

A formal model for the SPARCv8 ISA and a proof of non-interference for the LEON3 processor

Zhé Hóu, David Sanán, Alwen Tiu and Yang Liu

September 13, 2023

Abstract

We formalise the SPARCv8 instruction set architecture (ISA) which is used in processors such as LEON3. Our formalisation can be specialised to any SPARCv8 CPU, here we use LEON3 as a running example. Our model covers the operational semantics for all the instructions in the integer unit of the SPARCv8 architecture and it supports Isabelle code export, which effectively turns the Isabelle model into a SPARCv8 CPU simulator. We prove the language-based non-interference property for the LEON3 processor.

Our model is based on deterministic monad, which is a modified version of the non-deterministic monad from NICTA/14v. We also use the Word library developed by Jeremy Dawson and Gerwin Klein.

Contents

1	SPARC V8 architecture CPU model	1
2	CPU Register Definitions	3
3	The Monad	22
3.1	Failure	23
3.2	Generic functions on top of the state monad	24
3.3	The Monad Laws	24
4	Adding Exceptions	25
4.1	Monad Laws for the Exception Monad	26
5	Syntax	27
5.1	Syntax for the Nondeterministic State Monad	27
5.2	Syntax for the Exception Monad	28
6	Library of Monadic Functions and Combinators	29
7	Catching and Handling Exceptions	31

8	Hoare Logic	32
8.1	Validity	32
8.2	Determinism	34
8.3	Non-Failure	34
9	Basic exception reasoning	35
10	General Lemmas Regarding the Deterministic State Monad	35
10.1	Congruence Rules for the Function Package	35
10.2	Simplifying Monads	36
11	Low-level monadic reasoning	38
12	Register Operations	39
13	Getting Fields	39
14	Setting Fields	40
15	Clearing Fields	40
16	Memory Management Unit (MMU)	41
17	MMU Sizing	41
17.1	MMU Types	41
17.2	MMU length values	41
17.3	MMU index values	41
18	MMU Definition	42
19	Virtual Memory	42
19.1	MMU Auxiliary Definitions	42
19.2	Virtual Address Translation	43
20	SPARC V8 state model	47
21	state as a function	47
22	functions for state member access	48
23	SPARC instruction model	67
24	Single step theorem	133
25	Privilege safty	175
26	Single step non-interference property.	211

1 SPARC V8 architecture CPU model

```
theory Sparc-Types  
imports Main ../lib/WordDecl Word-Lib.Bit-Shifts-Infix-Syntax  
begin
```

The following type definitions are taken from David Sanan's definitions for SPARC machines.

```
type-synonym machine-word = word32  
type-synonym byte = word8  
type-synonym phys-address = word36
```

```
type-synonym virtua-address = word32  
type-synonym page-address = word24  
type-synonym offset = word12
```

```
type-synonym table-entry = word8
```

```
definition page-size :: word32 where page-size  $\equiv$  4096
```

```
type-synonym virtua-page-address = word20  
type-synonym context-type = word8
```

```
type-synonym word-length-t1 = word-length8  
type-synonym word-length-t2 = word-length6  
type-synonym word-length-t3 = word-length6  
type-synonym word-length-offset = word-length12  
type-synonym word-length-page = word-length24  
type-synonym word-length-phys-address = word-length36  
type-synonym word-length-virtua-address = word-length32  
type-synonym word-length-entry-type = word-length2  
type-synonym word-length-machine-word = word-length32
```

```
definition length-machine-word :: nat  
where length-machine-word  $\equiv$  LENGTH(word-length-machine-word)
```

2 CPU Register Definitions

The definitions below come from the SPARC Architecture Manual, Version 8. The LEON3 processor has been certified SPARC V8 conformant (2005).

definition *leon3khz* :: *word32*

where

leon3khz \equiv 33000

The following type definitions for MMU is taken from David Sanan's definitions for MMU.

The definitions below come from the UT699 LEON 3FT/SPARC V8 Microprocessor Functional Manual, Aeroflex, June 20, 2012, p35.

datatype *MMU-register*

= *CR* — Control Register
| *CTP* — ConText Pointer register
| *CNR* — Context Register
| *F TSR* — Fault Status Register
| *FAR* — Fault Address Register

lemma *MMU-register-induct*:

$P\ CR \implies P\ CTP \implies P\ CNR \implies P\ F\ TSR \implies P\ FAR$

$\implies P\ x$

by (*cases x*) *auto*

lemma *UNIV-MMU-register* [*no-atp*]: $UNIV = \{CR, CTP, CNR, F\ TSR, FAR\}$

apply (*safe*)

apply (*case-tac x*)

apply (*auto intro:MMU-register-induct*)

done

instantiation *MMU-register* :: *enum begin*

definition *enum-MMU-register* = [*CR, CTP, CNR, F\ TSR, FAR*]

definition

enum-all-MMU-register $P \longleftrightarrow P\ CR \wedge P\ CTP \wedge P\ CNR \wedge P\ F\ TSR \wedge P\ FAR$

definition

enum-ex-MMU-register $P \longleftrightarrow P\ CR \vee P\ CTP \vee P\ CNR \vee P\ F\ TSR \vee P\ FAR$

instance proof

qed (*simp-all only: enum-MMU-register-def enum-all-MMU-register-def*

enum-ex-MMU-register-def UNIV-MMU-register, simp-all)

end

type-synonym *MMU-context* = *MMU-register* \Rightarrow *machine-word*

PTE-flags is the last 8 bits of a PTE. See page 242 of SPARCV8 manual.

- C - bit 7
- M - bit 6,
- R - bit 5
- ACC - bit 4 2
- ET - bit 1 0.

type-synonym *PTE-flags* = *word8*

CPU-register datatype is an enumeration with the CPU registers defined in the SPARC V8 architecture.

datatype *CPU-register* =
PSR — Processor State Register
| *WIM* — Window Invalid Mask
| *TBR* — Trap Base Register
| *Y* — Multiply/Divide Register
| *PC* — Program Counter
| *nPC* — next Program Counter
| *DTQ* — Deferred-Trap Queue
| *FSR* — Floating-Point State Register
| *FQ* — Floating-Point Deferred-Trap Queue
| *CSR* — Coprocessor State Register
| *CQ* — Coprocessor Deferred-Trap Queue

| *ASR word5* — Ancillary State Register

The following two functions are dummies since we will not use ASRs. Future formalisation may add more details to this.

context
includes *bit-operations-syntax*
begin

definition *privileged-ASR* :: *word5* ⇒ *bool*
where
privileged-ASR *r* ≡ *False*

definition *illegal-instruction-ASR* :: *word5* ⇒ *bool*
where
illegal-instruction-ASR *r* ≡ *False*

definition *get-tt* :: *word32* ⇒ *word8*
where
get-tt *tbr* ≡
ucast (((*AND*) *tbr* *0b00000000000000000000111111110000*) >> 4)

Write the tt field of the TBR register. Return the new value of TBR.

```
definition write-tt :: word8 ⇒ word32 ⇒ word32
where
write-tt new-tt-val tbr-val ≡
  let tmp = (AND) tbr-val 0b11111111111111111111111111111111000000001111 in
    (OR) tmp (((ucast new-tt-val)::word32) << 4)
```

Get the nth bit of WIM. This equals ((AND) WIM 2ⁿ). N.B. the first bit of WIM is the 0th bit.

```
definition get-WIM-bit :: nat ⇒ word32 ⇒ word1
where
get-WIM-bit n wim ≡
  let mask = ((ucast (0b1::word1))::word32) << n in
    ucast (((AND) mask wim) >> n)
```

```
definition get-CWP :: word32 ⇒ word5
where
get-CWP psr ≡
  ucast ((AND) psr 0b000000000000000000000000000011111)
```

```
definition get-ET :: word32 ⇒ word1
where
get-ET psr ≡
  ucast (((AND) psr 0b0000000000000000000000000000100000) >> 5)
```

```
definition get-PIL :: word32 ⇒ word4
where
get-PIL psr ≡
  ucast (((AND) psr 0b000000000000000000000000111100000000) >> 8)
```

```
definition get-PS :: word32 ⇒ word1
where
get-PS psr ≡
  ucast (((AND) psr 0b00000000000000000000000000001000000) >> 6)
```

```
definition get-S :: word32 ⇒ word1
where
get-S psr ≡
  ucast (((AND) psr 0b00000000000000000000000000001000000) >> 6)
  if (((AND) psr (0b00000000000000000000000010000000::word32)) = 0 then 0
  else 1
```

definition *get-icc-N* :: *word32* ⇒ *word1*
where
get-icc-N psr ≡
ucast (((AND) psr 0b000000001000000000000000000000) >> 23)

definition *get-icc-Z* :: *word32* ⇒ *word1*
where
get-icc-Z psr ≡
ucast (((AND) psr 0b000000000100000000000000000000) >> 22)

definition *get-icc-V* :: *word32* ⇒ *word1*
where
get-icc-V psr ≡
ucast (((AND) psr 0b000000000100000000000000000000) >> 21)

definition *get-icc-C* :: *word32* ⇒ *word1*
where
get-icc-C psr ≡
ucast (((AND) psr 0b000000000100000000000000000000) >> 20)

definition *update-S* :: *word1* ⇒ *word32* ⇒ *word32*
where
update-S s-val psr-val ≡
let tmp0 = (AND) psr-val 0b1111111111111111111111111111111101111111 in
(OR) tmp0 (((ucast s-val)::word32) << 7)

Update the CWP field of PSR. Return the new value of PSR.

definition *update-CWP* :: *word5* ⇒ *word32* ⇒ *word32*
where
update-CWP cwp-val psr-val ≡
let tmp0 = (AND) psr-val (0b1111111111111111111111111111111100000::word32);
s-val = ((ucast (get-S psr-val))::word1)
in
if s-val = 0 then
(AND) ((OR) tmp0 (((ucast cwp-val)::word32)) (0b1111111111111111111111111111111101111111::word32))
else
(OR) ((OR) tmp0 (((ucast cwp-val)::word32)) (0b0000000000000000000000000000000010000000::word32))

Update the the ET, CWP, and S fields of PSR. Return the new value of PSR.

definition *update-PSR-rett* :: *word5* ⇒ *word1* ⇒ *word1* ⇒ *word32* ⇒ *word32*


```

    tmp1 = (OR) tmp0 (((ucast et)::word32) << 5);
    tmp2 = (OR) tmp1 (((ucast pil)::word32) << 8)
in
    tmp2

```

end

SPARC V8 architecture is organized in windows of 32 user registers. The data stored in a register is defined as a 32 bits word *reg-type*:

type-synonym *reg-type* = *word32*

The access to the value of a CPU register of type *CPU-register* is defined by a total function *cpu-context*

type-synonym *cpu-context* = *CPU-register* \Rightarrow *reg-type*

User registers are defined with the type *user-reg* represented by a 5 bits word.

type-synonym *user-reg-type* = *word5*

definition *PSR-S* :: *reg-type*
where *PSR-S* \equiv 6

Each window context is defined by a total function *window-context* from *user-register* to *reg-type* (32 bits word storing the actual value of the register).

type-synonym *window-context* = *user-reg-type* \Rightarrow *reg-type*

The number of windows is implementation dependent. The LEON architecture is composed of 16 different windows (a 4 bits word).

definition *NWINDOWS* :: *int*
where *NWINDOWS* \equiv 8

Maximum number of windows is 32 in SPARCV8.

type-synonym ('a) *window-size* = 'a *word*

Finally the user context is defined by another total function *user-context* from *window-size* to *window-context*. That is, the user context is a function taking as argument a register set window and a register within that window, and it returns the value stored in that user register.

type-synonym ('a) *user-context* = ('a) *window-size* \Rightarrow *window-context*

datatype *sys-reg* =
 CCR — Cache control register
 | *ICCR* — Instruction cache configuration register
 | *DCCR* — Data cache configuration register

type-synonym *sys-context* = *sys-reg* \Rightarrow *reg-type*

The memory model is defined by a total function from 32 bits words to 8 bits words

type-synonym *asi-type* = *word8*

The memory is defined as a function from page address to page, which is also defined as a function from physical address to *machine-word*

type-synonym *mem-val-type* = *word8*

type-synonym *mem-context* = *asi-type* \Rightarrow *phys-address* \Rightarrow *mem-val-type option*

type-synonym *cache-tag* = *word20*

type-synonym *cache-line-size* = *word12*

type-synonym *cache-type* = (*cache-tag* \times *cache-line-size*)

type-synonym *cache-context* = *cache-type* \Rightarrow *mem-val-type option*

The delayed-write pool generated from write state register instructions.

type-synonym *delayed-write-pool* = (*int* \times *reg-type* \times *CPU-register*) *list*

definition *DELAYNUM* :: *int*

where *DELAYNUM* \equiv 0

Convert a set to a list.

definition *list-of-set* :: '*a set* \Rightarrow '*a list*

where *list-of-set* *s* = (*SOME l. set l = s*)

lemma *set-list-of-set*: *finite s* \Longrightarrow *set (list-of-set s) = s*

unfolding *list-of-set-def*

by (*metis (mono-tags) finite-list some-eq-ex*)

type-synonym *ANNUL* = *bool*

type-synonym *RESET-TRAP* = *bool*

type-synonym *EXECUTE-MODE* = *bool*

type-synonym *RESET-MODE* = *bool*

type-synonym *ERROR-MODE* = *bool*

type-synonym *TICC-TRAP-TYPE* = *word7*

type-synonym *INTERRUPT-LEVEL* = *word3*

type-synonym *STORE-BARRIER-PENDING* = *bool*

The processor asserts this signal to ensure that the memory system will not process another SWAP or LDSTUB operation to the same memory byte.

type-synonym *pb-block-ldst-byte* = *virtua-address* \Rightarrow *bool*

The processor asserts this signal to ensure that the memory system will not process another SWAP or LDSTUB operation to the same memory word.

type-synonym *pb-block-ldst-word* = *virtua-address* \Rightarrow *bool*

```

record sparc-state-var =
  annul:: ANNUL
  resett:: RESET-TRAP
  exe:: EXECUTE-MODE
  reset:: RESET-MODE
  err:: ERROR-MODE
  ticc:: TICC-TRAP-TYPE
  itrpt-lvl:: INTERRUPT-LEVEL
  st-bar:: STORE-BARRIER-PENDING
  atm-ldst-byte:: pb-block-ldst-byte
  atm-ldst-word:: pb-block-ldst-word

```

```

definition get-annul :: sparc-state-var ⇒ bool
where get-annul v ≡ annul v

```

```

definition get-reset-trap :: sparc-state-var ⇒ bool
where get-reset-trap v ≡ resett v

```

```

definition get-exe-mode :: sparc-state-var ⇒ bool
where get-exe-mode v ≡ exe v

```

```

definition get-reset-mode :: sparc-state-var ⇒ bool
where get-reset-mode v ≡ reset v

```

```

definition get-err-mode :: sparc-state-var ⇒ bool
where get-err-mode v ≡ err v

```

```

definition get-ticc-trap-type :: sparc-state-var ⇒ word7
where get-ticc-trap-type v ≡ ticc v

```

```

definition get-interrupt-level :: sparc-state-var ⇒ word3
where get-interrupt-level v ≡ itrpt-lvl v

```

```

definition get-store-barrier-pending :: sparc-state-var ⇒ bool
where get-store-barrier-pending v ≡ st-bar v

```

```

definition write-annul :: bool ⇒ sparc-state-var ⇒ sparc-state-var
where write-annul b v ≡ v(|annul := b|)

```

```

definition write-reset-trap :: bool ⇒ sparc-state-var ⇒ sparc-state-var
where write-reset-trap b v ≡ v(|resett := b|)

```

```

definition write-exe-mode :: bool ⇒ sparc-state-var ⇒ sparc-state-var
where write-exe-mode b v ≡ v(|exe := b|)

```

```

definition write-reset-mode :: bool ⇒ sparc-state-var ⇒ sparc-state-var
where write-reset-mode b v ≡ v(|reset := b|)

```



```

let highest-bit = ((AND) w 0b1000000000000) >> 12 in
if highest-bit = 0 then
  (ucast w)::word32
else (OR) ((ucast w)::word32) 0b11111111111111111000000000000000

```

definition *zero-ext16* :: *word16* ⇒ *word32*
where
zero-ext16 w ≡ (ucast w)::word32

Given a word16 value, find the highest bit, and fill the left bits to be the highest bit.

definition *sign-ext16*::*word16* ⇒ *word32*
where
sign-ext16 w ≡
 let highest-bit = ((AND) w 0b1000000000000000) >> 15 in
 if highest-bit = 0 then
 (ucast w)::word32
 else (OR) ((ucast w)::word32) 0b11111111111111111000000000000000

Given a word22 value, find the highest bit, and fill the left bits to be the highest bit.

definition *sign-ext22*::*word22* ⇒ *word32*
where
sign-ext22 w ≡
 let highest-bit = ((AND) w 0b1000000000000000000000) >> 21 in
 if highest-bit = 0 then
 (ucast w)::word32
 else (OR) ((ucast w)::word32) 0b11111111110000000000000000000000

Given a word24 value, find the highest bit, and fill the left bits to be the highest bit.

definition *sign-ext24*::*word24* ⇒ *word32*
where
sign-ext24 w ≡
 let highest-bit = ((AND) w 0b100000000000000000000000) >> 23 in
 if highest-bit = 0 then
 (ucast w)::word32
 else (OR) ((ucast w)::word32) 0b11111111100000000000000000000000

Operations to be defined. The SPARC V8 architecture is composed of the following set of instructions:

- Load Integer Instructions

- Load Floating-point Instructions
- Load Coprocessor Instructions
- Store Integer Instructions
- Store Floating-point Instructions
- Store Coprocessor Instructions
- Atomic Load-Store Unsigned Byte Instructions
- SWAP Register With Memory Instruction
- SETHI Instructions
- NOP Instruction
- Logical Instructions
- Shift Instructions
- Add Instructions
- Tagged Add Instructions
- Subtract Instructions
- Tagged Subtract Instructions
- Multiply Step Instruction
- Multiply Instructions
- Divide Instructions
- SAVE and RESTORE Instructions
- Branch on Integer Condition Codes Instructions
- Branch on Floating-point Condition Codes Instructions
- Branch on Coprocessor Condition Codes Instructions
- Call and Link Instruction
- Jump and Link Instruction
- Return from Trap Instruction
- Trap on Integer Condition Codes Instructions
- Read State Register Instructions

- Write State Register Instructions
- STBAR Instruction
- Unimplemented Instruction
- Flush Instruction Memory
- Floating-point Operate (FPop) Instructions
- Convert Integer to Floating point Instructions
- Convert Floating point to Integer Instructions
- Convert Between Floating-point Formats Instructions
- Floating-point Move Instructions
- Floating-point Square Root Instructions
- Floating-point Add and Subtract Instructions
- Floating-point Multiply and Divide Instructions
- Floating-point Compare Instructions
- Coprocessor Operate Instructions

The CALL instruction.

datatype *call-type* = *CALL* — Call and Link

The SETHI instruction.

datatype *sethi-type* = *SETHI* — Set High 22 bits of r Register

The NOP instruction.

datatype *nop-type* = *NOP* — No Operation

The Branch on integer condition codes instructions.

datatype *bicc-type* =

- BE* — Branch on Equal
- | *BNE* — Branch on Not Equal
- | *BGU* — Branch on Greater Unsigned
- | *BLE* — Branch on Less or Equal
- | *BL* — Branch on Less
- | *BGE* — Branch on Greater or Equal
- | *BNEG* — Branch on Negative
- | *BG* — Branch on Greater
- | *BCS* — Branch on Carry Set (Less than, Unsigned)
- | *BLEU* — Branch on Less or Equal Unsigned

- | *BCC* — Branch on Carry Clear (Greater than or Equal, Unsigned)
- | *BA* — Branch Always
- | *BN* — Branch Never — Added for unconditional branches
- | *BPOS* — Branch on Positive
- | *BVC* — Branch on Overflow Clear
- | *BVS* — Branch on Overflow Set

Memory instructions. That is, load and store.

datatype *load-store-type* =

- LDSB* — Load Signed Byte
- | *LDUB* — Load Unsigned Byte
- | *LDUBA* — Load Unsigned Byte from Alternate space
- | *LDUH* — Load Unsigned Halfword
- | *LD* — Load Word
- | *LDA* — Load Word from Alternate space
- | *LDD* — Load Doubleword
- | *STB* — Store Byte
- | *STH* — Store Halfword
- | *ST* — Store Word
- | *STA* — Store Word into Alternate space
- | *STD* — Store Doubleword
- | *LDSBA* — Load Signed Byte from Alternate space
- | *LDSH* — Load Signed Halfword
- | *LDSHA* — Load Signed Halfword from Alternate space
- | *LDUHA* — Load Unsigned Halfword from Alternate space
- | *LDDA* — Load Doubleword from Alternate space
- | *STBA* — Store Byte into Alternate space
- | *STHA* — Store Halfword into Alternate space
- | *STDA* — Store Doubleword into Alternate space
- | *LDSTUB* — Atomic Load Store Unsigned Byte
- | *LDSTUBA* — Atomic Load Store Unsigned Byte in Alternate space
- | *SWAP* — Swap r Register with Mmemory
- | *SWAPA* — Swap r Register with Mmemory in Alternate space
- | *FLUSH* — Flush Instruction Memory
- | *STBAR* — Store Barrier

Arithmetic instructions.

datatype *arith-type* =

- ADD* — Add
- | *ADDcc* — Add and modify *icc*
- | *ADDX* — Add with Carry
- | *SUB* — Subtract
- | *SUBcc* — Subtract and modify *icc*
- | *SUBX* — Subtract with Carry
- | *UMUL* — Unsigned Integer Multiply
- | *SMUL* — Signed Integer Multiply
- | *SMULcc* — Signed Integer Multiply and modify *icc*
- | *UDIV* — Unsigned Integer Divide
- | *UDIVcc* — Unsigned Integer Divide and modify *icc*

- | *SDIV* — Signed Integer Divide
- | *ADDXcc* — Add with Carry and modify icc
- | *TADDcc* — Tagged Add and modify icc
- | *TADDccTV* — Tagged Add and modify icc and Trap on overflow
- | *SUBXcc* — Subtract with Carry and modify icc
- | *TSUBcc* — Tagged Subtract and modify icc
- | *TSUBccTV* — Tagged Subtract and modify icc and Trap on overflow
- | *MULScc* — Multiply Step and modify icc
- | *UMULcc* — Unsigned Integer Multiply and modify icc
- | *SDIVcc* — Signed Integer Divide and modify icc

Logical instructions.

datatype *logic-type* =

- ANDs* — And
- | *ANDcc* — And and modify icc
- | *ANDN* — And Not
- | *ANDNcc* — And Not and modify icc
- | *ORs* — Inclusive-Or
- | *ORcc* — Inclusive-Or and modify icc
- | *ORN* — Inclusive Or Not
- | *XORs* — Exclusive-Or
- | *XNOR* — Exclusive-Nor
- | *ORNcc* — Inclusive-Or Not and modify icc
- | *XORcc* — Exclusive-Or and modify icc
- | *XNORcc* — Exclusive-Nor and modify icc

Shift instructions.

datatype *shift-type* =

- SLL* — Shift Left Logical
- | *SRL* — Shift Right Logical
- | *SRA* — Shift Right Arithmetic

Other Control-transfer instructions.

datatype *ctrl-type* =

- JMPL* — Jump and Link
- | *RETT* — Return from Trap
- | *SAVE* — Save caller's window
- | *RESTORE* — Restore caller's window

Access state registers instructions.

datatype *sreg-type* =

- RDASR* — Read Ancillary State Register
- | *RDY* — Read Y Register
- | *RDPSR* — Read Processor State Register
- | *RDWIM* — Read Window Invalid Mask Register
- | *RDTBR* — Read Trap Base Register
- | *WRASR* — Write Ancillary State Register
- | *WRY* — Write Y Register

- | *WRPSR* — Write Processor State Register
- | *WRWIM* — Write Window Invalid Mask Register
- | *WRTBR* — Write Trap Base Register

Unimplemented instruction.

datatype *uimp-type* = *UNIMP* — Unimplemented

Trap on integer condition code instructions.

datatype *ticc-type* =

- TA* — Trap Always
- | *TN* — Trap Never
- | *TNE* — Trap on Not Equal
- | *TE* — Trap on Equal
- | *TG* — Trap on Greater
- | *TLE* — Trap on Less or Equal
- | *TGE* — Trap on Greater or Equal
- | *TL* — Trap on Less
- | *TGU* — Trap on Greater Unsigned
- | *TLEU* — Trap on Less or Equal Unsigned
- | *TCC* — Trap on Carry Clear (Greater than or Equal, Unsigned)
- | *TCS* — Trap on Carry Set (Less Than, Unsigned)
- | *TPOS* — Trap on Postive
- | *TNEG* — Trap on Negative
- | *TVC* — Trap on Overflow Clear
- | *TVS* — Trap on Overflow Set

datatype *sparc-operation* =

- call-type call-type*
- | *sethi-type sethi-type*
- | *nop-type nop-type*
- | *bicc-type bicc-type*
- | *load-store-type load-store-type*
- | *arith-type arith-type*
- | *logic-type logic-type*
- | *shift-type shift-type*
- | *ctrl-type ctrl-type*
- | *sreg-type sreg-type*
- | *uimp-type uimp-type*
- | *ticc-type ticc-type*

datatype *Trap* =

- reset*
- | *data-store-error*
- | *instruction-access-MMU-miss*
- | *instruction-access-error*
- | *r-register-access-error*
- | *instruction-access-exception*
- | *privileged-instruction*
- | *illegal-instruction*

```

|unimplemented-FLUSH
|watchpoint-detected
|fp-disabled
|cp-disabled
|window-overflow
|window-underflow
|mem-address-not-aligned
|fp-exception
|cp-exception
|data-access-error
|data-access-MMU-miss
|data-access-exception
|tag-overflow
|division-by-zero
|trap-instruction
|interrupt-level-n

```

datatype *Exception* =

— The following are processor states that are not in the instruction model,
— but we MAY want to deal with these from hardware perspective.

```

|execute-mode
|reset-mode
|error-mode

```

— The following are self-defined exceptions.

```

invalid-cond-f2
|invalid-op2-f2
|illegal-instruction2 — when  $i = 0$  for load/store not from alternate space
|invalid-op3-f3-op11
|case-impossible
|invalid-op3-f3-op10
|invalid-op-f3
|unsupported-instruction
|fetch-instruction-error
|invalid-trap-cond

```

end

end

```

theory Lib
imports Main
begin

```

```

lemma hd-map-simp:

```

$b \neq [] \implies \text{hd } (\text{map } a \ b) = a \ (\text{hd } b)$
by (*rule hd-map*)

lemma *tl-map-simp*:
 $\text{tl } (\text{map } a \ b) = \text{map } a \ (\text{tl } b)$
by (*induct b,auto*)

lemma *Collect-eq*:
 $\{x. P \ x\} = \{x. Q \ x\} \longleftrightarrow (\forall x. P \ x = Q \ x)$
by (*rule iffI*) *auto*

lemma *iff-impI*: $\llbracket P \implies Q = R \rrbracket \implies (P \longrightarrow Q) = (P \longrightarrow R)$ **by** *blast*

definition
 $\text{fun-app} :: ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ (**infixr** \$ 10) **where**
 $f \ \$ \ x \equiv f \ x$

declare *fun-app-def* [*iff*]

lemma *fun-app-cong*[*fundef-cong*]:
 $\llbracket f \ x = f' \ x' \rrbracket \implies (f \ \$ \ x) = (f' \ \$ \ x')$
by *simp*

lemma *fun-app-apply-cong*[*fundef-cong*]:
 $f \ x \ y = f' \ x' \ y' \implies (f \ \$ \ x) \ y = (f' \ \$ \ x') \ y'$
by *simp*

lemma *if-apply-cong*[*fundef-cong*]:
 $\llbracket P = P'; x = x'; P' \implies f \ x' = f' \ x'; \neg P' \implies g \ x' = g' \ x' \rrbracket$
 $\implies (\text{if } P \ \text{then } f \ \text{else } g) \ x = (\text{if } P' \ \text{then } f' \ \text{else } g') \ x'$
by *simp*

abbreviation (*input*) *split* :: ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'c **where**
 $\text{split} \equiv \text{case-prod}$

lemma *split-apply-cong*[*fundef-cong*]:
 $\llbracket f \ (\text{fst } p) \ (\text{snd } p) \ s = f' \ (\text{fst } p') \ (\text{snd } p') \ s' \rrbracket \implies \text{split } f \ p \ s = \text{split } f' \ p' \ s'$
by (*simp add: split-def*)

definition
 $\text{pred-conj} :: ('a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow \text{bool})$ (**infixl** and 35)
where
 $\text{pred-conj } P \ Q \equiv \lambda x. P \ x \ \wedge \ Q \ x$

definition
 $\text{pred-disj} :: ('a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow \text{bool})$ (**infixl** or 30)
where
 $\text{pred-disj } P \ Q \equiv \lambda x. P \ x \ \vee \ Q \ x$

definition

$$\text{pred-neg} :: ('a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow \text{bool}) \text{ (not - [40] 40)}$$
where

$$\text{pred-neg } P \equiv \lambda x. \neg P x$$

definition $K \equiv \lambda x y. x$

definition

$$\text{zipWith} :: ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a \text{ list} \Rightarrow 'b \text{ list} \Rightarrow 'c \text{ list} \text{ where}$$

$$\text{zipWith } f \text{ xs ys} \equiv \text{map (split } f) (\text{zip xs ys})$$
primrec

$$\text{delete} :: 'a \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$$
where

$$\text{delete } y [] = []$$

$$| \text{delete } y (x \# xs) = (\text{if } y=x \text{ then } xs \text{ else } x \# \text{delete } y \text{ xs})$$
primrec

$$\text{find} :: ('a \Rightarrow \text{bool}) \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ option}$$
where

$$\text{find } f [] = \text{None}$$

$$| \text{find } f (x \# xs) = (\text{if } f x \text{ then } \text{Some } x \text{ else } \text{find } f \text{ xs})$$
definition

$$\text{swp } f \equiv \lambda x y. f y x$$
primrec (*nonexhaustive*)
$$\text{theRight} :: 'a + 'b \Rightarrow 'b \text{ where}$$

$$\text{theRight (Inr } x) = x$$
primrec (*nonexhaustive*)
$$\text{theLeft} :: 'a + 'b \Rightarrow 'a \text{ where}$$

$$\text{theLeft (Inl } x) = x$$
definition

$$\text{isLeft } x \equiv (\exists y. x = \text{Inl } y)$$
definition

$$\text{isRight } x \equiv (\exists y. x = \text{Inr } y)$$
definition

$$\text{const } x \equiv \lambda y. x$$
lemma *tranclD2*:
$$(x, y) \in R^+ \implies \exists z. (x, z) \in R^* \wedge (z, y) \in R$$

by (*erule tranclE*) *auto*

lemma *linorder-min-same1* [*simp*]:
$$(\text{min } y \text{ } x = y) = (y \leq (x :: 'a :: \text{linorder}))$$

by (*auto simp: min-def linorder-not-less*)

lemma *linorder-min-same2* [*simp*]:

$(\min x y = y) = (y \leq (x::'a::\text{linorder}))$

by (*auto simp: min-def linorder-not-le*)

A combinator for pairing up well-formed relations. The divisor function splits the population in halves, with the True half greater than the False half, and the supplied relations control the order within the halves.

definition

$\text{wf-sum} :: ('a \Rightarrow \text{bool}) \Rightarrow ('a \times 'a) \text{ set} \Rightarrow ('a \times 'a) \text{ set} \Rightarrow ('a \times 'a) \text{ set}$

where

$\text{wf-sum divisor } r \ r' \equiv$

$(\{(x, y). \neg \text{divisor } x \wedge \neg \text{divisor } y\} \cap r')$

$\cup \{(x, y). \neg \text{divisor } x \wedge \text{divisor } y\}$

$\cup (\{(x, y). \text{divisor } x \wedge \text{divisor } y\} \cap r)$

lemma *wf-sum-wf*:

$\llbracket \text{wf } r; \text{wf } r' \rrbracket \Longrightarrow \text{wf } (\text{wf-sum divisor } r \ r')$

apply (*simp add: wf-sum-def*)

apply (*rule wf-Un*)⁺

apply (*erule wf-Int2*)

apply (*rule wf-subset*

where $r = \text{measure } (\lambda x. \text{If } (\text{divisor } x) \ 1 \ 0)$)])

apply *simp*

apply *clarsimp*

apply *blast*

apply (*erule wf-Int2*)

apply *blast*

done

abbreviation(*input*)

$\text{option-map} == \text{map-option}$

lemmas *option-map-def = map-option-case*

lemma *False-implies-equals* [*simp*]:

$((\text{False} \Longrightarrow P) \Longrightarrow \text{PROP } Q) \equiv \text{PROP } Q$

apply (*rule equal-intr-rule*)

apply (*erule meta-mp*)

apply *simp*

apply *simp*

done

lemma *split-paired-Ball*:

$(\forall x \in A. P x) = (\forall x y. (x, y) \in A \longrightarrow P (x, y))$

by *auto*

lemma *split-paired-Bex*:

```

( $\exists x \in A. P x$ ) = ( $\exists x y. (x,y) \in A \wedge P (x,y)$ )
by auto

```

end

```

theory DetMonad
imports ../Lib
begin

```

State monads are used extensively in the seL4 specification. They are defined below.

3 The Monad

The basic type of the deterministic state monad with failure is very similar to the normal state monad. Instead of a pair consisting of result and new state, we return a pair coupled with a failure flag. The flag is *True* if the computation have failed. Conversely, if the flag is *False*, the computation resulting in the returned result have succeeded.

```

type-synonym ('s,'a) det-monad = 's  $\Rightarrow$  ('a  $\times$  's)  $\times$  bool

```

The definition of fundamental monad functions *return* and *bind*. The monad function *return* x does not change the state, does not fail, and returns x .

definition

```

return :: 'a  $\Rightarrow$  ('s,'a) det-monad where
return a  $\equiv$   $\lambda s. ((a,s),False)$ 

```

The monad function *bind* $f g$, also written $f \gg= g$, is the execution of f followed by the execution of g . The function g takes the result value *and* the result state of f as parameter. The definition says that the result of the combined operation is the result which is created by g applied to the result of f . The combined operation may have failed, if f may have failed or g may have failed on the result of f .

David Sanan and Zhe Hou: The original definition of bind is very inefficient when converted to executable code. Here we change it to a more efficient version for execution. The idea remains the same.

definition $h1 f s = f s$

definition $h2 g fs = (let (a,b) = fst (fs) in g a b)$

definition $bind :: ('s, 'a) det-monad \Rightarrow ('a \Rightarrow ('s, 'b) det-monad) \Rightarrow ('s, 'b) det-monad$ (**infixl** $>>= 60$)

where

$bind\ f\ g \equiv \lambda s. (\text{let } fs = h1\ f\ s;$
 $\quad v = h2\ g\ fs$
in
 $(fst\ v, (snd\ v \vee snd\ fs)))$

Sometimes it is convenient to write $bind$ in reverse order.

abbreviation(*input*)

$bind-rev :: ('c \Rightarrow ('a, 'b) det-monad) \Rightarrow ('a, 'c) det-monad \Rightarrow ('a, 'b) det-monad$ (**infixl** $=<< 60$) **where**
 $g =<< f \equiv f >>= g$

The basic accessor functions of the state monad. get returns the current state as result, does not fail, and does not change the state. $put\ s$ returns nothing (*unit*), changes the current state to s and does not fail.

definition

$get :: ('s, 's) det-monad$ **where**
 $get \equiv \lambda s. ((s, s), False)$

definition

$put :: 's \Rightarrow ('s, unit) det-monad$ **where**
 $put\ s \equiv \lambda-. (((), s), False)$

3.1 Failure

The monad function that always fails. Returns the current state and sets the failure flag.

definition

$fail :: 'a \Rightarrow ('s, 'a) det-monad$ **where**
 $fail\ a \equiv \lambda s. ((a, s), True)$

Assertions: fail if the property P is not true

definition

$assert :: bool \Rightarrow ('a, unit) det-monad$ **where**
 $assert\ P \equiv \text{if } P \text{ then return } () \text{ else fail } ()$

An assertion that also can introspect the current state.

definition

$state-assert :: ('s \Rightarrow bool) \Rightarrow ('s, unit) det-monad$
where
 $state-assert\ P \equiv get >>= (\lambda s. assert\ (P\ s))$

3.2 Generic functions on top of the state monad

Apply a function to the current state and return the result without changing the state.

definition

$gets :: ('s \Rightarrow 'a) \Rightarrow ('s, 'a) \text{ det-monad}$ **where**
 $gets\ f \equiv get\ \>\>= (\lambda s. return\ (f\ s))$

Modify the current state using the function passed in.

definition

$modify :: ('s \Rightarrow 's) \Rightarrow ('s, unit) \text{ det-monad}$ **where**
 $modify\ f \equiv get\ \>\>= (\lambda s. put\ (f\ s))$

lemma *simpler-gets-def*: $gets\ f = (\lambda s. ((f\ s, s), False))$

apply (*simp add*: $gets\text{-def}$ $return\text{-def}$ $bind\text{-def}$ $h1\text{-def}$ $h2\text{-def}$ $get\text{-def}$)
done

lemma *simpler-modify-def*:

$modify\ f = (\lambda s. (((), f\ s), False))$
by (*simp add*: $modify\text{-def}$ $bind\text{-def}$ $h1\text{-def}$ $h2\text{-def}$ $get\text{-def}$ $put\text{-def}$)

Execute the given monad when the condition is true, return () otherwise.

definition

$when1 :: bool \Rightarrow ('s, unit) \text{ det-monad} \Rightarrow$
 $('s, unit) \text{ det-monad}$ **where**
 $when1\ P\ m \equiv if\ P\ then\ m\ else\ return\ ()$

Execute the given monad unless the condition is true, return () otherwise.

definition

$unless :: bool \Rightarrow ('s, unit) \text{ det-monad} \Rightarrow$
 $('s, unit) \text{ det-monad}$ **where**
 $unless\ P\ m \equiv when1\ (\neg P)\ m$

Perform a test on the current state, performing the left monad if the result is true or the right monad if the result is false.

definition

$condition :: ('s \Rightarrow bool) \Rightarrow ('s, 'r) \text{ det-monad} \Rightarrow ('s, 'r) \text{ det-monad} \Rightarrow ('s, 'r)$
 det-monad

where

$condition\ P\ L\ R \equiv \lambda s. if\ (P\ s)\ then\ (L\ s)\ else\ (R\ s)$

notation (output)

$condition\ ((condition\ (-)//\ (-)//\ (-))\ [1000,1000,1000]\ 1000)$

3.3 The Monad Laws

Each monad satisfies at least the following three laws.

return is absorbed at the left of a ($\gg=$), applying the return value directly:

lemma *return-bind* [*simp*]: $(\text{return } x \gg= f) = f \ x$
by (*simp add: return-def bind-def h1-def h2-def*)

return is absorbed on the right of a ($\gg=$)

lemma *bind-return* [*simp*]: $(m \gg= \text{return}) = m$
apply (*rule ext*)
apply (*simp add: bind-def h1-def h2-def return-def split-def*)
done

($\gg=$) is associative

lemma *bind-assoc*:
fixes $m :: ('a, 'b) \text{ det-monad}$
fixes $f :: 'b \Rightarrow ('a, 'c) \text{ det-monad}$
fixes $g :: 'c \Rightarrow ('a, 'd) \text{ det-monad}$
shows $(m \gg= f) \gg= g = m \gg= (\lambda x. f \ x \gg= g)$
apply (*unfold bind-def h1-def h2-def Let-def split-def*)
apply (*rule ext*)
apply *clarsimp*
done

4 Adding Exceptions

The type $('s, 'a) \text{ det-monad}$ gives us determinism and failure. We now extend this monad with exceptional return values that abort normal execution, but can be handled explicitly. We use the sum type to indicate exceptions. In $('s, 'e + 'a) \text{ det-monad}$, $'s$ is the state, $'e$ is an exception, and $'a$ is a normal return value.

This new type itself forms a monad again. Since type classes in Isabelle are not powerful enough to express the class of monads, we provide new names for the *return* and ($\gg=$) functions in this monad. We call them *returnOk* (for normal return values) and *bindE* (for composition). We also define *throwError* to return an exceptional value.

definition
 $\text{returnOk} :: 'a \Rightarrow ('s, 'e + 'a) \text{ det-monad}$ **where**
 $\text{returnOk} \equiv \text{return } o \text{ Inr}$

definition
 $\text{throwError} :: 'e \Rightarrow ('s, 'e + 'a) \text{ det-monad}$ **where**
 $\text{throwError} \equiv \text{return } o \text{ Inl}$

Lifting a function over the exception type: if the input is an exception, return that exception; otherwise continue execution.

definition
 $\text{lift} :: ('a \Rightarrow ('s, 'e + 'b) \text{ det-monad}) \Rightarrow$

$$'e + 'a \Rightarrow ('s, 'e + 'b) \text{ det-monad}$$

where

$$\begin{aligned} \text{lift } f \ v \equiv & \text{ case } v \text{ of } \text{Inl } e \Rightarrow \text{throwError } e \\ & | \text{Inr } v' \Rightarrow f \ v' \end{aligned}$$

The definition of ($\gg=$) in the exception monad (new name bindE): the same as normal ($\gg=$), but the right-hand side is skipped if the left-hand side produced an exception.

definition

$$\begin{aligned} \text{bindE} :: ('s, 'e + 'a) \text{ det-monad} \Rightarrow \\ ('a \Rightarrow ('s, 'e + 'b) \text{ det-monad}) \Rightarrow \\ ('s, 'e + 'b) \text{ det-monad} \quad (\mathbf{infixl} \gg=E \ 60) \end{aligned}$$

where

$$\text{bindE } f \ g \equiv \text{bind } f \ (\text{lift } g)$$

Lifting a normal deterministic monad into the exception monad is achieved by always returning its result as normal result and never throwing an exception.

definition

$$\text{liftE} :: ('s, 'a) \text{ det-monad} \Rightarrow ('s, 'e + 'a) \text{ det-monad}$$

where

$$\text{liftE } f \equiv f \ \gg= \ (\lambda r. \text{return } (\text{Inr } r))$$

Since the underlying type and *return* function changed, we need new definitions for when and unless:

definition

$$\begin{aligned} \text{whenE} :: \text{bool} \Rightarrow ('s, 'e + \text{unit}) \text{ det-monad} \Rightarrow \\ ('s, 'e + \text{unit}) \text{ det-monad} \end{aligned}$$

where

$$\text{whenE } P \ f \equiv \text{if } P \ \text{then } f \ \text{else } \text{returnOk } ()$$

definition

$$\begin{aligned} \text{unlessE} :: \text{bool} \Rightarrow ('s, 'e + \text{unit}) \text{ det-monad} \Rightarrow \\ ('s, 'e + \text{unit}) \text{ det-monad} \end{aligned}$$

where

$$\text{unlessE } P \ f \equiv \text{if } P \ \text{then } \text{returnOk } () \ \text{else } f$$

Throwing an exception when the parameter is *None*, otherwise returning *v* for *Some v*.

definition

$$\begin{aligned} \text{throw-opt} :: 'e \Rightarrow 'a \ \text{option} \Rightarrow ('s, 'e + 'a) \text{ det-monad} \quad \mathbf{where} \\ \text{throw-opt } ex \ x \equiv \\ \text{case } x \ \text{of } \text{None} \Rightarrow \text{throwError } ex \ | \ \text{Some } v \Rightarrow \text{returnOk } v \end{aligned}$$

4.1 Monad Laws for the Exception Monad

More direct definition of liftE :

lemma *liftE-def2*:

liftE $f = (\lambda s. ((\lambda(v,s'). (Inr\ v, s'))\ (fst\ (f\ s)),\ snd\ (f\ s)))$
by (*auto simp: Let-def liftE-def return-def split-def bind-def h1-def h2-def*)

Left *returnOk* absorption over ($\gg=E$):

lemma *returnOk-bindE [simp]*: (*returnOk* $x \gg=E f$) = $f\ x$
apply (*unfold bindE-def returnOk-def*)
apply (*clarsimp simp: lift-def*)
done

lemma *lift-return [simp]*:

lift (*return* \circ *Inr*) = *return*
by (*rule ext*)
(*simp add: lift-def throwError-def split: sum.splits*)

Right *returnOk* absorption over ($\gg=E$):

lemma *bindE-returnOk [simp]*: ($m \gg=E$ *returnOk*) = m
by (*simp add: bindE-def returnOk-def*)

Associativity of ($\gg=E$):

lemma *bindE-assoc*:

($m \gg=E f$) $\gg=E g$ = $m \gg=E (\lambda x. f\ x \gg=E g)$
apply (*simp add: bindE-def bind-assoc*)
apply (*rule arg-cong [where f= $\lambda x. m \gg=E x$]*)
apply (*rule ext*)
apply (*case-tac x, simp-all add: lift-def throwError-def*)
done

returnOk could also be defined via *liftE*:

lemma *returnOk-liftE*:

returnOk $x = liftE$ (*return* x)
by (*simp add: liftE-def returnOk-def*)

Execution after throwing an exception is skipped:

lemma *throwError-bindE [simp]*:

(*throwError* $E \gg=E f$) = *throwError* E
by (*simp add: bindE-def bind-def h1-def h2-def throwError-def lift-def return-def*)

5 Syntax

This section defines traditional Haskell-like do-syntax for the state monad in Isabelle.

5.1 Syntax for the Nondeterministic State Monad

We use *K-bind* to syntactically indicate the case where the return argument of the left side of a ($\gg=$) is ignored

definition

K-bind-def [iff]: $K\text{-bind} \equiv \lambda x y. x$

nonterminal

dobinds and *dobind* and *nobind*

syntax

$-dobind \quad :: [pttrn, 'a] \Rightarrow dobind \quad ((- \leftarrow -) 10)$
 $\quad \quad \quad :: dobind \Rightarrow dobinds \quad (-)$
 $-nobind \quad :: 'a \Rightarrow dobind \quad (-)$
 $-dobinds \quad :: [dobind, dobinds] \Rightarrow dobinds \quad ((-);//(-))$

 $-do \quad \quad :: [dobinds, 'a] \Rightarrow 'a \quad ((do ((-);//(-))//od) 100)$

translations

$-do (-dobinds b bs) e \quad == -do b (-do bs e)$
 $-do (-nobind b) e \quad == b >>= (CONST K\text{-bind } e)$
 $do x \leftarrow a; e od \quad == a >>= (\lambda x. e)$

Syntax examples:

lemma $do x \leftarrow return 1;$
 $\quad return (2::nat);$
 $\quad return x$
 $od =$
 $return 1 >>=$
 $(\lambda x. return (2::nat)) >>=$
 $\quad K\text{-bind } (return x)$
by (*rule refl*)

lemma $do x \leftarrow return 1;$
 $\quad return 2;$
 $\quad return x$
 $od = return 1$
by *simp*

5.2 Syntax for the Exception Monad

Since the exception monad is a different type, we need to syntactically distinguish it in the syntax. We use *doE/odE* for this, but can re-use most of the productions from *do/od* above.

syntax

$-doE \quad :: [dobinds, 'a] \Rightarrow 'a \quad ((doE ((-);//(-))//odE) 100)$

translations

$-doE (-dobinds b bs) e \quad == -doE b (-doE bs e)$
 $-doE (-nobind b) e \quad == b >>=E (CONST K\text{-bind } e)$
 $doE x \leftarrow a; e odE \quad == a >>=E (\lambda x. e)$

Syntax examples:

lemma $doE\ x \leftarrow returnOk\ 1;$
 $returnOk\ (2::nat);$
 $returnOk\ x$
 $odE =$
 $returnOk\ 1 >>=E$
 $(\lambda x. returnOk\ (2::nat) >>=E$
 $\quad K\text{-bind}\ (returnOk\ x))$
by $(rule\ refl)$

lemma $doE\ x \leftarrow returnOk\ 1;$
 $returnOk\ 2;$
 $returnOk\ x$
 $odE = returnOk\ 1$
by $simp$

6 Library of Monadic Functions and Combinators

Lifting a normal function into the monad type:

definition
 $liftM :: ('a \Rightarrow 'b) \Rightarrow ('s, 'a)\ det\text{-monad} \Rightarrow ('s, 'b)\ det\text{-monad}$
where
 $liftM\ f\ m \equiv do\ x \leftarrow m; return\ (f\ x)\ od$

The same for the exception monad:

definition
 $liftME :: ('a \Rightarrow 'b) \Rightarrow ('s, 'e+'a)\ det\text{-monad} \Rightarrow ('s, 'e+'b)\ det\text{-monad}$
where
 $liftME\ f\ m \equiv doE\ x \leftarrow m; returnOk\ (f\ x)\ odE$

Run a sequence of monads from left to right, ignoring return values.

definition
 $sequence\text{-}x :: ('s, 'a)\ det\text{-monad}\ list \Rightarrow ('s, unit)\ det\text{-monad}$
where
 $sequence\text{-}x\ xs \equiv foldr\ (\lambda x\ y. x >>= (\lambda_. y))\ xs\ (return\ ())$

Map a monadic function over a list by applying it to each element of the list from left to right, ignoring return values.

definition
 $mapM\text{-}x :: ('a \Rightarrow ('s, 'b)\ det\text{-monad}) \Rightarrow 'a\ list \Rightarrow ('s, unit)\ det\text{-monad}$
where
 $mapM\text{-}x\ f\ xs \equiv sequence\text{-}x\ (map\ f\ xs)$

Map a monadic function with two parameters over two lists, going through both lists simultaneously, left to right, ignoring return values.

definition
 $zipWithM\text{-}x :: ('a \Rightarrow 'b \Rightarrow ('s, 'c)\ det\text{-monad}) \Rightarrow$
 $'a\ list \Rightarrow 'b\ list \Rightarrow ('s, unit)\ det\text{-monad}$

where

$zipWithM\text{-}x\ f\ xs\ ys \equiv sequence\text{-}x\ (zipWith\ f\ xs\ ys)$

The same three functions as above, but returning a list of return values instead of *unit*

definition

$sequence :: ('s, 'a)\ det\text{-}monad\ list \Rightarrow ('s, 'a\ list)\ det\text{-}monad$

where

$sequence\ xs \equiv let\ mcons = (\lambda p\ q.\ p \gg = (\lambda x.\ q \gg = (\lambda y.\ return\ (x\#\ y))))$
 $in\ foldr\ mcons\ xs\ (return\ [])$

definition

$mapM :: ('a \Rightarrow ('s, 'b)\ det\text{-}monad) \Rightarrow 'a\ list \Rightarrow ('s, 'b\ list)\ det\text{-}monad$

where

$mapM\ f\ xs \equiv sequence\ (map\ f\ xs)$

definition

$zipWithM :: ('a \Rightarrow 'b \Rightarrow ('s, 'c)\ det\text{-}monad) \Rightarrow$
 $'a\ list \Rightarrow 'b\ list \Rightarrow ('s, 'c\ list)\ det\text{-}monad$

where

$zipWithM\ f\ xs\ ys \equiv sequence\ (zipWith\ f\ xs\ ys)$

definition

$foldM :: ('b \Rightarrow 'a \Rightarrow ('s, 'a)\ det\text{-}monad) \Rightarrow 'b\ list \Rightarrow 'a \Rightarrow ('s, 'a)\ det\text{-}monad$

where

$foldM\ m\ xs\ a \equiv foldr\ (\lambda p\ q.\ q \gg = m\ p)\ xs\ (return\ a)$

The sequence and map functions above for the exception monad, with and without lists of return value

definition

$sequenceE\text{-}x :: ('s, 'e+'a)\ det\text{-}monad\ list \Rightarrow ('s, 'e+unit)\ det\text{-}monad$

where

$sequenceE\text{-}x\ xs \equiv foldr\ (\lambda x\ y.\ doE\ -\ \leftarrow\ x;\ y\ odE)\ xs\ (returnOk\ ())$

definition

$mapME\text{-}x :: ('a \Rightarrow ('s, 'e+'b)\ det\text{-}monad) \Rightarrow 'a\ list \Rightarrow$
 $('s, 'e+unit)\ det\text{-}monad$

where

$mapME\text{-}x\ f\ xs \equiv sequenceE\text{-}x\ (map\ f\ xs)$

definition

$sequenceE :: ('s, 'e+'a)\ det\text{-}monad\ list \Rightarrow ('s, 'e+'a\ list)\ det\text{-}monad$

where

$sequenceE\ xs \equiv let\ mcons = (\lambda p\ q.\ p \gg = E\ (\lambda x.\ q \gg = E\ (\lambda y.\ returnOk\ (x\#\ y))))$
 $in\ foldr\ mcons\ xs\ (returnOk\ [])$

definition

$mapME :: ('a \Rightarrow ('s, 'e+'b)\ det\text{-}monad) \Rightarrow 'a\ list \Rightarrow$
 $('s, 'e+'b\ list)\ det\text{-}monad$

where
 $mapME\ f\ xs \equiv sequenceE\ (map\ f\ xs)$

Filtering a list using a monadic function as predicate:

primrec
 $filterM :: ('a \Rightarrow ('s, bool)\ det-monad) \Rightarrow 'a\ list \Rightarrow ('s, 'a\ list)\ det-monad$
where
 $filterM\ P\ [] = return\ []$
 $| filterM\ P\ (x\ \#\ xs) = do$
 $\quad b \leftarrow P\ x;$
 $\quad ys \leftarrow filterM\ P\ xs;$
 $\quad return\ (if\ b\ then\ (x\ \#\ ys)\ else\ ys)$
 od

7 Catching and Handling Exceptions

Turning an exception monad into a normal state monad by catching and handling any potential exceptions:

definition
 $catch :: ('s, 'e + 'a)\ det-monad \Rightarrow$
 $\quad ('e \Rightarrow ('s, 'a)\ det-monad) \Rightarrow$
 $\quad ('s, 'a)\ det-monad\ (\mathbf{infix}\ <catch>\ 10)$
where
 $f\ <catch>\ handler \equiv$
 $\quad do\ x \leftarrow f;$
 $\quad case\ x\ of$
 $\quad\quad Inr\ b \Rightarrow return\ b$
 $\quad\quad | Inl\ e \Rightarrow handler\ e$
 $\quad od$

Handling exceptions, but staying in the exception monad. The handler may throw a type of exceptions different from the left side.

definition
 $handleE' :: ('s, 'e1 + 'a)\ det-monad \Rightarrow$
 $\quad ('e1 \Rightarrow ('s, 'e2 + 'a)\ det-monad) \Rightarrow$
 $\quad ('s, 'e2 + 'a)\ det-monad\ (\mathbf{infix}\ <handle2>\ 10)$
where
 $f\ <handle2>\ handler \equiv$
 $\quad do$
 $\quad\quad v \leftarrow f;$
 $\quad\quad case\ v\ of$
 $\quad\quad\quad Inl\ e \Rightarrow handler\ e$
 $\quad\quad\quad | Inr\ v' \Rightarrow return\ (Inr\ v')$
 $\quad od$

A type restriction of the above that is used more commonly in practice: the exception handle (potentially) throws exception of the same type as the left-hand side.

definition

$$\begin{aligned} \text{handleE} &:: ('s, 'x + 'a) \text{ det-monad} \Rightarrow \\ &('x \Rightarrow ('s, 'x + 'a) \text{ det-monad}) \Rightarrow \\ &('s, 'x + 'a) \text{ det-monad} \text{ (infix } <\text{handle}> 10) \end{aligned}$$
where

$$\text{handleE} \equiv \text{handleE}'$$

Handling exceptions, and additionally providing a continuation if the left-hand side throws no exception:

definition

$$\begin{aligned} \text{handle-elseE} &:: ('s, 'e + 'a) \text{ det-monad} \Rightarrow \\ &('e \Rightarrow ('s, 'ee + 'b) \text{ det-monad}) \Rightarrow \\ &('a \Rightarrow ('s, 'ee + 'b) \text{ det-monad}) \Rightarrow \\ &('s, 'ee + 'b) \text{ det-monad} \\ &(- <\text{handle}> - <\text{else}> - 10) \end{aligned}$$
where

$$\begin{aligned} f <\text{handle}> \text{ handler } <\text{else}> \text{ continue} &\equiv \\ \text{do } v \leftarrow f; & \\ \text{case } v \text{ of } \text{Inl } e \Rightarrow \text{ handler } e & \\ \quad | \text{Inr } v' \Rightarrow \text{ continue } v' & \\ \text{od} & \end{aligned}$$

8 Hoare Logic

8.1 Validity

This section defines a Hoare logic for partial correctness for the deterministic state monad as well as the exception monad. The logic talks only about the behaviour part of the monad and ignores the failure flag.

The logic is defined semantically. Rules work directly on the validity predicate.

In the deterministic state monad, validity is a triple of precondition, monad, and postcondition. The precondition is a function from state to bool (a state predicate), the postcondition is a function from return value to state to bool. A triple is valid if for all states that satisfy the precondition, all result values and result states that are returned by the monad satisfy the postcondition. Note that if the computation returns the empty set, the triple is trivially valid. This means *assert P* does not require us to prove that *P* holds, but rather allows us to assume *P*! Proving non-failure is done via separate predicate and calculus (see below).

definition

$$\begin{aligned} \text{valid} &:: ('s \Rightarrow \text{bool}) \Rightarrow ('s, 'a) \text{ det-monad} \Rightarrow ('a \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow \text{bool} \\ &(\{\!\{-\}\!\} / - / \{\!\{-\}\!\}) \end{aligned}$$
where

$$\{\!\{P}\!\} f \{\!\{Q}\!\} \equiv \forall s. P s \longrightarrow (\forall r s'. ((r, s') = \text{fst } (f s) \longrightarrow Q r s'))$$

Validity for the exception monad is similar and build on the standard validity above. Instead of one postcondition, we have two: one for normal and one for exceptional results.

definition

$$\begin{aligned} \text{validE} &:: ('s \Rightarrow \text{bool}) \Rightarrow ('s, 'a + 'b) \text{ det-monad} \Rightarrow \\ &\quad ('b \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow \\ &\quad ('a \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow \text{bool} \\ &(\{\{-\}\} / - / (\{\{-\}\} / \{\{-\}\})) \end{aligned}$$

where

$$\{\{P\}\} f \{\{Q\}\}, \{\{E\}\} \equiv \{\{P\}\} f \{\{ \lambda v s. \text{ case } v \text{ of } \text{Inr } r \Rightarrow Q \ r \ s \mid \text{Inl } e \Rightarrow E \ e \ s \}\}$$

The following two instantiations are convenient to separate reasoning for exceptional and normal case.

definition

$$\begin{aligned} \text{validE-R} &:: ('s \Rightarrow \text{bool}) \Rightarrow ('s, 'e + 'a) \text{ det-monad} \Rightarrow \\ &\quad ('a \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow \text{bool} \\ &(\{\{-\}\} / - / \{\{-\}\}, -) \end{aligned}$$

where

$$\{\{P\}\} f \{\{Q\}\}, - \equiv \text{validE } P \ f \ Q \ (\lambda x y. \text{ True})$$

definition

$$\begin{aligned} \text{validE-E} &:: ('s \Rightarrow \text{bool}) \Rightarrow ('s, 'e + 'a) \text{ det-monad} \Rightarrow \\ &\quad ('e \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow \text{bool} \\ &(\{\{-\}\} / - / -, \{\{-\}\}) \end{aligned}$$

where

$$\{\{P\}\} f \ -, \{\{Q\}\} \equiv \text{validE } P \ f \ (\lambda x y. \text{ True}) \ Q$$

Abbreviations for trivial preconditions:

abbreviation(*input*)

$$\text{top} :: 'a \Rightarrow \text{bool} \ (\top)$$

where

$$\top \equiv \lambda -. \text{ True}$$

abbreviation(*input*)

$$\text{bottom} :: 'a \Rightarrow \text{bool} \ (\perp)$$

where

$$\perp \equiv \lambda -. \text{ False}$$

Abbreviations for trivial postconditions (taking two arguments):

abbreviation(*input*)

$$\text{toptop} :: 'a \Rightarrow 'b \Rightarrow \text{bool} \ (\top\top)$$

where

$$\top\top \equiv \lambda - -. \text{ True}$$

abbreviation(*input*)

$$\text{botbot} :: 'a \Rightarrow 'b \Rightarrow \text{bool} \ (\perp\perp)$$

where

$$\perp\perp \equiv \lambda - -. \text{ False}$$

Lifting \wedge and \vee over two arguments. Lifting \wedge and \vee over one argument is already defined (written *and* and *or*).

definition

$bipred-conj :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool)$
(infixl *And 96*)

where

$bipred-conj P Q \equiv \lambda x y. P x y \wedge Q x y$

definition

$bipred-disj :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool)$
(infixl *Or 91*)

where

$bipred-disj P Q \equiv \lambda x y. P x y \vee Q x y$

8.2 Determinism

A monad of type *det-monad* is deterministic iff it returns exactly one state and result and does not fail

definition

$det :: ('a, 's) det-monad \Rightarrow bool$

where

$det f \equiv \forall s. \exists r. f s = (r, False)$

A deterministic *det-monad* can be turned into a normal state monad:

definition

$the-run-state :: ('s, 'a) det-monad \Rightarrow 's \Rightarrow 'a \times 's$

where

$the-run-state M \equiv \lambda s. THE s'. fst (M s) = s'$

8.3 Non-Failure

With the failure flag, we can formulate non-failure separately from validity. A monad *m* does not fail under precondition *P*, if for no start state in that precondition it sets the failure flag.

definition

$no-fail :: ('s \Rightarrow bool) \Rightarrow ('s, 'a) det-monad \Rightarrow bool$

where

$no-fail P m \equiv \forall s. P s \longrightarrow \neg (snd (m s))$

It is often desired to prove non-failure and a Hoare triple simultaneously, as the reasoning is often similar. The following definitions allow such reasoning to take place.

definition

$validNF :: ('s \Rightarrow bool) \Rightarrow ('s, 'a) det-monad \Rightarrow ('a \Rightarrow 's \Rightarrow bool) \Rightarrow bool$
($\{\!\{-\}\!\} / - / \{\!\{-\}\!\}$)

where

$validNF\ P\ f\ Q \equiv valid\ P\ f\ Q \wedge no-fail\ P\ f$

definition

$validE-NF :: ('s \Rightarrow bool) \Rightarrow ('s, 'a + 'b)\ det-monad \Rightarrow$
 $('b \Rightarrow 's \Rightarrow bool) \Rightarrow$
 $('a \Rightarrow 's \Rightarrow bool) \Rightarrow bool$
 $(\{\{-\}\} / - / (\{\{-\}\} / \{\{-\}\}!))$

where

$validE-NF\ P\ f\ Q\ E \equiv validE\ P\ f\ Q\ E \wedge no-fail\ P\ f$

lemma *validE-NF-alt-def*:

$\{\{ P \} B \{ Q \}, \{ E \}!\} = \{\{ P \} B \{ \lambda v\ s.\ case\ v\ of\ Inl\ e \Rightarrow E\ e\ s \mid Inr\ r \Rightarrow Q\ r\ s \}!\}$
by (*clarsimp simp: validE-NF-def validE-def validNF-def*)

9 Basic exception reasoning

The following predicates *no-throw* and *no-return* allow reasoning that functions in the exception monad either do no throw an exception or never return normally.

definition *no-throw* $P\ A \equiv \{\{ P \} A \{ \lambda\ -. True \}, \{ \lambda\ -. False \} \}$

definition *no-return* $P\ A \equiv \{\{ P \} A \{ \lambda\ -. False \}, \{ \lambda\ -. True \} \}$

end

theory *DetMonadLemmas*

imports *DetMonad*

begin

10 General Lemmas Regarding the Deterministic State Monad

10.1 Congruence Rules for the Function Package

lemma *bind-cong*[*fundef-cong*]:

$\llbracket f = f'; \bigwedge v\ s\ s'. (v, s') = fst\ (f'\ s) \implies g\ v\ s' = g'\ v\ s' \rrbracket \implies f\ >>= g = f'\ >>= g'$

apply (*rule ext*)

apply (*auto simp: bind-def h1-def h2-def Let-def split-def intro: rev-image-eqI*)

done

lemma *bind-apply-cong* [*fundef-cong*]:

$\llbracket f\ s = f'\ s'; \bigwedge rv\ st. (rv, st) = fst\ (f'\ s') \implies g\ rv\ st = g'\ rv\ st \rrbracket$

$\implies (f \gg = g) s = (f' \gg = g') s'$
apply (*simp add: bind-def h1-def h2-def*)
apply (*auto simp: split-def intro: SUP-cong [OF refl] intro: rev-image-eqI*)
done

lemma *bindE-cong[fundef-cong]*:
 $\llbracket M = M' ; \bigwedge v s s'. (Inr v, s') = fst (M' s) \implies N v s' = N' v s' \rrbracket \implies bindE$
 $M N = bindE M' N'$
apply (*simp add: bindE-def*)
apply (*rule bind-cong*)
apply (*rule refl*)
apply (*unfold lift-def*)
apply (*case-tac v, simp-all*)
done

lemma *bindE-apply-cong[fundef-cong]*:
 $\llbracket f s = f' s' ; \bigwedge rv st. (Inr rv, st) = fst (f' s') \implies g rv st = g' rv st \rrbracket$
 $\implies (f \gg =E g) s = (f' \gg =E g') s'$
apply (*simp add: bindE-def*)
apply (*rule bind-apply-cong*)
apply *assumption*
apply (*case-tac rv, simp-all add: lift-def*)
done

lemma *K-bind-apply-cong[fundef-cong]*:
 $\llbracket f st = f' st' \rrbracket \implies K\text{-bind } f \text{ arg } st = K\text{-bind } f' \text{ arg}' st'$
by *simp*

lemma *when-apply-cong[fundef-cong]*:
 $\llbracket C = C' ; s = s' ; C' \implies m s' = m' s' \rrbracket \implies whenE C m s = whenE C' m' s'$
by (*simp add: whenE-def*)

lemma *unless-apply-cong[fundef-cong]*:
 $\llbracket C = C' ; s = s' ; \neg C' \implies m s' = m' s' \rrbracket \implies unlessE C m s = unlessE C' m'$
 s'
by (*simp add: unlessE-def*)

lemma *whenE-apply-cong[fundef-cong]*:
 $\llbracket C = C' ; s = s' ; C' \implies m s' = m' s' \rrbracket \implies whenE C m s = whenE C' m' s'$
by (*simp add: whenE-def*)

lemma *unlessE-apply-cong[fundef-cong]*:
 $\llbracket C = C' ; s = s' ; \neg C' \implies m s' = m' s' \rrbracket \implies unlessE C m s = unlessE C' m'$
 s'
by (*simp add: unlessE-def*)

10.2 Simplifying Monads

lemma *nested-bind [simp]*:

```

do x ← do y ← f; return (g y) od; h x od =
do y ← f; h (g y) od
apply (clarsimp simp add: bind-def h1-def h2-def)
apply (rule ext)
apply (clarsimp simp add: Let-def split-def return-def)
done

lemma assert-True [simp]:
  assert True >>= f = f ()
by (simp add: assert-def)

lemma when-True-bind [simp]:
  when1 True g >>= f = g >>= f
by (simp add: when1-def bind-def return-def)

lemma whenE-False-bind [simp]:
  whenE False g >>=E f = f ()
by (simp add: whenE-def bindE-def returnOk-def lift-def)

lemma whenE-True-bind [simp]:
  whenE True g >>=E f = g >>=E f
by (simp add: whenE-def bindE-def returnOk-def lift-def)

lemma when-True [simp]: when1 True X = X
by (clarsimp simp: when1-def)

lemma when-False [simp]: when1 False X = return ()
by (clarsimp simp: when1-def)

lemma unless-False [simp]: unless False X = X
by (clarsimp simp: unless-def)

lemma unless-True [simp]: unless True X = return ()
by (clarsimp simp: unless-def)

lemma unlessE-whenE:
  unlessE P = whenE (~P)
by (rule ext)+ (simp add: unlessE-def whenE-def)

lemma unless-when:
  unless P = when1 (~P)
by (rule ext)+ (simp add: unless-def when1-def)

lemma gets-to-return [simp]: gets (λs. v) = return v
by (clarsimp simp: gets-def put-def get-def bind-def h1-def h2-def return-def)

lemma liftE-handleE' [simp]: ((liftE a) <handle2> b) = liftE a
apply (clarsimp simp: liftE-def handleE'-def)
done

```

lemma *liftE-handleE [simp]*: $((\text{liftE } a) <\text{handle}> b) = \text{liftE } a$
apply (*unfold handleE-def*)
apply *simp*
done

lemma *condition-split*:
 $P (\text{condition } C \ a \ b \ s) = (((C \ s) \longrightarrow P \ (a \ s)) \wedge (\neg (C \ s) \longrightarrow P \ (b \ s)))$
apply (*clarsimp simp: condition-def*)
done

lemma *condition-split-asm*:
 $P (\text{condition } C \ a \ b \ s) = (\neg (C \ s \wedge \neg P \ (a \ s)) \vee \neg C \ s \wedge \neg P \ (b \ s))$
apply (*clarsimp simp: condition-def*)
done

lemmas *condition-splits = condition-split condition-split-asm*

lemma *condition-true-triv [simp]*:
 $\text{condition } (\lambda-. \text{True}) \ A \ B = A$
apply (*rule ext*)
apply (*clarsimp split: condition-splits*)
done

lemma *condition-false-triv [simp]*:
 $\text{condition } (\lambda-. \text{False}) \ A \ B = B$
apply (*rule ext*)
apply (*clarsimp split: condition-splits*)
done

lemma *condition-true*: $\llbracket P \ s \rrbracket \Longrightarrow \text{condition } P \ A \ B \ s = A \ s$
apply (*clarsimp simp: condition-def*)
done

lemma *condition-false*: $\llbracket \neg P \ s \rrbracket \Longrightarrow \text{condition } P \ A \ B \ s = B \ s$
apply (*clarsimp simp: condition-def*)
done

11 Low-level monadic reasoning

lemma *valid-make-schematic-post*:
 $(\forall s0. \llbracket \lambda s. P \ s0 \ s \rrbracket f \llbracket \lambda rv \ s. Q \ s0 \ rv \ s \rrbracket) \Longrightarrow$
 $\llbracket \lambda s. \exists s0. P \ s0 \ s \wedge (\forall rv \ s'. Q \ s0 \ rv \ s' \longrightarrow Q' \ rv \ s') \rrbracket f \llbracket Q' \rrbracket$
by (*auto simp add: valid-def no-fail-def split: prod.splits*)

lemma *validNF-make-schematic-post*:
 $(\forall s0. \llbracket \lambda s. P \ s0 \ s \rrbracket f \llbracket \lambda rv \ s. Q \ s0 \ rv \ s \rrbracket!) \Longrightarrow$
 $\llbracket \lambda s. \exists s0. P \ s0 \ s \wedge (\forall rv \ s'. Q \ s0 \ rv \ s' \longrightarrow Q' \ rv \ s') \rrbracket f \llbracket Q' \rrbracket!$
by (*auto simp add: valid-def validNF-def no-fail-def split: prod.splits*)

lemma *validE-make-schematic-post*:

$(\forall s0. \{ \lambda s. P s0 s \} f \{ \lambda rv s. Q s0 rv s \}, \{ \lambda rv s. E s0 rv s \}) \implies$
 $\{ \lambda s. \exists s0. P s0 s \wedge (\forall rv s'. Q s0 rv s' \longrightarrow Q' rv s') \wedge (\forall rv s'. E s0 rv s' \longrightarrow E' rv s') \} f \{ Q' \}, \{ E' \}$
by (*auto simp add: validE-def valid-def no-fail-def split: prod.splits sum.splits*)

lemma *validE-NF-make-schematic-post*:

$(\forall s0. \{ \lambda s. P s0 s \} f \{ \lambda rv s. Q s0 rv s \}, \{ \lambda rv s. E s0 rv s \}!) \implies$
 $\{ \lambda s. \exists s0. P s0 s \wedge (\forall rv s'. Q s0 rv s' \longrightarrow Q' rv s') \wedge (\forall rv s'. E s0 rv s' \longrightarrow E' rv s') \} f \{ Q' \}, \{ E' \}!$
by (*auto simp add: validE-NF-def validE-def valid-def no-fail-def split: prod.splits sum.splits*)

lemma *validNF-conjD1*: $\{ P \} f \{ \lambda rv s. Q rv s \wedge Q' rv s \}! \implies \{ P \} f \{ Q \}!$
by (*fastforce simp: validNF-def valid-def no-fail-def*)

lemma *validNF-conjD2*: $\{ P \} f \{ \lambda rv s. Q rv s \wedge Q' rv s \}! \implies \{ P \} f \{ Q' \}!$
by (*fastforce simp: validNF-def valid-def no-fail-def*)

lemma *exec-gets*:

$(\text{gets } f \gg = m) s = m (f s) s$
by (*simp add: simpler-gets-def bind-def h1-def h2-def*)

lemma *in-gets*:

$(r, s') = \text{fst } (\text{gets } f s) = (r = f s \wedge s' = s)$
by (*simp add: simpler-gets-def*)

end

12 Register Operations

theory *RegistersOps*

imports *Main ../lib/WordDecl Word-Lib.Bit-Shifts-Infix-Syntax*

begin

context

includes *bit-operations-syntax*

begin

This theory provides operations to get, set and clear bits in registers

13 Getting Fields

Get a field of type *'b word* starting at *index* from *addr* of type *'a word*

definition *get-field-from-word-a-b*: $'a::\text{len word} \Rightarrow \text{nat} \Rightarrow 'b::\text{len word}$
where

get-field-from-word-a-b *addr index*
 \equiv let *off* = (*size addr* - *LENGTH('b')*)
in *ucast* ((*addr* << (*off-index*)) >> *off*)

Obtain, from *addr* of type '*a word*', another '*a word*' containing the field of length *len* starting at *index* in *addr*.

definition *get-field-from-word-a-a*:: '*a::len word* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow '*a::len word*
where
get-field-from-word-a-a *addr index len*
 \equiv (*addr* << (*size addr* - (*index+len*)) >> (*size addr* - *len*))

14 Setting Fields

Set the field of type '*b word*' at *index* from *record* of type '*a word*'.

definition *set-field* :: '*a::len word* \Rightarrow '*b::len word* \Rightarrow *nat* \Rightarrow '*a::len word*
where
set-field *record field index*
 \equiv let *mask*:: ('*a::len word*) = (*mask* (*size field*)) << *index*
in (*record* AND (*NOT mask*)) OR ((*ucast field*) << *index*)

15 Clearing Fields

Zero the *n* initial bits of *addr*.

definition *clear-n-bits*:: '*a::len word* \Rightarrow *nat* \Rightarrow '*a::len word*
where
clear-n-bits *addr n* \equiv *addr* AND (*NOT* (*mask n*))

Gets the natural value of a 32 bit mask

definition *get-nat-from-mask*:: *word32* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow (*word32* \times *nat*)
where

get-nat-from-mask *w m v* \equiv if (*w* AND (*mask m*) = 0) then (*w* >> *m*, *v+m*)
else (*w,m*)

definition *get-nat-from-mask32*:: *word32* \Rightarrow *nat*

where

get-nat-from-mask32 *w* \equiv
if (*w*=0) then *len-of TYPE* (*word-length32*)
else
let (*w,res*) = *get-nat-from-mask* *w* 16 0 in
let (*w,res*) = *get-nat-from-mask* *w* 8 *res* in
let (*w,res*) = *get-nat-from-mask* *w* 4 *res* in
let (*w,res*) = *get-nat-from-mask* *w* 2 *res* in
let (*w,res*) = *get-nat-from-mask* *w* 1 *res* in
res

end

end

16 Memory Management Unit (MMU)

```
theory MMU
imports Main RegistersOps Sparc-Types
begin
```

17 MMU Sizing

We need some citation here for documentation about the MMU.

The MMU uses the Address Space Identifiers (ASI) to control memory access. ASI = 8, 10 are for user; ASI = 9, 11 are for supervisor.

17.1 MMU Types

```
type-synonym word-PTE-flags = word8
type-synonym word-length-PTE-flags = word-length8
```

17.2 MMU length values

Definitions for the length of the virtua address, page size, virtual translation tables indexes, virtual address offset and Page protection flags

```
definition length-entry-type :: nat
where length-entry-type  $\equiv$  LENGTH(word-length-entry-type)
definition length-phys-address :: nat
where length-phys-address  $\equiv$  LENGTH(word-length-phys-address)
definition length-virtua-address :: nat
where length-virtua-address  $\equiv$  LENGTH(word-length-virtua-address)
definition length-page :: nat where length-page  $\equiv$  LENGTH(word-length-page)
definition length-t1 :: nat where length-t1  $\equiv$  LENGTH(word-length-t1)
definition length-t2 :: nat where length-t2  $\equiv$  LENGTH(word-length-t2)
definition length-t3 :: nat where length-t3  $\equiv$  LENGTH(word-length-t3)
definition length-offset :: nat where length-offset  $\equiv$  LENGTH(word-length-offset)
definition length-PTE-flags :: nat where
length-PTE-flags  $\equiv$  LENGTH(word-length-PTE-flags)
```

17.3 MMU index values

```
definition va-t1-index :: nat where va-t1-index  $\equiv$  length-virtua-address - length-t1
definition va-t2-index :: nat where va-t2-index  $\equiv$  va-t1-index - length-t2
definition va-t3-index :: nat where va-t3-index  $\equiv$  va-t2-index - length-t3
```

definition *va-offset-index* :: nat **where** *va-offset-index* \equiv *va-t3-index* - *length-offset*
definition *pa-page-index* :: nat
where *pa-page-index* \equiv *length-phys-address* - *length-page*
definition *pa-offset-index* :: nat **where**
pa-offset-index \equiv *pa-page-index* - *length-page*

18 MMU Definition

record *MMU-state* =
registers :: *MMU-context*

The following functions access MMU registers via addresses. See UT699LEON3FT manual page 35.

definition *mmu-reg-val*:: *MMU-state* \Rightarrow *virtua-address* \Rightarrow *machine-word option*
where *mmu-reg-val* *mmu-state* *addr* \equiv
if *addr* = 0x000 *then* — MMU control register
Some ((*registers* *mmu-state*) *CR*)
else if *addr* = 0x100 *then* — Context pointer register
Some ((*registers* *mmu-state*) *CTP*)
else if *addr* = 0x200 *then* — Context register
Some ((*registers* *mmu-state*) *CNR*)
else if *addr* = 0x300 *then* — Fault status register
Some ((*registers* *mmu-state*) *FTSR*)
else if *addr* = 0x400 *then* — Fault address register
Some ((*registers* *mmu-state*) *FAR*)
else None

definition *mmu-reg-mod*:: *MMU-state* \Rightarrow *virtua-address* \Rightarrow *machine-word* \Rightarrow
MMU-state option **where**
mmu-reg-mod *mmu-state* *addr* *w* \equiv
if *addr* = 0x000 *then* — MMU control register
Some (*mmu-state*(*registers* := (*registers* *mmu-state*)(*CR* := *w*)))
else if *addr* = 0x100 *then* — Context pointer register
Some (*mmu-state*(*registers* := (*registers* *mmu-state*)(*CTP* := *w*)))
else if *addr* = 0x200 *then* — Context register
Some (*mmu-state*(*registers* := (*registers* *mmu-state*)(*CNR* := *w*)))
else if *addr* = 0x300 *then* — Fault status register
Some (*mmu-state*(*registers* := (*registers* *mmu-state*)(*FTSR* := *w*)))
else if *addr* = 0x400 *then* — Fault address register
Some (*mmu-state*(*registers* := (*registers* *mmu-state*)(*FAR* := *w*)))
else None

19 Virtual Memory

19.1 MMU Auxiliary Definitions

definition *getCTPVal*:: *MMU-state* \Rightarrow *machine-word*
where *getCTPVal* *mmu* \equiv (*registers* *mmu*) *CTP*

definition $getCNRVal :: MMU\text{-state} \Rightarrow machine\text{-word}$
where $getCNRVal\ mmu \equiv (registers\ mmu)\ CNR$

The physical context table address is got from the ConText Pointer register (CTP) and the Context Register (CNR) MMU registers. The CTP is shifted to align it with the physical address (36 bits) and we add the table index given on CNR. CTP is right shifted 2 bits, cast to phys address and left shifted 6 bytes to be aligned with the context register. CNR is 2 bits left shifted for alignment with the context table.

definition $compose\text{-context}\text{-table}\text{-addr} :: machine\text{-word} \Rightarrow machine\text{-word}$
 $\Rightarrow phys\text{-address}$

where

$compose\text{-context}\text{-table}\text{-addr}\ ctp\ cnr$
 $\equiv ((ucast\ (ctp\ >> 2)) \ll 6) + (ucast\ cnr \ll 2)$

19.2 Virtual Address Translation

Get the context table phys address from the MMU registers

definition $get\text{-context}\text{-table}\text{-addr} :: MMU\text{-state} \Rightarrow phys\text{-address}$
where

$get\text{-context}\text{-table}\text{-addr}\ mmu$
 $\equiv compose\text{-context}\text{-table}\text{-addr}\ (getCTPVal\ mmu)\ (getCNRVal\ mmu)$

definition $va\text{-list}\text{-index} :: nat\ list$ **where**

$va\text{-list}\text{-index} \equiv [va\text{-t1}\text{-index}, va\text{-t2}\text{-index}, va\text{-t3}\text{-index}, 0]$

definition $offset\text{-index} :: nat\ list$ **where**

$offset\text{-index}$
 $\equiv [length\text{-machine}\text{-word}$
 $\ ,\ length\text{-machine}\text{-word} - length\text{-t1}$
 $\ ,\ length\text{-machine}\text{-word} - length\text{-t1} - length\text{-t2}$
 $\ ,\ length\text{-machine}\text{-word} - length\text{-t1} - length\text{-t2} - length\text{-t3}$
 $\]$

definition $index\text{-len}\text{-table} :: nat\ list$ **where** $index\text{-len}\text{-table} \equiv [8, 6, 6, 0]$

definition $n\text{-context}\text{-tables} :: nat$ **where** $n\text{-context}\text{-tables} \equiv 3$

The following are basic physical memory read functions. At this level we don't need the write memory yet.

definition $mem\text{-context}\text{-val} :: asi\text{-type} \Rightarrow phys\text{-address} \Rightarrow$
 $mem\text{-context} \Rightarrow mem\text{-val}\text{-type}\ option$

where

$mem\text{-context}\text{-val}\ asi\ add\ m \equiv$
 $let\ asi8 = word\text{-of}\text{-int}\ 8;$
 $r1 = m\ asi\ add$

```

in
if r1 = None then
  m asi8 add
else r1

```

context

includes *bit-operations-syntax*

begin

Given an ASI (word8), an address (word32) *addr*, read the 32bit value from the memory addresses starting from address *addr'* where *addr' = addr* exception that the last two bits are 0's. That is, read the data from *addr'*, *addr'+1*, *addr'+2*, *addr'+3*.

definition *mem-context-val-w32* :: *asi-type* ⇒ *phys-address* ⇒ *mem-context* ⇒ *word32 option*

where

mem-context-val-w32 asi addr m ≡

```

let addr' = (AND) addr 0b1111111111111111111111111111111100;
  addr0 = (OR) addr' 0b00000000000000000000000000000000;
  addr1 = (OR) addr' 0b00000000000000000000000000000001;
  addr2 = (OR) addr' 0b00000000000000000000000000000010;
  addr3 = (OR) addr' 0b00000000000000000000000000000011;
  r0 = mem-context-val asi addr0 m;
  r1 = mem-context-val asi addr1 m;
  r2 = mem-context-val asi addr2 m;
  r3 = mem-context-val asi addr3 m

```

in

if r0 = None ∨ r1 = None ∨ r2 = None ∨ r3 = None then

None

else

```

let byte0 = case r0 of Some v ⇒ v;
  byte1 = case r1 of Some v ⇒ v;
  byte2 = case r2 of Some v ⇒ v;
  byte3 = case r3 of Some v ⇒ v

```

in

```

Some ((OR) ((OR) ((OR) ((ucast(byte0)) << 24)
  ((ucast(byte1)) << 16))
  ((ucast(byte2)) << 8))
  (ucast(byte3)))

```

get-addr-from-table browses the page description tables until it finds a PTE (bits==suc (suc 0)).

If it is a PTE it aligns the 24 most significant bits of the entry with the most significant bits of the phys address and or-ed with the offset, which will vary depending on the entry level. In the case we are looking at the last table level (level 3), the offset is aligned to 0 otherwise it will be 2.

If the table entry is a PTD (bits== Suc 0), the index is obtained from the virtual address depending on the current level and or-ed with the PTD.

```

function ptd-lookup:: virtua-address ⇒ virtua-address ⇒
mem-context ⇒ nat ⇒ (phys-address × PTE-flags) option
where ptd-lookup va pt m lvl = (
  if lvl > 3 then None
  else
    let thislvl-offset = (
      if lvl = 1 then (ucast ((ucast (va >> 24))::word8))::word32
      else if lvl = 2 then (ucast ((ucast (va >> 18))::word6))::word32
      else (ucast ((ucast (va >> 12))::word6))::word32);
      thislvl-addr = (OR) pt thislvl-offset;
      thislvl-data = mem-context-val-w32 (word-of-int 9) (ucast thislvl-addr) m
    in
      case thislvl-data of
      Some v ⇒ (
        let et-val = (AND) v 0b00000000000000000000000000000011 in
        if et-val = 0 then — Invalid
          None
        else if et-val = 1 then — Page Table Descriptor
          let ptp = (AND) v 0b11111111111111111111111111111100 in
          ptd-lookup va ptp m (lvl+1)
        else if et-val = 2 then — Page Table Entry
          let ppn = (ucast (v >> 8))::word24;
              va-offset = (ucast ((ucast va)::word12))::word36
          in
            Some (((OR) (((ucast ppn)::word36) << 12) va-offset),
              ((ucast v)::word8))
          else — et-val = 3, reserved.
            None
        )
      |None ⇒ None)

```

by pat-completeness auto

termination

by (relation measure (λ (va, (pt, (m, lvl))). 4 - lvl)) auto

definition get-acc-flag:: PTE-flags ⇒ word3 **where**
get-acc-flag w8 ≡ (ucast (w8 >> 2))::word3

definition mmu-readable:: word3 ⇒ asi-type ⇒ bool **where**
mmu-readable f asi ≡
 if uint asi ∈ {8, 10} then
 if uint f ∈ {0,1,2,3,5} then True
 else False
 else if uint asi ∈ {9, 11} then
 if uint f ∈ {0,1,2,3,5,6,7} then True
 else False
 else False

definition *mmu-writable*:: *word3* \Rightarrow *asi-type* \Rightarrow *bool* **where**

```
mmu-writable f asi  $\equiv$   
  if uint asi  $\in$  {8, 10} then  
    if uint f  $\in$  {1,3} then True  
    else False  
  else if uint asi  $\in$  {9, 11} then  
    if uint f  $\in$  {1,3,5,7} then True  
    else False  
  else False
```

definition *virt-to-phys* :: *virtua-address* \Rightarrow *MMU-state* \Rightarrow *mem-context* \Rightarrow
 (*phys-address* \times *PTE-flags*) *option*

where

```
virt-to-phys va mmu m  $\equiv$   
  let ctp-val = mmu-reg-val mmu (0x100);  
      cnr-val = mmu-reg-val mmu (0x200);  
      mmu-cr-val = (registers mmu) CR  
  in  
  if (AND) mmu-cr-val 1  $\neq$  0 then — MMU enabled.  
  case (ctp-val,cnr-val) of  
  (Some v1, Some v2)  $\Rightarrow$   
    let context-table-entry = (OR) ((v1 >> 11) << 11)  
        (((AND) v2 0b00000000000000000000000011111111) << 2);  
        context-table-data = mem-context-val-w32 (word-of-int 9)  
        (ucast context-table-entry) m  
    in (  
      case context-table-data of  
      Some lvl1-page-table  $\Rightarrow$   
        ptd-lookup va lvl1-page-table m 1  
      | None  $\Rightarrow$  None  
    )  
  | -  $\Rightarrow$  None  
  else Some ((ucast va), ((0b11101111)::word8))
```

The below function gives the initial values of MMU registers. In particular, the MMU context register CR is 0 because: We don't know the bits for IMPL, VER, and SC; the bits for PSO are 0s because we use TSO; the reserved bits are 0s; we assume NF bits are 0s; and most importantly, the E bit is 0 because when the machine starts up, MMU is disabled. An initial boot procedure (bootloader or something like that) should configure the MMU and then enable it if the OS uses MMU.

```
definition MMU-registers-init :: MMU-context
where MMU-registers-init r ≡ 0
```

```
definition mmu-setup :: MMU-state
where mmu-setup ≡ (|registers=MMU-registers-init)
```

```
end
```

```
end
```

20 SPARC V8 state model

```
theory Sparc-State
imports Main Sparc-Types ../lib/wp/DetMonadLemmas MMU
begin
```

21 state as a function

```
record cpu-cache =
  dcache:: cache-context
  icache:: cache-context
```

The state *sparc-state* is defined as a tuple *cpu-context*, *user-context*, *mem-context*, defining the state of the CPU registers, user registers, memory, cache, and delayed write pool respectively. Additionally, a boolean indicates whether the state is undefined or not.

```
record (overloaded) ('a) sparc-state =
  cpu-reg:: cpu-context
  user-reg:: ('a) user-context
  sys-reg:: sys-context
  mem:: mem-context
  mmu:: MMU-state
  cache:: cpu-cache
  dwrite:: delayed-write-pool
  state-var:: sparc-state-var
  traps:: Trap set
  undef:: bool
```

22 functions for state member access

definition *cpu-reg-val*:: CPU-register \Rightarrow ('a) sparc-state \Rightarrow reg-type
where
cpu-reg-val reg state \equiv (*cpu-reg* state) reg

definition *cpu-reg-mod* :: word32 \Rightarrow CPU-register \Rightarrow ('a) sparc-state \Rightarrow
('a) sparc-state
where *cpu-reg-mod* data-w32 cpu state \equiv
state(*cpu-reg* := ((*cpu-reg* state)(cpu := data-w32)))

r[0] = 0. Otherwise read the actual value.

definition *user-reg-val*:: ('a) window-size \Rightarrow user-reg-type \Rightarrow ('a) sparc-state \Rightarrow
reg-type
where
user-reg-val window ur state \equiv
if ur = 0 then 0
else (*user-reg* state) window ur

Write a global register. win should be initialised as NWINDOWS.

fun (*sequential*) *global-reg-mod* :: word32 \Rightarrow nat \Rightarrow user-reg-type \Rightarrow
('a::len) sparc-state \Rightarrow ('a) sparc-state
where
global-reg-mod data-w32 0 ur state = state
|
global-reg-mod data-w32 win ur state = (
let win-word = word-of-int (int (win-1));
ns = state(*user-reg* :=
(*user-reg* state)(win-word := ((*user-reg* state) win-word)(ur := data-w32)))

in
global-reg-mod data-w32 (win-1) ur ns
)

Compute the next window.

definition *next-window* :: ('a::len) window-size \Rightarrow ('a) window-size
where
next-window win \equiv
if (uint win) < (NWINDOWS - 1) then (win + 1)
else 0

Compute the previous window.

definition *pre-window* :: ('a::len) window-size \Rightarrow ('a::len) window-size
where
pre-window win \equiv
if (uint win) > 0 then (win - 1)
else (word-of-int (NWINDOWS - 1))

write an output register. Also write $ur+16$ of the previous window.

definition $out\text{-}reg\text{-}mod :: word32 \Rightarrow ('a::len) \text{ window-size} \Rightarrow user\text{-}reg\text{-}type \Rightarrow ('a) \text{ sparc-state} \Rightarrow ('a) \text{ sparc-state}$

where

```

out-reg-mod data-w32 win ur state ≡
  let state' = state(user-reg :=
    (user-reg state)(win := ((user-reg state) win)(ur := data-w32)));
    win' = pre-window win;
    ur' = ur + 16
  in
  state'(user-reg :=
    (user-reg state')(win' := ((user-reg state') win')(ur' := data-w32)))

```

Write a input register. Also write $ur-16$ of the next window.

definition $in\text{-}reg\text{-}mod :: word32 \Rightarrow ('a::len) \text{ window-size} \Rightarrow user\text{-}reg\text{-}type \Rightarrow ('a) \text{ sparc-state} \Rightarrow ('a) \text{ sparc-state}$

where

```

in-reg-mod data-w32 win ur state ≡
  let state' = state(user-reg :=
    (user-reg state)(win := ((user-reg state) win)(ur := data-w32)));
    win' = next-window win;
    ur' = ur - 16
  in
  state'(user-reg :=
    (user-reg state')(win' := ((user-reg state') win')(ur' := data-w32)))

```

Do not modify $r[0]$.

definition $user\text{-}reg\text{-}mod :: word32 \Rightarrow ('a::len) \text{ window-size} \Rightarrow user\text{-}reg\text{-}type \Rightarrow ('a) \text{ sparc-state} \Rightarrow ('a) \text{ sparc-state}$

where

```

user-reg-mod data-w32 win ur state ≡
  if ur = 0 then state
  else if 0 < ur ∧ ur < 8 then
    global-reg-mod data-w32 (nat NWINDOWS) ur state
  else if 7 < ur ∧ ur < 16 then
    out-reg-mod data-w32 win ur state
  else if 15 < ur ∧ ur < 24 then
    state(user-reg :=
      (user-reg state)(win := ((user-reg state) win)(ur := data-w32)))
  else if 23 < ur ∧ ur < 32 then
    in-reg-mod data-w32 win ur state
  else state

```

definition $sys\text{-}reg\text{-}val :: sys\text{-}reg \Rightarrow ('a) \text{ sparc-state} \Rightarrow reg\text{-}type$

where

```

sys-reg-val reg state ≡ (sys-reg state) reg

```

definition *sys-reg-mod* :: *word32* \Rightarrow *sys-reg* \Rightarrow
 (*'a*) *sparc-state* \Rightarrow (*'a*) *sparc-state*

where

sys-reg-mod data-w32 sys state \equiv *state*(\setminus *sys-reg* := (*sys-reg state*)(*sys* := *data-w32*))

The following functions deal with physical memory. N.B. Physical memory address in SPARCv8 is 36-bit.

LEON3 doesn't distinguish ASI 8 and 9; 10 and 11 for read access for both user and supervisor. We recently discovered that the compiled machine code by the *sparc-elf* compiler often reads *asi* = 10 (user data) when the actual content is store in *asi* = 8 (user instruction). For testing purposes, we don't distinguish *asi* = 8,9,10,11 for reading access.

definition *mem-val*:: *asi-type* \Rightarrow *phys-address* \Rightarrow
 (*'a*) *sparc-state* \Rightarrow *mem-val-type option*

where

mem-val asi add state \equiv
 let *asi8* = *word-of-int* 8;
 asi9 = *word-of-int* 9;
 asi10 = *word-of-int* 10;
 asi11 = *word-of-int* 11;
 r1 = (*mem state*) *asi8 add*
 in
 if *r1* = *None* *then*
 let *r2* = (*mem state*) *asi9 add in*
 if *r2* = *None* *then*
 let *r3* = (*mem state*) *asi10 add in*
 if *r3* = *None* *then*
 (*mem state*) *asi11 add*
 else *r3*
 else *r2*
 else *r1*

An alternative way to read values from memory. Some implementations may use this definition.

definition *mem-val-alt*:: *asi-type* \Rightarrow *phys-address* \Rightarrow
 (*'a*) *sparc-state* \Rightarrow *mem-val-type option*

where

mem-val-alt asi add state \equiv
 let *r1* = (*mem state*) *asi add*;
 asi8 = *word-of-int* 8;
 asi9 = *word-of-int* 9;
 asi10 = *word-of-int* 10;
 asi11 = *word-of-int* 11
 in
 if *r1* = *None* \wedge (*uint asi*) = 8 *then*

```

    let r2 = (mem state) asi9 add in
    r2
  else if r1 = None ∧ (uint asi) = 9 then
    let r2 = (mem state) asi8 add in
    r2
  else if r1 = None ∧ (uint asi) = 10 then
    let r2 = (mem state) asi11 add in
    if r2 = None then
      let r3 = (mem state) asi8 add in
      if r3 = None then
        (mem state) asi9 add
      else r3
    else r2
  else if r1 = None ∧ (uint asi) = 11 then
    let r2 = (mem state) asi10 add in
    if r2 = None then
      let r3 = (mem state) asi8 add in
      if r3 = None then
        (mem state) asi9 add
      else r3
    else r2
  else r1

```

definition *mem-mod* :: *asi-type* ⇒ *phys-address* ⇒ *mem-val-type* ⇒
 ('a) *sparc-state* ⇒ ('a) *sparc-state*

where

```

mem-mod asi addr val state ≡
  let state1 = state(\mem := (mem state)
    (asi := ((mem state) asi)(addr := Some val)))
  in — Only allow one of asi 8 and 9 (10 and 11) to have value.
  if (uint asi) = 8 ∨ (uint asi) = 10 then
    let asi2 = word-of-int ((uint asi) + 1) in
    state1(\mem := (mem state1)
      (asi2 := ((mem state1) asi2)(addr := None)))
  else if (uint asi) = 9 ∨ (uint asi) = 11 then
    let asi2 = word-of-int ((uint asi) - 1) in
    state1(\mem := (mem state1)(asi2 := ((mem state1) asi2)(addr := None)))
  else state1

```

An alternative way to write memory. This method insists that for each address, it can only hold a value in one of ASI = 8,9,10,11.

definition *mem-mod-alt* :: *asi-type* ⇒ *phys-address* ⇒ *mem-val-type* ⇒
 ('a) *sparc-state* ⇒ ('a) *sparc-state*

where

```

mem-mod-alt asi addr val state ≡
  let state1 = state(\mem := (mem state)
    (asi := ((mem state) asi)(addr := Some val)));
  asi8 = word-of-int 8;

```

```

    asi9 = word-of-int 9;
    asi10 = word-of-int 10;
    asi11 = word-of-int 11
in
— Only allow one of asi 8, 9, 10, 11 to have value.
if (uint asi) = 8 then
    let state2 = state1(|mem := (mem state1)
        (asi9 := ((mem state1) asi9)(addr := None)));
    state3 = state2(|mem := (mem state2)
        (asi10 := ((mem state2) asi10)(addr := None)));
    state4 = state3(|mem := (mem state3)
        (asi11 := ((mem state3) asi11)(addr := None)))
    in
    state4
else if (uint asi) = 9 then
    let state2 = state1(|mem := (mem state1)
        (asi8 := ((mem state1) asi8)(addr := None)));
    state3 = state2(|mem := (mem state2)
        (asi10 := ((mem state2) asi10)(addr := None)));
    state4 = state3(|mem := (mem state3)
        (asi11 := ((mem state3) asi11)(addr := None)))
    in
    state4
else if (uint asi) = 10 then
    let state2 = state1(|mem := (mem state1)
        (asi9 := ((mem state1) asi9)(addr := None)));
    state3 = state2(|mem := (mem state2)
        (asi8 := ((mem state2) asi8)(addr := None)));
    state4 = state3(|mem := (mem state3)
        (asi11 := ((mem state3) asi11)(addr := None)))
    in
    state4
else if (uint asi) = 11 then
    let state2 = state1(|mem := (mem state1)
        (asi9 := ((mem state1) asi9)(addr := None)));
    state3 = state2(|mem := (mem state2)
        (asi10 := ((mem state2) asi10)(addr := None)));
    state4 = state3(|mem := (mem state3)
        (asi8 := ((mem state3) asi8)(addr := None)))
    in
    state4
else state1

```

context

includes *bit-operations-syntax*

begin

Given an ASI (word8), an address (word32) addr, read the 32bit value from


```

        mem-mod asi addr0 byte0 state
    else state;
s1 = if (((AND) byte-mask (0b0100::word4)) >> 2) = 1 then
    mem-mod asi addr1 byte1 s0
    else s0;
s2 = if (((AND) byte-mask (0b0010::word4)) >> 1) = 1 then
    mem-mod asi addr2 byte2 s1
    else s1;
s3 = if ((AND) byte-mask (0b0001::word4)) = 1 then
    mem-mod asi addr3 byte3 s2
    else s2
in
s3

```

The following functions deal with virtual addresses. These are based on functions written by David Sanan.

definition *load-word-mem* :: ('a) sparc-state ⇒ virtua-address ⇒ asi-type ⇒ machine-word option

where *load-word-mem* state va asi ≡
let pair = (virt-to-phys va (mmu state) (mem state)) in
case pair of
Some pair ⇒ (
if mmu-readable (get-acc-flag (snd pair)) asi then
(mem-val-w32 asi (fst pair) state)
else None)
| None ⇒ None

definition *store-word-mem* :: ('a) sparc-state ⇒ virtua-address ⇒ machine-word ⇒

word4 ⇒ asi-type ⇒ ('a) sparc-state option

where *store-word-mem* state va wd byte-mask asi ≡
let pair = (virt-to-phys va (mmu state) (mem state)) in
case pair of
Some pair ⇒ (
if mmu-writable (get-acc-flag (snd pair)) asi then
Some (mem-mod-w32 asi (fst pair) byte-mask wd state)
else None)
| None ⇒ None

definition *icache-val*:: cache-type ⇒ ('a) sparc-state ⇒ mem-val-type option

where *icache-val* c state ≡ icache (cache state) c

definition *dcache-val*:: cache-type ⇒ ('a) sparc-state ⇒ mem-val-type option

where *dcache-val* c state ≡ dcache (cache state) c

definition *icache-mod* :: cache-type ⇒ mem-val-type ⇒

('a) sparc-state ⇒ ('a) sparc-state

where *icache-mod* c val state ≡

```
state(|cache := ((cache state)
  (|icache := (icache (cache state))(c := Some val)|))|)
```

definition *dcache-mod* :: *cache-type* \Rightarrow *mem-val-type* \Rightarrow
 ('a) *sparc-state* \Rightarrow ('a) *sparc-state*
where *dcache-mod* c val state \equiv
 state(|cache := ((cache state)
 (|dcache := (dcache (cache state))(c := Some val)|))|)

Check if the memory address is in the cache or not.

definition *icache-miss* :: *virtua-address* \Rightarrow ('a) *sparc-state* \Rightarrow *bool*
where
icache-miss addr state \equiv
 let line-len = 12;
 tag = (ucast (addr >> line-len))::cache-tag;
 line = (ucast (0b0::word1))::cache-line-size
 in
 if (icache-val (tag,line) state) = None then True
 else False

Check if the memory address is in the cache or not.

definition *dcache-miss* :: *virtua-address* \Rightarrow ('a) *sparc-state* \Rightarrow *bool*
where
dcache-miss addr state \equiv
 let line-len = 12;
 tag = (ucast (addr >> line-len))::cache-tag;
 line = (ucast (0b0::word1))::cache-line-size
 in
 if (dcache-val (tag,line) state) = None then True
 else False

definition *read-data-cache*:: ('a) *sparc-state* \Rightarrow *virtua-address* \Rightarrow *machine-word*
option

where *read-data-cache* state va \equiv
 let tag = (ucast (va >> 12))::word20;
 offset0 = (AND) ((ucast va)::word12) 0b111111111100;
 offset1 = (OR) offset0 0b000000000001;
 offset2 = (OR) offset0 0b000000000010;
 offset3 = (OR) offset0 0b000000000011;
 r0 = dcache-val (tag,offset0) state;
 r1 = dcache-val (tag,offset1) state;
 r2 = dcache-val (tag,offset2) state;
 r3 = dcache-val (tag,offset3) state
 in
 if r0 = None \vee r1 = None \vee r2 = None \vee r3 = None then

```

None
else
  let byte0 = case r0 of Some v ⇒ v;
      byte1 = case r1 of Some v ⇒ v;
      byte2 = case r2 of Some v ⇒ v;
      byte3 = case r3 of Some v ⇒ v
  in
  Some ((OR) ((OR) ((OR) ((ucast(byte0)) << 24)
                      ((ucast(byte1)) << 16))
              ((ucast(byte2)) << 8))
        (ucast(byte3))))

```

definition *read-instr-cache*:: ('a) sparc-state ⇒ virtua-address ⇒ machine-word option

where *read-instr-cache state va* ≡

```

let tag = (ucast (va >> 12))::word20;
  offset0 = (AND) ((ucast va)::word12) 0b111111111100;
  offset1 = (OR) offset0 0b000000000001;
  offset2 = (OR) offset0 0b000000000010;
  offset3 = (OR) offset0 0b000000000011;
  r0 = icache-val (tag,offset0) state;
  r1 = icache-val (tag,offset1) state;
  r2 = icache-val (tag,offset2) state;
  r3 = icache-val (tag,offset3) state
in
if r0 = None ∨ r1 = None ∨ r2 = None ∨ r3 = None then
  None
else
  let byte0 = case r0 of Some v ⇒ v;
      byte1 = case r1 of Some v ⇒ v;
      byte2 = case r2 of Some v ⇒ v;
      byte3 = case r3 of Some v ⇒ v
  in
  Some ((OR) ((OR) ((OR) ((ucast(byte0)) << 24)
                      ((ucast(byte1)) << 16))
              ((ucast(byte2)) << 8))
        (ucast(byte3))))

```

definition *add-data-cache* :: ('a) sparc-state ⇒ virtua-address ⇒ machine-word ⇒

word4 ⇒ ('a) sparc-state

where *add-data-cache state va word byte-mask* ≡

```

let tag = (ucast (va >> 12))::word20;
  offset0 = (AND) ((ucast va)::word12) 0b111111111100;
  offset1 = (OR) offset0 0b000000000001;
  offset2 = (OR) offset0 0b000000000010;

```

```

offset3 = (OR) offset0 0b000000000011;
byte0 = (ucast (word >> 24))::mem-val-type;
byte1 = (ucast (word >> 16))::mem-val-type;
byte2 = (ucast (word >> 8))::mem-val-type;
byte3 = (ucast word)::mem-val-type;
s0 = if (((AND) byte-mask (0b1000::word4)) >> 3) = 1 then
      dcache-mod (tag,offset0) byte0 state
    else state;
s1 = if (((AND) byte-mask (0b0100::word4)) >> 2) = 1 then
      dcache-mod (tag,offset1) byte1 s0
    else s0;
s2 = if (((AND) byte-mask (0b0010::word4)) >> 1) = 1 then
      dcache-mod (tag,offset2) byte2 s1
    else s1;
s3 = if ((AND) byte-mask (0b0001::word4)) = 1 then
      dcache-mod (tag,offset3) byte3 s2
    else s2
in s3

```

definition *add-instr-cache* :: ('a) *sparc-state* \Rightarrow *virtua-address* \Rightarrow *machine-word*
 \Rightarrow

word4 \Rightarrow ('a) *sparc-state*

where

```

add-instr-cache state va word byte-mask  $\equiv$ 
let tag = (ucast (va >> 12))::word20;
offset0 = (AND) ((ucast va)::word12) 0b111111111100;
offset1 = (OR) offset0 0b000000000001;
offset2 = (OR) offset0 0b000000000010;
offset3 = (OR) offset0 0b000000000011;
byte0 = (ucast (word >> 24))::mem-val-type;
byte1 = (ucast (word >> 16))::mem-val-type;
byte2 = (ucast (word >> 8))::mem-val-type;
byte3 = (ucast word)::mem-val-type;
s0 = if (((AND) byte-mask (0b1000::word4)) >> 3) = 1 then
      icache-mod (tag,offset0) byte0 state
    else state;
s1 = if (((AND) byte-mask (0b0100::word4)) >> 2) = 1 then
      icache-mod (tag,offset1) byte1 s0
    else s0;
s2 = if (((AND) byte-mask (0b0010::word4)) >> 1) = 1 then
      icache-mod (tag,offset2) byte2 s1
    else s1;
s3 = if ((AND) byte-mask (0b0001::word4)) = 1 then
      icache-mod (tag,offset3) byte3 s2
    else s2
in s3

```

definition *empty-cache* :: *cache-context* **where** *empty-cache* *c* \equiv *None*

definition *flush-data-cache*:: ('*a*) *sparc-state* \Rightarrow ('*a*) *sparc-state* **where**
flush-data-cache *state* \equiv *state*(*cache* := ((*cache state*)(*dcache* := *empty-cache*)))

definition *flush-instr-cache*:: ('*a*) *sparc-state* \Rightarrow ('*a*) *sparc-state* **where**
flush-instr-cache *state* \equiv *state*(*cache* := ((*cache state*)(*icache* := *empty-cache*)))

definition *flush-cache-all*:: ('*a*) *sparc-state* \Rightarrow ('*a*) *sparc-state* **where**
flush-cache-all *state* \equiv *state*(*cache* := ((*cache state*)(
icache := *empty-cache*, *dcache* := *empty-cache*)))

Check if the FI or FD bit of CCR is 1. If FI is 1 then flush instruction cache.
If FD is 1 then flush data cache.

definition *ccr-flush* :: ('*a*) *sparc-state* \Rightarrow ('*a*) *sparc-state*
where

ccr-flush *state* \equiv
let *ccr-val* = *sys-reg-val* *CCR* *state*;
— *FI* is bit 21 of *CCR*
fi-val = ((*AND*) *ccr-val* (0b0000000001000000000000000000)) >> 21;
fd-val = ((*AND*) *ccr-val* (0b0000000001000000000000000000)) >> 22;
state1 = (if *fi-val* = 1 then *flush-instr-cache* *state* else *state*)
in
if *fd-val* = 1 then *flush-data-cache* *state1* else *state1*

definition *get-delayed-pool* :: ('*a*) *sparc-state* \Rightarrow *delayed-write-pool*
where *get-delayed-pool* *state* \equiv *dwrite* *state*

definition *exe-pool* :: (*int* \times *reg-type* \times *CPU-register*) \Rightarrow (*int* \times *reg-type* \times *CPU-register*)
where *exe-pool* *w* \equiv *case* *w* of (*n,v,c*) \Rightarrow ((*n-1*),*v,c*)

Minus 1 to the delayed count for all the members in the set. Assuming all members have delay > 0.

primrec *delayed-pool-minus* :: *delayed-write-pool* \Rightarrow *delayed-write-pool*
where

delayed-pool-minus [] = []
|
delayed-pool-minus (*x#xs*) = (*exe-pool* *x*)#(*delayed-pool-minus* *xs*)

Add a delayed-write to the pool.

definition *delayed-pool-add* :: (*int* \times *reg-type* \times *CPU-register*) \Rightarrow
('*a*) *sparc-state* \Rightarrow ('*a*) *sparc-state*

where

delayed-pool-add *dw* *s* \equiv
let (*i,v,cr*) = *dw* in
if *i* = 0 then — Write the value to the register immediately.
cpu-reg-mod *v* *cr* *s*
else — Add to delayed write pool.

let curr-pool = get-delayed-pool s in
s(dwwrite := curr-pool@[dw])

Remove a delayed-write from the pool. Assume that the delayed-write to be removed has delay 0. i.e., it has been executed.

definition *delayed-pool-rm* :: (int × reg-type × CPU-register) ⇒ ('a) sparc-state ⇒ ('a) sparc-state

where

delayed-pool-rm dw s ≡
let curr-pool = get-delayed-pool s in
case dw of (n,v,cr) ⇒
(if n = 0 then
s(dwwrite := List.remove1 dw curr-pool)
else s)

Remove all the entries with delay = 0, i.e., those that are written.

primrec *delayed-pool-rm-written* :: delayed-write-pool ⇒ delayed-write-pool

where

delayed-pool-rm-written [] = []
|
delayed-pool-rm-written (x#xs) =
(if fst x = 0 then *delayed-pool-rm-written* xs else x#(*delayed-pool-rm-written* xs))

definition *annul-val* :: ('a) sparc-state ⇒ bool

where *annul-val* state ≡ get-annul (state-var state)

definition *annul-mod* :: bool ⇒ ('a) sparc-state ⇒ ('a) sparc-state

where *annul-mod* b s ≡ s(state-var := write-annul b (state-var s))

definition *reset-trap-val* :: ('a) sparc-state ⇒ bool

where *reset-trap-val* state ≡ get-reset-trap (state-var state)

definition *reset-trap-mod* :: bool ⇒ ('a) sparc-state ⇒ ('a) sparc-state

where *reset-trap-mod* b s ≡ s(state-var := write-reset-trap b (state-var s))

definition *exe-mode-val* :: ('a) sparc-state ⇒ bool

where *exe-mode-val* state ≡ get-exe-mode (state-var state)

definition *exe-mode-mod* :: bool ⇒ ('a) sparc-state ⇒ ('a) sparc-state

where *exe-mode-mod* b s ≡ s(state-var := write-exe-mode b (state-var s))

definition *reset-mode-val* :: ('a) sparc-state ⇒ bool

where *reset-mode-val* state ≡ get-reset-mode (state-var state)

definition *reset-mode-mod* :: bool ⇒ ('a) sparc-state ⇒ ('a) sparc-state

where *reset-mode-mod* b s ≡ s(state-var := write-reset-mode b (state-var s))

definition *err-mode-val* :: ('a) sparc-state \Rightarrow bool
where *err-mode-val* state \equiv get-err-mode (state-var state)

definition *err-mode-mod* :: bool \Rightarrow ('a) sparc-state \Rightarrow ('a) sparc-state
where *err-mode-mod* b s \equiv s(|state-var := write-err-mode b (state-var s)|)

definition *ticc-trap-type-val* :: ('a) sparc-state \Rightarrow word7
where *ticc-trap-type-val* state \equiv get-ticc-trap-type (state-var state)

definition *ticc-trap-type-mod* :: word7 \Rightarrow ('a) sparc-state \Rightarrow ('a) sparc-state
where *ticc-trap-type-mod* w s \equiv s(|state-var := write-ticc-trap-type w (state-var s)|)

definition *interrupt-level-val* :: ('a) sparc-state \Rightarrow word3
where *interrupt-level-val* state \equiv get-interrupt-level (state-var state)

definition *interrupt-level-mod* :: word3 \Rightarrow ('a) sparc-state \Rightarrow ('a) sparc-state
where *interrupt-level-mod* w s \equiv s(|state-var := write-interrupt-level w (state-var s)|)

definition *store-barrier-pending-val* :: ('a) sparc-state \Rightarrow bool
where *store-barrier-pending-val* state \equiv
get-store-barrier-pending (state-var state)

definition *store-barrier-pending-mod* :: bool \Rightarrow
('a) sparc-state \Rightarrow ('a) sparc-state
where *store-barrier-pending-mod* w s \equiv
s(|state-var := write-store-barrier-pending w (state-var s)|)

definition *pb-block-ldst-byte-val* :: virtua-address \Rightarrow ('a) sparc-state
 \Rightarrow bool
where *pb-block-ldst-byte-val* add state \equiv
(atm-ldst-byte (state-var state)) add

definition *pb-block-ldst-byte-mod* :: virtua-address \Rightarrow bool \Rightarrow
('a) sparc-state \Rightarrow ('a) sparc-state
where *pb-block-ldst-byte-mod* add b s \equiv
s(|state-var := ((state-var s)
(atm-ldst-byte := (atm-ldst-byte (state-var s))(add := b))|)

We only read the address such that add mod 4 = 0. add mod 4 represents the current word.

definition *pb-block-ldst-word-val* :: virtua-address \Rightarrow ('a) sparc-state
 \Rightarrow bool
where *pb-block-ldst-word-val* add state \equiv
let add0 = ((AND) add (0b11111111111111111111111111111100::word32)) in
(atm-ldst-word (state-var state)) add0

We only write the address such that add mod 4 = 0. add mod 4 represents

the current word.

definition *pb-block-ldst-word-mod* :: *virtua-address* \Rightarrow *bool* \Rightarrow
('a) sparc-state \Rightarrow *('a) sparc-state*
where *pb-block-ldst-word-mod* *add b s* \equiv
let add0 = ((AND) add (0b11111111111111111111111111111100::word32)) in
s(|state-var := ((state-var s)
(atm-ldst-word := (atm-ldst-word (state-var s))(add0 := b)))|)

definition *get-trap-set* :: *('a) sparc-state* \Rightarrow *Trap set*
where *get-trap-set* *state* \equiv (*traps state*)

definition *add-trap-set* :: *Trap* \Rightarrow *('a) sparc-state* \Rightarrow *('a) sparc-state*
where *add-trap-set* *t s* \equiv *s(|traps := (traps s) \cup {t}|)*

definition *emp-trap-set* :: *('a) sparc-state* \Rightarrow *('a) sparc-state*
where *emp-trap-set* *s* \equiv *s(|traps := {}|)*

definition *state-undef*:: *('a) sparc-state* \Rightarrow *bool*
where *state-undef* *state* \equiv (*undef state*)

The *memory-read* interface that conforms with the SPARCv8 manual.

definition *memory-read* :: *asi-type* \Rightarrow *virtua-address* \Rightarrow
('a) sparc-state \Rightarrow
((word32 option) \times ('a) sparc-state)
where *memory-read* *asi addr state* \equiv
let asi-int = uint asi in — See Page 25 and 35 for ASI usage in LEON 3FT.
if asi-int = 1 then — Forced cache miss.
— Directly read from memory.
let r1 = load-word-mem state addr (word-of-int 8) in
if r1 = None then
let r2 = load-word-mem state addr (word-of-int 10) in
if r2 = None then
(None, state)
else (r2, state)
else (r1, state)
else if asi-int = 2 then — System registers.
— See Table 19, Page 34 for System Register address map in LEON 3FT.
if uint addr = 0 then — Cache control register.
((Some (sys-reg-val CCR state)), state)
else if uint addr = 8 then — Instruction cache configuration register.
((Some (sys-reg-val ICCR state)), state)
else if uint addr = 12 then — Data cache configuration register.
((Some (sys-reg-val DCCR state)), state)
else — Invalid address.
(None, state)
else if asi-int \in {8,9} then — Access instruction memory.
let ccr-val = (sys-reg state) CCR in
if ccr-val AND 1 \neq 0 then — Cache is enabled. Update cache.
— We don't go through the tradition, i.e., read from cache first,

- if the address is not cached, then read from memory,
- because performance is not an issue here.
- Thus we directly read from memory and update the cache.
- let data = load-word-mem state addr asi in*
- case data of*
- Some w ⇒ (Some w, (add-instr-cache state addr w (0b1111::word4)))*
- |None ⇒ (None, state)*
- else* — Cache is disabled. Just read from memory.
- ((load-word-mem state addr asi), state)*
- else if asi-int ∈ {10,11} then* — Access data memory.
- let ccr-val = (sys-reg state) CCR in*
- if ccr-val AND 1 ≠ 0 then* — Cache is enabled. Update cache.
- We don't go through the tradition, i.e., read from cache first,
- if the address is not cached, then read from memory,
- because performance is not an issue here.
- Thus we directly read from memory and update the cache.
- let data = load-word-mem state addr asi in*
- case data of*
- Some w ⇒ (Some w, (add-data-cache state addr w (0b1111::word4)))*
- |None ⇒ (None, state)*
- else* — Cache is disabled. Just read from memory.
- ((load-word-mem state addr asi), state)*
- We don't access instruction cache tag. i.e., *asi = 12*.
- else if asi-int = 13 then* — Read instruction cache data.
- let cache-result = read-instr-cache state addr in*
- case cache-result of*
- Some w ⇒ (Some w, state)*
- |None ⇒ (None, state)*
- We don't access data cache tag. i.e., *asi = 14*.
- else if asi-int = 15 then* — Read data cache data.
- let cache-result = read-data-cache state addr in*
- case cache-result of*
- Some w ⇒ (Some w, state)*
- |None ⇒ (None, state)*
- else if asi-int ∈ {16,17} then* — Flush entire instruction/data cache.
- (None, state)* — Has no effect for memory read.
- else if asi-int ∈ {20,21} then* — MMU diagnostic cache access.
- (None, state)* — Not considered in this model.
- else if asi-int = 24 then* — Flush cache and TLB in LEON3.
- But is not used for memory read.
- (None, state)*
- else if asi-int = 25 then* — MMU registers.
- Treat MMU registers as memory addresses that are not in the main memory.
- ((mmu-reg-val (mmu state) addr), state)*
- else if asi-int = 28 then* — MMU bypass.
- Directly use *addr* as a physical address.
- Append 0000 in the front of *addr*.
- In this case, *(ucast addr)* suffices.
- ((mem-val-w32 asi (ucast addr) state), state)*

— Assuming writing into $asi = 10$.
store-word-mem state addr data-w32 byte-mask (word-of-int 10)
 else if $asi-int = 2$ then — System registers.
 — See Table 19, Page 34 for System Register address map in LEON 3FT.
 if $uint\ addr = 0$ then — Cache control register.
 let s1 = (sys-reg-mod data-w32 CCR state) in
 — Flush the instruction cache if FI of CCR is 1;
 — flush the data cache if FD of CCR is 1.
 Some (ccr-flush s1)
 else if $uint\ addr = 8$ then — Instruction cache configuration register.
 Some (sys-reg-mod data-w32 ICCR state)
 else if $uint\ addr = 12$ then — Data cache configuration register.
 Some (sys-reg-mod data-w32 DCCR state)
 else — Invalid address.
 None
 else if $asi-int \in \{8,9\}$ then — Access instruction memory.
 — Write to memory. LEON3 does write-through. Both cache and the memory are updated.
 let ns = add-instr-cache state addr data-w32 byte-mask in
 store-word-mem ns addr data-w32 byte-mask asi
 else if $asi-int \in \{10,11\}$ then — Access data memory.
 — Write to memory. LEON3 does write-through. Both cache and the memory are updated.
 let ns = add-data-cache state addr data-w32 byte-mask in
 store-word-mem ns addr data-w32 byte-mask asi
 — We don't access instruction cache tag. i.e., $asi = 12$.
 else if $asi-int = 13$ then — Write instruction cache data.
 Some (add-instr-cache state addr data-w32 (0b1111::word4))
 — We don't access data cache tag. i.e., $asi = 14$.
 else if $asi-int = 15$ then — Write data cache data.
 Some (add-data-cache state addr data-w32 (0b1111::word4))
 else if $asi-int = 16$ then — Flush instruction cache.
 Some (flush-instr-cache state)
 else if $asi-int = 17$ then — Flush data cache.
 Some (flush-data-cache state)
 else if $asi-int \in \{20,21\}$ then — MMU diagnostic cache access.
 None — Not considered in this model.
 else if $asi-int = 24$ then — Flush TLB and cache in LEON3.
 — We don't consider TLB here.
 Some (flush-cache-all state)
 else if $asi-int = 25$ then — MMU registers.
 — Treat MMU registers as memory addresses that are not in the main memory.
 let mmu-state' = mmu-reg-mod (mmu state) addr data-w32 in
 case mmu-state' of
 Some mmus \Rightarrow Some (state(|mmu := mmus))
 |None \Rightarrow None
 else if $asi-int = 28$ then — MMU bypass.
 — Write to virtual address as physical address.
 — Append 0000 in front of $addr$.

Some (mem-mod-w32 asi (ucast addr) byte-mask data-w32 state)
else if asi-int = 29 then — MMU diagnostic access.
None — Not considered in this model.
else — Not considered in this model.
None

definition *memory-write* :: *asi-type* ⇒ *virtua-address* ⇒ *word4* ⇒ *word32* ⇒
 (*'a*) *sparc-state* ⇒
 (*'a*) *sparc-state option*

where

memory-write asi addr byte-mask data-w32 state ≡
let result = memory-write-asi asi addr byte-mask data-w32 state in
case result of
None ⇒ None
| Some s1 ⇒ Some (store-barrier-pending-mod False s1)

monad for sequential operations over the register representation

type-synonym (*'a,'e*) *sparc-state-monad* = ((*'a*) *sparc-state,'e*) *det-monad*

Given a *word32* value, a cpu register, write the value in the cpu register.

definition *write-cpu* :: *word32* ⇒ *CPU-register* ⇒ (*'a,unit*) *sparc-state-monad*

where *write-cpu w cr* ≡
do
 modify (λs. (cpu-reg-mod w cr s));
 return ()
od

definition *write-cpu-tt* :: *word8* ⇒ (*'a,unit*) *sparc-state-monad*

where *write-cpu-tt w* ≡
do
 tbr-val ← gets (λs. (cpu-reg-val TBR s));
 new-tbr-val ← gets (λs. (write-tt w tbr-val));
 write-cpu new-tbr-val TBR;
 return ()
od

Given a *word32* value, a *word4* window, a user register, write the value in the user register. N.B. CWP is a 5 bit value, but we only use the last 4 bits, since there are only 16 windows.

definition *write-reg* :: *word32* ⇒ (*'a::len*) *word* ⇒ *user-reg-type* ⇒

 (*'a,unit*) *sparc-state-monad*
where *write-reg w win ur* ≡
do
 modify (λs.(user-reg-mod w win ur s));
 return ()
od

definition *set-annul* :: *bool* ⇒ ('a,unit) *sparc-state-monad*

where *set-annul* *b* ≡

```
do
  modify (λs. (annul-mod b s));
  return ()
od
```

definition *set-reset-trap* :: *bool* ⇒ ('a,unit) *sparc-state-monad*

where *set-reset-trap* *b* ≡

```
do
  modify (λs. (reset-trap-mod b s));
  return ()
od
```

definition *set-exe-mode* :: *bool* ⇒ ('a,unit) *sparc-state-monad*

where *set-exe-mode* *b* ≡

```
do
  modify (λs. (exe-mode-mod b s));
  return ()
od
```

definition *set-reset-mode* :: *bool* ⇒ ('a,unit) *sparc-state-monad*

where *set-reset-mode* *b* ≡

```
do
  modify (λs. (reset-mode-mod b s));
  return ()
od
```

definition *set-err-mode* :: *bool* ⇒ ('a,unit) *sparc-state-monad*

where *set-err-mode* *b* ≡

```
do
  modify (λs. (err-mode-mod b s));
  return ()
od
```

fun *get-delayed-0* :: (*int* × *reg-type* × *CPU-register*) *list* ⇒

(*int* × *reg-type* × *CPU-register*) *list*

where

get-delayed-0 [] = []

|

get-delayed-0 (*x* # *xs*) =

(if *fst* *x* = 0 then *x* # (*get-delayed-0* *xs*)

else *get-delayed-0* *xs*)

Get a list of delayed-writes with delay 0.

definition *get-delayed-write* :: *delayed-write-pool* ⇒ (*int* × *reg-type* × *CPU-register*)

list

where

get-delayed-write *dwp* ≡ *get-delayed-0* *dwp*

definition *delayed-write* :: (int × reg-type × CPU-register) ⇒ ('a) sparc-state ⇒ ('a) sparc-state
where *delayed-write* dw s ≡
 let (n,v,r) = dw in
 if n = 0 then
 cpu-reg-mod v r s
 else s

primrec *delayed-write-all* :: (int × reg-type × CPU-register) list ⇒ ('a) sparc-state ⇒ ('a) sparc-state
where *delayed-write-all* [] s = s
| *delayed-write-all* (x # xs) s =
delayed-write-all xs (*delayed-write* x s)

primrec *delayed-pool-rm-list* :: (int × reg-type × CPU-register) list ⇒ ('a) sparc-state ⇒ ('a) sparc-state
where *delayed-pool-rm-list* [] s = s
| *delayed-pool-rm-list* (x # xs) s =
delayed-pool-rm-list xs (*delayed-pool-rm* x s)

definition *delayed-pool-write* :: ('a) sparc-state ⇒ ('a) sparc-state
where *delayed-pool-write* s ≡
 let dwp0 = *get-delayed-pool* s;
 dwp1 = *delayed-pool-minus* dwp0;
 wl = *get-delayed-write* dwp1;
 s1 = *delayed-write-all* wl s;
 s2 = *delayed-pool-rm-list* wl s1
 in s2

definition *raise-trap* :: Trap ⇒ ('a,unit) sparc-state-monad
where *raise-trap* t ≡
 do
 modify (λs. (*add-trap-set* t s));
 return ()
 od

end

end

23 SPARC instruction model

theory *Sparc-Instruction*
imports *Main Sparc-Types Sparc-State HOL-Eisbach.Eisbach-Tools*
begin

This theory provides a formal model for assembly instruction to be executed in the model.

An instruction is defined as a tuple composed of a *sparc-operation* element, defining the operation the instruction carries out, and a list of operands *inst-operand*. *inst-operand* can be a user register *user-reg* or a memory address *mem-add-type*.

```
datatype inst-operand =
  W5 word5
| W30 word30
| W22 word22
| Cond word4
| Flag word1
| Asi asi-type
| Simm13 word13
| Opf word9
| Imm7 word7
```

```
primrec get-operand-w5::inst-operand  $\Rightarrow$  word5
where get-operand-w5 (W5 r) = r
```

```
primrec get-operand-w30::inst-operand  $\Rightarrow$  word30
where get-operand-w30 (W30 r) = r
```

```
primrec get-operand-w22::inst-operand  $\Rightarrow$  word22
where get-operand-w22 (W22 r) = r
```

```
primrec get-operand-cond::inst-operand  $\Rightarrow$  word4
where get-operand-cond (Cond r) = r
```

```
primrec get-operand-flag::inst-operand  $\Rightarrow$  word1
where get-operand-flag (Flag r) = r
```

```
primrec get-operand-asi::inst-operand  $\Rightarrow$  asi-type
where get-operand-asi (Asi r) = r
```

```
primrec get-operand-simm13::inst-operand  $\Rightarrow$  word13
where get-operand-simm13 (Simm13 r) = r
```

```
primrec get-operand-opf::inst-operand  $\Rightarrow$  word9
where get-operand-opf (Opf r) = r
```

```
primrec get-operand-imm7::inst-operand  $\Rightarrow$  word7
where get-operand-imm7 (Imm7 r) = r
```

```
context
  includes bit-operations-syntax
begin
```

```
type-synonym instruction = (sparc-operation  $\times$  inst-operand list)
```

```
definition get-op::word32  $\Rightarrow$  int
```

where *get-op* *w* \equiv *uint* (*w* >> 30)

definition *get-op2*::*word32* \Rightarrow *int*

where *get-op2* *w* \equiv

let *mask-op2* = 0b00000001110000000000000000000000 *in*
uint (((*AND*) *mask-op2* *w*) >> 22)

definition *get-op3*::*word32* \Rightarrow *int*

where *get-op3* *w* \equiv

let *mask-op3* = 0b00000001111110000000000000000000 *in*
uint (((*AND*) *mask-op3* *w*) >> 19)

definition *get-disp30*::*word32* \Rightarrow *int*

where *get-disp30* *w* \equiv

let *mask-disp30* = 0b00111111111111111111111111111111 *in*
uint (((*AND*) *mask-disp30* *w*) >> 2)

definition *get-a*::*word32* \Rightarrow *int*

where *get-a* *w* \equiv

let *mask-a* = 0b00100000000000000000000000000000 *in*
uint (((*AND*) *mask-a* *w*) >> 29)

definition *get-cond*::*word32* \Rightarrow *int*

where *get-cond* *w* \equiv

let *mask-cond* = 0b00011110000000000000000000000000 *in*
uint (((*AND*) *mask-cond* *w*) >> 25)

definition *get-disp-imm22*::*word32* \Rightarrow *int*

where *get-disp-imm22* *w* \equiv

let *mask-disp-imm22* = 0b00000000011111111111111111111111 *in*
uint (((*AND*) *mask-disp-imm22* *w*) >> 10)

definition *get-rd*::*word32* \Rightarrow *int*

where *get-rd* *w* \equiv

let *mask-rd* = 0b00111110000000000000000000000000 *in*
uint (((*AND*) *mask-rd* *w*) >> 25)

definition *get-rs1*::*word32* \Rightarrow *int*

where *get-rs1* *w* \equiv

let *mask-rs1* = 0b00000000000001111000000000000000 *in*
uint (((*AND*) *mask-rs1* *w*) >> 14)

definition *get-i*::*word32* \Rightarrow *int*

where *get-i* *w* \equiv

let *mask-i* = 0b00000000000000000000000010000000000000 *in*
uint (((*AND*) *mask-i* *w*) >> 13)

definition *get-opf*::*word32* \Rightarrow *int*

where *get-opf* *w* \equiv

let mask-opf = 0b000000000000000001111111100000 in
uint (((AND) mask-opf w) >> 5)

definition get-rs2::word32 ⇒ int

where get-rs2 w ≡

let mask-rs2 = 0b000000000000000000000000011111 in
uint ((AND) mask-rs2 w)

definition get-simm13::word32 ⇒ int

where get-simm13 w ≡

let mask-simm13 = 0b000000000000000001111111111111 in
uint ((AND) mask-simm13 w)

definition get-asi::word32 ⇒ int

where get-asi w ≡

let mask-asi = 0b00000000000000000111111100000 in
uint (((AND) mask-asi w) >> 5)

definition get-trap-cond:: word32 ⇒ int

where get-trap-cond w ≡

let mask-cond = 0b000111100000000000000000000000 in
uint (((AND) mask-cond w) >> 25)

definition get-trap-imm7:: word32 ⇒ int

where get-trap-imm7 w ≡

let mask-imm7 = 0b000000000000000000000001111111 in
uint ((AND) mask-imm7 w)

definition parse-instr-f1::word32 ⇒

(Exception list + instruction)

where — CALL, with a single operand disp30+00

parse-instr-f1 w ≡

Inr (call-type CALL,[W30 (word-of-int (get-disp30 w))])

definition parse-instr-f2::word32 ⇒

(Exception list + instruction)

where parse-instr-f2 w ≡

let op2 = get-op2 w in

if op2 = uint(0b100::word3) then — SETHI or NOP

let rd = get-rd w in

let imm22 = get-disp-imm22 w in

if rd = 0 ∧ imm22 = 0 then — NOP

Inr (nop-type NOP,[])

else — SETHI, with operands [imm22,rd]

Inr (sethi-type SETHI,[(W22 (word-of-int imm22)),
(W5 (word-of-int rd))])

else if op2 = uint(0b010::word3) then — Bicc, with operands [a,disp22]

let cond = get-cond w in

let flaga = Flag (word-of-int (get-a w)) in

```

let disp22 = W22 (word-of-int (get-disp-imm22 w)) in
if cond = uint(0b0001::word4) then — BE
  Inr (bicc-type BE,[flaga,disp22])
else if cond = uint(0b1001::word4) then — BNE
  Inr (bicc-type BNE,[flaga,disp22])
else if cond = uint(0b1100::word4) then — BGU
  Inr (bicc-type BGU,[flaga,disp22])
else if cond = uint(0b0010::word4) then — BLE
  Inr (bicc-type BLE,[flaga,disp22])
else if cond = uint(0b0011::word4) then — BL
  Inr (bicc-type BL,[flaga,disp22])
else if cond = uint(0b1011::word4) then — BGE
  Inr (bicc-type BGE,[flaga,disp22])
else if cond = uint(0b0110::word4) then — BNEG
  Inr (bicc-type BNEG,[flaga,disp22])
else if cond = uint(0b1010::word4) then — BG
  Inr (bicc-type BG,[flaga,disp22])
else if cond = uint(0b0101::word4) then — BCS
  Inr (bicc-type BCS,[flaga,disp22])
else if cond = uint(0b0100::word4) then — BLEU
  Inr (bicc-type BLEU,[flaga,disp22])
else if cond = uint(0b1101::word4) then — BCC
  Inr (bicc-type BCC,[flaga,disp22])
else if cond = uint(0b1000::word4) then — BA
  Inr (bicc-type BA,[flaga,disp22])
else if cond = uint(0b0000::word4) then — BN
  Inr (bicc-type BN,[flaga,disp22])
else if cond = uint(0b1110::word4) then — BPOS
  Inr (bicc-type BPOS,[flaga,disp22])
else if cond = uint(0b1111::word4) then — BVC
  Inr (bicc-type BVC,[flaga,disp22])
else if cond = uint(0b0111::word4) then — BVS
  Inr (bicc-type BVS,[flaga,disp22])
else Inl [invalid-cond-f2]
else Inl [invalid-op2-f2]

```

We don't consider floating-point operations, so we don't consider the third type of format 3.

definition $\text{parse-instr-f3}::\text{word32} \Rightarrow (\text{Exception list} + \text{instruction})$

where $\text{parse-instr-f3 } w \equiv$
 let $\text{this-op} = \text{get-op } w$ in
 let $\text{rd} = \text{get-rd } w$ in
 let $\text{op3} = \text{get-op3 } w$ in
 let $\text{rs1} = \text{get-rs1 } w$ in
 let $\text{flagi} = \text{get-i } w$ in
 let $\text{asi} = \text{get-asi } w$ in
 let $\text{rs2} = \text{get-rs2 } w$ in
 let $\text{simm13} = \text{get-simm13 } w$ in

if this-op = uint(0b11::word2) then — Load and Store
 — If an instruction accesses alternative space but *flagi = 1*,
 — may need to throw a trap.

if op3 = uint(0b001001::word6) then — LDSB
if flagi = 1 then — Operant list is [*i,rs1,simm13,rd*]
 Inr (load-store-type LDSB,[(Flag (word-of-int flagi)),
 (*W5 (word-of-int rs1)*),
 (*Simm13 (word-of-int simm13)*),
 (*W5 (word-of-int rd)*)]])
else — Operant list is [*i,rs1,rs2,rd*]
 Inr (load-store-type LDSB,[(Flag (word-of-int flagi)),
 (*W5 (word-of-int rs1)*),
 (*W5 (word-of-int rs2)*),
 (*W5 (word-of-int rd)*)]])

else if op3 = uint(0b011001::word6) then — LDSBA
 Inr (load-store-type LDSBA,[(Flag (word-of-int flagi)),
 (*W5 (word-of-int rs1)*),
 (*W5 (word-of-int rs2)*),
 (*Asi (word-of-int asi)*),
 (*W5 (word-of-int rd)*)]])

else if op3 = uint(0b001010::word6) then — LDSH
if flagi = 1 then — Operant list is [*i,rs1,simm13,rd*]
 Inr (load-store-type LDSH,[(Flag (word-of-int flagi)),
 (*W5 (word-of-int rs1)*),
 (*Simm13 (word-of-int simm13)*),
 (*W5 (word-of-int rd)*)]])
else — Operant list is [*i,rs1,rs2,rd*]
 Inr (load-store-type LDSH,[(Flag (word-of-int flagi)),
 (*W5 (word-of-int rs1)*),
 (*W5 (word-of-int rs2)*),
 (*W5 (word-of-int rd)*)]])

else if op3 = uint(0b011010::word6) then — LDSHA
 Inr (load-store-type LDSHA,[(Flag (word-of-int flagi)),
 (*W5 (word-of-int rs1)*),
 (*W5 (word-of-int rs2)*),
 (*Asi (word-of-int asi)*),
 (*W5 (word-of-int rd)*)]])

else if op3 = uint(0b000001::word6) then — LDUB
if flagi = 1 then — Operant list is [*i,rs1,simm13,rd*]
 Inr (load-store-type LDUB,[(Flag (word-of-int flagi)),
 (*W5 (word-of-int rs1)*),
 (*Simm13 (word-of-int simm13)*),
 (*W5 (word-of-int rd)*)]])
else — Operant list is [*i,rs1,rs2,rd*]
 Inr (load-store-type LDUB,[(Flag (word-of-int flagi)),
 (*W5 (word-of-int rs1)*),
 (*W5 (word-of-int rs2)*),
 (*W5 (word-of-int rd)*)]])

else if op3 = uint(0b010001::word6) then — LDUBA

```

    Inr (load-store-type LDUBA,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (Asi (word-of-int asi)),
      (W5 (word-of-int rd))])
else if op3 = uint(0b000010::word6) then — LDUH
  if flagi = 1 then — Operant list is [i,rs1,simm13,rd]
    Inr (load-store-type LDUH,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))])
  else — Operant list is [i,rs1,rs2,rd]
    Inr (load-store-type LDUH,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])
else if op3 = uint(0b010010::word6) then — LDUHA
  Inr (load-store-type LDUHA,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (W5 (word-of-int rs2)),
    (Asi (word-of-int asi)),
    (W5 (word-of-int rd))])
else if op3 = uint(0b000000::word6) then — LD
  if flagi = 1 then — Operant list is [i,rs1,simm13,rd]
    Inr (load-store-type LD,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))])
  else — Operant list is [i,rs1,rs2,rd]
    Inr (load-store-type LD,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])
else if op3 = uint(0b010000::word6) then — LDA
  Inr (load-store-type LDA,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (W5 (word-of-int rs2)),
    (Asi (word-of-int asi)),
    (W5 (word-of-int rd))])
else if op3 = uint(0b000011::word6) then — LDD
  if flagi = 1 then — Operant list is [i,rs1,simm13,rd]
    Inr (load-store-type LDD,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))])
  else — Operant list is [i,rs1,rs2,rd]
    Inr (load-store-type LDD,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),

```

```

      (W5 (word-of-int rd)))
else if op3 = uint(0b010011::word6) then — LDDA
  Inr (load-store-type LDDA,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (W5 (word-of-int rs2)),
    (Asi (word-of-int asi)),
    (W5 (word-of-int rd))])
else if op3 = uint(0b001101::word6) then — LDSTUB
  if flagi = 1 then — Operant list is [i,rs1,simm13,rd]
    Inr (load-store-type LDSTUB,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))])
  else — Operant list is [i,rs1,rs2,rd]
    Inr (load-store-type LDSTUB,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])
else if op3 = uint(0b011101::word6) then — LDSTUBA
  Inr (load-store-type LDSTUBA,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (W5 (word-of-int rs2)),
    (Asi (word-of-int asi)),
    (W5 (word-of-int rd))])
else if op3 = uint(0b000101::word6) then — STB
  if flagi = 1 then — Operant list is [i,rs1,simm13,rd]
    Inr (load-store-type STB,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))])
  else — Operant list is [i,rs1,rs2,rd]
    Inr (load-store-type STB,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])
else if op3 = uint(0b010101::word6) then — STBA
  Inr (load-store-type STBA,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (W5 (word-of-int rs2)),
    (Asi (word-of-int asi)),
    (W5 (word-of-int rd))])
else if op3 = uint(0b000110::word6) then — STH
  if flagi = 1 then — Operant list is [i,rs1,simm13,rd]
    Inr (load-store-type STH,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))])
  else — Operant list is [i,rs1,rs2,rd]
    Inr (load-store-type STH,[(Flag (word-of-int flagi)),

```

```

        (W5 (word-of-int rs1)),
        (W5 (word-of-int rs2)),
        (W5 (word-of-int rd))]
else if op3 = uint(0b010110::word6) then — STHA
  Inr (load-store-type STHA,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (W5 (word-of-int rs2)),
    (Asi (word-of-int asi)),
    (W5 (word-of-int rd))])
else if op3 = uint(0b000100::word6) then — ST
  if flagi = 1 then — Operant list is [i,rs1,simm13,rd]
    Inr (load-store-type ST,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))])
  else — Operant list is [i,rs1,rs2,rd]
    Inr (load-store-type ST,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])
else if op3 = uint(0b010100::word6) then — STA
  Inr (load-store-type STA,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (W5 (word-of-int rs2)),
    (Asi (word-of-int asi)),
    (W5 (word-of-int rd))])
else if op3 = uint(0b000111::word6) then — STD
  if flagi = 1 then — Operant list is [i,rs1,simm13,rd]
    Inr (load-store-type STD,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))])
  else — Operant list is [i,rs1,rs2,rd]
    Inr (load-store-type STD,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])
else if op3 = uint(0b010111::word6) then — STDA
  Inr (load-store-type STDA,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (W5 (word-of-int rs2)),
    (Asi (word-of-int asi)),
    (W5 (word-of-int rd))])
else if op3 = uint(0b001111::word6) then — SWAP
  if flagi = 1 then — Operant list is [i,rs1,simm13,rd]
    Inr (load-store-type SWAP,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))])

```

```

else — Operant list is  $[i,rs1,rs2,rd]$ 
  Inr (load-store-type SWAP,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (W5 (word-of-int rs2)),
    (W5 (word-of-int rd))])
else if op3 = uint(0b011111::word6) then — SWAPA
  Inr (load-store-type SWAPA,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (W5 (word-of-int rs2)),
    (Asi (word-of-int asi)),
    (W5 (word-of-int rd))])
else Inl [invalid-op3-f3-op11]
else if this-op = uint(0b10::word2) then — Others
  if op3 = uint(0b111000::word6) then — JMPL
    if flagi = 0 then — return  $[i,rs1,rs2,rd]$ 
      Inr (ctrl-type JMPL,[(Flag (word-of-int flagi)),
        (W5 (word-of-int rs1)),
        (W5 (word-of-int rs2)),
        (W5 (word-of-int rd))])
    else — return  $[i,rs1,simm13,rd]$ 
      Inr (ctrl-type JMPL,[(Flag (word-of-int flagi)),
        (W5 (word-of-int rs1)),
        (Simm13 (word-of-int simm13)),
        (W5 (word-of-int rd))])
  else if op3 = uint(0b111001::word6) then — RETT
    if flagi = 0 then — return  $[i,rs1,rs2]$ 
      Inr (ctrl-type RETT,[(Flag (word-of-int flagi)),
        (W5 (word-of-int rs1)),
        (W5 (word-of-int rs2))])
    else — return  $[i,rs1,simm13]$ 
      Inr (ctrl-type RETT,[(Flag (word-of-int flagi)),
        (W5 (word-of-int rs1)),
        (Simm13 (word-of-int simm13))])
— The following are Read and Write instructions,
— only return  $[rs1,rd]$  as operand.
else if op3 = uint(0b101000::word6)  $\wedge$  rs1  $\neq$  0 then — RDASR
  if rs1 = uint(0b011111::word6)  $\wedge$  rd = 0 then — STBAR is a special case of
RDASR
  Inr (load-store-type STBAR,[])
  else Inr (sreg-type RDASR,[(W5 (word-of-int rs1)),
    (W5 (word-of-int rd))])
else if op3 = uint(0b101000::word6)  $\wedge$  rs1 = 0 then — RDY
  Inr (sreg-type RDY,[(W5 (word-of-int rs1)),
    (W5 (word-of-int rd))])
else if op3 = uint(0b101001::word6) then — RDPSR
  Inr (sreg-type RDPSR,[(W5 (word-of-int rs1)),
    (W5 (word-of-int rd))])
else if op3 = uint(0b101010::word6) then — RDWIM
  Inr (sreg-type RDWIM,[(W5 (word-of-int rs1)),

```

```

      (W5 (word-of-int rd)))
else if op3 = uint(0b101011::word6) then — RDTBR
  Inr (sreg-type RDTBR,[(W5 (word-of-int rs1)),
    (W5 (word-of-int rd))])
else if op3 = uint(0b110000::word6) ∧ rd ≠ 0 then — WRASR
  if flagi = 0 then — return [i,rs1,rs2,rd]
  Inr (sreg-type WRASR,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (W5 (word-of-int rs2)),
    (W5 (word-of-int rd))])
else — return [i,rs1,simm13,rd]
  Inr (sreg-type WRASR,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (Simm13 (word-of-int simm13)),
    (W5 (word-of-int rd))])
else if op3 = uint(0b110000::word6) ∧ rd = 0 then — WRY
  if flagi = 0 then — return [i,rs1,rs2,rd]
  Inr (sreg-type WRY,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (W5 (word-of-int rs2)),
    (W5 (word-of-int rd))])
else — return [i,rs1,simm13,rd]
  Inr (sreg-type WRY,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (Simm13 (word-of-int simm13)),
    (W5 (word-of-int rd))])
else if op3 = uint(0b110001::word6) then — WRPSR
  if flagi = 0 then — return [i,rs1,rs2,rd]
  Inr (sreg-type WRPSR,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (W5 (word-of-int rs2)),
    (W5 (word-of-int rd))])
else — return [i,rs1,simm13,rd]
  Inr (sreg-type WRPSR,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (Simm13 (word-of-int simm13)),
    (W5 (word-of-int rd))])
else if op3 = uint(0b110010::word6) then — WRWIM
  if flagi = 0 then — return [i,rs1,rs2,rd]
  Inr (sreg-type WRWIM,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (W5 (word-of-int rs2)),
    (W5 (word-of-int rd))])
else — return [i,rs1,simm13,rd]
  Inr (sreg-type WRWIM,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (Simm13 (word-of-int simm13)),
    (W5 (word-of-int rd))])
else if op3 = uint(0b110011::word6) then — WRTBR

```

```

if flagi = 0 then — return [i,rs1,rs2,rd]
  Inr (sreg-type WRTBR,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (W5 (word-of-int rs2)),
    (W5 (word-of-int rd))])
else — return [i,rs1,simm13,rd]
  Inr (sreg-type WRTBR,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (Simm13 (word-of-int simm13)),
    (W5 (word-of-int rd))])
— FLUSH instruction
else if op3 = uint(0b111011::word6) then — FLUSH
  if flagi = 0 then — return [1,rs1,rs2]
    Inr (load-store-type FLUSH,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2))])
  else — return [i,rs1,simm13]
    Inr (load-store-type FLUSH,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13))])
— The following are arithmetic instructions.
else if op3 = uint(0b000001::word6) then — AND
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (logic-type ANDs,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])
  else — return [i,rs1,simm13,rd]
    Inr (logic-type ANDs,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))])
else if op3 = uint(0b010001::word6) then — ANDcc
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (logic-type ANDcc,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])
  else — return [i,rs1,simm13,rd]
    Inr (logic-type ANDcc,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))])
else if op3 = uint(0b000101::word6) then — ANDN
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (logic-type ANDN,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])

```

```

else — return [i,rs1,simm13,rd]
      Inr (logic-type ANDN,[ (Flag (word-of-int flagi)),
                             (W5 (word-of-int rs1)),
                             (Simm13 (word-of-int simm13)),
                             (W5 (word-of-int rd))])
else if op3 = uint(0b010101::word6) then — ANDNcc
if flagi = 0 then — return [i,rs1,rs2,rd]
      Inr (logic-type ANDNcc,[ (Flag (word-of-int flagi)),
                                (W5 (word-of-int rs1)),
                                (W5 (word-of-int rs2)),
                                (W5 (word-of-int rd))])
else — return [i,rs1,simm13,rd]
      Inr (logic-type ANDNcc,[ (Flag (word-of-int flagi)),
                                (W5 (word-of-int rs1)),
                                (Simm13 (word-of-int simm13)),
                                (W5 (word-of-int rd))])
else if op3 = uint(0b000010::word6) then — OR
if flagi = 0 then — return [i,rs1,rs2,rd]
      Inr (logic-type ORs,[ (Flag (word-of-int flagi)),
                             (W5 (word-of-int rs1)),
                             (W5 (word-of-int rs2)),
                             (W5 (word-of-int rd))])
else — return [i,rs1,simm13,rd]
      Inr (logic-type ORs,[ (Flag (word-of-int flagi)),
                             (W5 (word-of-int rs1)),
                             (Simm13 (word-of-int simm13)),
                             (W5 (word-of-int rd))])
else if op3 = uint(0b010010::word6) then — ORcc
if flagi = 0 then — return [i,rs1,rs2,rd]
      Inr (logic-type ORcc,[ (Flag (word-of-int flagi)),
                              (W5 (word-of-int rs1)),
                              (W5 (word-of-int rs2)),
                              (W5 (word-of-int rd))])
else — return [i,rs1,simm13,rd]
      Inr (logic-type ORcc,[ (Flag (word-of-int flagi)),
                              (W5 (word-of-int rs1)),
                              (Simm13 (word-of-int simm13)),
                              (W5 (word-of-int rd))])
else if op3 = uint(0b000110::word6) then — ORN
if flagi = 0 then — return [i,rs1,rs2,rd]
      Inr (logic-type ORN,[ (Flag (word-of-int flagi)),
                             (W5 (word-of-int rs1)),
                             (W5 (word-of-int rs2)),
                             (W5 (word-of-int rd))])
else — return [i,rs1,simm13,rd]
      Inr (logic-type ORN,[ (Flag (word-of-int flagi)),
                             (W5 (word-of-int rs1)),
                             (Simm13 (word-of-int simm13)),
                             (W5 (word-of-int rd))])

```

```

else if op3 = uint(0b010110::word6) then — ORNcc
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (logic-type ORNcc,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])
  else — return [i,rs1,simm13,rd]
    Inr (logic-type ORNcc,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))])
else if op3 = uint(0b000011::word6) then — XORs
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (logic-type XORs,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])
  else — return [i,rs1,simm13,rd]
    Inr (logic-type XORs,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))])
else if op3 = uint(0b010011::word6) then — XORcc
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (logic-type XORcc,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])
  else — return [i,rs1,simm13,rd]
    Inr (logic-type XORcc,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))])
else if op3 = uint(0b000111::word6) then — XNOR
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (logic-type XNOR,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])
  else — return [i,rs1,simm13,rd]
    Inr (logic-type XNOR,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))])
else if op3 = uint(0b010111::word6) then — XNORcc
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (logic-type XNORcc,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),

```

```

        (W5 (word-of-int rd)))
else — return [i,rs1,simm13,rd]
      Inr (logic-type XNORcc,[(Flag (word-of-int flagi)),
        (W5 (word-of-int rs1)),
        (Simm13 (word-of-int simm13)),
        (W5 (word-of-int rd))])
else if op3 = uint(0b100101::word6) then — SLL
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (shift-type SLL,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])
  else — return [i,rs1,shcnt,rd]
    let shcnt = rs2 in
      Inr (shift-type SLL,[(Flag (word-of-int flagi)),
        (W5 (word-of-int rs1)),
        (W5 (word-of-int shcnt)),
        (W5 (word-of-int rd))])
else if op3 = uint (0b100110::word6) then — SRL
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (shift-type SRL,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])
  else — return [i,rs1,shcnt,rd]
    let shcnt = rs2 in
      Inr (shift-type SRL,[(Flag (word-of-int flagi)),
        (W5 (word-of-int rs1)),
        (W5 (word-of-int shcnt)),
        (W5 (word-of-int rd))])
else if op3 = uint(0b100111::word6) then — SRA
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (shift-type SRA,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])
  else — return [i,rs1,shcnt,rd]
    let shcnt = rs2 in
      Inr (shift-type SRA,[(Flag (word-of-int flagi)),
        (W5 (word-of-int rs1)),
        (W5 (word-of-int shcnt)),
        (W5 (word-of-int rd))])
else if op3 = uint(0b000000::word6) then — ADD
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (arith-type ADD,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])
  else — return [i,rs1,simm13,rd]

```

```

      Inr (arith-type ADD,[(Flag (word-of-int flagi)),
        (W5 (word-of-int rs1)),
        (Simm13 (word-of-int simm13)),
        (W5 (word-of-int rd))]
else if op3 = uint(0b010000::word6) then — ADDcc
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (arith-type ADDcc,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))]
  else — return [i,rs1,simm13,rd]
    Inr (arith-type ADDcc,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))]
else if op3 = uint(0b001000::word6) then — ADDX
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (arith-type ADDX,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))]
  else — return [i,rs1,simm13,rd]
    Inr (arith-type ADDX,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))]
else if op3 = uint(0b011000::word6) then — ADDXcc
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (arith-type ADDXcc,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))]
  else — return [i,rs1,simm13,rd]
    Inr (arith-type ADDXcc,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))]
else if op3 = uint(0b100000::word6) then — TADDcc
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (arith-type TADDcc,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))]
  else — return [i,rs1,simm13,rd]
    Inr (arith-type TADDcc,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))]
else if op3 = uint(0b100010::word6) then — TADDccTV

```

```

if flagi = 0 then — return [i,rs1,rs2,rd]
  Inr (arith-type TADDccTV,[ (Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (W5 (word-of-int rs2)),
    (W5 (word-of-int rd))])
else — return [i,rs1,simm13,rd]
  Inr (arith-type TADDccTV,[ (Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (Simm13 (word-of-int simm13)),
    (W5 (word-of-int rd))])
else if op3 = uint(0b000100::word6) then — SUB
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (arith-type SUB,[ (Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])
  else — return [i,rs1,simm13,rd]
    Inr (arith-type SUB,[ (Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))])
else if op3 = uint(0b010100::word6) then — SUBcc
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (arith-type SUBcc,[ (Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])
  else — return [i,rs1,simm13,rd]
    Inr (arith-type SUBcc,[ (Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))])
else if op3 = uint(0b001100::word6) then — SUBX
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (arith-type SUBX,[ (Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])
  else — return [i,rs1,simm13,rd]
    Inr (arith-type SUBX,[ (Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))])
else if op3 = uint(0b011100::word6) then — SUBXcc
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (arith-type SUBXcc,[ (Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])

```

```

else — return [i,rs1,simm13,rd]
      Inr (arith-type SUBXcc,[(Flag (word-of-int flagi)),
                             (W5 (word-of-int rs1)),
                             (Simm13 (word-of-int simm13)),
                             (W5 (word-of-int rd))])
else if op3 = uint(0b100001::word6) then — TSUBcc
      if flagi = 0 then — return [i,rs1,rs2,rd]
            Inr (arith-type TSUBcc,[(Flag (word-of-int flagi)),
                                     (W5 (word-of-int rs1)),
                                     (W5 (word-of-int rs2)),
                                     (W5 (word-of-int rd))])
      else — return [i,rs1,simm13,rd]
            Inr (arith-type TSUBcc,[(Flag (word-of-int flagi)),
                                     (W5 (word-of-int rs1)),
                                     (Simm13 (word-of-int simm13)),
                                     (W5 (word-of-int rd))])
else if op3 = uint(0b100011::word6) then — TSUBccTV
      if flagi = 0 then — return [i,rs1,rs2,rd]
            Inr (arith-type TSUBccTV,[(Flag (word-of-int flagi)),
                                       (W5 (word-of-int rs1)),
                                       (W5 (word-of-int rs2)),
                                       (W5 (word-of-int rd))])
      else — return [i,rs1,simm13,rd]
            Inr (arith-type TSUBccTV,[(Flag (word-of-int flagi)),
                                       (W5 (word-of-int rs1)),
                                       (Simm13 (word-of-int simm13)),
                                       (W5 (word-of-int rd))])
else if op3 = uint(0b100100::word6) then — MULScC
      if flagi = 0 then — return [i,rs1,rs2,rd]
            Inr (arith-type MULScC,[(Flag (word-of-int flagi)),
                                     (W5 (word-of-int rs1)),
                                     (W5 (word-of-int rs2)),
                                     (W5 (word-of-int rd))])
      else — return [i,rs1,simm13,rd]
            Inr (arith-type MULScC,[(Flag (word-of-int flagi)),
                                     (W5 (word-of-int rs1)),
                                     (Simm13 (word-of-int simm13)),
                                     (W5 (word-of-int rd))])
else if op3 = uint(0b001010::word6) then — UMUL
      if flagi = 0 then — return [i,rs1,rs2,rd]
            Inr (arith-type UMUL,[(Flag (word-of-int flagi)),
                                   (W5 (word-of-int rs1)),
                                   (W5 (word-of-int rs2)),
                                   (W5 (word-of-int rd))])
      else — return [i,rs1,simm13,rd]
            Inr (arith-type UMUL,[(Flag (word-of-int flagi)),
                                   (W5 (word-of-int rs1)),
                                   (Simm13 (word-of-int simm13)),
                                   (W5 (word-of-int rd))])

```

```

else if op3 = uint(0b011010::word6) then — UMULcc
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (arith-type UMULcc,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])
  else — return [i,rs1,simm13,rd]
    Inr (arith-type UMULcc,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))])
else if op3 = uint(0b001011::word6) then — SMUL
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (arith-type SMUL,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])
  else — return [i,rs1,simm13,rd]
    Inr (arith-type SMUL,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))])
else if op3 = uint(0b011011::word6) then — SMULcc
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (arith-type SMULcc,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])
  else — return [i,rs1,simm13,rd]
    Inr (arith-type SMULcc,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))])
else if op3 = uint(0b001110::word6) then — UDIV
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (arith-type UDIV,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])
  else — return [i,rs1,simm13,rd]
    Inr (arith-type UDIV,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))])
else if op3 = uint(0b011110::word6) then — UDIVcc
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (arith-type UDIVcc,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),

```

```

      (W5 (word-of-int rd)))
else — return [i,rs1,simm13,rd]
      Inr (arith-type UDIVcc,[(Flag (word-of-int flagi)),
        (W5 (word-of-int rs1)),
        (Simm13 (word-of-int simm13)),
        (W5 (word-of-int rd))])
else if op3 = uint(0b001111::word6) then — SDIV
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (arith-type SDIV,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])
  else — return [i,rs1,simm13,rd]
    Inr (arith-type SDIV,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))])
else if op3 = uint(0b011111::word6) then — SDIVcc
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (arith-type SDIVcc,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])
  else — return [i,rs1,simm13,rd]
    Inr (arith-type SDIVcc,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))])
else if op3 = uint(0b111100::word6) then — SAVE
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (ctrl-type SAVE,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])
  else — return [i,rs1,simm13,rd]
    Inr (ctrl-type SAVE,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),
      (W5 (word-of-int rd))])
else if op3 = uint(0b111101::word6) then — RESTORE
  if flagi = 0 then — return [i,rs1,rs2,rd]
    Inr (ctrl-type RESTORE,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2)),
      (W5 (word-of-int rd))])
  else — return [i,rs1,simm13,rd]
    Inr (ctrl-type RESTORE,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Simm13 (word-of-int simm13)),

```

```

      (W5 (word-of-int rd)))
else if op3 = uint(0b111010::word6) then — Ticc
  let trap-cond = get-trap-cond w in
  let trap-imm7 = get-trap-imm7 w in
  if trap-cond = uint(0b1000::word4) then — TA
    if flagi = 0 then — return [i,rs1,rs2]
      Inr (ticc-type TA,[(Flag (word-of-int flagi)),
        (W5 (word-of-int rs1)),
        (W5 (word-of-int rs2))])
    else — return [i,rs1,trap-imm7]
      Inr (ticc-type TA,[(Flag (word-of-int flagi)),
        (W5 (word-of-int rs1)),
        (Imm7 (word-of-int trap-imm7))])
else if trap-cond = uint(0b0000::word4) then — TN
  if flagi = 0 then — return [i,rs1,rs2]
    Inr (ticc-type TN,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2))])
  else — return [i,rs1,trap-imm7]
    Inr (ticc-type TN,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Imm7 (word-of-int trap-imm7))])
else if trap-cond = uint(0b1001::word4) then — TNE
  if flagi = 0 then — return [i,rs1,rs2]
    Inr (ticc-type TNE,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2))])
  else — return [i,rs1,trap-imm7]
    Inr (ticc-type TNE,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Imm7 (word-of-int trap-imm7))])
else if trap-cond = uint(0b0001::word4) then — TE
  if flagi = 0 then — return [i,rs1,rs2]
    Inr (ticc-type TE,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2))])
  else — return [i,rs1,trap-imm7]
    Inr (ticc-type TE,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Imm7 (word-of-int trap-imm7))])
else if trap-cond = uint(0b1010::word4) then — TG
  if flagi = 0 then — return [i,rs1,rs2]
    Inr (ticc-type TG,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2))])
  else — return [i,rs1,trap-imm7]
    Inr (ticc-type TG,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Imm7 (word-of-int trap-imm7))])

```

```

else if trap-cond = uint(0b0010::word4) then — TLE
  if flagi = 0 then — return [i,rs1,rs2]
  Inr (ticc-type TLE,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (W5 (word-of-int rs2))])
else — return [i,rs1,trap-imm7]
  Inr (ticc-type TLE,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (Imm7 (word-of-int trap-imm7))])
else if trap-cond = uint(0b1011::word4) then — TGE
  if flagi = 0 then — return [i,rs1,rs2]
  Inr (ticc-type TGE,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (W5 (word-of-int rs2))])
else — return [i,rs1,trap-imm7]
  Inr (ticc-type TGE,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (Imm7 (word-of-int trap-imm7))])
else if trap-cond = uint(0b0011::word4) then — TL
  if flagi = 0 then — return [i,rs1,rs2]
  Inr (ticc-type TL,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (W5 (word-of-int rs2))])
else — return [i,rs1,trap-imm7]
  Inr (ticc-type TL,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (Imm7 (word-of-int trap-imm7))])
else if trap-cond = uint(0b1100::word4) then — TGU
  if flagi = 0 then — return [i,rs1,rs2]
  Inr (ticc-type TGU,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (W5 (word-of-int rs2))])
else — return [i,rs1,trap-imm7]
  Inr (ticc-type TGU,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (Imm7 (word-of-int trap-imm7))])
else if trap-cond = uint(0b0100::word4) then — TLEU
  if flagi = 0 then — return [i,rs1,rs2]
  Inr (ticc-type TLEU,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (W5 (word-of-int rs2))])
else — return [i,rs1,trap-imm7]
  Inr (ticc-type TLEU,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),
    (Imm7 (word-of-int trap-imm7))])
else if trap-cond = uint(0b1101::word4) then — TCC
  if flagi = 0 then — return [i,rs1,rs2]
  Inr (ticc-type TCC,[(Flag (word-of-int flagi)),
    (W5 (word-of-int rs1)),

```

```

      (W5 (word-of-int rs2)))
else — return [i,rs1,trap-imm7]
      Inr (ticc-type TCC,[(Flag (word-of-int flagi)),
        (W5 (word-of-int rs1)),
        (Imm7 (word-of-int trap-imm7))])
else if trap-cond = uint(0b0101::word4) then — TCS
  if flagi = 0 then — return [i,rs1,rs2]
    Inr (ticc-type TCS,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2))])
  else — return [i,rs1,trap-imm7]
    Inr (ticc-type TCS,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Imm7 (word-of-int trap-imm7))])
else if trap-cond = uint(0b1110::word4) then — TPOS
  if flagi = 0 then — return [i,rs1,rs2]
    Inr (ticc-type TPOS,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2))])
  else — return [i,rs1,trap-imm7]
    Inr (ticc-type TPOS,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Imm7 (word-of-int trap-imm7))])
else if trap-cond = uint(0b0110::word4) then — TNEG
  if flagi = 0 then — return [i,rs1,rs2]
    Inr (ticc-type TNEG,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2))])
  else — return [i,rs1,trap-imm7]
    Inr (ticc-type TNEG,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Imm7 (word-of-int trap-imm7))])
else if trap-cond = uint(0b1111::word4) then — TVC
  if flagi = 0 then — return [i,rs1,rs2]
    Inr (ticc-type TVC,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2))])
  else — return [i,rs1,trap-imm7]
    Inr (ticc-type TVC,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (Imm7 (word-of-int trap-imm7))])
else if trap-cond = uint(0b0111::word4) then — TVS
  if flagi = 0 then — return [i,rs1,rs2]
    Inr (ticc-type TVS,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),
      (W5 (word-of-int rs2))])
  else — return [i,rs1,trap-imm7]
    Inr (ticc-type TVS,[(Flag (word-of-int flagi)),
      (W5 (word-of-int rs1)),

```

```

      (Imm7 (word-of-int trap-imm7)))
    else Inl [invalid-trap-cond]
    else Inl [invalid-op3-f3-op10]
    else Inl [invalid-op-f3]

```

Read the word32 value from the Program Counter in the current state. Find the instruction in the memory address of the word32 value. Return a word32 value of the instruction.

definition *fetch-instruction*::('a) sparc-state \Rightarrow
 (Exception list + word32)
where *fetch-instruction* *s* \equiv
 — *pc-val* is the 32-bit memory address of the instruction.
 let *pc-val* = *cpu-reg-val* PC *s*;
 psr-val = *cpu-reg-val* PSR *s*;
 s-val = *get-S* *psr-val*;
 asi = if *s-val* = 0 then word-of-int 8 else word-of-int 9
 in
 — Check if *pc-val* is aligned to 4-byte (32-bit) boundary.
 — That is, check if the least significant two bits of
 — *pc-val* are 0s.
 if *uint*((AND) (0b00000000000000000000000000000011) *pc-val*) = 0 then
 — Get the 32-bit value from the address of *pc-val*
 — to the address of *pc-val*+3
 let (*mem-result*,*n-s*) = *memory-read asi pc-val s* in
 case *mem-result* of
 None \Rightarrow Inl [*fetch-instruction-error*]
 |Some *v* \Rightarrow Inr *v*
 else Inl [*fetch-instruction-error*]

Decode the word32 value of an instruction into the name of the instruction and its operands.

definition *decode-instruction*::word32 \Rightarrow
 Exception list + instruction
where *decode-instruction* *w* \equiv
 let *this-op* = *get-op* *w* in
 if *this-op* = *uint*(0b01::word2) then — Instruction format 1
 parse-instr-f1 *w*
 else if *this-op* = *uint*(0b00::word2) then — Instruction format 2
 parse-instr-f2 *w*
 else — *op* = 11 0r 10, instruction format 3
 parse-instr-f3 *w*

Get the current window from the PSR

definition *get-curr-win*::unit \Rightarrow ('a,('a::len window-size)) sparc-state-monad
where *get-curr-win* - \equiv

```

do
  curr-win ← gets (λs. (ucast (get-CWP (cpu-reg-val PSR s))));
  return curr-win
od

```

Operational semantics for CALL

definition *call-instr::instruction* \Rightarrow ('a::len,unit) *sparc-state-monad*
where *call-instr instr* \equiv
 let *op-list* = *snd instr*;
 mem-addr = ((ucast (get-operand-w30 (op-list!0)))::word32) << 2
 in
 do
 curr-win ← *get-curr-win*();
 pc-val ← *gets* (λs. (cpu-reg-val PC s));
 npc-val ← *gets* (λs. (cpu-reg-val nPC s));
 write-reg pc-val curr-win (word-of-int 15);
 write-cpu npc-val PC;
 write-cpu (pc-val + mem-addr) nPC;
 return ()
 od

Evaluate icc based on the bits N, Z, V, C in PSR and the type of branching instruction. See Sparcv8 manual Page 178.

definition *eval-icc::sparc-operation* \Rightarrow *word1* \Rightarrow *word1* \Rightarrow *word1* \Rightarrow *word1* \Rightarrow *int*
where
eval-icc instr-name n-val z-val v-val c-val \equiv
 if *instr-name* = *bicc-type BNE* then
 if *z-val* = 0 then 1 else 0
 else if *instr-name* = *bicc-type BE* then
 if *z-val* = 1 then 1 else 0
 else if *instr-name* = *bicc-type BG* then
 if ((OR) *z-val* (*n-val XOR v-val*)) = 0 then 1 else 0
 else if *instr-name* = *bicc-type BLE* then
 if ((OR) *z-val* (*n-val XOR v-val*)) = 1 then 1 else 0
 else if *instr-name* = *bicc-type BGE* then
 if (*n-val XOR v-val*) = 0 then 1 else 0
 else if *instr-name* = *bicc-type BL* then
 if (*n-val XOR v-val*) = 1 then 1 else 0
 else if *instr-name* = *bicc-type BGU* then
 if (*c-val* = 0 \wedge *z-val* = 0) then 1 else 0
 else if *instr-name* = *bicc-type BLEU* then
 if (*c-val* = 1 \vee *z-val* = 1) then 1 else 0
 else if *instr-name* = *bicc-type BCC* then
 if *c-val* = 0 then 1 else 0
 else if *instr-name* = *bicc-type BCS* then
 if *c-val* = 1 then 1 else 0
 else if *instr-name* = *bicc-type BNEG* then
 if *n-val* = 1 then 1 else 0
 else if *instr-name* = *bicc-type BA* then 1

```

else if instr-name = bicc-type BN then 0
else if instr-name = bicc-type BPOS then
  if n-val = 0 then 1 else 0
else if instr-name = bicc-type BVC then
  if v-val = 0 then 1 else 0
else if instr-name = bicc-type BVS then
  if v-val = 1 then 1 else 0
else -1

```

definition *branch-instr-sub1*:: *sparc-operation* \Rightarrow ('a) *sparc-state* \Rightarrow *int*

where *branch-instr-sub1 instr-name s* \equiv

```

let n-val = get-icc-N ((cpu-reg s) PSR);
z-val = get-icc-Z ((cpu-reg s) PSR);
v-val = get-icc-V ((cpu-reg s) PSR);
c-val = get-icc-C ((cpu-reg s) PSR)

```

in

```

eval-icc instr-name n-val z-val v-val c-val

```

Operational semantics for Branching instructions. Return exception or a bool value for annulment. If the bool value is 1, then the delay instruction is not executed, otherwise the delay instruction is executed.

definition *branch-instr::instruction* \Rightarrow ('a,unit) *sparc-state-monad*

where *branch-instr instr* \equiv

```

let instr-name = fst instr;
op-list = snd instr;
disp22 = get-operand-w22 (op-list!1);
flaga = get-operand-flag (op-list!0)

```

in

do

```

icc-val  $\leftarrow$  gets ( $\lambda s$ . (branch-instr-sub1 instr-name s));

```

```

npc-val  $\leftarrow$  gets ( $\lambda s$ . (cpu-reg-val nPC s));

```

```

pc-val  $\leftarrow$  gets ( $\lambda s$ . (cpu-reg-val PC s));

```

```

write-cpu npc-val PC;

```

```

if icc-val = 1 then

```

do

```

write-cpu (pc-val + (sign-ext24 (((ucast(disp22))::word24) << 2))) nPC;

```

```

if (instr-name = bicc-type BA)  $\wedge$  (flaga = 1) then

```

do

```

set-annul True;

```

```

return ()

```

od

else

```

return ()

```

od

```

else — icc-val = 0

```

do

```

write-cpu (npc-val + 4) nPC;

```

```

if flaga = 1 then

```

```

    do
      set-annul True;
      return ()
    od
  else return ()
od
od

```

Operational semantics for NOP

definition *nop-instr::instruction* \Rightarrow ('a,unit) *sparc-state-monad*
where *nop-instr instr* \equiv return ()

Operational semantics for SETHI

definition *sethi-instr::instruction* \Rightarrow ('a::len,unit) *sparc-state-monad*
where *sethi-instr instr* \equiv
 let *op-list* = snd *instr*;
 imm22 = get-operand-w22 (*op-list*!0);
 rd = get-operand-w5 (*op-list*!1)
 in
 if *rd* \neq 0 then
 do
 curr-win \leftarrow get-*curr-win*();
 write-reg (((ucast(*imm22*))::word32) << 10) *curr-win rd*;
 return ()
 od
 else return ()

Get *operand2* based on the flag *i*, *rs1*, *rs2*, and *simm13*. If *i* = 0 then *operand2* = *r[rs2]*, else *operand2* = *sign-ext13(simm13)*. *op-list* should be [*i,rs1,rs2,...*] or [*i,rs1,simm13,...*].

definition *get-operand2::inst-operand list* \Rightarrow ('a::len) *sparc-state*
 \Rightarrow *virtua-address*
where *get-operand2 op-list s* \equiv
 let *flagi* = get-operand-flag (*op-list*!0);
 curr-win = ucast (get-CWP (cpu-reg-val PSR s))
 in
 if *flagi* = 0 then
 let *rs2* = get-operand-w5 (*op-list*!2);
 rs2-val = user-reg-val *curr-win rs2 s*
 in *rs2-val*
 else
 let *ext-simm13* = sign-ext13 (get-operand-simm13 (*op-list*!2)) in
 ext-simm13

Get *operand2-val* based on the flag *i*, *rs1*, *rs2*, and *simm13*. If *i* = 0 then *operand2-val* = *uint r[rs2]*, else *operand2-val* = *sint sign-ext13(simm13)*. *op-list* should be [*i,rs1,rs2,...*] or [*i,rs1,simm13,...*].

definition $get\text{-}operand2\text{-}val::inst\text{-}operand\ list \Rightarrow ('a::len)\ sparc\text{-}state \Rightarrow int$
where $get\text{-}operand2\text{-}val\ op\text{-}list\ s \equiv$
 $let\ flagi = get\text{-}operand\text{-}flag\ (op\text{-}list!0);$
 $curr\text{-}win = ucast\ (get\text{-}CWP\ (cpu\text{-}reg\text{-}val\ PSR\ s))$
 in
 $if\ flagi = 0\ then$
 $let\ rs2 = get\text{-}operand\text{-}w5\ (op\text{-}list!2);$
 $rs2\text{-}val = user\text{-}reg\text{-}val\ curr\text{-}win\ rs2\ s$
 $in\ sint\ rs2\text{-}val$
 $else$
 $let\ ext\text{-}simm13 = sign\text{-}ext13\ (get\text{-}operand\text{-}simm13\ (op\text{-}list!2))\ in$
 $sint\ ext\text{-}simm13$

Get the address based on the flag i , $rs1$, $rs2$, and $simm13$. If $i = 0$ then $addr = r[rs1] + r[rs2]$, else $addr = r[rs1] + sign\text{-}ext13(simm13)$. $op\text{-}list$ should be $[i,rs1,rs2,\dots]$ or $[i,rs1,simm13,\dots]$.

definition $get\text{-}addr::inst\text{-}operand\ list \Rightarrow ('a::len)\ sparc\text{-}state \Rightarrow virtua\text{-}address$
where $get\text{-}addr\ op\text{-}list\ s \equiv$
 $let\ rs1 = get\text{-}operand\text{-}w5\ (op\text{-}list!1);$
 $curr\text{-}win = ucast\ (get\text{-}CWP\ (cpu\text{-}reg\text{-}val\ PSR\ s));$
 $rs1\text{-}val = user\text{-}reg\text{-}val\ curr\text{-}win\ rs1\ s;$
 $op2 = get\text{-}operand2\ op\text{-}list\ s$
 in
 $(rs1\text{-}val + op2)$

Operational semantics for JMPL

definition $jmp\text{-}instr::instruction \Rightarrow ('a::len,unit)\ sparc\text{-}state\text{-}monad$
where $jmp\text{-}instr\ instr \equiv$
 $let\ op\text{-}list = snd\ instr;$
 $rd = get\text{-}operand\text{-}w5\ (op\text{-}list!3)$
 in
 do
 $curr\text{-}win \leftarrow get\text{-}curr\text{-}win();$
 $jmp\text{-}addr \leftarrow gets\ (\lambda s. (get\text{-}addr\ op\text{-}list\ s));$
 $if\ ((AND)\ jmp\text{-}addr\ 0b000000000000000000000000000011) \neq 0\ then$
 do
 $raise\text{-}trap\ mem\text{-}address\text{-}not\text{-}aligned;$
 $return\ ()$
 od
 $else$
 do
 $rd\text{-}next\text{-}val \leftarrow gets\ (\lambda s. (if\ rd \neq 0\ then$
 $(cpu\text{-}reg\text{-}val\ PC\ s)$
 $else$
 $user\text{-}reg\text{-}val\ curr\text{-}win\ rd\ s));$
 $write\text{-}reg\ rd\text{-}next\text{-}val\ curr\text{-}win\ rd;$
 $npc\text{-}val \leftarrow gets\ (\lambda s. (cpu\text{-}reg\text{-}val\ nPC\ s));$

```

    write-cpu npc-val PC;
    write-cpu jmp-addr nPC;
    return ()
  od
od

```

Operational semantics for RETT

definition *rett-instr* :: *instruction* \Rightarrow ('a::len,unit) *sparc-state-monad*

where *rett-instr instr* \equiv

```

let op-list = snd instr in
do
  psr-val  $\leftarrow$  gets ( $\lambda s$ . (cpu-reg-val PSR s));
  curr-win  $\leftarrow$  gets ( $\lambda s$ . (get-CWP (cpu-reg-val PSR s)));
  new-cwp  $\leftarrow$  gets ( $\lambda s$ . (word-of-int (((uint curr-win) + 1) mod NWINDOWS)));
  new-cwp-int  $\leftarrow$  gets ( $\lambda s$ . ((uint curr-win) + 1) mod NWINDOWS);
  addr  $\leftarrow$  gets ( $\lambda s$ . (get-addr op-list s));
  et-val  $\leftarrow$  gets ( $\lambda s$ . ((ucast (get-ET psr-val))::word1));
  s-val  $\leftarrow$  gets ( $\lambda s$ . ((ucast (get-S psr-val))::word1));
  ps-val  $\leftarrow$  gets ( $\lambda s$ . ((ucast (get-PS psr-val))::word1));
  wim-val  $\leftarrow$  gets ( $\lambda s$ . (cpu-reg-val WIM s));
  npc-val  $\leftarrow$  gets ( $\lambda s$ . (cpu-reg-val nPC s));
  if et-val = 1 then
    if s-val = 0 then
      do
        raise-trap privileged-instruction;
        return ()
      od
    else
      do
        raise-trap illegal-instruction;
        return ()
      od
  else if s-val = 0 then
    do
      write-cpu-tt (0b00000011::word8);
      set-exe-mode False;
      set-err-mode True;
      raise-trap privileged-instruction;
      fail ()
    od
  else if (get-WIM-bit (nat new-cwp-int) wim-val)  $\neq$  0 then
    do
      write-cpu-tt (0b00000110::word8);
      set-exe-mode False;
      set-err-mode True;
      raise-trap window-underflow;
      fail ()
    od
  else if ((AND) addr (0b00000000000000000000000000000011::word32))  $\neq$  0

```

```

then
  do
    write-cpu-tt (0b00000111::word8);
    set-exe-mode False;
    set-err-mode True;
    raise-trap mem-address-not-aligned;
    fail ()
  od
else
  do
    write-cpu npc-val PC;
    write-cpu addr nPC;
    new-psr-val ← gets (λs. (update-PSR-rett new-cwp 1 ps-val psr-val));
    write-cpu new-psr-val PSR;
    return ()
  od
od

```

definition *save-restore-sub1* :: *word32* ⇒ *word5* ⇒ *word5* ⇒ ('a::len,unit) *sparc-state-monad*

where *save-restore-sub1 result new-cwp rd* ≡

```

do
  psr-val ← gets (λs. (cpu-reg-val PSR s));
  new-psr-val ← gets (λs. (update-CWP new-cwp psr-val));
  write-cpu new-psr-val PSR; — Change CWP to the new window value.
  write-reg result (ucast new-cwp) rd; — Write result in rd of the new window.
  return ()
od

```

Operational semantics for SAVE and RESTORE.

definition *save-restore-instr* :: *instruction* ⇒ ('a::len,unit) *sparc-state-monad*

where *save-restore-instr instr* ≡

```

let instr-name = fst instr;
    op-list = snd instr;
    rd = get-operand-w5 (op-list!3)
in
do
  psr-val ← gets (λs. (cpu-reg-val PSR s));
  curr-win ← get-curr-win();
  wim-val ← gets (λs. (cpu-reg-val WIM s));
  if instr-name = ctrl-type SAVE then
    do
      new-cwp ← gets (λs. ((word-of-int (((uint curr-win) - 1) mod NWIN-
DOWS))):word5);
      if (get-WIM-bit (unat new-cwp) wim-val) ≠ 0 then
        do
          raise-trap window-overflow;
          return ()
        od
      else

```

```

    do
      result ← gets (λs. (get-addr op-list s)); — operands are from the old
window.
      save-retore-sub1 result new-cwp rd
    od
  od
else — instr-name = RESTORE
  do
    new-cwp ← gets (λs. ((word-of-int (((uint curr-win) + 1) mod NWIN-
DOWS))::word5);
    if (get-WIM-bit (unat new-cwp) wim-val) ≠ 0 then
      do
        raise-trap window-underflow;
        return ()
      od
    else
      do
        result ← gets (λs. (get-addr op-list s)); — operands are from the old
window.
        save-retore-sub1 result new-cwp rd
      od
    od
  od

```

definition *flush-cache-line* :: *word32* ⇒ ('a,unit) *sparc-state-monad*
where *flush-cache-line* ≡ *undefined*

definition *flush-Ibuf-and-pipeline* :: *word32* ⇒ ('a,unit) *sparc-state-monad*
where *flush-Ibuf-and-pipeline* ≡ *undefined*

Operational semantics for FLUSH. Flush the all the caches.

definition *flush-instr* :: *instruction* ⇒ ('a::len,unit) *sparc-state-monad*
where *flush-instr instr* ≡
 let *op-list* = *snd instr in*
 do
addr ← *gets* (λs. (*get-addr op-list s*));
modify (λs. (*flush-cache-all s*));
~~*flush-cache-line addr*;~~
~~*flush-Ibuf-and-pipeline addr*;~~
 return ()
 od

Operational semantics for read state register instructions. We do not consider RDASR here.

definition *read-state-reg-instr* :: *instruction* ⇒ ('a::len,unit) *sparc-state-monad*
where *read-state-reg-instr instr* ≡
 let *instr-name* = *fst instr*;
op-list = *snd instr*;
rs1 = *get-operand-w5 (op-list!0)*;

```

    rd = get-operand-w5 (op-list!1)
in
do
  curr-win ← get-curr-win();
  psr-val ← gets (λs. (cpu-reg-val PSR s));
  s-val ← gets (λs. (get-S psr-val));
  if (instr-name ∈ {sreg-type RDPSR, sreg-type RDWIM, sreg-type RDTBR} ∨
      (instr-name = sreg-type RDASR ∧ privileged-ASR rs1))
    ∧ ((ucast s-val)::word1) = 0 then
  do
    raise-trap privileged-instruction;
    return ()
  od
  else if illegal-instruction-ASR rs1 then
  do
    raise-trap illegal-instruction;
    return ()
  od
  else if rd ≠ 0 then
  if instr-name = sreg-type RDY then
  do
    y-val ← gets (λs. (cpu-reg-val Y s));
    write-reg y-val curr-win rd;
    return ()
  od
  else if instr-name = sreg-type RDASR then
  do
    asr-val ← gets (λs. (cpu-reg-val (ASR rs1) s));
    write-reg asr-val curr-win rd;
    return ()
  od
  else if instr-name = sreg-type RDPSR then
  do
    write-reg psr-val curr-win rd;
    return ()
  od
  else if instr-name = sreg-type RDWIM then
  do
    wim-val ← gets (λs. (cpu-reg-val WIM s));
    write-reg wim-val curr-win rd;
    return ()
  od
  else — Must be RDTBR.
  do
    tbr-val ← gets (λs. (cpu-reg-val TBR s));
    write-reg tbr-val curr-win rd;
    return ()
  od
  else return ()

```

od

Operational semantics for write state register instructions. We do not consider WRASR here.

definition *write-state-reg-instr* :: *instruction* \Rightarrow (*a::len,unit*) *sparc-state-monad*

where *write-state-reg-instr instr* \equiv

let instr-name = *fst instr*;

op-list = *snd instr*;

rs1 = *get-operand-w5 (op-list!1)*;

rd = *get-operand-w5 (op-list!3)*

in

do

curr-win \leftarrow *get-curr-win()*;

psr-val \leftarrow *gets* ($\lambda s. (cpu-reg-val\ PSR\ s)$);

s-val \leftarrow *gets* ($\lambda s. (get-S\ psr-val)$);

op2 \leftarrow *gets* ($\lambda s. (get-operand2\ op-list\ s)$);

rs1-val \leftarrow *gets* ($\lambda s. (user-reg-val\ curr-win\ rs1\ s)$);

result \leftarrow *gets* ($\lambda s. ((XOR)\ rs1-val\ op2)$);

if instr-name = *sreg-type WRY* *then*

do

modify ($\lambda s. (delayed-pool-add\ (DELAYNUM,\ result,\ Y)\ s)$);

return ()

od

else if instr-name = *sreg-type WRASR* *then*

if privileged-ASR rd \wedge *s-val* = 0 *then*

do

raise-trap privileged-instruction;

return ()

od

else if illegal-instruction-ASR rd *then*

do

raise-trap illegal-instruction;

return ()

od

else

do

modify ($\lambda s. (delayed-pool-add\ (DELAYNUM,\ result,\ (ASR\ rd))\ s)$);

return ()

od

else if instr-name = *sreg-type WRPSR* *then*

if s-val = 0 *then*

do

raise-trap privileged-instruction;

return ()

od

else if (uint ((ucast result)::word5)) \geq NWINDOWS *then*

do

raise-trap illegal-instruction;

return ()

```

    od
  else
    do — ET and PIL appear to be written IMMEDIATELY w.r.t. interrupts.
      pil-val ← gets (λs. (get-PIL result));
      et-val ← gets (λs. (get-ET result));
      new-psr-val ← gets (λs. (update-PSR-et-pil et-val pil-val psr-val));
      write-cpu new-psr-val PSR;
      modify (λs. (delayed-pool-add (DELAYNUM, result, PSR) s));
      return ()
    od
  else if instr-name = sreg-type WRWIM then
    if s-val = 0 then
      do
        raise-trap privileged-instruction;
        return ()
      od
    else
      do — Don't write bits corresponding to non-existent windows.
        result-f ← gets (λs. ((result << nat (32 - NWINDOWS)) >> nat (32 -
NWINDOWS)));
        modify (λs. (delayed-pool-add (DELAYNUM, result-f, WIM) s));
        return ()
      od
    else — Must be WRTBR
      if s-val = 0 then
        do
          raise-trap privileged-instruction;
          return ()
        od
      else
        do — Only write the bits <31:12> of the result to TBR.
          tbr-val ← gets (λs. (cpu-reg-val TBR s));
          tbr-val-11-0 ← gets (λs. ((AND) tbr-val 0b00000000000000000000111111111111));
          result-tmp ← gets (λs. ((AND) result 0b11111111111111111111100000000000));
          result-f ← gets (λs. ((OR) tbr-val-11-0 result-tmp));
          modify (λs. (delayed-pool-add (DELAYNUM, result-f, TBR) s));
          return ()
        od
      od
    od
  od

```

definition *logical-result* :: *sparc-operation* ⇒ *word32* ⇒ *word32* ⇒ *word32*

where *logical-result instr-name rs1-val operand2* ≡

```

if (instr-name = logic-type ANDs) ∨
  (instr-name = logic-type ANDcc) then
  (AND) rs1-val operand2
else if (instr-name = logic-type ANDN) ∨
  (instr-name = logic-type ANDNcc) then
  (AND) rs1-val (NOT operand2)
else if (instr-name = logic-type ORs) ∨

```

```

      (instr-name = logic-type ORcc) then
      (OR) rs1-val operand2
    else if instr-name ∈ {logic-type ORN,logic-type ORNcc} then
      (OR) rs1-val (NOT operand2)
    else if instr-name ∈ {logic-type XORs,logic-type XORcc} then
      (XOR) rs1-val operand2
    else — Must be XNOR or XNORcc
      (XOR) rs1-val (NOT operand2)

```

definition *logical-new-psr-val* :: *word32* ⇒ ('a) *sparc-state* ⇒ *word32*
where *logical-new-psr-val result s* ≡
 let *psr-val* = *cpu-reg-val PSR s*;
 n-val = (*ucast (result >> 31)*)::*word1*;
 z-val = if (*result* = 0) then 1 else 0;
 v-val = 0;
 c-val = 0
 in
update-PSR-icc n-val z-val v-val c-val psr-val

definition *logical-instr-sub1* :: *sparc-operation* ⇒ *word32* ⇒
 ('a::len,unit) *sparc-state-monad*
where
logical-instr-sub1 instr-name result ≡
 if *instr-name* ∈ {*logic-type ANDcc,logic-type ANDNcc,logic-type ORcc,*
logic-type ORNcc,logic-type XORcc,logic-type XNORcc} then
 do
 new-psr-val ← *gets (λs. (logical-new-psr-val result s))*;
 write-cpu new-psr-val PSR;
 return ()
 od
 else return ()

Operational semantics for logical instructions.

definition *logical-instr* :: *instruction* ⇒ ('a::len,unit) *sparc-state-monad*
where *logical-instr instr* ≡
 let *instr-name* = *fst instr*;
 op-list = *snd instr*;
 rs1 = *get-operand-w5 (op-list!1)*;
 rd = *get-operand-w5 (op-list!3)*
 in
 do
 operand2 ← *gets (λs. (get-operand2 op-list s))*;
 curr-win ← *get-curr-win()*;
 rs1-val ← *gets (λs. (user-reg-val curr-win rs1 s))*;
 rd-val ← *gets (λs. (user-reg-val curr-win rd s))*;
 result ← *gets (λs. (logical-result instr-name rs1-val operand2))*;

```

    new-rd-val ← gets (λs. (if rd ≠ 0 then result else rd-val));
    write-reg new-rd-val curr-win rd;
    logical-instr-sub1 instr-name result
  od

```

Operational semantics for shift instructions.

definition *shift-instr* :: *instruction* ⇒ ('a::len,unit) *sparc-state-monad*

where *shift-instr instr* ≡

```

  let instr-name = fst instr;
      op-list = snd instr;
      flagi = get-operand-flag (op-list!0);
      rs1 = get-operand-w5 (op-list!1);
      rs2-shcnt = get-operand-w5 (op-list!2);
      rd = get-operand-w5 (op-list!3)
  in
  do
    curr-win ← get-curr-win();
    shift-count ← gets (λs. (if flagi = 0 then
      ucast (user-reg-val curr-win rs2-shcnt s)
      else rs2-shcnt));
    rs1-val ← gets (λs. (user-reg-val curr-win rs1 s));
    if (instr-name = shift-type SLL) ∧ (rd ≠ 0) then
      do
        rd-val ← gets (λs. (rs1-val << (unat shift-count)));
        write-reg rd-val curr-win rd;
        return ()
      od
    else if (instr-name = shift-type SRL) ∧ (rd ≠ 0) then
      do
        rd-val ← gets (λs. (rs1-val >> (unat shift-count)));
        write-reg rd-val curr-win rd;
        return ()
      od
    else if (instr-name = shift-type SRA) ∧ (rd ≠ 0) then
      do
        rd-val ← gets (λs. (rs1-val >>> (unat shift-count)));
        write-reg rd-val curr-win rd;
        return ()
      od
    else return ()
  od

```

definition *add-instr-sub1* :: *sparc-operation* ⇒ *word32* ⇒ *word32* ⇒ *word32*
 ⇒ ('a::len,unit) *sparc-state-monad*

where *add-instr-sub1 instr-name result rs1-val operand2* ≡

```

  if instr-name ∈ {arith-type ADDcc,arith-type ADDXcc} then
  do
    psr-val ← gets (λs. (cpu-reg-val PSR s));
    result-31 ← gets (λs. ((ucast (result >> 31))::word1));

```

```

rs1-val-31 ← gets (λs. ((ucast (rs1-val >> 31))::word1));
operand2-31 ← gets (λs. ((ucast (operand2 >> 31))::word1));
new-n-val ← gets (λs. (result-31));
new-z-val ← gets (λs. (if result = 0 then 1::word1 else 0::word1));
new-v-val ← gets (λs. ((OR) ((AND) rs1-val-31
                             ((AND) operand2-31
                              (NOT result-31)))
                      ((AND) (NOT rs1-val-31)
                              ((AND) (NOT operand2-31)
                                       result-31)))));
new-c-val ← gets (λs. ((OR) ((AND) rs1-val-31
                             operand2-31)
                      ((AND) (NOT result-31)
                              ((OR) rs1-val-31
                                       operand2-31)))));
new-psr-val ← gets (λs. (update-PSR-icc new-n-val
                               new-z-val
                               new-v-val
                               new-c-val psr-val));

write-cpu new-psr-val PSR;
return ()
od
else return ()

```

Operational semantics for add instructions. These include ADD, ADDcc, ADDX.

definition $add-instr :: instruction \Rightarrow ('a::len,unit) \text{ sparc-state-monad}$

where $add-instr \text{ instr} \equiv$

```

let instr-name = fst instr;
    op-list = snd instr;
    rs1 = get-operand-w5 (op-list!1);
    rd = get-operand-w5 (op-list!3)
in
do
  operand2 ← gets (λs. (get-operand2 op-list s));
  curr-win ← get-curr-win();
  rs1-val ← gets (λs. (user-reg-val curr-win rs1 s));
  psr-val ← gets (λs. (cpu-reg-val PSR s));
  c-val ← gets (λs. (get-icc-C psr-val));
  result ← gets (λs. (if (instr-name = arith-type ADD) ∨
                        (instr-name = arith-type ADDcc) then
                      rs1-val + operand2
                      else — Must be ADDX or ADDXcc
                      rs1-val + operand2 + (ucast c-val)));
  rd-val ← gets (λs. (user-reg-val curr-win rd s));
  new-rd-val ← gets (λs. (if rd ≠ 0 then result else rd-val));
  write-reg new-rd-val curr-win rd;
  add-instr-sub1 instr-name result rs1-val operand2

```

od

definition *sub-instr-sub1* :: *sparc-operation* \Rightarrow *word32* \Rightarrow *word32* \Rightarrow *word32*
 \Rightarrow ('a::len,unit) *sparc-state-monad*
where *sub-instr-sub1* *instr-name* *result* *rs1-val* *operand2* \equiv
 if *instr-name* \in {*arith-type SUBcc,arith-type SUBXcc*} then
 do
 psr-val \leftarrow *gets* ($\lambda s.$ (*cpu-reg-val* *PSR* *s*));
 result-31 \leftarrow *gets* ($\lambda s.$ ((*ucast* (*result* \gg 31))::*word1*));
 rs1-val-31 \leftarrow *gets* ($\lambda s.$ ((*ucast* (*rs1-val* \gg 31))::*word1*));
 operand2-31 \leftarrow *gets* ($\lambda s.$ ((*ucast* (*operand2* \gg 31))::*word1*));
 new-n-val \leftarrow *gets* ($\lambda s.$ (*result-31*));
 new-z-val \leftarrow *gets* ($\lambda s.$ (if *result* = 0 then 1::*word1* else 0::*word1*));
 new-v-val \leftarrow *gets* ($\lambda s.$ ((*OR*) ((*AND*) *rs1-val-31*
 ((*AND*) (*NOT* *operand2-31*)
 (*NOT* *result-31*)))
 ((*AND*) (*NOT* *rs1-val-31*)
 ((*AND*) *operand2-31*
 result-31)))));
 new-c-val \leftarrow *gets* ($\lambda s.$ ((*OR*) ((*AND*) (*NOT* *rs1-val-31*)
 operand2-31)
 ((*AND*) *result-31*
 ((*OR*) (*NOT* *rs1-val-31*)
 operand2-31)))));
 new-psr-val \leftarrow *gets* ($\lambda s.$ (*update-PSR-icc* *new-n-val*
 new-z-val
 new-v-val
 new-c-val *psr-val*));
 write-cpu *new-psr-val* *PSR*;
 return ()
 od
 else *return* ()

Operational semantics for subtract instructions. These include SUB, SUBcc, SUBX.

definition *sub-instr* :: *instruction* \Rightarrow ('a::len,unit) *sparc-state-monad*
where *sub-instr* *instr* \equiv
 let *instr-name* = *fst* *instr*;
 op-list = *snd* *instr*;
 rs1 = *get-operand-w5* (*op-list*!1);
 rd = *get-operand-w5* (*op-list*!3)
 in
 do
 operand2 \leftarrow *gets* ($\lambda s.$ (*get-operand2* *op-list* *s*));
 curr-win \leftarrow *get-curr-win*();
 rs1-val \leftarrow *gets* ($\lambda s.$ (*user-reg-val* *curr-win* *rs1* *s*));
 psr-val \leftarrow *gets* ($\lambda s.$ (*cpu-reg-val* *PSR* *s*));
 c-val \leftarrow *gets* ($\lambda s.$ (*get-icc-C* *psr-val*));

```

result ← gets (λs. (if (instr-name = arith-type SUB) ∨
                    (instr-name = arith-type SUBcc) then
                    rs1-val - operand2
                    else — Must be SUBX or SUBXcc
                    rs1-val - operand2 - (ucast c-val)));
rd-val ← gets (λs. (user-reg-val curr-win rd s));
new-rd-val ← gets (λs. (if rd ≠ 0 then result else rd-val));
write-reg new-rd-val curr-win rd;
sub-instr-sub1 instr-name result rs1-val operand2
od

```

definition *mul-instr-sub1* :: *sparc-operation* ⇒ *word32* ⇒
('a::len,unit) *sparc-state-monad*

where *mul-instr-sub1* *instr-name* *result* ≡
if *instr-name* ∈ {*arith-type SMULcc*,*arith-type UMULcc*} then
do
psr-val ← gets (λs. (cpu-reg-val PSR s));
new-n-val ← gets (λs. ((ucast (result >> 31))::word1));
new-z-val ← gets (λs. (if result = 0 then 1 else 0));
new-v-val ← gets (λs. 0);
new-c-val ← gets (λs. 0);
new-psr-val ← gets (λs. (update-PSR-icc new-n-val
new-z-val
new-v-val
new-c-val psr-val));
write-cpu new-psr-val PSR;
return ()
od
else return ()

Operational semantics for multiply instructions.

definition *mul-instr* :: *instruction* ⇒ ('a::len,unit) *sparc-state-monad*

where *mul-instr* *instr* ≡
let *instr-name* = fst *instr*;
op-list = snd *instr*;
rs1 = get-operand-w5 (*op-list*!1);
rd = get-operand-w5 (*op-list*!3)
in
do
operand2 ← gets (λs. (get-operand2 *op-list* s));
curr-win ← get-curr-win();
rs1-val ← gets (λs. (user-reg-val *curr-win* *rs1* s));
psr-val ← gets (λs. (cpu-reg-val PSR s));
result0 ← gets (λs. (if *instr-name* ∈ {*arith-type UMUL*,*arith-type UMULcc*}
then
(word-of-int ((uint *rs1-val*) *
(uint *operand2*)))::word64
else — Must be *SMUL* or *SMULcc*

```

      (word-of-int ((sint rs1-val) *
                   (sint operand2))::word64));
— whether to use ucast or scast does not matter below.
y-val ← gets (λs. ((ucast (result0 >> 32))::word32));
write-cpu y-val Y;
result ← gets (λs. ((ucast result0)::word32));
rd-val ← gets (λs. (user-reg-val curr-win rd s));
new-rd-val ← gets (λs. (if rd ≠ 0 then result else rd-val));
write-reg new-rd-val curr-win rd;
mul-instr-sub1 instr-name result
od

```

definition *div-comp-temp-64bit* :: *instruction* ⇒ *word64* ⇒ *virtua-address* ⇒ *word64*

where *div-comp-temp-64bit* *i y-rs1 operand2* ≡
if ((*fst i*) = *arith-type UDIV*) ∨ ((*fst i*) = *arith-type UDIVcc*) *then*
 (word-of-int ((uint *y-rs1*) div (uint *operand2*))::word64
else — Must be *SDIV* or *SDIVcc*.
 — Due to Isabelle’s rounding method is not nearest to zero,
 — we have to implement division in a different way.
let sop1 = sint y-rs1;
 sop2 = sint operand2;
 pop1 = abs sop1;
 pop2 = abs sop2
in
if sop1 > 0 ∧ sop2 > 0 then
 (word-of-int (*sop1 div sop2*))
else if sop1 > 0 ∧ sop2 < 0 then
 (word-of-int (− (*sop1 div pop2*)))
else if sop1 < 0 ∧ sop2 > 0 then
 (word-of-int (− (*pop1 div sop2*)))
else — *sop1 < 0 ∧ sop2 < 0*
 (word-of-int (*pop1 div pop2*))

definition *div-comp-temp-V* :: *instruction* ⇒ *word32* ⇒ *word33* ⇒ *word1*

where *div-comp-temp-V* *i w32 w33* ≡
if ((*fst i*) = *arith-type UDIV*) ∨ ((*fst i*) = *arith-type UDIVcc*) *then*
if w32 = 0 then 0 else 1
else — Must be *SDIV* or *SDIVcc*.
if (w33 = 0) ∨ (w33 = (0b11111111111111111111111111111111::word33))
then 0 else 1

definition *div-comp-result* :: *instruction* ⇒ *word1* ⇒ *word64* ⇒ *word32*

where *div-comp-result* *i temp-V temp-64bit* ≡
if temp-V = 1 then
if ((*fst i*) = *arith-type UDIV*) ∨ ((*fst i*) = *arith-type UDIVcc*) *then*
 (0b11111111111111111111111111111111::word32)
else if (fst i) ∈ {arith-type SDIV,arith-type SDIVcc} *then*
if temp-64bit > 0 then


```

    op-list = snd instr;
    rs1 = get-operand-w5 (op-list!1);
    rd = get-operand-w5 (op-list!3)
  in
  do
    operand2 ← gets (λs. (get-operand2 op-list s));
    if (uint operand2) = 0 then
      do
        raise-trap division-by-zero;
        return ()
      od
    else
      div-comp instr rs1 rd operand2
    od

```

definition $ld\text{-}word0 :: instruction \Rightarrow word32 \Rightarrow virtua\text{-}address \Rightarrow word32$

where $ld\text{-}word0\ instr\ data\text{-}word\ address \equiv$

```

  if (fst instr) ∈ {load-store-type LDSB,load-store-type LDUB,
    load-store-type LDUBA,load-store-type LDSBA} then
    let byte = if (uint ((ucast address)::word2)) = 0 then
      (ucast (data-word >> 24))::word8
    else if (uint ((ucast address)::word2)) = 1 then
      (ucast (data-word >> 16))::word8
    else if (uint ((ucast address)::word2)) = 2 then
      (ucast (data-word >> 8))::word8
    else — Must be 3.
      (ucast data-word)::word8
  in
  if (fst instr) = load-store-type LDSB ∨ (fst instr) = load-store-type LDSBA
  then
    sign-ext8 byte
  else
    zero-ext8 byte
  else if (fst instr) = load-store-type LDUH ∨ (fst instr) = load-store-type LDSH
  ∨
  (fst instr) = load-store-type LDSHA ∨ (fst instr) = load-store-type LDUHA
  then
    let halfword = if (uint ((ucast address)::word2)) = 0 then
      (ucast (data-word >> 16))::word16
    else — Must be 2.
      (ucast data-word)::word16
  in
  if (fst instr) = load-store-type LDSH ∨ (fst instr) = load-store-type LDSHA
  then
    sign-ext16 halfword
  else
    zero-ext16 halfword
  else — Must be LDD
    data-word

```

definition *ld-asi* :: *instruction* ⇒ *word1* ⇒ *asi-type*
where *ld-asi instr s-val* ≡
 if (*fst instr*) ∈ {*load-store-type LDD*,*load-store-type LD*,*load-store-type LDUH*,
load-store-type LDSB,*load-store-type LDUB*,*load-store-type LDSH*} then
 if *s-val* = 0 then (*word-of-int 10*)::*asi-type*
 else (*word-of-int 11*)::*asi-type*
 else — Must be *LDA*, *LDUBA*, *LDSBA*, *LDSHA*, *LDUHA*, or *LDDA*.
get-operand-asi ((*snd instr*)!3)

definition *load-sub2* :: *virtua-address* ⇒ *asi-type* ⇒ *word5* ⇒
 (*'a::len*) *window-size* ⇒ *word32* ⇒ (*'a,unit*) *sparc-state-monad*
where *load-sub2 address asi rd curr-win word0* ≡
 do
 write-reg word0 curr-win ((AND) rd 0b111110);
 (*result1,new-state1*) ← *gets* (*λs. (memory-read asi (address + 4) s)*);
 if *result1* = *None* then
 do
 raise-trap data-access-exception;
 return ()
 od
 else
 do
 word1 ← *gets* (*λs. (case result1 of Some v ⇒ v)*);
 modify (*λs. (new-state1)*);
 write-reg word1 curr-win ((OR) rd 1);
 return ()
 od
 od

definition *load-sub3* :: *instruction* ⇒ (*'a::len*) *window-size* ⇒
word5 ⇒ *asi-type* ⇒ *virtua-address* ⇒
 (*'a::len,unit*) *sparc-state-monad*
where *load-sub3 instr curr-win rd asi address* ≡
 do
 (*result,new-state*) ← *gets* (*λs. (memory-read asi address s)*);
 if *result* = *None* then
 do
 raise-trap data-access-exception;
 return ()
 od
 else
 do
 data-word ← *gets* (*λs. (case result of Some v ⇒ v)*);
 modify (*λs. (new-state)*);
 word0 ← *gets* (*λs. (ld-word0 instr data-word address)*);
 if *rd* ≠ 0 ∧ (*fst instr*) ∈ {*load-store-type LD*,*load-store-type LDA*,

```

    load-store-type LDUH,load-store-type LDSB,load-store-type LDUB,
    load-store-type LDUBA,load-store-type LDSH,load-store-type LDSHA,
    load-store-type LDUHA,load-store-type LDSBA} then
  do
    write-reg word0 curr-win rd;
    return ()
  od
  else — Must be LDD or LDDA
    load-sub2 address asi rd curr-win word0
  od
od

```

definition $load-sub1 :: instruction \Rightarrow word5 \Rightarrow word1 \Rightarrow$
 $('a::len,unit) \text{ sparc-state-monad}$

where $load-sub1 \text{ instr rd s-val} \equiv$

```

  do
    curr-win  $\leftarrow$  get-curr-win();
    address  $\leftarrow$  gets ( $\lambda s. (get-addr (snd \text{instr}) s)$ );
    asi  $\leftarrow$  gets ( $\lambda s. (ld-asi \text{instr} s\text{-val})$ );
    if (((fst instr) = load-store-type LDD  $\vee$  (fst instr) = load-store-type LDDA)
         $\wedge$  ((ucast address)::word3)  $\neq$  0)
         $\vee$  ((fst instr)  $\in$  {load-store-type LD,load-store-type LDA}
             $\wedge$  ((ucast address)::word2)  $\neq$  0)
         $\vee$  (((fst instr) = load-store-type LDUH  $\vee$  (fst instr) = load-store-type
LDUHA
LDSHA)
             $\vee$  (fst instr) = load-store-type LDSH  $\vee$  (fst instr) = load-store-type
LDSHA)
             $\wedge$  ((ucast address)::word1)  $\neq$  0)
    then
      do
        raise-trap mem-address-not-aligned;
        return ()
      od
    else
      load-sub3 instr curr-win rd asi address
    od

```

Operational semantics for Load instructions.

definition $load-instr :: instruction \Rightarrow ('a::len,unit) \text{ sparc-state-monad}$

where $load-instr \text{ instr} \equiv$

```

  let instr-name = fst instr;
      op-list = snd instr;
      flagi = get-operand-flag (op-list!0);
      rd = if instr-name  $\in$  {load-store-type LDUBA,load-store-type LDA,
load-store-type LDSBA,load-store-type LDSHA,
load-store-type LDSHA,load-store-type LDDA} then — rd is member 4
        get-operand-w5 (op-list!4)
      else — rd is member 3
        get-operand-w5 (op-list!3)

```

```

in
do
  psr-val ← gets (λs. (cpu-reg-val PSR s));
  s-val ← gets (λs. (get-S psr-val));
  if instr-name ∈ {load-store-type LDA,load-store-type LDUBA,
    load-store-type LDSBA,load-store-type LDSHA,
    load-store-type LDUHA,load-store-type LDDA} ∧ s-val = 0 then
  do
    raise-trap privileged-instruction;
    return ()
  od
  else if instr-name ∈ {load-store-type LDA,load-store-type LDUBA,
    load-store-type LDSBA,load-store-type LDSHA,load-store-type LDUHA,
    load-store-type LDDA} ∧ flagi = 1 then
  do
    raise-trap illegal-instruction;
    return ()
  od
  else
    load-sub1 instr rd s-val
od

```

definition *st-asi* :: instruction ⇒ word1 ⇒ asi-type

where *st-asi instr s-val* ≡

```

if (fst instr) ∈ {load-store-type STD,load-store-type ST,
  load-store-type STH,load-store-type STB} then
  if s-val = 0 then (word-of-int 10)::asi-type
  else (word-of-int 11)::asi-type
else — Must be STA, STBA, STHA, STDA.
  get-operand-asi ((snd instr)!3)

```

definition *st-byte-mask* :: instruction ⇒ virtua-address ⇒ word4

where *st-byte-mask instr address* ≡

```

if (fst instr) ∈ {load-store-type STD,load-store-type ST,
  load-store-type STA,load-store-type STDA} then
  (0b1111::word4)
else if (fst instr) ∈ {load-store-type STH,load-store-type STHA} then
  if ((ucast address)::word2) = 0 then
    (0b1100::word4)
  else — Must be 2.
    (0b0011::word4)
else — Must be STB or STBA.
  if ((ucast address)::word2) = 0 then
    (0b1000::word4)
  else if ((ucast address)::word2) = 1 then
    (0b0100::word4)
  else if ((ucast address)::word2) = 2 then
    (0b0010::word4)

```

else — Must be 3.
 (0b0001::word4)

definition *st-data0* :: instruction ⇒ ('a::len) window-size ⇒
 word5 ⇒ virtua-address ⇒ ('a) sparc-state ⇒ reg-type
where *st-data0 instr curr-win rd address s* ≡
 if (fst instr) ∈ {load-store-type STD,load-store-type STDA} then
 user-reg-val curr-win ((AND) rd 0b11110) s
 else if (fst instr) ∈ {load-store-type ST,load-store-type STA} then
 user-reg-val curr-win rd s
 else if (fst instr) ∈ {load-store-type STH,load-store-type STHA} then
 if ((ucast address)::word2) = 0 then
 (user-reg-val curr-win rd s) << 16
 else — Must be 2.
 user-reg-val curr-win rd s
 else — Must be STB or STBA.
 if ((ucast address)::word2) = 0 then
 (user-reg-val curr-win rd s) << 24
 else if ((ucast address)::word2) = 1 then
 (user-reg-val curr-win rd s) << 16
 else if ((ucast address)::word2) = 2 then
 (user-reg-val curr-win rd s) << 8
 else — Must be 3.
 user-reg-val curr-win rd s

definition *store-sub2* :: instruction ⇒ ('a::len) window-size ⇒
 word5 ⇒ asi-type ⇒ virtua-address ⇒
 ('a::len,unit) sparc-state-monad
where *store-sub2 instr curr-win rd asi address* ≡
 do
 byte-mask ← gets (λs. (st-byte-mask instr address));
 data0 ← gets (λs. (st-data0 instr curr-win rd address s));
 result0 ← gets (λs. (memory-write asi address byte-mask data0 s));
 if result0 = None then
 do
 raise-trap data-access-exception;
 return ()
 od
 else
 do
 new-state ← gets (λs. (case result0 of Some v ⇒ v));
 modify (λs. (new-state));
 if (fst instr) ∈ {load-store-type STD,load-store-type STDA} then
 do
 data1 ← gets (λs. (user-reg-val curr-win ((OR) rd 0b00001) s));
 result1 ← gets (λs. (memory-write asi (address + 4) (0b1111::word4) data1
 s));

```

    if result1 = None then
    do
        raise-trap data-access-exception;
        return ()
    od
    else
    do
        new-state1 ← gets (λs. (case result1 of Some v ⇒ v));
        modify (λs. (new-state1));
        return ()
    od
    od
    else
    return ()
    od
od

```

definition *store-sub1* :: *instruction* ⇒ *word5* ⇒ *word1* ⇒
 ('a::len,unit) *sparc-state-monad*

where *store-sub1 instr rd s-val* ≡

```

do
    curr-win ← get-curr-win();
    address ← gets (λs. (get-addr (snd instr) s));
    asi ← gets (λs. (st-asi instr s-val));
    — The following code is intentionally long to match the definitions in SPARCV8.
    if ((fst instr) = load-store-type STH ∨ (fst instr) = load-store-type STHA)
        ∧ ((ucast address)::word1) ≠ 0 then
    do
        raise-trap mem-address-not-aligned;
        return ()
    od
    else if (fst instr) ∈ {load-store-type ST, load-store-type STA}
        ∧ ((ucast address)::word2) ≠ 0 then
    do
        raise-trap mem-address-not-aligned;
        return ()
    od
    else if (fst instr) ∈ {load-store-type STD, load-store-type STDA}
        ∧ ((ucast address)::word3) ≠ 0 then
    do
        raise-trap mem-address-not-aligned;
        return ()
    od
    else
        store-sub2 instr curr-win rd asi address
    od
od

```

Operational semantics for Store instructions.

definition *store-instr* :: *instruction* ⇒

```

('a::len,unit) sparc-state-monad
where store-instr instr ≡
  let instr-name = fst instr;
      op-list = snd instr;
      flagi = get-operand-flag (op-list!0);
      rd = if instr-name ∈ {load-store-type STA,load-store-type STBA,
        load-store-type STHA,load-store-type STDA} then — rd is member 4
            get-operand-w5 (op-list!4)
          else — rd is member 3
            get-operand-w5 (op-list!3)
  in
  do
    psr-val ← gets (λs. (cpu-reg-val PSR s));
    s-val ← gets (λs. (get-S psr-val));
    if instr-name ∈ {load-store-type STA,load-store-type STDA,
      load-store-type STHA,load-store-type STBA} ∧ s-val = 0 then
      do
        raise-trap privileged-instruction;
        return ()
      od
    else if instr-name ∈ {load-store-type STA,load-store-type STDA,
      load-store-type STHA,load-store-type STBA} ∧ flagi = 1 then
      do
        raise-trap illegal-instruction;
        return ()
      od
    else
      store-sub1 instr rd s-val
  od

```

The instructions below are not used by Xtratum and they are not tested.

```

definition ldst-asi :: instruction ⇒ word1 ⇒ asi-type
where ldst-asi instr s-val ≡
  if (fst instr) ∈ {load-store-type LDSTUB} then
    if s-val = 0 then (word-of-int 10)::asi-type
    else (word-of-int 11)::asi-type
  else — Must be LDSTUBA.
    get-operand-asi ((snd instr)!3)

```

```

definition ldst-word0 :: instruction ⇒ word32 ⇒ virtua-address ⇒ word32
where ldst-word0 instr data-word address ≡
  let byte = if (wint ((ucast address)::word2)) = 0 then
    (ucast (data-word >> 24))::word8
  else if (wint ((ucast address)::word2)) = 1 then
    (ucast (data-word >> 16))::word8
  else if (wint ((ucast address)::word2)) = 2 then
    (ucast (data-word >> 8))::word8
  else — Must be 3.

```

```

                (ucast data-word)::word8
in
zero-ext8 byte

```

definition *ldst-byte-mask* :: *instruction* ⇒ *virtua-address* ⇒ *word4*
where *ldst-byte-mask instr address* ≡
 if ((ucast address)::word2) = 0 then
 (0b1000::word4)
 else if ((ucast address)::word2) = 1 then
 (0b0100::word4)
 else if ((ucast address)::word2) = 2 then
 (0b0010::word4)
 else — Must be 3.
 (0b0001::word4)

definition *load-store-sub1* :: *instruction* ⇒ *word5* ⇒ *word1* ⇒
 ('a::len,unit) *sparc-state-monad*

where *load-store-sub1 instr rd s-val* ≡
 do
 curr-win ← get-curr-win();
 address ← gets (λs. (get-addr (snd instr) s));
 asi ← gets (λs. (ldst-asi instr s-val));
 — wait for locks to be lifted.
 — an implementation actually need only block when another *LDSTUB* or *SWAP*
 — is pending on the same byte in memory as the one addressed by this *LDSTUB*
 — Should wait when *block-type* = 1 ∨ *block-word* = 1
 — until another processes write both to be 0.
 — We implement this as setting *pc* as *npc* when the instruction
 — is blocked. This way, in the next iteration, we will still execution
 — the current instruction.
 block-byte ← gets (λs. (pb-block-ldst-byte-val address s));
 block-word ← gets (λs. (pb-block-ldst-word-val address s));
 if block-byte ∨ block-word then
 do
 pc-val ← gets (λs. (cpu-reg-val PC s));
 write-cpu pc-val npc;
 return ()
 od
 else
 do
 modify (λs. (pb-block-ldst-byte-mod address True s));
 (result,new-state) ← gets (λs. (memory-read asi address s));
 if result = None then
 do
 raise-trap data-access-exception;
 return ()
 od

```

else
do
  data-word ← gets (λs. (case result of Some v ⇒ v));
  modify (λs. (new-state));
  byte-mask ← gets (λs. (ldst-byte-mask instr address));
  data0 ← gets (λs. (0b11111111111111111111111111111111::word32));
  result0 ← gets (λs. (memory-write asi address byte-mask data0 s));
  modify (λs. (pb-block-ldst-byte-mod address False s));
  if result0 = None then
  do
    raise-trap data-access-exception;
    return ()
  od
else
do
  new-state1 ← gets (λs. (case result0 of Some v ⇒ v));
  modify (λs. (new-state1));
  word0 ← gets (λs. (ldst-word0 instr data-word address));
  if rd ≠ 0 then
  do
    write-reg word0 curr-win rd;
    return ()
  od
else
  return ()
od
od
od
od

```

Operational semantics for atomic load-store.

definition *load-store-instr* :: *instruction* ⇒ ('a::len,unit) *sparc-state-monad*
where *load-store-instr instr* ≡
 let *instr-name* = *fst instr*;
 op-list = *snd instr*;
 flagi = *get-operand-flag (op-list!0)*;
 rd = if *instr-name* ∈ {*load-store-type LDSTUBA*} then — *rd* is member 4
 get-operand-w5 (op-list!4)
 else — *rd* is member 3
 get-operand-w5 (op-list!3)
 in
 do
 psr-val ← *gets (λs. (cpu-reg-val PSR s))*;
 s-val ← *gets (λs. (get-S psr-val))*;
 if *instr-name* ∈ {*load-store-type LDSTUBA*} ∧ *s-val* = 0 then
 do
 raise-trap privileged-instruction;
 return ()
 od
 od

```

else if instr-name ∈ {load-store-type LDSTUBA} ∧ flagi = 1 then
  do
    raise-trap illegal-instruction;
    return ()
  od
else
  load-store-sub1 instr rd s-val
od

```

definition *swap-sub1* :: instruction ⇒ word5 ⇒ word1 ⇒
('a::len,unit) sparc-state-monad

where *swap-sub1* instr rd s-val ≡

```

do
  curr-win ← get-curr-win();
  address ← gets (λs. (get-addr (snd instr) s));
  asi ← gets (λs. (ldst-asi instr s-val));
  temp ← gets (λs. (user-reg-val curr-win rd s));
  — wait for locks to be lifted.
  — an implementation actually need only block when another LDSTUB or SWAP
  — is pending on the same byte in memory as the one addressed by this LDSTUB
  — Should wait when block-type = 1 ∨ block-word = 1
  — until another processes write both to be 0.
  — We implement this as setting pc as npc when the instruction
  — is blocked. This way, in the next iteration, we will still execution
  — the current instruction.
  block-byte ← gets (λs. (pb-block-ldst-byte-val address s));
  block-word ← gets (λs. (pb-block-ldst-word-val address s));
  if block-byte ∨ block-word then
  do
    pc-val ← gets (λs. (cpu-reg-val PC s));
    write-cpu pc-val nPC;
    return ()
  od
else
  do
    modify (λs. (pb-block-ldst-word-mod address True s));
    (result,new-state) ← gets (λs. (memory-read asi address s));
    if result = None then
    do
      raise-trap data-access-exception;
      return ()
    od
  else
  do
    word ← gets (λs. (case result of Some v ⇒ v));
    modify (λs. (new-state));
    byte-mask ← gets (λs. (0b1111::word4));
    result0 ← gets (λs. (memory-write asi address byte-mask temp s));
    modify (λs. (pb-block-ldst-word-mod address False s));

```

```

    if result0 = None then
    do
        raise-trap data-access-exception;
        return ()
    od
    else
    do
        new-state1 ← gets (λs. (case result0 of Some v ⇒ v));
        modify (λs. (new-state1));
        if rd ≠ 0 then
        do
            write-reg word curr-win rd;
            return ()
        od
        else
            return ()
        od
    od
od
od
od

```

Operational semantics for swap.

definition *swap-instr* :: *instruction* ⇒ ('a::len,unit) *sparc-state-monad*
where *swap-instr instr* ≡
 let *instr-name* = *fst instr*;
 op-list = *snd instr*;
 flagi = *get-operand-flag (op-list!0)*;
 rd = if *instr-name* ∈ {*load-store-type SWAPA*} then — *rd* is member 4
 get-operand-w5 (op-list!4)
 else — *rd* is member 3
 get-operand-w5 (op-list!3)
 in
 do
 psr-val ← *gets (λs. (cpu-reg-val PSR s))*;
 s-val ← *gets (λs. (get-S psr-val))*;
 if *instr-name* ∈ {*load-store-type SWAPA*} ∧ *s-val* = 0 then
 do
 raise-trap *privileged-instruction*;
 return ()
 od
 else if *instr-name* ∈ {*load-store-type SWAPA*} ∧ *flagi* = 1 then
 do
 raise-trap *illegal-instruction*;
 return ()
 od
 else
 swap-sub1 instr rd s-val
 od

definition *bit2-zero* :: *word2* \Rightarrow *word1*
where *bit2-zero* *w2* \equiv if *w2* \neq 0 then 1 else 0

Operational semantics for tagged add instructions.

definition *tadd-instr* :: *instruction* \Rightarrow ('a::len,unit) *sparc-state-monad*
where *tadd-instr* *instr* \equiv

let *instr-name* = *fst instr*;

op-list = *snd instr*;

rs1 = *get-operand-w5 (op-list!1)*;

rd = *get-operand-w5 (op-list!3)*

in

do

operand2 \leftarrow *gets* ($\lambda s.$ (*get-operand2 op-list s*));

curr-win \leftarrow *get-curr-win*();

rs1-val \leftarrow *gets* ($\lambda s.$ (*user-reg-val curr-win rs1 s*));

psr-val \leftarrow *gets* ($\lambda s.$ (*cpu-reg-val PSR s*));

c-val \leftarrow *gets* ($\lambda s.$ (*get-icc-C psr-val*));

result \leftarrow *gets* ($\lambda s.$ (*rs1-val + operand2*));

result-31 \leftarrow *gets* ($\lambda s.$ (*ucast (result >> 31)::word1*));

rs1-val-31 \leftarrow *gets* ($\lambda s.$ (*ucast (rs1-val >> 31)::word1*));

operand2-31 \leftarrow *gets* ($\lambda s.$ (*ucast (operand2 >> 31)::word1*));

rs1-val-2 \leftarrow *gets* ($\lambda s.$ (*bit2-zero (ucast rs1-val)::word2*));

operand2-2 \leftarrow *gets* ($\lambda s.$ (*bit2-zero (ucast operand2)::word2*));

temp-V \leftarrow *gets* ($\lambda s.$ (*(OR) ((OR) ((AND) rs1-val-31*
 (*(AND) operand2-31*
 (*NOT result-31*)))
 (*(AND) (NOT rs1-val-31)*
 (*(AND) (NOT operand2-31)*
 result-31)))

 (*(OR) rs1-val-2 operand2-2*));

 if *instr-name* = *arith-type TADDccTV* \wedge *temp-V* = 1 then

 do

raise-trap tag-overflow;

 return ()

 od

 else

 do

rd-val \leftarrow *gets* ($\lambda s.$ (*user-reg-val curr-win rd s*));

new-rd-val \leftarrow *gets* ($\lambda s.$ (*if rd \neq 0 then result else rd-val*));

write-reg new-rd-val curr-win rd;

new-n-val \leftarrow *gets* ($\lambda s.$ (*result-31*));

new-z-val \leftarrow *gets* ($\lambda s.$ (*if result = 0 then 1::word1 else 0::word1*));

new-v-val \leftarrow *gets* ($\lambda s.$ *temp-V*);

new-c-val \leftarrow *gets* ($\lambda s.$ (*(OR) ((AND) rs1-val-31*
 (*operand2-31*
 (*(AND) (NOT result-31)*
 (*(OR) rs1-val-31*
 (*operand2-31*)))));

new-psr-val \leftarrow *gets* ($\lambda s.$ (*update-PSR-icc new-n-val*

```

new-z-val
new-v-val
new-c-val psr-val));
write-cpu new-psr-val PSR;
rd-val ← gets (λs. (user-reg-val curr-win rd s));
new-rd-val ← gets (λs. (if rd ≠ 0 then result else rd-val));
write-reg new-rd-val curr-win rd;
return ()
od
od

```

Operational semantics for tagged add instructions.

definition $tsub\text{-}instr :: instruction \Rightarrow ('a::len,unit) \text{ sparc-state-monad}$

where $tsub\text{-}instr\ instr \equiv$

```

let instr-name = fst instr;
op-list = snd instr;
rs1 = get-operand-w5 (op-list!1);
rd = get-operand-w5 (op-list!3)
in
do
operand2 ← gets (λs. (get-operand2 op-list s));
curr-win ← get-curr-win();
rs1-val ← gets (λs. (user-reg-val curr-win rs1 s));
psr-val ← gets (λs. (cpu-reg-val PSR s));
c-val ← gets (λs. (get-icc-C psr-val));
result ← gets (λs. (rs1-val - operand2));
result-31 ← gets (λs. ((ucast (result >> 31))::word1));
rs1-val-31 ← gets (λs. ((ucast (rs1-val >> 31))::word1));
operand2-31 ← gets (λs. ((ucast (operand2 >> 31))::word1));
rs1-val-2 ← gets (λs. (bit2-zero ((ucast rs1-val)::word2)));
operand2-2 ← gets (λs. (bit2-zero ((ucast operand2)::word2)));
temp-V ← gets (λs. ((OR) ((OR) ((AND) rs1-val-31
((AND) operand2-31
(NOT result-31)))
((AND) (NOT rs1-val-31)
((AND) (NOT operand2-31)
result-31))))
((OR) rs1-val-2 operand2-2));
if instr-name = arith-type TSUBccTV ∧ temp-V = 1 then
do
raise-trap tag-overflow;
return ()
od
else
do
rd-val ← gets (λs. (user-reg-val curr-win rd s));
new-rd-val ← gets (λs. (if rd ≠ 0 then result else rd-val));
write-reg new-rd-val curr-win rd;
new-n-val ← gets (λs. (result-31));

```

```

new-z-val ← gets (λs. (if result = 0 then 1::word1 else 0::word1));
new-v-val ← gets (λs. temp-V);
new-c-val ← gets (λs. ((OR) ((AND) rs1-val-31
                             operand2-31)
                          ((AND) (NOT result-31)
                                  ((OR) rs1-val-31
                                         operand2-31))));
new-psr-val ← gets (λs. (update-PSR-icc new-n-val
                                       new-z-val
                                       new-v-val
                                       new-c-val psr-val));

write-cpu new-psr-val PSR;
rd-val ← gets (λs. (user-reg-val curr-win rd s));
new-rd-val ← gets (λs. (if rd ≠ 0 then result else rd-val));
write-reg new-rd-val curr-win rd;
return ()
od
od

```

definition *muls-op2* :: *inst-operand list* ⇒ ('a::len) *sparc-state* ⇒ *word32*
where *muls-op2 op-list s* ≡
let y-val = cpu-reg-val Y s in
if ((ucast y-val)::word1) = 0 then 0
else get-operand2 op-list s

Operational semantics for multiply step instruction.

definition *muls-instr* :: *instruction* ⇒ ('a::len,unit) *sparc-state-monad*
where *muls-instr instr* ≡
let instr-name = fst instr;
op-list = snd instr;
rs1 = get-operand-w5 (op-list!1);
rd = get-operand-w5 (op-list!3)
in
do
curr-win ← get-curr-win();
rs1-val ← gets (λs. (user-reg-val curr-win rs1 s));
psr-val ← gets (λs. (cpu-reg-val PSR s));
n-val ← gets (λs. (get-icc-N psr-val));
v-val ← gets (λs. (get-icc-V psr-val));
c-val ← gets (λs. (get-icc-C psr-val));
y-val ← gets (λs. (cpu-reg-val Y s));
operand1 ← gets (λs. (word-cat ((XOR) n-val v-val)
 ((ucast (rs1-val >> 1))::word31)));
operand2 ← gets (λs. (muls-op2 op-list s));
result ← gets (λs. (operand1 + operand2));
new-y-val ← gets (λs. (word-cat ((ucast rs1-val)::word1) ((ucast (y-val >>
1))::word31)));
write-cpu new-y-val Y;

```

rd-val ← gets (λs. (user-reg-val curr-win rd s));
new-rd-val ← gets (λs. (if rd ≠ 0 then result else rd-val));
write-reg new-rd-val curr-win rd;
result-31 ← gets (λs. ((ucast (result >> 31))::word1));
operand1-31 ← gets (λs. ((ucast (operand1 >> 31))::word1));
operand2-31 ← gets (λs. ((ucast (operand2 >> 31))::word1));
new-n-val ← gets (λs. (result-31));
new-z-val ← gets (λs. (if result = 0 then 1::word1 else 0::word1));
new-v-val ← gets (λs. ((OR) ((AND) operand1-31
                             ((AND) operand2-31
                              (NOT result-31)))
                      ((AND) (NOT operand1-31)
                              ((AND) (NOT operand2-31)
                                       result-31))));
new-c-val ← gets (λs. ((OR) ((AND) operand1-31
                             operand2-31)
                      ((AND) (NOT result-31)
                              ((OR) operand1-31
                                       operand2-31))));
new-psr-val ← gets (λs. (update-PSR-icc new-n-val
                                   new-z-val
                                   new-v-val
                                   new-c-val psr-val));

write-cpu new-psr-val PSR;
return ()
od

```

Evaluate *icc* based on the bits N, Z, V, C in PSR and the type of *ticc* instruction. See Sparcv8 manual Page 182.

definition *trap-eval-icc::sparc-operation* ⇒ *word1* ⇒ *word1* ⇒ *word1* ⇒ *word1* ⇒ *int*

where *trap-eval-icc instr-name n-val z-val v-val c-val* ≡
 if *instr-name* = *ticc-type TNE* then
 if *z-val* = 0 then 1 else 0
 else if *instr-name* = *ticc-type TE* then
 if *z-val* = 1 then 1 else 0
 else if *instr-name* = *ticc-type TG* then
 if ((OR) *z-val* (*n-val XOR v-val*)) = 0 then 1 else 0
 else if *instr-name* = *ticc-type TLE* then
 if ((OR) *z-val* (*n-val XOR v-val*)) = 1 then 1 else 0
 else if *instr-name* = *ticc-type TGE* then
 if (*n-val XOR v-val*) = 0 then 1 else 0
 else if *instr-name* = *ticc-type TL* then
 if (*n-val XOR v-val*) = 1 then 1 else 0
 else if *instr-name* = *ticc-type TGU* then
 if (*c-val* = 0 ∧ *z-val* = 0) then 1 else 0
 else if *instr-name* = *ticc-type TLEU* then
 if (*c-val* = 1 ∨ *z-val* = 1) then 1 else 0
 else if *instr-name* = *ticc-type TCC* then

```

    if c-val = 0 then 1 else 0
  else if instr-name = ticc-type TCS then
    if c-val = 1 then 1 else 0
  else if instr-name = ticc-type TPOS then
    if n-val = 0 then 1 else 0
  else if instr-name = ticc-type TNEG then
    if n-val = 1 then 1 else 0
  else if instr-name = ticc-type TVC then
    if v-val = 0 then 1 else 0
  else if instr-name = ticc-type TVS then
    if v-val = 1 then 1 else 0
  else if instr-name = ticc-type TA then 1
  else if instr-name = ticc-type TN then 0
  else -1

```

Get *operand2* for *ticc* based on the flag *i*, *rs1*, *rs2*, and *trap-imm7*. If *i* = 0 then *operand2* = *r[rs2]*, else *operand2* = *sign-ext7(trap-imm7)*. *op-list* should be [*i,rs1,rs2*] or [*i,rs1,trap-imm7*].

definition *get-trap-op2::inst-operand list* \Rightarrow (*a::len*) *sparc-state*
 \Rightarrow *virtua-address*
where *get-trap-op2 op-list s* \equiv
 let *flagi* = *get-operand-flag (op-list!0)*;
 curr-win = *ucast (get-CWP (cpu-reg-val PSR s))*
 in
 if *flagi* = 0 then
 let *rs2* = *get-operand-w5 (op-list!2)*;
 rs2-val = *user-reg-val curr-win rs2 s*
 in *rs2-val*
 else
 let *ext-simm7* = *sign-ext7 (get-operand-imm7 (op-list!2))* in
 ext-simm7

Operational semantics for Ticc instructions.

definition *ticc-instr::instruction* \Rightarrow
 (*a::len,unit*) *sparc-state-monad*
where *ticc-instr instr* \equiv
 let *instr-name* = *fst instr*;
 op-list = *snd instr*;
 rs1 = *get-operand-w5 (op-list!1)*
 in
 do
 n-val \leftarrow *gets* ($\lambda s.$ *get-icc-N ((cpu-reg s) PSR)*);
 z-val \leftarrow *gets* ($\lambda s.$ *get-icc-Z ((cpu-reg s) PSR)*);
 v-val \leftarrow *gets* ($\lambda s.$ *get-icc-V ((cpu-reg s) PSR)*);
 c-val \leftarrow *gets* ($\lambda s.$ *get-icc-C ((cpu-reg s) PSR)*);
 icc-val \leftarrow *gets*($\lambda s.$ (*trap-eval-icc instr-name n-val z-val v-val c-val*));
 curr-win \leftarrow *get-curr-win()*;

```

rs1-val ← gets (λs. (user-reg-val curr-win rs1 s));
trap-number ← gets (λs. (rs1-val + (get-trap-op2 op-list s)));
npc-val ← gets (λs. (cpu-reg-val nPC s));
pc-val ← gets (λs. (cpu-reg-val PC s));
if icc-val = 1 then
  do
    raise-trap trap-instruction;
    trap-number7 ← gets (λs. ((ucast trap-number)::word7));
    modify (λs. (ticc-trap-type-mod trap-number7 s));
    return ()
  od
else — icc-val = 0
  do
    write-cpu npc-val PC;
    write-cpu (npc-val + 4) nPC;
    return ()
  od
od

```

Operational semantics for store barrier.

```

definition store-barrier-instr::instruction ⇒ ('a::len,unit) sparc-state-monad
where store-barrier-instr instr ≡
  do
    modify (λs. (store-barrier-pending-mod True s));
    return ()
  od
end
end

```

theory Sparc-Execution

imports Main Sparc-Instruction Sparc-State Sparc-Types
HOL-Eisbach.Eisbach-Tools

begin

```

primrec sum :: nat ⇒ nat where
sum 0 = 0 |
sum (Suc n) = Suc n + sum n

```

```

definition select-trap :: unit ⇒ ('a,unit) sparc-state-monad
where select-trap - ≡
  do
    traps ← gets (λs. (get-trap-set s));
    rt-val ← gets (λs. (reset-trap-val s));
    psr-val ← gets (λs. (cpu-reg-val PSR s));
    et-val ← gets (λs. (get-ET psr-val));
    modify (λs. (emp-trap-set s));

```

```

if rt-val = True then — ignore ET, and leave tt unchanged
    return ()
else if et-val = 0 then — go to error mode, machine needs reset
    do
        set-err-mode True;
        set-exe-mode False;
        fail ()
    od
— By the SPARCv8 manual only 1 of the following traps could be in traps.
else if data-store-error ∈ traps then
    do
        write-cpu-tt (0b00101011::word8);
        return ()
    od
else if instruction-access-error ∈ traps then
    do
        write-cpu-tt (0b00100001::word8);
        return ()
    od
else if r-register-access-error ∈ traps then
    do
        write-cpu-tt (0b00100000::word8);
        return ()
    od
else if instruction-access-exception ∈ traps then
    do
        write-cpu-tt (0b00000001::word8);
        return ()
    od
else if privileged-instruction ∈ traps then
    do
        write-cpu-tt (0b00000011::word8);
        return ()
    od
else if illegal-instruction ∈ traps then
    do
        write-cpu-tt (0b00000010::word8);
        return ()
    od
else if fp-disabled ∈ traps then
    do
        write-cpu-tt (0b00000100::word8);
        return ()
    od
else if cp-disabled ∈ traps then
    do
        write-cpu-tt (0b00100100::word8);
        return ()
    od

```

```

else if unimplemented-FLUSH ∈ traps then
  do
    write-cpu-tt (0b00100101::word8);
    return ()
  od
else if window-overflow ∈ traps then
  do
    write-cpu-tt (0b00000101::word8);
    return ()
  od
else if window-underflow ∈ traps then
  do
    write-cpu-tt (0b00000110::word8);
    return ()
  od
else if mem-address-not-aligned ∈ traps then
  do
    write-cpu-tt (0b00000111::word8);
    return ()
  od
else if fp-exception ∈ traps then
  do
    write-cpu-tt (0b00001000::word8);
    return ()
  od
else if cp-exception ∈ traps then
  do
    write-cpu-tt (0b00101000::word8);
    return ()
  od
else if data-access-error ∈ traps then
  do
    write-cpu-tt (0b00101001::word8);
    return ()
  od
else if data-access-exception ∈ traps then
  do
    write-cpu-tt (0b00001001::word8);
    return ()
  od
else if tag-overflow ∈ traps then
  do
    write-cpu-tt (0b00001010::word8);
    return ()
  od
else if division-by-zero ∈ traps then
  do
    write-cpu-tt (0b00101010::word8);
    return ()
  od

```

```

    od
  else if trap-instruction ∈ traps then
    do
      ticc-trap-type ← gets (λs. (ticc-trap-type-val s));
      write-cpu-tt (word-cat (1::word1) ticc-trap-type);
      return ()
    od
  else if interrupt-level ≠ 0 then
  — We don't consider interrupt-level
  else return ()
od

```

definition *exe-trap-st-pc* :: unit ⇒ ('a::len,unit) sparc-state-monad

where *exe-trap-st-pc* - ≡

```

do
  annul ← gets (λs. (annul-val s));
  pc-val ← gets (λs. (cpu-reg-val PC s));
  npc-val ← gets (λs. (cpu-reg-val nPC s));
  curr-win ← get-curr-win();
  if annul = False then
    do
      write-reg pc-val curr-win (word-of-int 17);
      write-reg npc-val curr-win (word-of-int 18);
      return ()
    od
  else — annul = True
    do
      write-reg npc-val curr-win (word-of-int 17);
      write-reg (npc-val + 4) curr-win (word-of-int 18);
      set-annul False;
      return ()
    od
od

```

definition *exe-trap-wr-pc* :: unit ⇒ ('a::len,unit) sparc-state-monad

where *exe-trap-wr-pc* - ≡

```

do
  psr-val ← gets (λs. (cpu-reg-val PSR s));
  new-psr-val ← gets (λs. (update-S (1::word1) psr-val));
  write-cpu new-psr-val PSR;
  reset-trap ← gets (λs. (reset-trap-val s));
  tbr-val ← gets (λs. (cpu-reg-val TBR s));
  if reset-trap = False then
    do
      write-cpu tbr-val PC;
      write-cpu (tbr-val + 4) nPC;
      return ()
    od
  else — reset-trap = True

```

```

do
  write-cpu 0 PC;
  write-cpu 4 nPC;
  set-reset-trap False;
  return ()
od
od

```

definition *execute-trap* :: unit ⇒ ('a::len,unit) sparc-state-monad

where *execute-trap* - ≡

```

do
  select-trap();
  err-mode ← gets (λs. (err-mode-val s));
  if err-mode = True then
    — The SparcV8 manual doesn't say what to do.
    return ()
  else
    do
      psr-val ← gets (λs. (cpu-reg-val PSR s));
      s-val ← gets (λs. ((ucast (get-S psr-val))::word1));
      curr-win ← get-curr-win();
      new-cwp ← gets (λs. ((word-of-int (((uint curr-win) - 1) mod NWIN-
DOWS))::word5));
      new-psr-val ← gets (λs. (update-PSR-exe-trap new-cwp (0::word1) s-val
psr-val));
      write-cpu new-psr-val PSR;
      exe-trap-st-pc();
      exe-trap-wr-pc();
      return ()
    od
od

```

definition *dispatch-instruction* :: instruction ⇒ ('a::len,unit) sparc-state-monad

where *dispatch-instruction instr* ≡

```

let instr-name = fst instr in
do
  traps ← gets (λs. (get-trap-set s));
  if traps = {} then
    if instr-name ∈ {load-store-type LDSB,load-store-type LDUB,
load-store-type LDUBA,load-store-type LDUH,load-store-type LD,
load-store-type LDA,load-store-type LDD} then
      load-instr instr
    else if instr-name ∈ {load-store-type STB,load-store-type STH,
load-store-type ST,load-store-type STA,load-store-type STD} then
      store-instr instr
    else if instr-name ∈ {sethi-type SETHI} then
      sethi-instr instr
    else if instr-name ∈ {nop-type NOP} then
      nop-instr instr

```

```

else if instr-name ∈ {logic-type ANDs,logic-type ANDcc,logic-type ANDN,
logic-type ANDNcc,logic-type ORs,logic-type ORcc,logic-type ORN,
logic-type XORs,logic-type XNOR} then
logical-instr instr
else if instr-name ∈ {shift-type SLL,shift-type SRL,shift-type SRA} then
shift-instr instr
else if instr-name ∈ {arith-type ADD,arith-type ADDcc,arith-type ADDX}
then
add-instr instr
else if instr-name ∈ {arith-type SUB,arith-type SUBcc,arith-type SUBX} then
sub-instr instr
else if instr-name ∈ {arith-type UMUL,arith-type SMUL,arith-type SMULcc}
then
mul-instr instr
else if instr-name ∈ {arith-type UDIV,arith-type UDIVcc,arith-type SDIV}
then
div-instr instr
else if instr-name ∈ {ctrl-type SAVE,ctrl-type RESTORE} then
save-restore-instr instr
else if instr-name ∈ {call-type CALL} then
call-instr instr
else if instr-name ∈ {ctrl-type JMPL} then
jmpl-instr instr
else if instr-name ∈ {ctrl-type RETT} then
rett-instr instr
else if instr-name ∈ {sreg-type RDY,sreg-type RDPSR,sreg-type RDWIM,
sreg-type RDTBR} then
read-state-reg-instr instr
else if instr-name ∈ {sreg-type WRYS,sreg-type WRPSR,sreg-type WRWIM,
sreg-type WRTBR} then
write-state-reg-instr instr
else if instr-name ∈ {load-store-type FLUSH} then
flush-instr instr
else if instr-name ∈ {bicc-type BE,bicc-type BNE,bicc-type BGU,
bicc-type BLE,bicc-type BL,bicc-type BGE,bicc-type BNEG,bicc-type BG,
bicc-type BCS,bicc-type BLEU,bicc-type BCC,bicc-type BA,bicc-type BN}
then
branch-instr instr
else fail ()
else return ()
od

```

definition *supported-instruction* :: *sparc-operation* ⇒ *bool*

where *supported-instruction instr* ≡

```

if instr ∈ {load-store-type LDSB,load-store-type LDUB,load-store-type LDUBA,
load-store-type LDUH,load-store-type LD,load-store-type LDA,
load-store-type LDD,
load-store-type STB,load-store-type STH,load-store-type ST,
load-store-type STA,load-store-type STD,

```

```

    sethi-type SETHI,
    nop-type NOP,
    logic-type ANDs,logic-type ANDcc,logic-type ANDN,logic-type ANDNcc,
    logic-type ORs,logic-type ORcc,logic-type ORN,logic-type XORs,
    logic-type XNOR,
    shift-type SLL,shift-type SRL,shift-type SRA,
    arith-type ADD,arith-type ADDcc,arith-type ADDX,
    arith-type SUB,arith-type SUBcc,arith-type SUBX,
    arith-type UMUL,arith-type SMUL,arith-type SMULcc,
    arith-type UDIV,arith-type UDIVcc,arith-type SDIV,
    ctrl-type SAVE,ctrl-type RESTORE,
    call-type CALL,
    ctrl-type JMPL,
    ctrl-type RETT,
    sreg-type RDY,sreg-type RDPSR,sreg-type RDWIM,sreg-type RDTBR,
    sreg-type WRY,sreg-type WRPSR,sreg-type WRWIM,sreg-type WRTBR,
    load-store-type FLUSH,
    bicc-type BE,bicc-type BNE,bicc-type BGU,bicc-type BLE,
    bicc-type BL,bicc-type BGE,bicc-type BNEG,bicc-type BG,
    bicc-type BCS,bicc-type BLEU,bicc-type BCC,bicc-type BA,
    bicc-type BN}
  then True
else False

```

definition *execute-instr-sub1* :: *instruction* \Rightarrow ('a::len,unit) *sparc-state-monad*

where *execute-instr-sub1 instr* \equiv

```

do
  instr-name  $\leftarrow$  gets ( $\lambda s$ . (fst instr));
  traps2  $\leftarrow$  gets ( $\lambda s$ . (get-trap-set s));
  if traps2 = {}  $\wedge$  instr-name  $\notin$  {call-type CALL,ctrl-type RETT,ctrl-type JMPL,
    bicc-type BE,bicc-type BNE,bicc-type BGU,
    bicc-type BLE,bicc-type BL,bicc-type BGE,
    bicc-type BNEG,bicc-type BG,
    bicc-type BCS,bicc-type BLEU,bicc-type BCC,
    bicc-type BA,bicc-type BN} then
    do
      npc-val  $\leftarrow$  gets ( $\lambda s$ . (cpu-reg-val nPC s));
      write-cpu npc-val PC;
      write-cpu (npc-val + 4) nPC;
      return ()
    od
  else return ()
od

```

definition *execute-instruction* :: *unit* \Rightarrow ('a::len,unit) *sparc-state-monad*

where *execute-instruction* - \equiv

```

do
  traps  $\leftarrow$  gets ( $\lambda s$ . (get-trap-set s));

```

```

if traps = {} then
do
  exe-mode ← gets (λs. (exe-mode-val s));
  if exe-mode = True then
  do
    modify (λs. (delayed-pool-write s));
    fetch-result ← gets (λs. (fetch-instruction s));
    case fetch-result of
    Inl e1 ⇒ (do — Memory address in PC is not aligned.
      — Actually, SparcV8 manual doesn't check alignment here.
      raise-trap instruction-access-exception;
      return ()
    od)
  | Inr v1 ⇒ (do
    dec ← gets (λs. (decode-instruction v1));
    case dec of
    Inl e2 ⇒ (— Instruction is ill-formatted.
      fail ()
    )
  | Inr v2 ⇒ (do
    instr ← gets (λs. (v2));
    annul ← gets (λs. (annul-val s));
    if annul = False then
    do
      dispatch-instruction instr;
      execute-instr-sub1 instr;
      return ()
    od
    else — annul ≠ False
    do
      set-annul False;
      npc-val ← gets (λs. (cpu-reg-val nPC s));
      write-cpu npc-val PC;
      write-cpu (npc-val + 4) nPC;
      return ()
    od
  od)
  od)
  else return () — Not in execute-mode.
od
else — traps is not empty, which means trap = 1.
do
  execute-trap();
  return ()
od
od

```

definition *NEXT* :: ('a::len)sparc-state ⇒ ('a)sparc-state option

where *NEXT* *s* \equiv *case execute-instruction* () *s* of *(-,True)* \Rightarrow *None*
| *(s',False)* \Rightarrow *Some (snd s')*

context

includes *bit-operations-syntax*

begin

definition *good-context* :: (*'a::len*) *sparc-state* \Rightarrow *bool*

where *good-context* *s* \equiv

let *traps* = *get-trap-set* *s*;
psr-val = *cpu-reg-val* *PSR* *s*;
et-val = *get-ET* *psr-val*;
rt-val = *reset-trap-val* *s*

in

if *traps* \neq {} \wedge *rt-val* = *False* \wedge *et-val* = 0 *then False* — *enter error-mode in select-traps.*

else

let *s'* = *delayed-pool-write* *s* *in*
case *fetch-instruction* *s'* *of*

— *instruction-access-exception* is handled in the next state.

Inl - \Rightarrow *True*

| *Inr* *v* \Rightarrow (

case *decode-instruction* *v* *of*

Inl - \Rightarrow *False*

| *Inr* *instr* \Rightarrow (

let *annul* = *annul-val* *s'* *in*

if *annul* = *True* *then True*

else — *annul* = *False*

if *supported-instruction* (*fst* *instr*) *then*

— The only instruction that could fail is *RETT*.

if (*fst* *instr*) = *ctrl-type* *RETT* *then*

let *curr-win-r* = (*get-CWP* (*cpu-reg-val* *PSR* *s'*));

new-cwp-int-r = (((*uint* *curr-win-r*) + 1) *mod* *NWINDOWS*);

wim-val-r = *cpu-reg-val* *WIM* *s'*;

psr-val-r = *cpu-reg-val* *PSR* *s'*;

et-val-r = *get-ET* *psr-val-r*;

s-val-r = (*ucast* (*get-S* *psr-val-r*))::*word1*;

op-list-r = *snd* *instr*;

addr-r = *get-addr* (*snd* *instr*) *s'*

in

if *et-val-r* = 1 *then True*

else if *s-val-r* = 0 *then False*

else if (*get-WIM-bit* (*nat* *new-cwp-int-r*) *wim-val-r*) \neq 0 *then False*

else if ((*AND*) *addr-r* (*0b00000000000000000000000000000011*::*word32*))

\neq 0 *then False*

else True

else True

else False — *Unsupported instruction.*

)

```

)

end

function (sequential) seq-exec:: nat ⇒ ('a::len,unit) sparc-state-monad
where seq-exec 0 = return ()
|
seq-exec n = (do execute-instruction();
              (seq-exec (n-1))
              od)

by pat-completeness auto
termination by lexicographic-order

type-synonym leon3-state = (word-length5) sparc-state

type-synonym ('e) leon3-state-monad = (leon3-state, 'e) det-monad

definition execute-leon3-instruction:: unit ⇒ (unit) leon3-state-monad
where execute-leon3-instruction ≡ execute-instruction

definition seq-exec-leon3:: nat ⇒ (unit) leon3-state-monad
where seq-exec-leon3 ≡ seq-exec

end

theory Sparc-Properties

imports Main Sparc-Execution

begin

```

24 Single step theorem

The following shows that, if the pre-state satisfies certain conditions called *good-context*, there must be a defined post-state after a single step execution.

```

method save-restore-proof =
((simp add: save-restore-instr-def),
 (simp add: Let-def simpler-gets-def bind-def h1-def h2-def),
 (simp add: case-prod-unfold),
 (simp add: raise-trap-def simpler-modify-def),
 (simp add: simpler-gets-def bind-def h1-def h2-def),
 (simp add: save-restore-sub1-def),
 (simp add: write-cpu-def simpler-modify-def),
 (simp add: write-reg-def simpler-modify-def),
 (simp add: get-curr-win-def),

```

(simp add: simplifier-def bind-def h1-def h2-def))

method *select-trap-proof0* =
((simp add: select-trap-def exec-gets return-def),
 (simp add: DetMonad.bind-def h1-def h2-def simplifier-modify-def),
 (simp add: write-cpu-tt-def write-cpu-def),
 (simp add: DetMonad.bind-def h1-def h2-def simplifier-modify-def),
 (simp add: return-def simplifier-def))

method *select-trap-proof1* =
((simp add: select-trap-def exec-gets return-def),
 (simp add: DetMonad.bind-def h1-def h2-def simplifier-modify-def),
 (simp add: write-cpu-tt-def write-cpu-def),
 (simp add: DetMonad.bind-def h1-def h2-def simplifier-modify-def),
 (simp add: return-def simplifier-def),
 (simp add: emp-trap-set-def err-mode-val-def cpu-reg-mod-def))

method *dispatch-instr-proof1* =
((simp add: dispatch-instruction-def),
 (simp add: simplifier-def bind-def h1-def h2-def),
 (simp add: Let-def))

method *exe-proof-to-decode* =
((simp add: execute-instruction-def),
 (simp add: exec-gets bind-def h1-def h2-def Let-def return-def),
 clarsimp,
 (simp add: simplifier-def bind-def h1-def h2-def Let-def simplifier-modify-def),
 (simp add: return-def))

method *exe-proof-dispatch-rett* =
((simp add: dispatch-instruction-def),
 (simp add: simplifier-def bind-def h1-def h2-def Let-def),
 (simp add: rett-instr-def),
 (simp add: simplifier-def bind-def h1-def h2-def Let-def))

lemma *write-cpu-result*: $\text{snd } (\text{write-cpu } w \ r \ s) = \text{False}$
by (simp add: write-cpu-def simplifier-modify-def)

lemma *set-annul-result*: $\text{snd } (\text{set-annul } b \ s) = \text{False}$
by (simp add: set-annul-def simplifier-modify-def)

lemma *raise-trap-result* : $\text{snd } (\text{raise-trap } t \ s) = \text{False}$
by (simp add: raise-trap-def simplifier-modify-def)

context
 includes *bit-operations-syntax*
begin

lemma *rett-instr-result*: $(\text{fst } i) = \text{ctrl-type } \text{RETT} \wedge$

```

(get-ET (cpu-reg-val PSR s) ≠ 1 ∧
(((get-S (cpu-reg-val PSR s))::word1) ≠ 0 ∧
(get-WIM-bit (nat (((uint (get-CWP (cpu-reg-val PSR s))) + 1) mod NWIN-
DOWS))
(cpu-reg-val WIM s)) = 0 ∧
((AND) (get-addr (snd i) s) (0b00000000000000000000000000000011::word32))
= 0) ⇒
snd (rett-instr i s) = False
apply (simp add: rett-instr-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def)
apply (simp add: write-cpu-def simpler-modify-def)
apply (auto simp add: Let-def return-def)
done

```

lemma *call-instr-result*: (fst i) = call-type CALL ⇒

```

snd (call-instr i s) = False
apply (simp add: call-instr-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def case-prod-unfold)
apply (simp add: write-cpu-def write-reg-def)
apply (simp add: get-curr-win-def get-CWP-def)
by (simp add: simpler-modify-def simpler-gets-def)

```

lemma *branch-instr-result*: (fst i) ∈ {bicc-type BE, bicc-type BNE, bicc-type BGU, bicc-type BLE, bicc-type BL, bicc-type BGE, bicc-type BNEG, bicc-type BG, bicc-type BCS, bicc-type BLEU, bicc-type BCC, bicc-type BA, bicc-type BN} ⇒
snd (branch-instr i s) = False

proof (cases eval-icc (fst i) (get-icc-N ((cpu-reg s) PSR)) (get-icc-Z ((cpu-reg s) PSR))

```

(get-icc-V ((cpu-reg s) PSR)) (get-icc-C ((cpu-reg s) PSR))
= 1)

```

case True

then have f1: eval-icc (fst i) (get-icc-N ((cpu-reg s) PSR)) (get-icc-Z ((cpu-reg s) PSR))

```

(get-icc-V ((cpu-reg s) PSR)) (get-icc-C ((cpu-reg s) PSR))
= 1

```

by auto

then show ?thesis

proof (cases (fst i) = bicc-type BA ∧ get-operand-flag ((snd i)!0) = 1)

case True

then show ?thesis using f1

apply (simp add: branch-instr-def)

apply (simp add: Let-def simpler-gets-def bind-def h1-def h2-def)

apply (simp add: set-annul-def case-prod-unfold)

apply (simp add: write-cpu-def simpler-modify-def)

by (simp add: return-def)

next

case False

then have f2: ¬ (fst i = bicc-type BA ∧ get-operand-flag (snd i ! 0) = 1) **by** auto

```

then show ?thesis using f1
apply (simp add: branch-instr-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def)
apply (simp add: write-cpu-def simpler-modify-def)
apply (simp add: branch-instr-sub1-def)
apply (simp add: Let-def)
apply auto
apply (simp add: write-cpu-def simpler-modify-def)
by (simp add: write-cpu-def simpler-modify-def)
qed
next
case False
then show ?thesis
apply (simp add: branch-instr-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def)
apply (simp add: write-cpu-def simpler-modify-def)
apply (simp add: branch-instr-sub1-def)
apply (simp add: Let-def)
apply auto
apply (simp add: Let-def bind-def h1-def h2-def)
apply (simp add: write-cpu-def simpler-modify-def)
apply (simp add: cpu-reg-mod-def set-annul-def simpler-modify-def)
by (simp add: write-cpu-def simpler-modify-def)
qed

lemma nop-instr-result: (fst i) = nop-type NOP  $\implies$ 
  snd (nop-instr i s) = False
apply (simp add: nop-instr-def)
by (simp add: returnOk-def return-def)

lemma sethi-instr-result: (fst i) = sethi-type SETHI  $\implies$ 
  snd (sethi-instr i s) = False
apply (simp add: sethi-instr-def)
apply (simp add: Let-def)
apply (simp add: get-curr-win-def get-CWP-def cpu-reg-val-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def)
apply (simp add: write-reg-def simpler-modify-def)
by (simp add: return-def)

lemma jmpl-instr-result: (fst i) = ctrl-type JMPL  $\implies$ 
  snd (jmpl-instr i s) = False
apply (simp add: jmpl-instr-def)
apply (simp add: get-curr-win-def get-CWP-def cpu-reg-val-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def)
apply (simp add: Let-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def)
apply (simp add: write-reg-def simpler-modify-def)
apply (simp add: write-cpu-def simpler-modify-def)
by (simp add: raise-trap-def simpler-modify-def)

```

lemma *save-restore-instr-result*: $(fst\ i) \in \{ctrl\text{-}type\ SAVE, ctrl\text{-}type\ RESTORE\}$
 \implies
 $snd\ (save\text{-}restore\text{-}instr\ i\ s) = False$
proof $(cases\ (fst\ i) = ctrl\text{-}type\ SAVE)$
 case *True*
 then show *?thesis*
 by *save-restore-proof*
next
 case *False*
 then show *?thesis*
 by *save-restore-proof*
qed

lemma *flush-instr-result*: $(fst\ i) = load\text{-}store\text{-}type\ FLUSH \implies$
 $snd\ (flush\text{-}instr\ i\ s) = False$
apply $(simp\ add: flush\text{-}instr\text{-}def)$
by $(simp\ add: simpler\text{-}gets\text{-}def\ bind\text{-}def\ h1\text{-}def\ h2\text{-}def\ simpler\text{-}modify\text{-}def)$

lemma *read-state-reg-instr-result*: $(fst\ i) \in \{sreg\text{-}type\ RDY, sreg\text{-}type\ RDPSR,$
 $sreg\text{-}type\ RDWIM, sreg\text{-}type\ RDTBR\} \implies$
 $snd\ (read\text{-}state\text{-}reg\text{-}instr\ i\ s) = False$
apply $(simp\ add: read\text{-}state\text{-}reg\text{-}instr\text{-}def)$
apply $(simp\ add: Let\text{-}def)$
apply $(simp\ add: simpler\text{-}gets\text{-}def\ bind\text{-}def\ h1\text{-}def\ h2\text{-}def\ Let\text{-}def)$
apply $(simp\ add: case\text{-}prod\text{-}unfold)$
apply $(simp\ add: simpler\text{-}gets\text{-}def\ bind\text{-}def)$
apply $(simp\ add: write\text{-}reg\text{-}def\ simpler\text{-}modify\text{-}def)$
apply $(simp\ add: raise\text{-}trap\text{-}def\ simpler\text{-}modify\text{-}def\ return\text{-}def)$
apply $(simp\ add: bind\text{-}def\ h1\text{-}def\ h2\text{-}def)$
by $(simp\ add: get\text{-}curr\text{-}win\text{-}def\ simpler\text{-}gets\text{-}def)$

lemma *write-state-reg-instr-result*: $(fst\ i) \in \{sreg\text{-}type\ WRY, sreg\text{-}type\ WRPSR,$
 $sreg\text{-}type\ WRWIM, sreg\text{-}type\ WRTBR\} \implies$
 $snd\ (write\text{-}state\text{-}reg\text{-}instr\ i\ s) = False$
apply $(simp\ add: write\text{-}state\text{-}reg\text{-}instr\text{-}def)$
apply $(simp\ add: Let\text{-}def)$
apply $(simp\ add: simpler\text{-}gets\text{-}def\ bind\text{-}def\ h1\text{-}def\ h2\text{-}def\ Let\text{-}def)$
apply $(simp\ add: case\text{-}prod\text{-}unfold)$
apply $(simp\ add: simpler\text{-}modify\text{-}def)$
apply $(simp\ add: raise\text{-}trap\text{-}def\ simpler\text{-}modify\text{-}def\ return\text{-}def)$
apply $(simp\ add: bind\text{-}def\ h1\text{-}def\ h2\text{-}def)$
apply $(simp\ add: simpler\text{-}gets\text{-}def)$
apply $(simp\ add: write\text{-}cpu\text{-}def\ simpler\text{-}modify\text{-}def)$
by $(simp\ add: get\text{-}curr\text{-}win\text{-}def\ simpler\text{-}gets\text{-}def)$

lemma *logical-instr-result*: $(fst\ i) \in \{logic\text{-}type\ ANDs, logic\text{-}type\ ANDcc,$
 $logic\text{-}type\ ANDN, logic\text{-}type\ ANDNcc, logic\text{-}type\ ORs, logic\text{-}type\ ORcc,$
 $logic\text{-}type\ ORN, logic\text{-}type\ XORs, logic\text{-}type\ XNOR\} \implies$

```

    snd (logical-instr i s) = False
apply (simp add: logical-instr-def)
apply (simp add: Let-def simpler-gets-def)
apply (simp add: write-reg-def simpler-modify-def)
apply (simp add: bind-def h1-def h2-def Let-def)
apply (simp add: case-prod-unfold)
apply (simp add: logical-instr-sub1-def)
apply (simp add: return-def)
apply (simp add: write-cpu-def simpler-modify-def)
apply (simp add: bind-def h1-def h2-def)
apply (simp add: case-prod-unfold)
apply (simp add: simpler-gets-def)
by (simp add: get-curr-win-def simpler-gets-def)

```

```

lemma shift-instr-result: (fst i) ∈ {shift-type SLL, shift-type
  SRL, shift-type SRA} ⇒
  snd (shift-instr i s) = False
apply (simp add: shift-instr-def)
apply (simp add: Let-def)
apply (simp add: get-curr-win-def simpler-gets-def bind-def h1-def h2-def)
apply (simp add: return-def)
apply (simp add: bind-def h1-def h2-def)
by (simp add: write-reg-def simpler-modify-def)

```

```

method add-sub-instr-proof =
  ((simp add: Let-def),
   auto,
   (simp add: write-reg-def simpler-modify-def),
   (simp add: simpler-gets-def bind-def),
   (simp add: get-curr-win-def simpler-gets-def),
   (simp add: write-reg-def write-cpu-def simpler-modify-def),
   (simp add: bind-def),
   (simp add: case-prod-unfold),
   (simp add: simpler-gets-def),
   (simp add: get-curr-win-def simpler-gets-def),
   (simp add: write-reg-def simpler-modify-def),
   (simp add: simpler-gets-def bind-def),
   (simp add: get-curr-win-def simpler-gets-def))

```

```

lemma add-instr-result: (fst i) ∈ {arith-type ADD, arith-type
  ADDcc, arith-type ADDX} ⇒
  snd (add-instr i s) = False
apply (simp add: add-instr-def)
apply (simp add: Let-def)
apply auto
apply (simp add: add-instr-sub1-def)
apply (simp add: write-reg-def simpler-modify-def)
apply (simp add: bind-def h1-def h2-def Let-def)
apply (simp add: case-prod-unfold)

```

```

apply (simp add: simpler-gets-def)
apply (simp add: get-curr-win-def simpler-gets-def)
apply (simp add: add-instr-sub1-def)
apply (simp add: write-reg-def simpler-modify-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def)
apply (simp add: get-curr-win-def simpler-gets-def)
apply (simp add: write-cpu-def simpler-modify-def)
apply (simp add: add-instr-sub1-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def)
apply (simp add: get-curr-win-def simpler-gets-def)
by (simp add: write-reg-def simpler-modify-def)

```

```

lemma sub-instr-result: (fst i) ∈ {arith-type SUB,arith-type SUBcc,
  arith-type SUBX} ⇒
  snd (sub-instr i s) = False
apply (simp add: sub-instr-def)
apply (simp add: Let-def)
apply auto
  apply (simp add: sub-instr-sub1-def)
  apply (simp add: write-reg-def simpler-modify-def)
  apply (simp add: bind-def h1-def h2-def Let-def)
  apply (simp add: case-prod-unfold)
  apply (simp add: simpler-gets-def)
  apply (simp add: get-curr-win-def simpler-gets-def)
apply (simp add: sub-instr-sub1-def)
apply (simp add: write-reg-def simpler-modify-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def)
apply (simp add: get-curr-win-def simpler-gets-def)
apply (simp add: write-cpu-def simpler-modify-def)
apply (simp add: sub-instr-sub1-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def)
apply (simp add: get-curr-win-def simpler-gets-def)
by (simp add: write-reg-def simpler-modify-def)

```

```

lemma mul-instr-result: (fst i) ∈ {arith-type UMUL,arith-type SMUL,
  arith-type SMULcc} ⇒
  snd (mul-instr i s) = False
apply (simp add: mul-instr-def)
apply (simp add: Let-def)
apply auto
  apply (simp add: mul-instr-sub1-def)
  apply (simp add: write-reg-def simpler-modify-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def)
  apply (simp add: get-curr-win-def simpler-gets-def)
  apply (simp add: write-reg-def write-cpu-def simpler-modify-def)
apply (simp add: mul-instr-sub1-def)
apply (simp add: simpler-gets-def)
apply (simp add: write-cpu-def write-reg-def simpler-modify-def)
apply (simp add: bind-def h1-def h2-def Let-def)

```

apply (*simp add: get-curr-win-def simpler-gets-def*)
apply (*simp add: mul-instr-sub1-def*)
apply (*simp add: simpler-gets-def*)
apply (*simp add: write-cpu-def write-reg-def simpler-modify-def*)
apply (*simp add: bind-def h1-def h2-def*)
by (*simp add: get-curr-win-def simpler-gets-def*)

lemma *div-write-new-val-result*: *snd (div-write-new-val i result temp-V s) = False*
apply (*simp add: div-write-new-val-def*)
apply (*simp add: return-def*)
apply (*simp add: simpler-gets-def bind-def h1-def h2-def*)
by (*simp add: write-cpu-def simpler-modify-def*)

lemma *div-result*: *snd (div-comp instr rs1 rd operand2 s) = False*
apply (*simp add: div-comp-def*)
apply (*simp add: simpler-gets-def*)
apply (*simp add: bind-def h1-def h2-def Let-def*)
apply (*simp add: case-prod-unfold*)
apply (*simp add: write-reg-def simpler-modify-def*)
apply (*simp add: get-curr-win-def simpler-gets-def*)
by (*simp add: div-write-new-val-result*)

lemma *div-instr-result*: *(fst i) ∈ {arith-type UDIV, arith-type UDIVcc, arith-type SDIV} ⇒*
snd (div-instr i s) = False
apply (*simp add: div-instr-def*)
apply (*simp add: Let-def*)
apply (*simp add: simpler-gets-def bind-def h1-def h2-def*)
apply (*simp add: raise-trap-def simpler-modify-def*)
apply (*simp add: return-def bind-def*)
by (*simp add: div-result*)

lemma *load-sub2-result*: *snd (load-sub2 address asi rd curr-win word0 s) = False*
apply (*simp add: load-sub2-def*)
apply (*simp add: write-reg-def simpler-modify-def*)
apply (*simp add: bind-def h1-def h2-def Let-def*)
apply (*simp add: case-prod-unfold*)
apply (*simp add: raise-trap-def simpler-modify-def*)
apply (*simp add: bind-def h1-def h2-def*)
apply (*simp add: write-reg-def simpler-modify-def*)
by (*simp add: simpler-gets-def*)

lemma *load-sub3-result*: *snd (load-sub3 instr curr-win rd asi address s) = False*
apply (*simp add: load-sub3-def*)
apply (*simp add: simpler-gets-def bind-def h1-def h2-def*)
apply (*simp add: case-prod-unfold*)
apply (*simp add: simpler-modify-def bind-def h1-def h2-def Let-def*)
apply (*simp add: write-reg-def simpler-modify-def*)
apply (*simp add: load-sub2-result*)

by (*simp add: raise-trap-def simpler-modify-def*)

lemma *load-sub1-result*: $\text{snd} (\text{load-sub1 } i \text{ rd } s\text{-val } s) = \text{False}$
apply (*simp add: load-sub1-def*)
apply (*simp add: bind-def h1-def h2-def Let-def*)
apply (*simp add: case-prod-unfold*)
apply (*simp add: raise-trap-def simpler-modify-def*)
apply (*simp add: get-curr-win-def simpler-gets-def*)
by (*simp add: load-sub3-result*)

lemma *load-instr-result*: $(\text{fst } i) \in \{\text{load-store-type LDSB}, \text{load-store-type LDUB},$
 $\text{load-store-type LDUBA}, \text{load-store-type LDUH}, \text{load-store-type LD},$
 $\text{load-store-type LDA}, \text{load-store-type LDD}\} \implies$
 $\text{snd} (\text{load-instr } i \text{ s}) = \text{False}$
apply (*simp add: load-instr-def*)
apply (*simp add: Let-def*)
apply (*simp add: simpler-gets-def bind-def h1-def h2-def*)
apply (*simp add: raise-trap-def simpler-modify-def*)
apply (*simp add: return-def*)
by (*simp add: load-sub1-result*)

lemma *store-sub2-result*: $\text{snd} (\text{store-sub2 } \text{instr } \text{curr-win } \text{rd } \text{asi } \text{address } s) = \text{False}$
apply (*simp add: store-sub2-def*)
apply (*simp add: simpler-gets-def bind-def h1-def h2-def*)
apply (*simp add: raise-trap-def simpler-modify-def*)
apply (*simp add: return-def*)
apply (*simp add: raise-trap-def simpler-modify-def*)
by (*simp add: bind-def h1-def h2-def*)

lemma *store-sub1-result*: $\text{snd} (\text{store-sub1 } \text{instr } \text{rd } s\text{-val } s) = \text{False}$
apply (*simp add: store-sub1-def*)
apply (*simp add: bind-def h1-def h2-def Let-def*)
apply (*simp add: case-prod-unfold*)
apply (*simp add: raise-trap-def simpler-modify-def*)
apply (*simp add: get-curr-win-def*)
apply (*simp add: simpler-gets-def*)
by (*simp add: store-sub2-result*)

lemma *store-instr-result*: $(\text{fst } i) \in \{\text{load-store-type STB}, \text{load-store-type STH},$
 $\text{load-store-type ST}, \text{load-store-type STA}, \text{load-store-type STD}\} \implies$
 $\text{snd} (\text{store-instr } i \text{ s}) = \text{False}$
apply (*simp add: store-instr-def*)
apply (*simp add: Let-def*)
apply (*simp add: simpler-gets-def bind-def h1-def h2-def*)
apply (*simp add: raise-trap-def simpler-modify-def*)
apply (*simp add: return-def*)
by (*simp add: store-sub1-result*)

lemma *supported-instr-set*: $\text{supported-instruction } i = \text{True} \implies$

$i \in \{$ load-store-type LDSB,load-store-type LDUB,load-store-type LDUBA,
load-store-type LDUH,load-store-type LD,load-store-type LDA,
load-store-type LDD,
load-store-type STB,load-store-type STH,load-store-type ST,
load-store-type STA,load-store-type STD,
sethi-type SETHI,
nop-type NOP,
logic-type ANDs,logic-type ANDcc,logic-type ANDN,logic-type ANDNcc,
logic-type ORs,logic-type ORcc,logic-type ORN,logic-type XORs,
logic-type XNOR,
shift-type SLL,shift-type SRL,shift-type SRA,
arith-type ADD,arith-type ADDcc,arith-type ADDX,
arith-type SUB,arith-type SUBcc,arith-type SUBX,
arith-type UMUL,arith-type SMUL,arith-type SMULcc,
arith-type UDIV,arith-type UDIVcc,arith-type SDIV,
ctrl-type SAVE,ctrl-type RESTORE,
call-type CALL,
ctrl-type JMPL,
ctrl-type RETT,
sreg-type RDY,sreg-type RDPSR,sreg-type RDWIM,sreg-type RDTBR,
sreg-type WRY,sreg-type WRPSR,sreg-type WRWIM,sreg-type WRTBR,
load-store-type FLUSH,
bicc-type BE,bicc-type BNE,bicc-type BGU,bicc-type BLE,
bicc-type BL,bicc-type BGE,bicc-type BNEG,bicc-type BG,
bicc-type BCS,bicc-type BLEU,bicc-type BCC,bicc-type BA,
bicc-type BN}

apply (*simp add: supported-instruction-def*)
by *presburger*

lemma *dispatch-instr-result*:

assumes *a1*: *supported-instruction (fst i) = True* \wedge *(fst i) \neq ctrl-type RETT*

shows *snd (dispatch-instruction i s) = False*

proof (*cases get-trap-set s = {}*)

case *True*

then have *f1: get-trap-set s = {}* **by** *auto*

then show *?thesis*

proof (*cases (fst i) \in {load-store-type LDSB,load-store-type LDUB,
load-store-type LDUBA,load-store-type LDUH,load-store-type LD,
load-store-type LDA,load-store-type LDD}*)

case *True*

then show *?thesis using f1*

apply *dispatch-instr-proof1*

by (*simp add: load-instr-result*)

next

case *False*

then have *f2: (fst i) \in {load-store-type STB,load-store-type STH,load-store-type
ST,
load-store-type STA,load-store-type STD,
sethi-type SETHI,*

```

    nop-type NOP,
    logic-type ANDs,logic-type ANDcc,logic-type ANDN,logic-type ANDNcc,
    logic-type ORs,logic-type ORcc,logic-type ORN,logic-type XORs,
    logic-type XNOR,
    shift-type SLL,shift-type SRL,shift-type SRA,
    arith-type ADD,arith-type ADDcc,arith-type ADDX,
    arith-type SUB,arith-type SUBcc,arith-type SUBX,
    arith-type UMUL,arith-type SMUL,arith-type SMULcc,
    arith-type UDIV,arith-type UDIVcc,arith-type SDIV,
    ctrl-type SAVE,ctrl-type RESTORE,
    call-type CALL,
    ctrl-type JMPL,
    sreg-type RDY,sreg-type RDPSR,sreg-type RDWIM,sreg-type RDTBR,
    sreg-type WRY,sreg-type WRPSR,sreg-type WRWIM,sreg-type WRTBR,
    load-store-type FLUSH,
    bicc-type BE,bicc-type BNE,bicc-type BGU,bicc-type BLE,
    bicc-type BL,bicc-type BGE,bicc-type BNEG,bicc-type BG,
    bicc-type BCS,bicc-type BLEU,bicc-type BCC,bicc-type BA,
    bicc-type BN}
using a1
apply (simp add: supported-instruction-def)
by presburger
then show ?thesis
proof (cases (fst i) ∈ {load-store-type STB,load-store-type STH,
    load-store-type ST,
    load-store-type STA,load-store-type STD})
case True
then show ?thesis using f1
apply dispatch-instr-proof1
by (auto simp add: store-instr-result)
next
case False
then have f3: (fst i) ∈ {sethi-type SETHI,
    nop-type NOP,
    logic-type ANDs,logic-type ANDcc,logic-type ANDN,logic-type ANDNcc,
    logic-type ORs,logic-type ORcc,logic-type ORN,logic-type XORs,
    logic-type XNOR,
    shift-type SLL,shift-type SRL,shift-type SRA,
    arith-type ADD,arith-type ADDcc,arith-type ADDX,
    arith-type SUB,arith-type SUBcc,arith-type SUBX,
    arith-type UMUL,arith-type SMUL,arith-type SMULcc,
    arith-type UDIV,arith-type UDIVcc,arith-type SDIV,
    ctrl-type SAVE,ctrl-type RESTORE,
    call-type CALL,
    ctrl-type JMPL,
    sreg-type RDY,sreg-type RDPSR,sreg-type RDWIM,sreg-type RDTBR,
    sreg-type WRY,sreg-type WRPSR,sreg-type WRWIM,sreg-type WRTBR,
    load-store-type FLUSH,
    bicc-type BE,bicc-type BNE,bicc-type BGU,bicc-type BLE,

```

```

    bicc-type BL,bicc-type BGE,bicc-type BNEG,bicc-type BG,
    bicc-type BCS,bicc-type BLEU,bicc-type BCC,bicc-type BA,
    bicc-type BN}
using f2 by auto
then show ?thesis
proof (cases (fst i) = sethi-type SETHI)
  case True
  then show ?thesis using f1
  apply dispatch-instr-proof1
  by (simp add: sethi-instr-result)
next
  case False
  then have f4: (fst i) ∈ {nop-type NOP,
    logic-type ANDs,logic-type ANDcc,logic-type ANDN,logic-type ANDNcc,
    logic-type ORs,logic-type ORcc,logic-type ORN,logic-type XORs,
    logic-type XNOR,
    shift-type SLL,shift-type SRL,shift-type SRA,
    arith-type ADD,arith-type ADDcc,arith-type ADDX,
    arith-type SUB,arith-type SUBcc,arith-type SUBX,
    arith-type UMUL,arith-type SMUL,arith-type SMULcc,
    arith-type UDIV,arith-type UDIVcc,arith-type SDIV,
    ctrl-type SAVE,ctrl-type RESTORE,
    call-type CALL,
    ctrl-type JMPL,
    sreg-type RDY,sreg-type RDPSR,sreg-type RDWIM,sreg-type RDTBR,
    sreg-type WRY,sreg-type WRPSR,sreg-type WRWIM,sreg-type WRTBR,
    load-store-type FLUSH,
    bicc-type BE,bicc-type BNE,bicc-type BGU,bicc-type BLE,
    bicc-type BL,bicc-type BGE,bicc-type BNEG,bicc-type BG,
    bicc-type BCS,bicc-type BLEU,bicc-type BCC,bicc-type BA,
    bicc-type BN}
  using f3 by auto
  then show ?thesis
  proof (cases fst i = nop-type NOP)
    case True
    then show ?thesis using f1
    apply dispatch-instr-proof1
    by (simp add: nop-instr-result)
  next
    case False
    then have f5: (fst i) ∈ {logic-type ANDs,logic-type ANDcc,
      logic-type ANDN,logic-type ANDNcc,
      logic-type ORs,logic-type ORcc,logic-type ORN,logic-type XORs,
      logic-type XNOR,
      shift-type SLL,shift-type SRL,shift-type SRA,
      arith-type ADD,arith-type ADDcc,arith-type ADDX,
      arith-type SUB,arith-type SUBcc,arith-type SUBX,
      arith-type UMUL,arith-type SMUL,arith-type SMULcc,
      arith-type UDIV,arith-type UDIVcc,arith-type SDIV,

```

```

ctrl-type SAVE,ctrl-type RESTORE,
call-type CALL,
ctrl-type JMPL,
sreg-type RDY,sreg-type RDPSR,sreg-type RDWIM,sreg-type RDTBR,
sreg-type WRY,sreg-type WRPSR,sreg-type WRWIM,sreg-type WRTBR,
load-store-type FLUSH,
bicc-type BE,bicc-type BNE,bicc-type BGU,bicc-type BLE,
bicc-type BL,bicc-type BGE,bicc-type BNEG,bicc-type BG,
bicc-type BCS,bicc-type BLEU,bicc-type BCC,bicc-type BA,
bicc-type BN}
using f4 by auto
then show ?thesis
proof (cases (fst i) ∈ {logic-type ANDs,logic-type ANDcc,
logic-type ANDN,logic-type ANDNcc,
logic-type ORs,logic-type ORcc,logic-type ORN,logic-type XORs,
logic-type XNOR})
case True
then show ?thesis using f1
apply dispatch-instr-proof1
by (auto simp add: logical-instr-result)
next
case False
then have f6: (fst i) ∈ {shift-type SLL,shift-type SRL,
shift-type SRA,
arith-type ADD,arith-type ADDcc,arith-type ADDX,
arith-type SUB,arith-type SUBcc,arith-type SUBX,
arith-type UMUL,arith-type SMUL,arith-type SMULcc,
arith-type UDIV,arith-type UDIVcc,arith-type SDIV,
ctrl-type SAVE,ctrl-type RESTORE,
call-type CALL,
ctrl-type JMPL,
sreg-type RDY,sreg-type RDPSR,sreg-type RDWIM,sreg-type RDTBR,
sreg-type WRY,sreg-type WRPSR,sreg-type WRWIM,sreg-type WRTBR,
load-store-type FLUSH,
bicc-type BE,bicc-type BNE,bicc-type BGU,bicc-type BLE,
bicc-type BL,bicc-type BGE,bicc-type BNEG,bicc-type BG,
bicc-type BCS,bicc-type BLEU,bicc-type BCC,bicc-type BA,
bicc-type BN}
using f5 by auto
then show ?thesis
proof (cases (fst i) ∈ {shift-type SLL,shift-type SRL,
shift-type SRA})
case True
then show ?thesis using f1
apply dispatch-instr-proof1
by (auto simp add: shift-instr-result)
next
case False
then have f7: (fst i) ∈ {arith-type ADD,arith-type ADDcc,

```

```

arith-type ADDX,
arith-type SUB,arith-type SUBcc,arith-type SUBX,
arith-type UMUL,arith-type SMUL,arith-type SMULcc,
arith-type UDIV,arith-type UDIVcc,arith-type SDIV,
ctrl-type SAVE,ctrl-type RESTORE,
call-type CALL,
ctrl-type JMPL,
sreg-type RDY,sreg-type RDPSR,sreg-type RDWIM,sreg-type RDTBR,
sreg-type WRY,sreg-type WRPSR,sreg-type WRWIM,sreg-type WRTBR,
load-store-type FLUSH,
bicc-type BE,bicc-type BNE,bicc-type BGU,bicc-type BLE,
bicc-type BL,bicc-type BGE,bicc-type BNEG,bicc-type BG,
bicc-type BCS,bicc-type BLEU,bicc-type BCC,bicc-type BA,
bicc-type BN}
using f6 by auto
then show ?thesis
proof (cases (fst i) ∈ {arith-type ADD,arith-type ADDcc,
arith-type ADDX})
  case True
    then show ?thesis using f1
    apply dispatch-instr-proof1
    by (auto simp add: add-instr-result)
  next
    case False
      then have f8: (fst i) ∈ {arith-type SUB,arith-type SUBcc,
arith-type SUBX,
arith-type UMUL,arith-type SMUL,arith-type SMULcc,
arith-type UDIV,arith-type UDIVcc,arith-type SDIV,
ctrl-type SAVE,ctrl-type RESTORE,
call-type CALL,
ctrl-type JMPL,
sreg-type RDY,sreg-type RDPSR,sreg-type RDWIM,sreg-type RDTBR,
sreg-type WRY,sreg-type WRPSR,sreg-type WRWIM,sreg-type WRTBR,
load-store-type FLUSH,
bicc-type BE,bicc-type BNE,bicc-type BGU,bicc-type BLE,
bicc-type BL,bicc-type BGE,bicc-type BNEG,bicc-type BG,
bicc-type BCS,bicc-type BLEU,bicc-type BCC,bicc-type BA,
bicc-type BN}
      using f7 by auto
      then show ?thesis
      proof (cases (fst i) ∈ {arith-type SUB,arith-type SUBcc,
arith-type SUBX})
        case True
          then show ?thesis using f1
          apply dispatch-instr-proof1
          by (auto simp add: sub-instr-result)
        next
          case False
            then have f9: (fst i) ∈ {arith-type UMUL,arith-type SMUL,

```

```

    arith-type SMULcc,
    arith-type UDIV,arith-type UDIVcc,arith-type SDIV,
    ctrl-type SAVE,ctrl-type RESTORE,
    call-type CALL,
    ctrl-type JMPL,
    sreg-type RDY,sreg-type RDPSR,sreg-type RDWIM,sreg-type RDTBR,
    sreg-type WRY,sreg-type WRPSR,sreg-type WRWIM,sreg-type WRTBR,
    load-store-type FLUSH,
    bicc-type BE,bicc-type BNE,bicc-type BGU,bicc-type BLE,
    bicc-type BL,bicc-type BGE,bicc-type BNEG,bicc-type BG,
    bicc-type BCS,bicc-type BLEU,bicc-type BCC,bicc-type BA,
    bicc-type BN}
  using f8 by auto
  then show ?thesis
  proof (cases (fst i) ∈ {arith-type UMUL,arith-type SMUL,
    arith-type SMULcc})
    case True
    then show ?thesis using f1
    apply dispatch-instr-proof1
    by (auto simp add: mul-instr-result)
  next
    case False
    then have f10: (fst i) ∈ {arith-type UDIV,arith-type UDIVcc,
    arith-type SDIV,
    ctrl-type SAVE,ctrl-type RESTORE,
    call-type CALL,
    ctrl-type JMPL,
    sreg-type RDY,sreg-type RDPSR,sreg-type RDWIM,sreg-type RDTBR,
    sreg-type WRY,sreg-type WRPSR,sreg-type WRWIM,sreg-type WRTBR,
    load-store-type FLUSH,
    bicc-type BE,bicc-type BNE,bicc-type BGU,bicc-type BLE,
    bicc-type BL,bicc-type BGE,bicc-type BNEG,bicc-type BG,
    bicc-type BCS,bicc-type BLEU,bicc-type BCC,bicc-type BA,
    bicc-type BN}
    using f9 by auto
    then show ?thesis
    proof (cases (fst i) ∈ {arith-type UDIV,arith-type UDIVcc,
    arith-type SDIV})
      case True
      then show ?thesis
      apply dispatch-instr-proof1 using f1
      by (auto simp add: div-instr-result)
    next
      case False
      then have f11: (fst i) ∈ {ctrl-type SAVE,ctrl-type RESTORE,
    call-type CALL,
    ctrl-type JMPL,
    sreg-type RDY,sreg-type RDPSR,sreg-type RDWIM,sreg-type RDTBR,
    sreg-type WRY,sreg-type WRPSR,sreg-type WRWIM,sreg-type WRTBR,

```

```

load-store-type FLUSH,
bicc-type BE,bicc-type BNE,bicc-type BGU,bicc-type BLE,
bicc-type BL,bicc-type BGE,bicc-type BNEG,bicc-type BG,
bicc-type BCS,bicc-type BLEU,bicc-type BCC,bicc-type BA,
bicc-type BN}
  using f10 by auto
  then show ?thesis
  proof (cases (fst i) ∈ {ctrl-type SAVE,ctrl-type RESTORE})
    case True
      then show ?thesis using f1
      apply dispatch-instr-proof1
      by (auto simp add: save-restore-instr-result)
    next
      case False
        then have f12: (fst i) ∈ {call-type CALL,
ctrl-type JMPL,
sreg-type RDY,sreg-type RDPSR,sreg-type RDWIM,sreg-type RDTBR,
sreg-type WRY,sreg-type WRPSR,sreg-type WRWIM,sreg-type WRTBR,
load-store-type FLUSH,
bicc-type BE,bicc-type BNE,bicc-type BGU,bicc-type BLE,
bicc-type BL,bicc-type BGE,bicc-type BNEG,bicc-type BG,
bicc-type BCS,bicc-type BLEU,bicc-type BCC,bicc-type BA,
bicc-type BN}
          using f11 by auto
          then show ?thesis
          proof (cases (fst i) = call-type CALL)
            case True
              then show ?thesis using f1
              apply dispatch-instr-proof1
              by (auto simp add: call-instr-result)
            next
              case False
                then have f13: (fst i) ∈ {ctrl-type JMPL,
sreg-type RDY,sreg-type RDPSR,sreg-type RDWIM,sreg-type RDTBR,
sreg-type WRY,sreg-type WRPSR,sreg-type WRWIM,sreg-type WRTBR,
load-store-type FLUSH,
bicc-type BE,bicc-type BNE,bicc-type BGU,bicc-type BLE,
bicc-type BL,bicc-type BGE,bicc-type BNEG,bicc-type BG,
bicc-type BCS,bicc-type BLEU,bicc-type BCC,bicc-type BA,
bicc-type BN}
                  using f12 by auto
                  then show ?thesis
                  proof (cases (fst i) = ctrl-type JMPL)
                    case True
                      then show ?thesis using f1
                      apply dispatch-instr-proof1
                      by (auto simp add: jmpl-instr-result)
                    next
                      case False

```

```

then have f14: (fst i) ∈ {
  sreg-type RDY,
  sreg-type RDPSR,sreg-type RDWIM,sreg-type RDTBR,
sreg-type WRY,sreg-type WRPSR,sreg-type WRWIM,sreg-type WRTBR,
load-store-type FLUSH,
bicc-type BE,bicc-type BNE,bicc-type BGU,bicc-type BLE,
bicc-type BL,bicc-type BGE,bicc-type BNEG,bicc-type BG,
bicc-type BCS,bicc-type BLEU,bicc-type BCC,bicc-type BA,
bicc-type BN}
using f13 by auto
then show ?thesis
proof (cases (fst i) ∈ {sreg-type RDY,
sreg-type RDPSR,sreg-type RDWIM,sreg-type RDTBR})
  case True
    then show ?thesis using f1
    apply dispatch-instr-proof1
    by (auto simp add: read-state-reg-instr-result)
  next
    case False
    then have f15: (fst i) ∈ {
      sreg-type WRY,
      sreg-type WRPSR,sreg-type WRWIM,sreg-type WRTBR,
load-store-type FLUSH,
bicc-type BE,bicc-type BNE,bicc-type BGU,bicc-type BLE,
bicc-type BL,bicc-type BGE,bicc-type BNEG,bicc-type BG,
bicc-type BCS,bicc-type BLEU,bicc-type BCC,bicc-type BA,
bicc-type BN}
    using f14 by auto
    then show ?thesis
    proof (cases (fst i) ∈ {sreg-type WRY,
sreg-type WRPSR,sreg-type WRWIM,sreg-type WRTBR})
      case True
        then show ?thesis using f1
        apply dispatch-instr-proof1
        by (auto simp add: write-state-reg-instr-result)
      next
        case False
        then have f16: (fst i) ∈ {
          load-store-type FLUSH,
bicc-type BE,bicc-type BNE,bicc-type BGU,bicc-type BLE,
bicc-type BL,bicc-type BGE,bicc-type BNEG,bicc-type BG,
bicc-type BCS,bicc-type BLEU,bicc-type BCC,bicc-type BA,
bicc-type BN}
        using f15 by auto
        then show ?thesis
        proof (cases (fst i) = load-store-type FLUSH)
          case True
            then show ?thesis using f1
            apply dispatch-instr-proof1

```



```

case False
then show ?thesis
apply (simp add: dispatch-instruction-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def)
apply (simp add: Let-def)
by (simp add: returnOk-def return-def)
qed

```

```

lemma dispatch-instr-result-rett:
assumes a1: (fst i) = ctrl-type RETT  $\wedge$  (get-ET (cpu-reg-val PSR s)  $\neq$  1  $\wedge$ 
  (((get-S (cpu-reg-val PSR s))::word1)  $\neq$  0  $\wedge$ 
  (get-WIM-bit (nat (((uint (get-CWP (cpu-reg-val PSR s))) + 1) mod NWINDOWS))
  (cpu-reg-val WIM s)) = 0  $\wedge$ 
  ((AND) (get-addr (snd i) s) (0b00000000000000000000000000000011::word32))
  = 0)
shows snd (dispatch-instruction i s) = False
proof (cases get-trap-set s = {})
case True
then show ?thesis using a1
apply (simp add: dispatch-instruction-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
by (simp add: rett-instr-result)
next
case False
then show ?thesis using a1
apply (simp add: dispatch-instruction-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
by (simp add: return-def)
qed

```

```

lemma execute-instr-sub1-result: snd (execute-instr-sub1 i s) = False
proof (cases get-trap-set s = {}  $\wedge$  (fst i)  $\in$  {call-type CALL,ctrl-type RETT,
  ctrl-type JMPL})
case True
then show ?thesis
apply (simp add: execute-instr-sub1-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def)
apply (simp add: write-cpu-def simpler-modify-def)
apply auto
by (auto simp add: return-def)
next
case False
then show ?thesis
apply (simp add: execute-instr-sub1-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def)
apply (simp add: write-cpu-def simpler-modify-def)
by (auto simp add: return-def)
qed

```

lemma *next-match* : $\text{snd} (\text{execute-instruction} () s) = \text{False} \implies$
 $\text{NEXT } s = \text{Some} (\text{snd} (\text{fst} (\text{execute-instruction} () s)))$
apply (*simp add: NEXT-def*)
by (*simp add: case-prod-unfold*)

lemma *exec-ss1* : $\exists s'. (\text{execute-instruction} () s = (s', \text{False})) \implies$
 $\exists s''. (\text{execute-instruction} () s = (s'', \text{False}))$
proof –
assume $\exists s'. (\text{execute-instruction} () s = (s', \text{False}))$
hence $(\text{snd} (\text{execute-instruction} () s)) = \text{False}$
by (*auto simp add: execute-instruction-def case-prod-unfold*)
hence $(\text{execute-instruction} () s) =$
 $((\text{fst} (\text{execute-instruction} () s)), \text{False})$
by (*metis (full-types) prod.collapse*)
hence $\exists s''. (\text{execute-instruction} () s = (s'', \text{False}))$
by *blast*
thus *?thesis* **by** *assumption*
qed

lemma *exec-ss2* : $\text{snd} (\text{execute-instruction} () s) = \text{False} \implies$
 $\text{snd} (\text{execute-instruction} () s) = \text{False}$
proof –
assume $\text{snd} (\text{execute-instruction} () s) = \text{False}$
hence $\text{snd} (\text{execute-instruction} () s) = \text{False}$
by (*auto simp add: execute-instruction-def*)
thus *?thesis* **by** *assumption*
qed

lemma *good-context-1* : $\text{good-context } s \wedge s' = s \wedge$
 $(\text{get-trap-set } s') \neq \{\} \wedge (\text{reset-trap-val } s') = \text{False} \wedge \text{get-ET} (\text{cpu-reg-val } \text{PSR } s')$
 $= 0$
 $\implies \text{False}$
proof –
assume *asm*: $\text{good-context } s \wedge s' = s \wedge$
 $(\text{get-trap-set } s') \neq \{\} \wedge (\text{reset-trap-val } s') = \text{False} \wedge \text{get-ET} (\text{cpu-reg-val } \text{PSR}$
 $s') = 0$
then have $(\text{get-trap-set } s') \neq \{\} \wedge (\text{reset-trap-val } s') = \text{False} \wedge$
 $\text{get-ET} (\text{cpu-reg-val } \text{PSR } s') = 0 \implies \text{False}$
by (*simp add: good-context-def get-ET-def cpu-reg-val-def*)
then show *?thesis* **using** *asm* **by** *auto*
qed

lemma *fetch-instr-result-1* : $\neg (\exists e. \text{fetch-instruction } s' = \text{Inl } e) \implies$
 $(\exists v. \text{fetch-instruction } s' = \text{Inr } v)$
by (*meson sumE*)

lemma *fetch-instr-result-2* : $(\exists v. \text{fetch-instruction } s' = \text{Inr } v) \implies$
 $\neg (\exists e. \text{fetch-instruction } s' = \text{Inl } e)$

by *force*

lemma *fetch-instr-result-3* : $(\exists e. \text{fetch-instruction } s' = \text{Inl } e) \implies$
 $\neg (\exists v. \text{fetch-instruction } s' = \text{Inr } v)$
by *auto*

lemma *decode-instr-result-1* :
 $\neg (\exists v2. ((\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inr } v2) \implies$
 $(\exists e. ((\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inl } e)$
by (*meson sumE*)

lemma *decode-instr-result-2* :
 $(\exists e. ((\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inl } e) \implies$
 $\neg (\exists v2. ((\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inr } v2)$
by *force*

lemma *decode-instr-result-3* : $x = \text{decode-instruction } v1 \wedge y = \text{decode-instruction } v2$
 $\wedge v1 = v2 \implies x = y$
by *auto*

lemma *decode-instr-result-4* :
 $\neg (\exists e. ((\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inl } e) \implies$
 $(\exists v2. ((\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inr } v2)$
by (*meson sumE*)

lemma *good-context-2* :
good-context (*s::('a::len) sparac-state*) \wedge
fetch-instruction (*delayed-pool-write s*) = *Inr v1* \wedge
 $\neg (\exists v2. (\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inr } v2)$
 $\implies \text{False}$

proof –

assume *good-context s* \wedge
fetch-instruction (*delayed-pool-write s*) = *Inr v1* \wedge
 $\neg (\exists v2. ((\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inr } v2)$
hence *fact1*: *good-context s* \wedge
fetch-instruction (*delayed-pool-write s*) = *Inr v1* \wedge
 $(\exists e. ((\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inl } e)$
using *decode-instr-result-1* **by** *auto*
hence *fact2*: $\neg (\exists e. \text{fetch-instruction } (\text{delayed-pool-write } s) = \text{Inl } e)$
using *fetch-instr-result-2* **by** *auto*
then have *fetch-instruction* (*delayed-pool-write s*) = *Inr v1* \wedge
 $(\exists e. ((\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inl } e)$
 $\implies \text{False}$

proof (*cases* (*get-trap-set s*) $\neq \{\}$) \wedge (*reset-trap-val s*) = *False* \wedge
get-ET (*cpu-reg-val PSR s*) = 0)
case *True*
from *this fact1* **show** *?thesis* **using** *good-context-1* **by** *blast*
next

```

case False
  then have fact3: (get-trap-set s) = {}  $\vee$  (reset-trap-val s)  $\neq$  False
   $\vee$  get-ET (cpu-reg-val PSR s)  $\neq$  0
  by auto
  then show ?thesis
  using fact1 decode-instr-result-3
  by (metis (no-types, lifting) good-context-def sum.case(1) sum.case(2))

qed
thus ?thesis using fact1 by auto
qed

```

```

lemma good-context-3 :
good-context (s::('a::len) sparc-state)  $\wedge$ 
s'' = delayed-pool-write s  $\wedge$ 
fetch-instruction s'' = Inr v1  $\wedge$ 
(decode-instruction v1::(Exception list + instruction)) = Inr v2  $\wedge$ 
annul-val s'' = False  $\wedge$  supported-instruction (fst v2) = False
 $\implies$  False

```

```

proof -
  assume asm: good-context (s::('a::len) sparc-state)  $\wedge$ 
  s'' = delayed-pool-write s  $\wedge$ 
  fetch-instruction s'' = Inr v1  $\wedge$ 
  (decode-instruction v1::(Exception list + instruction)) = Inr v2  $\wedge$ 
  annul-val s'' = False  $\wedge$  supported-instruction (fst v2) = False
  then have annul-val s'' = False  $\wedge$  supported-instruction (fst v2) = False
   $\implies$  False
  proof (cases (get-trap-set s)  $\neq$  {}  $\wedge$  (reset-trap-val s) = False  $\wedge$ 
  get-ET (cpu-reg-val PSR s) = 0)
  case True
  from this asm show ?thesis using good-context-1 by blast
  next
  case False
  then have fact3: (get-trap-set s) = {}  $\vee$  (reset-trap-val s)  $\neq$  False  $\vee$ 
  get-ET (cpu-reg-val PSR s)  $\neq$  0
  by auto
  thus ?thesis using asm by (auto simp add: good-context-def)
  qed
thus ?thesis using asm by auto
qed

```

```

lemma good-context-4 :
good-context (s::('a::len) sparc-state)  $\wedge$ 
s'' = delayed-pool-write s  $\wedge$ 
fetch-instruction s'' = Inr v1  $\wedge$ 
((decode-instruction v1::(Exception list + instruction)) = Inr v2  $\wedge$ 
annul-val s'' = False  $\wedge$ 
supported-instruction (fst v2) = True  $\wedge$  — This line is redundant
(fst v2) = ctrl-type RETT  $\wedge$  get-ET (cpu-reg-val PSR s'')  $\neq$  1  $\wedge$ 

```

$((\text{get-S } (\text{cpu-reg-val PSR } s''))::\text{word1}) = 0$
 $\implies \text{False}$

proof –

assume *asm*: $\text{good-context } (s::('a::\text{len}) \text{ sparc-state}) \wedge$
 $s'' = \text{delayed-pool-write } s \wedge$
 $\text{fetch-instruction } s'' = \text{Inr } v1 \wedge$
 $((\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inr } v2 \wedge$
 $\text{annul-val } s'' = \text{False} \wedge$
 $\text{supported-instruction } (\text{fst } v2) = \text{True} \wedge$ — This line is redundant
 $(\text{fst } v2) = \text{ctrl-type RETT} \wedge \text{get-ET } (\text{cpu-reg-val PSR } s'') \neq 1 \wedge$
 $((\text{get-S } (\text{cpu-reg-val PSR } s''))::\text{word1}) = 0$
then have $(\text{fst } v2) = \text{ctrl-type RETT} \wedge \text{get-ET } (\text{cpu-reg-val PSR } s'') \neq 1 \wedge$
 $((\text{get-S } (\text{cpu-reg-val PSR } s''))::\text{word1}) = 0 \implies \text{False}$
proof ($\text{cases } (\text{get-trap-set } s) \neq \{\} \wedge (\text{reset-trap-val } s) = \text{False} \wedge$
 $\text{get-ET } (\text{cpu-reg-val PSR } s) = 0$)
case *True*
from *this asm* **show** *?thesis* **using** *good-context-1* **by** *blast*
next
case *False*
then have *fact3*: $(\text{get-trap-set } s) = \{\} \vee (\text{reset-trap-val } s) \neq \text{False} \vee$
 $\text{get-ET } (\text{cpu-reg-val PSR } s) \neq 0$
by *auto*
thus *?thesis* **using** *asm* **by** (*auto simp add: good-context-def*)
qed
thus *?thesis* **using** *asm* **by** *auto*
qed

lemma *good-context-5* :

$\text{good-context } (s::('a::\text{len}) \text{ sparc-state}) \wedge$
 $s'' = \text{delayed-pool-write } s \wedge$
 $\text{fetch-instruction } s'' = \text{Inr } v1 \wedge$
 $((\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inr } v2 \wedge$
 $\text{annul-val } s'' = \text{False} \wedge$
 $\text{supported-instruction } (\text{fst } v2) = \text{True} \wedge$ — This line is redundant
 $(\text{fst } v2) = \text{ctrl-type RETT} \wedge \text{get-ET } (\text{cpu-reg-val PSR } s'') \neq 1 \wedge$
 $((\text{get-S } (\text{cpu-reg-val PSR } s''))::\text{word1}) \neq 0 \wedge$
 $(\text{get-WIM-bit } (\text{nat } (((\text{uint } (\text{get-CWP } (\text{cpu-reg-val PSR } s'')) + 1) \text{ mod } \text{NWIN-}$
 $\text{DOWS})))$
 $(\text{cpu-reg-val WIM } s'')) \neq 0$
 $\implies \text{False}$

proof –

assume *asm*: $\text{good-context } (s::('a::\text{len}) \text{ sparc-state}) \wedge$
 $s'' = \text{delayed-pool-write } s \wedge$
 $\text{fetch-instruction } s'' = \text{Inr } v1 \wedge$
 $((\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inr } v2 \wedge$
 $\text{annul-val } s'' = \text{False} \wedge$
 $\text{supported-instruction } (\text{fst } v2) = \text{True} \wedge$ — This line is redundant
 $(\text{fst } v2) = \text{ctrl-type RETT} \wedge \text{get-ET } (\text{cpu-reg-val PSR } s'') \neq 1 \wedge$
 $((\text{get-S } (\text{cpu-reg-val PSR } s''))::\text{word1}) \neq 0 \wedge$

```

  (get-WIM-bit (nat (((uint (get-CWP (cpu-reg-val PSR s''))) + 1) mod NWIN-
DOWS))
  (cpu-reg-val WIM s'')) ≠ 0
then have (fst v2) = ctrl-type RETT ∧ get-ET (cpu-reg-val PSR s') ≠ 1 ∧
  (((get-S (cpu-reg-val PSR s''))::word1) ≠ 0 ∧
  (get-WIM-bit (nat (((uint (get-CWP (cpu-reg-val PSR s''))) + 1) mod NWIN-
DOWS))
  (cpu-reg-val WIM s'')) ≠ 0
  ⇒ False
proof (cases (get-trap-set s) ≠ {} ∧ (reset-trap-val s) = False ∧
  get-ET (cpu-reg-val PSR s) = 0)
  case True
  from this asm show ?thesis using good-context-1 by blast
next
  case False
  then have fact3: (get-trap-set s) = {} ∨ (reset-trap-val s) ≠ False ∨
  get-ET (cpu-reg-val PSR s) ≠ 0
  by auto
  thus ?thesis using asm by (auto simp add: good-context-def)
qed
thus ?thesis using asm by auto
qed

```

lemma *good-context-6* :

```

good-context (s::('a::len) sparc-state) ∧
s'' = delayed-pool-write s ∧
fetch-instruction s'' = Inr v1 ∧
((decode-instruction v1)::(Exception list + instruction)) = Inr v2 ∧
annul-val s'' = False ∧
supported-instruction (fst v2) = True ∧ — This line is redundant
(fst v2) = ctrl-type RETT ∧ get-ET (cpu-reg-val PSR s') ≠ 1 ∧
(((get-S (cpu-reg-val PSR s''))::word1) ≠ 0 ∧
(get-WIM-bit (nat (((uint (get-CWP (cpu-reg-val PSR s''))) + 1) mod NWIN-
DOWS))
(cpu-reg-val WIM s'')) = 0 ∧
((AND) (get-addr (snd v2) s'') (0b000000000000000000000000000011::word32))
≠ 0
⇒ False
proof —
assume asm: good-context (s::('a::len) sparc-state) ∧
s'' = delayed-pool-write s ∧
fetch-instruction s'' = Inr v1 ∧
((decode-instruction v1)::(Exception list + instruction)) = Inr v2 ∧
annul-val s'' = False ∧
supported-instruction (fst v2) = True ∧ — This line is redundant
(fst v2) = ctrl-type RETT ∧ get-ET (cpu-reg-val PSR s') ≠ 1 ∧
(((get-S (cpu-reg-val PSR s''))::word1) ≠ 0 ∧
(get-WIM-bit (nat (((uint (get-CWP (cpu-reg-val PSR s''))) + 1) mod NWIN-
DOWS))
(cpu-reg-val WIM s'')) = 0

```

```

    (cpu-reg-val WIM s'') = 0 ∧
    ((AND) (get-addr (snd v2) s'') (0b00000000000000000000000000000011::word32))
  ≠ 0
  then have (fst v2) = ctrl-type RETT ∧ get-ET (cpu-reg-val PSR s'') ≠ 1 ∧
    (((get-S (cpu-reg-val PSR s''))::word1) ≠ 0 ∧
    (get-WIM-bit (nat (((uint (get-CWP (cpu-reg-val PSR s'')) + 1) mod NWIN-
DOWS))
    (cpu-reg-val WIM s'') = 0 ∧
    ((AND) (get-addr (snd v2) s'') (0b00000000000000000000000000000011::word32))
  ≠ 0
  ⇒ False
  proof (cases (get-trap-set s) ≠ {} ∧ (reset-trap-val s) = False ∧
    get-ET (cpu-reg-val PSR s) = 0)
    case True
      from this asm show ?thesis using good-context-1 by blast
    next
      case False
      then have fact3: (get-trap-set s) = {} ∨ (reset-trap-val s) ≠ False ∨
        get-ET (cpu-reg-val PSR s) ≠ 0
        by auto
      thus ?thesis using asm by (auto simp add: good-context-def)
    qed
  thus ?thesis using asm by auto
qed

lemma good-context-all :
good-context (s::('a::len) sparc-state)) ∧
s'' = delayed-pool-write s ⇒
(get-trap-set s = {} ∨ (reset-trap-val s) ≠ False ∨ get-ET (cpu-reg-val PSR s) ≠
0) ∧
(∃ e. fetch-instruction s'' = Inl e) ∨
(∃ v1 v2. fetch-instruction s'' = Inr v1 ∧
((decode-instruction v1)::(Exception list + instruction)) = Inr v2 ∧
(annul-val s'' = True ∨
(annul-val s'' = False ∧
(∀ v1' v2'. fetch-instruction s'' = Inr v1' ∧
((decode-instruction v1')::(Exception list + instruction)) = Inr v2' →
supported-instruction (fst v2') = True) ∧
((fst v2) ≠ ctrl-type RETT ∨
((fst v2) = ctrl-type RETT ∧
(get-ET (cpu-reg-val PSR s'') = 1 ∨
(get-ET (cpu-reg-val PSR s'') ≠ 1 ∧
(((get-S (cpu-reg-val PSR s''))::word1) ≠ 0 ∧
(get-WIM-bit (nat (((uint (get-CWP (cpu-reg-val PSR s'')) + 1) mod
NWINDOWS))
(cpu-reg-val WIM s'') = 0 ∧
((AND) (get-addr (snd v2) s'') (0b00000000000000000000000000000011::word32))
= 0)))))))))
proof -

```



```

case True
then show ?thesis using fact1 by auto
next
case False
then have fact8: ∃ v1. fetch-instruction s'' = Inr v1 ∧
(∃ v2. ((decode-instruction v1)::(Exception list + instruction)) = Inr v2)
using fact2 fact3 by auto
then show ?thesis
proof (cases annul-val s'' = True)
  case True
  then show ?thesis using fact1 fact8 by auto
next
case False
then have fact9: ∃ v1 v2. fetch-instruction s'' = Inr v1 ∧
((decode-instruction v1)::(Exception list + instruction)) = Inr v2 ∧
annul-val s'' = False ∧ supported-instruction (fst v2) = True
using fact4 fact8 by blast
then show ?thesis
proof (cases ∃ v1 v2. fetch-instruction s'' = Inr v1 ∧
((decode-instruction v1)::(Exception list + instruction)) = Inr v2 ∧
(fst v2) ≠ ctrl-type RETT)
  case True
  then show ?thesis using fact1 fact9 by auto
next
case False
then have fact10: ∃ v1 v2. fetch-instruction s'' = Inr v1 ∧
((decode-instruction v1)::(Exception list + instruction)) = Inr v2 ∧
annul-val s'' = False ∧ supported-instruction (fst v2) = True ∧
(fst v2) = ctrl-type RETT
using fact9 by auto
then show ?thesis
proof (cases get-ET (cpu-reg-val PSR s'') = 1)
  case True
  then show ?thesis using fact1 fact9 by auto
next
case False
then have fact11: get-ET (cpu-reg-val PSR s'') ≠ 1 ∧
((get-S (cpu-reg-val PSR s''))::word1) ≠ 0
using fact10 fact5 by auto
then have fact12: (get-WIM-bit (nat (((uint (get-CWP (cpu-reg-val
PSR s'')) + 1)
mod NWINDOWS)) (cpu-reg-val WIM s'')) = 0
using fact10 fact6 by auto
then have fact13: ∃ v1 v2. fetch-instruction s'' = Inr v1 ∧
((decode-instruction v1)::(Exception list + instruction)) = Inr v2 ∧
((AND) (get-addr (snd v2) s'') (0b00000000000000000000000000000011::word32))
= 0
using fact10 fact11 fact7 by blast
thus ?thesis using fact1 fact10 fact11 fact12 by auto

```

qed
 qed
 qed
 qed
 qed

lemma *select-trap-result1* : (reset-trap-val s) = True \implies
 snd (select-trap() s) = False
apply (simp add: select-trap-def exec-gets return-def)
by (simp add: bind-def h1-def h2-def simpler-modify-def)

lemma *select-trap-result2* :
assumes a1: $\neg(\text{reset-trap-val } s = \text{False} \wedge \text{get-ET } (\text{cpu-reg-val } \text{PSR } s) = 0)$
shows snd (select-trap() s) = False
proof (cases reset-trap-val s = True)
 case True
 then show ?thesis using select-trap-result1
 by blast
next
 case False
 then have f1: reset-trap-val s = False \wedge get-ET (cpu-reg-val PSR s) \neq 0
 using a1 by auto
 then show ?thesis
 proof (cases data-store-error \in get-trap-set s)
 case True
 then show ?thesis using f1
 by select-trap-proof0
next
 case False
 then have f2: data-store-error \notin get-trap-set s by auto
 then show ?thesis
 proof (cases instruction-access-error \in get-trap-set s)
 case True
 then show ?thesis using f1 f2
 by select-trap-proof0
next
 case False
 then have f3: instruction-access-error \notin get-trap-set s by auto
 then show ?thesis
 proof (cases r-register-access-error \in get-trap-set s)
 case True
 then show ?thesis using f1 f2 f3
 by select-trap-proof0
next
 case False
 then have f4: r-register-access-error \notin get-trap-set s by auto
 then show ?thesis
 proof (cases instruction-access-exception \in get-trap-set s)
 case True

```

then show ?thesis using f1 f2 f3 f4
by select-trap-proof0
next
  case False
  then have f5: instruction-access-exception  $\notin$  get-trap-set s by auto
  then show ?thesis
  proof (cases privileged-instruction  $\in$  get-trap-set s)
    case True
    then show ?thesis using f1 f2 f3 f4 f5
    by select-trap-proof0
  next
  case False
  then have f6: privileged-instruction  $\notin$  get-trap-set s by auto
  then show ?thesis
  proof (cases illegal-instruction  $\in$  get-trap-set s)
    case True
    then show ?thesis using f1 f2 f3 f4 f5 f6
    by select-trap-proof0
  next
  case False
  then have f7: illegal-instruction  $\notin$  get-trap-set s by auto
  then show ?thesis
  proof (cases fp-disabled  $\in$  get-trap-set s)
    case True
    then show ?thesis using f1 f2 f3 f4 f5 f6 f7
    by select-trap-proof0
  next
  case False
  then have f8: fp-disabled  $\notin$  get-trap-set s by auto
  then show ?thesis
  proof (cases cp-disabled  $\in$  get-trap-set s)
    case True
    then show ?thesis using f1 f2 f3 f4 f5 f6 f7 f8
    by select-trap-proof0
  next
  case False
  then have f9: cp-disabled  $\notin$  get-trap-set s by auto
  then show ?thesis
  proof (cases unimplemented-FLUSH  $\in$  get-trap-set s)
    case True
    then show ?thesis using f1 f2 f3 f4 f5 f6 f7 f8 f9
    by select-trap-proof0
  next
  case False
  then have f10: unimplemented-FLUSH  $\notin$  get-trap-set s by auto
  then show ?thesis
  proof (cases window-overflow  $\in$  get-trap-set s)
    case True
    then show ?thesis using f1 f2 f3 f4 f5 f6 f7 f8 f9 f10

```

```

    by select-trap-proof0
  next
  case False
  then have f11: window-overflow  $\notin$  get-trap-set s by auto
  then show ?thesis
  proof (cases window-underflow  $\in$  get-trap-set s)
    case True
    then show ?thesis using f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11
    by select-trap-proof0
  next
  case False
  then have f12: window-underflow  $\notin$  get-trap-set s by auto
  then show ?thesis
  proof (cases mem-address-not-aligned  $\in$  get-trap-set s)
    case True
    then show ?thesis using f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 f12
    by select-trap-proof0
  next
  case False
    then have f13: mem-address-not-aligned  $\notin$  get-trap-set s

by auto

    then show ?thesis
    proof (cases fp-exception  $\in$  get-trap-set s)
      case True
      then show ?thesis using f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11

f12 f13

      by select-trap-proof0
    next
    case False
    then have f14: fp-exception  $\notin$  get-trap-set s by auto
    then show ?thesis
    proof (cases cp-exception  $\in$  get-trap-set s)
      case True
      then show ?thesis using f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11

f12 f13

      f14
      by select-trap-proof0
    next
    case False
    then have f15: cp-exception  $\notin$  get-trap-set s by auto
    then show ?thesis
    proof (cases data-access-error  $\in$  get-trap-set s)
      case True
      then show ?thesis using f1 f2 f3 f4 f5 f6 f7 f8 f9 f10

f11 f12 f13

      f14 f15
      by select-trap-proof0
    next
    case False

```

```

    then have f16: data-access-error ∉ get-trap-set s by
auto
    then show ?thesis
    proof (cases data-access-exception ∈ get-trap-set s)
      case True
    then show ?thesis using f1 f2 f3 f4 f5 f6 f7 f8 f9 f10
f11 f12
      f13 f14 f15 f16
      by select-trap-proof0
    next
      case False
    then have f17: data-access-exception ∉ get-trap-set s
by auto
    then show ?thesis
    proof (cases tag-overflow ∈ get-trap-set s)
      case True
    then show ?thesis using f1 f2 f3 f4 f5 f6 f7 f8 f9
f10 f11 f12
      f13 f14 f15 f16 f17
      by select-trap-proof0
    next
      case False
    then have f18: tag-overflow ∉ get-trap-set s by auto
    then show ?thesis
    proof (cases division-by-zero ∈ get-trap-set s)
      case True
    then show ?thesis using f1 f2 f3 f4 f5 f6 f7 f8 f9
f10 f11
      f12 f13 f14 f15 f16 f17 f18
      by select-trap-proof0
    next
      case False
    then show ?thesis using f1 f2 f3 f4 f5 f6 f7 f8 f9
f10 f11
      f12 f13 f14 f15 f16 f17 f18
    apply (simp add: select-trap-def exec-gets return-def)
    apply (simp add: DetMonad.bind-def h1-def h2-def
simpler-modify-def)
      apply (simp add: return-def simpler-gets-def)
      apply (simp add: case-prod-unfold)
      apply (simp add: return-def)
      apply (simp add: write-cpu-tt-def write-cpu-def)
      by (simp add: simpler-gets-def bind-def h1-def
h2-def simpler-modify-def)
    qed
    qed
    qed
    qed
    qed

```



```

by select-trap-proof1
next
case False
then have f2: data-store-error  $\notin$  get-trap-set s by auto
then show ?thesis
proof (cases instruction-access-error  $\in$  get-trap-set s)
  case True
  then show ?thesis using f1 f2 a1
  by select-trap-proof1
next
case False
then have f3: instruction-access-error  $\notin$  get-trap-set s by auto
then show ?thesis
proof (cases r-register-access-error  $\in$  get-trap-set s)
  case True
  then show ?thesis using f1 f2 f3 a1
  by select-trap-proof1
next
case False
then have f4: r-register-access-error  $\notin$  get-trap-set s by auto
then show ?thesis
proof (cases instruction-access-exception  $\in$  get-trap-set s)
  case True
  then show ?thesis using f1 f2 f3 f4 a1
  by select-trap-proof1
next
case False
then have f5: instruction-access-exception  $\notin$  get-trap-set s by auto
then show ?thesis
proof (cases privileged-instruction  $\in$  get-trap-set s)
  case True
  then show ?thesis using f1 f2 f3 f4 f5 a1
  by select-trap-proof1
next
case False
then have f6: privileged-instruction  $\notin$  get-trap-set s by auto
then show ?thesis
proof (cases illegal-instruction  $\in$  get-trap-set s)
  case True
  then show ?thesis using f1 f2 f3 f4 f5 f6 a1
  by select-trap-proof1
next
case False
then have f7: illegal-instruction  $\notin$  get-trap-set s by auto
then show ?thesis
proof (cases fp-disabled  $\in$  get-trap-set s)
  case True
  then show ?thesis using f1 f2 f3 f4 f5 f6 f7 a1
  by select-trap-proof1

```

```

next
  case False
  then have f8: fp-disabled  $\notin$  get-trap-set s by auto
  then show ?thesis
  proof (cases cp-disabled  $\in$  get-trap-set s)
    case True
    then show ?thesis using f1 f2 f3 f4 f5 f6 f7 f8 a1
    by select-trap-proof1
  next
  case False
  then have f9: cp-disabled  $\notin$  get-trap-set s by auto
  then show ?thesis
  proof (cases unimplemented-FLUSH  $\in$  get-trap-set s)
    case True
    then show ?thesis using f1 f2 f3 f4 f5 f6 f7 f8 f9 a1
    by select-trap-proof1
  next
  case False
    then have f10: unimplemented-FLUSH  $\notin$  get-trap-set s by
auto

    then show ?thesis
    proof (cases window-overflow  $\in$  get-trap-set s)
      case True
      then show ?thesis using f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 a1
      by select-trap-proof1
    next
    case False
    then have f11: window-overflow  $\notin$  get-trap-set s by auto
    then show ?thesis
    proof (cases window-underflow  $\in$  get-trap-set s)
      case True
      then show ?thesis using f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 a1
      by select-trap-proof1
    next
    case False
    then have f12: window-underflow  $\notin$  get-trap-set s by auto
    then show ?thesis
    proof (cases mem-address-not-aligned  $\in$  get-trap-set s)
      case True
      then show ?thesis using f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11
f12 a1

      by select-trap-proof1
    next
    case False
    then have f13: mem-address-not-aligned  $\notin$  get-trap-set s
    by auto

    then show ?thesis
    proof (cases fp-exception  $\in$  get-trap-set s)
      case True

```

```

f12 f13
    then show ?thesis using f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11
        a1
        by select-trap-proof1
next
case False
then have f14: fp-exception  $\notin$  get-trap-set s by auto
then show ?thesis
proof (cases cp-exception  $\in$  get-trap-set s)
  case True
    then show ?thesis using f1 f2 f3 f4 f5 f6 f7 f8 f9 f10
f11 f12 f13
    f14 a1
    by select-trap-proof1
next
case False
then have f15: cp-exception  $\notin$  get-trap-set s by auto
then show ?thesis
proof (cases data-access-error  $\in$  get-trap-set s)
  case True
    then show ?thesis using f1 f2 f3 f4 f5 f6 f7 f8 f9 f10
f11 f12 f13
    f14 f15 a1
    by select-trap-proof1
next
case False
then have f16: data-access-error  $\notin$  get-trap-set s by
auto
    then show ?thesis
    proof (cases data-access-exception  $\in$  get-trap-set s)
      case True
        then show ?thesis using f1 f2 f3 f4 f5 f6 f7 f8 f9
f10 f11 f12
    f13 f14 f15 f16 a1
    by select-trap-proof1
next
case False
then have f17: data-access-exception  $\notin$  get-trap-set
s by auto
    then show ?thesis
    proof (cases tag-overflow  $\in$  get-trap-set s)
      case True
        then show ?thesis using f1 f2 f3 f4 f5 f6 f7 f8 f9
f10 f11 f12
    f13 f14 f15 f16 f17 a1
    by select-trap-proof1
next
case False
    then have f18: tag-overflow  $\notin$  get-trap-set s by

```

```

auto
    then show ?thesis
    proof (cases division-by-zero ∈ get-trap-set s)
      case True
        then show ?thesis using f1 f2 f3 f4 f5 f6 f7 f8
          f9 f10 f11
            f12 f13 f14 f15 f16 f17 f18 a1
            by select-trap-proof1
        next
          case False
            then show ?thesis using f1 f2 f3 f4 f5 f6 f7 f8
              f9 f10 f11
                f12 f13 f14 f15 f16 f17 f18 a1
                apply (simp add: select-trap-def exec-gets
                  return-def)
                  apply (simp add: bind-def h1-def h2-def
                    simpler-modify-def)
                    apply (simp add: return-def simpler-gets-def)
                    apply (simp add: emp-trap-set-def err-mode-val-def
                      cpu-reg-mod-def)
                    apply (simp add: case-prod-unfold)
                    apply (simp add: return-def)
                    apply clarsimp
                    apply (simp add: write-cpu-tt-def write-cpu-def
                      write-tt-def)
                      apply (simp add: simpler-gets-def bind-def h1-def
                        h2-def)
                        apply (simp add: simpler-modify-def)
                        by (simp add: cpu-reg-val-def cpu-reg-mod-def)
                        qed
                      qed
                    qed
                  qed
                qed
              qed
            qed
          qed
        qed
      qed
    qed
  qed

```

lemma *exe-trap-st-pc-result* : $\text{snd} (\text{exe-trap-st-pc}() s) = \text{False}$

proof (*cases annul-val s = True*)

case True

then show *?thesis*

apply (*simp add: exe-trap-st-pc-def get-curr-win-def*)

apply (*simp add: exec-gets return-def*)

apply (*simp add: DetMonad.bind-def h1-def h2-def*)

by (*simp add: set-annul-def write-reg-def simpler-modify-def*)

next

case False

then show *?thesis*

apply (*simp add: exe-trap-st-pc-def get-curr-win-def*)

apply (*simp add: exec-gets return-def*)

apply (*simp add: DetMonad.bind-def h1-def h2-def*)

by (*simp add: write-reg-def simpler-modify-def*)

qed

lemma *exe-trap-wr-pc-result* : $\text{snd} (\text{exe-trap-wr-pc}() s) = \text{False}$

proof (*cases reset-trap-val s = True*)

case True

then show *?thesis*

apply (*simp add: exe-trap-wr-pc-def get-curr-win-def*)

apply (*simp add: exec-gets return-def*)

apply (*simp add: DetMonad.bind-def h1-def h2-def*)

apply (*simp add: write-cpu-def simpler-modify-def*)

apply (*simp add: simpler-gets-def*)

apply (*simp add: cpu-reg-val-def update-S-def cpu-reg-mod-def reset-trap-val-def*)

apply (*simp add: write-cpu-def simpler-modify-def DetMonad.bind-def h1-def h2-def*)

apply (*simp add: return-def*)

by (*simp add: set-reset-trap-def simpler-modify-def DetMonad.bind-def h1-def h2-def return-def*)

next

case False

then show *?thesis*

apply (*simp add: exe-trap-wr-pc-def get-curr-win-def*)

apply (*simp add: exec-gets return-def*)

apply (*simp add: DetMonad.bind-def h1-def h2-def*)

apply (*simp add: write-cpu-def simpler-modify-def*)

apply (*simp add: simpler-gets-def*)

apply (*simp add: cpu-reg-val-def update-S-def cpu-reg-mod-def reset-trap-val-def*)

apply (*simp add: write-cpu-def simpler-modify-def DetMonad.bind-def h1-def h2-def*)

by (*simp add: return-def*)

qed

lemma *execute-trap-result* : $\neg(\text{reset-trap-val } s = \text{False} \wedge \text{get-ET} (\text{cpu-reg-val } \text{PSR } s) = 0) \implies$

```

    snd (execute-trap() s) = False
proof –
assume  $\neg(\text{reset-trap-val } s = \text{False} \wedge \text{get-ET } (\text{cpu-reg-val PSR } s) = 0)$ 
then have fact1:  $\text{snd } (\text{select-trap}() s) = \text{False}$  using select-trap-result2 by blast
then show ?thesis
  proof (cases err-mode-val s = True)
    case True
      then show ?thesis using fact1
        apply (simp add: execute-trap-def exec-gets return-def)
        apply (simp add: DetMonad.bind-def h1-def h2-def Let-def)
        apply (simp add: case-prod-unfold)
        by (simp add: in-gets return-def select-trap-monad simpler-gets-def)
      next
        case False
          then show ?thesis using fact1 select-trap-monad
            apply (simp add: execute-trap-def exec-gets return-def)
            apply (simp add: DetMonad.bind-def h1-def h2-def)
            apply (simp add: case-prod-unfold)
            apply (simp add: simpler-gets-def)
            apply (auto simp add: select-trap-monad)
            apply (simp add: DetMonad.bind-def h1-def h2-def get-curr-win-def)
            apply (simp add: get-CWP-def cpu-reg-val-def)
            apply (simp add: simpler-gets-def return-def write-cpu-def)
            apply (simp add: simpler-modify-def DetMonad.bind-def h1-def h2-def)
            apply (simp add: exe-trap-st-pc-result)
            by (simp add: case-prod-unfold exe-trap-wr-pc-result)
          qed
        qed

```

```

lemma execute-trap-result2 :  $\neg(\text{reset-trap-val } s = \text{False} \wedge \text{get-ET } (\text{cpu-reg-val PSR } s) = 0) \implies$ 
   $\text{snd } (\text{execute-trap}() s) = \text{False}$ 
using execute-trap-result
by blast

```

```

lemma exe-instr-all :
  good-context (s::('a::len) sparc-state)  $\implies$ 
     $\text{snd } (\text{execute-instruction}() s) = \text{False}$ 
proof –
assume asm1: good-context s
let ?s' = delayed-pool-write s
from asm1 have f1 :  $(\text{get-trap-set } s = \{\}) \vee (\text{reset-trap-val } s) \neq \text{False} \vee$ 
   $\text{get-ET } (\text{cpu-reg-val PSR } s) \neq 0) \wedge$ 
   $((\exists e. \text{fetch-instruction } ?s' = \text{Inl } e) \vee$ 
   $(\exists v1 v2. \text{fetch-instruction } ?s' = \text{Inr } v1 \wedge$ 
   $((\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inr } v2 \wedge$ 
   $(\text{annul-val } ?s' = \text{True} \vee$ 
   $(\text{annul-val } ?s' = \text{False} \wedge$ 
   $(\forall v1' v2'. \text{fetch-instruction } ?s' = \text{Inr } v1' \wedge$ 

```

```

    ((decode-instruction v1)::(Exception list + instruction)) = Inr v2' →
    supported-instruction (fst v2') = True) ∧
    ((fst v2) ≠ ctrl-type RETT ∨
    (fst v2) = ctrl-type RETT ∧
    (get-ET (cpu-reg-val PSR ?s') = 1 ∨
    (get-ET (cpu-reg-val PSR ?s') ≠ 1 ∧
    (((get-S (cpu-reg-val PSR ?s'))::word1) ≠ 0 ∧
    (get-WIM-bit (nat (((uint (get-CWP (cpu-reg-val PSR ?s')) + 1) mod
NWINDOWS))
    (cpu-reg-val WIM ?s')) = 0 ∧
    ((AND) (get-addr (snd v2) ?s') (0b00000000000000000000000000000011::word32))
= 0))))))
    using good-context-all by blast
from f1 have f2: get-trap-set s ≠ {} ⇒
(reset-trap-val s) ≠ False ∨ get-ET (cpu-reg-val PSR s) ≠ 0
by auto
show ?thesis
proof (cases get-trap-set s = {})
case True
then have f3: get-trap-set s = {} by auto
then show ?thesis
proof (cases exe-mode-val s = True)
case True
then have f4: exe-mode-val s = True by auto
then show ?thesis
proof (cases ∃ e1. fetch-instruction ?s' = Inl e1)
case True
then show ?thesis using f3
apply exe-proof-to-decode
apply (simp add: raise-trap-def simpler-modify-def)
by (simp add: bind-def h1-def h2-def return-def)
next
case False
then have f5: ∃ v1. fetch-instruction ?s' = Inr v1 using fetch-instr-result-1
by blast
then have f6: ∃ v1 v2. fetch-instruction ?s' = Inr v1 ∧
    ((decode-instruction v1)::(Exception list + instruction)) = Inr v2
using f1 fetch-instr-result-2 by blast
then show ?thesis
proof (cases annul-val ?s' = True)
case True
then show ?thesis using f3 f4 f6
apply exe-proof-to-decode
apply (simp add: set-annul-def annul-mod-def simpler-modify-def bind-def
h1-def h2-def)
apply (simp add: return-def simpler-gets-def)
by (simp add: write-cpu-def simpler-modify-def)
next
case False

```

```

then have f7:  $\exists v1\ v2. \text{fetch-instruction } ?s' = \text{Inr } v1 \wedge$ 
 $((\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inr } v2 \wedge$ 
 $(\forall v1'\ v2'. \text{fetch-instruction } ?s' = \text{Inr } v1' \wedge$ 
 $((\text{decode-instruction } v1')::(\text{Exception list} + \text{instruction})) = \text{Inr } v2' \longrightarrow$ 
 $\text{supported-instruction } (\text{fst } v2') = \text{True}) \wedge \text{annul-val } ?s' = \text{False}$ 
using f1 f6 fetch-instr-result-2 by auto
then have f7':  $\exists v1\ v2. \text{fetch-instruction } ?s' = \text{Inr } v1 \wedge$ 
 $((\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inr } v2 \wedge$ 
 $\text{supported-instruction } (\text{fst } v2) = \text{True} \wedge \text{annul-val } ?s' = \text{False}$ 
by auto
then show ?thesis
proof (cases  $\exists v1\ v2. \text{fetch-instruction } ?s' = \text{Inr } v1 \wedge$ 
 $((\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inr } v2 \wedge$ 
 $(\text{fst } v2) = \text{ctrl-type } \text{RETT}$ )
  case True
    then have f8:  $\exists v1\ v2. \text{fetch-instruction } ?s' = \text{Inr } v1 \wedge$ 
 $((\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inr } v2 \wedge$ 
 $(\text{fst } v2) = \text{ctrl-type } \text{RETT}$  by auto
    then show ?thesis
    proof (cases  $\text{get-trap-set } ?s' = \{\}$ )
      case True
        then have f9:  $\text{get-trap-set } ?s' = \{\}$  by auto
        then show ?thesis
        proof (cases  $\text{get-ET } (\text{cpu-reg-val } \text{PSR } ?s') = 1$ )
          case True
            then have f10:  $\text{get-ET } (\text{cpu-reg-val } \text{PSR } ?s') = 1$  by auto
            then show ?thesis
            proof (cases  $((\text{get-S } (\text{cpu-reg-val } \text{PSR } ?s'))::\text{word1}) = 0$ )
              case True
                then show ?thesis using f3 f4 f7 f8 f9 f10
                apply exe-proof-to-decode
                apply exe-proof-dispatch-rett
                apply (simp add: raise-trap-def simpler-modify-def)
                apply (auto simp add: execute-instr-sub1-result return-def)
                by (simp add: case-prod-unfold)
              next
                case False
                  then show ?thesis using f3 f4 f7 f8 f9 f10
                  apply exe-proof-to-decode
                  apply exe-proof-dispatch-rett
                  apply (simp add: raise-trap-def simpler-modify-def)
                  apply (auto simp add: execute-instr-sub1-result return-def)
                  by (simp add: case-prod-unfold)
                qed
            next
              case False
                then have f11:  $\exists v1\ v2. \text{fetch-instruction } ?s' = \text{Inr } v1 \wedge$ 
 $((\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inr } v2 \wedge$ 
 $\text{annul-val } ?s' = \text{False} \wedge$ 

```

```

      (fst v2) = ctrl-type RETT ∧
      (get-ET (cpu-reg-val PSR ?s') ≠ 1 ∧
      (((get-S (cpu-reg-val PSR ?s'))::word1) ≠ 0 ∧
      (get-WIM-bit (nat (((uint (get-CWP (cpu-reg-val PSR ?s'))) + 1)
mod NWINDOWS))
      (cpu-reg-val WIM ?s')) = 0 ∧
      ((AND) (get-addr (snd v2) ?s') (0b00000000000000000000000000000011::word32))
= 0)

      using f1 fetch-instr-result-2 f7' f8 by auto
      then show ?thesis using f3 f4
      proof (cases get-trap-set ?s' = {})
        case True
          then show ?thesis using f3 f4 f11
          apply (simp add: execute-instruction-def)
          apply (simp add: simpler-gets-def bind-def h1-def h2-def sim-
pler-modify-def)
          apply clarsimp
          apply (simp add: return-def)
          apply (simp add: bind-def h1-def h2-def Let-def)
          apply (simp add: case-prod-unfold)
          apply auto
          apply (simp add: execute-instr-sub1-result)
          apply (simp add: dispatch-instruction-def)
          apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
          by (simp add: rett-instr-result)
        next
          case False
            then show ?thesis using f3 f4 f11
            apply (simp add: execute-instruction-def)
            apply (simp add: simpler-gets-def bind-def h1-def h2-def sim-
pler-modify-def)
            apply clarsimp
            apply (simp add: return-def)
            apply (simp add: bind-def h1-def h2-def)
            apply (simp add: case-prod-unfold)
            apply (simp add: execute-instr-sub1-result)
            apply (simp add: dispatch-instruction-def)
            apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
            by (simp add: return-def)
          qed
        qed
      next
        case False
          then show ?thesis using f3 f4 f7 f8
          apply exe-proof-to-decode
          apply (simp add: dispatch-instruction-def)
          apply (simp add: simpler-gets-def bind-def h1-def h2-def)
          apply (simp add: case-prod-unfold)
          by (auto simp add: execute-instr-sub1-result return-def Let-def)

```

```

qed
next
  case False — Instruction is not RETT.
  then have  $\exists v1\ v2. \text{fetch-instruction } ?s' = \text{Inr } v1 \wedge$ 
     $((\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inr } v2 \wedge$ 
     $(\text{fst } v2) \neq \text{ctrl-type } \text{RETT}$  using f7 by auto
  then have  $\exists v1\ v2. \text{fetch-instruction } ?s' = \text{Inr } v1 \wedge$ 
     $((\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inr } v2 \wedge$ 
     $(\text{fst } v2) \neq \text{ctrl-type } \text{RETT} \wedge$ 
     $\text{supported-instruction } (\text{fst } v2) = \text{True} \wedge \text{annul-val } ?s' = \text{False}$ 
    using f7 by auto
  then have  $\exists v1\ v2. \text{fetch-instruction } ?s' = \text{Inr } v1 \wedge$ 
     $((\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inr } v2 \wedge$ 
     $(\text{fst } v2) \neq \text{ctrl-type } \text{RETT} \wedge$ 
     $\text{supported-instruction } (\text{fst } v2) = \text{True} \wedge \text{annul-val } ?s' = \text{False} \wedge$ 
     $\text{snd } (\text{dispatch-instruction } v2\ ?s') = \text{False}$ 
    by (auto simp add: dispatch-instr-result)
  then show ?thesis using f3 f4
    apply exe-proof-to-decode
    apply (simp add: bind-def h1-def h2-def)
    apply (simp add: case-prod-unfold)
    by (simp add: execute-instr-sub1-result)
  qed
qed
qed
next
  case False
  then show ?thesis using f3
  apply (simp add: execute-instruction-def)
  by (simp add: exec-gets return-def)
qed
next
  case False
  then have  $\text{get-trap-set } s \neq \{\}$   $\wedge$ 
     $((\text{reset-trap-val } s) \neq \text{False} \vee \text{get-ET } (\text{cpu-reg-val } \text{PSR } s) \neq 0)$ 
    using f2 by auto
  then show ?thesis
  apply (simp add: execute-instruction-def exec-gets)
  by (simp add: execute-trap-result2)
qed
qed

```

lemma *dispatch-fail*:

```

 $\text{snd } (\text{execute-instruction } ()) (s::('a::\text{len}) \text{sparc-state})) = \text{False} \wedge$ 
 $\text{get-trap-set } s = \{\} \wedge$ 
 $\text{exe-mode-val } s \wedge$ 
 $\text{fetch-instruction } (\text{delayed-pool-write } s) = \text{Inr } v \wedge$ 
 $((\text{decode-instruction } v)::(\text{Exception list} + \text{instruction})) = \text{Inl } e$ 
 $\implies \text{False}$ 

```

```

using decode-instr-result-2
apply (simp add: execute-instruction-def)
apply (simp add: exec-gets bind-def)
apply clarsimp
apply (simp add: simplifier-gets-def bind-def h1-def h2-def)
apply (simp add: simplifier-modify-def return-def)
by (simp add: fail-def)

```

lemma no-error : good-context s \implies snd (execute-instruction () s) = False

proof –

assume good-context s

hence snd (execute-instruction() s) = False

using exe-instr-all **by** auto

hence snd (execute-instruction () s) = False **by** (simp add: exec-ss2)

thus ?thesis **by** assumption

qed

theorem single-step : good-context s \implies NEXT s = Some (snd (fst (execute-instruction () s)))

by (simp add: no-error next-match)

25 Privilege safty

The following shows that, if the pre-state is under user mode, then after a singel step execution, the post-state is aslo under user mode.

lemma write-cpu-pc-privilege: $s' = \text{snd} (\text{fst} (\text{write-cpu } w \text{ PC } s)) \wedge$

$((\text{get-S } (\text{cpu-reg-val } \text{PSR } s))::\text{word1}) = 0 \implies$

$((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$

apply (simp add: write-cpu-def simplifier-modify-def)

apply (simp add: cpu-reg-mod-def)

by (simp add: cpu-reg-val-def)

lemma write-cpu-npc-privilege: $s' = \text{snd} (\text{fst} (\text{write-cpu } w \text{ nPC } s)) \wedge$

$((\text{get-S } (\text{cpu-reg-val } \text{PSR } s))::\text{word1}) = 0 \implies$

$((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$

apply (simp add: write-cpu-def simplifier-modify-def)

apply (simp add: cpu-reg-mod-def)

by (simp add: cpu-reg-val-def)

lemma write-cpu-y-privilege: $s' = \text{snd} (\text{fst} (\text{write-cpu } w \text{ Y } s)) \wedge$

$((\text{get-S } (\text{cpu-reg-val } \text{PSR } s))::\text{word1}) = 0$

$\implies ((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$

apply (simp add: write-cpu-def simplifier-modify-def)

apply (simp add: cpu-reg-mod-def)

by (simp add: cpu-reg-val-def)

lemma cpu-reg-mod-y-privilege: $s' = \text{cpu-reg-mod } w \text{ Y } s \wedge$

$((\text{get-S } (\text{cpu-reg-val } \text{PSR } s))::\text{word1}) = 0$

$\implies (((\text{get-}S \text{ (cpu-reg-val PSR } s^{\wedge)}))::\text{word1}) = 0$
by (*simp add: cpu-reg-mod-def cpu-reg-val-def*)

lemma *cpu-reg-mod-asr-privilege*: $s' = \text{cpu-reg-mod } w \text{ (ASR } r) s \wedge$
 $((\text{get-}S \text{ (cpu-reg-val PSR } s))::\text{word1}) = 0$
 $\implies (((\text{get-}S \text{ (cpu-reg-val PSR } s^{\wedge)}))::\text{word1}) = 0$
by (*simp add: cpu-reg-mod-def cpu-reg-val-def*)

lemma *global-reg-mod-privilege*: $s' = \text{global-reg-mod } w1 \text{ } n \text{ } w2 \text{ } s \wedge$
 $((\text{get-}S \text{ (cpu-reg-val PSR } s))::\text{word1}) = 0 \implies$
 $((\text{get-}S \text{ (cpu-reg-val PSR } s^{\wedge}))::\text{word1}) = 0$
apply (*induction n arbitrary:s*)
apply (*clarsimp*)
apply (*auto*)
apply (*simp add: Let-def*)
by (*simp add: cpu-reg-val-def*)

lemma *out-reg-mod-privilege*: $s' = \text{out-reg-mod } a \text{ } w \text{ } r \text{ } s \wedge$
 $((\text{get-}S \text{ (cpu-reg-val PSR } s))::\text{word1}) = 0 \implies$
 $((\text{get-}S \text{ (cpu-reg-val PSR } s^{\wedge}))::\text{word1}) = 0$
apply (*simp add: out-reg-mod-def Let-def*)
by (*simp add: cpu-reg-val-def*)

lemma *in-reg-mod-privilege*: $s' = \text{in-reg-mod } a \text{ } w \text{ } r \text{ } s \wedge$
 $((\text{get-}S \text{ (cpu-reg-val PSR } s))::\text{word1}) = 0 \implies$
 $((\text{get-}S \text{ (cpu-reg-val PSR } s^{\wedge}))::\text{word1}) = 0$
apply (*simp add: in-reg-mod-def Let-def*)
by (*simp add: cpu-reg-val-def*)

lemma *user-reg-mod-privilege*:
assumes *a1*: $s' = \text{user-reg-mod } d \text{ (} w::('a::\text{len}) \text{ window-size) } r$
 $(s::('a::\text{len}) \text{ sparc-state}) \wedge$
 $((\text{get-}S \text{ (cpu-reg-val PSR } s))::\text{word1}) = 0$
shows $((\text{get-}S \text{ (cpu-reg-val PSR } s^{\wedge}))::\text{word1}) = 0$
proof (*cases r = 0*)
case *True*
then show *?thesis using a1*
by (*simp add: user-reg-mod-def*)
next
case *False*
then have *f1*: $r \neq 0$ **by** *auto*
then show *?thesis*
proof (*cases 0 < r \wedge r < 8*)
case *True*
then show *?thesis using a1 f1*
apply (*simp add: user-reg-mod-def*)
by (*auto intro: global-reg-mod-privilege*)
next
case *False*

```

then have f2:  $\neg(0 < r \wedge r < 8)$  by auto
then show ?thesis
proof (cases  $7 < r \wedge r < 16$ )
  case True
    then show ?thesis using a1 f1 f2
    apply (simp add: user-reg-mod-def)
    by (auto intro: out-reg-mod-privilege)
  next
    case False
    then have f3:  $\neg(7 < r \wedge r < 16)$  by auto
    then show ?thesis
    proof (cases  $15 < r \wedge r < 24$ )
      case True
        then show ?thesis using a1 f1 f2 f3
        apply (simp add: user-reg-mod-def)
        by (simp add: cpu-reg-val-def)
      next
        case False
        then show ?thesis using a1 f1 f2 f3
        apply (simp add: user-reg-mod-def)
        by (auto intro: in-reg-mod-privilege)
    qed
  qed
qed
qed

```

lemma *write-reg-privilege*: $s' = \text{snd} (\text{fst} (\text{write-reg } w1 \ w2 \ w3$
 $(s::('a::\text{len}) \ \text{sparc-state}))) \wedge$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s))::\text{word1}) = 0 \implies$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$
apply (*simp add: write-reg-def simpler-modify-def*)
by (*auto intro: user-reg-mod-privilege*)

lemma *set-annul-privilege*: $s' = \text{snd} (\text{fst} (\text{set-annul } b \ s)) \wedge$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s))::\text{word1}) = 0 \implies$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$
apply (*simp add: set-annul-def simpler-modify-def*)
apply (*simp add: annul-mod-def write-annul-def*)
by (*simp add: cpu-reg-val-def*)

lemma *set-reset-trap-privilege*: $s' = \text{snd} (\text{fst} (\text{set-reset-trap } b \ s)) \wedge$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s))::\text{word1}) = 0 \implies$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$
apply (*simp add: set-reset-trap-def simpler-modify-def*)
apply (*simp add: reset-trap-mod-def write-annul-def*)
by (*simp add: cpu-reg-val-def*)

lemma *empty-delayed-pool-write-privilege*: $\text{get-delayed-pool } s = [] \wedge$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s))::\text{word1}) = 0 \wedge$

$s' = \text{delayed-pool-write } s \implies$
 $((\text{get-S } (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$
apply (*simp add: delayed-pool-write-def*)
by (*simp add: get-delayed-write-def delayed-write-all-def delayed-pool-rm-list-def*)

lemma *raise-trap-privilege:*
 $((\text{get-S } (\text{cpu-reg-val PSR } s))::\text{word1}) = 0 \wedge$
 $s' = \text{snd } (\text{fst } (\text{raise-trap } t \ s)) \implies$
 $((\text{get-S } (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$
apply (*simp add: raise-trap-def*)
apply (*simp add: simpler-modify-def add-trap-set-def*)
by (*simp add: cpu-reg-val-def*)

lemma *write-cpu-tt-privilege:* $s' = \text{snd } (\text{fst } (\text{write-cpu-tt } w \ s)) \wedge$
 $((\text{get-S } (\text{cpu-reg-val PSR } s))::\text{word1}) = 0$
 $\implies ((\text{get-S } (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$
apply (*simp add: write-cpu-tt-def*)
apply (*simp add: exec-gets*)
apply (*simp add: write-cpu-def cpu-reg-mod-def write-tt-def*)
apply (*simp add: simpler-modify-def*)
by (*simp add: cpu-reg-val-def*)

lemma *emp-trap-set-privilege:* $s' = \text{emp-trap-set } s \wedge$
 $((\text{get-S } (\text{cpu-reg-val PSR } s))::\text{word1}) = 0$
 $\implies ((\text{get-S } (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$
apply (*simp add: emp-trap-set-def*)
by (*simp add: cpu-reg-val-def*)

lemma *sys-reg-mod-privilege:* $s' = \text{sys-reg-mod } w \ r \ s$
 $\wedge ((\text{get-S } (\text{cpu-reg-val PSR } s))::\text{word1}) = 0$
 $\implies ((\text{get-S } (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$
apply (*simp add: sys-reg-mod-def*)
by (*simp add: cpu-reg-val-def*)

lemma *mem-mod-privilege:*
assumes $a1: s' = \text{mem-mod } a1 \ a2 \ v \ s \wedge$
 $((\text{get-S } (\text{cpu-reg-val PSR } s))::\text{word1}) = 0$
shows $((\text{get-S } (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$
proof (*cases (uint a1) = 8 \vee (uint a1) = 10*)
 case True
 then show *?thesis using a1*
 apply (*simp add: mem-mod-def*)
 apply (*simp add: Let-def*)
 by (*simp add: cpu-reg-val-def*)
next
 case False
 then have $f1: \neg((\text{uint } a1) = 8 \vee (\text{uint } a1) = 10)$ **by** *auto*
 then show *?thesis*
 proof (*cases (uint a1) = 9 \vee (uint a1) = 11*)

```

    case True
    then show ?thesis using a1 f1
    apply (simp add: mem-mod-def)
    apply (simp add: Let-def)
    by (simp add: cpu-reg-val-def)
  next
  case False
  then show ?thesis using a1 f1
  apply (simp add: mem-mod-def)
  by (simp add: cpu-reg-val-def)
qed

```

```

lemma mem-mod-w32-privilege: s' = mem-mod-w32 a1 a2 b d s  $\wedge$ 
  (((get-S (cpu-reg-val PSR s)))::word1) = 0
   $\implies$  (((get-S (cpu-reg-val PSR s'))::word1) = 0
  apply (simp add: mem-mod-w32-def)
  apply (simp add: Let-def)
  by (auto intro: mem-mod-privilege)

```

```

lemma add-instr-cache-privilege: s' = add-instr-cache s addr y m  $\implies$ 
  (((get-S (cpu-reg-val PSR s)))::word1) = 0  $\implies$ 
  (((get-S (cpu-reg-val PSR s'))::word1) = 0
  apply (simp add: add-instr-cache-def)
  apply (simp add: Let-def)
  by (simp add: icache-mod-def cpu-reg-val-def)

```

```

lemma add-data-cache-privilege: s' = add-data-cache s addr y m  $\implies$ 
  (((get-S (cpu-reg-val PSR s)))::word1) = 0  $\implies$ 
  (((get-S (cpu-reg-val PSR s'))::word1) = 0
  apply (simp add: add-data-cache-def)
  apply (simp add: Let-def)
  by (simp add: dcache-mod-def cpu-reg-val-def)

```

```

lemma memory-read-privilege:
  assumes a1: s' = snd (memory-read asi addr s)  $\wedge$ 
    (((get-S (cpu-reg-val PSR s)))::word1) = 0
  shows (((get-S (cpu-reg-val PSR s'))::word1) = 0
  proof (cases uint asi = 1)
  case True
  then show ?thesis using a1
  apply (simp add: memory-read-def)
  by (simp add: Let-def)
  next
  case False
  then have f1: uint asi  $\neq$  1 by auto
  then show ?thesis
  proof (cases uint asi = 2)
  case True

```

```

then show ?thesis using a1 f1
by (simp add: memory-read-def)
next
case False
then have f2: uint asi ≠ 2 by auto
then show ?thesis
proof (cases uint asi ∈ {8,9})
  case True
  then have f3: uint asi ∈ {8,9} by auto
  then show ?thesis
  proof (cases load-word-mem s addr asi = None)
    case True
    then have f4: load-word-mem s addr asi = None by auto
    then show ?thesis
    using a1 f1 f2 f3 f4
    by (simp add: memory-read-def)
  next
  case False
  then show ?thesis using a1 f1 f2 f3
  apply (simp add: memory-read-def)
  apply auto
  apply (simp add: add-instr-cache-privilege)
  by (simp add: add-instr-cache-privilege)
qed
next
case False
then have f5: uint asi ∉ {8, 9} by auto
then show ?thesis
proof (cases uint asi ∈ {10,11})
  case True
  then have f6: uint asi ∈ {10,11} by auto
  then show ?thesis
  proof (cases load-word-mem s addr asi = None)
    case True
    then have f7: load-word-mem s addr asi = None by auto
    then show ?thesis
    using a1 f1 f2 f5 f6 f7
    by (simp add: memory-read-def)
  next
  case False
  then show ?thesis using a1 f1 f2 f5 f6
  apply (simp add: memory-read-def)
  apply auto
  apply (simp add: add-data-cache-privilege)
  by (simp add: add-data-cache-privilege)
qed
next
case False
then have f8: uint asi ∉ {10,11} by auto

```

```

then show ?thesis
proof (cases uint asi = 13)
  case True
    then have f9: uint asi = 13 by auto
    then show ?thesis
    proof (cases read-instr-cache s addr = None)
      case True
        then show ?thesis using a1 f1 f2 f5 f8 f9
        by (simp add: memory-read-def)
      next
        case False
          then show ?thesis using a1 f1 f2 f5 f8 f9
          apply (simp add: memory-read-def)
          by auto
        qed
    next
      case False
        then have f10: uint asi ≠ 13 by auto
        then show ?thesis
        proof (cases uint asi = 15)
          case True
            then show ?thesis using a1 f1 f2 f5 f8 f10
            apply (simp add: memory-read-def)
            apply (cases read-data-cache s addr = None)
            by auto
          next
            case False
              then show ?thesis using a1 f1 f2 f5 f8 f10
              apply (simp add: memory-read-def) — The rest cases are easy.
              by (simp add: Let-def)
            qed
          qed
        qed
      qed
    qed
  qed

```

```

lemma get-curr-win-privilege: s' = snd (fst (get-curr-win() s)) ∧
  (((get-S (cpu-reg-val PSR s))::word1) = 0
  ⇒ (((get-S (cpu-reg-val PSR s'))::word1) = 0)
apply (simp add: get-curr-win-def)
by (simp add: simpler-gets-def)

```

```

lemma load-sub2-privilege:
assumes a1: s' = snd (fst (load-sub2 addr asi r win w s))
  ∧ (((get-S (cpu-reg-val PSR s))::word1) = 0
  shows (((get-S (cpu-reg-val PSR s'))::word1) = 0)
proof (cases fst (memory-read asi (addr + 4)
  (snd (fst (write-reg w win (r AND 30) s)))) =

```

```

      None)
    case True
    then show ?thesis using a1
    apply (simp add: load-sub2-def)
    apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
    apply (simp add: case-prod-unfold)
    by (auto intro: raise-trap-privilege write-reg-privilege)
  next
  case False
  then show ?thesis using a1
  apply (simp add: load-sub2-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def)
  apply (simp add: case-prod-unfold)
  apply clarsimp
  apply (simp add: simpler-modify-def bind-def h1-def h2-def Let-def)
  by (auto intro: write-reg-privilege memory-read-privilege)
qed

lemma load-sub3-privilege:
  assumes a1: s' = snd (fst (load-sub3 instr curr-win rd asi address s))
    ∧ (((get-S (cpu-reg-val PSR s))::word1) = 0
  shows (((get-S (cpu-reg-val PSR s'))::word1) = 0
  proof (cases fst (memory-read asi address s) = None)
  case True
  then show ?thesis using a1
  apply (simp add: load-sub3-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def)
  apply (simp add: case-prod-unfold)
  by (auto intro: raise-trap-privilege)
  next
  case False
  then have f1: fst (memory-read asi address s) ≠ None by auto
  then show ?thesis
  proof (cases rd ≠ 0 ∧
    (fst instr = load-store-type LD ∨
    fst instr = load-store-type LDA ∨
    fst instr = load-store-type LDUH ∨
    fst instr = load-store-type LDSB ∨
    fst instr = load-store-type LDUB ∨
    fst instr = load-store-type LDUBA ∨
    fst instr = load-store-type LDSH ∨
    fst instr = load-store-type LDSHA ∨
    fst instr = load-store-type LDUHA ∨
    fst instr = load-store-type LDSBA))
  case True
  then show ?thesis using a1 f1
  apply (simp add: load-sub3-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def)
  apply (simp add: case-prod-unfold)

```

```

apply clarsimp
apply (simp add: simpler-modify-def bind-def h1-def h2-def Let-def)
by (auto intro: write-reg-privilege memory-read-privilege)
next
  case False
  then show ?thesis using a1 f1
  apply (simp add: load-sub3-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def)
  apply (simp add: case-prod-unfold)
  apply auto
  apply (simp add: simpler-modify-def bind-def h1-def h2-def)
  apply (auto intro: load-sub2-privilege memory-read-privilege)
  apply (simp add: simpler-modify-def bind-def h1-def h2-def)
  by (auto intro: load-sub2-privilege memory-read-privilege)
qed
qed

lemma load-sub1-privilege:
assumes a1: s' = snd (fst (load-sub1 instr rd s-val s))
   $\wedge (((\text{get-S } (\text{cpu-reg-val PSR } s)))::\text{word1}) = 0$ 
shows  $((\text{get-S } (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$ 
using a1
apply (simp add: load-sub1-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
apply (simp add: case-prod-unfold)
apply auto
  by (auto intro: get-curr-win-privilege raise-trap-privilege load-sub3-privilege)

lemma load-instr-privilege: s' = snd (fst (load-instr i s))
   $\wedge (((\text{get-S } (\text{cpu-reg-val PSR } s)))::\text{word1}) = 0$ 
   $\implies (((\text{get-S } (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$ 
apply (simp add: load-instr-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def)
apply (simp add: Let-def)
apply clarsimp
by (auto intro: get-curr-win-privilege raise-trap-privilege load-sub1-privilege)

lemma store-barrier-pending-mod-privilege: s' = store-barrier-pending-mod b s
   $\wedge (((\text{get-S } (\text{cpu-reg-val PSR } s)))::\text{word1}) = 0$ 
   $\implies (((\text{get-S } (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$ 
apply (simp add: store-barrier-pending-mod-def)
apply (simp add: write-store-barrier-pending-def)
by (simp add: cpu-reg-val-def)

lemma store-word-mem-privilege:
assumes a1: store-word-mem s addr data byte-mask asi = Some s'  $\wedge$ 
   $((\text{get-S } (\text{cpu-reg-val PSR } s)))::\text{word1}) = 0$ 
shows  $((\text{get-S } (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$ 
using a1 apply (simp add: store-word-mem-def)

```

```

apply (case-tac virt-to-phys addr (mmu s) (mem s) = None)
apply auto
apply (case-tac mmu-writable (get-acc-flag b) asi)
apply auto
by (simp add: mem-mod-w32-privilege)

```

```

lemma flush-instr-cache-privilege: (((get-S (cpu-reg-val PSR s))):word1) = 0  $\implies$ 
s' = flush-instr-cache s  $\implies$ 
(((get-S (cpu-reg-val PSR s'))):word1) = 0
apply (simp add: flush-instr-cache-def)
by (simp add: cpu-reg-val-def)

```

```

lemma flush-data-cache-privilege: (((get-S (cpu-reg-val PSR s))):word1) = 0  $\implies$ 
s' = flush-data-cache s  $\implies$ 
(((get-S (cpu-reg-val PSR s'))):word1) = 0
apply (simp add: flush-data-cache-def)
by (simp add: cpu-reg-val-def)

```

```

lemma flush-cache-all-privilege: (((get-S (cpu-reg-val PSR s))):word1) = 0  $\implies$ 
s' = flush-cache-all s  $\implies$ 
(((get-S (cpu-reg-val PSR s'))):word1) = 0
apply (simp add: flush-cache-all-def)
by (simp add: cpu-reg-val-def)

```

```

lemma memory-write-asi-privilege:
assumes a1: r = memory-write-asi asi addr byte-mask data s  $\wedge$ 
r = Some s'  $\wedge$ 
(((get-S (cpu-reg-val PSR s))):word1) = 0
shows (((get-S (cpu-reg-val PSR s'))):word1) = 0
proof (cases uint asi = 1)
  case True
  then show ?thesis using a1
  apply (simp add: memory-write-asi-def)
  by (auto intro: store-word-mem-privilege)
next
  case False
  then have f1: uint asi  $\neq$  1 by auto
  then show ?thesis
  proof (cases uint asi = 2)
    case True
    then have f01: uint asi = 2 by auto
    then show ?thesis
    proof (cases uint addr = 0)
      case True
      then show ?thesis using a1 f1 f01
      apply (simp add: memory-write-asi-def)
      apply (simp add: ccr-flush-def)
      apply (simp add: Let-def)
      apply auto
    
```

```

apply (metis flush-data-cache-privilege flush-instr-cache-privilege sys-reg-mod-privilege)
  apply (metis flush-instr-cache-privilege sys-reg-mod-privilege)
  apply (metis flush-data-cache-privilege sys-reg-mod-privilege)
by (simp add: sys-reg-mod-privilege)
next
  case False
  then show ?thesis using a1 f1 f01
  apply (simp add: memory-write-asi-def)
  apply clarsimp
  by (metis option.distinct(1) option.sel sys-reg-mod-privilege)
qed
next
  case False
  then have f2: uint asi ≠ 2 by auto
  then show ?thesis
  proof (cases uint asi ∈ {8,9})
    case True
    then show ?thesis using a1 f1 f2
    apply (simp add: memory-write-asi-def)
    using store-word-mem-privilege add-instr-cache-privilege
    by blast
  next
  case False
  then have f3: uint asi ∉ {8,9} by auto
  then show ?thesis
  proof (cases uint asi ∈ {10,11})
    case True
    then show ?thesis using a1 f1 f2 f3
    apply (simp add: memory-write-asi-def)
    using store-word-mem-privilege add-data-cache-privilege
    by blast
  next
  case False
  then have f4: uint asi ∉ {10,11} by auto
  then show ?thesis
  proof (cases uint asi = 13)
    case True
    then show ?thesis using a1 f1 f2 f3 f4
    apply (simp add: memory-write-asi-def)
    by (auto simp add: add-instr-cache-privilege)
  next
  case False
  then have f5: uint asi ≠ 13 by auto
  then show ?thesis
  proof (cases uint asi = 15)
    case True
    then show ?thesis using a1 f1 f2 f3 f4 f5
    apply (simp add: memory-write-asi-def)
    by (auto simp add: add-data-cache-privilege)

```

```

next
  case False
  then have f6: uint asi  $\neq$  15 by auto
  then show ?thesis
  proof (cases uint asi = 16)
    case True
    then show ?thesis using a1
    apply (simp add: memory-write-asi-def)
    by (auto simp add: flush-instr-cache-privilege)
  next
  case False
  then have f7: uint asi  $\neq$  16 by auto
  then show ?thesis
  proof (cases uint asi = 17)
    case True
    then show ?thesis using a1
    apply (simp add: memory-write-asi-def)
    by (auto simp add: flush-data-cache-privilege)
  next
  case False
  then have f8: uint asi  $\neq$  17 by auto
  then show ?thesis
  proof (cases uint asi = 24)
    case True
    then show ?thesis using a1
    apply (simp add: memory-write-asi-def)
    by (auto simp add: flush-cache-all-privilege)
  next
  case False
  then have f9: uint asi  $\neq$  24 by auto
  then show ?thesis
  proof (cases uint asi = 25)
    case True
    then show ?thesis using a1
    apply (simp add: memory-write-asi-def)
    apply (case-tac mmu-reg-mod (mmu s) addr data = None)
    apply auto
    by (simp add: cpu-reg-val-def)
  next
  case False
  then have f10: uint asi  $\neq$  25 by auto
  then show ?thesis
  proof (cases uint asi = 28)
    case True
    then show ?thesis using a1
    apply (simp add: memory-write-asi-def)
    by (auto simp add: mem-mod-w32-privilege)
  next
  case False — The remaining cases are easy.

```

```

      then show ?thesis using a1 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10
      apply (simp add: memory-write-asi-def)
      apply (auto simp add: Let-def)
      apply (case-tac uint asi = 20 ∨ uint asi = 21)
      by auto
    qed
  qed
qed
qed
qed
qed
qed
qed
qed
qed
qed
qed

```

lemma *memory-write-privilege*:

assumes *a1*: $r = \text{memory-write asi addr byte-mask data}$

$(s::('a::len) \text{sparc-state})) \wedge$

$r = \text{Some } s' \wedge$

$((\text{get-S (cpu-reg-val PSR s)}))::\text{word1}) = 0$

shows $((\text{get-S (cpu-reg-val PSR}$

$(s'::('a::len) \text{sparc-state})))::\text{word1}) = 0$

proof –

have $\forall x. \text{Some } x \neq \text{None}$ **by** *auto*

then have $r \neq \text{None}$ **using** *a1*

by (*simp add: $\langle r = \text{memory-write asi addr byte-mask data } s \wedge$*
 $r = \text{Some } s' \wedge (\text{get-S (cpu-reg-val PSR } s)) = 0 \rangle$)

then have $\exists s''. r = \text{Some (store-barrier-pending-mod False } s'')$ **using** *a1*

by (*metis (no-types, lifting) memory-write-def option.case-eq-if*)

then have $\exists s''. s' = \text{store-barrier-pending-mod False } s''$ **using** *a1*

by *blast*

then have $\exists s''. \text{memory-write-asi asi addr byte-mask data } s = \text{Some } s'' \wedge$
 $s' = \text{store-barrier-pending-mod False } s''$

by (*metis (no-types, lifting) assms memory-write-def not-None-eq option.case-eq-if*
option.sel)

then show ?thesis **using** *a1*

using *memory-write-asi-privilege store-barrier-pending-mod-privilege* **by** *blast*

qed

lemma *store-sub2-privilege*:

assumes *a1*: $s' = \text{snd (fst (store-sub2 instr curr-win rd asi address } s))}$

$\wedge ((\text{get-S (cpu-reg-val PSR } s))::\text{word1}) = 0$

shows $((\text{get-S (cpu-reg-val PSR } s'))::\text{word1}) = 0$

proof (*cases memory-write asi address (st-byte-mask instr address)*

$(\text{st-data0 instr curr-win rd address } s) s =$
 None)

```

case True
then show ?thesis using a1
apply (simp add: store-sub2-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
apply (simp add: case-prod-unfold)
by (metis fst-conv raise-trap-privilege return-def snd-conv)
next
case False
then have f1: ¬(memory-write asi address (st-byte-mask instr address)
              (st-data0 instr curr-win rd address s) s =
              None)
  by auto
then show ?thesis
proof (cases (fst instr) ∈ {load-store-type STD,load-store-type STDA})
  case True
  then have f2: (fst instr) ∈ {load-store-type STD,load-store-type STDA} by
  auto
  then show ?thesis using a1 f1
  apply (simp add: store-sub2-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def)
  apply (simp add: simpler-modify-def bind-def h1-def h2-def Let-def)
  apply (simp add: return-def)
  apply (simp add: bind-def case-prod-unfold)
  apply (simp add: simpler-modify-def)
  apply clarsimp
  apply (simp add: case-prod-unfold bind-def h1-def h2-def Let-def simpler-modify-def)
  apply (simp add: simpler-gets-def)
  apply auto
  using memory-write-privilege raise-trap-privilege apply blast
  apply (simp add: simpler-modify-def simpler-gets-def bind-def)
  apply (meson memory-write-privilege)
  using memory-write-privilege raise-trap-privilege apply blast
  by (meson memory-write-privilege)
next
case False
then show ?thesis using a1 f1
apply (simp add: store-sub2-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
apply clarsimp
apply (simp add: simpler-modify-def return-def)
by (auto intro: memory-write-privilege)
qed
qed

```

lemma *store-sub1-privilege*:
assumes $a1$: $s' = \text{snd} (\text{fst} (\text{store-sub1 } \text{instr } \text{rd } \text{s-val}$
 $(s::('a::\text{len}) \text{ sparc-state})))$
 $\wedge (((\text{get-S } (\text{cpu-reg-val } \text{PSR } s)))::\text{word1}) = 0$
shows $((\text{get-S } (\text{cpu-reg-val } \text{PSR}$

```

(s'::('a::len) sparc-state))))::word1) = 0
proof (cases (fst instr = load-store-type STH  $\vee$  fst instr = load-store-type STHA)
 $\wedge$ 
      ((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s))))))::word1)  $\neq$ 
0)
  case True
  then show ?thesis using a1
  apply (simp add: store-sub1-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
  apply (simp add: case-prod-unfold)
  using get-curr-win-privilege raise-trap-privilege by blast
next
  case False
  then have f1:  $\neg$ ((fst instr = load-store-type STH  $\vee$  fst instr = load-store-type
STHA)  $\wedge$ 
      ((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s))))))::word1)  $\neq$ 
0)
    by auto
  then show ?thesis
  proof (cases (fst instr  $\in$  {load-store-type ST,load-store-type STA})  $\wedge$ 
      ((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s))))))::word2)  $\neq$ 
0)
    case True
    then show ?thesis using a1 f1
    apply (simp add: store-sub1-def)
    apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
    apply (simp add: case-prod-unfold)
    using get-curr-win-privilege raise-trap-privilege by blast
    next
    case False
    then have f2:  $\neg$ ((fst instr  $\in$  {load-store-type ST,load-store-type STA})  $\wedge$ 
      ((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s))))))::word2)  $\neq$ 
0)
      by auto
    then show ?thesis
    proof (cases (fst instr  $\in$  {load-store-type STD,load-store-type STDA})  $\wedge$ 
      ((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s))))))::word3)  $\neq$ 
0)
      case True
      then show ?thesis using a1 f1 f2
      apply (simp add: store-sub1-def)
      apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
      apply (simp add: case-prod-unfold)
      using get-curr-win-privilege raise-trap-privilege by blast
      next
      case False
      then show ?thesis using a1 f1 f2
      apply (simp add: store-sub1-def)
      apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)

```



```

apply (simp add: Let-def)
apply auto
apply (simp add: cpu-reg-val-def)
using s-0-word by blast

lemma logical-instr-sub1-privilege:
assumes a1: s' = snd (fst (logical-instr-sub1 instr-name result
  (s::('a::len) sparc-state))))
  ∧ (((get-S (cpu-reg-val PSR s))::word1) = 0
shows (((get-S (cpu-reg-val PSR s'))::word1) = 0
proof (cases instr-name = logic-type ANDcc ∨
  instr-name = logic-type ANDNcc ∨
  instr-name = logic-type ORcc ∨
  instr-name = logic-type ORNcc ∨
  instr-name = logic-type XORcc ∨ instr-name = logic-type XNORcc)
  case True
  then show ?thesis using a1
  apply (simp add: logical-instr-sub1-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
  apply (simp add: logical-new-psr-val-def)
  using write-cpu-PSR-icc-privilege by blast
next
  case False
  then show ?thesis using a1
  apply (simp add: logical-instr-sub1-def)
  by (simp add: return-def)
qed

lemma logical-instr-privilege:
assumes a1: s' = snd (fst (logical-instr instr
  (s::('a::len) sparc-state))))
  ∧ (((get-S (cpu-reg-val PSR s))::word1) = 0
shows (((get-S (cpu-reg-val PSR s'))::word1) = 0
using a1
apply (simp add: logical-instr-def)
apply (simp add: Let-def simpler-gets-def bind-def h1-def h2-def)
apply (simp add: case-prod-unfold)
apply auto
apply (meson get-curr-win-privilege logical-instr-sub1-privilege write-reg-privilege)
by (meson get-curr-win-privilege logical-instr-sub1-privilege write-reg-privilege)

method shift-instr-privilege-proof = (
  (simp add: shift-instr-def),
  (simp add: Let-def),
  (simp add: simpler-gets-def),
  (simp add: bind-def h1-def h2-def Let-def case-prod-unfold),
  auto,
  (blast intro: get-curr-win-privilege write-reg-privilege),
  (blast intro: get-curr-win-privilege write-reg-privilege)

```

)

lemma *shift-instr-privilege*:

assumes $a1: s' = \text{snd } (\text{fst } (\text{shift-instr } \text{instr}$
 $(s::('a::\text{len}) \text{sparc-state})))$

$\wedge (((\text{get-S } (\text{cpu-reg-val } \text{PSR } s)))::\text{word1}) = 0$

shows $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$

proof $(\text{cases } (\text{fst } \text{instr} = \text{shift-type } \text{SLL}) \wedge (\text{get-operand-w5 } ((\text{snd } \text{instr})!3) \neq 0))$

case *True*

then show *?thesis* **using** $a1$

by *shift-instr-privilege-proof*

next

case *False*

then have $f1: \neg((\text{fst } \text{instr} = \text{shift-type } \text{SLL}) \wedge (\text{get-operand-w5 } ((\text{snd } \text{instr})!3) \neq 0))$

by *auto*

then show *?thesis*

proof $(\text{cases } (\text{fst } \text{instr} = \text{shift-type } \text{SRL}) \wedge (\text{get-operand-w5 } ((\text{snd } \text{instr})!3) \neq 0))$

case *True*

then show *?thesis* **using** $a1 f1$

by *shift-instr-privilege-proof*

next

case *False*

then have $f2: \neg((\text{fst } \text{instr} = \text{shift-type } \text{SRL}) \wedge (\text{get-operand-w5 } ((\text{snd } \text{instr})!3) \neq 0))$

by *auto*

then show *?thesis*

proof $(\text{cases } (\text{fst } \text{instr} = \text{shift-type } \text{SRA}) \wedge (\text{get-operand-w5 } ((\text{snd } \text{instr})!3) \neq 0))$

case *True*

then show *?thesis* **using** $a1 f1 f2$

by *shift-instr-privilege-proof*

next

case *False*

then show *?thesis* **using** $a1 f1 f2$

apply $(\text{simp add: } \text{shift-instr-def})$

apply $(\text{simp add: } \text{Let-def})$

apply $(\text{simp add: } \text{simpler-gets-def})$

apply $(\text{simp add: } \text{bind-def } \text{h1-def } \text{h2-def } \text{Let-def } \text{case-prod-unfold})$

apply $(\text{simp add: } \text{return-def})$

using *get-curr-win-privilege* **by** *blast*

qed

qed

qed

lemma *add-instr-sub1-privilege*:

assumes $a1: s' = \text{snd } (\text{fst } (\text{add-instr-sub1 } \text{instr-name } \text{result } \text{rs1-val } \text{operand2}$
 $(s::('a::\text{len}) \text{sparc-state})))$

```

   $\wedge (((get-S (cpu-reg-val PSR s))::word1) = 0$ 
shows  $((((get-S (cpu-reg-val PSR s'))::word1) = 0$ 
proof (cases instr-name = arith-type ADDcc  $\vee$  instr-name = arith-type ADDXcc)
  case True
  then show ?thesis using a1
  apply (simp add: add-instr-sub1-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
  by (blast intro: write-cpu-PSR-icc-privilege)
next
  case False
  then show ?thesis using a1
  apply (simp add: add-instr-sub1-def)
  by (simp add: return-def)
qed

```

```

lemma add-instr-privilege:
assumes a1:  $s' = snd (fst (add-instr instr$ 
   $(s::('a::len) sparc-state)))$ 
   $\wedge (((get-S (cpu-reg-val PSR s))::word1) = 0$ 
shows  $((((get-S (cpu-reg-val PSR s'))::word1) = 0$ 
using a1
apply (simp add: add-instr-def)
apply (simp add: Let-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
apply (simp add: case-prod-unfold)
by (meson add-instr-sub1-privilege get-curr-win-privilege write-reg-privilege)

```

```

lemma sub-instr-sub1-privilege:
assumes a1:  $s' = snd (fst (sub-instr-sub1 instr-name result rs1-val operand2$ 
   $(s::('a::len) sparc-state)))$ 
   $\wedge (((get-S (cpu-reg-val PSR s))::word1) = 0$ 
shows  $((((get-S (cpu-reg-val PSR s'))::word1) = 0$ 
proof (cases instr-name = arith-type SUBcc  $\vee$  instr-name = arith-type SUBXcc)
  case True
  then show ?thesis using a1
  apply (simp add: sub-instr-sub1-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
  by (blast intro: write-cpu-PSR-icc-privilege)
next
  case False
  then show ?thesis using a1
  apply (simp add: sub-instr-sub1-def)
  by (simp add: return-def)
qed

```

```

lemma sub-instr-privilege:
assumes a1:  $s' = snd (fst (sub-instr instr$ 
   $(s::('a::len) sparc-state)))$ 
   $\wedge (((get-S (cpu-reg-val PSR s))::word1) = 0$ 

```

```

shows (((get-S (cpu-reg-val PSR s'))::word1) = 0
using a1
apply (simp add: sub-instr-def)
apply (simp add: Let-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
apply (simp add: case-prod-unfold)
by (meson sub-instr-sub1-privilege get-curr-win-privilege write-reg-privilege)

```

```

lemma mul-instr-sub1-privilege:
assumes a1: s' = snd (fst (mul-instr-sub1 instr-name result
  (s::('a::len) sparc-state)))
  ∧ (((get-S (cpu-reg-val PSR s))::word1) = 0
shows (((get-S (cpu-reg-val PSR s'))::word1) = 0
proof (cases instr-name ∈ {arith-type SMULcc,arith-type UMULcc})
  case True
    then show ?thesis using a1
    apply (simp add: mul-instr-sub1-def)
    apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
    by (blast intro: write-cpu-PSR-icc-privilege)
  next
    case False
    then show ?thesis using a1
    apply (simp add: mul-instr-sub1-def)
    by (simp add: return-def)
qed

```

```

lemma mul-instr-privilege:
assumes a1: s' = snd (fst (mul-instr instr
  (s::('a::len) sparc-state)))
  ∧ (((get-S (cpu-reg-val PSR s))::word1) = 0
shows (((get-S (cpu-reg-val PSR s'))::word1) = 0
using a1
apply (simp add: mul-instr-def)
apply (simp add: Let-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
apply (simp add: case-prod-unfold)
by (meson get-curr-win-privilege mul-instr-sub1-privilege write-cpu-y-privilege write-reg-privilege)

```

```

lemma div-write-new-val-privilege:
assumes a1: s' = snd (fst (div-write-new-val i result temp-V
  (s::('a::len) sparc-state)))
  ∧ (((get-S (cpu-reg-val PSR s))::word1) = 0
shows (((get-S (cpu-reg-val PSR s'))::word1) = 0
proof (cases (fst i) ∈ {arith-type UDIVcc,arith-type SDIVcc})
  case True
    then show ?thesis using a1
    apply (simp add: div-write-new-val-def)
    apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
    by (blast intro: write-cpu-PSR-icc-privilege)

```

```

next
  case False
  then show ?thesis using a1
  apply (simp add: div-write-new-val-def)
  by (simp add: return-def)
qed

lemma div-comp-privilege:
assumes a1: s' = snd (fst (div-comp instr rs1 rd operand2
  (s::('a::len) sparc-state)))
   $\wedge$  (((get-S (cpu-reg-val PSR s))::word1) = 0)
shows (((get-S (cpu-reg-val PSR s'))::word1) = 0)
using a1
apply (simp add: div-comp-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
apply (simp add: case-prod-unfold)
by (meson get-curr-win-privilege div-write-new-val-privilege write-reg-privilege)

lemma div-instr-privilege:
assumes a1: s' = snd (fst (div-instr instr
  (s::('a::len) sparc-state)))
   $\wedge$  (((get-S (cpu-reg-val PSR s))::word1) = 0)
shows (((get-S (cpu-reg-val PSR s'))::word1) = 0)
using a1
apply (simp add: div-instr-def)
apply (simp add: Let-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
apply (simp add: case-prod-unfold)
apply (simp add: return-def)
apply auto
  using raise-trap-privilege apply blast
using div-comp-privilege by blast

lemma save-restore-sub1-privilege:
assumes a1: s' = snd (fst (save-restore-sub1 result new-cwp rd
  (s::('a::len) sparc-state)))
   $\wedge$  (((get-S (cpu-reg-val PSR s))::word1) = 0)
shows (((get-S (cpu-reg-val PSR s'))::word1) = 0)
using a1
apply (simp add: save-restore-sub1-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
apply (simp add: case-prod-unfold)
using write-cpu-PSR-CWP-privilege write-reg-privilege by blast

method save-restore-instr-privilege-proof = (
  (simp add: save-restore-instr-def),
  (simp add: Let-def),
  (simp add: simpler-gets-def bind-def h1-def h2-def Let-def),
  (simp add: case-prod-unfold),

```

```

auto,
(blast intro: get-curr-win-privilege raise-trap-privilege),
(simp add: simpler-gets-def bind-def h1-def h2-def Let-def case-prod-unfold),
(blast intro: get-curr-win-privilege save-restore-sub1-privilege)
)

```

```

lemma save-restore-instr-privilege:
assumes a1: s' = snd (fst (save-restore-instr instr
  (s::('a::len) sparc-state)))
  ∧ (((get-S (cpu-reg-val PSR s))::word1) = 0
shows (((get-S (cpu-reg-val PSR s'))::word1) = 0
proof (cases fst instr = ctrl-type SAVE)
  case True
  then have f1: fst instr = ctrl-type SAVE by auto
  then show ?thesis using a1
  by save-restore-instr-privilege-proof
next
  case False
  then show ?thesis using a1
  by save-restore-instr-privilege-proof
qed

```

```

lemma call-instr-privilege:
assumes a1: s' = snd (fst (call-instr instr
  (s::('a::len) sparc-state)))
  ∧ (((get-S (cpu-reg-val PSR s))::word1) = 0
shows (((get-S (cpu-reg-val PSR s'))::word1) = 0
using a1
apply (simp add: call-instr-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
apply (simp add: case-prod-unfold)
by (meson get-curr-win-privilege write-cpu-npc-privilege write-cpu-pc-privilege write-reg-privilege)

```

```

lemma jmpl-instr-privilege:
assumes a1: s' = snd (fst (jmpl-instr instr
  (s::('a::len) sparc-state)))
  ∧ (((get-S (cpu-reg-val PSR s))::word1) = 0
shows (((get-S (cpu-reg-val PSR s'))::word1) = 0
using a1
apply (simp add: jmpl-instr-def)
apply (simp add: Let-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
apply (simp add: case-prod-unfold)
apply auto
  using get-curr-win-privilege raise-trap-privilege apply blast
apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
apply (simp add: case-prod-unfold)
by (meson get-curr-win-privilege write-cpu-npc-privilege write-cpu-pc-privilege write-reg-privilege)

```

```

lemma rett-instr-privilege:
assumes a1: snd (rett-instr i s) = False ∧
  s' = snd (fst (rett-instr instr
    (s::('a::len) sparc-state))))
  ∧ (((get-S (cpu-reg-val PSR s))::word1) = 0
shows (((get-S (cpu-reg-val PSR s'))::word1) = 0
using a1
apply (simp add: rett-instr-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
apply auto
apply (simp add: case-prod-unfold)
apply (simp add: return-def)
apply (blast intro: raise-trap-privilege)
apply (simp add: bind-def h1-def h2-def Let-def)
by (simp add: case-prod-unfold fail-def)

method read-state-reg-instr-privilege-proof = (
  (simp add: read-state-reg-instr-def),
  (simp add: Let-def),
  (simp add: simpler-gets-def bind-def h1-def h2-def Let-def),
  (simp add: case-prod-unfold)
)

lemma read-state-reg-instr-privilege:
assumes a1: s' = snd (fst (read-state-reg-instr instr
  (s::('a::len) sparc-state))))
  ∧ (((get-S (cpu-reg-val PSR s))::word1) = 0
shows (((get-S (cpu-reg-val PSR s'))::word1) = 0
proof (cases (fst instr ∈ {sreg-type RDPSR,sreg-type RDWIM,sreg-type RDTBR}
  ∨
    (fst instr = sreg-type RDASR ∧ privileged-ASR (get-operand-w5 ((snd
instr)!0))))))
    case True
    then have (fst instr ∈ {sreg-type RDPSR,sreg-type RDWIM,sreg-type RDTBR}
  ∨
    (fst instr = sreg-type RDASR ∧ privileged-ASR (get-operand-w5 ((snd
instr)!0))))
    ∧ (((get-S (cpu-reg-val PSR (snd (fst (get-curr-win () s))))))::word1) = 0
    by (metis assms get-curr-win-privilege)
    then show ?thesis using a1
    apply read-state-reg-instr-privilege-proof
    by (blast intro: raise-trap-privilege get-curr-win-privilege)
  next
  case False
  then have f1: ¬((fst instr = sreg-type RDPSR ∨
    fst instr = sreg-type RDWIM ∨
    fst instr = sreg-type RDTBR ∨
    fst instr = sreg-type RDASR ∧ privileged-ASR (get-operand-w5
(snd instr ! 0))) ∧

```

```

      (get-S (cpu-reg-val PSR (snd (fst (get-curr-win () s)))) = 0)
    by blast
  then show ?thesis
proof (cases illegal-instruction-ASR (get-operand-w5 ((snd instr)!0)))
  case True
  then show ?thesis using a1 f1
  apply read-state-reg-instr-privilege-proof
  by (simp add: illegal-instruction-ASR-def)
next
  case False
  then have f2: ¬(illegal-instruction-ASR (get-operand-w5 ((snd instr)!0)))
  by auto
  then show ?thesis
proof (cases (get-operand-w5 ((snd instr)!1)) ≠ 0)
  case True
  then have f3: (get-operand-w5 ((snd instr)!1)) ≠ 0
  by auto
  then show ?thesis
proof (cases fst instr = sreg-type RDY)
  case True
  then show ?thesis using a1 f1 f2 f3
  apply (simp add: read-state-reg-instr-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
  apply (simp add: case-prod-unfold)
  by (blast intro: get-curr-win-privilege write-reg-privilege)
next
  case False
  then have f4: ¬(fst instr = sreg-type RDY) by auto
  then show ?thesis
proof (cases fst instr = sreg-type RDASR)
  case True
  then show ?thesis using a1 f1 f2 f3 f4
  apply read-state-reg-instr-privilege-proof
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
  by (blast intro: get-curr-win-privilege write-reg-privilege)
next
  case False
  then have f5: ¬(fst instr = sreg-type RDASR) by auto
  then show ?thesis
proof (cases fst instr = sreg-type RDPSR)
  case True
  then show ?thesis using a1 f1 f2 f3 f4 f5
  apply read-state-reg-instr-privilege-proof
  by (blast intro: get-curr-win-privilege write-reg-privilege)
next
  case False
  then show ?thesis using a1 f1 f2 f3 f4 f5
  apply read-state-reg-instr-privilege-proof
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)

```

```

      by (blast intro: get-curr-win-privilege write-reg-privilege)
    qed
  qed
  qed
next
case False
then show ?thesis using a1
apply read-state-reg-instr-privilege-proof
apply (simp add: return-def)
using f1 f2 get-curr-win-privilege by blast
qed
qed
qed

```

```

method write-state-reg-instr-privilege-proof = (
  (simp add: write-state-reg-instr-def),
  (simp add: Let-def),
  (simp add: simplifier-modify-def bind-def h1-def h2-def Let-def),
  (simp add: case-prod-unfold)
)

```

```

lemma write-state-reg-instr-privilege:
  assumes a1:  $s' = \text{snd } (\text{fst } (\text{write-state-reg-instr } \text{instr}
    (\text{s}::('a::\text{len}) \text{sparc-state})))$ 
     $\wedge (((\text{get-S } (\text{cpu-reg-val } \text{PSR } \text{s}))::\text{word1}) = 0$ 
  shows  $((\text{get-S } (\text{cpu-reg-val } \text{PSR } \text{s}'))::\text{word1}) = 0$ 
  proof (cases  $\text{fst } \text{instr} = \text{sreg-type } \text{WRY}$ )
  case True
  then show ?thesis using a1
  apply write-state-reg-instr-privilege-proof
  apply (simp add: simplifier-modify-def)
  apply (simp add: delayed-pool-add-def DELAYNUM-def)
  by (blast intro: cpu-reg-mod-y-privilege get-curr-win-privilege)
next
case False
then have f1:  $\neg(\text{fst } \text{instr} = \text{sreg-type } \text{WRY})$  by auto
then show ?thesis
proof (cases  $\text{fst } \text{instr} = \text{sreg-type } \text{WRASR}$ )
case True
then show ?thesis
  using a1 f1
  apply write-state-reg-instr-privilege-proof
  apply (simp add: simplifier-modify-def)
  apply auto
  using illegal-instruction-ASR-def apply blast
  using illegal-instruction-ASR-def apply blast
  using illegal-instruction-ASR-def apply blast
  using raise-trap-privilege get-curr-win-privilege apply blast
  apply (simp add: simplifier-modify-def delayed-pool-add-def DELAYNUM-def)

```

```

    using cpu-reg-mod-asr-privilege get-curr-win-privilege apply blast
apply (simp add: simpler-modify-def delayed-pool-add-def DELAYNUM-def)
using cpu-reg-mod-asr-privilege get-curr-win-privilege by blast
next
  case False
then have f2:  $\neg(\text{fst instr} = \text{sreg-type WRASR})$  by auto
have f3:  $\text{get-S}(\text{cpu-reg-val PSR}(\text{snd}(\text{fst}(\text{get-curr-win}(\ ) s)))) = 0$ 
  using get-curr-win-privilege a1 by (metis ucast-id)
then show ?thesis
proof (cases fst instr = sreg-type WRPSR)
  case True
then show ?thesis using a1 f1 f2 f3
apply write-state-reg-instr-privilege-proof
by (metis raise-trap-privilege ucast-0)
next
  case False
then have f4:  $\neg(\text{fst instr} = \text{sreg-type WRPSR})$  by auto
then show ?thesis
proof (cases fst instr = sreg-type WRWIM)
  case True
then show ?thesis using a1 f1 f2 f3 f4
apply write-state-reg-instr-privilege-proof
by (metis raise-trap-privilege ucast-0)
next
  case False
then have f5:  $\neg(\text{fst instr} = \text{sreg-type WRWIM})$  by auto
then show ?thesis using a1 f1 f2 f3 f4 f5
apply write-state-reg-instr-privilege-proof
by (metis raise-trap-privilege ucast-0)
qed
qed
qed
qed

```

lemma *flush-instr-privilege*:

assumes *a1*: $s' = \text{snd}(\text{fst}(\text{flush-instr instr}(\text{s}::('a::\text{len}) \text{sparc-state})))$

$\wedge (((\text{get-S}(\text{cpu-reg-val PSR } s)))::\text{word1}) = 0$

shows $((\text{get-S}(\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$

using *a1*

apply (*simp add: flush-instr-def*)

apply (*simp add: simpler-gets-def bind-def h1-def h2-def Let-def simpler-modify-def*)

by (*auto simp add: flush-cache-all-privilege*)

lemma *branch-instr-privilege*:

assumes *a1*: $s' = \text{snd}(\text{fst}(\text{branch-instr instr}(\text{s}::('a::\text{len}) \text{sparc-state})))$

$\wedge (((\text{get-S}(\text{cpu-reg-val PSR } s)))::\text{word1}) = 0$

shows $((\text{get-S}(\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$

```

using a1
apply (simp add: branch-instr-def)
apply (simp add: Let-def simpler-gets-def bind-def h1-def h2-def)
apply (simp add: case-prod-unfold return-def)
by (meson set-annul-privilege write-cpu-npc-privilege write-cpu-pc-privilege)

```

```

method dispath-instr-privilege-proof = (
  (simp add: dispatch-instruction-def),
  (simp add: simpler-gets-def bind-def h1-def h2-def Let-def),
  (simp add: Let-def)
)

```

lemma *dispath-instr-privilege*:

```

assumes a1: snd (dispatch-instruction instr s) = False ∧
  s' = snd (fst (dispatch-instruction instr s))
  ∧ (((get-S (cpu-reg-val PSR s))::word1) = 0
shows (((get-S (cpu-reg-val PSR s'))::word1) = 0
proof (cases get-trap-set s = {})
  case True
  then have f1: get-trap-set s = {} by auto
  show ?thesis
  proof (cases fst instr ∈ {load-store-type LDSB,load-store-type LDUB,
    load-store-type LDUBA,load-store-type LDUH,load-store-type LD,
    load-store-type LDA,load-store-type LDD})
    case True
    then show ?thesis using a1 f1
    apply dispath-instr-privilege-proof
    by (blast intro: load-instr-privilege)
  next
  case False
  then have f2: ¬(fst instr ∈ {load-store-type LDSB,load-store-type LDUB,
    load-store-type LDUBA,load-store-type LDUH,load-store-type LD,
    load-store-type LDA,load-store-type LDD})
    by auto
  then show ?thesis
  proof (cases fst instr ∈ {load-store-type STB,load-store-type STH,
    load-store-type ST,load-store-type STA,load-store-type STD})
    case True
    then show ?thesis using a1 f1 f2
    apply dispath-instr-privilege-proof
    by (blast intro: store-instr-privilege)
  next
  case False
  then have f3: ¬(fst instr ∈ {load-store-type STB,load-store-type STH,
    load-store-type ST,load-store-type STA,load-store-type STD})
    by auto
  then show ?thesis
  proof (cases fst instr ∈ {sethi-type SETHI})
    case True

```

```

then show ?thesis using a1 f1 f2 f3
apply dispath-instr-privilege-proof
by (blast intro: sethi-instr-privilege)
next
  case False
  then have f4:  $\neg(\text{fst instr} \in \{\text{sethi-type SETHI}\})$ 
    by auto
  then show ?thesis
  proof (cases fst instr  $\in$  {nop-type NOP})
    case True
    then show ?thesis using a1 f1 f2 f3 f4
    apply dispath-instr-privilege-proof
    by (blast intro: nop-instr-privilege)
  next
    case False
    then have f5:  $\neg(\text{fst instr} \in \{\text{nop-type NOP}\})$ 
      by auto
    then show ?thesis
      proof (cases fst instr  $\in$  {logic-type ANDs,logic-type ANDcc,logic-type
ANDN,
  logic-type ANDNcc,logic-type ORs,logic-type ORcc,logic-type ORN,
  logic-type XORs,logic-type XNOR})
        case True
        then show ?thesis using a1 f1 f2 f3 f4 f5
        apply dispath-instr-privilege-proof
        by (blast intro: logical-instr-privilege)
      next
        case False
        then have f6:  $\neg(\text{fst instr} \in \{\text{logic-type ANDs,logic-type ANDcc,logic-type
ANDN,
  logic-type ANDNcc,logic-type ORs,logic-type ORcc,logic-type ORN,
  logic-type XORs,logic-type XNOR}\})$ 
          by auto
        show ?thesis
        proof (cases fst instr  $\in$  {shift-type SLL,shift-type SRL,shift-type SRA})
          case True
          then show ?thesis using a1 f1 f2 f3 f4 f5 f6
          apply dispath-instr-privilege-proof
          by (blast intro: shift-instr-privilege)
        next
          case False
          then have f7:  $\neg(\text{fst instr} \in \{\text{shift-type SLL,shift-type SRL,shift-type
SRA}\})$ 
            by auto
          then show ?thesis
          proof (cases fst instr  $\in$  {arith-type ADD,arith-type ADDcc,arith-type
ADDX})
            case True
            then show ?thesis using a1 f1 f2 f3 f4 f5 f6 f7

```

```

    apply dispath-instr-privilege-proof
    by (blast intro: add-instr-privilege)
  next
  case False
  then have f8: ¬(fst instr ∈ {arith-type ADD,arith-type ADDcc,arith-type
ADDX})
    by auto
    then show ?thesis
  proof (cases fst instr ∈ {arith-type SUB,arith-type SUBcc,arith-type
SUBX})
    case True
    then show ?thesis using a1 f1 f2 f3 f4 f5 f6 f7 f8
    apply dispath-instr-privilege-proof
    by (blast intro: sub-instr-privilege)
  next
  case False
  then have f9: ¬(fst instr ∈ {arith-type SUB,arith-type SUBcc,arith-type
SUBX})
    by auto
    then show ?thesis
  proof (cases fst instr ∈ {arith-type UMUL,arith-type SMUL,arith-type
SMULcc})
    case True
    then show ?thesis using a1 f1 f2 f3 f4 f5 f6 f7 f8 f9
    apply dispath-instr-privilege-proof
    by (blast intro: mul-instr-privilege)
  next
  case False
  then have f10: ¬(fst instr ∈ {arith-type UMUL,arith-type SMUL,
arith-type SMULcc})
    by auto
    then show ?thesis
  proof (cases fst instr ∈ {arith-type UDIV,arith-type UDIVcc,arith-type
SDIV})
    case True
    then show ?thesis using a1 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10
    apply dispath-instr-privilege-proof
    by (blast intro: div-instr-privilege)
  next
  case False
  then have f11: ¬(fst instr ∈ {arith-type UDIV,
arith-type UDIVcc,arith-type SDIV})
    by auto
    then show ?thesis
  proof (cases fst instr ∈ {ctrl-type SAVE,ctrl-type RESTORE})
    case True
    then show ?thesis using a1 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11
    apply dispath-instr-privilege-proof
    by (blast intro: save-restore-instr-privilege)
  
```

```

next
  case False
    then have f12:  $\neg(\text{fst instr} \in \{\text{ctrl-type SAVE}, \text{ctrl-type}$ 
RESTORE})
      by auto
    then show ?thesis
    proof (cases fst instr  $\in \{\text{call-type CALL}\}$ )
      case True
        then show ?thesis using a1 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11
          f12
          apply dispath-instr-privilege-proof
          by (blast intro: call-instr-privilege)
        next
          case False
            then have f13:  $\neg(\text{fst instr} \in \{\text{call-type CALL}\})$  by auto
            then show ?thesis
            proof (cases fst instr  $\in \{\text{ctrl-type JMPL}\}$ )
              case True
                then show ?thesis using a1 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11
                  f12 f13
                apply dispath-instr-privilege-proof
                by (blast intro: jmpl-instr-privilege)
              next
                case False
                  then have f14:  $\neg(\text{fst instr} \in \{\text{ctrl-type JMPL}\})$  by auto
                  then show ?thesis
                  proof (cases fst instr  $\in \{\text{ctrl-type RETT}\}$ )
                    case True
                      then show ?thesis using a1 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10
                        f11 f12 f13
                      apply dispath-instr-privilege-proof
                      by (blast intro: rett-instr-privilege)
                    next
                      case False
                        then have f15:  $\neg(\text{fst instr} \in \{\text{ctrl-type RETT}\})$  by auto
                        then show ?thesis
                        proof (cases fst instr  $\in \{\text{sreg-type RDY}, \text{sreg-type RDPSR},$ 
                          sreg-type RDWIM}, \text{sreg-type RDTBR}\})
                          case True
                            then show ?thesis using a1 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10
                              f11 f12
                            apply dispath-instr-privilege-proof
                            by (blast intro: read-state-reg-instr-privilege)
                          next
                            case False
                              then have f16:  $\neg(\text{fst instr} \in \{\text{sreg-type RDY}, \text{sreg-type}$ 
RDPSR,

```

```

sreg-type RDWIM, sreg-type RDTBR}) by auto
then show ?thesis
  proof (cases fst instr ∈ {sreg-type WRY,sreg-type
WRPSR,
sreg-type WRWIM, sreg-type WRTBR})
  case True
  then show ?thesis using a1 f1 f2 f3 f4 f5 f6 f7 f8 f9
f10 f11 f12
f13 f14 f15 f16
apply dispath-instr-privilege-proof
by (blast intro: write-state-reg-instr-privilege)
next
  case False
  then have f17: ¬(fst instr ∈ {sreg-type WRY,sreg-type
WRPSR,
sreg-type WRWIM, sreg-type WRTBR}) by auto
then show ?thesis
proof (cases fst instr ∈ {load-store-type FLUSH})
  case True
  then show ?thesis using a1 f1 f2 f3 f4 f5 f6 f7 f8 f9
f10 f11
f12 f13 f14 f15 f16 f17
apply dispath-instr-privilege-proof
by (blast intro: flush-instr-privilege)
next
  case False
  then have f18: ¬(fst instr ∈ {load-store-type FLUSH})
by auto
  then show ?thesis
proof (cases fst instr ∈ {bicc-type BE,bicc-type BNE,
BGE,
bicc-type BGU,bicc-type BLE,bicc-type BL,bicc-type
BLEU,
bicc-type BNEG,bicc-type BG,bicc-type BCS,bicc-type
bicc-type BCC,bicc-type BA,bicc-type BN})
  case True
  then show ?thesis using a1 f1 f2 f3 f4 f5 f6 f7 f8
f9 f10 f11
f12 f13 f14 f15 f16 f17 f18
apply dispath-instr-privilege-proof
by (blast intro: branch-instr-privilege)
next
  case False
  then show ?thesis using a1 f1 f2 f3 f4 f5 f6 f7 f8
f9 f10 f11
f12 f13 f14 f15 f16 f17 f18
apply dispath-instr-privilege-proof
by (simp add: fail-def)
qed

```

```

      qed
    qed
  qed
  qed
  qed
  qed
  qed
  qed
  qed
  qed
  qed
  qed
  qed
  qed
  qed
  next
  case False
  then show ?thesis using a1
  apply (simp add: dispatch-instruction-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def)
  apply (simp add: Let-def)
  by (simp add: return-def)
qed

lemma execute-instr-sub1-privilege:
  assumes a1: snd (execute-instr-sub1 i s) = False ∧
    s' = snd (fst (execute-instr-sub1 i s))
    ∧ (((get-S (cpu-reg-val PSR s)))::word1) = 0
    ∧ (((get-S (cpu-reg-val PSR s'))::word1) = 0)
  shows (((get-S (cpu-reg-val PSR s'))::word1) = 0)
  proof (cases get-trap-set s = {} ∧ fst i ∉ {call-type CALL,ctrl-type RETT,ctrl-type
JMPL,
    bicc-type BE,bicc-type BNE,bicc-type BGU,
    bicc-type BLE,bicc-type BL,bicc-type BGE,
    bicc-type BNEG,bicc-type BG,
    bicc-type BCS,bicc-type BLEU,bicc-type BCC,
    bicc-type BA,bicc-type BN})
  case True
  then show ?thesis using a1
  apply (simp add: execute-instr-sub1-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
  apply (simp add: case-prod-unfold return-def)
  by (auto intro: write-cpu-pc-privilege write-cpu-npc-privilege)
  next
  case False
  then show ?thesis using a1
  apply (simp add: execute-instr-sub1-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)

```

```

apply (simp add: case-prod-unfold return-def)
by auto
qed

```

Assume that there is no *delayed-write* and there is no traps to be executed. If an instruction is executed as a user, the privilege will not be changed to supervisor after the execution.

```

theorem safe-privilege :
assumes a1: get-delayed-pool s = []  $\wedge$  get-trap-set s = {}  $\wedge$ 
  snd (execute-instruction() s) = False  $\wedge$ 
  s' = snd (fst (execute-instruction() s))  $\wedge$ 
  (((get-S (cpu-reg-val PSR s))::word1) = 0
shows (((get-S (cpu-reg-val PSR s'))::word1) = 0
proof (cases exe-mode-val s)
  case True
  then have f2: exe-mode-val s = True by auto
  then show ?thesis
  proof (cases  $\exists e$ . fetch-instruction (delayed-pool-write s) = Inl e)
    case True
    then have f3:  $\exists e$ . fetch-instruction (delayed-pool-write s) = Inl e
      by auto
    then have f4:  $\neg (\exists v$ . fetch-instruction (delayed-pool-write s) = Inr v)
      using fetch-instr-result-3 by auto
    then show ?thesis using a1 f2 f3 raise-trap-result empty-delayed-pool-write-privilege
      raise-trap-privilege
    apply (simp add: execute-instruction-def)
    apply (simp add: exec-gets return-def)
    apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
    apply (simp add: simpler-modify-def)
    apply clarsimp
    apply (simp add: case-prod-unfold)
    by (blast intro: empty-delayed-pool-write-privilege raise-trap-privilege)
  next
  case False
  then have f5:  $\exists v$ . fetch-instruction (delayed-pool-write s) = Inr v
    using fetch-instr-result-1 by blast
  then have f6:  $\exists v$ . fetch-instruction (delayed-pool-write s) = Inr v  $\wedge$ 
 $\neg (\exists e$ . ((decode-instruction v)::(Exception list + instruction)) = Inl e)
    using a1 f2 dispatch-fail by blast
  then have f7:  $\exists v$ . fetch-instruction (delayed-pool-write s) = Inr v  $\wedge$ 
 $(\exists v1$ . ((decode-instruction v)::(Exception list + instruction)) = Inr v1)
    using decode-instr-result-4 by auto
  then show ?thesis
  proof (cases annul-val (delayed-pool-write s))
    case True
    then show ?thesis using a1 f2 f7
    apply (simp add: execute-instruction-def)
    apply (simp add: exec-gets return-def)
    apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)

```

```

apply (simp add: simpler-modify-def)
apply clarsimp
apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
apply (simp add: case-prod-unfold)
by (auto intro: empty-delayed-pool-write-privilege
      set-annul-privilege write-cpu-npc-privilege write-cpu-pc-privilege)
next
  case False
  then show ?thesis using a1 f2 f7
  apply (simp add: execute-instruction-def)
  apply (simp add: exec-gets return-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
  apply (simp add: simpler-modify-def)
  apply clarsimp
  apply (simp add: bind-def h1-def h2-def Let-def)
  apply (simp add: case-prod-unfold)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
  apply (simp add: simpler-modify-def return-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
  apply (simp add: case-prod-unfold)
  by (auto intro: empty-delayed-pool-write-privilege dispath-instr-privilege
      execute-instr-sub1-privilege)
  qed
qed
next
  case False
  then show ?thesis using a1
  apply (simp add: execute-instruction-def)
  by (simp add: simpler-gets-def bind-def h1-def h2-def Let-def return-def)
qed

```

26 Single step non-interference property.

definition *user-accessible*:: ('a::len) sparc-state \Rightarrow phys-address \Rightarrow bool **where**
user-accessible s pa $\equiv \exists va p. (virt-to-phys va (mmu s) (mem s)) = \text{Some } p \wedge$
 mmu-readable (get-acc-flag (snd p)) 10 \wedge
 (fst p) = pa — Passing *asi* = 8 is the same.

lemma *user-accessible-8*:
assumes a1: mmu-readable (get-acc-flag (snd p)) 8
shows mmu-readable (get-acc-flag (snd p)) 10
using a1 **by** (simp add: mmu-readable-def)

definition *mem-equal*:: ('a) sparc-state \Rightarrow ('a) sparc-state \Rightarrow
 phys-address \Rightarrow bool **where**
mem-equal s1 s2 pa \equiv
 (mem s1) 8 (pa AND 68719476732) = (mem s2) 8 (pa AND 68719476732) \wedge
 (mem s1) 8 ((pa AND 68719476732) + 1) = (mem s2) 8 ((pa AND 68719476732)
 + 1) \wedge

$$\begin{aligned}
& (\text{mem } s1) \ 8 \ ((pa \ \text{AND} \ 68719476732) + 2) = (\text{mem } s2) \ 8 \ ((pa \ \text{AND} \ 68719476732) \\
& + 2) \wedge \\
& (\text{mem } s1) \ 8 \ ((pa \ \text{AND} \ 68719476732) + 3) = (\text{mem } s2) \ 8 \ ((pa \ \text{AND} \ 68719476732) \\
& + 3) \wedge \\
& (\text{mem } s1) \ 9 \ (pa \ \text{AND} \ 68719476732) = (\text{mem } s2) \ 9 \ (pa \ \text{AND} \ 68719476732) \wedge \\
& (\text{mem } s1) \ 9 \ ((pa \ \text{AND} \ 68719476732) + 1) = (\text{mem } s2) \ 9 \ ((pa \ \text{AND} \ 68719476732) \\
& + 1) \wedge \\
& (\text{mem } s1) \ 9 \ ((pa \ \text{AND} \ 68719476732) + 2) = (\text{mem } s2) \ 9 \ ((pa \ \text{AND} \ 68719476732) \\
& + 2) \wedge \\
& (\text{mem } s1) \ 9 \ ((pa \ \text{AND} \ 68719476732) + 3) = (\text{mem } s2) \ 9 \ ((pa \ \text{AND} \ 68719476732) \\
& + 3) \wedge \\
& (\text{mem } s1) \ 10 \ (pa \ \text{AND} \ 68719476732) = (\text{mem } s2) \ 10 \ (pa \ \text{AND} \ 68719476732) \wedge \\
& (\text{mem } s1) \ 10 \ ((pa \ \text{AND} \ 68719476732) + 1) = (\text{mem } s2) \ 10 \ ((pa \ \text{AND} \ 68719476732) \\
& + 1) \wedge \\
& (\text{mem } s1) \ 10 \ ((pa \ \text{AND} \ 68719476732) + 2) = (\text{mem } s2) \ 10 \ ((pa \ \text{AND} \ 68719476732) \\
& + 2) \wedge \\
& (\text{mem } s1) \ 10 \ ((pa \ \text{AND} \ 68719476732) + 3) = (\text{mem } s2) \ 10 \ ((pa \ \text{AND} \ 68719476732) \\
& + 3) \wedge \\
& (\text{mem } s1) \ 11 \ (pa \ \text{AND} \ 68719476732) = (\text{mem } s2) \ 11 \ (pa \ \text{AND} \ 68719476732) \wedge \\
& (\text{mem } s1) \ 11 \ ((pa \ \text{AND} \ 68719476732) + 1) = (\text{mem } s2) \ 11 \ ((pa \ \text{AND} \ 68719476732) \\
& + 1) \wedge \\
& (\text{mem } s1) \ 11 \ ((pa \ \text{AND} \ 68719476732) + 2) = (\text{mem } s2) \ 11 \ ((pa \ \text{AND} \ 68719476732) \\
& + 2) \wedge \\
& (\text{mem } s1) \ 11 \ ((pa \ \text{AND} \ 68719476732) + 3) = (\text{mem } s2) \ 11 \ ((pa \ \text{AND} \ 68719476732) \\
& + 3)
\end{aligned}$$

low-equal defines the equivalence relation over two sparcs states that is an analogy to the $=_L$ relation over memory contexts in the traditional non-interference theorem.

definition *low-equal*:: ('a::len) sparcs-state \Rightarrow ('a) sparcs-state \Rightarrow bool **where**

low-equal s1 s2 \equiv

$$\begin{aligned}
& (\text{cpu-reg } s1) = (\text{cpu-reg } s2) \wedge \\
& (\text{user-reg } s1) = (\text{user-reg } s2) \wedge \\
& (\text{sys-reg } s1) = (\text{sys-reg } s2) \wedge \\
& (\forall va. (\text{virt-to-phys } va \ (\text{mmu } s1) \ (\text{mem } s1)) = (\text{virt-to-phys } va \ (\text{mmu } s2) \ (\text{mem } \\
& s2))) \wedge \\
& (\forall pa. (\text{user-accessible } s1 \ pa) \longrightarrow \text{mem-equal } s1 \ s2 \ pa) \wedge \\
& (\text{mmu } s1) = (\text{mmu } s2) \wedge \\
& (\text{state-var } s1) = (\text{state-var } s2) \wedge \\
& (\text{traps } s1) = (\text{traps } s2) \wedge \\
& (\text{undef } s1) = (\text{undef } s2)
\end{aligned}$$

lemma *low-equal-com*: *low-equal* s1 s2 \implies *low-equal* s2 s1

apply (*simp add*: *low-equal-def*)

apply (*simp add*: *mem-equal-def user-accessible-def*)

by *metis*

lemma *non-exe-mode-equal*: *exe-mode-val* s = False \wedge

```

get-trap-set s = {} ∧
Some t = NEXT s ⇒
t = s
apply (simp add: NEXT-def execute-instruction-def)
apply auto
by (simp add: simpler-gets-def bind-def h1-def h2-def Let-def return-def)

```

```

lemma exe-mode-low-equal:
assumes a1: low-equal s1 s2
shows exe-mode-val s1 = exe-mode-val s2
using a1 apply (simp add: low-equal-def)
by (simp add: exe-mode-val-def)

```

```

lemma mem-val-mod-state: mem-val-alt asi a s = mem-val-alt asi a
(s(cpu-reg := new-cpu-reg,
 user-reg := new-user-reg,
 dwrite := new-dwrite,
 state-var := new-state-var,
 traps := new-traps,
 undef := new-undef))
apply (simp add: mem-val-alt-def)
by (simp add: Let-def)

```

```

lemma mem-val-w32-mod-state: mem-val-w32 asi a s = mem-val-w32 asi a
(s(cpu-reg := new-cpu-reg,
 user-reg := new-user-reg,
 dwrite := new-dwrite,
 state-var := new-state-var,
 traps := new-traps,
 undef := new-undef))
apply (simp add: mem-val-w32-def)
apply (simp add: Let-def)
by (metis mem-val-mod-state)

```

```

lemma load-word-mem-mod-state: load-word-mem s addr asi = load-word-mem
(s(cpu-reg := new-cpu-reg,
 user-reg := new-user-reg,
 dwrite := new-dwrite,
 state-var := new-state-var,
 traps := new-traps,
 undef := new-undef)) addr asi
apply (simp add: load-word-mem-def)
apply (case-tac virt-to-phys addr (mmu s) (mem s) = None)
apply auto
by (auto simp add: mem-val-w32-mod-state)

```

```

lemma load-word-mem2-mod-state:
fst (case load-word-mem s addr asi of None ⇒ (None, s)
 | Some w ⇒ (Some w, add-data-cache s addr w 15)) =

```

```

fst (case load-word-mem (s\cpu-reg := new-cpu-reg,
  user-reg := new-user-reg,
  dwrite := new-dwrite,
  state-var := new-state-var,
  traps := new-traps,
  undef := new-undef)) addr asi of
  None => (None, (s\cpu-reg := new-cpu-reg,
  user-reg := new-user-reg,
  dwrite := new-dwrite,
  state-var := new-state-var,
  traps := new-traps,
  undef := new-undef))
  | Some w => (Some w, add-data-cache (s\cpu-reg := new-cpu-reg,
  user-reg := new-user-reg,
  dwrite := new-dwrite,
  state-var := new-state-var,
  traps := new-traps,
  undef := new-undef)) addr w 15))
proof (cases load-word-mem s addr asi = None)
case True
then have load-word-mem s addr asi = None ∧
  load-word-mem (s\cpu-reg := new-cpu-reg,
  user-reg := new-user-reg,
  dwrite := new-dwrite,
  state-var := new-state-var,
  traps := new-traps,
  undef := new-undef)) addr asi = None
using load-word-mem-mod-state by metis
then show ?thesis by auto
next
case False
then have ∃ w. load-word-mem s addr asi = Some w by auto
then have ∃ w. load-word-mem s addr asi = Some w ∧
  load-word-mem (s\cpu-reg := new-cpu-reg,
  user-reg := new-user-reg,
  dwrite := new-dwrite,
  state-var := new-state-var,
  traps := new-traps,
  undef := new-undef)) addr asi = Some w
using load-word-mem-mod-state by metis
then show ?thesis by auto
qed

```

lemma load-word-mem3-mod-state:

```

fst (case load-word-mem s addr asi of None => (None, s)
  | Some w => (Some w, add-instr-cache s addr w 15)) =
fst (case load-word-mem (s\cpu-reg := new-cpu-reg,
  user-reg := new-user-reg,
  dwrite := new-dwrite,

```

```

state-var := new-state-var,
traps := new-traps,
undef := new-undef)) addr asi of
  None ⇒ (None, (s(cpu-reg := new-cpu-reg,
user-reg := new-user-reg,
dwrite := new-dwrite,
state-var := new-state-var,
traps := new-traps,
undef := new-undef)))
  | Some w ⇒ (Some w, add-instr-cache (s(cpu-reg := new-cpu-reg,
user-reg := new-user-reg,
dwrite := new-dwrite,
state-var := new-state-var,
traps := new-traps,
undef := new-undef)) addr w 15))
proof (cases load-word-mem s addr asi = None)
case True
then have load-word-mem s addr asi = None ∧
  load-word-mem (s(cpu-reg := new-cpu-reg,
user-reg := new-user-reg,
dwrite := new-dwrite,
state-var := new-state-var,
traps := new-traps,
undef := new-undef)) addr asi = None
using load-word-mem-mod-state by metis
then show ?thesis by auto
next
case False
then have ∃ w. load-word-mem s addr asi = Some w by auto
then have ∃ w. load-word-mem s addr asi = Some w ∧
  load-word-mem (s(cpu-reg := new-cpu-reg,
user-reg := new-user-reg,
dwrite := new-dwrite,
state-var := new-state-var,
traps := new-traps,
undef := new-undef)) addr asi = Some w
using load-word-mem-mod-state by metis
then show ?thesis by auto
qed

lemma read-dcache-mod-state: read-data-cache s addr = read-data-cache
(s(cpu-reg := new-cpu-reg,
user-reg := new-user-reg,
dwrite := new-dwrite,
state-var := new-state-var,
traps := new-traps,
undef := new-undef)) addr
apply (simp add: read-data-cache-def)
by (simp add: dcache-val-def)

```

lemma *read-dcache2-mod-state*:
fst (case read-data-cache *s addr* of *None* \Rightarrow (*None*, *s*)
| *Some w* \Rightarrow (*Some w*, *s*)) =
fst (case read-data-cache (*s* | *cpu-reg* := *new-cpu-reg*,
user-reg := *new-user-reg*,
dwrite := *new-dwrite*,
state-var := *new-state-var*,
traps := *new-traps*,
undef := *new-undef*)) *addr* of
None \Rightarrow (*None*, (*s* | *cpu-reg* := *new-cpu-reg*,
user-reg := *new-user-reg*,
dwrite := *new-dwrite*,
state-var := *new-state-var*,
traps := *new-traps*,
undef := *new-undef*)))
| *Some w* \Rightarrow (*Some w*, (*s* | *cpu-reg* := *new-cpu-reg*,
user-reg := *new-user-reg*,
dwrite := *new-dwrite*,
state-var := *new-state-var*,
traps := *new-traps*,
undef := *new-undef*))))
proof (cases read-data-cache *s addr* = *None*)
case *True*
then have read-data-cache *s addr* = *None* \wedge
read-data-cache (*s* | *cpu-reg* := *new-cpu-reg*,
user-reg := *new-user-reg*,
dwrite := *new-dwrite*,
state-var := *new-state-var*,
traps := *new-traps*,
undef := *new-undef*)) *addr* = *None*
using *read-dcache-mod-state* **by** *metis*
then show ?*thesis* **by** *auto*
next
case *False*
then have $\exists w$. read-data-cache *s addr* = *Some w* **by** *auto*
then have $\exists w$. read-data-cache *s addr* = *Some w* \wedge
read-data-cache (*s* | *cpu-reg* := *new-cpu-reg*,
user-reg := *new-user-reg*,
dwrite := *new-dwrite*,
state-var := *new-state-var*,
traps := *new-traps*,
undef := *new-undef*)) *addr* = *Some w*
using *read-dcache-mod-state* **by** *metis*
then show ?*thesis* **by** *auto*
qed

lemma *read-icache-mod-state*: read-instr-cache *s addr* = read-instr-cache
(*s* | *cpu-reg* := *new-cpu-reg*,

```

    user-reg := new-user-reg,
    dwrite := new-dwrite,
    state-var := new-state-var,
    traps := new-traps,
    undef := new-undef)) addr
apply (simp add: read-instr-cache-def)
by (simp add: icache-val-def)

lemma read-icache2-mod-state:
fst (case read-instr-cache s addr of None  $\Rightarrow$  (None, s)
    | Some w  $\Rightarrow$  (Some w, s)) =
fst (case read-instr-cache (s\cpu-reg := new-cpu-reg,
    user-reg := new-user-reg,
    dwrite := new-dwrite,
    state-var := new-state-var,
    traps := new-traps,
    undef := new-undef)) addr of
    None  $\Rightarrow$  (None, (s\cpu-reg := new-cpu-reg,
    user-reg := new-user-reg,
    dwrite := new-dwrite,
    state-var := new-state-var,
    traps := new-traps,
    undef := new-undef)))
    | Some w  $\Rightarrow$  (Some w, (s\cpu-reg := new-cpu-reg,
    user-reg := new-user-reg,
    dwrite := new-dwrite,
    state-var := new-state-var,
    traps := new-traps,
    undef := new-undef))))
proof (cases read-instr-cache s addr = None)
case True
then have read-instr-cache s addr = None  $\wedge$ 
    read-instr-cache (s\cpu-reg := new-cpu-reg,
    user-reg := new-user-reg,
    dwrite := new-dwrite,
    state-var := new-state-var,
    traps := new-traps,
    undef := new-undef)) addr = None
using read-icache-mod-state by metis
then show ?thesis by auto
next
case False
then have  $\exists w$ . read-instr-cache s addr = Some w by auto
then have  $\exists w$ . read-instr-cache s addr = Some w  $\wedge$ 
    read-instr-cache (s\cpu-reg := new-cpu-reg,
    user-reg := new-user-reg,
    dwrite := new-dwrite,
    state-var := new-state-var,
    traps := new-traps,

```

```

    undef := new-undef)) addr = Some w
  using read-icache-mod-state by metis
  then show ?thesis by auto
qed

```

```

lemma mem-read-mod-state: fst (memory-read asi addr s) =
fst (memory-read asi addr
(s(cpu-reg := new-cpu-reg,
  user-reg := new-user-reg,
  dwrite := new-dwrite,
  state-var := new-state-var,
  traps := new-traps,
  undef := new-undef)))
apply (simp add: memory-read-def)
apply (case-tac uint asi = 1)
apply (simp add: Let-def)
apply (metis load-word-mem-mod-state option.distinct(1))
apply (case-tac uint asi = 2)
apply (simp add: Let-def)
apply (simp add: sys-reg-val-def)
apply (case-tac uint asi ∈ {8,9})
apply (simp add: Let-def)
apply (simp add: load-word-mem3-mod-state)
apply (simp add: load-word-mem-mod-state)
apply (case-tac uint asi ∈ {10,11})
apply (simp add: Let-def)
apply (simp add: load-word-mem2-mod-state)
apply (simp add: load-word-mem-mod-state)
apply (case-tac uint asi = 13)
apply (simp add: Let-def)
apply (simp add: read-icache2-mod-state)
apply (case-tac uint asi = 15)
apply (simp add: Let-def)
apply (simp add: read-dcache2-mod-state)
apply (case-tac uint asi = 25)
apply (simp add: Let-def)
apply (case-tac uint asi = 28)
apply (simp add: Let-def)
apply (simp add: mem-val-w32-mod-state)
by (simp add: Let-def)

```

```

lemma insert-trap-mem: fst (memory-read asi addr s) =
fst (memory-read asi addr (s(traps := new-traps)))

```

```

proof –
  have fst (memory-read asi addr s) =
    fst (memory-read asi addr
      (s(cpu-reg := (cpu-reg s),
        user-reg := (user-reg s),
        dwrite := (dwrite s),

```

```

      state-var := (state-var s),
      traps := new-traps,
      undef := (undef s)))
    using mem-read-mod-state by blast
    then show ?thesis by auto
qed

```

lemma *cpu-reg-mod-mem*: $\text{fst} (\text{memory-read asi addr } s) = \text{fst} (\text{memory-read asi addr } (s \setminus \text{cpu-reg} := \text{new-cpu-reg}))$

```

proof –
  have  $\text{fst} (\text{memory-read asi addr } s) =$ 
     $\text{fst} (\text{memory-read asi addr}$ 
       $(s \setminus \text{cpu-reg} := \text{new-cpu-reg},$ 
         $\text{user-reg} := (\text{user-reg } s),$ 
         $\text{dwrite} := (\text{dwrite } s),$ 
         $\text{state-var} := (\text{state-var } s),$ 
         $\text{traps} := (\text{traps } s),$ 
         $\text{undef} := (\text{undef } s)))$ 
    using mem-read-mod-state by blast
    then show ?thesis by auto
qed

```

lemma *user-reg-mod-mem*: $\text{fst} (\text{memory-read asi addr } s) = \text{fst} (\text{memory-read asi addr } (s \setminus \text{user-reg} := \text{new-user-reg}))$

```

proof –
  have  $\text{fst} (\text{memory-read asi addr } s) =$ 
     $\text{fst} (\text{memory-read asi addr}$ 
       $(s \setminus \text{cpu-reg} := (\text{cpu-reg } s),$ 
         $\text{user-reg} := \text{new-user-reg},$ 
         $\text{dwrite} := (\text{dwrite } s),$ 
         $\text{state-var} := (\text{state-var } s),$ 
         $\text{traps} := (\text{traps } s),$ 
         $\text{undef} := (\text{undef } s)))$ 
    using mem-read-mod-state by blast
    then show ?thesis by auto
qed

```

lemma *annul-mem*: $\text{fst} (\text{memory-read asi addr } s) = \text{fst} (\text{memory-read asi addr } (s \setminus \text{state-var} := \text{new-state-var}, \text{cpu-reg} := \text{new-cpu-reg}))$

```

proof –
  have  $\text{fst} (\text{memory-read asi addr } s) =$ 
     $\text{fst} (\text{memory-read asi addr}$ 
       $(s \setminus \text{cpu-reg} := \text{new-cpu-reg},$ 
         $\text{user-reg} := (\text{user-reg } s),$ 
         $\text{dwrite} := (\text{dwrite } s),$ 
         $\text{state-var} := \text{new-state-var},$ 
         $\text{traps} := (\text{traps } s),$ 

```

```

      undef := (undef s)))
    using mem-read-mod-state by blast
  then have fst (memory-read asi addr s) =
    fst (memory-read asi addr
      (s(|cpu-reg := new-cpu-reg,
        state-var := new-state-var|)))
    by auto
  then show ?thesis
    by (metis Sparc-State.sparc-state.surjective Sparc-State.sparc-state.update-convs(1)
      Sparc-State.sparc-state.update-convs(8))
qed

```

lemma *state-var-mod-mem*: $\text{fst}(\text{memory-read asi addr } s) = \text{fst}(\text{memory-read asi addr } (s(|\text{state-var} := \text{new-state-var}|)))$

```

proof –
  have fst (memory-read asi addr s) =
    fst (memory-read asi addr
      (s(|cpu-reg := (cpu-reg s),
        user-reg := (user-reg s),
        dwrite := (dwrite s),
        state-var := new-state-var,
        traps := (traps s),
        undef := (undef s)|)))
    using mem-read-mod-state by blast
  then show ?thesis by auto
qed

```

lemma *mod-state-low-equal*: $\text{low-equal } s1 \ s2 \wedge$

```

t1 = (s1(|cpu-reg := new-cpu-reg,
  user-reg := new-user-reg,
  dwrite := new-dwrite,
  state-var := new-state-var,
  traps := new-traps,
  undef := new-undef|)) ∧
t2 = (s2(|cpu-reg := new-cpu-reg,
  user-reg := new-user-reg,
  dwrite := new-dwrite,
  state-var := new-state-var,
  traps := new-traps,
  undef := new-undef|)) ⇒
low-equal t1 t2

```

```

apply (simp add: low-equal-def)
apply clarsimp
apply (simp add: mem-equal-def)
by (simp add: user-accessible-def)

```

lemma *user-reg-state-mod-low-equal*:

```

assumes a1: low-equal s1 s2 ∧
t1 = (s1(|user-reg := new-user-reg|)) ∧

```

```

t2 = (s2(|user-reg := new-user-reg|))
shows low-equal t1 t2
proof –
  have low-equal s1 s2 ∧
t1 = (s1(|cpu-reg := (cpu-reg s1),
  user-reg := new-user-reg,
  dwrite := (dwrite s1),
  state-var := (state-var s1),
  traps := (traps s1),
  undef := (undef s1)|)) ∧
t2 = (s2(|cpu-reg := (cpu-reg s2),
  user-reg := new-user-reg,
  dwrite := (dwrite s2),
  state-var := (state-var s2),
  traps := (traps s2),
  undef := (undef s2)|)) ⇒
low-equal t1 t2
  using mod-state-low-equal apply (simp add: low-equal-def)
  apply (simp add: user-accessible-def mem-equal-def)
  by clarsimp
  then show ?thesis using a1
  by clarsimp
qed

```

```

lemma mod-trap-low-equal:
assumes a1: low-equal s1 s2 ∧
t1 = (s1(|traps := new-traps|)) ∧
t2 = (s2(|traps := new-traps|))
shows low-equal t1 t2
proof –
  have low-equal s1 s2 ∧
t1 = (s1(|cpu-reg := (cpu-reg s1),
  user-reg := (user-reg s1),
  dwrite := (dwrite s1),
  state-var := (state-var s1),
  traps := new-traps,
  undef := (undef s1)|)) ∧
t2 = (s2(|cpu-reg := (cpu-reg s2),
  user-reg := (user-reg s2),
  dwrite := (dwrite s2),
  state-var := (state-var s2),
  traps := new-traps,
  undef := (undef s2)|)) ⇒
low-equal t1 t2
  using mod-state-low-equal apply (simp add: low-equal-def)
  apply (simp add: user-accessible-def mem-equal-def)
  by clarsimp
  then show ?thesis using a1
  by clarsimp

```

qed

lemma *state-var-low-equal*: $low\text{-}equal\ s1\ s2 \implies$
 $state\text{-}var\ s1 = state\text{-}var\ s2$
by (*simp add: low-equal-def*)

lemma *state-var2-low-equal*:
assumes $a1: low\text{-}equal\ s1\ s2 \wedge$
 $t1 = (s1\ (state\text{-}var := new\text{-}state\text{-}var)) \wedge$
 $t2 = (s2\ (state\text{-}var := new\text{-}state\text{-}var))$
shows $low\text{-}equal\ t1\ t2$
proof –

have $low\text{-}equal\ s1\ s2 \wedge$
 $t1 = (s1\ (cpu\text{-}reg := (cpu\text{-}reg\ s1),$
 $user\text{-}reg := (user\text{-}reg\ s1),$
 $dwrite := (dwrite\ s1),$
 $state\text{-}var := new\text{-}state\text{-}var,$
 $traps := (traps\ s1),$
 $undef := (undef\ s1))) \wedge$
 $t2 = (s2\ (cpu\text{-}reg := (cpu\text{-}reg\ s2),$
 $user\text{-}reg := (user\text{-}reg\ s2),$
 $dwrite := (dwrite\ s2),$
 $state\text{-}var := new\text{-}state\text{-}var,$
 $traps := (traps\ s2),$
 $undef := (undef\ s2))) \implies$
 $low\text{-}equal\ t1\ t2$
using *mod-state-low-equal* **apply** (*simp add: low-equal-def*)
apply (*simp add: user-accessible-def mem-equal-def*)
by *clarsimp*
then show *?thesis* **using** $a1$
by *clarsimp*
qed

lemma *traps-low-equal*: $low\text{-}equal\ s1\ s2 \implies traps\ s1 = traps\ s2$
by (*simp add: low-equal-def*)

lemma *s-low-equal*: $low\text{-}equal\ s1\ s2 \implies$
 $(get\text{-}S\ (cpu\text{-}reg\text{-}val\ PSR\ s1)) = (get\text{-}S\ (cpu\text{-}reg\text{-}val\ PSR\ s2))$
by (*simp add: low-equal-def cpu-reg-val-def*)

lemma *cpu-reg-val-low-equal*: $low\text{-}equal\ s1\ s2 \implies$
 $(cpu\text{-}reg\text{-}val\ cr\ s1) = (cpu\text{-}reg\text{-}val\ cr\ s2)$
by (*simp add: cpu-reg-val-def low-equal-def*)

lemma *get-curr-win-low-equal*: $low\text{-}equal\ s1\ s2 \implies$
 $(fst\ (fst\ (get\text{-}curr\text{-}win\ ()\ s1))) = (fst\ (fst\ (get\text{-}curr\text{-}win\ ()\ s2)))$
apply (*simp add: low-equal-def*)
apply (*simp add: get-curr-win-def cpu-reg-val-def get-CWP-def*)
by (*simp add: simplifier-gets-def*)

lemma *get-curr-win2-low-equal: low-equal s1 s2* \implies
t1 = (snd (fst (get-curr-win () s1))) \implies
t2 = (snd (fst (get-curr-win () s2))) \implies
low-equal t1 t2
apply (*simp add: low-equal-def*)
apply (*simp add: get-curr-win-def cpu-reg-val-def get-CWP-def*)
by (*auto simp add: simpler-gets-def*)

lemma *get-curr-win3-low-equal: low-equal s1 s2* \implies
(traps (snd (fst (get-curr-win () s1)))) =
(traps (snd (fst (get-curr-win () s2))))
using *low-equal-def get-curr-win2-low-equal* **by** *blast*

lemma *get-addr-low-equal: low-equal s1 s2* \implies
((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s1)))))::word3) =
((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s2)))))::word3) \wedge
((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s1)))))::word2) =
((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s2)))))::word2) \wedge
((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s1)))))::word1) =
((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s2)))))::word1)
apply (*simp add: low-equal-def*)
apply (*simp add: get-curr-win-def cpu-reg-val-def get-CWP-def*)
apply (*simp add: simpler-gets-def get-addr-def user-reg-val-def*)
apply (*simp add: Let-def*)
apply (*simp add: get-CWP-def cpu-reg-val-def get-operand2-def*)
by (*simp add: user-reg-val-def*)

lemma *get-addr2-low-equal: low-equal s1 s2* \implies
get-addr (snd instr) (snd (fst (get-curr-win () s1))) =
get-addr (snd instr) (snd (fst (get-curr-win () s2)))
apply (*simp add: low-equal-def*)
apply (*simp add: get-curr-win-def cpu-reg-val-def get-CWP-def*)
apply (*simp add: simpler-gets-def get-addr-def user-reg-val-def*)
apply (*simp add: Let-def*)
apply (*simp add: get-CWP-def cpu-reg-val-def get-operand2-def*)
by (*simp add: user-reg-val-def*)

lemma *sys-reg-low-equal: low-equal s1 s2* \implies
sys-reg s1 = sys-reg s2
by (*simp add: low-equal-def*)

lemma *user-reg-low-equal: low-equal s1 s2* \implies
user-reg s1 = user-reg s2
by (*simp add: low-equal-def*)

lemma *user-reg-val-low-equal: low-equal s1 s2* \implies
user-reg-val win ur s1 = user-reg-val win ur s2
apply (*simp add: user-reg-val-def*)

by (*simp add: user-reg-low-equal*)

lemma *get-operand2-low-equal: low-equal s1 s2 \implies*
get-operand2 op-list s1 = get-operand2 op-list s2
apply (*simp add: get-operand2-def*)
apply (*simp add: cpu-reg-val-low-equal*)
apply *auto*
apply (*simp add: user-reg-val-def*)
using *user-reg-low-equal* **by** *fastforce*

lemma *mem-val-mod-cache: mem-val-alt asi a s =*
mem-val-alt asi a (s\|cache := new-cache))
apply (*simp add: mem-val-alt-def*)
by (*simp add: Let-def*)

lemma *mem-val-w32-mod-cache: mem-val-w32 asi a s =*
mem-val-w32 asi a (s\|cache := new-cache))
apply (*simp add: mem-val-w32-def*)
apply (*simp add: Let-def*)
by (*metis mem-val-mod-cache*)

lemma *load-word-mem-mod-cache:*
load-word-mem s addr asi =
load-word-mem (s\|cache := new-cache)) addr asi
apply (*simp add: load-word-mem-def*)
apply (*case-tac virt-to-phys addr (mmu s) (mem s) = None*)
apply *auto*
by (*simp add: mem-val-w32-mod-cache*)

lemma *memory-read-8-mod-cache:*
fst (memory-read 8 addr s) = fst (memory-read 8 addr (s\|cache := new-cache))
apply (*simp add: memory-read-def*)
apply (*case-tac sys-reg s CCR AND 1 \neq 0*)
apply *auto*
apply (*simp add: option.case-eq-if load-word-mem-mod-cache*)
apply (*auto intro: load-word-mem-mod-cache*)
apply (*metis load-word-mem-mod-cache option.distinct(1)*)
by (*metis load-word-mem-mod-cache option.distinct(1)*)

lemma *memory-read-10-mod-cache:*
fst (memory-read 10 addr s) = fst (memory-read 10 addr (s\|cache := new-cache))
apply (*simp add: memory-read-def*)
apply (*case-tac sys-reg s CCR AND 1 \neq 0*)
apply *auto*
apply (*simp add: option.case-eq-if load-word-mem-mod-cache*)
apply (*auto intro: load-word-mem-mod-cache*)
apply (*metis load-word-mem-mod-cache option.distinct(1)*)
by (*metis load-word-mem-mod-cache option.distinct(1)*)

lemma *mem-equal-mod-cache*: $mem\text{-}equal\ s1\ s2\ pa \implies$
 $mem\text{-}equal\ (s1(\backslash cache := new\text{-}cache1))\ (s2(\backslash cache := new\text{-}cache2))\ pa$
by (*simp add: mem-equal-def*)

lemma *user-accessible-mod-cache*: $user\text{-}accessible\ (s(\backslash cache := new\text{-}cache))\ pa =$
 $user\text{-}accessible\ s\ pa$
by (*simp add: user-accessible-def*)

lemma *mem-equal-mod-user-reg*: $mem\text{-}equal\ s1\ s2\ pa \implies$
 $mem\text{-}equal\ (s1(\backslash user\text{-}reg := new\text{-}user\text{-}reg1))\ (s2(\backslash user\text{-}reg := user\text{-}reg2))\ pa$
by (*simp add: mem-equal-def*)

lemma *user-accessible-mod-user-reg*: $user\text{-}accessible\ (s(\backslash user\text{-}reg := new\text{-}user\text{-}reg))$
 $pa =$
 $user\text{-}accessible\ s\ pa$
by (*simp add: user-accessible-def*)

lemma *mem-equal-mod-cpu-reg*: $mem\text{-}equal\ s1\ s2\ pa \implies$
 $mem\text{-}equal\ (s1(\backslash cpu\text{-}reg := new\text{-}cpu1))\ (s2(\backslash cpu\text{-}reg := cpu\text{-}reg2))\ pa$
by (*simp add: mem-equal-def*)

lemma *user-accessible-mod-cpu-reg*: $user\text{-}accessible\ (s(\backslash cpu\text{-}reg := new\text{-}cpu\text{-}reg))\ pa$
 $=$
 $user\text{-}accessible\ s\ pa$
by (*simp add: user-accessible-def*)

lemma *mem-equal-mod-trap*: $mem\text{-}equal\ s1\ s2\ pa \implies$
 $mem\text{-}equal\ (s1(\backslash traps := new\text{-}traps1))\ (s2(\backslash traps := traps2))\ pa$
by (*simp add: mem-equal-def*)

lemma *user-accessible-mod-trap*: $user\text{-}accessible\ (s(\backslash traps := new\text{-}traps))\ pa =$
 $user\text{-}accessible\ s\ pa$
by (*simp add: user-accessible-def*)

lemma *mem-equal-annul*: $mem\text{-}equal\ s1\ s2\ pa \implies$
 $mem\text{-}equal\ (s1(\backslash state\text{-}var := new\text{-}state\text{-}var,$
 $\quad cpu\text{-}reg := new\text{-}cpu\text{-}reg))\ (s2(\backslash state\text{-}var := new\text{-}state\text{-}var2,$
 $\quad cpu\text{-}reg := new\text{-}cpu\text{-}reg2))\ pa$
by (*simp add: mem-equal-def*)

lemma *user-accessible-annul*: $user\text{-}accessible\ (s(\backslash state\text{-}var := new\text{-}state\text{-}var,$
 $\quad cpu\text{-}reg := new\text{-}cpu\text{-}reg))\ pa =$
 $user\text{-}accessible\ s\ pa$
by (*simp add: user-accessible-def*)

lemma *mem-val-alt-10-mem-equal-0*: $mem\text{-}equal\ s1\ s2\ pa \implies$
 $mem\text{-}val\text{-}alt\ 10\ (pa\ AND\ 68719476732)\ s1 = mem\text{-}val\text{-}alt\ 10\ (pa\ AND\ 68719476732)$
 $s2$
apply (*simp add: mem-val-alt-def*)

apply (*simp add: Let-def*)
apply (*simp add: mem-equal-def*)
by (*metis option.distinct(1)*)

lemma *mem-val-alt-10-mem-equal-1: mem-equal s1 s2 pa \implies*
mem-val-alt 10 ((pa AND 68719476732) + 1) s1 = mem-val-alt 10 ((pa AND
68719476732) + 1) s2
apply (*simp add: mem-val-alt-def*)
apply (*simp add: Let-def*)
apply (*simp add: mem-equal-def*)
by (*metis option.distinct(1)*)

lemma *mem-val-alt-10-mem-equal-2: mem-equal s1 s2 pa \implies*
mem-val-alt 10 ((pa AND 68719476732) + 2) s1 = mem-val-alt 10 ((pa AND
68719476732) + 2) s2
apply (*simp add: mem-val-alt-def*)
apply (*simp add: Let-def*)
apply (*simp add: mem-equal-def*)
by (*metis option.distinct(1)*)

lemma *mem-val-alt-10-mem-equal-3: mem-equal s1 s2 pa \implies*
mem-val-alt 10 ((pa AND 68719476732) + 3) s1 = mem-val-alt 10 ((pa AND
68719476732) + 3) s2
apply (*simp add: mem-val-alt-def*)
apply (*simp add: Let-def*)
apply (*simp add: mem-equal-def*)
by (*metis option.distinct(1)*)

lemma *mem-val-alt-10-mem-equal:*
assumes *a1: mem-equal s1 s2 pa*
shows *mem-val-alt 10 (pa AND 68719476732) s1 = mem-val-alt 10 (pa AND*
68719476732) s2 \wedge
mem-val-alt 10 ((pa AND 68719476732) + 1) s1 = mem-val-alt 10 ((pa AND
68719476732) + 1) s2 \wedge
mem-val-alt 10 ((pa AND 68719476732) + 2) s1 = mem-val-alt 10 ((pa AND
68719476732) + 2) s2 \wedge
mem-val-alt 10 ((pa AND 68719476732) + 3) s1 = mem-val-alt 10 ((pa AND
68719476732) + 3) s2
using *mem-val-alt-10-mem-equal-0 mem-val-alt-10-mem-equal-1*
mem-val-alt-10-mem-equal-2 mem-val-alt-10-mem-equal-3 a1
by *blast*

lemma *mem-val-w32-10-mem-equal:*
assumes *a1: mem-equal s1 s2 a*
shows *mem-val-w32 10 a s1 = mem-val-w32 10 a s2*
apply (*simp add: mem-val-w32-def*)
apply (*simp add: Let-def*)
using *mem-val-alt-10-mem-equal a1* **apply** *auto*
apply *fastforce*

apply *fastforce*
apply *fastforce*
apply *fastforce*
apply *fastforce*
apply *fastforce*
apply *fastforce*
apply *fastforce*
by *fastforce*

lemma *mem-val-alt-8-mem-equal-0*: $\text{mem-equal } s1 \ s2 \ pa \implies$
 $\text{mem-val-alt } 8 \ (pa \ \text{AND} \ 68719476732) \ s1 = \text{mem-val-alt } 8 \ (pa \ \text{AND} \ 68719476732)$
 $s2$
apply (*simp add: mem-val-alt-def*)
apply (*simp add: Let-def*)
apply (*simp add: mem-equal-def*)
by (*metis option.distinct(1)*)

lemma *mem-val-alt-8-mem-equal-1*: $\text{mem-equal } s1 \ s2 \ pa \implies$
 $\text{mem-val-alt } 8 \ ((pa \ \text{AND} \ 68719476732) + 1) \ s1 = \text{mem-val-alt } 8 \ ((pa \ \text{AND} \ 68719476732)$
 $+ 1) \ s2$
apply (*simp add: mem-val-alt-def*)
apply (*simp add: Let-def*)
apply (*simp add: mem-equal-def*)
by (*metis option.distinct(1)*)

lemma *mem-val-alt-8-mem-equal-2*: $\text{mem-equal } s1 \ s2 \ pa \implies$
 $\text{mem-val-alt } 8 \ ((pa \ \text{AND} \ 68719476732) + 2) \ s1 = \text{mem-val-alt } 8 \ ((pa \ \text{AND} \ 68719476732)$
 $+ 2) \ s2$
apply (*simp add: mem-val-alt-def*)
apply (*simp add: Let-def*)
apply (*simp add: mem-equal-def*)
by (*metis option.distinct(1)*)

lemma *mem-val-alt-8-mem-equal-3*: $\text{mem-equal } s1 \ s2 \ pa \implies$
 $\text{mem-val-alt } 8 \ ((pa \ \text{AND} \ 68719476732) + 3) \ s1 = \text{mem-val-alt } 8 \ ((pa \ \text{AND} \ 68719476732)$
 $+ 3) \ s2$
apply (*simp add: mem-val-alt-def*)
apply (*simp add: Let-def*)
apply (*simp add: mem-equal-def*)
by (*metis option.distinct(1)*)

lemma *mem-val-alt-8-mem-equal*:
assumes *a1*: $\text{mem-equal } s1 \ s2 \ pa$
shows $\text{mem-val-alt } 8 \ (pa \ \text{AND} \ 68719476732) \ s1 = \text{mem-val-alt } 8 \ (pa \ \text{AND} \ 68719476732)$
 $s2 \ \wedge$
 $\text{mem-val-alt } 8 \ ((pa \ \text{AND} \ 68719476732) + 1) \ s1 = \text{mem-val-alt } 8 \ ((pa \ \text{AND} \ 68719476732)$
 $+ 1) \ s2 \ \wedge$
 $\text{mem-val-alt } 8 \ ((pa \ \text{AND} \ 68719476732) + 2) \ s1 = \text{mem-val-alt } 8 \ ((pa \ \text{AND} \ 68719476732)$
 $+ 2) \ s2 \ \wedge$

```

    mem-val-alt 8 ((pa AND 68719476732) + 3) s1 = mem-val-alt 8 ((pa AND
68719476732) + 3) s2
using mem-val-alt-8-mem-equal-0 mem-val-alt-8-mem-equal-1
mem-val-alt-8-mem-equal-2 mem-val-alt-8-mem-equal-3 a1
by blast

```

```

lemma mem-val-w32-8-mem-equal:
assumes a1: mem-equal s1 s2 a
shows mem-val-w32 8 a s1 = mem-val-w32 8 a s2
apply (simp add: mem-val-w32-def)
apply (simp add: Let-def)
using mem-val-alt-8-mem-equal a1 apply auto
    apply fastforce
    apply fastforce
by fastforce

```

```

lemma load-word-mem-10-low-equal:
assumes a1: low-equal s1 s2
shows load-word-mem s1 address 10 = load-word-mem s2 address 10
using a1 apply (simp add: low-equal-def load-word-mem-def)
apply clarsimp
apply (case-tac virt-to-phys address (mmu s2) (mem s2) = None)
apply auto
apply (simp add: user-accessible-def)
using mem-val-w32-10-mem-equal apply blast
apply (simp add: user-accessible-def)
using mem-val-w32-10-mem-equal by blast

```

```

lemma load-word-mem-8-low-equal:
assumes a1: low-equal s1 s2
shows load-word-mem s1 address 8 = load-word-mem s2 address 8
using a1 apply (simp add: low-equal-def load-word-mem-def)
apply clarsimp
apply (case-tac virt-to-phys address (mmu s2) (mem s2) = None)
apply auto
apply (simp add: user-accessible-def)
using mem-val-w32-8-mem-equal user-accessible-8 apply fastforce
apply (simp add: user-accessible-def)
using mem-val-w32-8-mem-equal user-accessible-8 by fastforce

```

```

lemma mem-read-low-equal:
assumes a1: low-equal s1 s2  $\wedge$  asi  $\in$  {8,10}
shows fst (memory-read asi address s1) = fst (memory-read asi address s2)

```

```

proof (cases asi = 8)
  case True
  then show ?thesis using a1
  apply (simp add: low-equal-def)
  apply (simp add: memory-read-def)
  using a1 load-word-mem-8-low-equal apply auto
  apply (simp add: option.case-eq-if)
  by (simp add: option.case-eq-if)
next
  case False
  then have asi = 10 using a1 by auto
  then show ?thesis using a1
  apply (simp add: low-equal-def)
  apply (simp add: memory-read-def)
  using a1 load-word-mem-10-low-equal apply auto
  apply (simp add: option.case-eq-if)
  by (simp add: option.case-eq-if)
qed

```

```

lemma read-mem-pc-low-equal:
assumes a1: low-equal s1 s2
shows fst (memory-read 8 (cpu-reg-val PC s1) s1) =
fst (memory-read 8 (cpu-reg-val PC s2) s2)
proof –
  have f2: cpu-reg-val PC s1 = cpu-reg-val PC s2 using a1
  by (simp add: low-equal-def cpu-reg-val-def)
  then show ?thesis using a1 f2 mem-read-low-equal
  by auto
qed

```

```

lemma dcache-mod-low-equal:
assumes a1: low-equal s1 s2  $\wedge$ 
t1 = dcache-mod c v s1  $\wedge$ 
t2 = dcache-mod c v s2
shows low-equal t1 t2
using a1 apply (simp add: low-equal-def)
apply (simp add: dcache-mod-def)
apply auto
apply (simp add: user-accessible-mod-cache mem-equal-mod-cache)
by (simp add: user-accessible-mod-cache mem-equal-mod-cache)

```

```

lemma add-data-cache-low-equal:
assumes a1: low-equal s1 s2  $\wedge$ 
t1 = add-data-cache s1 address w bm  $\wedge$ 
t2 = add-data-cache s2 address w bm
shows low-equal t1 t2
using a1 apply (simp add: add-data-cache-def)
apply (case-tac bm AND 8 >> 3 = 1)
apply auto

```

```

apply (case-tac bm AND 4 >> 2 = 1)
apply auto
apply (case-tac bm AND 2 >> Suc 0 = 1)
apply auto
apply (case-tac bm AND 1 = 1)
apply auto
apply (meson dcache-mod-low-equal)
apply (meson dcache-mod-low-equal)
apply (case-tac bm AND 1 = 1)
apply auto
apply (meson dcache-mod-low-equal)
apply (meson dcache-mod-low-equal)
apply (case-tac bm AND 2 >> Suc 0 = 1)
apply auto
apply (case-tac bm AND 1 = 1)
apply auto
apply (meson dcache-mod-low-equal)
apply (meson dcache-mod-low-equal)
apply (case-tac bm AND 1 = 1)
apply auto
apply (meson dcache-mod-low-equal)
apply (meson dcache-mod-low-equal)
apply (case-tac bm AND 4 >> 2 = 1)
apply auto
apply (case-tac bm AND 2 >> Suc 0 = 1)
apply auto
apply (case-tac bm AND 1 = 1)
apply auto
apply (meson dcache-mod-low-equal)
apply (meson dcache-mod-low-equal)
apply (case-tac bm AND 1 = 1)
apply auto
apply (meson dcache-mod-low-equal)
apply (meson dcache-mod-low-equal)
apply (case-tac bm AND 2 >> Suc 0 = 1)
apply auto
apply (case-tac bm AND 1 = 1)
apply auto
apply (meson dcache-mod-low-equal)
apply (meson dcache-mod-low-equal)
by (meson dcache-mod-low-equal)

lemma mem-read2-low-equal:
assumes a1: low-equal s1 s2  $\wedge$ 
t1 = snd (memory-read (10::word8) address s1)  $\wedge$ 
t2 = snd (memory-read (10::word8) address s2)
shows low-equal t1 t2
using a1 apply (simp add: memory-read-def)
using a1 apply (auto simp add: sys-reg-low-equal mod-2-eq-odd)

```

```

using a1 apply (simp add: load-word-mem-10-low-equal)
apply (auto split: option.splits)
using add-data-cache-low-equal apply force
using add-data-cache-low-equal apply force
done

```

```

lemma mem-read-delayed-write-low-equal:
assumes a1: low-equal s1 s2  $\wedge$  get-delayed-pool s1 = []  $\wedge$  get-delayed-pool s2 = []
shows fst (memory-read 8 (cpu-reg-val PC (delayed-pool-write s1)) (delayed-pool-write
s1)) =
fst (memory-read 8 (cpu-reg-val PC (delayed-pool-write s2)) (delayed-pool-write
s2))
using a1 apply (simp add: delayed-pool-write-def)
apply (simp add: Let-def)
apply (simp add: get-delayed-write-def)
by (simp add: read-mem-pc-low-equal)

```

```

lemma global-reg-mod-low-equal:
assumes a1: low-equal s1 s2  $\wedge$ 
t1 = (global-reg-mod w n rd s1)  $\wedge$ 
t2 = (global-reg-mod w n rd s2)
shows low-equal t1 t2
using a1 apply (induction n arbitrary: s1 s2)
apply clarsimp
apply auto
apply (simp add: Let-def)
apply (simp add: user-reg-low-equal)
using user-reg-state-mod-low-equal by blast

```

```

lemma out-reg-mod-low-equal:
assumes a1: low-equal s1 s2  $\wedge$ 
t1 = (out-reg-mod w curr-win rd s1)  $\wedge$ 
t2 = (out-reg-mod w curr-win rd s2)
shows low-equal t1 t2
using a1 apply (simp add: out-reg-mod-def Let-def)
apply auto
apply (simp add: user-reg-low-equal)
using user-reg-state-mod-low-equal apply fastforce
apply (simp add: user-reg-low-equal)
using user-reg-state-mod-low-equal by blast

```

```

lemma in-reg-mod-low-equal:
assumes a1: low-equal s1 s2  $\wedge$ 
t1 = (in-reg-mod w curr-win rd s1)  $\wedge$ 
t2 = (in-reg-mod w curr-win rd s2)
shows low-equal t1 t2
using a1 apply (simp add: in-reg-mod-def Let-def)
apply auto
apply (simp add: user-reg-low-equal)

```

```

using user-reg-state-mod-low-equal apply fastforce
apply (simp add: user-reg-low-equal)
using user-reg-state-mod-low-equal by blast

lemma user-reg-mod-low-equal:
assumes a1: low-equal s1 s2  $\wedge$ 
t1 = user-reg-mod w curr-win rd s1  $\wedge$  t2 = user-reg-mod w curr-win rd s2
shows low-equal t1 t2
proof (cases rd = 0)
  case True
    then show ?thesis using a1
    by (simp add: user-reg-mod-def)
next
  case False
    then have f1: rd  $\neq$  0 by auto
    then show ?thesis
    proof (cases 0 < rd  $\wedge$  rd < 8)
      case True
        then show ?thesis using a1 f1
        apply (simp add: user-reg-mod-def)
        using global-reg-mod-low-equal by blast
      next
        case False
          then have f2:  $\neg$  (0 < rd  $\wedge$  rd < 8) by auto
          then show ?thesis
          proof (cases 7 < rd  $\wedge$  rd < 16)
            case True
              then show ?thesis using a1 f1 f2
              apply (simp add: user-reg-mod-def)
              by (auto intro: out-reg-mod-low-equal)
            next
              case False
                then have f3:  $\neg$  (7 < rd  $\wedge$  rd < 16) by auto
                then show ?thesis
                proof (cases 15 < rd  $\wedge$  rd < 24)
                  case True
                    then show ?thesis using a1 f1 f2 f3
                    apply (simp add: user-reg-mod-def)
                    apply (simp add: low-equal-def)
                    apply clarsimp
                    by (simp add: user-accessible-mod-user-reg mem-equal-mod-user-reg)
                  next
                    case False
                      then show ?thesis using a1 f1 f2 f3
                      apply (simp add: user-reg-mod-def)
                      by (auto intro: in-reg-mod-low-equal)
                qed
            qed
          qed
        qed
    qed
  qed

```

qed

lemma *virt-to-phys-low-equal*: $low\text{-}equal\ s1\ s2 \implies$
 $virt\text{-}to\text{-}phys\ addr\ (mmu\ s1)\ (mem\ s1) = virt\text{-}to\text{-}phys\ addr\ (mmu\ s2)\ (mem\ s2)$
by (*auto simp add: low-equal-def*)

lemma *write-reg-low-equal*:
assumes $a1: low\text{-}equal\ s1\ s2 \wedge$
 $t1 = (snd\ (fst\ (write\text{-}reg\ w\ curr\text{-}win\ rd\ s1))) \wedge$
 $t2 = (snd\ (fst\ (write\text{-}reg\ w\ curr\text{-}win\ rd\ s2)))$
shows $low\text{-}equal\ t1\ t2$
using $a1$ **apply** (*simp add: write-reg-def*)
apply (*simp add: simpler-modify-def*)
by (*auto intro: user-reg-mod-low-equal*)

lemma *write-cpu-low-equal*:
assumes $a1: low\text{-}equal\ s1\ s2 \wedge$
 $t1 = snd\ (fst\ (write\text{-}cpu\ w\ cr\ s1)) \wedge$
 $t2 = (snd\ (fst\ (write\text{-}cpu\ w\ cr\ s2)))$
shows $low\text{-}equal\ t1\ t2$
using $a1$
apply (*simp add: write-cpu-def simpler-modify-def*)
apply (*simp add: cpu-reg-mod-def*)
apply (*simp add: low-equal-def*)
using *user-accessible-mod-cpu-reg mem-equal-mod-cpu-reg*
by *metis*

lemma *cpu-reg-mod-low-equal*:
assumes $a1: low\text{-}equal\ s1\ s2 \wedge$
 $t1 = cpu\text{-}reg\text{-}mod\ w\ cr\ s1 \wedge$
 $t2 = cpu\text{-}reg\text{-}mod\ w\ cr\ s2$
shows $low\text{-}equal\ t1\ t2$
using $a1$
apply (*simp add: cpu-reg-mod-def*)
apply (*simp add: low-equal-def*)
using *user-accessible-mod-cpu-reg mem-equal-mod-cpu-reg*
by *metis*

lemma *load-sub2-low-equal*:
assumes $a1: low\text{-}equal\ s1\ s2 \wedge$
 $t1 = (snd\ (fst\ (load\text{-}sub2\ address\ 10\ rd\ curr\text{-}win\ w\ s1))) \wedge$
 $t2 = (snd\ (fst\ (load\text{-}sub2\ address\ 10\ rd\ curr\text{-}win\ w\ s2)))$
shows $low\text{-}equal\ t1\ t2$
proof (*cases fst (memory-read 10 (address + 4)*
(snd (fst (write-reg w curr-win (rd AND 30) s1)))) = None)
case *True*
then **have** $f0: fst\ (memory\text{-}read\ 10\ (address + 4)$
 $(snd\ (fst\ (write\text{-}reg\ w\ curr\text{-}win\ (rd\ AND\ 30)\ s1)))) = None$ **by** *auto*
have $f1: low\text{-}equal\ (snd\ (fst\ (write\text{-}reg\ w\ curr\text{-}win\ (rd\ AND\ 30)\ s1)))$

```

      (snd (fst (write-reg w curr-win (rd AND 30) s2))))
    using a1 by (auto intro: write-reg-low-equal)
  then have fst (memory-read 10 (address + 4))
    (snd (fst (write-reg w curr-win (rd AND 30) s1)))) = None ∧
    fst (memory-read 10 (address + 4))
    (snd (fst (write-reg w curr-win (rd AND 30) s1)))) =
    fst (memory-read 10 (address + 4))
    (snd (fst (write-reg w curr-win (rd AND 30) s2))))
  using f0 by (blast intro: mem-read-low-equal)
  then have fst (memory-read 10 (address + 4))
    (snd (fst (write-reg w curr-win (rd AND 30) s1)))) = None ∧
    fst (memory-read 10 (address + 4))
    (snd (fst (write-reg w curr-win (rd AND 30) s2)))) = None
  by auto
  then show ?thesis using a1
  apply (simp add: load-sub2-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
  apply (simp add: case-prod-unfold)
  apply (simp add: raise-trap-def add-trap-set-def)
  apply (simp add: simpler-modify-def)
  using f1 apply (simp add: traps-low-equal)
  using f1 by (auto intro: mod-trap-low-equal)
next
case False
  then have f2: fst (memory-read 10 (address + 4))
    (snd (fst (write-reg w curr-win (rd AND 30) s1)))) ≠ None
  by auto
  have f3: low-equal (snd (fst (write-reg w curr-win (rd AND 30) s1)))
    (snd (fst (write-reg w curr-win (rd AND 30) s2)))
  using a1 by (auto intro: write-reg-low-equal)
  then have f4: fst (memory-read 10 (address + 4))
    (snd (fst (write-reg w curr-win (rd AND 30) s1)))) =
    fst (memory-read 10 (address + 4))
    (snd (fst (write-reg w curr-win (rd AND 30) s2))))
  using f2 by (blast intro: mem-read-low-equal)
  then have fst (memory-read 10 (address + 4))
    (snd (fst (write-reg w curr-win (rd AND 30) s1)))) ≠ None ∧
    fst (memory-read 10 (address + 4))
    (snd (fst (write-reg w curr-win (rd AND 30) s2)))) ≠ None
  using f2 by auto
  then show ?thesis using a1
  apply (simp add: load-sub2-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
  apply (simp add: case-prod-unfold)
  apply clarsimp
  apply (simp add: simpler-modify-def bind-def h1-def h2-def Let-def)
  using f4 apply clarsimp
  using f3 by (auto intro: mem-read2-low-equal write-reg-low-equal)
qed

```

```

lemma load-sub3-low-equal:
assumes a1: low-equal s1 s2  $\wedge$ 
t1 = snd (fst (load-sub3 instr curr-win rd (10::word8) address s1))  $\wedge$ 
t2 = snd (fst (load-sub3 instr curr-win rd (10::word8) address s2))
shows low-equal t1 t2
proof (cases fst (memory-read 10 address s1) = None)
  case True
  then have fst (memory-read 10 address s1) = None  $\wedge$ 
    fst (memory-read 10 address s2) = None
  using a1 by (auto simp add: mem-read-low-equal)
  then show ?thesis using a1
  apply (simp add: load-sub3-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def)
  apply (simp add: case-prod-unfold)
  apply (simp add: raise-trap-def add-trap-set-def)
  apply (simp add: simpler-modify-def)
  apply (auto simp add: traps-low-equal)
  by (auto intro: mod-trap-low-equal)
next
  case False
  then have f1: fst (memory-read 10 address s1)  $\neq$  None  $\wedge$ 
    fst (memory-read 10 address s2)  $\neq$  None
  using a1 by (auto simp add: mem-read-low-equal)
  then show ?thesis
  proof (cases rd  $\neq$  0  $\wedge$ 
    (fst instr = load-store-type LD  $\vee$ 
    fst instr = load-store-type LDA  $\vee$ 
    fst instr = load-store-type LDUH  $\vee$ 
    fst instr = load-store-type LDSB  $\vee$ 
    fst instr = load-store-type LDUB  $\vee$ 
    fst instr = load-store-type LDUBA  $\vee$ 
    fst instr = load-store-type LDSH  $\vee$ 
    fst instr = load-store-type LDSHA  $\vee$ 
    fst instr = load-store-type LDUHA  $\vee$ 
    fst instr = load-store-type LDSBA))
    case True
    then show ?thesis using a1 f1
    apply (simp add: load-sub3-def)
    apply (simp add: simpler-gets-def bind-def h1-def h2-def)
    apply (simp add: case-prod-unfold)
    apply clarsimp
    apply (simp add: simpler-modify-def bind-def h1-def h2-def Let-def)
    apply (simp add: mem-read-low-equal)
    by (meson mem-read2-low-equal write-reg-low-equal)
  next
  case False
  then show ?thesis using a1 f1
  apply (simp add: load-sub3-def)

```

```

apply (simp add: simpler-gets-def bind-def h1-def h2-def)
apply (simp add: case-prod-unfold)
apply clarsimp
apply (simp add: simpler-modify-def bind-def h1-def h2-def Let-def)
apply (simp add: mem-read-low-equal)
by (meson load-sub2-low-equal mem-read2-low-equal)
qed
qed

```

```

lemma ld-asi-user:
(fst instr = load-store-type LDSB  $\vee$ 
fst instr = load-store-type LDUB  $\vee$ 
fst instr = load-store-type LDUH  $\vee$ 
fst instr = load-store-type LD  $\vee$ 
fst instr = load-store-type LDD)  $\implies$ 
ld-asi instr 0 = 10
apply (simp add: ld-asi-def)
by auto

```

```

lemma load-sub1-low-equal:
assumes a1: low-equal s1 s2  $\wedge$ 
(fst instr = load-store-type LDSB  $\vee$ 
fst instr = load-store-type LDUB  $\vee$ 
fst instr = load-store-type LDUH  $\vee$ 
fst instr = load-store-type LD  $\vee$ 
fst instr = load-store-type LDD)  $\wedge$ 
t1 = snd (fst (load-sub1 instr rd 0 s1))  $\wedge$ 
t2 = snd (fst (load-sub1 instr rd 0 s2))
shows low-equal t1 t2
proof (cases (fst instr = load-store-type LDD  $\vee$  fst instr = load-store-type LDDA)
 $\wedge$ 
((ucast (get-addr (snd instr) (snd (fst (get-curr-win ()
s1))))):word3)  $\neq$  0  $\vee$ 
(fst instr = load-store-type LD  $\vee$  fst instr = load-store-type
LDA)  $\wedge$ 
((ucast (get-addr (snd instr) (snd (fst (get-curr-win ()
s1))))):word2)  $\neq$  0  $\vee$ 
(fst instr = load-store-type LDUH  $\vee$ 
fst instr = load-store-type LDUHA  $\vee$ 
fst instr = load-store-type LDSH  $\vee$  fst instr = load-store-type
LDSHA)  $\wedge$ 
((ucast (get-addr (snd instr) (snd (fst (get-curr-win ()
s1))))):word1)  $\neq$  0)
case True
then have ((fst instr = load-store-type LDD  $\vee$  fst instr = load-store-type LDDA)
 $\wedge$ 
((ucast (get-addr (snd instr) (snd (fst (get-curr-win ()
s1))))):word3)  $\neq$  0  $\vee$ 
(fst instr = load-store-type LD  $\vee$  fst instr = load-store-type

```

$LDA) \wedge$
 $((ucast (get-addr (snd instr) (snd (fst (get-curr-win ()$
 $s1))))):word2) \neq 0 \vee$
 $(fst instr = load-store-type LDUH \vee$
 $fst instr = load-store-type LDUHA \vee$
 $fst instr = load-store-type LDSH \vee fst instr = load-store-type$
 $LDSHA) \wedge$
 $((ucast (get-addr (snd instr) (snd (fst (get-curr-win ()$
 $s1))))):word1) \neq 0) \wedge$
 $((fst instr = load-store-type LDD \vee fst instr = load-store-type LDDA) \wedge$
 $((ucast (get-addr (snd instr) (snd (fst (get-curr-win ()$
 $s2))))):word3) \neq 0 \vee$
 $(fst instr = load-store-type LD \vee fst instr = load-store-type$
 $LDA) \wedge$
 $((ucast (get-addr (snd instr) (snd (fst (get-curr-win ()$
 $s2))))):word2) \neq 0 \vee$
 $(fst instr = load-store-type LDUH \vee$
 $fst instr = load-store-type LDUHA \vee$
 $fst instr = load-store-type LDSH \vee fst instr = load-store-type$
 $LDSHA) \wedge$
 $((ucast (get-addr (snd instr) (snd (fst (get-curr-win ()$
 $s2))))):word1) \neq 0)$
by (*metis (mono-tags, lifting) assms get-addr-low-equal*)
then show *?thesis using a1*
apply (*simp add: load-sub1-def*)
apply (*simp add: simpler-gets-def bind-def h1-def h2-def Let-def*)
apply (*simp add: case-prod-unfold*)
apply (*simp add: raise-trap-def add-trap-set-def*)
apply (*simp add: simpler-modify-def*)
apply *clarsimp*
apply (*simp add: get-curr-win3-low-equal*)
by (*auto intro: get-curr-win2-low-equal mod-trap-low-equal*)
next
case *False*
then have *f1: $\neg ((fst instr = load-store-type LDD \vee fst instr = load-store-type$*
 $LDDA) \wedge$
 $((ucast (get-addr (snd instr) (snd (fst (get-curr-win ()$
 $s1))))):word3) \neq 0 \vee$
 $(fst instr = load-store-type LD \vee fst instr = load-store-type$
 $LDA) \wedge$
 $((ucast (get-addr (snd instr) (snd (fst (get-curr-win ()$
 $s1))))):word2) \neq 0 \vee$
 $(fst instr = load-store-type LDUH \vee$
 $fst instr = load-store-type LDUHA \vee$
 $fst instr = load-store-type LDSH \vee fst instr = load-store-type$
 $LDSHA) \wedge$
 $((ucast (get-addr (snd instr) (snd (fst (get-curr-win ()$
 $s1))))):word1) \neq 0) \wedge$
 $\neg ((fst instr = load-store-type LDD \vee fst instr = load-store-type LDDA)$

\wedge
 $((\text{ucast } (\text{get-addr } (\text{snd } \text{instr})) (\text{snd } (\text{fst } (\text{get-curr-win } () s2))))::\text{word3}) \neq 0 \vee$
 $(\text{fst } \text{instr} = \text{load-store-type } \text{LD} \vee \text{fst } \text{instr} = \text{load-store-type } \text{LDA}) \wedge$
 $((\text{ucast } (\text{get-addr } (\text{snd } \text{instr})) (\text{snd } (\text{fst } (\text{get-curr-win } () s2))))::\text{word2}) \neq 0 \vee$
 $(\text{fst } \text{instr} = \text{load-store-type } \text{LDUH} \vee$
 $\text{fst } \text{instr} = \text{load-store-type } \text{LDUHA} \vee$
 $\text{fst } \text{instr} = \text{load-store-type } \text{LDSH} \vee \text{fst } \text{instr} = \text{load-store-type } \text{LDSHA}) \wedge$
 $((\text{ucast } (\text{get-addr } (\text{snd } \text{instr})) (\text{snd } (\text{fst } (\text{get-curr-win } () s2))))::\text{word1}) \neq 0)$
by (*metis* *assms* *get-addr-low-equal*)
show *?thesis*
proof –
have *low-equal* *s1 s2* \implies
 $\text{low-equal } (\text{snd } (\text{fst } (\text{get-curr-win } () s1)))$
 $(\text{snd } (\text{fst } (\text{get-curr-win } () s2)))$
using *get-curr-win2-low-equal* **by** *auto*
then have *f2*: *low-equal* *s1 s2* \implies
 $\text{low-equal } (\text{snd } (\text{fst } (\text{load-sub3 } \text{instr } (\text{fst } (\text{fst } (\text{get-curr-win } () s2)))) \text{rd } 10$
 $(\text{get-addr } (\text{snd } \text{instr}) (\text{snd } (\text{fst } (\text{get-curr-win } () s2))))$
 $(\text{snd } (\text{fst } (\text{get-curr-win } () s1))))))$
 $(\text{snd } (\text{fst } (\text{load-sub3 } \text{instr } (\text{fst } (\text{fst } (\text{get-curr-win } () s2)))) \text{rd } 10$
 $(\text{get-addr } (\text{snd } \text{instr}) (\text{snd } (\text{fst } (\text{get-curr-win } () s2))))$
 $(\text{snd } (\text{fst } (\text{get-curr-win } () s2))))))$
using *load-sub3-low-equal* **by** *blast*
show *?thesis* **using** *a1*
unfolding *load-sub1-def* *simpler-gets-def* *bind-def* *h1-def* *h2-def* *Let-def* *case-prod-unfold*
using *f1* *f2* **apply** *clarsimp*
by (*simp* *add*: *get-addr2-low-equal* *get-curr-win-low-equal* *ld-asi-user*)
qed
qed

lemma *load-instr-low-equal*:
assumes *a1*: *low-equal* *s1 s2* \wedge
 $(\text{fst } \text{instr} = \text{load-store-type } \text{LDSB} \vee$
 $\text{fst } \text{instr} = \text{load-store-type } \text{LDUB} \vee$
 $\text{fst } \text{instr} = \text{load-store-type } \text{LDUBA} \vee$
 $\text{fst } \text{instr} = \text{load-store-type } \text{LDUH} \vee$
 $\text{fst } \text{instr} = \text{load-store-type } \text{LD} \vee$
 $\text{fst } \text{instr} = \text{load-store-type } \text{LDA} \vee$
 $\text{fst } \text{instr} = \text{load-store-type } \text{LDD}) \wedge$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s1))::\text{word1}) = 0 \wedge$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s2))::\text{word1}) = 0 \wedge$
 $t1 = \text{snd } (\text{fst } (\text{load-instr } \text{instr } s1)) \wedge t2 = \text{snd } (\text{fst } (\text{load-instr } \text{instr } s2))$
shows *low-equal* *t1 t2*
proof –

```

have get-S (cpu-reg-val PSR s1) = 0  $\wedge$  get-S (cpu-reg-val PSR s2) = 0
  using a1 by (simp add: ucast-id)
then show ?thesis using a1
apply (simp add: load-instr-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def)
apply (simp add: Let-def)
apply clarsimp
apply (simp add: raise-trap-def add-trap-set-def)
apply (simp add: simpler-modify-def)
apply (simp add: traps-low-equal)
by (auto intro: mod-trap-low-equal load-sub1-low-equal)
qed

```

```

lemma st-data0-low-equal: low-equal s1 s2  $\implies$ 
st-data0 instr curr-win rd addr s1 = st-data0 instr curr-win rd addr s2
apply (simp add: st-data0-def)
by (simp add: user-reg-val-def low-equal-def)

```

```

lemma store-word-mem-low-equal-none: low-equal s1 s2  $\implies$ 
store-word-mem (add-data-cache s1 addr data bm) addr data bm 10 = None  $\implies$ 
store-word-mem (add-data-cache s2 addr data bm) addr data bm 10 = None
apply (simp add: store-word-mem-def)

```

proof –

```

  assume a1: low-equal s1 s2
  assume a2: (case virt-to-phys addr (mmu (add-data-cache s1 addr data bm))
(mem (add-data-cache s1 addr data bm)) of None  $\Rightarrow$  None | Some pair  $\Rightarrow$  if
mmu-writable (get-acc-flag (snd pair)) 10 then Some (mem-mod-w32 10 (fst pair))
bm data (add-data-cache s1 addr data bm)) else None) = None
  have f3: (if mmu-writable (get-acc-flag (snd (v1-2 (virt-to-phys addr (mmu
(add-data-cache s2 addr data bm)) (mem (add-data-cache s2 addr data bm)))))) 10
then Some (mem-mod-w32 10 (fst (v1-2 (virt-to-phys addr (mmu (add-data-cache
s2 addr data bm)) (mem (add-data-cache s2 addr data bm)))))) bm data (add-data-cache
s2 addr data bm)) else None) = (case Some (v1-2 (virt-to-phys addr (mmu (add-data-cache
s2 addr data bm)) (mem (add-data-cache s2 addr data bm)))) of None  $\Rightarrow$  if mmu-writable
(get-acc-flag (snd (v1-2 (virt-to-phys addr (mmu (add-data-cache s2 addr data
bm)) (mem (add-data-cache s2 addr data bm)))))) 10 then Some (mem-mod-w32
10 (fst (v1-2 (virt-to-phys addr (mmu (add-data-cache s2 addr data bm)) (mem
(add-data-cache s2 addr data bm)))))) bm data (add-data-cache s1 addr data bm))
else None | Some p  $\Rightarrow$  if mmu-writable (get-acc-flag (snd p)) 10 then Some (mem-mod-w32
10 (fst p)) bm data (add-data-cache s2 addr data bm)) else None)
  by auto
  obtain pp :: (word36  $\times$  word8) option  $\Rightarrow$  word36  $\times$  word8 where
f4: virt-to-phys addr (mmu (add-data-cache s1 addr data bm)) (mem (add-data-cache
s1 addr data bm)) = None  $\vee$  virt-to-phys addr (mmu (add-data-cache s1 addr
data bm)) (mem (add-data-cache s1 addr data bm)) = Some (pp (virt-to-phys
addr (mmu (add-data-cache s1 addr data bm)) (mem (add-data-cache s1 addr data
bm))))
  by (metis (no-types) option.exhaust)
  have f5: virt-to-phys addr (mmu (add-data-cache s1 addr data bm)) (mem (add-data-cache

```

$s1 \text{ addr data bm}) = \text{virt-to-phys addr (mmu (add-data-cache s2 addr data bm))}$
 $(\text{mem (add-data-cache s2 addr data bm)})$
using $a1$ **by** $(\text{meson add-data-cache-low-equal virt-to-phys-low-equal})$
{ assume $\text{Some (mem-mod-w32 10 (fst (pp (virt-to-phys addr (mmu (add-data-cache s2 addr data bm)) (mem (add-data-cache s2 addr data bm)))))) bm data (add-data-cache s1 addr data bm))} \neq (\text{case Some (pp (virt-to-phys addr (mmu (add-data-cache s2 addr data bm)) (mem (add-data-cache s2 addr data bm)))) of None} \Rightarrow \text{None} \mid \text{Some } p \Rightarrow \text{if mmu-writable (get-acc-flag (snd p)) 10 then Some (mem-mod-w32 10 (fst p) bm data (add-data-cache s1 addr data bm)) else None})$
then have $\text{None} = (\text{if mmu-writable (get-acc-flag (snd (pp (virt-to-phys addr (mmu (add-data-cache s2 addr data bm)) (mem (add-data-cache s2 addr data bm)))))) 10 then Some (mem-mod-w32 10 (fst (pp (virt-to-phys addr (mmu (add-data-cache s2 addr data bm)) (mem (add-data-cache s2 addr data bm)))))) bm data (add-data-cache s2 addr data bm)) else None})$
by fastforce
moreover
{ assume $(\text{if mmu-writable (get-acc-flag (snd (pp (virt-to-phys addr (mmu (add-data-cache s2 addr data bm)) (mem (add-data-cache s2 addr data bm)))))) 10 then Some (mem-mod-w32 10 (fst (pp (virt-to-phys addr (mmu (add-data-cache s2 addr data bm)) (mem (add-data-cache s2 addr data bm)))))) bm data (add-data-cache s2 addr data bm)) else None} \neq (\text{case virt-to-phys addr (mmu (add-data-cache s2 addr data bm)) (mem (add-data-cache s2 addr data bm)) of None} \Rightarrow \text{None} \mid \text{Some } p \Rightarrow \text{if mmu-writable (get-acc-flag (snd p)) 10 then Some (mem-mod-w32 10 (fst p) bm data (add-data-cache s2 addr data bm)) else None})$
then have $(\text{case Some (pp (virt-to-phys addr (mmu (add-data-cache s2 addr data bm)) (mem (add-data-cache s2 addr data bm)))) of None} \Rightarrow \text{if mmu-writable (get-acc-flag (snd (pp (virt-to-phys addr (mmu (add-data-cache s2 addr data bm)) (mem (add-data-cache s2 addr data bm)))))) 10 then Some (mem-mod-w32 10 (fst (pp (virt-to-phys addr (mmu (add-data-cache s2 addr data bm)) (mem (add-data-cache s2 addr data bm)))))) bm data (add-data-cache s1 addr data bm)) else None} \mid \text{Some } p \Rightarrow \text{if mmu-writable (get-acc-flag (snd p)) 10 then Some (mem-mod-w32 10 (fst p) bm data (add-data-cache s2 addr data bm)) else None} \neq (\text{case virt-to-phys addr (mmu (add-data-cache s2 addr data bm)) (mem (add-data-cache s2 addr data bm)) of None} \Rightarrow \text{None} \mid \text{Some } p \Rightarrow \text{if mmu-writable (get-acc-flag (snd p)) 10 then Some (mem-mod-w32 10 (fst p) bm data (add-data-cache s2 addr data bm)) else None})$
using $f3$ **by simp**
then have $\text{Some (pp (virt-to-phys addr (mmu (add-data-cache s2 addr data bm)) (mem (add-data-cache s2 addr data bm))))} \neq \text{virt-to-phys addr (mmu (add-data-cache s2 addr data bm)) (mem (add-data-cache s2 addr data bm))} \vee (\text{if mmu-writable (get-acc-flag (snd (pp (virt-to-phys addr (mmu (add-data-cache s2 addr data bm)) (mem (add-data-cache s2 addr data bm)))))) 10 then Some (mem-mod-w32 10 (fst (pp (virt-to-phys addr (mmu (add-data-cache s2 addr data bm)) (mem (add-data-cache s2 addr data bm)))))) bm data (add-data-cache s1 addr data bm)) else None} \neq \text{None})$
proof –
have $(\text{case virt-to-phys addr (mmu (add-data-cache s2 addr data bm)) (mem (add-data-cache s2 addr data bm)) of None} \Rightarrow \text{if mmu-writable (get-acc-flag (snd (pp (virt-to-phys addr (mmu (add-data-cache s2 addr data bm)) (mem (add-data-cache s2 addr data bm)))))) 10 then Some (mem-mod-w32 10 (fst (pp (virt-to-phys addr (mmu (add-data-cache s2 addr data bm)) (mem (add-data-cache s2 addr data bm)))))) bm data (add-data-cache s2 addr data bm)) else None} \neq \text{None})$

```

(mmu (add-data-cache s2 addr data bm)) (mem (add-data-cache s2 addr data bm))))
bm data (add-data-cache s1 addr data bm)) else None | Some p ⇒ if mmu-writable
(get-acc-flag (snd p)) 10 then Some (mem-mod-w32 10 (fst p) bm data (add-data-cache
s2 addr data bm)) else None) = (case virt-to-phys addr (mmu (add-data-cache
s2 addr data bm)) (mem (add-data-cache s2 addr data bm)) of None ⇒ None |
Some p ⇒ if mmu-writable (get-acc-flag (snd p)) 10 then Some (mem-mod-w32
10 (fst p) bm data (add-data-cache s2 addr data bm)) else None) ∨ Some (pp
(virt-to-phys addr (mmu (add-data-cache s2 addr data bm)) (mem (add-data-cache
s2 addr data bm)))) ≠ virt-to-phys addr (mmu (add-data-cache s2 addr data bm))
(mem (add-data-cache s2 addr data bm)) ∨ (if mmu-writable (get-acc-flag (snd (pp
(virt-to-phys addr (mmu (add-data-cache s2 addr data bm)) (mem (add-data-cache
s2 addr data bm)))))) 10 then Some (mem-mod-w32 10 (fst (pp (virt-to-phys addr
(mmu (add-data-cache s2 addr data bm)) (mem (add-data-cache s2 addr data bm))))))
bm data (add-data-cache s1 addr data bm)) else None) ≠ None
  by simp
  then show ?thesis
    using ⟨(case Some (pp (virt-to-phys addr (mmu (add-data-cache s2 addr
data bm)) (mem (add-data-cache s2 addr data bm)))) of None ⇒ if mmu-writable
(get-acc-flag (snd (pp (virt-to-phys addr (mmu (add-data-cache s2 addr data bm))
(mem (add-data-cache s2 addr data bm)))))) 10 then Some (mem-mod-w32 10 (fst
(pp (virt-to-phys addr (mmu (add-data-cache s2 addr data bm)) (mem (add-data-cache
s2 addr data bm)))))) bm data (add-data-cache s1 addr data bm)) else None | Some
p ⇒ if mmu-writable (get-acc-flag (snd p)) 10 then Some (mem-mod-w32 10 (fst
p) bm data (add-data-cache s2 addr data bm)) else None) ≠ (case virt-to-phys addr
(mmu (add-data-cache s2 addr data bm)) (mem (add-data-cache s2 addr data bm))
of None ⇒ None | Some p ⇒ if mmu-writable (get-acc-flag (snd p)) 10 then Some
(mem-mod-w32 10 (fst p) bm data (add-data-cache s2 addr data bm)) else None)⟩
  by force
  qed
  moreover
    { assume Some (pp (virt-to-phys addr (mmu (add-data-cache s2 addr
data bm)) (mem (add-data-cache s2 addr data bm)))) ≠ virt-to-phys addr (mmu
(add-data-cache s2 addr data bm)) (mem (add-data-cache s2 addr data bm))
      then have virt-to-phys addr (mmu (add-data-cache s1 addr data bm))
(mem (add-data-cache s1 addr data bm)) ≠ Some (pp (virt-to-phys addr (mmu
(add-data-cache s1 addr data bm)) (mem (add-data-cache s1 addr data bm))))
        using f5 by simp }
    ultimately have virt-to-phys addr (mmu (add-data-cache s1 addr data bm))
(mem (add-data-cache s1 addr data bm)) ≠ Some (pp (virt-to-phys addr (mmu
(add-data-cache s2 addr data bm)) (mem (add-data-cache s2 addr data bm)))) ∨
virt-to-phys addr (mmu (add-data-cache s1 addr data bm)) (mem (add-data-cache
s1 addr data bm)) ≠ Some (pp (virt-to-phys addr (mmu (add-data-cache s1 addr
data bm)) (mem (add-data-cache s1 addr data bm))))
      using a2 by force }
    ultimately have virt-to-phys addr (mmu (add-data-cache s1 addr data bm))
(mem (add-data-cache s1 addr data bm)) = Some (pp (virt-to-phys addr (mmu
(add-data-cache s1 addr data bm)) (mem (add-data-cache s1 addr data bm)))) ∧
virt-to-phys addr (mmu (add-data-cache s1 addr data bm)) (mem (add-data-cache
s1 addr data bm)) = Some (pp (virt-to-phys addr (mmu (add-data-cache s2 addr

```

$data\ bm))\ (mem\ (add-data-cache\ s2\ addr\ data\ bm)))\ \longrightarrow\ (case\ virt-to-phys\ addr\ (mmu\ (add-data-cache\ s2\ addr\ data\ bm))\ (mem\ (add-data-cache\ s2\ addr\ data\ bm))\ of\ None\ \Rightarrow\ None\ |\ Some\ p\ \Rightarrow\ if\ mmu-writable\ (get-acc-flag\ (snd\ p))\ 10\ then\ Some\ (mem-mod-w32\ 10\ (fst\ p)\ bm\ data\ (add-data-cache\ s2\ addr\ data\ bm))\ else\ None)\ =\ None$
by fastforce }
then have $virt-to-phys\ addr\ (mmu\ (add-data-cache\ s1\ addr\ data\ bm))\ (mem\ (add-data-cache\ s1\ addr\ data\ bm))\ =\ Some\ (pp\ (virt-to-phys\ addr\ (mmu\ (add-data-cache\ s1\ addr\ data\ bm))\ (mem\ (add-data-cache\ s1\ addr\ data\ bm))))\ \wedge\ virt-to-phys\ addr\ (mmu\ (add-data-cache\ s1\ addr\ data\ bm))\ (mem\ (add-data-cache\ s1\ addr\ data\ bm))\ =\ Some\ (pp\ (virt-to-phys\ addr\ (mmu\ (add-data-cache\ s2\ addr\ data\ bm))\ (mem\ (add-data-cache\ s2\ addr\ data\ bm))))\ \longrightarrow\ (case\ virt-to-phys\ addr\ (mmu\ (add-data-cache\ s2\ addr\ data\ bm))\ (mem\ (add-data-cache\ s2\ addr\ data\ bm))\ of\ None\ \Rightarrow\ None\ |\ Some\ p\ \Rightarrow\ if\ mmu-writable\ (get-acc-flag\ (snd\ p))\ 10\ then\ Some\ (mem-mod-w32\ 10\ (fst\ p)\ bm\ data\ (add-data-cache\ s2\ addr\ data\ bm))\ else\ None)\ =\ None$
using a2 by force
then show $(case\ virt-to-phys\ addr\ (mmu\ (add-data-cache\ s2\ addr\ data\ bm))\ (mem\ (add-data-cache\ s2\ addr\ data\ bm))\ of\ None\ \Rightarrow\ None\ |\ Some\ p\ \Rightarrow\ if\ mmu-writable\ (get-acc-flag\ (snd\ p))\ 10\ then\ Some\ (mem-mod-w32\ 10\ (fst\ p)\ bm\ data\ (add-data-cache\ s2\ addr\ data\ bm))\ else\ None)\ =\ None$
using f5 f4 by force
qed

lemma memory-write-asi-low-equal-none: $low-equal\ s1\ s2\ \Longrightarrow\ memory-write-asi\ 10\ addr\ bm\ data\ s1\ =\ None\ \Longrightarrow\ memory-write-asi\ 10\ addr\ bm\ data\ s2\ =\ None$
apply (*simp add: memory-write-asi-def*)
by (*simp add: store-word-mem-low-equal-none*)

lemma memory-write-low-equal-none: $low-equal\ s1\ s2\ \Longrightarrow\ memory-write\ 10\ addr\ bm\ data\ s1\ =\ None\ \Longrightarrow\ memory-write\ 10\ addr\ bm\ data\ s2\ =\ None$
apply (*simp add: memory-write-def*)
by (*metis map-option-case memory-write-asi-low-equal-none option.map-disc-iff*)

lemma memory-write-low-equal-none2: $low-equal\ s1\ s2\ \Longrightarrow\ memory-write\ 10\ addr\ bm\ data\ s2\ =\ None\ \Longrightarrow\ memory-write\ 10\ addr\ bm\ data\ s1\ =\ None$
apply (*simp add: memory-write-def*)
by (*metis low-equal-com memory-write-def memory-write-low-equal-none*)

lemma mem-context-val-9-unchanged:
 $mem-context-val\ 9\ addr1\ (mem\ s1)\ =\ mem-context-val\ 9\ addr1\ ((mem\ s1)(10\ :=\ (mem\ s1\ 10)(addr\ \mapsto\ val),\ 11\ :=\ (mem\ s1\ 11)(addr\ :=\ None)))$
apply (*simp add: mem-context-val-def*)
by (*simp add: Let-def*)

lemma *mem-context-val-w32-9-unchanged*:
mem-context-val-w32 9 addr1 (mem s1) =
mem-context-val-w32 9 addr1
((mem s1)(10 := (mem s1 10)(addr ↦ val), 11 := (mem s1 11)(addr
:= None)))
apply (*simp add: mem-context-val-w32-def*)
apply (*simp add: Let-def*)
by (*metis mem-context-val-9-unchanged*)

lemma *ptd-lookup-unchanged-4*:
ptd-lookup va ptp (mem s1) 4 =
ptd-lookup va ptp ((mem s1)(10 := (mem s1 10)(addr ↦ val),
11 := (mem s1 11)(addr := None))) 4
by *auto*

lemma *ptd-lookup-unchanged-3*:
ptd-lookup va ptp (mem s1) 3 =
ptd-lookup va ptp ((mem s1)(10 := (mem s1 10)(addr ↦ val),
11 := (mem s1 11)(addr := None))) 3
proof (*cases mem-context-val-w32 9 ((ucast (ptp OR ((ucast ((ucast (va >>*
12))::word6))::word32)))::word36 (mem s1) = None)
case *True*
then have *mem-context-val-w32 9 ((ucast (ptp OR ((ucast ((ucast (va >>*
12))::word6))::word32)))::word36 (mem s1) = None ∧
mem-context-val-w32 9 ((ucast (ptp OR ((ucast ((ucast (va >> 12))::word6))::word32)))::word36
((mem s1)(10 := (mem s1 10)(addr ↦ val), 11 := (mem s1 11)(addr :=
None))) = None
using *mem-context-val-w32-9-unchanged by metis*
then show *?thesis*
by *auto*
next
case *False*
then have *mem-context-val-w32 9*
((ucast (ptp OR ((ucast ((ucast (va >> 12))::word6))::word32)))::word36 (mem
s1) ≠ None ∧
mem-context-val-w32 9 ((ucast (ptp OR ((ucast ((ucast (va >> 12))::word6))::word32)))::word36
((mem s1)(10 := (mem s1 10)(addr ↦ val), 11 := (mem s1 11)(addr :=
None))) ≠ None
using *mem-context-val-w32-9-unchanged by metis*
then have *mem-context-val-w32 9*
((ucast (ptp OR ((ucast ((ucast (va >> 12))::word6))::word32)))::word36 (mem
s1) ≠ None ∧
mem-context-val-w32 9 ((ucast (ptp OR ((ucast ((ucast (va >> 12))::word6))::word32)))::word36
((mem s1)(10 := (mem s1 10)(addr ↦ val), 11 := (mem s1 11)(addr :=
None))) ≠ None ∧
(∀ y. (mem-context-val-w32 9
((ucast (ptp OR ((ucast ((ucast (va >> 12))::word6))::word32)))::word36 (mem

```

s1) = Some y) →
  (mem-context-val-w32 9
    ((ucast (ptp OR ((ucast ((ucast (va >> 12))::word6))::word32)))::word36)
    ((mem s1)(10 := (mem s1 10)(addr ↦ val), 11 := (mem s1 11)(addr :=
None)))= Some y))
  using mem-context-val-w32-9-unchanged by metis
  then show ?thesis
  apply auto
  by (simp add: Let-def)
qed

lemma ptd-lookup-unchanged-2:
  ptd-lookup va ptp (mem s1) 2 =
  ptd-lookup va ptp ((mem s1)(10 := (mem s1 10)(addr ↦ val),
    11 := (mem s1 11)(addr := None))) 2
proof (cases mem-context-val-w32 9 ((ucast (ptp OR ((ucast ((ucast (va >>
18))::word6))::word32)))::word36) (mem s1) = None)
  case True
  then have mem-context-val-w32 9 ((ucast (ptp OR ((ucast ((ucast (va >>
18))::word6))::word32)))::word36) (mem s1) = None ∧
    mem-context-val-w32 9 ((ucast (ptp OR ((ucast ((ucast (va >> 18))::word6))::word32)))::word36)
      ((mem s1)(10 := (mem s1 10)(addr ↦ val), 11 := (mem s1 11)(addr :=
None))) = None
  using mem-context-val-w32-9-unchanged by metis
  then show ?thesis
  by auto
next
  case False
  then have mem-context-val-w32 9
    ((ucast (ptp OR ((ucast ((ucast (va >> 18))::word6))::word32)))::word36) (mem
s1) ≠ None ∧
    mem-context-val-w32 9 ((ucast (ptp OR ((ucast ((ucast (va >> 18))::word6))::word32)))::word36)
      ((mem s1)(10 := (mem s1 10)(addr ↦ val), 11 := (mem s1 11)(addr :=
None))) ≠ None
  using mem-context-val-w32-9-unchanged by metis
  then have mem-context-val-w32 9
    ((ucast (ptp OR ((ucast ((ucast (va >> 18))::word6))::word32)))::word36) (mem
s1) ≠ None ∧
    mem-context-val-w32 9 ((ucast (ptp OR ((ucast ((ucast (va >> 18))::word6))::word32)))::word36)
      ((mem s1)(10 := (mem s1 10)(addr ↦ val), 11 := (mem s1 11)(addr :=
None))) ≠ None ∧
    (∀ y. (mem-context-val-w32 9
      ((ucast (ptp OR ((ucast ((ucast (va >> 18))::word6))::word32)))::word36) (mem
s1) = Some y) →
      (mem-context-val-w32 9
        ((ucast (ptp OR ((ucast ((ucast (va >> 18))::word6))::word32)))::word36)
          ((mem s1)(10 := (mem s1 10)(addr ↦ val), 11 := (mem s1 11)(addr :=

```

```

None)))= Some y))
  using mem-context-val-w32-9-unchanged by metis
  then show ?thesis
  apply auto
  using ptd-lookup-unchanged-3
  unfolding Let-def
  by auto
qed

lemma ptd-lookup-unchanged-1:
  ptd-lookup va ptp (mem s1) 1 =
  ptd-lookup va ptp ((mem s1)(10 := (mem s1 10) (addr ↦ val),
    11 := (mem s1 11)(addr := None))) 1
proof (cases mem-context-val-w32 9 ((ucast (ptp OR ((ucast ((ucast (va >>
24))::word8))::word32)))::word36) (mem s1) = None)
  case True
  then have mem-context-val-w32 9 ((ucast (ptp OR ((ucast ((ucast (va >>
24))::word8))::word32)))::word36) (mem s1) = None ∧
    mem-context-val-w32 9 ((ucast (ptp OR ((ucast ((ucast (va >> 24))::word8))::word32)))::word36)
      ((mem s1)(10 := (mem s1 10)(addr ↦ val), 11 := (mem s1 11)(addr :=
None))) = None
  using mem-context-val-w32-9-unchanged by metis
  then show ?thesis
  by auto
next
  case False
  then have mem-context-val-w32 9
    ((ucast (ptp OR ((ucast ((ucast (va >> 24))::word8))::word32)))::word36) (mem
s1) ≠ None ∧
    mem-context-val-w32 9 ((ucast (ptp OR ((ucast ((ucast (va >> 24))::word8))::word32)))::word36)
      ((mem s1)(10 := (mem s1 10)(addr ↦ val), 11 := (mem s1 11)(addr :=
None))) ≠ None
  using mem-context-val-w32-9-unchanged by metis
  then have mem-context-val-w32 9
    ((ucast (ptp OR ((ucast ((ucast (va >> 24))::word8))::word32)))::word36) (mem
s1) ≠ None ∧
    mem-context-val-w32 9 ((ucast (ptp OR ((ucast ((ucast (va >> 24))::word8))::word32)))::word36)
      ((mem s1)(10 := (mem s1 10)(addr ↦ val), 11 := (mem s1 11)(addr :=
None))) ≠ None ∧
    (∀ y. (mem-context-val-w32 9
      ((ucast (ptp OR ((ucast ((ucast (va >> 24))::word8))::word32)))::word36) (mem
s1) = Some y) →
      (mem-context-val-w32 9
        ((ucast (ptp OR ((ucast ((ucast (va >> 24))::word8))::word32)))::word36)
          ((mem s1)(10 := (mem s1 10)(addr ↦ val), 11 := (mem s1 11)(addr :=
None)))= Some y))
  using mem-context-val-w32-9-unchanged by metis

```

```

then show ?thesis
apply auto
using ptd-lookup-unchanged-2
unfolding Let-def
proof –
  fix y :: word32
  have (y AND 3 ≠ 0 ∨ y AND 3 = 0 ∨ (y AND 3 ≠ 1 ∨ ptd-lookup va (y
AND 4294967292) (mem s1) (Suc 0 + 1) = None) ∧ (y AND 3 = 1 ∨ y AND 3
≠ 2 ∨ None = Some ((ucast (ucast (y >> 8)::word24) << 12) OR (ucast (ucast
va::word12)::word36), ucast y::word8))) ∧ (y AND 3 = 0 ∨ (y AND 3 ≠ 1 ∨ (y
AND 3 ≠ 0 ∨ ptd-lookup va (y AND 4294967292) ((mem s1) (10 := (mem s1
10)(addr ↦ val), 11 := (mem s1 11)(addr := None))) (Suc 0 + 1) = None) ∧
(y AND 3 = 0 ∨ (y AND 3 ≠ 1 ∨ ptd-lookup va (y AND 4294967292) (mem s1)
(Suc 0 + 1) = ptd-lookup va (y AND 4294967292) ((mem s1) (10 := (mem s1
10)(addr ↦ val), 11 := (mem s1 11)(addr := None))) (Suc 0 + 1)) ∧ (y AND
3 = 1 ∨ (y AND 3 ≠ 2 ∨ ptd-lookup va (y AND 4294967292) ((mem s1) (10
:= (mem s1 10)(addr ↦ val), 11 := (mem s1 11)(addr := None))) (Suc 0 + 1)
= Some ((ucast (ucast (y >> 8)::word24) << 12) OR ucast (ucast va::word12),
ucast y)) ∧ (y AND 3 = 2 ∨ ptd-lookup va (y AND 4294967292) ((mem s1) (10
:= (mem s1 10)(addr ↦ val), 11 := (mem s1 11)(addr := None))) (Suc 0 + 1)
= None)))) ∧ (y AND 3 = 1 ∨ (y AND 3 ≠ 2 ∨ (y AND 3 ≠ 0 ∨ None = Some
((ucast (ucast (y >> 8)::word24) << 12) OR (ucast (ucast va::word12)::word36),
ucast y::word8))) ∧ (y AND 3 = 0 ∨ (y AND 3 ≠ 1 ∨ ptd-lookup va (y AND
4294967292) (mem s1) (Suc 0 + 1) = Some ((ucast (ucast (y >> 8)::word24)
<< 12) OR ucast (ucast va::word12), ucast y)) ∧ (y AND 3 = 1 ∨ y AND 3 =
2 ∨ None = Some ((ucast (ucast (y >> 8)::word24) << 12) OR (ucast (ucast
va::word12)::word36), ucast y::word8)))) ∧ (y AND 3 = 2 ∨ y AND 3 = 0 ∨
(y AND 3 ≠ 1 ∨ ptd-lookup va (y AND 4294967292) (mem s1) (Suc 0 + 1) =
None) ∧ (y AND 3 = 1 ∨ y AND 3 ≠ 2 ∨ None = Some ((ucast (ucast (y >>
8)::word24) << 12) OR (ucast (ucast va::word12)::word36), ucast y::word8))))
∨ (∀ w. mem s1 w = ((mem s1) (10 := (mem s1 10)(addr ↦ val), 11 := (mem
s1 11)(addr := None))) w)
  by (metis (no-types) One-nat-def Suc-1 Suc-eq-plus1 ptd-lookup-unchanged-2)
  then show (if y AND 3 = 0 then None else if y AND 3 = 1 then ptd-lookup
va (y AND 4294967292) (mem s1) (Suc 0 + 1) else if y AND 3 = 2 then Some
((ucast (ucast (y >> 8)::word24) << 12) OR ucast (ucast va::word12), ucast y)
else None) = (if y AND 3 = 0 then None else if y AND 3 = 1 then ptd-lookup va
(y AND 4294967292) ((mem s1) (10 := (mem s1 10)(addr ↦ val), 11 := (mem
s1 11)(addr := None))) (Suc 0 + 1) else if y AND 3 = 2 then Some ((ucast (ucast
(y >> 8)::word24) << 12) OR ucast (ucast va::word12), ucast y) else None)
  proof –
    have f1: 2 = Suc 0 + 1
    by (metis One-nat-def Suc-1 Suc-eq-plus1)
    { assume y AND 3 = 1
      moreover
        { assume y AND 3 = 1 ∧ (if y AND 3 = 0 then None else if y AND
3 = 1 then ptd-lookup va (y AND 4294967292) (mem s1) (Suc 0 + 1) else if y
AND 3 = 2 then Some ((ucast (ucast (y >> 8)::word24) << 12) OR ucast (ucast
va::word12), ucast y) else None) ≠ (if y AND 3 = 0 then None else if y AND 3 =

```

1 then ptd-lookup va (y AND 4294967292) ((mem s1) (10 := (mem s1 10)(addr ↦ val), 11 := (mem s1 11)(addr := None))) (Suc 0 + 1) else if y AND 3 = 2 then Some ((ucast (ucast (y >> 8)::word24) << 12) OR ucast (ucast va::word12), ucast y) else None)

have y AND 3 = 1 ∧ (if y AND 3 = 0 then None else if y AND 3 = 1 then ptd-lookup va (y AND 4294967292) (mem s1) (Suc 0 + 1) else if y AND 3 = 2 then Some ((ucast (ucast (y >> 8)::word24) << 12) OR ucast (ucast va::word12), ucast y) else None) ≠ ptd-lookup va (y AND 4294967292) ((mem s1) (10 := (mem s1 10)(addr ↦ val), 11 := (mem s1 11)(addr := None))) (Suc 0 + 1) ∨ (if y AND 3 = 0 then None else if y AND 3 = 1 then ptd-lookup va (y AND 4294967292) (mem s1) (Suc 0 + 1) else if y AND 3 = 2 then Some ((ucast (ucast (y >> 8)::word24) << 12) OR ucast (ucast va::word12), ucast y) else None) = (if y AND 3 = 0 then None else if y AND 3 = 1 then ptd-lookup va (y AND 4294967292) ((mem s1) (10 := (mem s1 10)(addr ↦ val), 11 := (mem s1 11)(addr := None))) (Suc 0 + 1) else if y AND 3 = 2 then Some ((ucast (ucast (y >> 8)::word24) << 12) OR ucast (ucast va::word12), ucast y) else None)

by presburger

moreover

{ **assume** y AND 3 = 1 ∧ (if y AND 3 = 0 then None else if y AND 3 = 1 then ptd-lookup va (y AND 4294967292) (mem s1) (Suc 0 + 1) else if y AND 3 = 2 then Some ((ucast (ucast (y >> 8)::word24) << 12) OR ucast (ucast va::word12), ucast y) else None) ≠ ptd-lookup va (y AND 4294967292) ((mem s1) (10 := (mem s1 10)(addr ↦ val), 11 := (mem s1 11)(addr := None))) (Suc 0 + 1)

then have y AND 3 = 1 ∧ (if y AND 3 = 0 then None else if y AND 3 = 1 then ptd-lookup va (y AND 4294967292) (mem s1) (Suc 0 + 1) else if y AND 3 = 2 then Some ((ucast (ucast (y >> 8)::word24) << 12) OR ucast (ucast va::word12), ucast y) else None) ≠ ptd-lookup va (y AND 4294967292) (mem s1) 2

by (metis One-nat-def Suc-1 Suc-eq-plus1 ptd-lookup-unchanged-2)

then have ?thesis

using f1 **by** auto }

ultimately have ?thesis

by blast }

ultimately have ?thesis

by blast }

then show ?thesis

by presburger

qed

qed

qed

lemma virt-to-phys-unchanged-sub1:

assumes a1: (let context-table-entry = (v1 >> 11 << 11) OR (v2 AND 511 << 2)

in Let (mem-context-val-w32 (word-of-int 9) (ucast context-table-entry) (mem s1))

(case-option None (λlvl1-page-table. ptd-lookup va lvl1-page-table (mem s1) 1)))

=

```

    (let context-table-entry = (v1 >> 11 << 11) OR (v2 AND 511 << 2)
      in Let (mem-context-val-w32 (word-of-int 9) (ucast context-table-entry) (mem
s2))
      (case-option None ( $\lambda$ lvl1-page-table. ptd-lookup va lvl1-page-table (mem s2) 1)))
shows (let context-table-entry = (v1 >> 11 << 11) OR (v2 AND 511 << 2)
  in Let (mem-context-val-w32 (word-of-int 9) (ucast context-table-entry)
    ((mem s1)(10 := (mem s1 10)(addr  $\mapsto$  val), 11 := (mem s1 11)(addr :=
None))))
    (case-option None ( $\lambda$ lvl1-page-table. ptd-lookup va lvl1-page-table
      ((mem s1)(10 := (mem s1 10)(addr  $\mapsto$  val), 11 := (mem s1 11)(addr :=
None))) 1))) =
    (let context-table-entry = (v1 >> 11 << 11) OR (v2 AND 511 << 2)
      in Let (mem-context-val-w32 (word-of-int 9) (ucast context-table-entry)
        ((mem s2)(10 := (mem s2 10)(addr  $\mapsto$  val), 11 := (mem s2 11)(addr :=
None))))
          (case-option None ( $\lambda$ lvl1-page-table. ptd-lookup va lvl1-page-table
            ((mem s2)(10 := (mem s2 10)(addr  $\mapsto$  val), 11 := (mem s2 11)(addr :=
None))) 1)))
proof –
  from a1 have
    (case mem-context-val-w32 (word-of-int 9) (ucast ((v1 >> 11 << 11) OR (v2
AND 511 << 2))) (mem s1) of
      None  $\Rightarrow$  None | Some lvl1-page-table  $\Rightarrow$  ptd-lookup va lvl1-page-table (mem s1)
1) =
    (case mem-context-val-w32 (word-of-int 9) (ucast ((v1 >> 11 << 11) OR (v2
AND 511 << 2))) (mem s2) of
      None  $\Rightarrow$  None | Some lvl1-page-table  $\Rightarrow$  ptd-lookup va lvl1-page-table (mem s2)
1)
    unfolding Let-def by auto
    then have (case mem-context-val-w32 (word-of-int 9) (ucast ((v1 >> 11 <<
11) OR (v2 AND 511 << 2)))
      ((mem s1)(10 := (mem s1 10)(addr  $\mapsto$  val), 11 := (mem s1 11)(addr :=
None))) of
        None  $\Rightarrow$  None | Some lvl1-page-table  $\Rightarrow$  ptd-lookup va lvl1-page-table (mem s1)
1) =
      (case mem-context-val-w32 (word-of-int 9) (ucast ((v1 >> 11 << 11) OR (v2
AND 511 << 2)))
        ((mem s2)(10 := (mem s2 10)(addr  $\mapsto$  val), 11 := (mem s2 11)(addr :=
None))) of
          None  $\Rightarrow$  None | Some lvl1-page-table  $\Rightarrow$  ptd-lookup va lvl1-page-table (mem s2)
1)
        using mem-context-val-w32-9-unchanged
        by (metis word-numeral-alt)
        then have (case mem-context-val-w32 (word-of-int 9) (ucast ((v1 >> 11 <<
11) OR (v2 AND 511 << 2)))
          ((mem s1)(10 := (mem s1 10)(addr  $\mapsto$  val), 11 := (mem s1 11)(addr :=
None))) of
            None  $\Rightarrow$  None | Some lvl1-page-table  $\Rightarrow$  ptd-lookup va lvl1-page-table
            ((mem s1)(10 := (mem s1 10)(addr  $\mapsto$  val), 11 := (mem s1 11)(addr :=

```

$\text{None})) 1) =$
 $(\text{case mem-context-val-w32 (word-of-int 9) (ucast ((v1 >> 11 << 11) OR (v2 AND 511 << 2)))$
 $((\text{mem s2})(10 := (\text{mem s2 } 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem s2 } 11)(\text{addr} := \text{None}))) \text{ of}$
 $\text{None} \Rightarrow \text{None} \mid \text{Some lvl1-page-table} \Rightarrow \text{ptd-lookup va lvl1-page-table}$
 $((\text{mem s2})(10 := (\text{mem s2 } 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem s2 } 11)(\text{addr} := \text{None}))) 1)$
using *ptd-lookup-unchanged-1*
proof –
obtain $ww :: \text{word32 option} \Rightarrow \text{word32}$ **where**
 $f1: \forall z. (z = \text{None} \vee z = \text{Some } (ww \ z)) \wedge (z \neq \text{None} \vee (\forall w. z \neq \text{Some } w))$
by *moura*
then have $f2: (\text{mem-context-val-w32 (word-of-int 9) (ucast ((v1 >> 11 << 11) OR (v2 AND 511 << 2))) ((\text{mem s1}) (10 := (\text{mem s1 } 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem s1 } 11)(\text{addr} := \text{None}))) = \text{None} \vee \text{mem-context-val-w32 (word-of-int 9) (ucast ((v1 >> 11 << 11) OR (v2 AND 511 << 2))) ((\text{mem s1}) (10 := (\text{mem s1 } 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem s1 } 11)(\text{addr} := \text{None}))) = \text{Some } (ww (\text{mem-context-val-w32 (word-of-int 9) (ucast ((v1 >> 11 << 11) OR (v2 AND 511 << 2))) ((\text{mem s1}) (10 := (\text{mem s1 } 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem s1 } 11)(\text{addr} := \text{None})))))) \wedge (\text{mem-context-val-w32 (word-of-int 9) (ucast ((v1 >> 11 << 11) OR (v2 AND 511 << 2))) ((\text{mem s1}) (10 := (\text{mem s1 } 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem s1 } 11)(\text{addr} := \text{None}))) \neq \text{None} \vee (\forall w. \text{mem-context-val-w32 (word-of-int 9) (ucast ((v1 >> 11 << 11) OR (v2 AND 511 << 2))) ((\text{mem s1}) (10 := (\text{mem s1 } 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem s1 } 11)(\text{addr} := \text{None}))) \neq \text{Some } w))$
by *blast*
then have $f3: (\text{case mem-context-val-w32 (word-of-int 9) (ucast ((v1 >> 11 << 11) OR (v2 AND 511 << 2))) ((\text{mem s1}) (10 := (\text{mem s1 } 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem s1 } 11)(\text{addr} := \text{None}))) \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } w \Rightarrow \text{ptd-lookup va w ((\text{mem s1}) (10 := (\text{mem s1 } 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem s1 } 11)(\text{addr} := \text{None}))) 1) \neq \text{None} \longrightarrow (\text{case mem-context-val-w32 (word-of-int 9) (ucast ((v1 >> 11 << 11) OR (v2 AND 511 << 2))) ((\text{mem s1}) (10 := (\text{mem s1 } 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem s1 } 11)(\text{addr} := \text{None}))) \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } w \Rightarrow \text{ptd-lookup va w ((\text{mem s1}) (10 := (\text{mem s1 } 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem s1 } 11)(\text{addr} := \text{None}))) 1) = (\text{case mem-context-val-w32 (word-of-int 9) (ucast ((v1 >> 11 << 11) OR (v2 AND 511 << 2))) ((\text{mem s1}) (10 := (\text{mem s1 } 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem s1 } 11)(\text{addr} := \text{None}))) \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } w \Rightarrow \text{ptd-lookup va w (mem s1) 1)$
by (*metis (no-types) <\wedge val va s1 ptp addr. ptd-lookup va ptp (mem s1) 1 = ptd-lookup va ptp ((\text{mem s1}) (10 := (\text{mem s1 } 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem s1 } 11)(\text{addr} := \text{None}))) 1> option.case(2) option.simps(4)*)
have $f4: \text{mem-context-val-w32 (word-of-int 9) (ucast ((v1 >> 11 << 11) OR (v2 AND 511 << 2))) ((\text{mem s2}) (10 := (\text{mem s2 } 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem s2 } 11)(\text{addr} := \text{None}))) = \text{Some } (ww (\text{mem-context-val-w32 (word-of-int 9) (ucast ((v1 >> 11 << 11) OR (v2 AND 511 << 2))) ((\text{mem s2}) (10 := (\text{mem s2 } 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem s2 } 11)(\text{addr} := \text{None})))))) \wedge \text{mem-context-val-w32 (word-of-int 9) (ucast ((v1 >> 11 << 11) OR (v2 AND 511 << 2))) ((\text{mem s1}) (10 := (\text{mem s1 } 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem s1 } 11)(\text{addr} := \text{None}))) = \text{Some } (ww (\text{mem-context-val-w32 (word-of-int 9) (ucast ((v1 >> 11 << 11) OR (v2 AND 511 << 2))) ((\text{mem s1}) (10 := (\text{mem s1 } 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem s1 } 11)(\text{addr} := \text{None})))$

$(v2 \text{ AND } 511 \ll 2))) ((\text{mem } s1) (10 := (\text{mem } s1 \ 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None})))) \longrightarrow (\text{case mem-context-val-w32 (word-of-int 9) (ucast ((v1 \gg 11 \ll 11) \text{ OR } (v2 \text{ AND } 511 \ll 2))) ((\text{mem } s1) (10 := (\text{mem } s1 \ 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None}))) \text{ of None} \Rightarrow \text{None} \mid \text{Some } w \Rightarrow \text{ptd-lookup va w ((\text{mem } s1) (10 := (\text{mem } s1 \ 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None}))) 1} = (\text{case Some (ww (mem-context-val-w32 (word-of-int 9) (ucast ((v1 \gg 11 \ll 11) \text{ OR } (v2 \text{ AND } 511 \ll 2))) ((\text{mem } s2) (10 := (\text{mem } s2 \ 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem } s2 \ 11)(\text{addr} := \text{None})))))) \text{ of None} \Rightarrow \text{None} \mid \text{Some } w \Rightarrow \text{ptd-lookup va w ((\text{mem } s2) (10 := (\text{mem } s2 \ 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem } s2 \ 11)(\text{addr} := \text{None}))) 1}$

by (*metis* (*no-types*) $\langle (\text{case mem-context-val-w32 (word-of-int 9) (ucast ((v1 \gg 11 \ll 11) \text{ OR } (v2 \text{ AND } 511 \ll 2))) ((\text{mem } s1) (10 := (\text{mem } s1 \ 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None}))) \text{ of None} \Rightarrow \text{None} \mid \text{Some } \text{lvl1-page-table} \Rightarrow \text{ptd-lookup va lvl1-page-table (mem } s1) 1} = (\text{case mem-context-val-w32 (word-of-int 9) (ucast ((v1 \gg 11 \ll 11) \text{ OR } (v2 \text{ AND } 511 \ll 2))) ((\text{mem } s2) (10 := (\text{mem } s2 \ 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem } s2 \ 11)(\text{addr} := \text{None}))) \text{ of None} \Rightarrow \text{None} \mid \text{Some } \text{lvl1-page-table} \Rightarrow \text{ptd-lookup va lvl1-page-table (mem } s2) 1} \rangle \langle \bigwedge \text{val va s1 ptp addr. ptd-lookup va ptp (mem } s1) 1 = \text{ptd-lookup va ptp ((\text{mem } s1) (10 := (\text{mem } s1 \ 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None}))) 1} \rangle \text{option.case}(2)$

have *f5*: $(\text{mem-context-val-w32 (word-of-int 9) (ucast ((v1 \gg 11 \ll 11) \text{ OR } (v2 \text{ AND } 511 \ll 2))) ((\text{mem } s2) (10 := (\text{mem } s2 \ 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem } s2 \ 11)(\text{addr} := \text{None}))) = \text{None} \vee \text{mem-context-val-w32 (word-of-int 9) (ucast ((v1 \gg 11 \ll 11) \text{ OR } (v2 \text{ AND } 511 \ll 2))) ((\text{mem } s2) (10 := (\text{mem } s2 \ 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem } s2 \ 11)(\text{addr} := \text{None}))) = \text{Some (ww (mem-context-val-w32 (word-of-int 9) (ucast ((v1 \gg 11 \ll 11) \text{ OR } (v2 \text{ AND } 511 \ll 2))) ((\text{mem } s2) (10 := (\text{mem } s2 \ 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem } s2 \ 11)(\text{addr} := \text{None})))))) \wedge (\text{mem-context-val-w32 (word-of-int 9) (ucast ((v1 \gg 11 \ll 11) \text{ OR } (v2 \text{ AND } 511 \ll 2))) ((\text{mem } s2) (10 := (\text{mem } s2 \ 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem } s2 \ 11)(\text{addr} := \text{None}))) \neq \text{None} \vee (\forall w. \text{mem-context-val-w32 (word-of-int 9) (ucast ((v1 \gg 11 \ll 11) \text{ OR } (v2 \text{ AND } 511 \ll 2))) ((\text{mem } s2) (10 := (\text{mem } s2 \ 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem } s2 \ 11)(\text{addr} := \text{None}))) \neq \text{Some } w)}$

using *f1* **by** *blast*

{ assume $(\text{case mem-context-val-w32 (word-of-int 9) (ucast ((v1 \gg 11 \ll 11) \text{ OR } (v2 \text{ AND } 511 \ll 2))) ((\text{mem } s1) (10 := (\text{mem } s1 \ 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None}))) \text{ of None} \Rightarrow \text{None} \mid \text{Some } w \Rightarrow \text{ptd-lookup va w ((\text{mem } s1) (10 := (\text{mem } s1 \ 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None}))) 1} \neq (\text{case mem-context-val-w32 (word-of-int 9) (ucast ((v1 \gg 11 \ll 11) \text{ OR } (v2 \text{ AND } 511 \ll 2))) ((\text{mem } s2) (10 := (\text{mem } s2 \ 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem } s2 \ 11)(\text{addr} := \text{None}))) \text{ of None} \Rightarrow \text{None} \mid \text{Some } w \Rightarrow \text{ptd-lookup va w ((\text{mem } s2) (10 := (\text{mem } s2 \ 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem } s2 \ 11)(\text{addr} := \text{None}))) 1}$

{ assume $(\text{case mem-context-val-w32 (word-of-int 9) (ucast ((v1 \gg 11 \ll 11) \text{ OR } (v2 \text{ AND } 511 \ll 2))) ((\text{mem } s2) (10 := (\text{mem } s2 \ 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem } s2 \ 11)(\text{addr} := \text{None}))) \text{ of None} \Rightarrow \text{None} \mid \text{Some } w \Rightarrow \text{ptd-lookup va w (mem } s2) 1} \neq \text{None} \wedge (\text{case mem-context-val-w32 (word-of-int 9) (ucast ((v1 \gg 11 \ll 11) \text{ OR } (v2 \text{ AND } 511 \ll 2))) ((\text{mem } s2) (10 := (\text{mem } s2 \ 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem } s2 \ 11)(\text{addr} := \text{None}))) \text{ of None} \Rightarrow \text{None} \mid \text{Some } w \Rightarrow \text{ptd-lookup va w (mem } s2) 1} \neq \text{None}$

then have $(\text{case mem-context-val-w32 (word-of-int 9) (ucast ((v1 \gg 11 \ll 11) \text{ OR } (v2 \text{ AND } 511 \ll 2))) ((\text{mem } s2) (10 := (\text{mem } s2 \ 10)(\text{addr} \mapsto \text{val}),$

$11 := (\text{mem } s2 \ 11)(\text{addr} := \text{None}))$ of $\text{None} \Rightarrow \text{None} \mid \text{Some } w \Rightarrow \text{ptd-lookup } va$
 $w (\text{mem } s2) \ 1 \neq \text{None} \wedge \text{mem-context-val-w32 } (\text{word-of-int } 9) (\text{ucast } ((v1 \gg$
 $11 \ll 11) \text{ OR } (v2 \text{ AND } 511 \ll 2))) ((\text{mem } s1) (10 := (\text{mem } s1 \ 10)(\text{addr} \mapsto$
 $val), 11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None})) \neq \text{None}$
by $(\text{metis } (\text{no-types}) \langle (\text{case mem-context-val-w32 } (\text{word-of-int } 9) (\text{ucast } ((v1 \gg$
 $\gg 11 \ll 11) \text{ OR } (v2 \text{ AND } 511 \ll 2))) ((\text{mem } s1) (10 := (\text{mem } s1 \ 10)(\text{addr} \mapsto$
 $val), 11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None}))$ of $\text{None} \Rightarrow \text{None} \mid \text{Some } \text{lvl1-page-table}$
 $\Rightarrow \text{ptd-lookup } va \ \text{lvl1-page-table } (\text{mem } s1) \ 1) = (\text{case mem-context-val-w32 } (\text{word-of-int}$
 $9) (\text{ucast } ((v1 \gg 11 \ll 11) \text{ OR } (v2 \text{ AND } 511 \ll 2))) ((\text{mem } s2) (10 := (\text{mem}$
 $s2 \ 10)(\text{addr} \mapsto val), 11 := (\text{mem } s2 \ 11)(\text{addr} := \text{None}))$ of $\text{None} \Rightarrow \text{None} \mid \text{Some}$
 $\text{lvl1-page-table} \Rightarrow \text{ptd-lookup } va \ \text{lvl1-page-table } (\text{mem } s2) \ 1) \rangle \text{option.simps}(4))$
then have $?thesis$
using $f5 \ f4 \ f2$ **by force** }
then have $?thesis$
using $f5 \ f3$ **by** $(\text{metis } (\text{no-types}) \langle (\text{case mem-context-val-w32 } (\text{word-of-int}$
 $9) (\text{ucast } ((v1 \gg 11 \ll 11) \text{ OR } (v2 \text{ AND } 511 \ll 2))) ((\text{mem } s1) (10 :=$
 $(\text{mem } s1 \ 10)(\text{addr} \mapsto val), 11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None}))$ of $\text{None} \Rightarrow$
 $\text{None} \mid \text{Some } \text{lvl1-page-table} \Rightarrow \text{ptd-lookup } va \ \text{lvl1-page-table } (\text{mem } s1) \ 1) = (\text{case}$
 $\text{mem-context-val-w32 } (\text{word-of-int } 9) (\text{ucast } ((v1 \gg 11 \ll 11) \text{ OR } (v2 \text{ AND } 511$
 $\ll 2))) ((\text{mem } s2) (10 := (\text{mem } s2 \ 10)(\text{addr} \mapsto val), 11 := (\text{mem } s2 \ 11)(\text{addr}$
 $:= \text{None}))$ of $\text{None} \Rightarrow \text{None} \mid \text{Some } \text{lvl1-page-table} \Rightarrow \text{ptd-lookup } va \ \text{lvl1-page-table}$
 $(\text{mem } s2) \ 1) \rangle \langle \wedge val \ va \ s1 \ \text{ptp } \text{addr. ptd-lookup } va \ \text{ptp } (\text{mem } s1) \ 1 = \text{ptd-lookup}$
 $va \ \text{ptp } ((\text{mem } s1) (10 := (\text{mem } s1 \ 10)(\text{addr} \mapsto val), 11 := (\text{mem } s1 \ 11)(\text{addr} :=$
 $\text{None})) \ 1) \rangle \text{option.case}(2) \ \text{option.simps}(4))$ }
then show $?thesis$
by blast
qed
then show $?thesis$
unfolding Let-def **by auto**
qed

lemma *virt-to-phys-unchanged*:

assumes $a1: (\forall va. \text{virt-to-phys } va \ (\text{mmu } s2) \ (\text{mem } s1) = \text{virt-to-phys } va \ (\text{mmu}$
 $s2) \ (\text{mem } s2))$

shows $(\forall va. \text{virt-to-phys } va \ (\text{mmu } s2) \ ((\text{mem } s1)(10 := (\text{mem } s1 \ 10)(\text{addr} \mapsto$
 $val),$

$$\begin{aligned}
 & 11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None})) = \\
 & \text{virt-to-phys } va \ (\text{mmu } s2) \ ((\text{mem } s2)(10 := (\text{mem } s2 \ 10)(\text{addr} \mapsto val), \\
 & 11 := (\text{mem } s2 \ 11)(\text{addr} := \text{None})))
 \end{aligned}$$

proof $(\text{cases registers } (\text{mmu } s2) \ \text{CR AND } 1 \neq 0)$

case True

then have $f1: \text{registers } (\text{mmu } s2) \ \text{CR AND } 1 \neq 0$ **by auto**

then show $?thesis$

proof $(\text{cases mmu-reg-val } (\text{mmu } s2) \ 256 = \text{None})$

case True

then show $?thesis$

by $(\text{simp add: virt-to-phys-def})$

next

case False

```

then have f2: mmu-reg-val (mmu s2) 256 ≠ None by auto
then show ?thesis
proof (cases mmu-reg-val (mmu s2) 512 = None)
  case True
    then show ?thesis using f1 f2
    apply (simp add: virt-to-phys-def)
    by auto
  next
    case False
    then show ?thesis using f1 f2 a1
    apply (simp add: virt-to-phys-def)
    apply clarify
    using virt-to-phys-unchanged-sub1 by fastforce
qed
qed
next
  case False
  then show ?thesis
  by (simp add: virt-to-phys-def)
qed

lemma virt-to-phys-unchanged2-sub1:
(case mem-context-val-w32 (word-of-int 9)
 (ucast ((v1 >> 11 << 11) OR (v2 AND 511 << 2))) (mem s2) of
 None ⇒ None | Some lvl1-page-table ⇒ ptd-lookup va lvl1-page-table (mem s2)
 1) =
(case mem-context-val-w32 (word-of-int 9)
 (ucast ((v1 >> 11 << 11) OR (v2 AND 511 << 2))) ((mem s2)
 (10 := (mem s2 10)(addr ↦ val), 11 := (mem s2 11)(addr := None))) of
 None ⇒ None | Some lvl1-page-table ⇒ ptd-lookup va lvl1-page-table ((mem s2)
 (10 := (mem s2 10)(addr ↦ val), 11 := (mem s2 11)(addr := None))) 1)
proof (cases mem-context-val-w32 9 (ucast ((v1 >> 11 << 11) OR (v2 AND
511 << 2))) (mem s2) = None)
  case True
    then have mem-context-val-w32 9 (ucast ((v1 >> 11 << 11) OR (v2 AND 511
<< 2))) (mem s2) = None ∧
      mem-context-val-w32 9 (ucast ((v1 >> 11 << 11) OR (v2 AND 511 << 2)))
      ((mem s2)
      (10 := (mem s2 10)(addr ↦ val), 11 := (mem s2 11)(addr := None))) = None
    using mem-context-val-w32-9-unchanged by metis
    then show ?thesis
    by auto
  next
    case False
    then have mem-context-val-w32 9 (ucast ((v1 >> 11 << 11) OR (v2 AND 511
<< 2))) (mem s2) ≠ None ∧
      (∀ y. mem-context-val-w32 9 (ucast ((v1 >> 11 << 11) OR (v2 AND 511 <<
2))) (mem s2) = Some y →
      mem-context-val-w32 9 (ucast ((v1 >> 11 << 11) OR (v2 AND 511 <<

```

```

2))) ((mem s2)
      (10 := (mem s2 10)(addr ↦ val), 11 := (mem s2 11)(addr := None))) =
Some y)
  using mem-context-val-w32-9-unchanged by metis
  then show ?thesis
  using ptd-lookup-unchanged-1 by fastforce
qed

```

```

lemma virt-to-phys-unchanged2:
virt-to-phys va (mmu s2) (mem s2) =
virt-to-phys va (mmu s2) ((mem s2)(10 := (mem s2 10)(addr ↦ val),
                                     11 := (mem s2 11)(addr := None)))

```

```

proof (cases registers (mmu s2) CR AND 1 ≠ 0)
case True
  then have f1: registers (mmu s2) CR AND 1 ≠ 0 by auto
  then show ?thesis
  proof (cases mmu-reg-val (mmu s2) 256 = None)
  case True
    then show ?thesis
    by (simp add: virt-to-phys-def)
  next
  case False
    then have f2: mmu-reg-val (mmu s2) 256 ≠ None by auto
    then show ?thesis
    proof (cases mmu-reg-val (mmu s2) 512 = None)
    case True
      then show ?thesis using f1 f2
      apply (simp add: virt-to-phys-def)
      by auto
    next
    case False
      then show ?thesis
      using f1 f2
      apply (simp add: virt-to-phys-def)
      apply clarify
      unfolding Let-def
      using virt-to-phys-unchanged2-sub1
      by auto
    qed
  qed
next
case False
  then show ?thesis
  by (simp add: virt-to-phys-def)
qed

```

```

lemma virt-to-phys-unchanged-low-equal:
assumes a1: low-equal s1 s2
shows (∀ va. virt-to-phys va (mmu s2) ((mem s1)(10 := (mem s1 10)(addr ↦

```

val),
 $11 := (mem\ s1\ 11)(addr := None))) =$
 $virt-to-phys\ va\ (mmu\ s2)\ ((mem\ s2)(10 := (mem\ s2\ 10)(addr \mapsto val),$
 $11 := (mem\ s2\ 11)(addr := None)))$
using $a1$ **apply** (*simp add: low-equal-def*)
using *virt-to-phys-unchanged*
by *metis*

lemma *mmu-low-equal: low-equal s1 s2 \implies mmu s1 = mmu s2*
by (*simp add: low-equal-def*)

lemma *mem-val-alt-8-unchanged0:*
assumes $a1: mem-equal\ s1\ s2\ pa$
shows $mem-val-alt\ 8\ (pa\ AND\ 68719476732)\ (s1\ (\!|mem := (mem\ s1)(10 := (mem$
 $s1\ 10)(addr \mapsto val),$
 $11 := (mem\ s1\ 11)(addr := None))) =$
 $mem-val-alt\ 8\ (pa\ AND\ 68719476732)\ (s2\ (\!|mem := (mem\ s2)(10 := (mem\ s2$
 $10)(addr \mapsto val),$
 $11 := (mem\ s2\ 11)(addr := None)))$
apply (*simp add: mem-val-alt-def*)
apply (*simp add: Let-def*)
using $a1$ **apply** (*simp add: mem-equal-def*)
by (*metis option.distinct(1)*)

lemma *mem-val-alt-8-unchanged1:*
assumes $a1: mem-equal\ s1\ s2\ pa$
shows $mem-val-alt\ 8\ ((pa\ AND\ 68719476732) + 1)\ (s1\ (\!|mem := (mem\ s1)(10 :=$
 $(mem\ s1\ 10)(addr \mapsto val),$
 $11 := (mem\ s1\ 11)(addr := None))) =$
 $mem-val-alt\ 8\ ((pa\ AND\ 68719476732) + 1)\ (s2\ (\!|mem := (mem\ s2)(10 := (mem$
 $s2\ 10)(addr \mapsto val),$
 $11 := (mem\ s2\ 11)(addr := None)))$
apply (*simp add: mem-val-alt-def*)
apply (*simp add: Let-def*)
using $a1$ **apply** (*simp add: mem-equal-def*)
by (*metis option.distinct(1)*)

lemma *mem-val-alt-8-unchanged2:*
assumes $a1: mem-equal\ s1\ s2\ pa$
shows $mem-val-alt\ 8\ ((pa\ AND\ 68719476732) + 2)\ (s1\ (\!|mem := (mem\ s1)(10 :=$
 $(mem\ s1\ 10)(addr \mapsto val),$
 $11 := (mem\ s1\ 11)(addr := None))) =$
 $mem-val-alt\ 8\ ((pa\ AND\ 68719476732) + 2)\ (s2\ (\!|mem := (mem\ s2)(10 := (mem$
 $s2\ 10)(addr \mapsto val),$
 $11 := (mem\ s2\ 11)(addr := None)))$
apply (*simp add: mem-val-alt-def*)
apply (*simp add: Let-def*)
using $a1$ **apply** (*simp add: mem-equal-def*)
by (*metis option.distinct(1)*)

lemma *mem-val-alt-8-unchanged3*:
assumes *a1*: *mem-equal s1 s2 pa*
shows *mem-val-alt 8 ((pa AND 68719476732) + 3) (s1 (mem := (mem s1)(10 := (mem s1 10)(addr ↦ val), 11 := (mem s1 11)(addr := None)))) = mem-val-alt 8 ((pa AND 68719476732) + 3) (s2 (mem := (mem s2)(10 := (mem s2 10)(addr ↦ val), 11 := (mem s2 11)(addr := None))))*
apply (*simp add: mem-val-alt-def*)
apply (*simp add: Let-def*)
using *a1* **apply** (*simp add: mem-equal-def*)
by (*metis option.distinct(1)*)

lemma *mem-val-alt-8-unchanged*:
assumes *a1*: *mem-equal s1 s2 pa*
shows *mem-val-alt 8 (pa AND 68719476732) (s1 (mem := (mem s1)(10 := (mem s1 10)(addr ↦ val), 11 := (mem s1 11)(addr := None)))) = mem-val-alt 8 (pa AND 68719476732) (s2 (mem := (mem s2)(10 := (mem s2 10)(addr ↦ val), 11 := (mem s2 11)(addr := None))))* \wedge
mem-val-alt 8 ((pa AND 68719476732) + 1) (s1 (mem := (mem s1)(10 := (mem s1 10)(addr ↦ val), 11 := (mem s1 11)(addr := None)))) = mem-val-alt 8 ((pa AND 68719476732) + 1) (s2 (mem := (mem s2)(10 := (mem s2 10)(addr ↦ val), 11 := (mem s2 11)(addr := None)))) \wedge
mem-val-alt 8 ((pa AND 68719476732) + 2) (s1 (mem := (mem s1)(10 := (mem s1 10)(addr ↦ val), 11 := (mem s1 11)(addr := None)))) = mem-val-alt 8 ((pa AND 68719476732) + 2) (s2 (mem := (mem s2)(10 := (mem s2 10)(addr ↦ val), 11 := (mem s2 11)(addr := None)))) \wedge
mem-val-alt 8 ((pa AND 68719476732) + 3) (s1 (mem := (mem s1)(10 := (mem s1 10)(addr ↦ val), 11 := (mem s1 11)(addr := None)))) = mem-val-alt 8 ((pa AND 68719476732) + 3) (s2 (mem := (mem s2)(10 := (mem s2 10)(addr ↦ val), 11 := (mem s2 11)(addr := None))))
using *a1 mem-val-alt-8-unchanged0 mem-val-alt-8-unchanged1 mem-val-alt-8-unchanged2 mem-val-alt-8-unchanged3*
by *blast*

lemma *mem-val-w32-8-unchanged*:
assumes *a1*: *mem-equal s1 s2 a*
shows *mem-val-w32 8 a (s1 (mem := (mem s1)(10 := (mem s1 10)(addr ↦ val), 11 := (mem s1 11)(addr := None)))) =*

```

mem-val-w32 8 a (s2(mem := (mem s2)(10 := (mem s2 10)(addr ↦ val),
  11 := (mem s2 11)(addr := None))))
apply (simp add: mem-val-w32-def)
apply (simp add: Let-def)
using mem-val-alt-8-unchanged a1 apply auto
  apply fastforce
  apply fastforce
by fastforce

lemma load-word-mem-8-unchanged:
assumes a1: low-equal s1 s2 ∧
load-word-mem s1 addra 8 = load-word-mem s2 addra 8
shows load-word-mem (s1(mem := (mem s1)(10 := (mem s1 10)(addr ↦ val),
  11 := (mem s1 11)(addr := None)))) addra 8 =
  load-word-mem (s2(mem := (mem s2)(10 := (mem s2 10)(addr ↦ val),
  11 := (mem s2 11)(addr := None)))) addra 8
proof (cases virt-to-phys addra (mmu s1) ((mem s1)(10 := (mem s1 10)(addr ↦
val),
  11 := (mem s1 11)(addr := None))) = None)
  case True
  then have virt-to-phys addra (mmu s1) ((mem s1)(10 := (mem s1 10)(addr ↦
val),
  11 := (mem s1 11)(addr := None))) = None ∧
    virt-to-phys addra (mmu s2) ((mem s2)(10 := (mem s2 10)(addr ↦
val),
  11 := (mem s2 11)(addr := None))) = None
  using a1 apply (auto simp add: mmu-low-equal)
  using a1 virt-to-phys-unchanged-low-equal by metis
  then show ?thesis
  by (simp add: load-word-mem-def)
next
  case False
  then have ∃ p. virt-to-phys addra (mmu s1) ((mem s1)(10 := (mem s1 10)(addr
↦ val),
  11 := (mem s1 11)(addr := None))) = Some p ∧
    virt-to-phys addra (mmu s2) ((mem s2)(10 := (mem s2 10)(addr ↦
val),
  11 := (mem s2 11)(addr := None))) = Some p
  using a1 apply (auto simp add: mmu-low-equal)
  using a1 virt-to-phys-unchanged-low-equal by metis
  then have ∃ p. virt-to-phys addra (mmu s1) ((mem s1)(10 := (mem s1 10)(addr
↦ val),
  11 := (mem s1 11)(addr := None))) = Some p ∧

```

```

      virt-to-phys addr (mmu s2) ((mem s2)(10 := (mem s2 10)(addr ↦
val),
      11 := (mem s2 11)(addr := None))) = Some p ∧
      virt-to-phys addr (mmu s1) (mem s1) = Some p ∧
      virt-to-phys addr (mmu s2) (mem s2) = Some p
    using virt-to-phys-unchanged2 by metis
  then show ?thesis using a1
  apply (simp add: load-word-mem-def)
  apply auto
  apply (simp add: low-equal-def)
  apply (simp add: user-accessible-def)
  using mem-val-w32-8-unchanged a1 user-accessible-8
  by (metis snd-conv)
qed

```

lemma *load-word-mem-select-8*:

```

assumes a1: fst (case load-word-mem s1 addr 8 of None ⇒ (None, s1)
  | Some w ⇒ (Some w, add-instr-cache s1 addr w 15)) =
fst (case load-word-mem s2 addr 8 of None ⇒ (None, s2)
  | Some w ⇒ (Some w, add-instr-cache s2 addr w 15))
shows load-word-mem s1 addr 8 = load-word-mem s2 addr 8
using a1
by (metis (mono-tags, lifting) fst-conv not-None-eq option.simps(4) option.simps(5))

```

lemma *memory-read-8-unchanged*:

```

assumes a1: low-equal s1 s2 ∧
fst (memory-read 8 addr s1) = fst (memory-read 8 addr s2)
shows fst (memory-read 8 addr
  (s1(mem := (mem s1)(10 := (mem s1 10)(addr ↦ val),
      11 := (mem s1 11)(addr := None)))))) =
fst (memory-read 8 addr
  (s2(mem := (mem s2)(10 := (mem s2 10)(addr ↦ val),
      11 := (mem s2 11)(addr := None))))))
proof (cases sys-reg s1 CCR AND 1 = 0)
  case True
  then have sys-reg s1 CCR AND 1 = 0 ∧ sys-reg s2 CCR AND 1 = 0
    using a1 sys-reg-low-equal by fastforce
  then show ?thesis using a1
  apply (simp add: memory-read-def)
  using load-word-mem-8-unchanged by blast
  next
  case False
  then have f1: sys-reg s1 CCR AND 1 ≠ 0 ∧ sys-reg s2 CCR AND 1 ≠ 0
    using a1 sys-reg-low-equal by fastforce
  then show ?thesis using a1
  proof (cases load-word-mem (s1(mem := (mem s1)(10 := (mem s1 10)(addr
↦ val),
      11 := (mem s1 11)(addr := None)))))) addr 8 = None)
  case True

```

```

then have load-word-mem (s1(mem := (mem s1)(10 := (mem s1 10)(addr ↦
val),
  11 := (mem s1 11)(addr := None))) addr 8 = None ∧
load-word-mem (s2(mem := (mem s2)(10 := (mem s2 10)(addr ↦ val),
  11 := (mem s2 11)(addr := None))) addr 8 = None
using a1 f1
apply (simp add: memory-read-def)
apply clarsimp
using load-word-mem-select-8 load-word-mem-8-unchanged
by fastforce
then show ?thesis
by (simp add: memory-read-def)
next
case False
then have ∃ y. load-word-mem (s1(mem := (mem s1)(10 := (mem s1 10)(addr
↦ val),
  11 := (mem s1 11)(addr := None))) addr 8 = Some y by auto
then have ∃ y. load-word-mem (s1(mem := (mem s1)(10 := (mem s1 10)(addr
↦ val),
  11 := (mem s1 11)(addr := None))) addr 8 = Some y ∧
load-word-mem (s2(mem := (mem s2)(10 := (mem s2 10)(addr
↦ val),
  11 := (mem s2 11)(addr := None))) addr 8 = Some y
using a1 f1
apply (simp add: memory-read-def)
apply clarsimp
using load-word-mem-select-8 load-word-mem-8-unchanged by fastforce
then show ?thesis using a1 f1
apply (simp add: memory-read-def)
by auto
qed
qed

```

```

lemma mem-val-alt-mod:
assumes a1: addr1 ≠ addr2
shows mem-val-alt 10 addr1 s =
mem-val-alt 10 addr1 (s(mem := (mem s)(10 := (mem s 10)(addr2 ↦ val),
  11 := (mem s 11)(addr2 := None))))
using a1 apply (simp add: mem-val-alt-def)
by (simp add: Let-def)

```

```

lemma mem-val-alt-mod2:
mem-val-alt 10 addr (s(mem := (mem s)(10 := (mem s 10)(addr ↦ val),
  11 := (mem s 11)(addr := None)))) = Some val
by (simp add: mem-val-alt-def)

```

```

lemma mem-val-alt-10-unchanged0:
assumes a1: mem-equal s1 s2 pa
shows mem-val-alt 10 (pa AND 68719476732) (s1(mem := (mem s1)(10 := (mem

```

```

s1 10)(addr ↦ val),
  11 := (mem s1 11)(addr := None))) =
  mem-val-alt 10 (pa AND 68719476732) (s2 (mem := (mem s2))(10 := (mem s2
10)(addr ↦ val),
  11 := (mem s2 11)(addr := None)))
apply (simp add: mem-val-alt-def)
apply (simp add: Let-def)
using a1 apply (simp add: mem-equal-def)
by (metis option.distinct(1))

```

```

lemma mem-val-alt-10-unchanged1:
assumes a1: mem-equal s1 s2 pa
shows mem-val-alt 10 ((pa AND 68719476732) + 1) (s1 (mem := (mem s1))(10
:= (mem s1 10)(addr ↦ val),
  11 := (mem s1 11)(addr := None))) =
  mem-val-alt 10 ((pa AND 68719476732) + 1) (s2 (mem := (mem s2))(10 :=
(mem s2 10)(addr ↦ val),
  11 := (mem s2 11)(addr := None)))
apply (simp add: mem-val-alt-def)
apply (simp add: Let-def)
using a1 apply (simp add: mem-equal-def)
by (metis option.distinct(1))

```

```

lemma mem-val-alt-10-unchanged2:
assumes a1: mem-equal s1 s2 pa
shows mem-val-alt 10 ((pa AND 68719476732) + 2) (s1 (mem := (mem s1))(10
:= (mem s1 10)(addr ↦ val),
  11 := (mem s1 11)(addr := None))) =
  mem-val-alt 10 ((pa AND 68719476732) + 2) (s2 (mem := (mem s2))(10 :=
(mem s2 10)(addr ↦ val),
  11 := (mem s2 11)(addr := None)))
apply (simp add: mem-val-alt-def)
apply (simp add: Let-def)
using a1 apply (simp add: mem-equal-def)
by (metis option.distinct(1))

```

```

lemma mem-val-alt-10-unchanged3:
assumes a1: mem-equal s1 s2 pa
shows mem-val-alt 10 ((pa AND 68719476732) + 3) (s1 (mem := (mem s1))(10
:= (mem s1 10)(addr ↦ val),
  11 := (mem s1 11)(addr := None))) =
  mem-val-alt 10 ((pa AND 68719476732) + 3) (s2 (mem := (mem s2))(10 :=
(mem s2 10)(addr ↦ val),
  11 := (mem s2 11)(addr := None)))
apply (simp add: mem-val-alt-def)
apply (simp add: Let-def)
using a1 apply (simp add: mem-equal-def)
by (metis option.distinct(1))

```

lemma *mem-val-alt-10-unchanged*:
assumes *a1*: *mem-equal s1 s2 pa*
shows *mem-val-alt 10 (pa AND 68719476732) (s1 (mem := (mem s1) (10 := (mem s1 10) (addr ↦ val),*
11 := (mem s1 11) (addr := None))) =
mem-val-alt 10 (pa AND 68719476732) (s2 (mem := (mem s2) (10 := (mem s2
10) (addr ↦ val),
11 := (mem s2 11) (addr := None)))) \wedge
mem-val-alt 10 ((pa AND 68719476732) + 1) (s1 (mem := (mem s1) (10 :=
(mem s1 10) (addr ↦ val),
11 := (mem s1 11) (addr := None))) =
mem-val-alt 10 ((pa AND 68719476732) + 1) (s2 (mem := (mem s2) (10 :=
(mem s2 10) (addr ↦ val),
11 := (mem s2 11) (addr := None)))) \wedge
mem-val-alt 10 ((pa AND 68719476732) + 2) (s1 (mem := (mem s1) (10 :=
(mem s1 10) (addr ↦ val),
11 := (mem s1 11) (addr := None))) =
mem-val-alt 10 ((pa AND 68719476732) + 2) (s2 (mem := (mem s2) (10 :=
(mem s2 10) (addr ↦ val),
11 := (mem s2 11) (addr := None)))) \wedge
mem-val-alt 10 ((pa AND 68719476732) + 3) (s1 (mem := (mem s1) (10 :=
(mem s1 10) (addr ↦ val),
11 := (mem s1 11) (addr := None))) =
mem-val-alt 10 ((pa AND 68719476732) + 3) (s2 (mem := (mem s2) (10 :=
(mem s2 10) (addr ↦ val),
11 := (mem s2 11) (addr := None))))
using *a1 mem-val-alt-10-unchanged0 mem-val-alt-10-unchanged1*
mem-val-alt-10-unchanged2 mem-val-alt-10-unchanged3
by *blast*

lemma *mem-val-w32-10-unchanged*:
assumes *a1*: *mem-equal s1 s2 a*
shows *mem-val-w32 10 a (s1 (mem := (mem s1) (10 := (mem s1 10) (addr ↦ val),*
11 := (mem s1 11) (addr := None))) =
mem-val-w32 10 a (s2 (mem := (mem s2) (10 := (mem s2 10) (addr ↦ val),
11 := (mem s2 11) (addr := None))))
apply (*simp add: mem-val-w32-def*)
apply (*simp add: Let-def*)
using *mem-val-alt-10-unchanged a1 apply auto*
apply *fastforce*
by *fastforce*

lemma *is-accessible: low-equal s1 s2* \implies
virt-to-phys addr (*mmu s1*) (*mem s1*) = *Some (a, b)* \implies
virt-to-phys addr (*mmu s2*) (*mem s2*) = *Some (a, b)* \implies
mmu-readable (get-acc-flag b) 10 \implies
mem-equal s1 s2 a
apply (*simp add: low-equal-def*)
apply (*simp add: user-accessible-def*)
by *fastforce*

lemma *load-word-mem-10-unchanged:*
assumes *a1: low-equal s1 s2* \wedge
load-word-mem s1 addr 10 = load-word-mem s2 addr 10
shows *load-word-mem (s1 (mem := (mem s1) (10 := (mem s1 10) (addr \mapsto val),*
11 := (mem s1 11) (addr := None))) addr 10 =
load-word-mem (s2 (mem := (mem s2) (10 := (mem s2 10) (addr \mapsto val),
11 := (mem s2 11) (addr := None))) addr 10
proof (*cases virt-to-phys addr (mmu s1) ((mem s1) (10 := (mem s1 10) (addr \mapsto*
val),
11 := (mem s1 11) (addr := None))) = None)
case *True*
then have *virt-to-phys addr (mmu s1) ((mem s1) (10 := (mem s1 10) (addr \mapsto*
val),
11 := (mem s1 11) (addr := None))) = None \wedge
virt-to-phys addr (mmu s2) ((mem s2) (10 := (mem s2 10) (addr \mapsto
val),
11 := (mem s2 11) (addr := None))) = None
using *a1 apply (auto simp add: mmu-low-equal)*
using *a1 virt-to-phys-unchanged-low-equal by metis*
then show *?thesis*
by (*simp add: load-word-mem-def*)
next
case *False*
then have $\exists p.$ *virt-to-phys addr (mmu s1) ((mem s1) (10 := (mem s1 10) (addr*
 \mapsto *val),*
11 := (mem s1 11) (addr := None))) = Some p \wedge
virt-to-phys addr (mmu s2) ((mem s2) (10 := (mem s2 10) (addr \mapsto
val),
11 := (mem s2 11) (addr := None))) = Some p
using *a1 apply (auto simp add: mmu-low-equal)*
using *a1 virt-to-phys-unchanged-low-equal by metis*
then have $\exists p.$ *virt-to-phys addr (mmu s1) ((mem s1) (10 := (mem s1 10) (addr*
 \mapsto *val),*
11 := (mem s1 11) (addr := None))) = Some p \wedge
virt-to-phys addr (mmu s2) ((mem s2) (10 := (mem s2 10) (addr \mapsto
val),
11 := (mem s2 11) (addr := None))) = Some p \wedge
virt-to-phys addr (mmu s1) (mem s1) = Some p \wedge
virt-to-phys addr (mmu s2) (mem s2) = Some p

```

    using virt-to-phys-unchanged2 by metis
  then show ?thesis using a1
  apply (simp add: load-word-mem-def)
  apply auto
  apply (simp add: low-equal-def)
  apply (simp add: user-accessible-def)
  using mem-val-w32-10-unchanged a1 by metis
qed

```

```

lemma load-word-mem-select-10:
  assumes a1: fst (case load-word-mem s1 addra 10 of None  $\Rightarrow$  (None, s1)
    | Some w  $\Rightarrow$  (Some w, add-data-cache s1 addra w 15)) =
    fst (case load-word-mem s2 addra 10 of None  $\Rightarrow$  (None, s2)
    | Some w  $\Rightarrow$  (Some w, add-data-cache s2 addra w 15))
  shows load-word-mem s1 addra 10 = load-word-mem s2 addra 10
  using a1
  by (metis (mono-tags, lifting) fst-conv not-None-eq option.simps(4) option.simps(5))

```

```

lemma memory-read-10-unchanged:
  assumes a1: low-equal s1 s2  $\wedge$ 
    fst (memory-read 10 addra s1) = fst (memory-read 10 addra s2)
  shows fst (memory-read 10 addra
    (s1(mem := (mem s1)(10 := (mem s1 10)(addr  $\mapsto$  val),
      11 := (mem s1 11)(addr := None)))))) =
    fst (memory-read 10 addra
    (s2(mem := (mem s2)(10 := (mem s2 10)(addr  $\mapsto$  val),
      11 := (mem s2 11)(addr := None))))))

```

```

proof (cases sys-reg s1 CCR AND 1 = 0)
  case True
  then have sys-reg s1 CCR AND 1 = 0  $\wedge$  sys-reg s2 CCR AND 1 = 0
    using a1 sys-reg-low-equal by fastforce
  then show ?thesis using a1
  apply (simp add: memory-read-def)
  using load-word-mem-10-unchanged by blast
next
  case False
  then have f1: sys-reg s1 CCR AND 1  $\neq$  0  $\wedge$  sys-reg s2 CCR AND 1  $\neq$  0
    using a1 sys-reg-low-equal by fastforce
  then show ?thesis using a1
  proof (cases load-word-mem (s1(mem := (mem s1)(10 := (mem s1 10)(addr
 $\mapsto$  val),
    11 := (mem s1 11)(addr := None)))))) addra 10 = None)
  case True
  then have load-word-mem (s1(mem := (mem s1)(10 := (mem s1 10)(addr  $\mapsto$ 
    val),
    11 := (mem s1 11)(addr := None)))) addra 10 = None  $\wedge$ 
    load-word-mem (s2(mem := (mem s2)(10 := (mem s2 10)(addr  $\mapsto$  val),
    11 := (mem s2 11)(addr := None)))) addra 10 = None
    using a1 f1

```

```

    apply (simp add: memory-read-def)
    apply clarsimp
    using load-word-mem-select-10 load-word-mem-10-unchanged by fastforce
    then show ?thesis
    by (simp add: memory-read-def)
  next
    case False
    then have  $\exists y. \text{load-word-mem } (s1 \text{ (mem := (mem s1) (10 := (mem s1 10) (addr }
\mapsto \text{val}),$ 
      11 := (mem s1 11)(addr := None))) addra 10 = Some y by auto
    then have  $\exists y. \text{load-word-mem } (s1 \text{ (mem := (mem s1) (10 := (mem s1 10) (addr }
\mapsto \text{val}),$ 
      11 := (mem s1 11)(addr := None))) addra 10 = Some y  $\wedge$ 
      load-word-mem (s2 (mem := (mem s2) (10 := (mem s2 10) (addr
\mapsto \text{val}),
      11 := (mem s2 11)(addr := None))) addra 10 = Some y
    using a1 f1
    apply (simp add: memory-read-def)
    apply clarsimp
    using load-word-mem-select-10 load-word-mem-10-unchanged by fastforce
    then show ?thesis using a1 f1
    apply (simp add: memory-read-def)
    by auto
  qed
qed

```

lemma *state-mem-mod-1011-low-equal-sub1*:

```

assumes a1: ( $\forall va. \text{virt-to-phys } va \text{ (mmu s2) (mem s1) =$ 
  virt-to-phys va (mmu s2) (mem s2))  $\wedge$ 
  ( $\forall pa. (\exists va b. \text{virt-to-phys } va \text{ (mmu s2) (mem s2) = Some (pa, b) } \wedge$ 
  mmu-readable (get-acc-flag b) 10)  $\longrightarrow$ 
  mem-equal s1 s2 pa)  $\wedge$ 
  mmu s1 = mmu s2  $\wedge$ 
  virt-to-phys va (mmu s2)
  ((mem s1)(10 := (mem s1 10)(addr  $\mapsto$  val), 11 := (mem s1 11)(addr := None)))
=
  Some (pa, b)  $\wedge$ 
  mmu-readable (get-acc-flag b) 10
shows mem-equal s1 s2 pa
proof -
  have virt-to-phys va (mmu s1)
  ((mem s1)(10 := (mem s1 10)(addr  $\mapsto$  val), 11 := (mem s1 11)(addr := None)))
=
  Some (pa, b)
  using a1 by auto
  then have virt-to-phys va (mmu s1) (mem s1) = Some (pa, b)
  using virt-to-phys-unchanged2 by metis
  then have virt-to-phys va (mmu s2) (mem s2) = Some (pa, b)
  using a1 by auto

```

then show *?thesis* **using** *a1* **by** *auto*
qed

lemma *mem-equal-unchanged*:
assumes *a1*: *mem-equal s1 s2 pa*
shows *mem-equal (s1 (mem := (mem s1) (10 := (mem s1 10) (addr ↦ val),
11 := (mem s1 11) (addr := None))))*
*(s2 (mem := (mem s2) (10 := (mem s2 10) (addr ↦ val),
11 := (mem s2 11) (addr := None))))*
pa
using *a1* **apply** (*simp add: mem-equal-def*)
by *auto*

lemma *state-mem-mod-1011-low-equal*:
assumes *a1*: *low-equal s1 s2* \wedge
*t1 = s1 (mem := (mem s1) (10 := (mem s1 10) (addr ↦ val), 11 := (mem s1
11) (addr := None)))* \wedge
*t2 = s2 (mem := (mem s2) (10 := (mem s2 10) (addr ↦ val), 11 := (mem s2
11) (addr := None)))*
shows *low-equal t1 t2*
using *a1*
apply (*simp add: low-equal-def*)
apply (*simp add: user-accessible-def*)
apply *auto*
apply (*simp add: asms virt-to-phys-unchanged-low-equal*)
using *state-mem-mod-1011-low-equal-sub1 mem-equal-unchanged*
apply *metis*
apply (*metis virt-to-phys-unchanged2*)
using *state-mem-mod-1011-low-equal-sub1 mem-equal-unchanged*
by *metis*

lemma *mem-mod-low-equal*:
assumes *a1*: *low-equal s1 s2* \wedge
t1 = (mem-mod 10 addr val s1) \wedge
t2 = (mem-mod 10 addr val s2)
shows *low-equal t1 t2*
using *a1*
apply (*simp add: mem-mod-def*)
by (*auto intro: state-mem-mod-1011-low-equal*)

lemma *mem-mod-w32-low-equal*:
assumes *a1*: *low-equal s1 s2* \wedge
t1 = mem-mod-w32 10 a bm data s1 \wedge
t2 = mem-mod-w32 10 a bm data s2
shows *low-equal t1 t2*
using *a1*
apply (*simp add: mem-mod-w32-def*)
apply (*simp add: Let-def*)
by (*meson mem-mod-low-equal*)

```

lemma store-word-mem-low-equal:
assumes a1: low-equal s1 s2  $\wedge$ 
Some t1 = store-word-mem s1 addr data bm 10  $\wedge$ 
Some t2 = store-word-mem s2 addr data bm 10
shows low-equal t1 t2 using a1
apply (simp add: store-word-mem-def)
apply (auto simp add: virt-to-phys-low-equal)
apply (case-tac virt-to-phys addr (mmu s2) (mem s2) = None)
  apply auto
apply (case-tac mmu-writable (get-acc-flag b) 10)
  apply auto
using mem-mod-w32-low-equal by blast

lemma memory-write-asi-low-equal:
assumes a1: low-equal s1 s2  $\wedge$ 
Some t1 = memory-write-asi 10 addr bm data s1  $\wedge$ 
Some t2 = memory-write-asi 10 addr bm data s2
shows low-equal t1 t2
using a1 apply (simp add: memory-write-asi-def)
by (meson add-data-cache-low-equal store-word-mem-low-equal)

lemma store-barrier-pending-mod-low-equal:
assumes a1: low-equal s1 s2  $\wedge$ 
t1 = store-barrier-pending-mod False s1  $\wedge$ 
t2 = store-barrier-pending-mod False s2
shows low-equal t1 t2
using a1 apply (simp add: store-barrier-pending-mod-def)
apply clarsimp
using a1 apply (auto simp add: state-var-low-equal)
by (auto intro: state-var2-low-equal)

lemma memory-write-low-equal:
assumes a1: low-equal s1 s2  $\wedge$ 
Some t1 = memory-write 10 addr bm data s1  $\wedge$ 
Some t2 = memory-write 10 addr bm data s2
shows low-equal t1 t2
apply (case-tac memory-write-asi 10 addr bm data s1 = None)
  using a1 apply (simp add: memory-write-def)
apply (case-tac memory-write-asi 10 addr bm data s2 = None)
  apply (meson assms low-equal-com memory-write-asi-low-equal-none)
using a1 apply (simp add: memory-write-def)
apply auto
by (metis memory-write-asi-low-equal store-barrier-pending-mod-low-equal)

lemma memory-write-low-equal2:
assumes a1: low-equal s1 s2  $\wedge$ 
Some t1 = memory-write 10 addr bm data s1
shows  $\exists$  t2. Some t2 = memory-write 10 addr bm data s2

```

```

using a1
apply (simp add: memory-write-def)
apply auto
by (metis (full-types) memory-write-def memory-write-low-equal-none2 not-None-eq)

```

```

lemma store-sub2-low-equal-sub1:
assumes a1: low-equal s1 s2 ∧
memory-write 10 addr (st-byte-mask instr addr)
(st-data0 instr curr-win rd addr s2) s1 = Some y ∧
memory-write 10 addr (st-byte-mask instr addr)
(st-data0 instr curr-win rd addr s2) s2 = Some ya
shows low-equal (y⟦traps := insert data-access-exception (traps y)⟧)
(ya⟦traps := insert data-access-exception (traps ya)⟧)
proof –
from a1 have f1: low-equal y ya using memory-write-low-equal by metis
then have traps y = traps ya by (simp add: low-equal-def)
then show ?thesis using f1 mod-trap-low-equal by fastforce
qed

```

```

lemma store-sub2-low-equal-sub2:
assumes a1: low-equal s1 s2 ∧
memory-write 10 addr (st-byte-mask instr addr)
(st-data0 instr curr-win rd addr s2) s1 = Some y ∧
memory-write 10 addr (st-byte-mask instr addr)
(st-data0 instr curr-win rd addr s2) s2 = Some ya ∧
memory-write 10 (addr + 4) 15 (user-reg-val curr-win (rd OR 1) y) y = None ∧
memory-write 10 (addr + 4) 15 (user-reg-val curr-win (rd OR 1) ya) ya = Some
yb
shows False
proof –
from a1 have f1: low-equal y ya using memory-write-low-equal by metis
then have (user-reg-val curr-win (rd OR 1) y) =
(user-reg-val curr-win (rd OR 1) ya)
by (simp add: low-equal-def user-reg-val-def)
then show ?thesis using a1
using f1 memory-write-low-equal-none by fastforce
qed

```

```

lemma store-sub2-low-equal-sub3:
assumes a1: low-equal s1 s2 ∧
memory-write 10 addr (st-byte-mask instr addr)
(st-data0 instr curr-win rd addr s2) s1 = Some y ∧
memory-write 10 addr (st-byte-mask instr addr)
(st-data0 instr curr-win rd addr s2) s2 = Some ya ∧
memory-write 10 (addr + 4) 15 (user-reg-val curr-win (rd OR 1) y) y = Some yb
∧
memory-write 10 (addr + 4) 15 (user-reg-val curr-win (rd OR 1) ya) ya = None
shows False

```

```

proof –
  from a1 have f1: low-equal y ya using memory-write-low-equal by metis
  then have (user-reg-val curr-win (rd OR 1) y) =
    (user-reg-val curr-win (rd OR 1) ya)
    by (simp add: low-equal-def user-reg-val-def)
  then show ?thesis using a1
  using f1 memory-write-low-equal-none2 by fastforce
qed

lemma store-sub2-low-equal-sub4:
assumes a1: low-equal s1 s2 ∧
memory-write 10 addr (st-byte-mask instr addr)
  (st-data0 instr curr-win rd addr s2) s1 = Some y ∧
memory-write 10 addr (st-byte-mask instr addr)
  (st-data0 instr curr-win rd addr s2) s2 = Some ya ∧
memory-write 10 (addr + 4) 15 (user-reg-val curr-win (rd OR 1) y) y = Some yb
  ∧
memory-write 10 (addr + 4) 15 (user-reg-val curr-win (rd OR 1) ya) ya = Some yc
shows low-equal yb yc
proof –
  from a1 have f1: low-equal y ya using memory-write-low-equal by metis
  then have (user-reg-val curr-win (rd OR 1) y) =
    (user-reg-val curr-win (rd OR 1) ya)
    by (simp add: low-equal-def user-reg-val-def)
  then show ?thesis using a1 f1
  by (metis memory-write-low-equal)
qed

lemma store-sub2-low-equal:
assumes a1: low-equal s1 s2 ∧
t1 = snd (fst (store-sub2 instr curr-win rd 10 addr s1)) ∧
t2 = snd (fst (store-sub2 instr curr-win rd 10 addr s2))
shows low-equal t1 t2
proof (cases memory-write 10 addr (st-byte-mask instr addr)
  (st-data0 instr curr-win rd addr s1) s1 = None)
  case True
  then have memory-write 10 addr (st-byte-mask instr addr)
    (st-data0 instr curr-win rd addr s1) s1 = None ∧
    memory-write 10 addr (st-byte-mask instr addr)
    (st-data0 instr curr-win rd addr s2) s2 = None
    using a1 by (metis memory-write-low-equal-none st-data0-low-equal)
  then show ?thesis using a1
  apply (simp add: store-sub2-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
  apply (simp add: case-prod-unfold return-def)
  apply (simp add: raise-trap-def add-trap-set-def)
  apply (simp add: simpler-modify-def)
  using mod-trap-low-equal traps-low-equal by fastforce

```

```

next
case False
then have f1: memory-write 10 addr (st-byte-mask instr addr)
  (st-data0 instr curr-win rd addr s1) s1  $\neq$  None  $\wedge$ 
  memory-write 10 addr (st-byte-mask instr addr)
  (st-data0 instr curr-win rd addr s2) s2  $\neq$  None
  using a1 by (metis memory-write-low-equal-none2 st-data0-low-equal)
then show ?thesis
proof (cases (fst instr)  $\in$  {load-store-type STD,load-store-type STDA})
  case True
  then show ?thesis using a1 f1
  apply (simp add: store-sub2-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def)
  apply (simp add: simpler-modify-def bind-def h1-def h2-def Let-def)
  apply (simp add: return-def)
  apply (simp add: bind-def case-prod-unfold)
  apply (simp add: simpler-modify-def)
  apply clarsimp
  apply (simp add: case-prod-unfold bind-def h1-def h2-def Let-def simpler-modify-def)
  apply (simp add: simpler-gets-def)
  apply auto
    apply (simp add: raise-trap-def add-trap-set-def)
    apply (simp add: simpler-modify-def)
    apply (simp add: st-data0-low-equal)
    apply (simp add: store-sub2-low-equal-sub1)
    apply (simp add: st-data0-low-equal)
    using store-sub2-low-equal-sub2 apply blast
    apply (simp add: st-data0-low-equal)
    using store-sub2-low-equal-sub3 apply blast
    apply (simp add: st-data0-low-equal)
    using store-sub2-low-equal-sub4 apply blast
    apply (simp add: st-data0-low-equal)
    apply (simp add: raise-trap-def add-trap-set-def)
    apply (simp add: simpler-modify-def)
    using store-sub2-low-equal-sub1 apply blast
    apply (simp add: st-data0-low-equal)
    using store-sub2-low-equal-sub2 apply blast
    apply (simp add: st-data0-low-equal)
    using store-sub2-low-equal-sub3 apply blast
    apply (simp add: st-data0-low-equal)
    using store-sub2-low-equal-sub4 by blast
  next
  case False
  then show ?thesis using a1 f1
  apply (simp add: store-sub2-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def)
  apply (simp add: simpler-modify-def bind-def h1-def h2-def Let-def)
  apply (simp add: return-def)
  apply (simp add: bind-def case-prod-unfold)

```

```

apply clarsimp
apply (simp add: simpler-modify-def)
apply (simp add: st-data0-low-equal)
using memory-write-low-equal by metis
qed
qed

lemma store-sub1-low-equal:
assumes a1: low-equal s1 s2  $\wedge$ 
(fst instr = load-store-type STB  $\vee$ 
fst instr = load-store-type STH  $\vee$ 
fst instr = load-store-type ST  $\vee$ 
fst instr = load-store-type STD)  $\wedge$ 
t1 = snd (fst (store-sub1 instr rd 0 s1))  $\wedge$ 
t2 = snd (fst (store-sub1 instr rd 0 s2))
shows low-equal t1 t2
proof (cases (fst instr = load-store-type STH  $\vee$  fst instr = load-store-type STHA)
 $\wedge$ 
      ((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s1))))::word1
 $\neq 0$ )
      case True
      then have ((fst instr = load-store-type STH  $\vee$  fst instr = load-store-type STHA)
 $\wedge$ 
        ((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s1))))::word1
 $\neq 0$ )  $\wedge$ 
          ((fst instr = load-store-type STH  $\vee$  fst instr = load-store-type STHA)  $\wedge$ 
            ((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s2))))::word1
 $\neq 0$ ))
          by (metis (mono-tags, lifting) assms get-addr-low-equal)
          then show ?thesis using a1
          apply (simp add: store-sub1-def)
          apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
          apply (simp add: case-prod-unfold)
          apply (simp add: raise-trap-def add-trap-set-def)
          apply (simp add: simpler-modify-def)
          apply clarsimp
          apply (simp add: get-curr-win3-low-equal)
          by (auto intro: get-curr-win2-low-equal mod-trap-low-equal)
        next
        case False
        then have f1:  $\neg$  ((fst instr = load-store-type STH  $\vee$  fst instr = load-store-type
STHA)  $\wedge$ 
          ((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s1))))::word1
 $\neq 0$ )
           $\wedge$ 
             $\neg$  ((fst instr = load-store-type STH  $\vee$  fst instr = load-store-type STHA)  $\wedge$ 
              ((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s2))))::word1
 $\neq 0$ ))
            by (metis (mono-tags, lifting) assms get-addr-low-equal)
          then show ?thesis
          proof (cases (fst instr  $\in$  {load-store-type ST,load-store-type STA})  $\wedge$ 

```

```

      ((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s1))))))::word2)
≠ 0)
  case True
  then have (fst instr ∈ {load-store-type ST,load-store-type STA}) ∧
    ((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s1))))))::word2)
≠ 0 ∧
    (fst instr ∈ {load-store-type ST,load-store-type STA}) ∧
    ((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s2))))))::word2)
≠ 0
  by (metis (mono-tags, lifting) assms get-addr-low-equal)
  then show ?thesis using a1 f1
  apply (simp add: store-sub1-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
  apply (simp add: case-prod-unfold)
  apply (simp add: raise-trap-def add-trap-set-def)
  apply (simp add: simpler-modify-def)
  apply clarsimp
  apply (simp add: get-curr-win3-low-equal)
  by (auto intro: get-curr-win2-low-equal mod-trap-low-equal)
next
case False
then have ¬((fst instr ∈ {load-store-type ST,load-store-type STA}) ∧
  ((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s1))))))::word2)
≠ 0) ∧
  ¬((fst instr ∈ {load-store-type ST,load-store-type STA}) ∧
  ((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s2))))))::word2)
≠ 0)
  by (metis (mono-tags, lifting) assms get-addr-low-equal)
  then have f2: ¬((fst instr = load-store-type ST ∨ fst instr = load-store-type
STA) ∧
  ((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s1))))))::word2)
≠ 0) ∧
  ¬((fst instr = load-store-type ST ∨ fst instr = load-store-type STA) ∧
  ((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s2))))))::word2)
≠ 0)
  by auto
  then show ?thesis
  proof (cases (fst instr ∈ {load-store-type STD,load-store-type STDA}) ∧
    ((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s1))))))::word3)
≠ 0)
  case True
  then have (fst instr ∈ {load-store-type STD,load-store-type STDA}) ∧
    ((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s1))))))::word3)
≠ 0 ∧
    (fst instr ∈ {load-store-type STD,load-store-type STDA}) ∧
    ((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s2))))))::word3)
≠ 0
  by (metis (mono-tags, lifting) assms get-addr-low-equal)
  then show ?thesis using a1

```

```

apply (simp add: store-sub1-def)
apply (simp add: simplifier-gets-def bind-def h1-def h2-def Let-def)
apply auto
apply (simp add: case-prod-unfold)
apply (simp add: raise-trap-def add-trap-set-def)
apply (simp add: simplifier-modify-def)
apply (simp add: get-curr-win3-low-equal)
by (auto intro: get-curr-win2-low-equal mod-trap-low-equal)
next
case False
then have  $\neg$  (fst instr  $\in$  {load-store-type STD, load-store-type STDA}  $\wedge$ 
  ((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s1))))):word3)  $\neq$  0)
 $\wedge$ 
   $\neg$  (fst instr  $\in$  {load-store-type STD, load-store-type STDA}  $\wedge$ 
  ((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s2))))):word3)  $\neq$  0)
  by (metis (mono-tags, lifting) assms get-addr-low-equal)
then have f3:  $\neg$  ((fst instr = load-store-type STD  $\vee$  fst instr = load-store-type
STDA)  $\wedge$ 
  ((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s1))))):word3)  $\neq$  0)
 $\wedge$ 
   $\neg$  ((fst instr = load-store-type STD  $\vee$  fst instr = load-store-type STDA)  $\wedge$ 
  ((ucast (get-addr (snd instr) (snd (fst (get-curr-win () s2))))):word3)  $\neq$  0)
  by auto
show ?thesis using a1
apply (simp add: store-sub1-def)
apply (simp add: simplifier-gets-def bind-def h1-def h2-def Let-def)
apply (unfold case-prod-beta)
apply (simp add: f1 f2 f3)
apply (simp-all add: st-asi-def)
using a1 apply clarsimp
apply (simp add: get-curr-win-low-equal get-addr2-low-equal)
by (metis store-sub2-low-equal get-curr-win2-low-equal)
qed
qed
qed

```

lemma *store-instr-low-equal*:

```

assumes a1: low-equal s1 s2  $\wedge$ 
  (fst instr = load-store-type STB  $\vee$ 
  fst instr = load-store-type STH  $\vee$ 
  fst instr = load-store-type ST  $\vee$ 
  fst instr = load-store-type STA  $\vee$ 
  fst instr = load-store-type STD)  $\wedge$ 
  (((get-S (cpu-reg-val PSR s1)):word1) = 0  $\wedge$ 
  (((get-S (cpu-reg-val PSR s2)):word1) = 0  $\wedge$ 
  t1 = snd (fst (store-instr instr s1))  $\wedge$  t2 = snd (fst (store-instr instr s2)))
shows low-equal t1 t2
proof -
  have get-S (cpu-reg-val PSR s1) = 0  $\wedge$  get-S (cpu-reg-val PSR s2) = 0

```

```

    using a1 by (simp add: ucast-id)
  then show ?thesis using a1
  apply (simp add: store-instr-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def)
  apply (simp add: Let-def)
  apply clarsimp
  apply (simp add: raise-trap-def add-trap-set-def)
  apply (simp add: simpler-modify-def)
  apply (simp add: traps-low-equal)
  by (auto intro: mod-trap-low-equal store-sub1-low-equal)
qed

```

```

lemma sethi-low-equal: low-equal s1 s2  $\wedge$ 
t1 = snd (fst (sethi-instr instr s1))  $\wedge$  t2 = snd (fst (sethi-instr instr s2))  $\implies$ 
low-equal t1 t2
apply (simp add: sethi-instr-def)
apply (simp add: Let-def)
apply (case-tac get-operand-w5 (snd instr ! Suc 0)  $\neq$  0)
apply auto
apply (simp add: bind-def h1-def h2-def Let-def)
apply (simp add: case-prod-unfold)
apply (simp add: get-curr-win-low-equal)
using get-curr-win2-low-equal write-reg-low-equal
apply metis
by (simp add: return-def)

```

```

lemma nop-low-equal: low-equal s1 s2  $\wedge$ 
t1 = snd (fst (nop-instr instr s1))  $\wedge$  t2 = snd (fst (nop-instr instr s2))  $\implies$ 
low-equal t1 t2
apply (simp add: nop-instr-def)
by (simp add: return-def)

```

```

lemma logical-instr-sub1-low-equal:
assumes a1: low-equal s1 s2  $\wedge$ 
t1 = snd (fst (logical-instr-sub1 instr-name result s1))  $\wedge$ 
t2 = snd (fst (logical-instr-sub1 instr-name result s2))
shows low-equal t1 t2
proof (cases instr-name = logic-type ANDcc  $\vee$ 
instr-name = logic-type ANDNcc  $\vee$ 
instr-name = logic-type ORcc  $\vee$ 
instr-name = logic-type ORNcc  $\vee$ 
instr-name = logic-type XORcc  $\vee$  instr-name = logic-type XNORcc)
case True
then show ?thesis using a1
apply (simp add: logical-instr-sub1-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
apply (simp add: logical-new-psr-val-def)
using write-cpu-low-equal cpu-reg-val-low-equal
by fastforce

```

```

next
  case False
  then show ?thesis using a1
  apply (simp add: logical-instr-sub1-def)
  by (simp add: return-def)
qed

lemma logical-instr-low-equal: low-equal s1 s2 ∧
t1 = snd (fst (logical-instr instr s1)) ∧ t2 = snd (fst (logical-instr instr s2)) ⇒
low-equal t1 t2
apply (simp add: logical-instr-def)
apply (simp add: Let-def simpler-gets-def bind-def h1-def h2-def)
apply (simp add: case-prod-unfold)
apply auto
  apply (simp-all add: get-curr-win-low-equal)
  apply (simp-all add: get-operand2-low-equal)
  using logical-instr-sub1-low-equal get-operand2-low-equal
  get-curr-win2-low-equal write-reg-low-equal user-reg-val-low-equal
proof -
  assume a1: low-equal s1 s2
  assume t2 = snd (fst (logical-instr-sub1 (fst instr) (logical-result (fst instr)
(user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0))
(snd (fst (get-curr-win () s2)))) (get-operand2 (snd instr) s2)) (snd (fst (write-reg
(logical-result (fst instr) (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) (get-operand2 (snd instr)
s2)) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst
(get-curr-win () s2))))))))))
  assume t1 = snd (fst (logical-instr-sub1 (fst instr) (logical-result (fst instr)
(user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0))
(snd (fst (get-curr-win () s1)))) (get-operand2 (snd instr) s2)) (snd (fst (write-reg
(logical-result (fst instr) (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) (get-operand2 (snd instr)
s2)) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst
(get-curr-win () s1))))))))))
  have  $\bigwedge w wa. \text{user-reg-val } w \text{ wa (snd (fst (get-curr-win () s2)))} = \text{user-reg-val } w$ 
  wa (snd (fst (get-curr-win () s1)))
  using a1 by (metis (no-types) get-curr-win2-low-equal user-reg-val-low-equal)
  then show low-equal (snd (fst (logical-instr-sub1 (fst instr) (logical-result (fst
instr) (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr !
Suc 0)) (snd (fst (get-curr-win () s1)))) (get-operand2 (snd instr) s2)) (snd (fst
(write-reg (logical-result (fst instr) (user-reg-val (fst (fst (get-curr-win () s2)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) (get-operand2
(snd instr) s2)) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd
(fst (get-curr-win () s1)))))))))) (snd (fst (logical-instr-sub1 (fst instr) (logical-result
(fst instr) (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr
! Suc 0)) (snd (fst (get-curr-win () s2)))) (get-operand2 (snd instr) s2)) (snd
(fst (write-reg (logical-result (fst instr) (user-reg-val (fst (fst (get-curr-win () s2)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) (get-operand2
(snd instr) s2)) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd
(fst (get-curr-win () s2)))))))))) (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! 3)) (snd

```

```

(fst (get-curr-win () s2)))))))))
  using a1 by (metis (no-types) get-curr-win2-low-equal logical-instr-sub1-low-equal
write-reg-low-equal)
next
  assume a2: low-equal s1 s2
  assume t1 = snd (fst (logical-instr-sub1 (fst instr)
(logical-result (fst instr)
(user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! Suc 0))
(snd (fst (get-curr-win () s1))))
(get-operand2 (snd instr) s2))
(snd (fst (write-reg
(user-reg-val (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s1))))
(fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s1))))))))))
  assume t2 = snd (fst (logical-instr-sub1 (fst instr)
(logical-result (fst instr)
(user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! Suc 0))
(snd (fst (get-curr-win () s2))))
(get-operand2 (snd instr) s2))
(snd (fst (write-reg
(user-reg-val (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s2))))
(fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s2))))))))))
  have  $\wedge w$  wa. user-reg-val w wa (snd (fst (get-curr-win () s2))) = user-reg-val w
wa (snd (fst (get-curr-win () s1)))
  using a2 by (metis (no-types) get-curr-win2-low-equal user-reg-val-low-equal)
  then show low-equal
    (snd (fst (logical-instr-sub1 (fst instr)
(logical-result (fst instr)
(user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! Suc 0))
(snd (fst (get-curr-win () s1))))
(get-operand2 (snd instr) s2))
(snd (fst (write-reg
(user-reg-val (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s1))))
(fst (fst (get-curr-win () s2))) 0 (snd (fst (get-curr-win ()
s1))))))))))
    (snd (fst (logical-instr-sub1 (fst instr)
(logical-result (fst instr)
(user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! Suc 0))
(snd (fst (get-curr-win () s2))))
(get-operand2 (snd instr) s2))
(snd (fst (write-reg

```

```

      (user-reg-val (fst (fst (get-curr-win () s2))) 0
        (snd (fst (get-curr-win () s2))))
      (fst (fst (get-curr-win () s2))) 0 (snd (fst (get-curr-win ()
s2)))))))))
proof –
  have low-equal (snd (fst (logical-instr-sub1 (fst instr) (logical-result (fst instr)
(user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0))
(snd (fst (get-curr-win () s1)))) (get-operand2 (snd instr) s2)) (snd (fst (write-reg
(user-reg-val (fst (fst (get-curr-win () s2))) 0 (snd (fst (get-curr-win () s1))))
(fst (fst (get-curr-win () s2))) 0 (snd (fst (get-curr-win () s1))))))))) (snd (fst
(logical-instr-sub1 (fst instr) (logical-result (fst instr) (user-reg-val (fst (fst (get-curr-win
() s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))))
(get-operand2 (snd instr) s2)) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win
() s2))) 0 (snd (fst (get-curr-win () s1)))) (fst (fst (get-curr-win () s2))) 0 (snd
(fst (get-curr-win () s2))))))))))
  by (meson a2 get-curr-win2-low-equal logical-instr-sub1-low-equal write-reg-low-equal)
  then show ?thesis
  using ⟨ $\wedge$  wa w. user-reg-val w wa (snd (fst (get-curr-win () s2))) = user-reg-val
w wa (snd (fst (get-curr-win () s1)))⟩ by presburger
qed
qed

```

lemma *shift-instr-low-equal*:

assumes a1: low-equal s1 s2 \wedge

t1 = snd (fst (shift-instr instr s1)) \wedge t2 = snd (fst (shift-instr instr s2))

shows low-equal t1 t2

proof (cases (fst instr = shift-type SLL) \wedge (get-operand-w5 ((snd instr)!3) \neq 0))

case True

then show ?thesis **using** a1

apply (simp add: shift-instr-def)

apply (simp add: Let-def)

apply (simp add: simpler-gets-def)

apply (simp add: bind-def h1-def h2-def Let-def case-prod-unfold)

apply auto

apply (simp-all add: get-curr-win-low-equal)

proof –

assume a1: low-equal s1 s2

assume t2 = snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win ()
s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2))) <<
unat (ucast (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr !
2)) (snd (fst (get-curr-win () s2))))::word5)) (fst (fst (get-curr-win () s2)))
(get-operand-w5 (snd instr ! 3)) (snd (fst (get-curr-win () s2))))))

assume t1 = snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win ()
s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))) <<
unat (ucast (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr !
2)) (snd (fst (get-curr-win () s1))))::word5)) (fst (fst (get-curr-win () s2)))
(get-operand-w5 (snd instr ! 3)) (snd (fst (get-curr-win () s1))))))

have \wedge w wa wb. low-equal (snd (fst (write-reg w wa wb s1))) (snd (fst
(write-reg w wa wb s2)))

```

using a1 by (metis write-reg-low-equal)
then show low-equal (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win
() s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))
<< unat (ucast (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! 2)) (snd (fst (get-curr-win () s1))))::word5)) (fst (fst (get-curr-win ()
s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (get-curr-win () s1)))))) (snd (fst
(write-reg (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr
! Suc 0)) (snd (fst (get-curr-win () s2))) << unat (ucast (user-reg-val (fst (fst
(get-curr-win () s2))) (get-operand-w5 (snd instr ! 2)) (snd (fst (get-curr-win ()
s2))))::word5)) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3))
(snd (fst (get-curr-win () s2))))))
using a1 by (simp add: get-curr-win-def simpler-gets-def user-reg-val-low-equal)
next
assume a2: low-equal s1 s2
assume t1 = snd (fst (write-reg
(user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! Suc 0))
(snd (fst (get-curr-win () s1))) <<
unat (get-operand-w5 (snd instr ! 2)))
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3))
(snd (fst (get-curr-win () s1))))))
assume t2 = snd (fst (write-reg
(user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! Suc 0))
(snd (fst (get-curr-win () s2))) <<
unat (get-operand-w5 (snd instr ! 2)))
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3))
(snd (fst (get-curr-win () s2))))))
have  $\bigwedge w wa wb.$  low-equal (snd (fst (write-reg w wa wb s1))) (snd (fst
(write-reg w wa wb s2)))
using a2 by (metis write-reg-low-equal)
then show low-equal
(snd (fst (write-reg
(user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! Suc 0))
(snd (fst (get-curr-win () s1))) <<
unat (get-operand-w5 (snd instr ! 2)))
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3))
(snd (fst (get-curr-win () s1))))))
(snd (fst (write-reg
(user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! Suc 0))
(snd (fst (get-curr-win () s2))) <<
unat (get-operand-w5 (snd instr ! 2)))
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3))
(snd (fst (get-curr-win () s2))))))
proof –
assume a1:  $\bigwedge w wa wb.$  low-equal (snd (fst (write-reg w wa wb s1))) (snd
(fst (write-reg w wa wb s2)))

```

```

      have  $\wedge u s. \text{fst} (\text{get-curr-win } u s) = (\text{ucast } (\text{get-CWP } (\text{cpu-reg-val } \text{PSR } s)))::'a \text{ word}, s)$ 
      by (simp add: get-curr-win-def simpler-gets-def)
      then show ?thesis
      using a1 assms user-reg-val-low-equal by fastforce
    qed
  qed
next
  case False
  then have f1:  $\neg((\text{fst } \text{instr} = \text{shift-type } \text{SLL}) \wedge (\text{get-operand-w5 } ((\text{snd } \text{instr})!3) \neq 0))$ 
  by auto
  then show ?thesis
  proof (cases (fst instr = shift-type SRL)  $\wedge$  (get-operand-w5 ((snd instr)!3)  $\neq$  0))
    case True
    then show ?thesis using a1 f1
    apply (simp add: shift-instr-def)
    apply (simp add: Let-def)
    apply (simp add: simpler-gets-def)
    apply (simp add: bind-def h1-def h2-def Let-def case-prod-unfold)
    apply auto
    apply (simp-all add: get-curr-win-low-equal)
    proof -
      assume a1: low-equal s1 s2
      assume t2 = snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2))) >> unat (ucast (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 2)) (snd (fst (get-curr-win () s2))))::word5)) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (get-curr-win () s2))))))
      assume t1 = snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))) >> unat (ucast (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 2)) (snd (fst (get-curr-win () s1))))::word5)) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (get-curr-win () s1))))))
      have  $\wedge u s. \text{fst} (\text{get-curr-win } u s) = (\text{ucast } (\text{get-CWP } (\text{cpu-reg-val } \text{PSR } s)))::'a \text{ word}, s)$ 
      by (simp add: get-curr-win-def simpler-gets-def)
      then show low-equal (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))) >> unat (ucast (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 2)) (snd (fst (get-curr-win () s1))))::word5)) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (get-curr-win () s1)))))) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2))) >> unat (ucast (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 2)) (snd (fst (get-curr-win () s2))))::word5)) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (get-curr-win () s2))))))
      using a1 user-reg-val-low-equal write-reg-low-equal by fastforce
    qed
  qed

```

```

next
  assume a2: low-equal s1 s2
  assume t1 = snd (fst (write-reg
    (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! Suc 0))
      (snd (fst (get-curr-win () s1))) >>
      unat (get-operand-w5 (snd instr ! 2)))
      (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3))
      (snd (fst (get-curr-win () s1))))))
  assume t2 = snd (fst (write-reg
    (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! Suc 0))
      (snd (fst (get-curr-win () s2))) >>
      unat (get-operand-w5 (snd instr ! 2)))
      (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3))
      (snd (fst (get-curr-win () s2))))))
  have  $\bigwedge u s. \text{fst (get-curr-win } u \text{ s)} = (\text{ucast (get-CWP (cpu-reg-val PSR }
s))::'a \text{ word, s)}$ 
  by (simp add: get-curr-win-def simpler-gets-def)
  then show low-equal
    (snd (fst (write-reg
      (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! Suc 0))
        (snd (fst (get-curr-win () s1))) >>
        unat (get-operand-w5 (snd instr ! 2)))
        (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3))
        (snd (fst (get-curr-win () s1))))))
      (snd (fst (write-reg
        (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! Suc 0))
          (snd (fst (get-curr-win () s2))) >>
          unat (get-operand-w5 (snd instr ! 2)))
          (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3))
          (snd (fst (get-curr-win () s2))))))
        using a2 user-reg-val-low-equal write-reg-low-equal by fastforce
    )
  qed
next
  case False
  then have f2:  $\neg((\text{fst instr} = \text{shift-type SRL}) \wedge (\text{get-operand-w5 } ((\text{snd instr})!3) \neq 0))$ 
  by auto
  then show ?thesis
  proof (cases (fst instr = shift-type SRA)  $\wedge$  (get-operand-w5 ((snd instr)!3)  $\neq$ 
0))
    case True
    then show ?thesis using a1 f1 f2
    apply (simp add: shift-instr-def)
    apply (simp add: Let-def)
    apply (simp add: simpler-gets-def)

```

```

apply (simp add: bind-def h1-def h2-def Let-def case-prod-unfold)
apply auto
apply (simp-all add: get-curr-win-low-equal)
proof –
  assume a1: low-equal s1 s2
    assume t1 = snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win ()
s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))) >>>
unat (ucast (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd in-
str ! 2)) (snd (fst (get-curr-win () s1))))::word5)) (fst (fst (get-curr-win () s2)))
(get-operand-w5 (snd instr ! 3)) (snd (fst (get-curr-win () s1))))))
    assume t2 = snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win ()
s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2))) >>>
unat (ucast (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd in-
str ! 2)) (snd (fst (get-curr-win () s2))))::word5)) (fst (fst (get-curr-win () s2)))
(get-operand-w5 (snd instr ! 3)) (snd (fst (get-curr-win () s2))))))
    have  $\forall w$  wa. user-reg-val wa w (snd (fst (get-curr-win () s1))) = user-reg-val
wa w (snd (fst (get-curr-win () s2)))
      using a1 by (meson get-curr-win2-low-equal user-reg-val-low-equal)
    then show low-equal (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win
() s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))
>>> unat (ucast (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! 2)) (snd (fst (get-curr-win () s1))))::word5)) (fst (fst (get-curr-win
() s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (get-curr-win () s1)))))) (snd
(fst (write-reg (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! Suc 0)) (snd (fst (get-curr-win () s2))) >>> unat (ucast (user-reg-val (fst
(fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 2)) (snd (fst (get-curr-win
() s2))))::word5)) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3))
(snd (fst (get-curr-win () s2))))))
      using a1 by (metis (no-types) get-curr-win2-low-equal write-reg-low-equal)
    next
      assume a2: low-equal s1 s2
      assume t1 = snd (fst (write-reg
(user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! Suc 0))
(snd (fst (get-curr-win () s1))) >>>
unat (get-operand-w5 (snd instr ! 2)))
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3))
(snd (fst (get-curr-win () s1))))))
      assume t2 = snd (fst (write-reg
(user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! Suc 0))
(snd (fst (get-curr-win () s2))) >>>
unat (get-operand-w5 (snd instr ! 2)))
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3))
(snd (fst (get-curr-win () s2))))))
      have  $\forall w$  wa. user-reg-val wa w (snd (fst (get-curr-win () s1))) = user-reg-val
wa w (snd (fst (get-curr-win () s2)))
      using a2 by (meson get-curr-win2-low-equal user-reg-val-low-equal)
      then show low-equal

```

```

      (snd (fst (write-reg
        (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! Suc 0))
          (snd (fst (get-curr-win () s1))) >>>
          unat (get-operand-w5 (snd instr ! 2)))
          (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3))
          (snd (fst (get-curr-win () s1))))))
      (snd (fst (write-reg
        (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! Suc 0))
          (snd (fst (get-curr-win () s2))) >>>
          unat (get-operand-w5 (snd instr ! 2)))
          (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3))
          (snd (fst (get-curr-win () s2))))))
    using a2 get-curr-win2-low-equal write-reg-low-equal by fastforce
  qed
next
case False
then show ?thesis using a1 f1 f2
apply (simp add: shift-instr-def)
apply (simp add: Let-def)
apply (simp add: simplifier-gets-def)
apply (simp add: bind-def h1-def h2-def Let-def case-prod-unfold)
apply (simp add: return-def)
using get-curr-win2-low-equal by blast
qed
qed
qed

```

lemma *add-instr-sub1-low-equal*:

assumes *a1*: *low-equal s1 s2* \wedge

t1 = *snd (fst (add-instr-sub1 instr-name result rs1-val operand2 s1))* \wedge

t2 = *snd (fst (add-instr-sub1 instr-name result rs1-val operand2 s2))*

shows *low-equal t1 t2*

proof (*cases instr-name = arith-type ADDcc* \vee *instr-name = arith-type ADDXcc*)

case *True*

then show ?thesis using *a1*

apply (*simp add: add-instr-sub1-def*)

apply (*simp add: simplifier-gets-def bind-def h1-def h2-def Let-def*)

apply (*clarsimp simp add: cpu-reg-val-low-equal*)

using *write-cpu-low-equal* by blast

next

case *False*

then show ?thesis using *a1*

apply (*simp add: add-instr-sub1-def*)

by (*simp add: return-def*)

qed

lemma *add-instr-low-equal*:

assumes $a1: \text{low-equal } s1 \ s2 \wedge$
 $t1 = \text{snd } (\text{fst } (\text{add-instr } \text{instr } s1)) \wedge t2 = \text{snd } (\text{fst } (\text{add-instr } \text{instr } s2))$
shows $\text{low-equal } t1 \ t2$
proof –
have $f1: \text{low-equal } s1 \ s2 \wedge$
 $t1 = \text{snd } (\text{fst } (\text{add-instr-sub1 } (\text{fst } \text{instr}))$
 $(\text{if } \text{fst } \text{instr} = \text{arith-type } \text{ADD} \vee \text{fst } \text{instr} = \text{arith-type } \text{ADDcc}$
 $\text{then } \text{user-reg-val } (\text{fst } (\text{fst } (\text{get-curr-win } ()) \ s1))) (\text{get-operand-w5}$
 $(\text{snd } \text{instr } ! \text{Suc } 0))$
 $(\text{snd } (\text{fst } (\text{get-curr-win } ()) \ s1))) +$
 $\text{get-operand2 } (\text{snd } \text{instr}) \ s1$
 $\text{else } \text{user-reg-val } (\text{fst } (\text{fst } (\text{get-curr-win } ()) \ s1))) (\text{get-operand-w5}$
 $(\text{snd } \text{instr } ! \text{Suc } 0))$
 $(\text{snd } (\text{fst } (\text{get-curr-win } ()) \ s1))) +$
 $\text{get-operand2 } (\text{snd } \text{instr}) \ s1 +$
 $\text{ucast } (\text{get-icc-C } (\text{cpu-reg-val } \text{PSR } (\text{snd } (\text{fst } (\text{get-curr-win } ())$
 $s1))))))$
 $(\text{user-reg-val } (\text{fst } (\text{fst } (\text{get-curr-win } ()) \ s1))) (\text{get-operand-w5 } (\text{snd}$
 $\text{instr } ! \text{Suc } 0))$
 $(\text{snd } (\text{fst } (\text{get-curr-win } ()) \ s1)))$
 $(\text{get-operand2 } (\text{snd } \text{instr}) \ s1)$
 $(\text{snd } (\text{fst } (\text{write-reg}$
 $(\text{if } \text{get-operand-w5 } (\text{snd } \text{instr } ! 3) \neq 0$
 $\text{then } \text{if } \text{fst } \text{instr} = \text{arith-type } \text{ADD} \vee \text{fst } \text{instr} = \text{arith-type}$
 ADDcc
 $\text{then } \text{user-reg-val } (\text{fst } (\text{fst } (\text{get-curr-win } ()) \ s1)))$
 $(\text{get-operand-w5 } (\text{snd } \text{instr } ! \text{Suc } 0))$
 $(\text{snd } (\text{fst } (\text{get-curr-win } ()) \ s1))) +$
 $\text{get-operand2 } (\text{snd } \text{instr}) \ s1$
 $\text{else } \text{user-reg-val } (\text{fst } (\text{fst } (\text{get-curr-win } ()) \ s1)))$
 $(\text{get-operand-w5 } (\text{snd } \text{instr } ! \text{Suc } 0))$
 $(\text{snd } (\text{fst } (\text{get-curr-win } ()) \ s1))) +$
 $\text{get-operand2 } (\text{snd } \text{instr}) \ s1 +$
 $\text{ucast } (\text{get-icc-C}$
 $(\text{cpu-reg-val } \text{PSR } (\text{snd } (\text{fst } (\text{get-curr-win}$
 $() \ s1))))))$
 $(\text{get-operand-w5 } (\text{snd } \text{instr } ! 3)) (\text{snd } (\text{fst } (\text{get-curr-win}$
 $() \ s1)))$
 $(\text{fst } (\text{fst } (\text{get-curr-win } ()) \ s1))) (\text{get-operand-w5 } (\text{snd } \text{instr}$
 $! 3))$
 $(\text{snd } (\text{fst } (\text{get-curr-win } ()) \ s1)))))) \wedge$
 $t2 = \text{snd } (\text{fst } (\text{add-instr-sub1 } (\text{fst } \text{instr}))$
 $(\text{if } \text{fst } \text{instr} = \text{arith-type } \text{ADD} \vee \text{fst } \text{instr} = \text{arith-type } \text{ADDcc}$
 $\text{then } \text{user-reg-val } (\text{fst } (\text{fst } (\text{get-curr-win } ()) \ s2))) (\text{get-operand-w5}$
 $(\text{snd } \text{instr } ! \text{Suc } 0))$
 $(\text{snd } (\text{fst } (\text{get-curr-win } ()) \ s2))) +$
 $\text{get-operand2 } (\text{snd } \text{instr}) \ s2$
 $\text{else } \text{user-reg-val } (\text{fst } (\text{fst } (\text{get-curr-win } ()) \ s2))) (\text{get-operand-w5}$

```

(snd instr ! Suc 0))
      (snd (fst (get-curr-win () s2))) +
      get-operand2 (snd instr) s2 +
      ucast (get-icc-C (cpu-reg-val PSR (snd (fst (get-curr-win ()
s2))))))
      (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! Suc 0))
      (snd (fst (get-curr-win () s2))))
      (get-operand2 (snd instr) s2)
      (snd (fst (write-reg
      (if get-operand-w5 (snd instr ! 3) ≠ 0
      then if fst instr = arith-type ADD ∨ fst instr = arith-type
ADDcc
      then user-reg-val (fst (fst (get-curr-win () s2)))
      (get-operand-w5 (snd instr ! Suc 0))
      (snd (fst (get-curr-win () s2))) +
      get-operand2 (snd instr) s2
      else user-reg-val (fst (fst (get-curr-win () s2)))
      (get-operand-w5 (snd instr ! Suc 0))
      (snd (fst (get-curr-win () s2))) +
      get-operand2 (snd instr) s2 +
      ucast (get-icc-C
      (cpu-reg-val PSR (snd (fst (get-curr-win
() s2))))))
      else user-reg-val (fst (fst (get-curr-win () s2)))
      (get-operand-w5 (snd instr ! 3)) (snd (fst (get-curr-win
() s2))))
      (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr
! 3))
      (snd (fst (get-curr-win () s2)))))))))
using a1 apply (simp add: add-instr-def)
apply (simp add: Let-def)
apply (simp add: simplifier-gets-def bind-def h1-def h2-def Let-def)
by (simp add: case-prod-unfold)
then show ?thesis
proof (cases get-operand-w5 (snd instr ! 3) ≠ 0)
  case True
    then have f2: get-operand-w5 (snd instr ! 3) ≠ 0 by auto
    then show ?thesis
      proof (cases fst instr = arith-type ADD ∨ fst instr = arith-type ADDcc)
        case True
          then show ?thesis
            using f1 f2 apply clarsimp
          proof –
            assume a1: low-equal s1 s2
            assume t1 = snd (fst (add-instr-sub1 (fst instr) (user-reg-val (fst (fst
(get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win
() s1))) + get-operand2 (snd instr) s1) (user-reg-val (fst (fst (get-curr-win () s1)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) (get-operand2

```

```

(snd instr) s1) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s1)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))) + get-operand2
(snd instr) s1) (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd
(fst (get-curr-win () s1))))))))
  assume a2: t2 = snd (fst (add-instr-sub1 (fst instr) (user-reg-val (fst (fst
(get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win
() s2))) + get-operand2 (snd instr) s2) (user-reg-val (fst (fst (get-curr-win () s2)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) (get-operand2
(snd instr) s2) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s2)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2))) + get-operand2
(snd instr) s2) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd
(fst (get-curr-win () s2))))))))))
  have f3:  $\forall is. \text{get-operand2 } is \ s1 = \text{get-operand2 } is \ s2$ 
  using a1 by (metis get-operand2-low-equal)
  have f4:  $\text{fst (fst (get-curr-win () s1))} = \text{fst (fst (get-curr-win () s2))}$ 
  using a1 by (meson get-curr-win-low-equal)
  have  $\forall s. \text{snd (fst (add-instr-sub1 (fst instr) (user-reg-val (fst (fst (get-curr-win
() s2))) (get-operand-w5 (snd instr ! Suc 0)) s + get-operand2 (snd instr) s2)
(user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) s)
(get-operand2 (snd instr) s2) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win
() s2))) (get-operand-w5 (snd instr ! Suc 0)) s + get-operand2 (snd instr) s2) (fst
(fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (get-curr-win
() s2)))))))))) = t2 \vee \neg \text{low-equal } s \ (\text{snd (fst (get-curr-win () s2))})$ 
  using a2 user-reg-val-low-equal by fastforce
  then show low-equal (snd (fst (add-instr-sub1 (fst instr) (user-reg-val (fst (fst
(get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win
() s1))) + get-operand2 (snd instr) s1) (user-reg-val (fst (fst (get-curr-win () s1)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) (get-operand2
(snd instr) s1) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s1)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))) + get-operand2
(snd instr) s1) (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd
(fst (get-curr-win () s1)))))))))) (snd (fst (add-instr-sub1 (fst instr) (user-reg-val
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst
(get-curr-win () s2))) + get-operand2 (snd instr) s2) (user-reg-val (fst (fst (get-curr-win
() s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2))))
(get-operand2 (snd instr) s2) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win
() s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2))) +
get-operand2 (snd instr) s2) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! 3)) (snd (fst (get-curr-win () s2))))))))))
  using f4 f3 a2 a1 by (metis (no-types) add-instr-sub1-low-equal get-curr-win2-low-equal
write-reg-low-equal)
qed
next
case False
then show ?thesis
using f1 f2 apply clarsimp
proof -
  assume a1: low-equal s1 s2
  have f2:  $\forall s \ sa \ sb \ w \ wa \ wb \ sc. (\neg \text{low-equal } s \ sa \vee \ sb \neq \text{snd (fst (write-reg$ 

```

w ($wa::'a$ word) wb s) \vee $sc \neq$ snd (fst ($write-reg$ w wa wb sa))) \vee $low-equal$ sb sc
by (*meson* *write-reg-low-equal*)
have $f3$: $gets$ ($\lambda s. ucast$ ($get-CWP$ ($cpu-reg-val$ PSR s))::' a word) =
 $get-curr-win$ ()
by (*simp* *add: get-curr-win-def*)
then have (($ucast$ ($get-CWP$ ($cpu-reg-val$ PSR $s1$))), $s1$), $False$) = (fst
($get-curr-win$ () $s1$), snd ($get-curr-win$ () $s1$))
by (*metis* (*no-types*) *prod.collapse simpler-gets-def*)
then have ($ucast$ ($get-CWP$ ($cpu-reg-val$ PSR $s1$))), $s1$) = fst ($get-curr-win$
() $s1$) \wedge \neg snd ($get-curr-win$ () $s1$)
by *blast*
then have $f4$: $ucast$ ($get-CWP$ ($cpu-reg-val$ PSR $s1$)) = fst (fst ($get-curr-win$
() $s1$)) \wedge $s1$ = snd (fst ($get-curr-win$ () $s1$))
by (*metis* (*no-types*) *prod.collapse prod.simps(1)*)
have (($ucast$ ($get-CWP$ ($cpu-reg-val$ PSR $s2$))), $s2$), $False$) = (fst ($get-curr-win$
() $s2$), snd ($get-curr-win$ () $s2$))
using $f3$ **by** (*metis* (*no-types*) *prod.collapse simpler-gets-def*)
then have ($ucast$ ($get-CWP$ ($cpu-reg-val$ PSR $s2$))), $s2$) = fst ($get-curr-win$
() $s2$) \wedge \neg snd ($get-curr-win$ () $s2$)
by *blast*
then have $f5$: $ucast$ ($get-CWP$ ($cpu-reg-val$ PSR $s2$)) = fst (fst ($get-curr-win$
() $s2$)) \wedge $s2$ = snd (fst ($get-curr-win$ () $s2$))
by (*metis* *prod.collapse prod.simps(1)*)
then have $f6$: $low-equal$ (snd (fst ($get-curr-win$ () $s1$))) (snd (fst ($get-curr-win$
() $s2$))) = $low-equal$ $s1$ $s2$
using $f4$ **by** *presburger*
have $f7$: fst (fst ($get-curr-win$ () $s1$)) = $ucast$ ($get-CWP$ ($cpu-reg-val$ PSR
 $s1$))
using $f4$ **by** *presburger*
have $f8$: $cpu-reg-val$ PSR $s1$ = $cpu-reg-val$ PSR $s2$
using $a1$ **by** (*meson* *cpu-reg-val-low-equal*)
have $f9$: $user-reg-val$ ($ucast$ ($get-CWP$ ($cpu-reg-val$ PSR $s2$))) ($get-operand-w5$
(snd $instr$! Suc 0)) (snd (fst ($get-curr-win$ () $s1$))) = $user-reg-val$ ($ucast$ ($get-CWP$
($cpu-reg-val$ PSR $s2$))) ($get-operand-w5$ (snd $instr$! Suc 0)) (snd (fst ($get-curr-win$
() $s2$)))
using $f6$ $a1$ **by** (*meson* *user-reg-val-low-equal*)
have $f10$: $ucast$ ($get-CWP$ ($cpu-reg-val$ PSR $s2$)) = fst (fst ($get-curr-win$ ()
 $s2$))
using $f5$ **by** *meson*
have $f11$: $\forall s$ sa $is. \neg$ $low-equal$ ($s::'a$ *sparc-state*) sa \vee $get-operand2$ is s =
 $get-operand2$ is sa
using *get-operand2-low-equal* **by** *blast*
then have $f12$: $user-reg-val$ (fst (fst ($get-curr-win$ () $s1$))) ($get-operand-w5$
(snd $instr$! Suc 0)) (snd (fst ($get-curr-win$ () $s1$))) + $get-operand2$ (snd $instr$) $s1$ + $ucast$ ($get-icc-C$ ($cpu-reg-val$ PSR (snd (fst ($get-curr-win$ () $s1$)))))) =
 $user-reg-val$ (fst (fst ($get-curr-win$ () $s2$))) ($get-operand-w5$ (snd $instr$! Suc 0))
(snd (fst ($get-curr-win$ () $s2$))) + $get-operand2$ (snd $instr$) $s2$ + $ucast$ ($get-icc-C$
($cpu-reg-val$ PSR (snd (fst ($get-curr-win$ () $s2$))))))
using $f9$ $f8$ $f5$ $f4$ $a1$ **by** *auto*

```

then have low-equal (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win
() s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))) +
get-operand2 (snd instr) s1 + ucast (get-icc-C (cpu-reg-val PSR (snd (fst (get-curr-win
() s1)))))) (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd (fst
(get-curr-win () s1)))))) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win
() s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2))) +
get-operand2 (snd instr) s2 + ucast (get-icc-C (cpu-reg-val PSR (snd (fst (get-curr-win
() s2)))))) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst
(get-curr-win () s2))))))
using f10 f8 f6 f4 f2 a1 by simp
then show low-equal (snd (fst (add-instr-sub1 (fst instr) (user-reg-val
(fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst
(get-curr-win () s1))) + get-operand2 (snd instr) s1 + ucast (get-icc-C (cpu-reg-val
PSR (snd (fst (get-curr-win () s1)))))) (user-reg-val (fst (fst (get-curr-win () s1)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))) (get-operand2
(snd instr) s1) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s1)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))) + get-operand2
(snd instr) s1 + ucast (get-icc-C (cpu-reg-val PSR (snd (fst (get-curr-win ()
s1)))))) (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd (fst
(get-curr-win () s1))))))))) (snd (fst (add-instr-sub1 (fst instr) (user-reg-val (fst
(fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win
() s2))) + get-operand2 (snd instr) s2 + ucast (get-icc-C (cpu-reg-val PSR (snd
(fst (get-curr-win () s2)))))) (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s2))) (get-operand2 (snd instr) s2)
(snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s2))) + get-operand2 (snd instr)
s2 + ucast (get-icc-C (cpu-reg-val PSR (snd (fst (get-curr-win () s2)))))) (fst (fst
(get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (get-curr-win ()
s2)))))))))
using f12 f11 f10 f9 f8 f7 a1 add-instr-sub1-low-equal by fastforce
qed
qed
next
case False
then have f3:  $\neg$  get-operand-w5 (snd instr ! 3)  $\neq$  0 by auto
then show ?thesis
proof (cases fst instr = arith-type ADD  $\vee$  fst instr = arith-type ADDcc)
case True
then show ?thesis
using f1 f3 apply clarsimp
proof –
assume a1: low-equal s1 s2
assume t1 = snd (fst (add-instr-sub1 (fst instr) (user-reg-val (fst (fst
(get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win
() s1))) + get-operand2 (snd instr) s1) (user-reg-val (fst (fst (get-curr-win () s1)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))) (get-operand2
(snd instr) s1) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s1)))
0 (snd (fst (get-curr-win () s1)))))) (fst (fst (get-curr-win () s1))) 0 (snd (fst
(get-curr-win () s1)))))))))

```

```

      assume t2 = snd (fst (add-instr-sub1 (fst instr) (user-reg-val (fst (fst
(get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win
() s2))) + get-operand2 (snd instr) s2) (user-reg-val (fst (fst (get-curr-win () s2)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) (get-operand2
(snd instr) s2) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s2)))
0 (snd (fst (get-curr-win () s2)))) (fst (fst (get-curr-win () s2))) 0 (snd (fst
(get-curr-win () s2))))))))))
      have f2:  $\forall is. \text{get-operand2 } is \ s1 = \text{get-operand2 } is \ s2$ 
      using a1 by (meson get-operand2-low-equal)
      have f3:  $\text{fst (fst (get-curr-win () s1))} = \text{fst (fst (get-curr-win () s2))}$ 
      using a1 by (meson get-curr-win-low-equal)
      have  $\forall w \ wa. \text{user-reg-val } wa \ w \ (\text{snd (fst (get-curr-win () s1)))} = \text{user-reg-val}$ 
 $\text{wa } w \ (\text{snd (fst (get-curr-win () s2)))}$ 
      using a1 by (meson get-curr-win2-low-equal user-reg-val-low-equal)
      then show low-equal (snd (fst (add-instr-sub1 (fst instr) (user-reg-val (fst (fst
(get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win
() s1))) + get-operand2 (snd instr) s1) (user-reg-val (fst (fst (get-curr-win () s1)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) (get-operand2
(snd instr) s1) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s1)))
0 (snd (fst (get-curr-win () s1)))) (fst (fst (get-curr-win () s1))) 0 (snd (fst
(get-curr-win () s1)))))))))) (snd (fst (add-instr-sub1 (fst instr) (user-reg-val (fst
(fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win
() s2))) + get-operand2 (snd instr) s2) (user-reg-val (fst (fst (get-curr-win () s2)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) (get-operand2
(snd instr) s2) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s2)))
0 (snd (fst (get-curr-win () s2)))) (fst (fst (get-curr-win () s2))) 0 (snd (fst
(get-curr-win () s2))))))))))
      using f3 f2 a1 by (metis (no-types) add-instr-sub1-low-equal get-curr-win2-low-equal
write-reg-low-equal)
    qed
  next
  case False
  then show ?thesis
  using f1 f3 apply clarsimp
  proof -
    assume a1: low-equal s1 s2
    have f2:  $\text{gets } (\lambda s. \text{ucast (get-CWP (cpu-reg-val PSR s))}) :: 'a \ \text{word} =$ 
 $\text{get-curr-win ()}$ 
    by (simp add: get-curr-win-def)
    then have  $((\text{ucast (get-CWP (cpu-reg-val PSR s1))}, s1), \text{False}) = (\text{fst}$ 
 $(\text{get-curr-win () } s1), \text{snd (get-curr-win () } s1))$ 
    by (metis (no-types) prod.collapse simpler-gets-def)
    then have  $(\text{ucast (get-CWP (cpu-reg-val PSR s1))}, s1) = \text{fst (get-curr-win}$ 
 $() \ s1) \wedge \neg \text{snd (get-curr-win () } s1)$ 
    by fastforce
    then have f3:  $\text{ucast (get-CWP (cpu-reg-val PSR s1))} = \text{fst (fst (get-curr-win}$ 
 $() \ s1)) \wedge s1 = \text{snd (fst (get-curr-win () } s1))$ 
    by (metis prod.collapse prod.simps(1))
    have  $((\text{ucast (get-CWP (cpu-reg-val PSR s2))}, s2), \text{False}) = (\text{fst (get-curr-win}$ 

```

```

() s2), snd (get-curr-win () s2))
  using f2 by (metis (no-types) prod.collapse simpler-gets-def)
  then have (ucast (get-CWP (cpu-reg-val PSR s2)), s2) = fst (get-curr-win
() s2) ∧ ¬ snd (get-curr-win () s2)
  by fastforce
  then have f4: ucast (get-CWP (cpu-reg-val PSR s2)) = fst (fst (get-curr-win
() s2)) ∧ s2 = snd (fst (get-curr-win () s2))
  by (metis (no-types) prod.collapse prod.simps(1))
  then have f5: low-equal (snd (fst (get-curr-win () s1))) (snd (fst (get-curr-win
() s2))) = low-equal s1 s2
  using f3 by presburger
  have f6: fst (fst (get-curr-win () s1)) = ucast (get-CWP (cpu-reg-val PSR
s1))
  using f3 by auto
  have f7: cpu-reg-val PSR s1 = cpu-reg-val PSR s2
  using a1 by (meson cpu-reg-val-low-equal)
  have f8: ∀ s sa w wa. ¬ low-equal s sa ∨ user-reg-val (w::'a word) wa s =
user-reg-val w wa sa
  by (meson user-reg-val-low-equal)
  have f9: ucast (get-CWP (cpu-reg-val PSR s2)) = fst (fst (get-curr-win ()
s2))
  using f4 by meson
  have ∀ s sa is. ¬ low-equal (s::'a sparc-state) sa ∨ get-operand2 is s =
get-operand2 is sa
  using get-operand2-low-equal by blast
  then have f10: get-operand2 (snd instr) s1 = get-operand2 (snd instr) s2
  using a1 by meson
  have f11: cpu-reg-val PSR (snd (fst (get-curr-win () s2))) = cpu-reg-val
PSR s1
  using f4 a1 by (simp add: cpu-reg-val-low-equal)
  have f12: user-reg-val (fst (fst (get-curr-win () s1))) 0 (snd (fst (get-curr-win
() s1))) = 0
  by (meson user-reg-val-def)
  have user-reg-val (fst (fst (get-curr-win () s2))) 0 (snd (fst (get-curr-win
() s2))) = 0
  by (meson user-reg-val-def)
  then have low-equal (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win
() s1))) 0 (snd (fst (get-curr-win () s1)))) (fst (fst (get-curr-win () s1))) 0 (snd
(fst (get-curr-win () s1)))))) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win
() s2))) 0 (snd (fst (get-curr-win () s2)))) (fst (fst (get-curr-win () s2))) 0 (snd
(fst (get-curr-win () s2))))))
  using f12 f9 f7 f5 f3 a1 write-reg-low-equal by fastforce
  then have low-equal (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win
() s1))) 0 (snd (fst (get-curr-win () s1)))) (fst (fst (get-curr-win () s1))) 0 (snd
(fst (get-curr-win () s1)))))) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win
() s2))) 0 (snd (fst (get-curr-win () s2)))) (fst (fst (get-curr-win () s2))) 0 (snd
(fst (get-curr-win () s2)))))) ∧ snd (fst (add-instr-sub1 (fst instr) (user-reg-val
(fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst
(get-curr-win () s1))) + get-operand2 (snd instr) s1 + ucast (get-icc-C (cpu-reg-val

```

```

PSR (snd (fst (get-curr-win () s1)))) (user-reg-val (fst (fst (get-curr-win () s1)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))) (get-operand2
(snd instr) s1) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s1)))
0 (snd (fst (get-curr-win () s1)))) (fst (fst (get-curr-win () s1))) 0 (snd (fst
(get-curr-win () s1)))))) = snd (fst (add-instr-sub1 (fst instr) (user-reg-val (fst
(fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win
() s2))) + get-operand2 (snd instr) s2 + ucast (get-icc-C (cpu-reg-val PSR (snd
(fst (get-curr-win () s2)))))) (if get-operand-w5 (snd instr ! Suc 0) = 0 then
0 else user-reg (snd (fst (get-curr-win () s2))) (fst (fst (get-curr-win () s2)))
(get-operand-w5 (snd instr ! Suc 0)) (get-operand2 (snd instr) s2) (snd (fst
(write-reg (user-reg-val (fst (fst (get-curr-win () s1))) 0 (snd (fst (get-curr-win
() s1)))) (fst (fst (get-curr-win () s1))) 0 (snd (fst (get-curr-win () s1))))))
^ snd (fst (add-instr-sub1 (fst instr) (user-reg-val (fst (fst (get-curr-win () s2)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2))) + get-operand2
(snd instr) s2 + ucast (get-icc-C (cpu-reg-val PSR (snd (fst (get-curr-win ()
s2)))))) (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr
! Suc 0)) (snd (fst (get-curr-win () s2))) (get-operand2 (snd instr) s2) (snd (fst
(write-reg (user-reg-val (fst (fst (get-curr-win () s2))) 0 (snd (fst (get-curr-win
() s2))) (fst (fst (get-curr-win () s2))) 0 (snd (fst (get-curr-win () s2))))))
= snd (fst (add-instr-sub1 (fst instr) (user-reg-val (fst (fst (get-curr-win () s2)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2))) + get-operand2
(snd instr) s2 + ucast (get-icc-C (cpu-reg-val PSR (snd (fst (get-curr-win ()
s2)))))) (if get-operand-w5 (snd instr ! Suc 0) = 0 then 0 else user-reg (snd (fst
(get-curr-win () s2))) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr !
Suc 0)) (get-operand2 (snd instr) s2) (snd (fst (write-reg (user-reg-val (fst (fst
(get-curr-win () s2))) 0 (snd (fst (get-curr-win () s2))) (fst (fst (get-curr-win ()
s2))) 0 (snd (fst (get-curr-win () s2))))))
)
using f11 f10 f9 f8 f7 f6 f5 f3 a1 by (simp add: user-reg-val-def)
then show low-equal (snd (fst (add-instr-sub1 (fst instr) (user-reg-val
(fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst
(get-curr-win () s1))) + get-operand2 (snd instr) s1 + ucast (get-icc-C (cpu-reg-val
PSR (snd (fst (get-curr-win () s1)))))) (user-reg-val (fst (fst (get-curr-win () s1)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))) (get-operand2
(snd instr) s1) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s1)))
0 (snd (fst (get-curr-win () s1)))) (fst (fst (get-curr-win () s1))) 0 (snd (fst
(get-curr-win () s1)))))) (snd (fst (add-instr-sub1 (fst instr) (user-reg-val (fst
(fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win
() s2))) + get-operand2 (snd instr) s2 + ucast (get-icc-C (cpu-reg-val PSR (snd
(fst (get-curr-win () s2)))))) (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s2))) (get-operand2 (snd instr)
s2) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s2))) 0 (snd (fst
(get-curr-win () s2))) (fst (fst (get-curr-win () s2))) 0 (snd (fst (get-curr-win ()
s2))))))
)
using add-instr-sub1-low-equal by blast
qed
qed
qed
qed

```

lemma *sub-instr-sub1-low-equal*:
assumes *a1*: *low-equal s1 s2* \wedge
t1 = *snd* (*fst* (*sub-instr-sub1 instr-name result rs1-val operand2 s1*)) \wedge
t2 = *snd* (*fst* (*sub-instr-sub1 instr-name result rs1-val operand2 s2*))
shows *low-equal t1 t2*
proof (*cases instr-name = arith-type SUBcc* \vee *instr-name = arith-type SUBXcc*)
 case *True*
 then show *?thesis using a1*
 apply (*simp add: sub-instr-sub1-def*)
 apply (*simp add: simpler-gets-def bind-def h1-def h2-def Let-def*)
 apply (*clarsimp simp add: cpu-reg-val-low-equal*)
 using *write-cpu-low-equal* **by** *blast*
next
 case *False*
 then show *?thesis using a1*
 apply (*simp add: sub-instr-sub1-def*)
 by (*simp add: return-def*)
qed

lemma *sub-instr-low-equal*:
assumes *a1*: *low-equal s1 s2* \wedge
t1 = *snd* (*fst* (*sub-instr instr s1*)) \wedge *t2* = *snd* (*fst* (*sub-instr instr s2*))
shows *low-equal t1 t2*
proof –
 have *f1*: *low-equal s1 s2* \wedge
 t1 = *snd* (*fst* (*sub-instr-sub1* (*fst instr*)
 (*if fst instr = arith-type SUB* \vee *fst instr = arith-type SUBcc*
 then user-reg-val (*fst* (*fst* (*get-curr-win* () *s1*))) (*get-operand-w5*
 (*snd instr* ! *Suc 0*))
 (*snd* (*fst* (*get-curr-win* () *s1*))) –
 get-operand2 (*snd instr*) *s1*
 else user-reg-val (*fst* (*fst* (*get-curr-win* () *s1*))) (*get-operand-w5*
 (*snd instr* ! *Suc 0*))
 (*snd* (*fst* (*get-curr-win* () *s1*))) –
 get-operand2 (*snd instr*) *s1* –
 ucast (*get-icc-C* (*cpu-reg-val PSR* (*snd* (*fst* (*get-curr-win* ()
 s1))))))
 (*user-reg-val* (*fst* (*fst* (*get-curr-win* () *s1*))) (*get-operand-w5* (*snd*
 instr ! *Suc 0*))
 (*snd* (*fst* (*get-curr-win* () *s1*))))
 (*get-operand2* (*snd instr*) *s1*)
 (*snd* (*fst* (*write-reg*
 (*if get-operand-w5* (*snd instr* ! *3*) \neq *0*
 then if fst instr = arith-type SUB \vee *fst instr = arith-type*
 SUBcc
 then user-reg-val (*fst* (*fst* (*get-curr-win* () *s1*)))
 (*get-operand-w5* (*snd instr* ! *Suc 0*))
 (*snd* (*fst* (*get-curr-win* () *s1*))) –
 get-operand2 (*snd instr*) *s1*

```

else user-reg-val (fst (fst (get-curr-win () s1)))
  (get-operand-w5 (snd instr ! Suc 0))
  (snd (fst (get-curr-win () s1))) –
  get-operand2 (snd instr) s1 –
  ucast (get-icc-C
    (cpu-reg-val PSR (snd (fst (get-curr-win
() s1))))))
else user-reg-val (fst (fst (get-curr-win () s1)))
  (get-operand-w5 (snd instr ! 3)) (snd (fst (get-curr-win
() s1))))
(fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr
! 3))
  (snd (fst (get-curr-win () s1)))))) ∧
t2 = snd (fst (sub-instr-sub1 (fst instr)
  (if fst instr = arith-type SUB ∨ fst instr = arith-type SUBcc
  then user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! Suc 0))
    (snd (fst (get-curr-win () s2))) –
    get-operand2 (snd instr) s2
  else user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! Suc 0))
    (snd (fst (get-curr-win () s2))) –
    get-operand2 (snd instr) s2 –
    ucast (get-icc-C (cpu-reg-val PSR (snd (fst (get-curr-win ()
s2))))))
  (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! Suc 0))
    (snd (fst (get-curr-win () s2))))
  (get-operand2 (snd instr) s2)
  (snd (fst (write-reg
    (if get-operand-w5 (snd instr ! 3) ≠ 0
    then if fst instr = arith-type SUB ∨ fst instr = arith-type
SUBcc
      then user-reg-val (fst (fst (get-curr-win () s2)))
        (get-operand-w5 (snd instr ! Suc 0))
        (snd (fst (get-curr-win () s2))) –
        get-operand2 (snd instr) s2
      else user-reg-val (fst (fst (get-curr-win () s2)))
        (get-operand-w5 (snd instr ! Suc 0))
        (snd (fst (get-curr-win () s2))) –
        get-operand2 (snd instr) s2 –
        ucast (get-icc-C
          (cpu-reg-val PSR (snd (fst (get-curr-win
() s2))))))
        else user-reg-val (fst (fst (get-curr-win () s2)))
          (get-operand-w5 (snd instr ! 3)) (snd (fst (get-curr-win
() s2))))
        (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr
! 3))

```

```

      (snd (fst (get-curr-win () s2)))))))))
using a1 apply (simp add: sub-instr-def)
apply (simp add: Let-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
by (simp add: case-prod-unfold)
then show ?thesis
proof (cases get-operand-w5 (snd instr ! 3) ≠ 0)
  case True
    then have f2: get-operand-w5 (snd instr ! 3) ≠ 0 by auto
    then show ?thesis
    proof (cases fst instr = arith-type SUB ∨ fst instr = arith-type SUBcc)
      case True
        then show ?thesis
        using f1 f2 apply clarsimp
        proof –
          assume a1: low-equal s1 s2
          assume a2: t1 = snd (fst (sub-instr-sub1 (fst instr) (user-reg-val (fst (fst
            (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win
            () s1))) – get-operand2 (snd instr) s1) (user-reg-val (fst (fst (get-curr-win () s1)))
            (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) (get-operand2
            (snd instr) s1) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s1)))
            (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))) – get-operand2
            (snd instr) s1) (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd
            (fst (get-curr-win () s1))))))))))
          assume a3: t2 = snd (fst (sub-instr-sub1 (fst instr) (user-reg-val (fst (fst
            (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win
            () s2))) – get-operand2 (snd instr) s2) (user-reg-val (fst (fst (get-curr-win () s2)))
            (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) (get-operand2
            (snd instr) s2) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s2)))
            (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2))) – get-operand2
            (snd instr) s2) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd
            (fst (get-curr-win () s2))))))))))
          then have f4: snd (fst (sub-instr-sub1 (fst instr) (user-reg-val (fst (fst
            (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win
            () s2))) – get-operand2 (snd instr) s1) (user-reg-val (fst (fst (get-curr-win () s2)))
            (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) (get-operand2
            (snd instr) s1) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s2)))
            (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2))) – get-operand2
            (snd instr) s1) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd
            (fst (get-curr-win () s2)))))))))) = t2
          using a1 by (simp add: get-operand2-low-equal)
          have ∀ s. ¬ low-equal (snd (fst (get-curr-win () s1))) s ∨ snd (fst (sub-instr-sub1
            (fst instr) (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr
            ! Suc 0)) s – get-operand2 (snd instr) s1) (user-reg-val (fst (fst (get-curr-win ()
            s2))) (get-operand-w5 (snd instr ! Suc 0)) s) (get-operand2 (snd instr) s1) (snd
            (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
            instr ! Suc 0)) s – get-operand2 (snd instr) s1) (fst (fst (get-curr-win () s2)))
            (get-operand-w5 (snd instr ! 3)) (snd (fst (get-curr-win () s1)))))))))) = t1
          using a2 a1 by (simp add: get-curr-win-low-equal user-reg-val-low-equal)

```

```

then show low-equal (snd (fst (sub-instr-sub1 (fst instr) (user-reg-val (fst (fst
(get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win
() s1))) – get-operand2 (snd instr) s1) (user-reg-val (fst (fst (get-curr-win () s1)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) (get-operand2
(snd instr) s1) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s1)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))) – get-operand2
(snd instr) s1) (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd
(fst (get-curr-win () s1)))))))) (snd (fst (sub-instr-sub1 (fst instr) (user-reg-val
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst
(get-curr-win () s2))) – get-operand2 (snd instr) s2) (user-reg-val (fst (fst (get-curr-win
() s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2))))
(get-operand2 (snd instr) s2) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win
() s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2))) –
get-operand2 (snd instr) s2) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! 3)) (snd (fst (get-curr-win () s2))))))))))
using f4 a3 a2 a1 by (metis (no-types) get-curr-win2-low-equal sub-instr-sub1-low-equal
write-reg-low-equal)
qed
next
case False
then show ?thesis
using f1 f2 apply clarsimp
proof –
  assume a1: low-equal s1 s2
  have f2: fst (get-curr-win () s1) = (ucast (get-CWP (cpu-reg-val PSR s1)),
s1)
    by (simp add: get-curr-win-def simpler-gets-def)
  have f3: cpu-reg-val PSR s1 = cpu-reg-val PSR s2
    using a1 by (meson cpu-reg-val-low-equal)
  then have f4: user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))) = user-reg-val (ucast (get-CWP
(cpu-reg-val PSR s2))) (get-operand-w5 (snd instr ! Suc 0)) s1
    using f2 by simp
  have f5:  $\forall s$  sa is.  $\neg$  low-equal (s:’a sparc-state) sa  $\vee$  get-operand2 is s =
get-operand2 is sa
    using get-operand2-low-equal by blast
  then have f6: sub-instr-sub1 (fst instr) (user-reg-val (fst (fst (get-curr-win
() s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))) –
get-operand2 (snd instr) s1 – ucast (get-icc-C (cpu-reg-val PSR (snd (fst (get-curr-win
() s1)))))) (user-reg-val (ucast (get-CWP (cpu-reg-val PSR s2))) (get-operand-w5
(snd instr ! Suc 0)) s2) (get-operand2 (snd instr) s2) (snd (fst (write-reg (user-reg-val
(fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst
(get-curr-win () s1))) – get-operand2 (snd instr) s1 – ucast (get-icc-C (cpu-reg-val
PSR (snd (fst (get-curr-win () s1)))))) (fst (fst (get-curr-win () s1))) (get-operand-w5
(snd instr ! 3)) (snd (fst (get-curr-win () s1)))))) = sub-instr-sub1 (fst instr)
(user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0))
(snd (fst (get-curr-win () s1))) – get-operand2 (snd instr) s1 – ucast (get-icc-C
(cpu-reg-val PSR (snd (fst (get-curr-win () s1)))))) (user-reg-val (fst (fst (get-curr-win
() s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))))

```



```

() s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2))) –
get-operand2 (snd instr) s2 – ucast (get-icc-C (cpu-reg-val PSR (snd (fst (get-curr-win
() s2)))))) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst
(get-curr-win () s2))))))
  using f3 f2 a1 by (metis (no-types) prod.sel(1) prod.sel(2) write-reg-low-equal)
  then show low-equal (snd (fst (sub-instr-sub1 (fst instr) (user-reg-val
(fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst
(get-curr-win () s1))) – get-operand2 (snd instr) s1 – ucast (get-icc-C (cpu-reg-val
PSR (snd (fst (get-curr-win () s1)))))) (user-reg-val (fst (fst (get-curr-win () s1)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))) (get-operand2
(snd instr) s1) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s1)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))) – get-operand2
(snd instr) s1 – ucast (get-icc-C (cpu-reg-val PSR (snd (fst (get-curr-win ()
s1)))))) (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd (fst
(get-curr-win () s1)))))))))) (snd (fst (sub-instr-sub1 (fst instr) (user-reg-val (fst
(fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win
() s2))) – get-operand2 (snd instr) s2 – ucast (get-icc-C (cpu-reg-val PSR (snd
(fst (get-curr-win () s2)))))) (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s2))) (get-operand2 (snd instr) s2)
(snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s2))) – get-operand2 (snd instr)
s2 – ucast (get-icc-C (cpu-reg-val PSR (snd (fst (get-curr-win () s2)))))) (fst (fst
(get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (get-curr-win ()
s2))))))))))
  using f9 f6 by (metis (no-types) sub-instr-sub1-low-equal)
qed
qed
next
case False
then have f3: ¬ get-operand-w5 (snd instr ! 3) ≠ 0 by auto
then show ?thesis
proof (cases fst instr = arith-type SUB ∨ fst instr = arith-type SUBcc)
  case True
  then show ?thesis
  using f1 f3 apply clarsimp
  proof –
    assume a1: low-equal s1 s2
    assume t1 = snd (fst (sub-instr-sub1 (fst instr) (user-reg-val (fst (fst
(get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win
() s1))) – get-operand2 (snd instr) s1) (user-reg-val (fst (fst (get-curr-win () s1)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))) (get-operand2
(snd instr) s1) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s1)))
0 (snd (fst (get-curr-win () s1)))))) (fst (fst (get-curr-win () s1))) 0 (snd (fst
(get-curr-win () s1))))))))))
    assume t2 = snd (fst (sub-instr-sub1 (fst instr) (user-reg-val (fst (fst
(get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win
() s2))) – get-operand2 (snd instr) s2) (user-reg-val (fst (fst (get-curr-win () s2)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) (get-operand2
(snd instr) s2) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s2)))

```

```

0 (snd (fst (get-curr-win () s2))) (fst (fst (get-curr-win () s2))) 0 (snd (fst
(get-curr-win () s2))))))
  have f2:  $\forall is. \text{get-operand2 } is \ s1 = \text{get-operand2 } is \ s2$ 
    using a1 get-operand2-low-equal by blast
  have f3:  $\text{fst (fst (get-curr-win () s1))} = \text{fst (fst (get-curr-win () s2))}$ 
    using a1 by (meson get-curr-win-low-equal)
  have  $\forall w \ wa. \text{user-reg-val } wa \ w \ (\text{snd (fst (get-curr-win () s1)))} = \text{user-reg-val}$ 
 $\text{wa } w \ (\text{snd (fst (get-curr-win () s2)))}$ 
    using a1 by (metis (no-types) get-curr-win2-low-equal user-reg-val-low-equal)
  then show low-equal (snd (fst (sub-instr-sub1 (fst instr) (user-reg-val (fst (fst
(get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win
() s1))) - get-operand2 (snd instr) s1) (user-reg-val (fst (fst (get-curr-win () s1)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) (get-operand2
(snd instr) s1) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s1)))
0 (snd (fst (get-curr-win () s1)))) (fst (fst (get-curr-win () s1))) 0 (snd (fst
(get-curr-win () s1)))))) (snd (fst (sub-instr-sub1 (fst instr) (user-reg-val (fst
(fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win
() s2))) - get-operand2 (snd instr) s2) (user-reg-val (fst (fst (get-curr-win () s2)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) (get-operand2
(snd instr) s2) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s2)))
0 (snd (fst (get-curr-win () s2)))) (fst (fst (get-curr-win () s2))) 0 (snd (fst
(get-curr-win () s2)))))))))
    using f3 f2 a1 by (metis (no-types) get-curr-win2-low-equal sub-instr-sub1-low-equal
write-reg-low-equal)
  qed
next
case False
then show ?thesis
using f1 f3 apply clarsimp
proof -
  assume a1: low-equal s1 s2
  have f2:  $\forall s \ sa \ sb \ w \ wa \ wb \ sc. (\neg \text{low-equal } s \ sa \vee \ sb \neq \text{snd (fst (write-reg}$ 
 $w \ (\text{wa}::'a \ \text{word}) \ wb \ s)) \vee \ sc \neq \text{snd (fst (write-reg } w \ wa \ wb \ sa))) \vee \text{low-equal } sb \ sc$ 
    by (meson write-reg-low-equal)
  have ((ucast (get-CWP (cpu-reg-val PSR s1)), s1), False) = get-curr-win
() s1
    by (simp add: get-curr-win-def simpler-gets-def)
  then have f3:  $\text{ucast (get-CWP (cpu-reg-val PSR s1))} = \text{fst (fst (get-curr-win$ 
 $() \ s1)) \wedge \ s1 = \text{snd (fst (get-curr-win () s1))}$ 
    by (metis (no-types) prod.collapse prod.simps(1))
  have ((ucast (get-CWP (cpu-reg-val PSR s2)), s2), False) = get-curr-win
() s2
    by (simp add: get-curr-win-def simpler-gets-def)
  then have f4:  $\text{ucast (get-CWP (cpu-reg-val PSR s2))} = \text{fst (fst (get-curr-win$ 
 $() \ s2)) \wedge \ s2 = \text{snd (fst (get-curr-win () s2))}$ 
    by (metis (no-types) prod.collapse prod.simps(1))
  have f5:  $\forall s \ sa \ sb \ sc \ w \ wa \ wb \ sd. (\neg \text{low-equal } (s::'a \ \text{sparc-state}) \ sa \vee \ sb \neq$ 
 $\text{snd (fst (sub-instr-sub1 } sc \ w \ wa \ wb \ s)) \vee \ sd \neq \text{snd (fst (sub-instr-sub1 } sc \ w \ wa \ wb$ 
 $\ sa))) \vee \text{low-equal } sb \ sd$ 

```

```

    by (meson sub-instr-sub1-low-equal)
    have low-equal (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win ()
s1))) 0 (snd (fst (get-curr-win () s1)))) (fst (fst (get-curr-win () s1))) 0 (snd (fst
(get-curr-win () s1)))))) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win ()
s2))) 0 (snd (fst (get-curr-win () s2)))) (fst (fst (get-curr-win () s2))) 0 (snd (fst
(get-curr-win () s2))))))
    using f4 f3 f2 a1 by (simp add: cpu-reg-val-low-equal user-reg-val-low-equal)
    then show low-equal (snd (fst (sub-instr-sub1 (fst instr) (user-reg-val
(fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst
(get-curr-win () s1))) - get-operand2 (snd instr) s1 - ucast (get-icc-C (cpu-reg-val
PSR (snd (fst (get-curr-win () s1)))))) (user-reg-val (fst (fst (get-curr-win () s1)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))) (get-operand2
(snd instr) s1) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s1)))
0 (snd (fst (get-curr-win () s1)))) (fst (fst (get-curr-win () s1))) 0 (snd (fst
(get-curr-win () s1)))))))))) (snd (fst (sub-instr-sub1 (fst instr) (user-reg-val (fst
(fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win
() s2))) - get-operand2 (snd instr) s2 - ucast (get-icc-C (cpu-reg-val PSR (snd
(fst (get-curr-win () s2)))))) (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) (get-operand2 (snd instr)
s2) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s2))) 0 (snd (fst
(get-curr-win () s2)))) (fst (fst (get-curr-win () s2))) 0 (snd (fst (get-curr-win ()
s2))))))))))
    using f5 f4 f3 a1 by (simp add: cpu-reg-val-low-equal get-operand2-low-equal
user-reg-val-low-equal)
    qed
  qed
  qed
  qed

```

lemma *mul-instr-sub1-low-equal*:

assumes *a1*: *low-equal s1 s2* \wedge

t1 = *snd (fst (mul-instr-sub1 instr-name result s1))* \wedge

t2 = *snd (fst (mul-instr-sub1 instr-name result s2))*

shows *low-equal t1 t2*

proof (*cases instr-name* \in {*arith-type SMULcc,arith-type UMULcc*})

case True

then show *?thesis using a1*

apply (*simp add: mul-instr-sub1-def*)

apply (*simp add: simpler-gets-def bind-def h1-def h2-def Let-def*)

apply (*clarsimp simp add: cpu-reg-val-low-equal*)

using write-cpu-low-equal by blast

next

case False

then show *?thesis using a1*

apply (*simp add: mul-instr-sub1-def*)

by (*simp add: return-def*)

qed

lemma *mul-instr-low-equal*:

```

  ⟨low-equal t1 t2⟩
  if ⟨low-equal s1 s2 ∧ t1 = snd (fst (mul-instr instr s1)) ∧ t2 = snd (fst (mul-instr
instr s2))⟩
  proof -
    from that have ⟨low-equal s1 s2⟩
      and t1: ⟨t1 = snd (fst (mul-instr instr s1))⟩
      and t2: ⟨t2 = snd (fst (mul-instr instr s2))⟩
      by simp-all
    have f2: ∀ s sa sb sc w sd. ¬ low-equal (s::'a sparc-state) sa ∨ sb ≠ snd (fst
(mul-instr-sub1 sc w s)) ∨ sd ≠ snd (fst (mul-instr-sub1 sc w sa)) ∨ low-equal sb
sd
      using mul-instr-sub1-low-equal by blast
    have f3: ∀ s sa sb w wa wb sc. ¬ low-equal s sa ∨ sb ≠ snd (fst (write-reg w
(wa::'a word) wb s)) ∨ sc ≠ snd (fst (write-reg w wa wb sa)) ∨ low-equal sb sc
      by (meson write-reg-low-equal)
    have f4: ∀ s sa sb w c sc. ¬ low-equal (s::'a sparc-state) sa ∨ sb ≠ snd (fst
(write-cpu w c s)) ∨ sc ≠ snd (fst (write-cpu w c sa)) ∨ low-equal sb sc
      by (meson write-cpu-low-equal)
    have f6: ((ucast (get-CWP (cpu-reg-val PSR s1)), s1), False) = (fst (get-curr-win
() s1), snd (get-curr-win () s1))
      by (simp add: get-curr-win-def simpler-gets-def)
    have f7: fst (fst (get-curr-win () s1)) = fst (fst (get-curr-win () s2))
      using ⟨low-equal s1 s2⟩ by (meson get-curr-win-low-equal)
    have ((ucast (get-CWP (cpu-reg-val PSR s2)), s2), False) = (fst (get-curr-win
() s2), snd (get-curr-win () s2))
      by (simp add: get-curr-win-def simpler-gets-def)
    then have f8: ucast (get-CWP (cpu-reg-val PSR s2)) = fst (fst (get-curr-win ()
s2)) ∧ s2 = snd (fst (get-curr-win () s2))
      by (metis prod.collapse prod.simps(1))
    then have f9: low-equal (snd (fst (get-curr-win () s1))) (snd (fst (get-curr-win
() s2)))
      using f6 ⟨low-equal s1 s2⟩ by (metis (no-types) prod.collapse prod.simps(1))
    have f10: ∀ s sa w wa. ¬ low-equal s sa ∨ user-reg-val (w::'a word) wa s =
user-reg-val w wa sa
      using user-reg-val-low-equal by blast
    have f11: get-operand2 (snd instr) s1 = get-operand2 (snd instr) (snd (fst
(get-curr-win () s2)))
      using f9 f6 by (metis (no-types) get-operand2-low-equal prod.collapse prod.simps(1))
    then have f12: uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd
instr) s2) = uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd
instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr)
s1)
      using f10 f9 f8 f7 by presburger
    then have f13: (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2
(snd instr) s1))::word64) ≠ (if fst instr = arith-type UMUL ∨ fst instr = arith-type
UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd in-

```

$str) s1)) \text{ else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr) s1))) \vee (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2))::word64) \neq (if fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)) else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr) s2))) \vee low-equal (snd (fst (write-cpu (ucast ((if fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1))::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr) s1))) >> 32)) Y (snd (fst (get-curr-win () s1)))) (snd (fst (write-cpu (ucast ((if fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2))::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr) s2))) >> 32)) Y (snd (fst (get-curr-win () s2))))))$

using $f9 f4$ **by** *presburger*

have $get-operand-w5 (snd instr ! 3) = 0 \wedge low-equal (snd (fst (write-cpu (ucast ((if fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1))::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr) s1))) >> 32)) Y (snd (fst (get-curr-win () s1)))) (snd (fst (write-cpu (ucast ((if fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2))::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr) s2))) >> 32)) Y (snd (fst (get-curr-win () s2)))))) \longrightarrow write-reg (if get-operand-w5 (snd instr ! 3) = 0 then user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((if fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2))::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr) s2))) >> 32)) Y (snd (fst (get-curr-win () s2)))))) else ucast (if fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2))::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr) s2))) >> 32)) Y (snd (fst (get-curr-win () s2))))))$


```

(user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0))
(snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64 else
word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd
instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr)
s1)))) (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd (fst
(write-cpu (ucast ((if fst instr = arith-type UMUL ∨ fst instr = arith-type UMULcc
then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd
instr) s1))::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win ()
s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint
(get-operand2 (snd instr) s1))) >> 32)) Y (snd (fst (get-curr-win () s1)))))))))
(snd (fst (mul-instr-sub1 (fst instr) (ucast (if fst instr = arith-type UMUL ∨ fst in-
str = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win
() s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2))))
* uint (get-operand2 (snd instr) s2))::word64 else word-of-int (sint (user-reg-val
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst
(get-curr-win () s2)))) * sint (get-operand2 (snd instr) s2)))) (snd (fst (write-reg
(if get-operand-w5 (snd instr ! 3) = 0 then user-reg-val (fst (fst (get-curr-win
() s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((if fst in-
str = arith-type UMUL ∨ fst instr = arith-type UMULcc then word-of-int (uint
(user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0))
(snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2))::word64
else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd
instr) s2))) >> 32)) Y (snd (fst (get-curr-win () s2)))))) else ucast (if fst in-
str = arith-type UMUL ∨ fst instr = arith-type UMULcc then word-of-int (uint
(user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0))
(snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2))::word64 else
word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr)
s2)))) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst
(write-cpu (ucast ((if fst instr = arith-type UMUL ∨ fst instr = arith-type UMULcc
then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd
instr) s2))::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win ()
s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint
(get-operand2 (snd instr) s2))) >> 32)) Y (snd (fst (get-curr-win () s2)))))))))
∨ fst instr ≠ arith-type UMULcc ∨ get-operand-w5 (snd instr ! 3) ≠ 0 ∨ (word-of-int
(uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc
0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1))::word64)
≠ (if fst instr = arith-type UMUL ∨ fst instr = arith-type UMULcc then word-of-int
(uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc
0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)) else
word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd
instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr)
s1))) ∨ (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd
instr) s2))::word64) ≠ (if fst instr = arith-type UMUL ∨ fst instr = arith-type
UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5

```

(snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr s2)) else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2)))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr s2)))

using f13 f12 f2 **by** fastforce

have f16: user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2))) = user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))

using f10 f9 f7 **by** presburger

{ **assume** fst instr \neq arith-type UMUL \vee low-equal (snd (fst (mul-instr-sub1 (arith-type UMUL) (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr s1)::word64)) (snd (fst (write-reg (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr s1)::word64)) (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr s1)::word64) >> 32)) Y (snd (fst (get-curr-win () s1)))))))))) (snd (fst (mul-instr-sub1 (arith-type UMUL) (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr s2)::word64)) (snd (fst (write-reg (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr s2)::word64)) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr s2)::word64) >> 32)) Y (snd (fst (get-curr-win () s2))))))))))

moreover

{ **assume** \neg low-equal (snd (fst (mul-instr-sub1 (arith-type UMULcc) (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr s1)::word64)) (snd (fst (write-reg (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr s1)::word64)) (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr s1)::word64) >> 32)) Y (snd (fst (get-curr-win () s1)))))))))) (snd (fst (mul-instr-sub1 (arith-type UMULcc) (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr s2)::word64)) (snd (fst (write-reg (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr s2)::word64)) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr s2)::word64) >> 32)) Y (snd (fst (get-curr-win () s2))))))))))

(get-curr-win () s2))))))))))

moreover

{ **assume** low-equal (snd (fst (mul-instr-sub1 (arith-type UMULcc) (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64)) (snd (fst (write-reg (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64)) (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64) >> 32)) Y (snd (fst (get-curr-win () s1)))))))))) (snd (fst (mul-instr-sub1 (arith-type UMULcc) (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64)) (snd (fst (write-reg (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64)) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64) >> 32)) Y (snd (fst (get-curr-win () s2)))))))))) ≠ low-equal (snd (fst (mul-instr-sub1 (fst instr) (ucast (if fst instr = arith-type UMUL ∨ fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr) s1)))) (snd (fst (write-reg (if get-operand-w5 (snd instr ! 3) = 0 then user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((if fst instr = arith-type UMUL ∨ fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr) s1))) >> 32)) Y (snd (fst (get-curr-win () s1)))))) else ucast (if fst instr = arith-type UMUL ∨ fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr) s1)))) (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((if fst instr = arith-type UMUL ∨ fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr) s1))) >> 32)) Y (snd (fst (get-curr-win () s1)))))))))) (snd (fst (mul-instr-sub1 (fst instr) (ucast (if fst instr = arith-type UMUL ∨ fst instr = arith-type UMULcc then word-of-int (uint

(user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr) s2))) >> 32)) Y (snd (fst (get-curr-win () s2)))))) ∨ fst instr ≠ arith-type UMULcc

by fastforce }

ultimately have write-reg (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64)) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64)) >> 32)) Y (snd (fst (get-curr-win () s2)))))) ≠ write-reg (if get-operand-w5 (snd instr ! 3) = 0 then user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((if fst instr = arith-type UMUL ∨ fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr) s2))) >> 32)) Y (snd (fst (get-curr-win () s2)))))) else ucast (if fst instr = arith-type UMUL ∨ fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr) s2)))) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((if fst instr = arith-type UMUL ∨ fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr) s2))) >> 32)) Y (snd (fst (get-curr-win () s2)))))) ∨ write-reg (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64)) (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64)) >> 32)) Y (snd (fst (get-curr-win () s1)))))) ≠ write-reg (if get-operand-w5 (snd instr ! 3) = 0 then user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((if fst instr = arith-type UMUL ∨ fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr) s1))) >> 32)) Y (snd (fst (get-curr-win () s1)))))) else ucast (if fst instr = arith-type UMUL ∨ fst instr = arith-type UMULcc then word-of-int (uint

*(if get-operand-w5 (snd instr ! 3) = 0 then user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((if fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2))::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr) s2))) >> 32)) Y (snd (fst (get-curr-win () s2)))))) else ucast (if fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2))::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr) s2))) >> 32)) Y (snd (fst (get-curr-win () s2))))))))) \vee fst instr \neq arith-type UMULcc
by fastforce }*

ultimately have *fst instr = arith-type UMULcc \wedge low-equal (snd (fst (mul-instr-sub1 (fst instr) (ucast (if fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1))::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr) s1))) (snd (fst (write-reg (if get-operand-w5 (snd instr ! 3) = 0 then user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((if fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1))::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr) s1))) >> 32)) Y (snd (fst (get-curr-win () s1)))))) else ucast (if fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1))::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr) s1))) (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((if fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1))::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr) s1))) >> 32)) Y (snd (fst (get-curr-win () s1))))))))) (snd (fst (mul-instr-sub1*


```

(snd (fst (get-curr-win () s1))) * sint (get-operand2 (snd instr) s1))) → get-operand-w5
(snd instr ! 3) = 0
  using f13 f7 f3 by fastforce }
moreover
  { assume mul-instr-sub1 (arith-type UMULcc) (if get-operand-w5 (snd instr !
3) = 0 then user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr !
3)) (snd (fst (write-cpu (ucast ((if fst instr = arith-type UMUL ∨ fst instr
= arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win
() s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))))
* uint (get-operand2 (snd instr) s1)::word64 else word-of-int (sint (user-reg-val
(fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst
(get-curr-win () s1)))) * sint (get-operand2 (snd instr) s1))) >> 32)) Y (snd
(fst (get-curr-win () s1)))))) else ucast (if fst instr = arith-type UMUL ∨ fst instr
= arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win
() s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))))
* uint (get-operand2 (snd instr) s1)::word64 else word-of-int (sint (user-reg-val
(fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst
(get-curr-win () s1)))) * sint (get-operand2 (snd instr) s1))) (snd (fst (write-reg
(if get-operand-w5 (snd instr ! 3) = 0 then user-reg-val (fst (fst (get-curr-win
() s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((if fst instr
= arith-type UMUL ∨ fst instr = arith-type UMULcc then word-of-int (uint
(user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0))
(snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64
else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd
instr) s2))) >> 32)) Y (snd (fst (get-curr-win () s2)))))) else ucast (if fst instr
= arith-type UMUL ∨ fst instr = arith-type UMULcc then word-of-int (uint
(user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0))
(snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64
else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd
instr) s2))) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd
(fst (write-cpu (ucast ((if fst instr = arith-type UMUL ∨ fst instr = arith-type UMULcc
then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd
instr) s2)::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win ()
s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint
(get-operand2 (snd instr) s2))) >> 32)) Y (snd (fst (get-curr-win () s2)))))))))
≠ mul-instr-sub1 (fst instr) (ucast (if fst instr = arith-type UMUL ∨ fst instr
= arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win
() s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2))))
* uint (get-operand2 (snd instr) s2)::word64 else word-of-int (sint (user-reg-val
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst
(get-curr-win () s2)))) * sint (get-operand2 (snd instr) s2))) (snd (fst (write-reg
(if get-operand-w5 (snd instr ! 3) = 0 then user-reg-val (fst (fst (get-curr-win
() s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((if fst instr
= arith-type UMUL ∨ fst instr = arith-type UMULcc then word-of-int (uint
(user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0))
(snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64
else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd
instr) s2))) >> 32)) Y (snd (fst (get-curr-win () s2)))))))))

```

*else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr s2))) >> 32)) Y (snd (fst (get-curr-win () s2)))))) else ucast (if fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr s2))::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr s2)))) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((if fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr s2))::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr s2))) >> 32)) Y (snd (fst (get-curr-win () s2))))))))))*

moreover

{ assume (if get-operand-w5 (snd instr ! 3) = 0 then user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((if fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr s1))::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr s1))) >> 32)) Y (snd (fst (get-curr-win () s1)))))) else ucast (if fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr s1))::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr s1)))) \neq ucast (if fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr s2))::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr s2))))

then have (if fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr s2))::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr s2))) = (if fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr s1)) else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr s1)))) \longrightarrow get-operand-w5 (snd instr ! 3) = 0

by (metis f11 f16 f8) }

ultimately have *fst instr = arith-type UMULcc \wedge (if fst instr = arith-type*

$(\text{snd instr ! Suc } 0)) (\text{snd (fst (get-curr-win () s2))}) * \text{sint (get-operand2 (snd instr s2))::word64}) (\text{snd (fst (write-reg (ucast (word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc } 0)) (snd (fst (get-curr-win () s2))}) * \text{sint (get-operand2 (snd instr s2))::word64}) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! } 3)) (snd (fst (write-cpu (ucast ((word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc } 0)) (snd (fst (get-curr-win () s2))}) * \text{sint (get-operand2 (snd instr s2))::word64}) >> } 32)) Y (snd (fst (get-curr-win () s2))))))))))$

by force

moreover

{ **assume** $\text{fst instr} \neq \text{arith-type UMULcc}$

{ **assume** $\text{fst instr} \neq \text{arith-type UMULcc} \wedge \text{low-equal (snd (fst (write-reg (if get-operand-w5 (snd instr ! } 3) = 0 \text{ then user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! } 3)) (snd (fst (write-cpu (ucast ((if fst instr = \text{arith-type UMUL} \vee \text{fst instr} = \text{arith-type UMULcc} \text{ then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc } 0)) (snd (fst (get-curr-win () s1))) * uint (get-operand2 (snd instr s1))::word64} \text{ else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc } 0)) (snd (fst (get-curr-win () s1))) * sint (get-operand2 (snd instr s1))) >> } 32)) Y (snd (fst (get-curr-win () s1)))))) \text{ else ucast (if fst instr = \text{arith-type UMUL} \vee \text{fst instr} = \text{arith-type UMULcc} \text{ then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc } 0)) (snd (fst (get-curr-win () s1))) * uint (get-operand2 (snd instr s1))::word64} \text{ else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc } 0)) (snd (fst (get-curr-win () s1))) * sint (get-operand2 (snd instr s1))) >> } 32)) Y (snd (fst (get-curr-win () s1))))))$

 $(\text{snd (fst (write-reg (if get-operand-w5 (snd instr ! } 3) = 0 \text{ then user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! } 3)) (snd (fst (write-cpu (ucast ((if fst instr = \text{arith-type UMUL} \vee \text{fst instr} = \text{arith-type UMULcc} \text{ then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc } 0)) (snd (fst (get-curr-win () s2))) * uint (get-operand2 (snd instr s2))::word64} \text{ else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc } 0)) (snd (fst (get-curr-win () s2))) * sint (get-operand2 (snd instr s2))) >> } 32)) Y (snd (fst (get-curr-win () s2)))))) \text{ else ucast (if fst instr = \text{arith-type UMUL} \vee \text{fst instr} = \text{arith-type UMULcc} \text{ then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc } 0)) (snd (fst (get-curr-win () s2))) * uint (get-operand2 (snd instr s2))::word64} \text{ else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc } 0)) (snd (fst (get-curr-win () s2))) * sint (get-operand2 (snd instr s2))) >> } 32)) Y (snd (fst (get-curr-win () s2))))))$

instr *s2*))::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () *s2*))) (get-operand-w5 (snd *instr* ! Suc 0)) (snd (fst (get-curr-win () *s2*)))) * sint (get-operand2 (snd *instr* *s2*))) >> 32)) Y (snd (fst (get-curr-win () *s2*))))))

moreover

{ **assume** *fst instr* ≠ arith-type UMULcc ∧ low-equal (snd (fst (write-reg (if get-operand-w5 (snd *instr* ! 3) = 0 then user-reg-val (fst (fst (get-curr-win () *s1*))) (get-operand-w5 (snd *instr* ! 3)) (snd (fst (write-cpu (ucast ((if *fst instr* = arith-type UMUL ∨ *fst instr* = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () *s1*))) (get-operand-w5 (snd *instr* ! Suc 0)) (snd (fst (get-curr-win () *s1*)))) * uint (get-operand2 (snd *instr* *s1*))::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () *s1*))) (get-operand-w5 (snd *instr* ! Suc 0)) (snd (fst (get-curr-win () *s1*)))) * sint (get-operand2 (snd *instr* *s1*))) >> 32)) Y (snd (fst (get-curr-win () *s1*)))))) else ucast (if *fst instr* = arith-type UMUL ∨ *fst instr* = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () *s1*))) (get-operand-w5 (snd *instr* ! Suc 0)) (snd (fst (get-curr-win () *s1*)))) * uint (get-operand2 (snd *instr* *s1*))::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () *s1*))) (get-operand-w5 (snd *instr* ! Suc 0)) (snd (fst (get-curr-win () *s1*)))) * sint (get-operand2 (snd *instr* *s1*))) (fst (fst (get-curr-win () *s1*))) (get-operand-w5 (snd *instr* ! 3)) (snd (fst (write-cpu (ucast ((if *fst instr* = arith-type UMUL ∨ *fst instr* = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () *s1*))) (get-operand-w5 (snd *instr* ! Suc 0)) (snd (fst (get-curr-win () *s1*)))) * uint (get-operand2 (snd *instr* *s1*))::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () *s1*))) (get-operand-w5 (snd *instr* ! Suc 0)) (snd (fst (get-curr-win () *s1*)))) * sint (get-operand2 (snd *instr* *s1*))) >> 32)) Y (snd (fst (get-curr-win () *s1*)))))) (snd (fst (write-reg (if get-operand-w5 (snd *instr* ! 3) = 0 then user-reg-val (fst (fst (get-curr-win () *s2*))) (get-operand-w5 (snd *instr* ! 3)) (snd (fst (write-cpu (ucast ((if *fst instr* = arith-type UMUL ∨ *fst instr* = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () *s2*))) (get-operand-w5 (snd *instr* ! Suc 0)) (snd (fst (get-curr-win () *s2*)))) * uint (get-operand2 (snd *instr* *s2*))::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () *s2*))) (get-operand-w5 (snd *instr* ! Suc 0)) (snd (fst (get-curr-win () *s2*)))) * sint (get-operand2 (snd *instr* *s2*))) >> 32)) Y (snd (fst (get-curr-win () *s2*)))))) else ucast (if *fst instr* = arith-type UMUL ∨ *fst instr* = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () *s2*))) (get-operand-w5 (snd *instr* ! Suc 0)) (snd (fst (get-curr-win () *s2*)))) * uint (get-operand2 (snd *instr* *s2*))::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () *s2*))) (get-operand-w5 (snd *instr* ! Suc 0)) (snd (fst (get-curr-win () *s2*)))) * sint (get-operand2 (snd *instr* *s2*))) (fst (fst (get-curr-win () *s2*))) (get-operand-w5 (snd *instr* ! 3)) (snd (fst (write-cpu (ucast ((if *fst instr* = arith-type UMUL ∨ *fst instr* = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () *s2*))) (get-operand-w5 (snd *instr* ! Suc 0)) (snd (fst (get-curr-win () *s2*)))) * uint (get-operand2 (snd *instr* *s2*))::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () *s2*))) (get-operand-w5 (snd *instr* ! Suc 0)) (snd (fst (get-curr-win () *s2*)))) * uint (get-operand2 (snd *instr* *s2*))) >> 32)) Y (snd (fst (get-curr-win () *s2*)))))) ∧ snd (fst (mul-instr-sub1 (fst *instr*) (ucast (if *fst instr* = arith-type UMUL ∨ *fst instr* = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () *s2*))) (get-operand-w5 (snd *instr* ! Suc 0)) (snd (fst (get-curr-win () *s2*)))) * uint (get-operand2 (snd *instr* *s2*))::word64 else word-of-int (sint (user-reg-val


```

(snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd
instr) s1)::word64)) (snd (fst (write-reg (ucast (word-of-int (uint (user-reg-val
(fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst
(get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64)) (fst (fst
(get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast
((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd
instr) s1)::word64) >> 32)) Y (snd (fst (get-curr-win () s1)))))))))) (snd
(fst (mul-instr-sub1 (arith-type UMULcc) (ucast (word-of-int (uint (user-reg-val
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst
(get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64)) (snd (fst
(write-reg (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2
(snd instr) s2)::word64)) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! 3)) (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst
(get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win
() s2)))) * uint (get-operand2 (snd instr) s2)::word64) >> 32)) Y (snd (fst
(get-curr-win () s2)))))))))) ^ (fst instr = arith-type UMUL ∨ fst instr =
arith-type UMULcc ∨ low-equal (snd (fst (mul-instr-sub1 (fst instr) (ucast (word-of-int
(sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc
0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr) s1)::word64))
(snd (fst (write-reg (ucast (word-of-int (sint (user-reg-val (fst (fst (get-curr-win
() s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))))
* sint (get-operand2 (snd instr) s1)::word64)) (fst (fst (get-curr-win () s1)))
(get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((word-of-int (sint
(user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0))
(snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr) s1)::word64) >>
32)) Y (snd (fst (get-curr-win () s1)))))))))) (snd (fst (mul-instr-sub1 (fst instr)
(ucast (word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd
instr) s2)::word64)) (snd (fst (write-reg (ucast (word-of-int (sint (user-reg-val
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst
(get-curr-win () s2)))) * sint (get-operand2 (snd instr) s2)::word64)) (fst (fst
(get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast
((word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd in-
str) s2)::word64) >> 32)) Y (snd (fst (get-curr-win () s2)))))))))) ∨ get-operand-w5
(snd instr ! 3) = 0

```

using f2 by presburger }

ultimately have $\text{fst instr} \neq \text{arith-type UMULcc} \wedge (\text{if } \text{get-operand-w5} (\text{snd instr} ! 3) = 0 \text{ then } \text{user-reg-val} (\text{fst} (\text{fst} (\text{get-curr-win} () s1))) (\text{get-operand-w5} (\text{snd instr} ! 3)) (\text{snd} (\text{fst} (\text{write-cpu} (\text{ucast} ((\text{if } \text{fst instr} = \text{arith-type UMUL} \vee \text{fst instr} = \text{arith-type UMULcc} \text{ then } \text{word-of-int} (\text{uint} (\text{user-reg-val} (\text{fst} (\text{fst} (\text{get-curr-win} () s1))) (\text{get-operand-w5} (\text{snd instr} ! \text{Suc } 0)) (\text{snd} (\text{fst} (\text{get-curr-win} () s1)))) * \text{uint} (\text{get-operand2} (\text{snd instr} s1)::\text{word64} \text{ else } \text{word-of-int} (\text{sint} (\text{user-reg-val} (\text{fst} (\text{fst} (\text{get-curr-win} () s1))) (\text{get-operand-w5} (\text{snd instr} ! \text{Suc } 0)) (\text{snd} (\text{fst} (\text{get-curr-win} () s1)))) * \text{sint} (\text{get-operand2} (\text{snd instr} s1))) >> 32)) Y (\text{snd} (\text{fst} (\text{get-curr-win} () s1)))))) \text{ else } \text{ucast} (\text{if } \text{fst instr} = \text{arith-type UMUL} \vee \text{fst instr} = \text{arith-type UMULcc} \text{ then } \text{word-of-int} (\text{uint} (\text{user-reg-val} (\text{fst} (\text{fst} (\text{get-curr-win} () s1))) (\text{get-operand-w5} (\text{snd instr} ! 3)) (\text{snd} (\text{fst} (\text{write-cpu} (\text{ucast} ((\text{word-of-int} (\text{uint} (\text{user-reg-val} (\text{fst} (\text{fst} (\text{get-curr-win} () s1))) (\text{get-operand-w5} (\text{snd instr} ! \text{Suc } 0)) (\text{snd} (\text{fst} (\text{get-curr-win} () s1)))) * \text{uint} (\text{get-operand2} (\text{snd instr} s1)::\text{word64} \text{ else } \text{word-of-int} (\text{sint} (\text{user-reg-val} (\text{fst} (\text{fst} (\text{get-curr-win} () s1))) (\text{get-operand-w5} (\text{snd instr} ! \text{Suc } 0)) (\text{snd} (\text{fst} (\text{get-curr-win} () s1)))) * \text{sint} (\text{get-operand2} (\text{snd instr} s1))) >> 32)) Y (\text{snd} (\text{fst} (\text{get-curr-win} () s1)))))) \vee \text{get-operand-w5} (\text{snd instr} ! 3) = 0$

\neq arith-type UMULcc \vee low-equal (snd (fst (mul-instr-sub1 (arith-type UMULcc) (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr s1)::word64)) (snd (fst (write-reg (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr s1)::word64)) (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr s1)::word64) >> 32)) Y (snd (fst (get-curr-win () s1)))))))))) (snd (fst (mul-instr-sub1 (arith-type UMULcc) (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr s2)::word64)) (snd (fst (write-reg (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr s2)::word64)) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr s2)::word64) >> 32)) Y (snd (fst (get-curr-win () s2))))))))))))) \wedge (fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc \vee low-equal (snd (fst (mul-instr-sub1 (fst instr) (ucast (word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr s1)::word64)) (snd (fst (write-reg (ucast (word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr s1)::word64)) (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr s1)::word64) >> 32)) Y (snd (fst (get-curr-win () s1)))))))))) (snd (fst (mul-instr-sub1 (fst instr) (ucast (word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr s2)::word64)) (snd (fst (write-reg (ucast (word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr s2)::word64)) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr s2)::word64) >> 32)) Y (snd (fst (get-curr-win () s2))))))))))))) \vee get-operand-w5 (snd instr ! 3) = 0

by fastforce }

then have (get-operand-w5 (snd instr ! 3) = 0 \vee (fst instr \neq arith-type UMUL \vee low-equal (snd (fst (mul-instr-sub1 (arith-type UMUL) (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr s1)::word64)) (snd (fst (write-reg (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr s1)::word64)) (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr s1)::word64) >> 32)) Y (snd (fst (get-curr-win () s1)))))))))))))


```

(snd instr ! Suc 0) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr s2)::word64) >> 32)) Y (snd (fst (get-curr-win () s2)))))) ∨ get-operand-w5
(snd instr ! 3) = 0
  using f16 f11 f9 f8 f7 f4 f3 f2 by force }
moreover
{ assume get-operand-w5 (snd instr ! 3) = 0
  moreover
  { assume (fst instr = arith-type UMUL ∨ fst instr = arith-type UMULcc) ∧
    get-operand-w5 (snd instr ! 3) = 0
    moreover
    { assume ((fst instr = arith-type UMUL ∨ fst instr = arith-type UMULcc)
      ∧ get-operand-w5 (snd instr ! 3) = 0) ∧ ¬ low-equal (snd (fst (mul-instr-sub1
      (arith-type UMUL) (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win
      () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))))
      * uint (get-operand2 (snd instr) s1)::word64)) (snd (fst (write-reg (user-reg-val
      (fst (fst (get-curr-win () s1))) 0 (snd (fst (write-cpu (ucast ((word-of-int (uint
      (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0))
      (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64)
      >> 32)) Y (snd (fst (get-curr-win () s1)))))) (fst (fst (get-curr-win () s1))) 0
      (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win
      () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) *
      uint (get-operand2 (snd instr) s1)::word64) >> 32)) Y (snd (fst (get-curr-win
      () s1)))))) (snd (fst (mul-instr-sub1 (arith-type UMUL) (ucast (word-of-int
      (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc
      0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64))
      (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s2))) 0 (snd (fst (write-cpu
      (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
      (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr)
      s2)::word64) >> 32)) Y (snd (fst (get-curr-win () s2)))))) (fst (fst (get-curr-win
      () s2))) 0 (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst
      (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win
      () s2)))) * uint (get-operand2 (snd instr) s2)::word64) >> 32)) Y (snd (fst
      (get-curr-win () s2)))))))))
    moreover
    { assume ((fst instr = arith-type UMUL ∨ fst instr = arith-type UMULcc) ∧
      get-operand-w5 (snd instr ! 3) = 0) ∧ low-equal (snd (fst (mul-instr-sub1 (arith-type
      UMUL) (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5
      (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr)
      s1)::word64)) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s1)))
      0 (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win
      () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) *
      uint (get-operand2 (snd instr) s1)::word64) >> 32)) Y (snd (fst (get-curr-win ()
      s1)))))) (fst (fst (get-curr-win () s1))) 0 (snd (fst (write-cpu (ucast ((word-of-int
      (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc
      0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64)
      >> 32)) Y (snd (fst (get-curr-win () s1)))))) (snd (fst (mul-instr-sub1 (arith-type
      UMUL) (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
      (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr)
      s2)::word64)) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s2)))
      0 (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win
      () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) *
      uint (get-operand2 (snd instr) s2)::word64) >> 32)) Y (snd (fst (get-curr-win
      () s2)))))))))

```

0 (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64) >> 32)) Y (snd (fst (get-curr-win () s2)))))) (fst (fst (get-curr-win () s2))) 0 (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64) >> 32)) Y (snd (fst (get-curr-win () s2))))))))) \neq low-equal (snd (fst (mul-instr-sub1 (arith-type UMULcc) (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64)) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s1))) 0 (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64) >> 32)) Y (snd (fst (get-curr-win () s1)))))) (fst (fst (get-curr-win () s1))) 0 (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64) >> 32)) Y (snd (fst (get-curr-win () s1))))))))) (snd (fst (mul-instr-sub1 (arith-type UMULcc) (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64)) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s2))) 0 (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64) >> 32)) Y (snd (fst (get-curr-win () s2)))))) (fst (fst (get-curr-win () s2))) 0 (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64) >> 32)) Y (snd (fst (get-curr-win () s2)))))))))

then have ((fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc) \wedge get-operand-w5 (snd instr ! 3) = 0) \wedge arith-type UMUL \neq arith-type UMULcc \vee fst instr \neq arith-type UMULcc \wedge (fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc) \wedge get-operand-w5 (snd instr ! 3) = 0

by fastforce

then have ((fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc) \wedge get-operand-w5 (snd instr ! 3) = 0) \wedge (fst instr \neq arith-type UMUL \vee low-equal (snd (fst (mul-instr-sub1 (arith-type UMUL) (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64)) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s1))) 0 (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64) >> 32)) Y (snd (fst (get-curr-win () s1)))))) (fst (fst (get-curr-win () s1))) 0 (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64) >> 32)) Y (snd (fst (get-curr-win () s1))))))))) (snd (fst (mul-instr-sub1 (arith-type UMUL) (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64)) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s2))) 0

$str ! 3)) (snd (fst (write-cpu (ucast ((if\ fst\ instr = arith-type\ UMUL \vee\ fst\ instr = arith-type\ UMULcc\ then\ word-of-int\ (uint\ (user-reg-val\ (fst\ (fst\ (get-curr-win\ ()\ s1)))) (get-operand-w5\ (snd\ instr\ !\ Suc\ 0)) (snd\ (fst\ (get-curr-win\ ()\ s1)))) * uint\ (get-operand2\ (snd\ instr\ s1))::word64\ else\ word-of-int\ (sint\ (user-reg-val\ (fst\ (fst\ (get-curr-win\ ()\ s1)))) (get-operand-w5\ (snd\ instr\ !\ Suc\ 0)) (snd\ (fst\ (get-curr-win\ ()\ s1)))) * sint\ (get-operand2\ (snd\ instr\ s1))) >> 32)) Y (snd\ (fst\ (get-curr-win\ ()\ s1)))))) else\ ucast\ (if\ fst\ instr = arith-type\ UMUL \vee\ fst\ instr = arith-type\ UMULcc\ then\ word-of-int\ (uint\ (user-reg-val\ (fst\ (fst\ (get-curr-win\ ()\ s1)))) (get-operand-w5\ (snd\ instr\ !\ Suc\ 0)) (snd\ (fst\ (get-curr-win\ ()\ s1)))) * uint\ (get-operand2\ (snd\ instr\ s1))::word64\ else\ word-of-int\ (sint\ (user-reg-val\ (fst\ (fst\ (get-curr-win\ ()\ s1)))) (get-operand-w5\ (snd\ instr\ !\ Suc\ 0)) (snd\ (fst\ (get-curr-win\ ()\ s1)))) * sint\ (get-operand2\ (snd\ instr\ s1)))) (fst\ (fst\ (get-curr-win\ ()\ s1))) (get-operand-w5\ (snd\ instr\ !\ 3)) (snd\ (fst\ (write-cpu\ (ucast\ ((if\ fst\ instr = arith-type\ UMUL \vee\ fst\ instr = arith-type\ UMULcc\ then\ word-of-int\ (uint\ (user-reg-val\ (fst\ (fst\ (get-curr-win\ ()\ s1)))) (get-operand-w5\ (snd\ instr\ !\ Suc\ 0)) (snd\ (fst\ (get-curr-win\ ()\ s1)))) * uint\ (get-operand2\ (snd\ instr\ s1))::word64\ else\ word-of-int\ (sint\ (user-reg-val\ (fst\ (fst\ (get-curr-win\ ()\ s1)))) (get-operand-w5\ (snd\ instr\ !\ Suc\ 0)) (snd\ (fst\ (get-curr-win\ ()\ s1)))) * sint\ (get-operand2\ (snd\ instr\ s1))) >> 32)) Y (snd\ (fst\ (get-curr-win\ ()\ s1)))))))))) (snd\ (fst\ (mul-instr-sub1\ (fst\ instr)\ (ucast\ (if\ fst\ instr = arith-type\ UMUL \vee\ fst\ instr = arith-type\ UMULcc\ then\ word-of-int\ (uint\ (user-reg-val\ (fst\ (fst\ (get-curr-win\ ()\ s2)))) (get-operand-w5\ (snd\ instr\ !\ Suc\ 0)) (snd\ (fst\ (get-curr-win\ ()\ s2)))) * uint\ (get-operand2\ (snd\ instr\ s2))::word64\ else\ word-of-int\ (sint\ (user-reg-val\ (fst\ (fst\ (get-curr-win\ ()\ s2)))) (get-operand-w5\ (snd\ instr\ !\ Suc\ 0)) (snd\ (fst\ (get-curr-win\ ()\ s2)))) * sint\ (get-operand2\ (snd\ instr\ s2)))) (snd\ (fst\ (write-reg\ (if\ get-operand-w5\ (snd\ instr\ !\ 3) = 0\ then\ user-reg-val\ (fst\ (fst\ (get-curr-win\ ()\ s2)))) (get-operand-w5\ (snd\ instr\ !\ 3)) (snd\ (fst\ (write-cpu\ (ucast\ ((if\ fst\ instr = arith-type\ UMUL \vee\ fst\ instr = arith-type\ UMULcc\ then\ word-of-int\ (uint\ (user-reg-val\ (fst\ (fst\ (get-curr-win\ ()\ s2)))) (get-operand-w5\ (snd\ instr\ !\ Suc\ 0)) (snd\ (fst\ (get-curr-win\ ()\ s2)))) * uint\ (get-operand2\ (snd\ instr\ s2))::word64\ else\ word-of-int\ (sint\ (user-reg-val\ (fst\ (fst\ (get-curr-win\ ()\ s2)))) (get-operand-w5\ (snd\ instr\ !\ Suc\ 0)) (snd\ (fst\ (get-curr-win\ ()\ s2)))) * sint\ (get-operand2\ (snd\ instr\ s2))) >> 32)) Y (snd\ (fst\ (get-curr-win\ ()\ s2)))))) else\ ucast\ (if\ fst\ instr = arith-type\ UMUL \vee\ fst\ instr = arith-type\ UMULcc\ then\ word-of-int\ (uint\ (user-reg-val\ (fst\ (fst\ (get-curr-win\ ()\ s2)))) (get-operand-w5\ (snd\ instr\ !\ Suc\ 0)) (snd\ (fst\ (get-curr-win\ ()\ s2)))) * uint\ (get-operand2\ (snd\ instr\ s2))::word64\ else\ word-of-int\ (sint\ (user-reg-val\ (fst\ (fst\ (get-curr-win\ ()\ s2)))) (get-operand-w5\ (snd\ instr\ !\ Suc\ 0)) (snd\ (fst\ (get-curr-win\ ()\ s2)))) * sint\ (get-operand2\ (snd\ instr\ s2)))) (fst\ (fst\ (get-curr-win\ ()\ s2))) (get-operand-w5\ (snd\ instr\ !\ 3)) (snd\ (fst\ (write-cpu\ (ucast\ ((if\ fst\ instr = arith-type\ UMUL \vee\ fst\ instr = arith-type\ UMULcc\ then\ word-of-int\ (uint\ (user-reg-val\ (fst\ (fst\ (get-curr-win\ ()\ s2)))) (get-operand-w5\ (snd\ instr\ !\ Suc\ 0)) (snd\ (fst\ (get-curr-win\ ()\ s2)))) * uint\ (get-operand2\ (snd\ instr\ s2))::word64\ else\ word-of-int\ (sint\ (user-reg-val\ (fst\ (fst\ (get-curr-win\ ()\ s2)))) (get-operand-w5\ (snd\ instr\ !\ Suc\ 0)) (snd\ (fst\ (get-curr-win\ ()\ s2)))) * sint\ (get-operand2\ (snd\ instr\ s2))) >> 32)) Y (snd\ (fst\ (get-curr-win\ ()\ s2)))))) \vee\ fst\ instr \neq\ arith-type\ UMULcc \wedge\ (fst\ instr = arith-type\ UMUL \vee\ fst\ instr = arith-type\ UMULcc) \wedge\ get-operand-w5\ (snd\ instr\ !\ 3) = 0 \vee\ (get-operand-w5\ (snd\ instr\ !\ 3) = 0 \vee\ (fst\ instr \neq\ arith-type\ UMUL \vee\ low-equal\ (snd\ (fst\ (mul-instr-sub1\ (arith-type\ UMUL)\ (ucast\ (word-of-int\ (uint\ (user-reg-val\ (fst\ (fst\ (get-curr-win\ ()\ s1)))) (get-operand-w5$

$\gg 32)) Y (snd (fst (get-curr-win () s2))))))))) \wedge (fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc \vee low-equal (snd (fst (mul-instr-sub1 (fst instr) (ucast (word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr) s1)::word64)) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s1))) 0 (snd (fst (write-cpu (ucast ((word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr) s1)::word64)) \gg 32)) Y (snd (fst (get-curr-win () s1))))))))) (fst (fst (get-curr-win () s1))) 0 (snd (fst (write-cpu (ucast ((word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr) s1)::word64)) \gg 32)) Y (snd (fst (get-curr-win () s1))))))))) (snd (fst (mul-instr-sub1 (fst instr) (ucast (word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr) s2)::word64)) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s2))) 0 (snd (fst (write-cpu (ucast ((word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr) s2)::word64)) \gg 32)) Y (snd (fst (get-curr-win () s2))))))))) (fst (fst (get-curr-win () s2))) 0 (snd (fst (write-cpu (ucast ((word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr) s2)::word64)) \gg 32)) Y (snd (fst (get-curr-win () s2)))))))))$

by auto

then have $fst instr \neq arith-type UMULcc \wedge (fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc) \wedge get-operand-w5 (snd instr ! 3) = 0 \vee (get-operand-w5 (snd instr ! 3) = 0 \vee (fst instr \neq arith-type UMUL \vee low-equal (snd (fst (mul-instr-sub1 (arith-type UMUL) (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64)) (snd (fst (write-reg (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64)) (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64)) \gg 32)) Y (snd (fst (get-curr-win () s1))))))))) (snd (fst (mul-instr-sub1 (arith-type UMUL) (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64)) (snd (fst (write-reg (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64)) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64)) \gg 32)) Y (snd (fst (get-curr-win () s2))))))))) \wedge (fst instr \neq arith-type UMULcc \vee low-equal (snd (fst (mul-instr-sub1 (arith-type UMULcc) (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64)) (snd (fst (write-reg (ucast (word-of-int$

```

(uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc
0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64))
(fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu
(ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd
instr) s1)::word64) >> 32)) Y (snd (fst (get-curr-win () s1)))))))))) (snd
(fst (mul-instr-sub1 (arith-type UMULcc) (ucast (word-of-int (uint (user-reg-val
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst
(get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64)) (snd (fst
(write-reg (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2
(snd instr) s2)::word64)) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! 3)) (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst
(get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win
() s2)))) * uint (get-operand2 (snd instr) s2)::word64) >> 32)) Y (snd (fst
(get-curr-win () s2))))))))))))) ^ (fst instr = arith-type UMUL ∨ fst instr =
arith-type UMULcc ∨ low-equal (snd (fst (mul-instr-sub1 (fst instr) (ucast (word-of-int
(sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc
0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr) s1)::word64))
(snd (fst (write-reg (ucast (word-of-int (sint (user-reg-val (fst (fst (get-curr-win
() s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))))
* sint (get-operand2 (snd instr) s1)::word64)) (fst (fst (get-curr-win () s1)))
(get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((word-of-int (sint
(user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0))
(snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr) s1)::word64) >>
32)) Y (snd (fst (get-curr-win () s1))))))))))))) (snd (fst (mul-instr-sub1 (fst instr)
(ucast (word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd
instr) s2)::word64)) (snd (fst (write-reg (ucast (word-of-int (sint (user-reg-val
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst
(get-curr-win () s2)))) * sint (get-operand2 (snd instr) s2)::word64)) (fst (fst
(get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast
((word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd in-
str) s2)::word64) >> 32)) Y (snd (fst (get-curr-win () s2))))))))))))) ^ (get-operand-w5
(snd instr ! 3) ≠ 0 ∨ (fst instr ≠ arith-type UMUL ∨ low-equal (snd (fst (mul-instr-sub1
(arith-type UMUL) (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win
() s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))))
* uint (get-operand2 (snd instr) s1)::word64)) (snd (fst (write-reg (user-reg-val
(fst (fst (get-curr-win () s1))) 0 (snd (fst (write-cpu (ucast ((word-of-int (uint
(user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0))
(snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64)
>> 32)) Y (snd (fst (get-curr-win () s1))))))))) (fst (fst (get-curr-win () s1))) 0
(snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win
() s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) *
uint (get-operand2 (snd instr) s1)::word64) >> 32)) Y (snd (fst (get-curr-win
() s1))))))))))))) (snd (fst (mul-instr-sub1 (arith-type UMUL) (ucast (word-of-int
(uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc
0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64))

```



```

sint (get-operand2 (snd instr) s2)::word64) >> 32)) Y (snd (fst (get-curr-win ()
s2))))))))))))))
  using f15 by presburger
  then have (get-operand-w5 (snd instr ! 3) = 0 ∨ (fst instr ≠ arith-type UMUL
∨ low-equal (snd (fst (mul-instr-sub1 (arith-type UMUL) (ucast (word-of-int (uint
(user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0))
(snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64))
(snd (fst (write-reg (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win
() s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))))
* uint (get-operand2 (snd instr) s1)::word64)) (fst (fst (get-curr-win () s1)))
(get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((word-of-int (uint
(user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0))
(snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64) >>
32)) Y (snd (fst (get-curr-win () s1)))))))))))))) (snd (fst (mul-instr-sub1 (arith-type
UMUL) (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd
instr) s2)::word64)) (snd (fst (write-reg (ucast (word-of-int (uint (user-reg-val
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst
(get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64)) (fst (fst
(get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast
((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd in-
str) s2)::word64) >> 32)) Y (snd (fst (get-curr-win () s2)))))))))))))) ∧ (fst instr
≠ arith-type UMULcc ∨ low-equal (snd (fst (mul-instr-sub1 (arith-type UMULcc)
(ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd
instr) s1)::word64)) (snd (fst (write-reg (ucast (word-of-int (uint (user-reg-val
(fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst
(get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64)) (fst (fst
(get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast
((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd
instr) s1)::word64) >> 32)) Y (snd (fst (get-curr-win () s1)))))))))))))) (snd
(fst (mul-instr-sub1 (arith-type UMULcc) (ucast (word-of-int (uint (user-reg-val
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst
(get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64)) (snd (fst
(write-reg (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2
(snd instr) s2)::word64)) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! 3)) (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst
(get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win
() s2)))) * uint (get-operand2 (snd instr) s2)::word64) >> 32)) Y (snd (fst
(get-curr-win () s2)))))))))))))) ∧ (fst instr = arith-type UMUL ∨ fst instr =
arith-type UMULcc ∨ low-equal (snd (fst (mul-instr-sub1 (fst instr) (ucast (word-of-int
(sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc
0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr) s1)::word64))
(snd (fst (write-reg (ucast (word-of-int (sint (user-reg-val (fst (fst (get-curr-win
() s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))))
* sint (get-operand2 (snd instr) s1)::word64)) (fst (fst (get-curr-win () s1)))

```


$str) s2)::word64) \gg 32)) Y (snd (fst (get-curr-win () s2))))))))) \wedge (fst instr \neq arith-type UMULcc \vee low-equal (snd (fst (mul-instr-sub1 (arith-type UMULcc) (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64)) (snd (fst (write-reg (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64)) (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64) \gg 32)) Y (snd (fst (get-curr-win () s1))))))))) (snd (fst (mul-instr-sub1 (arith-type UMULcc) (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64)) (snd (fst (write-reg (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64)) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64) \gg 32)) Y (snd (fst (get-curr-win () s2))))))))) \wedge (fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc \vee low-equal (snd (fst (mul-instr-sub1 (fst instr) (ucast (word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr) s1)::word64)) (snd (fst (write-reg (ucast (word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr) s1)::word64)) (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr) s1)::word64) \gg 32)) Y (snd (fst (get-curr-win () s1))))))))) (snd (fst (mul-instr-sub1 (fst instr) (ucast (word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr) s2)::word64)) (snd (fst (write-reg (ucast (word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr) s2)::word64)) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr) s2)::word64) \gg 32)) Y (snd (fst (get-curr-win () s2))))))))) \wedge (get-operand-w5 (snd instr ! 3) \neq 0 \vee (fst instr \neq arith-type UMUL \vee low-equal (snd (fst (mul-instr-sub1 (arith-type UMUL) (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64)) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s1))) 0 (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64) \gg 32)) Y (snd (fst (get-curr-win () s1))))))))) (fst (fst (get-curr-win () s1))) 0 (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst (get-curr-win$


```

(user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0))
(snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr) s2)::word64
>> 32)) Y (snd (fst (get-curr-win () s2)))))) (fst (fst (get-curr-win () s2))) 0
(snd (fst (write-cpu (ucast ((word-of-int (sint (user-reg-val (fst (fst (get-curr-win
() s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) *
sint (get-operand2 (snd instr) s2)::word64 >> 32)) Y (snd (fst (get-curr-win ()
s2))))))))))
using f16 f14 f11 f9 f8 f4 f2 by fastforce }
ultimately have (get-operand-w5 (snd instr ! 3) ≠ 0 → (fst instr = arith-type
UMUL → low-equal (snd (fst (mul-instr-sub1 (arith-type UMUL) (ucast (word-of-int
(uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc
0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64))
(snd (fst (write-reg (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win
() s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))))
* uint (get-operand2 (snd instr) s1)::word64)) (fst (fst (get-curr-win () s1)))
(get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((word-of-int (uint
(user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0))
(snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64 >>
32)) Y (snd (fst (get-curr-win () s1)))))))))) (snd (fst (mul-instr-sub1 (arith-type
UMUL) (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd
instr) s2)::word64)) (snd (fst (write-reg (ucast (word-of-int (uint (user-reg-val
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst
(get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64)) (fst (fst
(get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast
((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd
instr) s2)::word64 >> 32)) Y (snd (fst (get-curr-win () s2)))))))))) ∧ (fst
instr = arith-type UMULcc → low-equal (snd (fst (mul-instr-sub1 (arith-type
UMULcc) (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2
(snd instr) s1)::word64)) (snd (fst (write-reg (ucast (word-of-int (uint (user-reg-val
(fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst
(get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64)) (fst (fst
(get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast
((word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd
instr) s1)::word64 >> 32)) Y (snd (fst (get-curr-win () s1)))))))))) (snd
(fst (mul-instr-sub1 (arith-type UMULcc) (ucast (word-of-int (uint (user-reg-val
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst
(get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64)) (snd (fst
(write-reg (ucast (word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2
(snd instr) s2)::word64)) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! 3)) (snd (fst (write-cpu (ucast ((word-of-int (uint (user-reg-val (fst (fst
(get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win
() s2)))) * uint (get-operand2 (snd instr) s2)::word64 >> 32)) Y (snd (fst
(get-curr-win () s2)))))))))) ∧ (fst instr ≠ arith-type UMUL ∧ fst instr ≠
arith-type UMULcc → low-equal (snd (fst (mul-instr-sub1 (fst instr) (ucast (word-of-int

```


(ucast ((if fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr) s1))) >> 32)) Y (snd (fst (get-curr-win () s1)))))) (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((if fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * uint (get-operand2 (snd instr) s1)::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s1))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))) * sint (get-operand2 (snd instr) s1))) >> 32)) Y (snd (fst (get-curr-win () s1))))))))))

by (simp add: mul-instr-def) (simp add: simpler-gets-def bind-def h1-def h2-def Let-def case-prod-unfold)

moreover from t2 have

\langle t2 = snd (fst (mul-instr-sub1 (fst instr) (ucast (if fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr) s2)))) (snd (fst (write-reg (if get-operand-w5 (snd instr ! 3) \neq 0 then ucast (if fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr) s2))) else user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((if fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr) s2))) >> 32)) Y (snd (fst (get-curr-win () s2)))))) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3)) (snd (fst (write-cpu (ucast ((if fst instr = arith-type UMUL \vee fst instr = arith-type UMULcc then word-of-int (uint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * uint (get-operand2 (snd instr) s2)::word64 else word-of-int (sint (user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))) * sint (get-operand2 (snd instr) s2))) >> 32)) Y (snd (fst (get-curr-win () s2))))))))))

by (simp add: mul-instr-def) (simp add: simpler-gets-def bind-def h1-def h2-def Let-def case-prod-unfold)

ultimately show ?thesis

by (simp add: mul-instr-def)

qed

lemma div-write-new-val-low-equal:

```

assumes a1: low-equal s1 s2  $\wedge$ 
t1 = snd (fst (div-write-new-val i result temp-V s1))  $\wedge$ 
t2 = snd (fst (div-write-new-val i result temp-V s2))
shows low-equal t1 t2
proof (cases (fst i)  $\in$  {arith-type UDIVcc,arith-type SDIVcc})
  case True
  then show ?thesis using a1
  apply (simp add: div-write-new-val-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
  apply (clarsimp simp add: cpu-reg-val-low-equal)
  using write-cpu-low-equal by blast
next
  case False
  then show ?thesis using a1
  apply (simp add: div-write-new-val-def)
  by (simp add: return-def)
qed

```

```

lemma div-comp-low-equal:
assumes a1: low-equal s1 s2  $\wedge$ 
t1 = snd (fst (div-comp instr rs1 rd operand2 s1))  $\wedge$ 
t2 = snd (fst (div-comp instr rs1 rd operand2 s2))
shows low-equal t1 t2
using a1
apply (simp add: div-comp-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
apply (simp add: case-prod-unfold)
apply (clarsimp simp add: get-curr-win-low-equal)
proof –
  assume a1: low-equal s1 s2
  have f2:  $\forall s$  sa sb w wa wb sc.  $\neg$  low-equal s sa  $\vee$  sb  $\neq$  snd (fst (write-reg w
(wa::'a word) wb s))  $\vee$  sc  $\neq$  snd (fst (write-reg w wa wb sa))  $\vee$  low-equal sb sc
  by (meson write-reg-low-equal)
  have f3: gets ( $\lambda s$ . ucast (get-CWP (cpu-reg-val PSR s))::'a word) = get-curr-win
()
  by (simp add: get-curr-win-def)
  then have ((ucast (get-CWP (cpu-reg-val PSR s1)), s1), False) = (fst (get-curr-win
() s1), snd (get-curr-win () s1))
  by (metis (no-types) prod.collapse simpler-gets-def)
  then have f4: ucast (get-CWP (cpu-reg-val PSR s1)) = fst (fst (get-curr-win ()
s1))  $\wedge$  s1 = snd (fst (get-curr-win () s1))
  by (metis prod.collapse prod.simps(1))
  have ((ucast (get-CWP (cpu-reg-val PSR s2)), s2), False) = (fst (get-curr-win
() s2), snd (get-curr-win () s2))
  using f3 by (metis (no-types) prod.collapse simpler-gets-def)
  then have f5: ucast (get-CWP (cpu-reg-val PSR s2)) = fst (fst (get-curr-win ()
s2))  $\wedge$  s2 = snd (fst (get-curr-win () s2))
  by (metis (no-types) prod.collapse prod.simps(1))
  then have f6: low-equal (snd (fst (get-curr-win () s1))) (snd (fst (get-curr-win

```

```

() s2)))
  using f4 a1 by presburger
  have f7:  $\forall s \ sa \ sb \ p \ w \ wa \ sc. \neg \text{low-equal} (s::'a \ \text{sparc-state}) \ sa \ \vee \ sb \neq \ \text{snd} (fst (div-write-new-val \ p \ w \ wa \ s)) \ \vee \ sc \neq \ \text{snd} (fst (div-write-new-val \ p \ w \ wa \ sa)) \ \vee \ \text{low-equal} \ sb \ sc$ 
    by (meson div-write-new-val-low-equal)
  have f8: cpu-reg-val PSR s2 = cpu-reg-val PSR s1
    using a1 by (simp add: cpu-reg-val-def low-equal-def)
  then have fst (fst (get-curr-win () s2)) = ucast (get-CWP (cpu-reg-val PSR s1))
    using f5 by presburger
  then have f9: fst (fst (get-curr-win () s2)) = fst (fst (get-curr-win () s1))
    using f4 by presburger
  have f10: fst (fst (get-curr-win () s1)) = fst (fst (get-curr-win () s2))
    using f8 f5 f4 by presburger
  have f11: (word-cat (cpu-reg-val Y (snd (fst (get-curr-win () s2)))) (user-reg-val (fst (fst (get-curr-win () s2))) rs1 (snd (fst (get-curr-win () s2))))::word64) = word-cat (cpu-reg-val Y (snd (fst (get-curr-win () s1)))) (user-reg-val (fst (fst (get-curr-win () s1))) rs1 (snd (fst (get-curr-win () s1))))
    using f5 f4 a1 by (metis (no-types) cpu-reg-val-def low-equal-def user-reg-val-low-equal)
  have f12: ucast (get-CWP (cpu-reg-val PSR s1)) = fst (fst (get-curr-win () s2))
    using f8 f5 by presburger
  then have rd = 0  $\longrightarrow$  (if rd = 0 then user-reg-val (fst (fst (get-curr-win () s2))) rd (snd (fst (get-curr-win () s2))) else div-comp-result instr (div-comp-temp-V instr (ucast (div-comp-temp-64bit instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win () s2)))) (user-reg-val (fst (fst (get-curr-win () s2))) rs1 (snd (fst (get-curr-win () s2)))))) operand2 >> 32)) (ucast (div-comp-temp-64bit instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win () s2)))) (user-reg-val (fst (fst (get-curr-win () s2))) rs1 (snd (fst (get-curr-win () s2)))))) operand2 >> 31)) (div-comp-temp-64bit instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win () s2)))) (user-reg-val (fst (fst (get-curr-win () s2))) rs1 (snd (fst (get-curr-win () s2)))))) operand2)) = user-reg-val (ucast (get-CWP (cpu-reg-val PSR s1))) 0 (snd (fst (get-curr-win () s1))))
    using f6 user-reg-val-low-equal by fastforce
  then have f13: rd = 0  $\longrightarrow$  write-reg (if rd = 0 then user-reg-val (fst (fst (get-curr-win () s2))) rd (snd (fst (get-curr-win () s2))) else div-comp-result instr (div-comp-temp-V instr (ucast (div-comp-temp-64bit instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win () s2)))) (user-reg-val (fst (fst (get-curr-win () s2))) rs1 (snd (fst (get-curr-win () s2)))))) operand2 >> 32)) (ucast (div-comp-temp-64bit instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win () s2)))) (user-reg-val (fst (fst (get-curr-win () s2))) rs1 (snd (fst (get-curr-win () s2)))))) operand2 >> 31)) (div-comp-temp-64bit instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win () s2)))) (user-reg-val (fst (fst (get-curr-win () s2))) rs1 (snd (fst (get-curr-win () s2)))))) operand2)) (ucast (get-CWP (cpu-reg-val PSR s1))) 0 (snd (fst (get-curr-win () s1)))) = write-reg (if rd = 0 then user-reg-val (fst (fst (get-curr-win () s1))) rd (snd (fst (get-curr-win () s1))) else div-comp-result instr (div-comp-temp-V instr (ucast (div-comp-temp-64bit instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win () s1)))) (user-reg-val (fst (fst (get-curr-win () s1))) rs1 (snd (fst (get-curr-win () s1)))))) operand2 >> 32)) (ucast (div-comp-temp-64bit instr (word-cat (cpu-reg-val

```



```

operand2 >> 32)) (ucast (div-comp-temp-64bit instr (word-cat (cpu-reg-val Y (snd
(fst (get-curr-win () s2)))) (user-reg-val (fst (fst (get-curr-win () s2))) rs1 (snd (fst
(get-curr-win () s2)))))) operand2 >> 31))) (snd (fst (write-reg (user-reg-val (fst
(fst (get-curr-win () s2))) 0 (snd (fst (get-curr-win () s2)))) (fst (fst (get-curr-win
() s2))) 0 (snd (fst (get-curr-win () s2))))))))))
  by fastforce }
moreover
{ assume write-reg (div-comp-result instr (div-comp-temp-V instr (ucast (div-comp-temp-64bit
instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win () s1)))) (user-reg-val (fst
(fst (get-curr-win () s1))) rs1 (snd (fst (get-curr-win () s1)))))) operand2 >> 32))
(ucast (div-comp-temp-64bit instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win
() s1)))) (user-reg-val (fst (fst (get-curr-win () s1))) rs1 (snd (fst (get-curr-win
() s1)))))) operand2 >> 31))) (div-comp-temp-64bit instr (word-cat (cpu-reg-val Y
(snd (fst (get-curr-win () s1)))) (user-reg-val (fst (fst (get-curr-win () s1))) rs1
(snd (fst (get-curr-win () s1)))))) operand2)) (ucast (get-CWP (cpu-reg-val PSR
s1))) rd (snd (fst (get-curr-win () s1))) ≠ write-reg (if rd = 0 then user-reg-val (fst
(fst (get-curr-win () s1))) rd (snd (fst (get-curr-win () s1))) else div-comp-result in-
str (div-comp-temp-V instr (ucast (div-comp-temp-64bit instr (word-cat (cpu-reg-val
Y (snd (fst (get-curr-win () s1)))) (user-reg-val (fst (fst (get-curr-win () s1))) rs1
(snd (fst (get-curr-win () s1)))))) operand2 >> 32)) (ucast (div-comp-temp-64bit
instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win () s1)))) (user-reg-val (fst
(fst (get-curr-win () s1))) rs1 (snd (fst (get-curr-win () s1)))))) operand2 >>
31))) (div-comp-temp-64bit instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win
() s1)))) (user-reg-val (fst (fst (get-curr-win () s1))) rs1 (snd (fst (get-curr-win
() s1)))))) operand2)) (fst (fst (get-curr-win () s2))) rd (snd (fst (get-curr-win ()
s1))))
  then have div-comp-result instr (div-comp-temp-V instr (ucast (div-comp-temp-64bit
instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win () s1)))) (user-reg-val (fst
(fst (get-curr-win () s2))) rs1 (snd (fst (get-curr-win () s1)))))) operand2 >> 32))
(ucast (div-comp-temp-64bit instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win
() s1)))) (user-reg-val (fst (fst (get-curr-win () s2))) rs1 (snd (fst (get-curr-win
() s1)))))) operand2 >> 31))) (div-comp-temp-64bit instr (word-cat (cpu-reg-val Y
(snd (fst (get-curr-win () s1)))) (user-reg-val (fst (fst (get-curr-win () s2))) rs1
(snd (fst (get-curr-win () s1)))))) operand2) ≠ (if rd = 0 then user-reg-val (fst (fst
(get-curr-win () s1))) rd (snd (fst (get-curr-win () s1))) else div-comp-result instr
(div-comp-temp-V instr (ucast (div-comp-temp-64bit instr (word-cat (cpu-reg-val Y
(snd (fst (get-curr-win () s1)))) (user-reg-val (fst (fst (get-curr-win () s1))) rs1
(snd (fst (get-curr-win () s1)))))) operand2 >> 32)) (ucast (div-comp-temp-64bit
instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win () s1)))) (user-reg-val (fst
(fst (get-curr-win () s1))) rs1 (snd (fst (get-curr-win () s1)))))) operand2 >>
31))) (div-comp-temp-64bit instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win
() s1)))) (user-reg-val (fst (fst (get-curr-win () s1))) rs1 (snd (fst (get-curr-win
() s1)))))) operand2))
  using f12 f9 by fastforce }
moreover
{ assume write-reg (div-comp-result instr (div-comp-temp-V instr (ucast (div-comp-temp-64bit
instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win () s1)))) (user-reg-val (fst
(fst (get-curr-win () s1))) rs1 (snd (fst (get-curr-win () s1)))))) operand2 >> 32))
(ucast (div-comp-temp-64bit instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win

```

```

() s1)))) (user-reg-val (fst (fst (get-curr-win () s1))) rs1 (snd (fst (get-curr-win
() s1)))))) operand2 >> 31))) (div-comp-temp-64bit instr (word-cat (cpu-reg-val Y
(snd (fst (get-curr-win () s1)))) (user-reg-val (fst (fst (get-curr-win () s1))) rs1
(snd (fst (get-curr-win () s1)))))) operand2)) (ucast (get-CWP (cpu-reg-val PSR
s1))) rd (snd (fst (get-curr-win () s2))) ≠ write-reg (if rd = 0 then user-reg-val (fst
(fst (get-curr-win () s2))) rd (snd (fst (get-curr-win () s2))) else div-comp-result instr
(div-comp-temp-V instr (ucast (div-comp-temp-64bit instr (word-cat (cpu-reg-val
Y (snd (fst (get-curr-win () s2)))) (user-reg-val (fst (fst (get-curr-win () s2))) rs1
(snd (fst (get-curr-win () s2)))))) operand2 >> 32)) (ucast (div-comp-temp-64bit
instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win () s2)))) (user-reg-val (fst
(fst (get-curr-win () s2))) rs1 (snd (fst (get-curr-win () s2)))))) operand2 >>
31))) (div-comp-temp-64bit instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win
() s2)))) (user-reg-val (fst (fst (get-curr-win () s2))) rs1 (snd (fst (get-curr-win
() s2)))))) operand2)) (fst (fst (get-curr-win () s2))) rd (snd (fst (get-curr-win ()
s2))))
  then have rd = 0
    using f14 by presburger }
moreover
{ assume rd = 0
  then have rd = 0 ∧ low-equal (snd (fst (write-reg (if rd = 0 then user-reg-val
(fst (fst (get-curr-win () s1))) rd (snd (fst (get-curr-win () s1))) else div-comp-result
instr (div-comp-temp-V instr (ucast (div-comp-temp-64bit instr (word-cat (cpu-reg-val
Y (snd (fst (get-curr-win () s1)))) (user-reg-val (fst (fst (get-curr-win () s1))) rs1
(snd (fst (get-curr-win () s1)))))) operand2 >> 32)) (ucast (div-comp-temp-64bit
instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win () s1)))) (user-reg-val (fst
(fst (get-curr-win () s1))) rs1 (snd (fst (get-curr-win () s1)))))) operand2 >>
31))) (div-comp-temp-64bit instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win
() s1)))) (user-reg-val (fst (fst (get-curr-win () s1))) rs1 (snd (fst (get-curr-win
() s1)))))) operand2)) (fst (fst (get-curr-win () s2))) rd (snd (fst (get-curr-win ()
s1)))))) (snd (fst (write-reg (if rd = 0 then user-reg-val (fst (fst (get-curr-win ()
s2))) rd (snd (fst (get-curr-win () s2))) else div-comp-result instr (div-comp-temp-V
instr (ucast (div-comp-temp-64bit instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win
() s2)))) (user-reg-val (fst (fst (get-curr-win () s2))) rs1 (snd (fst (get-curr-win ()
s2)))))) operand2 >> 32)) (ucast (div-comp-temp-64bit instr (word-cat (cpu-reg-val
Y (snd (fst (get-curr-win () s2)))) (user-reg-val (fst (fst (get-curr-win () s2))) rs1
(snd (fst (get-curr-win () s2)))))) operand2 >> 31))) (div-comp-temp-64bit instr
(word-cat (cpu-reg-val Y (snd (fst (get-curr-win () s2)))) (user-reg-val (fst (fst
(get-curr-win () s2))) rs1 (snd (fst (get-curr-win () s2)))))) operand2)) (fst (fst
(get-curr-win () s2))) rd (snd (fst (get-curr-win () s2))))))
    using f13 f12 f6 f2 by metis
  then have rd = 0 ∧ low-equal (snd (fst (div-write-new-val instr (div-comp-result
instr (div-comp-temp-V instr (ucast (div-comp-temp-64bit instr (word-cat (cpu-reg-val
Y (snd (fst (get-curr-win () s1)))) (user-reg-val (fst (fst (get-curr-win () s2))) rs1
(snd (fst (get-curr-win () s1)))))) operand2 >> 32)) (ucast (div-comp-temp-64bit
instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win () s1)))) (user-reg-val (fst
(fst (get-curr-win () s2))) rs1 (snd (fst (get-curr-win () s1)))))) operand2 >>
31))) (div-comp-temp-64bit instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win
() s1)))) (user-reg-val (fst (fst (get-curr-win () s2))) rs1 (snd (fst (get-curr-win
() s1)))))) operand2)) (div-comp-temp-V instr (ucast (div-comp-temp-64bit instr

```



```

(get-curr-win () s2)))) operand2 >> 31))) (snd (fst (write-reg (div-comp-result instr
(div-comp-temp-V instr (ucast (div-comp-temp-64bit instr (word-cat (cpu-reg-val
Y (snd (fst (get-curr-win () s2)))) (user-reg-val (fst (fst (get-curr-win () s2))) rs1
(snd (fst (get-curr-win () s2)))))) operand2 >> 32)) (ucast (div-comp-temp-64bit
instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win () s2)))) (user-reg-val (fst
(fst (get-curr-win () s2))) rs1 (snd (fst (get-curr-win () s2)))))) operand2 >>
31))) (div-comp-temp-64bit instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win
() s2)))) (user-reg-val (fst (fst (get-curr-win () s2))) rs1 (snd (fst (get-curr-win
() s2)))))) operand2)) (fst (fst (get-curr-win () s2))) rd (snd (fst (get-curr-win ()
s2))))))))) ^ (rd = 0 → low-equal (snd (fst (div-write-new-val instr (div-comp-result
instr (div-comp-temp-V instr (ucast (div-comp-temp-64bit instr (word-cat (cpu-reg-val
Y (snd (fst (get-curr-win () s1)))) (user-reg-val (fst (fst (get-curr-win () s2))) rs1
(snd (fst (get-curr-win () s1)))))) operand2 >> 32)) (ucast (div-comp-temp-64bit
instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win () s1)))) (user-reg-val (fst
(fst (get-curr-win () s2))) rs1 (snd (fst (get-curr-win () s1)))))) operand2 >>
31))) (div-comp-temp-64bit instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win
() s1)))) (user-reg-val (fst (fst (get-curr-win () s2))) rs1 (snd (fst (get-curr-win
() s1)))))) operand2)) (div-comp-temp-V instr (ucast (div-comp-temp-64bit instr
(word-cat (cpu-reg-val Y (snd (fst (get-curr-win () s1)))) (user-reg-val (fst (fst
(get-curr-win () s2))) rs1 (snd (fst (get-curr-win () s1)))))) operand2 >> 32))
(ucast (div-comp-temp-64bit instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win
() s1)))) (user-reg-val (fst (fst (get-curr-win () s2))) rs1 (snd (fst (get-curr-win
() s1)))))) operand2 >> 31))) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win
() s2))) 0 (snd (fst (get-curr-win () s1)))) (fst (fst (get-curr-win () s2))) 0 (snd (fst
(get-curr-win () s1))))))))) (snd (fst (div-write-new-val instr (div-comp-result instr
(div-comp-temp-V instr (ucast (div-comp-temp-64bit instr (word-cat (cpu-reg-val Y
(snd (fst (get-curr-win () s2)))) (user-reg-val (fst (fst (get-curr-win () s2))) rs1
(snd (fst (get-curr-win () s2)))))) operand2 >> 32)) (ucast (div-comp-temp-64bit
instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win () s2)))) (user-reg-val (fst
(fst (get-curr-win () s2))) rs1 (snd (fst (get-curr-win () s2)))))) operand2 >>
31))) (div-comp-temp-64bit instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win
() s2)))) (user-reg-val (fst (fst (get-curr-win () s2))) rs1 (snd (fst (get-curr-win
() s2)))))) operand2)) (div-comp-temp-V instr (ucast (div-comp-temp-64bit instr
(word-cat (cpu-reg-val Y (snd (fst (get-curr-win () s2)))) (user-reg-val (fst (fst
(get-curr-win () s2))) rs1 (snd (fst (get-curr-win () s2)))))) operand2 >> 32))
(ucast (div-comp-temp-64bit instr (word-cat (cpu-reg-val Y (snd (fst (get-curr-win
() s2)))) (user-reg-val (fst (fst (get-curr-win () s2))) rs1 (snd (fst (get-curr-win
() s2)))))) operand2 >> 31))) (snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win
() s2))) 0 (snd (fst (get-curr-win () s2)))) (fst (fst (get-curr-win () s2))) 0 (snd
(fst (get-curr-win () s2)))))))))

```

using f9 by fastforce

qed

lemma *div-instr-low-equal*:

assumes *a1*: low-equal *s1 s2* ^

t1 = snd (fst (div-instr instr *s1*)) ^ *t2* = snd (fst (div-instr instr *s2*))

shows low-equal *t1 t2*

using *a1*

apply (*simp add: div-instr-def*)

```

apply (simp add: Let-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
apply (simp add: case-prod-unfold)
apply (simp add: return-def)
apply (auto simp add: get-operand2-low-equal)
  apply (simp add: raise-trap-def add-trap-set-def simpler-modify-def)
  apply (auto simp add: traps-low-equal)
  apply (blast intro: mod-trap-low-equal)
using div-comp-low-equal by blast

lemma get-curr-win-traps-low-equal:
assumes a1: low-equal s1 s2
shows low-equal
  (snd (fst (get-curr-win () s1)))
    (traps := insert some-trap (traps (snd (fst (get-curr-win () s1)))))
  (snd (fst (get-curr-win () s2)))
    (traps := insert some-trap (traps (snd (fst (get-curr-win () s2)))))
proof –
  from a1 have f1: low-equal (snd (fst (get-curr-win () s1))) (snd (fst (get-curr-win
  () s2)))
    using get-curr-win2-low-equal by auto
  then have f2: (traps (snd (fst (get-curr-win () s1)))) =
    (traps (snd (fst (get-curr-win () s2))))
    using traps-low-equal by auto
  then show ?thesis using f1 f2 mod-trap-low-equal
    by fastforce
qed

lemma save-restore-instr-sub1-low-equal:
assumes a1: low-equal s1 s2 ∧
  t1 = snd (fst (save-restore-sub1 result new-cwp rd s1)) ∧
  t2 = snd (fst (save-restore-sub1 result new-cwp rd s2))
shows low-equal t1 t2
using a1
apply (simp add: save-restore-sub1-def)
apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
apply (simp add: case-prod-unfold)
apply (auto simp add: cpu-reg-val-low-equal)
using write-cpu-low-equal write-reg-low-equal
by fastforce

lemma get-WIM-bit-low-equal:
  ⟨get-WIM-bit (nat ((uint (fst (fst (get-curr-win () s1))) – 1) mod NWINDOWS))
    (cpu-reg-val WIM (snd (fst (get-curr-win () s1)))) =
  get-WIM-bit (nat ((uint (fst (fst (get-curr-win () s2))) – 1) mod NWINDOWS))
    (cpu-reg-val WIM (snd (fst (get-curr-win () s2))))⟩
  if ⟨low-equal s1 s2⟩
proof –
  from that have f1: low-equal (snd (fst (get-curr-win () s1))) (snd (fst (get-curr-win

```

```

() s2)))
  using get-curr-win2-low-equal by blast
  then have f2: (cpu-reg-val WIM (snd (fst (get-curr-win () s1)))) =
    (cpu-reg-val WIM (snd (fst (get-curr-win () s2))))
  using cpu-reg-val-low-equal by auto
  from that have (fst (fst (get-curr-win () s1))) = (fst (fst (get-curr-win () s2)))
  using get-curr-win-low-equal by auto
  then show ?thesis using f1 f2
    by auto
qed

```

lemma *get-WIM-bit-low-equal2*:

```

⟨get-WIM-bit (nat ((uint (fst (fst (get-curr-win () s1))) + 1) mod NWINDOWS))
  (cpu-reg-val WIM (snd (fst (get-curr-win () s1)))) =
  get-WIM-bit (nat ((uint (fst (fst (get-curr-win () s2))) + 1) mod NWINDOWS))
  (cpu-reg-val WIM (snd (fst (get-curr-win () s2))))⟩
  if ⟨low-equal s1 s2⟩

```

proof –

```

  from that have f1: low-equal (snd (fst (get-curr-win () s1))) (snd (fst (get-curr-win
() s2)))
  using get-curr-win2-low-equal by blast
  then have f2: (cpu-reg-val WIM (snd (fst (get-curr-win () s1)))) =
    (cpu-reg-val WIM (snd (fst (get-curr-win () s2))))
  using cpu-reg-val-low-equal by auto
  from that have (fst (fst (get-curr-win () s1))) = (fst (fst (get-curr-win () s2)))
  using get-curr-win-low-equal by auto
  then show ?thesis using f1 f2
    by auto
qed

```

lemma *take-bit-5-mod-NWINDOWS-eq [simp]*:

```

⟨take-bit 5 (k mod NWINDOWS) = k mod NWINDOWS⟩
  by (simp add: NWINDOWS-def take-bit-eq-mod)

```

lemma *save-restore-instr-low-equal*:

assumes *a1*: *low-equal s1 s2* \wedge

t1 = *snd (fst (save-restore-instr instr s1))* \wedge *t2* = *snd (fst (save-restore-instr instr s2))*

shows *low-equal t1 t2*

proof (*cases fst instr = ctrl-type SAVE*)

case *True*

then have *f1*: *fst instr = ctrl-type SAVE* **by** *auto*

then show ?thesis **using** *a1*

apply (*simp add: save-restore-instr-def*)

apply (*simp add: Let-def*)

apply (*simp add: simpler-gets-def bind-def h1-def h2-def Let-def*)

apply (*simp add: case-prod-unfold*)

apply (*auto simp add: unsigned-of-int*)

apply (*simp add: raise-trap-def add-trap-set-def simpler-modify-def*)

```

    apply (simp add: get-curr-win-traps-low-equal)
    apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
    apply (simp add: get-WIM-bit-low-equal)
    apply (simp add: get-WIM-bit-low-equal)
    apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
    apply (simp add: get-curr-win-low-equal)
    using get-curr-win2-low-equal save-restore-instr-sub1-low-equal get-addr2-low-equal
    apply metis
  done
next
case False
then show ?thesis using a1
  apply (simp add: save-restore-instr-def)
  apply (simp add: Let-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
  apply (simp add: case-prod-unfold)
  apply (auto simp add: unsigned-of-int)
    apply (simp add: raise-trap-def add-trap-set-def simpler-modify-def)
    apply (simp add: get-curr-win-traps-low-equal)
    apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
    apply (simp add: get-WIM-bit-low-equal2)
    apply (simp add: get-WIM-bit-low-equal2)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
  apply (simp add: get-curr-win-low-equal)
  using get-curr-win2-low-equal save-restore-instr-sub1-low-equal get-addr2-low-equal
  apply metis
  done
qed

```

lemma *call-instr-low-equal*:

assumes *a1*: *low-equal s1 s2* \wedge
t1 = *snd* (*fst* (*call-instr instr s1*)) \wedge *t2* = *snd* (*fst* (*call-instr instr s2*))

shows *low-equal t1 t2*

using *a1*

```

  apply (simp add: call-instr-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
  apply (simp add: case-prod-unfold)
  apply (auto simp add: get-curr-win-low-equal)
  using cpu-reg-val-low-equal get-curr-win2-low-equal
  write-cpu-low-equal write-reg-low-equal
proof –
  assume a1: low-equal s1 s2
  assume t1 = snd (fst (write-cpu (cpu-reg-val PC (snd (fst (get-curr-win ()
s1))) + (ucast (get-operand-w30 (snd instr ! 0) << 2)) nPC (snd (fst (write-cpu
(cpu-reg-val nPC (snd (fst (get-curr-win () s1)))) PC (snd (fst (write-reg (cpu-reg-val
PC (snd (fst (get-curr-win () s1)))) (fst (fst (get-curr-win () s2))) 15 (snd (fst
(get-curr-win () s1))))))))))
  assume t2 = snd (fst (write-cpu (cpu-reg-val PC (snd (fst (get-curr-win ()
s2))) + (ucast (get-operand-w30 (snd instr ! 0) << 2)) nPC (snd (fst (write-cpu

```

```

(cpu-reg-val nPC (snd (fst (get-curr-win () s2)))) PC (snd (fst (write-reg (cpu-reg-val
PC (snd (fst (get-curr-win () s2)))) (fst (fst (get-curr-win () s2))) 15 (snd (fst
(get-curr-win () s2))))))))))
  have  $\forall c.$  cpu-reg-val c (snd (fst (get-curr-win () s1))) = cpu-reg-val c (snd (fst
(get-curr-win () s2)))
    using a1 by (meson cpu-reg-val-low-equal get-curr-win2-low-equal)
  then show low-equal (snd (fst (write-cpu (cpu-reg-val PC (snd (fst (get-curr-win
() s1))) + (ucast (get-operand-w30 (snd instr ! 0)) << 2)) nPC (snd (fst (write-cpu
(cpu-reg-val nPC (snd (fst (get-curr-win () s1)))) PC (snd (fst (write-reg (cpu-reg-val
PC (snd (fst (get-curr-win () s1)))) (fst (fst (get-curr-win () s2))) 15 (snd (fst
(get-curr-win () s1)))))))))) (snd (fst (write-cpu (cpu-reg-val PC (snd (fst (get-curr-win
() s2))) + (ucast (get-operand-w30 (snd instr ! 0)) << 2)) nPC (snd (fst (write-cpu
(cpu-reg-val nPC (snd (fst (get-curr-win () s2)))) PC (snd (fst (write-reg (cpu-reg-val
PC (snd (fst (get-curr-win () s2)))) (fst (fst (get-curr-win () s2))) 15 (snd (fst
(get-curr-win () s2)))))))))))))
    using a1 by (metis (no-types) get-curr-win2-low-equal write-cpu-low-equal
write-reg-low-equal)
qed

```

lemma *jmp1-instr-low-equal-sub1*:

assumes *a1*: low-equal *s1 s2* \wedge

t1 = snd (fst (write-cpu (get-addr (snd instr) (snd (fst (get-curr-win () s2))))
nPC

```

(snd (fst (write-cpu (cpu-reg-val nPC
(snd (fst (write-reg (cpu-reg-val PC (snd (fst (get-curr-win () s1))))
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3))
(snd (fst (get-curr-win () s1))))))
PC (snd (fst (write-reg (cpu-reg-val PC (snd (fst (get-curr-win () s1))))
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3))
(snd (fst (get-curr-win () s1))))))))))  $\wedge$ 

```

t2 = snd (fst (write-cpu (get-addr (snd instr) (snd (fst (get-curr-win () s2))))
nPC

```

(snd (fst (write-cpu (cpu-reg-val nPC
(snd (fst (write-reg (cpu-reg-val PC (snd (fst (get-curr-win () s2))))
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3))
(snd (fst (get-curr-win () s2))))))
PC (snd (fst (write-reg (cpu-reg-val PC (snd (fst (get-curr-win () s2))))
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3))
(snd (fst (get-curr-win () s2))))))))))

```

shows low-equal *t1 t2*

proof –

from *a1* **have** *f1*: low-equal (snd (fst (get-curr-win () s1))) (snd (fst (get-curr-win
(*s2*)))

using *get-curr-win2-low-equal* **by** *blast*

then have *f2*: (cpu-reg-val PC (snd (fst (get-curr-win () s1)))) =
(cpu-reg-val PC (snd (fst (get-curr-win () s2))))

using *cpu-reg-val-low-equal* **by** *blast*

then have *f3*: low-equal

(snd (fst (write-reg (cpu-reg-val PC (snd (fst (get-curr-win () s1))))

```

(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3))
(snd (fst (get-curr-win () s1))))))
(snd (fst (write-reg (cpu-reg-val PC (snd (fst (get-curr-win () s2))))
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3))
(snd (fst (get-curr-win () s2))))))
using f1 write-reg-low-equal by fastforce
then have (cpu-reg-val nPC
(snd (fst (write-reg (cpu-reg-val PC (snd (fst (get-curr-win () s1))))
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3))
(snd (fst (get-curr-win () s1)))))) =
(cpu-reg-val nPC
(snd (fst (write-reg (cpu-reg-val PC (snd (fst (get-curr-win () s2))))
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3))
(snd (fst (get-curr-win () s2))))))
using cpu-reg-val-low-equal by auto
then have f4: low-equal
(snd (fst (write-cpu (cpu-reg-val nPC
(snd (fst (write-reg (cpu-reg-val PC (snd (fst (get-curr-win () s1))))
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3))
(snd (fst (get-curr-win () s1))))))
PC (snd (fst (write-reg (cpu-reg-val PC (snd (fst (get-curr-win () s1))))
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3))
(snd (fst (get-curr-win () s1))))))
(snd (fst (write-cpu (cpu-reg-val nPC
(snd (fst (write-reg (cpu-reg-val PC (snd (fst (get-curr-win () s2))))
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3))
(snd (fst (get-curr-win () s2))))))
PC (snd (fst (write-reg (cpu-reg-val PC (snd (fst (get-curr-win () s2))))
(fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! 3))
(snd (fst (get-curr-win () s2))))))
using f3 write-cpu-low-equal by fastforce
then show ?thesis using write-cpu-low-equal
using assms by blast
qed

lemma jmpl-instr-low-equal-sub2:
assumes a1: low-equal s1 s2  $\wedge$ 
t1 = snd (fst (write-cpu (get-addr (snd instr) (snd (fst (get-curr-win () s2))))
nPC
(snd (fst (write-cpu (cpu-reg-val nPC
(snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s1)))) (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s1)))))) PC (snd (fst (write-reg
(user-reg-val (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s1)))) (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s1)))))))))  $\wedge$ 
t2 = snd (fst (write-cpu (get-addr (snd instr) (snd (fst (get-curr-win () s2))))
nPC
(snd (fst (write-cpu (cpu-reg-val nPC (snd (fst (write-reg

```

```

(user-reg-val (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s2)))) (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s2)))))) PC (snd (fst (write-reg
(user-reg-val (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s2)))) (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s2))))))))))
shows low-equal t1 t2
proof –
from a1 have f1: low-equal (snd (fst (get-curr-win () s1))) (snd (fst (get-curr-win
() s2)))
using get-curr-win2-low-equal by blast
then have f2: (user-reg-val (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s1)))) =
(user-reg-val (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s2))))
using user-reg-val-low-equal by blast
then have f3: low-equal
(snd (fst (write-reg
(user-reg-val (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s1)))) (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s1))))))
(snd (fst (write-reg
(user-reg-val (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s2)))) (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s2))))))
using f1 write-reg-low-equal by fastforce
then have (cpu-reg-val nPC
(snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s1)))) (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s1)))))) =
(cpu-reg-val nPC (snd (fst (write-reg
(user-reg-val (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s2)))) (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s2))))))
using cpu-reg-val-low-equal by blast
then have low-equal
(snd (fst (write-cpu (cpu-reg-val nPC
(snd (fst (write-reg (user-reg-val (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s1)))) (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s1)))))) PC (snd (fst (write-reg
(user-reg-val (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s1)))) (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s1))))))))
(snd (fst (write-cpu (cpu-reg-val nPC (snd (fst (write-reg
(user-reg-val (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s2)))) (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s2)))))) PC (snd (fst (write-reg
(user-reg-val (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s2)))) (fst (fst (get-curr-win () s2))) 0
(snd (fst (get-curr-win () s2))))))

```

```

      (snd (fst (get-curr-win () s2)))))))))
    using f1 f2 f3 write-cpu-low-equal by fastforce
    then show ?thesis
    using write-cpu-low-equal
    using assms by blast
qed

```

```

lemma jmpl-instr-low-equal:
  assumes a1: low-equal s1 s2 ∧
    t1 = snd (fst (jmpl-instr instr s1)) ∧ t2 = snd (fst (jmpl-instr instr s2))
  shows low-equal t1 t2
  using a1
  apply (simp add: jmpl-instr-def)
  apply (simp add: Let-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
  apply (simp add: case-prod-unfold)
  apply auto
    apply (simp add: raise-trap-def add-trap-set-def simpler-modify-def)
    apply (simp add: get-curr-win-traps-low-equal)
    apply (simp add: get-addr2-low-equal)
    apply (simp add: get-addr2-low-equal)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
  apply (simp add: case-prod-unfold)
  apply (simp-all add: get-addr2-low-equal)
  apply (simp-all add: get-curr-win-low-equal)
  apply (case-tac get-operand-w5 (snd instr ! 3) ≠ 0)
  apply auto
    using jmpl-instr-low-equal-sub1 apply blast
  apply (simp-all add: get-curr-win-low-equal)
  using jmpl-instr-low-equal-sub2 by blast

```

```

lemma rett-instr-low-equal:
  assumes a1: low-equal s1 s2 ∧
    ¬ snd (rett-instr instr s1) ∧
    ¬ snd (rett-instr instr s2) ∧
    (((get-S (cpu-reg-val PSR s1)))::word1) = 0 ∧
    (((get-S (cpu-reg-val PSR s2)))::word1) = 0 ∧
    t1 = snd (fst (rett-instr instr s1)) ∧ t2 = snd (fst (rett-instr instr s2))
  shows low-equal t1 t2
  using a1
  apply (simp add: rett-instr-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
  apply auto
    apply (simp add: raise-trap-def add-trap-set-def simpler-modify-def)
    apply (simp add: return-def)
    using mod-trap-low-equal traps-low-equal apply fastforce
    using cpu-reg-val-low-equal apply fastforce
  using cpu-reg-val-low-equal apply fastforce
  apply (simp add: bind-def h1-def h2-def Let-def)

```

by (simp add: case-prod-unfold fail-def)

lemma read-state-reg-low-equal:

assumes a1: low-equal s1 s2 \wedge

$((\text{get-S } (\text{cpu-reg-val PSR } s1))::\text{word1}) = 0 \wedge$

$((\text{get-S } (\text{cpu-reg-val PSR } s2))::\text{word1}) = 0 \wedge$

$t1 = \text{snd } (\text{fst } (\text{read-state-reg-instr instr } s1)) \wedge$

$t2 = \text{snd } (\text{fst } (\text{read-state-reg-instr instr } s2))$

shows low-equal t1 t2

proof (cases (fst instr \in {sreg-type RDPSR,sreg-type RDWIM,sreg-type RDTBR})
 \vee

(fst instr = sreg-type RDASR \wedge privileged-ASR (get-operand-w5 ((snd instr)!0))))

case True

then have (fst instr \in {sreg-type RDPSR,sreg-type RDWIM,sreg-type RDTBR})

\vee

(fst instr = sreg-type RDASR \wedge privileged-ASR (get-operand-w5 ((snd instr)!0))))

$\wedge ((\text{get-S } (\text{cpu-reg-val PSR } (\text{snd } (\text{fst } (\text{get-curr-win } () s1))))::\text{word1}) = 0$

$\wedge ((\text{get-S } (\text{cpu-reg-val PSR } (\text{snd } (\text{fst } (\text{get-curr-win } () s2))))::\text{word1}) = 0$

by (metis assms get-curr-win-privilege)

then show ?thesis using a1

apply (simp add: read-state-reg-instr-def)

apply (simp add: Let-def)

apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)

apply (simp add: case-prod-unfold)

apply (simp add: raise-trap-def add-trap-set-def simpler-modify-def)

apply clarsimp

using get-curr-win-traps-low-equal

by auto

next

case False

then have f1: $\neg((\text{fst instr} = \text{sreg-type RDPSR} \vee$

$\text{fst instr} = \text{sreg-type RDWIM} \vee$

$\text{fst instr} = \text{sreg-type RDTBR} \vee$

$\text{fst instr} = \text{sreg-type RDASR} \wedge \text{privileged-ASR } (\text{get-operand-w5}$

$(\text{snd instr } ! 0))))$

by blast

then show ?thesis

proof (cases illegal-instruction-ASR (get-operand-w5 ((snd instr)!0)))

case True

then show ?thesis using a1 f1

apply read-state-reg-instr-privilege-proof

by (simp add: illegal-instruction-ASR-def)

next

case False

then have f2: $\neg(\text{illegal-instruction-ASR } (\text{get-operand-w5 } ((\text{snd instr})!0)))$

by auto

then show ?thesis

```

proof (cases (get-operand-w5 ((snd instr)!1)) ≠ 0)
  case True
  then have f3: (get-operand-w5 ((snd instr)!1)) ≠ 0
    by auto
  then show ?thesis
  proof (cases fst instr = sreg-type RDY)
    case True
    then show ?thesis using a1 f1 f2 f3
    apply (simp add: read-state-reg-instr-def)
    apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
    apply (simp add: case-prod-unfold)
    apply (auto simp add: get-curr-win-low-equal)
    using cpu-reg-val-low-equal get-curr-win2-low-equal write-reg-low-equal
    proof –
      assume low-equal s1 s2
      then have low-equal (snd (fst (get-curr-win () s1))) (snd (fst (get-curr-win
() s2)))
        by (meson get-curr-win2-low-equal)
        then show low-equal (snd (fst (write-reg (cpu-reg-val Y (snd (fst
(get-curr-win () s1)))) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd in-
str ! Suc 0)) (snd (fst (get-curr-win () s1)))))) (snd (fst (write-reg (cpu-reg-val
Y (snd (fst (get-curr-win () s2)))) (fst (fst (get-curr-win () s2))) (get-operand-w5
(snd instr ! Suc 0)) (snd (fst (get-curr-win () s2)))))))
          using cpu-reg-val-low-equal write-reg-low-equal by fastforce
    qed
  next
  case False
  then have f4: ¬(fst instr = sreg-type RDY) by auto
  then show ?thesis
  proof (cases fst instr = sreg-type RDASR)
    case True
    then show ?thesis using a1 f1 f2 f3 f4
    apply read-state-reg-instr-privilege-proof
    apply (clarsimp simp add: get-curr-win-low-equal)
    using cpu-reg-val-low-equal get-curr-win2-low-equal write-reg-low-equal
    proof –
      assume a1: low-equal s1 s2
      then have cpu-reg-val (ASR (get-operand-w5 (snd instr ! 0))) (snd (fst
(get-curr-win () s1))) = cpu-reg-val (ASR (get-operand-w5 (snd instr ! 0))) (snd
(fst (get-curr-win () s2)))
        by (meson cpu-reg-val-low-equal get-curr-win2-low-equal)
      then show low-equal (snd (fst (write-reg (cpu-reg-val (ASR (get-operand-w5
(snd instr ! 0))) (snd (fst (get-curr-win () s1)))) (fst (fst (get-curr-win () s2)))
(get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1)))))) (snd (fst
(write-reg (cpu-reg-val (ASR (get-operand-w5 (snd instr ! 0))) (snd (fst (get-curr-win
() s2)))) (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd
(fst (get-curr-win () s2)))))))
        using a1 by (metis (no-types) get-curr-win2-low-equal write-reg-low-equal)
    qed

```

```

next
  case False
  then have f5:  $\neg(\text{fst instr} = \text{sreg-type RDASR})$  by auto
  then show ?thesis using a1 f1 f2 f3 f4 f5
  apply read-state-reg-instr-privilege-proof
  apply (clarsimp simp add: get-curr-win-low-equal)
  using cpu-reg-val-low-equal get-curr-win2-low-equal write-reg-low-equal
  proof -
    assume a1: low-equal s1 s2
    assume a2:  $t1 = \text{snd} (\text{fst} (\text{write-reg} (\text{cpu-reg-val TBR} (\text{snd} (\text{fst} (\text{get-curr-win} () s1)))) (\text{fst} (\text{fst} (\text{get-curr-win} () s2)))) (\text{get-operand-w5} (\text{snd instr} ! \text{Suc } 0)) (\text{snd} (\text{fst} (\text{get-curr-win} () s1))))))$ 
    assume t2 =  $\text{snd} (\text{fst} (\text{write-reg} (\text{cpu-reg-val TBR} (\text{snd} (\text{fst} (\text{get-curr-win} () s2)))) (\text{fst} (\text{fst} (\text{get-curr-win} () s2)))) (\text{get-operand-w5} (\text{snd instr} ! \text{Suc } 0)) (\text{snd} (\text{fst} (\text{get-curr-win} () s2))))))$ 
    have  $\forall s. \neg \text{low-equal} (\text{snd} (\text{fst} (\text{get-curr-win} () s1))) s \vee \text{snd} (\text{fst} (\text{write-reg} (\text{cpu-reg-val TBR} s) (\text{fst} (\text{fst} (\text{get-curr-win} () s2)))) (\text{get-operand-w5} (\text{snd instr} ! \text{Suc } 0)) (\text{snd} (\text{fst} (\text{get-curr-win} () s1)))))) = t1$ 
    using a2 by (simp add: cpu-reg-val-low-equal)
    then show low-equal ( $\text{snd} (\text{fst} (\text{write-reg} (\text{cpu-reg-val TBR} (\text{snd} (\text{fst} (\text{get-curr-win} () s1)))) (\text{fst} (\text{fst} (\text{get-curr-win} () s2)))) (\text{get-operand-w5} (\text{snd instr} ! \text{Suc } 0)) (\text{snd} (\text{fst} (\text{get-curr-win} () s1)))))) (\text{snd} (\text{fst} (\text{write-reg} (\text{cpu-reg-val TBR} (\text{snd} (\text{fst} (\text{get-curr-win} () s2)))) (\text{fst} (\text{fst} (\text{get-curr-win} () s2)))) (\text{get-operand-w5} (\text{snd instr} ! \text{Suc } 0)) (\text{snd} (\text{fst} (\text{get-curr-win} () s2))))))$ )
    using a2 a1 by (metis (no-types) get-curr-win2-low-equal write-reg-low-equal)
  qed
qed
qed
next
  case False
  then show ?thesis using a1 f1 f2
  apply (simp add: read-state-reg-instr-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
  apply (simp add: return-def)
  apply clarsimp
  apply (simp add: case-prod-unfold)
  using get-curr-win2-low-equal by auto
  qed
qed
qed

lemma get-s-get-curr-win:
  assumes a1: low-equal s1 s2
  shows get-S (cpu-reg-val PSR ( $\text{snd} (\text{fst} (\text{get-curr-win} () s1)))) =$ 
get-S (cpu-reg-val PSR ( $\text{snd} (\text{fst} (\text{get-curr-win} () s2))))$ )
  proof -
    from a1 have low-equal ( $\text{snd} (\text{fst} (\text{get-curr-win} () s1)))$ 
      ( $\text{snd} (\text{fst} (\text{get-curr-win} () s2)))$ )
    using get-curr-win2-low-equal by blast

```

```

then show ?thesis
using cpu-reg-val-low-equal
by fastforce
qed

```

lemma write-state-reg-low-equal:

```

assumes a1: low-equal s1 s2  $\wedge$ 
  (((get-S (cpu-reg-val PSR s1))::word1) = 0  $\wedge$ 
   ((get-S (cpu-reg-val PSR s2))::word1) = 0  $\wedge$ 
   t1 = snd (fst (write-state-reg-instr instr s1))  $\wedge$ 
   t2 = snd (fst (write-state-reg-instr instr s2)))
shows low-equal t1 t2
proof (cases fst instr = sreg-type WRY)
  case True
    then show ?thesis using a1
    apply write-state-reg-instr-privilege-proof
    apply (simp add: simpler-modify-def)
    apply (simp add: delayed-pool-add-def DELAYNUM-def)
    apply (auto simp add: get-curr-win-low-equal)
    using get-curr-win2-low-equal cpu-reg-mod-low-equal
      user-reg-val-low-equal get-operand2-low-equal
    proof -
      assume a1: low-equal s1 s2
      assume t2 = cpu-reg-mod (user-reg-val (fst (fst (get-curr-win () s2)))) (get-operand-w5
        (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2))) XOR get-operand2 (snd instr)
        (snd (fst (get-curr-win () s2)))) Y (snd (fst (get-curr-win () s2)))
      assume t1 = cpu-reg-mod (user-reg-val (fst (fst (get-curr-win () s2)))) (get-operand-w5
        (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))) XOR get-operand2 (snd instr)
        (snd (fst (get-curr-win () s1)))) Y (snd (fst (get-curr-win () s1)))
      have f2: low-equal (snd (fst (get-curr-win () s1))) (snd (fst (get-curr-win ()
        s2)))
      using a1 by (meson get-curr-win2-low-equal)
      then have f3:  $\bigwedge w wa.$  user-reg-val w wa (snd (fst (get-curr-win () s2))) =
        user-reg-val w wa (snd (fst (get-curr-win () s1)))
      by (simp add: user-reg-val-low-equal)
      have  $\bigwedge is.$  get-operand2 is (snd (fst (get-curr-win () s2))) = get-operand2 is
        (snd (fst (get-curr-win () s1)))
      using f2 by (simp add: get-operand2-low-equal)
      then show low-equal (cpu-reg-mod (user-reg-val (fst (fst (get-curr-win ()
        s2)))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))) XOR
        get-operand2 (snd instr) (snd (fst (get-curr-win () s1)))) Y (snd (fst (get-curr-win
        () s1)))) (cpu-reg-mod (user-reg-val (fst (fst (get-curr-win () s2)))) (get-operand-w5
        (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2))) XOR get-operand2 (snd instr)
        (snd (fst (get-curr-win () s2)))) Y (snd (fst (get-curr-win () s2))))
      using f3 f2 by (metis cpu-reg-mod-low-equal)
    qed
  next
    case False
    then have f1:  $\neg$ (fst instr = sreg-type WRY) by auto

```

```

then show ?thesis
proof (cases fst instr = sreg-type WRASR)
  case True
  then have f1-1: fst instr = sreg-type WRASR by auto
  then show ?thesis
  proof (cases privileged-ASR (get-operand-w5 (snd instr ! 3)) ∧
    get-S (cpu-reg-val PSR (snd (fst (get-curr-win () s2)))) = 0)
    case True
    then show ?thesis using a1 f1 f1-1
    apply write-state-reg-instr-privilege-proof
    apply (clarsimp simp add: get-s-get-curr-win)
    apply (simp add: raise-trap-def add-trap-set-def simpler-modify-def)
    apply (clarsimp simp add: get-curr-win3-low-equal)
    using traps-low-equal mod-trap-low-equal get-curr-win2-low-equal
    by fastforce
  next
  case False
  then have f1-2: ¬ (privileged-ASR (get-operand-w5 (snd instr ! 3)) ∧
    get-S (cpu-reg-val PSR (snd (fst (get-curr-win () s2)))) = 0)
  by auto
  then show ?thesis
  proof (cases illegal-instruction-ASR (get-operand-w5 (snd instr ! 3)))
    case True
    then show ?thesis using a1 f1 f1-1 f1-2
    apply write-state-reg-instr-privilege-proof
    apply (clarsimp simp add: get-s-get-curr-win)
    apply auto
    apply (simp add: raise-trap-def add-trap-set-def simpler-modify-def)
    apply (clarsimp simp add: get-curr-win3-low-equal)
    using traps-low-equal mod-trap-low-equal get-curr-win2-low-equal
    apply fastforce
    apply (simp add: raise-trap-def add-trap-set-def simpler-modify-def)
    apply (clarsimp simp add: get-curr-win3-low-equal)
    using traps-low-equal mod-trap-low-equal get-curr-win2-low-equal
    by fastforce
  next
  case False
  then show ?thesis using a1 f1 f1-1 f1-2
  apply write-state-reg-instr-privilege-proof
  apply (clarsimp simp add: get-s-get-curr-win)
  apply auto
  apply (simp add: simpler-modify-def)
  apply (simp add: delayed-pool-add-def DELAYNUM-def)
  apply (auto simp add: get-curr-win-low-equal)
  using get-curr-win2-low-equal cpu-reg-mod-low-equal
  user-reg-val-low-equal get-operand2-low-equal
  proof -
    assume a1: low-equal s1 s2
    assume t2 = cpu-reg-mod (user-reg-val (fst (fst (get-curr-win ()

```

```

s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s2))) XOR
get-operand2 (snd instr) (snd (fst (get-curr-win () s2)))) (ASR (get-operand-w5
(snd instr ! 3))) (snd (fst (get-curr-win () s2)))
  assume t1 = cpu-reg-mod (user-reg-val (fst (fst (get-curr-win ()
s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))) XOR
get-operand2 (snd instr) (snd (fst (get-curr-win () s1)))) (ASR (get-operand-w5
(snd instr ! 3))) (snd (fst (get-curr-win () s1))))
  have low-equal (snd (fst (get-curr-win () s1))) (snd (fst (get-curr-win ()
s2)))
    using a1 by (meson get-curr-win2-low-equal)
    then show low-equal (cpu-reg-mod (user-reg-val (fst (fst (get-curr-win
() s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win () s1))) XOR
get-operand2 (snd instr) (snd (fst (get-curr-win () s1)))) (ASR (get-operand-w5
(snd instr ! 3))) (snd (fst (get-curr-win () s1)))) (cpu-reg-mod (user-reg-val (fst (fst
(get-curr-win () s2))) (get-operand-w5 (snd instr ! Suc 0)) (snd (fst (get-curr-win
() s2))) XOR get-operand2 (snd instr) (snd (fst (get-curr-win () s2)))) (ASR
(get-operand-w5 (snd instr ! 3))) (snd (fst (get-curr-win () s2))))))
      using cpu-reg-mod-low-equal get-operand2-low-equal user-reg-val-low-equal
by fastforce
  next
    assume f1: ¬ illegal-instruction-ASR (get-operand-w5 (snd instr ! 3))
    assume f2: fst instr = sreg-type WRASR
    assume f3: snd (fst (write-state-reg-instr instr s1)) =
snd (fst (modify
  (delayed-pool-add
    (DELAYNUM,
      user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! Suc 0))
        (snd (fst (get-curr-win () s1))) XOR
get-operand2 (snd instr) (snd (fst (get-curr-win () s1))),
ASR (get-operand-w5 (snd instr ! 3))))
        (snd (fst (get-curr-win () s1))))))
    assume f4: snd (fst (write-state-reg-instr instr s2)) =
snd (fst (modify
  (delayed-pool-add
    (DELAYNUM,
      user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! Suc 0))
        (snd (fst (get-curr-win () s2))) XOR
get-operand2 (snd instr) (snd (fst (get-curr-win () s2))),
ASR (get-operand-w5 (snd instr ! 3))))
        (snd (fst (get-curr-win () s2))))))
    assume f5: low-equal s1 s2
    assume f6: (get-S (cpu-reg-val PSR s1)) = 0
    assume f7: (get-S (cpu-reg-val PSR s2)) = 0
    assume f8: t1 = snd (fst (modify
  (delayed-pool-add
    (DELAYNUM,
      user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd

```

```

instr ! Suc 0))
  (snd (fst (get-curr-win () s1))) XOR
  get-operand2 (snd instr) (snd (fst (get-curr-win () s1))),
  ASR (get-operand-w5 (snd instr ! 3)))
(snd (fst (get-curr-win () s1))))
assume f9: t2 = snd (fst (modify
  (delayed-pool-add
  (DELAYNUM,
  user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! Suc 0))
  (snd (fst (get-curr-win () s2))) XOR
  get-operand2 (snd instr) (snd (fst (get-curr-win () s2))),
  ASR (get-operand-w5 (snd instr ! 3)))
(snd (fst (get-curr-win () s2))))))
assume f10: get-S (cpu-reg-val PSR (snd (fst (get-curr-win () s2)))) ≠ 0
assume f11: (∧ s1 s2 t1 t2.
  low-equal s1 s2 ⇒
  t1 = snd (fst (get-curr-win () s1)) ⇒ t2 = snd (fst (get-curr-win ()
s2)) ⇒ low-equal t1 t2)
assume f12: (∧ s1 s2 t1 w cr t2.
  low-equal s1 s2 ∧ t1 = cpu-reg-mod w cr s1 ∧ t2 = cpu-reg-mod w cr s2
⇒ low-equal t1 t2)
assume f13: (∧ s1 s2 win ur. low-equal s1 s2 ⇒ user-reg-val win ur s1
= user-reg-val win ur s2)
assume f14: (∧ s1 s2 op-list. low-equal s1 s2 ⇒ get-operand2 op-list s1
= get-operand2 op-list s2)
show low-equal
(snd (fst (modify
  (delayed-pool-add
  (DELAYNUM,
  user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! Suc 0))
  (snd (fst (get-curr-win () s1))) XOR
  get-operand2 (snd instr) (snd (fst (get-curr-win () s1))),
  ASR (get-operand-w5 (snd instr ! 3)))
(snd (fst (get-curr-win () s1))))))
(snd (fst (modify
  (delayed-pool-add
  (DELAYNUM,
  user-reg-val (fst (fst (get-curr-win () s2))) (get-operand-w5 (snd
instr ! Suc 0))
  (snd (fst (get-curr-win () s2))) XOR
  get-operand2 (snd instr) (snd (fst (get-curr-win () s2))),
  ASR (get-operand-w5 (snd instr ! 3)))
(snd (fst (get-curr-win () s2))))))
using f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 f12 f13 f14
using Sparc-Properties.ucast-0 assms get-curr-win-privilege by blast
qed
qed

```

```

qed
next
case False
then have f2:  $\neg(\text{fst instr} = \text{sreg-type WRASR})$  by auto
have f3:  $\text{get-S}(\text{cpu-reg-val PSR}(\text{snd}(\text{fst}(\text{get-curr-win}() s1)))) = 0 \wedge$ 
 $\text{get-S}(\text{cpu-reg-val PSR}(\text{snd}(\text{fst}(\text{get-curr-win}() s2)))) = 0$ 
using get-curr-win-privilege a1 by (metis ucast-id)
then show ?thesis
proof (cases  $\text{fst instr} = \text{sreg-type WRPSR}$ )
case True
then show ?thesis using a1 f1 f2 f3
apply write-state-reg-instr-privilege-proof
apply (simp add: raise-trap-def add-trap-set-def simpler-modify-def)
apply (clarsimp simp add: get-curr-win3-low-equal)
using traps-low-equal mod-trap-low-equal get-curr-win2-low-equal
by fastforce
next
case False
then have f4:  $\neg(\text{fst instr} = \text{sreg-type WRPSR})$  by auto
then show ?thesis
proof (cases  $\text{fst instr} = \text{sreg-type WRWIM}$ )
case True
then show ?thesis using a1 f1 f2 f3 f4
apply write-state-reg-instr-privilege-proof
apply (simp add: raise-trap-def add-trap-set-def simpler-modify-def)
apply (clarsimp simp add: get-curr-win3-low-equal)
using traps-low-equal mod-trap-low-equal get-curr-win2-low-equal
by fastforce
next
case False
then have f5:  $\neg(\text{fst instr} = \text{sreg-type WRWIM})$  by auto
then show ?thesis using a1 f1 f2 f3 f4 f5
apply write-state-reg-instr-privilege-proof
apply (simp add: raise-trap-def add-trap-set-def simpler-modify-def)
apply (clarsimp simp add: get-curr-win3-low-equal)
using traps-low-equal mod-trap-low-equal get-curr-win2-low-equal
by fastforce
qed
qed
qed
qed

lemma flush-instr-low-equal:
assumes a1: low-equal s1 s2  $\wedge$ 
t1 = snd (fst (flush-instr instr s1))  $\wedge$ 
t2 = snd (fst (flush-instr instr s2))
shows low-equal t1 t2
using a1
apply (simp add: flush-instr-def)

```

```

apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def simpler-modify-def)
apply (simp add: flush-cache-all-def)
apply (simp add: low-equal-def)
apply (simp add: user-accessible-def)
apply (simp add: mem-equal-def)
by auto

```

```

lemma branch-instr-sub1-low-equal:
assumes a1: low-equal s1 s2
shows branch-instr-sub1 instr-name s1 = branch-instr-sub1 instr-name s2
using a1 apply (simp add: branch-instr-sub1-def)
by (simp add: low-equal-def)

```

```

lemma set-annul-low-equal:
assumes a1: low-equal s1 s2  $\wedge$ 
t1 = snd (fst (set-annul True s1))  $\wedge$ 
t2 = snd (fst (set-annul True s2))
shows low-equal t1 t2
using a1 apply (simp add: set-annul-def)
apply (simp add: simpler-modify-def annul-mod-def)
using state-var2-low-equal state-var-low-equal
by fastforce

```

```

lemma branch-instr-low-equal-sub0:
assumes a1: low-equal s1 s2  $\wedge$ 
t1 = snd (fst (write-cpu (cpu-reg-val PC s2 +
  sign-ext24 (ucast (get-operand-w22 (snd instr ! Suc 0)) << 2))
  nPC (snd (fst (write-cpu (cpu-reg-val nPC s2) PC s1))))))  $\wedge$ 
t2 = snd (fst (write-cpu (cpu-reg-val PC s2 +
  sign-ext24 (ucast (get-operand-w22 (snd instr ! Suc 0)) << 2))
  nPC (snd (fst (write-cpu (cpu-reg-val nPC s2) PC s2))))))
shows low-equal t1 t2
proof –
  from a1 have low-equal
    (snd (fst (write-cpu (cpu-reg-val nPC s2) PC s1)))
    (snd (fst (write-cpu (cpu-reg-val nPC s2) PC s2)))
  using write-cpu-low-equal by blast
  then show ?thesis
  using a1 write-cpu-low-equal by blast
qed

```

```

lemma branch-instr-low-equal-sub1:
assumes a1: low-equal s1 s2  $\wedge$ 
t1 = snd (fst (set-annul True (snd (fst (write-cpu
  (cpu-reg-val PC s2 + sign-ext24
  (ucast (get-operand-w22 (snd instr ! Suc 0)) << 2))
  nPC (snd (fst (write-cpu (cpu-reg-val nPC s2) PC s1))))))))))  $\wedge$ 
t2 = snd (fst (set-annul True (snd (fst (write-cpu
  (cpu-reg-val PC s2 + sign-ext24

```

```

    (ucast (get-operand-w22 (snd instr ! Suc 0)) << 2))
    nPC (snd (fst (write-cpu (cpu-reg-val nPC s2) PC s2)))))))))
shows low-equal t1 t2
proof –
  from a1 have low-equal
    (snd (fst (write-cpu
      (cpu-reg-val PC s2 + sign-ext24
        (ucast (get-operand-w22 (snd instr ! Suc 0)) << 2))
        nPC (snd (fst (write-cpu (cpu-reg-val nPC s2) PC s1)))))))
    (snd (fst (write-cpu
      (cpu-reg-val PC s2 + sign-ext24
        (ucast (get-operand-w22 (snd instr ! Suc 0)) << 2))
        nPC (snd (fst (write-cpu (cpu-reg-val nPC s2) PC s2)))))))
  using branch-instr-low-equal-sub0 by blast
  then show ?thesis using a1
  using set-annul-low-equal by blast
qed

```

```

lemma branch-instr-low-equal-sub2:
assumes a1: low-equal s1 s2 ∧
t1 = snd (fst (set-annul True
  (snd (fst (write-cpu (cpu-reg-val nPC s2 + 4) nPC
    (snd (fst (write-cpu (cpu-reg-val nPC s2) PC s1))))))) ∧
t2 = snd (fst (set-annul True
  (snd (fst (write-cpu (cpu-reg-val nPC s2 + 4) nPC
    (snd (fst (write-cpu (cpu-reg-val nPC s2) PC s2)))))))
shows low-equal t1 t2
proof –
  from a1 have low-equal
    (snd (fst (write-cpu (cpu-reg-val nPC s2) PC s1)))
    (snd (fst (write-cpu (cpu-reg-val nPC s2) PC s2)))
  using write-cpu-low-equal by blast
  then have low-equal
    (snd (fst (write-cpu (cpu-reg-val nPC s2 + 4) nPC
      (snd (fst (write-cpu (cpu-reg-val nPC s2) PC s1)))))))
    (snd (fst (write-cpu (cpu-reg-val nPC s2 + 4) nPC
      (snd (fst (write-cpu (cpu-reg-val nPC s2) PC s2)))))))
  using write-cpu-low-equal by blast
  then show ?thesis using a1
  using set-annul-low-equal by blast
qed

```

```

lemma branch-instr-low-equal:
assumes a1: low-equal s1 s2 ∧
t1 = snd (fst (branch-instr instr s1)) ∧
t2 = snd (fst (branch-instr instr s2))
shows low-equal t1 t2
using a1
apply (simp add: branch-instr-def)

```

```

apply (simp add: Let-def simpler-gets-def bind-def h1-def h2-def)
apply (simp add: case-prod-unfold return-def)
apply clarsimp
apply (simp add: branch-instr-sub1-low-equal)
apply (simp-all add: cpu-reg-val-low-equal)
apply (cases branch-instr-sub1 (fst instr) s2 = 1)
  apply clarsimp
  apply (simp add: bind-def h1-def h2-def Let-def)
  apply (simp-all add: cpu-reg-val-low-equal)
  apply (simp add: case-prod-unfold)
  apply (cases fst instr = bicc-type BA  $\wedge$  get-operand-flag (snd instr ! 0) = 1)
    apply clarsimp
    using branch-instr-low-equal-sub1 apply blast
  apply clarsimp
  apply (simp add: return-def)
  using branch-instr-low-equal-sub0 apply fastforce
apply (simp add: bind-def h1-def h2-def Let-def)
apply (simp add: case-prod-unfold)
apply (cases get-operand-flag (snd instr ! 0) = 1)
  apply clarsimp
  apply (simp-all add: cpu-reg-val-low-equal)
  using branch-instr-low-equal-sub2 apply metis
apply (simp add: return-def)
using write-cpu-low-equal by metis

```

```

lemma dispath-instr-low-equal:
assumes a1: low-equal s1 s2  $\wedge$ 
  (((get-S (cpu-reg-val PSR s1)))::word1) = 0  $\wedge$ 
  (((get-S (cpu-reg-val PSR s2)))::word1) = 0  $\wedge$ 
   $\neg$  snd (dispatch-instruction instr s1)  $\wedge$ 
   $\neg$  snd (dispatch-instruction instr s2)  $\wedge$ 
  t1 = (snd (fst (dispatch-instruction instr s1)))  $\wedge$ 
  t2 = (snd (fst (dispatch-instruction instr s2)))
shows low-equal t1 t2
proof (cases get-trap-set s1 = {})
  case True
  then have f-no-traps: get-trap-set s1 = {}  $\wedge$  get-trap-set s2 = {}
    using a1 by (simp add: low-equal-def get-trap-set-def)
  then show ?thesis
  proof (cases fst instr  $\in$  {load-store-type LDSB,load-store-type LDUB,
    load-store-type LDUBA,load-store-type LDUH,load-store-type LD,
    load-store-type LDA,load-store-type LDD})
    case True
    then show ?thesis using a1 f-no-traps
    apply dispath-instr-privilege-proof
    by (blast intro: load-instr-low-equal)
  next
  case False
  then have f1: fst instr  $\notin$  {load-store-type LDSB, load-store-type LDUB,

```

```

    load-store-type LDUBA, load-store-type LDUH,
    load-store-type LD, load-store-type LDA, load-store-type LDD}
  by auto
then show ?thesis
proof (cases fst instr ∈ {load-store-type STB,load-store-type STH,
  load-store-type ST,load-store-type STA,load-store-type STD})
  case True
  then show ?thesis using a1 f-no-traps f1
  apply dispath-instr-privilege-proof
  using store-instr-low-equal by blast
next
  case False
  then have f2: ¬(fst instr ∈ {load-store-type STB,load-store-type STH,
    load-store-type ST,load-store-type STA,load-store-type STD})
    by auto
  then show ?thesis
  proof (cases fst instr ∈ {sethi-type SETHI})
    case True
    then show ?thesis using a1 f-no-traps f1 f2
    apply dispath-instr-privilege-proof
    by (auto intro: sethi-low-equal)
  next
    case False
    then have f3: ¬(fst instr ∈ {sethi-type SETHI})
      by auto
    then show ?thesis
    proof (cases fst instr ∈ {nop-type NOP})
      case True
      then show ?thesis using a1 f-no-traps f1 f2 f3
      apply dispath-instr-privilege-proof
      by (auto intro: nop-low-equal)
    next
      case False
      then have f4: ¬(fst instr ∈ {nop-type NOP})
        by auto
      then show ?thesis
      proof (cases fst instr ∈ {logic-type ANDs,logic-type ANDcc,logic-type
ANDN,
    logic-type ANDNcc,logic-type ORs,logic-type ORcc,logic-type ORN,
    logic-type XORs,logic-type XNOR})
        case True
        then show ?thesis using a1 f-no-traps f1 f2 f3 f4
        apply dispath-instr-privilege-proof
        using logical-instr-low-equal by blast
      next
        case False
        then have f5: ¬(fst instr ∈ {logic-type ANDs,logic-type ANDcc,logic-type
ANDN,
    logic-type ANDNcc,logic-type ORs,logic-type ORcc,logic-type ORN,

```

```

logic-type XORs,logic-type XNOR})
  by auto
then show ?thesis
proof (cases fst instr ∈ {shift-type SLL,shift-type SRL,shift-type SRA})
  case True
  then show ?thesis using a1 f-no-traps f1 f2 f3 f4 f5
  apply dispath-instr-privilege-proof
  using shift-instr-low-equal by blast
next
  case False
  then have f6: ¬(fst instr ∈ {shift-type SLL,shift-type SRL,shift-type
SRA})
    by auto
  then show ?thesis
  proof (cases fst instr ∈ {arith-type ADD,arith-type ADDcc,arith-type
ADDX})
    case True
    then show ?thesis using a1 f-no-traps f1 f2 f3 f4 f5 f6
    apply dispath-instr-privilege-proof
    using add-instr-low-equal by blast
  next
    case False
  then have f7: ¬(fst instr ∈ {arith-type ADD,arith-type ADDcc,arith-type
ADDX})
    by auto
  then show ?thesis
  proof (cases fst instr ∈ {arith-type SUB,arith-type SUBcc,arith-type
SUBX})
    case True
    then show ?thesis using a1 f-no-traps f1 f2 f3 f4 f5 f6 f7
    apply dispath-instr-privilege-proof
    using sub-instr-low-equal by blast
  next
    case False
  then have f8: ¬(fst instr ∈ {arith-type SUB,arith-type SUBcc,arith-type
SUBX})
    by auto
  then show ?thesis
  proof (cases fst instr ∈ {arith-type UMUL,arith-type SMUL,arith-type
SMULcc})
    case True
    then show ?thesis using a1 f-no-traps f1 f2 f3 f4 f5 f6 f7 f8
    apply dispath-instr-privilege-proof
    using mul-instr-low-equal by blast
  next
    case False
  then have f9: ¬(fst instr ∈ {arith-type UMUL,arith-type SMUL,
arith-type SMULcc})
    by auto

```

```

then show ?thesis
proof (cases fst instr ∈ {arith-type UDIV,arith-type UDIVcc,arith-type
SDIV})
  case True
  then show ?thesis using a1 f-no-traps f1 f2 f3 f4 f5 f6 f7 f8 f9
  apply dispath-instr-privilege-proof
  using div-instr-low-equal by blast
next
  case False
  then have f10: ¬(fst instr ∈ {arith-type UDIV,
arith-type UDIVcc,arith-type SDIV})
  by auto
  then show ?thesis
  proof (cases fst instr ∈ {ctrl-type SAVE,ctrl-type RESTORE})
    case True
    then show ?thesis using a1 f-no-traps f1 f2 f3 f4 f5 f6 f7 f8 f9
f10
    apply dispath-instr-privilege-proof
    using save-restore-instr-low-equal by blast
    next
    case False
    then have f11: ¬(fst instr ∈ {ctrl-type SAVE,ctrl-type
RESTORE})
    by auto
    then show ?thesis
    proof (cases fst instr ∈ {call-type CALL})
      case True
      then show ?thesis using a1 f-no-traps f1 f2 f3 f4 f5 f6 f7 f8
f9 f10 f11
      apply dispath-instr-privilege-proof
      using call-instr-low-equal by blast
      next
      case False
      then have f12: ¬(fst instr ∈ {call-type CALL}) by auto
      then show ?thesis
      proof (cases fst instr ∈ {ctrl-type JMPL})
        case True
        then show ?thesis using a1 f-no-traps f1 f2 f3 f4 f5 f6 f7
f8 f9 f10 f11 f12
        apply dispath-instr-privilege-proof
        using jmpl-instr-low-equal by blast
        next
        case False
        then have f13: ¬(fst instr ∈ {ctrl-type JMPL}) by auto
        then show ?thesis
        proof (cases fst instr ∈ {ctrl-type RETT})
          case True
          then show ?thesis using a1 f-no-traps f1 f2 f3 f4 f5 f6 f7
f8 f9 f10

```

```

    f11 f12 f13
  apply dispath-instr-privilege-proof
  using rett-instr-low-equal by blast
next
case False
then have f14:  $\neg(\text{fst instr} \in \{\text{ctrl-type RETT}\})$  by auto
then show ?thesis
proof (cases  $\text{fst instr} \in \{\text{sreg-type RDY}, \text{sreg-type RDPSR},$ 
   $\text{sreg-type RDWIM}, \text{sreg-type RDTBR}\})$ 
  case True
  then show ?thesis using a1 f-no-traps f1 f2 f3 f4 f5 f6
f7 f8 f9 f10

    f11 f12 f13 f14
  apply dispath-instr-privilege-proof
  using read-state-reg-low-equal by blast
next
case False
then have f15:  $\neg(\text{fst instr} \in \{\text{sreg-type RDY}, \text{sreg-type}$ 
RDPSR,
   $\text{sreg-type RDWIM}, \text{sreg-type RDTBR}\})$  by auto
then show ?thesis
  proof (cases  $\text{fst instr} \in \{\text{sreg-type WRY}, \text{sreg-type}$ 
WRPSR,
   $\text{sreg-type WRWIM}, \text{sreg-type WRTBR}\})$ 
  case True
  then show ?thesis using a1 f-no-traps f1 f2 f3 f4 f5 f6
f7 f8 f9

    f10 f11 f12 f13 f14 f15
  apply dispath-instr-privilege-proof
  using write-state-reg-low-equal by blast
next
case False
then have f16:  $\neg(\text{fst instr} \in \{\text{sreg-type WRY}, \text{sreg-type}$ 
WRPSR,
   $\text{sreg-type WRWIM}, \text{sreg-type WRTBR}\})$  by auto
then show ?thesis
proof (cases  $\text{fst instr} \in \{\text{load-store-type FLUSH}\})$ 
  case True
  then show ?thesis using a1 f-no-traps f1 f2 f3 f4 f5
f6 f7 f8 f9

    f10 f11 f12 f13 f14 f15 f16
  apply dispath-instr-privilege-proof
  using flush-instr-low-equal by blast
next
case False
then have f17:  $\neg(\text{fst instr} \in \{\text{load-store-type FLUSH}\})$ 
by auto

  then show ?thesis
  proof (cases  $\text{fst instr} \in \{\text{bicc-type BE}, \text{bicc-type BNE},$ 

```



```

¬ snd (execute-instr-sub1 instr s1) ∧
¬ snd (execute-instr-sub1 instr s2) ∧
t1 = (snd (fst (execute-instr-sub1 instr s1))) ∧
t2 = (snd (fst (execute-instr-sub1 instr s2)))
shows low-equal t1 t2
proof (cases get-trap-set s1 = {})
  case True
  then have get-trap-set s1 = {} ∧ get-trap-set s2 = {}
    using a1 by (simp add: low-equal-def get-trap-set-def)
  then show ?thesis using a1
  apply (simp add: execute-instr-sub1-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
  apply (case-tac fst instr ≠ call-type CALL ∧
    fst instr ≠ ctrl-type RETT ∧
    fst instr ≠ ctrl-type JMPL ∧
    fst instr ≠ bicc-type BE ∧
    fst instr ≠ bicc-type BNE ∧
    fst instr ≠ bicc-type BGU ∧
    fst instr ≠ bicc-type BLE ∧
    fst instr ≠ bicc-type BL ∧
    fst instr ≠ bicc-type BGE ∧
    fst instr ≠ bicc-type BNEG ∧
    fst instr ≠ bicc-type BG ∧
    fst instr ≠ bicc-type BCS ∧
    fst instr ≠ bicc-type BLEU ∧
    fst instr ≠ bicc-type BCC ∧
    fst instr ≠ bicc-type BA ∧ fst instr ≠ bicc-type BN)
  apply clarsimp
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
  apply (simp add: case-prod-unfold)
  apply (simp add: low-equal-def)
  apply (simp add: cpu-reg-val-def write-cpu-def cpu-reg-mod-def)
  apply (simp add: simpler-modify-def return-def)
  apply (simp add: user-accessible-mod-cpu-reg mem-equal-mod-cpu-reg)
  apply clarsimp
  by (auto simp add: return-def)
next
  case False
  then have get-trap-set s1 ≠ {} ∧ get-trap-set s2 ≠ {}
    using a1 by (simp add: low-equal-def get-trap-set-def)
  then show ?thesis using a1
  apply (simp add: execute-instr-sub1-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
  by (simp add: return-def)
qed

```

theorem non-interference-step:

assumes a1: (((get-S (cpu-reg-val PSR s1)))::word1) = 0 ∧
good-context s1 ∧

```

get-delayed-pool s1 = []  $\wedge$  get-trap-set s1 = {}  $\wedge$ 
(((get-S (cpu-reg-val PSR s2)))::word1) = 0  $\wedge$ 
get-delayed-pool s2 = []  $\wedge$  get-trap-set s2 = {}  $\wedge$ 
good-context s2  $\wedge$ 
low-equal s1 s2
shows  $\exists t1 t2. \text{Some } t1 = \text{NEXT } s1 \wedge \text{Some } t2 = \text{NEXT } s2 \wedge$ 
(((get-S (cpu-reg-val PSR t1)))::word1) = 0  $\wedge$ 
(((get-S (cpu-reg-val PSR t2)))::word1) = 0  $\wedge$ 
low-equal t1 t2
proof –
  from a1 have good-context s1  $\wedge$  good-context s2 by auto
  then have NEXT s1 = Some (snd (fst (execute-instruction () s1)))  $\wedge$ 
    NEXT s2 = Some (snd (fst (execute-instruction () s2)))
    by (simp add: single-step)
  then have  $\exists t1 t2. \text{Some } t1 = \text{NEXT } s1 \wedge \text{Some } t2 = \text{NEXT } s2$ 
    by auto
  then have f0: snd (execute-instruction() s1) = False  $\wedge$ 
    snd (execute-instruction() s2) = False
    by (auto simp add: NEXT-def case-prod-unfold)
  then have f1:  $\exists t1 t2. \text{Some } t1 = \text{NEXT } s1 \wedge$ 
    Some t2 = NEXT s2  $\wedge$ 
    (((get-S (cpu-reg-val PSR t1)))::word1) = 0  $\wedge$ 
    (((get-S (cpu-reg-val PSR t2)))::word1) = 0
    using a1
    apply (auto simp add: NEXT-def case-prod-unfold)
    by (auto simp add: safe-privilege)
  then show ?thesis
  proof (cases exe-mode-val s1)
    case True
    then have f-exe0: exe-mode-val s1 by auto
    then have f-exe: exe-mode-val s1  $\wedge$  exe-mode-val s2
    proof –
      have low-equal s1 s2 using a1 by auto
      then have state-var s1 = state-var s2 by (simp add: low-equal-def)
      then have exe-mode-val s1 = exe-mode-val s2 by (simp add: exe-mode-val-def)
      then show ?thesis using f-exe0 by auto
    qed
  then show ?thesis
  proof (cases  $\exists e. \text{fetch-instruction (delayed-pool-write s1) = Inl } e$ )
    case True
    then have f-fetch-error:  $\exists e. \text{fetch-instruction (delayed-pool-write s1) = Inl } e$ 
by auto
    then have f-fetch-error2: ( $\exists e. \text{fetch-instruction (delayed-pool-write s1) = Inl } e$ )  $\wedge$ 
      ( $\exists e. \text{fetch-instruction (delayed-pool-write s2) = Inl } e$ )
    proof –
      have cpu-reg s1 = cpu-reg s2
      using a1 by (simp add: low-equal-def)
      then have cpu-reg-val PC s1 = cpu-reg-val PC s2

```

```

    by (simp add: cpu-reg-val-def)
  then have cpu-reg-val PC s1 = cpu-reg-val PC s2 ∧
    (((get-S (cpu-reg-val PSR (delayed-pool-write s1))))::word1) = 0 ∧
    (((get-S (cpu-reg-val PSR (delayed-pool-write s2))))::word1) = 0
    using a1
    by (auto simp add: empty-delayed-pool-write-privilege)
  then show ?thesis using a1 f-fetch-error
  apply (simp add: fetch-instruction-def)
  apply (simp add: Let-def)
  apply clarsimp
  apply (case-tac wint (3 AND cpu-reg-val PC (delayed-pool-write s1)) = 0)
  apply auto
  apply (case-tac fst (memory-read 8 (cpu-reg-val PC (delayed-pool-write
s1)))
    (delayed-pool-write s1)) = None)
  apply auto
  apply (simp add: case-prod-unfold)
  using a1 apply (auto simp add: mem-read-delayed-write-low-equal)
  apply (simp add: case-prod-unfold)
  using a1 apply (auto simp add: mem-read-delayed-write-low-equal)
  apply (simp add: delayed-pool-write-def)
  by (simp add: Let-def get-delayed-write-def)
qed
then show ?thesis
proof (cases exe-mode-val s1)
  case True
  then have exe-mode-val s1 ∧ exe-mode-val s2 using exe-mode-low-equal a1
by auto
  then show ?thesis using f1
  apply (simp add: NEXT-def execute-instruction-def)
  apply (simp add: bind-def h1-def h2-def Let-def simplifier-def)
  using a1 apply clarsimp
  apply (simp add: simplifier-def bind-def h1-def h2-def Let-def)
  apply (simp add: simplifier-modify-def)
  using f-fetch-error2 apply clarsimp
  apply (simp add: raise-trap-def simplifier-modify-def return-def)
  apply (simp add: simplifier-def bind-def h1-def h2-def Let-def)
  apply (simp add: return-def simplifier-modify-def)
  apply (simp add: raise-trap-def simplifier-modify-def return-def)
  apply (simp add: simplifier-def bind-def h1-def h2-def Let-def)
  apply (simp add: return-def)
  apply (simp add: delayed-pool-write-def get-delayed-write-def Let-def)
  apply (simp add: low-equal-def)
  apply (simp add: add-trap-set-def)
  apply (simp add: cpu-reg-val-def)
  apply clarsimp
  by (simp add: mem-equal-mod-trap user-accessible-mod-trap)
next
  case False

```

```

then have  $\neg$  (exe-mode-val s1)  $\wedge$   $\neg$  (exe-mode-val s2)
  using exe-mode-low-equal a1 by auto
then show ?thesis using f1
apply (simp add: NEXT-def execute-instruction-def)
apply (simp add: bind-def h1-def h2-def Let-def simpler-gets-def)
using a1 apply clarsimp
apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
by (simp add: return-def)
qed
next
case False
then have f-fetch-suc: ( $\exists v$ . fetch-instruction (delayed-pool-write s1) = Inr v)

  using fetch-instr-result-1 by auto
then have ( $\exists v$ . fetch-instruction (delayed-pool-write s1) = Inr v  $\wedge$ 
  fetch-instruction (delayed-pool-write s2) = Inr v)

  proof –
    have cpu-reg s1 = cpu-reg s2
      using a1 by (simp add: low-equal-def)
    then have cpu-reg-val PC s1 = cpu-reg-val PC s2
      by (simp add: cpu-reg-val-def)
    then have cpu-reg-val PC s1 = cpu-reg-val PC s2  $\wedge$ 
      ((get-S (cpu-reg-val PSR (delayed-pool-write s1))))::word1) = 0  $\wedge$ 
      ((get-S (cpu-reg-val PSR (delayed-pool-write s2))))::word1) = 0
      using a1
      by (auto simp add: empty-delayed-pool-write-privilege)
    then show ?thesis using a1 f-fetch-suc
    apply (simp add: fetch-instruction-def)
    apply (simp add: Let-def)
    apply clarsimp
    apply (case-tac uint (3 AND cpu-reg-val PC (delayed-pool-write s1)) = 0)
    apply auto
    apply (case-tac fst (memory-read 8 (cpu-reg-val PC (delayed-pool-write
s1))
      (delayed-pool-write s1)) = None)
      apply auto
      apply (simp add: case-prod-unfold)
      using a1 apply (auto simp add: mem-read-delayed-write-low-equal)
      apply (simp add: case-prod-unfold)
      using a1 apply (auto simp add: mem-read-delayed-write-low-equal)
      apply (simp add: delayed-pool-write-def)
      by (simp add: Let-def get-delayed-write-def)
    qed
then have ( $\exists v$ . fetch-instruction (delayed-pool-write s1) = Inr v  $\wedge$ 
  fetch-instruction (delayed-pool-write s2) = Inr v  $\wedge$ 
   $\neg$  ( $\exists e$ . (decode-instruction v) = Inl e))

  using dispatch-fail f0 a1 f-exe by auto
then have f-fetch-dec: ( $\exists v$ . fetch-instruction (delayed-pool-write s1) = Inr v

```

\wedge

```

      fetch-instruction (delayed-pool-write s2) = Inr v ∧
      (∃ v1. (decode-instruction v) = Inr v1))
    using decode-instr-result-4 by auto
  then show ?thesis
  proof (cases annul-val (delayed-pool-write s1))
    case True
    then have annul-val (delayed-pool-write s1) ∧ annul-val (delayed-pool-write
s2)
      using a1
      apply (simp add: low-equal-def)
      by (simp add: delayed-pool-write-def get-delayed-write-def annul-val-def)
    then show ?thesis using a1 f1 f-exe f-fetch-dec
    apply (simp add: NEXT-def execute-instruction-def)
    apply (simp add: exec-gets return-def)
    apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
    apply (simp add: simpler-modify-def)
    apply clarsimp
    apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
    apply (simp add: case-prod-unfold)
    apply (simp add: write-cpu-def cpu-reg-val-def set-annul-def)
    apply (simp add: simpler-modify-def)
    apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
    apply (simp add: write-cpu-def cpu-reg-val-def set-annul-def)
    apply (simp add: simpler-modify-def)
    apply (simp add: cpu-reg-mod-def annul-mod-def)
    apply (simp add: delayed-pool-write-def get-delayed-write-def)
    apply (simp add: write-annul-def)
    apply clarsimp
    apply (simp add: low-equal-def)
    apply (simp add: user-accessible-annul mem-equal-annul)
    by (metis)
  next
  case False
  then have ¬ annul-val (delayed-pool-write s1) ∧
    ¬ annul-val (delayed-pool-write s2)
    using a1 apply (simp add: low-equal-def)
    apply (simp add: delayed-pool-write-def get-delayed-write-def)
    by (simp add: annul-val-def)
  then show ?thesis using a1 f1 f-exe f-fetch-dec
  apply (simp add: NEXT-def execute-instruction-def)
  apply (simp add: exec-gets return-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
  apply (simp add: simpler-modify-def)
  apply clarsimp
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
  apply (simp add: case-prod-unfold)
  apply (case-tac snd (execute-instr-sub1 (a, b)
    (snd (fst (dispatch-instruction (a, b)
      (delayed-pool-write s1)))))) ∨

```

```

      snd (dispatch-instruction (a, b) (delayed-pool-write s1)))
    apply auto
  apply (case-tac snd (execute-instr-sub1 (a, b)
    (snd (fst (dispatch-instruction (a, b)
      (delayed-pool-write s2)))))) ∨
    snd (dispatch-instruction (a, b) (delayed-pool-write s2)))
  apply auto
  apply (simp add: simpler-modify-def)
  apply (simp add: simpler-gets-def bind-def h1-def h2-def Let-def)
  apply (simp add: case-prod-unfold)
  apply (simp add: delayed-pool-write-def get-delayed-write-def)
by (meson dispath-instr-low-equal dispath-instr-privilege execute-instr-sub1-low-equal)

qed
qed
next
case False
then have f-non-exe: exe-mode-val s1 = False by auto
then have exe-mode-val s1 = False ∧ exe-mode-val s2 = False
proof -
  have low-equal s1 s2 using a1 by auto
  then have state-var s1 = state-var s2 by (simp add: low-equal-def)
  then have exe-mode-val s1 = exe-mode-val s2 by (simp add: exe-mode-val-def)
  then show ?thesis using f-non-exe by auto
qed
then show ?thesis using f1 a1
apply (simp add: NEXT-def execute-instruction-def)
by (simp add: simpler-gets-def bind-def h1-def h2-def Let-def return-def)
qed
qed

function (sequential) SEQ:: nat ⇒ ('a::len) sparc-state ⇒ ('a) sparc-state option
where SEQ 0 s = Some s
| SEQ n s = (
  case SEQ (n-1) s of None ⇒ None
  | Some t ⇒ NEXT t
)
by pat-completeness auto
termination by lexicographic-order

lemma SEQ-suc: SEQ n s = Some t ⇒ SEQ (Suc n) s = NEXT t
apply (induction n)
apply clarsimp
by (simp add: option.case-eq-if)

definition user-seq-exe:: nat ⇒ ('a::len) sparc-state ⇒ bool where
user-seq-exe n s ≡ ∀ i t. (i ≤ n ∧ SEQ i s = Some t) →
(good-context t ∧ get-delayed-pool t = [] ∧ get-trap-set t = {})

```

NIA is short for non-interference assumption.

definition *NIA* $t1\ t2 \equiv$
 $((\text{get-}S\ (\text{cpu-reg-val}\ \text{PSR}\ t1))::\text{word1}) = 0 \wedge$
 $((\text{get-}S\ (\text{cpu-reg-val}\ \text{PSR}\ t2))::\text{word1}) = 0 \wedge$
 $\text{good-context}\ t1 \wedge \text{get-delayed-pool}\ t1 = [] \wedge \text{get-trap-set}\ t1 = \{\}$ \wedge
 $\text{good-context}\ t2 \wedge \text{get-delayed-pool}\ t2 = [] \wedge \text{get-trap-set}\ t2 = \{\}$ \wedge
 $\text{low-equal}\ t1\ t2$

NIC is short for non-interference conclusion.

definition *NIC* $t1\ t2 \equiv (\exists u1\ u2. \text{Some}\ u1 = \text{NEXT}\ t1 \wedge \text{Some}\ u2 = \text{NEXT}\ t2$
 \wedge
 $((\text{get-}S\ (\text{cpu-reg-val}\ \text{PSR}\ u1))::\text{word1}) = 0 \wedge$
 $((\text{get-}S\ (\text{cpu-reg-val}\ \text{PSR}\ u2))::\text{word1}) = 0 \wedge$
 $\text{low-equal}\ u1\ u2)$

lemma *NIS-short*: $\forall t1\ t2. \text{NIA}\ t1\ t2 \longrightarrow \text{NIC}\ t1\ t2$

apply (*simp add: NIA-def NIC-def*)

using *non-interference-step by auto*

lemma *non-interference-induct-case-sub1*:

assumes $a1: (\exists t1. \text{Some}\ t1 = \text{SEQ}\ n\ s1 \wedge$
 $(\exists t2. \text{Some}\ t2 = \text{SEQ}\ n\ s2 \wedge$
 $\text{NIA}\ t1\ t2))$

shows $(\exists t1. \text{Some}\ t1 = \text{SEQ}\ n\ s1 \wedge$
 $(\exists t2. \text{Some}\ t2 = \text{SEQ}\ n\ s2 \wedge$
 $\text{NIA}\ t1\ t2 \wedge$
 $\text{NIC}\ t1\ t2))$

using *NIS-short*

using *assms by auto*

lemma *non-interference-induct-case*:

assumes $a1:$

$((\forall i\ t. i \leq n \wedge \text{SEQ}\ i\ s1 = \text{Some}\ t \longrightarrow$
 $\text{good-context}\ t \wedge \text{get-delayed-pool}\ t = [] \wedge \text{get-trap-set}\ t = \{\}) \wedge$
 $(\forall i\ t. i \leq n \wedge \text{SEQ}\ i\ s2 = \text{Some}\ t \longrightarrow$
 $\text{good-context}\ t \wedge \text{get-delayed-pool}\ t = [] \wedge \text{get-trap-set}\ t = \{\}) \longrightarrow$
 $(\exists t1. \text{Some}\ t1 = \text{SEQ}\ n\ s1 \wedge$
 $(\exists t2. \text{Some}\ t2 = \text{SEQ}\ n\ s2 \wedge$
 $((\text{get-}S\ (\text{cpu-reg-val}\ \text{PSR}\ t1))::\text{word1}) = 0 \wedge$
 $((\text{get-}S\ (\text{cpu-reg-val}\ \text{PSR}\ t2))::\text{word1}) = 0 \wedge \text{low-equal}\ t1\ t2))) \wedge$
 $(\forall i\ t. i \leq \text{Suc}\ n \wedge \text{SEQ}\ i\ s1 = \text{Some}\ t \longrightarrow$
 $\text{good-context}\ t \wedge \text{get-delayed-pool}\ t = [] \wedge \text{get-trap-set}\ t = \{\}) \wedge$
 $(\forall i\ t. i \leq \text{Suc}\ n \wedge \text{SEQ}\ i\ s2 = \text{Some}\ t \longrightarrow$
 $\text{good-context}\ t \wedge \text{get-delayed-pool}\ t = [] \wedge \text{get-trap-set}\ t = \{\})$

shows $\exists t1. \text{Some}\ t1 = (\text{case}\ \text{SEQ}\ n\ s1\ \text{of}\ \text{None} \Rightarrow \text{None} \mid \text{Some}\ x \Rightarrow \text{NEXT}\ x) \wedge$
 $(\exists t2. \text{Some}\ t2 = (\text{case}\ \text{SEQ}\ n\ s2\ \text{of}\ \text{None} \Rightarrow \text{None} \mid \text{Some}\ x \Rightarrow \text{NEXT}$
 $x) \wedge$

$((\text{get-}S\ (\text{cpu-reg-val}\ \text{PSR}\ t1))::\text{word1}) = 0 \wedge$

$((\text{get-}S\ (\text{cpu-reg-val}\ \text{PSR}\ t2))::\text{word1}) = 0 \wedge \text{low-equal}\ t1\ t2)$

proof –

from $a1$ **have** $f1$: $((\forall i t. i \leq n \wedge SEQ\ i\ s1 = Some\ t \longrightarrow$
 $good\ context\ t \wedge get\ delayed\ pool\ t = [] \wedge get\ trap\ set\ t = \{\}) \wedge$
 $(\forall i t. i \leq n \wedge SEQ\ i\ s2 = Some\ t \longrightarrow$
 $good\ context\ t \wedge get\ delayed\ pool\ t = [] \wedge get\ trap\ set\ t = \{\}))$
by (*metis le-SucI*)
then **have** $f2$: $(\exists t1. Some\ t1 = SEQ\ n\ s1 \wedge$
 $(\exists t2. Some\ t2 = SEQ\ n\ s2 \wedge$
 $((get\ S\ (cpu\ reg\ val\ PSR\ t1))::word1) = 0 \wedge$
 $((get\ S\ (cpu\ reg\ val\ PSR\ t2))::word1) = 0 \wedge$
 $low\ equal\ t1\ t2))$
using $a1$ **by** *auto*
then **have** $f3$: $(\exists t1. Some\ t1 = SEQ\ n\ s1 \wedge$
 $(\exists t2. Some\ t2 = SEQ\ n\ s2 \wedge$
 $NIA\ t1\ t2))$
using $f1$ *NIA-def* **by** (*metis (full-types) dual-order.refl*)
then **have** $(\exists t1. Some\ t1 = SEQ\ n\ s1 \wedge$
 $(\exists t2. Some\ t2 = SEQ\ n\ s2 \wedge$
 $NIA\ t1\ t2 \wedge$
 $NIC\ t1\ t2))$
using *non-interference-induct-case-sub1* **by** *blast*
then **have** $(\exists t1. Some\ t1 = SEQ\ n\ s1 \wedge$
 $(\exists t2. Some\ t2 = SEQ\ n\ s2 \wedge$
 $((get\ S\ (cpu\ reg\ val\ PSR\ t1))::word1) = 0 \wedge$
 $((get\ S\ (cpu\ reg\ val\ PSR\ t2))::word1) = 0 \wedge$
 $good\ context\ t1 \wedge get\ delayed\ pool\ t1 = [] \wedge get\ trap\ set\ t1 = \{\} \wedge$
 $good\ context\ t2 \wedge get\ delayed\ pool\ t2 = [] \wedge get\ trap\ set\ t2 = \{\} \wedge$
 $low\ equal\ t1\ t2) \wedge$
 $(\exists u1\ u2. Some\ u1 = NEXT\ t1 \wedge Some\ u2 = NEXT\ t2 \wedge$
 $((get\ S\ (cpu\ reg\ val\ PSR\ u1))::word1) = 0 \wedge$
 $((get\ S\ (cpu\ reg\ val\ PSR\ u2))::word1) = 0 \wedge$
 $low\ equal\ u1\ u2))$
using *NIA-def* *NIC-def* **by** *fastforce*
then **show** *?thesis*
by (*metis option.simps(5)*)
qed

lemma *non-interference-induct-case-sub2*:

assumes $a1$:

$(user\ seq\ exe\ n\ s1 \wedge$
 $user\ seq\ exe\ n\ s2 \longrightarrow$
 $(\exists t1. Some\ t1 = SEQ\ n\ s1 \wedge$
 $(\exists t2. Some\ t2 = SEQ\ n\ s2 \wedge$
 $((get\ S\ (cpu\ reg\ val\ PSR\ t1))::word1) = 0 \wedge$
 $((get\ S\ (cpu\ reg\ val\ PSR\ t2))::word1) = 0 \wedge low\ equal\ t1\ t2))) \wedge$
 $user\ seq\ exe\ (Suc\ n)\ s1 \wedge$
 $user\ seq\ exe\ (Suc\ n)\ s2$

shows $\exists t1. Some\ t1 = (case\ SEQ\ n\ s1\ of\ None \Rightarrow None \mid Some\ x \Rightarrow NEXT\ x) \wedge$
 $(\exists t2. Some\ t2 = (case\ SEQ\ n\ s2\ of\ None \Rightarrow None \mid Some\ x \Rightarrow NEXT$
 $x) \wedge$

$((\text{get-}S(\text{cpu-reg-val } PSR\ t1))::\text{word1}) = 0 \wedge$
 $((\text{get-}S(\text{cpu-reg-val } PSR\ t2))::\text{word1}) = 0 \wedge \text{low-equal } t1\ t2)$
using *a1*
by (*simp add: non-interference-induct-case user-seq-exe-def*)

theorem *non-interference:*

assumes *a1:*

$((\text{get-}S(\text{cpu-reg-val } PSR\ s1))::\text{word1}) = 0 \wedge$
good-context s1 \wedge
get-delayed-pool s1 = $\square \wedge$ *get-trap-set s1* = $\{\}$ \wedge
 $((\text{get-}S(\text{cpu-reg-val } PSR\ s2))::\text{word1}) = 0 \wedge$
get-delayed-pool s2 = $\square \wedge$ *get-trap-set s2* = $\{\}$ \wedge
good-context s2 \wedge
user-seq-exe n s1 \wedge *user-seq-exe n s2* \wedge
low-equal s1 s2

shows $(\exists t1\ t2. \text{Some } t1 = SEQ\ n\ s1 \wedge \text{Some } t2 = SEQ\ n\ s2 \wedge$

$((\text{get-}S(\text{cpu-reg-val } PSR\ t1))::\text{word1}) = 0 \wedge$
 $((\text{get-}S(\text{cpu-reg-val } PSR\ t2))::\text{word1}) = 0 \wedge$
 $\text{low-equal } t1\ t2)$

using *a1*

apply (*induction n*)

apply (*simp add: user-seq-exe-def*)

apply *clarsimp*

by (*simp add: non-interference-induct-case-sub2*)

end

end