

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	23
3.3	The Monad Laws	24
4	Adding Exceptions	24
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	28
7	Catching and Handling Exceptions	30

8	Hoare Logic	31
8.1	Validity	31
8.2	Determinism	33
8.3	Non-Failure	34
9	Basic exception reasoning	34
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	37
12	Register Operations	38
13	Getting Fields	38
14	Setting Fields	39
15	Clearing Fields	39
16	Memory Management Unit (MMU)	39
17	MMU Sizing	40
17.1	MMU Types	40
17.2	MMU length values	40
17.3	MMU index values	40
18	MMU Definition	40
19	Virtual Memory	41
19.1	MMU Auxiliary Definitions	41
19.2	Virtual Address Translation	42
20	SPARC V8 state model	46
21	state as a function	46
22	functions for state member access	47
23	SPARC instruction model	66
24	Single step theorem	132
25	Privilege safty	141
26	Single step non-interference property.	152

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 Micro-processor Functional Manual, Aeroflex, June 20, 2012, p35.

datatype *MMU-register*

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

lemma *MMU-register-induct*:

$P \text{ CR} \implies P \text{ CTP} \implies P \text{ CNR} \implies P \text{ FTSR} \implies P \text{ FAR}$
 $\implies P \ x$
{*proof*}

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

{*proof*}

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

definition *enum-MMU-register* = [*CR*, *CTP*, *CNR*, *FTSR*, *FAR*]

definition

enum-all-MMU-register $P \longleftrightarrow P \text{ CR} \wedge P \text{ CTP} \wedge P \text{ CNR} \wedge P \text{ FTSR} \wedge P \text{ FAR}$

definition

enum-ex-MMU-register $P \longleftrightarrow P \text{ CR} \vee P \text{ CTP} \vee P \text{ CNR} \vee P \text{ FTSR} \vee P \text{ FAR}$

instance {*proof*}

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

$ucast \ ((AND) \ psr \ 0b000000001000000000000000000000) \gg 23)$

definition $get-icc-Z :: word32 \Rightarrow word1$

where

$get-icc-Z \ psr \equiv$

$ucast \ ((AND) \ psr \ 0b000000000100000000000000000000) \gg 22)$

definition $get-icc-V :: word32 \Rightarrow word1$

where

$get-icc-V \ psr \equiv$

$ucast \ ((AND) \ psr \ 0b000000000100000000000000000000) \gg 21)$

definition $get-icc-C :: word32 \Rightarrow word1$

where

$get-icc-C \ psr \equiv$

$ucast \ ((AND) \ psr \ 0b000000000010000000000000000000) \gg 20)$

definition $update-S :: word1 \Rightarrow word32 \Rightarrow word32$

where

$update-S \ s-val \ psr-val \equiv$

$let \ tmp0 = (AND) \ psr-val \ 0b1111111111111111111111111111111101111111 \ in$
 $(OR) \ tmp0 \ (((ucast \ s-val)::word32) \ll 7)$

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

definition $update-CWP :: word5 \Rightarrow word32 \Rightarrow word32$

where

$update-CWP \ cwp-val \ psr-val \equiv$

$let \ tmp0 = (AND) \ psr-val \ (0b1111111111111111111111111111111100000::word32);$
 $s-val = ((ucast \ (get-S \ psr-val))::word1)$

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 $update-PSR-rett :: word5 \Rightarrow word1 \Rightarrow word1 \Rightarrow word32 \Rightarrow word32$

where

$update-PSR-rett \ cwp-val \ et-val \ s-val \ psr-val \equiv$

$let \ tmp0 = (AND) \ psr-val \ 0b1111111111111111111111111111111101000000;$

$tmp1 = (OR) \ tmp0 \ (((ucast \ cwp-val)::word32);$

$tmp2 = (OR) \ tmp1 \ (((ucast \ et-val)::word32) \ll 5);$

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*
{*proof*}

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* \Rightarrow *bool*
where *get-annul* *v* \equiv *annul* *v*

definition *get-reset-trap* :: *sparc-state-var* \Rightarrow *bool*
where *get-reset-trap* *v* \equiv *resett* *v*

definition *get-exe-mode* :: *sparc-state-var* \Rightarrow *bool*
where *get-exe-mode* *v* \equiv *exe* *v*

definition *get-reset-mode* :: *sparc-state-var* \Rightarrow *bool*
where *get-reset-mode* *v* \equiv *reset* *v*

definition *get-err-mode* :: *sparc-state-var* \Rightarrow *bool*
where *get-err-mode* *v* \equiv *err* *v*

definition *get-ticc-trap-type* :: *sparc-state-var* \Rightarrow *word7*
where *get-ticc-trap-type* *v* \equiv *ticc* *v*

definition *get-interrupt-level* :: *sparc-state-var* \Rightarrow *word3*
where *get-interrupt-level* *v* \equiv *itrpt-lvl* *v*

definition *get-store-barrier-pending* :: *sparc-state-var* \Rightarrow *bool*
where *get-store-barrier-pending* *v* \equiv *st-bar* *v*

definition *write-annul* :: *bool* \Rightarrow *sparc-state-var* \Rightarrow *sparc-state-var*
where *write-annul* *b* *v* \equiv *v*(*annul* := *b*)

definition *write-reset-trap* :: *bool* \Rightarrow *sparc-state-var* \Rightarrow *sparc-state-var*
where *write-reset-trap* *b* *v* \equiv *v*(*resett* := *b*)

definition *write-exe-mode* :: *bool* \Rightarrow *sparc-state-var* \Rightarrow *sparc-state-var*
where *write-exe-mode* *b* *v* \equiv *v*(*exe* := *b*)

definition *write-reset-mode* :: *bool* \Rightarrow *sparc-state-var* \Rightarrow *sparc-state-var*
where *write-reset-mode* *b* *v* \equiv *v*(*reset* := *b*)

definition *write-err-mode* :: *bool* \Rightarrow *sparc-state-var* \Rightarrow *sparc-state-var*
where *write-err-mode* *b* *v* \equiv *v*(*err* := *b*)

definition *write-ticc-trap-type* :: *word7* \Rightarrow *sparc-state-var* \Rightarrow *sparc-state-var*
where *write-ticc-trap-type* *w* *v* \equiv *v*(*ticc* := *w*)

definition *write-interrupt-level* :: *word3* \Rightarrow *sparc-state-var* \Rightarrow *sparc-state-var*

where

zero-ext16 w \equiv (*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* \Rightarrow *word32*

where

sign-ext16 w \equiv

let *highest-bit* = ((*AND*) *w* 0b1000000000000000) >> 15 in

if *highest-bit* = 0 then

(*ucast w*)::*word32*

else (*OR*) ((*ucast w*)::*word32*) 0b11111111111111110000000000000000

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

definition *sign-ext22::word22* \Rightarrow *word32*

where

sign-ext22 w \equiv

let *highest-bit* = ((*AND*) *w* 0b1000000000000000000000) >> 21 in

if *highest-bit* = 0 then

(*ucast w*)::*word32*

else (*OR*) ((*ucast w*)::*word32*) 0b11111111100000000000000000000000

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

definition *sign-ext24::word24* \Rightarrow *word32*

where

sign-ext24 w \equiv

let *highest-bit* = ((*AND*) *w* 0b100000000000000000000000) >> 23 in

if *highest-bit* = 0 then

(*ucast w*)::*word32*

else (*OR*) ((*ucast w*)::*word32*) 0b11111111000000000000000000000000

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
- | *MULSec* — 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*:

$b \neq [] \implies \text{hd} (\text{map } a \ b) = a (\text{hd } b)$
 $\langle \text{proof} \rangle$

lemma *tl-map-simp*:

$\text{tl} (\text{map } a \ b) = \text{map } a \ (\text{tl } b)$
 $\langle \text{proof} \rangle$

lemma *Collect-eq*:

$\{x. P x\} = \{x. Q x\} \longleftrightarrow (\forall x. P x = Q x)$
<proof>

lemma *iff-impI*: $\llbracket P \implies Q = R \rrbracket \implies (P \longrightarrow Q) = (P \longrightarrow R)$ *<proof>*

definition

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*]:
 $\llbracket f x = f' x' \rrbracket \implies (f \$ x) = (f' \$ x')$
<proof>

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

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

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

lemma *split-apply-cong*[*fundef-cong*]:
 $\llbracket f (fst p) (snd p) s = f' (fst p') (snd p') s' \rrbracket \implies \text{split } f p s = \text{split } f' p' s'$
<proof>

definition

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

definition

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

definition

pred-neg :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool)$ (*not* - [40] 40)
where
pred-neg *P* $\equiv \lambda x. \neg P x$

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

definition

zipWith :: ('a ⇒ 'b ⇒ 'c) ⇒ 'a list ⇒ 'b list ⇒ 'c list **where**
zipWith f xs ys ≡ map (split f) (zip xs ys)

primrec

delete :: 'a ⇒ 'a list ⇒ 'a list

where

delete y [] = []

| *delete* y (x#xs) = (if y=x then xs else x # *delete* y xs)

primrec

find :: ('a ⇒ bool) ⇒ 'a list ⇒ 'a option

where

find f [] = None

| *find* f (x # xs) = (if f x then Some x else *find* f xs)

definition

swp f ≡ λx y. f y x

primrec (*nonexhaustive*)

theRight :: 'a + 'b ⇒ 'b **where**

theRight (Inr x) = x

primrec (*nonexhaustive*)

theLeft :: 'a + 'b ⇒ 'a **where**

theLeft (Inl x) = x

definition

isLeft x ≡ (∃ y. x = Inl y)

definition

isRight x ≡ (∃ y. x = Inr y)

definition

const x ≡ λy. x

lemma *tranclD2*:

$(x, y) \in R^+ \implies \exists z. (x, z) \in R^* \wedge (z, y) \in R$
 ⟨proof⟩

lemma *linorder-min-same1* [simp]:

$(\min y x = y) = (y \leq (x::'a::linorder))$
 ⟨proof⟩

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

$(\min x y = y) = (y \leq (x::'a::linorder))$
 ⟨proof⟩

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

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

where

$wf\text{-sum divisor } r \ r' \equiv$
 $(\{(x, y). \neg \text{divisor } x \wedge \neg \text{divisor } y\} \cap r')$
 $\cup \{(x, y). \neg \text{divisor } x \wedge \text{divisor } y\}$
 $\cup (\{(x, y). \text{divisor } x \wedge \text{divisor } y\} \cap r)$

lemma *wf-sum-wf*:

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

abbreviation(*input*)

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

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

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

$((False \Longrightarrow P) \Longrightarrow PROP \ Q) \equiv PROP \ Q$
 $\langle proof \rangle$

lemma *split-paired-Ball*:

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

lemma *split-paired-Bex*:

$(\exists x \in A. P \ x) = (\exists x \ y. (x,y) \in A \wedge P \ (x,y))$
 $\langle proof \rangle$

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 $('s, 'a) \text{ det-monad} = 's \Rightarrow ('a \times 's) \times \text{bool}$

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

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

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 $h2 g fs = (\text{let } (a, b) = \text{fst } (fs) \text{ in } g a b)$

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

where

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

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

abbreviation(*input*)

$\text{bind-rev} :: ('c \Rightarrow ('a, 'b) \text{ det-monad}) \Rightarrow ('a, 'c) \text{ det-monad} \Rightarrow ('a, 'b) \text{ det-monad}$ (**infixl** $=\langle\langle 60$) **where**
 $g =\langle\langle f \equiv f \gg= g$

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 ≡ λs. ((s,s), *False*)

definition

put :: 's ⇒ ('s, *unit*) *det-monad* **where**
put s ≡ λ-. (((),s), *False*)

3.1 Failure

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

definition

fail :: 'a ⇒ ('s, 'a) *det-monad* **where**
fail a ≡ λs. ((a,s), *True*)

Assertions: fail if the property *P* is not true

definition

assert :: bool ⇒ ('a, *unit*) *det-monad* **where**
assert *P* ≡ if *P* then return () else fail ()

An assertion that also can introspect the current state.

definition

state-assert :: ('s ⇒ bool) ⇒ ('s, *unit*) *det-monad*
where
state-assert *P* ≡ *get* >>= (λs. *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 ⇒ 'a) ⇒ ('s, 'a) *det-monad* **where**
gets *f* ≡ *get* >>= (λs. *return* (*f* s))

Modify the current state using the function passed in.

definition

modify :: ('s ⇒ 's) ⇒ ('s, *unit*) *det-monad* **where**
modify *f* ≡ *get* >>= (λs. *put* (*f* s))

lemma *simpler-gets-def*: *gets* *f* = (λs. ((*f* s, s), *False*))
⟨*proof*⟩

lemma *simpler-modify-def*:

modify *f* = (λs. (((), *f* s), *False*))
⟨*proof*⟩

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

definition

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

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

definition

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

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

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 $(>>=)$, applying the return value directly:

lemma *return-bind [simp]*: $(return\ x\ >>= f) = f\ x$
 $\langle proof \rangle$

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

lemma *bind-return [simp]*: $(m\ >>= return) = m$
 $\langle proof \rangle$

$(>>=)$ 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 $(m\ >>= f)\ >>= g = m\ >>= (\lambda x. f\ x\ >>= g)$
 $\langle proof \rangle$

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 $(\text{'s}, \text{'e} + \text{'a})$ *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

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

definition

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

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

definition

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

where

$\text{lift } f \ v \equiv \text{case } v \text{ of } \text{Inl } e \Rightarrow \text{throwError } e$
 $\quad \quad \quad | \text{Inr } v' \Rightarrow f \ v'$

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

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

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

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

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

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

$(\text{'s}, \text{'e} + \text{unit}) \text{ det-monad}$

where

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

definition

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

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

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

4.1 Monad Laws for the Exception Monad

More direct definition of *liftE*:

lemma *liftE-def2*:

$\text{liftE } f = (\lambda s. ((\lambda(v,s'). (\text{Inr } v, s')) (\text{fst } (f s)), \text{snd } (f s)))$
 $\langle \text{proof} \rangle$

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

lemma *returnOk-bindE [simp]*: $(\text{returnOk } x \gg=E f) = f x$
 $\langle \text{proof} \rangle$

lemma *lift-return [simp]*:

$\text{lift } (\text{return} \circ \text{Inr}) = \text{return}$
 $\langle \text{proof} \rangle$

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

lemma *bindE-returnOk [simp]*: $(m \gg=E \text{returnOk}) = m$
 $\langle \text{proof} \rangle$

Associativity of ($\gg=E$):

lemma *bindE-assoc*:

$(m \gg=E f) \gg=E g = m \gg=E (\lambda x. f x \gg=E g)$
 $\langle \text{proof} \rangle$

returnOk could also be defined via *liftE*:

lemma *returnOk-liftE*:

$\text{returnOk } x = \text{liftE } (\text{return } x)$
 $\langle \text{proof} \rangle$

Execution after throwing an exception is skipped:

lemma *throwError-bindE* [*simp*]:
 $(\text{throwError } E \gg= E f) = \text{throwError } E$
 ⟨*proof*⟩

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 :: [*pstrn*, 'a'] => *dobind* ((- ←/ -) 10)
 :: *dobind* => *dobinds* (-)
-nobind :: 'a => *dobind* (-)
-dobinds :: [*dobind*, *dobinds*] => *dobinds* ((-);//(-))

-do :: [*dobinds*, 'a'] => 'a ((do ((-);//(-))//od) 100)

translations

-do (*-dobinds* *b* *bs*) *e* == *-do* *b* (*-do* *bs* *e*)
-do (*-nobind* *b*) *e* == *b* $\gg=$ (*CONST* *K-bind* *e*)
do *x* ← *a*; *e* *od* == *a* $\gg=$ ($\lambda x. e$)

Syntax examples:

lemma *do x ← return 1;*
 return (2::nat);
 return x
 od =
 return 1 $\gg=$
 ($\lambda x. \text{return } (2::\text{nat}) \gg=$
 K-bind (*return* *x*))
 ⟨*proof*⟩

lemma *do x ← return 1;*
 return 2;
 return x
 od = return 1
 ⟨*proof*⟩

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 :: [*dobinds*, 'a] => 'a ((*doE* ((-);/(-))/odE) 100)

translations

-doE (*-dobinds* b bs) e == *-doE* b (*-doE* bs e)
-doE (*-nobind* b) e == b >>=E (CONST K-bind e)
doE x ← a; e odE == a >>=E (λx. e)

Syntax examples:

lemma *doE* x ← *returnOk* 1;
returnOk (2::nat);
returnOk x
odE =
returnOk 1 >>=E
(λx. *returnOk* (2::nat) >>=E
K-bind (*returnOk* x))
⟨proof⟩

lemma *doE* x ← *returnOk* 1;
returnOk 2;
returnOk x
odE = *returnOk* 1
⟨proof⟩

6 Library of Monadic Functions and Combinators

Lifting a normal function into the monad type:

definition

liftM :: ('a ⇒ 'b) ⇒ ('s,'a) *det-monad* ⇒ ('s, 'b) *det-monad*

where

liftM f m ≡ *do* x ← m; *return* (f x) od

The same for the exception monad:

definition

liftME :: ('a ⇒ 'b) ⇒ ('s,'e+'a) *det-monad* ⇒ ('s,'e+'b) *det-monad*

where

liftME f m ≡ *doE* x ← m; *returnOk* (f x) odE

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

definition

sequence-x :: ('s, 'a) *det-monad list* ⇒ ('s, unit) *det-monad*

where

$sequence-x\ xs \equiv foldr\ (\lambda x\ y.\ x \gg= (\lambda-. y))\ xs\ (return\ ())$

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

definition

$mapM-x :: ('a \Rightarrow ('s, 'b)\ det-monad) \Rightarrow 'a\ list \Rightarrow ('s, unit)\ det-monad$

where

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

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

definition

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

where

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

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

definition

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

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

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

where

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

definition

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

where

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

definition

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

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

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

where

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

definition

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

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

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

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

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

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

Filtering a list using a monadic function as predicate:

primrec

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

$$\begin{aligned} & \text{filterM } P [] = \text{return } [] \\ | & \text{filterM } P (x \# xs) = \text{do} \\ & \quad b \leftarrow P x; \\ & \quad ys \leftarrow \text{filterM } P xs; \\ & \quad \text{return } (\text{if } b \text{ then } (x \# ys) \text{ else } ys) \\ & \text{od} \end{aligned}$$

7 Catching and Handling Exceptions

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

definition

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

$$\begin{aligned} f <\text{catch}> \text{ handler} \equiv \\ & \text{do } x \leftarrow f; \\ & \text{case } x \text{ of} \\ & \quad \text{Inr } b \Rightarrow \text{return } b \\ & \quad | \text{Inl } e \Rightarrow \text{handler } e \\ & \text{od} \end{aligned}$$

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

definition

$$\begin{aligned} \text{handleE}' &:: ('s, 'e1 + 'a) \text{det-monad} \Rightarrow \\ &('e1 \Rightarrow ('s, 'e2 + 'a) \text{det-monad}) \Rightarrow \\ &('s, 'e2 + 'a) \text{det-monad} \text{ (infix } \langle \text{handle2} \rangle 10) \end{aligned}$$

where

$$\begin{aligned} f \langle \text{handle2} \rangle \text{ handler} &\equiv \\ \text{do} & \\ &v \leftarrow f; \\ &\text{case } v \text{ of} \\ &\quad \text{Inl } e \Rightarrow \text{handler } e \\ &\quad | \text{Inr } v' \Rightarrow \text{return } (\text{Inr } v') \\ \text{od} & \end{aligned}$$

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

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

where

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

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

definition

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

where

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

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

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

$$(\{\!\{-\}\!/ - / \{\!\{-\}\}\!)$$

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

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

$$('b \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow$$

$$('a \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow \text{bool}$$

$$(\{\!\{-\}\!/ - / (\{\!\{-\}\!/ \{\!\{-\}\}\!))$$

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

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

$$('a \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow \text{bool}$$

$$(\{\!\{-\}\!/ - / \{\!\{-\}\!/ -, -)$$

where

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

definition

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

$$('e \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow \text{bool}$$

$$(\{\!\{-\}\!/ - / -, \{\!\{-\}\}\!)$$

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*)
bottom :: 'a ⇒ bool (⊥)
where
⊥ ≡ λ-. *False*

Abbreviations for trivial postconditions (taking two arguments):

abbreviation(*input*)
toptop :: 'a ⇒ 'b ⇒ bool (⊤⊤)
where
⊤⊤ ≡ λ- -. *True*

abbreviation(*input*)
botbot :: 'a ⇒ 'b ⇒ bool (⊥⊥)
where
⊥⊥ ≡ λ- -. *False*

Lifting ∧ and ∨ over two arguments. Lifting ∧ and ∨ over one argument is already defined (written *and* and *or*).

definition
bipred-conj :: ('a ⇒ 'b ⇒ bool) ⇒ ('a ⇒ 'b ⇒ bool) ⇒ ('a ⇒ 'b ⇒ bool)
(infixl *And 96*)
where
bipred-conj P Q ≡ λx y. P x y ∧ Q x y

definition
bipred-disj :: ('a ⇒ 'b ⇒ bool) ⇒ ('a ⇒ 'b ⇒ bool) ⇒ ('a ⇒ 'b ⇒ bool)
(infixl *Or 91*)
where
bipred-disj P Q ≡ λx y. P x y ∨ 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* ⇒ bool
where
det f ≡ ∀ s. ∃ 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* ⇒ 's ⇒ 'a × 's
where
the-run-state M ≡ λs. *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

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

where

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

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

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

$$(\{\!\{-\}\!\} / - / \{\!\{-\}\!\}!)$$

where

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

definition

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

$$('b \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow$$

$$('a \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow \text{bool}$$

$$(\{\!\{-\}\!\} / - / (\{\!\{-\}\!\} / \{\!\{-\}\!\}!))$$

where

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

lemma *validE-NF-alt-def*:

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

<proof>

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* $P \ A \equiv \{\!\{ P \}\!\} A \ \{\!\{ \lambda - \ . \ \text{True} \}\!\}, \{\!\{ \lambda - \ . \ \text{False} \}\!\}$

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

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*]:

$\llbracket f = f'; \bigwedge v s s'. (v, s') = fst (f' s) \implies g v s' = g' v s' \rrbracket \implies f \gg = g = f' \gg = g'$
 ⟨*proof*⟩

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

$\llbracket f s = f' s'; \bigwedge rv st. (rv, st) = fst (f' s') \implies g rv st = g' rv st \rrbracket$
 $\implies (f \gg = g) s = (f' \gg = g') s'$
 ⟨*proof*⟩

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

$\llbracket M = M'; \bigwedge v s s'. (Inr v, s') = fst (M' s) \implies N v s' = N' v s' \rrbracket \implies bindE M N = bindE M' N'$
 ⟨*proof*⟩

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

$\llbracket f s = f' s'; \bigwedge rv st. (Inr rv, st) = fst (f' s') \implies g rv st = g' rv st \rrbracket$
 $\implies (f \gg =E g) s = (f' \gg =E g') s'$
 ⟨*proof*⟩

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

$\llbracket f st = f' st' \rrbracket \implies K\text{-bind } f \text{ arg } st = K\text{-bind } f' \text{ arg}' st'$
 ⟨*proof*⟩

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

$\llbracket C = C'; s = s'; C' \implies m s' = m' s' \rrbracket \implies whenE C m s = whenE C' m' s'$
 ⟨*proof*⟩

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

$\llbracket C = C'; s = s'; \neg C' \implies m s' = m' s' \rrbracket \implies unlessE C m s = unlessE C' m' s'$
 ⟨*proof*⟩

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

$\llbracket C = C'; s = s'; C' \implies m s' = m' s' \rrbracket \implies whenE C m s = whenE C' m' s'$
 ⟨*proof*⟩

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

$\llbracket C = C'; s = s'; \neg C' \implies m s' = m' s' \rrbracket \implies unlessE C m s = unlessE C' m' s'$
 ⟨*proof*⟩

10.2 Simplifying Monads

lemma *nested-bind* [*simp*]:
do $x \leftarrow do\ y \leftarrow f; return\ (g\ y)\ od; h\ x\ od =$
do $y \leftarrow f; h\ (g\ y)\ od$
(*proof*)

lemma *assert-True* [*simp*]:
assert $True\ \>\>= f = f\ ()$
(*proof*)

lemma *when-True-bind* [*simp*]:
when1 $True\ g\ \>\>= f = g\ \>\>= f$
(*proof*)

lemma *whenE-False-bind* [*simp*]:
whenE $False\ g\ \>\>=E f = f\ ()$
(*proof*)

lemma *whenE-True-bind* [*simp*]:
whenE $True\ g\ \>\>=E f = g\ \>\>=E f$
(*proof*)

lemma *when-True* [*simp*]: $when1\ True\ X = X$
(*proof*)

lemma *when-False* [*simp*]: $when1\ False\ X = return\ ()$
(*proof*)

lemma *unless-False* [*simp*]: $unless\ False\ X = X$
(*proof*)

lemma *unless-True* [*simp*]: $unless\ True\ X = return\ ()$
(*proof*)

lemma *unlessE-whenE*:
 $unlessE\ P = whenE\ (\sim P)$
(*proof*)

lemma *unless-when*:
 $unless\ P = when1\ (\sim P)$
(*proof*)

lemma *gets-to-return* [*simp*]: $gets\ (\lambda s.\ v) = return\ v$
(*proof*)

lemma *liftE-handleE'* [*simp*]: $((liftE\ a)\ <handle2>\ b) = liftE\ a$
(*proof*)

lemma *liftE-handleE* [*simp*]: $((liftE\ a)\ <handle>\ b) = liftE\ a$

<proof>

lemma *condition-split*:

$P \text{ (condition } C \ a \ b \ s) = (((C \ s) \longrightarrow P \ (a \ s)) \wedge (\neg (C \ s) \longrightarrow P \ (b \ s)))$
<proof>

lemma *condition-split-asm*:

$P \text{ (condition } C \ a \ b \ s) = (\neg (C \ s \wedge \neg P \ (a \ s)) \vee \neg C \ s \wedge \neg P \ (b \ s))$
<proof>

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

lemma *condition-true-triv [simp]*:

$\text{condition } (\lambda_. \text{ True}) \ A \ B = A$
<proof>

lemma *condition-false-triv [simp]*:

$\text{condition } (\lambda_. \text{ False}) \ A \ B = B$
<proof>

lemma *condition-true*: $\llbracket P \ s \rrbracket \Longrightarrow \text{condition } P \ A \ B \ s = A \ s$

<proof>

lemma *condition-false*: $\llbracket \neg P \ s \rrbracket \Longrightarrow \text{condition } P \ A \ B \ s = B \ s$

<proof>

11 Low-level monadic reasoning

lemma *valid-make-schematic-post*:

$(\forall s0. \llbracket \lambda s. P \ s0 \ s \rrbracket f \llbracket \lambda rv \ s. Q \ s0 \ rv \ s \rrbracket) \Longrightarrow$
 $\llbracket \lambda s. \exists s0. P \ s0 \ s \wedge (\forall rv \ s'. Q \ s0 \ rv \ s' \longrightarrow Q' \ rv \ s') \rrbracket f \llbracket Q' \rrbracket$
<proof>

lemma *validNF-make-schematic-post*:

$(\forall s0. \llbracket \lambda s. P \ s0 \ s \rrbracket f \llbracket \lambda rv \ s. Q \ s0 \ rv \ s \rrbracket!) \Longrightarrow$
 $\llbracket \lambda s. \exists s0. P \ s0 \ s \wedge (\forall rv \ s'. Q \ s0 \ rv \ s' \longrightarrow Q' \ rv \ s') \rrbracket f \llbracket Q' \rrbracket!$
<proof>

lemma *validE-make-schematic-post*:

$(\forall s0. \llbracket \lambda s. P \ s0 \ s \rrbracket f \llbracket \lambda rv \ s. Q \ s0 \ rv \ s \rrbracket, \llbracket \lambda rv \ s. E \ s0 \ rv \ s \rrbracket) \Longrightarrow$
 $\llbracket \lambda s. \exists s0. P \ s0 \ s \wedge (\forall rv \ s'. Q \ s0 \ rv \ s' \longrightarrow Q' \ rv \ s') \wedge$
 $(\forall rv \ s'. E \ s0 \ rv \ s' \longrightarrow E' \ rv \ s') \rrbracket f \llbracket Q' \rrbracket, \llbracket E' \rrbracket$
<proof>

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

$(\forall s0. \llbracket \lambda s. P \ s0 \ s \rrbracket f \llbracket \lambda rv \ s. Q \ s0 \ rv \ s \rrbracket, \llbracket \lambda rv \ s. E \ s0 \ rv \ s \rrbracket!) \Longrightarrow$
 $\llbracket \lambda s. \exists s0. P \ s0 \ s \wedge (\forall rv \ s'. Q \ s0 \ rv \ s' \longrightarrow Q' \ rv \ s') \wedge$
 $(\forall rv \ s'. E \ s0 \ rv \ s' \longrightarrow E' \ rv \ s') \rrbracket f \llbracket Q' \rrbracket, \llbracket E' \rrbracket!$
<proof>

lemma *validNF-conjD1*: $\{\!| P \!\} f \{\!| \lambda r v s. Q r v s \wedge Q' r v s \!\}! \implies \{\!| P \!\} f \{\!| Q \!\}!$
<proof>

lemma *validNF-conjD2*: $\{\!| P \!\} f \{\!| \lambda r v s. Q r v s \wedge Q' r v s \!\}! \implies \{\!| P \!\} f \{\!| Q' \!\}!$
<proof>

lemma *exec-gets*:
 $(gets\ f\ \gg = m)\ s = m\ (f\ s)\ s$
<proof>

lemma *in-gets*:
 $(r, s') = fst\ (gets\ f\ s) = (r = f\ s \wedge s' = s)$
<proof>

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::len word* \Rightarrow *nat* \Rightarrow *'b::len word*

where
get-field-from-word-a-b *addr index*
 $\equiv let\ off = (size\ addr - LENGTH('b))$
 $in\ ucast\ ((addr\ \ll\ (off - index))\ \gg\ off)$

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*
 $\equiv (addr\ \ll\ (size\ addr - (index + len))\ \gg\ (size\ addr - len))$

14 Setting Fields

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

definition *set-field* :: '*a*::len word ⇒ '*b*::len word ⇒ nat ⇒ '*a*::len word

where

set-field record field index

≡ 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 ⇒ nat ⇒ '*a*::len word

where

clear-n-bits addr n ≡ addr AND (NOT (mask n))

Gets the natural value of a 32 bit mask

definition *get-nat-from-mask*::word32 ⇒ nat ⇒ nat ⇒ (word32 × nat)

where

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

definition *get-nat-from-mask32*::word32 ⇒ nat

where

get-nat-from-mask32 w ≡

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-state* \Rightarrow *machine-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-context-table-addr* :: *machine-word* \Rightarrow *machine-word*
 \Rightarrow *phys-address*

where

```

compose-context-table-addr ctp cnr
≡ ((ucast (ctp >> 2)) << 6) + (ucast cnr << 2)

```

19.2 Virtual Address Translation

Get the context table phys address from the MMU registers

definition *get-context-table-addr* :: *MMU-state* ⇒ *phys-address*

where

```

get-context-table-addr mmu
≡ compose-context-table-addr (getCTPVal mmu) (getCNRVal mmu)

```

definition *va-list-index* :: *nat list* **where**

va-list-index ≡ [*va-t1-index*, *va-t2-index*, *va-t3-index*, 0]

definition *offset-index* :: *nat list* **where**

```

offset-index
≡ [ length-machine-word
    , length-machine-word-length-t1
    , length-machine-word-length-t1-length-t2
    , length-machine-word-length-t1-length-t2-length-t3
    ]

```

definition *index-len-table* :: *nat list* **where** *index-len-table* ≡ [8,6,6,0]

definition *n-context-tables* :: *nat* **where** *n-context-tables* ≡ 3

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

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

mem-context ⇒ *mem-val-type option*

where

mem-context-val asi add m ≡

let *asi8* = *word-of-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* \Rightarrow *phys-address* \Rightarrow
mem-context \Rightarrow *word32 option*

where

```

mem-context-val-w32 asi addr m  $\equiv$ 
  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  $\vee$  r1 = None  $\vee$  r2 = None  $\vee$  r3 = None then
    None
  else
    let byte0 = case r0 of Some v  $\Rightarrow$  v;
        byte1 = case r1 of Some v  $\Rightarrow$  v;
        byte2 = case r2 of Some v  $\Rightarrow$  v;
        byte3 = case r3 of Some v  $\Rightarrow$  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* \Rightarrow *virtua-address* \Rightarrow
mem-context \Rightarrow *nat* \Rightarrow (*phys-address* \times *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)

```

⟨proof⟩

termination

⟨proof⟩

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 ⇒ asi-type ⇒ bool **where**
mmu-writable f asi ≡
 if uint asi ∈ {8, 10} then
 if uint f ∈ {1,3} then True
 else False
 else if uint asi ∈ {9, 11} then
 if uint f ∈ {1,3,5,7} then True
 else False
 else False

definition *virt-to-phys* :: virtua-address ⇒ MMU-state ⇒ mem-context ⇒

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

where

```

out-reg-mod data-w32 win ur state  $\equiv$ 
  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 $in-reg-mod :: word32 \Rightarrow ('a::len) \text{ window-size} \Rightarrow user-reg-type \Rightarrow ('a) \text{ sparc-state} \Rightarrow ('a) \text{ sparc-state}$

where

```

in-reg-mod data-w32 win ur state  $\equiv$ 
  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 $user-reg-mod :: word32 \Rightarrow ('a::len) \text{ window-size} \Rightarrow user-reg-type \Rightarrow ('a) \text{ sparc-state} \Rightarrow ('a) \text{ sparc-state}$

where

```

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

```

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

where

```

sys-reg-val reg state  $\equiv$  (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* = 0b00000000000001111100000000000000 *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 ⇒
 (Exception list + word32)
where *fetch-instruction* *s* ≡
 — *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 ⇒ Inl [fetch-instruction-error]
 |Some *v* ⇒ 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 ⇒
 Exception list + instruction
where *decode-instruction* *w* ≡
 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 ⇒ ('a,('a::len window-size)) sparc-state-monad
where *get-curr-win* - ≡

```

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-word0* :: *instruction* ⇒ *word32* ⇒ *virtua-address* ⇒ *word32*

where *ld-word0 instr data-word address* ≡

```

  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

```

Operational semantics for swap.

definition $swap\text{-instr} :: instruction \Rightarrow ('a::len,unit) \text{sparc-state-monad}$
where $swap\text{-instr } instr \equiv$
 let $instr\text{-name} = fst \ instr;$
 $op\text{-list} = snd \ instr;$
 $flagi = get\text{-operand-flag } (op\text{-list}!0);$
 $rd = if \ instr\text{-name} \in \{load\text{-store-type } SWAPA\} \text{ then } rd \text{ is member } 4$
 $get\text{-operand-w5 } (op\text{-list}!4)$
 $else \text{ — } rd \text{ is member } 3$
 $get\text{-operand-w5 } (op\text{-list}!3)$
 in
 do
 $psr\text{-val} \leftarrow gets \ (\lambda s. (cpu\text{-reg-val } PSR \ s));$
 $s\text{-val} \leftarrow gets \ (\lambda s. (get\text{-S } psr\text{-val}));$
 if $instr\text{-name} \in \{load\text{-store-type } SWAPA\} \wedge s\text{-val} = 0$ then
 do
 raise-trap privileged-instruction;
 return ()
 od
 else if $instr\text{-name} \in \{load\text{-store-type } SWAPA\} \wedge flagi = 1$ then
 do
 raise-trap illegal-instruction;
 return ()
 od
 else
 $swap\text{-sub1 } instr \ rd \ s\text{-val}$
 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)
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*

\neq 0 *then False*

else True

else True

else False — *Unsupported instruction.*

)

```

)

end

function (sequential) seq-exec:: nat  $\Rightarrow$  ('a::len,unit) sparc-state-monad
where seq-exec 0 = return ()
|
seq-exec n = (do execute-instruction();
              (seq-exec (n-1))
              od)

<proof>
termination <proof>

type-synonym leon3-state = (word-length5) sparc-state

type-synonym ('e) leon3-state-monad = (leon3-state, 'e) det-monad

definition execute-leon3-instruction:: unit  $\Rightarrow$  (unit) leon3-state-monad
where execute-leon3-instruction  $\equiv$  execute-instruction

definition seq-exec-leon3:: nat  $\Rightarrow$  (unit) leon3-state-monad
where seq-exec-leon3  $\equiv$  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: simpler-gets-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 simpler-modify-def*),
(*simp add: write-cpu-tt-def write-cpu-def*),
(*simp add: DetMonad.bind-def h1-def h2-def simpler-modify-def*),
(*simp add: return-def simpler-gets-def*))

method *select-trap-proof1* =
((*simp add: select-trap-def exec-gets return-def*),
(*simp add: DetMonad.bind-def h1-def h2-def simpler-modify-def*),
(*simp add: write-cpu-tt-def write-cpu-def*),
(*simp add: DetMonad.bind-def h1-def h2-def simpler-modify-def*),
(*simp add: return-def simpler-gets-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: simpler-gets-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: simpler-gets-def bind-def h1-def h2-def Let-def simpler-modify-def*),
(*simp add: return-def*))

method *exe-proof-dispatch-rett* =
((*simp add: dispatch-instruction-def*),
(*simp add: simpler-gets-def bind-def h1-def h2-def Let-def*),
(*simp add: rett-instr-def*),
(*simp add: simpler-gets-def bind-def h1-def h2-def Let-def*))

lemma *write-cpu-result: snd (write-cpu w r s) = False*
{*proof*}

lemma *set-annul-result: snd (set-annul b s) = False*
{*proof*}

lemma *raise-trap-result : snd (raise-trap t s) = False*
{*proof*}

context
 includes *bit-operations-syntax*
begin

lemma *rett-instr-result: (fst i) = ctrl-type RETT ∧*

$(\text{get-ET } (\text{cpu-reg-val PSR } s) \neq 1 \wedge$
 $((\text{get-S } (\text{cpu-reg-val PSR } s))::\text{word1}) \neq 0 \wedge$
 $(\text{get-WIM-bit } (\text{nat } (((\text{uint } (\text{get-CWP } (\text{cpu-reg-val PSR } s))) + 1) \bmod \text{NWINDOWS})))$
 $(\text{cpu-reg-val WIM } s) = 0 \wedge$
 $((\text{AND}) (\text{get-addr } (\text{snd } i) s) (0b00000000000000000000000000000011::\text{word32}))$
 $= 0) \implies$
 $\text{snd } (\text{rett-instr } i s) = \text{False}$
 $\langle \text{proof} \rangle$

lemma *call-instr-result*: $(\text{fst } i) = \text{call-type CALL} \implies$
 $\text{snd } (\text{call-instr } i s) = \text{False}$
 $\langle \text{proof} \rangle$

lemma *branch-instr-result*: $(\text{fst } i) \in \{\text{bicc-type BE}, \text{bicc-type BNE}, \text{bicc-type BGU},$
 $\text{bicc-type BLE}, \text{bicc-type BL}, \text{bicc-type BGE}, \text{bicc-type BNEG}, \text{bicc-type BG},$
 $\text{bicc-type BCS}, \text{bicc-type BLEU}, \text{bicc-type BCC}, \text{bicc-type BA}, \text{bicc-type BN}\} \implies$
 $\text{snd } (\text{branch-instr } i s) = \text{False}$
 $\langle \text{proof} \rangle$

lemma *nop-instr-result*: $(\text{fst } i) = \text{nop-type NOP} \implies$
 $\text{snd } (\text{nop-instr } i s) = \text{False}$
 $\langle \text{proof} \rangle$

lemma *sethi-instr-result*: $(\text{fst } i) = \text{sethi-type SETHI} \implies$
 $\text{snd } (\text{sethi-instr } i s) = \text{False}$
 $\langle \text{proof} \rangle$

lemma *jmpl-instr-result*: $(\text{fst } i) = \text{ctrl-type JMPL} \implies$
 $\text{snd } (\text{jmpl-instr } i s) = \text{False}$
 $\langle \text{proof} \rangle$

lemma *save-restore-instr-result*: $(\text{fst } i) \in \{\text{ctrl-type SAVE}, \text{ctrl-type RESTORE}\}$
 \implies
 $\text{snd } (\text{save-restore-instr } i s) = \text{False}$
 $\langle \text{proof} \rangle$

lemma *flush-instr-result*: $(\text{fst } i) = \text{load-store-type FLUSH} \implies$
 $\text{snd } (\text{flush-instr } i s) = \text{False}$
 $\langle \text{proof} \rangle$

lemma *read-state-reg-instr-result*: $(\text{fst } i) \in \{\text{sreg-type RDY}, \text{sreg-type RDPSR},$
 $\text{sreg-type RDWIM}, \text{sreg-type RDTBR}\} \implies$
 $\text{snd } (\text{read-state-reg-instr } i s) = \text{False}$
 $\langle \text{proof} \rangle$

lemma *write-state-reg-instr-result*: $(\text{fst } i) \in \{\text{sreg-type WRY}, \text{sreg-type WRPSR},$
 $\text{sreg-type WRWIM}, \text{sreg-type WRTBR}\} \implies$
 $\text{snd } (\text{write-state-reg-instr } i s) = \text{False}$

<proof>

lemma *logical-instr-result*: $(fst\ i) \in \{logic\text{-type}\ ANDs, logic\text{-type}\ ANDcc, logic\text{-type}\ ANDN, logic\text{-type}\ ANDNcc, logic\text{-type}\ ORs, logic\text{-type}\ ORcc, logic\text{-type}\ ORN, logic\text{-type}\ XORs, logic\text{-type}\ XNOR\} \implies$
 $snd\ (logical\text{-instr}\ i\ s) = False$
<proof>

lemma *shift-instr-result*: $(fst\ i) \in \{shift\text{-type}\ SLL, shift\text{-type}\ SRL, shift\text{-type}\ SRA\} \implies$
 $snd\ (shift\text{-instr}\ i\ s) = False$
<proof>

method *add-sub-instr-proof* =
 $((simp\ add:\ Let\text{-def}),$
auto,
 $(simp\ add:\ write\text{-reg}\text{-def}\ simpler\text{-modify}\text{-def}),$
 $(simp\ add:\ simpler\text{-gets}\text{-def}\ bind\text{-def}),$
 $(simp\ add:\ get\text{-curr}\text{-win}\text{-def}\ simpler\text{-gets}\text{-def}),$
 $(simp\ add:\ write\text{-reg}\text{-def}\ write\text{-cpu}\text{-def}\ simpler\text{-modify}\text{-def}),$
 $(simp\ add:\ bind\text{-def}),$
 $(simp\ add:\ case\text{-prod}\text{-unfold}),$
 $(simp\ add:\ simpler\text{-gets}\text{-def}),$
 $(simp\ add:\ get\text{-curr}\text{-win}\text{-def}\ simpler\text{-gets}\text{-def}),$
 $(simp\ add:\ write\text{-reg}\text{-def}\ simpler\text{-modify}\text{-def}),$
 $(simp\ add:\ simpler\text{-gets}\text{-def}\ bind\text{-def}),$
 $(simp\ add:\ get\text{-curr}\text{-win}\text{-def}\ simpler\text{-gets}\text{-def}))$

lemma *add-instr-result*: $(fst\ i) \in \{arith\text{-type}\ ADD, arith\text{-type}\ ADDcc, arith\text{-type}\ ADDX\} \implies$
 $snd\ (add\text{-instr}\ i\ s) = False$
<proof>

lemma *sub-instr-result*: $(fst\ i) \in \{arith\text{-type}\ SUB, arith\text{-type}\ SUBcc, arith\text{-type}\ SUBX\} \implies$
 $snd\ (sub\text{-instr}\ i\ s) = False$
<proof>

lemma *mul-instr-result*: $(fst\ i) \in \{arith\text{-type}\ UMUL, arith\text{-type}\ SMUL, arith\text{-type}\ SMULcc\} \implies$
 $snd\ (mul\text{-instr}\ i\ s) = False$
<proof>

lemma *div-write-new-val-result*: $snd\ (div\text{-write}\text{-new}\text{-val}\ i\ result\ temp\text{-}V\ s) = False$
<proof>

lemma *div-result*: $snd\ (div\text{-comp}\ instr\ rs1\ rd\ operand2\ s) = False$
<proof>

lemma *div-instr-result*: $(fst\ i) \in \{arith\text{-}type\ UDIV, arith\text{-}type\ UDIVcc, arith\text{-}type\ SDIV\} \implies$
 $snd\ (div\text{-}instr\ i\ s) = False$
 $\langle proof \rangle$

lemma *load-sub2-result*: $snd\ (load\text{-}sub2\ address\ asi\ rd\ curr\text{-}win\ word0\ s) = False$
 $\langle proof \rangle$

lemma *load-sub3-result*: $snd\ (load\text{-}sub3\ instr\ curr\text{-}win\ rd\ asi\ address\ s) = False$
 $\langle proof \rangle$

lemma *load-sub1-result*: $snd\ (load\text{-}sub1\ i\ rd\ s\text{-}val\ s) = False$
 $\langle proof \rangle$

lemma *load-instr-result*: $(fst\ i) \in \{load\text{-}store\text{-}type\ LDSB, load\text{-}store\text{-}type\ LDUB, load\text{-}store\text{-}type\ LDUBA, load\text{-}store\text{-}type\ LDUH, load\text{-}store\text{-}type\ LD, load\text{-}store\text{-}type\ LDA, load\text{-}store\text{-}type\ LDD\} \implies$
 $snd\ (load\text{-}instr\ i\ s) = False$
 $\langle proof \rangle$

lemma *store-sub2-result*: $snd\ (store\text{-}sub2\ instr\ curr\text{-}win\ rd\ asi\ address\ s) = False$
 $\langle proof \rangle$

lemma *store-sub1-result*: $snd\ (store\text{-}sub1\ instr\ rd\ s\text{-}val\ s) = False$
 $\langle proof \rangle$

lemma *store-instr-result*: $(fst\ i) \in \{load\text{-}store\text{-}type\ STB, load\text{-}store\text{-}type\ STH, load\text{-}store\text{-}type\ ST, load\text{-}store\text{-}type\ STA, load\text{-}store\text{-}type\ STD\} \implies$
 $snd\ (store\text{-}instr\ i\ s) = False$
 $\langle proof \rangle$

lemma *supported-instr-set*: $supported\text{-}instruction\ i = True \implies$
 $i \in \{load\text{-}store\text{-}type\ LDSB, load\text{-}store\text{-}type\ LDUB, load\text{-}store\text{-}type\ LDUBA, load\text{-}store\text{-}type\ LDUH, load\text{-}store\text{-}type\ LD, load\text{-}store\text{-}type\ LDA, load\text{-}store\text{-}type\ LDD, load\text{-}store\text{-}type\ STB, load\text{-}store\text{-}type\ STH, load\text{-}store\text{-}type\ ST, load\text{-}store\text{-}type\ STA, load\text{-}store\text{-}type\ STD, sethi\text{-}type\ SETHI, nop\text{-}type\ NOP, logic\text{-}type\ ANDs, logic\text{-}type\ ANDcc, logic\text{-}type\ ANDN, logic\text{-}type\ ANDNcc, logic\text{-}type\ ORs, logic\text{-}type\ ORcc, logic\text{-}type\ ORN, logic\text{-}type\ XORs, logic\text{-}type\ XNOR, shift\text{-}type\ SLL, shift\text{-}type\ SRL, shift\text{-}type\ SRA, arith\text{-}type\ ADD, arith\text{-}type\ ADDcc, arith\text{-}type\ ADDX, arith\text{-}type\ SUB, arith\text{-}type\ SUBcc, arith\text{-}type\ SUBX, arith\text{-}type\ UMUL, arith\text{-}type\ SMUL, arith\text{-}type\ SMULcc, arith\text{-}type\ UDIV, arith\text{-}type\ UDIVcc, arith\text{-}type\ SDIV, ctrl\text{-}type\ SAVE, ctrl\text{-}type\ RESTORE, call\text{-}type\ CALL,$

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}

<proof>

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>

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>

lemma *execute-instr-sub1-result:* *snd (execute-instr-sub1 i s) = False*

<proof>

lemma *next-match :* *snd (execute-instruction () s) = False \implies*

NEXT s = Some (snd (fst (execute-instruction () s)))

<proof>

lemma *exec-ss1 :* $\exists s'. (execute-instruction () s = (s', False)) \implies$

$\exists s''. (execute-instruction() s = (s'', False))$

<proof>

lemma *exec-ss2 :* *snd (execute-instruction() s) = False \implies*

snd (execute-instruction () s) = False

<proof>

lemma *good-context-1 :* *good-context s \wedge s' = s \wedge*

(get-trap-set s') \neq {} \wedge (reset-trap-val s') = False \wedge get-ET (cpu-reg-val PSR s')

= 0

\implies False

<proof>

lemma *fetch-instr-result-1 :* $\neg (\exists e. \text{fetch-instruction } s' = \text{Inl } e) \implies$

$(\exists v. \text{fetch-instruction } s' = \text{Inr } v)$
 $\langle \text{proof} \rangle$

lemma *fetch-instr-result-2* : $(\exists v. \text{fetch-instruction } s' = \text{Inr } v) \implies$
 $\neg (\exists e. \text{fetch-instruction } s' = \text{Inl } e)$
 $\langle \text{proof} \rangle$

lemma *fetch-instr-result-3* : $(\exists e. \text{fetch-instruction } s' = \text{Inl } e) \implies$
 $\neg (\exists v. \text{fetch-instruction } s' = \text{Inr } v)$
 $\langle \text{proof} \rangle$

lemma *decode-instr-result-1* :
 $\neg (\exists v2. ((\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inr } v2) \implies$
 $(\exists e. ((\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inl } e)$
 $\langle \text{proof} \rangle$

lemma *decode-instr-result-2* :
 $(\exists e. ((\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inl } e) \implies$
 $\neg (\exists v2. ((\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inr } v2)$
 $\langle \text{proof} \rangle$

lemma *decode-instr-result-3* : $x = \text{decode-instruction } v1 \wedge y = \text{decode-instruction}$
 $v2$
 $\wedge v1 = v2 \implies x = y$
 $\langle \text{proof} \rangle$

lemma *decode-instr-result-4* :
 $\neg (\exists e. ((\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inl } e) \implies$
 $(\exists v2. ((\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inr } v2)$
 $\langle \text{proof} \rangle$

lemma *good-context-2* :
 $\text{good-context } (s::('a::\text{len}) \text{ sparc-state}) \wedge$
 $\text{fetch-instruction } (\text{delayed-pool-write } s) = \text{Inr } v1 \wedge$
 $\neg (\exists v2. (\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inr } v2)$
 $\implies \text{False}$
 $\langle \text{proof} \rangle$

lemma *good-context-3* :
 $\text{good-context } (s::('a::\text{len}) \text{ sparc-state}) \wedge$
 $s'' = \text{delayed-pool-write } s \wedge$
 $\text{fetch-instruction } s'' = \text{Inr } v1 \wedge$
 $(\text{decode-instruction } v1)::(\text{Exception list} + \text{instruction})) = \text{Inr } v2 \wedge$
 $\text{annul-val } s'' = \text{False} \wedge \text{supported-instruction } (\text{fst } v2) = \text{False}$
 $\implies \text{False}$
 $\langle \text{proof} \rangle$

lemma *good-context-4* :
 $\text{good-context } (s::('a::\text{len}) \text{ sparc-state}) \wedge$

lemma *execute-trap-result2* : $\neg(\text{reset-trap-val } s = \text{False} \wedge \text{get-ET } (\text{cpu-reg-val } \text{PSR } s) = 0) \implies$
 $\text{snd } (\text{execute-trap}() s) = \text{False}$
 $\langle \text{proof} \rangle$

lemma *exe-instr-all* :
 $\text{good-context } (s::('a::\text{len}) \text{ sparc-state}) \implies$
 $\text{snd } (\text{execute-instruction}() s) = \text{False}$
 $\langle \text{proof} \rangle$

lemma *dispatch-fail*:
 $\text{snd } (\text{execute-instruction}() (s::('a::\text{len}) \text{ sparc-state})) = \text{False} \wedge$
 $\text{get-trap-set } s = \{\}$ \wedge
 $\text{exe-mode-val } s \wedge$
 $\text{fetch-instruction } (\text{delayed-pool-write } s) = \text{Inr } v \wedge$
 $((\text{decode-instruction } v)::(\text{Exception list} + \text{instruction})) = \text{Inl } e$
 $\implies \text{False}$
 $\langle \text{proof} \rangle$

lemma *no-error* : $\text{good-context } s \implies \text{snd } (\text{execute-instruction } () s) = \text{False}$
 $\langle \text{proof} \rangle$

theorem *single-step* : $\text{good-context } s \implies \text{NEXT } s = \text{Some } (\text{snd } (\text{fst } (\text{execute-instruction } () s)))$
 $\langle \text{proof} \rangle$

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*: $s' = \text{snd } (\text{fst } (\text{write-cpu } w \text{ PC } s)) \wedge$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s))::\text{word1}) = 0 \implies$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *write-cpu-npc-privilege*: $s' = \text{snd } (\text{fst } (\text{write-cpu } w \text{ nPC } s)) \wedge$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s))::\text{word1}) = 0 \implies$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *write-cpu-y-privilege*: $s' = \text{snd } (\text{fst } (\text{write-cpu } w \text{ Y } s)) \wedge$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s))::\text{word1}) = 0$
 $\implies ((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *cpu-reg-mod-y-privilege*: $s' = \text{cpu-reg-mod } w \text{ Y } s \wedge$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s))::\text{word1}) = 0$
 $\implies ((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$

<proof>

lemma *cpu-reg-mod-asr-privilege*: $s' = \text{cpu-reg-mod } w \text{ (ASR } r) s \wedge$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s))::\text{word1}) = 0$
 $\implies ((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$
<proof>

lemma *global-reg-mod-privilege*: $s' = \text{global-reg-mod } w1 \text{ } n \text{ } w2 \text{ } s \wedge$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s))::\text{word1}) = 0 \implies$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$
<proof>

lemma *out-reg-mod-privilege*: $s' = \text{out-reg-mod } a \text{ } w \text{ } r \text{ } s \wedge$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s))::\text{word1}) = 0 \implies$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$
<proof>

lemma *in-reg-mod-privilege*: $s' = \text{in-reg-mod } a \text{ } w \text{ } r \text{ } s \wedge$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s))::\text{word1}) = 0 \implies$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$
<proof>

lemma *user-reg-mod-privilege*:
assumes $a1$: $s' = \text{user-reg-mod } d \text{ (} w::('a::\text{len}) \text{ window-size) } r$
 $(s::('a::\text{len}) \text{ sparc-state}) \wedge$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s))::\text{word1}) = 0$
shows $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$
<proof>

lemma *write-reg-privilege*: $s' = \text{snd } (\text{fst } (\text{write-reg } w1 \text{ } w2 \text{ } w3$
 $(s::('a::\text{len}) \text{ sparc-state}))) \wedge$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s))::\text{word1}) = 0 \implies$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$
<proof>

lemma *set-annul-privilege*: $s' = \text{snd } (\text{fst } (\text{set-annul } b \text{ } s)) \wedge$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s))::\text{word1}) = 0 \implies$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$
<proof>

lemma *set-reset-trap-privilege*: $s' = \text{snd } (\text{fst } (\text{set-reset-trap } b \text{ } s)) \wedge$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s))::\text{word1}) = 0 \implies$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$
<proof>

lemma *empty-delayed-pool-write-privilege*: $\text{get-delayed-pool } s = [] \wedge$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s))::\text{word1}) = 0 \wedge$
 $s' = \text{delayed-pool-write } s \implies$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$

$\langle \text{proof} \rangle$

lemma *raise-trap-privilege*:

$((\text{get-}S \text{ (cpu-reg-val PSR } s))::\text{word1}) = 0 \wedge$
 $s' = \text{snd (fst (raise-trap } t \ s)) \implies$
 $((\text{get-}S \text{ (cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *write-cpu-tt-privilege*: $s' = \text{snd (fst (write-cpu-tt } w \ s)) \wedge$

$((\text{get-}S \text{ (cpu-reg-val PSR } s))::\text{word1}) = 0$
 $\implies ((\text{get-}S \text{ (cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *emp-trap-set-privilege*: $s' = \text{emp-trap-set } s \wedge$

$((\text{get-}S \text{ (cpu-reg-val PSR } s))::\text{word1}) = 0$
 $\implies ((\text{get-}S \text{ (cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *sys-reg-mod-privilege*: $s' = \text{sys-reg-mod } w \ r \ s$

$\wedge ((\text{get-}S \text{ (cpu-reg-val PSR } s))::\text{word1}) = 0$
 $\implies ((\text{get-}S \text{ (cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

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 $((\text{get-}S \text{ (cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *mem-mod-w32-privilege*: $s' = \text{mem-mod-w32 } a1 \ a2 \ b \ d \ s \wedge$

$((\text{get-}S \text{ (cpu-reg-val PSR } s))::\text{word1}) = 0$
 $\implies ((\text{get-}S \text{ (cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *add-instr-cache-privilege*: $s' = \text{add-instr-cache } s \ \text{addr } y \ m \implies$

$((\text{get-}S \text{ (cpu-reg-val PSR } s))::\text{word1}) = 0 \implies$
 $((\text{get-}S \text{ (cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *add-data-cache-privilege*: $s' = \text{add-data-cache } s \ \text{addr } y \ m \implies$

$((\text{get-}S \text{ (cpu-reg-val PSR } s))::\text{word1}) = 0 \implies$
 $((\text{get-}S \text{ (cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *memory-read-privilege*:

assumes $a1: s' = \text{snd (memory-read asi addr } s) \wedge$
 $((\text{get-}S \text{ (cpu-reg-val PSR } s))::\text{word1}) = 0$
shows $((\text{get-}S \text{ (cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *get-curr-win-privilege*: $s' = \text{snd} (\text{fst} (\text{get-curr-win}() s)) \wedge$
 $((\text{get-S} (\text{cpu-reg-val PSR } s))::\text{word1}) = 0$
 $\implies ((\text{get-S} (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *load-sub2-privilege*:
assumes $a1: s' = \text{snd} (\text{fst} (\text{load-sub2} \text{ addr asi r win w } s))$
 $\wedge ((\text{get-S} (\text{cpu-reg-val PSR } s))::\text{word1}) = 0$
shows $((\text{get-S} (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *load-sub3-privilege*:
assumes $a1: s' = \text{snd} (\text{fst} (\text{load-sub3} \text{ instr curr-win rd asi address } s))$
 $\wedge ((\text{get-S} (\text{cpu-reg-val PSR } s))::\text{word1}) = 0$
shows $((\text{get-S} (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *load-sub1-privilege*:
assumes $a1: s' = \text{snd} (\text{fst} (\text{load-sub1} \text{ instr rd s-val } s))$
 $\wedge ((\text{get-S} (\text{cpu-reg-val PSR } s))::\text{word1}) = 0$
shows $((\text{get-S} (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *load-instr-privilege*: $s' = \text{snd} (\text{fst} (\text{load-instr} i s))$
 $\wedge ((\text{get-S} (\text{cpu-reg-val PSR } s))::\text{word1}) = 0$
 $\implies ((\text{get-S} (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *store-barrier-pending-mod-privilege*: $s' = \text{store-barrier-pending-mod } b s$
 $\wedge ((\text{get-S} (\text{cpu-reg-val PSR } s))::\text{word1}) = 0$
 $\implies ((\text{get-S} (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *store-word-mem-privilege*:
assumes $a1: \text{store-word-mem } s \text{ addr data byte-mask asi} = \text{Some } s' \wedge$
 $((\text{get-S} (\text{cpu-reg-val PSR } s))::\text{word1}) = 0$
shows $((\text{get-S} (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *flush-instr-cache-privilege*: $((\text{get-S} (\text{cpu-reg-val PSR } s))::\text{word1}) = 0 \implies$
 $s' = \text{flush-instr-cache } s \implies$
 $((\text{get-S} (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *flush-data-cache-privilege*: $((\text{get-S} (\text{cpu-reg-val PSR } s))::\text{word1}) = 0 \implies$
 $s' = \text{flush-data-cache } s \implies$
 $((\text{get-S} (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *flush-cache-all-privilege*: $((\text{get-S } (\text{cpu-reg-val PSR } s)))::\text{word1} = 0 \implies$
 $s' = \text{flush-cache-all } s \implies$
 $((\text{get-S } (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *memory-write-asi-privilege*:
assumes $a1: r = \text{memory-write-asi } \text{asi } \text{addr } \text{byte-mask } \text{data } s \wedge$
 $r = \text{Some } s' \wedge$
 $((\text{get-S } (\text{cpu-reg-val PSR } s)))::\text{word1} = 0$
shows $((\text{get-S } (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *memory-write-privilege*:
assumes $a1: r = \text{memory-write } \text{asi } \text{addr } \text{byte-mask } \text{data}$
 $(s::('a::\text{len}) \text{sparc-state})) \wedge$
 $r = \text{Some } s' \wedge$
 $((\text{get-S } (\text{cpu-reg-val PSR } s)))::\text{word1} = 0$
shows $((\text{get-S } (\text{cpu-reg-val PSR}$
 $(s'::('a::\text{len}) \text{sparc-state}))))::\text{word1} = 0$
 $\langle \text{proof} \rangle$

lemma *store-sub2-privilege*:
assumes $a1: s' = \text{snd } (\text{fst } (\text{store-sub2 } \text{instr } \text{curr-win } \text{rd } \text{asi } \text{address } s))$
 $\wedge ((\text{get-S } (\text{cpu-reg-val PSR } s)))::\text{word1} = 0$
shows $((\text{get-S } (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *store-sub1-privilege*:
assumes $a1: s' = \text{snd } (\text{fst } (\text{store-sub1 } \text{instr } \text{rd } s\text{-val}$
 $(s::('a::\text{len}) \text{sparc-state}))))$
 $\wedge ((\text{get-S } (\text{cpu-reg-val PSR } s)))::\text{word1} = 0$
shows $((\text{get-S } (\text{cpu-reg-val PSR}$
 $(s'::('a::\text{len}) \text{sparc-state}))))::\text{word1} = 0$
 $\langle \text{proof} \rangle$

lemma *store-instr-privilege*:
assumes $a1: s' = \text{snd } (\text{fst } (\text{store-instr } \text{instr}$
 $(s::('a::\text{len}) \text{sparc-state}))))$
 $\wedge ((\text{get-S } (\text{cpu-reg-val PSR } s)))::\text{word1} = 0$
shows $((\text{get-S } (\text{cpu-reg-val PSR}$
 $(s'::('a::\text{len}) \text{sparc-state}))))::\text{word1} = 0$
 $\langle \text{proof} \rangle$

lemma *sethi-instr-privilege*:
assumes $a1: s' = \text{snd } (\text{fst } (\text{sethi-instr } \text{instr}$
 $(s::('a::\text{len}) \text{sparc-state}))))$
 $\wedge ((\text{get-S } (\text{cpu-reg-val PSR } s)))::\text{word1} = 0$
shows $((\text{get-S } (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$

$\langle proof \rangle$

lemma *nop-instr-privilege*:
assumes $a1: s' = snd (fst (nop-instr instr$
 $(s::('a::len) sparc-state)))$
 $\wedge (((get-S (cpu-reg-val PSR s))::word1) = 0$
shows $(((get-S (cpu-reg-val PSR s'))::word1) = 0$
 $\langle proof \rangle$

lemma *ucast-0*: $(((get-S w)::word1) = 0 \implies get-S w = 0$
 $\langle proof \rangle$

lemma *ucast-02*: $get-S w = 0 \implies (((get-S w)::word1) = 0$
 $\langle proof \rangle$

lemma *ucast-s*: $(((get-S w)::word1) = 0 \implies$
 $(AND) w (0b0000000000000000000000010000000::word32) = 0$
 $\langle proof \rangle$

lemma *ucast-s2*: $(AND) w 0b0000000000000000000000010000000 = 0$
 $\implies (((get-S w)::word1) = 0$
 $\langle proof \rangle$

lemma *update-PSR-icc-1*: $w' = (AND) w (0b11111111000011111111111111111111::word32)$
 $\wedge (((get-S w)::word1) = 0$
 $\implies (((get-S w')::word1) = 0$
 $\langle proof \rangle$

lemma *and-num-1048576-128*: $(AND) (0b0000000000100000000000000000000::word32)$
 $(0b00000000000000000000000010000000::word32) = 0$
 $\langle proof \rangle$

lemma *and-num-2097152-128*: $(AND) (0b0000000001000000000000000000000::word32)$
 $(0b00000000000000000000000010000000::word32) = 0$
 $\langle proof \rangle$

lemma *and-num-4194304-128*: $(AND) (0b0000000010000000000000000000000::word32)$
 $(0b00000000000000000000000010000000::word32) = 0$
 $\langle proof \rangle$

lemma *and-num-8388608-128*: $(AND) (0b0000000100000000000000000000000::word32)$
 $(0b00000000000000000000000010000000::word32) = 0$
 $\langle proof \rangle$

lemma *or-and-s*: $(AND) w1 (0b00000000000000000000000010000000::word32) =$
 0
 $\wedge (AND) w2 (0b00000000000000000000000010000000::word32) = 0$
 $\implies (AND) ((OR) w1 w2) (0b00000000000000000000000010000000::word32) =$
 0

$\langle proof \rangle$

lemma *and-or-s*:

assumes $((get-S\ w1)::word1) = 0 \wedge$
 $(AND\ w2\ (0b00000000000000000000000010000000::word32) = 0$
shows $((get-S\ ((OR\ ((AND\ w1$
 $(0b11111111000011111111111111111111::word32))\ w2)))::word1) = 0$
 $\langle proof \rangle$

lemma *and-or-or-s*:

assumes $a1: ((get-S\ w1)::word1) = 0 \wedge$
 $(AND\ w2\ (0b00000000000000000000000010000000::word32) = 0 \wedge$
 $(AND\ w3\ (0b00000000000000000000000010000000::word32) = 0$
shows $((get-S\ ((OR\ ((OR\ ((AND\ w1$
 $(0b11111111000011111111111111111111::word32))\ w2)\ w3)))::word1) = 0$
 $\langle proof \rangle$

lemma *and-or-or-or-s*:

assumes $a1: ((get-S\ w1)::word1) = 0 \wedge$
 $(AND\ w2\ (0b00000000000000000000000010000000::word32) = 0 \wedge$
 $(AND\ w3\ (0b00000000000000000000000010000000::word32) = 0 \wedge$
 $(AND\ w4\ (0b00000000000000000000000010000000::word32) = 0$
shows $((get-S\ ((OR\ ((OR\ ((OR\ ((AND\ w1$
 $(0b11111111000011111111111111111111::word32))\ w2)\ w3)\ w4)))::word1) = 0$
 $\langle proof \rangle$

lemma *and-or-or-or-or-s*:

assumes $a1: ((get-S\ w1)::word1) = 0 \wedge$
 $(AND\ w2\ (0b00000000000000000000000010000000::word32) = 0 \wedge$
 $(AND\ w3\ (0b00000000000000000000000010000000::word32) = 0 \wedge$
 $(AND\ w4\ (0b00000000000000000000000010000000::word32) = 0 \wedge$
 $(AND\ w5\ (0b00000000000000000000000010000000::word32) = 0$
shows $((get-S\ ((OR\ ((OR\ ((OR\ ((OR\ ((AND\ w1$
 $(0b11111111000011111111111111111111::word32))\ w2)\ w3)\ w4)\ w5)))::word1)$
 $= 0$
 $\langle proof \rangle$

lemma *write-cpu-PSR-icc-privilege*:

assumes $a1: s' = snd\ (fst\ (write-cpu\ (update-PSR-icc\ n-val\ z-val\ v-val\ c-val$
 $(cpu-reg-val\ PSR\ s))$
 PSR
 $(s::('a::len)\ sparc-state)))$
 $\wedge\ (((get-S\ (cpu-reg-val\ PSR\ s)))::word1) = 0$
shows $((get-S\ (cpu-reg-val\ PSR\ s'))::word1) = 0$
 $\langle proof \rangle$

lemma *and-num-4294967167-128*: $(AND)\ (0b1111111111111111111111111011111111::word32)$
 $(0b00000000000000000000000010000000::word32) = 0$
 $\langle proof \rangle$

lemma *add-instr-sub1-privilege:*
assumes $a1: s' = \text{snd } (\text{fst } (\text{add-instr-sub1 } \text{instr-name } \text{result } \text{rs1-val } \text{operand2}$
 $(s::('a::\text{len}) \text{sparc-state})))$
 $\wedge (((\text{get-S } (\text{cpu-reg-val } \text{PSR } s)))::\text{word1}) = 0$
shows $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *add-instr-privilege:*
assumes $a1: s' = \text{snd } (\text{fst } (\text{add-instr } \text{instr}$
 $(s::('a::\text{len}) \text{sparc-state})))$
 $\wedge (((\text{get-S } (\text{cpu-reg-val } \text{PSR } s)))::\text{word1}) = 0$
shows $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *sub-instr-sub1-privilege:*
assumes $a1: s' = \text{snd } (\text{fst } (\text{sub-instr-sub1 } \text{instr-name } \text{result } \text{rs1-val } \text{operand2}$
 $(s::('a::\text{len}) \text{sparc-state})))$
 $\wedge (((\text{get-S } (\text{cpu-reg-val } \text{PSR } s)))::\text{word1}) = 0$
shows $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *sub-instr-privilege:*
assumes $a1: s' = \text{snd } (\text{fst } (\text{sub-instr } \text{instr}$
 $(s::('a::\text{len}) \text{sparc-state})))$
 $\wedge (((\text{get-S } (\text{cpu-reg-val } \text{PSR } s)))::\text{word1}) = 0$
shows $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *mul-instr-sub1-privilege:*
assumes $a1: s' = \text{snd } (\text{fst } (\text{mul-instr-sub1 } \text{instr-name } \text{result}$
 $(s::('a::\text{len}) \text{sparc-state})))$
 $\wedge (((\text{get-S } (\text{cpu-reg-val } \text{PSR } s)))::\text{word1}) = 0$
shows $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *mul-instr-privilege:*
assumes $a1: s' = \text{snd } (\text{fst } (\text{mul-instr } \text{instr}$
 $(s::('a::\text{len}) \text{sparc-state})))$
 $\wedge (((\text{get-S } (\text{cpu-reg-val } \text{PSR } s)))::\text{word1}) = 0$
shows $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *div-write-new-val-privilege:*
assumes $a1: s' = \text{snd } (\text{fst } (\text{div-write-new-val } i \text{result } \text{temp-V}$
 $(s::('a::\text{len}) \text{sparc-state})))$
 $\wedge (((\text{get-S } (\text{cpu-reg-val } \text{PSR } s)))::\text{word1}) = 0$
shows $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *div-comp-privilege*:
assumes $a1: s' = \text{snd } (\text{fst } (\text{div-comp instr rs1 rd operand2}$
 $(s::('a::\text{len}) \text{sparc-state})))$
 $\wedge (((\text{get-S } (\text{cpu-reg-val PSR } s)))::\text{word1}) = 0$
shows $((\text{get-S } (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *div-instr-privilege*:
assumes $a1: s' = \text{snd } (\text{fst } (\text{div-instr instr}$
 $(s::('a::\text{len}) \text{sparc-state})))$
 $\wedge (((\text{get-S } (\text{cpu-reg-val PSR } s)))::\text{word1}) = 0$
shows $((\text{get-S } (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *save-restore-sub1-privilege*:
assumes $a1: s' = \text{snd } (\text{fst } (\text{save-restore-sub1 result new-cwp rd}$
 $(s::('a::\text{len}) \text{sparc-state})))$
 $\wedge (((\text{get-S } (\text{cpu-reg-val PSR } s)))::\text{word1}) = 0$
shows $((\text{get-S } (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

method *save-restore-instr-privilege-proof* = (
 $(\text{simp add: save-restore-instr-def}),$
 $(\text{simp add: Let-def}),$
 $(\text{simp add: simpler-gets-def bind-def h1-def h2-def Let-def}),$
 $(\text{simp add: case-prod-unfold}),$
 $\text{auto},$
 $(\text{blast intro: get-curr-win-privilege raise-trap-privilege}),$
 $(\text{simp add: simpler-gets-def bind-def h1-def h2-def Let-def case-prod-unfold}),$
 $(\text{blast intro: get-curr-win-privilege save-restore-sub1-privilege})$
 $)$

lemma *save-restore-instr-privilege*:
assumes $a1: s' = \text{snd } (\text{fst } (\text{save-restore-instr instr}$
 $(s::('a::\text{len}) \text{sparc-state})))$
 $\wedge (((\text{get-S } (\text{cpu-reg-val PSR } s)))::\text{word1}) = 0$
shows $((\text{get-S } (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *call-instr-privilege*:
assumes $a1: s' = \text{snd } (\text{fst } (\text{call-instr instr}$
 $(s::('a::\text{len}) \text{sparc-state})))$
 $\wedge (((\text{get-S } (\text{cpu-reg-val PSR } s)))::\text{word1}) = 0$
shows $((\text{get-S } (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *jmp-instr-privilege*:
assumes $a1: s' = \text{snd } (\text{fst } (\text{jmp-instr instr}$

$(s::('a::len) \text{ sparc-state}))$
 $\wedge (((\text{get-S } (\text{cpu-reg-val PSR } s)))::\text{word1}) = 0$
shows $(((\text{get-S } (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *rett-instr-privilege*:
assumes $a1: \text{snd } (\text{rett-instr } i \ s) = \text{False} \wedge$
 $s' = \text{snd } (\text{fst } (\text{rett-instr } \text{instr}$
 $(s::('a::len) \text{ sparc-state}))$
 $\wedge (((\text{get-S } (\text{cpu-reg-val PSR } s)))::\text{word1}) = 0$
shows $(((\text{get-S } (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

method *read-state-reg-instr-privilege-proof* = (
 $(\text{simp add: read-state-reg-instr-def}),$
 $(\text{simp add: Let-def}),$
 $(\text{simp add: simpler-gets-def bind-def h1-def h2-def Let-def}),$
 $(\text{simp add: case-prod-unfold})$
 $)$

lemma *read-state-reg-instr-privilege*:
assumes $a1: s' = \text{snd } (\text{fst } (\text{read-state-reg-instr } \text{instr}$
 $(s::('a::len) \text{ sparc-state}))$
 $\wedge (((\text{get-S } (\text{cpu-reg-val PSR } s)))::\text{word1}) = 0$
shows $(((\text{get-S } (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

method *write-state-reg-instr-privilege-proof* = (
 $(\text{simp add: write-state-reg-instr-def}),$
 $(\text{simp add: Let-def}),$
 $(\text{simp add: simpler-gets-def bind-def h1-def h2-def Let-def}),$
 $(\text{simp add: case-prod-unfold})$
 $)$

lemma *write-state-reg-instr-privilege*:
assumes $a1: s' = \text{snd } (\text{fst } (\text{write-state-reg-instr } \text{instr}$
 $(s::('a::len) \text{ sparc-state}))$
 $\wedge (((\text{get-S } (\text{cpu-reg-val PSR } s)))::\text{word1}) = 0$
shows $(((\text{get-S } (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *flush-instr-privilege*:
assumes $a1: s' = \text{snd } (\text{fst } (\text{flush-instr } \text{instr}$
 $(s::('a::len) \text{ sparc-state}))$
 $\wedge (((\text{get-S } (\text{cpu-reg-val PSR } s)))::\text{word1}) = 0$
shows $(((\text{get-S } (\text{cpu-reg-val PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma *branch-instr-privilege*:

assumes $a1: s' = \text{snd } (\text{fst } (\text{branch-instr } \text{instr} \text{ (s::('a::len) \text{sparc-state}))))$
 $\wedge (((\text{get-S } (\text{cpu-reg-val } \text{PSR } s)))::\text{word1}) = 0$
shows $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

method $\text{dispath-instr-privilege-proof} =$ (
 $(\text{simp add: dispatch-instruction-def}),$
 $(\text{simp add: simpler-gets-def bind-def h1-def h2-def Let-def}),$
 $(\text{simp add: Let-def})$
 $)$

lemma $\text{dispath-instr-privilege}$:
assumes $a1: \text{snd } (\text{dispatch-instruction } \text{instr } s) = \text{False} \wedge$
 $s' = \text{snd } (\text{fst } (\text{dispatch-instruction } \text{instr } s))$
 $\wedge (((\text{get-S } (\text{cpu-reg-val } \text{PSR } s)))::\text{word1}) = 0$
shows $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

lemma $\text{execute-instr-sub1-privilege}$:
assumes $a1: \text{snd } (\text{execute-instr-sub1 } i s) = \text{False} \wedge$
 $s' = \text{snd } (\text{fst } (\text{execute-instr-sub1 } i s))$
 $\wedge (((\text{get-S } (\text{cpu-reg-val } \text{PSR } s)))::\text{word1}) = 0$
shows $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

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 $a1: \text{get-delayed-pool } s = [] \wedge \text{get-trap-set } s = \{\}$ \wedge
 $\text{snd } (\text{execute-instruction } () s) = \text{False} \wedge$
 $s' = \text{snd } (\text{fst } (\text{execute-instruction } () s)) \wedge$
 $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s)))::\text{word1} = 0$
shows $((\text{get-S } (\text{cpu-reg-val } \text{PSR } s'))::\text{word1}) = 0$
 $\langle \text{proof} \rangle$

26 Single step non-interference property.

definition $\text{user-accessible}:: ('a::\text{len}) \text{sparc-state} \Rightarrow \text{phys-address} \Rightarrow \text{bool}$ **where**
 $\text{user-accessible } s \text{ pa} \equiv \exists va \text{ p. } (\text{virt-to-phys } va (\text{mmu } s) (\text{mem } s)) = \text{Some } p \wedge$
 $\text{mmu-readable } (\text{get-acc-flag } (\text{snd } p)) \text{ 10} \wedge$
 $(\text{fst } p) = \text{pa} \text{ — Passing } \text{asi} = 8 \text{ is the same.}$

lemma user-accessible-8 :
assumes $a1: \text{mmu-readable } (\text{get-acc-flag } (\text{snd } p)) \text{ 8}$
shows $\text{mmu-readable } (\text{get-acc-flag } (\text{snd } p)) \text{ 10}$
 $\langle \text{proof} \rangle$

definition *mem-equal*:: ('a) *sparc-state* \Rightarrow ('a) *sparc-state* \Rightarrow *phys-address* \Rightarrow *bool* **where**
mem-equal *s1* *s2* *pa* \equiv
 $(\text{mem } s1) \ 8 \ (pa \ \text{AND} \ 68719476732) = (\text{mem } s2) \ 8 \ (pa \ \text{AND} \ 68719476732) \wedge$
 $(\text{mem } s1) \ 8 \ ((pa \ \text{AND} \ 68719476732) + 1) = (\text{mem } s2) \ 8 \ ((pa \ \text{AND} \ 68719476732)$
 $+ 1) \wedge$
 $(\text{mem } s1) \ 8 \ ((pa \ \text{AND} \ 68719476732) + 2) = (\text{mem } s2) \ 8 \ ((pa \ \text{AND} \ 68719476732)$
 $+ 2) \wedge$
 $(\text{mem } s1) \ 8 \ ((pa \ \text{AND} \ 68719476732) + 3) = (\text{mem } s2) \ 8 \ ((pa \ \text{AND} \ 68719476732)$
 $+ 3) \wedge$
 $(\text{mem } s1) \ 9 \ (pa \ \text{AND} \ 68719476732) = (\text{mem } s2) \ 9 \ (pa \ \text{AND} \ 68719476732) \wedge$
 $(\text{mem } s1) \ 9 \ ((pa \ \text{AND} \ 68719476732) + 1) = (\text{mem } s2) \ 9 \ ((pa \ \text{AND} \ 68719476732)$
 $+ 1) \wedge$
 $(\text{mem } s1) \ 9 \ ((pa \ \text{AND} \ 68719476732) + 2) = (\text{mem } s2) \ 9 \ ((pa \ \text{AND} \ 68719476732)$
 $+ 2) \wedge$
 $(\text{mem } s1) \ 9 \ ((pa \ \text{AND} \ 68719476732) + 3) = (\text{mem } s2) \ 9 \ ((pa \ \text{AND} \ 68719476732)$
 $+ 3) \wedge$
 $(\text{mem } s1) \ 10 \ (pa \ \text{AND} \ 68719476732) = (\text{mem } s2) \ 10 \ (pa \ \text{AND} \ 68719476732) \wedge$
 $(\text{mem } s1) \ 10 \ ((pa \ \text{AND} \ 68719476732) + 1) = (\text{mem } s2) \ 10 \ ((pa \ \text{AND} \ 68719476732)$
 $+ 1) \wedge$
 $(\text{mem } s1) \ 10 \ ((pa \ \text{AND} \ 68719476732) + 2) = (\text{mem } s2) \ 10 \ ((pa \ \text{AND} \ 68719476732)$
 $+ 2) \wedge$
 $(\text{mem } s1) \ 10 \ ((pa \ \text{AND} \ 68719476732) + 3) = (\text{mem } s2) \ 10 \ ((pa \ \text{AND} \ 68719476732)$
 $+ 3) \wedge$
 $(\text{mem } s1) \ 11 \ (pa \ \text{AND} \ 68719476732) = (\text{mem } s2) \ 11 \ (pa \ \text{AND} \ 68719476732) \wedge$
 $(\text{mem } s1) \ 11 \ ((pa \ \text{AND} \ 68719476732) + 1) = (\text{mem } s2) \ 11 \ ((pa \ \text{AND} \ 68719476732)$
 $+ 1) \wedge$
 $(\text{mem } s1) \ 11 \ ((pa \ \text{AND} \ 68719476732) + 2) = (\text{mem } s2) \ 11 \ ((pa \ \text{AND} \ 68719476732)$
 $+ 2) \wedge$
 $(\text{mem } s1) \ 11 \ ((pa \ \text{AND} \ 68719476732) + 3) = (\text{mem } s2) \ 11 \ ((pa \ \text{AND} \ 68719476732)$
 $+ 3)$

low-equal defines the equivalence relation over two sparc states that is an analogy to the $=_L$ relation over memory contexts in the traditional non-interference theorem.

definition *low-equal*:: ('a::len) *sparc-state* \Rightarrow ('a) *sparc-state* \Rightarrow *bool* **where**
low-equal *s1* *s2* \equiv
 $(\text{cpu-reg } s1) = (\text{cpu-reg } s2) \wedge$
 $(\text{user-reg } s1) = (\text{user-reg } s2) \wedge$
 $(\text{sys-reg } s1) = (\text{sys-reg } s2) \wedge$
 $(\forall va. (\text{virt-to-phys } va \ (\text{mmu } s1) \ (\text{mem } s1)) = (\text{virt-to-phys } va \ (\text{mmu } s2) \ (\text{mem } s2))) \wedge$
 $(\forall pa. (\text{user-accessible } s1 \ pa) \longrightarrow \text{mem-equal } s1 \ s2 \ pa) \wedge$
 $(\text{mmu } s1) = (\text{mmu } s2) \wedge$
 $(\text{state-var } s1) = (\text{state-var } s2) \wedge$
 $(\text{traps } s1) = (\text{traps } s2) \wedge$
 $(\text{undef } s1) = (\text{undef } s2)$

lemma *low-equal-com*: $low\text{-}equal\ s1\ s2 \implies low\text{-}equal\ s2\ s1$
(proof)

lemma *non-exe-mode-equal*: $exe\text{-}mode\text{-}val\ s = False \wedge$
 $get\text{-}trap\text{-}set\ s = \{\}$ \wedge
 $Some\ t = NEXT\ s \implies$
 $t = s$
(proof)

lemma *exe-mode-low-equal*:
assumes *a1*: $low\text{-}equal\ s1\ s2$
shows $exe\text{-}mode\text{-}val\ s1 = exe\text{-}mode\text{-}val\ s2$
(proof)

lemma *mem-val-mod-state*: $mem\text{-}val\text{-}alt\ asi\ a\ s = mem\text{-}val\text{-}alt\ asi\ a$
($s(|cpu\text{-}reg := new\text{-}cpu\text{-}reg,$
 $user\text{-}reg := new\text{-}user\text{-}reg,$
 $dwrite := new\text{-}dwrite,$
 $state\text{-}var := new\text{-}state\text{-}var,$
 $traps := new\text{-}traps,$
 $undef := new\text{-}undef|)$)
(proof)

lemma *mem-val-w32-mod-state*: $mem\text{-}val\text{-}w32\ asi\ a\ s = mem\text{-}val\text{-}w32\ asi\ a$
($s(|cpu\text{-}reg := new\text{-}cpu\text{-}reg,$
 $user\text{-}reg := new\text{-}user\text{-}reg,$
 $dwrite := new\text{-}dwrite,$
 $state\text{-}var := new\text{-}state\text{-}var,$
 $traps := new\text{-}traps,$
 $undef := new\text{-}undef|)$)
(proof)

lemma *load-word-mem-mod-state*: $load\text{-}word\text{-}mem\ s\ addr\ asi = load\text{-}word\text{-}mem$
($s(|cpu\text{-}reg := new\text{-}cpu\text{-}reg,$
 $user\text{-}reg := new\text{-}user\text{-}reg,$
 $dwrite := new\text{-}dwrite,$
 $state\text{-}var := new\text{-}state\text{-}var,$
 $traps := new\text{-}traps,$
 $undef := new\text{-}undef|)$) $addr\ asi$
(proof)

lemma *load-word-mem2-mod-state*:
 $fst\ (case\ load\text{-}word\text{-}mem\ s\ addr\ asi\ of\ None \implies (None, s)$
 $\quad | Some\ w \implies (Some\ w, add\text{-}data\text{-}cache\ s\ addr\ w\ 15)) =$
 $fst\ (case\ load\text{-}word\text{-}mem\ (s(|cpu\text{-}reg := new\text{-}cpu\text{-}reg,$
 $user\text{-}reg := new\text{-}user\text{-}reg,$
 $dwrite := new\text{-}dwrite,$
 $state\text{-}var := new\text{-}state\text{-}var,$

traps := new-traps,
undef := new-undef)) addr asi 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, 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>

lemma *load-word-mem3-mod-state:*

fst (case load-word-mem s addr asi of None \Rightarrow (None, s
| Some w \Rightarrow (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 \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, 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>

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
 <proof>

lemma *read-dcache2-mod-state:*

$$\text{fst (case read-data-cache } s \text{ addr of None } \Rightarrow (\text{None}, s)$$

$$\quad | \text{Some } w \Rightarrow (\text{Some } w, s)) =$$

$$\text{fst (case read-data-cache (s(|cpu-reg := new-cpu-reg,$$

$$\quad \text{user-reg := new-user-reg,$$

$$\quad \text{dwrite := new-dwrite,$$

$$\quad \text{state-var := new-state-var,$$

$$\quad \text{traps := new-traps,$$

$$\quad \text{undef := new-undef})) addr of}$$

$$\quad \text{None } \Rightarrow (\text{None}, (\text{s(|cpu-reg := new-cpu-reg,$$

$$\quad \text{user-reg := new-user-reg,$$

$$\quad \text{dwrite := new-dwrite,$$

$$\quad \text{state-var := new-state-var,$$

$$\quad \text{traps := new-traps,$$

$$\quad \text{undef := new-undef}))$$

$$\quad | \text{Some } w \Rightarrow (\text{Some } w, (\text{s(|cpu-reg := new-cpu-reg,$$

$$\quad \text{user-reg := new-user-reg,$$

$$\quad \text{dwrite := new-dwrite,$$

$$\quad \text{state-var := new-state-var,$$

$$\quad \text{traps := new-traps,$$

$$\quad \text{undef := new-undef}))))$$

$$\langle \text{proof} \rangle$$

lemma *read-icache-mod-state: read-instr-cache s addr = read-instr-cache*

$$(\text{s(|cpu-reg := new-cpu-reg,$$

$$\quad \text{user-reg := new-user-reg,$$

$$\quad \text{dwrite := new-dwrite,$$

$$\quad \text{state-var := new-state-var,$$

$$\quad \text{traps := new-traps,$$

$$\quad \text{undef := new-undef})) addr$$

$$\langle \text{proof} \rangle$$

lemma *read-icache2-mod-state:*

$$\text{fst (case read-instr-cache } s \text{ addr of None } \Rightarrow (\text{None}, s)$$

$$\quad | \text{Some } w \Rightarrow (\text{Some } w, s)) =$$

$$\text{fst (case read-instr-cache (s(|cpu-reg := new-cpu-reg,$$

$$\quad \text{user-reg := new-user-reg,$$

$$\quad \text{dwrite := new-dwrite,$$

$$\quad \text{state-var := new-state-var,$$

$$\quad \text{traps := new-traps,$$

$$\quad \text{undef := new-undef})) addr of}$$

$$\quad \text{None } \Rightarrow (\text{None}, (\text{s(|cpu-reg := new-cpu-reg,$$

$$\quad \text{user-reg := new-user-reg,$$

$$\quad \text{dwrite := new-dwrite,$$

$$\quad \text{state-var := new-state-var,$$

$$\quad \text{traps := new-traps,$$

$$\quad \text{undef := new-undef}))$$

$$\quad | \text{Some } w \Rightarrow (\text{Some } w, (\text{s(|cpu-reg := new-cpu-reg,$$

$$\quad \text{user-reg := new-user-reg,$$

$$\quad \text{dwrite := new-dwrite,$$

$state-var := new-state-var,$
 $traps := new-traps,$
 $undef := new-undef))))$
 ⟨proof⟩

lemma *mem-read-mod-state*: $fst (memory-read asi addr s) =$
 $fst (memory-read asi addr$
 $(s(|cpu-reg := new-cpu-reg,$
 $user-reg := new-user-reg,$
 $dwrite := new-dwrite,$
 $state-var := new-state-var,$
 $traps := new-traps,$
 $undef := new-undef))))$
 ⟨proof⟩

lemma *insert-trap-mem*: $fst (memory-read asi addr s) =$
 $fst (memory-read asi addr (s(|traps := new-traps))))$
 ⟨proof⟩

lemma *cpu-reg-mod-mem*: $fst (memory-read asi addr s) =$
 $fst (memory-read asi addr (s(|cpu-reg := new-cpu-reg))))$
 ⟨proof⟩

lemma *user-reg-mod-mem*: $fst (memory-read asi addr s) =$
 $fst (memory-read asi addr (s(|user-reg := new-user-reg))))$
 ⟨proof⟩

lemma *annul-mem*: $fst (memory-read asi addr s) =$
 $fst (memory-read asi addr$
 $(s(|state-var := new-state-var,$
 $cpu-reg := new-cpu-reg))))$
 ⟨proof⟩

lemma *state-var-mod-mem*: $fst (memory-read asi addr s) =$
 $fst (memory-read asi addr (s(|state-var := new-state-var))))$
 ⟨proof⟩

lemma *mod-state-low-equal*: $low-equal s1 s2 \wedge$
 $t1 = (s1(|cpu-reg := new-cpu-reg,$
 $user-reg := new-user-reg,$
 $dwrite := new-dwrite,$
 $state-var := new-state-var,$
 $traps := new-traps,$
 $undef := new-undef)) \wedge$
 $t2 = (s2(|cpu-reg := new-cpu-reg,$
 $user-reg := new-user-reg,$
 $dwrite := new-dwrite,$
 $state-var := new-state-var,$
 $traps := new-traps,$

$undef := new-undef)) \implies$
 $low-equal\ t1\ t2$
 $\langle proof \rangle$

lemma *user-reg-state-mod-low-equal*:
assumes $a1: low-equal\ s1\ s2 \wedge$
 $t1 = (s1\ (\!|user-reg := new-user-reg\!|)) \wedge$
 $t2 = (s2\ (\!|user-reg := new-user-reg\!|))$
shows $low-equal\ t1\ t2$
 $\langle proof \rangle$

lemma *mod-trap-low-equal*:
assumes $a1: low-equal\ s1\ s2 \wedge$
 $t1 = (s1\ (\!|traps := new-traps\!|)) \wedge$
 $t2 = (s2\ (\!|traps := new-traps\!|))$
shows $low-equal\ t1\ t2$
 $\langle proof \rangle$

lemma *state-var-low-equal*: $low-equal\ s1\ s2 \implies$
 $state-var\ s1 = state-var\ s2$
 $\langle proof \rangle$

lemma *state-var2-low-equal*:
assumes $a1: low-equal\ s1\ s2 \wedge$
 $t1 = (s1\ (\!|state-var := new-state-var\!|)) \wedge$
 $t2 = (s2\ (\!|state-var := new-state-var\!|))$
shows $low-equal\ t1\ t2$
 $\langle proof \rangle$

lemma *traps-low-equal*: $low-equal\ s1\ s2 \implies traps\ s1 = traps\ s2$
 $\langle proof \rangle$

lemma *s-low-equal*: $low-equal\ s1\ s2 \implies$
 $(get-S\ (cpu-reg-val\ PSR\ s1)) = (get-S\ (cpu-reg-val\ PSR\ s2))$
 $\langle proof \rangle$

lemma *cpu-reg-val-low-equal*: $low-equal\ s1\ s2 \implies$
 $(cpu-reg-val\ cr\ s1) = (cpu-reg-val\ cr\ s2)$
 $\langle proof \rangle$

lemma *get-curr-win-low-equal*: $low-equal\ s1\ s2 \implies$
 $(fst\ (fst\ (get-curr-win\ ()\ s1))) = (fst\ (fst\ (get-curr-win\ ()\ s2)))$
 $\langle proof \rangle$

lemma *get-curr-win2-low-equal*: $low-equal\ s1\ s2 \implies$
 $t1 = (snd\ (fst\ (get-curr-win\ ()\ s1))) \implies$
 $t2 = (snd\ (fst\ (get-curr-win\ ()\ s2))) \implies$
 $low-equal\ t1\ t2$
 $\langle proof \rangle$

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))))
 <proof>

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)
 <proof>

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)))
 <proof>

lemma *sys-reg-low-equal*: *low-equal s1 s2* \implies
sys-reg s1 = *sys-reg s2*
 <proof>

lemma *user-reg-low-equal*: *low-equal s1 s2* \implies
user-reg s1 = *user-reg s2*
 <proof>

lemma *user-reg-val-low-equal*: *low-equal s1 s2* \implies
user-reg-val win ur s1 = *user-reg-val win ur s2*
 <proof>

lemma *get-operand2-low-equal*: *low-equal s1 s2* \implies
get-operand2 op-list s1 = *get-operand2 op-list s2*
 <proof>

lemma *mem-val-mod-cache*: *mem-val-alt asi a s* =
mem-val-alt asi a (s|cache := new-cache)
 <proof>

lemma *mem-val-w32-mod-cache*: *mem-val-w32 asi a s* =
mem-val-w32 asi a (s|cache := new-cache)
 <proof>

lemma *load-word-mem-mod-cache*:
load-word-mem s addr asi =
load-word-mem (s|cache := new-cache) addr asi
 <proof>

lemma *memory-read-8-mod-cache*:

$\text{fst } (\text{memory-read } 8 \text{ addr } s) = \text{fst } (\text{memory-read } 8 \text{ addr } (s(\text{cache} := \text{new-cache})))$
 $\langle \text{proof} \rangle$

lemma *memory-read-10-mod-cache*:

$\text{fst } (\text{memory-read } 10 \text{ addr } s) = \text{fst } (\text{memory-read } 10 \text{ addr } (s(\text{cache} := \text{new-cache})))$
 $\langle \text{proof} \rangle$

lemma *mem-equal-mod-cache*: $\text{mem-equal } s1 \ s2 \ pa \implies$

$\text{mem-equal } (s1(\text{cache} := \text{new-cache1})) \ (s2(\text{cache} := \text{new-cache2})) \ pa$
 $\langle \text{proof} \rangle$

lemma *user-accessible-mod-cache*: $\text{user-accessible } (s(\text{cache} := \text{new-cache})) \ pa =$
 $\text{user-accessible } s \ pa$

$\langle \text{proof} \rangle$

lemma *mem-equal-mod-user-reg*: $\text{mem-equal } s1 \ s2 \ pa \implies$

$\text{mem-equal } (s1(\text{user-reg} := \text{new-user-reg1})) \ (s2(\text{user-reg} := \text{user-reg2})) \ pa$
 $\langle \text{proof} \rangle$

lemma *user-accessible-mod-user-reg*: $\text{user-accessible } (s(\text{user-reg} := \text{new-user-reg}))$
 $pa =$

$\text{user-accessible } s \ pa$

$\langle \text{proof} \rangle$

lemma *mem-equal-mod-cpu-reg*: $\text{mem-equal } s1 \ s2 \ pa \implies$

$\text{mem-equal } (s1(\text{cpu-reg} := \text{new-cpu1})) \ (s2(\text{cpu-reg} := \text{cpu-reg2})) \ pa$
 $\langle \text{proof} \rangle$

lemma *user-accessible-mod-cpu-reg*: $\text{user-accessible } (s(\text{cpu-reg} := \text{new-cpu-reg})) \ pa =$
 $=$

$\text{user-accessible } s \ pa$

$\langle \text{proof} \rangle$

lemma *mem-equal-mod-trap*: $\text{mem-equal } s1 \ s2 \ pa \implies$

$\text{mem-equal } (s1(\text{traps} := \text{new-traps1})) \ (s2(\text{traps} := \text{traps2})) \ pa$
 $\langle \text{proof} \rangle$

lemma *user-accessible-mod-trap*: $\text{user-accessible } (s(\text{traps} := \text{new-traps})) \ pa =$
 $\text{user-accessible } s \ pa$

$\langle \text{proof} \rangle$

lemma *mem-equal-annul*: $\text{mem-equal } s1 \ s2 \ pa \implies$

$\text{mem-equal } (s1(\text{state-var} := \text{new-state-var},$
 $\text{cpu-reg} := \text{new-cpu-reg})) \ (s2(\text{state-var} := \text{new-state-var2},$
 $\text{cpu-reg} := \text{new-cpu-reg2})) \ pa$

$\langle \text{proof} \rangle$

lemma *user-accessible-annul*: $\text{user-accessible } (s(\text{state-var} := \text{new-state-var},$

$cpu_reg := new_cpu_reg)) pa =$
user-accessible s pa
{proof}

lemma *mem-val-alt-10-mem-equal-0*: $mem_equal\ s1\ s2\ pa \implies$
 $mem_val_alt\ 10\ (pa\ AND\ 68719476732)\ s1 = mem_val_alt\ 10\ (pa\ AND\ 68719476732)$
 $s2$
{proof}

lemma *mem-val-alt-10-mem-equal-1*: $mem_equal\ s1\ s2\ pa \implies$
 $mem_val_alt\ 10\ ((pa\ AND\ 68719476732) + 1)\ s1 = mem_val_alt\ 10\ ((pa\ AND\ 68719476732) + 1)\ s2$
{proof}

lemma *mem-val-alt-10-mem-equal-2*: $mem_equal\ s1\ s2\ pa \implies$
 $mem_val_alt\ 10\ ((pa\ AND\ 68719476732) + 2)\ s1 = mem_val_alt\ 10\ ((pa\ AND\ 68719476732) + 2)\ s2$
{proof}

lemma *mem-val-alt-10-mem-equal-3*: $mem_equal\ s1\ s2\ pa \implies$
 $mem_val_alt\ 10\ ((pa\ AND\ 68719476732) + 3)\ s1 = mem_val_alt\ 10\ ((pa\ AND\ 68719476732) + 3)\ s2$
{proof}

lemma *mem-val-alt-10-mem-equal*:

assumes *a1*: $mem_equal\ s1\ s2\ pa$

shows $mem_val_alt\ 10\ (pa\ AND\ 68719476732)\ s1 = mem_val_alt\ 10\ (pa\ AND\ 68719476732)\ s2 \wedge$

$mem_val_alt\ 10\ ((pa\ AND\ 68719476732) + 1)\ s1 = mem_val_alt\ 10\ ((pa\ AND\ 68719476732) + 1)\ s2 \wedge$

$mem_val_alt\ 10\ ((pa\ AND\ 68719476732) + 2)\ s1 = mem_val_alt\ 10\ ((pa\ AND\ 68719476732) + 2)\ s2 \wedge$

$mem_val_alt\ 10\ ((pa\ AND\ 68719476732) + 3)\ s1 = mem_val_alt\ 10\ ((pa\ AND\ 68719476732) + 3)\ s2$

{proof}

lemma *mem-val-w32-10-mem-equal*:

assumes *a1*: $mem_equal\ s1\ s2\ a$

shows $mem_val_w32\ 10\ a\ s1 = mem_val_w32\ 10\ a\ s2$

{proof}

lemma *mem-val-alt-8-mem-equal-0*: $mem_equal\ s1\ s2\ pa \implies$

$mem_val_alt\ 8\ (pa\ AND\ 68719476732)\ s1 = mem_val_alt\ 8\ (pa\ AND\ 68719476732)$
 $s2$

{proof}

lemma *mem-val-alt-8-mem-equal-1*: $mem_equal\ s1\ s2\ pa \implies$

$mem_val_alt\ 8\ ((pa\ AND\ 68719476732) + 1)\ s1 = mem_val_alt\ 8\ ((pa\ AND\ 68719476732) + 1)\ s2$

<proof>

lemma *mem-val-alt-8-mem-equal-2*: *mem-equal s1 s2 pa* \implies
mem-val-alt 8 ((pa AND 68719476732) + 2) s1 = mem-val-alt 8 ((pa AND 68719476732)
+ 2) s2
<proof>

lemma *mem-val-alt-8-mem-equal-3*: *mem-equal s1 s2 pa* \implies
mem-val-alt 8 ((pa AND 68719476732) + 3) s1 = mem-val-alt 8 ((pa AND 68719476732)
+ 3) s2
<proof>

lemma *mem-val-alt-8-mem-equal*:
assumes *a1*: *mem-equal s1 s2 pa*
shows *mem-val-alt 8 (pa AND 68719476732) s1 = mem-val-alt 8 (pa AND 68719476732)*
s2 \wedge
mem-val-alt 8 ((pa AND 68719476732) + 1) s1 = mem-val-alt 8 ((pa AND
68719476732) + 1) s2 \wedge
mem-val-alt 8 ((pa AND 68719476732) + 2) s1 = mem-val-alt 8 ((pa AND
68719476732) + 2) s2 \wedge
mem-val-alt 8 ((pa AND 68719476732) + 3) s1 = mem-val-alt 8 ((pa AND
68719476732) + 3) s2
<proof>

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*
<proof>

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*
<proof>

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*
<proof>

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>

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>

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*
 \langle *proof* \rangle

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*
 \langle *proof* \rangle

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*
 \langle *proof* \rangle

lemma *mem-read-delayed-write-low-equal*:
assumes *a1*: *low-equal s1 s2* \wedge *get-delayed-pool s1* = \square \wedge *get-delayed-pool s2* = \square
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))
 \langle *proof* \rangle

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*
 \langle *proof* \rangle

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*
 \langle *proof* \rangle

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*

<proof>

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>

lemma *virt-to-phys-low-equal*: *low-equal s1 s2* \implies

virt-to-phys addr (mmu s1) (mem s1) = virt-to-phys addr (mmu s2) (mem s2)

<proof>

lemma *write-reg-low-equal*:

assumes *a1*: *low-equal s1 s2* \wedge

t1 = (snd (fst (write-reg w curr-win rd s1))) \wedge

t2 = (snd (fst (write-reg w curr-win rd s2)))

shows *low-equal t1 t2*

<proof>

lemma *write-cpu-low-equal*:

assumes *a1*: *low-equal s1 s2* \wedge

t1 = snd (fst (write-cpu w cr s1)) \wedge

t2 = (snd (fst (write-cpu w cr s2)))

shows *low-equal t1 t2*

<proof>

lemma *cpu-reg-mod-low-equal*:

assumes *a1*: *low-equal s1 s2* \wedge

t1 = cpu-reg-mod w cr s1 \wedge

t2 = cpu-reg-mod w cr s2

shows *low-equal t1 t2*

<proof>

lemma *load-sub2-low-equal*:

assumes *a1*: *low-equal s1 s2* \wedge

t1 = (snd (fst (load-sub2 address 10 rd curr-win w s1))) \wedge

t2 = (snd (fst (load-sub2 address 10 rd curr-win w s2)))

shows *low-equal t1 t2*

<proof>

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>

lemma *ld-asi-user*:

(fst instr = load-store-type LDSB \vee

$fst\ instr = load-store-type\ LDUB \vee$
 $fst\ instr = load-store-type\ LDUH \vee$
 $fst\ instr = load-store-type\ LD \vee$
 $fst\ instr = load-store-type\ LDD) \implies$
 $ld-asi\ instr\ 0 = 10$
 <proof>

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>

lemma *load-instr-low-equal*:
assumes $a1: low-equal\ s1\ s2 \wedge$
 $(fst\ instr = load-store-type\ LDSB \vee$
 $fst\ instr = load-store-type\ LDUB \vee$
 $fst\ instr = load-store-type\ LDUBA \vee$
 $fst\ instr = load-store-type\ LDUH \vee$
 $fst\ instr = load-store-type\ LD \vee$
 $fst\ instr = load-store-type\ LDA \vee$
 $fst\ instr = load-store-type\ LDD) \wedge$
 $((get-S\ (cpu-reg-val\ PSR\ s1))::word1) = 0 \wedge$
 $((get-S\ (cpu-reg-val\ PSR\ s2))::word1) = 0 \wedge$
 $t1 = snd\ (fst\ (load-instr\ instr\ s1)) \wedge t2 = snd\ (fst\ (load-instr\ instr\ s2))$
shows $low-equal\ t1\ t2$
 <proof>

lemma *st-data0-low-equal*: $low-equal\ s1\ s2 \implies$
 $st-data0\ instr\ curr-win\ rd\ addr\ s1 = st-data0\ instr\ curr-win\ rd\ addr\ s2$
 <proof>

lemma *store-word-mem-low-equal-none*: $low-equal\ s1\ s2 \implies$
 $store-word-mem\ (add-data-cache\ s1\ addr\ data\ bm)\ addr\ data\ bm\ 10 = None \implies$
 $store-word-mem\ (add-data-cache\ s2\ addr\ data\ bm)\ addr\ data\ bm\ 10 = None$
 <proof>

lemma *memory-write-asi-low-equal-none*: $low-equal\ s1\ s2 \implies$
 $memory-write-asi\ 10\ addr\ bm\ data\ s1 = None \implies$
 $memory-write-asi\ 10\ addr\ bm\ data\ s2 = None$
 <proof>

lemma *memory-write-low-equal-none*: $low-equal\ s1\ s2 \implies$

memory-write 10 addr bm data s1 = None \implies
memory-write 10 addr bm data s2 = None
 <proof>

lemma *memory-write-low-equal-none2: low-equal s1 s2* \implies
memory-write 10 addr bm data s2 = None \implies
memory-write 10 addr bm data s1 = None
 <proof>

lemma *mem-context-val-9-unchanged:*
mem-context-val 9 addr1 (mem s1) =
mem-context-val 9 addr1
 $((\text{mem } s1)(10 := (\text{mem } s1 \ 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None})))$
 <proof>

lemma *mem-context-val-w32-9-unchanged:*
mem-context-val-w32 9 addr1 (mem s1) =
mem-context-val-w32 9 addr1
 $((\text{mem } s1)(10 := (\text{mem } s1 \ 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None})))$
 <proof>

lemma *ptd-lookup-unchanged-4:*
ptd-lookup va ptp (mem s1) 4 =
ptd-lookup va ptp ((mem s1)(10 := (mem s1 10)(addr ↦ val),
 $11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None}))) \ 4$
 <proof>

lemma *ptd-lookup-unchanged-3:*
ptd-lookup va ptp (mem s1) 3 =
ptd-lookup va ptp ((mem s1)(10 := (mem s1 10)(addr ↦ val),
 $11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None}))) \ 3$
 <proof>

lemma *ptd-lookup-unchanged-2:*
ptd-lookup va ptp (mem s1) 2 =
ptd-lookup va ptp ((mem s1)(10 := (mem s1 10)(addr ↦ val),
 $11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None}))) \ 2$
 <proof>

lemma *ptd-lookup-unchanged-1:*
ptd-lookup va ptp (mem s1) 1 =
ptd-lookup va ptp ((mem s1)(10 := (mem s1 10) (addr ↦ val),
 $11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None}))) \ 1$
 <proof>

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 (λ lvl1-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 (λ 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 (λ 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 (λ 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>

lemma *virt-to-phys-unchanged*:

assumes a1: (\forall va. *virt-to-phys* va (mmu s2) (mem s1) = *virt-to-phys* va (mmu s2) (mem s2))

shows (\forall va. *virt-to-phys* va (mmu s2) ((mem s1)(10 := (mem s1 10)(addr \mapsto val),

$$11 := (mem s1 11)(addr := None))) = \\
virt-to-phys va (mmu s2) ((mem s2)(10 := (mem s2 10)(addr \mapsto val), \\
 11 := (mem s2 11)(addr := None))))$$

<proof>

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 \Rightarrow None | Some lvl1-page-table \Rightarrow 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 \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)
 (10 := (mem s2 10)(addr \mapsto val), 11 := (mem s2 11)(addr := None))) 1)
 <proof>

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*⟩

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* ↦ *val*),
11 := (*mem s2* 11)(*addr* := None))))

⟨*proof*⟩

lemma *mmu-low-equal*: *low-equal s1 s2* ⇒ *mmu s1* = *mmu s2*

⟨*proof*⟩

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* ↦ *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))))

⟨*proof*⟩

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* ↦ *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))))

⟨*proof*⟩

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* ↦ *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))))

⟨*proof*⟩

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 :=

$(\text{mem } s1 \ 10)(\text{addr} \mapsto \text{val}),$
 $11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None})) =$
 $\text{mem-val-alt } 8 \ ((\text{pa AND } 68719476732) + 3) \ (s2 \ (\text{mem} := (\text{mem } s2)(10 := (\text{mem}$
 $s2 \ 10)(\text{addr} \mapsto \text{val}),$
 $11 := (\text{mem } s2 \ 11)(\text{addr} := \text{None})))$
 $\langle \text{proof} \rangle$

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),
 $11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None})) =$
 $\text{mem-val-alt } 8 \ (\text{pa AND } 68719476732) \ (s2 \ (\text{mem} := (\text{mem } s2)(10 := (\text{mem } s2$
 $10)(\text{addr} \mapsto \text{val}),$

$11 := (\text{mem } s2 \ 11)(\text{addr} := \text{None})) \wedge$
 $\text{mem-val-alt } 8 \ ((\text{pa AND } 68719476732) + 1) \ (s1 \ (\text{mem} := (\text{mem } s1)(10 := (\text{mem}$
 $s1 \ 10)(\text{addr} \mapsto \text{val}),$

$11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None})) =$
 $\text{mem-val-alt } 8 \ ((\text{pa AND } 68719476732) + 1) \ (s2 \ (\text{mem} := (\text{mem } s2)(10 := (\text{mem}$
 $s2 \ 10)(\text{addr} \mapsto \text{val}),$

$11 := (\text{mem } s2 \ 11)(\text{addr} := \text{None})) \wedge$
 $\text{mem-val-alt } 8 \ ((\text{pa AND } 68719476732) + 2) \ (s1 \ (\text{mem} := (\text{mem } s1)(10 := (\text{mem}$
 $s1 \ 10)(\text{addr} \mapsto \text{val}),$

$11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None})) =$
 $\text{mem-val-alt } 8 \ ((\text{pa AND } 68719476732) + 2) \ (s2 \ (\text{mem} := (\text{mem } s2)(10 := (\text{mem}$
 $s2 \ 10)(\text{addr} \mapsto \text{val}),$

$11 := (\text{mem } s2 \ 11)(\text{addr} := \text{None})) \wedge$
 $\text{mem-val-alt } 8 \ ((\text{pa AND } 68719476732) + 3) \ (s1 \ (\text{mem} := (\text{mem } s1)(10 := (\text{mem}$
 $s1 \ 10)(\text{addr} \mapsto \text{val}),$

$11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None})) =$
 $\text{mem-val-alt } 8 \ ((\text{pa AND } 68719476732) + 3) \ (s2 \ (\text{mem} := (\text{mem } s2)(10 := (\text{mem}$
 $s2 \ 10)(\text{addr} \mapsto \text{val}),$

$11 := (\text{mem } s2 \ 11)(\text{addr} := \text{None})))$
 $\langle \text{proof} \rangle$

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))) =
 $\text{mem-val-w32 } 8 \ a \ (s2 \ (\text{mem} := (\text{mem } s2)(10 := (\text{mem } s2 \ 10)(\text{addr} \mapsto \text{val}),$

$11 := (\text{mem } s2 \ 11)(\text{addr} := \text{None})))$
 $\langle \text{proof} \rangle$

lemma *load-word-mem-8-unchanged:*

assumes *a1: low-equal s1 s2 \wedge*

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 =

$load\text{-}word\text{-}mem\ (s2\ (mem := (mem\ s2)\ (10 := (mem\ s2\ 10)\ (addr \mapsto val),$
 $11 := (mem\ s2\ 11)\ (addr := None))))\ addr\ 8$
 <proof>

lemma *load-word-mem-select-8:*

assumes *a1:* $fst\ (case\ load\text{-}word\text{-}mem\ s1\ addr\ 8\ of\ None \Rightarrow (None, s1)$
 $| Some\ w \Rightarrow (Some\ w, add\text{-}instr\text{-}cache\ s1\ addr\ w\ 15)) =$
 $fst\ (case\ load\text{-}word\text{-}mem\ s2\ addr\ 8\ of\ None \Rightarrow (None, s2)$
 $| Some\ w \Rightarrow (Some\ w, add\text{-}instr\text{-}cache\ s2\ addr\ w\ 15))$
shows $load\text{-}word\text{-}mem\ s1\ addr\ 8 = load\text{-}word\text{-}mem\ s2\ addr\ 8$
 <proof>

lemma *memory-read-8-unchanged:*

assumes *a1:* $low\text{-}equal\ s1\ s2 \wedge$
 $fst\ (memory\text{-}read\ 8\ addr\ s1) = fst\ (memory\text{-}read\ 8\ addr\ s2)$
shows $fst\ (memory\text{-}read\ 8\ addr$
 $(s1\ (mem := (mem\ s1)\ (10 := (mem\ s1\ 10)\ (addr \mapsto val),$
 $11 := (mem\ s1\ 11)\ (addr := None)))) =$
 $fst\ (memory\text{-}read\ 8\ addr$
 $(s2\ (mem := (mem\ s2)\ (10 := (mem\ s2\ 10)\ (addr \mapsto val),$
 $11 := (mem\ s2\ 11)\ (addr := None))))$
 <proof>

lemma *mem-val-alt-mod:*

assumes *a1:* $addr1 \neq addr2$
shows $mem\text{-}val\text{-}alt\ 10\ addr1\ s =$
 $mem\text{-}val\text{-}alt\ 10\ addr1\ (s\ (mem := (mem\ s)\ (10 := (mem\ s\ 10)\ (addr2 \mapsto val),$
 $11 := (mem\ s\ 11)\ (addr2 := None))))$
 <proof>

lemma *mem-val-alt-mod2:*

$mem\text{-}val\text{-}alt\ 10\ addr\ (s\ (mem := (mem\ s)\ (10 := (mem\ s\ 10)\ (addr \mapsto val),$
 $11 := (mem\ s\ 11)\ (addr := None)))) = Some\ val$
 <proof>

lemma *mem-val-alt-10-unchanged0:*

assumes *a1:* $mem\text{-}equal\ s1\ s2\ pa$
shows $mem\text{-}val\text{-}alt\ 10\ (pa\ AND\ 68719476732)\ (s1\ (mem := (mem\ s1)\ (10 := (mem$
 $s1\ 10)\ (addr \mapsto val),$
 $11 := (mem\ s1\ 11)\ (addr := None)))) =$
 $mem\text{-}val\text{-}alt\ 10\ (pa\ AND\ 68719476732)\ (s2\ (mem := (mem\ s2)\ (10 := (mem\ s2$
 $10)\ (addr \mapsto val),$
 $11 := (mem\ s2\ 11)\ (addr := None))))$
 <proof>

lemma *mem-val-alt-10-unchanged1:*

assumes *a1:* $mem\text{-}equal\ s1\ s2\ pa$
shows $mem\text{-}val\text{-}alt\ 10\ ((pa\ AND\ 68719476732) + 1)\ (s1\ (mem := (mem\ s1)\ (10$
 $:= (mem\ s1\ 10)\ (addr \mapsto val),$

$11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None})) =$
 $\text{mem-val-alt } 10 \ ((pa \ \text{AND } 68719476732) + 1) \ (s2 \ (\text{mem} := (\text{mem } s2))(10 :=$
 $(\text{mem } s2 \ 10)(\text{addr} \mapsto \text{val}),$
 $11 := (\text{mem } s2 \ 11)(\text{addr} := \text{None}))$
 $\langle \text{proof} \rangle$

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),*
 $11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None})) =$
 $\text{mem-val-alt } 10 \ ((pa \ \text{AND } 68719476732) + 2) \ (s2 \ (\text{mem} := (\text{mem } s2))(10 :=$
 $(\text{mem } s2 \ 10)(\text{addr} \mapsto \text{val}),$
 $11 := (\text{mem } s2 \ 11)(\text{addr} := \text{None}))$
 $\langle \text{proof} \rangle$

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),*
 $11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None})) =$
 $\text{mem-val-alt } 10 \ ((pa \ \text{AND } 68719476732) + 3) \ (s2 \ (\text{mem} := (\text{mem } s2))(10 :=$
 $(\text{mem } s2 \ 10)(\text{addr} \mapsto \text{val}),$
 $11 := (\text{mem } s2 \ 11)(\text{addr} := \text{None}))$
 $\langle \text{proof} \rangle$

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),*
 $11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None})) =$
 $\text{mem-val-alt } 10 \ (pa \ \text{AND } 68719476732) \ (s2 \ (\text{mem} := (\text{mem } s2))(10 := (\text{mem } s2$
 $10)(\text{addr} \mapsto \text{val}),$
 $11 := (\text{mem } s2 \ 11)(\text{addr} := \text{None})) \wedge$
 $\text{mem-val-alt } 10 \ ((pa \ \text{AND } 68719476732) + 1) \ (s1 \ (\text{mem} := (\text{mem } s1))(10 :=$
 $(\text{mem } s1 \ 10)(\text{addr} \mapsto \text{val}),$
 $11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None})) =$
 $\text{mem-val-alt } 10 \ ((pa \ \text{AND } 68719476732) + 1) \ (s2 \ (\text{mem} := (\text{mem } s2))(10 :=$
 $(\text{mem } s2 \ 10)(\text{addr} \mapsto \text{val}),$
 $11 := (\text{mem } s2 \ 11)(\text{addr} := \text{None})) \wedge$
 $\text{mem-val-alt } 10 \ ((pa \ \text{AND } 68719476732) + 2) \ (s1 \ (\text{mem} := (\text{mem } s1))(10 :=$
 $(\text{mem } s1 \ 10)(\text{addr} \mapsto \text{val}),$
 $11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None})) =$
 $\text{mem-val-alt } 10 \ ((pa \ \text{AND } 68719476732) + 2) \ (s2 \ (\text{mem} := (\text{mem } s2))(10 :=$
 $(\text{mem } s2 \ 10)(\text{addr} \mapsto \text{val}),$
 $11 := (\text{mem } s2 \ 11)(\text{addr} := \text{None})) \wedge$
 $\text{mem-val-alt } 10 \ ((pa \ \text{AND } 68719476732) + 3) \ (s1 \ (\text{mem} := (\text{mem } s1))(10 :=$
 $(\text{mem } s1 \ 10)(\text{addr} \mapsto \text{val}),$
 $11 := (\text{mem } s1 \ 11)(\text{addr} := \text{None})) =$

$mem\text{-}val\text{-}alt\ 10\ ((pa\ AND\ 68719476732) + 3)\ (s2\ (\!mem := (mem\ s2)(10 := (mem\ s2\ 10)(addr \mapsto val),$
 $11 := (mem\ s2\ 11)(addr := None)))$)
 <proof>

lemma *mem-val-w32-10-unchanged:*

assumes *a1: mem-equal s1 s2 a*

shows $mem\text{-}val\text{-}w32\ 10\ a\ (s1\ (\!mem := (mem\ s1)(10 := (mem\ s1\ 10)(addr \mapsto val),$

$11 := (mem\ s1\ 11)(addr := None))) =$
 $mem\text{-}val\text{-}w32\ 10\ a\ (s2\ (\!mem := (mem\ s2)(10 := (mem\ s2\ 10)(addr \mapsto val),$
 $11 := (mem\ s2\ 11)(addr := None)))$)
 <proof>

lemma *is-accessible: low-equal s1 s2 \implies*

virt-to-phys addr a (mmu s1) (mem s1) = Some (a, b) \implies

virt-to-phys addr a (mmu s2) (mem s2) = Some (a, b) \implies

mmu-readable (get-acc-flag b) 10 \implies

mem-equal s1 s2 a

<proof>

lemma *load-word-mem-10-unchanged:*

assumes *a1: low-equal s1 s2 \wedge*

load-word-mem s1 addr a 10 = load-word-mem s2 addr a 10

shows $load\text{-}word\text{-}mem\ (s1\ (\!mem := (mem\ s1)(10 := (mem\ s1\ 10)(addr \mapsto val),$
 $11 := (mem\ s1\ 11)(addr := None)))\ addr\ a\ 10 =$
 $load\text{-}word\text{-}mem\ (s2\ (\!mem := (mem\ s2)(10 := (mem\ s2\ 10)(addr \mapsto val),$
 $11 := (mem\ s2\ 11)(addr := None)))\ addr\ a\ 10$

<proof>

lemma *load-word-mem-select-10:*

assumes *a1: fst (case load-word-mem s1 addr a 10 of None \Rightarrow (None, s1)*

| Some w \Rightarrow (Some w, add-data-cache s1 addr w 15)) =

fst (case load-word-mem s2 addr a 10 of None \Rightarrow (None, s2)

| Some w \Rightarrow (Some w, add-data-cache s2 addr w 15))

shows $load\text{-}word\text{-}mem\ s1\ addr\ a\ 10 = load\text{-}word\text{-}mem\ s2\ addr\ a\ 10$

<proof>

lemma *memory-read-10-unchanged:*

assumes *a1: low-equal s1 s2 \wedge*

fst (memory-read 10 addr a s1) = fst (memory-read 10 addr a s2)

shows $fst\ (memory\text{-}read\ 10\ addr\ a$

$(s1\ (\!mem := (mem\ s1)(10 := (mem\ s1\ 10)(addr \mapsto val),$

$11 := (mem\ s1\ 11)(addr := None))) =$

$fst\ (memory\text{-}read\ 10\ addr\ a$

$(s2\ (\!mem := (mem\ s2)(10 := (mem\ s2\ 10)(addr \mapsto val),$

$11 := (mem\ s2\ 11)(addr := None)))$)

<proof>

lemma *state-mem-mod-1011-low-equal-sub1*:
assumes *a1*: $(\forall va. \text{virt-to-phys } va \text{ (mmu } s2) \text{ (mem } s1) = \text{virt-to-phys } va \text{ (mmu } s2) \text{ (mem } s2)) \wedge$
 $(\forall pa. (\exists va b. \text{virt-to-phys } va \text{ (mmu } s2) \text{ (mem } s2) = \text{Some } (pa, b) \wedge$
 $\text{mmu-readable } (\text{get-acc-flag } b) \text{ } 10) \longrightarrow$
 $\text{mem-equal } s1 \text{ } s2 \text{ } pa) \wedge$
 $\text{mmu } s1 = \text{mmu } s2 \wedge$
 $\text{virt-to-phys } va \text{ (mmu } s2)$
 $((\text{mem } s1)(10 := (\text{mem } s1 \text{ } 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem } s1 \text{ } 11)(\text{addr} := \text{None})))$
 $=$
 $\text{Some } (pa, b) \wedge$
 $\text{mmu-readable } (\text{get-acc-flag } b) \text{ } 10$
shows *mem-equal s1 s2 pa*
 $\langle \text{proof} \rangle$

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))))*
 $(s2 (\text{mem} := (\text{mem } s2)(10 := (\text{mem } s2 \text{ } 10)(\text{addr} \mapsto \text{val}),$
 $11 := (\text{mem } s2 \text{ } 11)(\text{addr} := \text{None}))))$
 pa
 $\langle \text{proof} \rangle$

lemma *state-mem-mod-1011-low-equal*:
assumes *a1*: *low-equal s1 s2* \wedge
 $t1 = s1 (\text{mem} := (\text{mem } s1)(10 := (\text{mem } s1 \text{ } 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem } s1 \text{ } 11)(\text{addr} := \text{None}))) \wedge$
 $t2 = s2 (\text{mem} := (\text{mem } s2)(10 := (\text{mem } s2 \text{ } 10)(\text{addr} \mapsto \text{val}), 11 := (\text{mem } s2 \text{ } 11)(\text{addr} := \text{None})))$
shows *low-equal t1 t2*
 $\langle \text{proof} \rangle$

lemma *mem-mod-low-equal*:
assumes *a1*: *low-equal s1 s2* \wedge
 $t1 = (\text{mem-mod } 10 \text{ } \text{addr } \text{val } s1) \wedge$
 $t2 = (\text{mem-mod } 10 \text{ } \text{addr } \text{val } s2)$
shows *low-equal t1 t2*
 $\langle \text{proof} \rangle$

lemma *mem-mod-w32-low-equal*:
assumes *a1*: *low-equal s1 s2* \wedge
 $t1 = \text{mem-mod-w32 } 10 \text{ } a \text{ } \text{bm } \text{data } s1 \wedge$
 $t2 = \text{mem-mod-w32 } 10 \text{ } a \text{ } \text{bm } \text{data } s2$
shows *low-equal t1 t2*
 $\langle \text{proof} \rangle$

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* \langle *proof* \rangle

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*
 \langle *proof* \rangle

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*
 \langle *proof* \rangle

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*
 \langle *proof* \rangle

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*
 \langle *proof* \rangle

lemma *store-sub2-low-equal-sub1:*
assumes *a1: low-equal s1 s2* \wedge
memory-write 10 addr (st-byte-mask instr addr)
(st-data0 instr curr-win rd addr s2) s1 = Some y \wedge
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)|))
 \langle *proof* \rangle

lemma *store-sub2-low-equal-sub2:*
assumes *a1: low-equal s1 s2* \wedge
memory-write 10 addr (st-byte-mask instr addr)
(st-data0 instr curr-win rd addr s2) s1 = Some y \wedge
memory-write 10 addr (st-byte-mask instr addr)
(st-data0 instr curr-win rd addr s2) s2 = Some ya \wedge
memory-write 10 (addr + 4) 15 (user-reg-val curr-win (rd OR 1) y) y = None \wedge
memory-write 10 (addr + 4) 15 (user-reg-val curr-win (rd OR 1) ya) ya = Some

yb
shows *False*
 ⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *store-sub1-low-equal*:
assumes *a1*: *low-equal s1 s2* ∧
(fst instr = load-store-type STB ∨
fst instr = load-store-type STH ∨
fst instr = load-store-type ST ∨
fst instr = load-store-type STD) ∧
t1 = snd (fst (store-sub1 instr rd 0 s1)) ∧
t2 = snd (fst (store-sub1 instr rd 0 s2))
shows *low-equal t1 t2*
 ⟨*proof*⟩

lemma *store-instr-low-equal*:
assumes *a1*: *low-equal s1 s2* ∧

$(fst\ instr = load-store-type\ STB \vee$
 $fst\ instr = load-store-type\ STH \vee$
 $fst\ instr = load-store-type\ ST \vee$
 $fst\ instr = load-store-type\ STA \vee$
 $fst\ instr = load-store-type\ STD) \wedge$
 $((get-S\ (cpu-reg-val\ PSR\ s1))::word1) = 0 \wedge$
 $((get-S\ (cpu-reg-val\ PSR\ s2))::word1) = 0 \wedge$
 $t1 = snd\ (fst\ (store-instr\ instr\ s1)) \wedge t2 = snd\ (fst\ (store-instr\ instr\ s2))$
shows $low-equal\ t1\ t2$
 $\langle proof \rangle$

lemma *sethi-low-equal*: $low-equal\ s1\ s2 \wedge$
 $t1 = snd\ (fst\ (sethi-instr\ instr\ s1)) \wedge t2 = snd\ (fst\ (sethi-instr\ instr\ s2)) \implies$
 $low-equal\ t1\ t2$
 $\langle proof \rangle$

lemma *nop-low-equal*: $low-equal\ s1\ s2 \wedge$
 $t1 = snd\ (fst\ (nop-instr\ instr\ s1)) \wedge t2 = snd\ (fst\ (nop-instr\ instr\ s2)) \implies$
 $low-equal\ t1\ t2$
 $\langle proof \rangle$

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 $low-equal\ t1\ t2$
 $\langle proof \rangle$

lemma *logical-instr-low-equal*: $low-equal\ s1\ s2 \wedge$
 $t1 = snd\ (fst\ (logical-instr\ instr\ s1)) \wedge t2 = snd\ (fst\ (logical-instr\ instr\ s2)) \implies$
 $low-equal\ t1\ t2$
 $\langle proof \rangle$

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 $low-equal\ t1\ t2$
 $\langle proof \rangle$

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 $low-equal\ t1\ t2$
 $\langle proof \rangle$

lemma *add-instr-low-equal*:
assumes $a1: low-equal\ s1\ s2 \wedge$
 $t1 = snd\ (fst\ (add-instr\ instr\ s1)) \wedge t2 = snd\ (fst\ (add-instr\ instr\ s2))$

shows *low-equal t1 t2*
<proof>

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>

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>

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>

lemma *mul-instr-low-equal*:

<low-equal t1 t2>

if *<low-equal s1 s2* \wedge *t1 = snd (fst (mul-instr instr s1))* \wedge *t2 = snd (fst (mul-instr instr s2))>*

<proof>

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>

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*

<proof>

lemma *div-instr-low-equal*:

assumes *a1: low-equal s1 s2* \wedge

t1 = snd (fst (div-instr instr s1)) \wedge *t2 = snd (fst (div-instr instr s2))*

shows *low-equal t1 t2*

<proof>

lemma *get-curr-win-traps-low-equal*:

assumes *a1*: *low-equal s1 s2*

shows *low-equal*

$(\text{snd } (\text{fst } (\text{get-curr-win } ()) \text{ s1}))$
 $\langle \text{traps} := \text{insert some-trap } (\text{traps } (\text{snd } (\text{fst } (\text{get-curr-win } ()) \text{ s1}))) \rangle$
 $(\text{snd } (\text{fst } (\text{get-curr-win } ()) \text{ s2}))$
 $\langle \text{traps} := \text{insert some-trap } (\text{traps } (\text{snd } (\text{fst } (\text{get-curr-win } ()) \text{ s2}))) \rangle$
<proof>

lemma *save-restore-instr-sub1-low-equal*:

assumes *a1*: *low-equal s1 s2* \wedge

t1 = $\text{snd } (\text{fst } (\text{save-restore-sub1 result new-cwp rd } \text{s1}))$ \wedge

t2 = $\text{snd } (\text{fst } (\text{save-restore-sub1 result new-cwp rd } \text{s2}))$

shows *low-equal t1 t2*

<proof>

lemma *get-WIM-bit-low-equal*:

$\langle \text{get-WIM-bit } (\text{nat } ((\text{uint } (\text{fst } (\text{fst } (\text{get-curr-win } ()) \text{ s1}))) - 1) \text{ mod } \text{NWINDOVS}))$
 $(\text{cpu-reg-val WIM } (\text{snd } (\text{fst } (\text{get-curr-win } ()) \text{ s1})))) =$
 $\text{get-WIM-bit } (\text{nat } ((\text{uint } (\text{fst } (\text{fst } (\text{get-curr-win } ()) \text{ s2}))) - 1) \text{ mod } \text{NWINDOVS}))$
 $(\text{cpu-reg-val WIM } (\text{snd } (\text{fst } (\text{get-curr-win } ()) \text{ s2})))) \rangle$
if *<low-equal s1 s2>*
<proof>

lemma *get-WIM-bit-low-equal2*:

$\langle \text{get-WIM-bit } (\text{nat } ((\text{uint } (\text{fst } (\text{fst } (\text{get-curr-win } ()) \text{ s1}))) + 1) \text{ mod } \text{NWINDOVS}))$
 $(\text{cpu-reg-val WIM } (\text{snd } (\text{fst } (\text{get-curr-win } ()) \text{ s1})))) =$
 $\text{get-WIM-bit } (\text{nat } ((\text{uint } (\text{fst } (\text{fst } (\text{get-curr-win } ()) \text{ s2}))) + 1) \text{ mod } \text{NWINDOVS}))$
 $(\text{cpu-reg-val WIM } (\text{snd } (\text{fst } (\text{get-curr-win } ()) \text{ s2})))) \rangle$
if *<low-equal s1 s2>*
<proof>

lemma *take-bit-5-mod-NWINDOVS-eq [simp]*:

$\langle \text{take-bit } 5 \text{ } (k \text{ mod } \text{NWINDOVS}) = k \text{ mod } \text{NWINDOVS} \rangle$
<proof>

lemma *save-restore-instr-low-equal*:

assumes *a1*: *low-equal s1 s2* \wedge

t1 = $\text{snd } (\text{fst } (\text{save-restore-instr instr } \text{s1}))$ \wedge *t2* = $\text{snd } (\text{fst } (\text{save-restore-instr instr } \text{s2}))$

shows *low-equal t1 t2*

<proof>

lemma *call-instr-low-equal*:

assumes *a1*: *low-equal s1 s2* \wedge

t1 = $\text{snd } (\text{fst } (\text{call-instr instr } \text{s1}))$ \wedge *t2* = $\text{snd } (\text{fst } (\text{call-instr instr } \text{s2}))$

shows *low-equal t1 t2*

<proof>

lemma *jmpl-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>

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>

lemma *jmpl-instr-low-equal*:

assumes *a1*: *low-equal s1 s2* \wedge

t1 = *snd* (*fst* (*jmpl-instr instr s1*)) \wedge *t2* = *snd* (*fst* (*jmpl-instr instr s2*))

shows *low-equal t1 t2*
{proof}

lemma *rett-instr-low-equal*:
assumes *a1: low-equal s1 s2* \wedge
 $\neg \text{snd (rett-instr instr s1)}$ \wedge
 $\neg \text{snd (rett-instr instr s2)}$ \wedge
 $((\text{get-S (cpu-reg-val PSR s1)})::\text{word1}) = 0$ \wedge
 $((\text{get-S (cpu-reg-val PSR s2)})::\text{word1}) = 0$ \wedge
 $t1 = \text{snd (fst (rett-instr instr s1))}$ \wedge $t2 = \text{snd (fst (rett-instr instr s2))}$
shows *low-equal t1 t2*
{proof}

lemma *read-state-reg-low-equal*:
assumes *a1: low-equal s1 s2* \wedge
 $((\text{get-S (cpu-reg-val PSR s1)})::\text{word1}) = 0$ \wedge
 $((\text{get-S (cpu-reg-val PSR s2)})::\text{word1}) = 0$ \wedge
 $t1 = \text{snd (fst (read-state-reg-instr instr s1))}$ \wedge
 $t2 = \text{snd (fst (read-state-reg-instr instr s2))}$
shows *low-equal t1 t2*
{proof}

lemma *get-s-get-curr-win*:
assumes *a1: low-equal s1 s2*
shows $\text{get-S (cpu-reg-val PSR (snd (fst (get-curr-win ()) s1)))} =$
 $\text{get-S (cpu-reg-val PSR (snd (fst (get-curr-win ()) s2)))}$
{proof}

lemma *write-state-reg-low-equal*:
assumes *a1: low-equal s1 s2* \wedge
 $((\text{get-S (cpu-reg-val PSR s1)})::\text{word1}) = 0$ \wedge
 $((\text{get-S (cpu-reg-val PSR s2)})::\text{word1}) = 0$ \wedge
 $t1 = \text{snd (fst (write-state-reg-instr instr s1))}$ \wedge
 $t2 = \text{snd (fst (write-state-reg-instr instr s2))}$
shows *low-equal t1 t2*
{proof}

lemma *flush-instr-low-equal*:
assumes *a1: low-equal s1 s2* \wedge
 $t1 = \text{snd (fst (flush-instr instr s1))}$ \wedge
 $t2 = \text{snd (fst (flush-instr instr s2))}$
shows *low-equal t1 t2*
{proof}

lemma *branch-instr-sub1-low-equal*:
assumes *a1: low-equal s1 s2*
shows $\text{branch-instr-sub1 instr-name s1} = \text{branch-instr-sub1 instr-name s2}$
{proof}

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*
<proof>

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*)) \ll 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*)) \ll 2))
nPC (*snd* (*fst* (*write-cpu* (*cpu-reg-val* *nPC s2*) *PC s2*))))))
shows *low-equal t1 t2*
<proof>

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*)) \ll 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*)) \ll 2))
nPC (*snd* (*fst* (*write-cpu* (*cpu-reg-val* *nPC s2*) *PC s2*))))))))))
shows *low-equal t1 t2*
<proof>

lemma *branch-instr-low-equal-sub2*:
assumes *a1*: *low-equal s1 s2* \wedge
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*)))))))))) \wedge
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>

lemma *branch-instr-low-equal*:
assumes *a1*: *low-equal s1 s2* \wedge
t1 = *snd* (*fst* (*branch-instr* *instr s1*)) \wedge
t2 = *snd* (*fst* (*branch-instr* *instr s2*))
shows *low-equal t1 t2*
<proof>

lemma *dispath-instr-low-equal*:
assumes $a1$: *low-equal* $s1$ $s2$ \wedge
 $((\text{get-}S \text{ (cpu-reg-val PSR } s1))::\text{word1}) = 0 \wedge$
 $((\text{get-}S \text{ (cpu-reg-val PSR } s2))::\text{word1}) = 0 \wedge$
 $\neg \text{snd (dispatch-instruction instr } s1) \wedge$
 $\neg \text{snd (dispatch-instruction instr } s2) \wedge$
 $t1 = (\text{snd (fst (dispatch-instruction instr } s1))) \wedge$
 $t2 = (\text{snd (fst (dispatch-instruction instr } s2)))$
shows *low-equal* $t1$ $t2$
 $\langle \text{proof} \rangle$

lemma *execute-instr-sub1-low-equal*:
assumes $a1$: *low-equal* $s1$ $s2$ \wedge
 $\neg \text{snd (execute-instr-sub1 instr } s1) \wedge$
 $\neg \text{snd (execute-instr-sub1 instr } s2) \wedge$
 $t1 = (\text{snd (fst (execute-instr-sub1 instr } s1))) \wedge$
 $t2 = (\text{snd (fst (execute-instr-sub1 instr } s2)))$
shows *low-equal* $t1$ $t2$
 $\langle \text{proof} \rangle$

theorem *non-interference-step*:
assumes $a1$: $((\text{get-}S \text{ (cpu-reg-val PSR } s1))::\text{word1}) = 0 \wedge$
good-context $s1 \wedge$
 $\text{get-delayed-pool } s1 = [] \wedge \text{get-trap-set } s1 = \{\}$ \wedge
 $((\text{get-}S \text{ (cpu-reg-val PSR } s2))::\text{word1}) = 0 \wedge$
 $\text{get-delayed-pool } s2 = [] \wedge \text{get-trap-set } s2 = \{\}$ \wedge
good-context $s2 \wedge$
low-equal $s1$ $s2$
shows $\exists t1 t2. \text{Some } t1 = \text{NEXT } s1 \wedge \text{Some } t2 = \text{NEXT } s2 \wedge$
 $((\text{get-}S \text{ (cpu-reg-val PSR } t1))::\text{word1}) = 0 \wedge$
 $((\text{get-}S \text{ (cpu-reg-val PSR } t2))::\text{word1}) = 0 \wedge$
low-equal $t1$ $t2$
 $\langle \text{proof} \rangle$

function (*sequential*) *SEQ*:: $\text{nat} \Rightarrow ('a::\text{len}) \text{sparc-state} \Rightarrow ('a) \text{sparc-state option}$
where $\text{SEQ } 0 \text{ } s = \text{Some } s$
 $|\text{SEQ } n \text{ } s = ($
 $\text{case } \text{SEQ } (n-1) \text{ } s \text{ of None} \Rightarrow \text{None}$
 $|\text{Some } t \Rightarrow \text{NEXT } t$
 $)$
 $\langle \text{proof} \rangle$
termination $\langle \text{proof} \rangle$

lemma *SEQ-suc*: $\text{SEQ } n \text{ } s = \text{Some } t \implies \text{SEQ } (\text{Suc } n) \text{ } s = \text{NEXT } t$
 $\langle \text{proof} \rangle$

definition *user-seq-exe*:: $\text{nat} \Rightarrow ('a::\text{len}) \text{sparc-state} \Rightarrow \text{bool}$ **where**
user-seq-exe $n \text{ } s \equiv \forall i t. (i \leq n \wedge \text{SEQ } i \text{ } s = \text{Some } t) \longrightarrow$
 $(\text{good-context } t \wedge \text{get-delayed-pool } t = [] \wedge \text{get-trap-set } t = \{\})$

NIA is short for non-interference assumption.

definition $NIA\ t1\ t2 \equiv$

$$\begin{aligned} & (((get-S\ (cpu-reg-val\ PSR\ t1)))::word1) = 0 \wedge \\ & (((get-S\ (cpu-reg-val\ PSR\ t2)))::word1) = 0 \wedge \\ & good-context\ t1 \wedge get-delayed-pool\ t1 = [] \wedge get-trap-set\ t1 = \{\} \wedge \\ & good-context\ t2 \wedge get-delayed-pool\ t2 = [] \wedge get-trap-set\ t2 = \{\} \wedge \\ & low-equal\ t1\ t2 \end{aligned}$$

NIC is short for non-interference conclusion.

definition $NIC\ t1\ t2 \equiv (\exists\ u1\ u2.\ Some\ u1 = NEXT\ t1 \wedge Some\ u2 = NEXT\ t2$

$$\begin{aligned} & \wedge \\ & (((get-S\ (cpu-reg-val\ PSR\ u1)))::word1) = 0 \wedge \\ & (((get-S\ (cpu-reg-val\ PSR\ u2)))::word1) = 0 \wedge \\ & low-equal\ u1\ u2) \end{aligned}$$

lemma $NIS-short: \forall\ t1\ t2.\ NIA\ t1\ t2 \longrightarrow NIC\ t1\ t2$

<proof>

lemma $non-interference-induct-case-sub1:$

assumes $a1: (\exists\ t1.\ Some\ t1 = SEQ\ n\ s1 \wedge$

$$(\exists\ t2.\ Some\ t2 = SEQ\ n\ s2 \wedge$$

$$NIA\ t1\ t2))$$

shows $(\exists\ t1.\ Some\ t1 = SEQ\ n\ s1 \wedge$

$$(\exists\ t2.\ Some\ t2 = SEQ\ n\ s2 \wedge$$

$$NIA\ t1\ t2 \wedge$$

$$NIC\ t1\ t2))$$

<proof>

lemma $non-interference-induct-case:$

assumes $a1:$

$$((\forall\ i\ t.\ i \leq n \wedge SEQ\ i\ s1 = Some\ t \longrightarrow$$

$$good-context\ t \wedge get-delayed-pool\ t = [] \wedge get-trap-set\ t = \{\}) \wedge$$

$$(\forall\ i\ t.\ i \leq n \wedge SEQ\ i\ s2 = Some\ t \longrightarrow$$

$$good-context\ t \wedge get-delayed-pool\ t = [] \wedge get-trap-set\ t = \{\}) \longrightarrow$$

$$(\exists\ t1.\ Some\ t1 = SEQ\ n\ s1 \wedge$$

$$(\exists\ t2.\ Some\ t2 = SEQ\ n\ s2 \wedge$$

$$(((get-S\ (cpu-reg-val\ PSR\ t1)))::word1) = 0 \wedge$$

$$(((get-S\ (cpu-reg-val\ PSR\ t2)))::word1) = 0 \wedge low-equal\ t1\ t2))) \wedge$$

$$(\forall\ i\ t.\ i \leq Suc\ n \wedge SEQ\ i\ s1 = Some\ t \longrightarrow$$

$$good-context\ t \wedge get-delayed-pool\ t = [] \wedge get-trap-set\ t = \{\}) \wedge$$

$$(\forall\ i\ t.\ i \leq Suc\ n \wedge SEQ\ i\ s2 = Some\ t \longrightarrow$$

$$good-context\ t \wedge get-delayed-pool\ t = [] \wedge get-trap-set\ t = \{\})$$

shows $\exists\ t1.\ Some\ t1 = (case\ SEQ\ n\ s1\ of\ None \Rightarrow None \mid Some\ x \Rightarrow NEXT\ x) \wedge$

$$(\exists\ t2.\ Some\ t2 = (case\ SEQ\ n\ s2\ of\ None \Rightarrow None \mid Some\ x \Rightarrow NEXT$$

$x) \wedge$

$$(((get-S\ (cpu-reg-val\ PSR\ t1)))::word1) = 0 \wedge$$

$$(((get-S\ (cpu-reg-val\ PSR\ t2)))::word1) = 0 \wedge low-equal\ t1\ t2)$$

<proof>

lemma *non-interference-induct-case-sub2*:

assumes *a1*:

$(user\text{-}seq\text{-}exe\ n\ s1 \wedge$

$user\text{-}seq\text{-}exe\ n\ s2 \longrightarrow$

$(\exists t1. Some\ t1 = SEQ\ n\ s1 \wedge$

$(\exists t2. Some\ t2 = SEQ\ n\ s2 \wedge$

$((get\text{-}S\ (cpu\text{-}reg\text{-}val\ PSR\ t1))::word1) = 0 \wedge$

$((get\text{-}S\ (cpu\text{-}reg\text{-}val\ PSR\ t2))::word1) = 0 \wedge low\text{-}equal\ t1\ t2))) \wedge$

$user\text{-}seq\text{-}exe\ (Suc\ n)\ s1 \wedge$

$user\text{-}seq\text{-}exe\ (Suc\ n)\ s2$

shows $\exists t1. Some\ t1 = (case\ SEQ\ n\ s1\ of\ None \Rightarrow None \mid Some\ x \Rightarrow NEXT\ x) \wedge$

$(\exists t2. Some\ t2 = (case\ SEQ\ n\ s2\ of\ None \Rightarrow None \mid Some\ x \Rightarrow NEXT$

$x) \wedge$

$((get\text{-}S\ (cpu\text{-}reg\text{-}val\ PSR\ t1))::word1) = 0 \wedge$

$((get\text{-}S\ (cpu\text{-}reg\text{-}val\ PSR\ t2))::word1) = 0 \wedge low\text{-}equal\ t1\ t2)$

$\langle proof \rangle$

theorem *non-interference*:

assumes *a1*:

$((get\text{-}S\ (cpu\text{-}reg\text{-}val\ PSR\ s1))::word1) = 0 \wedge$

$good\text{-}context\ s1 \wedge$

$get\text{-}delayed\text{-}pool\ s1 = [] \wedge get\text{-}trap\text{-}set\ s1 = \{\} \wedge$

$((get\text{-}S\ (cpu\text{-}reg\text{-}val\ PSR\ s2))::word1) = 0 \wedge$

$get\text{-}delayed\text{-}pool\ s2 = [] \wedge get\text{-}trap\text{-}set\ s2 = \{\} \wedge$

$good\text{-}context\ s2 \wedge$

$user\text{-}seq\text{-}exe\ n\ s1 \wedge user\text{-}seq\text{-}exe\ n\ s2 \wedge$

$low\text{-}equal\ s1\ s2$

shows $(\exists t1\ t2. Some\ t1 = SEQ\ n\ s1 \wedge Some\ t2 = SEQ\ n\ s2 \wedge$

$((get\text{-}S\ (cpu\text{-}reg\text{-}val\ PSR\ t1))::word1) = 0 \wedge$

$((get\text{-}S\ (cpu\text{-}reg\text{-}val\ PSR\ t2))::word1) = 0 \wedge$

$low\text{-}equal\ t1\ t2)$

$\langle proof \rangle$

end

end