The Semantics of Power and ARM Multiprocessor Machine Code

The HOL Specification

Jade Alglave[2] Anthony Fox[1] Samin Ishtiaq[3] Magnus O. Myreen[1]
Susmit Sarkar[1] Peter Sewell[1] Francesco Zappa Nardelli[2]

[1]University of Cambridge [2]INRIA [3]Microsoft Research Cambridge

October 14, 2008

Brief Contents

Full Contents

Part I

common_coretypes

type_abbrev proc : num

type_abbrev program_order_index : num

iiid =⟨[proc : proc; *poi* : program_order_index]⟩

Part II

common_types

type_abbrev Ximm : word32

type_abbrev address : Ximm

type_abbrev value : Ximm

type_abbrev eiid : num

type_abbrev reln : $'a\#'a \rightarrow$ bool

dirn = R | W

location = LOCATION_REG **of** proc $'reg$
| LOCATION_MEM **of** address
| LOCATION_RES **of** proc
| LOCATION_RES_ADDR **of** proc

synchronization = SYNC

$action$ = ACCESS **of** dirn $'reg$ location value
| BARRIER **of** synchronization

event =⟪ eiid : eiid;
$iiid$: $iiid$;
$action$: $'reg$ $action$⟫

architecture = POWER205 | ARMV7

event_structure =⟪ $events$: ($'reg$ event)set;
$intra\_causality\_data$: ($'reg$ event)reln;
$intra\_causality\_control$: ($'reg$ event)reln;
$atomicity$: ($'reg$ event)set set;
$arch$: architecture;
$granule\_size\_exponent$: num⟫

type_abbrev state_constraint : ($'reg$ location) $\rightarrow$ value option

type_abbrev view_orders : proc $\rightarrow$ ($'reg$ event)reln

execution_witness =
⟪ $initial\_state$: ($'reg$ state_constraint);
vo : ($'reg$ view_orders);
$write\_serialization$: ($'reg$ event reln)⟫

type_abbrev eiid_state : eiid set

(next_eiid : eiid_state → (eiid#eiid_state)set)$eiids = \{(\mathsf{eiid}, eiids \cup \{\mathsf{eiid}\}) \mid \mathsf{eiid} \mid \neg(\mathsf{eiid} \in eiids)\}$

(initial_eiid_state : eiid_state) $= \{\}$

type_abbrev program_word8 : (address → word8 option)

type_abbrev run_skeleton : (proc → (program_order_index → address option))

(run_skeleton_wf : address set → run_skeleton → bool)$addrs\ rs =$
$(\forall p\ i\ i'.((\neg((rs\ p\ i') = \text{NONE})) \wedge (i < i')) \implies (\neg((rs\ p\ i) = \text{NONE}))) \wedge$
$(\forall p\ i\ a.(rs\ p\ i = \text{SOME}\ \ a) \implies a \in addrs) \wedge$
finite$\{p \mid \exists i\ a.rs\ p\ i = \text{SOME}\ \ a\}$

Part III

ppc_arm_axiomatic_model

```
loc e =
case e.action of
  ACCESS d l v → SOME l
| _ → NONE

value_of e =
case e.action of
  ACCESS d l v → SOME v
| _ → NONE

proc e = e.iiid.proc

mem_load e =
case e.action of
  ACCESS R(LOCATION_MEM a)v → T
| _ → F

mem_store e =
case e.action of
  ACCESS W(LOCATION_MEM a)v → T
| _ → F

mem_access e =
case e.action of
  ACCESS d(LOCATION_MEM a)v → T
| _ → F

reg_load e =
case e.action of
  ACCESS R(LOCATION_REG p r)v → T
| _ → F

reg_store e =
case e.action of
  ACCESS W(LOCATION_REG p r)v → T
| _ → F

reg_access e =
case e.action of
  ACCESS d(LOCATION_REG p r)v → T
| _ → F
```

```
reg_or_mem_location l =
case l of
     (LOCATION_REG p r) → T
 || (LOCATION_MEM a) → T
 || (LOCATION_RES p) → F
 || (LOCATION_RES_ADDR p) → F

reg_or_mem_or_resaddr l =
case l of
     (LOCATION_REG p r) → T
 || (LOCATION_MEM a) → T
 || (LOCATION_RES p) → F
 || (LOCATION_RES_ADDR p) → T

store e =
case e.action of
   ACCESS W l v → T
 || _ → F

load e =
case e.action of
   ACCESS R l v → T
 || _ → F

is_barrier e =
case e.action of
   BARRIER bar → T
 || _ → F

is_sync e =
case e.action of
   BARRIER SYNC → T
 || _ → F

procs E = {proc e | e ∈ E.events}

iiids E = {e.iiid | e ∈ E.events}

writes E = {e | e ∈ E.events ∧ ∃l v.e.action = ACCESS W l v}

reads E = {e | e ∈ E.events ∧ ∃l v.e.action = ACCESS R l v}
```

$$\begin{array}{l}
\text{po } E = \\
\{(e_1, e_2) \mid (e_1.iiid.\,\text{proc} = e_2.iiid.\,\text{proc}) \wedge \\
\qquad e_1.iiid.poi \leq e_2.iiid.poi \wedge \\
\qquad e_1 \in E.events \wedge e_2 \in E.events\}
\end{array}$$

$$\begin{array}{l}
\text{po_strict } E = \\
\{(e_1, e_2) \mid (e_1.iiid.\,\text{proc} = e_2.iiid.\,\text{proc}) \wedge \\
\qquad e_1.iiid.poi < e_2.iiid.poi \wedge \\
\qquad e_1 \in E.events \wedge e_2 \in E.events\}
\end{array}$$

$$\begin{array}{l}
\text{intra_causality } E = E.intra_causality_data \cup \\
\qquad\qquad\qquad E.intra_causality_control
\end{array}$$

$$\begin{array}{l}
\text{po_iico_both } E = \text{po_strict } E \cup E.intra_causality_data \cup \\
\qquad\qquad\qquad E.intra_causality_control
\end{array}$$

$$\text{po\_iico\_data } E = \text{po\_strict } E \cup E.intra\_causality\_data$$

$$\begin{array}{l}
\text{well_formed_event_structure } E = \\
(\forall e_1\ e_2 \in (E.events).(e_1.\mathsf{eiid} = e_2.\mathsf{eiid}) \wedge (e_1.iiid = e_2.iiid) \\
\qquad \Longrightarrow (e_1 = e_2)) \wedge \\
(DOM(\text{intra_causality } E)) \subseteq E.events \wedge \\
(\mathbf{range}(\text{intra_causality } E)) \subseteq E.events \wedge \\
\text{acyclic}(\text{intra_causality } E) \wedge \\
(\forall (e_1, e_2) \in (\text{intra_causality } E).(e_1.iiid = e_2.iiid)) \wedge \\
(\forall (e_1 \in \text{writes } E) e_2. \\
\neg(e_1 = e_2) \wedge \\
(e_2 \in \text{writes } E \vee e_2 \in \text{reads } E) \wedge \\
(e_1.iiid = e_2.iiid) \wedge \\
(\text{loc } e_1 = \text{loc } e_2) \wedge \\
(\exists p\ r.\,\text{loc } e_1 = \textsc{Some } (\textsc{Location_reg } p\ r)) \\
\Longrightarrow \\
(e_1, e_2) \in (\text{intra_causality } E)^+ \vee \\
(e_2, e_1) \in (\text{intra_causality } E)^+) \wedge \\
PER\ E.events\ E.atomicity \wedge \\
(\forall es \in (E.atomicity).\forall e_1\ e_2 \in es.(e_1.iiid = e_2.iiid))
\end{array}$$

$$\begin{array}{l}
\text{local_register_data_dependency } E\ p = \\
\{(e_1, e_2) \mid \\
\quad (e_1, e_2) \in \text{po_iico_data } E \wedge \\
\quad (\text{proc } e_1 = p) \wedge (\text{proc } e_2 = p) \wedge \\
\quad (\exists r\ v_1\ v_2. \\
\qquad (e_1.action = \textsc{Access } \text{W}(\textsc{Location_reg } p\ r) v_1) \wedge \\
\qquad (e_2.action = \textsc{Access } \text{R}(\textsc{Location_reg } p\ r) v_2) \wedge \\
\qquad (\neg(\exists e_3\ v_3.(e_1, e_3) \in \text{po_iico_data } E \wedge \\
\qquad\quad (e_3, e_2) \in \text{po_iico_data } E \wedge \\
\qquad\quad (e_3.action = \textsc{Access } \text{W}(\textsc{Location_reg } p\ r) v_3))))\}
\end{array}$$

address_or_data_dependency_load_load $E\ p =$
$\{(e_1, e_2) \mid$
$\quad (\text{mem\_load}\ e_1 \wedge \text{mem\_load}\ e_2 \wedge (e_1, e_2) \in \text{po}\ E \wedge$
$\quad (\text{proc}\ e_1 = p) \wedge (\text{proc}\ e_2 = p) \wedge$
$\quad (e_1, e_2) \in ((E.\mathit{intra\_causality\_data} \cup$
$\qquad \text{local\_register\_data\_dependency}\ E\ p)^+))\}$

address_or_data_or_control_dependency_load_store $E\ p =$
$\{(e_1, e_2) \mid$
$\quad (\text{mem\_load}\ e_1 \wedge \text{mem\_store}\ e_2 \wedge (e_1, e_2) \in \text{po}\ E \wedge$
$\quad (\text{proc}\ e_1 = p) \wedge (\text{proc}\ e_2 = p) \wedge$
$\quad (e_1, e_2) \in ((E.\mathit{intra\_causality\_data} \cup$
$\qquad E.\mathit{intra\_causality\_control} \cup$
$\qquad \text{local\_register\_data\_dependency}\ E\ p)^+))\}$

preserved_program_order_mem_loc $E\ p =$
$\{(e_1, e_2) \mid (e_1, e_2) \in \text{po\_iico\_data}\ E \wedge$
$\quad (\text{proc}\ e_1 = p) \wedge (\text{proc}\ e_2 = p) \wedge (\text{loc}\ e_1 = \text{loc}\ e_2) \wedge$
$\quad \text{mem\_access}\ e_1 \wedge \text{mem\_access}\ e_2\}$

preserved_program_order $E\ p =$
$\{(e_1, e_2) \mid (e_1, e_2) \in E.\mathit{intra\_causality\_data} \wedge$
$\qquad\qquad (\text{proc}\ e_1 = p) \wedge (\text{proc}\ e_2 = p)\} \cup$
$\{(e_1, e_2) \mid (e_1, e_2) \in E.\mathit{intra\_causality\_control} \wedge$
$\qquad\qquad (\text{proc}\ e_1 = p) \wedge (\text{proc}\ e_2 = p)\} \cup$
$\text{address\_or\_data\_dependency\_load\_load}\ E\ p \cup$
$\text{address\_or\_data\_or\_control\_dependency\_load\_store}\ E\ p \cup$
$\text{preserved\_program\_order\_mem\_loc}\ E\ p$

preserved_coherence_order $E\ p =$
$\{(e_1, e_2) \mid (e_1, e_2) \in \text{po\_iico\_data}\ E \wedge$
$\quad (\text{proc}\ e_1 = p) \wedge (\text{proc}\ e_2 = p) \wedge (\text{loc}\ e_1 = \text{loc}\ e_2) \wedge$
$\quad (\text{mem\_store}\ e_1 \wedge \text{mem\_store}\ e_2)\}$

viewed_events $E\ p =$
$\{e \mid e \in E.\mathit{events} \wedge ((\text{proc}\ e = p) \vee \text{mem\_store}\ e)\}$

view_orders_well_formed $E\ \mathit{vo} =$
$\forall p \in (\text{procs}\ E).$
$\quad \text{strict\_linear\_order}(\mathit{vo}\ p)(\text{viewed\_events}\ E\ p)$

get_mem_l_stores $E\ l =$
$\{e \mid e \in E.\mathit{events} \wedge \text{mem\_store}\ e \wedge (\text{loc}\ e = \textsc{Some}\ l)\}$

write_serialization_candidates E *cand* =
$(\forall(e_1, e_2) \in$ *cand*.
  $\exists l.e_1 \in$ (get_mem_l_stores E l) $\wedge$
    $e_2 \in$ (get_mem_l_stores E l)) $\wedge$
$(\forall l.$ strict_linear_order(*cand*$|_{\text{(get\_mem\_l\_stores } E\ l)}$)
                (get_mem_l_stores E l))

state_updates E *vo* e =
$\{ew \mid ew \in$ (writes E) $\wedge$ $(ew, e) \in$ *vo*(proc e) $\wedge$
    (loc ew = loc e)$\}$

read_most_recent_value E *initial_state* *vo* =
$\forall e \in (E.events).\forall l\ v.$
  (($e.action$ = Access R l v) $\wedge$ reg_or_mem_or_resaddr l)
    $\Longrightarrow$
  (**if** (state_updates E *vo* e) = $\{\}$ **then**
    (Some v = *initial_state* l)
  else
    ((Some v) $\in$ {value_of ew |
      $ew \in$ maximal_elements(state_updates E *vo* e)
                (*vo*(proc e))}))

FIX f x = **biginter**$\{X \mid (f\ X \cup x) \subseteq X\}$

check_sync_power_2_05 E *vos* =
  $\forall es \in (E.events).(es.action$ = Barrier Sync) $\Longrightarrow$
  let *group_A* = $\{e \mid ((e, es) \in$ po $E \vee (e, es) \in$ *vos*(proc es)) $\wedge$ mem_access $e\}$ **in**
  let *group_B_base* = $\{e \mid (es, e) \in$ po $E \wedge$ mem_access $e\}$ **in**
  let *group_B_ind* B_0 =
    $\{e \mid$ mem_access $e \wedge$
      $(\neg($proc e = proc $es)) \wedge$
      $\exists er.$ mem_load $er \wedge (er, e) \in$ *vos*(proc er) $\wedge$ (proc er = proc e) $\wedge$
        $\exists ew.$ mem_store $ew \wedge ew \in B_0 \wedge (ew, er) \in$ *vos*(proc er) $\wedge$ (loc er = loc ew) $\wedge$
          $(\neg(\exists ew'.(ew, ew') \in$ *vos*(proc er) $\wedge$ $(ew', er) \in$ *vos*(proc er) $\wedge$
            (loc ew' = loc er) $\wedge$ mem_store ew'))$\}$ **in**
  let *group_B* = FIX *group_B_ind* *group_B_base* **in**
  $\forall p \in$ (procs E).$\forall ea \in$ *group_A*.$\forall eb \in$ *group_B*.($ea \in$ viewed_events E $p \wedge eb \in$ viewed_events E p) $\Longrightarrow$
    if (p = $es.iiid.$proc) **then** $((ea, es) \in$ *vos* $p \wedge (es, eb) \in$ *vos* p) **else** $(ea, eb) \in$ *vos* p

check_dmb_arm E *vos* =
  $\forall es \in (E.events).(es.action$ = Barrier Sync) $\Longrightarrow$
  let *group_A_base* = $\{e \mid ((e, es) \in$ *vos*(proc es)) $\wedge$ mem_access $e\}$ **in**
  let *group_A_ind* A_0 = $\{er \mid$ mem_load $er \wedge$
            $\exists e \in A_0.(er, e) \in$ *vos*(proc e)$\}$ **in**
  let *group_B_base* = $\{e \mid (es, e) \in$ po $E \wedge$ mem_access $e\}$ **in**
  let *group_B_ind* B_0 = $\{e \mid$ mem_access $e \wedge$

```
                    ∃ew. mem_store ew ∧ ew ∈ B0 ∧ (ew, e) ∈ vos(proc e)} in
  let group_A = FIX group_A_ind group_A_base in
  let group_B = FIX group_B_ind group_B_base in
  ∀p ∈ (procs E).∀ea ∈ group_A.∀eb ∈ group_B.(ea ∈ viewed_events E p ∧ eb ∈ viewed_events E p) ⟹
    if (p = es.iiid. proc) then ((ea, es) ∈ vos p ∧ (es, eb) ∈ vos p) else (ea, eb) ∈ vos p


same_granule E a a' =
let f x = x//(1w ≪ E.granule_size_exponent) in (f a = f a')


location_res_value E vo e =
let prior_reservations = {ew | ew ∈ (writes E) ∧
  (ew, e) ∈ vo(proc e) ∧
  (loc ew = SOME (LOCATION_RES_ADDR(proc e)))} in
if (prior_reservations = {}) then 0w else
  let reservation = maximal_elements
    prior_reservations(vo(proc e)) in
  let intervening_writes = {ew |
    ∃ew' ∈ reservation.∃a'.∃a v.
      (ew.action = ACCESS W(LOCATION_MEM a)v) ∧
      (ew', ew) ∈ vo(proc e) ∧ (ew, e) ∈ vo(proc e) ∧
      (value_of ew' = SOME a') ∧
       same_granule E a a'} in
  if intervening_writes = {} then 0w else 1w


read_location_res_value E initial_state vo =
∀e ∈ (E.events).∀p v.
  (e.action = ACCESS R(LOCATION_RES p)v) ⟹
  (v = location_res_value E vo e)


check_atomicity E vo =
  ∀p ∈ (procs E).∀es ∈ (E.atomicity).
    ∀e1 e2 ∈ es.(e1, e2) ∈ (vo p) ⟹
      ∀e.(e1, e) ∈ (vo p) ∧ (e, e2) ∈ (vo p) ⟹ e ∈ es


valid_execution E X =
   view_orders_well_formed E X.vo ∧
   read_most_recent_value E X.initial_state X.vo ∧
   X.write_serialization ∈ write_serialization_candidates E ∧
   (∀p ∈ (procs E).
      preserved_coherence_order E p ⊆ X.write_serialization ∧
      X.write_serialization ⊆ X.vo p ∧
      preserved_program_order E p ⊆ X.vo p ∧
       (*no intervening writes in local register data dependency*)
      local_register_data_dependency E p ⊆ X.vo p ∧
      (∀(e1, e2) ∈ (local_register_data_dependency E p).
        ¬(∃e3.(e1, e3) ∈ X.vo p ∧ (e3, e2) ∈ X.vo p ∧
```

$$(\text{loc}\ e_3 = \text{loc}\ e_1) \wedge \text{store}\ e_3))) \wedge$$
(*no intervening writes before a reg read from initial state*)
$(\forall e \in (E.events).(\text{reg\_load}\ e \wedge$
$(\neg(\exists e_0.(e_0, e) \in \text{po\_iico\_both}\ E \wedge \text{reg\_store}\ e_0 \wedge$
$(\text{loc}\ e_0 = \text{loc}\ e)))) \implies$
$(\neg(\exists e_0.(e_0, e) \in X.vo(\text{proc}\ e) \wedge \text{reg\_store}\ e_0 \wedge$
$(\text{loc}\ e_0 = \text{loc}\ e)))) \wedge$
(*no intervening writes after a reg write to the final state*)
$(\forall e \in (E.events).(\text{reg\_store}\ e \wedge$
$(\neg(\exists e_1.(e, e_1) \in \text{po\_iico\_both}\ E \wedge \text{reg\_store}\ e_1 \wedge$
$(\text{loc}\ e_1 = \text{loc}\ e)))) \implies$
$(\neg(\exists e_1.(e, e_1) \in X.vo(\text{proc}\ e) \wedge \text{reg\_store}\ e_1 \wedge$
$(\text{loc}\ e_1 = \text{loc}\ e)))) \wedge$
(**case** $E.arch$ **of**
POWER205 $\rightarrow$ check_sync_power_2_05 $E\ X.vo$
$\|$ ARMv7 $\rightarrow$ check_dmb_arm $E\ X.vo$) $\wedge$
read_location_res_value $E\ X.initial\_state\ X.vo \wedge$
check_atomicity $E\ X.vo$

Part IV

arm_coretypes

type_abbrev Aimm : word32

ARMreg =
R0 | R1 | R2 | R3 | R4 | R5 | R6 | R7 |
R8 | R9 | R10 | R11 | R12 | R13 | R14 | R15 |
R8_FIQ | R9_FIQ | R10_FIQ | R11_FIQ | R12_FIQ | R13_FIQ | R14_FIQ |
R13_IRQ | R14_IRQ |
R13_SVC | R14_SVC |
R13_ABT | R14_ABT |
R13_UND | R14_UND

ARMpsr =
CPSR | SPSR_FIQ | SPSR_IRQ | SPSR_SVC | SPSR_ABT | SPSR_UND

ARMmode = USR | FIQ | IRQ | SVC | ABT | UND | SYS

ARMstatus =
⟪ N : bool; Z : bool; C : bool; V : bool;
Q : bool; IT : word8; J : bool; $Reserved$: word4; GE : word4;
E : bool; A : bool;
$\mathbf{I}$: bool; $\mathbf{F}$: bool; $\mathbf{T}$: bool; M : word5⟫

ARMsctlr =
⟪ TE : bool; AFE : bool; TRE : bool; $NMFI$: bool; EE : bool;
VE : bool; U : bool; FI : bool; HA : bool; RR : bool;
V : bool; $\mathbf{I}$: bool; Z : bool; SW : bool; B : bool;
C : bool; A : bool; M : bool⟫

ARMcp_registers =⟪ $SCTLR$: ARMsctlr⟫

ARMcondition =
EQ | CS | MI | VS | HI | GE | GT | AL |
NE | CC | PL | VC | LS | LT | LE | NV

ARMexception =
Exception_Reset |
Exception_Undefined |
Exception_Supervisor |
Exception_PrefetchAbort |
Exception_DataAbort |
Exception_Address |
Exception_Interrupt |
Exception_FIQ

ARMversion =
ARMv4 | ARMv4T |
ARMv5T | ARMv5TE |
ARMv6 | ARMv6K | ARMv6T2 |
ARMv7_A | ARMv7_R | ARMv7_M

ARMextensions =
Extension_ThumbEE | Extension_VFP | Extension_AdvanvedSIMD |
Extension_Security | Extension_Jazelle | Extension_Multiprocessing

ARMinfo =
⟪ *version* : ARMversion;
extensions : ARMextensions set⟫

InstrSet =
InstrSet_ARM | InstrSet_Thumb | InstrSet_Jazelle | InstrSet_ThumbEE

MemType =
MemType_Normal | MemType_Device | MemType_StringlyOrdered

MemoryAttributes =
⟪ *type* : MemType;
innerattrs : word2;
outerattrs : word2;
shareable : bool;
outershareable : bool⟫

FullAddress =
⟪ *physicaladdress* : word32;
physicaladdressext : word8;
NS : bool(* F = Secure; T = Non-secure *)⟫

AddressDescriptor =
⟪ *memattrs* : MemoryAttributes;
paddress : FullAddress⟫

type_abbrev ExclusiveMonitor : FullAddress#*iiid*#num → bool

ExclusiveMonitors =
⟪ *TranslateAddress* :
word32#bool#bool → AddressDescriptor;
(* TranslateAddress(VA,ispriv,iswrite) converts a virtual address to an address descriptor. Implementation depends on the memory architecture. *)
MarkExclusiveGlobal :

```
  (FullAddress#iiid#num) → ExclusiveMonitor → ExclusiveMonitor;
  MarkExclusiveLocal :
  (FullAddress#iiid#num) → ExclusiveMonitor → ExclusiveMonitor;
  ClearExclusiveLocal : iiid → ExclusiveMonitor#ExclusiveMonitor →
                               ExclusiveMonitor#ExclusiveMonitor;
  IsExclusiveLocal : ExclusiveMonitor;
  IsExclusiveGlobal : ExclusiveMonitor⟫

MBReqDomain =
MBReqDomain_FullSystem |
MBReqDomain_OuterShareable |
MBReqDomain_InnerShareable |
MBReqDomain_Nonshareable

MBReqTypes = MBReqTypes_All | MBReqTypes_Writes

word5_to_mode(m : word5) =
case m of
   16w → Some usr
 || 17w → Some fiq
 || 18w → Some irq
 || 19w → Some svc
 || 23w → Some abt
 || 27w → Some und
 || 31w → Some sys
 || _ → None

align(w : 'a word, n : num) : 'a word =
n2w(n * (w2n w div n))

decode_psr(psr : word32) =
⟪ N := psr[31];
  Z := psr[30];
  C := psr[29];
  V := psr[28];
  Q := psr[27];
  IT :=((15 >< 10)psr : word6)@@((26 >< 25)psr : word2);
  J := psr[24];
  Reserved :=(23 >< 20)psr;
  GE :=(19 >< 16)psr;
  E := psr[9];
  A := psr[8];
  I := psr[7];
  F := psr[6];
  T := psr[5];
  M :=(4 >< 0)psr⟫
```

encode_psr(psr : ARMstatus) : word32 =
word_modify($\lambda x\ b$.
if $x < 5$ **then** psr.M[x] **else**
if $x = 5$ **then** psr.**T** **else**
if $x = 6$ **then** psr.**F** **else**
if $x = 7$ **then** psr.**I** **else**
if $x = 8$ **then** $psr.A$ **else**
if $x = 9$ **then** $psr.E$ **else**
if $x < 16$ **then** $psr.IT[(x - 8)]$ **else**
if $x < 20$ **then** $psr.GE[(x - 16)]$ **else**
if $x < 24$ **then** $psr.Reserved[(x - 20)]$ **else**
if $x = 24$ **then** $psr.J$ **else**
if $x < 27$ **then** $psr.IT[(x - 25)]$ **else**
if $x = 27$ **then** $psr.Q$ **else**
if $x = 28$ **then** $psr.V$ **else**
if $x = 29$ **then** $psr.C$ **else**
if $x = 30$ **then** $psr.Z$ **else**
(* x = 31 *)$psr.N$)0w

(version_number ARMv4 = 4) ∧
(version_number ARMv4T = 4) ∧
(version_number ARMv5T = 5) ∧
(version_number ARMv5TE = 5) ∧
(version_number ARMv6 = 6) ∧
(version_number ARMv6K = 6) ∧
(version_number ARMv6T2 = 6) ∧
(version_number ARMv7_A = 7) ∧
(version_number ARMv7_R = 7) ∧
(version_number ARMv7_M = 7)

Part V

arm_types

arm_reg = REG32 **of** ARMreg | REGPSR **of** ARMpsr

Part VI

arm_ast

```
addressing_mode1 =
MODE1_IMMEDIATE of word4 word8
 | MODE1_SHIFT_IMMEDIATE of word5 word2 word4
 | MODE1_SHIFT_REGISTER of word4 word2 word4

addressing_mode2 =
MODE2_IMMEDIATE of word12
 | MODE2_SHIFT_IMMEDIATE of word5 word2 word4

addressing_mode3 =
MODE3_IMMEDIATE of word8
 | MODE3_REGISTER of word4

ARMinstruction =
UNPREDICTABLE
 | UNDEFINED
 | BRANCH of bool word24
 | BRANCH_EXCHANGE of word4
 | BRANCH_LINK_EXCHANGE1 of bool word24
 | BRANCH_LINK_EXCHANGE2 of word4
 | COUNT_LEADING_ZEROES of word4 word4
 | BREAKPOINT of word16
 | SUPERVISOR_CALL of word24
 | MULTIPLY of bool bool word4 word4 word4 word4
 | MULTIPLY_LONG of bool bool bool word4 word4 word4 word4
 | STATUS_TO_REGISTER of bool word4
 | REGISTER_TO_STATUS of bool word4 word4
 | IMMEDIATE_TO_STATUS of bool word4 word4 word8
 | DATA_PROCESSING of word4 bool word4 word4 addressing_mode1
 | SWAP of bool word4 word4 word4
 | DATA_MEMORY_BARRIER of word4
 | LOAD_STORE_MULTIPLE of bool bool bool bool bool word4 word16
 | LOAD_EXCLUSIVE of word4 word4 word8
 | STORE_EXCLUSIVE of word4 word4 word4 word8
 | LOAD_STORE of bool bool bool bool bool word4 word4
                 addressing_mode2
 | LOAD_STORE_HALFWORD of bool bool bool bool word4 word4 bool
                 addressing_mode3
 | LOAD_STORE_TWO_WORDS of bool bool bool word4 word4 bool
                 addressing_mode3
 | CP_LOAD_STORE of bool bool bool bool bool word4 word4
                 word4 word8
 | CP_LOAD_STORE2 of bool bool bool bool bool word4 word4
                 word4 word8
 | CP_DATA_PROCESSING of word4 word4 word4 word4 word3 word4
 | CP_DATA_PROCESSING2 of word4 word4 word4 word4 word3 word4
 | CP_TRANSFER of word3 bool word4 word4 word4 word3 word4
 | CP_TRANSFER2 of word3 bool word4 word4 word4 word3 word4
```

| DSP_ADD_SUBTRACT **of** word2 word4 word4 word4
| DSP_MULTIPLY **of** word4 word4 word4 bool word4

Part VII

arm_opsem

```
user_or_system_mode m = (m = USR) ∨ (m = SYS)

word4_mode_to_reg(w : word4, m) =
num2ARMreg(let n = w2n w in
(if (n = 15) ∨ user_or_system_mode m ∨
   (m = FIQ) ∧ n < 8 ∨
   ¬(m = FIQ) ∧ n < 13
 then
   n
 else case m of
   FIQ → n + 8
 || IRQ → n + 10
 || SVC → n + 12
 || ABT → n + 14
 || UND → n + 16
 || _ → ARB))

read_regm ii x = read_reg ii(word4_mode_to_reg x)

write_regm ii x = write_reg ii(word4_mode_to_reg x)

mode_to_psr mode =
case mode of
   USR → CPSR
 || FIQ → SPSR_FIQ
 || IRQ → SPSR_IRQ
 || SVC → SPSR_SVC
 || ABT → SPSR_ABT
 || UND → SPSR_UND
 || SYS → CPSR

exception_to_mode type =
case type of
   Exception_Reset → SVC
 || Exception_Undefined → UND
 || Exception_Supervisor → SVC
 || Exception_Address → SVC
 || Exception_Prefetch_Abort → ABT
 || Exception_Data_Abort → ABT
 || Exception_Interrupt → IRQ
 || Exception_Fast → FIQ

mode_to_word5 mode : word5 =
case mode of
   USR → 16w
 || FIQ → 17w
```

$\|$ IRQ $\rightarrow 18w$
$\|$ SVC $\rightarrow 19w$
$\|$ ABT $\rightarrow 23w$
$\|$ UND $\rightarrow 27w$
$\|$ SYS $\rightarrow 31w$

exception_to_address *high_vectors type* : word32 =
(**if** *high_vectors* **then** $0xFFFF0000w$ **else** $0w$) +
n2w$(4 * ARMexception2num\ type)$

condition_passed(N, Z, C, V)*cond* =
case *cond* **of**
  EQ $\rightarrow Z$
$\|$ NE $\rightarrow \neg Z$
$\|$ CS $\rightarrow C$
$\|$ CC $\rightarrow \neg C$
$\|$ MI $\rightarrow N$
$\|$ PL $\rightarrow \neg N$
$\|$ VS $\rightarrow V$
$\|$ VC $\rightarrow \neg V$
$\|$ HI $\rightarrow C \wedge \neg Z$
$\|$ LS $\rightarrow \neg C \vee Z$
$\|$ *GE* $\rightarrow N = V$
$\|$ LT $\rightarrow \neg(N = V)$
$\|$ GT $\rightarrow \neg Z \wedge (N = V)$
$\|$ LE $\rightarrow Z \vee \neg(N = V)$
$\|$ AL $\rightarrow$ **T**
$\|$ NV $\rightarrow$ **F**

inc_pc *ii* = read_reg *ii* R15 >>= $(\lambda x.$ write_reg *ii* R15$(x + 4w))$

branch_write_pc *ii*$(addr$: word32$)$ =
(read_version *ii*!! read_instr_set *ii*) >>=
$(\lambda(version, iset)$.
  if *iset* = InstrSet_ARM **then**
    if *version* $< 6 \wedge ((1 >< 0)addr = 0w$: word2$)$
    then
      failureT
    else
      write_reg *ii* R15$((31 \neq 2)addr)$
  else
    write_reg *ii* R15$((31 \neq 1)addr))$

LSL$(m$: word32$)(n$: word8$)c$ =
if $n = 0w$ **then** (c, m) **else**
$(n <= +32w \wedge m[(32 -$ **w2n** $n)], m \ll$ **w2n** $n)$

```
LSR(m : word32)(n : word8)c =
if n = 0w then LSL m 0w c else
(n <= +32w ∧ m[(w2n n − 1)], m >>> w2n n)


ASR(m : word32)(n : word8)c =
if n = 0w then LSL m 0w c else
(m[(min 31(w2n n − 1))], m ≫ w2n n)


ROR(m : word32)(n : word8)c =
if n = 0w then LSL m 0w c else
(m[(w2n((w2w n) : word5) − 1)], m# >> w2n n)


immediate ii(rot : word4)(imm : word8) =
(inc_pc ii!! read_flags ii) >>=
(λ(unit, (n, z, c, v)). constT(ROR(w2w imm)(2w ∗ w2w rot)c))


shift_immediate ii(shift : word5)(sh : word2)rm =
(read_flags ii!! read_mode ii) >>=
(λ((n, z, c, v), m).
   read_regm ii(rm, m) >>=
   (λrm. inc_pc ii >> −
          constT
            case sh of
              0w → LSL rm(w2w shift)c
            || 1w → LSR rm(if shift = 0w then 32w else w2w shift)c
            || 2w → ASR rm(if shift = 0w then 32w else w2w shift)c
            || _ → if shift = 0w then
                       word_rrx(c, rm)
                   else
                     ROR rm(w2w shift)c))


shift_register ii rs(sh : word2)rm =
(read_flags ii!! read_mode ii) >>=
(λ((n, z, c, v), m).
   read_regm ii(rs, m) >>=
   (λrs. inc_pc ii >> −
          read_regm ii(rm, m) >>=
          (λrm. constT
            case sh of
                0w → LSL rm(w2w rs)c
            || 1w → LSR rm(w2w rs)c
            || 2w → ASR rm(w2w rs)c
            || _ → ROR rm(w2w rs)c)))
```

```
(addr_mode1 ii(MODE1_IMMEDIATE rot imm) =
 immediate ii rot imm) ∧
(addr_mode1 ii(MODE1_SHIFT_IMMEDIATE shift sh rm) =
 shift_immediate ii shift sh rm) ∧
(addr_mode1 ii(MODE1_SHIFT_REGISTER rs sh rm) =
 shift_register ii rs sh rm)


(addr_mode2 ii u w rn(MODE2_IMMEDIATE offset) =
 read_mode ii >>=
 (λm.
    read_regm ii(rn, m) >>=
    (λrn. inc_pc ii >> −
          let address = if u then rn + w2w offset else rn − w2w offset in
            constT(if w then address else rn, address)))) ∧
(addr_mode2 ii u w rn(MODE2_SHIFT_IMMEDIATE shift sh rm) =
 read_mode ii >>=
 (λm.
    read_regm ii(rn, m) >>=
    (λrn. shift_immediate ii shift sh rm >>=
        (λ(c_shift, offset).
          let address = if u then rn + offset else rn − offset in
            constT(if w then address else rn, address)))))


ALU_arith op(rn : word32)(op2 : word32) =
let sign = word_msb rn
 and (q, r) = DIVMOD_2EXP 32(op(w2n rn)(w2n op2)) in
let res = (n2w r) : word32 in
((word_msb res, r = 0, ODD q,
   (word_msb op2 = sign) ∧ ¬(word_msb res = sign)), res)


ALU_logic(res : word32) = ((word_msb res, res = 0w, F, F), res)


ALU_multiply a rn rs rm =
let res : word32 = rm ∗ rs + (if a then rn else 0w) in
(word_msb res, res = 0w, res)


ALU_multiply_long s a(rdhi : word32)(rdlo : word32)(rs : word32)(rm : word32) =
let res : word64 =
       (if a then rdhi@@rdlo else 0w : word64) +
       (if s then sw2sw rm ∗ sw2sw rs else w2w rm ∗ w2w rs)
 in
let reshi = (63 >< 32)res : word32
 and reslo = (31 >< 0)res : word32
 in
(word_msb res, res = 0w, reshi, reslo)
```

```
ADD a b c = ALU_arith(λx y.x + y + (if c then 1 else 0))a b

SUB a b c = ADD a(¬b)c

AND a b = ALU_logic(a&&b)

EOR a b = ALU_logic(a??b)

ORR a b = ALU_logic(a!!b)

ALU(opc : word4)rn op2 c =
case opc of
  0w → AND rn op2
| 1w → EOR rn op2
| 2w → SUB rn op2 T
| 4w → ADD rn op2 F
| 3w → SUB op2 rn T
| 5w → ADD rn op2 c
| 6w → SUB rn op2 c
| 7w → SUB op2 rn c
| 8w → AND rn op2
| 9w → EOR rn op2
| 10w → SUB rn op2 T
| 11w → ADD rn op2 F
| 12w → ORR rn op2
| 13w → ALU_logic op2
| 14w → AND rn(¬op2)
| _ → ALU_logic(¬op2)

arithmetic(opcode : word4) =
(opcode[2] ∨ opcode[1]) ∧ (¬(opcode[3]) ∨ ¬(opcode[2]))

test_or_compare(opcode : word4) = ((3 − −2)opcode = 2w)

rotate_mem32(oareg : word2)(w : word32) = w# >> (8 ∗ w2n oareg)

exception_exec ii type =
 discardT
((read_reg ii R15!! read_sctlr ii!! read_psr ii CPSR) >>=
  (λ(pc, sctlr, cpsr).
    (let mode = exception_to_mode type in
         write_regm ii(14w, mode)(pc − 4w)!!
         branch_write_pc ii(exception_to_address sctlr.V type)!!
         write_psr ii(mode_to_psr mode)cpsr!!
```

```
          write_psr ii CPSR(cpsr
            ⟪ I := T;
              F :=(type ∈ {EXCEPTION_RESET; EXCEPTION_FAST} ∨ cpsr.F);
              A :=((type = EXCEPTION_RESET) ∨ cpsr.A); IT := 0w; J := F;
              T := sctlr.TE; E := sctlr.EE;
              M := mode_to_word5 mode⟫)))))

branch_exec ii(BRANCH l offset) =
read_reg ii R15 >>=
(λpc.
  let target = pc + sw2sw offset ≪ 2 in
    if ¬(word_msb target = word_msb(pc − 8w)) then
      failureT
    else
      if l then
        discardT(read_mode ii >>=
        (λm.
          write_regm ii(14w, m)(pc − 4w)!!
          branch_write_pc ii target))
      else
        branch_write_pc ii target)

data_processing_exec ii(DATA_PROCESSING opcode s rn rd op2) =
(addr_mode1 ii op2!! read_flags ii!! read_mode ii) >>=
(λ((c_shift, opnd2), (n, z, c, v), m).
  read_regm ii(rn, m) >>=
  (λrn.
    let ((n_alu, z_alu, c_alu, v_alu), res) = ALU opcode rn opnd2 c_shift in
      discardT
        (condT(¬ test_or_compare opcode)
          (write_regm ii(rd, m)res)!!
        condT s
          (if (rd = 15w) ∧ ¬(test_or_compare opcode) then
            read_psr ii(mode_to_psr m) >>=
            (λspsr. write_psr ii CPSR spsr)
          else
            write_flags ii
              (if arithmetic opcode then
                (n_alu, z_alu, c_alu, v_alu)
              else
                (n_alu, z_alu, c_shift, v))))))

multiply_exec ii(MULTIPLY a s rd rn rs rm) =
if (rd = 15w) ∨ (rm = 15w) ∨ (rs = 15w) ∨ (rd = rm) then
failureT
else
discardT((read_flags ii!! read_mode ii) >>=
(λ((n, z, c, v), m).
```

```
      (read_regm ii(rn, m)!!
       read_regm ii(rs, m)!!
       read_regm ii(rm, m)) >>=
      (λ(rn, rs, rm).
          let (n_alu, z_alu, res) = ALU_multiply a rn rs rm in
             inc_pc ii!!
             write_regm ii(rd, m)res!!
             condT s(read_version ii >>=
             (λversion.
               let c_flag = if version = 4 then ARB else c in
                  write_flags ii(n_alu, z_alu, c_flag, v))))))


multiply_long_exec ii(MULTIPLY_LONG u a s rdhi rdlo rs rm) =
if (rdhi = 15w) ∨ (rdlo = 15w) ∨ (rm = 15w) ∨ (rs = 15w) ∨
   (rdhi = rm) ∨ (rdlo = rm) ∨ (rdhi = rdlo) then
 failureT
 else
 discardT((read_flags ii!! read_mode ii) >>=
(λ((n, z, c, v), m).
      (read_regm ii(rdhi, m)!!
       read_regm ii(rdlo, m)!!
       read_regm ii(rs, m)!!
       read_regm ii(rm, m)) >>=
      (λ(rdhi′, rdlo′, rs, rm).
          let (n_alu, z_alu, reshi, reslo) =
                 ALU_multiply_long u a rdhi′ rdlo′ rs rm
           in
             inc_pc ii!!
             write_regm ii(rdhi, m)reshi!!
             write_regm ii(rdlo, m)reslo!!
             condT s(read_version ii >>=
             (λversion.
               let c_flag = if version = 4 then ARB else c in
                  write_flags ii(n_alu, z_alu, c_flag, v))))))


load_store_exec ii(LOAD_STORE p u b w l rn rd op2) =
let wb = p ⟹ w in
if wb ∧ (rn = rd) then
   failureT
 else
   discardT((addr_mode2 ii u w rn op2!! read_mode ii) >>=
   (λ((address, wb_address), m).
     if l then
        condT wb(write_regm ii(rn, m)wb_address)!!
       (if b then
             read_mem8 ii address >>= (λd. constT(w2w d))
          else
             read_mem32 ii address >>=
```

```
            (λd. constT(rotate_mem32(w2w address)d))) >>=
            (λdata. write_regm ii(rd, m)data)
      else
        read_regm ii(rd, m) >>=
        (λrd.
            if b then
              write_mem8 ii address(w2w rd)
            else
              write_mem32 ii address rd)!!
        condT wb(write_regm ii(rn, m)wb_address)))


load_exclusive_exec ii(LOAD_EXCLUSIVE rn rt imm8) =
read_version ii >>=
(λversion.
  if version < 6 then
    exception_exec ii EXCEPTION_UNDEFINED
  else
    if (rt = 15w) ∨ (rn = 15w) then
      failureT
    else
      read_mode ii >>=
      (λm.
        read_regm ii(rn, m) >>=
        (λrn.let address = rn + w2w imm8 in
            set_exclusive_monitorsT ii(address, 4) >> −
            read_mem32 ii address >>=
            (λd. write_regm ii(rt, m)d))))


store_exclusive_exec ii(STORE_EXCLUSIVE rn rd rt imm8) =
read_version ii >>=
(λversion.
  if version < 6 then
    exception_exec ii EXCEPTION_UNDEFINED
  else
    if (rd = 15w) ∨ (rt = 15w) ∨ (rn = 15w) ∨
      (rd = rt) ∨ (rd = rt)
    then
      failureT
    else
      read_mode ii >>=
      (λm.
        read_regm ii(rn, m) >>=
        (λrn.let address = rn + w2w imm8 in
            (exclusive_monitors_passT ii(address, 4) >>=
            (λpass.
              if pass then
                read_regm ii(rt, m) >>=
                (λrt.
```

```
                discardT
                  (write_mem32 ii address rt!!
                     write_regm ii(rd, m)0w))
            else
              write_regm ii(rd, m)1w)))))

swap_exec ii(SWAP b rn rd rm) =
if (rd = 15w) ∨ (rn = 15w) ∨ (rm = 15w) ∨ (rn = rm) ∨ (rn = rd) then
failureT
 else
lockT(read_mode ii >>=
(λm.
    (read_regm ii(rn, m)!! read_regm ii(rm, m)) >>=
    (λ(address, rm).
      (if b then
          read_mem8 ii address >>= (λd. constT(w2w d))
        else
          read_mem32 ii address >>=
          (λd. constT(rotate_mem32(w2w address)d))) >>=
          (λdata. discardT
            (inc_pc ii!!
               write_mem32 ii address rm!!
               write_regm ii(rd, m)data)))))

status_to_register_exec ii(STATUS_TO_REGISTER r rd) =
read_psr ii CPSR >>=
(λcpsr.
  let mode = word5_to_mode cpsr.M in let m = the mode in
    if (rd = 15w) ∨ IS_NONE mode ∨ user_or_system_mode m ∧ r then
      failureT
     else
      (if r then
         read_psr ii(mode_to_psr m) >>= (λspsr. constT(encode_psr spsr))
       else
         constT(encode_psr cpsr&&
                  0b11111000_11111111_00000011_11011111w)) >>=
      (λpsr. discardT(inc_pc ii!! write_regm ii(rd, m)psr)))

breakpoint_exec ii cond =
if ¬(cond = AL) then
failureT
 else
read_version ii >>=
(λversion.
    if version = 4 then
      exception_exec ii EXCEPTION_UNDEFINED
     else
      exception_exec ii EXCEPTION_PREFETCH_ABORT)
```

```
data_memory_barrier_exec ii(DATA_MEMORY_BARRIER option) =
read_version ii >>=
(λversion.
  if version < 7 then
    exception_exec ii EXCEPTION_UNDEFINED
  else
    discardT(inc_pc ii!! dmbT ii
      (case option of
          0b0010w → (MBREQDOMAIN_OUTERSHAREABLE, MBREQTYPES_WRITES)
        || 0b0011w → (MBREQDOMAIN_OUTERSHAREABLE, MBREQTYPES_ALL)
        || 0b0110w → (MBREQDOMAIN_NONSHAREABLE, MBREQTYPES_WRITES)
        || 0b0111w → (MBREQDOMAIN_NONSHAREABLE, MBREQTYPES_ALL)
        || 0b1010w → (MBREQDOMAIN_INNERSHAREABLE, MBREQTYPES_WRITES)
        || 0b1011w → (MBREQDOMAIN_INNERSHAREABLE, MBREQTYPES_ALL)
        || 0b1110w → (MBREQDOMAIN_FULLSYSTEM, MBREQTYPES_WRITES)
        || _ → (MBREQDOMAIN_FULLSYSTEM, MBREQTYPES_ALL))))

arm_execute ii(cond, inst) =
read_flags ii >>=
(λflags.
  if condition_passed flags cond then
    case inst of
      BRANCH l offset →
        branch_exec ii inst
    || SWAP b rn rd rm →
        swap_exec ii inst
    || DATA_PROCESSING opcode s rn rd op2 →
        data_processing_exec ii inst
    || LOAD_STORE p u b w l rn rd op2_1 →
        load_store_exec ii inst
    || LOAD_EXCLUSIVE rn rt imm8 →
        load_exclusive_exec ii inst
    || STORE_EXCLUSIVE rn rd rt imm8 →
        store_exclusive_exec ii inst
    || MULTIPLY_LONG u a s rdhi rdlo rs rm →
        multiply_long_exec ii inst
    || MULTIPLY a s rd rn rs rm →
        multiply_exec ii inst
    || STATUS_TO_REGISTER r rd →
        status_to_register_exec ii inst
    || BREAKPOINT number16 →
        breakpoint_exec ii cond
    || DATA_MEMORY_BARRIER option →
        data_memory_barrier_exec ii inst
    || SUPERVISOR_CALL number24 →
        exception_exec ii EXCEPTION_SUPERVISOR
    || UNDEFINED →
        exception_exec ii EXCEPTION_UNDEFINED
    || _ →
```

```
        failureT
  else
    inc_pc ii)
```

Part VIII

arm_seq_monad

type_abbrev arm_state : (ARMreg → word32)#(* - general-purpose registers *)
(ARMpsr → ARMstatus)#(* - program-status registers *)
ARMcp_registers#(* - co-processor registers *)
(word32 → word8 option)#(* - unsegmented memory *)
ARMinfo#(* - info on ISA version and extensions *)
ExclusiveMonitors(* - for synchronization & semaphores *)

AREAD_REG xˆarm_state $= r\ x$

AREAD_PSR xˆarm_state $= p\ x$

AREAD_CP ˆarm_state $= c$

AREAD_MEM xˆarm_state $= m\ x$

AREAD_INFO ˆarm_state $= v$

AREAD_EXCL ˆarm_state $= e$

AWRITE_REG $i\ x$ˆarm_state $= ((i = +x)r, p, c, m, v, e)$: arm_state

AWRITE_PSR $i\ x$ˆarm_state $= (r, (i = +x)p, c, m, v, e)$: arm_state

AWRITE_MEM $i\ x$ˆarm_state $= (r, p, c, (i = +x)m, v, e)$: arm_state

AWRITE_EXCL xˆarm_state $= (r, p, c, m, v, x)$: arm_state

type_abbrev M : arm_state → ($'a$#arm_state) option

(constT_seq : $'a \to 'a$ M)$x = \lambda y.$SOME (x, y)

(failureT_seq : $'a$ M) $= \lambda y.$NONE

(bindT_seq : $'a$ M → ($'a \to 'b$ M) → $'b$ M)$s\ f =$
$\lambda y.$**case** $s\ y$ **of** NONE → NONE ‖ SOME $(z, t) \to f\ z\ t$

(seqT_seq : $'a$ M → $'b$ M → $'b$ M)$s\ f =$
bindT_seq $s(\lambda x.f)$

(parT_seq : $'a$ M → $'b$ M → ($'a$#$'b$)M)$s\ t =$
bindT_seq $s(\lambda x.$ bindT_seq $t(\lambda y.$ constT_seq$(x, y)))$

```
(lockT_seq : 'a M → 'a M)s = s

(condT_seq : bool → unit M → unit M)b s =
if b then s else constT_seq()

(discardT_seq : 'a M → unit M)s =
seqT_seq s(constT_seq())

(addT_seq : 'a → 'b M → ('a#'b)M)x s =
bindT_seq s(λz. constT_seq(x, z))

(write_reg_seq ii r x) : unit M =
λs.SOME ((), AWRITE_REG r x s)

(read_reg_seq ii r) : Aimm M =
λs.SOME (let x = AREAD_REG r s in if r = R15 then x + 8w else x, s)

(write_psr_seq ii r x) : unit M =
λs.case word5_to_mode x.M of
      NONE → NONE
   || SOME m → SOME ((), AWRITE_PSR r x s)

(read_psr_seq ii r) : ARMstatus M =
λs.SOME (AREAD_PSR r s, s)

(write_flags_seq ii(n, z, c, v)) : unit M =
λs.SOME ((),
  let cpsr = AREAD_PSR CPSR s ⟦ N := n; Z := z; C := c; V := v⟧ in
    AWRITE_PSR CPSR cpsr s)

(read_flags_seq ii) : (bool#bool#bool#bool)M =
λs.SOME (let cpsr = AREAD_PSR CPSR s in (cpsr.N, cpsr.Z, cpsr.C, cpsr.V), s)

(read_endian_seq ii) : bool M =
λs.SOME ((AREAD_PSR CPSR s).E, s)

(read_mode_seq ii) : ARMmode M =
λs.case word5_to_mode(AREAD_PSR CPSR s).M of
     NONE → NONE
|| SOME m → SOME (m, s)
```

```
(read_instr_set_seq ii) : InstrSet M =
λs.SOME (let cpsr = AREAD_PSR CPSR s in
  (case (cpsr.J, cpsr.T) of
      (F, F) → INSTRSET_ARM
    ‖ (F, T) → INSTRSET_THUMB
    ‖ (T, F) → INSTRSET_JAZELLE
    ‖ (T, T) → INSTRSET_THUMBEE), s)


(read_sctlr_seq ii) : ARMsctlr M =
λs.SOME ((AREAD_CP s).SCTLR, s)


(read_info_seq ii) : ARMinfo M =
λs.SOME (AREAD_INFO s, s)


(read_version_seq ii) : num M =
λs.SOME (version_number(AREAD_INFO s).version, s)


(set_exclusive_monitorsT_seq ii(a : word32, size : num)) : unit M =
(λs.case word5_to_mode(AREAD_PSR CPSR s).M of
    NONE → NONE
‖ SOME m → SOME ((),
      let monitor = AREAD_EXCL s in
      let memaddrdesc = monitor.TranslateAddress(a, ¬(m = USR), F)
       in
        AWRITE_EXCL
          (monitor
            ⟪ IsExclusiveGlobal :=
              (if memaddrdesc.memattrs.shareable then
              monitor.MarkExclusiveGlobal
              (memaddrdesc.paddress, ii, size)
              monitor.IsExclusiveGlobal
               else monitor.IsExclusiveGlobal);
              IsExclusiveLocal :=
              monitor.MarkExclusiveLocal(memaddrdesc.paddress, ii, size)
              monitor.IsExclusiveLocal⟫)s))


(exclusive_monitors_passT_seq ii(a : word32, size : num)) : bool M =
λs.case word5_to_mode(AREAD_PSR CPSR s).M of
    NONE → NONE
‖ SOME m →
      let monitor = AREAD_EXCL s in
      let memaddrdesc = monitor.TranslateAddress(a, ¬(m = USR), F) in
      let local_pass = monitor.IsExclusiveLocal
                          (memaddrdesc.paddress, ii, size) in
      let passed =
            if memaddrdesc.memattrs.shareable then
```

```
                monitor.IsExclusiveLocal(memaddrdesc.paddress, ii, size) ∧
                local_pass
              else local_pass
        in
          SOME (passed,
            if passed then
              AWRITE_EXCL
                (let (local, global) =
                       monitor.ClearExclusiveLocal ii
                         (monitor.IsExclusiveLocal, monitor.IsExclusiveGlobal)
                 in
                   monitor
                   ⦇ IsExclusiveLocal := local;
                     IsExclusiveGlobal := global⦈)s
             else s)


(dmbT_seq : iiid → MBReqDomain#MBReqTypes → unit M)ii x =
λs.SOME ((), s)


(write_mem8_seq ii a x) : unit M =
(λs.case AREAD_MEM a s of
    NONE → NONE
  ‖ SOME y → SOME ((), AWRITE_MEM a(SOME x)s))


(write_mem16_seq ii a(x : word16)) : unit M =
if a[0] then
failureT_seq
else
bindT_seq(read_endian_seq ii)
(λe.let l = word2bytes 4 x in
  discardT_seq
    (if e then
       parT_seq(write_mem8_seq ii a(EL 1 l))
                (write_mem8_seq ii(a + 1w)(EL 0 l))
     else
       parT_seq(write_mem8_seq ii a(EL 0 l))
                (write_mem8_seq ii(a + 1w)(EL 1 l))))


(write_mem32_seq ii a(x : word32)) : unit M =
bindT_seq(read_endian_seq ii)
(λe.let aa = align(a, 4) and l = word2bytes 4 x in
discardT_seq
  (if e then
      parT_seq(write_mem8_seq ii aa(EL 3 l))
    (parT_seq(write_mem8_seq ii(aa + 1w)(EL 2 l))
    (parT_seq(write_mem8_seq ii(aa + 2w)(EL 1 l))
              (write_mem8_seq ii(aa + 3w)(EL 0 l)))))
```

```
      else
        parT_seq(write_mem8_seq ii aa(EL 0 l))
      (parT_seq(write_mem8_seq ii(aa + 1w)(EL 1 l))
      (parT_seq(write_mem8_seq ii(aa + 2w)(EL 2 l))
                 (write_mem8_seq ii(aa + 3w)(EL 3 l))))))

(read_mem8_seq ii a) : word8 M =
(λs.case AREAD_MEM a s of NONE → NONE ‖ SOME x → SOME (x, s))

(read_mem16_seq ii a) : word16 M =
if a[0] then
failureT_seq
else
bindT_seq(parT_seq(read_mem8_seq ii a)(read_mem8_seq ii(a + 1w)))
(λ(b0, b1). constT_seq(b1@@b0))

(read_mem32_seq ii a) : word32 M =
bindT_seq
(let aa = align(a, 4) in
   parT_seq(read_mem8_seq ii aa)
   (parT_seq(read_mem8_seq ii(aa + 1w))
   (parT_seq(read_mem8_seq ii(aa + 2w))
              (read_mem8_seq ii(aa + 3w)))))
(λ(b0, b1, b2, b3). constT_seq((b3@@b2@@b1@@b0)))

(constT : 'a → 'a M) = constT_seq

(addT : 'a → 'b M → ('a#'b)M) = addT_seq

(lockT : unit M → unit M) = lockT_seq

(failureT : unit M) = failureT_seq

(discardT : 'a M → unit M) = discardT_seq

(condT : bool → unit M → unit M) = condT_seq

(bindT : 'a M → (('a → 'b M) → 'b M)) = bindT_seq

(seqT : 'a M → 'b M → 'b M) = seqT_seq

(parT : 'a M → 'b M → ('a#'b)M) = parT_seq
```

(read_info : *iiid* → ARMinfo M) = read_info_seq

(read_version : *iiid* → num M) = read_version_seq

(write_reg : *iiid* → ARMreg → Aimm → unit M) = write_reg_seq

(read_reg : *iiid* → ARMreg → Aimm M) = read_reg_seq

(write_psr :
iiid → ARMpsr → ARMstatus → unit M) = write_psr_seq

(read_psr : *iiid* → ARMpsr → ARMstatus M) = read_psr_seq

(write_flags :
iiid → bool#bool#bool#bool → unit M) = write_flags_seq

(read_flags :
iiid → (bool#bool#bool#bool)M) = read_flags_seq

(read_mode : *iiid* → ARMmode M) = read_mode_seq

(read_instr_set : *iiid* → InstrSet M) = read_instr_set_seq

(read_sctlr : *iiid* → ARMsctlr M) = read_sctlr_seq

(write_mem8 : *iiid* → word32 → word8 → unit M) = write_mem8_seq

(read_mem8 : *iiid* → word32 → word8 M) = read_mem8_seq

(write_mem16 :
iiid → word32 → word16 → unit M) = write_mem16_seq

(read_mem16 : *iiid* → word32 → word16 M) = read_mem16_seq

(write_mem32 :
iiid → word32 → word32 → unit M) = write_mem32_seq

(read_mem32 : *iiid* → word32 → Aimm M) = read_mem32_seq

(dmbT :
iiid → MBReqDomain#MBReqTypes → unit M) = dmbT_seq

(set_exclusive_monitorsT :
iiid → (word32#num) → unit M) = set_exclusive_monitorsT_seq

(exclusive_monitors_passT :
iiid → (word32#num) → bool M) = exclusive_monitors_passT_seq

option_apply $x\ f =$ **if** $x =$ NONE **then** NONE **else** f(**the** x)

Part IX

arm_event_monad

type_abbrev M : eiid_state → ((eiid_state#′*a*#(arm_reg event_structure))set)

event_structure_empty =⟪ *events* :={}; *intra_causality_data* :={}; *intra_causality_control* :={}; *atomicity* :={}⟫

event_structure_lock *es* =⟪ *events* := *es*.*events*; *intra_causality_data* := *es*.*intra_causality_data*; *intra_causality_contro*

event_structure_union es_1 es_2 =
⟪ *events* := es_1.*events* ∪ es_2.*events*;
intra_causality_data := es_1.*intra_causality_data* ∪ es_2.*intra_causality_data*;
intra_causality_control := es_1.*intra_causality_control* ∪ es_2.*intra_causality_control*;
atomicity := es_1.*atomicity* ∪ es_2.*atomicity*⟫

event_structure_bigunion(*ess* : (arm_reg event_structure)set) =
⟪ *events* := **bigunion**{*es*.*events* | *es* ∈ *ess*};
intra_causality_data := **bigunion**{*es*.*intra_causality_data* | *es* ∈ *ess*};
intra_causality_control := **bigunion**{*es*.*intra_causality_control* | *es* ∈ *ess*};
atomicity := **bigunion**{*es*.*atomicity* | *es* ∈ *ess*}⟫

event_structure_seq_union es_1 es_2 =
⟪ *events* := es_1.*events* ∪ es_2.*events*;
intra_causality_data := es_1.*intra_causality_data*
∪ es_2.*intra_causality_data*
∪ {(e_1, e_2)
| e_1 ∈ (maximal_elements es_1.*events* es_1.*intra_causality_data*)
∧ e_2 ∈ (*minimal_elements* es_2.*events* es_2.*intra_causality_data*)};
intra_causality_control := es_1.*intra_causality_control*
∪ es_2.*intra_causality_control*;
atomicity := es_1.*atomicity* ∪ es_2.*atomicity*⟫

event_structure_control_seq_union es_1 es_2 =
⟪ *events* := es_1.*events* ∪ es_2.*events*;
intra_causality_data := es_1.*intra_causality_data*
∪ es_2.*intra_causality_data*;
intra_causality_control := es_1.*intra_causality_control*
∪ es_2.*intra_causality_control*
∪ {(e_1, e_2)
| e_1 ∈ (maximal_elements es_1.*events* es_1.*intra_causality_control*)
∧ e_2 ∈ (*minimal_elements* es_2.*events* es_2.*intra_causality_control*)};
atomicity := es_1.*atomicity* ∪ es_2.*atomicity*⟫

(mapT_ev : (′*a* → ′*b*) → ′*a* M → ′*b* M)*f* *s* =
λ*eiid_next* : eiid_state.
let *t* = *s* *eiid_next* **in**
{(*eiid_next*′, *f* *x*, *es*)
| (*eiid_next*′, *x*, *es*) ∈ *t*}

```
(choiceT_ev : 'a M → 'a M → 'a M)s s' =
λeiid_next : eiid_state.s eiid_next ∪ s' eiid_next


(constT_ev : 'a → 'a M)x = λeiid_next.{(eiid_next, x, event_structure_empty)}


(discardT_ev : 'a M → unit M)s =
λeiid_next.let (t : (eiid_state#'a#(arm_reg event_structure))set) = s eiid_next in
   image(λ(eiid_next', v, es).(eiid_next', (), es))t


(addT_ev : 'a → 'b M → ('a#'b)M)x s =
λeiid_next.let (t : (eiid_state#'b#(arm_reg event_structure))set) = s eiid_next in
   image(λ(eiid_next', v, es).(eiid_next', (x, v), es))t


(lockT_ev : 'a M → 'a M)s =
λeiid_next.let (t : (eiid_state#'a#(arm_reg event_structure))set) = s eiid_next in
   image(λ(eiid_next', v, es).(eiid_next', v, event_structure_lock es))t


(failureT_ev : 'a M) = λeiid_next.{}


(condT_ev : bool → unit M → unit M)b s =
if b then s else constT_ev()


(bindT_ev : 'a M → ('a → 'b M) → 'b M)s f =
λeiid_next : eiid_state.
let t = s eiid_next in
   bigunion{let t' = f x eiid_next' in
                    {(eiid_next'', x', event_structure_seq_union es es')
                     | (eiid_next'', x', es') ∈ t'}
            | (eiid_next', x, es) ∈ t}


(control_seqT_ev : 'a M → ('a → 'b M) → 'b M)s f =
λeiid_next : eiid_state.
let t = s eiid_next in
   bigunion{let t' = f x eiid_next' in
                    {(eiid_next'', x', event_structure_control_seq_union es es')
                     | (eiid_next'', x', es') ∈ t'}
            | (eiid_next', x, es) ∈ t}


(seqT_ev : 'a M → 'b M → 'b M)s s' =
bindT_ev s(λx.s')
```

```
(parT_ev : 'a M → 'b M → ('a#'b)M)s s' =
λeiid_next : eiid_state.
let  t = s eiid_next  in
   bigunion{let  t' = s' eiid_next'  in
                    {(eiid_next'', (x, x'), event_structure_union  es es')
                     | (eiid_next'', x', es') ∈  t'}
              | (eiid_next', x, es) ∈  t}


(parT_unit_ev : unit M → unit M → unit M)s s' =
λeiid_next : eiid_state.
let  t = s eiid_next  in
   bigunion{let  t' = s' eiid_next'  in
                    {(eiid_next'', (), event_structure_union  es es')
                     | (eiid_next'', (), es') ∈  t'}
              | (eiid_next', (), es) ∈  t}


(syncT_ev  ii) : unit M =
λeiid_next.{(eiid_next',
             (),
             ⦇ events :={⦇ eiid := eiid';
                           iiid := ii;
                           action := Barrier Sync⦈};
               intra_causality_data :={};
               intra_causality_control :={};
               atomicity :={}⦈) | (eiid', eiid_next') ∈ next_eiid  eiid_next}


(write_location_ev  ii l x) : unit M =
λeiid_next.{(eiid_next',
             (),
             ⦇ events :={⦇ eiid := eiid';
                           iiid := ii;
                           action := Access W  l  x⦈};
               intra_causality_data :={};
               intra_causality_control :={};
               atomicity :={}⦈) | (eiid', eiid_next') ∈ next_eiid  eiid_next}


(read_location_ev  ii l) : value M =
λeiid_next.{(eiid_next',
             x,
             ⦇ events :={⦇ eiid := eiid';
                           iiid := ii;
                           action := Access R  l  x⦈};
               intra_causality_data :={};
               intra_causality_control :={};
               atomicity :={}⦈)
        | x  ∈  UNIV ∧ (eiid', eiid_next') ∈ next_eiid  eiid_next}
```

(write_reg_ev $ii\ r\ x$) : unit M =
write_location_ev ii(LOCATION_REG ii. proc(REG32 r))x

(read_reg_ev $ii\ r$) : value M =
read_location_ev ii(LOCATION_REG ii. proc(REG32 r))

(write_psr_ev $ii\ r\ x$) : unit M =
write_location_ev ii(LOCATION_REG ii. proc(REGPSR r))(encode_psr x)

(read_psr_ev $ii\ r$) : ARMstatus M =
bindT_ev(read_location_ev ii(LOCATION_REG ii. proc(REGPSR r)))($\lambda psrw$. constT_ev(decode_psr $psrw$))

(write_flags_ev $ii(bn, bz, bc, bv)$) : unit M =
lockT_ev(bindT_ev(read_psr_ev ii CPSR)
(λpsr. write_psr_ev ii CPSR(psr ⟦ $N := bn; Z := bz; C := bc; V := bv$⟧)))

(read_flags_ev ii) : (bool#bool#bool#bool)M =
bindT_ev(read_psr_ev ii CPSR)(λpsr. constT_ev($psr.N, psr.Z, psr.C, psr.V$))

OUR_VERSION = ARMv7_A

OUR_INFO =⟦ $version$:= OUR_VERSION; $extensions$:={}⟧

OUR_MODE = USR

OUR_INSTR_SET = INSTRSET_ARM

OUR_SCTLR =⟦ $TE := \mathbf{T}; AFE := \mathbf{T}; TRE := \mathbf{T}; NMFI := \mathbf{T}; EE := \mathbf{T}; VE := \mathbf{T}; U := \mathbf{T}; FI := \mathbf{T}; HA := \mathbf{T}; RR := \mathbf{T}$;

(read_version_ev ii) : num M =
constT_ev(version_number OUR_VERSION)

(read_info_ev ii) : ARMinfo M =
constT_ev(OUR_INFO)

(read_mode_ev ii) : ARMmode M =
constT_ev(OUR_MODE)

(read_instr_set_ev ii) : InstrSet M =
constT_ev(OUR_INSTR_SET)

(read_sctlr_ev ii) : ARMsctlr M =
constT_ev(OUR_SCTLR)

(set_exclusive_monitorsT_ev $ii(a$: word32, $size$: num)) : unit M =
failureT_ev

(exclusive_monitors_passT_ev $ii(a$: word32, $size$: num)) : bool M =
failureT_ev

aligned32 $a = ((a\&\&3w) = 0w)$

(write_mem8_ev ii a x) : unit M =
failureT_ev

(write_mem16_ev ii $a(x$: word16)) : unit M =
failureT_ev

(write_mem32_ev ii $a(x$: word32)) : unit M =
if aligned32 a **then**
write_location_ev ii(LOCATION_MEM a)x
else
failureT_ev

(read_mem8_ev ii a) : word8 M =
failureT_ev

(read_mem16_ev ii a) : word16 M =
failureT_ev

(read_mem32_ev ii a) : word32 M =
if aligned32 a **then**
read_location_ev ii(LOCATION_MEM a)
else
failureT_ev

(dmbT_ev $ii(d, t)$) : unit M =
syncT_ev ii

(constT : $'a \rightarrow 'a$ M) = constT_ev

(addT : $'a \rightarrow 'b$ M $\rightarrow$ ($'a\#'b$)M) = addT_ev

(lockT : unit M → unit M) = lockT_ev

(failureT : unit M) = failureT_ev

(discardT : $'a$ M → unit M) = discardT_ev

(condT : bool → unit M → unit M) = condT_ev

(bindT : $'a$ M → (($'a$ → $'b$ M) → $'b$ M)) = bindT_ev

(seqT : $'a$ M → $'b$ M → $'b$ M) = seqT_ev

(parT : $'a$ M → $'b$ M → ($'a$#$'b$)M) = parT_ev

(parT_unit : unit M → unit M → unit M) = parT_unit_ev

(read_info : *iiid* → ARMinfo M) = read_info_ev

(read_version : *iiid* → num M) = read_version_ev

(write_reg : *iiid* → ARMreg → Aimm → unit M) = write_reg_ev

(read_reg : *iiid* → ARMreg → Aimm M) = read_reg_ev

(write_psr : *iiid* → ARMpsr → ARMstatus → unit M) = write_psr_ev

(read_psr : *iiid* → ARMpsr → ARMstatus M) = read_psr_ev

(write_flags :
iiid → bool#bool#bool#bool → unit M) = write_flags_ev

(read_flags :
iiid → (bool#bool#bool#bool)M) = read_flags_ev

(read_mode : *iiid* → ARMmode M) = read_mode_ev

(read_instr_set : *iiid* → InstrSet M) = read_instr_set_ev

(read_sctlr : *iiid* → ARMsctlr M) = read_sctlr_ev

(write_mem8 : *iiid* → word32 → word8 → unit M) = write_mem8_ev

(read_mem8 : *iiid* → word32 → word8 M) = read_mem8_ev

(write_mem16 : *iiid* → word32 → word16 → unit M) = write_mem16_ev

(read_mem16 : *iiid* → word32 → word16 M) = read_mem16_ev

(write_mem32 : *iiid* → word32 → word32 → unit M) = write_mem32_ev

(read_mem32 : *iiid* → word32 → Aimm M) = read_mem32_ev

(dmbT :
iiid → MBReqDomain#MBReqTypes → unit M) = dmbT_ev

(set_exclusive_monitorsT :
iiid → (word32#num) → unit M) = set_exclusive_monitorsT_ev

(exclusive_monitors_passT :
iiid → (word32#num) → bool M) = exclusive_monitors_passT_ev

Part X

arm_decoder

```
condition_decode(cond : word4) =
let  n = w2n((3 − −1)cond) in
num2ARMcondition(if  cond[0] then  n + 8  else  n)


arm_decode  version(ireg : word32) =
let  b n = ireg[n]
 and  i2 n = (n + 1 >< n)ireg : word2
 and  i3 n = (n + 2 >< n)ireg : word3
 and  i4 n = (n + 3 >< n)ireg : word4
 and  i5 n = (n + 4 >< n)ireg : word5
 and  i8 n = (n + 7 >< n)ireg : word8
 and  i12 n = (n + 11 >< n)ireg : word12
 and  i16 n = (n + 15 >< n)ireg : word16
 and  i24 n = (23 >< 0)ireg : word24  in
let  cond = i4 28  and  r = i4
 in
if  cond = 15w  then
if  version < 5  then
   (AL, UNPREDICTABLE)
 else
   case (b 27, b 26, b 25, b 24, b 23, b 22, b 21, b 20, b 7, b 6, b 5, b 4) of
         (F, T, F, T, F, T, T, T, F, T, F, T) →
            (AL, DATA_MEMORY_BARRIER(i4 0))
    || (F, _26, _25, _24, _23, _22, _21, _20, _7, _6, _5, _4) → (AL, UNDEFINED)
    || (T, F, F, _24, _23, _22, _21, _20, _7, _6, _5, _4) → (AL, UNDEFINED)
    || (T, F, T, b24, _23, _22, _21, _20, _7, _6, _5, _4) →
            (AL, BRANCH_LINK_EXCHANGE1 b24(i24 0))
    || (T, T, F, b24, _23, _22, _21, _20, _7, _6, _5, _4) →
            (AL,
               CP_LOAD_STORE2 b24(b 23)(b 22)(b 21)(b 20)(r 16)(r 12)
                  (i4 8)(i8 0))
    || (T, T, T, F, _23, _22, _21, _20, _7, _6, _5, F) →
            (AL, CP_DATA_PROCESSING2(i4 20)(r 16)(r 12)(i4 8)(i3 5)(r 0))
    || (T, T, T, F, _23, _22, _21, _20, _7, _6, _5, T) →
            (AL, CP_TRANSFER2(i3 21)(b 20)(r 16)(r 12)(i4 8)(i3 5)(r 0))
    || (T, T, T, T, _23, _22, _21, _20, _7, _6, _5, _4) → (AL, UNDEFINED)
 else
(condition_decode cond,
case (b 27, b 26, b 25, b 24, b 23, b 22, b 21, b 20, b 7, b 6, b 5, b 4) of
(* v — "Miscellaneous instructions" ————————————————————-v *)
      (F, F, F, T, F, b22, F, F, F, F, F, F) →
         STATUS_TO_REGISTER b22(r 12)
 || (F, F, F, T, F, b22, T, F, F, F, F, F) →
         REGISTER_TO_STATUS b22(i4 16)(r 0)
 || (F, F, F, T, F, F, T, F, F, F, F, T) →
         BRANCH_EXCHANGE(r 0)
 || (F, F, F, T, F, T, T, F, F, F, F, T) →
            COUNT_LEADING_ZEROES(r 12)(r 0)
 || (F, F, F, T, F, F, T, F, F, F, T, T) →
```

```
        BRANCH_LINK_EXCHANGE2(r 0)
 || (F, F, F, T, F, _22, _21, F, F, T, F, T) →
        DSP_ADD_SUBTRACT(i2 21)(r 16)(r 12)(r 0)
 || (F, F, F, T, F, F, T, F, F, T, T, T) →
        BREAKPOINT(i12 8@@i4 0)
 || (F, F, F, T, F, _22, _21, F, T, b6, _5, F) →
        DSP_MULTIPLY(r 16)(r 12)(r 8)b6(r 0)
(* ^————————————————————————————————————————^ *)
 || (F, F, F, _24, _23, _22, _21, b20, _7, _6, _5, F) →
        DATA_PROCESSING(i4 21)b20(r 16)(r 12)
          (MODE1_SHIFT_IMMEDIATE(i5 7)(i2 5)(r 0))
 || (F, F, F, _24, _23, _22, _21, b20, F, _6, _5, T) →
        DATA_PROCESSING(i4 21)b20(r 16)(r 12)
          (MODE1_SHIFT_REGISTER(r 8)(i2 5)(r 0))
(* v — "Multiplies and extra load/store instruction" ————————————v *)
 || (F, F, F, F, F, F, b21, b20, T, F, F, T) →
        MULTIPLY b21 b20(r 16)(r 12)(r 8)(r 0)
 || (F, F, F, F, T, b22, b21, b20, T, F, F, T) →
        MULTIPLY_LONG b22 b21 b20(r 16)(r 12)(r 8)(r 0)
 || (F, F, F, T, F, b22, F, F, T, F, F, T) →
        SWAP b22(r 16)(r 12)(r 0)
 || (F, F, F, T, T, F, F, F, T, F, F, T) →
        STORE_EXCLUSIVE(r 16)(r 12)(r 0)0w
 || (F, F, F, T, T, F, F, T, T, F, F, T) →
        LOAD_EXCLUSIVE(r 16)(r 12)0w
 || (F, F, F, b24, b23, F, b21, b20, T, F, T, T) →
        LOAD_STORE_HALFWORD b24 b23 b21 b20(r 16)(r 12)ARB
          (MODE3_REGISTER(r 0))
 || (F, F, F, b24, b23, T, b21, b20, T, F, T, T) →
        LOAD_STORE_HALFWORD b24 b23 b21 b20(r 16)(r 12)ARB
          (MODE3_IMMEDIATE(i4 8@@i4 0))
 || (F, F, F, b24, b23, F, b21, F, T, T, b5, T) →
        LOAD_STORE_TWO_WORDS b24 b23 b21(r 16)(r 12)b5
          (MODE3_REGISTER(r 0))
 || (F, F, F, b24, b23, F, b21, T, T, T, b5, T) →
        LOAD_STORE_HALFWORD b24 b23 b21 T(r 16)(r 12)b5
          (MODE3_REGISTER(r 0))
 || (F, F, F, b24, b23, T, b21, F, T, T, b5, T) →
        LOAD_STORE_TWO_WORDS b24 b23 b21(r 16)(r 12)b5
          (MODE3_IMMEDIATE(i4 8@@i4 0))
 || (F, F, F, b24, b23, T, b21, T, T, T, b5, T) →
        LOAD_STORE_HALFWORD b24 b23 b21 T(r 16)(r 12)b5
          (MODE3_IMMEDIATE(i4 8@@i4 0))
(* ^————————————————————————————————————————^ *)
 || (F, F, T, T, F, _22, F, F, _7, _6, _5, _4) →
        if version < 4 then UNPREDICTABLE else UNDEFINED
 || (F, F, T, T, F, b22, T, F, _7, _6, _5, _4) →
        IMMEDIATE_TO_STATUS b22(i4 16)(i4 8)(i8 0)
 || (F, F, T, _24, _23, _22, _21, b20, _7, _6, _5, _4) →
```

```
        DATA_PROCESSING(i4 21)b20(r 16)(r 12)
          (MODE1_IMMEDIATE(i4 8)(i8 0))
|| (F, T, F, b24, b23, b22, b21, b20, _7, _6, _5, _4) →
        LOAD_STORE b24 b23 b22 b21 b20(r 16)(r 12)
          (MODE2_IMMEDIATE(i12 0))
|| (F, T, T, b24, b23, b22, b21, b20, _7, _6, _5, F) →
        LOAD_STORE b24 b23 b22 b21 b20(r 16)(r 12)
          (MODE2_SHIFT_IMMEDIATE(i5 7)(i2 5)(r 0))
|| (F, T, T, _24, _23, _22, _21, _20, _7, _6, _5, T) →
        UNDEFINED
|| (T, F, F, b24, b23, b22, b21, b20, _7, _6, _5, _4) →
        LOAD_STORE_MULTIPLE b24 b23 b22 b21 b20(r 16)(i16 0)
|| (T, F, T, b24, _23, _22, _21, _20, _7, _6, _5, _4) →
        BRANCH b24(i24 0)
|| (T, T, F, b24, b23, b22, b21, b20, _7, _6, _5, _4) →
        CP_LOAD_STORE b24 b23 b22 b21 b20(r 16)(r 12)(i4 8)(i8 0)
|| (T, T, T, F, _23, _22, _21, _20, _7, _6, _5, F) →
        CP_DATA_PROCESSING(i4 20)(r 16)(r 12)(i4 8)(i3 5)(r 0)
|| (T, T, T, F, _23, _22, _21, b20, _7, _6, _5, T) →
        CP_TRANSFER(i3 21)b20(r 16)(r 12)(i4 8)(i3 5)(r 0)
|| (T, T, T, T, _23, _22, _21, _20, _7, _6, _5, _4) →
        SUPERVISOR_CALL(i24 0)
|| _ → if version < 4 then UNPREDICTABLE else UNDEFINED)
```

Part XI

arm_program

VERSION_MULTICORE = 7

type_abbrev Ainstruction : (ARMcondition#ARMinstruction)

type_abbrev program_Ainstruction : (address → (Ainstruction#num) option)

(arm_decode_program_fun : program_word8 → address → (Ainstruction#num) option)*prog_word8 a* =
if (*a*&&3*w* = 0*w*) **then** NONE
else
let *w0* = *prog_word8*(*a* + 0*w*) **in**
let *w1* = *prog_word8*(*a* + 1*w*) **in**
let *w2* = *prog_word8*(*a* + 2*w*) **in**
let *w3* = *prog_word8*(*a* + 3*w*) **in**
if mem NONE [*w0*; *w1*; *w2*; *w3*] **then** NONE **else**
let (*raw_instruction* : word32) = (**the** *w0*)@@(((**the** *w1*)@@(((**the** *w2*)@@(**the** *w3*)) : word16)) : word24) **in**
let *i* = arm_decode VERSION_MULTICORE *raw_instruction* **in**
SOME (*i*, 4)

(arm_decode_program_rel : program_word8 → program_Ainstruction → bool)
prog_word8 prog_Xinst =
∀*a*.**case** *prog_Xinst a* **of**
NONE → **T**
‖ SOME (*inst*, *n*) → arm_decode_program_fun *prog_word8 a* = SOME (*inst*, *n*)

arm_event_execute = *arm_event_opsem* $*arm_execute*

arm_execute_with_pc_check *ii inst pc* =
let *s* = (arm_event_execute *ii inst*){} **in**
{*E*
| ∃*eiid_next x*.
(*eiid_next*, *x*, *E*) ∈ *s* ∧
∀*v ev*.((*ev.action* = (ACCESS R(LOCATION_REG *ii*. proc(REG32 r15))*v*)) ∧
ev ∈ *E.events*) ⟹ (*v* = *pc*)
}

(arm_event_structures_of_run_skeleton : program_Ainstruction → run_skeleton → (arm_reg event_structure)set)
prog_Ainstruction rs =
let *Ess* = {arm_execute_with_pc_check⟦ proc := *p*; *poi* := *i*⟧*inst pc* | ∃*n*.
(*rs p i* = SOME *pc*) ∧ (SOME (*inst*, *n*) = *prog_Ainstruction pc*)}
in
{event_structure_bigunion *Es* | *Es* ∈ *all_choices Ess*}

(arm_semantics : program_word8 → (arm_reg state_constraint) →
(run_skeleton#program_Ainstruction#(((arm_reg event_structure)#((arm_reg execution_witness)set))set))set)
prog_word8 initial_state =

let *x1* = {(*rs*, *prog_Xinst*) | *rs*, *prog_Xinst* | run_skeleton_wf(*DOMAIN prog_Xinst*)*rs* ∧
arm_decode_program_rel *prog_word8 prog_Xinst*} **in**

let *x2* = {(*rs*, *prog_Xinst*, *Es*) | (*rs*, *prog_Xinst*) ∈ *x1* ∧
(*Es* = arm_event_structures_of_run_skeleton *prog_Xinst rs*)} **in**

let *x3* = {(*rs*, *prog_Xinst*, {(*E*, *Xs*) | *E* ∈ *Es* ∧
(*Xs* = {*X* | valid_execution *E X* ∧
(*X*.*initial_state* = *initial_state*)})})
| (*rs*, *prog_Xinst*, *Es*) ∈ *x2*} **in**
x3

Part XII

ppc_coretypes

type_abbrev ireg : word5

type_abbrev freg : word5

type_abbrev ppc_constant : word16

type_abbrev crbit : word2

ppc_bit = PPC_CARRY(* carry bit of the status register *)
| PPC_CR0 **of** word2(* bit i of the condition register *)

ppc_reg32 = PPC_IR **of** word5(* integer registers *)
| PPC_LR(* link register (return address) *)
| PPC_CTR(* count register, used for some branches *)
| PPC_PC(* program counter *)

Part XIII

ppc_types

ppc_reg = REG32 **of** ppc_reg32 | REGBIT **of** ppc_bit

Part XIV

ppc_ast

```
Pinstruction =
Padd of ireg ireg ireg(* integer addition *)
| Paddi of ireg ireg ppc_constant(* add immediate *)
| Paddis of ireg ireg ppc_constant(* add immediate high *)
| Paddze of ireg ireg(* add Carry bit *)
| Pand_ of ireg ireg ireg(* bitwise and *)
| Pandc of ireg ireg ireg(* bitwise and-complement *)
| Pandi_ of ireg ireg ppc_constant(* and immediate and set conditions *)
| Pandis_ of ireg ireg ppc_constant(* and immediate high and set conditions *)
| Pb of word24(* unconditional branch *)
| Pbctr(* branch to contents of register CTR *)
| Pbctrl(* branch to contents of CTR and link *)
| Pbf of crbit 14 word(* branch if false *)
| Pbl of word24(* branch and link *)
| Pbs of word24(* branch to symbol *)
| Pblr(* branch to contents of register LR *)
| Pbt of crbit 14 word(* branch if true *)
| Pcmplw of ireg ireg(* unsigned integer comparison *)
| Pcmplwi of ireg ppc_constant(* same, with immediate argument *)
| Pcmpw of ireg ireg(* signed integer comparison *)
| Pcmpwi of ireg ppc_constant(* same, with immediate argument *)
| Pcror of crbit crbit crbit(* or between condition bits *)
| Pdivw of ireg ireg ireg(* signed division *)
| Pdivwu of ireg ireg ireg(* unsigned division *)
| Peqv of ireg ireg ireg(* bitwise not-xor *)
| Pextsb of ireg ireg(* 8-bit sign extension *)
| Pextsh of ireg ireg(* 16-bit sign extension *)
| Pfabs of freg freg(* float absolute value *)
| Pfadd of freg freg freg(* float addition *)
| Pfcmpu of freg freg(* float comparison *)
| Pfcti of ireg freg(* float-to-int conversion *)
| Pfdiv of freg freg freg(* float division *)
| Pfmadd of freg freg freg freg(* float multiply-add *)
| Pfmr of freg freg(* float move *)
| Pfmsub of freg freg freg freg(* float multiply-sub *)
| Pfmul of freg freg freg(* float multiply *)
| Pfneg of freg freg(* float negation *)
| Pfrsp of freg freg(* float round to single precision *)
| Pfsub of freg freg freg(* float subtraction *)
| Pictf of freg ireg(* int-to-float conversion *)
| Piuctf of freg ireg(* unsigned int-to-float conversion *)
| Plbz of ireg ppc_constant ireg(* load 8-bit unsigned word32 *)
| Plbzx of ireg ireg ireg(* same, with 2 index regs *)
| Plfd of freg ppc_constant ireg(* load 64-bit float *)
| Plfdx of freg ireg ireg(* same, with 2 index regs *)
| Plfs of freg ppc_constant ireg(* load 32-bit float *)
| Plfsx of freg ireg ireg(* same, with 2 index regs *)
| Plha of ireg ppc_constant ireg(* load 16-bit signed word32 *)
| Plhax of ireg ireg ireg(* same, with 2 index regs *)
```

```
| Plhz of ireg ppc_constant ireg(* load 16-bit unsigned word32 *)
| Plhzx of ireg ireg ireg(* same, with 2 index regs *)
| Plwarx of ireg ireg ireg(* load 32-bit word and reserve index *)
| Plwz of ireg ppc_constant ireg(* load 32-bit word32 *)
| Plwzx of ireg ireg ireg(* same, with 2 index regs *)
| Pmfcrbit of ireg crbit(* move condition bit to reg *)
| Pmflr of ireg(* move LR to reg *)
| Pmr of ireg ireg(* integer move *)
| Pmtctr of ireg(* move ireg to CTR *)
| Pmtlr of ireg(* move ireg to LR *)
| Pmulli of ireg ireg ppc_constant(* integer multiply immediate *)
| Pmullw of ireg ireg ireg(* integer multiply *)
| Pnand of ireg ireg ireg(* bitwise not-and *)
| Pnor of ireg ireg ireg(* bitwise not-or *)
| Por of ireg ireg ireg(* bitwise or *)
| Porc of ireg ireg ireg(* bitwise or-complement *)
| Pori of ireg ireg ppc_constant(* or with immediate *)
| Poris of ireg ireg ppc_constant(* or with immediate high *)
| Prlwinm of ireg ireg word5 word5 word5(* rotate and mask *)
| Pslw of ireg ireg ireg(* shift left *)
| Psraw of ireg ireg ireg(* shift right signed *)
| Psrawi of ireg ireg word5(* shift right signed immediate *)
| Psrw of ireg ireg ireg(* shift right unsigned *)
| Pstb of ireg ppc_constant ireg(* store 8-bit word *)
| Pstbx of ireg ireg ireg(* same, with 2 index regs *)
| Pstfd of freg ppc_constant ireg(* store 64-bit float *)
| Pstfdx of freg ireg ireg(* same, with 2 index regs *)
| Pstfs of freg ppc_constant ireg(* store 32-bit float *)
| Pstfsx of freg ireg ireg(* same, with 2 index regs *)
| Psth of ireg ppc_constant ireg(* store 16-bit word *)
| Psthx of ireg ireg ireg(* same, with 2 index regs *)
| Pstw of ireg ppc_constant ireg(* store 32-bit word *)
| Pstwcx of ireg ireg ireg(* store word conditional indexed *)
| Pstwx of ireg ireg ireg(* store 32-bit word, with 2 index regs *)
| Psubfc of ireg ireg ireg(* reversed integer subtraction *)
| Psubfic of ireg ireg ppc_constant(* integer subtraction from immediate *)
| Psync(* synchronize *)
| Pxor of ireg ireg ireg(* bitwise xor *)
| Pxori of ireg ireg ppc_constant(* bitwise xor with immediate *)
| Pxoris of ireg ireg ppc_constant(* bitwise xor with immediate high *)
```

Part XV

ppc_opsem

ppc_sint_cmp $ii(a : \mathsf{word32})(b : \mathsf{word32}) =$
(parT_unit(write_status ii(PPC_CR0 $0w$)(SOME $(a < b)$)))
(parT_unit(write_status ii(PPC_CR0 $1w$)(SOME $(b < a)$)))
(parT_unit(write_status ii(PPC_CR0 $2w$)(SOME $(a = b)$)))
(write_status ii(PPC_CR0 $3w$)NONE))))

ppc_uint_cmp $ii(a : \mathsf{word32})(b : \mathsf{word32}) =$
(parT_unit(write_status ii(PPC_CR0 $0w$)(SOME $(a < +b)$)))
(parT_unit(write_status ii(PPC_CR0 $1w$)(SOME $(b < +a)$)))
(parT_unit(write_status ii(PPC_CR0 $2w$)(SOME $(a = b)$)))
(write_status ii(PPC_CR0 $3w$)NONE))))

OK_nextinstr $ii\ f =$
parT_unit f(seqT(read_reg ii PPC_PC)($\lambda x.$ write_reg ii PPC_PC$(x + 4w)$))

reg_update $ii\ rd\ f\ s1\ s2 =$
seqT(parT $s1\ s2$)($\lambda(x, y).$ write_reg ii(PPC_IR rd)($f\ x\ y$))

uint_reg_update $ii\ rd\ f\ s1\ s2 =$
seqT(parT $s1\ s2$)
($\lambda(x, y).$ parT_unit(write_reg ii(PPC_IR rd)($f\ x\ y$))(ppc_uint_cmp $ii(f\ x\ y)0w$))

sint_reg_update $ii\ rd\ f\ s1\ s2 =$
seqT(parT $s1\ s2$)
($\lambda(x, y).$ parT_unit(write_reg ii(PPC_IR rd)($f\ x\ y$))(ppc_sint_cmp $ii(f\ x\ y)0w$))

uint_compare $ii\ s1\ s2 =$
control_seqT(parT $s1\ s2$)($\lambda(x, y).$ ppc_uint_cmp $ii\ x\ y$)

sint_compare $ii\ s1\ s2 =$
control_seqT(parT $s1\ s2$)($\lambda(x, y).$ ppc_sint_cmp $ii\ x\ y$)

bit_update $ii\ bd(f : \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool})s1\ s2 =$
seqT(parT $s1\ s2$)($\lambda(x, y).$ write_status $ii\ bd$(SOME $(f\ x\ y)$))

const_low $w =$ constT$((w2w : \mathsf{word16} \to \mathsf{word32})w)$

const_high $w =$ constT$((w2w : \mathsf{word16} \to \mathsf{word32})w \ll 16)$

conditional $x\ y\ z =$ **if** x **then** y **else** z

read_bit_word $ii\ bit =$
seqT(read_status $ii\ bit$)($\lambda x.$ constT(conditional $x\ 1w\ 0w$))

read_ireg *ii rd* = read_reg *ii*(PPC_IR *rd*)

gpr_or_zero *ii d* = **if** *d* = 0*w* **then** const_low 0*w* **else** read_ireg *ii d*

no_carry *ii* = write_status *ii* PPC_CARRY None

goto_label *ii l* =
seqT(read_reg *ii* PPC_PC)(λ*x*. write_reg *ii* PPC_PC(*x* + *sw2sw l* ∗ 4*w*))

effective_address *s1 s2* = seqT(parT *s1 s2*)(λ(*x* : word32, *y* : word32). constT(*x* + *y*))

assertT *b f* = seqT(**if** *b* **then** constT() **else** failureT)(λ*x*.*f*)

(write_mem_aux *ii addr*[ ] = constT()) ∧
(write_mem_aux *ii addr*(*b* ∈ *bytes*) =
parT_unit(write_mem8 *ii addr b*)
(write_mem_aux *ii*(*addr* + 1*w*)*bytes*))

store_word *ii size addr* value =
assertT((*address_aligned size addr*) ∧ (*size* = 4))
(write_mem32 *ii addr* value)

register_store *ii size rd s1 s2* =
seqT(parT(effective_address *s1 s2*)(read_ireg *ii rd*))
(λ(*addr*, *x*). store_word *ii size addr x*)

read_mem_aux *ii size addr* =
if *size* = 1 **then**
seqT(read_mem8 *ii addr*)
(λ*x*. constT((*bytes2word*[*x*]) : word32))
else if *size* = 2 **then**
seqT(parT(read_mem8 *ii addr*)(read_mem8 *ii*(*addr* + 1*w*)))
(λ(*x*0, *x*1). constT(*bytes2word*[*x*1; *x*0]))
else
seqT(parT(parT(read_mem8 *ii*(*addr* + 0*w*))(read_mem8 *ii*(*addr* + 1*w*)))
(parT(read_mem8 *ii*(*addr* + 2*w*))(read_mem8 *ii*(*addr* + 3*w*))))
(λ((*x*0, *x*1), (*x*2, *x*3)). constT(*bytes2word*[*x*3; *x*2; *x*1; *x*0]))

load_word *ii size addr* =
assertT((*address_aligned size addr*) ∧ (*size* = 2))
(read_mem32 *ii addr*)

register_load *ii size rd s1 s2* =
seqT(effective_address *s1 s2*)
(λ*addr*. seqT(load_word *ii size addr*)
(write_reg *ii*(PPC_IR *rd*)))

set_CR0_to_00xNONE *ii b* =
(parT_unit(write_status *ii*(PPC_CR0 0*w*)(SOME **F**))
(parT_unit(write_status *ii*(PPC_CR0 1*w*)(SOME **F**))
(parT_unit(write_status *ii*(PPC_CR0 2*w*)(SOME *b*))
(write_status *ii*(PPC_CR0 3*w*)NONE))))

(ppc_exec_instr *ii*(PADD *rd* R1 R2) =
OK_nextinstr *ii*(reg_update *ii rd*$ + (read_ireg *ii* R1)(read_ireg *ii* R2))) ∧

(ppc_exec_instr *ii*(PADDI *rd* R1 *cst*) =
OK_nextinstr *ii*(reg_update *ii rd*$ + (gpr_or_zero *ii* R1)(const_low *cst*))) ∧

(ppc_exec_instr *ii*(PADDIS *rd* R1 *cst*) =
OK_nextinstr *ii*(reg_update *ii rd*$ + (gpr_or_zero *ii* R1)(const_high *cst*))) ∧

(ppc_exec_instr *ii*(PADDZE *rd* R1) =
OK_nextinstr *ii*(reg_update *ii rd*$ + (read_ireg *ii* R1)(read_bit_word *ii* PPC_CARRY))) ∧

(ppc_exec_instr *ii*(PAND_ *rd* R1 R2) =
OK_nextinstr *ii*(sint_reg_update *ii rd*$&&&(read_ireg *ii* R1)(read_ireg *ii* R2))) ∧

(ppc_exec_instr *ii*(PANDC *rd* R1 R2) =
OK_nextinstr *ii*(reg_update *ii rd*(λ*x y*.*x*&&&¬*y*)(read_ireg *ii* R1)(read_ireg *ii* R2))) ∧

(ppc_exec_instr *ii*(PANDL_ *rd* R1 *cst*) =
OK_nextinstr *ii*(sint_reg_update *ii rd*$&&&(read_ireg *ii* R1)(const_low *cst*))) ∧

(ppc_exec_instr *ii*(PANDIS_ *rd* R1 *cst*) =
OK_nextinstr *ii*(sint_reg_update *ii rd*$&&&(read_ireg *ii* R1)(const_high *cst*))) ∧

(ppc_exec_instr *ii*(PB *lbl*) =
goto_label *ii lbl*) ∧

(ppc_exec_instr *ii*(PBCTR) =
seqT(read_reg *ii* PPC_CTR)(write_reg *ii* PPC_PC)) ∧

(ppc_exec_instr *ii*(PBCTRL) =
seqT(parT(read_reg *ii* PPC_PC)(read_reg *ii* PPC_CTR))
(λ(*pc*, *ctr*). parT_unit(write_reg *ii* PPC_PC *ctr*)(write_reg *ii* PPC_LR(*pc* + 4*w*)))) ∧

(ppc_exec_instr *ii*(PBF *bit lb1*) =
seqT(read_status *ii*(PPC_CR0 *bit*))
(λ*b*.**if** *b* **then** goto_label *ii lb1* **else** OK_nextinstr *ii*(constT()))) ∧

(ppc_exec_instr *ii*(PBL *ident*) =
  seqT(read_reg *ii* PPC_PC)
    (λ*x*. parT_unit(write_reg *ii* PPC_PC(*x* + *sw2sw ident* ∗ 4*w*))(write_reg *ii* PPC_LR(*x* + 4*w*)))) ∧

(ppc_exec_instr *ii*(PBS *ident*) =
  goto_label *ii ident*) ∧

(ppc_exec_instr *ii*(PBLR) =
  seqT(read_reg *ii* PPC_LR)(write_reg *ii* PPC_PC)) ∧

(ppc_exec_instr *ii*(PBT *bit lb1*) =
  control_seqT(read_status *ii*(PPC_CR0 *bit*))
    (λ*b*.**if** ¬*b* **then** goto_label *ii lb1* **else** OK_nextinstr *ii*(constT()))) ∧

(ppc_exec_instr *ii*(PCMPLW R1 R2) =
OK_nextinstr *ii*(uint_compare *ii*(read_ireg *ii* R1)(read_ireg *ii* R2))) ∧

(ppc_exec_instr *ii*(PCMPLWI R1 *cst*) =
OK_nextinstr *ii*(uint_compare *ii*(read_ireg *ii* R1)(const_low *cst*))) ∧

(ppc_exec_instr *ii*(PCMPW R1 R2) =
OK_nextinstr *ii*(sint_compare *ii*(read_ireg *ii* R1)(read_ireg *ii* R2))) ∧

(ppc_exec_instr *ii*(PCMPWI R1 *cst*) =
OK_nextinstr *ii*(sint_compare *ii*(read_ireg *ii* R1)(const_low *cst*))) ∧

(ppc_exec_instr *ii*(PCROR *bd* b_1 b_2) =
OK_nextinstr *ii*(bit_update *ii*(PPC_CR0 *bd*)((read_status *ii*(PPC_CR0 b_1)) ∨ (read_status *ii*(PPC_CR0 b_2))))) ∧

(ppc_exec_instr *ii*(PDIVW *rd* R1 R2) = failureT) ∧

(ppc_exec_instr *ii*(PDIVWU *rd* R1 R2) = failureT) ∧

(ppc_exec_instr *ii*(PEQV *rd* R1 R2) =
OK_nextinstr *ii*(reg_update *ii rd*(λ*x y*.¬(*x*??*y*))(read_ireg *ii* R1)(read_ireg *ii* R2))) ∧

(ppc_exec_instr *ii*(PEXTSB *rd* R1) =
OK_nextinstr *ii*(reg_update *ii rd*(λ*x y.sw2sw*((*w2w x*) : word8))
    (read_ireg *ii* R1)(constT()))) ∧

(ppc_exec_instr *ii*(PEXTSH *rd* R1) =
OK_nextinstr *ii*(reg_update *ii rd*(λ*x y.sw2sw*((*w2w x*) : word16))
  (read_ireg *ii* R1)(constT()))) ∧

(ppc_exec_instr *ii*(PFABS *rd* R1) = failureT) ∧

(ppc_exec_instr *ii*(PFADD *rd* R1 R2) = failureT) ∧

(ppc_exec_instr ii(PFCMPU R1 R2) = failureT) ∧

(ppc_exec_instr ii(PFCTI rd R1) = failureT) ∧

(ppc_exec_instr ii(PFDIV rd R1 R2) = failureT) ∧

(ppc_exec_instr ii(PFMADD rd R1 R2 R3) = failureT) ∧

(ppc_exec_instr ii(PFMR rd R1) = failureT) ∧

(ppc_exec_instr ii(PFMSUB rd R1 R2 R3) = failureT) ∧

(ppc_exec_instr ii(PFMUL rd R1 R2) = failureT) ∧

(ppc_exec_instr ii(PFNEG rd R1) = failureT) ∧

(ppc_exec_instr ii(PFRSP rd R1) = failureT) ∧

(ppc_exec_instr ii(PFSUB rd R1 R2) = failureT) ∧

(ppc_exec_instr ii(PICTF rd R1) = failureT) ∧

(ppc_exec_instr ii(PIUCTF rd R1) = failureT) ∧

(ppc_exec_instr ii(PLBZ rd cst R1) =
OK_nextinstr ii(register_load ii 1 rd(read_ireg ii R1)(const_low cst))) ∧

(ppc_exec_instr ii(PLBZX rd R1 R2) =
OK_nextinstr ii(register_load ii 1 rd(read_ireg ii R1)(read_ireg ii R2))) ∧

(ppc_exec_instr ii(PLFD rd cst R1) = failureT) ∧

(ppc_exec_instr ii(PLFDX rd R1 R2) = failureT) ∧

(ppc_exec_instr ii(PLFS rd cst R1) = failureT) ∧

(ppc_exec_instr ii(PLFSX rd R1 R2) = failureT) ∧

(ppc_exec_instr ii(PLHA rd cst R1) =
OK_nextinstr ii(register_load ii 2 rd(read_ireg ii R1)(const_low cst))) ∧

(ppc_exec_instr ii(PLHAX rd R1 R2) =
OK_nextinstr ii(register_load ii 2 rd(read_ireg ii R1)(read_ireg ii R2))) ∧

(ppc_exec_instr ii(PLHZ rd cst R1) =
OK_nextinstr ii(register_load ii 2 rd(read_ireg ii R1)(const_low cst))) ∧

(ppc_exec_instr ii(PLHZX rd R1 R2) =
OK_nextinstr ii(register_load ii 2 rd(read_ireg ii R1)(read_ireg ii R2))) ∧

(ppc_exec_instr *ii*(PLWARX *rd ra rb*) =
OK_nextinstr *ii*(
  seqT(effective_address(gpr_or_zero *ii ra*)(read_ireg *ii rb*))
      (λ*ea*. lockT(parT_unit(write_reserve_bit *ii* **T**)(
                  parT_unit(write_reserve_address *ii ea*)
                      (seqT(load_word *ii* 4 *ea*)(write_reg *ii*(PPC_IR *rd*))))))))) ∧

(ppc_exec_instr *ii*(PLWZ *rd cst* R1) =
OK_nextinstr *ii*(register_load *ii* 4 *rd*(read_ireg *ii* R1)(const_low *cst*))) ∧

(ppc_exec_instr *ii*(PLWZX *rd* R1 R2) =
OK_nextinstr *ii*(register_load *ii* 4 *rd*(read_ireg *ii* R1)(read_ireg *ii* R2))) ∧

(ppc_exec_instr *ii*(PMFCRBIT *v162 v163*) = failureT) ∧

(ppc_exec_instr *ii*(PMFLR *rd*) =
OK_nextinstr *ii*(seqT(read_reg *ii* PPC_LR)(write_reg *ii*(PPC_IR *rd*)))) ∧

(ppc_exec_instr *ii*(PMR *rd* R1) =
OK_nextinstr *ii*(seqT(read_ireg *ii* R1)(write_reg *ii*(PPC_IR *rd*)))) ∧

(ppc_exec_instr *ii*(PMTCTR R1) =
OK_nextinstr *ii*(seqT(read_ireg *ii* R1)(write_reg *ii* PPC_CTR))) ∧

(ppc_exec_instr *ii*(PMTLR R1) =
OK_nextinstr *ii*(seqT(read_ireg *ii* R1)(write_reg *ii* PPC_LR))) ∧

(ppc_exec_instr *ii*(PMULLI *rd* R1 *cst*) =
OK_nextinstr *ii*(reg_update *ii rd*$ ∗ (read_ireg *ii* R1)(const_low *cst*))) ∧

(ppc_exec_instr *ii*(PMULLW *rd* R1 R2) =
OK_nextinstr *ii*(reg_update *ii rd*$ ∗ (read_ireg *ii* R1)(read_ireg *ii* R2))) ∧

(ppc_exec_instr *ii*(PNAND *rd* R1 R2) =
OK_nextinstr *ii*(reg_update *ii rd*(λ*x y*.¬(*x*&&*y*))(read_ireg *ii* R1)(read_ireg *ii* R2))) ∧

(ppc_exec_instr *ii*(PNOR *rd* R1 R2) =
OK_nextinstr *ii*(reg_update *ii rd*(λ*x y*.¬(*x*!!*y*))(read_ireg *ii* R1)(read_ireg *ii* R2))) ∧

(ppc_exec_instr *ii*(POR *rd* R1 R2) =
OK_nextinstr *ii*(reg_update *ii rd*$!!(read_ireg *ii* R1)(read_ireg *ii* R2))) ∧

(ppc_exec_instr *ii*(PORC *rd* R1 R2) =
OK_nextinstr *ii*(reg_update *ii rd*(λ*x y*.*x*!!¬*y*)(read_ireg *ii* R1)(read_ireg *ii* R2))) ∧

(ppc_exec_instr *ii*(PORI *rd* R1 *cst*) =
OK_nextinstr *ii*(reg_update *ii rd*$!!(read_ireg *ii* R1)(const_low *cst*))) ∧

(ppc_exec_instr *ii*(PORIS *rd* R1 *cst*) =
OK_nextinstr *ii*(reg_update *ii* *rd*$!!(read_ireg *ii* R1)(const_high *cst*))) ∧

(ppc_exec_instr *ii*(PRLWINM *rd* R1 *sh* *mb* *me*) = failureT) ∧

(ppc_exec_instr *ii*(PSLW *rd* R1 R2) =
OK_nextinstr *ii*(reg_update *ii* *rd*(λ*x* *y*.*x* ≪ **w2n**((*w2w* *y*) : word6))(read_ireg *ii* R1)(read_ireg *ii* R2))) ∧

(ppc_exec_instr *ii*(PSRAW *rd* R1 R2) =
OK_nextinstr *ii*(parT_unit(reg_update *ii* *rd*(λ*x* *y*.*x* >>> **w2n**((*w2w* *y*) : word6))(read_ireg *ii* R1)(read_ireg *ii* R2))
(no_carry *ii*))) ∧

(ppc_exec_instr *ii*(PSRAWI *rd* R1 *sh*) =
OK_nextinstr *ii*(parT_unit(reg_update *ii* *rd*(λ*x* : *word*$_{32}$ *y* : *word*$_{32}$.*x* >>> **w2n**((*w2w* *y*) : word6))(read_ireg *ii* R1)
(no_carry *ii*))) ∧

(ppc_exec_instr *ii*(PSRW *rd* R1 R2) =
OK_nextinstr *ii*(reg_update *ii* *rd*(λ*x* *y*.*x* ≫ **w2n**((*w2w* *y*) : word6))(read_ireg *ii* R1)(read_ireg *ii* R2))) ∧

(ppc_exec_instr *ii*(PSTB *rd* *cst* R1) =
OK_nextinstr *ii*(register_store *ii* 1 *rd*(read_ireg *ii* R1)(const_low *cst*))) ∧

(ppc_exec_instr *ii*(PSTBX *rd* R1 R2) =
OK_nextinstr *ii*(register_store *ii* 1 *rd*(read_ireg *ii* R1)(read_ireg *ii* R2))) ∧

(ppc_exec_instr *ii*(PSTFD *rd* *cst* R1) = failureT) ∧

(ppc_exec_instr *ii*(PSTFDX *rd* R1 R2) = failureT) ∧

(ppc_exec_instr *ii*(PSTFS *rd* *cst* R1) = failureT) ∧

(ppc_exec_instr *ii*(PSTFSX *rd* R1 R2) = failureT) ∧

(ppc_exec_instr *ii*(PSTH *rd* *cst* R1) =
OK_nextinstr *ii*(register_store *ii* 2 *rd*(read_ireg *ii* R1)(const_low *cst*))) ∧

(ppc_exec_instr *ii*(PSTHX *rd* R1 R2) =
OK_nextinstr *ii*(register_store *ii* 2 *rd*(read_ireg *ii* R1)(read_ireg *ii* R2))) ∧

(ppc_exec_instr *ii*(PSTWCX *rs* *ra* *rb*) =
OK_nextinstr *ii*(
seqT(parT(effective_address(gpr_or_zero *ii* *ra*)(read_ireg *ii* *rb*))
(read_reserve_bit *ii*))
(λ(*ea*, *reserve*).
if *reserve* **then**
lockT(parT_unit(seqT(read_ireg *ii* *rs*)(λ*x*.store_word *ii* 4 *ea* *x*))
(parT_unit(set_CR0_to_00xNONE *ii* **T**)
(write_reserve_bit *ii* **F**)))
else

(set_CR0_to_00xNONE *ii* **T**)))) ∧

(ppc_exec_instr *ii*(PSTW *rd cst* R1) =
OK_nextinstr *ii*(register_store *ii* 4 *rd*(read_ireg *ii* R1)(const_low *cst*))) ∧

(ppc_exec_instr *ii*(PSTWX *rd* R1 R2) =
OK_nextinstr *ii*(register_store *ii* 4 *rd*(read_ireg *ii* R1)(read_ireg *ii* R2))) ∧

(ppc_exec_instr *ii*(PSUBFC *rd* R1 R2) =
OK_nextinstr *ii*(parT_unit(reg_update *ii* *rd*$ − (read_ireg *ii* R2)(read_ireg *ii* R1))
(no_carry *ii*))) ∧

(ppc_exec_instr *ii*(PSUBFIC *rd* R1 *cst*) =
OK_nextinstr *ii*(parT_unit(reg_update *ii* *rd*$ − (const_low *cst*)(read_ireg *ii* R1))
(no_carry *ii*))) ∧

(ppc_exec_instr *ii*(PSYNC) =
OK_nextinstr *ii*(syncT *ii*)) ∧

(ppc_exec_instr *ii*(PXOR *rd* R1 R2) =
OK_nextinstr *ii*(reg_update *ii* *rd*$??(read_ireg *ii* R1)(read_ireg *ii* R2))) ∧

(ppc_exec_instr *ii*(PXORI *rd* R1 *cst*) =
OK_nextinstr *ii*(reg_update *ii* *rd*$??(read_ireg *ii* R1)(const_low *cst*))) ∧

(ppc_exec_instr *ii*(PXORIS *rd* R1 *cst*) =
OK_nextinstr *ii*(reg_update *ii* *rd*$??(read_ireg *ii* R1)(const_high *cst*)))

Part XVI

ppc_decoder

```
ppc_match_step name =
if name = "0" then DF else
if name = "1" then DT else
if mem name ["A";"B";"C";"D";"S";"BI";"crbA";"crbB";"crbD";"SH";"MB";"ME"] then
assign_drop name 5
 else if mem name ["BD"] then
assign_drop name 14
 else if mem name ["SIMM";"UIMM";"d"] then
assign_drop name 16
 else if mem name ["LI"] then
assign_drop name 24
 else if mem name ["AA";"Rc";"OE";"y";"z"] then
assign_drop name 1
 else
option_fail

ppc_decode = match_list ppc_match_step(REVERSE o tokenise)(λk x.SOME (k(fst x)))^ppc_syntax
```

Part XVII

ppc_

iiid_dummy =⟦ proc := 0; *poi* := 0⟧

PPC_NEXT *s* =
let *pc* = PREAD_R PPC_PC *s* **in**
let *w0* = PREAD_M(*pc* + 0*w*)*s* **in**
let *w1* = PREAD_M(*pc* + 1*w*)*s* **in**
let *w2* = PREAD_M(*pc* + 2*w*)*s* **in**
let *w3* = PREAD_M(*pc* + 3*w*)*s* **in**
let *raw_instruction* = *w2bits*(**the** *w0*) + +*w2bits*(**the** *w1*) + +*w2bits*(**the** *w2*) + +*w2bits*(**the** *w3*) **in**
let *i* = ppc_decode *raw_instruction* **in**
let *s*′ = ppc_exec_instr iiid_dummy(**the** *i*)*s* **in**
if ¬(*pc*&&3*w* = 0*w*) ∨ **mem** NONE [*w0*; *w1*; *w2*; *w3*] ∨ (*i* = NONE) ∨ (*s*′ = NONE) **then** NONE
else SOME (**snd**(**the** *s*′))

Part XVIII

ppc_seq_monad

type_abbrev ppc_state : (ppc_reg32 → word32)#(ppc_bit → bool option)#(word32 → word8 option)#bool#word32

PREAD_R $rd((r, s, m, rb, ra) : \mathsf{ppc\_state}) = r\ rd$

PREAD_S $rd((r, s, m, rb, ra) : \mathsf{ppc\_state}) = s\ rd$

PREAD_M $rd((r, s, m, rb, ra) : \mathsf{ppc\_state}) = m\ rd$

PWRITE_R $rd\ x(r, s, m, rb, ra) = ((rd = +x)r, s, m, rb, ra) : \mathsf{ppc\_state}$

PWRITE_S $rd\ x(r, s, m, rb, ra) = (r, (rd = +x)s, m, rb, ra) : \mathsf{ppc\_state}$

PWRITE_M $rd\ x(r, s, m, rb, ra) = (r, s, (rd = +x)m, rb, ra) : \mathsf{ppc\_state}$

PREAD_REVERSE_BIT$((r, s, m, rb, ra) : \mathsf{ppc_state}) = rb$

PREAD_REVERSE_ADDRESS$((r, s, m, rb, ra) : \mathsf{ppc_state}) = ra$

PWRITE_REVERSE_BIT $x((r, s, m, rb, ra) : \mathsf{ppc\_state}) = ((r, s, m, x, ra) : \mathsf{ppc\_state})$

PWRITE_REVERSE_ADDRESS $x((r, s, m, rb, ra) : \mathsf{ppc\_state}) = ((r, s, m, rb, x) : \mathsf{ppc\_state})$

type_abbrev M : ppc_state → ('a#ppc_state) option

(constT_seq : 'a → 'a M)$x = \lambda y.\textsc{Some}\ (x, y)$

(addT_seq : 'a → 'b M → ('a#'b)M)$x\ s =$
$\lambda y.\mathbf{case}\ s\ y\ \mathbf{of}\ \textsc{None} \to \textsc{None} \parallel \textsc{Some}\ (z, t) \to \textsc{Some}\ ((x, z), t)$

(lockT_seq : 'a M → 'a M)$s = s$

(syncT_seq : $iiid$ → unit M)x = constT_seq()

(failureT_seq : 'a M) $= \lambda y.\textsc{None}$

(seqT_seq : 'a M → ('a → 'b M) → 'b M)$s\ f =$
$\lambda y.\mathbf{case}\ s\ y\ \mathbf{of}\ \textsc{None} \to \textsc{None} \parallel \textsc{Some}\ (z, t) \to f\ z\ t$

$(\text{parT\_seq} : {'a}\ \mathsf{M} \to {'b}\ \mathsf{M} \to ({'a}\#{'b})\mathsf{M})s\ t =$
$\lambda y.\mathbf{case}\ s\ y\ \mathbf{of}\ \textsc{None} \to \textsc{None} \parallel \textsc{Some}\ (a, z) \to$
$\quad \mathbf{case}\ t\ z\ \mathbf{of}\ \textsc{None} \to \textsc{None} \parallel \textsc{Some}\ (b, x) \to \textsc{Some}\ ((a, b), x)$

$(\text{parT\_unit\_seq} : \mathsf{unit}\ \mathsf{M} \to \mathsf{unit}\ \mathsf{M} \to \mathsf{unit}\ \mathsf{M})s\ t =$
$\lambda y.\mathbf{case}\ s\ y\ \mathbf{of}\ \textsc{None} \to \textsc{None} \parallel \textsc{Some}\ (a, z) \to$
$\quad \mathbf{case}\ t\ z\ \mathbf{of}\ \textsc{None} \to \textsc{None} \parallel \textsc{Some}\ (b, x) \to \textsc{Some}\ ((), x)$

$(\text{write\_reg\_seq}\ ii\ r\ x) : \mathsf{unit}\ \mathsf{M} =$
$\lambda s.\textsc{Some}\ ((), \text{PWRITE\_R}\ r\ x\ s)$

$(\text{read\_reg\_seq}\ ii\ r) : \mathsf{word32}\ \mathsf{M} =$
$\lambda s.\textsc{Some}\ (\text{PREAD\_R}\ r\ s, s)$

$(\text{write\_status\_seq}\ ii\ f\ x) : \mathsf{unit}\ \mathsf{M} =$
$(\lambda s.\textsc{Some}\ ((), \text{PWRITE\_S}\ f\ x\ s))$

$(\text{read\_status\_seq}\ ii\ f) : \mathsf{bool}\ \mathsf{M} =$
$(\lambda s.\mathbf{case}\ \ \text{PREAD\_S}\ f\ s\ \mathbf{of}\ \textsc{None} \to \textsc{None} \parallel \textsc{Some}\ \ b \to \textsc{Some}\ (b, s))$

$(\text{write\_mem8\_seq}\ ii\ a\ x) : \mathsf{unit}\ \mathsf{M} =$
$(\lambda s.\mathbf{case}\ \ \text{PREAD\_M}\ a\ s\ \mathbf{of}\ \textsc{None} \to \textsc{None} \parallel \textsc{Some}\ \ y \to \textsc{Some}\ ((), \text{PWRITE\_M}\ a(\textsc{Some}\ \ x)s))$

$(\text{read\_mem8\_seq}\ ii\ a) : \mathsf{word8}\ \mathsf{M} =$
$(\lambda s.\mathbf{case}\ \ \text{PREAD\_M}\ a\ s\ \mathbf{of}\ \textsc{None} \to \textsc{None} \parallel \textsc{Some}\ \ x \to \textsc{Some}\ (x, s))$

$(\text{read\_mem32\_seq}\ ii\ a) : \mathsf{word32}\ \mathsf{M} =$
$\text{seqT\_seq}(\text{parT\_seq}(\text{read\_mem8\_seq}\ ii(a + 0w))(\text{parT\_seq}(\text{read\_mem8\_seq}\ ii(a + 1w))$
$\qquad (\text{parT\_seq}(\text{read\_mem8\_seq}\ ii(a + 2w))(\text{read\_mem8\_seq}\ ii(a + 3w)))))$
$\quad (\lambda(x0, x1, x2, x3).\ \text{constT\_seq}(bytes2word[x0; x1; x2; x3]))$

$(\text{write\_mem32\_seq}\ ii\ a\ w) : \mathsf{unit}\ \mathsf{M} =$
$(\mathbf{let}\ \ bs = word2bytes\ 4\ w\ \ \mathbf{in}$
$\quad \text{parT\_unit\_seq}(\text{write\_mem8\_seq}\ ii(a + 0w)(EL\ 0\ bs))(\text{parT\_unit\_seq}(\text{write\_mem8\_seq}\ ii(a + 1w)(EL\ 1\ bs))$
$(\text{parT\_unit\_seq}(\text{write\_mem8\_seq}\ ii(a + 2w)(EL\ 2\ bs))(\text{write\_mem8\_seq}\ ii(a + 3w)(EL\ 3\ bs)))))$

$(\text{write\_reserve\_bit\_seq}(ii : iiid)x) : \mathsf{unit}\ \mathsf{M} =$
$\lambda s.\textsc{Some}\ ((), \text{PWRITE\_REVERSE\_BIT}\ x\ s)$

$(\text{read\_reserve\_bit\_seq}(ii : iiid)) : \mathsf{bool}\ \mathsf{M} =$
$\lambda s.\textsc{Some}\ (\text{PREAD\_REVERSE\_BIT}\ s, s)$

(write_reserve_address_seq(ii : $iiid$)x) : unit M =
λs.SOME ((), PWRITE_REVERSE_ADDRESS x s)

(read_reserve_address_seq(ii : $iiid$)) : word32 M =
λs.SOME (PREAD_REVERSE_ADDRESS s, s)

(constT : $'a \to 'a$ M) = constT_seq

(addT : $'a \to 'b$ M $\to$ ($'a$#$'b$)M) = addT_seq

(lockT : unit M $\to$ unit M) = lockT_seq

(syncT : $iiid$ $\to$ unit M) = syncT_seq

(failureT : unit M) = failureT_seq

(control_seqT : $'a$ M $\to$ (($'a \to 'b$ M) $\to 'b$ M)) = seqT_seq

(seqT : $'a$ M $\to$ (($'a \to 'b$ M) $\to 'b$ M)) = seqT_seq

(parT : $'a$ M $\to 'b$ M $\to$ ($'a$#$'b$)M) = parT_seq

(parT_unit : unit M $\to$ unit M $\to$ unit M) = parT_unit_seq

(write_reg : $iiid$ $\to$ ppc_reg32 $\to$ word32 $\to$ unit M) = write_reg_seq

(read_reg : $iiid$ $\to$ ppc_reg32 $\to$ word32 M) = read_reg_seq

(write_status : $iiid$ $\to$ ppc_bit $\to$ bool option $\to$ unit M) = write_status_seq

(read_status : $iiid$ $\to$ ppc_bit $\to$ bool M) = read_status_seq

(write_mem8 : $iiid$ $\to$ word32 $\to$ word8 $\to$ unit M) = write_mem8_seq

(read_mem8 : $iiid$ $\to$ word32 $\to$ word8 M) = read_mem8_seq

(write_mem32 : $iiid$ $\to$ word32 $\to$ word32 $\to$ unit M) = write_mem32_seq

(read_mem32 : $iiid$ $\to$ word32 $\to$ word32 M) = read_mem32_seq

(write_reserve_bit : $iiid$ $\to$ bool $\to$ unit M) = write_reserve_bit_seq

(read_reserve_bit : $iiid$ $\to$ bool M) = read_reserve_bit_seq

(write_reserve_address : $iiid$ $\to$ word32 $\to$ unit M) = write_reserve_address_seq

(read_reserve_address : $iiid$ $\to$ word32 M) = read_reserve_address_seq

option_apply x f = **if** x = NONE **then** NONE **else** f(**the** x)

Part XIX

ppc_event_monad

type_abbrev M : eiid_state → ((eiid_state#′a#(ppc_reg event_structure))set)

event_structure_empty =⟪ *events* :={}; *intra_causality_data* :={}; *intra_causality_control* :={}; *atomicity* :={}⟫

event_structure_lock *es* =⟪ *events* := *es*.*events*; *intra_causality_data* := *es*.*intra_causality_data*; *intra_causality_contro*

event_structure_union es_1 es_2 =
⟪ *events* := es_1.*events* ∪ es_2.*events*;
intra_causality_data := es_1.*intra_causality_data* ∪ es_2.*intra_causality_data*;
intra_causality_control := es_1.*intra_causality_control* ∪ es_2.*intra_causality_control*;
atomicity := es_1.*atomicity* ∪ es_2.*atomicity*⟫

event_structure_bigunion(*ess* : (ppc_reg event_structure)set) =
⟪ *events* := **bigunion**{*es*.*events* | *es* ∈ *ess*};
intra_causality_data := **bigunion**{*es*.*intra_causality_data* | *es* ∈ *ess*};
intra_causality_control := **bigunion**{*es*.*intra_causality_control* | *es* ∈ *ess*};
atomicity := **bigunion**{*es*.*atomicity* | *es* ∈ *ess*}⟫

event_structure_seq_union es_1 es_2 =
⟪ *events* := es_1.*events* ∪ es_2.*events*;
intra_causality_data := es_1.*intra_causality_data*
∪ es_2.*intra_causality_data*
∪ {(e_1, e_2)
| e_1 ∈ (maximal_elements es_1.*events* es_1.*intra_causality_data*)
∧ e_2 ∈ (*minimal_elements* es_2.*events* es_2.*intra_causality_data*)};
intra_causality_control := es_1.*intra_causality_control*
∪ es_2.*intra_causality_control*;
atomicity := es_1.*atomicity* ∪ es_2.*atomicity*⟫

event_structure_control_seq_union es_1 es_2 =
⟪ *events* := es_1.*events* ∪ es_2.*events*;
intra_causality_data := es_1.*intra_causality_data*
∪ es_2.*intra_causality_data*;
intra_causality_control := es_1.*intra_causality_control*
∪ es_2.*intra_causality_control*
∪ {(e_1, e_2)
| e_1 ∈ (maximal_elements es_1.*events* es_1.*intra_causality_control*)
∧ e_2 ∈ (*minimal_elements* es_2.*events* es_2.*intra_causality_control*)};
atomicity := es_1.*atomicity* ∪ es_2.*atomicity*⟫

(mapT_ev : (′a → ′b) → ′a M → ′b M)*f* *s* =
λ*eiid_next* : eiid_state.
let *t* = *s* *eiid_next* **in**
{(*eiid_next*′, *f* *x*, *es*)
| (*eiid_next*′, *x*, *es*) ∈ *t*}

```
(choiceT_ev : 'a M → 'a M → 'a M)s s' =
λeiid_next : eiid_state.s eiid_next ∪ s' eiid_next

(constT_ev : 'a → 'a M)x = λeiid_next.{(eiid_next, x, event_structure_empty)}

(addT_ev : 'a → 'b M → ('a#'b)M)x s =
λeiid_next.let (t : (eiid_state#'b#(ppc_reg event_structure))set) = s eiid_next in
  image(λ(eiid_next', v, es).(eiid_next', (x, v), es))t

(lockT_ev : 'a M → 'a M)s =
λeiid_next.let (t : (eiid_state#'a#(ppc_reg event_structure))set) = s eiid_next in
  image(λ(eiid_next', v, es).(eiid_next', v, event_structure_lock es))t

(failureT_ev : 'a M) = λeiid_next.{}

(condT_ev : bool → unit M → unit M)b s =
if b then s else constT_ev()

(seqT_ev : 'a M → ('a → 'b M) → 'b M)s f =
λeiid_next : eiid_state.
let t = s eiid_next in
  bigunion{let t' = f x eiid_next' in
                {(eiid_next'', x', event_structure_seq_union es es')
                 | (eiid_next'', x', es') ∈ t'}
           | (eiid_next', x, es) ∈ t}

(control_seqT_ev : 'a M → ('a → 'b M) → 'b M)s f =
λeiid_next : eiid_state.
let t = s eiid_next in
  bigunion{let t' = f x eiid_next' in
                {(eiid_next'', x', event_structure_control_seq_union es es')
                 | (eiid_next'', x', es') ∈ t'}
           | (eiid_next', x, es) ∈ t}

(parT_ev : 'a M → 'b M → ('a#'b)M)s s' =
λeiid_next : eiid_state.
let t = s eiid_next in
  bigunion{let t' = s' eiid_next' in
                {(eiid_next'', (x, x'), event_structure_union es es')
                 | (eiid_next'', x', es') ∈ t'}
           | (eiid_next', x, es) ∈ t}
```

```
(parT_unit_ev : unit M → unit M → unit M)s s′ =
λeiid_next : eiid_state.
let t = s eiid_next in
  bigunion{let t′ = s′ eiid_next′ in
                {(eiid_next″, (), event_structure_union es es′)
                 | (eiid_next″, (), es′) ∈ t′}
           | (eiid_next′, (), es) ∈ t}


(write_location_ev ii l x) : unit M =
λeiid_next.{(eiid_next′,
             (),
             ⦇ events :={⦇ eiid := eiid′;
                           iiid := ii;
                           action := ACCESS W l x⦈};
               intra_causality_data :={};
               intra_causality_control :={};
               atomicity :={}⦈) | (eiid′, eiid_next′) ∈ next_eiid eiid_next}


(read_location_ev ii l) : value M =
λeiid_next.{(eiid_next′,
             x,
             ⦇ events :={⦇ eiid := eiid′;
                           iiid := ii;
                           action := ACCESS R l x⦈};
               intra_causality_data :={};
               intra_causality_control :={};
               atomicity :={}⦈)
          | x ∈ UNIV ∧ (eiid′, eiid_next′) ∈ next_eiid eiid_next}


(syncT_ev ii) : unit M =
λeiid_next.{(eiid_next′,
             (),
             ⦇ events :={⦇ eiid := eiid′;
                           iiid := ii;
                           action := BARRIER SYNC⦈};
               intra_causality_data :={};
               intra_causality_control :={};
               atomicity :={}⦈) | (eiid′, eiid_next′) ∈ next_eiid eiid_next}


(write_reg_ev ii r x) : unit M =
write_location_ev ii(LOCATION_REG ii. proc(REG32 r))x


(read_reg_ev ii r) : value M =
read_location_ev ii(LOCATION_REG ii. proc(REG32 r))
```

```
(write_status_ev ii f x) : unit M =
case x of
SOME b → write_location_ev ii(LOCATION_REG ii. proc(REGBIT f))(if b then 1w else 0w)
 || NONE → choiceT_ev(write_location_ev ii(LOCATION_REG ii. proc(REGBIT f))1w)
                      (write_location_ev ii(LOCATION_REG ii. proc(REGBIT f))0w)

(read_status_ev ii f) : bool M =
seqT_ev(read_location_ev ii(LOCATION_REG ii. proc(REGBIT f)))(λw. constT_ev(w[0]))

aligned32 a = ((a&&&3w) = 0w)

(write_mem8_ev ii a x) = failureT_ev

(read_mem8_ev ii a) = failureT_ev

(write_mem32_ev ii a(x : word32)) : unit M =
if aligned32 a then
write_location_ev ii(LOCATION_MEM a)x
else
failureT_ev

(read_mem32_ev ii a) : word32 M =
if aligned32 a then
read_location_ev ii(LOCATION_MEM a)
else
failureT_ev

(write_reserve_bit_ev(ii : iiid)x) : unit M =
write_location_ev ii(LOCATION_RES ii. proc)(if x then 1w else 0w)

(read_reserve_bit_ev(ii : iiid)) : bool M =
seqT_ev(read_location_ev ii(LOCATION_RES ii. proc))(λw. constT_ev(w[0]))

(write_reserve_address_ev(ii : iiid)x) : unit M =
write_location_ev ii(LOCATION_RES_ADDR ii. proc)x

(read_reserve_address_ev(ii : iiid)) : word32 M =
read_location_ev ii(LOCATION_RES_ADDR ii. proc)

(constT : 'a → 'a M) = constT_ev

(addT : 'a → 'b M → ('a#'b)M) = addT_ev
```

(lockT : unit M → unit M) = lockT_ev

(syncT : *iiid* → unit M) = syncT_ev

(failureT : unit M) = failureT_ev

(seqT : $'a$ M → (($'a$ → $'b$ M) → $'b$ M)) = seqT_ev

(control_seqT : $'a$ M → (($'a$ → $'b$ M) → $'b$ M)) = control_seqT_ev

(parT : $'a$ M → $'b$ M → ($'a$#$'b$)M) = parT_ev

(parT_unit : unit M → unit M → unit M) = parT_unit_ev

(write_reg : *iiid* → ppc_reg32 → word32 → unit M) = write_reg_ev

(read_reg : *iiid* → ppc_reg32 → word32 M) = read_reg_ev

(write_status : *iiid* → ppc_bit → bool option → unit M) = write_status_ev

(read_status : *iiid* → ppc_bit → bool M) = read_status_ev

(write_mem8 : *iiid* → word32 → word8 → unit M) = write_mem8_ev

(read_mem8 : *iiid* → word32 → word8 M) = read_mem8_ev

(write_mem32 : *iiid* → word32 → word32 → unit M) = write_mem32_ev

(read_mem32 : *iiid* → word32 → word32 M) = read_mem32_ev

(write_reserve_bit : *iiid* → bool → unit M) = write_reserve_bit_ev

(read_reserve_bit : *iiid* → bool M) = read_reserve_bit_ev

(write_reserve_address : *iiid* → word32 → unit M) = write_reserve_address_ev

(read_reserve_address : *iiid* → word32 M) = read_reserve_address_ev

Part XX

ppc_program

type_abbrev program_Pinstruction : (address → (Pinstruction#num) option)

(ppc_decode_program_fun : program_word8 → address → (Pinstruction#num) option)*prog_word8 a* =
if $(a\&\&3w = 0w)$ **then** NONE
else
let *w0* = *prog_word8*$(a + 0w)$ **in**
let *w1* = *prog_word8*$(a + 1w)$ **in**
let *w2* = *prog_word8*$(a + 2w)$ **in**
let *w3* = *prog_word8*$(a + 3w)$ **in**
if mem NONE $[w0; w1; w2; w3]$ **then** NONE **else**
let *raw_instruction* = *w2bits*(**the** *w0*) ++ *w2bits*(**the** *w1*) ++ *w2bits*(**the** *w2*) ++ *w2bits*(**the** *w3*) **in**
let *io* = ppc_decode *raw_instruction* **in**
case *io* **of**
NONE → NONE
‖ SOME *i* → SOME $(i, 4)$

(ppc_decode_program_rel : program_word8 → program_Pinstruction → bool)
prog_word8 prog_Xinst =
$\forall a$.**case** *prog_Xinst a* **of**
NONE → **T**
‖ SOME $(inst, n)$ → ppc_decode_program_fun *prog_word8 a* = SOME $(inst, n)$

ppc_event_execute = *ppc_event_opsem* $*ppc_exec_instr*

ppc_execute_with_pc_check *ii inst pc* =
let *s* = (ppc_event_execute *ii inst*){} **in**
{*E*
| $\exists$*eiid_next x*.
$(eiid\_next, x, E) \in s \wedge$
$\forall v\ ev.((ev.action = (\text{ACCESS R}(\text{LOCATION\_REG}\ ii.\,\text{proc}(\text{REG32 PPC\_PC}))v)) \wedge$
$ev \in E.events) \implies (v = pc)$
}

(ppc_event_structures_of_run_skeleton : program_Pinstruction → run_skeleton → (ppc_reg event_structure)set)
prog_Pinstruction rs =
let *Ess* = {ppc_execute_with_pc_check⟦ proc := *p*; *poi* := *i*⟧*inst pc* | $\exists n$.
(*rs p i* = SOME *pc*) ∧ (SOME $(inst, n)$ = *prog_Pinstruction pc*)}
in
{event_structure_bigunion *Es* | *Es* ∈ *all_choices Ess*}

(ppc_semantics : program_word8 → (ppc_reg state_constraint) →
(run_skeleton#program_Pinstruction#(((ppc_reg event_structure)#((ppc_reg execution_witness)set))set))set)
prog_word8 initial_state =

let *x1* = {$(rs, prog\_Xinst)$ | *rs*, *prog_Xinst* | run_skeleton_wf(*DOMAIN prog_Xinst*)*rs* ∧
ppc_decode_program_rel *prog_word8 prog_Xinst*} **in**

let $x2 = \{(rs, prog\_Xinst, Es) \mid (rs, prog\_Xinst) \in x1 \wedge$
$(Es = \text{ppc\_event\_structures\_of\_run\_skeleton } prog\_Xinst\ rs)\}$ **in**

let $x3 = \{(rs, prog\_Xinst, \{(E, Xs) \mid E \in Es \wedge$
$(Xs = \{X \mid \text{valid\_execution } E\ X \wedge$
$(X.initial\_state = initial\_state)\})\})$
$\mid (rs, prog\_Xinst, Es) \in x2\}$ **in**
$x3$

Index