```
(*=======================================================================*)
(*                                                                       *)
(*  Copyright (c) 2015-2016 Robert M. Norton                             *)
(*  Copyright (c) 2015-2016 Kathyrn Gray                                 *)
(*  All rights reserved.                                                 *)
(*                                                                       *)
(*  This software was developed by the University of Cambridge Computer  *)
(*  Laboratory as part of the Rigorous Engineering of Mainstream Systems *)
(*  (REMS) project, funded by EPSRC grant EP/K008528/1.                  *)
(*                                                                       *)
(*  Redistribution and use in source and binary forms, with or without   *)
(*  modification, are permitted provided that the following conditions   *)
(*  are met:                                                             *)
(*  1. Redistributions of source code must retain the above copyright    *)
(*     notice, this list of conditions and the following disclaimer.     *)
(*  2. Redistributions in binary form must reproduce the above copyright *)
(*     notice, this list of conditions and the following disclaimer in   *)
(*     the documentation and/or other materials provided with the        *)
(*     distribution.                                                     *)
(*                                                                       *)
(*  THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ''AS IS''   *)
(*  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED    *)
(*  TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A      *)
(*  PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR  *)
(*  CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,         *)
(*  SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT     *)
(*  LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF     *)
(*  USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND  *)
(*  ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,   *)
(*  OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT   *)
(*  OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF   *)
(*  SUCH DAMAGE.                                                         *)
(*=======================================================================*)

(* mips_prelude.sail: declarations of mips registers, and functions common
   to mips instructions (e.g. address translation) *)

(* bit vectors have indices decreasing from left i.e. msb is highest index *)
default Order dec

register (bit[64]) PC
register (bit[64]) nextPC

(* CP0 Registers *)

typedef CauseReg = register bits [ 31 : 0 ] {
  31       : BD; (* branch delay *)
  (*30       : Z0;*)
  29 .. 28 : CE; (* for coprocessor enable exception *)
  (*27 .. 24 : Z1;*)
  23       : IV; (* special interrupt vector not supported *)
  22       : WP; (* watchpoint exception occurred *)
  (*21 .. 16 : Z2; *)
  15 .. 8  : IP; (* interrupt pending bits *)
```

```
  (*7        : Z3;*)
  6 .. 2   : ExcCode; (* code of last exception *)
  (*1 .. 0   : Z4;*)
}

typedef TLBEntryLoReg = register bits [63 : 0] {
  63      : CapS;
  62      : CapL;
  29 .. 6 : PFN;
  5  .. 3 : C;
  2       : D;
  1       : V;
  0       : G;
}

typedef TLBEntryHiReg = register bits [63 : 0] {
  63 .. 62 : R;
  39 .. 13 : VPN2;
  7  ..  0 : ASID;
}

typedef ContextReg = register bits [63 : 0] {
  63 .. 23 : PTEBase;
  22 .. 4  : BadVPN2;
  (*3  .. 0  : ZR;*)
}

typedef XContextReg = register bits [63 : 0] {
  63 .. 33: PTEBase;
  32 .. 31: R;
  30 ..  4: BadVPN2;
}

let ([:8:]) TLBNumEntries = 8
typedef TLBIndexT = (bit[3])
let (TLBIndexT) TLBIndexMax = 0b111

let MAX_VA = unsigned(0xfffffffff)
let MAX_PA = unsigned(0xfffffffff)

typedef TLBEntry = register bits [116 : 0] {
  116 ..  101: pagemask;
  100 ..  99 : r       ;
  98  ..  72 : vpn2    ;
  71  ..  64 : asid    ;
  63         : g       ;
  62         : valid   ;
  61         : caps1   ;
  60         : capl1   ;
  59  ..  36 : pfn1    ;
  35  ..  33 : c1      ;
  32         : d1      ;
  31         : v1      ;
  30         : caps0   ;
```

```
  29         : capl0   ;
  28  ..  5  : pfn0    ;
  4   ..  2  : c0      ;
  1          : d0      ;
  0          : v0      ;
}

register (bit[1])        TLBProbe
register (bit[3])     TLBIndex
register (bit[3])     TLBRandom
register (TLBEntryLoReg) TLBEntryLo0
register (TLBEntryLoReg) TLBEntryLo1
register (ContextReg)    TLBContext
register (bit[16])       TLBPageMask
register (bit[3])     TLBWired
register (TLBEntryHiReg) TLBEntryHi
register (XContextReg)   TLBXContext

register (TLBEntry) TLBEntry00
register (TLBEntry) TLBEntry01
register (TLBEntry) TLBEntry02
register (TLBEntry) TLBEntry03
register (TLBEntry) TLBEntry04
register (TLBEntry) TLBEntry05
register (TLBEntry) TLBEntry06
register (TLBEntry) TLBEntry07

let (vector <0, 8, inc, (TLBEntry)>) TLBEntries = [
TLBEntry00,
TLBEntry01,
TLBEntry02,
TLBEntry03,
TLBEntry04,
TLBEntry05,
TLBEntry06,
TLBEntry07
]

register (bit[32])  CP0Compare
register (CauseReg) CP0Cause
register (bit[64])  CP0EPC
register (bit[64])  CP0ErrorEPC
register (bit[1])   CP0LLBit
register (bit[64])  CP0LLAddr
register (bit[64])  CP0BadVAddr
register (bit[32])  CP0Count
register (bit[32])  CP0Compare
register (bit[32])  CP0HWREna
register (bit[64])  CP0UserLocal

typedef StatusReg = register bits [31:0] {
  31 .. 28        : CU;  (* co-processor enable bits *)
  (* RP/FR/RE/MX/PX not implemented *)
  22              : BEV; (* use boot exception vectors (initialised to one) *)
```

```
  (* TS/SR/NMI not implemented *)
  15 .. 8         : IM;  (* Interrupt mask *)
  7               : KX;  (* kernel 64-bit enable *)
  6               : SX;  (* supervisor 64-bit enable *)
  5               : UX;  (* user 64-bit enable *)
  4 .. 3          : KSU; (* Processor mode *)
  2               : ERL; (* error level (should be initialised to one, but BERI is different) *)
  1               : EXL; (* exception level *)
  0               : IE;  (* interrupt enable *)
}
register (StatusReg) CP0Status

(* Implementation registers -- not ISA defined *)
register (bit[1])  branchPending      (* Set by branch instructions to implement branch delay *)
register (bit[1])  inBranchDelay      (* Needs to be set by outside world when in branch delay slot *)
register (bit[64]) delayedPC          (* Set by branch instructions to implement branch delay *)

(* General purpose registers *)

register (bit[64]) GPR00 (* should never be read or written *)
register (bit[64]) GPR01
register (bit[64]) GPR02
register (bit[64]) GPR03
register (bit[64]) GPR04
register (bit[64]) GPR05
register (bit[64]) GPR06
register (bit[64]) GPR07
register (bit[64]) GPR08
register (bit[64]) GPR09
register (bit[64]) GPR10
register (bit[64]) GPR11
register (bit[64]) GPR12
register (bit[64]) GPR13
register (bit[64]) GPR14
register (bit[64]) GPR15
register (bit[64]) GPR16
register (bit[64]) GPR17
register (bit[64]) GPR18
register (bit[64]) GPR19
register (bit[64]) GPR20
register (bit[64]) GPR21
register (bit[64]) GPR22
register (bit[64]) GPR23
register (bit[64]) GPR24
register (bit[64]) GPR25
register (bit[64]) GPR26
register (bit[64]) GPR27
register (bit[64]) GPR28
register (bit[64]) GPR29
register (bit[64]) GPR30
register (bit[64]) GPR31

(* Special registers For MUL/DIV *)
register (bit[64]) HI
```

```
register (bit[64]) LO

let (vector <0, 32, inc, (register<(bit[64])>) >) GPR =
  [ GPR00, GPR01, GPR02, GPR03, GPR04, GPR05, GPR06, GPR07, GPR08, GPR09, GPR10,
  GPR11, GPR12, GPR13, GPR14, GPR15, GPR16, GPR17, GPR18, GPR19, GPR20,
  GPR21, GPR22, GPR23, GPR24, GPR25, GPR26, GPR27, GPR28, GPR29, GPR30, GPR31
  ]

(* Check whether a given 64-bit vector is a properly sign extended 32-bit word *)
val bit[64] -> bool effect pure NotWordVal
function bool NotWordVal (word) =
    (word[31] ^^ 32) != word[63..32]

(* Read numbered GP reg. (r0 is always zero) *)
val bit[5] -> bit[64] effect {rreg} rGPR
function bit[64] rGPR idx = {
         if idx == 0 then 0 else GPR[idx]
}

(* Write numbered GP reg. (writes to r0 ignored) *)
val (bit[5], bit[64]) -> unit effect {wreg} wGPR
function unit wGPR (idx, v) = {
         if idx == 0 then () else GPR[idx] := v
}

val extern forall Nat 'n. ( bit[64] , [|'n|] , bit[8*'n]) -> unit effect { wmem } MEMw
val extern forall Nat 'n. ( bit[64] , [|'n|] ) -> (bit[8 * 'n]) effect { rmem } MEMr
val extern forall Nat 'n. ( bit[64] , [|'n|] ) -> (bit[8 * 'n]) effect { rmem } MEMr_reserve
val extern forall Nat 'n. ( bit[64] , [|'n|] , bit[8*'n]) -> bool effect { wmem } MEMw_conditional
val extern unit -> unit effect { barr } MEM_sync

typedef Exception = enumerate
{
   Int; TLBMod; TLBL; TLBS; AdEL; AdES; Sys; Bp; ResI; CpU; Ov; Tr; C2E; C2Trap;
   XTLBRefillL; XTLBRefillS; XTLBInvL; XTLBInvS; MCheck
}

(* Return the ISA defined code for an exception condition *)
function (bit[5]) ExceptionCode ((Exception) ex) =
   switch (ex)
   {
      case Int         -> mask(0x00) (* Interrupt *)
      case TLBMod      -> mask(0x01) (* TLB modification exception *)
      case TLBL        -> mask(0x02) (* TLB exception (load or fetch) *)
      case TLBS        -> mask(0x03) (* TLB exception (store) *)
      case AdEL        -> mask(0x04) (* Address error (load or fetch) *)
      case AdES        -> mask(0x05) (* Address error (store) *)
      case Sys         -> mask(0x08) (* Syscall *)
      case Bp          -> mask(0x09) (* Breakpoint *)
      case ResI        -> mask(0x0a) (* Reserved instruction *)
      case CpU         -> mask(0x0b) (* Coprocessor Unusable *)
      case Ov          -> mask(0x0c) (* Arithmetic overflow *)
      case Tr          -> mask(0x0d) (* Trap *)
      case C2E         -> mask(0x12) (* C2E coprocessor 2 exception *)
```

```
      case C2Trap      -> mask(0x12) (* C2Trap maps to same exception code, different vector *)
      case XTLBRefillL -> mask(0x02)
      case XTLBRefillS -> mask(0x03)
      case XTLBInvL    -> mask(0x02)
      case XTLBInvS    -> mask(0x03)
      case MCheck      -> mask(0x18)
   }


val Exception -> unit effect {rreg, wreg} SignalException

function unit SignalExceptionMIPS ((Exception) ex) =
  {
    (* Only update EPC and BD if not already in EXL mode *)
    if (~ (CP0Status.EXL)) then
      {
        if (inBranchDelay[0]) then
          {
            CP0EPC := PC - 4;
            CP0Cause.BD := 1;
          }
        else
          {
            CP0EPC := PC;
            CP0Cause.BD := 0;
          }
      };

      (* choose an exception vector to branch to. Some are not supported
         e.g. Reset *)
      vectorOffset :=
        if (CP0Status.EXL) then
          0x180 (* Always use common vector if in exception mode already *)
        else if ((ex == XTLBRefillL) | (ex == XTLBRefillS)) then
          0x080
        else if (ex == C2Trap) then
          0x280 (* Special vector for CHERI traps *)
        else
          0x180; (* Common vector *)
      (bit[64]) vectorBase  := if CP0Status.BEV then
                    0xFFFFFFFFBFC00200
                 else
                    0xFFFFFFFF80000000;
      nextPC := vectorBase + EXTS(vectorOffset);
      CP0Cause.ExcCode      := ExceptionCode(ex);
      CP0Status.EXL         := 1;
  }

function unit SignalExceptionBadAddr((Exception) ex, (bit[64]) badAddr) =
  {
    CP0BadVAddr := badAddr;
    SignalException(ex);
  }
```

```
function unit SignalExceptionTLB((Exception) ex, (bit[64]) badAddr) = {
  CP0BadVAddr := badAddr;
  (TLBContext.BadVPN2) := (badAddr[31..13]);
  (TLBXContext.BadVPN2):= (badAddr[39..13]);
  (TLBXContext.R)      := (badAddr[63..62]);
  (TLBEntryHi.R)       := (badAddr[63..62]);
  (TLBEntryHi.VPN2)    := (badAddr[39..13]);
  SignalException(ex);
}

typedef MemAccessType = enumerate {Instruction; LoadData; StoreData}
typedef AccessLevel   = enumerate {User; Supervisor; Kernel}

function AccessLevel getAccessLevel() =
  if ((CP0Status.EXL) | (CP0Status.ERL)) then
    Kernel
  else switch (CP0Status.KSU)
    {
      case 0b00 -> Kernel
      case 0b01 -> Supervisor
      case 0b10 -> User
      case _    -> User (* behaviour undefined, assume user *)
    }

function unit checkCP0Access () =
  {
    let accessLevel = getAccessLevel() in
    if ((accessLevel != Kernel) & (~((CP0Status.CU)[28]))) then
      {
        (CP0Cause.CE) := 0b00;
        exit (SignalException(CpU));
      }
  }

function unit incrementCP0Count() = {
  TLBRandom := (if (unsigned(TLBRandom) == unsigned(TLBWired))
    then (TLBIndexMax) else (TLBRandom - 1));

  CP0Count := (CP0Count + 1);
  if (unsigned(CP0Count) == unsigned(CP0Compare)) then {
    (CP0Cause[15]) := bitone;
  };
  (* XXX Sail does not allow reading fields here :-( *)
  let (bit[32])status = CP0Status in
  let (bit[32])cause  = CP0Cause in
  let (bit[8]) ims = status[15..8] in
  let (bit[8]) ips = cause[15..8] in
  let ie           = status[0] in
  let exl          = status[1] in
  let erl          = status[2] in
  if ((~(exl)) & (~(erl)) & ie & ((ips & ims) != 0x00)) then
    exit (SignalException(Int));
}
```

```
function bool tlbEntryMatch(r, vpn2, asid, (TLBEntry) entry) =
      let entryVal   = (bit[117]) entry in
      let entryValid = entryVal[62] in
      let entryR     = entryVal[100..99] in
      let entryMask  = entryVal[116..101] in
      let entryVPN   = entryVal[98..72] in
      let entryASID  = entryVal[71..64] in
      let entryG     = entryVal[63] in
      (entryValid &
       (r == entryR) &
       ((vpn2 & ~(EXTZ(entryMask))) == ((entryVPN) & ~(EXTZ(entryMask)))) &
       ((asid == (entryASID)) | (entryG)))

function option<TLBIndexT> tlbSearch((bit[64]) VAddr) =
  let r    = (VAddr[63..62]) in
  let vpn2 = (VAddr[39..13]) in
  let asid = (((bit[64])TLBEntryHi)[7..0]) in (* XXX workaround sail bug *)
  if (tlbEntryMatch(r, vpn2, asid, TLBEntry00)) then
    Some(0b000)
  else if (tlbEntryMatch(r, vpn2, asid, TLBEntry01)) then
    Some(0b001)
  else if (tlbEntryMatch(r, vpn2, asid, TLBEntry02)) then
    Some(0b010)
  else if (tlbEntryMatch(r, vpn2, asid, TLBEntry03)) then
    Some(0b011)
  else if (tlbEntryMatch(r, vpn2, asid, TLBEntry04)) then
    Some(0b100)
  else if (tlbEntryMatch(r, vpn2, asid, TLBEntry05)) then
    Some(0b101)
  else if (tlbEntryMatch(r, vpn2, asid, TLBEntry06)) then
    Some(0b110)
  else if (tlbEntryMatch(r, vpn2, asid, TLBEntry07)) then
    Some(0b111)
  else
    None

function (bit[64], bool) TLBTranslate2 ((bit[64]) vAddr, (MemAccessType) accessType) = {
  let idx = tlbSearch(vAddr) in
  switch(idx) {
    case (Some(idx)) ->
    let entry = ((bit[117])(TLBEntries[idx])) in
    let entryMask = (entry[116..101]) in

    let evenOddBit = switch(entryMask) {
      case 0x0000 -> 12
      case 0x0003 -> 14
      case 0x000f -> 16
      case 0x003f -> 18
      case 0x00ff -> 20
      case 0x03ff -> 22
      case 0x0fff -> 24
      case 0x3fff -> 26
      case 0xffff -> 28
      case _      -> undefined
```

```
    } in
    let isOdd = (vAddr[evenOddBit]) in
    let (caps, capl, pfn, d, v) = if (isOdd) then
        (entry[61], entry[60], entry[59..36], entry[32], entry[31])
      else
        (entry[30], entry[29], entry[28..5], entry[1], entry[0]) in
    if (~(v)) then
      exit (SignalExceptionTLB(if (accessType == StoreData) then XTLBInvS else XTLBInvL, vAddr))
    else if ((accessType == StoreData) & ~(d)) then
      exit (SignalExceptionTLB(TLBMod, vAddr))
    else
      (EXTZ(pfn[23..((evenOddBit)-(12))] : vAddr[((evenOddBit)-(1)) .. 0]),
       if (accessType == StoreData) then caps else capl)
    case None -> exit (SignalExceptionTLB(
      if (accessType == StoreData) then XTLBRefillS else XTLBRefillL, vAddr))
  }
}

function (bit[64], bool) TLBTranslateC ((bit[64]) vAddr, (MemAccessType) accessType) =
  {
    let currentAccessLevel = getAccessLevel() in
    let compat32 = (vAddr[61..31] == 0b11111111111111111111111111111111) in
    let (requiredLevel, addr) = switch(vAddr[63..62]) {
      case 0b11 -> switch(compat32, vAddr[30..29]) { (* xkseg *)
        case (true, 0b11) -> (Kernel, None)               (* kseg3 mapped 32-bit compat *)
        case (true, 0b10) -> (Supervisor, None)           (* sseg  mapped 32-bit compat *)
        case (true, 0b01) -> (Kernel, Some(EXTZ(vAddr[28..0]))) (* kseg1 unmapped uncached 32-bit compat *)
        case (true, 0b00) -> (Kernel, Some(EXTZ(vAddr[28..0]))) (* kseg0 unmapped cached 32-bit compat *)
        case (_, _) -> (Kernel, None) (* xkseg mapped *)
      }
      case 0b10 -> (Kernel, Some(EXTZ(vAddr[58..0])))       (* xkphys bits 61-59 are cache mode (ignored) *)
      case 0b01 -> (Supervisor, None)                       (* xsseg - supervisor mapped *)
      case 0b00 -> (User, None) (* xuseg - user mapped *)
    } in
    if (((nat)currentAccessLevel) < ((nat)requiredLevel)) then
      exit (SignalExceptionBadAddr(if (accessType == StoreData) then AdES else AdEL, vAddr))
    else
      let (pa, c) = switch(addr) {
      case (Some(a)) -> (a, false)
      case None      -> if ((~(compat32)) & (unsigned(vAddr[61..0]) > MAX_VA)) then
          exit (SignalExceptionBadAddr(if (accessType == StoreData) then AdES else AdEL, vAddr))
        else
          TLBTranslate2(vAddr, accessType)
      }
      in if (unsigned(pa) > MAX_PA) then
          exit (SignalExceptionBadAddr(if (accessType == StoreData) then AdES else AdEL, vAddr))
        else
          (pa, c)
  }

function (bit[64]) TLBTranslate ((bit[64]) vAddr, (MemAccessType) accessType) =
      let (addr, c) = TLBTranslateC(vAddr, accessType) in addr

typedef regno = bit[5]                    (* a register number *)
```

```
typedef imm16 = bit[16]                    (* 16-bit immediate *)
(* a commonly used instruction format with three register operands *)
typedef regregreg   = (regno, regno, regno)
(* a commonly used instruction format with two register operands and 16-bit immediate *)
typedef regregimm16 = (regno, regno, imm16)

typedef decode_failure = enumerate {
  no_matching_pattern;
  unsupported_instruction;
  illegal_instruction;
  internal_error
}

(* Used by branch and trap instructions *)
typedef Comparison = enumerate {
  EQ; (* equal *)
  NE; (* not equal *)
  GE; (* signed greater than or equal *)
  GEU;(* unsigned greater than or equal *)
  GT; (* signed strictly greater than *)
  LE; (* signed less than or equal *)
  LT; (* signed strictly less than *)
  LTU;(* unsigned less than or qual *)
}
function bool compare ((Comparison)cmp, (bit[64]) valA, (bit[64]) valB) =
   (* sail comparisons are signed so extend to 65 bits for unsigned comparisons *)
  let valA65 = (0b0 : valA) in
  let valB65 = (0b0 : valB) in
  switch(cmp) {
    case EQ  -> valA ==   valB
    case NE  -> valA !=   valB
    case GE  -> valA >=   valB
    case GEU -> valA65 >= valB65
    case GT  -> valA >    valB
    case LE  -> valA <=   valB
    case LT  -> valA <    valB
    case LTU -> valA65 <  valB65
  }
typedef WordType = enumerate { B; H; W; D}

function forall Nat 'r, 'r IN {1,2,4,8}.[:'r:] wordWidthBytes((WordType) w) =
  switch(w) {
    case B -> 1
    case H -> 2
    case W -> 4
    case D -> 8
  }

function bool isAddressAligned(addr, (WordType) wordType) =
  switch (wordType) {
    case B -> true
    case H -> (addr[0]    == 0)
    case W -> (addr[1..0] == 0b00)
    case D -> (addr[2..0] == 0b000)
```

```
  }

(*=========================================================================*)
(*                                                                         *)
(*  Copyright (c) 2015-2016 Robert M. Norton                               *)
(*  Copyright (c) 2015-2016 Kathyrn Gray                                   *)
(*  All rights reserved.                                                   *)
(*                                                                         *)
(*  This software was developed by the University of Cambridge Computer    *)
(*  Laboratory as part of the Rigorous Engineering of Mainstream Systems   *)
(*  (REMS) project, funded by EPSRC grant EP/K008528/1.                    *)
(*                                                                         *)
(*  Redistribution and use in source and binary forms, with or without     *)
(*  modification, are permitted provided that the following conditions     *)
(*  are met:                                                               *)
(*  1. Redistributions of source code must retain the above copyright      *)
(*     notice, this list of conditions and the following disclaimer.       *)
(*  2. Redistributions in binary form must reproduce the above copyright   *)
(*     notice, this list of conditions and the following disclaimer in     *)
(*     the documentation and/or other materials provided with the          *)
(*     distribution.                                                       *)
(*                                                                         *)
(*  THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS''     *)
(*  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED      *)
(*  TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A        *)
(*  PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR    *)
(*  CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,           *)
(*  SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT       *)
(*  LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF       *)
(*  USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND    *)
(*  ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,     *)
(*  OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT     *)
(*  OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF     *)
(*  SUCH DAMAGE.                                                           *)
(*=========================================================================*)

(* mips_wrappers.sail: wrappers functions and hooks for CHERI extensibility
   (mostly identity functions here) *)

function unit effect {wmem} MEMw_wrapper(addr, size, data) = MEMw(addr, size, data)
function bool effect {wmem} MEMw_conditional_wrapper(addr, size, data) =
  MEMw_conditional(addr, size, data)

function bit[64] addrWrapper((bit[64]) addr, (MemAccessType) accessType, (WordType) width) =
    addr

function (bit[64]) TranslateAddress ((bit[64]) vAddr, (MemAccessType) accessType) = {
    incrementCP0Count();
    if (vAddr[1..0] != 0b00) then (* bad PC alignment *)
      exit (SignalExceptionBadAddr(AdEL, vAddr))
    else
      TLBTranslate(vAddr, accessType)
}
```

```
let have_cp2 = false

function unit SignalException ((Exception) ex) = SignalExceptionMIPS(ex)

function unit ERETHook() = ()
(*=========================================================================*)
(*                                                                         *)
(*  Copyright (c) 2015-2016 Robert M. Norton                               *)
(*  Copyright (c) 2015-2016 Kathyrn Gray                                   *)
(*  All rights reserved.                                                   *)
(*                                                                         *)
(*  This software was developed by the University of Cambridge Computer    *)
(*  Laboratory as part of the Rigorous Engineering of Mainstream Systems   *)
(*  (REMS) project, funded by EPSRC grant EP/K008528/1.                    *)
(*                                                                         *)
(*  Redistribution and use in source and binary forms, with or without     *)
(*  modification, are permitted provided that the following conditions     *)
(*  are met:                                                               *)
(*  1. Redistributions of source code must retain the above copyright      *)
(*     notice, this list of conditions and the following disclaimer.       *)
(*  2. Redistributions in binary form must reproduce the above copyright   *)
(*     notice, this list of conditions and the following disclaimer in     *)
(*     the documentation and/or other materials provided with the          *)
(*     distribution.                                                       *)
(*                                                                         *)
(*  THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS''     *)
(*  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED      *)
(*  TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A        *)
(*  PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR    *)
(*  CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,           *)
(*  SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT       *)
(*  LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF       *)
(*  USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND    *)
(*  ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,     *)
(*  OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT     *)
(*  OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF     *)
(*  SUCH DAMAGE.                                                           *)
(*=========================================================================*)

(* misp_insts.sail: mips instruction decode and execute clauses and AST node
   declarations *)

scattered function unit execute
scattered typedef ast = const union

val bit[32] -> option<ast> effect pure decode
scattered function option<ast> decode

(*****************************************************************************)
(* [D]ADD[I][U] various forms of ADD                                         *)
(*****************************************************************************)

(* DADDIU Doubleword Add Immediate Unsigned --
   the simplest possible instruction, no undefined behaviour or exceptions
```

```
   reg, reg, immediate *)

union ast member regregimm16 DADDIU

function clause decode (0b011001 : (regno) rs : (regno) rt : (imm16) imm) =
  Some(DADDIU(rs, rt, imm))

function clause execute (DADDIU (rs, rt, imm)) =
  {
    wGPR(rt) := rGPR(rs) + EXTS(imm)
  }

(* DADDU Doubleword Add Unsigned -- another very simple instruction,
   reg, reg, reg *)

union ast member regregreg DADDU

function clause decode (0b000000 : (regno) rs : (regno) rt : (regno) rd : 0b00000 : 0b101101) =
  Some(DADDU(rs, rt, rd))

function clause execute (DADDU (rs, rt, rd)) =
  {
    wGPR(rd) := rGPR(rs) + rGPR(rt)
  }

(* DADDI Doubleword Add Immediate
   reg, reg, imm with possible exception *)

union ast member regregimm16 DADDI

function clause decode (0b011000 : (regno) rs : (regno) rt : (imm16) imm) =
  Some(DADDI(rs, rt, imm))

function clause execute (DADDI (rs, rt, imm)) =
  {
    let (bit[65]) sum65 = (EXTS(rGPR(rs)) + EXTS(imm)) in
    {
      if (sum65[64] != sum65[63]) then
        exit (SignalException(Ov))
      else
        wGPR(rt) := sum65[63..0]
    }
  }

(* DADD Doubleword Add
   reg, reg, reg with possible exception *)

union ast member regregreg DADD

function clause decode (0b000000 : (regno) rs : (regno) rt : (regno) rd : 0b00000 : 0b101100) =
  Some(DADD(rs, rt, rd))

function clause execute (DADD (rs, rt, rd)) =
  {
```

```
    let (bit[65]) sum65 = (EXTS(rGPR(rs)) + EXTS(rGPR(rt))) in
    {
      if sum65[64] != sum65[63] then
        exit (SignalException(Ov))
      else
        wGPR(rd) := sum65[63..0]
    }
  }

(* ADD   32-bit add           -- reg, reg, reg with possible undefined behaviour or exception *)

union ast member regregreg ADD

function clause decode (0b000000 : (regno) rs : (regno) rt : (regno) rd : 0b00000 : 0b100000) =
  Some(ADD(rs, rt, rd))

function clause execute (ADD(rs, rt, rd)) =
  {
    (bit[64]) opA := rGPR(rs);
    (bit[64]) opB := rGPR(rt);
    if NotWordVal(opA) | NotWordVal(opB) then
      wGPR(rd) := undefined (* XXX could exit instead *)
    else
      let (bit[33]) sum33 = (EXTS(opA[31 .. 0]) + EXTS(opB[31 .. 0])) in
        if sum33[32] != sum33[31] then
          exit (SignalException(Ov))
        else
          wGPR(rd) := EXTS(sum33[31..0])
  }

(* ADDI  32-bit add immediate -- reg, reg, imm with possible undefined behaviour or exception *)

union ast member regregimm16 ADDI

function clause decode (0b001000 : (regno) rs : (regno) rt : (imm16) imm) =
  Some(ADDI(rs, rt, imm))

function clause execute (ADDI(rs, rt, imm)) =
  {
    (bit[64]) opA := rGPR(rs);
    if NotWordVal(opA) then
      wGPR(rt) := undefined (* XXX could exit instead *)
    else
      let (bit[33]) sum33 = (EXTS(opA[31 .. 0]) + EXTS(imm)) in
        if sum33[32] != sum33[31] then
          exit (SignalException(Ov))
        else
          wGPR(rt) := EXTS(sum33[31..0])
  }

(* ADDU  32-bit add immediate -- reg, reg, reg with possible undefined behaviour *)

union ast member regregreg ADDU
```

```
function clause decode (0b000000 : (regno) rs : (regno) rt : (regno) rd : 0b00000 : 0b100001) =
  Some(ADDU(rs, rt, rd))

function clause execute (ADDU(rs, rt, rd)) =
  {
    (bit[64]) opA := rGPR(rs);
    (bit[64]) opB := rGPR(rt);
    if NotWordVal(opA) | NotWordVal(opB) then
      wGPR(rd) := undefined
    else
      wGPR(rd) := EXTS(opA[31..0] + opB[31..0])
  }


(* ADDIU 32-bit add immediate -- reg, reg, imm with possible undefined behaviour *)

union ast member regregimm16 ADDIU

function clause decode (0b001001 : (regno) rs : (regno) rt : (imm16) imm) =
  Some(ADDIU(rs, rt, imm))

function clause execute (ADDIU(rs, rt, imm)) =
  {
    (bit[64]) opA := rGPR(rs);
    if NotWordVal(opA) then
      wGPR(rt) := undefined (* XXX could exit instead *)
    else
      wGPR(rt) := EXTS((opA[31 .. 0]) + EXTS(imm))
  }

(*****************************************************************************)
(* [D]SUB[U] various forms of SUB                                            *)
(*****************************************************************************)

(* DSUBU doubleword subtract 'unsigned' reg, reg, reg *)

union ast member regregreg DSUBU

function clause decode (0b000000 : (regno) rs : (regno) rt : (regno) rd : 0b00000 : 0b101111) =
  Some(DSUBU(rs, rt, rd))

function clause execute (DSUBU (rs, rt, rd)) =
  {
    wGPR(rd) := rGPR(rs) - rGPR(rt)
  }

(* DSUB reg, reg, reg with possible exception *)

union ast member regregreg DSUB

function clause decode (0b000000 : (regno) rs : (regno) rt : (regno) rd : 0b00000 : 0b101110) =
  Some(DSUB(rs, rt, rd))

function clause execute (DSUB (rs, rt, rd)) =
```

```
  {
    let (bit[65]) temp65 = (EXTS(rGPR(rs)) - EXTS(rGPR(rt))) in
    {
      if temp65[64] != temp65[63] then
        exit (SignalException(Ov))
      else
        wGPR(rd) := temp65[63..0]
    }
  }

(* SUB   32-bit sub           -- reg, reg, reg with possible undefined behaviour or exception *)

union ast member regregreg SUB

function clause decode (0b000000 : (regno) rs : (regno) rt : (regno) rd : 0b00000 : 0b100010) =
  Some(SUB(rs, rt, rd))

function clause execute (SUB(rs, rt, rd)) =
  {
    (bit[64]) opA := rGPR(rs);
    (bit[64]) opB := rGPR(rt);
    if NotWordVal(opA) | NotWordVal(opB) then
      wGPR(rd) := undefined (* XXX could exit instead *)
    else
      let (bit[33]) temp33 = (EXTS(opA[31..0]) - EXTS(opB[31..0])) in
      if temp33[32] != temp33[31] then
        exit (SignalException(Ov))
      else
        wGPR(rd) := EXTS(temp33[31..0])
  }

(* SUBU  32-bit 'unsigned' sub -- reg, reg, reg with possible undefined behaviour *)

union ast member regregreg SUBU
function clause decode (0b000000 : (regno) rs : (regno) rt : (regno) rd : 0b00000 : 0b100011) =
    Some(SUBU(rs, rt, rd))

function clause execute (SUBU(rs, rt, rd)) =
  {
    (bit[64]) opA := rGPR(rs);
    (bit[64]) opB := rGPR(rt);
    if NotWordVal(opA) | NotWordVal(opB) then
      wGPR(rd) := undefined
    else
      wGPR(rd) := EXTS(opA[31..0] - opB[31..0])
  }

(*****************************************************************************)
(* Logical bitwise operations                                                *)
(*****************************************************************************)

(* AND reg, reg, reg *)

union ast member regregreg AND
```

```
function clause decode (0b000000 : (regno) rs : (regno) rt : (regno) rd : 0b00000 : 0b100100) =
    Some(AND(rs, rt, rd))

function clause execute (AND (rs, rt, rd)) =
  {
    wGPR(rd) := (rGPR(rs) & rGPR(rt))
  }

(* ANDI reg, reg, imm *)

union ast member regregimm16 ANDI
function clause decode (0b001100 : (regno) rs : (regno) rt : (imm16) imm) =
  Some(ANDI(rs, rt, imm))
function clause execute (ANDI (rs, rt, imm)) =
  {
    wGPR(rt) := (rGPR(rs) & EXTZ(imm))
  }

(* OR reg, reg, reg *)

union ast member regregreg OR
function clause decode (0b000000 : (regno) rs : (regno) rt : (regno) rd : 0b00000 : 0b100101) =
  Some(OR(rs, rt, rd))
function clause execute (OR (rs, rt, rd)) =
  {
    wGPR(rd) := (rGPR(rs) | rGPR(rt))
  }

(* ORI reg, reg, imm *)

union ast member regregimm16 ORI
function clause decode (0b001101 : (regno) rs : (regno) rt : (imm16) imm) =
  Some(ORI(rs, rt, imm))
function clause execute (ORI (rs, rt, imm)) =
  {
    wGPR(rt) := (rGPR(rs) | EXTZ(imm))
  }

(* NOR reg, reg, reg *)

union ast member regregreg NOR
function clause decode (0b000000 : (regno) rs : (regno) rt : (regno) rd : 0b00000 : 0b100111) =
  Some(NOR(rs, rt, rd))
function clause execute (NOR (rs, rt, rd)) =
  {
    wGPR(rd) := ~(rGPR(rs) | rGPR(rt))
  }

(* XOR reg, reg, reg *)

union ast member regregreg XOR
function clause decode (0b000000 : (regno) rs : (regno) rt : (regno) rd : 0b00000 : 0b100110) =
  Some(XOR(rs, rt, rd))
function clause execute (XOR (rs, rt, rd)) =
```

```
  {
    wGPR(rd) := (rGPR(rs) ^ rGPR(rt))
  }

(* XORI reg, reg, imm *)
union ast member regregimm16 XORI
function clause decode (0b001110 : (regno) rs : (regno) rt : (imm16) imm) =
  Some(XORI(rs, rt, imm))
function clause execute (XORI (rs, rt, imm)) =
  {
    wGPR(rt) := (rGPR(rs) ^ EXTZ(imm))
  }

(* LUI reg, imm 32-bit load immediate into upper 16 bits *)
union ast member (regno, imm16) LUI
function clause decode (0b001111 : 0b00000 : (regno) rt : (imm16) imm) =
  Some(LUI(rt, imm))
function clause execute (LUI (rt, imm)) =
  {
    wGPR(rt) := EXTS(imm : 0x0000)
  }

(*****************************************************************************)
(* 64-bit  shift operations                                                  *)
(*****************************************************************************)

(* DSLL reg, reg, imm5 *)

union ast member regregreg DSLL
function clause decode (0b000000 : 0b00000 : (regno) rt : (regno) rd : (bit[5]) sa : 0b111000) =
  Some(DSLL(rt, rd, sa))
function clause execute (DSLL (rt, rd, sa)) =
  {

    wGPR(rd) := (rGPR(rt) << sa) (* make tuareg mode less blue >> *)
  }

(* DSLL32 reg, reg, imm5 *)

union ast member regregreg DSLL32
function clause decode (0b000000 : 0b00000 : (regno) rt : (regno) rd : (bit[5]) sa : 0b111100) =
  Some(DSLL32(rt, rd, sa))
function clause execute (DSLL32 (rt, rd, sa)) =
  {
    wGPR(rd) := (rGPR(rt) << (0b1 : sa)) (* make tuareg mode less blue >> *)
  }

(* DSLLV reg, reg, reg *)

union ast member regregreg DSLLV
function clause decode (0b000000 : (regno) rs : (regno) rt : (regno) rd : 0b00000 : 0b010100) =
  Some(DSLLV(rs, rt, rd))
function clause execute (DSLLV (rs, rt, rd)) =
  {
```

```
    wGPR(rd) := (rGPR(rt) << ((rGPR(rs))[5 .. 0])) (* make tuareg mode less blue >> *)
    (* alternative slicing based version of above
    sa       := (rGPR(rt))[5 .. 0];
    v        := rGPR(rs);
    wGPR(rd) := v[(63-sa) .. 0] : (0b0 ^^ sa)  *)
  }

(* DSRA arithmetic shift duplicating sign bit - reg, reg, imm5 *)

union ast member regregreg DSRA
function clause decode (0b000000 : 0b00000 : (regno) rt : (regno) rd : (bit[5]) sa : 0b111011) =
  Some(DSRA(rt, rd, sa))
function clause execute (DSRA (rt, rd, sa)) =
  {
    temp     := rGPR(rt);
    wGPR(rd) := ((temp[63] ^^ sa) : (temp[63 .. sa]))
  }

(* DSRA32 reg, reg, imm5 *)

union ast member regregreg DSRA32
function clause decode (0b000000 : 0b00000 : (regno) rt : (regno) rd : (bit[5]) sa : 0b111111) =
  Some(DSRA32(rt, rd, sa))
function clause execute (DSRA32 (rt, rd, sa)) =
  {
    temp     := rGPR(rt);
    sa32     := (0b1 : sa); (* sa+32 *)
    wGPR(rd) := ((temp[63] ^^ sa32) : (temp[63 .. sa32]))
  }

(* DSRAV reg, reg, reg *)
union ast member regregreg DSRAV
function clause decode (0b000000 : (regno) rs : (regno) rt : (regno) rd : 0b00000 : 0b010111) =
  Some(DSRAV(rs, rt, rd))
function clause execute (DSRAV (rs, rt, rd)) =
  {
    temp     := rGPR(rt);
    sa       := (rGPR(rs)) [5..0];
    wGPR(rd) := ((temp[63] ^^ sa) : temp[63 .. sa])
  }

(* DSRL shift right logical - reg, reg, imm5 *)

union ast member regregreg DSRL
function clause decode (0b000000 : 0b00000 : (regno) rt : (regno) rd : (bit[5]) sa : 0b111010) =
  Some(DSRL(rt, rd, sa))
function clause execute (DSRL (rt, rd, sa)) =
  {
    temp     := rGPR(rt);
    wGPR(rd) := ((bitzero ^^ sa) : (temp[63 .. sa]))
  }

(* DSRL32 reg, reg, imm5 *)
```

```
union ast member regregreg DSRL32
function clause decode (0b000000 : 0b00000 : (regno) rt : (regno) rd : (bit[5]) sa : 0b111110) =
  Some(DSRL32(rt, rd, sa))
function clause execute (DSRL32 (rt, rd, sa)) =
  {
    temp     := rGPR(rt);
    sa32     := (0b1 : sa); (* sa+32 *)
    wGPR(rd) := ((bitzero ^^ sa32) : (temp[63 .. sa32]))
  }

(* DSRLV reg, reg, reg *)

union ast member regregreg DSRLV
function clause decode (0b000000 : (regno) rs : (regno) rt : (regno) rd : 0b00000 : 0b010110) =
  Some(DSRLV(rs, rt, rd))
function clause execute (DSRLV (rs, rt, rd)) =
  {
    temp     := rGPR(rt);
    sa       := (rGPR(rs)) [5..0];
    wGPR(rd) := ((bitzero ^^ sa) : temp[63 .. sa])
  }

(*****************************************************************************)
(* 32-bit  shift operations                                                  *)
(*****************************************************************************)

(* SLL 32-bit shift left *)

union ast member regregreg SLL
function clause decode (0b000000 : 0b00000 : (regno) rt : (regno) rd : (regno) sa : 0b000000) =
  Some(SLL(rt, rd, sa))
function clause execute (SLL(rt, rd, sa)) =
  {
    wGPR(rd) := EXTS((rGPR(rt)) [(31-sa)..0] : (bitzero ^^ sa))
  }

(* SLLV 32-bit shift left variable *)

union ast member regregreg SLLV
function clause decode (0b000000 : (regno) rs : (regno) rt : (regno) rd : 0b00000 : 0b000100) =
  Some(SLLV(rs, rt, rd))
function clause execute (SLLV(rs, rt, rd)) =
  {
    sa       := (rGPR(rs))[4..0];
    wGPR(rd) := EXTS((rGPR(rt)) [(31-sa)..0] : (bitzero ^^ sa))
  }

(* SRA 32-bit arithmetic shift right *)

union ast member regregreg SRA
function clause decode (0b000000 : 0b00000 : (regno) rt : (regno) rd : (regno) sa : 0b000011) =
  Some(SRA(rt, rd, sa))
function clause execute (SRA(rt, rd, sa)) =
  {
```

```
    temp := rGPR(rt);
    if (NotWordVal(temp)) then
      wGPR(rd) := undefined
    else
      wGPR(rd) := (temp[31] ^^ (sa+32)) : temp [31..sa]
  }

(* SRAV 32-bit arithmetic shift right variable *)

union ast member regregreg SRAV
function clause decode (0b000000 : (regno) rs : (regno) rt : (regno) rd : 0b00000 : 0b000111) =
  Some(SRAV(rs, rt, rd))
function clause execute (SRAV(rs, rt, rd)) =
  {
    temp := rGPR(rt);
    sa   := (rGPR(rs))[4..0];
    if (NotWordVal(temp)) then
      wGPR(rd) := undefined
    else
      wGPR(rd) := (temp[31] ^^ (sa+32)) : temp [31..sa]
  }

(* SRL 32-bit shift right *)

union ast member regregreg SRL
function clause decode (0b000000 : 0b00000 : (regno) rt : (regno) rd : (regno) sa : 0b000010) =
  Some(SRL(rt, rd, sa))
function clause execute (SRL(rt, rd, sa)) =
  {
    temp := rGPR(rt);
    if (NotWordVal(temp)) then
      wGPR(rd) := undefined
    else
      wGPR(rd) := EXTS((bitzero ^^ sa) : (temp [31..sa]))
  }

(* SRLV 32-bit shift right variable *)

union ast member regregreg SRLV
function clause decode (0b000000 : (regno) rs : (regno) rt : (regno) rd : 0b00000 : 0b000110) =
  Some(SRLV(rs, rt, rd))
function clause execute (SRLV(rs, rt, rd)) =
  {
    temp := rGPR(rt);
    sa   := (rGPR(rs))[4..0];
    if (NotWordVal(temp)) then
      wGPR(rd) := undefined
    else
      wGPR(rd) := EXTS((bitzero ^^ sa) : (temp [31..sa]))
  }

(*****************************************************************************)
(* set less than / conditional move                                          *)
(*****************************************************************************)
```

```
(* SLT set if less than (signed) *)

union ast member regregreg SLT
function clause decode (0b000000 : (regno) rs : (regno) rt : (regno) rd : 0b00000 : 0b101010) =
  Some(SLT(rs, rt, rd))
function clause execute (SLT(rs, rt, rd)) =
  {
    wGPR(rd) := if (rGPR(rs) <_s rGPR(rt)) then 1 else 0
  }

(* SLT set if less than immediate (signed) *)

union ast member regregimm16 SLTI
function clause decode (0b001010 : (regno) rs : (regno) rt : (imm16) imm) =
  Some(SLTI(rs, rt, imm))
function clause execute (SLTI(rs, rt, imm)) =
  {
    let imm_val = signed(imm) in
    let rs_val  = signed(rGPR(rs)) in
    wGPR(rt) := if (rs_val < imm_val) then 0x0000000000000001 else 0x0000000000000000
  }

(* SLTU set if less than unsigned *)

union ast member regregreg SLTU
function clause decode (0b000000 : (regno) rs : (regno) rt : (regno) rd : 0b00000 : 0b101011) =
  Some(SLTU(rs, rt, rd))
function clause execute (SLTU(rs, rt, rd)) =
  {
    let rs_val = (0b0 : (rGPR(rs))) in
    let rt_val = (0b0 : (rGPR(rt))) in
    wGPR(rd) := (if (rs_val < rt_val) then 1 else 0)
  }

(* SLTIU set if less than immediate unsigned *)

union ast member regregimm16 SLTIU
function clause decode (0b001011 : (regno) rs : (regno) rt : (imm16) imm) =
  Some(SLTIU(rs, rt, imm))
function clause execute (SLTIU(rs, rt, imm)) =
  {
    let rs_val  = (0b0 : (rGPR(rs))) in
    let imm_val = (0b0 : ((bit[64])(EXTS(imm)))) in
    wGPR(rt) := (if (rs_val < imm_val) then 1 else 0)
  }

(* MOVN move if non-zero *)

union ast member regregreg MOVN
function clause decode (0b000000 : (regno) rs : (regno) rt : (regno) rd : 0b00000 : 0b001011) =
  Some(MOVN(rs, rt, rd))
function clause execute (MOVN(rs, rt, rd)) =
  {
```

```
    if (rGPR(rt) != 0x0000000000000000) then
      wGPR(rd) := rGPR(rs)
  }

(* MOVZ move if zero *)

union ast member regregreg MOVZ
function clause decode (0b000000 : (regno) rs : (regno) rt : (regno) rd : 0b00000 : 0b001010) =
  Some(MOVZ(rs, rt, rd))
function clause execute (MOVZ(rs, rt, rd)) =
  {
    if (rGPR(rt) == 0x0000000000000000) then
      wGPR(rd) := rGPR(rs)
  }

(*****************************************************************************)
(* MUL/DIV instructions                                                      *)
(*****************************************************************************)

(* MFHI move from HI register *)
union ast member regno MFHI
function clause decode (0b000000 : 0b0000000000 : (regno) rd : 0b00000 : 0b010000) =
  Some(MFHI(rd))
function clause execute (MFHI(rd)) =
  {
    wGPR(rd) := HI
  }

(* MFLO move from LO register *)
union ast member regno MFLO
function clause decode (0b000000 : 0b0000000000 : (regno) rd : 0b00000 : 0b010010) =
  Some(MFLO(rd))
function clause execute (MFLO(rd)) =
  {
    wGPR(rd) := LO
  }

(* MTHI move to HI register *)
union ast member regno MTHI
function clause decode (0b000000 : (regno) rs : 0b000000000000000 : 0b010001) =
  Some(MTHI(rs))
function clause execute (MTHI(rs)) =
  {
    HI := rGPR(rs)
  }

(* MTLO move to LO register *)
union ast member regno MTLO
function clause decode (0b000000 : (regno) rs : 0b000000000000000 : 0b010011) =
  Some(MTLO(rs))
function clause execute (MTLO(rs)) =
  {
    LO := rGPR(rs)
  }
```

```
(* MUL 32-bit multiply into GPR *)
union ast member regregreg MUL
function clause decode (0b011100 : (regno) rs : (regno) rt : (regno) rd : 0b00000 : 0b000010) =
  Some(MUL(rs, rt, rd))
function clause execute (MUL(rs, rt, rd)) =
  {
    rsVal := rGPR(rs);
    rtVal := rGPR(rt);
    (bit[64]) result := (rsVal[31..0]) *_s (rtVal[31..0]);
    wGPR(rd) := if (NotWordVal(rsVal) | NotWordVal(rtVal)) then
        undefined
      else
        EXTS(result[31..0]);
    (* HI and LO are technically undefined after MUL, but this causes problems with tests and
       (potentially) context switch so just leave them alone
    HI := undefined;
    LO := undefined;
    *)
  }

(* MULT 32-bit multiply into HI/LO *)
union ast member (regno, regno) MULT
function clause decode (0b000000 : (regno) rs : (regno) rt : 0b00000 : 0b00000 : 0b011000) =
  Some(MULT(rs, rt))
function clause execute (MULT(rs, rt)) =
  {
    rsVal := rGPR(rs);
    rtVal := rGPR(rt);
    (bit[64]) result := if (NotWordVal(rsVal) | NotWordVal(rtVal)) then
        undefined
      else
        (rsVal[31..0]) *_s (rtVal[31..0]);
    HI := EXTS(result[63..32]);
    LO := EXTS(result[31..0]);
  }

(* MULTU 32-bit unsignedm rultiply into HI/LO *)
union ast member (regno, regno) MULTU
function clause decode (0b000000 : (regno) rs : (regno) rt : 0b00000 : 0b00000 : 0b011001) =
  Some(MULTU(rs, rt))
function clause execute (MULTU(rs, rt)) =
  {
    rsVal := rGPR(rs);
    rtVal := rGPR(rt);
    (bit[64]) result := if (NotWordVal(rsVal) | NotWordVal(rtVal)) then
        undefined
      else
        (rsVal[31..0]) * (rtVal[31..0]);
    HI := EXTS(result[63..32]);
    LO := EXTS(result[31..0]);
  }

(* DMULT 64-bit multiply into HI/LO *)
```

```
union ast member (regno, regno) DMULT
function clause decode (0b000000 : (regno) rs : (regno) rt : 0b00000 : 0b00000 : 0b011100) =
  Some(DMULT(rs, rt))
function clause execute (DMULT(rs, rt)) =
  {
    (bit[128]) result := (rGPR(rs)) *_s (rGPR(rt));
    HI := (result[127..64]);
    LO := (result[63..0]);
  }

(* DMULTU 64-bit unsigned multiply into HI/LO *)
union ast member (regno, regno) DMULTU
function clause decode (0b000000 : (regno) rs : (regno) rt : 0b00000 : 0b00000 : 0b011101) =
  Some(DMULTU(rs, rt))
function clause execute (DMULTU(rs, rt)) =
  {
    (bit[128]) result := (rGPR(rs)) * (rGPR(rt));
    HI := (result[127..64]);
    LO := (result[63..0]);
  }

(* MADD 32-bit signed multiply and add into HI/LO *)
union ast member (regno, regno) MADD
function clause decode (0b011100 : (regno) rs : (regno) rt : 0b00000 : 0b00000 : 0b000000) =
  Some(MADD(rs, rt))
function clause execute (MADD(rs, rt)) =
  {
    rsVal := rGPR(rs);
    rtVal := rGPR(rt);
    (bit[64]) mul_result := if (NotWordVal(rsVal) | NotWordVal(rtVal)) then
        undefined
      else
        ((rsVal[31..0]) *_s (rtVal[31..0]));
    (bit[64]) result := mul_result + (HI[31..0] : LO[31..0]);
    HI := EXTS(result[63..32]);
    LO := EXTS(result[31..0]);
  }

(* MADDU 32-bit unsigned multiply and add into HI/LO *)
union ast member (regno, regno) MADDU
function clause decode (0b011100 : (regno) rs : (regno) rt : 0b00000 : 0b00000 : 0b000001) =
  Some(MADDU(rs, rt))
function clause execute (MADDU(rs, rt)) =
  {
    rsVal := rGPR(rs);
    rtVal := rGPR(rt);
    (bit[64]) mul_result := if (NotWordVal(rsVal) | NotWordVal(rtVal)) then
        undefined
      else
        ((rsVal[31..0]) * (rtVal[31..0]));
    (bit[64]) result := mul_result + (HI[31..0] : LO[31..0]);
    HI := EXTS(result[63..32]);
    LO := EXTS(result[31..0]);
  }
```

```
(* MSUB 32-bit signed multiply and sub from HI/LO *)
union ast member (regno, regno) MSUB
function clause decode (0b011100 : (regno) rs : (regno) rt : 0b00000 : 0b00000 : 0b000100) =
  Some(MSUB(rs, rt))
function clause execute (MSUB(rs, rt)) =
  {
    rsVal := rGPR(rs);
    rtVal := rGPR(rt);
    (bit[64]) mul_result := if (NotWordVal(rsVal) | NotWordVal(rtVal)) then
        undefined
      else
        ((rsVal[31..0]) *_s (rtVal[31..0]));
    (bit[64]) result := (HI[31..0] : LO[31..0]) - mul_result;
    HI := EXTS(result[63..32]);
    LO := EXTS(result[31..0]);
  }

(* MSUBU 32-bit unsigned multiply and sub from HI/LO *)
union ast member (regno, regno) MSUBU
function clause decode (0b011100 : (regno) rs : (regno) rt : 0b00000 : 0b00000 : 0b000101) =
  Some(MSUBU(rs, rt))
function clause execute (MSUBU(rs, rt)) =
  {
    rsVal := rGPR(rs);
    rtVal := rGPR(rt);
    (bit[64]) mul_result := if (NotWordVal(rsVal) | NotWordVal(rtVal)) then
        undefined
      else
        ((rsVal[31..0]) * (rtVal[31..0]));
    (bit[64]) result := (HI[31..0] : LO[31..0]) - mul_result;
    HI := EXTS(result[63..32]);
    LO := EXTS(result[31..0]);
  }

(* DIV 32-bit divide into HI/LO *)
union ast member (regno, regno) DIV
function clause decode (0b000000 : (regno) rs : (regno) rt : 0b00000 : 0b00000 : 0b011010) =
  Some(DIV(rs, rt))
function clause execute (DIV(rs, rt)) =
  {
    rsVal := rGPR(rs);
    rtVal := rGPR(rt);
    let ((bit[32]) q, (bit[32])r) = (
      if (NotWordVal(rsVal) | NotWordVal(rtVal) | (rtVal == 0)) then
        (undefined, undefined)
      else
        let si = (signed((rsVal[31..0]))) in
        let ti = (signed((rtVal[31..0]))) in
        let qi = (si quot ti)  in
        let ri = (si - (ti*qi))   in
        ((bit[32]) qi, (bit[32]) ri))
    in
    {
```

```
      HI := EXTS(r);
      LO := EXTS(q);
    }
  }

(* DIVU 32-bit unsigned divide into HI/LO *)
union ast member (regno, regno) DIVU
function clause decode (0b000000 : (regno) rs : (regno) rt : 0b00000 : 0b00000 : 0b011011) =
  Some(DIVU(rs, rt))
function clause execute (DIVU(rs, rt)) =
  {
    rsVal := rGPR(rs);
    rtVal := rGPR(rt);
    let ((bit[32]) q, (bit[32])r) = (
      if (NotWordVal(rsVal) | NotWordVal(rtVal) | rtVal == 0) then
        (undefined, undefined)
      else
        let si = ((int)(rsVal[31..0])) in
        let ti = ((int)(rtVal[31..0])) in
        let qi = (si quot ti)  in
        let ri = (si mod ti)   in
        ((bit[32]) qi, (bit[32]) ri))
    in
    {
      HI := EXTS(r);
      LO := EXTS(q);
    }
  }

(* DDIV 64-bit divide into HI/LO *)
union ast member (regno, regno) DDIV
function clause decode (0b000000 : (regno) rs : (regno) rt : 0b00000 : 0b00000 : 0b011110) =
  Some(DDIV(rs, rt))
function clause execute (DDIV(rs, rt)) =
  {
    rsVal := signed(rGPR(rs));
    rtVal := signed(rGPR(rt));
    let ((bit[64])q, (bit[64])r) = (if (rtVal == 0)
      then (undefined, undefined)
      else
           let qi = (rsVal quot_s rtVal) in
           let ri = (rsVal - (qi * rtVal)) in
           ((bit[64]) qi, (bit[64]) ri)) in
    {
      LO    := q;
      HI    := r;
    }
  }

(* DDIV 64-bit divide into HI/LO *)
union ast member (regno, regno) DDIVU
function clause decode (0b000000 : (regno) rs : (regno) rt : 0b00000 : 0b00000 : 0b011111) =
  Some(DDIVU(rs, rt))
function clause execute (DDIVU(rs, rt)) =
```

```
  {
    (int) rsVal := rGPR(rs);
    (int) rtVal := rGPR(rt);
    let ((bit[64])q, (bit[64])r) = (if (rtVal == 0)
      then (undefined, undefined)
      else let qi = (rsVal quot rtVal) in
           let ri = (rsVal mod rtVal) in
           ((bit[64]) qi, (bit[64]) ri)) in
    {
      LO    := q;
      HI    := r;
    }
  }

(*******************************************************************************)
(* Jump instructions -- unconditional branches                                 *)
(*******************************************************************************)

(* J    - jump, the simplest control flow instruction, with branch delay slot *)
union ast member (bit[26]) J
function clause decode (0b000010 : (bit[26]) offset) =
  Some(J(offset))
function clause execute (J(offset)) =
  {
    delayedPC     := (PC + 4)[63..28] : offset : 0b00;
    branchPending := 1
  }

(* JAL    - jump and link *)
union ast member (bit[26]) JAL
function clause decode (0b000011 : (bit[26]) offset) =
  Some(JAL(offset))
function clause execute (JAL(offset)) =
  {
    delayedPC     := (PC + 4)[63..28] : offset : 0b00;
    branchPending := 1;
    wGPR(31)      := PC + 8;
  }

(* JR -- jump via register *)
union ast member regno JR
function clause decode (0b000000 : (regno) rs : 0b00000 : 0b00000 : (regno) hint : 0b001000) =
  Some(JR(rs)) (* hint is ignored *)
function clause execute (JR(rs)) =
  {
    delayedPC     := rGPR(rs);
    branchPending := 1;
  }

(* JALR -- jump via register with link *)
union ast member (regno, regno) JALR
function clause decode (0b000000 : (regno) rs : 0b00000 : (regno) rd : (regno) hint : 0b001001) =
  Some(JALR(rs, rd)) (* hint is ignored *)
function clause execute (JALR(rs, rd)) =
```

```
  {
    delayedPC     := rGPR(rs);
    branchPending := 1;
    wGPR(rd)      := PC + 8;
  }

(*****************************************************************************)
(* B[N]EQ[L]  - branch on (not) equal (likely)                               *)
(* Conditional branch instructions with two register operands                *)
(*****************************************************************************)

union ast member (regno, regno, imm16, bool, bool) BEQ
function clause decode (0b000100 : (regno) rs : (regno) rt : (imm16) imm) =
  Some(BEQ(rs, rt, imm, false, false))  (* BEQ  *)
function clause decode (0b010100 : (regno) rs : (regno) rt : (imm16) imm) =
  Some(BEQ(rs, rt, imm, false, true))   (* BEQL *)
function clause decode (0b000101 : (regno) rs : (regno) rt : (imm16) imm) =
  Some(BEQ(rs, rt, imm, true , false))  (* BNE  *)
function clause decode (0b010101 : (regno) rs : (regno) rt : (imm16) imm) =
  Some(BEQ(rs, rt, imm, true , true))   (* BNEL *)
function clause execute (BEQ(rs, rd, imm, ne, likely)) =
  {
    if ((rGPR(rs) == rGPR(rd)) ^ ne) then
      {
        let (bit[64]) offset = (EXTS(imm : 0b00) + 4) in
        delayedPC     := PC + offset;
        branchPending := 1;
      }
    else
      {
        if (likely) then
          nextPC := PC + 8; (* skip branch delay *)
      }
  }

(*

   Branches comparing with zero (single register operand, possible link in r31)
*)

(*****************************************************************************)
(* BGEZ[AL][L] - branch on comparison with zero (possibly  link, likely)     *)
(* Conditional branch instructions with single register operand              *)
(*****************************************************************************)

union ast member (regno, imm16, Comparison, bool, bool) BCMPZ
function clause decode (0b000001 : (regno) rs : 0b00000 : (imm16) imm) =
  Some(BCMPZ(rs, imm, LT, false, false)) (* BLTZ *)
function clause decode (0b000001 : (regno) rs : 0b10000 : (imm16) imm) =
  Some(BCMPZ(rs, imm, LT, true, false))  (* BLTZAL *)
function clause decode (0b000001 : (regno) rs : 0b00010 : (imm16) imm) =
  Some(BCMPZ(rs, imm, LT, false, true))  (* BLTZL *)
function clause decode (0b000001 : (regno) rs : 0b10010 : (imm16) imm) =
  Some(BCMPZ(rs, imm, LT, true, true))   (* BLTZALL *)
```

```
function clause decode (0b000001 : (regno) rs : 0b00001 : (imm16) imm) =
  Some(BCMPZ(rs, imm, GE, false, false)) (* BGEZ *)
function clause decode (0b000001 : (regno) rs : 0b10001 : (imm16) imm) =
  Some(BCMPZ(rs, imm, GE, true, false))  (* BGEZAL *)
function clause decode (0b000001 : (regno) rs : 0b00011 : (imm16) imm) =
  Some(BCMPZ(rs, imm, GE, false, true))  (* BGEZL *)
function clause decode (0b000001 : (regno) rs : 0b10011 : (imm16) imm) =
  Some(BCMPZ(rs, imm, GE, true, true))   (* BGEZALL *)

function clause decode (0b000111 : (regno) rs : 0b00000 : (imm16) imm) =
  Some(BCMPZ(rs, imm, GT, false, false)) (* BGTZ *)
function clause decode (0b010111 : (regno) rs : 0b00000 : (imm16) imm) =
  Some(BCMPZ(rs, imm, GT, false, true))  (* BGTZL *)

function clause decode (0b000110 : (regno) rs : 0b00000 : (imm16) imm) =
  Some(BCMPZ(rs, imm, LE, false, false)) (* BLEZ *)
function clause decode (0b010110 : (regno) rs : 0b00000 : (imm16) imm) =
  Some(BCMPZ(rs, imm, LE, false, true))  (* BLEZL *)

function clause execute (BCMPZ(rs, imm, cmp, link, likely)) =
  {
    (bit[64]) linkVal := PC + 8;
    regVal := rGPR(rs);
    condition := compare(cmp, regVal, 0);
    if (condition) then
      {
        let (bit[64]) offset = (EXTS(imm : 0b00) + 4) in
        delayedPC     :=  PC + offset;
        branchPending := 1;
      }
    else if (likely) then
      {
        nextPC := PC + 8  (* skip branch delay *)
      };
    if (link) then
      wGPR(31) := linkVal
  }

(*****************************************************************************)
(* SYSCALL/BREAK/WAIT/Trap                                                   *)
(*****************************************************************************)

union ast member unit SYSCALL
function clause decode (0b000000 : (bit[20]) code : 0b001100) =
  Some(SYSCALL) (* code is ignored *)
function clause execute (SYSCALL) =
  {
    exit (SignalException(Sys))
  }

(* BREAK is identical to SYSCALL exception for the exception raised *)
union ast member unit BREAK
function clause decode (0b000000 : (bit[20]) code : 0b001101) =
```

```
  Some(BREAK) (* code is ignored *)
function clause execute (BREAK) =
  {
    exit (SignalException(Bp))
  }

(* Accept WAIT as a NOP *)
union ast member unit WAIT
function clause decode (0b010000 : 0x80000 : 0b100000) =
  Some(WAIT) (* we only accept code == 0 *)
function clause execute (WAIT) = {
  nextPC := PC;
}

(* Trap instructions with two register operands *)
union ast member (regno, regno, Comparison) TRAPREG
function clause decode (0b000000 : (regno) rs : (regno) rt : (bit[10]) code : 0b110000) =
  Some(TRAPREG(rs, rt, GE)) (* TGE *)
function clause decode (0b000000 : (regno) rs : (regno) rt : (bit[10]) code : 0b110001) =
  Some(TRAPREG(rs, rt, GEU)) (* TGEU *)
function clause decode (0b000000 : (regno) rs : (regno) rt : (bit[10]) code : 0b110010) =
  Some(TRAPREG(rs, rt, LT)) (* TLT *)
function clause decode (0b000000 : (regno) rs : (regno) rt : (bit[10]) code : 0b110011) =
  Some(TRAPREG(rs, rt, LTU)) (* TLTU *)
function clause decode (0b000000 : (regno) rs : (regno) rt : (bit[10]) code : 0b110100) =
  Some(TRAPREG(rs, rt, EQ)) (* TEQ *)
function clause decode (0b000000 : (regno) rs : (regno) rt : (bit[10]) code : 0b110110) =
  Some(TRAPREG(rs, rt, NE)) (* TNE *)
function clause execute (TRAPREG(rs, rt, cmp)) =
  {
    rs_val := rGPR(rs);
    rt_val := rGPR(rt);
    condition := compare(cmp, rs_val, rt_val);
    if (condition) then
      exit (SignalException(Tr))
  }


(* Trap instructions with one register and one immediate operand *)
union ast member (regno, imm16, Comparison) TRAPIMM
function clause decode (0b000001 : (regno) rs : 0b01100 : (imm16) imm) =
  Some(TRAPIMM(rs, imm, EQ)) (* TEQI *)
function clause decode (0b000001 : (regno) rs : 0b01110 : (imm16) imm) =
  Some(TRAPIMM(rs, imm, NE)) (* TNEI *)
function clause decode (0b000001 : (regno) rs : 0b01000 : (imm16) imm) =
  Some(TRAPIMM(rs, imm, GE)) (* TGEI *)
function clause decode (0b000001 : (regno) rs : 0b01001 : (imm16) imm) =
  Some(TRAPIMM(rs, imm, GEU)) (* TGEIU *)
function clause decode (0b000001 : (regno) rs : 0b01010 : (imm16) imm) =
  Some(TRAPIMM(rs, imm, LT))  (* TLTI *)
function clause decode (0b000001 : (regno) rs : 0b01011 : (imm16) imm) =
  Some(TRAPIMM(rs, imm, LTU)) (* TLTIU *)
function clause execute (TRAPIMM(rs, imm, cmp)) =
  {
```

```
    rs_val := rGPR(rs);
    imm_val := EXTS(imm);
    condition := compare(cmp, rs_val, imm_val);
    if (condition) then
      exit (SignalException(Tr))
  }

(*****************************************************************************)
(* Load instructions -- various width/signs                                  *)
(*****************************************************************************)

union ast member (WordType, bool, bool, regno, regno, imm16) Load
function clause decode (0b100000 : (regno) base : (regno) rt : (imm16) offset) =
  Some(Load(B, true,  false, base, rt, offset)) (* LB *)
function clause decode (0b100100 : (regno) base : (regno) rt : (imm16) offset) =
  Some(Load(B, false, false, base, rt, offset)) (* LBU *)
function clause decode (0b100001 : (regno) base : (regno) rt : (imm16) offset) =
  Some(Load(H, true,  false, base, rt, offset)) (* LH *)
function clause decode (0b100101 : (regno) base : (regno) rt : (imm16) offset) =
  Some(Load(H, false, false, base, rt, offset)) (* LHU *)
function clause decode (0b100011 : (regno) base : (regno) rt : (imm16) offset) =
  Some(Load(W, true,  false, base, rt, offset)) (* LW *)
function clause decode (0b100111 : (regno) base : (regno) rt : (imm16) offset) =
  Some(Load(W, false, false, base, rt, offset)) (* LWU *)
function clause decode (0b110111 : (regno) base : (regno) rt : (imm16) offset) =
  Some(Load(D, false, false, base, rt, offset)) (* LD *)
function clause decode (0b110000 : (regno) base : (regno) rt : (imm16) offset) =
  Some(Load(W, true,  true, base, rt, offset))  (* LL *)
function clause decode (0b110100 : (regno) base : (regno) rt : (imm16) offset) =
  Some(Load(D, false, true, base, rt, offset))  (* LLD *)
function clause execute (Load(width, signed, linked, base, rt, offset)) =
  {
    (bit[64]) vAddr := addrWrapper(EXTS(offset) + rGPR(base), LoadData, width);
    if ~ (isAddressAligned(vAddr, width)) then
      exit (SignalExceptionBadAddr(AdEL, vAddr)) (* unaligned access *)
    else
      let pAddr = (TLBTranslate(vAddr, LoadData)) in
          {
            memResult := if (linked) then
                {
                  CP0LLBit  := 0b1;
                  CP0LLAddr := pAddr;
                  MEMr_reserve(pAddr, wordWidthBytes(width));
                }
              else
                MEMr(pAddr, wordWidthBytes(width));
            if (signed) then
              wGPR(rt) := EXTS(memResult)
            else
              wGPR(rt) := EXTZ(memResult)
          }
  }

(*****************************************************************************)
```

```
(* Store instructions -- various widths                                          *)
(*******************************************************************************)

union ast member (WordType, bool, regno, regno, imm16) Store
function clause decode (0b101000 : (regno) base : (regno) rt : (imm16) offset) =
  Some(Store(B, false, base, rt, offset)) (* SB *)
function clause decode (0b101001 : (regno) base : (regno) rt : (imm16) offset) =
  Some(Store(H, false, base, rt, offset)) (* SH *)
function clause decode (0b101011 : (regno) base : (regno) rt : (imm16) offset) =
  Some(Store(W, false, base, rt, offset)) (* SW *)
function clause decode (0b111111 : (regno) base : (regno) rt : (imm16) offset) =
  Some(Store(D, false, base, rt, offset)) (* SD *)
function clause decode (0b111000 : (regno) base : (regno) rt : (imm16) offset) =
  Some(Store(W, true, base, rt, offset))  (* SC *)
function clause decode (0b111100 : (regno) base : (regno) rt : (imm16) offset) =
  Some(Store(D, true, base, rt, offset))  (* SCD *)
function clause execute (Store(width, conditional, base, rt, offset)) =
      {
        (bit[64]) vAddr := addrWrapper(EXTS(offset) + rGPR(base), StoreData, width);
        (bit[64]) rt_val := rGPR(rt);
        if ~ (isAddressAligned(vAddr, width)) then
          exit (SignalExceptionBadAddr(AdES, vAddr)) (* unaligned access *)
        else
          let pAddr = (TLBTranslate(vAddr, StoreData)) in
          {
                if (conditional) then
                  {
                    success := if (CP0LLBit[0]) then switch(width)
                    {
                      case B -> MEMw_conditional_wrapper(pAddr, 1, rt_val[7..0])
                      case H -> MEMw_conditional_wrapper(pAddr, 2, rt_val[15..0])
                      case W -> MEMw_conditional_wrapper(pAddr, 4, rt_val[31..0])
                      case D -> MEMw_conditional_wrapper(pAddr, 8, rt_val)
                    } else false;
                    wGPR(rt) := EXTZ([success])
                  }
                else
                  switch(width)
                    {
                      case B -> MEMw_wrapper(pAddr, 1) := rt_val[7..0]
                      case H -> MEMw_wrapper(pAddr, 2) := rt_val[15..0]
                      case W -> MEMw_wrapper(pAddr, 4) := rt_val[31..0]
                      case D -> MEMw_wrapper(pAddr, 8) := rt_val
                    }
          }
      }

(* LWL   - Load word left (big-endian only) *)

union ast member regregimm16 LWL
function clause decode(0b100010 : (regno) base : (regno) rt : (imm16) offset) =
      Some(LWL(base, rt, offset))
function clause execute(LWL(base, rt, offset)) =
      {
```

```
        (* XXX length check not quite right, but conservative *)
        (bit[64]) vAddr := addrWrapper(EXTS(offset) + rGPR(base), LoadData, W);
        let pAddr = (TLBTranslate(vAddr, LoadData)) in
          {
            mem_val   := MEMr (pAddr[63..2] : 0b00, 4); (* read word of interest *)
            reg_val   := rGPR(rt);
            wGPR(rt)  := EXTS(switch(vAddr[1..0])
              {
                case 0b00 -> mem_val
                case 0b01 -> mem_val[23..0] : reg_val[07..0]
                case 0b10 -> mem_val[15..0] : reg_val[15..0]
                case 0b11 -> mem_val[07..0] : reg_val[23..0]
              });
          }
      }
union ast member regregimm16 LWR
function clause decode(0b100110 : (regno) base : (regno) rt : (imm16) offset) =
      Some(LWR(base, rt, offset))
function clause execute(LWR(base, rt, offset)) =
      {
        (* XXX length check not quite right, but conservative *)
        (bit[64]) vAddr := addrWrapper(EXTS(offset) + rGPR(base), LoadData, W);
        let pAddr = (TLBTranslate(vAddr, LoadData)) in
          {
            mem_val   := MEMr (pAddr[63..2] : 0b00, 4); (* read word of interest *)
            reg_val   := rGPR(rt);
            wGPR(rt)  := EXTS(switch(vAddr[1..0]) (* it is acceptable to sign extend in all cases *)
              {
                case 0b00 -> reg_val[31..8]  : mem_val[31..24]
                case 0b01 -> reg_val[31..16] : mem_val[31..16]
                case 0b10 -> reg_val[31..24] : mem_val[31..8]
                case 0b11 -> mem_val
              });
          }
      }

(* SWL   - Store word left *)
union ast member regregimm16 SWL
function clause decode(0b101010 : (regno) base : (regno) rt : (imm16) offset) =
      Some(SWL(base, rt, offset))
function clause execute(SWL(base, rt, offset)) =
      {
        (* XXX length check not quite right, but conservative *)
        (bit[64]) vAddr := addrWrapper(EXTS(offset) + rGPR(base), StoreData, W);
        let pAddr = (TLBTranslate(vAddr, StoreData)) in
          {
            reg_val   := rGPR(rt);
            switch (vAddr[1..0])
            {
              case 0b00 -> (MEMw_wrapper(pAddr, 4) := reg_val[31..0])
              case 0b01 -> (MEMw_wrapper(pAddr, 3) := reg_val[31..8])
              case 0b10 -> (MEMw_wrapper(pAddr, 2) := reg_val[31..16])
              case 0b11 -> (MEMw_wrapper(pAddr, 1) := reg_val[31..24])
            }
```

```
          }
      }

union ast member regregimm16 SWR
function clause decode(0b101110 : (regno) base : (regno) rt : (imm16) offset) =
      Some(SWR(base, rt, offset))
function clause execute(SWR(base, rt, offset)) =
      {
        (* XXX length check not quite right, but conservative *)
        (bit[64]) vAddr := addrWrapper(EXTS(offset) + rGPR(base), StoreData, W);
        let (pAddr) = (TLBTranslate(vAddr, StoreData)) in
          {
            wordAddr := pAddr[63..2] : 0b00;
            reg_val   := rGPR(rt);
            switch (vAddr[1..0])
              {
                case 0b00 -> (MEMw_wrapper(wordAddr, 1) := reg_val[7..0])
                case 0b01 -> (MEMw_wrapper(wordAddr, 2) := reg_val[15..0])
                case 0b10 -> (MEMw_wrapper(wordAddr, 3) := reg_val[23..0])
                case 0b11 -> (MEMw_wrapper(wordAddr, 4) := reg_val[31..0])
              }
          }
      }

(* Load double-word left *)
union ast member regregimm16 LDL
function clause decode(0b011010 : (regno) base : (regno) rt : (imm16) offset) =
  Some(LDL(base, rt, offset))
function clause execute(LDL(base, rt, offset)) =
  {
    (* XXX length check not quite right, but conservative *)
    (bit[64]) vAddr := addrWrapper(EXTS(offset) + rGPR(base), LoadData, D);
    let pAddr = (TLBTranslate(vAddr, StoreData)) in
      {
        mem_val   := MEMr (pAddr[63..3] : 0b000, 8); (* read double of interest *)
        reg_val   := rGPR(rt);
        wGPR(rt)  := switch(vAddr[2..0])
              {
                case 0b000 -> mem_val
                case 0b001 -> mem_val[55..0] : reg_val[7..0]
                case 0b010 -> mem_val[47..0] : reg_val[15..0]
                case 0b011 -> mem_val[39..0] : reg_val[23..0]
                case 0b100 -> mem_val[31..0] : reg_val[31..0]
                case 0b101 -> mem_val[23..0] : reg_val[39..0]
                case 0b110 -> mem_val[15..0] : reg_val[47..0]
                case 0b111 -> mem_val[07..0] : reg_val[55..0]
              };
      }
  }

(* Load double-word right *)
union ast member regregimm16 LDR
function clause decode(0b011011 : (regno) base : (regno) rt : (imm16) offset) =
  Some(LDR(base, rt, offset))
```

```
function clause execute(LDR(base, rt, offset)) =
  {
    (* XXX length check not quite right, but conservative *)
    (bit[64]) vAddr := addrWrapper(EXTS(offset) + rGPR(base), LoadData, D);
    let pAddr = (TLBTranslate(vAddr, StoreData)) in
      {
        mem_val   := MEMr (pAddr[63..3] : 0b000, 8); (* read double of interest *)
        reg_val   := rGPR(rt);
        wGPR(rt)  := switch(vAddr[2..0])
              {
                case 0b000 -> reg_val[63..08] : mem_val[63..56]
                case 0b001 -> reg_val[63..16] : mem_val[63..48]
                case 0b010 -> reg_val[63..24] : mem_val[63..40]
                case 0b011 -> reg_val[63..32] : mem_val[63..32]
                case 0b100 -> reg_val[63..40] : mem_val[63..24]
                case 0b101 -> reg_val[63..48] : mem_val[63..16]
                case 0b110 -> reg_val[63..56] : mem_val[63..08]
                case 0b111 -> mem_val
              };
      }
  }

(* SDL   - Store double-word left *)
union ast member regregimm16 SDL
function clause decode(0b101100 : (regno) base : (regno) rt : (imm16) offset) =
      Some(SDL(base, rt, offset))
function clause execute(SDL(base, rt, offset)) =
      {
        (* XXX length check not quite right, but conservative *)
        (bit[64]) vAddr := addrWrapper(EXTS(offset) + rGPR(base), StoreData, D);
        let pAddr = (TLBTranslate(vAddr, StoreData)) in
          {
            reg_val := rGPR(rt);
            switch (vAddr[2..0])
            {
              case 0b000 -> (MEMw_wrapper(pAddr, 8) := reg_val[63..00])
              case 0b001 -> (MEMw_wrapper(pAddr, 7) := reg_val[63..08])
              case 0b010 -> (MEMw_wrapper(pAddr, 6) := reg_val[63..16])
              case 0b011 -> (MEMw_wrapper(pAddr, 5) := reg_val[63..24])
              case 0b100 -> (MEMw_wrapper(pAddr, 4) := reg_val[63..32])
              case 0b101 -> (MEMw_wrapper(pAddr, 3) := reg_val[63..40])
              case 0b110 -> (MEMw_wrapper(pAddr, 2) := reg_val[63..48])
              case 0b111 -> (MEMw_wrapper(pAddr, 1) := reg_val[63..56])
            }
          }
      }


(* SDR   - Store double-word right *)
union ast member regregimm16 SDR
function clause decode(0b101101 : (regno) base : (regno) rt : (imm16) offset) =
      Some(SDR(base, rt, offset))
function clause execute(SDR(base, rt, offset)) =
  {
```

```
        (* XXX length check not quite right, but conservative *)
        (bit[64]) vAddr := addrWrapper(EXTS(offset) + rGPR(base), StoreData, D);
        let pAddr = (TLBTranslate(vAddr, StoreData)) in
          {
            reg_val := rGPR(rt);
            wordAddr := pAddr[63..3] : 0b000;
            switch (vAddr[2..0])
            {
              case 0b000 -> (MEMw_wrapper(wordAddr, 1) := reg_val[07..0])
              case 0b001 -> (MEMw_wrapper(wordAddr, 2) := reg_val[15..0])
              case 0b010 -> (MEMw_wrapper(wordAddr, 3) := reg_val[23..0])
              case 0b011 -> (MEMw_wrapper(wordAddr, 4) := reg_val[31..0])
              case 0b100 -> (MEMw_wrapper(wordAddr, 5) := reg_val[39..0])
              case 0b101 -> (MEMw_wrapper(wordAddr, 6) := reg_val[47..0])
              case 0b110 -> (MEMw_wrapper(wordAddr, 7) := reg_val[55..0])
              case 0b111 -> (MEMw_wrapper(wordAddr, 8) := reg_val[63..0])
            }
          }
      }

(* CACHE - manipulate (non-existent) caches *)

union ast member regregimm16 CACHE
function clause decode (0b101111 : (regno) base : (regno) op : (imm16) imm) =
  Some(CACHE(base, op, imm))
function clause execute (CACHE(base, op, imm)) =
  checkCP0Access () (* pretty much a NOP because no caches *)

(* PREF - prefetching into (non-existent) caches *)

union ast member regregimm16 PREF
function clause decode (0b110011 : (regno) base : (regno) op : (imm16) imm) =
  Some(PREF(base, op, imm))
function clause execute (PREF(base, op, imm)) =
  () (* XXX NOP *)


(* SYNC  - Memory barrier *)
union ast member unit SYNC
function clause decode (0b000000 : 0b00000 : 0b00000 : 0b00000 : (regno) stype : 0b001111) =
  Some(SYNC()) (* stype is currently ignored *)
function clause execute(SYNC) =
  MEM_sync()

union ast member (regno, regno, bit[3], bool) MFC0
function clause decode (0b010000 : 0b00000 : (regno) rt : (regno) rd : 0b00000000 : (bit[3]) sel) =
  Some(MFC0(rt, rd, sel, false)) (* MFC0 *)
function clause decode (0b010000 : 0b00001 : (regno) rt : (regno) rd : 0b00000000 : (bit[3]) sel) =
  Some(MFC0(rt, rd, sel, true))  (* DMFC0 *)
function clause execute (MFC0(rt, rd, sel, double)) = {
  checkCP0Access();
  let (bit[64]) result = switch (rd, sel)
        {
          case (0b00000,0b000) -> (0x00000000 : [TLBProbe] : 0x0000000 : TLBIndex)  (* 0, TLB Index *)
```

```
case (0b00001,0b000) -> EXTZ(TLBRandom)  (* 1, TLB Random *)
case (0b00010,0b000) -> TLBEntryLo0      (* 2, TLB EntryLo0 *)
case (0b00011,0b000) -> TLBEntryLo1      (* 3, TLB EntryLo1 *)
case (0b00100,0b000) -> TLBContext       (* 4, TLB Context *)
case (0b00101,0b000) -> EXTZ(TLBPageMask : 0x000) (* 5, TLB PageMask *)
case (0b00110,0b000) -> EXTZ(TLBWired)   (* 6, TLB Wired *)
case (0b00111,0b000) -> EXTZ(CP0HWREna)  (* 7, HWREna *)
case (0b01000,0b000) -> CP0BadVAddr      (* 8, BadVAddr reg *)
case (0b01000,0b001) -> 0                (* 8, BadInstr reg XXX TODO *)
case (0b01001,0b000) -> EXTZ(CP0Count)   (* 9, Count reg *)
case (0b01010,0b000) -> TLBEntryHi       (* 10, TLB EntryHi *)
case (0b01011,0b000) -> EXTZ(CP0Compare) (* 11, Compare reg *)
case (0b01100,0b000) -> EXTZ(CP0Status)  (* 12, Status reg *)
case (0b01101,0b000) -> EXTZ(CP0Cause)   (* 13, Cause reg *)
case (0b01110,0b000) -> CP0EPC           (* 14, EPC *)
case (0b01111,0b000) -> EXTZ(0x00000400) (* 15, sel 0: PrID processor ID *)
case (0b01111,0b110) -> 0                (* 15, sel 6: CHERI core ID *)
case (0b01111,0b111) -> 0                (* 15, sel 7: CHERI thread ID *)
case (0b10000,0b000) -> EXTZ(0b1 (* M *)      (* 16, sel 0: Config0 *)
                        : 0b000000000000000 (* Impl *)
                        : 0b1               (* BE *)
                        : 0b10              (* AT *)
                        : 0b000             (* AR *)
                        : 0b001             (* MT standard TLB *)
                        : 0b0000            (* zero *)
                        : 0b000) (* K0 TODO should be writable*)
case (0b10000,0b001) -> EXTZ( (* 16, sel 1: Config1 *)
                                0b1      (* M *)
                              : 0b000111 (* MMU size-1 *)
                              : 0b000    (* IS icache sets *)
                              : 0b000    (* IL icache lines *)
                              : 0b000    (* IA icache assoc. *)
                              : 0b000    (* DS dcache sets *)
                              : 0b000    (* DL dcache lines *)
                              : 0b000    (* DA dcache assoc. *)
                              : [have_cp2] (* C2 CP2 presence *)
                              : 0b0      (* MD MDMX implemented *)
                              : 0b0      (* PC performance counters *)
                              : 0b0      (* WR watch registers *)
                              : 0b0      (* CA 16e code compression *)
                              : 0b0      (* EP EJTAG *)
                              : 0b0)     (* FP FPU present *)
case (0b10000,0b010) -> EXTZ( (* 16, sel 2: Config2 *)
                                0b1     (* M *)
                              : 0b000   (* TU L3 control  *)
                              : 0b0000  (* TS L3 sets *)
                              : 0b0000  (* TL L3 lines *)
                              : 0b0000  (* TA L3 assoc. *)
                              : 0b0000  (* SU L2 control *)
                              : 0b0000  (* SS L2 sets *)
                              : 0b0000  (* SL L2 lines *)
                              : 0b0000) (* SA L2 assoc. *)
case (0b10000,0b011) -> 0x0000000000002000 (* 16, sel 3: Config3 zero except for bit 13 == ulri *)
case (0b10000,0b101) -> 0x0000000000000000 (* 16, sel 5: Config5 beri specific -- no extended TLB *)
```

```
          case (0b10001,0b000) -> CP0LLAddr        (* 17, sel 0: LLAddr *)
          case (0b10010,0b000) -> 0                (* 18, WatchLo *)
          case (0b10011,0b000) -> 0                (* 19, WatchHi *)
          case (0b10100,0b000) -> TLBXContext      (* 20, XContext *)
          case (0b11110,0b000) -> CP0ErrorEPC      (* 30, ErrorEPC *)
          case _               -> {exit (SignalException(ResI)); 0}
        } in
  wGPR(rt) := if (double) then result else EXTS(result[31..0])
}

union ast member (regno, regno, bit[3], bool) MTC0
function clause decode (0b010000 : 0b00100 : (regno) rt : (regno) rd : 0b00000000 : (bit[3]) sel) =
  Some(MTC0(rt, rd, sel, false)) (* MTC0 *)
function clause decode (0b010000 : 0b00101 : (regno) rt : (regno) rd : 0b00000000 : (bit[3]) sel) =
  Some(MTC0(rt, rd, sel, true))  (* DMTC0 *)
function clause execute (MTC0(rt, rd, sel, double)) = {
  checkCP0Access();
  let reg_val = (rGPR(rt)) in
  switch (rd, sel)
    {
      case (0b00000,0b000) -> TLBIndex := mask(reg_val) (* NB no write to TLBProbe *)
      case (0b00001,0b000) -> ()                        (* TLBRandom is read only *)
      case (0b00010,0b000) -> TLBEntryLo0 := reg_val
      case (0b00011,0b000) -> TLBEntryLo1 := reg_val
      case (0b00100,0b000) -> (TLBContext.PTEBase) := (reg_val[63..23])
      case (0b00100,0b010) -> CP0UserLocal := reg_val
      case (0b00101,0b000) -> TLBPageMask  := (reg_val[28..13])
      case (0b00110,0b000) -> {
        TLBWired     := mask(reg_val);
        TLBRandom    := TLBIndexMax;
      }
      case (0b00111,0b000) -> CP0HWREna := (reg_val[31..29] : 0b00000000000000000000000000 : reg_val[3..0])
      case (0b01000,0b000) -> () (* BadVAddr read only *)
      case (0b01001,0b000) -> CP0Count := reg_val[31..0]
      case (0b01010,0b000) -> {
        (TLBEntryHi.R)    := (reg_val[63..62]);
        (TLBEntryHi.VPN2) := (reg_val[39..13]);
        (TLBEntryHi.ASID) := (reg_val[7..0]);
      }
      case (0b01011,0b000) -> {  (* 11, sel 0: Compare reg *)
        CP0Compare := reg_val[31..0];
        (CP0Cause[15]) := bitzero;
      }
      case (0b01100,0b000) -> {  (* 12 Status *)
        (CP0Status.CU)  := reg_val[31..28];
        (CP0Status.BEV) := reg_val[22];
        (CP0Status.IM)  := reg_val[15..8];
        (CP0Status.KX)  := reg_val[7];
        (CP0Status.SX)  := reg_val[6];
        (CP0Status.UX)  := reg_val[5];
        (CP0Status.KSU) := reg_val[4..3];
        (CP0Status.ERL) := reg_val[2];
        (CP0Status.EXL) := reg_val[1];
        (CP0Status.IE)  := reg_val[0];
```

```
      }
      case (0b01100,0b000) -> {   (* 13 Cause *)
        CP0Cause.IV := reg_val[23]; (* TODO special interrupt vector not implemeneted *)
        (CP0Cause.IP)[9..8] := reg_val[9..8];
      }
      case (0b01110,0b000) -> CP0EPC      := reg_val         (* 14, EPC *)
      case (0b10100,0b000) -> (TLBXContext.PTEBase) := (reg_val[63..33])
      case (0b11110,0b000) -> CP0ErrorEPC := reg_val         (* 30, ErrorEPC *)
      case _               -> exit (SignalException(ResI))
    }
}

function unit TLBWriteEntry((TLBIndexT) idx) = {
  pagemask := ((bit[16]) TLBPageMask);
  switch(pagemask) {
      case 0x0000 -> ()
      case 0x0003 -> ()
      case 0x000f -> ()
      case 0x003f -> ()
      case 0x00ff -> ()
      case 0x03ff -> ()
      case 0x0fff -> ()
      case 0x3fff -> ()
      case 0xffff -> ()
      case _      -> exit (SignalException(MCheck))
  };
  ((TLBEntries[idx]).pagemask) := pagemask;
  ((TLBEntries[idx]).r       ) := TLBEntryHi.R;
  ((TLBEntries[idx]).vpn2    ) := TLBEntryHi.VPN2;
  ((TLBEntries[idx]).asid    ) := TLBEntryHi.ASID;
  ((TLBEntries[idx]).g       ) := ((TLBEntryLo0.G) & (TLBEntryLo1.G));
  ((TLBEntries[idx]).valid   ) := bitone;
  ((TLBEntries[idx]).caps0   ) := TLBEntryLo0.CapS;
  ((TLBEntries[idx]).capl0   ) := TLBEntryLo0.CapL;
  ((TLBEntries[idx]).pfn0    ) := TLBEntryLo0.PFN;
  ((TLBEntries[idx]).c0      ) := TLBEntryLo0.C;
  ((TLBEntries[idx]).d0      ) := TLBEntryLo0.D;
  ((TLBEntries[idx]).v0      ) := TLBEntryLo0.V;
  ((TLBEntries[idx]).caps1   ) := TLBEntryLo1.CapS;
  ((TLBEntries[idx]).capl1   ) := TLBEntryLo1.CapL;
  ((TLBEntries[idx]).pfn1    ) := TLBEntryLo1.PFN;
  ((TLBEntries[idx]).c1      ) := TLBEntryLo1.C;
  ((TLBEntries[idx]).d1      ) := TLBEntryLo1.D;
  ((TLBEntries[idx]).v1      ) := TLBEntryLo1.V;
}

union ast member TLBWI
function clause decode (0b010000 : 0b10000000000000000000 : 0b000010) = Some(TLBWI)
function clause execute (TLBWI) = {
  checkCP0Access();
  TLBWriteEntry(TLBIndex);
}

union ast member TLBWR
```

```
function clause decode (0b010000 : 0b10000000000000000000 : 0b000110) = Some(TLBWR)
function clause execute (TLBWR) = {
  checkCP0Access();
  TLBWriteEntry(TLBRandom);
}

union ast member TLBR
function clause decode (0b010000 : 0b10000000000000000000 : 0b000001) = Some(TLBR)
function clause execute (TLBR) = {
  checkCP0Access();
  let entry = TLBEntries[TLBIndex] in {
    TLBPageMask     := entry.pagemask;
    TLBEntryHi.R    := entry.r;
    TLBEntryHi.VPN2 := entry.vpn2;
    TLBEntryHi.ASID := entry.asid;
    TLBEntryLo0.CapS:= entry.caps0;
    TLBEntryLo0.CapL:= entry.capl0;
    TLBEntryLo0.PFN := entry.pfn0;
    TLBEntryLo0.C   := entry.c0;
    TLBEntryLo0.D   := entry.d0;
    TLBEntryLo0.V   := entry.v0;
    TLBEntryLo0.G   := entry.g;
    TLBEntryLo1.CapS:= entry.caps1;
    TLBEntryLo1.CapL:= entry.capl1;
    TLBEntryLo1.PFN := entry.pfn1;
    TLBEntryLo1.C   := entry.c1;
    TLBEntryLo1.D   := entry.d1;
    TLBEntryLo1.V   := entry.v1;
    TLBEntryLo1.G   := entry.g;
  }
}

union ast member TLBP
function clause decode (0b010000 : 0b10000000000000000000 : 0b001000) = Some(TLBP)
function clause execute ((TLBP)) = {
  checkCP0Access();
  let result = tlbSearch(TLBEntryHi) in
  switch(result) {
    case (Some(idx)) -> {
      TLBProbe := [bitzero];
      TLBIndex := idx;
    }
    case None -> {
      TLBProbe := [bitone];
      TLBIndex := 0;
    }
  }
}

union ast member (regno, regno) RDHWR
function clause decode (0b011111 : 0b00000 : (regno) rt : (regno) rd : 0b00000 : 0b111011) =
  Some(RDHWR(rt, rd))
function clause execute (RDHWR(rt, rd)) = {
  let accessLevel = getAccessLevel() in
```

```
  if ((accessLevel != Kernel) & (~((CP0Status.CU)[0])) & ~(CP0HWREna[rd])) then
    exit (SignalException(ResI));
  let (bit[64]) temp = switch (rd) {
    case 0b00000 -> EXTZ([bitzero])  (* CPUNum *)
    case 0b00001 -> EXTZ([bitzero])  (* SYNCI_step *)
    case 0b00010 -> EXTZ(CP0Count) (* Count *)
    case 0b00011 -> EXTZ([bitone])   (* Count resolution *)
    case 0b11101 -> CP0UserLocal   (* User local register *)
    case _       -> exit (SignalException(ResI))
  } in
  wGPR(rt) := temp;
}

union ast member unit ERET
function clause decode (0b010000 : 0b1 : 0b0000000000000000000 : 0b011000) =
      Some(ERET)
function clause execute (ERET) =
      {
        checkCP0Access();
        ERETHook();
        CP0LLBit := 0b0;
        if (CP0Status.ERL == bitone) then
          {
            nextPC := CP0ErrorEPC;
            CP0Status.ERL := 0;
          }
        else
          {
            nextPC := CP0EPC;
            CP0Status.EXL := 0;
          }
      }

(* simulator halt instruction "MTC0 rt, r23" (cheri specific behaviour) *)
union ast member unit HCF
function clause decode (0b010000 : 0b00100 : (regno) rt : 0b10111 : 0b00000000000) =
  Some(HCF())

function clause decode (0b010000 : 0b00100 : (regno) rt : 0b11010 : 0b00000000000) =
  Some(HCF())

function clause execute (HCF) =
  () (* halt instruction actually executed by interpreter framework *)
(*-------------------------------------------------------------------------*)
(*                                                                         *)
(*  Copyright (c) 2015-2016 Robert M. Norton                               *)
(*  Copyright (c) 2015-2016 Kathyrn Gray                                   *)
(*  All rights reserved.                                                   *)
(*                                                                         *)
(*  This software was developed by the University of Cambridge Computer    *)
(*  Laboratory as part of the Rigorous Engineering of Mainstream Systems   *)
(*  (REMS) project, funded by EPSRC grant EP/K008528/1.                    *)
(*                                                                         *)
(*  Redistribution and use in source and binary forms, with or without     *)
```

```
(*  modification, are permitted provided that the following conditions    *)
(*  are met:                                                              *)
(*  1. Redistributions of source code must retain the above copyright     *)
(*     notice, this list of conditions and the following disclaimer.      *)
(*  2. Redistributions in binary form must reproduce the above copyright  *)
(*     notice, this list of conditions and the following disclaimer in    *)
(*     the documentation and/or other materials provided with the         *)
(*     distribution.                                                      *)
(*                                                                        *)
(*  THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS''    *)
(*  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED     *)
(*  TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A       *)
(*  PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR   *)
(*  CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,          *)
(*  SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT      *)
(*  LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF      *)
(*  USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND   *)
(*  ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,    *)
(*  OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT    *)
(*  OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF    *)
(*  SUCH DAMAGE.                                                          *)
(*========================================================================*)

(* mips_epilogue.sail: end of decode, execute and AST definitions. *)

union ast member unit RI
function clause decode _ = Some(RI)
function clause execute (RI) =
      exit (SignalException (ResI))

end decode
end execute
end ast

function option<ast> supported_instructions (instr) = Some(instr)
```