```
***
*** "A 32-bit Pipelined RISC Microprocessor Specification"
*** Sean Handley, sean.handley@gmail.com
*** 2006-11-10
***

***
*** Definitions for binary arithmetic
***
fmod BINARY is
   protecting INT .

   sorts Bit Bits .

   subsort Bit < Bits .

   ops 0 1 : -> Bit .
   op __ : Bits Bits -> Bits [assoc prec 1 gather (e E)] .
   op |_| : Bits -> Int .
   op normalize : Bits -> Bits .
   op bits : Bits Int Int -> Bits .
   op _++_  : Bits Bits -> Bits [assoc comm prec 5 gather (E e)] .
   op _**_  : Bits Bits -> Bits [assoc comm prec 4 gather (E e)] .
   op _>_  : Bits Bits -> Bool [prec 6 gather (E E)] .
   op not_ : Bits -> Bits [prec 2 gather (E)] .
   op _and_ : Bits Bits -> Bits [assoc comm prec 2 gather (E e)] .
   op _or_ : Bits Bits -> Bits [assoc comm prec 2 gather (E e)] .
   op _sl_ : Bits Bits -> Bits [prec 2 gather (E e)] .
   op _-- : Bits -> Bits [prec 2 gather (E)] .
   op bin2int : Bits -> Int .

   vars S T : Bits .
   vars B C : Bit .
   var L : Bool .
   vars I J : Int .

   op constzero32 : -> Bits .
   op constzero8 : -> Bits .
   op constminus1 : -> Bits .

   *** define constants for zero^32 and zero^8
   eq constzero32 =   0 0 0 0 0 0 0 0
                      0 0 0 0 0 0 0 0
                      0 0 0 0 0 0 0 0
                      0 0 0 0 0 0 0 0 .

   eq constzero8 =    0 0 0 0 0 0 0 0 .

   eq constminus1 =   1 1 1 1 1 1 1 1
                      1 1 1 1 1 1 1 1
                      1 1 1 1 1 1 1 1
                      1 1 1 1 1 1 1 1 .

   *** Binary to Integer
   ceq bin2int(B) = 0 if normalize(B) == 0 .
   ceq bin2int(B) = 1 if normalize(B) == 1 .
   eq bin2int(S) = 1 + bin2int((S)--) .

   *** Length
   eq | B | = 1 .
```

```
eq | S B | = | S | + 1 .

*** Extract Bits...
eq bits(S B,0,0) = B .
eq bits(B,J,0) = B .
ceq bits(S B,J,0) = bits(S, J - 1,0) B if J > 0 .
ceq bits(S B,J,I) = bits(S,J - 1,I - 1) if I > 0 and J > 0 .

*** Not
eq not (S T) = (not S) (not T) .
eq not 0 = 1 .
eq not 1 = 0 .

*** And
eq B and 0 = 0 .
eq B and 1 = B .
eq (S B) and (T C) = (S and T) (B and C) .

*** Or
eq B or 0 = B .
eq B or 1 = 1 .
eq (S B) or (T C) = (S or T) (B or C) .

*** Normalize supresses zeros at the
*** left of a binary number
eq normalize(0) = 0 .
eq normalize(1) = 1 .
eq normalize(0 S) = normalize(S) .
eq normalize(1 S) = 1 S .

*** Greater than
eq 0 > S = false .
eq 1 > (0).Bit = true .
eq 1 > (1).Bit = false .
eq B > (0 S) = B > S .
eq B > (1 S) = false .
eq (1 S) > B = true .
eq (B S) > (C T)
= if | normalize(B S) | > | normalize(C T) |
    then true
    else if | normalize(B S) | < | normalize(C T) |
        then false
    else (S > T)
    fi
fi .

*** Binary addition
eq 0 ++ S = S .
eq 1 ++ 1 = 1 0 .
eq 1 ++ (T 0) = T 1 .
eq 1 ++ (T 1) = (T ++ 1) 0 .
eq (S B) ++ (T 0) = (S ++ T) B .
eq (S 1) ++ (T 1) = (S ++ T ++ 1) 0 .

*** Binary multiplication
eq 0 ** T = 0 .
eq 1 ** T = T .
eq (S B) ** T = ((S ** T) 0) ++ (B ** T) .

*** Decrement
eq 0 -- = 0 .
```

```
   eq 1 -- = 0 .
   eq (S 1) -- = normalize(S 0) .
   ceq (S 0) -- = normalize(S --) 1 if normalize(S) =/= 0 .
   ceq (S 0) -- = 0 if normalize(S) == 0 .

   *** Shift left
   ceq S sl T = ((S 0) sl (T --)) if bin2int(T) > 0 .
   eq S sl T = S .
endfm

***
*** Module for dealing with machine words and instruction formats.
***
fmod MACHINE-WORD is
   protecting BINARY .

   *** 32-bit machine word, 1 byte per opcode/reg address
   *** Opfields and register addresses are both 1 byte so they share a name

   sorts OpField Word .

   subsort OpField < Bits .
   subsort Word < Bits .

   op opcode : Word -> OpField .
   ops rega regb regc : Word -> OpField .

   op _+_ : Word Word -> Word .
   op _+_ : OpField OpField -> OpField .

   op _*_ : Word Word -> Word .
   op _*_ : OpField OpField -> OpField .

   op _&_ : Word Word -> Word .
   op _&_ : OpField OpField -> OpField .

   op _|_ : Word Word -> Word .
   op _|_ : OpField OpField -> OpField .

   op !_ : Word -> Word .
   op !_ : OpField -> OpField .

   op _<<_ : Word Word -> Word .
   op _<<_ : OpField OpField -> OpField .

   op _gt_ : Word Word -> Bool .
   op _gt_ : OpField OpField -> Bool .

   op Four : -> Bits .

   vars B1 B2 B3 B4 B5 B6 B7 B8 : Bit .
   vars B9 B10 B11 B12 B13 B14 B15 B16 : Bit .
   vars B17 B18 B19 B20 B21 B22 B23 B24 : Bit .
   vars B25 B26 B27 B28 B29 B30 B31 B32 : Bit .

   vars V W : Word .
   vars A B : OpField .

   *** 8 bits = opfield
   mb (B1 B2 B3 B4 B5 B6 B7 B8) : OpField .

   *** 32 bits = word and/or memory address
   mb (B1 B2 B3 B4 B5 B6 B7 B8
```

```
      B9 B10 B11 B12 B13 B14 B15 B16
      B17 B18 B19 B20 B21 B22 B23 B24
      B25 B26 B27 B28 B29 B30 B31 B32) : Word .

   *** 1 byte per opcode/reg address
   eq opcode(W) = bits(W,31,24) .
   *** eq opcode(W) = bits(W,7,0) .
   eq rega(W) = bits(W,23,16) .
   *** eq rega(W) = bits(W,15,8) .
   eq regb(W) = bits(W,15,8) .
   *** eq regb(W) = bits(W,23,16) .
   eq regc(W) = bits(W,7,0) .
   *** eq regc(W) = bits(W,31,24) .

   *** truncate the last 32 bits/8 bits resp
   eq V + W = bits(V ++ W,31,0) .
   eq A + B = bits(A ++ B,7,0) .
   eq V gt W = V > W .
   eq A gt B = A > B .
   eq V * W = bits(V ** W,31,0) .
   eq A * B = bits(A ** B,7,0) .
   eq ! V = bits(not V,31,0) .
   eq ! A = bits(not A,7,0) .
   eq V & W = bits(V and W,31,0) .
   eq A & B = bits(A and B,7,0) .
   eq V | W = bits(V or W,31,0) .
   eq A | B = bits(A or B,7,0) .
   eq V << W = bits(V sl W,31,0) .
   eq A << B = bits(A sl B,7,0) .

   *** constant four to jump to the next instruction
   eq Four    = 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0
                0 0 0 0 0 1 0 0 .

endfm

***
*** Module for representing memory. Words are 32 bits.
***
fmod MEM is
   protecting MACHINE-WORD .

   sorts Mem .

   op _[_] : Mem Word -> Word .          *** read
   op _[_/_] : Mem Word Word -> Mem .     *** write

   var M : Mem .

   var A B : Word .
   var W : Word .

   eq M[W / A][A] = W .
   eq M[W / A][B] = M[B] [owise] .
endfm

***
*** Module for representing registers.
***
```

```
fmod REG is
   protecting MACHINE-WORD .

   sorts Reg .

   op _[_] : Reg OpField -> Word .          *** read
   op _[_/_] : Reg Word OpField -> Reg .    *** write

   var R : Reg .
   var A B : OpField .
   var W : Word .

   eq R[W / A][A] = W .
   eq R[W / A][B] = R[B] [owise] .
endfm

***
*** Instruction definitions
***
fmod INSTRUCTION-SET is
   protecting MACHINE-WORD .

   ops ADD MULT AND OR NOT : -> OpField .
   ops SLL LD ST EQ GT JMP NOP : -> OpField .
   op NOPWORD : -> Word .

   *** define the opcodes
   eq ADD    = 0 0 0 0 0 0 0 0 1 .
   eq MULT   = 0 0 0 0 0 0 0 1 0 .
   eq AND    = 0 0 0 0 0 0 0 1 1 .
   eq OR     = 0 0 0 0 0 0 1 0 0 .
   eq NOT    = 0 0 0 0 0 0 1 0 1 .
   eq SLL    = 0 0 0 0 0 0 1 1 0 .
   eq LD     = 0 0 0 0 0 0 1 1 1 .
   eq ST     = 0 0 0 0 0 1 0 0 0 .
   eq EQ     = 0 0 0 0 0 1 0 0 1 .
   eq GT     = 0 0 0 0 0 1 0 1 0 .
   eq JMP    = 0 0 0 0 0 1 0 1 1 .
   eq NOP    = 0 0 0 0 0 0 0 0 0 .

   eq NOPWORD = 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 .

endfm

***
*** Functional Units
***
fmod FUNCTIONAL-UNITS is
   protecting MEM .
   protecting REG .
   protecting INSTRUCTION-SET .

   sort FeState .
   sort ExState .
   sort WbState .

*** FETCH OPS
```

```
   op (_,_,_,_) : Mem Word Word Word -> FeState .

   op pm_ : FeState -> Mem .
   ops pc_ cir_ pir_ : FeState -> Word .

   op feu : Int FeState ExState WbState -> FeState .

   op fenext : FeState ExState WbState -> FeState .

*** EXECUTE OPS

   *** Result, Taken Flag, WBFlag, MemWBLoc, RegWBLoc
   op (_,_,_,_,_) : Word Bool Int Word OpField -> ExState .

   op exu : Int FeState ExState WbState -> ExState .

   op exnext : FeState ExState WbState -> ExState .

   ops result_ memwbloc_ : ExState -> Word .
   op taken_ : ExState -> Bool .
   op wbflag_ : ExState -> Int .
   op regwbloc : ExState -> OpField .

*** WRITEBACK OPS

   op (_,_) : Mem Reg -> WbState .

   op dm_ : WbState -> Mem .
   op reg_ : WbState -> Reg .

   op wbu : Int FeState ExState WbState -> WbState .

   op wbnext : FeState ExState WbState -> WbState .

*** FETCH VARIABLES

   var PM : Mem .
   var PC PIR CIR : Word .

   var feS : FeState . *** state
   var T : Int .    *** time

*** EXECUTE VARIABLES

   var TAKEN : Bool .
   var WBFLAG : Int . *** 0 = no writeback, 1 = mem, 2 = reg
   var RESULT MEMWBLOC : Word .
   var REGWBLOC : OpField .

   var exS : ExState . *** state

*** WRITEBACK VARIABLES

   var wbS : WbState . *** state
   var DM : Mem .
   var REG : Reg .

*** FETCH EQUATIONS

   eq pm(PM,PC,CIR,PIR) = PM .
   eq pc(PM,PC,CIR,PIR) = PC .
   eq cir(PM,PC,CIR,PIR) = CIR .
   eq pir(PM,PC,CIR,PIR) = PIR .
```

```
   *** iterated map
   eq feu(0,feS,exS,wbS) = feS .
   eq feu(T,feS,exS,wbS) = fenext(feu(T - 1,feS,exS,wbS),exu(T - 1,feS,exS,wbS),wbu(T - 1,feS,exS,wbS)) [owise] .

   ceq fenext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) = (PM,PC + Four,PM[PC],CIR)
      if opcode(cir(PM,PC,CIR,PIR)) =/= JMP .
   ceq fenext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) = (PM,PC + Four,PM[PC],CIR)
      if opcode(cir(PM,PC,CIR,PIR)) == JMP and taken(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) == false .
   ceq fenext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) = (PM,REG[regc(CIR)],PM[REG[regc(CIR)]],CIR
      if opcode(cir(PM,PC,CIR,PIR)) == JMP and taken(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) == true .

*** EXECUTE EQUATIONS

   eq result(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) = RESULT .
   eq taken(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) = TAKEN .
   eq wbflag(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) = WBFLAG .
   eq memwbloc(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) = MEMWBLOC .
   eq regwbloc(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) = REGWBLOC .

   *** iterated map
   eq exu(0,feS,exS,wbS) = exS .
   eq exu(T,feS,exS,wbS) = exnext(feu(T - 1,feS,exS,wbS),exu(T - 1,feS,exS,wbS),wbu(T - 1,feS,exS,wbS)) [owise] .

   *** define instructions

   *** NOP (opcode = 0)
   ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) = (NOPWORD,false,0,NOPWORD,NOP)
      if opcode(cir(PM,PC,CIR,PIR)) == NOP .
   *** ADD (opcode = 1)
   ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
      (REG[rega(CIR)] + REG[regb(CIR)],false,2,NOPWORD,REG[regc(CIR)])
      if opcode(cir(PM,PC,CIR,PIR)) == ADD .
   *** MULT (opcode = 10)
   ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
      (REG[rega(CIR)] * REG[regb(CIR)],false,2,NOPWORD,REG[regc(CIR)])
      if opcode(cir(PM,PC,CIR,PIR)) == MULT .
   *** AND (opcode = 11)
   ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
      (REG[rega(CIR)] & REG[regb(CIR)],false,2,NOPWORD,REG[regc(CIR)])
      if opcode(cir(PM,PC,CIR,PIR)) == AND .
   *** OR (opcode = 100)
   ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
      (REG[rega(CIR)] | REG[regb(CIR)],false,2,NOPWORD,REG[regc(CIR)])
      if opcode(cir(PM,PC,CIR,PIR)) == OR .
   *** NOT (opcode = 101)
   ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
      (!(REG[rega(CIR)]),false,2,NOPWORD,REG[regc(CIR)])
      if opcode(cir(PM,PC,CIR,PIR)) == NOT .
   *** SLL (opcode = 110)
   ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
      (REG[rega(CIR)] << REG[regb(CIR)],false,2,NOPWORD,REG[regc(CIR)])
      if opcode(cir(PM,PC,CIR,PIR)) == SLL .
   *** LD (opcode = 111)
   ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
      (DM[REG[rega(CIR)] + REG[regb(CIR)]],false,2,NOPWORD,REG[regc(CIR)])
      if opcode(cir(PM,PC,CIR,PIR)) == LD .
   *** ST (opcode = 1000)
   ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
      (REG[rega(CIR)] + REG[regb(CIR)],false,1,REG[regc(CIR)],NOP)
```

```
      if opcode(cir(PM,PC,CIR,PIR)) == ST .
   *** EQ (opcode = 1001) [RA == RB]
   ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
      (constzero32,false,2,NOPWORD,REG[regc(CIR)])
      if opcode(cir(PM,PC,CIR,PIR)) == EQ and REG[rega(CIR)] == REG[regb(CIR)] .
   *** EQ (opcode = 1001) [RA =/= RB]
   ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
      (constminus1,false,2,NOPWORD,REG[regc(CIR)])
      if opcode(cir(PM,PC,CIR,PIR)) == EQ and REG[rega(CIR)] =/= REG[regb(CIR)] .
   *** GT (opcode = 1010) [RA > RB]
   ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
      (constzero32,false,2,NOPWORD,REG[regc(CIR)])
      if opcode(cir(PM,PC,CIR,PIR)) == GT and REG[rega(CIR)] gt REG[regb(CIR)] .
   *** GT (opcode = 1010) [RA <= RB]
   ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
      (constminus1,false,2,NOPWORD,REG[regc(CIR)])
      if opcode(cir(PM,PC,CIR,PIR)) == GT and not REG[rega(CIR)] gt REG[regb(CIR)] .
   *** JMP (opcode = 1011) [branch not taken]
   ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
      (constzero32,false,0,NOPWORD,NOP)
      if opcode(cir(PM,PC,CIR,PIR)) == JMP and REG[rega(CIR)] =/= REG[constzero8] .
   *** JMP (opcode = 1011) [branch taken]
   ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
      (PC + Four,true,2,NOPWORD,REG[regb(CIR)])
      if opcode(cir(PM,PC,CIR,PIR)) == JMP and REG[rega(CIR)] == REG[constzero8] .

*** WRITEBACK EQUATIONS

   eq dm(DM,REG) = DM .
   eq reg(DM,REG) = REG .

   eq wbu(0,feS,exS,wbS) = wbS .
   eq wbu(T,feS,exS,wbS) = wbnext(feu(T - 1,feS,exS,wbS),exu(T - 1,feS,exS,wbS),wbu(T - 1,feS,exS,wbS)) [owise] .

   ceq wbnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
      (DM[result(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) / memwbloc(RESULT,TAKEN,WBFLAG,MEMWBLOC,REG
      if wbflag(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) == 1 .
   ceq wbnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
      (DM,REG[result(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) / regwbloc(RESULT,TAKEN,WBFLAG,MEMWBLOC,
      if wbflag(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) == 2 .

endfm

***
*** Pipeline
***
*** This is the "main" function, where we define the state funtion pmp and the
*** next-state function pnext.
***
fmod PMP is
    protecting FUNCTIONAL-UNITS .

   sort PState .

   op (_,_,_) : FeState ExState WbState -> PState .
   op pmp : Int PState -> PState .
   op pnext : PState -> PState .

   op fetchunit_ : PState -> FeState .
   op executeunit_ : PState -> ExState .
```

```
   op writebackunit_ : PState -> WbState .

   var FETCHUNIT : FeState .
   var EXECUTEUNIT : ExState .
   var WRITEBACKUNIT : WbState .

   var PMP : PState .
   var Time : Int .

   eq fetchunit(FETCHUNIT,EXECUTEUNIT,WRITEBACKUNIT) = FETCHUNIT .
   eq executeunit(FETCHUNIT,EXECUTEUNIT,WRITEBACKUNIT) = EXECUTEUNIT .
   eq writebackunit(FETCHUNIT,EXECUTEUNIT,WRITEBACKUNIT) = WRITEBACKUNIT .

   eq pmp(0,PMP) = PMP .
   eq pmp(Time,PMP) = pnext(pmp(Time - 1,PMP)) [owise] .

   eq pnext(FETCHUNIT,EXECUTEUNIT,WRITEBACKUNIT) =
      (fenext(FETCHUNIT,EXECUTEUNIT,WRITEBACKUNIT),
      exnext(FETCHUNIT,EXECUTEUNIT,WRITEBACKUNIT),
      wbnext(FETCHUNIT,EXECUTEUNIT,WRITEBACKUNIT)) .
endfm

***
*** The final module is to define an actual program and run it.
***
fmod RUNPROGS is
   protecting PMP .      *** import the microprocessor representation

   ops Dm Pm : -> Mem .
   op Rg : -> Reg .
   ops Pc Pir Cir Res : -> Word .
   op Tk : -> Bool .
   op Wflag : -> Int .
   op Wmemloc : -> Word .
   op Wregloc : -> OpField .

   *** Set the values to zero
   eq Pc = constzero32 .
   eq Cir = constzero32 .
   eq Pir = constzero32 .
   eq Res = constzero32 .
   eq Wmemloc = constzero32 .
   eq Tk = false .
   eq Wflag = 0 .
   eq Wregloc = constzero8 .

   *** R1 = 1
   eq Rg[0 0 0 0 0 0 0 1] =
         0 0 0 0 0 0 0 0
         0 0 0 0 0 0 0 0
         0 0 0 0 0 0 0 0
         0 0 0 0 0 0 0 1 .

   *** R2 = 0
   eq Rg[0 0 0 0 0 0 1 0] =
         0 0 0 0 0 0 0 0
         0 0 0 0 0 0 0 0
         0 0 0 0 0 0 0 0
         0 0 0 0 0 0 0 0 .
```

```
*** R13 = 252
eq Rg[0 0 0 0 1 1 0 1] =
      0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0
      1 1 1 1 1 1 0 0 .

*** Mem[1] = 6
eq Dm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1] =
      0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0
      0 0 0 0 0 1 1 0 .

*** Mem[2] = 5
eq Dm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0] =
      0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0
      0 0 0 0 0 1 0 1 .

*** INST 1 [ Load (RG1+RG1) into RG3 ] -> Mem[2] = 5
eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0] =
      0 0 0 0 0 1 1 1 *** LOAD
      0 0 0 0 0 0 0 1 *** RG1
      0 0 0 0 0 0 0 1 *** RG1
      0 0 0 0 0 0 1 1 . *** RG3

*** INST 2 [ Load (RG1+RG2) into RG4 ] -> Mem[2] = 5
eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0] =
      0 0 0 0 0 1 1 1 *** LOAD
      0 0 0 0 0 0 0 1 *** RG1
      0 0 0 0 0 0 0 1 *** RG1
      0 0 0 0 0 1 0 0 . *** RG4

*** INST 3 [ Shift left R3 by R4 and store in R5 ]
eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0] =
      0 0 0 0 0 1 1 0 *** SHIFTL
      0 0 0 0 0 0 1 1 *** RG3
      0 0 0 0 0 1 0 0 *** RG4
      0 0 0 0 0 1 0 1 . *** RG5

*** INST 4 [ Store RG5 in Mem[RG2+RG3] ] -> Mem[5] = 160
eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0] =
      0 0 0 0 1 0 0 0 *** STORE
      0 0 0 0 0 0 1 0 *** RG2
      0 0 0 0 0 0 1 1 *** RG3
      0 0 0 0 0 1 0 1 . *** RG5

*** INST 5 [ Add RG3 and RG4 and store in RG6 ] -> RG[6] = 10
eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0] =
      0 0 0 0 0 0 0 1 *** ADD
      0 0 0 0 0 0 1 1 *** RG3
      0 0 0 0 0 1 0 0 *** RG4
      0 0 0 0 0 1 1 0 . *** RG6

*** INST 6 [ Mult RG3 by RG4 and store in RG7 ] -> RG[7] = 25
eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0] =
      0 0 0 0 0 0 1 0 *** MULT
```

```
      0 0 0 0 0 0 1 1 *** RG3
      0 0 0 0 0 1 0 0 *** RG4
      0 0 0 0 0 1 1 1 . *** RG7

*** INST 7 [ Bitwise and of RG3 and RG4, stored in RG8 ] -> RG[8] = 5
eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0] =
      0 0 0 0 0 0 1 1 *** AND
      0 0 0 0 0 0 1 1 *** RG3
      0 0 0 0 0 1 0 0 *** RG4
      0 0 0 0 1 0 0 0 . *** RG8

*** INST 8 [ Bitwise or of RG3 and RG4, stored in RG9 ] -> RG[9] = 5
eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0] =
      0 0 0 0 0 1 0 0 *** OR
      0 0 0 0 0 0 1 1 *** RG3
      0 0 0 0 0 1 0 0 *** RG4
      0 0 0 0 1 0 0 1 . *** RG9

*** INST 9 [ Inverse of RG3, stored in R10 ]
eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0] =
      0 0 0 0 0 1 0 1 *** NOT
      0 0 0 0 0 0 1 1 *** RG3
      0 0 0 0 0 0 0 0 *** RG0
      0 0 0 0 1 0 1 0 . *** RG10

*** INST 10 [ Test if RG7 = RG8 ] -> RG[7] = 25, RG[8] = 5, Answer = -1 (false)
eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0] =
      0 0 0 0 1 0 0 1 *** EQ
      0 0 0 0 0 1 1 1 *** RG7
      0 0 0 0 1 0 0 0 *** RG8
      0 0 0 0 1 0 1 1 . *** R11

*** INST 11 [ Test if R11 == 0 ] -> false, no jump
eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0] =
      0 0 0 0 1 0 1 1 *** JMP
      0 0 0 0 1 0 1 1 *** R11
      0 0 0 0 1 1 0 0 *** R12
      0 0 0 0 1 1 0 1 . *** R13

*** INST 12 [ Test if RG7 > RG8 ] -> RG[7] = 25, RG[8] = 5, Answer = 0 (true)
eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 0] =
      0 0 0 0 1 0 1 0 *** GT
      0 0 0 0 0 1 1 1 *** RG7
      0 0 0 0 1 0 0 0 *** RG8
      0 0 0 0 1 0 1 1 . *** RG11

*** INST 13 [ Test if RG11 == 0 ] -> true, jump to R13, store PC+4 in RG12
eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0] =
      0 0 0 0 1 0 1 1 *** JMP
      0 0 0 0 1 0 1 1 *** RG11
      0 0 0 0 1 1 0 0 *** RG12
      0 0 0 0 1 1 0 1 . *** RG13

*** INST 14 [ Subroutine ][ Add RG3 to RG4 and store in RG14 ] -> R[14] = 10
eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0] =
      0 0 0 0 0 0 0 1 *** ADD
      0 0 0 0 0 0 1 1 *** RG3
      0 0 0 0 0 1 0 0 *** RG4
      0 0 0 0 1 1 1 0 . *** RG14
```

```
   *** INST 15 [ Test if R11 == 0] -> True, jump back to R[12]
   eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0] =
         0 0 0 0 1 0 1 1 *** JMP
         0 0 0 0 1 0 1 1 *** RG11
         0 0 0 0 1 1 1 1 *** RG15
         0 0 0 0 1 1 0 0 . *** RG12

   *** INST 16 [ Return from subroutine ] [ Add R7 to R8 and store in R16 ] -> R[16] = 25 + 5 =30
   eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 0] =
         0 0 0 0 0 0 0 0 *** ADD
         0 0 0 0 0 1 1 1 *** RG7
         0 0 0 0 1 0 0 0 *** RG8
         0 0 0 1 0 0 0 0 . *** R16
endfm

***
*** Now run the program
***
reduce pmp(1, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(2, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(3, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(4, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(5, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(6, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(7, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(8, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(9, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(10, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(11, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(12, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(13, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(14, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(15, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(16, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
q
```