```
***
*** "A 32-bit Generic RISC Microprocessor Specification"
*** Sean Handley, sean.handley@gmail.com
*** 2006-08-02
***

***
*** 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 .

    *** 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 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 .

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

    *** 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 _+8_ : Word Word -> Word .
    op _+8_ : 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 .

    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) .
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] . *** seek if not found
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] . *** seek if not found
endfm

***
*** State of SPM, together with tupling and projection functions
***

fmod SPM-STATE is
    protecting MEM .
    protecting REG .

    sort SPMstate .

    op (_,_,_,_) : Mem Mem Word Reg -> SPMstate .

    *** project out program and data mem
    ops mp_ md_ : SPMstate -> Mem .

    *** project out PC
    op pc_ : SPMstate -> Word .

    *** project out regs
    op reg_ : SPMstate -> Reg .

    var S : SPMstate .
    vars MP MD : Mem .
    var PC : Word .
    var REG : Reg .

    *** tuple member accessor functions
    eq mp(MP,MD,PC,REG) = MP .
    eq md(MP,MD,PC,REG) = MD .
    eq pc(MP,MD,PC,REG) = PC .
    eq reg(MP,MD,PC,REG) = REG .
endfm

***
*** SPM
***
*** This is the "main" function, where we define the state funtion spm and the
*** next-state function next.
***
```

```
fmod SPM is
    protecting SPM-STATE .

    ops ADD32 MULT AND OR NOT : -> OpField .
    ops SLL LD32 ST32 EQ GT JMP : -> OpField .
    op Four : -> Word .

    op spm : Int SPMstate -> SPMstate .

    op next : SPMstate -> SPMstate .

    var SPM : SPMstate .
    var T : Int .
    var MP MD : Mem .
    var PC A : Word .
    var REG : Reg .
    var O P : OpField .

    *** define the opcodes
    eq ADD32 = 0 0 0 0 0 0 0 0 .
    eq MULT = 0 0 0 0 0 0 1 0 .
    eq AND = 0 0 0 0 0 0 1 1 .
    eq OR = 0 0 0 0 0 1 0 0 .
    eq NOT = 0 0 0 0 0 1 0 1 .
    eq SLL = 0 0 0 0 0 1 1 0 .
    eq LD32 = 0 0 0 0 0 1 1 1 .
    eq ST32 = 0 0 0 0 1 0 0 0 .
    eq EQ = 0 0 0 0 1 0 0 1 .
    eq GT = 0 0 0 0 1 0 1 0 .
    eq JMP = 0 0 0 0 1 0 1 1 .

    *** 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 .

    eq spm(0,SPM) = SPM .
    eq spm(T,SPM) = next(spm(T - 1,SPM)) [owise] .

    *** Fix the zero register
    eq REG[0 0 0 0 0 0 0 0] = constzero32 .
    ceq REG[A / O][O] = constzero32 if O == constzero8 .
    eq REG[A / O][P] = REG[P] [owise].

    *** define instructions

    *** ADD32 (opcode = 0)
    ceq next(MP,MD,PC,REG) = (MP, MD, PC + Four,
        REG[REG[rega(MP[PC])] + REG[regb(MP[PC])] / regc(MP[PC])])
        if opcode(MP[PC]) == ADD32 .
    *** MULT (opcode = 10)
    ceq next(MP,MD,PC,REG) = (MP, MD, PC + Four,
        REG[REG[rega(MP[PC])] * REG[regb(MP[PC])] / regc(MP[PC])])
        if opcode(MP[PC]) == MULT .
    *** AND (opcode = 11)
    ceq next(MP,MD,PC,REG) = (MP, MD, PC + Four,
        REG[REG[rega(MP[PC])] & REG[regb(MP[PC])] / regc(MP[PC])])
```

```
        if opcode(MP[PC]) == AND .
    *** OR (opcode = 100)
    ceq next(MP,MD,PC,REG) = (MP, MD, PC + Four,
        REG[REG[rega(MP[PC])] | REG[regb(MP[PC])] / regc(MP[PC])])
        if opcode(MP[PC]) == OR .
    *** NOT (opcode = 101)
    ceq next(MP,MD,PC,REG) = (MP, MD, PC + Four,
        REG[!(REG[rega(MP[PC])]) / regc(MP[PC])])
        if opcode(MP[PC]) == NOT .
    *** SLL (opcode = 110)
    ceq next(MP,MD,PC,REG) = (MP, MD, PC + Four,
        REG[REG[rega(MP[PC])] << REG[regb(MP[PC])] / regc(MP[PC])])
        if opcode(MP[PC]) == SLL .
    *** LD32 (opcode = 111)
    ceq next(MP,MD,PC,REG) = (MP, MD, PC + Four,
        REG[MD[REG[rega(MP[PC])] + REG[regb(MP[PC])]] / regc(MP[PC])])
        if opcode(MP[PC]) == LD32 .
    *** ST32 (opcode = 1000)
    ceq next(MP,MD,PC,REG) = (MP,
        MD[REG[regc(MP[PC])] / (REG[rega(MP[PC])] + REG[regb(MP[PC])])], PC + Four, REG)
        if opcode(MP[PC]) == ST32 .
    *** EQ (opcode = 1001) [RA == RB]
    ceq next(MP,MD,PC,REG) = (MP, MD, PC + Four,
        REG[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 / regc(MP[PC])])
        if opcode(MP[PC]) == EQ and REG[rega(MP[PC])] == REG[regb(MP[PC])] .
    *** EQ (opcode = 1001) [RA =/= RB]
    ceq next(MP,MD,PC,REG) = (MP, MD, PC + Four,
        REG[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 / regc(MP[PC])])
        if opcode(MP[PC]) == EQ and REG[rega(MP[PC])] =/= REG[regb(MP[PC])] .
    *** GT (opcode = 1010) [RA > RB]
    ceq next(MP,MD,PC,REG) = (MP, MD, PC + Four,
        REG[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 / regc(MP[PC])])
        if opcode(MP[PC]) == GT and REG[rega(MP[PC])] gt REG[regb(MP[PC])] .
    *** GT (opcode = 1010) [RA <= RB]
    ceq next(MP,MD,PC,REG) = (MP, MD, PC + Four,
        REG[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 / regc(MP[PC])])
        if opcode(MP[PC]) == GT and not (REG[rega(MP[PC])] gt REG[regb(MP[PC])]) .
    *** JMP (opcode = 1011) [branch not taken]
    ceq next(MP,MD,PC,REG) = (MP, MD, PC + Four,REG)
        if opcode(MP[PC]) == JMP and REG[rega(MP[PC])] =/= REG[0 0 0 0 0 0 0 0] .
    *** JMP (opcode = 1011) [branch taken]
    ceq next(MP,MD,PC,REG) = (MP,MD,
        REG[regc(MP[PC])], REG[PC + Four / regb(MP[PC])])
        if opcode(MP[PC]) == JMP and REG[rega(MP[PC])] == REG[0 0 0 0 0 0 0 0] .
endfm

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

    ops Md Mp : -> Mem .
    op Rg : -> Reg .
    op Pc : -> Word .

    *** Set the PC to zero
```

```
eq Pc = 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 .

*** 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 Md[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 Md[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 Mp[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 Mp[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 Mp[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 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 Mp[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 Mp[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 *** 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 Mp[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 Mp[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 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 Mp[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 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 Mp[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 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 Mp[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 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 Mp[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 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 Mp[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 Mp[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 Mp[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 0 *** 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 Mp[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 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 Mp[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 spm(1, (Mp,Md,Pc,Rg)) .
reduce spm(2, (Mp,Md,Pc,Rg)) .
reduce spm(3, (Mp,Md,Pc,Rg)) .
reduce spm(4, (Mp,Md,Pc,Rg)) .
reduce spm(5, (Mp,Md,Pc,Rg)) .
reduce spm(6, (Mp,Md,Pc,Rg)) .
reduce spm(7, (Mp,Md,Pc,Rg)) .
reduce spm(8, (Mp,Md,Pc,Rg)) .
reduce spm(9, (Mp,Md,Pc,Rg)) .
reduce spm(10, (Mp,Md,Pc,Rg)) .
reduce spm(11, (Mp,Md,Pc,Rg)) .
reduce spm(12, (Mp,Md,Pc,Rg)) .
reduce spm(13, (Mp,Md,Pc,Rg)) .
reduce spm(14, (Mp,Md,Pc,Rg)) .
reduce spm(15, (Mp,Md,Pc,Rg)) .
reduce spm(16, (Mp,Md,Pc,Rg)) .

q
```