course: CA

exercise: 2

date: 2021-12-29

author: Haotian Shi, 2520366S

author: Jiwei Ma, 2638638M

author: Jiyuan He, 2682781H

All parts of exercise have been completed successfully. It compiles and works.

$ghci

$:load run

Test loadxi

$:main programs/Loadxi

$run

Test mul

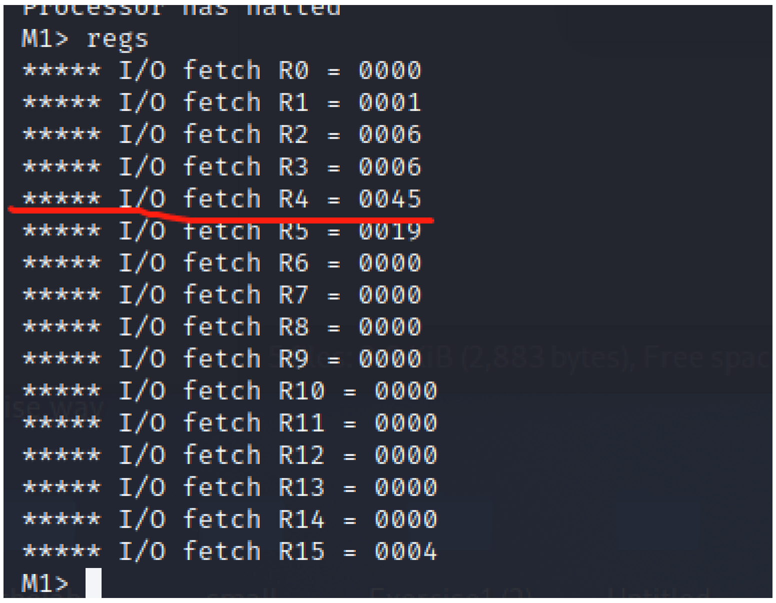
$:main programs/Mul

$run

Extraction from simulation output:

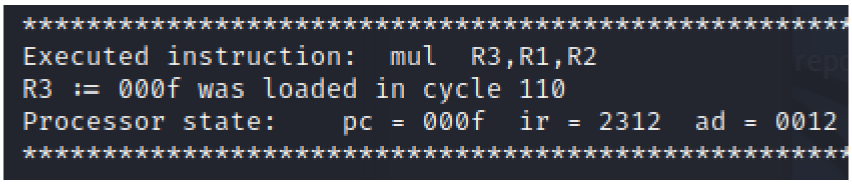
Loadxi:

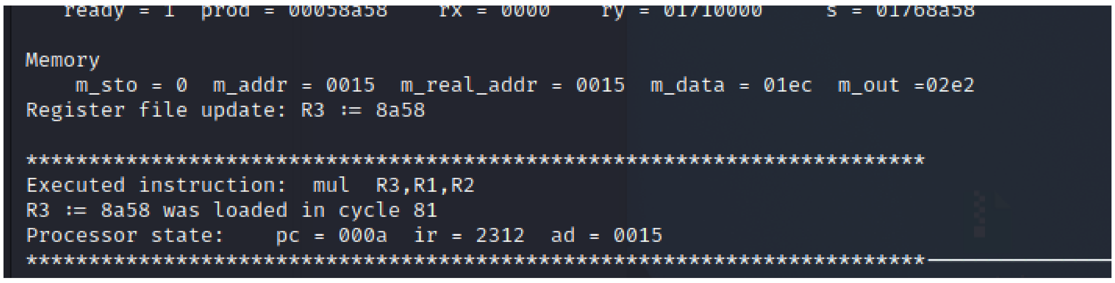
Regs command showing R4 is the sum value. R3 is the index register, you can see it is set to 6 since the array length is 6.



```
Processor has halted
M1> regs
***** I/O fetch R0 = 0000
***** I/O fetch R1 = 0001
***** I/O fetch R2 = 0006
***** I/O fetch R3 = 0006
***** I/O fetch R4 = 0045
***** I/O fetch R5 = 0019
***** I/O fetch R6 = 0000
***** I/O fetch R7 = 0000
***** I/O fetch R8 = 0000
***** I/O fetch R9 = 0000
***** I/O fetch R10 = 0000
***** I/O fetch R11 = 0000
***** I/O fetch R12 = 0000
***** I/O fetch R13 = 0000
***** I/O fetch R14 = 0000
***** I/O fetch R15 = 0004
M1>
```

Mul:

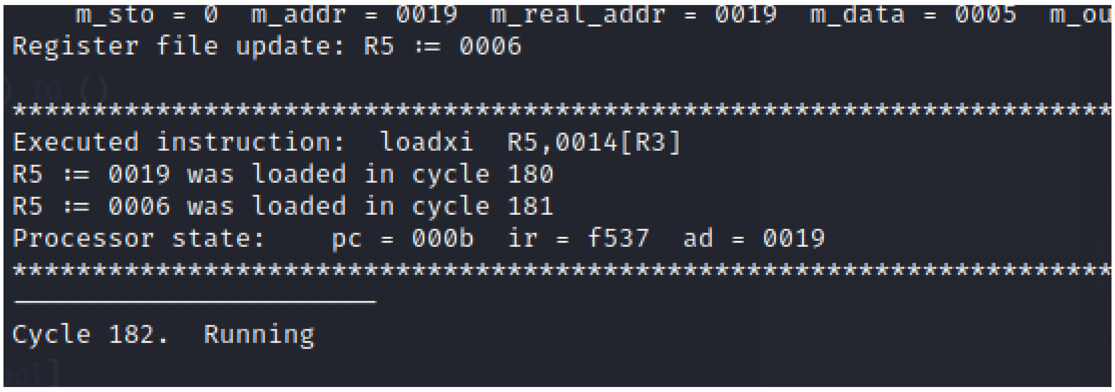

```
******************************************************
Executed instruction:  mul  R3,R1,R2
R3 := 000f was loaded in cycle 110
Processor state:    pc = 000f  ir = 2312  ad = 0012
******************************************************
```
this is 3 * 5


```
   ready = 1  prod = 00058a58    rx = 0000    ry = 01710000     s = 01768a58

Memory
    m_sto = 0  m_addr = 0015  m_real_addr = 0015  m_data = 01ec  m_out =02e2
Register file update: R3 := 8a58

************************************************************************
Executed instruction:  mul  R3,R1,R2
R3 := 8a58 was loaded in cycle 81
Processor state:    pc = 000a  ir = 2312  ad = 0015
************************************************************************
```

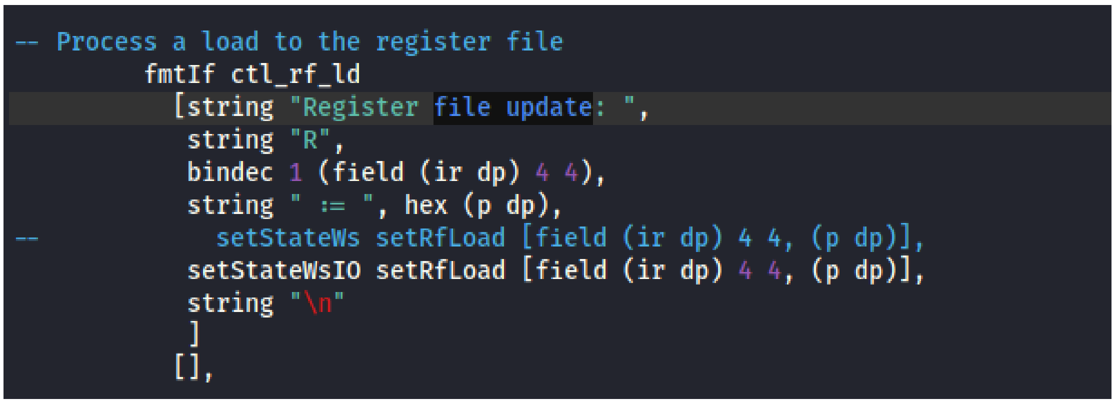
This is 492 * 739 = 58a58(hex), we choose the lower 16bits which is 8a58.

Something I want to highlight during my working:

Before I hand in my work, I notice that when I run loadxi command, it is working well.. but later on, from the simulation output I find one strange point:


```
    m_sto = 0  m_addr = 0019  m_real_addr = 0019  m_data = 0005  m_ou
Register file update: R5 := 0006

*********************************************************************
Executed instruction:  loadxi  R5,0014[R3]
R5 := 0019 was loaded in cycle 180
R5 := 0006 was loaded in cycle 181
Processor state:    pc = 000b  ir = f537  ad = 0019
*********************************************************************
______________________
Cycle 182.  Running
```

Here you can see R5 is updated to 6. But it is strange, the index register R3 should be updated and not the destination register! We think it is the format reason, we locate the format position in Run.hs.


```
-- Process a load to the register file
         fmtIf ctl_rf_ld
           [string "Register file update: ",
            string "R",
            bindec 1 (field (ir dp) 4 4),
            string " := ", hex (p dp),
--            setStateWs setRfLoad [field (ir dp) 4 4, (p dp)],
            setStateWsIO setRfLoad [field (ir dp) 4 4, (p dp)],
            string "\n"
            ]
           [],
```

So here there is
bindec 1 (field (ir dp) 4 4)

The M1 circuit will not think about incrementing the index register so the message about updating index register is never going to appear!!

So what if I change it to

bindec 1 (field (ir dp) 8 4)

which is showing index register updated?

That would be messy because it would affect other RX instruction which update destination register.

The only way I think to make updating index regsiter appear is by defining a new function in Run.hs, but I didn't implement it.

Part 1. Implementing a new instruction: loadxi

Step 1 Control algorithm:

For loadxi. (Control.hs) Like most RX instructions, which is decided by ir_sb. If we add the loadxi instruction, we can add this function to 7.

The effect of executing the instruction is to perform a load, and also to increment the index register. So we can copy 'load' instruction here (st_loadxi0 st_loadxi1 st_loadxi2) and beyond that, we need to add st_loadxi3 to increment the index register. To do that, we need to do 'reg[ir_sa] = reg[ir_sa] + 1', but we don't know what signals we need to assert to achieve that.
7 -> -- loadxi instruction

```
        st_loadxi0:  ad := mem[pc], pc++

          assert [ctl_ma_pc, ctl_ad_ld, ctl_x_pc, ctl_alu_abc=011, ctl_pc_ld]

        st_loadxi1:  ad := reg[ir_sa] + ad

          assert [ctl_y_ad, ctl_alu_abc=000, ctl_ad_ld, ctl_ad_alu]

        st_loadxi2:  reg[ir_d] := mem[ad]

          assert [ctl_rf_ld]

        st_loadxi3: reg[ir_sa] = reg[ir_sa] + 1

          assert []
```

Step 2 Analyse the datapath and figure out how to do the operation:

(Datapath.hs) Now we need to make the datapath perform the operation 'reg[ir_sa] = reg[ir_sa] + 1'.

To perform this, we need to use ALU to perform “x+1”.

ctl_alu_abc = 011

So we need x = a, a = reg[rf_sa] as normal.

And the most important thing is to let destination register to be the index register.

ir_d = mux1w ctl_rf_sa (field ir 4 4) ir_sa

Here we implement a new control signal ctl_rf_sa to let destination register to be index register, so that index register can increment itself. We also introduce a new multiplexer to choose between ctl_rf_sa and ir_sa.

Step 3 Find the required control signals:

Assert the new control signal ctl_rf_sa, to let destination register be index register.

Load control and finally we’ve got

```
7 -> -- loadxi instruction
   st_loadxi0:  ad := mem[pc], pc++
     assert [ctl_ma_pc, ctl_ad_ld, ctl_x_pc, ctl_alu_abc=011, ctl_pc_ld]
   st_loadxi1:  ad := reg[ir_sa] + ad
     assert [ctl_y_ad, ctl_alu_abc=000, ctl_ad_ld, ctl_ad_alu]
   st_loadxi2:  reg[ir_d] := mem[ad]
     assert [ctl_rf_ld]
   st_loadxi3: reg[ir_sa] = reg[ir_sa] + 1
     assert [ctl_alu_abc = 011, ctl_rf_alu, ctl_rf_ld, ctl_rf_ldcc, ctl_rf_sa]
```

Step 4 Create new control signal and states.

(Interface.hs)

Add new control ctl_rf_sa and new control states.

```
data CtlSig a = CtlSig {
...
-- Controls for register file
...
ctl_rf_sa, -- Choose ir as input (if 0, use pc)
```

...

:: a -- all controls are bit signals

}

data CtlState a = CtlState

{dff_instr_fet, st_instr_fet,

... dff_loadxi0, st_loadxi0, dff_loadxi1, st_loadxi1, dff_loadxi2, st_loadxi2, dff_loadxi3, st_loadxi3 ...

Step 5 Implement state transitions:

We need to define the new state flip flops in control.hs

```
-- loadxi control states
     dff_loadxi0 = dff (or2 (pRX!!7) (and2 dff_loadxi0 io_DMA))
     st_loadxi0  = and2 dff_loadxi0 cpu
     dff_loadxi1 = dff (or2 st_loadxi0 (and2 dff_loadxi1 io_DMA))
     st_loadxi1  = and2 cpu dff_loadxi1
     dff_loadxi2 = dff (or2 st_loadxi1 (and2 dff_loadxi2 io_DMA))
     st_loadxi2  = and2 cpu dff_loadxi2
     dff_loadxi3 = dff (or2 st_loadxi2 (and2 dff_loadxi3 io_DMA))
     st_loadxi3  = and2 cpu dff_loadxi3
```

Step 6 Terminate the instruction:

Every instruction has a last state, and when that is reached the control should go next to the top of the instruction fetch–execute cycle.

In control.hs

```
start = orw
        [reset,
         …. st_loadxi3,
         ……]
```

Step 7 Generate the control signals:

Because loadxi is a new instruction so we need to add each control state to generate the control signal again. This includes **st_loadxi0 st_loadxi1 st_loadxi2 st_loadxi3**.

"st_loadxi0 st_loadxi1 st_loadxi2" are same as the 'load' instruction... but for st_loadxi3, it is a new state, so we need to add it additionally.

ctl_rf_ld   = orw [……**st_loadxi3**]

ctl_rf_ldcc = orw [……**st_loadxi3**]

ctl_rf_alu = orw [st_lea1,st_add,st_sub,**st_loadxi3**]

ctl_alu_c   = orw [st_instr_fet,st_load0,st_store0,st_lea0,

st_jump0, st_jumpc00, st_jumpc10,

st_sub,st_jumpc00,st_jal0,st_loadxi0,**st_loadxi3**]

ctl_rf_sa   = orw [st_loadxi3]

ctl_alu_b   = orw [……..st_loadxi3]

ctl_alu_c   = orw [……st_loadxi3]

Step 8 Write test data and object code:

Use loadxi instruction.

arraySum.asm.txt:

```
     lea    R1,1[R0]       ; R1 = constant 1
     load   R2,n[R0]       ; R2 = n
; sum := 0
     add    R4,R0,R0       ; R4 = sum = 0
; i := 0
     add    R3,R0,R0       ; R3 = i = 0

; Top of loop, determine whether to remain in loop
whileloop
; if i >= n then goto done
     cmp    R3,R2          ; compare i, n
```

```
       jumpge done[R0]       ; if i>=n then goto done

; sum := sum + x[i]
       loadxi R5,x[R3]       ; R5 = x[i], i = i + 1
       add    R4,R4,R5       ; sum := sum + R5

; i := i + 1
       add    R0,R0,R0       ; do nothing

; goto whileloop
       jump   whileloop[R0]  ; goto whileloop

done   store  R4,sum[R0]     ; sum := R4
       trap   R0,R0,R0       ; terminate

; Data area
; With the following initial values, the expected result
;   = 18 + -33 + 21 + -2 + 40 + 25
;   = 69 = hex 0045

n       data   6
sum     data   0
x       data  18
        data -33
        data  21
        data  -2
        data  40
        data  25
```

After this assembly code we need change it to obj code. Previously we can use sigma16 to generate obj code for using load instruction instead of loadxi…

So what we need to do is to manually change the obj code to run loadxi instruction!!!

For RX instruction **loadxi R5,x[R3]**

[f] [destination register] [index register] [instruction op code] [displacement]

So we can get

f537,0014

because 7 is used for loadxi in control.hs

Step 9 Update simulation driver:

Finally in run.hs

Update several lines in it. So I just put what I added here.

```
 let ctlStateLookupTable =
       [ ("reset", reset)
.....
       , ("st_loadxi0",    st_loadxi0)
.......
       ]
......
```

Format

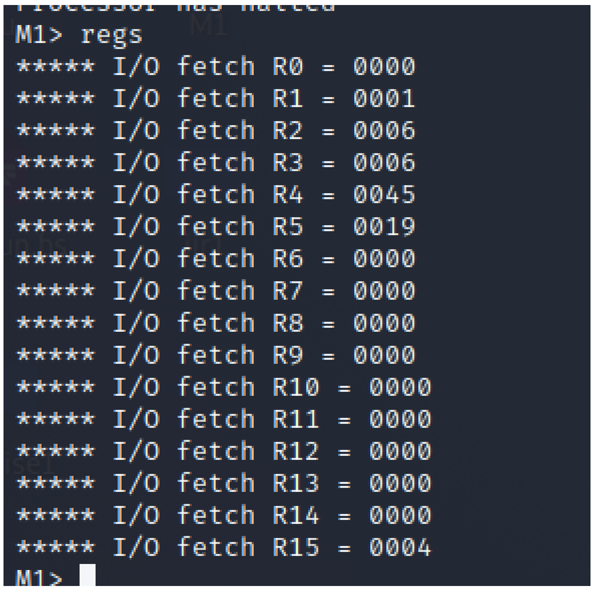
```
   [
   .....
       , string "   st_loadxi0 = ", bit dff_loadxi0, bit st_loadxi0
       , string "   st_loadxi1 = ", bit dff_loadxi1, bit st_loadxi1
       , string "   st_loadxi2 = ", bit dff_loadxi2, bit st_loadxi2
       , string "   st_loadxi3 = ", bit dff_loadxi3, bit st_loadxi3
   ....
        , string "    ctl_rf_sa = ", bit ctl_rf_sa
```

```
 ….
  ]
  …..
  fmtIf (orw [……st_loadxi1])
  ……
  mnemonics_RX =
         [……..
  "loadxi",
          ……..]
```

Step 10 Run the program on improved circuit.

Since we’ve changed the obj code, we can run it now.

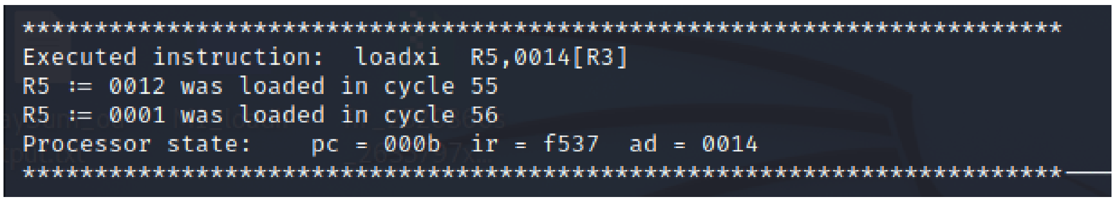
```
$cd M1_loadxi
$ghci
Ghci > : load Run
Ghci > : main programs/arraySum
M1 > run
….
….
M1 > regs
```



```
M1> regs
***** I/O fetch R0 = 0000
***** I/O fetch R1 = 0001
***** I/O fetch R2 = 0006
***** I/O fetch R3 = 0006
***** I/O fetch R4 = 0045
***** I/O fetch R5 = 0019
***** I/O fetch R6 = 0000
***** I/O fetch R7 = 0000
***** I/O fetch R8 = 0000
***** I/O fetch R9 = 0000
***** I/O fetch R10 = 0000
***** I/O fetch R11 = 0000
***** I/O fetch R12 = 0000
***** I/O fetch R13 = 0000
***** I/O fetch R14 = 0000
***** I/O fetch R15 = 0004
M1>
```

R4 is the final sum of all array elements.

Hex 0045 is exactly the answer. It demonstrates my circuit works.



```
************************************************************************
Executed instruction:  loadxi  R5,0014[R3]
R5 := 0012 was loaded in cycle 55
R5 := 0001 was loaded in cycle 56
Processor state:    pc = 000b  ir = f537  ad = 0014
************************************************************************
```

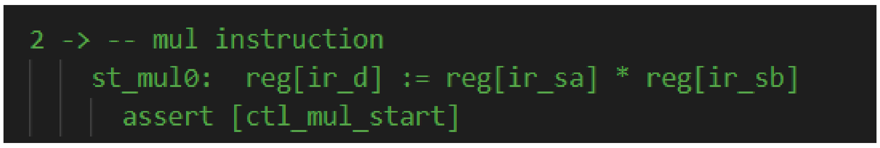
Executed loadxi instruction. Cycle 56.

Part 2. Controlling a functional unit: mul

Step 1 Control algorithm:

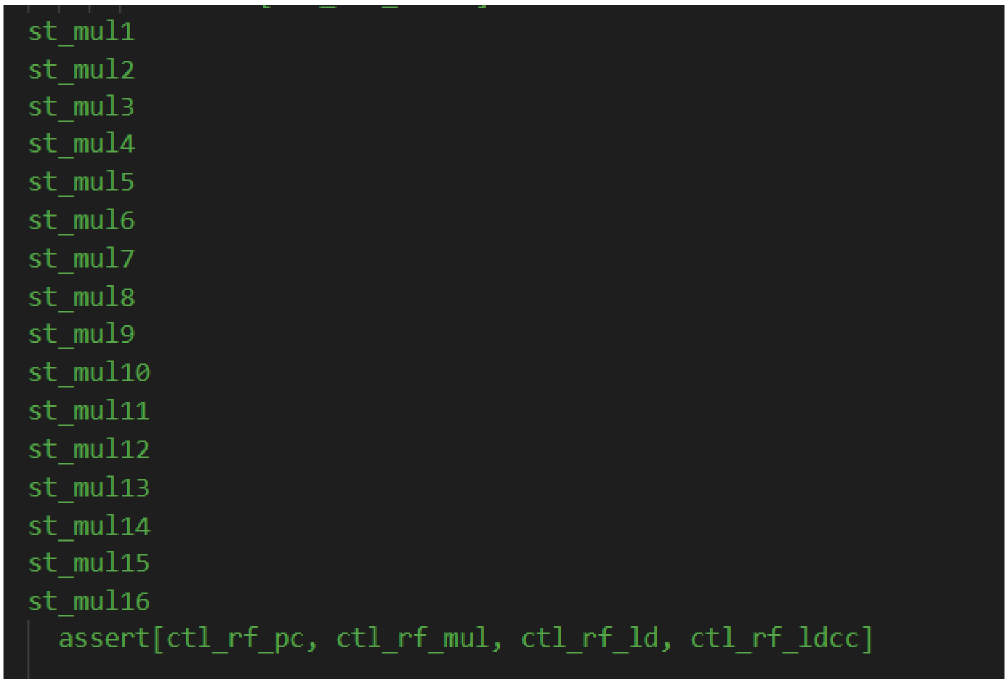
To implement the Multiply in M1 circuit, we write the control algorithm in the comment of the control.hs, and M1 has already provide 2 to implement this function

1. The algorithm which are needed to implement:



```
2 -> -- mul instruction
    st_mul0:  reg[ir_d] := reg[ir_sa] * reg[ir_sb]
      assert [ctl_mul_start]
```

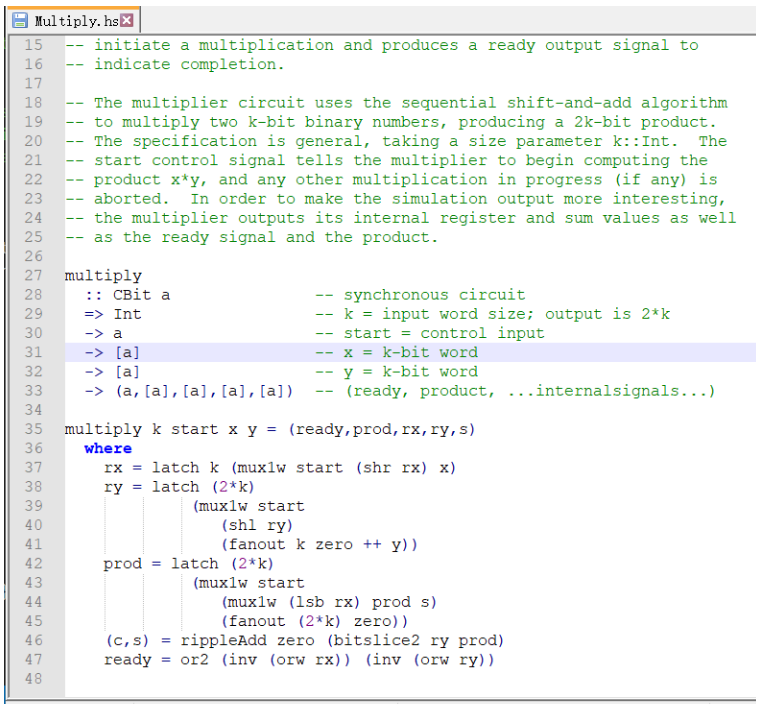
2. Because we need to multiply two 16-bit numbers. In that way we need to introduce 16 control signals to implement our algorithm



```
st_mul1
st_mul2
st_mul3
st_mul4
st_mul5
st_mul6
st_mul7
st_mul8
st_mul9
st_mul10
st_mul11
st_mul12
st_mul13
st_mul14
st_mul15
st_mul16
  assert[ctl_rf_pc, ctl_rf_mul, ctl_rf_ld, ctl_rf_ldcc]
```

Step 2 implementing the combinational logic (DataPath.hs)

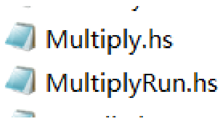
Implementing the combinational logic of k-bit multiplication in Multiply.hs;



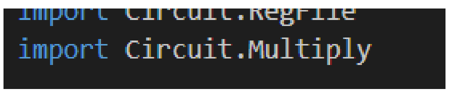
Multiply.hs

```
-- initiate a multiplication and produces a ready output signal to
-- indicate completion.

-- The multiplier circuit uses the sequential shift-and-add algorithm
-- to multiply two k-bit binary numbers, producing a 2k-bit product.
-- The specification is general, taking a size parameter k::Int.  The
-- start control signal tells the multiplier to begin computing the
-- product x*y, and any other multiplication in progress (if any) is
-- aborted.  In order to make the simulation output more interesting,
-- the multiplier outputs its internal register and sum values as well
-- as the ready signal and the product.

multiply
  :: CBit a                 -- synchronous circuit
  => Int                    -- k = input word size; output is 2*k
  -> a                      -- start = control input
  -> [a]                    -- x = k-bit word
  -> [a]                    -- y = k-bit word
  -> (a,[a],[a],[a],[a])    -- (ready, product, ...internalsignals...)

multiply k start x y = (ready,prod,rx,ry,s)
  where
    rx = latch k (mux1w start (shr rx) x)
    ry = latch (2*k)
             (mux1w start
                (shl ry)
                (fanout k zero ++ y))
    prod = latch (2*k)
             (mux1w start
                (mux1w (lsb rx) prod s)
                (fanout (2*k) zero))
    (c,s) = rippleAdd zero (bitslice2 ry prod)
    ready = or2 (inv (orw rx)) (inv (orw ry))
```

import this file in Datapath.hs to add Multiply combinational logic circuit to M1 circuit.

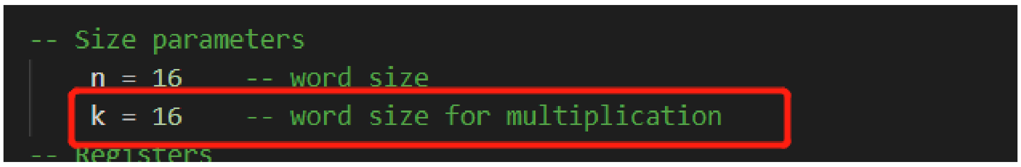


Multiply.hs
MultiplyRun.hs



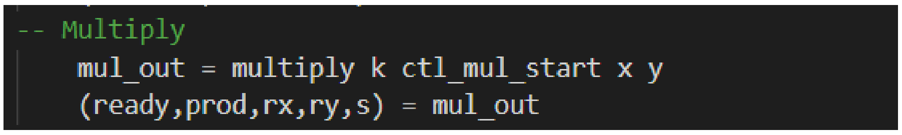
```
import Circuit.RegFile
import Circuit.Multiply
```

Define 16 bits;



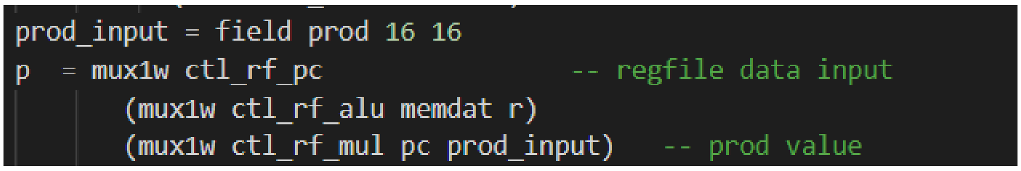
```
-- Size parameters
    n = 16     -- word size
    k = 16     -- word size for multiplication
-- Registers
```

Implement the Multiply algorithm in datapath.hs



```
-- Multiply
    mul_out = multiply k ctl_mul_start x y
    (ready,prod,rx,ry,s) = mul_out
```

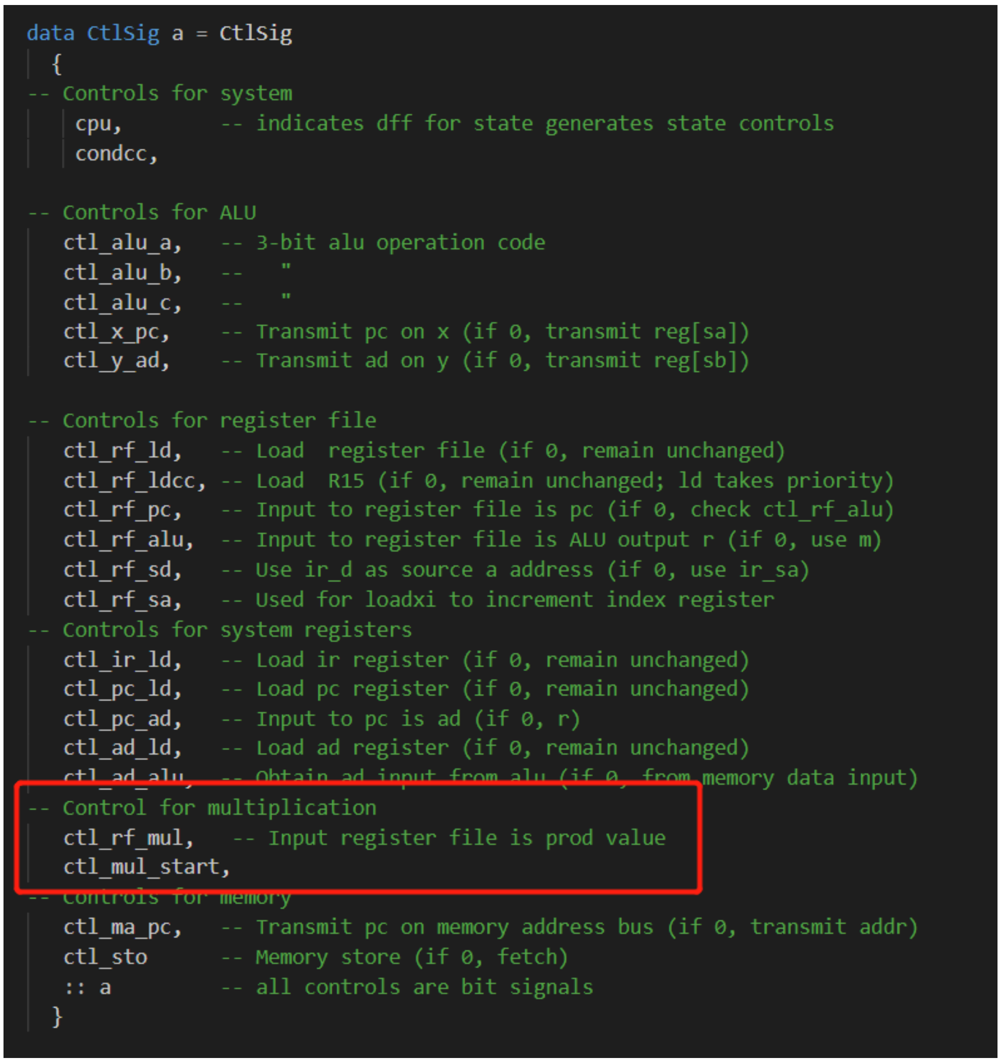
Adjust the registers, The lower 16 bits of Calculation results are output to Reg4 as a result:



```
prod_input = field prod 16 16
p   = mux1w ctl_rf_pc                    -- regfile data input
       (mux1w ctl_rf_alu memdat r)
       (mux1w ctl_rf_mul pc prod_input)    -- prod value
```

Step3 Create new control signal and states

Add control signal:



```
data CtlSig a = CtlSig
  {
-- Controls for system
    cpu,          -- indicates dff for state generates state controls
    condcc,

-- Controls for ALU
   ctl_alu_a,    -- 3-bit alu operation code
   ctl_alu_b,    --    "
   ctl_alu_c,    --    "
   ctl_x_pc,     -- Transmit pc on x (if 0, transmit reg[sa])
   ctl_y_ad,     -- Transmit ad on y (if 0, transmit reg[sb])

-- Controls for register file
   ctl_rf_ld,    -- Load  register file (if 0, remain unchanged)
   ctl_rf_ldcc,  -- Load  R15 (if 0, remain unchanged; ld takes priority)
   ctl_rf_pc,    -- Input to register file is pc (if 0, check ctl_rf_alu)
   ctl_rf_alu,   -- Input to register file is ALU output r (if 0, use m)
   ctl_rf_sd,    -- Use ir_d as source a address (if 0, use ir_sa)
   ctl_rf_sa,    -- Used for loadxi to increment index register
-- Controls for system registers
   ctl_ir_ld,    -- Load ir register (if 0, remain unchanged)
   ctl_pc_ld,    -- Load pc register (if 0, remain unchanged)
   ctl_pc_ad,    -- Input to pc is ad (if 0, r)
   ctl_ad_ld,    -- Load ad register (if 0, remain unchanged)
   ctl_ad_alu,   -- Obtain ad input from alu (if 0, from memory data input)
-- Control for multiplication
   ctl_rf_mul,    -- Input register file is prod value
   ctl_mul_start,
-- Controls for memory
   ctl_ma_pc,    -- Transmit pc on memory address bus (if 0, transmit addr)
   ctl_sto       -- Memory store (if 0, fetch)
   :: a          -- all controls are bit signals
  }
```

Add control state

data CtlState a = CtlState

{

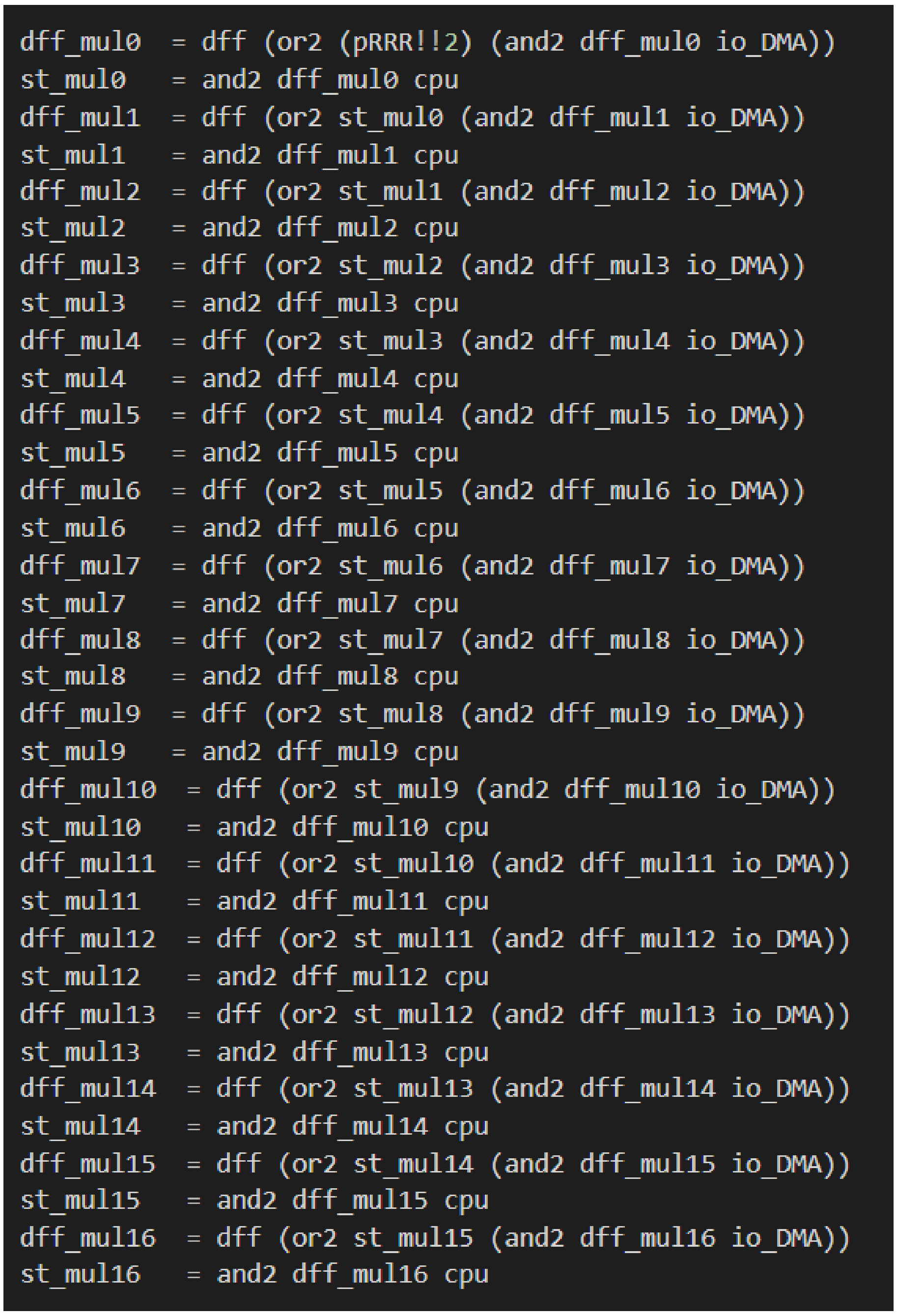
…

dff_mul0, st_mul0, dff_mul1, st_mul1, dff_mul2, st_mul2, dff_mul3, st_mul3, dff_mul4, st_mul4, dff_mul5, st_mul5, dff_mul6, st_mul6, dff_mul7, st_mul7, dff_mul8, st_mul8, dff_mul9, st_mul9, dff_mul10, st_mul10, dff_mul11, st_mul11, dff_mul12, st_mul12, dff_mul13, st_mul13, dff_mul14, st_mul14, dff_mul15, st_mul15, dff_mul16, st_mul16,

…

}

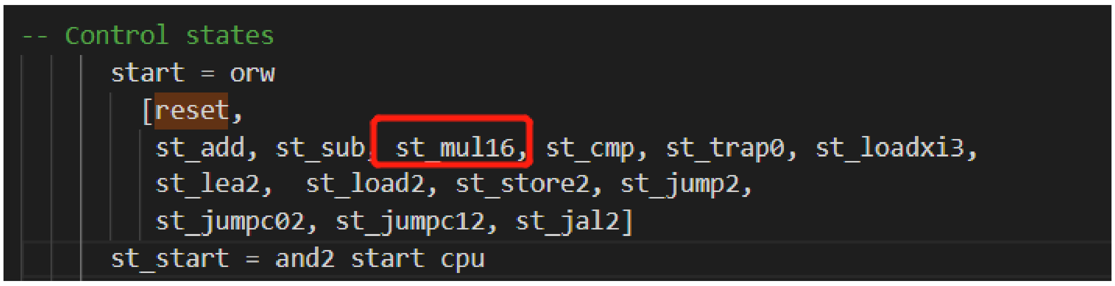
Step 4 Define the new state flip flops in control.hs



```
dff_mul0  = dff (or2 (pRRR!!2) (and2 dff_mul0 io_DMA))
st_mul0   = and2 dff_mul0 cpu
dff_mul1  = dff (or2 st_mul0 (and2 dff_mul1 io_DMA))
st_mul1   = and2 dff_mul1 cpu
dff_mul2  = dff (or2 st_mul1 (and2 dff_mul2 io_DMA))
st_mul2   = and2 dff_mul2 cpu
dff_mul3  = dff (or2 st_mul2 (and2 dff_mul3 io_DMA))
st_mul3   = and2 dff_mul3 cpu
dff_mul4  = dff (or2 st_mul3 (and2 dff_mul4 io_DMA))
st_mul4   = and2 dff_mul4 cpu
dff_mul5  = dff (or2 st_mul4 (and2 dff_mul5 io_DMA))
st_mul5   = and2 dff_mul5 cpu
dff_mul6  = dff (or2 st_mul5 (and2 dff_mul6 io_DMA))
st_mul6   = and2 dff_mul6 cpu
dff_mul7  = dff (or2 st_mul6 (and2 dff_mul7 io_DMA))
st_mul7   = and2 dff_mul7 cpu
dff_mul8  = dff (or2 st_mul7 (and2 dff_mul8 io_DMA))
st_mul8   = and2 dff_mul8 cpu
dff_mul9  = dff (or2 st_mul8 (and2 dff_mul9 io_DMA))
st_mul9   = and2 dff_mul9 cpu
dff_mul10  = dff (or2 st_mul9 (and2 dff_mul10 io_DMA))
st_mul10   = and2 dff_mul10 cpu
dff_mul11  = dff (or2 st_mul10 (and2 dff_mul11 io_DMA))
st_mul11   = and2 dff_mul11 cpu
dff_mul12  = dff (or2 st_mul11 (and2 dff_mul12 io_DMA))
st_mul12   = and2 dff_mul12 cpu
dff_mul13  = dff (or2 st_mul12 (and2 dff_mul13 io_DMA))
st_mul13   = and2 dff_mul13 cpu
dff_mul14  = dff (or2 st_mul13 (and2 dff_mul14 io_DMA))
st_mul14   = and2 dff_mul14 cpu
dff_mul15  = dff (or2 st_mul14 (and2 dff_mul15 io_DMA))
st_mul15   = and2 dff_mul15 cpu
dff_mul16  = dff (or2 st_mul15 (and2 dff_mul16 io_DMA))
st_mul16   = and2 dff_mul16 cpu
```

Step 5 Terminate the instruction

As what we did in step6 in loadxi. But we change st_mul0 which was define by original M1 to st_mul16 which was defined by ourselves.



```
-- Control states
      start = orw
        [reset,
         st_add, st_sub, st_mul16, st_cmp, st_trap0, st_loadxi3,
         st_lea2,  st_load2, st_store2, st_jump2,
         st_jumpc02, st_jumpc12, st_jal2]
      st_start = and2 start cpu
```

Step 6 modify control signals

New control states st_mul16 which was generated by our selves needs to be added to the corresponding control signal:

```
ctl_rf_ld    = orw [st_load2,st_lea1,st_add,st_sub,
                        st_jal2,st_loadxi2,st_loadxi3, st_mul16]
```

```
ctl_rf_pc    = orw [st_jal2, st_mul16]
```

```
ctl_rf_mul   = orw [st_mul16]
```

step 7 generate test data and obj code

write the test program:

Multi.asm.txt:

```
  load   R1,a[R0]

  load   R2,b[R0]

  mul    R3,R1,R2   -- expect R3 = 12


  load   R1,e[R0]

  load   R2,f[R0]   -- product is 363,096

  mul    R3,R1,R2


  load   R1,a[R0]

  load   R2,c[R0]

  mul    R3,R1,R2   -- expect R3 = 15
```

```
    trap R0,R0,R0


a    data     3
b    data     4
c    data     5
d    data     1
e    data   492
f    data   738
g    data   598
```

And then generated obj by sigma16

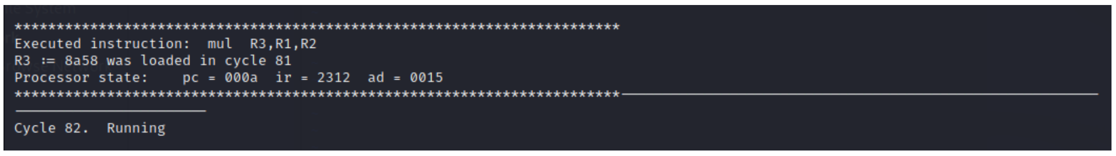
```
module   anonymous
data     f101,0010,f201,0011,2312,f101,0014,f201
data     0015,2312,f101,0010,f201,0012,2312,b000
data     0003,0004,0005,0001,01ec,02e2,0256
relocate 0001,0003,0006,0008,000b,000d
```

Step 8 update the Run.hs

Update the simulation print. Printing the related signal in console. Format them, so that become easy to understand.

Step 9 Run the simulation

In our testing program, we test 492 * 739. The result would be 58a58(hex), but we only show its lower 16bits, so the result would be 8a58. This is the boundary case.



```
************************************************************************
Executed instruction:  mul  R3,R1,R2
R3 := 8a58 was loaded in cycle 81
Processor state:    pc = 000a  ir = 2312  ad = 0015
************************************************************************
────────────────────────
Cycle 82.  Running
```

Before cycle 82.

We also run two simple cases 3*4 and 3*5. It is working well from the simulation output.