System Synthesis and Modeling

CSCE 3953

Fall 2018

Mon, Wed, Fri (9:40 AM – 10:30 AM)

Final Project

Simple Microcontroller Design

10-06-2018

Suhail Basalama

010786839

sebasala@email.uark.edu

Project Objective

The goal of this project is to integrate all the skills learned in this class to build a whole system in Verilog. A microcontroller is a perfect system that involve both sequential and combinational logic like registers, decoders, ALU. This project aims at giving us experience with Finite State Machines to control all the internal signals of the microcontroller. Merging all these parts together create a complex structure that needs careful observation and debugging skills. Finally, another objective is to simulate and synthesize our circuits using CAD tools like ModelSim and Synopsys to produce a performing microcontroller.

Design Methodology

Nomenclature and Standards

Predicting the complexity of this project, I decided to set a constant nomenclature standard. I considered the Memory Module and the Bus to be masters, and I set all other components as slaves. The word “read” means that the bus or the memory is reading from a slave module; similarly, the word “write” means that the bus is writing into that slave module. If the memory module is the master, I added “mem” as a prefix to the word “read” or “write” like “mem_read” meaning that the memory is reading one of the slave modules. I also set the ports in a constant order for all my modules, where I start with “clk”, “reset” and then all other ports.

Ports

I made my microcontroller synchronous with rising edge for all my modules except the decoder which does not have a clk. For all my modules I added an active high asynchronous reset to reset all the registers to 0. Top module has five ports as follow: clk, reset, input_pin, output_pin, input_enable. Other modules have different ports based on their functionality.

Top

My top module starts with including all the modules from the whole hierarchy using the command (`include). Next is declaring my ports. After that, I added my bus as a 16-bit wires and all other wires to connect the different modules. Then, I added all the seven FSMs to produce the control signals. Next, I ORed all my control signals to put them into the microcontroller components like registers. After that I added other components with the proper signals assigned to them.

Finite State Machines

I decided to create seven different Moore FSMs to produce all the control signals which are:

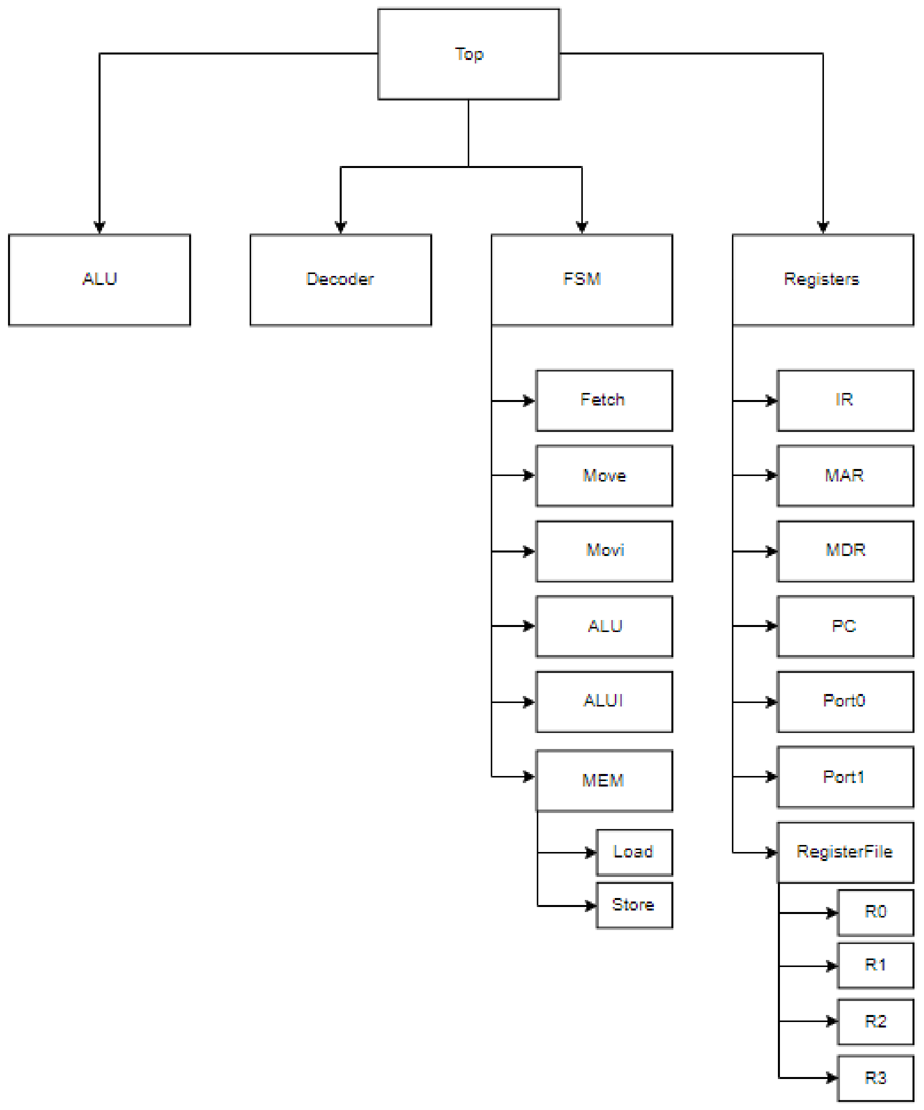
1. Fetch:

   State1: PC(read) and MAR(write)
   State2: MAR(mem_read) and MEM(RW,EN)
   State3: wait for memory MEM(MFC)
   State4: MDR(mem_write)
   State5: MDR(read) and IR(write)
   Done: PC(increment) and done = 1

2. Move
   State1: Ri(write) and Rj(read)
   Done: done = 1
3. Movi
   State1: FSM(read) and Ri(write)
   Done: done = 1
4. ALU
   State1: Ri(read) and ALU(writeIN1)
   State2: Rj(read) and ALU(writeIN2)
   State3: ALU(alu_out_en) and ALU(opControl)
   State4: ALU(read) and Ri(write)
   Done: done = 1
5. ALUI
   State1: Ri(read) and ALU(writeIN1)
   State2: FSM(read) and ALU(writeIN2)
   State3: ALU(alu_out_en) and ALU(opControl)
   State4: ALU(read) and Ri(write)
   Done: done = 1
6. Store
   State1: Ri(read) and MDR(write)
   State2: Rj(read) and MAR(write)
   State3: MDR(mem_read) and MEM(!RW) and MEM(EN)
   Done: done = 1
7. Load
   State1: Ri(read) and MAR(write)
   State2: MAR(mem_read) and MEM(RW) and MEM(EN)
   State3: wait for MFC
   State4: MDR(mem_write)
   State5: MDR(read) and Rj(write)
   Done: done = 1

Project File Structure

I created my modules in a hierarchical structure as shown in the diagram below:



Instruction Set with Machine Code

Add Ri, Rj	0000	Addi Ri, #	1000	Load (Ri), Rj	1010
Sub Ri, Rj	0001	Subi Ri, #	1001	Store Ri, (Rj)	1011
Not Ri	0010	Movi Ri, #	1111		
And Ri, Rj	0011				
Or Ri, Rj	0100				
Xor Ri, Rj	0101				
Xnor Ri, Rj	0110				
Mov Ri, Rj	0111				

```
Ri = i in 6-bit binary
Rj = j in 6-bit binary
P0 = 000100
P1 = 000101
```

Assumptions

1. Instructions can take as many Clock cycles as necessary.
2. Asynchronous active high reset.
3. All modules are positive edge clock triggered.
4. One memory cell only, so a max of one store can be used.
5. P1 is already integrated in the FSMs, but it can be enabled from the top level as well using the input_enable signal.
6. ALU opControl signals are:
```
parameter ADD  = 0;
parameter SUB  = 1;
parameter NOT  = 2;
parameter AND  = 3;
parameter OR   = 4;
parameter XOR  = 5;
parameter XNOR = 6;
```

Simulation Method

I used the instructions provided in the document which translate to machine code as follows:

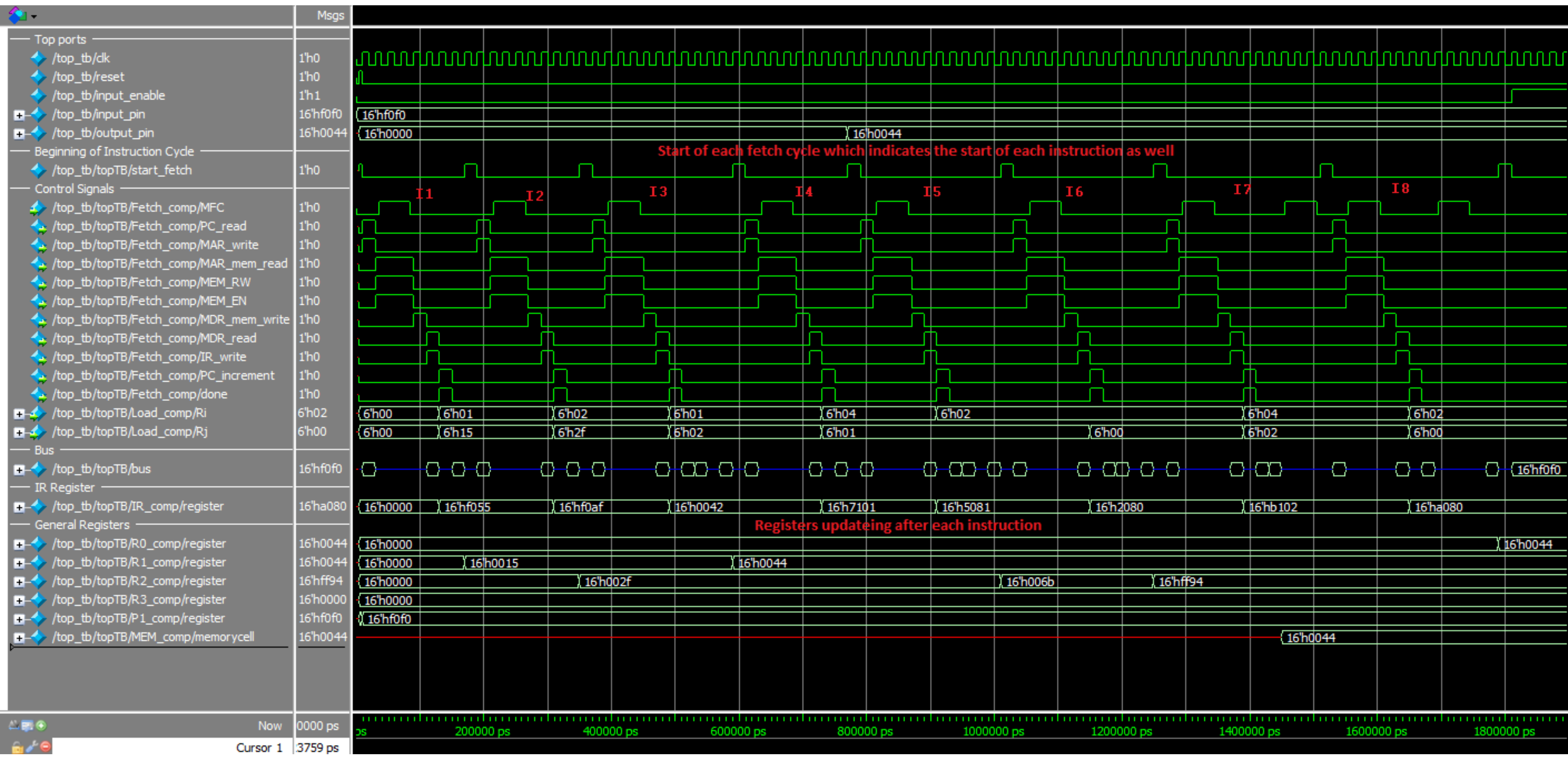
```
010786839
MOVI  R1, #15      1111 000001 010101
MOVI  R2, #2F      1111 000010 101111
ADD   R1, R2       0000 000001 000010
MOV   P0, R1       0111 000100 000001
XOR   R2, R1       0101 000010 000001
INV   R2           0010 000010 000000
STORE P0, (R2)     1011 000100 000010
LOAD (R2), R0      1010 000010 000000
```

The microcontroller can perform all the instructions, and Port 1 is integrated as well.

Results

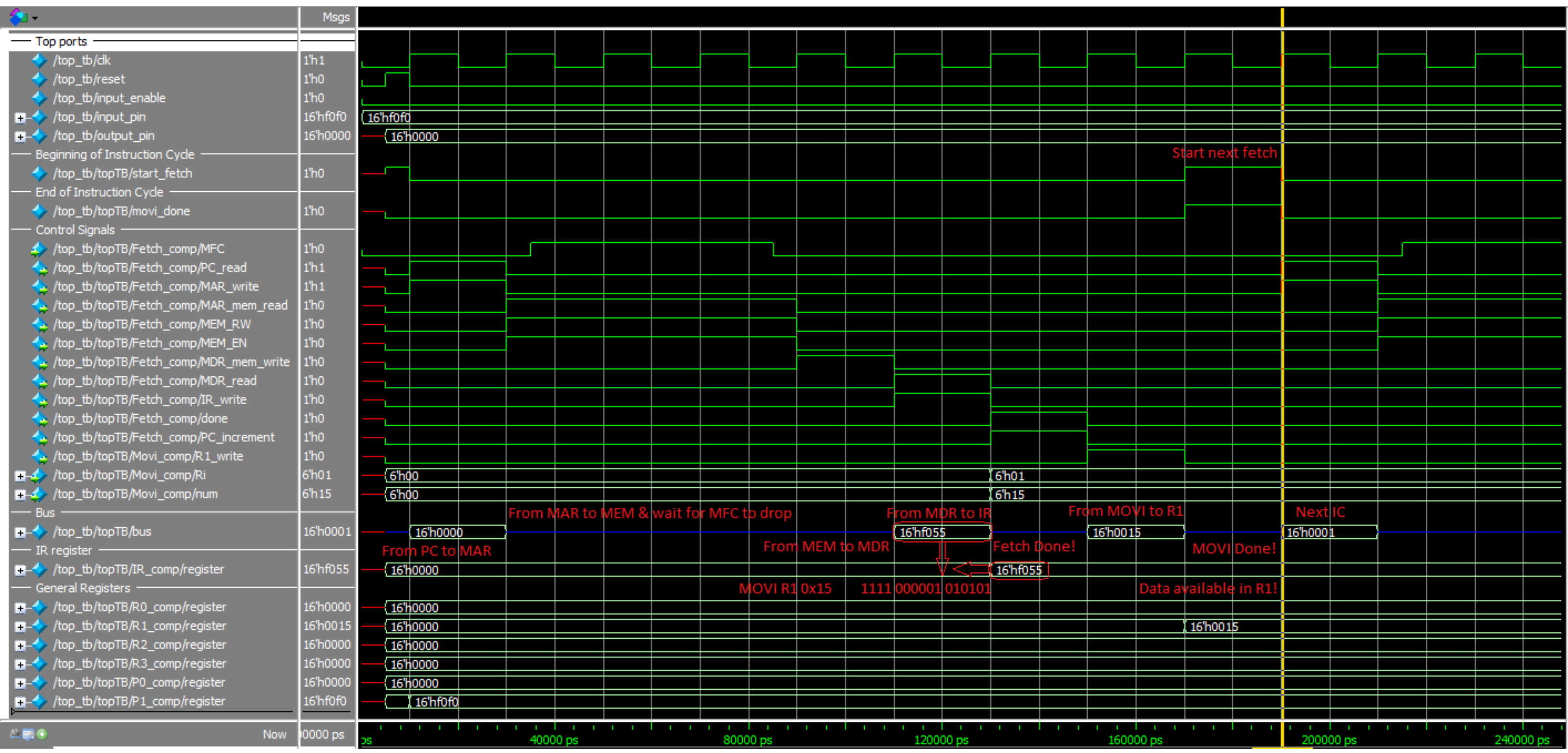
Fetch instruction starts by moving the data from the Program Counter (PC) register to the Memory Address (MAR) register, and the bus will contain the PC value at this stage; second, it moves the data from MAR to the memory and wait for MFC to drop so the bus will be in high impedance in this stage; third, the memory writes the data into the Memory Data Register (MDR) register, and the bus is still in high impedance because this transition is between MDR and MEM; fourth, the data (instruction in this case) is released from the MDR to the IR through the bus, so the bus will have the Instruction at this stage. That was the fetch cycle and it applies for all the instructions.

Here is a general overview of the all the instructions:


Msgs
Top ports
/top_tb/clk 1'h0
/top_tb/reset 1'h0
/top_tb/input_enable 1'h1
/top_tb/input_pin 16'hf0f0
/top_tb/output_pin 16'h0044
Beginning of Instruction Cycle
/top_tb/topTB/start_fetch 1'h0
Control Signals
/top_tb/topTB/Fetch_comp/MFC 1'h0
/top_tb/topTB/Fetch_comp/PC_read 1'h0
/top_tb/topTB/Fetch_comp/MAR_write 1'h0
/top_tb/topTB/Fetch_comp/MAR_mem_read 1'h0
/top_tb/topTB/Fetch_comp/MEM_RW 1'h0
/top_tb/topTB/Fetch_comp/MEM_EN 1'h0
/top_tb/topTB/Fetch_comp/MDR_mem_write 1'h0
/top_tb/topTB/Fetch_comp/MDR_read 1'h0
/top_tb/topTB/Fetch_comp/IR_write 1'h0
/top_tb/topTB/Fetch_comp/PC_increment 1'h0
/top_tb/topTB/Fetch_comp/done 1'h0
/top_tb/topTB/Load_comp/Ri 6'h02
/top_tb/topTB/Load_comp/Rj 6'h00
Bus
/top_tb/topTB/bus 16'hf0f0
IR Register
/top_tb/topTB/IR_comp/register 16'ha080
General Registers
/top_tb/topTB/R0_comp/register 16'h0044
/top_tb/topTB/R1_comp/register 16'h0044
/top_tb/topTB/R2_comp/register 16'hff94
/top_tb/topTB/R3_comp/register 16'h0000
/top_tb/topTB/P1_comp/register 16'hf0f0
/top_tb/topTB/MEM_comp/memorycell 16'h0044
Start of each fetch cycle which indicates the start of each instruction as well
I1
I2
I3
I4
I5
I6
I7
I8
16'hf0f0
16'h0000
16'h0044
6'h00 6'h01 6'h02 6'h01 6'h04 6'h02 6'h04 6'h02
6'h00 6'h15 6'h2f 6'h02 6'h01 6'h00 6'h02 6'h00
16'hf0f0
16'h0000 16'hf055 16'hf0af 16'h0042 16'h7101 16'h5081 16'h2080 16'hb102 16'ha080
Registers updateing after each instruction
16'h0000 16'h0044
16'h0000 16'h0015 16'h0044
16'h0000 16'h002f 16'h006b 16'hff94
16'h0000
16'hf0f0
16'h0044
Now 0000 ps
Cursor 1 3759 ps
ps 200000 ps 400000 ps 600000 ps 800000 ps 1000000 ps 1200000 ps 1400000 ps 1600000 ps 1800000 ps

Instruction 1 (`MOVI   R1,  #15        1111 000001 010101`)

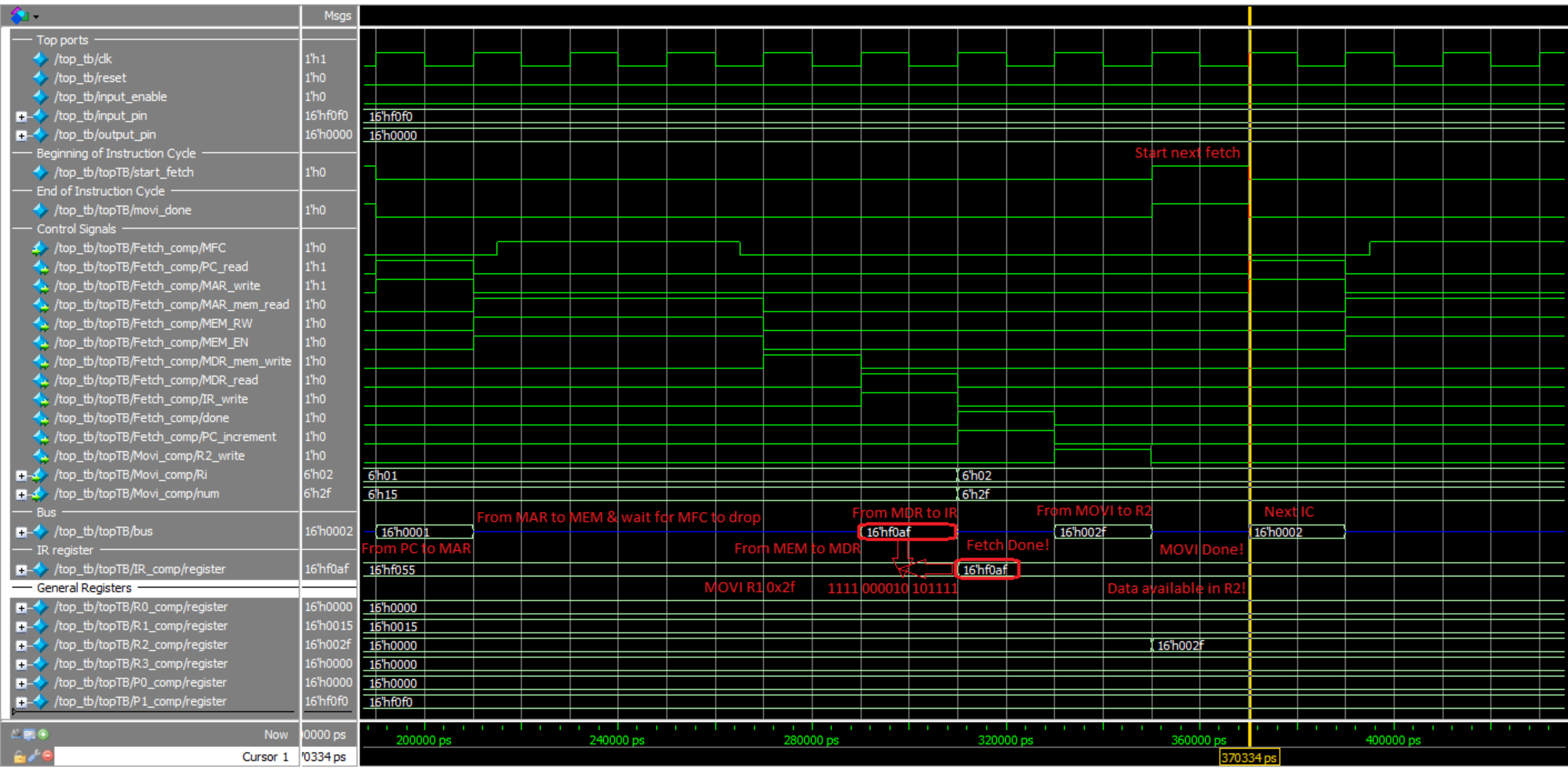
Fetching the instruction is the same as explained above. When fetch done, MOVI FSM is started. Next the signal R1_write is high and the FSM internal tri-state buffer lets the immediate value to be outputted to the bus through which then goes through the bus to R1.



```
                                          Msgs
Top ports
  /top_tb/clk                             1'h1
  /top_tb/reset                           1'h0
  /top_tb/input_enable                    1'h0
  /top_tb/input_pin                       16'hf0f0
  /top_tb/output_pin                      16'h0000
Beginning of Instruction Cycle
  /top_tb/topTB/start_fetch               1'h0
End of Instruction Cycle
  /top_tb/topTB/movi_done                 1'h0
Control Signals
  /top_tb/topTB/Fetch_comp/MFC            1'h0
  /top_tb/topTB/Fetch_comp/PC_read        1'h1
  /top_tb/topTB/Fetch_comp/MAR_write      1'h1
  /top_tb/topTB/Fetch_comp/MAR_mem_read   1'h0
  /top_tb/topTB/Fetch_comp/MEM_RW         1'h0
  /top_tb/topTB/Fetch_comp/MEM_EN         1'h0
  /top_tb/topTB/Fetch_comp/MDR_mem_write  1'h0
  /top_tb/topTB/Fetch_comp/MDR_read       1'h0
  /top_tb/topTB/Fetch_comp/IR_write       1'h0
  /top_tb/topTB/Fetch_comp/done           1'h0
  /top_tb/topTB/Fetch_comp/PC_increment   1'h0
  /top_tb/topTB/Movi_comp/R1_write        1'h0
  /top_tb/topTB/Movi_comp/Ri              6'h01
  /top_tb/topTB/Movi_comp/num             6'h15
Bus
  /top_tb/topTB/bus                       16'h0001
IR register
  /top_tb/topTB/IR_comp/register          16'hf055
General Registers
  /top_tb/topTB/R0_comp/register          16'h0000
  /top_tb/topTB/R1_comp/register          16'h0015
  /top_tb/topTB/R2_comp/register          16'h0000
  /top_tb/topTB/R3_comp/register          16'h0000
  /top_tb/topTB/P0_comp/register          16'h0000
  /top_tb/topTB/P1_comp/register          16'hf0f0
Now                                       0000 ps
```

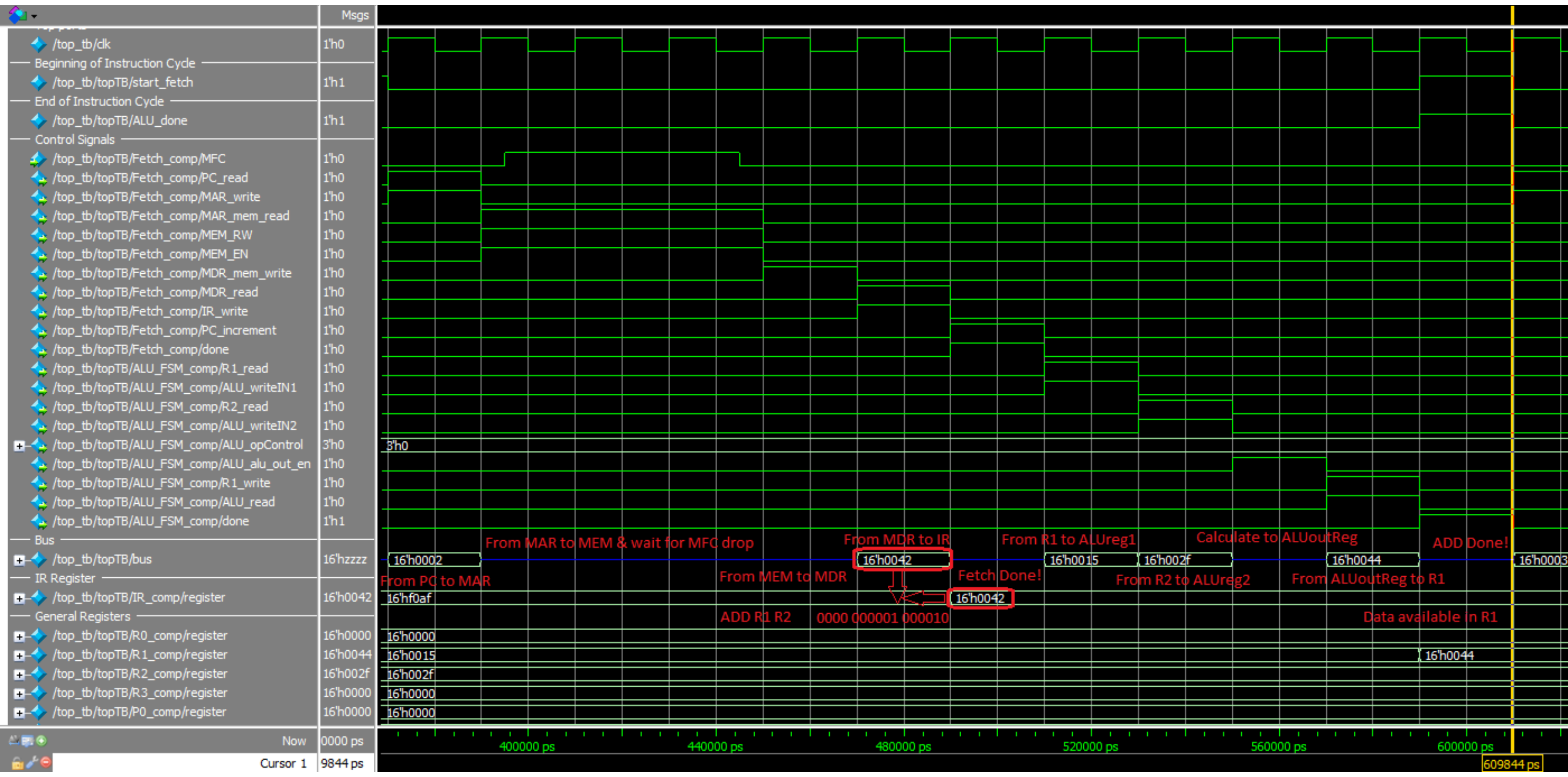
Instruction 2 (MOVI R2, #2F   1111 000010 101111)

Fetching the instruction is the same as explained above. When fetch done, MOVI FSM is started. Next the signal R2_write is high and the FSM internal tri-state buffer lets the immediate value to be outputted to the bus through which then goes through the bus to R2.


Msgs
Top ports
/top_tb/clk 1'h1
/top_tb/reset 1'h0
/top_tb/input_enable 1'h0
/top_tb/input_pin 16'hf0f0
/top_tb/output_pin 16'h0000
Beginning of Instruction Cycle
/top_tb/topTB/start_fetch 1'h0
End of Instruction Cycle
/top_tb/topTB/movi_done 1'h0
Control Signals
/top_tb/topTB/Fetch_comp/MFC 1'h0
/top_tb/topTB/Fetch_comp/PC_read 1'h1
/top_tb/topTB/Fetch_comp/MAR_write 1'h1
/top_tb/topTB/Fetch_comp/MAR_mem_read 1'h0
/top_tb/topTB/Fetch_comp/MEM_RW 1'h0
/top_tb/topTB/Fetch_comp/MEM_EN 1'h0
/top_tb/topTB/Fetch_comp/MDR_mem_write 1'h0
/top_tb/topTB/Fetch_comp/MDR_read 1'h0
/top_tb/topTB/Fetch_comp/IR_write 1'h0
/top_tb/topTB/Fetch_comp/done 1'h0
/top_tb/topTB/Fetch_comp/PC_increment 1'h0
/top_tb/topTB/Movi_comp/R2_write 1'h0
/top_tb/topTB/Movi_comp/Ri 6'h02
/top_tb/topTB/Movi_comp/num 6'h2f
Bus
/top_tb/topTB/bus 16'h0002
IR register
/top_tb/topTB/IR_comp/register 16'hf0af
General Registers
/top_tb/topTB/R0_comp/register 16'h0000
/top_tb/topTB/R1_comp/register 16'h0015
/top_tb/topTB/R2_comp/register 16'h002f
/top_tb/topTB/R3_comp/register 16'h0000
/top_tb/topTB/P0_comp/register 16'h0000
/top_tb/topTB/P1_comp/register 16'hf0f0
16'hf0f0
16'h0000
Start next fetch
6'h01
6'h02
6'h15
6'h2f
From MAR to MEM & wait for MFC to drop
From MDR to IR
From MOVI to R2
Next IC
16'h0001
16'hf0af
16'h002f
16'h0002
From PC to MAR
From MEM to MDR
Fetch Done!
MOVI Done!
16'hf055
16'hf0af
MOVI R1 0x2f
1111 000010 101111
Data available in R2!
16'h0000
16'h0015
16'h0000
16'h002f
16'h0000
16'h0000
16'hf0f0
Now 0000 ps
Cursor 1 '0334 ps
200000 ps
240000 ps
280000 ps
320000 ps
360000 ps
400000 ps
370334 ps

Instruction 3 (`ADD    R1, R2          0000 000001 000010`)

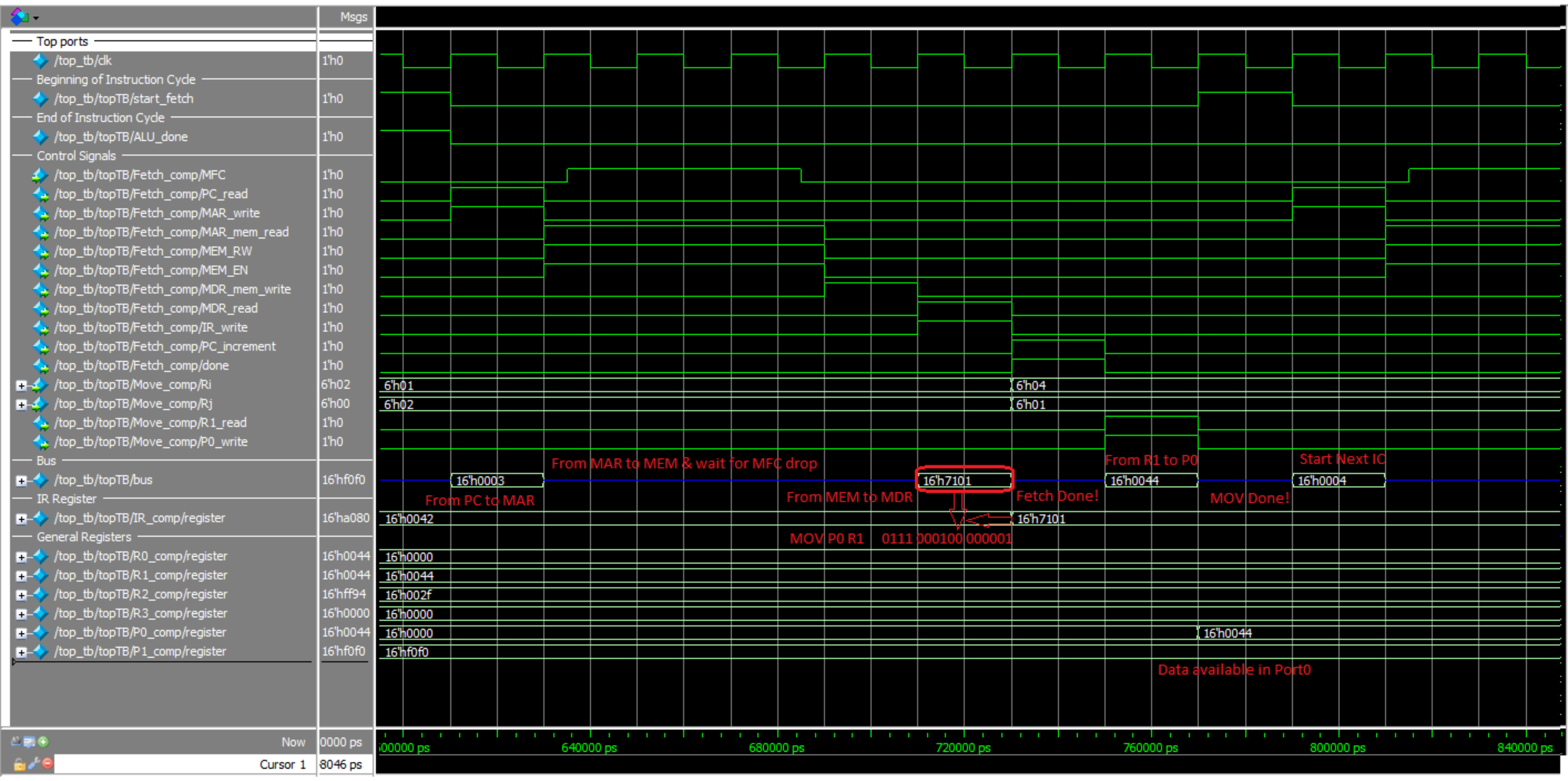
Fetch is the same as explained before. Then the ALU FSM starts and it asserts Ri (R1) and ALU_in1 at the same time allowing the data to go through the bus from R1 to ALUreg1. Next, the same happens with Rj (R2) and the data moves to ALU_in2. After that the ALU_opControl is generated by the FSM (000) and sent to the ALU which evaluates the result and the ALU_out reg is enabled to store the result. Next, the result is allowed to go to the bus by asserting ALU_read (meaning the bus will read the ALU) and asserting the destination Register (R1_write) at the same time.



```
                                                   Msgs
/top_tb/clk                                        1'h0
Beginning of Instruction Cycle
/top_tb/topTB/start_fetch                          1'h1
End of Instruction Cycle
/top_tb/topTB/ALU_done                             1'h1
Control Signals
/top_tb/topTB/Fetch_comp/MFC                       1'h0
/top_tb/topTB/Fetch_comp/PC_read                   1'h0
/top_tb/topTB/Fetch_comp/MAR_write                 1'h0
/top_tb/topTB/Fetch_comp/MAR_mem_read              1'h0
/top_tb/topTB/Fetch_comp/MEM_RW                    1'h0
/top_tb/topTB/Fetch_comp/MEM_EN                    1'h0
/top_tb/topTB/Fetch_comp/MDR_mem_write             1'h0
/top_tb/topTB/Fetch_comp/MDR_read                  1'h0
/top_tb/topTB/Fetch_comp/IR_write                  1'h0
/top_tb/topTB/Fetch_comp/PC_increment              1'h0
/top_tb/topTB/Fetch_comp/done                      1'h0
/top_tb/topTB/ALU_FSM_comp/R1_read                 1'h0
/top_tb/topTB/ALU_FSM_comp/ALU_writeIN1            1'h0
/top_tb/topTB/ALU_FSM_comp/R2_read                 1'h0
/top_tb/topTB/ALU_FSM_comp/ALU_writeIN2            1'h0
/top_tb/topTB/ALU_FSM_comp/ALU_opControl           3'h0
/top_tb/topTB/ALU_FSM_comp/ALU_alu_out_en          1'h0
/top_tb/topTB/ALU_FSM_comp/R1_write                1'h0
/top_tb/topTB/ALU_FSM_comp/ALU_read                1'h0
/top_tb/topTB/ALU_FSM_comp/done                    1'h1
Bus
/top_tb/topTB/bus                                  16'hzzzz
IR Register
/top_tb/topTB/IR_comp/register                     16'h0042
General Registers
/top_tb/topTB/R0_comp/register                     16'h0000
/top_tb/topTB/R1_comp/register                     16'h0044
/top_tb/topTB/R2_comp/register                     16'h002f
/top_tb/topTB/R3_comp/register                     16'h0000
/top_tb/topTB/P0_comp/register                     16'h0000
Now                                                0000 ps
Cursor 1                                           9844 ps
```

Instruction 4 (`MOV    P0, R1        0111 000100 000001`)

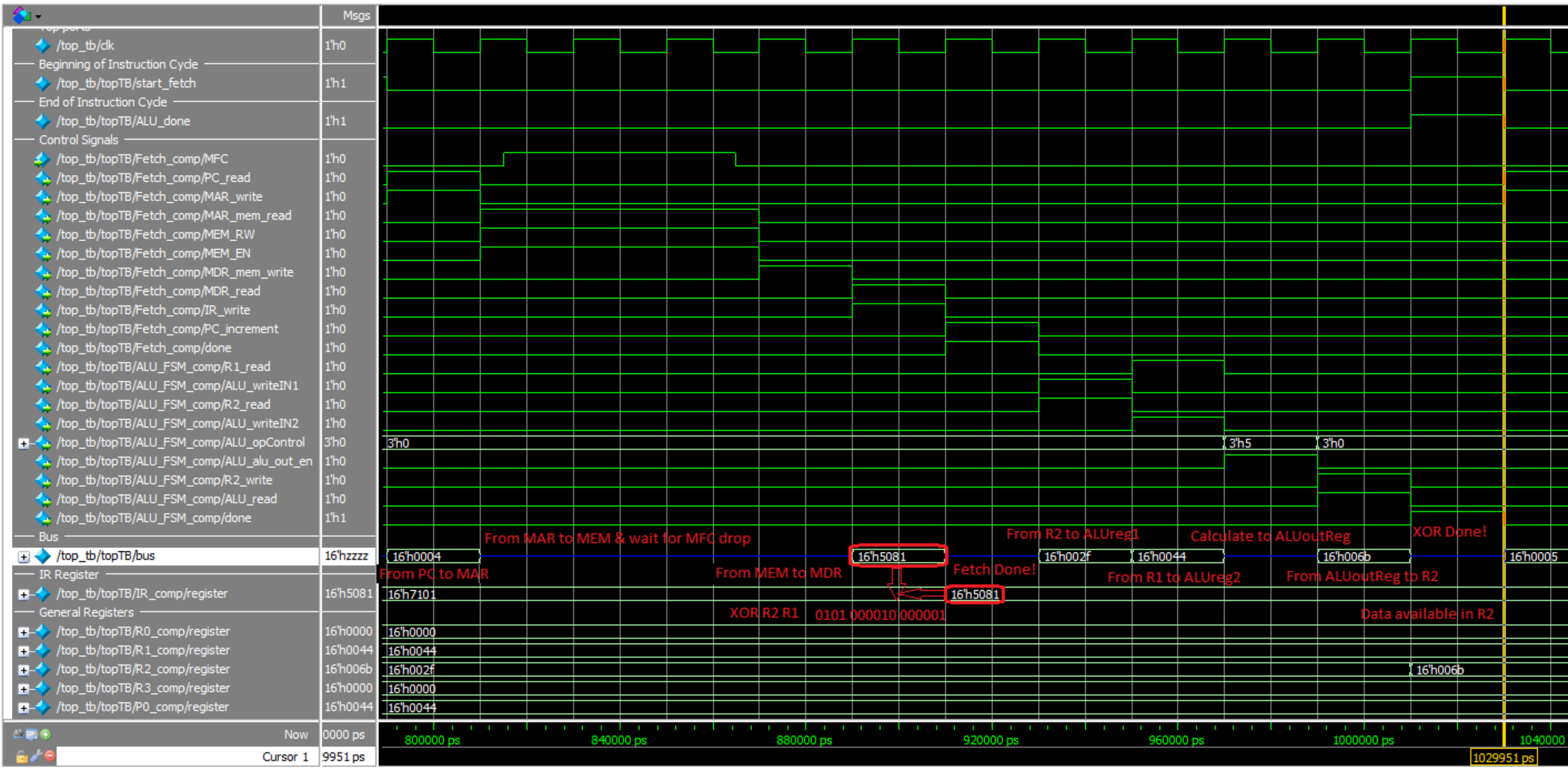
The fetch is the same here as well. When Move FSM starts both signals P0_write and R1_read are asserted at the same time, which means that the bus will read from R1 and write to P0.



Msgs
Top ports
/top_tb/clk 1'h0
Beginning of Instruction Cycle
/top_tb/topTB/start_fetch 1'h0
End of Instruction Cycle
/top_tb/topTB/ALU_done 1'h0
Control Signals
/top_tb/topTB/Fetch_comp/MFC 1'h0
/top_tb/topTB/Fetch_comp/PC_read 1'h0
/top_tb/topTB/Fetch_comp/MAR_write 1'h0
/top_tb/topTB/Fetch_comp/MAR_mem_read 1'h0
/top_tb/topTB/Fetch_comp/MEM_RW 1'h0
/top_tb/topTB/Fetch_comp/MEM_EN 1'h0
/top_tb/topTB/Fetch_comp/MDR_mem_write 1'h0
/top_tb/topTB/Fetch_comp/MDR_read 1'h0
/top_tb/topTB/Fetch_comp/IR_write 1'h0
/top_tb/topTB/Fetch_comp/PC_increment 1'h0
/top_tb/topTB/Fetch_comp/done 1'h0
/top_tb/topTB/Move_comp/Ri 6'h02
/top_tb/topTB/Move_comp/Rj 6'h00
/top_tb/topTB/Move_comp/R1_read 1'h0
/top_tb/topTB/Move_comp/P0_write 1'h0
Bus
/top_tb/topTB/bus 16'hf0f0
IR Register
/top_tb/topTB/IR_comp/register 16'ha080
General Registers
/top_tb/topTB/R0_comp/register 16'h0044
/top_tb/topTB/R1_comp/register 16'h0044
/top_tb/topTB/R2_comp/register 16'hff94
/top_tb/topTB/R3_comp/register 16'h0000
/top_tb/topTB/P0_comp/register 16'h0044
/top_tb/topTB/P1_comp/register 16'hf0f0
6'h01
6'h04
6'h02
6'h01
16'h0003
16'h7101
16'h0044
16'h0004
16'h0042
16'h7101
16'h0000
16'h0044
16'h002f
16'h0000
16'h0000
16'h0044
16'hf0f0
From MAR to MEM & wait for MFC drop
From PC to MAR
From MEM to MDR
Fetch Done!
From R1 to P0
MOV Done!
Start Next IC
MOV P0 R1
0111 000100 000001
Data available in Port0
Now 0000 ps
Cursor 1 8046 ps
00000 ps
640000 ps
680000 ps
720000 ps
760000 ps
800000 ps
840000 ps

Instruction 5 (XOR R2, R1 0101 000010 000001)

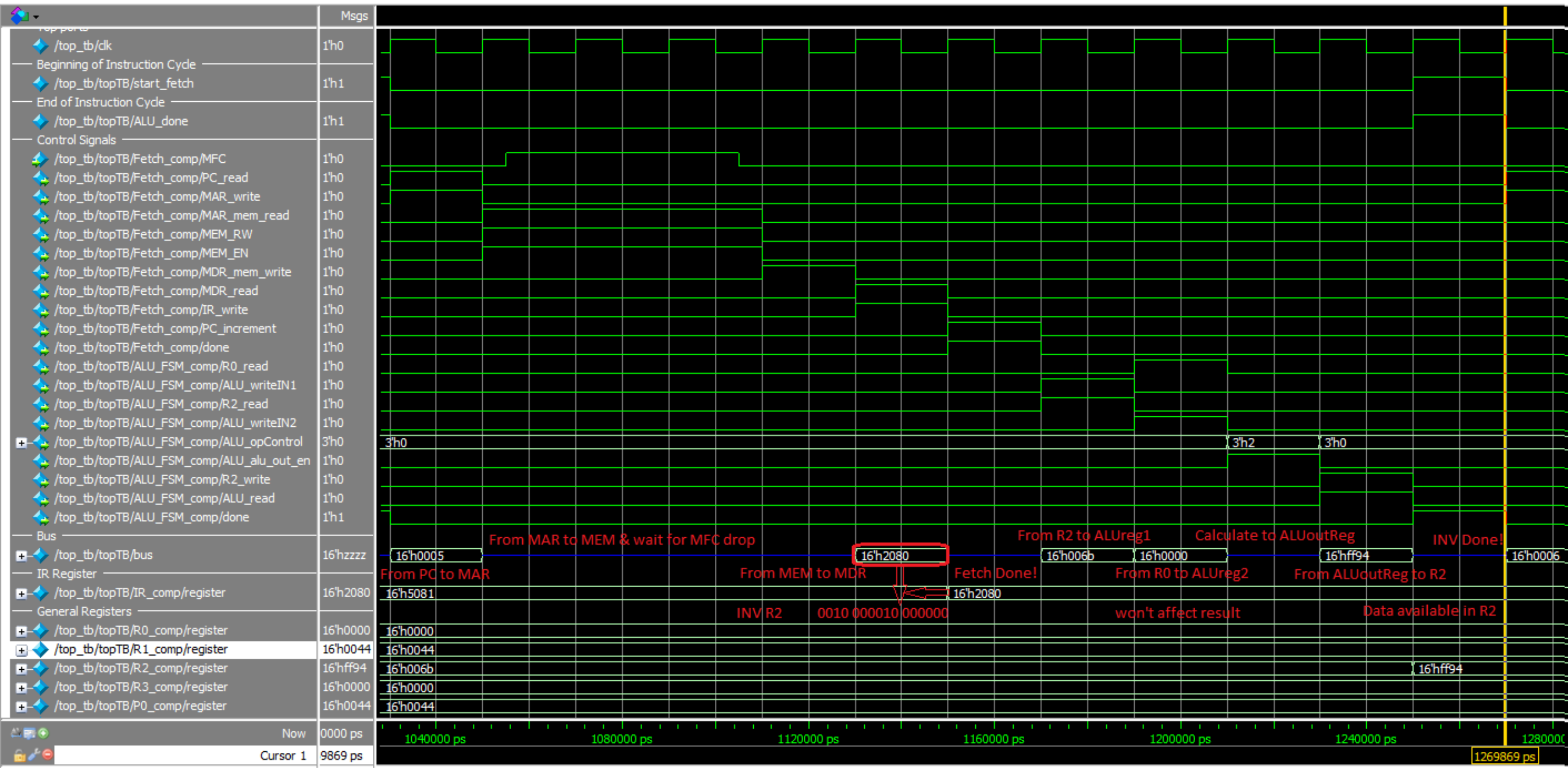
Fetch is the same as explained before. Then the ALU FSM starts and it asserts Ri (R2) and ALU_in1 at the same time allowing the data to go through the bus from R2 to ALUreg1. Next, the same happens with Rj (R1) and the data moves to ALU_in2. After that the ALU_opControl is generated by the FSM (101) and sent to the ALU which evaluates the result and the ALU_out reg is enabled to store the result. Next, the result is allowed to go to the bus by asserting ALU_read (meaning the bus will read the ALU) and asserting the destination Register (R2_write) at the same time.



```
                                                  Msgs
/top_tb/clk                                       1'h0
Beginning of Instruction Cycle
/top_tb/topTB/start_fetch                         1'h1
End of Instruction Cycle
/top_tb/topTB/ALU_done                            1'h1
Control Signals
/top_tb/topTB/Fetch_comp/MFC                      1'h0
/top_tb/topTB/Fetch_comp/PC_read                  1'h0
/top_tb/topTB/Fetch_comp/MAR_write                1'h0
/top_tb/topTB/Fetch_comp/MAR_mem_read             1'h0
/top_tb/topTB/Fetch_comp/MEM_RW                   1'h0
/top_tb/topTB/Fetch_comp/MEM_EN                   1'h0
/top_tb/topTB/Fetch_comp/MDR_mem_write            1'h0
/top_tb/topTB/Fetch_comp/MDR_read                 1'h0
/top_tb/topTB/Fetch_comp/IR_write                 1'h0
/top_tb/topTB/Fetch_comp/PC_increment             1'h0
/top_tb/topTB/Fetch_comp/done                     1'h0
/top_tb/topTB/ALU_FSM_comp/R1_read                1'h0
/top_tb/topTB/ALU_FSM_comp/ALU_writeIN1           1'h0
/top_tb/topTB/ALU_FSM_comp/R2_read                1'h0
/top_tb/topTB/ALU_FSM_comp/ALU_writeIN2           1'h0
/top_tb/topTB/ALU_FSM_comp/ALU_opControl          3'h0
/top_tb/topTB/ALU_FSM_comp/ALU_alu_out_en         1'h0
/top_tb/topTB/ALU_FSM_comp/R2_write               1'h0
/top_tb/topTB/ALU_FSM_comp/ALU_read               1'h0
/top_tb/topTB/ALU_FSM_comp/done                   1'h1
Bus
/top_tb/topTB/bus                                 16'hzzzz
IR Register
/top_tb/topTB/IR_comp/register                    16'h5081
General Registers
/top_tb/topTB/R0_comp/register                    16'h0000
/top_tb/topTB/R1_comp/register                    16'h0044
/top_tb/topTB/R2_comp/register                    16'h006b
/top_tb/topTB/R3_comp/register                    16'h0000
/top_tb/topTB/P0_comp/register                    16'h0044
Now                                               0000 ps
Cursor 1                                          9951 ps
```

Instruction 6 (INV   R2   0010 000010 000000)

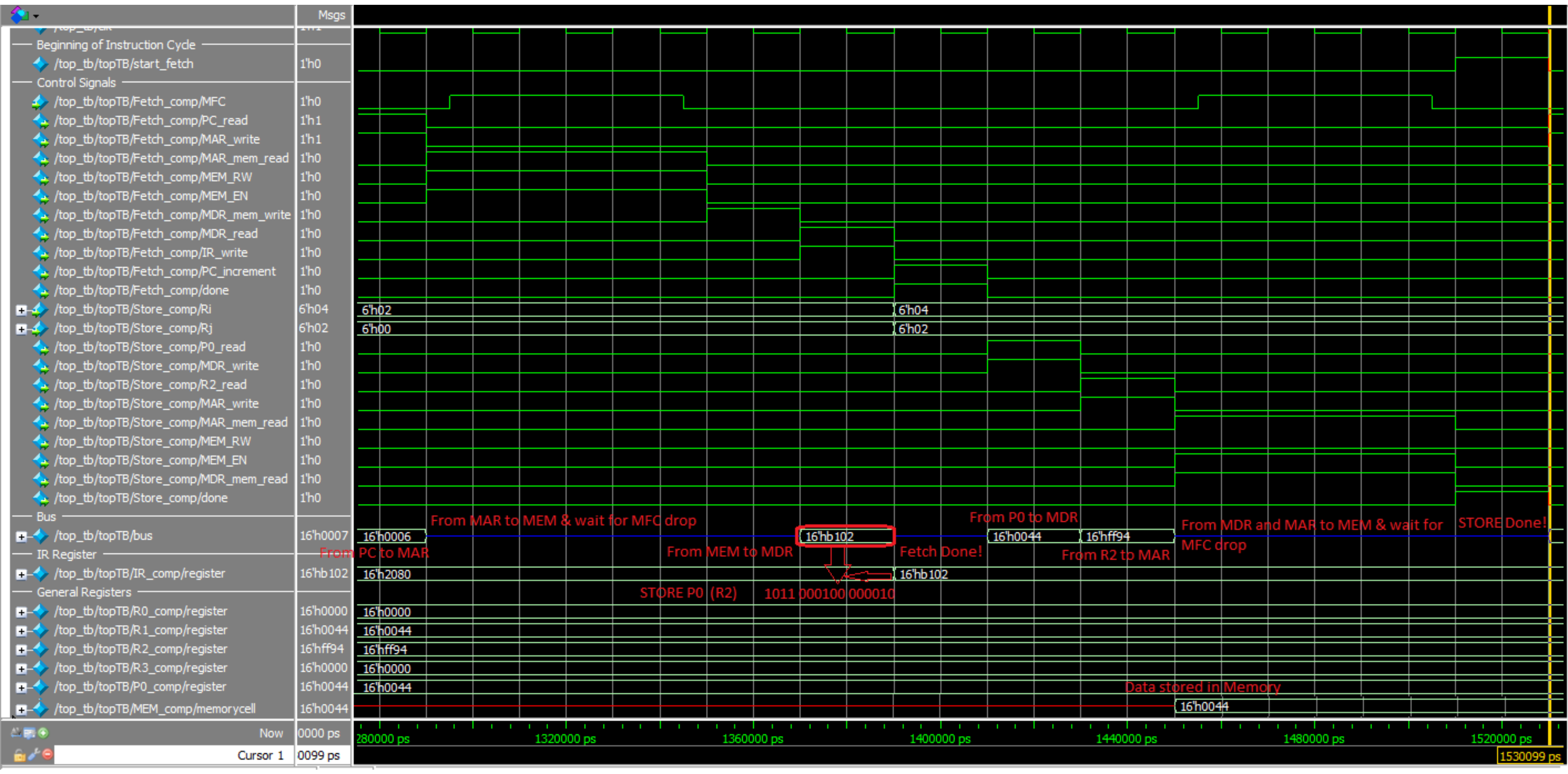
Fetch is the same as explained before. Then the ALU FSM starts and it asserts Ri (R2) and ALU_in1 at the same time allowing the data to go through the bus from R2 to ALUreg1. Next, the same happens with Rj (R0 or any other register will not affect the result) and the data moves to ALU_in2. After that the ALU_opControl is generated by the FSM (010) and sent to the ALU which evaluates the result and the ALU_out reg is enabled to store the result. Next, the result is allowed to go to the bus by asserting ALU_read (meaning the bus will read the ALU) and asserting the destination Register (R2_write) at the same time.


```
                                          Msgs
  /top_tb/clk                             1'h0
Beginning of Instruction Cycle
  /top_tb/topTB/start_fetch               1'h1
End of Instruction Cycle
  /top_tb/topTB/ALU_done                  1'h1
Control Signals
  /top_tb/topTB/Fetch_comp/MFC            1'h0
  /top_tb/topTB/Fetch_comp/PC_read        1'h0
  /top_tb/topTB/Fetch_comp/MAR_write      1'h0
  /top_tb/topTB/Fetch_comp/MAR_mem_read   1'h0
  /top_tb/topTB/Fetch_comp/MEM_RW         1'h0
  /top_tb/topTB/Fetch_comp/MEM_EN         1'h0
  /top_tb/topTB/Fetch_comp/MDR_mem_write  1'h0
  /top_tb/topTB/Fetch_comp/MDR_read       1'h0
  /top_tb/topTB/Fetch_comp/IR_write       1'h0
  /top_tb/topTB/Fetch_comp/PC_increment   1'h0
  /top_tb/topTB/Fetch_comp/done           1'h0
  /top_tb/topTB/ALU_FSM_comp/R0_read      1'h0
  /top_tb/topTB/ALU_FSM_comp/ALU_writeIN1 1'h0
  /top_tb/topTB/ALU_FSM_comp/R2_read      1'h0
  /top_tb/topTB/ALU_FSM_comp/ALU_writeIN2 1'h0
  /top_tb/topTB/ALU_FSM_comp/ALU_opControl 3'h0    3'h0  3'h2  3'h0
  /top_tb/topTB/ALU_FSM_comp/ALU_alu_out_en 1'h0
  /top_tb/topTB/ALU_FSM_comp/R2_write     1'h0
  /top_tb/topTB/ALU_FSM_comp/ALU_read     1'h0
  /top_tb/topTB/ALU_FSM_comp/done         1'h1
Bus
  /top_tb/topTB/bus                       16'hzzzz   16'h0005  16'h2080  16'h006b  16'h0000  16'hff94  16'h0006
IR Register
  /top_tb/topTB/IR_comp/register          16'h2080   16'h5081  16'h2080
General Registers
  /top_tb/topTB/R0_comp/register          16'h0000   16'h0000
  /top_tb/topTB/R1_comp/register          16'h0044   16'h0044
  /top_tb/topTB/R2_comp/register          16'hff94   16'h006b  16'hff94
  /top_tb/topTB/R3_comp/register          16'h0000   16'h0000
  /top_tb/topTB/P0_comp/register          16'h0044   16'h0044

Now      0000 ps    1040000 ps  1080000 ps  1120000 ps  1160000 ps  1200000 ps  1240000 ps  128000
Cursor 1 9869 ps                                                                    1269869 ps
```

Instruction 7 (`STORE P0, (R2)     1011 000100 000010`)

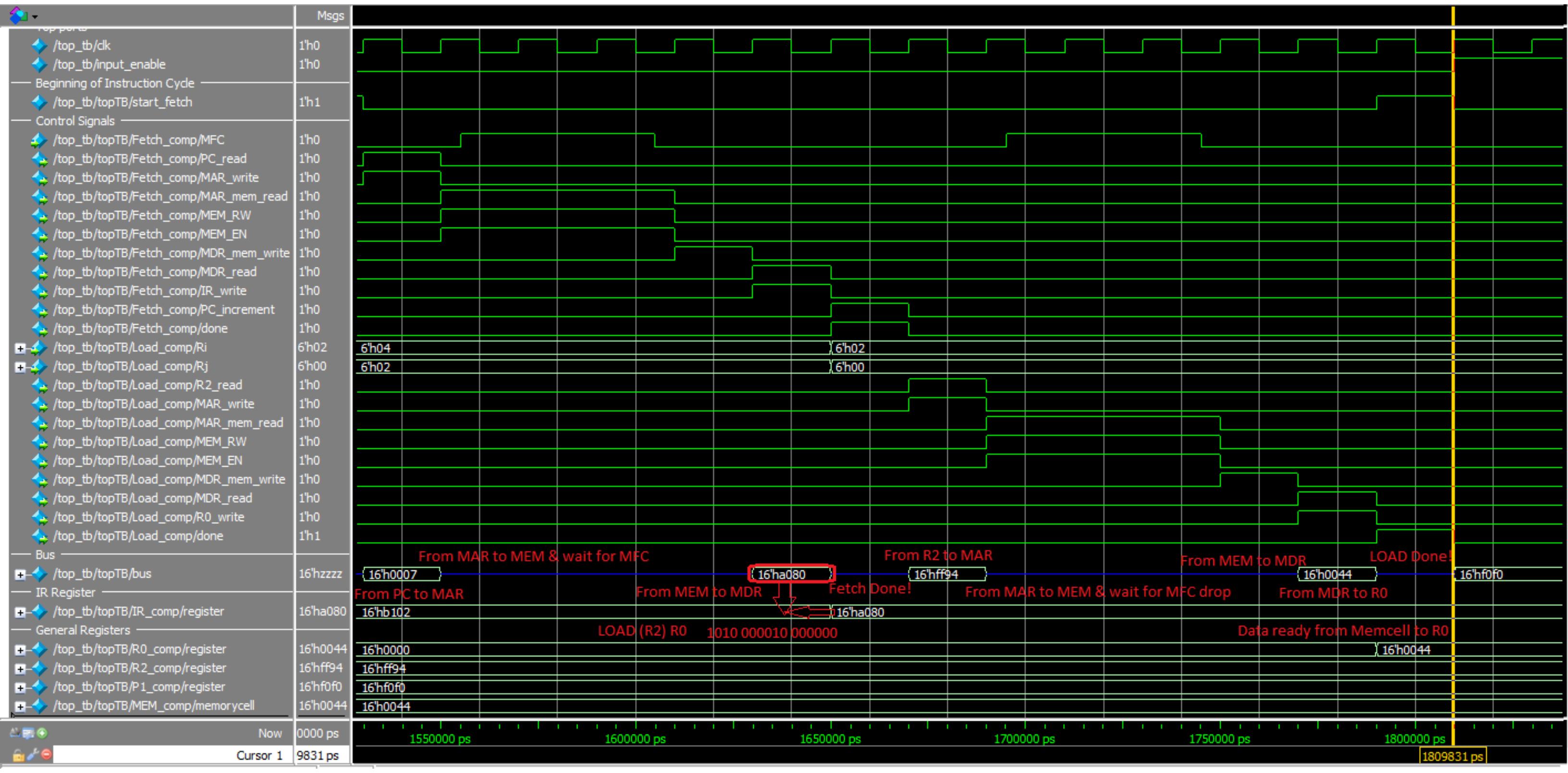
Fetch is the same. After that, the data moves from the Ri (P0) to the memory data register MDR by asserting P0_read and MDR_write at the same time. Next step is to copy the address from R2 to memory address register MAR by asserting R2_read and MAR_write simultaneously. Finally, the FSM enables the memory by setting EN high, and set RW to 0 to write to memory, and then wait for the data to be written into the memory which is indicated by MFC signal going to zero.



```
                                                     Msgs
Beginning of Instruction Cycle
  /top_tb/topTB/start_fetch                          1'h0
Control Signals
  /top_tb/topTB/Fetch_comp/MFC                       1'h0
  /top_tb/topTB/Fetch_comp/PC_read                   1'h1
  /top_tb/topTB/Fetch_comp/MAR_write                 1'h1
  /top_tb/topTB/Fetch_comp/MAR_mem_read              1'h0
  /top_tb/topTB/Fetch_comp/MEM_RW                    1'h0
  /top_tb/topTB/Fetch_comp/MEM_EN                    1'h0
  /top_tb/topTB/Fetch_comp/MDR_mem_write             1'h0
  /top_tb/topTB/Fetch_comp/MDR_read                  1'h0
  /top_tb/topTB/Fetch_comp/IR_write                  1'h0
  /top_tb/topTB/Fetch_comp/PC_increment              1'h0
  /top_tb/topTB/Fetch_comp/done                      1'h0
  /top_tb/topTB/Store_comp/Ri                        6'h04
  /top_tb/topTB/Store_comp/Rj                        6'h02
  /top_tb/topTB/Store_comp/P0_read                   1'h0
  /top_tb/topTB/Store_comp/MDR_write                 1'h0
  /top_tb/topTB/Store_comp/R2_read                   1'h0
  /top_tb/topTB/Store_comp/MAR_write                 1'h0
  /top_tb/topTB/Store_comp/MAR_mem_read              1'h0
  /top_tb/topTB/Store_comp/MEM_RW                    1'h0
  /top_tb/topTB/Store_comp/MEM_EN                    1'h0
  /top_tb/topTB/Store_comp/MDR_mem_read              1'h0
  /top_tb/topTB/Store_comp/done                      1'h0
Bus
  /top_tb/topTB/bus                                  16'h0007
IR Register
  /top_tb/topTB/IR_comp/register                     16'hb102
General Registers
  /top_tb/topTB/R0_comp/register                     16'h0000
  /top_tb/topTB/R1_comp/register                     16'h0044
  /top_tb/topTB/R2_comp/register                     16'hff94
  /top_tb/topTB/R3_comp/register                     16'h0000
  /top_tb/topTB/P0_comp/register                     16'h0044
  /top_tb/topTB/MEM_comp/memorycell                  16'h0044
Now                                                  0000 ps
Cursor 1                                             0099 ps
```

Instruction 8 (`LOAD (R2), R0        1010 000010 000000`)

For fetch, it is the same. And the load is almost the same with fetch except that load takes data from memory to another register other than IR. So, load FSM starts by sending the address from Ri (R0) to the memory address register MAR by asserting R2_read and MAR_write simultaneously. Then we send the data to memory by asserting EN and RW to read from memory and we wait for MFC to drop. The FSM then allow the data to be written to the MDR register by enabling MDR_mem_write. Finally, the FSM sends the data from MDR to Rj (R0) by asserting MDR_read and R0_write simultaneously.



	Msgs
/top_tb/clk	1'h0
/top_tb/input_enable	1'h0
Beginning of Instruction Cycle	
/top_tb/topTB/start_fetch	1'h1
Control Signals	
/top_tb/topTB/Fetch_comp/MFC	1'h0
/top_tb/topTB/Fetch_comp/PC_read	1'h0
/top_tb/topTB/Fetch_comp/MAR_write	1'h0
/top_tb/topTB/Fetch_comp/MAR_mem_read	1'h0
/top_tb/topTB/Fetch_comp/MEM_RW	1'h0
/top_tb/topTB/Fetch_comp/MEM_EN	1'h0
/top_tb/topTB/Fetch_comp/MDR_mem_write	1'h0
/top_tb/topTB/Fetch_comp/MDR_read	1'h0
/top_tb/topTB/Fetch_comp/IR_write	1'h0
/top_tb/topTB/Fetch_comp/PC_increment	1'h0
/top_tb/topTB/Fetch_comp/done	1'h0
/top_tb/topTB/Load_comp/Ri	6'h02
/top_tb/topTB/Load_comp/Rj	6'h00
/top_tb/topTB/Load_comp/R2_read	1'h0
/top_tb/topTB/Load_comp/MAR_write	1'h0
/top_tb/topTB/Load_comp/MAR_mem_read	1'h0
/top_tb/topTB/Load_comp/MEM_RW	1'h0
/top_tb/topTB/Load_comp/MEM_EN	1'h0
/top_tb/topTB/Load_comp/MDR_mem_write	1'h0
/top_tb/topTB/Load_comp/MDR_read	1'h0
/top_tb/topTB/Load_comp/R0_write	1'h0
/top_tb/topTB/Load_comp/done	1'h1
Bus	
/top_tb/topTB/bus	16'hzzzz
IR Register	
/top_tb/topTB/IR_comp/register	16'ha080
General Registers	
/top_tb/topTB/R0_comp/register	16'h0044
/top_tb/topTB/R2_comp/register	16'hff94
/top_tb/topTB/P1_comp/register	16'hf0f0
/top_tb/topTB/MEM_comp/memorycell	16'h0044
Now	0000 ps
Cursor 1	9831 ps

Analysis and Conclusion

This was an excellent project that combined all what we have learned in this class starting from implementing simple logic devices like registers and decoder, to more complex ones like ALU. It also had excellent examples to practice what we learned about Finite State Machines and how to use them in our designs in a practical way. In addition, one of the main aspects of this project is the complexity and how hardware systems can be very complicated even if it seems easy from the outside. Therefore, I learned a very important skill which is standardizing my naming and methodology so that I can handle large systems like this one. If I did not spend some time organizing the structure and design of my microcontroller, I would not be able to finish it in about four days. Finally, I did synthesize my registers and they work properly, but my FSMs have some issues after synthesis since I am raising my control signals for one clock cycle only, but I am planning to complete the synthesis process in the break and deploy my project to an FPGA to test it in reality.

Source Code

Top.v

```verilog
`include "/FSM/Fetch/Fetch.v"
`include "/FSM/Move/Move.v"
`include "/FSM/Movi/Movi.v"
`include "/FSM/ALU_FSM/ALU_FSM.v"
`include "/FSM/ALUI_FSM/ALUI_FSM.v"
`include "/FSM/Mem_FSM/Store.v"
`include "/FSM/Mem_FSM/Load.v"
`include "/Registers/PC/PC.v"
`include "/Registers/MAR/MAR.v"
`include "/Registers/MDR/MDR.v"
`include "MEM.v"
`include "/Registers/IR/IR.v"
`include "/Registers/RegisterFile/R0.v"
`include "/Registers/RegisterFile/R1.v"
`include "/Registers/RegisterFile/R2.v"
`include "/Registers/RegisterFile/R3.v"
`include "/Decoder/Decoder.v"
`include "/ALU/ALU_pre.v"
`include "/Registers/Port0/Port0.v"
`include "/Registers/Port1/Port1.v"

module top(clk, reset, input_pin, output_pin, input_enable);

input clk, reset, input_enable;
input[15:0] input_pin;
output[15:0] output_pin;

wire[15:0] bus;

wire[15:0] MAR_to_MEM;
wire[15:0] MDR_to_MEM;
wire[15:0] MEM_to_MDR;

wire MFC;
wire PC_read, PC_increment;
wire MAR_write, MAR_mem_read;
wire MEM_RW, MEM_EN;
wire MDR_mem_write, MDR_read;
wire IR_write;
wire[15:0] out_to_decoder;
wire move_done, movi_done, ALU_done, ALUI_done, load_done, store_done;
wire start_fetch, start_move, start_movi, start_ALU, start_ALUI, start_load,
start_store;

wire R0_write, R0_read;
wire R1_write, R1_read;
wire R2_write, R2_read;
wire R3_write, R3_read;
wire P0_write, P0_read;
wire decode_done, fetch_done;

wire ALU_read;
wire[2:0] opControl;

wire[5:0] Ri, Rj;
wire[3:0] opCode;

wire[2:0] ALU_opControl, ALUI_opControl;

assign Rj = out_to_decoder[5:0];
assign Ri = out_to_decoder[11:6];
assign opCode = out_to_decoder[15:12];


assign start_fetch = reset || move_done || movi_done || ALU_done || ALUI_done ||
load_done || store_done;

Decoder Decoder_comp(fetch_done, opCode, start_move, start_movi, start_ALU,
start_ALUI, start_load, start_store);

reg flag = 1;
always@(PC_read)
if(bus==7) flag = 0;

Fetch Fetch_comp
(
    clk, reset, start_fetch && flag, MFC,
    PC_read, PC_increment,
    MAR_write_fetch, MAR_mem_read_fetch,
    MEM_RW_fetch, MEM_EN_fetch,
    MDR_mem_write_fetch, MDR_read_fetch,
    IR_write, fetch_done
);

Move Move_comp
(
    clk, reset, start_move, Ri, Rj,
    move_done,
    R0_write_move, R0_read_move,
    R1_write_move, R1_read_move,
    R2_write_move, R2_read_move,
    R3_write_move, R3_read_move,
    P0_write_move, P0_read_move,
                   P1_read_move
);

Movi Movi_comp
(
    clk, reset, start_movi, Ri, Rj,
    bus,
    movi_done,
    R0_write_movi,
    R1_write_movi,
    R2_write_movi,
    R3_write_movi,
    P0_write_movi
);

ALU_FSM ALU_FSM_comp
(
    clk, reset, start_ALU, opCode, Ri, Rj,
    ALU_done,
    R0_write_ALU, R0_read_ALU,
    R1_write_ALU, R1_read_ALU,
    R2_write_ALU, R2_read_ALU,
    R3_write_ALU, R3_read_ALU,
    P0_write_ALU, P0_read_ALU,
                  P1_read_ALU,
    ALU_opControl,
    ALU_alu_out_en, ALU_writeIN1, ALU_writeIN2, ALU_read
);

ALUI_FSM ALUI_FSM_comp
(
    clk, reset, start_ALUI, opCode, Ri, Rj,
    bus,
    ALUI_done,
    R0_write_ALUI, R0_read_ALUI,
    R1_write_ALUI, R1_read_ALUI,
    R2_write_ALUI, R2_read_ALUI,
    R3_write_ALUI, R3_read_ALUI,
    P0_write_ALUI, P0_read_ALUI,
    ALUI_opControl,
    ALUI_alu_out_en, ALUI_writeIN1, ALUI_writeIN2, ALUI_read
);

Load Load_comp
(
    clk, reset, start_load, MFC, Ri, Rj,
    R0_read_load, R0_write_load,
    R1_read_load, R1_write_load,
    R2_read_load, R2_write_load,
    R3_read_load, R3_write_load,
    P0_read_load, P0_write_load,
    MAR_write_load, MAR_mem_read_load,
    MEM_RW_load, MEM_EN_load,
    MDR_mem_write_load, MDR_read_load,
    load_done
);

Store Store_comp
(
    clk, reset, start_store, MFC, Ri, Rj,
    R0_read_store,
    R1_read_store,
    R2_read_store,
    R3_read_store,
    P0_read_store,
    P1_read_Store,
    MAR_write_store, MAR_mem_read_store,
    MEM_RW_store, MEM_EN_store,
    MDR_mem_read_store, MDR_write_store,
    store_done
);

assign R0_read = R0_read_move || R0_read_ALU || R0_read_ALUI || R0_read_load ||
R0_read_store;
assign R1_read = R1_read_move || R1_read_ALU || R1_read_ALUI || R1_read_load ||
R1_read_store;
assign R2_read = R2_read_move || R2_read_ALU || R2_read_ALUI || R2_read_load ||
R2_read_store;
assign R3_read = R3_read_move || R3_read_ALU || R3_read_ALUI || R3_read_load ||
R3_read_store;
assign P0_read = P0_read_move || P0_read_ALU || P0_read_ALUI || P0_read_load ||
P0_read_store;
assign P1_read = P1_read_move || P1_read_ALU || P1_read_Store|| input_enable;
assign R0_write = R0_write_move || R0_write_movi || R0_write_ALU || R0_write_ALUI
|| R0_write_load;
assign R1_write = R1_write_move || R1_write_movi || R1_write_ALU || R1_write_ALUI
|| R1_write_load;
assign R2_write = R2_write_move || R2_write_movi || R2_write_ALU || R2_write_ALUI
|| R2_write_load;
assign R3_write = R3_write_move || R3_write_movi || R3_write_ALU || R3_write_ALUI
|| R3_write_load;
assign P0_write = P0_write_move || P0_write_movi || P0_write_ALU || P0_write_ALUI
|| P0_write_load;

assign read_ALU     = ALU_read       || ALUI_read;
assign writeIN1     = ALU_writeIN1   || ALUI_writeIN1;
assign writeIN2     = ALU_writeIN2   || ALUI_writeIN2;
assign alu_out_en   = ALU_alu_out_en || ALUI_alu_out_en;
assign opControl    = ALU_opControl   | ALUI_opControl;

assign MAR_write     = MAR_write_fetch     || MAR_write_load     ||
MAR_write_store;
assign MAR_mem_read  = MAR_mem_read_fetch  || MAR_mem_read_load  ||
MAR_mem_read_store;
assign MEM_RW        = MEM_RW_fetch        || MEM_RW_load        || MEM_RW_store;
assign MEM_EN        = MEM_EN_fetch        || MEM_EN_load        || MEM_EN_store;
assign MDR_mem_write = MDR_mem_write_fetch || MDR_mem_write_load;
assign MDR_read      = MDR_read_fetch      || MDR_read_load;
// assign MDR_mem_read  = MDR_mem_read_store;
// assign MDR_write     = MDR_write_store;

ALU ALU_comp (clk, reset, bus, bus, read_ALU, writeIN1, writeIN2, alu_out_en,
opControl);
R0 R0_comp   (clk, reset, bus, bus, R0_read, R0_write);
R1 R1_comp   (clk, reset, bus, bus, R1_read, R1_write);
R2 R2_comp   (clk, reset, bus, bus, R2_read, R2_write);
R3 R3_comp   (clk, reset, bus, bus, R3_read, R3_write);
Port0 P0_comp(clk, reset, bus, bus, P0_read, P0_write, output_pin);
Port1 P1_comp(clk, reset, bus,      P1_read,            input_pin);
PC PC_comp   (clk, reset, bus, PC_read, PC_increment);
MAR MAR_comp (clk, reset, bus, MAR_to_MEM, MAR_write, MAR_mem_read);
MEM MEM_comp (MEM_to_MDR, MFC, MEM_EN, MAR_to_MEM, MDR_to_MEM, MEM_RW);
MDR MDR_comp (clk, reset, bus, bus, MDR_read, MDR_write_store, MEM_to_MDR ,
MDR_to_MEM, MDR_mem_read_store, MDR_mem_write);
IR IR_comp   (clk, reset, bus, out_to_decoder, IR_write, 1);
endmodule
```

Top_tb.v

```
`timescale 1ns/1ps

module top_tb;

//inputs
reg clk, reset, input_enable;
reg[15:0] input_pin;
wire[15:0] output_pin;

top topTB(clk, reset, input_pin, output_pin, input_enable);

initial // Clock generator
  begin
    clk = 0;
    forever #10 clk = !clk;
  end

initial

begin

input_enable = 0;
input_pin = 16'hf0f0;
reset = 0;
#5 reset = 1;
#5 reset = 0;

#1800

input_enable = 1;
end

endmodule
```

Fetch.v

```verilog
module Fetch
(
    clk, reset, start, MFC,
    PC_read, PC_increment,
    MAR_write, MAR_mem_read,
    MEM_RW, MEM_EN,
    MDR_mem_write, MDR_read,
    IR_write, done
);

input clk, reset, start, MFC;
output reg PC_read, PC_increment;
output reg MAR_write, MAR_mem_read;
output reg MEM_RW, MEM_EN;
output reg MDR_mem_write, MDR_read;
output reg IR_write;
output reg done;

reg[2:0] pres_state, next_state;
parameter  init = 0, st0 = 1, st1 = 2, st2 = 3, st3 = 4, WAIT1 = 5,  DONE = 6;

//FSM register
always @(posedge clk or posedge reset)
    begin:statereg
        if (reset)  pres_state <= init;
        else        pres_state <= next_state;
    end //stateregFSM Description by VerilogExample

//FSM next state logic
always @(pres_state or MFC or start)
begin: fsm
    case (pres_state)

        init: if(start) next_state <= st0;

        st0: next_state <= st1;

        st1: next_state <= WAIT1;

        WAIT1: if(!MFC) next_state <= st2;

        st2: next_state <= st3;
        st3: next_state <= DONE;

        DONE: next_state <= init;

        default: next_state<= init;

    endcase end //fsm
//Moore output definition using pres_state only
always   @(pres_state)
begin: outputs
    case (pres_state)

        init: begin
            PC_read         <= 0;
            MAR_write       <= 0;
            MAR_mem_read    <= 0;
            MEM_RW          <= 0;
            MEM_EN          <= 0;
            MDR_mem_write   <= 0;
            MDR_read        <= 0;
            IR_write        <= 0;
            PC_increment    <= 0;
            done <= 0;
        end

        st0: begin
            PC_read         <= 1;
            MAR_write       <= 1;
            MAR_mem_read    <= 0;
            MEM_RW          <= 0;
            MEM_EN          <= 0;
            MDR_mem_write   <= 0;
            MDR_read        <= 0;
            IR_write        <= 0;
            PC_increment    <= 0;
        end

        st1: begin
            PC_read         <= 0;
            MAR_write       <= 0;
            MAR_mem_read    <= 1;
            MEM_RW          <= 1;
            MEM_EN          <= 1;
            MDR_mem_write   <= 0;
            MDR_read        <= 0;
            IR_write        <= 0;
            PC_increment    <= 0;
        end

        WAIT1: begin
            // PC_read         <= 0;
            // MAR_write       <= 0;
            // MAR_mem_read    <= 0;
            // MEM_RW          <= 0;
            // MEM_EN          <= 0;
            // MDR_mem_write   <= 0;
            // MDR_read        <= 0;
            // IR_write        <= 0;
            // PC_increment    <= 0;
        end

        st2: begin
            PC_read         <= 0;
            MAR_write       <= 0;
            MAR_mem_read    <= 0;
            MEM_RW          <= 0;
            MEM_EN          <= 0;
            MDR_mem_write   <= 1;
            MDR_read        <= 0;
            IR_write        <= 0;
            PC_increment    <= 0;
        end

        st3: begin
            PC_read         <= 0;
            MAR_write       <= 0;
            MAR_mem_read    <= 0;
            MEM_RW          <= 0;
            MEM_EN          <= 0;
            MDR_mem_write   <= 0;
            MDR_read        <= 1;
            IR_write        <= 1;

        end

        DONE: begin
            PC_read         <= 0;
            MAR_write       <= 0;
            MAR_mem_read    <= 0;
            MEM_RW          <= 0;
```

```
            MEM_EN          <= 0;
            MDR_mem_write   <= 0;
            MDR_read        <= 0;
            IR_write        <= 0;
            done <= 1;
            PC_increment    <= 1;
        end

    endcase
end //fsm

endmodule
```

Fetch_tb.v

```
`timescale 1ns/1ps

module Fetch_tb;

reg clk, reset, start, MFC;
wire PC_read;
wire MAR_write, MAR_mem_read;
wire MEM_RW, MEM_EN;
wire MDR_mem_write, MDR_read;
wire IR_write;


Fetch TB
(
    clk, reset, start, MFC,
    PC_read,
    MAR_write, MAR_mem_read,
    MEM_RW, MEM_EN,
    MDR_mem_write, MDR_read,
    IR_write
);


initial // Clock generator
  begin
    clk = 1;
    forever #10 clk = !clk;
  end

initial
```

```
begin

  reset = 0;
  MFC = 1;
  start = 0;
  #1 reset = 1;
  #1 reset = 0;
  #20 start = 1;
  #10 start = 0;

  #100 MFC = 0;
  #1 MFC = 1;

  #100 start = 1;
  #5 start = 0;

  #200 MFC = 0;

end

endmodule
```

Move.v

```
module Move
(
    clk, reset, start, Ri, Rj,
    done,
    R0_write, R0_read,
    R1_write, R1_read,
    R2_write, R2_read,
    R3_write, R3_read,
    P0_write, P0_read,
              P1_read
);

input clk, reset, start;
input[5:0] Ri, Rj;

output reg R0_write, R0_read;
output reg R1_write, R1_read;
output reg R2_write, R2_read;
output reg R3_write, R3_read;
output reg P0_write, P0_read;
output reg P1_read;

output reg done;

reg[1:0] pres_state, next_state;
parameter  INIT = 0, MAIN = 1, NEXT_I = 2;


//FSM register
always @(posedge clk or posedge reset)
    begin:statereg
        if (reset)  pres_state <= INIT;
        else        pres_state <= next_state;
    end //stateregFSM Description by VerilogExample

//FSM next state logic
always @(pres_state or start)
begin: fsm
    case (pres_state)

        INIT: begin
            if(start) next_state <= MAIN;
            else next_state <= INIT;
        end
        MAIN:   next_state <= NEXT_I;
        NEXT_I: next_state <= INIT;
        default: next_state<= INIT;

    endcase end //fsm


always   @(pres_state)
begin: outputs
    case (pres_state)
        INIT: begin

            R0_write <= 0; R0_read <= 0;
            R1_write <= 0; R1_read <= 0;
            R2_write <= 0; R2_read <= 0;
            R3_write <= 0; R3_read <= 0;
            P0_write <= 0; P0_read <= 0;
            P1_read  <= 0;
            done <= 0;

        end

        MAIN: begin

            case(Ri)
                0: R0_write <= 1;
                1: R1_write <= 1;
                2: R2_write <= 1;
                3: R3_write <= 1;
                4: P0_write <= 1;
            endcase

            case(Rj)
                0: R0_read <= 1;
                1: R1_read <= 1;
                2: R2_read <= 1;
                3: R3_read <= 1;
                4: P0_read <= 1;
                5: P1_read <= 1;
            endcase
        end

        NEXT_I: begin
            R0_write <= 0; R0_read <= 0;
            R1_write <= 0; R1_read <= 0;
            R2_write <= 0; R2_read <= 0;
            R3_write <= 0; R3_read <= 0;
            P0_write <= 0; P0_read <= 0;
            P1_read  <= 0;
            done <= 1;
        end

    endcase
end //fsm

endmodule
```

Move_tb.v

```
`timescale 1ns/1ps

module Move_tb;
```

```
reg clk, reset, start;
reg[5:0] Ri, Rj;

wire R0_write, R0_read;
wire R1_write, R1_read;
wire R2_write, R2_read;
wire R3_write, R3_read;


Move TB
(
    clk, reset, start, Ri, Rj,
    start_next_I,
    R0_write, R0_read,
    R1_write, R1_read,
    R2_write, R2_read,
    R3_write, R3_read
);


initial // Clock generator
  begin
    clk = 1;
    forever #10 clk = !clk;
  end

initial

begin

    Ri = 0;
    Rj = 3;
    start = 0;
    reset = 0;
    #1 reset = 1;
    #1 reset = 0;
    #20 start = 1;
    #4 start = 0;

end

endmodule
```

Movi.v

```
// 0111 Ri Rj
module Movi
(
    clk, reset, start, Ri, num,
    out_to_bus,
    done,
    R0_write,
    R1_write,
    R2_write,
    R3_write,
    P0_write
);

input clk, reset, start;
input[5:0] Ri, num;

output reg R0_write;
output reg R1_write;
output reg R2_write;
output reg R3_write;
output reg P0_write;
output reg done;

output[15:0] out_to_bus;

reg[1:0] pres_state, next_state;

reg read;

parameter  INIT = 0, MAIN = 1, NEXT_I = 2;


//FSM register
always @(posedge clk or posedge reset)
    begin:statereg
        if (reset)  pres_state <= INIT;
        else        pres_state <= next_state;
    end //stateregFSM Description by VerilogExample

//FSM next state logic
always @(pres_state or start)
begin: fsm
    case (pres_state)
```

```
        INIT: begin
            if(start) next_state <= MAIN;
            else next_state <= INIT;
        end
        MAIN:   next_state <= NEXT_I;
        NEXT_I: next_state <= INIT;
        default: next_state<= INIT;

    endcase end //fsm


always   @(pres_state)
begin: outputs
    case (pres_state)
        INIT: begin

            R0_write <= 0;
            R1_write <= 0;
            R2_write <= 0;
            R3_write <= 0;
            P0_write <= 0;
            read <= 0;
            done <= 0;

        end

        MAIN: begin

            case(Ri)
                0: R0_write <= 1;
                1: R1_write <= 1;
                2: R2_write <= 1;
                3: R3_write <= 1;
                4: P0_write <= 1;
            endcase

            read <= 1;

        end

        NEXT_I: begin
            R0_write <= 0;
            R1_write <= 0;
            R2_write <= 0;
            R3_write <= 0;
            P0_write <= 0;
            read <= 0;
            done <= 1;
        end

    endcase
end //fsm

assign out_to_bus = read? num : 16'hzzzz;

endmodule
```

Movi_tb.v

```
`timescale 1ns/1ps

module Movi_tb;

reg clk, reset, start;
reg[5:0] Ri, num;

wire R0_write;
wire R1_write;
wire R2_write;
wire R3_write;
wire[15:0] out_to_bus;


Movi TB
(
    clk, reset, start, Ri, num,
    out_to_bus,
    start_next_I,
    R0_write,
    R1_write,
    R2_write,
    R3_write
);


initial // Clock generator
  begin
    clk = 1;
    forever #10 clk = !clk;
  end
```

```
initial

begin

    Ri = 0;
    num = 15;
    start = 0;
    reset = 0;
    #1 reset = 1;
    #1 reset = 0;
    #20 start = 1;
    #4 start = 0;

end

endmodule
```

ALU_FSM.v

```
// 0111 Ri Rj

module ALU_FSM
(
    clk, reset, start, opCode, Ri, Rj,
    done,
    R0_write, R0_read,
    R1_write, R1_read,
    R2_write, R2_read,
    R3_write, R3_read,
    P0_write, P0_read,
              P1_read,
    ALU_opControl,
    ALU_alu_out_en, ALU_writeIN1, ALU_writeIN2, ALU_read
);

input clk, reset, start;
input[3:0] opCode;
input[5:0] Ri, Rj;

output reg R0_write, R0_read;
output reg R1_write, R1_read;
output reg R2_write, R2_read;
output reg R3_write, R3_read;
```

```verilog
output reg P0_write, P0_read;
output reg P1_read;

output reg[2:0] ALU_opControl;
output reg ALU_alu_out_en, ALU_writeIN1, ALU_writeIN2, ALU_read;
output reg done;

reg[2:0] pres_state, next_state;
parameter  INIT = 0, IN1 = 1, IN2 = 2, EVAL = 3, OUT = 4, NEXT_I = 5;


//FSM register
always @(posedge clk or posedge reset)
    begin:statereg
        if (reset)  pres_state <= INIT;
        else        pres_state <= next_state;
    end //stateregFSM Description by VerilogExample

//FSM next state logic
always @(pres_state or start)
begin: fsm
    case (pres_state)

        INIT: begin
            if(start) next_state <= IN1;
            else next_state <= INIT;
        end
        IN1:   next_state <= IN2;
        IN2:   next_state <= EVAL;
        EVAL:  next_state <= OUT;
        OUT:   next_state <= NEXT_I;
        NEXT_I: next_state <= INIT;
        default: next_state<= INIT;

    endcase end //fsm


always   @(pres_state)
begin: outputs
    case (pres_state)
        INIT: begin

            R0_write <= 0; R0_read <= 0;
            R1_write <= 0; R1_read <= 0;
            R2_write <= 0; R2_read <= 0;
            R3_write <= 0; R3_read <= 0;
            P0_write <= 0; P0_read <= 0;
            P1_read  <= 0;
            ALU_opControl <= 0;
            ALU_alu_out_en <= 0; ALU_writeIN1 <=0; ALU_writeIN2 <= 0; ALU_read <=
0;
            done <= 0;

        end

        IN1: begin

            /////////////////
            R0_write <= 0;
            R1_write <= 0;
            R2_write <= 0;
            R3_write <= 0;
            P0_write <= 0;
            P1_read  <= 0;
            ALU_opControl <= 0;
            ALU_alu_out_en <= 0; ALU_writeIN2 <= 0; ALU_read <= 0;
            ////////////////////////
            case(Ri)
                0: R0_read <= 1;
                1: R1_read <= 1;
                2: R2_read <= 1;
                3: R3_read <= 1;
                4: P0_read <= 1;
            endcase

           ALU_writeIN1 <= 1;

        end

        IN2: begin

            //may be redundant
            R0_write <= 0;
            R1_write <= 0;
            R2_write <= 0;
            R3_write <= 0;
            P0_write <= 0;
            ALU_opControl <= 0;
            ALU_alu_out_en <= 0; ALU_writeIN1 <= 0; ALU_read <= 0;
            case(Ri)
                0: R0_read <= 0;
                1: R1_read <= 0;
                2: R2_read <= 0;
                3: R3_read <= 0;
                4: P0_read <= 0;
            endcase
            ///////////////////

            case(Rj)
                0: R0_read <= 1;
                1: R1_read <= 1;
                2: R2_read <= 1;
                3: R3_read <= 1;
                4: P0_read <= 1;
                5: P1_read <= 1;
            endcase

           ALU_writeIN2 <= 1;

        end

        EVAL: begin //if ALU_IN1 or ALU_IN2 changed at this phase the output will change as well because they are not set to zero in this state
            ////////////////////////////////////
            R0_write <= 0; R0_read <= 0;
            R1_write <= 0; R1_read <= 0;
            R2_write <= 0; R2_read <= 0;
            R3_write <= 0; R3_read <= 0;
            P0_write <= 0; P0_read <= 0;
            P1_read  <= 0;
            ALU_writeIN1 <=0; ALU_writeIN2 <= 0; ALU_read <= 0;
            //////////////////////////////////////////////
            ALU_alu_out_en <= 1;

            ALU_opControl <= opCode[2:0];

        end

        OUT: begin
            ////////////////////////////////////
            R0_read <= 0;
            R1_read <= 0;
            R2_read <= 0;
            R3_read <= 0;
            P0_read <= 0;
            P1_read <= 0;
            ALU_opControl <= 0;
            ALU_alu_out_en <= 0; ALU_writeIN1 <=0; ALU_writeIN2 <= 0;
            //////////////////////////////////////////////////

            ALU_read <= 1;

            case(Ri)
                0: R0_write <= 1;
                1: R1_write <= 1;
                2: R2_write <= 1;
                3: R3_write <= 1;
                4: P0_write <= 1;
            endcase
        end

        NEXT_I: begin
            R0_write <= 0; R0_read <= 0;
            R1_write <= 0; R1_read <= 0;
            R2_write <= 0; R2_read <= 0;
            R3_write <= 0; R3_read <= 0;
            P0_write <= 0; P0_read <= 0;
            P1_read  <= 0;
            ALU_opControl <= 0;
            ALU_alu_out_en <= 0; ALU_writeIN1 <=0; ALU_writeIN2 <= 0; ALU_read <=
0;
            done <= 1;
        end

    endcase
end //fsm

endmodule
```

ALU_FSM_tb.v

```
`timescale 1ns/1ps

module ALU_FSM_tb;

reg clk, reset, start;
reg[3:0] opCode;
reg[5:0] Ri, Rj;
```

```verilog
wire R0_write, R0_read;
wire R1_write, R1_read;
wire R2_write, R2_read;
wire R3_write, R3_read;

wire[2:0] ALU_opControl;
wire  ALU_alu_out_en, ALU_writeIN1, ALU_writeIN2, ALU_read;
wire  done;

ALU_FSM TB
(
    clk, reset, start, opCode, Ri, Rj,
    done,
    R0_write, R0_read,
    R1_write, R1_read,
    R2_write, R2_read,
    R3_write, R3_read,
    ALU_opControl,
    ALU_alu_out_en, ALU_writeIN1, ALU_writeIN2, ALU_read
);


initial // Clock generator
  begin
    clk = 1;
    forever #10 clk = !clk;
  end

initial

begin

    Ri = 0;
    Rj = 3;
    opCode = 1;
    start = 0;
    reset = 0;
    #1 reset = 1;
    #1 reset = 0;
    #20 start = 1;
    #4 start = 0;

end

endmodule
```

ALUI_FSM.v

```
// 0111 Ri Rj

module ALUI_FSM
(
    clk, reset, start, opCode, Ri, num,
    out_to_bus,
    done,
    R0_write, R0_read,
    R1_write, R1_read,
    R2_write, R2_read,
    R3_write, R3_read,
    P0_write, P0_read,
    ALU_opControl,
    ALU_alu_out_en, ALU_writeIN1, ALU_writeIN2, ALU_read
);

input clk, reset, start;
input[3:0] opCode;
input[5:0] Ri, num;

output reg R0_write, R0_read;
output reg R1_write, R1_read;
output reg R2_write, R2_read;
output reg R3_write, R3_read;
output reg P0_write, P0_read;
output reg[2:0] ALU_opControl;
output reg ALU_alu_out_en, ALU_writeIN1, ALU_writeIN2, ALU_read;
output reg done;

output[15:0] out_to_bus;

reg read;

reg[2:0] pres_state, next_state;
parameter  INIT = 0, IN1 = 1, IN2 = 2, EVAL = 3, OUT = 4, NEXT_I = 5;


//FSM register
always @(posedge clk or posedge reset)
    begin:statereg
        if (reset)  pres_state <= INIT;
        else        pres_state <= next_state;
    end //stateregFSM Description by VerilogExample
```

```
//FSM next state logic
always @(pres_state or start)
begin: fsm
    case (pres_state)

        INIT: begin
            if(start) next_state <= IN1;
            else next_state <= INIT;
        end
        IN1:   next_state <= IN2;
        IN2:   next_state <= EVAL;
        EVAL:  next_state <= OUT;
        OUT:   next_state <= NEXT_I;
        NEXT_I: next_state <= INIT;
        default: next_state<= INIT;

    endcase end //fsm


always   @(pres_state)
begin: outputs
    case (pres_state)
        INIT: begin

            R0_write <= 0; R0_read <= 0;
            R1_write <= 0; R1_read <= 0;
            R2_write <= 0; R2_read <= 0;
            R3_write <= 0; R3_read <= 0;
            P0_write <= 0; P0_read <= 0;
            ALU_opControl <= 0;
            ALU_alu_out_en <= 0; ALU_writeIN1 <=0; ALU_writeIN2 <= 0; ALU_read <=
0;
            read <= 0;
            done <= 0;

        end

        IN1: begin

            /////////////////
            R0_write <= 0;
            R1_write <= 0;
            R2_write <= 0;
            R3_write <= 0;
            P0_write <= 0;
            ALU_opControl <= 0;
            ALU_alu_out_en <= 0; ALU_writeIN2 <= 0; ALU_read <= 0;
            read <= 0;
            ////////////////////////
            case(Ri)
                0: R0_read <= 1;
                1: R1_read <= 1;
                2: R2_read <= 1;
                3: R3_read <= 1;
                4: P0_read <= 1;
            endcase

           ALU_writeIN1 <= 1;

        end

        IN2: begin

            //may be redundant
            R0_write <= 0;
            R1_write <= 0;
            R2_write <= 0;
            R3_write <= 0;
            P0_write <= 0;
            ALU_opControl <= 0;
            ALU_alu_out_en <= 0; ALU_writeIN1 <= 0; ALU_read <= 0;

            case(Ri)
                0: R0_read <= 0;
                1: R1_read <= 0;
                2: R2_read <= 0;
                3: R3_read <= 0;
                4: P0_read <= 0;
            endcase
            ///////////////////

           read <= 1;

           ALU_writeIN2 <= 1;

        end

        EVAL: begin //if ALU_IN1 or ALU_IN2 changed at this phase the output will
change as well because they are not set to zero in this state
            ////////////////////////////////////
            R0_write <= 0; R0_read <= 0;
            R1_write <= 0; R1_read <= 0;
            R2_write <= 0; R2_read <= 0;
            R3_write <= 0; R3_read <= 0;
            P0_write <= 0; P0_read <= 0;
            ALU_writeIN1 <=0; ALU_writeIN2 <= 0; ALU_read <= 0;
            read <= 0;
            //////////////////////////////////////////////

            ALU_alu_out_en <= 1;

            ALU_opControl <= opCode;

        end

        OUT: begin
            ////////////////////////////////////
            R0_read <= 0;
            R1_read <= 0;
            R2_read <= 0;
            R3_read <= 0;
            P0_read <= 0;
            ALU_opControl <= 0;
            ALU_alu_out_en <= 0; ALU_writeIN1 <=0; ALU_writeIN2 <= 0;
            read <= 0;
            //////////////////////////////////////////////////

            ALU_read <= 1;

            case(Ri)
                0: R0_write <= 1;
                1: R1_write <= 1;
                2: R2_write <= 1;
                3: R3_write <= 1;
                4: P0_write <= 1;
            endcase
        end

        NEXT_I: begin
            R0_write <= 0; R0_read <= 0;
            R1_write <= 0; R1_read <= 0;
            R2_write <= 0; R2_read <= 0;
            R3_write <= 0; R3_read <= 0;
            P0_write <= 0; P0_read <= 0;
            ALU_opControl <= 0;
```

```
            ALU_alu_out_en <= 0; ALU_writeIN1 <=0; ALU_writeIN2 <= 0; ALU_read <=
0;
            read <= 0;
            done <= 1;
        end

    endcase
end //fsm

assign out_to_bus = read? num : 16'hzzzz;

endmodule
```

ALUI_FSM_tb.v

```
`timescale 1ns/1ps

module ALU_FSM_tb;

reg clk, reset, start;
reg[3:0] opCode;
reg[5:0] Ri, num;

wire R0_write, R0_read;
wire R1_write, R1_read;
wire R2_write, R2_read;
wire R3_write, R3_read;

wire[2:0] ALU_opControl;
wire  ALU_alu_out_en, ALU_writeIN1, ALU_writeIN2, ALU_read;
wire  done;
wire[15:0] out_to_bus;

ALUI_FSM TB
(
    clk, reset, start, opCode, Ri, num,
    out_to_bus,
    done,
    R0_write, R0_read,
    R1_write, R1_read,
    R2_write, R2_read,
    R3_write, R3_read,
    ALU_opControl,
    ALU_alu_out_en, ALU_writeIN1, ALU_writeIN2, ALU_read
);
```

```
initial // Clock generator
  begin
    clk = 1;
    forever #10 clk = !clk;
  end

initial

begin

    Ri = 0;
    num = 6'b111111;
    opCode = 5;
    start = 0;
    reset = 0;
    #1 reset = 1;
    #1 reset = 0;
    #20 start = 1;
    #4 start = 0;

end

endmodule
```

Store.v

```
module Store
(
    clk, reset, start, MFC, Ri, Rj,
    R0_read,
    R1_read,
    R2_read,
    R3_read,
    P0_read,
    P1_read,
    MAR_write, MAR_mem_read,
    MEM_RW, MEM_EN,
    MDR_mem_read, MDR_write,
    done
);
```

```verilog
input clk, reset, start, MFC;
input[5:0] Ri, Rj;

output reg R0_read;
output reg R1_read;
output reg R2_read;
output reg R3_read;
output reg P0_read;
output reg P1_read;
output reg MAR_write, MAR_mem_read;
output reg MEM_RW, MEM_EN;
output reg MDR_mem_read, MDR_write;
output reg done;


reg[2:0] pres_state, next_state;
parameter  st0 = 0, st1 = 1, st2 = 2, WAIT1 = 4, init = 5, DONE = 6;

//FSM register
always @(posedge clk or posedge reset)
    begin:statereg
        if (reset)  pres_state <= init;
        else        pres_state <= next_state;
    end //stateregFSM Description by VerilogExample

//FSM next state logic
always @(pres_state or MFC or start)
begin: fsm
    case (pres_state)

        init: begin
            if(start) next_state <= st0;
            else next_state <= init;
        end

        st0: next_state <= st1;

        st1: next_state <= st2;

        st2: next_state <= WAIT1;

        WAIT1: if(!MFC) next_state <= DONE;

        DONE: next_state <= init;
        default: next_state<= init;

    endcase end //fsm

//Moore output definition using pres_state only
always   @(pres_state)
begin: outputs
    case (pres_state)

        init: begin

            R0_read      <= 0;
            R1_read      <= 0;
            R2_read      <= 0;
            R3_read      <= 0;
            P0_read      <= 0;
            P1_read      <= 0;
            MAR_write    <= 0; MAR_mem_read <= 0;
            MEM_RW       <= 0; MEM_EN       <= 0;
            MDR_mem_read <= 0; MDR_write    <= 0;
            done <= 0;

        end

        st0: begin

            case(Ri)

                0: R0_read <= 1;
                1: R1_read <= 1;
                2: R2_read <= 1;
                3: R3_read <= 1;
                4: P0_read <= 1;
                5: P1_read <= 1;

            endcase

            MAR_write    <= 0; MAR_mem_read <= 0;
            MEM_RW       <= 0; MEM_EN       <= 0;
            MDR_mem_read <= 0; MDR_write    <= 1;

            end

        st1: begin
        case(Ri)

        0: R0_read <= 0;
        1: R1_read <= 0;
        2: R2_read <= 0;
        3: R3_read <= 0;
        4: P0_read <= 0;
        5: P1_read <= 0;

        endcase

        case(Rj)

            0: R0_read <= 1;
            1: R1_read <= 1;
            2: R2_read <= 1;
            3: R3_read <= 1;
            4: P0_read <= 1;
            5: P1_read <= 1;

        endcase

        MAR_write    <= 1; MAR_mem_read <= 0;
        MEM_RW       <= 0; MEM_EN       <= 0;
        MDR_mem_read <= 0; MDR_write    <= 0;

    end

    st2: begin
        R0_read      <= 0;
        R1_read      <= 0;
        R2_read      <= 0;
        R3_read      <= 0;
        P0_read      <= 0;
        P1_read      <= 0;
        MAR_write    <= 0; MAR_mem_read <= 1;
        MEM_RW       <= 0; MEM_EN       <= 1;
        MDR_mem_read <= 1; MDR_write    <= 0;
    end

    WAIT1: begin

        // R0_read      <= 0;
        // R1_read      <= 0;
        // R2_read      <= 0;
```

```
            // R3_read      <= 0;
            // P0_read      <= 0;
            // P1_read      <= 0;
            // MAR_write    <= 0; MAR_mem_read <= 0;
            // MEM_RW       <= 0; MEM_EN       <= 0;
            // MDR_mem_read <= 0; MDR_write    <= 0;

        end

        DONE: begin

            R0_read      <= 0;
            R1_read      <= 0;
            R2_read      <= 0;
            R3_read      <= 0;
            P0_read      <= 0;
            P1_read      <= 0;
            MAR_write    <= 0; MAR_mem_read <= 0;
            MEM_RW       <= 0; MEM_EN       <= 0;
            MDR_mem_read <= 0; MDR_write    <= 0;
            done <= 0;
            done <= 1;

        end

    endcase
end //fsm

endmodule
```

Store_tb.v

```
`timescale 1ns/1ps

module Store_tb;

reg clk, reset, start, MFC;
reg[5:0] Ri, Rj;
wire R0_read;
wire R1_read;
wire R2_read;
wire R3_read;
wire P0_read;
wire MAR_write, MAR_mem_read;
wire MEM_RW, MEM_EN;
wire MDR_mem_read, MDR_write;
wire done;

Store TB
(
    clk, reset, start, MFC, Ri, Rj,
    R0_read,
    R1_read,
    R2_read,
    R3_read,
    P0_read,
    MAR_write, MAR_mem_read,
    MEM_RW, MEM_EN,
    MDR_mem_read, MDR_write,
    done
);


initial // Clock generator
  begin
    clk = 1;
    forever #10 clk = !clk;
  end

initial

begin

  reset = 0;
  MFC = 1;
  start = 0;
  Ri = 6'hffffff;
  Rj = 0;
  #1 reset = 1;
  #1 reset = 0;
  #20 start = 1;
  #10 start = 0;

  #100 MFC = 0;
  #1 MFC = 1;

  #100 start = 1;
  #5 start = 0;

  #200 MFC = 0;
```

```
end

endmodule
```

Load.v

```
module Load
(
    clk, reset, start, MFC, Ri, Rj,
    R0_read, R0_write,
    R1_read, R1_write,
    R2_read, R2_write,
    R3_read, R3_write,
    P0_read, P0_write,
    MAR_write, MAR_mem_read,
    MEM_RW, MEM_EN,
    MDR_mem_write, MDR_read,
    done
);

input clk, reset, start, MFC;
input[5:0] Ri, Rj;

output reg R0_read, R0_write;
output reg R1_read, R1_write;
output reg R2_read, R2_write;
output reg R3_read, R3_write;
output reg P0_read, P0_write;
output reg MAR_write, MAR_mem_read;
output reg MEM_RW, MEM_EN;
output reg MDR_mem_write, MDR_read;
output reg done;


reg[2:0] pres_state, next_state;
parameter  st0 = 0, st1 = 1, st2 = 2, st3 = 3, WAIT1 = 4, init = 5, DONE = 6;

//FSM register
always @(posedge clk or posedge reset)
    begin:statereg
        if (reset)  pres_state <= init;
        else        pres_state <= next_state;
    end //stateregFSM Description by VerilogExample
```

```
//FSM next state logic
always @(pres_state or MFC or start)
begin: fsm
    case (pres_state)

        init: begin
            if(start) next_state <= st0;
            else next_state <= init;
        end

        st0: next_state <= st1;

        st1: next_state <= WAIT1;

        WAIT1: if(!MFC) next_state <= st2;

        st2: next_state <= st3;

        st3: next_state <= DONE;

        DONE: next_state <= init;

        default: next_state<= init;

    endcase end //fsm

//Moore output definition using pres_state only
always   @(pres_state)
begin: outputs
    case (pres_state)

        init: begin
            R0_write <= 0; R0_read <= 0;
            R1_write <= 0; R1_read <= 0;
            R2_write <= 0; R2_read <= 0;
            R3_write <= 0; R3_read <= 0;
            P0_write <= 0; P0_read <= 0;
            MAR_write       <= 0; MAR_mem_read  <= 0;
            MEM_RW          <= 0; MEM_EN       <= 0;
                                  MDR_mem_write <= 0;
            MDR_read        <= 0;
            done <= 0;
        end
        st0: begin

            R0_write <= 0;
            R1_write <= 0;
            R2_write <= 0;
            R3_write <= 0;
            P0_write <= 0;

            case(Ri)

                0: R0_read <= 1;
                1: R1_read <= 1;
                2: R2_read <= 1;
                3: R3_read <= 1;
                4: P0_read <= 1;

            endcase

            MAR_write       <= 1; MAR_mem_read  <= 0;
            MEM_RW          <= 0; MEM_EN        <= 0;
                                  MDR_mem_write <= 0;
            MDR_read        <= 0;

            end

        st1: begin

            R0_write <= 0; R0_read <= 0;
            R1_write <= 0; R1_read <= 0;
            R2_write <= 0; R2_read <= 0;
            R3_write <= 0; R3_read <= 0;
            P0_write <= 0; P0_read <= 0;
            MAR_write       <= 0; MAR_mem_read  <= 1;
            MEM_RW          <= 1; MEM_EN        <= 1;
                                  MDR_mem_write <= 0;
            MDR_read        <= 0;

        end

        WAIT1: begin

            // R0_write <= 0; R0_read <= 0;
            // R1_write <= 0; R1_read <= 0;
            // R2_write <= 0; R2_read <= 0;
            // R3_write <= 0; R3_read <= 0;
```

```verilog
            // P0_write <= 0; P0_read <= 0;
            // MAR_write       <= 0; MAR_mem_read  <= 0;
            // MEM_RW          <= 0; MEM_EN        <= 0;
            //                       MDR_mem_write <= 0;
            // MDR_read        <= 0;

        end

        st2: begin
            R0_write <= 0; R0_read <= 0;
            R1_write <= 0; R1_read <= 0;
            R2_write <= 0; R2_read <= 0;
            R3_write <= 0; R3_read <= 0;
            P0_write <= 0; P0_read <= 0;
            MAR_write       <= 0; MAR_mem_read  <= 0;
            MEM_RW          <= 0; MEM_EN        <= 0;
                                  MDR_mem_write <= 1;
            MDR_read        <= 0;
        end

        st3: begin

            R0_read <= 0;
            R1_read <= 0;
            R2_read <= 0;
            R3_read <= 0;
            P0_read <= 0;
            MAR_write       <= 0; MAR_mem_read  <= 0;
            MEM_RW          <= 0; MEM_EN        <= 0;
                                  MDR_mem_write <= 0;
            MDR_read        <= 1;
            case(Rj)

                0: R0_write <= 1;
                1: R1_write <= 1;
                2: R2_write <= 1;
                3: R3_write <= 1;
                4: P0_write <= 1;

            endcase

        end

        DONE: begin
```

```
            R0_write <= 0; R0_read <= 0;
            R1_write <= 0; R1_read <= 0;
            R2_write <= 0; R2_read <= 0;
            R3_write <= 0; R3_read <= 0;
            P0_write <= 0; P0_read <= 0;
            MAR_write       <= 0; MAR_mem_read  <= 0;
            MEM_RW          <= 0; MEM_EN        <= 0;
                                  MDR_mem_write <= 0;
            MDR_read        <= 0;

            done <= 1;
        end

    endcase
end //fsm

endmodule
```

Load_tb.v

```
`timescale 1ns/1ps

module Load_tb;

reg clk, reset, start, MFC;
reg[5:0] Ri, Rj;
wire R0_read, R0_write;
wire R1_read, R1_write;
wire R2_read, R2_write;
wire R3_read, R3_write;
wire P0_read, P0_write;
wire MAR_write, MAR_mem_read;
wire MEM_RW, MEM_EN;
wire MDR_mem_write, MDR_read;
wire done;

Load TB
(
    clk, reset, start, MFC, Ri, Rj,
    R0_read, R0_write,
    R1_read, R1_write,
    R2_read, R2_write,
    R3_read, R3_write,
    P0_read, P0_write,
    MAR_write, MAR_mem_read,
    MEM_RW, MEM_EN,
    MDR_mem_write, MDR_read,
    done
);


initial // Clock generator
  begin
    clk = 1;
    forever #10 clk = !clk;
  end

initial

begin

  reset = 0;
  MFC = 1;
  start = 0;
  Ri = 6'hffffff;
  Rj = 0;
  #1 reset = 1;
  #1 reset = 0;
  #20 start = 1;
  #10 start = 0;

  #100 MFC = 0;
  #1 MFC = 1;

  #100 start = 1;
  #5 start = 0;

  #200 MFC = 0;

end

endmodule
```

ALU.v

```
//module ALU (reset, clk, in_from_bus, writeIN1, writeIN2, alu_out_en, read,
OpControl, out_to_bus);
module ALU (clk, reset, in_from_bus, out_to_bus, read, writeIN1, writeIN2,
alu_out_en, OpControl);
```

```verilog
//parameters
parameter ADD  = 0;
parameter SUB  = 1;
parameter NOT  = 2;
parameter AND  = 3;
parameter OR   = 4;
parameter XOR  = 5;
parameter XNOR = 6;

//inputs
input clk;
input reset;
input[15:0] in_from_bus;
input writeIN1;
input writeIN2;
input alu_out_en;
input read;
input[2:0] OpControl;

//outputs
output [15:0] out_to_bus;

//internal signals
reg[15:0] IN1_reg;
reg[15:0] IN2_reg;
reg[15:0] OUT_reg;

always @(posedge clk or posedge reset)
begin
    if(reset) begin
        IN1_reg <= 0;
        IN2_reg <= 0;
        end
    else begin
        if(writeIN1) IN1_reg <= in_from_bus;
        if(writeIN2) IN2_reg <= in_from_bus;
        end
end

//implementation
always @(*)
begin
if(reset) OUT_reg <= 0;
else if(alu_out_en) begin
        case(OpControl)
```

```
            ADD:        OUT_reg <= IN1_reg + IN2_reg ;
            SUB:        OUT_reg <= IN1_reg - IN2_reg ;
            NOT:        OUT_reg <= ~IN1_reg          ;
            AND:        OUT_reg <= IN1_reg & IN2_reg ;
            OR:         OUT_reg <= IN1_reg | IN2_reg ;
            XOR:        OUT_reg <= IN1_reg ^ IN2_reg ;
            XNOR:       OUT_reg <= IN1_reg ~^ IN2_reg;
            default:    OUT_reg <= 16'h0000          ;
        endcase
    end
end

assign out_to_bus = read? OUT_reg : 16'hzzzz;

endmodule
```

ALU_tb.v

```
`timescale 1ns/1ps

module ALU_tb;

//parameters
parameter ADD  = 0;
parameter SUB  = 1;
parameter NOT  = 2;
parameter AND  = 3;
parameter OR   = 4;
parameter XOR  = 5;
parameter XNOR = 6;

//inputs
reg reset;
reg clk;
reg[15:0] Bus_in;
reg IN1_en;
reg IN2_en;
reg OUT_en;
reg OUT_reg_en;
reg[2:0] OpControl;

//outputs
wire[15:0] Bus_out;
```

```
ALU ALU_c (reset, clk, Bus_in, IN1_en, IN2_en, OUT_en, OUT_reg_en, OpControl,
Bus_out);

initial // Clock generator
  begin
    clk = 1;
    forever #10 clk = !clk;
  end

initial

begin

    Bus_in = 16'h55;
    reset = 0;
    IN1_en = 0;
    IN2_en = 0;
    OUT_en = 0;
    OUT_reg_en = 0;
    OpControl = 0;
    #1
    reset = 1;
    #1
    reset = 0;
    IN1_en = 1;
    IN2_en = 1;
    OUT_en = 1;
    #10 OpControl = ADD;
    OUT_reg_en = 1;
    #10 OpControl = SUB;
    #10 OpControl = NOT;
    #10 OpControl = AND;
    #10 OpControl = OR;
    #10 OpControl = XOR;
    #10 OpControl = XNOR;
    #10 OpControl = 7;
    #10 OUT_en = 0;
    #10;
    $stop;

end

endmodule
```

Decoder.v

```
module Decoder(enable, opCode, start_move, start_movi, start_ALU, start_ALUI,
start_load, start_store);

input enable;
input[3:0] opCode;

output reg start_move, start_movi, start_ALU, start_ALUI, start_load,
start_store;

always @(*)
begin
    if(enable) begin
    case(opCode)

        4'b0000: begin
            start_move <= 0;
            start_movi <= 0;
            start_ALU  <= 1;
            start_ALUI <= 0;
            start_load <= 0;
            start_store<= 0;
        end
        4'b0001: begin
            start_move <= 0;
            start_movi <= 0;
            start_ALU  <= 1;
            start_ALUI <= 0;
            start_load <= 0;
            start_store<= 0;
        end
        4'b0010: begin
            start_move <= 0;
            start_movi <= 0;
            start_ALU  <= 1;
            start_ALUI <= 0;
            start_load <= 0;
            start_store<= 0;
        end
        4'b0011: begin
            start_move <= 0;
            start_movi <= 0;
            start_ALU  <= 1;
            start_ALUI <= 0;
            start_load <= 0;
            start_store<= 0;
        end
        4'b0100: begin
            start_move <= 0;
            start_movi <= 0;
            start_ALU  <= 1;
            start_ALUI <= 0;
            start_load <= 0;
            start_store<= 0;
        end
        4'b0101: begin
            start_move <= 0;
            start_movi <= 0;
            start_ALU  <= 1;
            start_ALUI <= 0;
            start_load <= 0;
            start_store<= 0;
        end
        4'b0110: begin
            start_move <= 0;
            start_movi <= 0;
            start_ALU  <= 1;
            start_ALUI <= 0;
            start_load <= 0;
            start_store<= 0;
        end

        4'b1000: begin
            start_move <= 0;
            start_movi <= 0;
            start_ALU  <= 0;
            start_ALUI <= 1;
            start_load <= 0;
            start_store<= 0;
        end

        4'b1001: begin
            start_move <= 0;
            start_movi <= 0;
            start_ALU  <= 0;
            start_ALUI <= 1;
            start_load <= 0;
            start_store<= 0;
        end
        4'b0111: begin
            start_move <= 1;
            start_movi <= 0;
            start_ALU  <= 0;
            start_ALUI <= 0;
            start_load <= 0;
            start_store<= 0;
        end
        4'b1111: begin
            start_move <= 0;
            start_movi <= 1;
            start_ALU  <= 0;
            start_ALUI <= 0;
            start_load <= 0;
            start_store<= 0;
        end

         4'b1010: begin
            start_move <= 0;
            start_movi <= 0;
            start_ALU  <= 0;
            start_ALUI <= 0;
            start_load <= 1;
            start_store<= 0;
        end

        4'b1011: begin
            start_move <= 0;
            start_movi <= 0;
            start_ALU  <= 0;
            start_ALUI <= 0;
            start_load <= 0;
            start_store<= 1;
        end

    endcase
    end
    else begin
        start_move <= 0;
        start_movi <= 0;
        start_ALU  <= 0;
        start_ALUI <= 0;
        start_load <= 0;
        start_store<= 0;
    end
end

endmodule
```

IR.v

```
//module IR(clk, reset, write, FSM_read, in_from_bus, out_to_decoder); //FSM_read
is not necessary
module IR   (clk, reset, in_from_bus, out_to_decoder,       write, FSM_read);

input clk, reset, write, FSM_read;
input [15:0] in_from_bus;
output[15:0] out_to_decoder;

reg[15:0] register;

always@(posedge clk or posedge reset)
begin
  if(reset) register <= 0;
  else if(write) register <= in_from_bus;
end

//I may not need the FSM_read
assign out_to_decoder = FSM_read? register : 16'hzzzz;

endmodule
```

MAR.v

```
//module MAR(clk, reset, write, read, in_from_bus, out_to_mem); //read is not
necessary
module MAR  (clk, reset, in_from_bus, out_to_mem, write, mem_read);

input clk, reset, write, mem_read;
input [15:0] in_from_bus;
output[15:0] out_to_mem;

reg[15:0] register;

always@(posedge clk or posedge reset)
begin
  if(reset) register <= 0;
  else if(write) register <= in_from_bus;
```

```
end

assign out_to_mem = mem_read? register : 16'hzzzz;

endmodule
```

MAR_tb.v

```
`timescale 1ns/1ns
module MAR_tb();

reg clk;
reg reset;
reg in_en;
reg out_en;
reg[15:0] data_in;
wire[15:0] data_out;

MAR TB(clk, reset, in_en, out_en, data_in, data_out);


initial // Clock generator
  begin
    clk = 1;
    forever #10 clk = !clk;
  end

initial
begin
   reset = 0;
   in_en = 0;
   out_en = 0;
   data_in = 16'hFFFF;
   out_en = 0;
#5 reset = 1;
#5 reset = 0;
#10 out_en = 1;
    in_en = 1;
#10 out_en = 0;

end
endmodule
```

MDR.v

```
//module MDR(clk, reset, write, mem_write, read, mem_read, in_from_bus,
out_to_bus, in_from_mem, out_to_mem);
module MDR  (clk, reset, in_from_bus, out_to_bus, read, write, in_from_mem,
out_to_mem, mem_read, mem_write);

input clk, reset, write, mem_write, read, mem_read;
input[15:0] in_from_bus;
input[15:0] in_from_mem;

output[15:0] out_to_bus;
output[15:0] out_to_mem;


reg[15:0] register;


always@(posedge clk or posedge reset)
begin
  if(reset) register <= 0;
  else begin//should make sure that one of them is enabled at a time
        if(write && !mem_write) register <= in_from_bus;
        if(mem_write && !write) register <= in_from_mem;
  end
end

assign out_to_bus = read? register: 16'hzzzz;
assign out_to_mem = mem_read? register : 16'hzzzz;

endmodule
```

MDR_tb.v

```
`timescale 1ns/1ns
module MDR_tb();

//inputs
reg clk;
reg reset;
reg in_bus_en, in_mem_en, out_bus_en, out_mem_en;
reg[15:0] bus_in;
reg[15:0] mem_in;

//outputs
wire[15:0] bus_out;
```

```
wire[15:0] mem_out;



MDR TB(clk, reset, in_bus_en, in_mem_en, out_bus_en, out_mem_en, bus_in, bus_out,
mem_in, mem_out);


initial // Clock generator
  begin
    clk = 1;
    forever #10 clk = !clk;
  end

initial begin

reset = 0;
in_bus_en = 0; in_mem_en = 0; out_bus_en = 0; out_mem_en = 0;
bus_in = 1;
mem_in = 15;
#1 reset = 1;
#1 reset = 0;
in_bus_en = 1;
#20
in_bus_en = 0;
in_mem_en = 1;
out_bus_en = 1;
out_mem_en = 1;
#20
in_bus_en = 0; in_mem_en = 0; out_bus_en = 0; out_mem_en = 0;
reset = 1;
#1 reset = 0;



end
endmodule
```

PC.v

```
//module PC(clk, reset, increment, read, out_to_bus);
module PC   (clk, reset,              out_to_bus, read,        increment);

input clk, reset, increment, read;
```

```
output[15:0] out_to_bus;

reg[15:0] register;

always@(posedge clk or posedge reset)
begin
  if(reset) register <= 0;
  else if(increment) register <= register + 1;
end

assign out_to_bus = read? register : 16'hzzzz;

endmodule
```

PC_tb.v

```
`timescale 1ns/1ns
module PC_tb();

reg clk, reset, inc, out_en;

wire[15:0] data_out;

PC TB(clk, reset, inc, out_en, data_out);

initial // Clock generator
  begin
    clk = 1;
    forever #10 clk = !clk;
  end

initial begin


reset = 0;
inc = 0;
out_en = 0;
#10
reset = 1;
#10
reset = 0;
out_en = 1;
#10
```

```
inc = 1;
#20 inc =0;



end
endmodule
```

Port0.v

```
//module Port0(clk, reset, read, write, in_from_bus, out_to_bus, out); //read is
for the tri state buffer
module Port0(clk, reset, in_from_bus, out_to_bus, read, write, out_pin);

input clk, reset, read, write; //read means read from the bus, the out_pin is
always out
input [15:0] in_from_bus;

output[15:0] out_to_bus;
output[15:0] out_pin;

reg[15:0] register;

always@(posedge clk or posedge reset)
begin
  if(reset) register <= 0;
  else if(write) register <= in_from_bus; //write into output when write == 1
end

assign out_pin = register;
assign out_to_bus = read? register : 16'hzzzz; //read from output when read == 1

endmodule
```

Port0_tb.v

```
`timescale 1ns/1ns
module Port0_tb();

reg clk, reset, read, write;
reg[15:0] bus_in;

wire[15:0] bus_out;
```

```
wire[15:0] out;

Port0 TB(clk, reset, read, write, bus_in, bus_out, out);


initial // Clock generator
  begin
    clk = 1;
    forever #10 clk = !clk;
  end

initial begin

//as an output
read = 0;
write = 0;
reset = 0;
bus_in = 16'hffff;
#3 reset = 1;
#3 reset = 0;
#10 read = 1;
write = 1;
#10 read = 0;



end
endmodule
```

Port1.v

```
//module Port1(clk, reset, read, in, out_to_bus);
module Port1(clk, reset,              out_to_bus, read,        in_pin);

input clk, reset, read;
input[15:0] in_pin;

output[15:0] out_to_bus;

reg[15:0] register;

always@(posedge clk or posedge reset)
begin
  if(reset) register <= 0;
  else register <= in_pin;
end

assign out_to_bus = read? register : 16'hzzzz;

endmodule
```

Port1_tb.v

```
`timescale 1ns/1ns
module Port1_tb();
//bus_out means out to bus
reg clk, reset, read;
reg[15:0] in;

wire[15:0] bus_out;


Port1 TB(clk, reset, read, in, bus_out);


initial // Clock generator
  begin
    clk = 1;
    forever #10 clk = !clk;
  end

initial begin

//as an output
read = 0;
reset = 0;
in = 0;
#3 reset = 1;
#3 reset = 0;
in = 9;
#10 read = 1;
#10 read = 0;



end
endmodule
```

R0.v

```verilog
//module R0(clk, reset, read, write, in_from_bus, out_to_bus);
module R0   (clk, reset, in_from_bus, out_to_bus, read, write);

input clk, reset, read, write;
input[15:0] in_from_bus;

output[15:0] out_to_bus;

reg[15:0] register;

always@(posedge clk or posedge reset)
begin
  if(reset) register <= 0;
  else if(write) register <= in_from_bus;
end

assign out_to_bus = read? register : 16'hzzzz;

endmodule
```

R1.v

```verilog
//module R0(clk, reset, read, write, in_from_bus, out_to_bus);
module R1   (clk, reset, in_from_bus, out_to_bus, read, write);

input clk, reset, read, write;
input[15:0] in_from_bus;

output[15:0] out_to_bus;

reg[15:0] register;

always@(posedge clk or posedge reset)
begin
  if(reset) register <= 0;
  else if(write) register <= in_from_bus;
end

assign out_to_bus = read? register : 16'hzzzz;

endmodule
```

R2.v

```
//module R0(clk, reset, read, write, in_from_bus, out_to_bus);
module R2   (clk, reset, in_from_bus, out_to_bus, read, write);

input clk, reset, read, write;
input[15:0] in_from_bus;

output[15:0] out_to_bus;

reg[15:0] register;

always@(posedge clk or posedge reset)
begin
  if(reset) register <= 0;
  else if(write) register <= in_from_bus;
end

assign out_to_bus = read? register : 16'hzzzz;

endmodule
```

R3.v

```
//module R0(clk, reset, read, write, in_from_bus, out_to_bus);
module R3   (clk, reset, in_from_bus, out_to_bus, read, write);

input clk, reset, read, write;
input[15:0] in_from_bus;

output[15:0] out_to_bus;

reg[15:0] register;

always@(posedge clk or posedge reset)
begin
  if(reset) register <= 0;
  else if(write) register <= in_from_bus;
end

assign out_to_bus = read? register : 16'hzzzz;
```

```
endmodule
```

R_tb.v

```
`timescale 1ns/1ns
module R_tb();

reg clk, reset, read, write;
reg[15:0] bus_in;

wire[15:0] bus_out;
wire[15:0] out;

R0 TB(clk, reset, read, write, bus_in, bus_out);


initial // Clock generator
  begin
    clk = 1;
    forever #10 clk = !clk;
  end

initial begin

reset = 0;
read = 0;
write = 0;
bus_in = 0;
#10 reset = 1;
#3 reset = 0;
read = 1;
#15
bus_in = 15;
write = 1;




end
endmodule
```