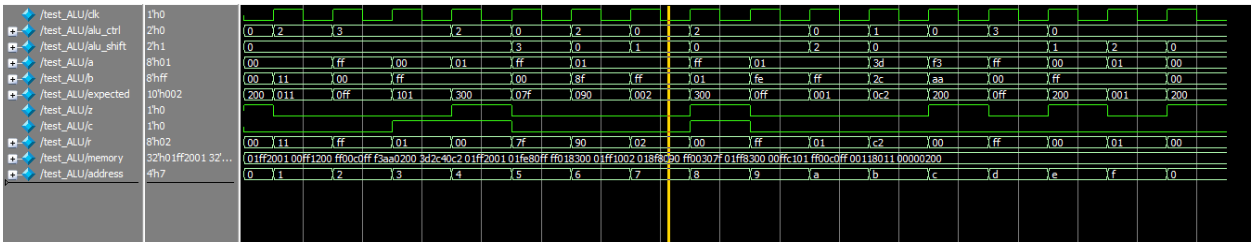
1. FSM diagram

The INITIAL STATE is asserted when processor is reset by CLB signal, all control signal is reset on the mean time. The FETCH STATE get instruction from instruction memory, loadIR is switched to high. The EXECUTE STATE executes instruction. If it is HALT instruction, the incPC will set to zero, the next instruction will always be HALT, the PC address is not updated.

Control Signal State	incPC	selPC	loadPC	loadReg	loadAcc	loadAlu	selAcc	SelAlu	loadIR
INITIAL	0								
FETCH	0							1000	1
EXECUTE	Depends on Opcode								0

2. Waveform result

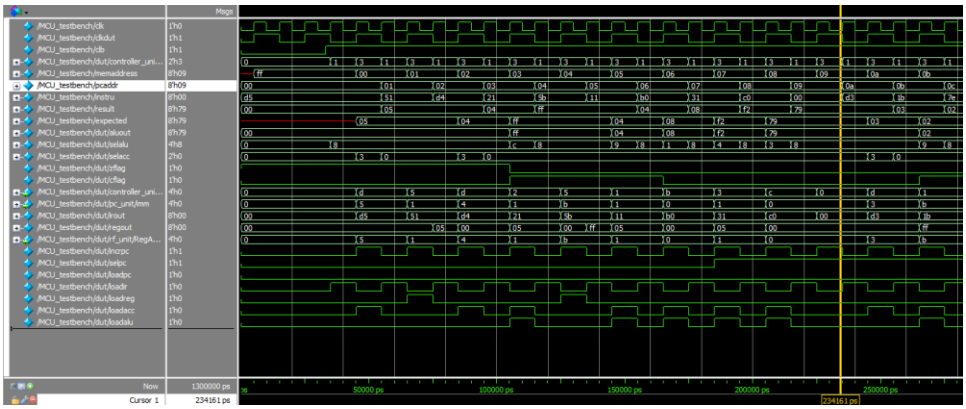
ALU test result:



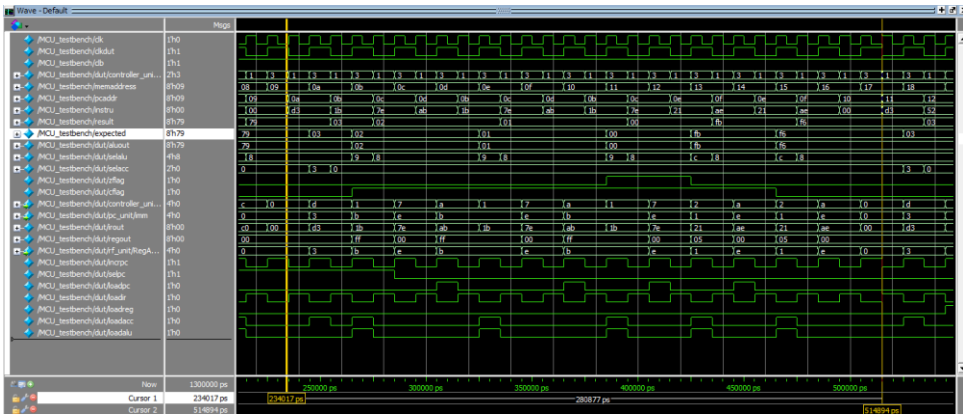
The processor has CPI as 2, each clock cycle is FETCH and EXECUTE. The result of current instruction is generated when next instruction is fetched from memory. The top-level testbench delays one cycle of PC address updating to align the result and instruction on the same clock cycle. There is clock signal for controller module controlling state, and clkdut signal lasting for 2 clk cycles that complete state transition in 1 loop.

Top design waveform:

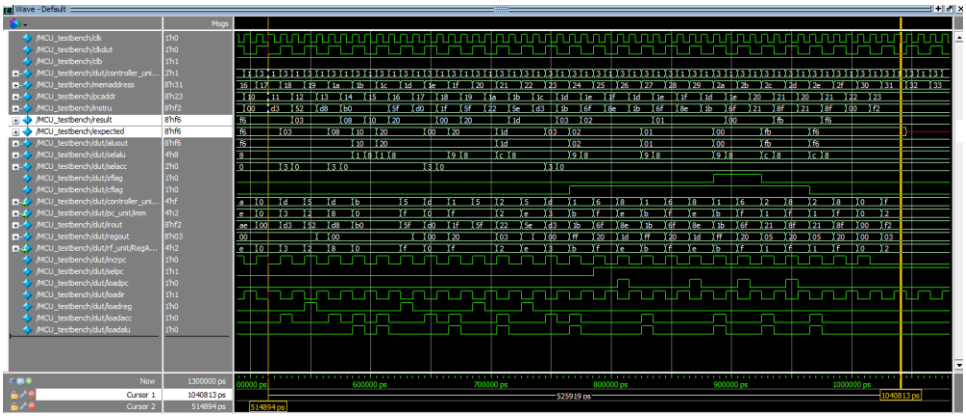
First part of testbench. The testbench first reset the processor, then start to run instruction in address 0.



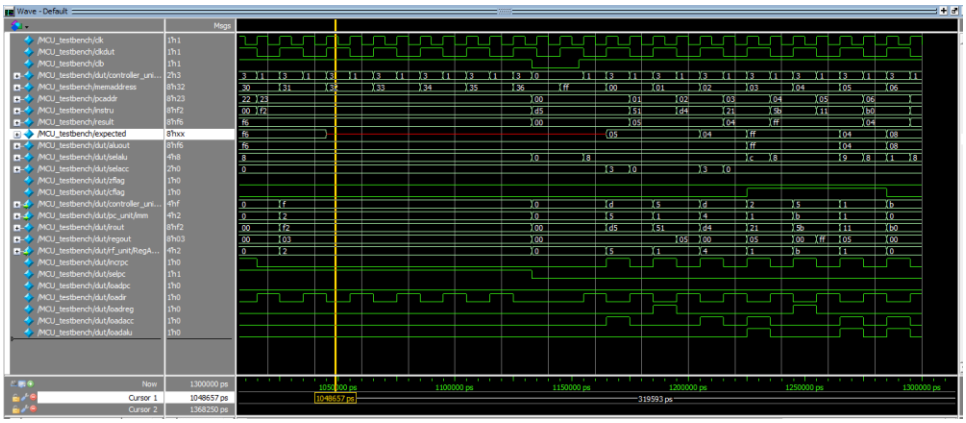
Second part of testbench. Two loops are executed in this part.



Third part of testbench. Similar two loops comparing to above are executed here.



Last part of simulation. Reset the processor after it enters HALT condition, then rerun the testbench.



3.

Testbench consist of three parts. First part is about the basic arithmetic operation taken place on ALU units, and run through the processor to ensure the top-level design working properly. ADD, SUB, NOR, SHL, SHR, MOV in 2 directions, LD imm, 8 operations are tested in first part of testbench. Second part is testing jumping operations of both JC and JZ that go to immediate addresses. Two loops are coded to test both the jumping and not jumping decision of cout and zout controlling. Third part tests jumping operations that go to addresses store in registers. It modifies the code from second part by calculating the jumping address ahead of jump operation, and storing the address value to register 16 and 15. NOP operations insert in the boundary of different testing parts, and HALT operation is placed on the end of testbench.

Confused operations:
Load immediate 0 to accumulator, then jump to another address if zout is high. Zout signal is controlled by ALU unit, while the accout is zero, the aluout stays on the previous state which might not be zero value. This operation might cause confusion to programmer. Code snippet is given below for example.
LD ACC #(00)
JZ #(0F)

im_hex.txt

```
D5// LD ACC #(05);
51// MOV R1 ACC;
D4// LD ACC #(04);
21// SUB ACC R1
5B// MOV R11 ACC, R11 = 8'hFF
11// ADD ACC R1;
B0// SHL ACC
31// NOR ACC R1;
C0// SHR ACC
41// MOV ACC R1
00// NOP ------------------------------
D3// LD ACC #(03)
1B// ADD ACC R11 --Loop1
7F// JZ #(), forward 2 adr
AC// JC #(), backward 2 adr
```

```
21// SUB ACC R1 --Loop2
AF// JC #(), backward 1 adr
00// NOP ------------------------------
D3// LD ACC #(03), R16=8'h21
52// MOV R2 ACC
D8// LD ACC #(), R16 upper bits
B0// SHL
B0// SHL
5F// MOV R16 ACC
D1// LD ACC #(), R16 lower bits
1F// ADD ACC R16
5F// MOV R16 ACC, R16 stored
22// SUB ACC R2
5E// MOV R15 ACC
D3// LD ACC #(03)
1B// ADD ACC R11 --Loop11
6F// JZ R16, forward 2 adr
8E// JC R15, backward 2 adr
21// SUB ACC R1 --Loop22
8F// JC R16, backward 1 adr
00// NOP ------------------------------
F2// HALT
# output_hex.txt

05 // ACC = 01
05 // MOV R1 ACC, R1 = 8'h05
04 // ACC = 00
FF // ACC - R1
FF // MOV R11 ACC
04 // ACC + R1 = FF+05
08 // ACC << 1
F2 // ACC NOR R1
79 // ACC >> 1
05 // MOV ACC R1
05 // NOP ------------------------------
03 // LD ACC #(03)
02 // ADD ACC R1(FF) --loop1 start
02 // JZ0
02 // JC1
01 // ADD ACC R1(FF)
01 // JZ0
01 // JC1
00 // ADD ACC R1(FF)
00 // JZ1 --loop1 end
FB // ACC - R1 = 0-5 --loop2 start
FB // JC1
F6 // ACC - R1
F6 // JC0 -- loop2 end
```

```
F6 // NOP ------------------------------
03 // ACC = 03
03 // MOV R2 ACC, R2 = 8'h03
08 // ACC #(), R16 upper bits
10 // ACC << 1
20 // ACC << 1
20 // MOV R16 ACC
01 // ACC #(), R16 lower bits
21 // ADD ACC R16
21 // MOV R16 ACC, R16 = 20
1E // ACC - R2
1E // MOV R15 ACC
03 // LD ACC #(03)
02 // ADD ACC R1(FF) --loop11 start
02 // JZ0
02 // JC1 R15
01 // ADD ACC R1(FF)
01 // JZ0
01 // JC1
00 // ADD ACC R1(FF)
00 // JZ1 R16 --loop11 end
FB // ACC - R1 = 0-5 --loop22 start
FB // JC1
F6 // ACC - R1
F6 // JC0 -- loop22 end
F6 // NOP ------------------------------
F6 // HALT
```

Note:
I found register_file.v initialize only 15 registers to 8'h0, for register addressed in 8'hff it is uninitialized as signal x. The for loop only iterates 15 times from address 0 to 14 according to the condition (i<15). The index should be 16 instead of 15.

I added loadalu signal to the design in addition to the diagram given. Loadalu signal is for controlling alu unit maintain a consistent output even if the input data is changed. The zout and cout signal depends on the output of alu unit. If the current instruction is not logic operation but the jumping or loading operations, the aluout should be consistent so that the conditional jumping can work properly.