CZ/

CE3001: ADVANCED COMPUTER ARCHITECTURE

Lab 4 Interim Report

TEAM: Dr.Teeth	DATE: [March 24, 2014]

Name	**Matric No**
HONG XIAOHUI	U1220523H
JIANG KEWEI	U1220848F
LONG HONGQUAN	U1220764A
TAN LI HAU	U1222847D

SCHOOL OF COMPUTER ENGINEERING
NANYANG TECHNOLOGICAL UNIVERSITY

Contents

1. Brief Description

1.1. Purpose

The project's purpose was to design and simulate a MIPS(Microprocessor without Interlocked Pipeline Stages) pipelined 16-bit CPU using the verilog. The objectives of the project consisted of furthering our understanding of pipelining and processor design and to further understand the MIPS instruction set.

This Project contains two phases. In phase 1 of the project, we will be able to handle arithmetic instructions (ADD, SUB, AND, OR, SLL, SRL, SRA,and RL), memory instructions (LW and SW), load immediate instructions (LHB and LLB), control instructions (B, JAL, JR, and EXEC). There are two memories to access which are instruction memory and data memory.

In our phase 1 design, we do not consider the fact of pipeline hazard. All the current design and result so far is based on Ideal Pipeline. Further improvements will be realized in Phase 2.

1.2. Design & Feature

1.2.1. Pipeline

We use a 5-stage pipeline to handle the insturctions, i.e. IF, ID, EX, MEM, WB. The registers in between each stage store information from the previous stage and deliver useful information to the next stage.

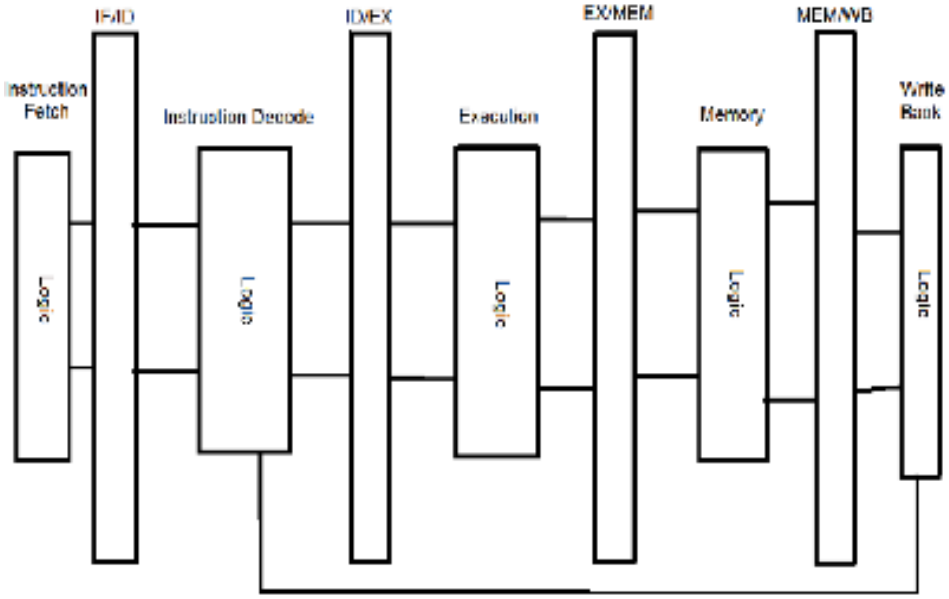

Figure 1

1.2.2. Control Unit

Control Unit controls signal input to each CPU component and multiplexers which selects connection.
For branch instruction, CU decides whether to branch based on the flag signal the last instruction sent to it. The flag signal is received from ALU out put on EXECUTE stage.

1.2.3. Single ALU in execute stage

For all the instructions including arithmetic, LW, SW, LHB, LLB, branch and jump, the calculations are all done be one single ALU. No further adder is implemented here. The ALU is in the Ex stage. This reduces CPU components and time cost. Thus the design will improve CPU efficiency.

The two ALU inputs are controlled by multiplexes, one 2-1 mux and one 4-1 mux. The implementation is shown as below in Figure 2. A table describe the control signals of source selections is also listed in Table 1.

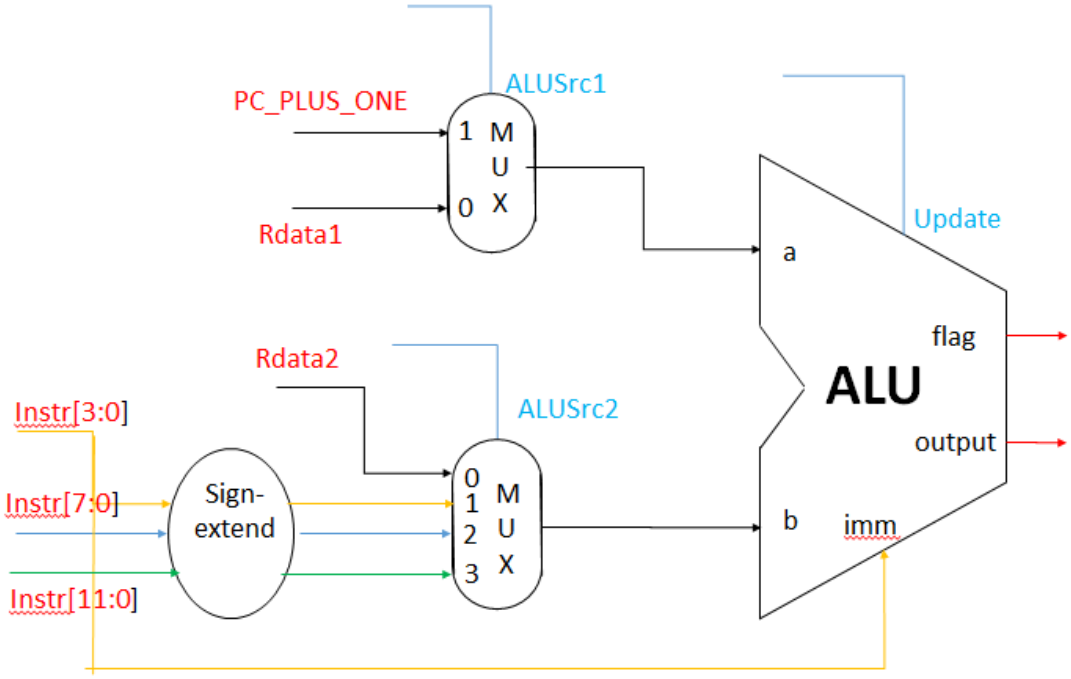

Figure 2

OPCODE	ALUSrc1	ALUSrc2
R-TYPE	0	0
LW	0	1
SW	0	1
LHB	0	2
LLB	0	2
B	1	2
JAL	1	3
JR/EXE	0	0

Table 1

2. Methodology

2.1. *Design Methodology*

The design of the project was divided into 6 steps, and each step was completed before going on to the next step. The testing steps are conducted into the following testing methodology in the order:

1). Create modules that are the components of the pipelined CPU
2). Test individual modules for functionality
3). Piece all of the individual modules together
4). Test complete CPU with the instruction program written by ourselves
5). Add support and test for one instruction at a time
6). Create program to demonstrate functionality of CPU

Our pipeline implementation of the CPU was subdivided into modules which were created before being pieced together. Before this project, we had created many basic modules like ALU and Register File in the previous labs which were used in the single cycle CPU. Because of this reason, we decided to start from the basic single cycle design and then create the control unit for the datapath in the pipeline registers. Each specialized pipeline register had to be created and created the datapath in the control unit to deal with the instructions in these different pipeline stages. After that, test values were then put through the modules by the memory to test their functionality.

After all of the modules were completed, we created the Phase1_toplevel for the CPU module which is used to join all of the individual modules together with logic part. This is the most difficult step because we need to add a large number of extra wires and registers in bits widths. The external clock which the test bench provided had to control these separate modules at the correct time correctly in the hardware and verilog. In the CPU, all of pipeline registers were clocked so that they would write on the positive edge of the clock and the register file would write on the both positive and negative edge. The final datapath diagram is shown below.

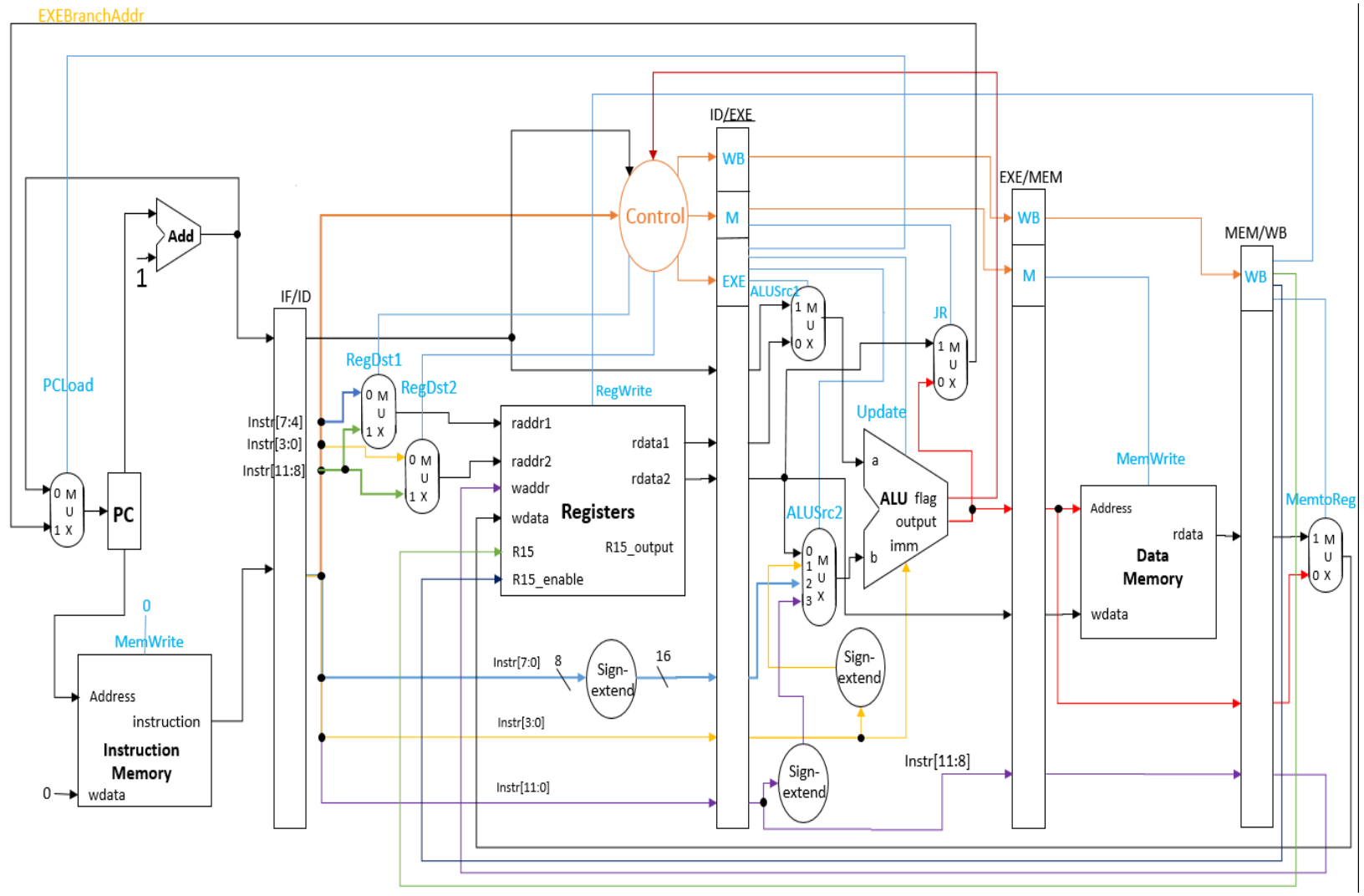

Figure 3

Once all the modules were packaged together and debugged already, we ran the instruction program which were written by ourselves as a test bench to debug the CPU when it is working.

2.2. *Testing Methodology*

We used a test bench that were written by ourselves to test our design. The test bench contains all of the instructions such as ADD, SUB, AND, OR, SLL, SRL, SRA, RL ,LW SW ,LHB LLB, B, JAL and JR which are used to test our multi-cycle implementations for this project.

The test bench loaded some values from the memory and used them to test all of our instructions. Some of these instructions were dependent on previous instructions, but some weren't. Each instruction was stored into different registers so that we could just run the whole program and look at the end result to decide if we ended up with the correct values.

3. Current status of project

3.1. Currently Working part

✓ We are able to implement 5-stage pipeline and it runs almost perfectly under ideal pipeline.
✓ Instructions ADD, SUB, AND, OR, SLL, SRL, SRA, RL ,LW SW ,LHB LLB, B, JAL and JR can be compiled and run successfully.
✓ We can access and write to the memory, register, pc (for branch).
✓ The control unit is fully functional currently.
✓ Regardless of data dependency, we are free to write any test bench instructions by ourselves and it seems to work fine under ideal pipeline.

3.2. Work ahead

✓ Instruction EXEC needs debugging.
✓ Code in memory.v has been edited for one single line. If we use the original version, which uses addr_r to access the memory, the **memory stage** will be delayed by one cycle.
✓ Now we change the memory.v file for one single word (use mem[addr] to access memory instead of using mem[addr_r]). This is to test our entire design and other components. We will consider to solve this problem by not changing the giving file in the future.
✓ Hazard Detection need to be implemented. They include data hazard, control hazards(phase 1) and structural hazard (phase 2). We are going to build a Hazard Control Unit or integrete it within our existing units. Stall signal or NOP will be added into our design.
✓ We are also considering to implement a Data Forwarding Unit in Phase 2 after hazard detection to improve our performance by reduce the cycle per instruction by forwarding data in pipeline.

4. Simulation and testing

4.1. Test case

Below is the test case we designed, such that there is no data dependency between each intstruction, so that there will be no hazards, since our Hazard Detection module is yet to be implemented.

4.1.1 . Intsruction

PC Address (hex)	Instruction	Machine Code (hex)
0000	LW R4, mem[R0+4]	8404
0001	LW R5, mem[R0+5]	8505
0002	LW R1, mem[R0+0]	8100

0003	LW	R2, mem[R0+1]	8201
0004	LW	R6, mem[R0+4]	8604
0005	LW	R7, mem[R0+1]	8701
0006	LW	R8, mem[R0+4]	8804
0007	LW	R9, mem[R0+4]	8904
0008	LW	RA, mem[R0+4]	8A04
0009	ADD	R3, R1, R2	0312
000A	LHB	RB, 0x06	AB06
000B	AND	R3, R6, R4	2364
000C	OR	R3, R6, R4	3364
000D	SLL	R5, R2, 2	4522
000E	LLB	RB, 0x03	BB03
*000F	SUB	R4, R4, R7	1447
0010	SLL	R7, R7, 1	4771
0011	ADD	R3, R1, R2	0312
0012	SW	R2, mem[R0+2]	9202
0013	SW	R3, mem[R0+3]	9303
0014	SUB	R4, R8, R7	1487

0015	B GT, (PC+1)+(-7)	C2F9

Table 2

4.1.2 Initial Memory

Address	Value (hex)
0000	0001
0001	0002
0002	0003
0003	0004
0004	000a
0005	0001
0006 - ffff	0000

Table 3

4.1.3. Description

Instruction 0000 - 0008
Load data from memory into register.
Instruction 0009 - 000F
Test of ALU operations: ADD, LHB, AND, OR, SLL, LLB, SUB
Instruction 0010, 0014, 0015
Instruction 0010, 0014, and 0015 implement a loop:
R7 multiplies by 2 for each loop and compares with R8 until R8 is greater than R7.
Instruction 0011, 0012, 0013
Random instruction that prevents data dependencies on R7 between instruction 0010 and 0014, as instruction 0010 and 0014 form a Read After Write (RAW) dependency.

4.2. Simulation results

We can note from the simulation result that the output is as expected. In our simulation process, we used the IDEAL PIPELINE instructions which contains no data dependency. All possible operations except for EXEC is tested and all output satisfied our expectation.

4.2.1. Instruction 0000 - 0008(Test for LW)

Figure 4 shows the simulation result for beginning (resetting) part of the CPU and the **SW** operations to initialize the registers.
We can see that the data from memory is loaded into the register. Each **STORE** is **5** cycles after the instruction is fetched from the instruction memory. In order, R1<-000A, R5<-0001, R1<-0001, R2<-0002, R6<-000A, R7<-000A, R8<-000A, R9<-000A, RA<-000A.

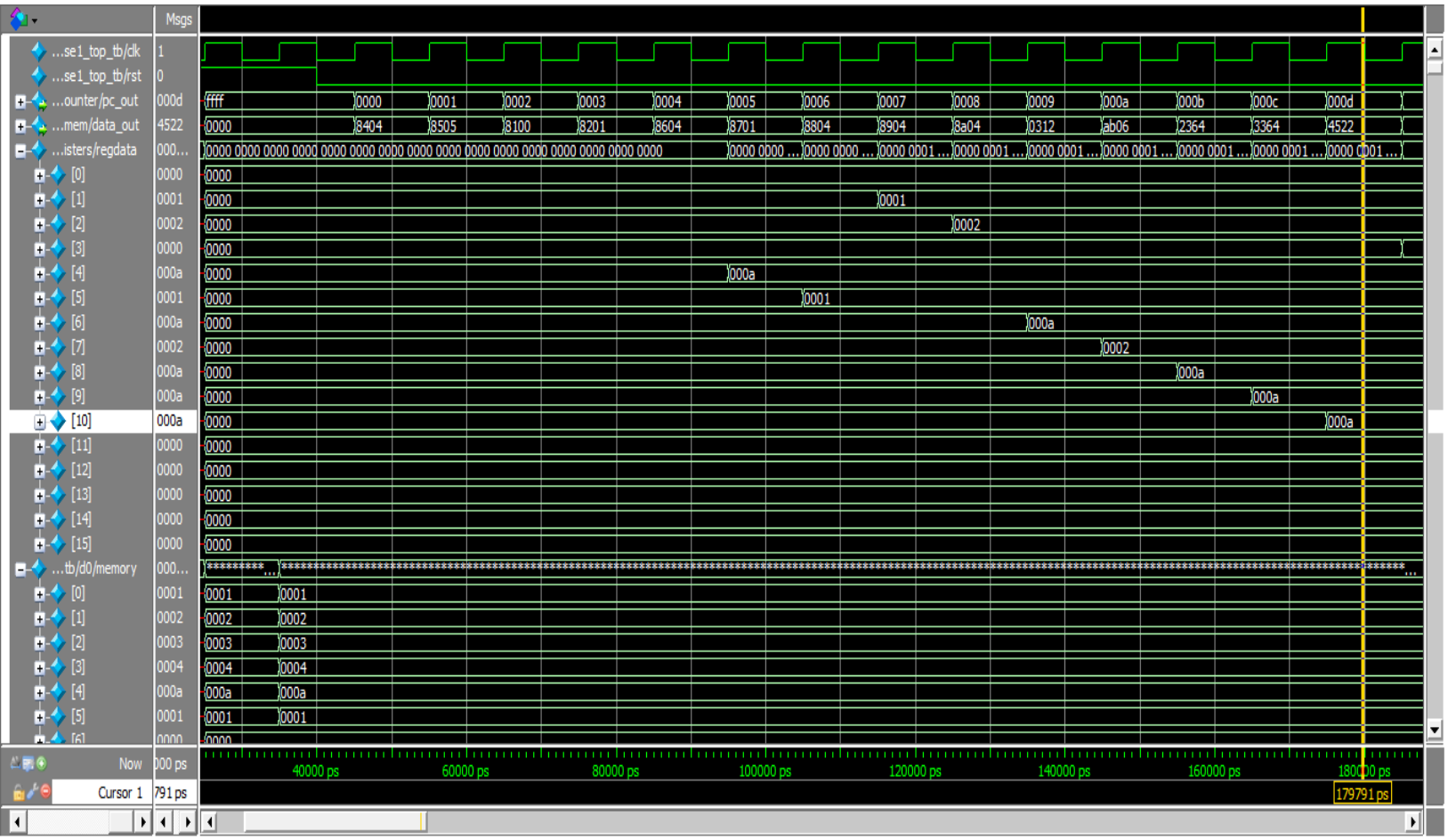

Figure 4

4.2.2. Instruction 0009 - 000E(Test of ALU operations and LHB/LLB)

Figure 5 shows the simulation result test for ADD, LHB, AND, OR, SLL, LLB in sequence. Each result is write back to register exactly **5** cycles after the instruction is fetched from the instruction memory.

For example, the when **program counter** is at 000E, fetched instruction is shown as below

000E	LLB RB, 0x03	BB03

This instruction load the 8 bit of 0x03 into the lowest 8 bit of register B. As register B is 0600 before, the result is 0603, which satisfies our expectation. When write back, the current program counter is already 0013, which is exactly 5 cycles after fetch.

The result for this part are all correct, which support our expectation.

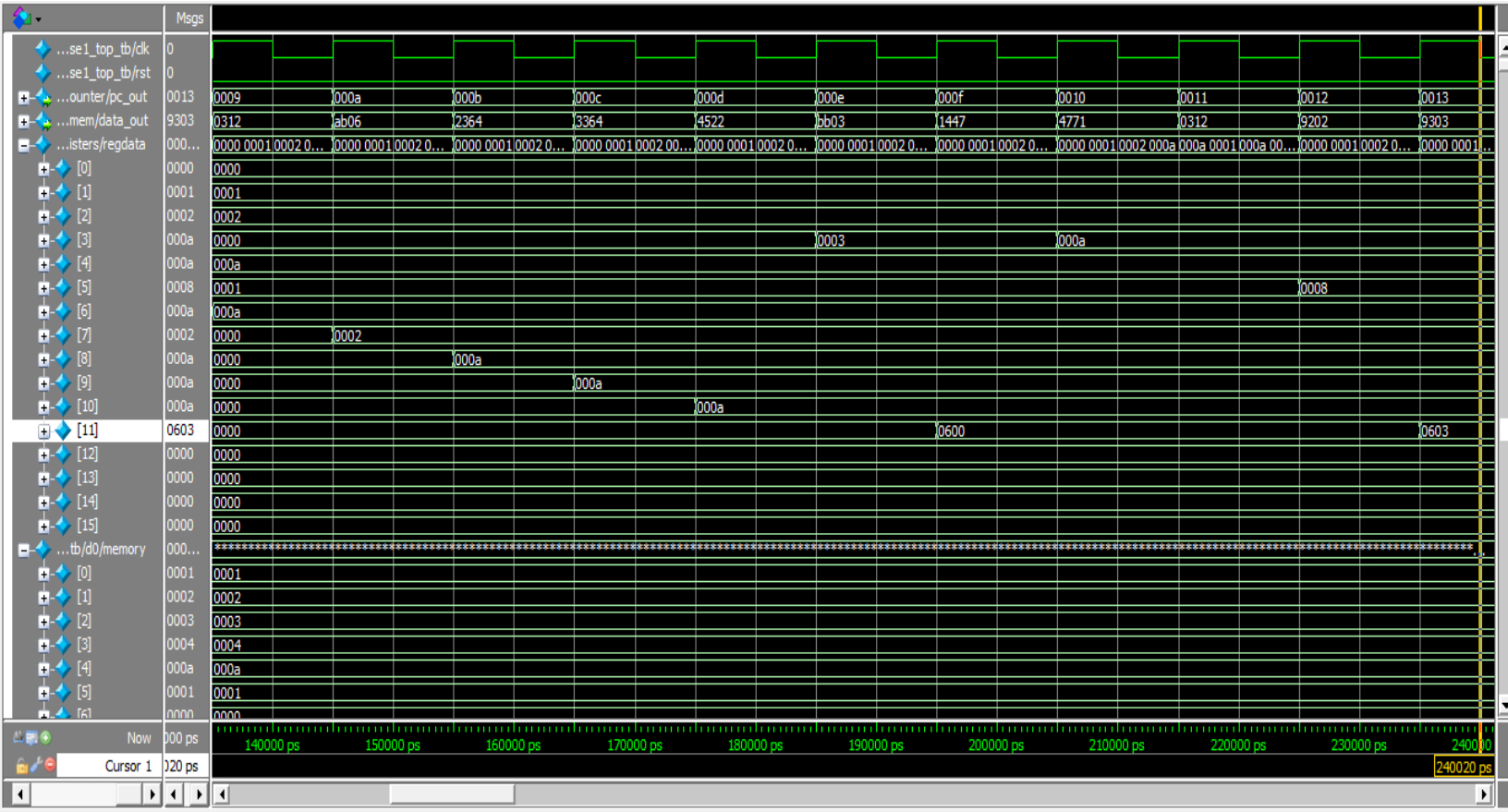

Figure 5

4.2.3. Instruction 0010, 0014, 0015(Test for branch)

Instruction 000F, 0010, 0014, and 0015 implement a loop:

0010	SLL R7, R7, 1	4771

...

0014	SUB R4, R8, R7	1487
0015	B GT, (PC+1)+(-7)	C2F9

Figure 6 show the simulation result for testing a successful branch. Here we compare the R8, which equals to 000A, with R7, which is initially equal to 2 and multiplied by two in every loop. If R8 is greater than R7, we jump back to 0015+1-7 = 000F.
Seen from the graph, the program counter jump back to 000F when at 0017. This is due to our special design -- if there is a jump instruction, we write back jump address and signal to **pc_mux** at **EXECUTE stage** where all data is ready**,** instead of having to wait

until **WRITE BACK stage**. That is why the jump instruction is performed only with a delay of **3** cycles.
As shown in the last part of the diagram, in **pc_mux** multiplex, **pc_load** is set to **1**and pc will load the branch address instead of increment by 1.

Figure 6

Figure 7 show the simulation result for testing an unsuccessful branch. According to

0010	SLL R7, R7, 1	4771

R7 is multiplied by 2 in every loop. After two loops, R7 will be 0x0010 which will be greater than R8, which is 0x000A. Logically speaking, the loop condition will be invalid so that there will be no branch.
Here we can see from the graph, when program counter reach **0x0016** this time, it is executing **SUB R4, R8, R7**, which set the flag which determine branch signal in **control unit**. As the output here is **FFFA**, the **flag[0]** is set to be **1**, which indicates **Negative**. According to the logic in **control unit**, branch signal will not be activated. Thus, there will be no loop this time.
As shown in the last part of the diagram, in **pc_mux** multiplex, **pc_load** is set to **0** and pc is incremented by 1. Which support our expectation.

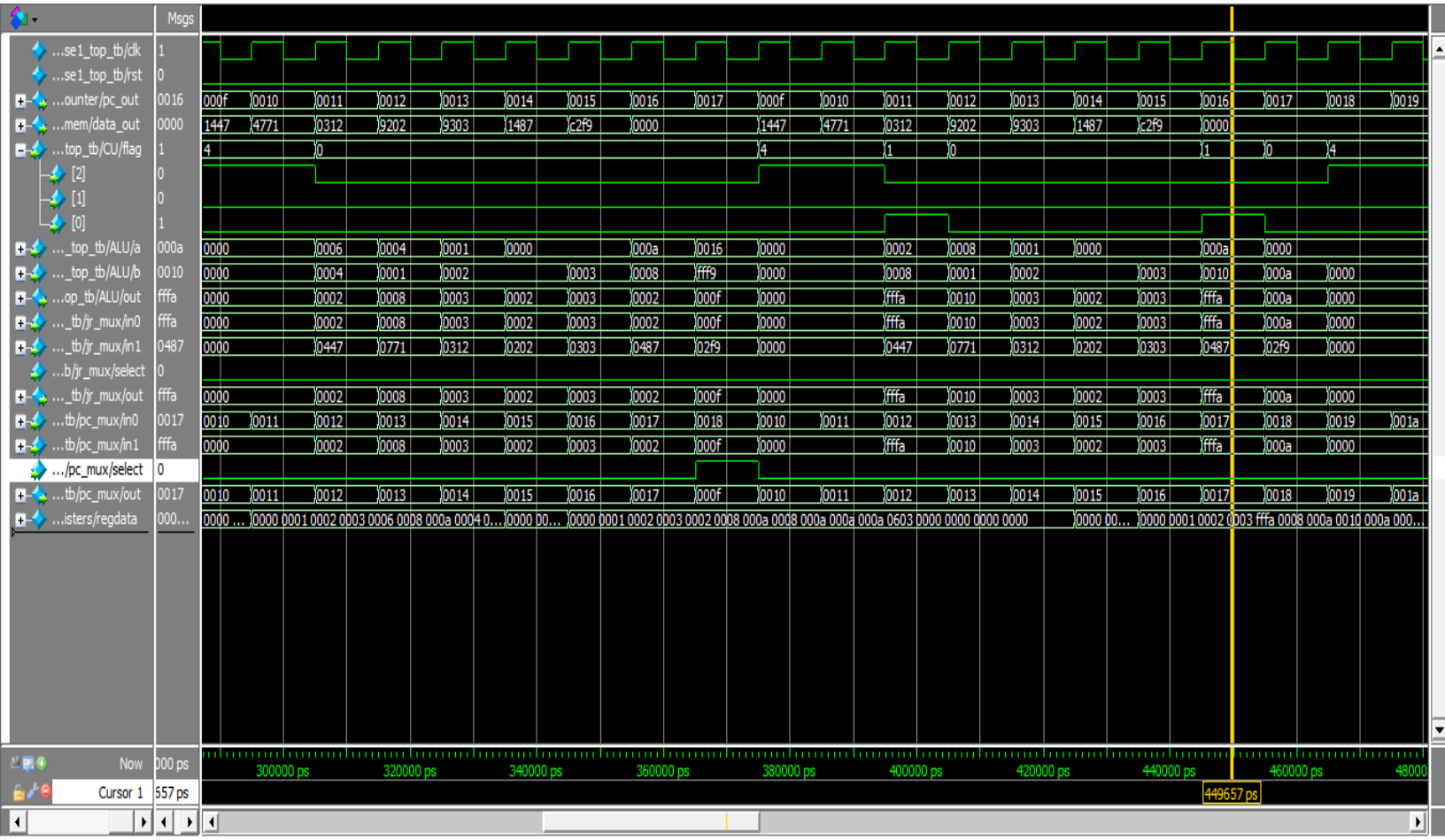

Figure 7

4.2.4. Instruction 0011, 0012, 0013 (Test of Memory Write, as well as Data Dependency Pitfall)

The instructions store the data in R2 into memory 2, data in R3 into memory 3. However, before the store instruction, there is a ADD R3, R1, R2 instruction. Thus, instruction 0011 and 0013 will form a Read After Write (RAW) dependency on register R3.
Initially, R2 = 0002, R3 = 000A.
The simulation test here focuses on testing for correct SW instruction, which is instruction 0012. Also, we want to test for the data dependency problem. Ideally, the out come should be mem[2] = 0002, mem[3] = 0003 (R2+R3). However, our expected result here would be mem[2] = 0002, mem[3] = 000A due to Read After Write (RAW) dependency.

0012	SW	R2, mem[R0+2]	9202
0013	SW	R3, mem[R0+3]	9303

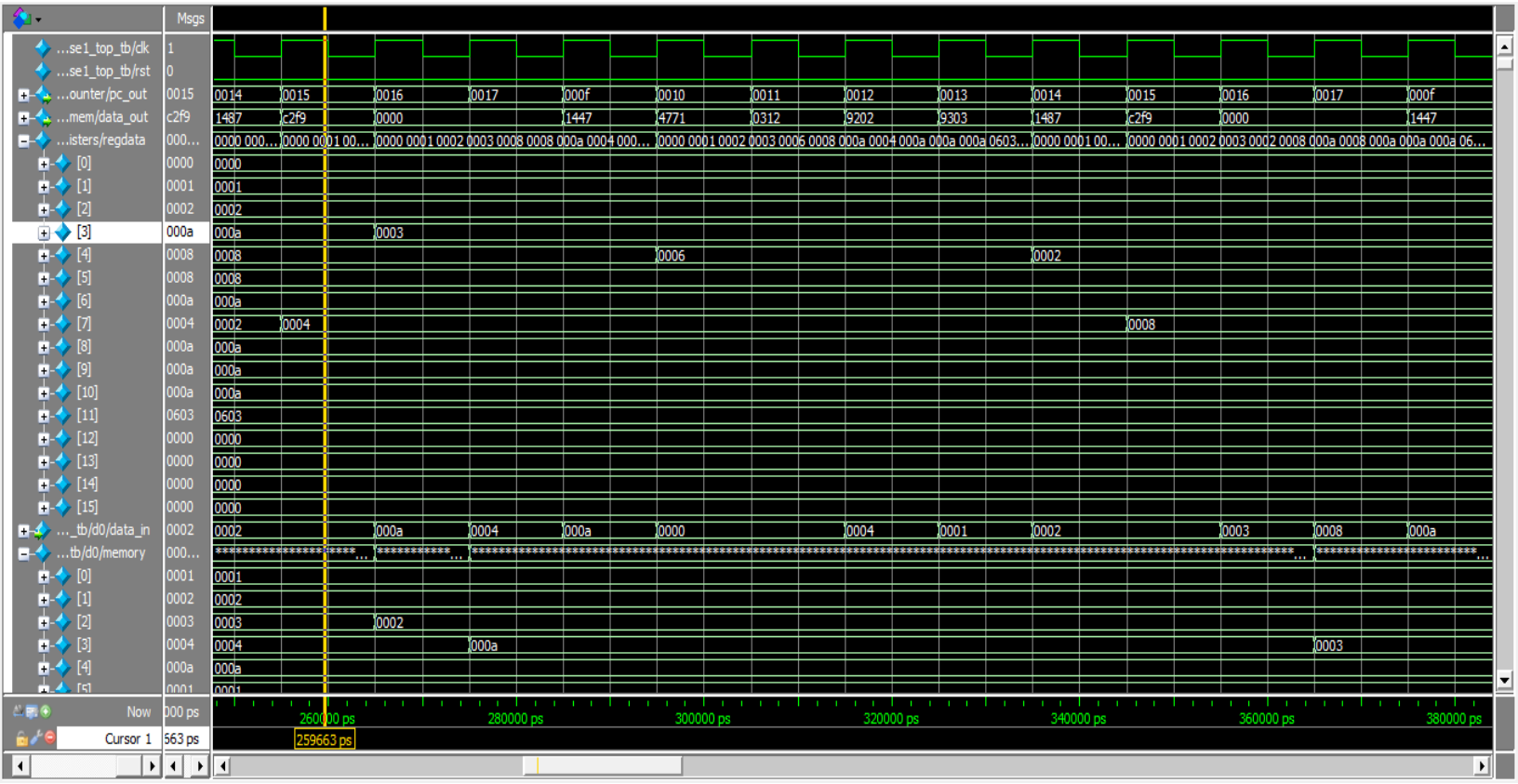

Figure 8

The figure 8 shows the simulation result for this part.
As shown in the last lines which represents the memory, mem[2] is correctly writen as 0002. However, mem[3] is writen to be 000A. If we look up at the registers in the same cycle, we can find that register 3 is now already changed into 0003. If we further look at the diagram in the second loop, the write data for mem[3] now is now 0003.
The simulation supports our expectation and shows the fatal flaw of our design -- data dependency.

5. Conclusion

Overall, the implementation of A basic CPU with 16-bit MIPS-like ISA written by our team in Verilog for this project is successful in this phase 1 where we just consider ideal pipeline.
This project helped us understand how a pipelined CPU works and indicated the importance of dividing the instructions into different register stages.
It is a very complex project, mostly because of these different modules with a large number of registers and wires. We need to keep track of a lot of different wires and the clock issues that lots of components need to be clocked. Furthermore, the debugging takes us lots of time and we still need to design the Hazard Detection unit and Forwarding unit in the Phase 2. Also, the instruction memory and data memory would be combined into a shared memory to store the address and data in the next phase.

6. Attachment 1 -- Instruction Memory.txt

```
// This file is used to initialize I_memory
//modified to test more operations
// Assembly
// LW   R4, mem[R0+4]
// LW   R5, mem[R0+5]
// LW   R1, mem[R0+0]                                                    // load 0
// LW   R2, mem[R0+1]                                                    // load 1
// Added Operation to remove data dependency
//// LW R6, mem[R0+4]
//// LW R7, mem[R0+1]
//// LW R8, mem[R0+4]
//// LW R9, mem[R0+4]
//// LW RA, mem[R0+4]
// End Added Operation to remove data dependency
// ADD R3, R1, R2                                                        // additon

// Added Operation to remove data dependency
//// LHB RB, 0X06
//// AND R3, R6, R4
//// OR  R3, R6, R4
//// SLL R5, R2,2
//// LLB RB, 0X03
// LOOP:  //It will loop for 5 times
//// SUB R4, R4, R7
//// SLL R7, R7,1 //shift left one logically every time
// ADD R3, R1, R2                                                        // additon
// SW   R2, mem[R0+2]                                          // store the 2nd operand in mem[2]
// SW   R3, mem[R0+3]                                             // store the addition in mem[3]
// SUB R4, R8, R7 //R8 = 000A, R7 = 2*R7, where R7 is initially 1.
// B      GT, (PC+1)+(-7) //Go to SUB R4, R4, R7
//
// Machine code in 16-bit Hex
0000 8404
0001 8505
0002 8100
0003 8201
0004 8604
0005 8701
0006 8804
0007 8904
0008 8A04
0009 0312
000A AB06
000B 2364
000C 3364
000D 4522
000E BB03
000F 1447
0010 4771
0011 0312
0012 9202
0013 9303
0014 1487
0015 C2F9
```

7. Attachment 2 -- Modified memory.v

```
`include "define.v"

module memory( clk, rst, wen, addr, data_in, fileid, data_out);

parameter ASIZE=16;
parameter DSIZE=16;

  input clk;
  input rst;
  input wen;
  input [ASIZE-1:0] addr;      // address input
  input [DSIZE-1:0] data_in;          // data input
  input [3:0] fileid;
  output [DSIZE-1:0] data_out;    // data output

  reg [DSIZE-1:0] memory [0:2**ASIZE-1];
  reg [8*`MAX_LINE_LENGTH:0] line; /* Line of text read from file */

integer fin, i, c, r;
reg [ASIZE-1:0] t_addr;
reg [DSIZE-1:0] t_data;

reg [ASIZE-1:0] addr_r;

//assign data_out = memory[addr_r];
assign data_out = memory[addr];

  always @(posedge clk)
    begin
      if(rst)
        begin
              addr_r <=0;
              case(fileid)
                  0: fin = $fopen("imem_test0.txt","r");
            1: fin = $fopen("imem_test1.txt","r");
                  2: fin = $fopen("imem_test2.txt","r");
            3: fin = $fopen("imem_test3.txt","r");
                  4: fin = $fopen("imem_test4.txt","r");
            5: fin = $fopen("imem_test5.txt","r");
                  6: fin = $fopen("imem_test6.txt","r");
            7: fin = $fopen("imem_test7.txt","r");
                  8: fin = $fopen("dmem_test0.txt","r");
            9: fin = $fopen("dmem_test1.txt","r");
                  10: fin = $fopen("dmem_test2.txt","r");
            11: fin = $fopen("dmem_test3.txt","r");
                  12: fin = $fopen("dmem_test4.txt","r");
            13: fin = $fopen("dmem_test5.txt","r");
                  14: fin = $fopen("dmem_test6.txt","r");
                  15: fin = $fopen("dmem_test7.txt","r");
              endcase
              $write("Opening Fileid %d\n", fileid);
              //First, initialize everything to 0
              for (i = 0; i < 2 ** ASIZE; i = i + 1)
                  begin
```

```
            memory[i] = 16'h0000;
                end
        //Now read in the input file
          while(!$feof(fin)) begin
          c = $fgetc(fin);
          // check for comment
          if (c == "/" | c == "#" | c == "%")
             r = $fgets(line, fin);
          else
            begin
              // Push the character back to the file then read the next time
              r = $ungetc(c, fin);
              r = $fscanf(fin, "%h %h",t_addr, t_data);
              memory[t_addr]=t_data;
            end
        end
        $fclose(fin);
        end
    else
      begin
          addr_r <= addr;
       if (!wen)
        begin           // active-low write enable
          memory[addr] <= data_in;
        end
        end
  end

endmodule
```

7. Attachment 3 – Design Diagram