PIPELINED MIPS CPU

COMPUTER ARCHITECTURE FINAL PROJECT

Abstract

This project implements the theory of pipeline form textbook in a set of program. And try to simulate the circuit by Verilog and ModelSim.

Haoyu Zhang, Li-Wei Li

Contents

Introduction

Through using Verilog and ModelSim, this project implements pipelined MIPS CPU. The datapath (see Appendix 1) shows the design of the project including four pipelines, a register, and a memory which are explained in textbook p.325 figure 4.65. There are five stages which are separated by four pipelines. The five stages are instruction fetch (IF), Instruction decode (ID), execution (EX), memory (MEM), and write back (WB). In this design, one clock cycle is 20 ns; every signals are triggered when at positive edge.

The data of register and memory, and the set of instruction are read from the text file (see Appendix 2). The three text files (instr_mem.txt, reg.txt, data_mem.txt) are read by readmemh() in test bench, which is used to read data from a file and save data into memory, so we can see every lines as one byte.

To prove this program's correctness, I will show the simulation of four instructions, lw, beq, or, and j. These four instructions represent four different instruction formats. If this program gets correct results, I can assume that this program can correctly execute every instruction which are belong to one of these four formats.

Simulation Result

This section will prove the program is correct by analyzing instructions and data. There are four instructions (lw, beq, or, and j), which represent four different formats in MIPS.

1. lw

The first instruction is lw $s1, $t7, 0.

```
// lw$t7, $s1, 0
00
00
2F
8E
```

In theory, this instruction will go to register s1 and get its value. After shifting the value 0 bit, the program will enter data memory to fetch the data according to the shifting result. And put the result into register t7. After calculating by human brain, the program should get s1 (0000_0002), and then shift 0 bit (still is 0000_0002), next, enter the data memory and get the data (00 00 01 00), in final, sent back the data to register.

The following figure is the result of simulation. The red boxes show the result of five levels pipeline; the blue box shows the signal of MemRead, which is 1.

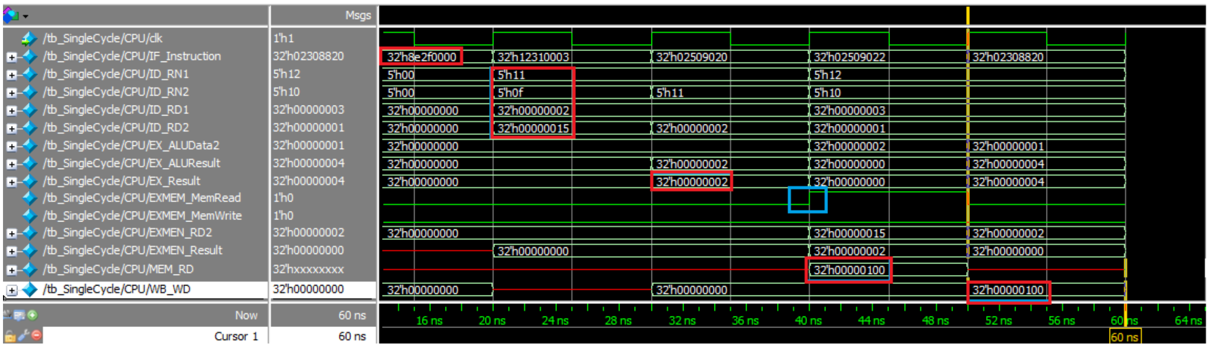


Figure 1. The red boxes show the result of five levels pipeline; the blue box shows the signal of MemRead, which is 1.

We can see that the content of the 32 bits wire - IF_Instruction – is 8e2f0000$_h$, which is the same as the first instruction. 8e2f0000$_h$ can be translated to 1000 1110 0010 1111 0000 0000 0000 0000 0000$_b$. According to the textbook, we can divide every bit into different fields (table 1).

Table 1. Load/store instruction format. Instruction format for load (opcode == 35_d). The register rs is the base register that is added to the 16-bit address field to form the memory address. Register rt is the destination register for the loaded value.[1]

$100011_b = 35_d$	10001	01111	0000000000000000
lw	rs = 17_d	rt = 15_d	address

In the second level, instruction decode, ID_RN1 is rs ($10001_b = 17_d$); ID_RN2 is rt ($01111_b = 15_d$). These two signals enter register memory and send two data (ID_RD1, ID_RD2) out. In figure1, ID_RD1 is 02_h; ID_RD2 is 15_h. To prove these two data is correct, we can check the reg.txt (see Appendix 2). The 17^{th} data (ID_RN1) in reg.txt is 0000_0002, and the 15^{th} data in reg.txt is 0000_0015.

In the third level (execution level), the result of ALU is 02_h. After filtering by the multiplexer, which is control by a shift wire which is used to determine whether result of the execution level is a shifted address or the data coming from register, the result of the third level is 02_h. According to what we learned from the class, for lw instruction, the result of the EX stage should come from the shift address, however in this case the shift bit is 0, therefore, the result is 02_h.

In memory part, due to this is a load instruction, it does not need to write data into the memory (Mem_Write == 0), but it should read data from the memory (Mem_Read == 1). The address we want to read from the memory is delivered by EXMEN_result, which is 0000_0002. In mem.txt, the data of 0000_0002 is 00 00 01 00_h. 0100h is as same as the result in figure 1.

In the last level, write back, the MemWB_MemtoReg wire controls the multiplexer to decide the write back data comes from the data memory or execution result. The job of a load instructions is loading the target data, whose address is determined by the instruction, from memory. What the program send back to register is 0100_h, which is as same as what we expected early.

2. beq

The next instruction is beq $s1, $s2, 3.

```
// beq $s1, $s1, 3
03
00
31
```

12

Through translating 03 00 31 12 to 32-bit instruction, we get 0001 0010 0011 0001 0000 0000 0000 0011$_b$ (12310003$_h$). Due to the opcode (opcode == 4$_d$), it is a beq instruction.

As table 2, it compares the same register (10001), so the next PC is the address which is shifted left two bits (1100$_b$), and add to PC + 4 (c$_h$+1100$_b$ = 24$_d$ = 18$_h$). Besides, the execution should only waste four clock cycles, because it does not need to complete WB. Once it completes the calculation of next PC, it sends the result back to IF.

Table 2. Branch equal format (opcode == 4$_d$). The register rs and rt are the source registers that are compared for equality. The 16-bit address field is sign-extended, shifted, and added to the PC+4 to compare the branch target address.

000100	10001	10001	0000000000000011
beq	rs	rt	address

Figure 2 shows the simulation of beq execution. Because this program is a pipelined CPU, when the last instruction (lw) is executed in instruction decode part, the current instruction is executed in the first part.

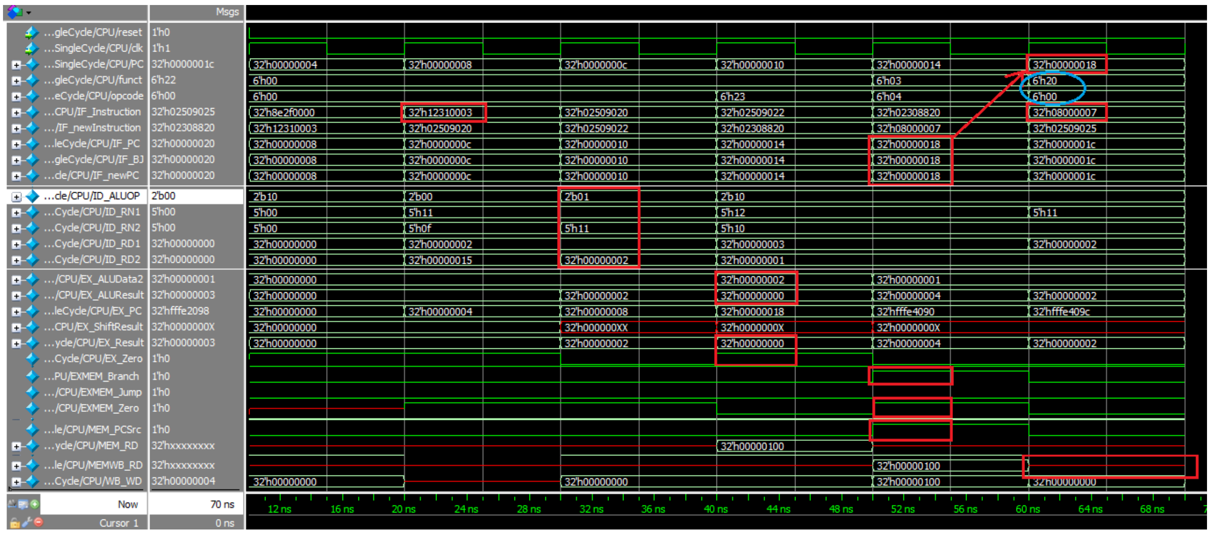


Figure 2. The simulation of beq instruction. In the fourth clock cycle, it sends the result to the next clock cycle. And in the next cycle, it starts to fetch the corresponding instruction. Lw instruction needs only four clock cycle to complete its job.

In the instruction fetch level, the program fetches the second instruction 12310003$_h$. The signal of ALU_Zero is 1. And the signal is sent to the third level, we can see that MEM_Branch, and MEM_Zero are 1; MEM_Jump is 0. According to the datapath (see

Appendix 1), the and gate generates signal 1 and then sends the signal to multiplexer. After altering by two multiplexers, 18_h is sent back to IF as the PC. The result is as same as what we expected before.

3. or

Current PC is 18_h, so the corresponding instruction is or $s2, $s0, $s2.

```
// or $s3, $s1, $t7
25
90
2F
02
```

02 4F 90 25_h = 0000 0010 0100 1111 1001 0000 0010 0101_b

Because of the opcode, this 32-bit instruction can be separated into 6 fields (table 3). According to the funct field, we know it is going to execute or instruction. The two source register is rs (18_d) and rt (16_d); the result of the operation will be stored in rd (18_d).

Table 3. R-type for or instruction

000000	10010	01111	10010	00000	100101
opcode	rs = 17_d	rt = 15_d	rd = 18_d	shamt	funct = 37_d

The data in register rs should be what we have read in the first instruction (lw $t7, $s1, 0), 0100h; the data in rt should be 0002h. After or these two data, the result is 0102h. Table 4 illustrate how to calculate 0002 or 0100.

Table 4. The operation of or two data.

0000	0000	0000	0010
0000	0001	0000	0000
0000	0001	0000	0010

According to the simulation (see figure 3), the two input data is 11_h and 02_h. And we get $0f_h$ and 0100_h from data memory. After calculating by ALU, the result is 0102h. in the fifth stage, the program writes 0102h back to the register s3.

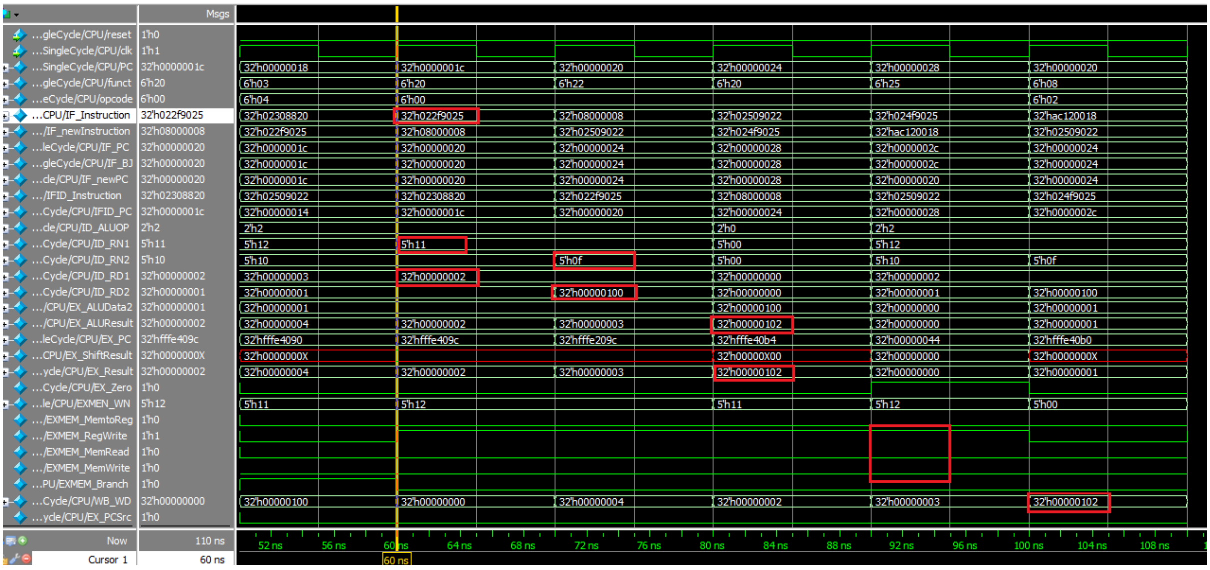


Figure 3. Simulation of or instruction.

4. j

Because of the last instruction (beq \$s1, \$s2, 3), the current PC is $1C_h$. The corresponding instruction is $08\ 00\ 00\ 07_h$.

```
// j   7
08
00
00
08
```

As the analysis in table 5, after translating 08 00 00 08h to binary type (0000 1000 0000 0000 0000 0000 0000 1000_b), the opcode is 2_d (000010_b), so this is a jump instruction; the other 26 bits is an address.

Table 5. Instruction format for the jump instruction.

000010	00 0000 0000 0000 0000 0000 1000
Opcode = 2 -> jump	address

The destination address for a jump instruction is formed by concatenating the upper 4 bits of the current PC+4 to the 26-bit address field in the jump instruction and adding 00 as the 2 low-order bits.[1] After concatenating the upper 4 bits of the current PC+4 to the 26-bit address and shifting left 2 bits, we get the result 0000 0000 0000 0000 0000 0000 0010 0000_b (==20_h).

As the result of the simulation (figure 4), in the beginning, the PC is 18h, which is delivered from the last instruction (beq \$s1, \$s2, 3), and the MSA is 08 00 00 08. In ID stage, the ID_Jump signal arise to 1. The program judges this instruction is a jump instruction, so we just need to focus on the address that the pipelined CPU generates.

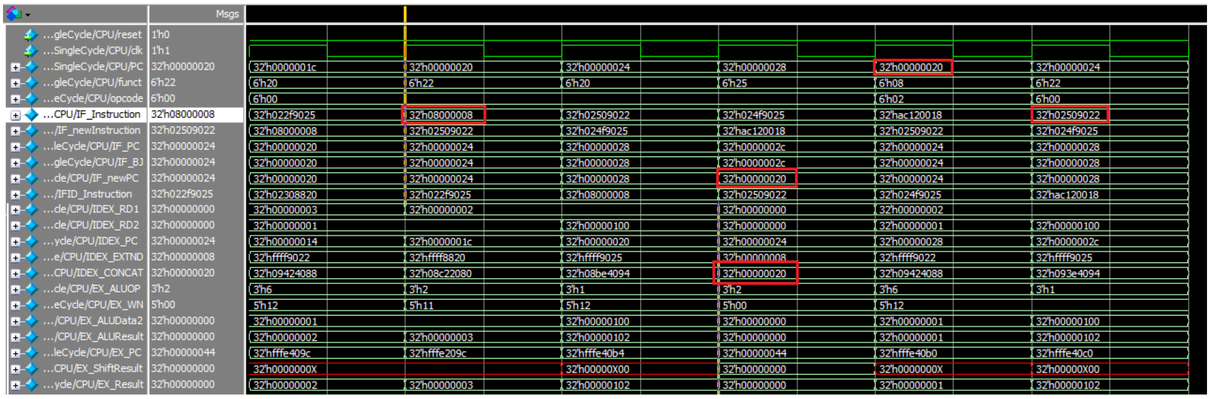


Figure 4. The simulation of jump instruction.

In the second clock cycle of jump instruction, the address, 20_h, is generated by the concatenating mechanism. After concatenating, 20_h is sent to the multiplexer. Because the jump flag is 1, the multiplexer releases 20_h to IF stage. In the figure 4, the next PC is 20_h, which is match what we anticipate.

Structure of the Program

The hierarchy of this program is illustrated as figure 5. There is a file named total_CPU, which controls how to operate the whole CPU, also represents the datapath which is attached in Appendix 1.

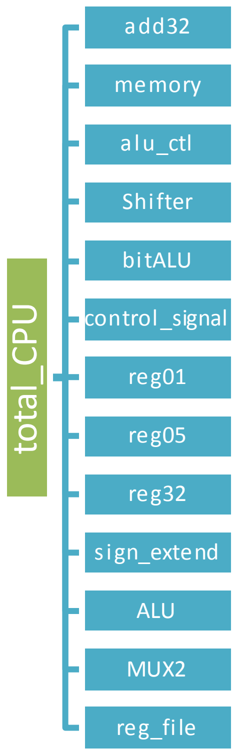


Figure 5. The structure of this program.

IF stage:

1. InstrMem. Fetch the corresponding instruction from the instruction memory.
2. Pcadd4. Add 4 to current PC as the next PC.
3. ifbramux. Decide the next PC come from branch address or PC+4.
4. ifjmpmux. Decide the next PC come from jump address or ifbramux.

ID stage:

1. Ifidpc. This is declared as a reg32, which implements pipeline.
2. ID_RN1. This is one of inputs, which send signal into RegFile. The value is 25^{th} to 21^{st} bits of current instruction.
3. ID_RN2. This is another input. The value is 20^{th} to 16^{th} bits of current instruction.

4. cu. This is a control unit, which depend on different instruction to send flags to corresponding unit. The flags it generates includes RegDst, Branch, MemtoReg, MemRead, ALUOp, MemWrite, ALUSrc, and RegWrite.
5. RegFile. RegFile can output appropriate registers which is supported by reg.txt.
6. tobig. Extend 16 bits to 32 bits.
7. ID_CONCAT. This concatenate the upper four bits of next PC to the current instruction (25 to 0 bit), and then concatenate two zero in the left most bit.

Ex stage:

1. exr01 … exr17. Those implement the IDEX pipeline.
2. totalAlu. ALUOp is sent to TotalAlu to control the operation of TotalAlu, which executes the main calculation of the datapath.
3. exaddpc. Use to calculate the branch address.
4. ac. According to the ALUOp, ac outputs ALUOp, which control totalALU, and IDEX_shift to control a mux.
5. sss. Input two data and shift the right most four bits, and output the result.
6. EX_PCSrc. Use logical and to process EX_PCSrc.

Mem stage:

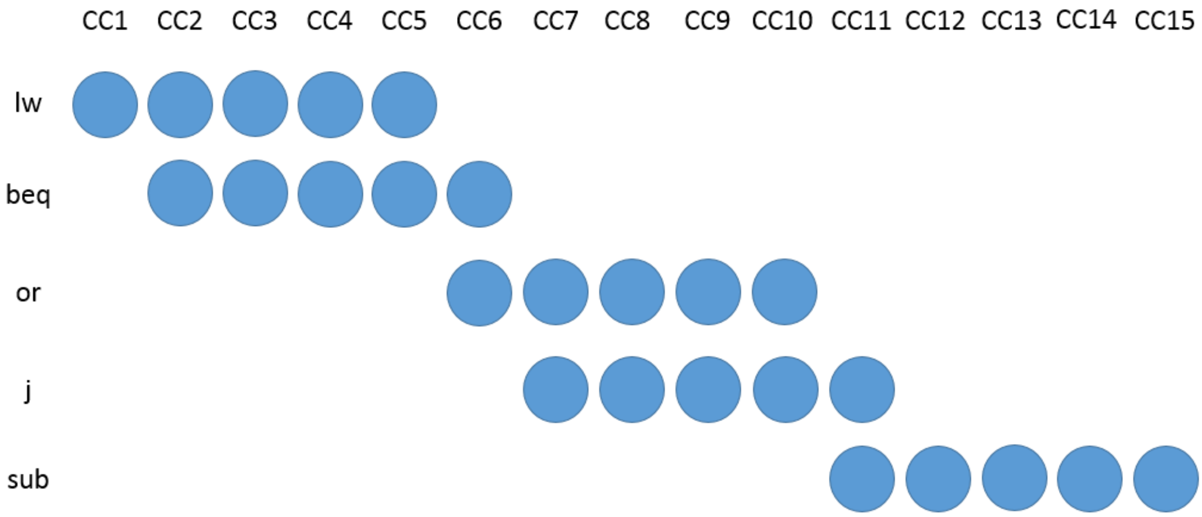
1. memr01 … memr14. Those registers implement the EXMEM pipeline.
2. DatMem. Input five signals, and depend on MemRead and MemWrite to decide the output data.

WB stage:

1. wbr1 … wbr2. Those registers implement the MEMWB pipeline.
2. wdmux. Use to decide what signal should be sent back to RegFile.

Conclusion

According to the simulation, this program is not efficient enough. Figure 5 shows the program's execution in every clock cycle. We can see that the IF stage occurs when beq instruction enter the fifth stage.



Appendix

1. Datapath

2. instr_mem.txt

```
// lw      $s1, $t7, 0
00
00
2F
8E
// beq     $s1, $s2, 3
03
00
31
12
// add     $s2, $s0, $s2
20
90
50
02
// sub     $s2, $s0, $s2
22
90
50
02
// add     $s1, $s0, $s1, 0
20
88
30
02
// j    7
07
00
00
08
// or  $s2, $s0, $s2
25
90
50
02
// add $s1, $s0, $s1, 0
```

```
20
88
30
02
// sub $s2, $s0, $s2
22
90
50
02
// or  $s2, $s0, $s2
25
90
50
02
// sw  $zero, $s2, 24
18
00
12
AC
```

3. reg.txt

```
// $at
0000_0001
// $v0-$v1
0000_0002
0000_0003
// $a0-$a3
0000_0004
0000_0005
0000_0006
0000_0007
// $t0-$t7
0000_0008
0000_0009
0000_0010
0000_0011
0000_0012
0000_0013
0000_0014
0000_0015
// $s0-$s7
0000_0001
0000_0002
0000_0003
0000_0004
0000_0005
0000_0006
0000_0007
0000_0008
// $t8-$t9
0000_0024
0000_0025
// $k0-$k1
0000_0026
0000_0027
// $gp
0000_0028
```

```
// $sp
0000_0029
// $fp
0000_0030
// $ra
0000_0031
```

4. data_mem.txt

```
00
00
00
01

00
00
00
02

00
00
00
03

00
00
00
04
```

Reference

[1] D. A. Patterson and J. L. Hennesy, “The Processor,” in *computer Organization and Design*, 5th ed, MA, 2014, p.262, p.270