Organization of Digital Computer Lab
EECS 112L

Lab 4
Yan Li
Date: 3/08/2025

Objective

The goal of this lab was to design and create a pipelined processor in Vivado (*Figure 1*).

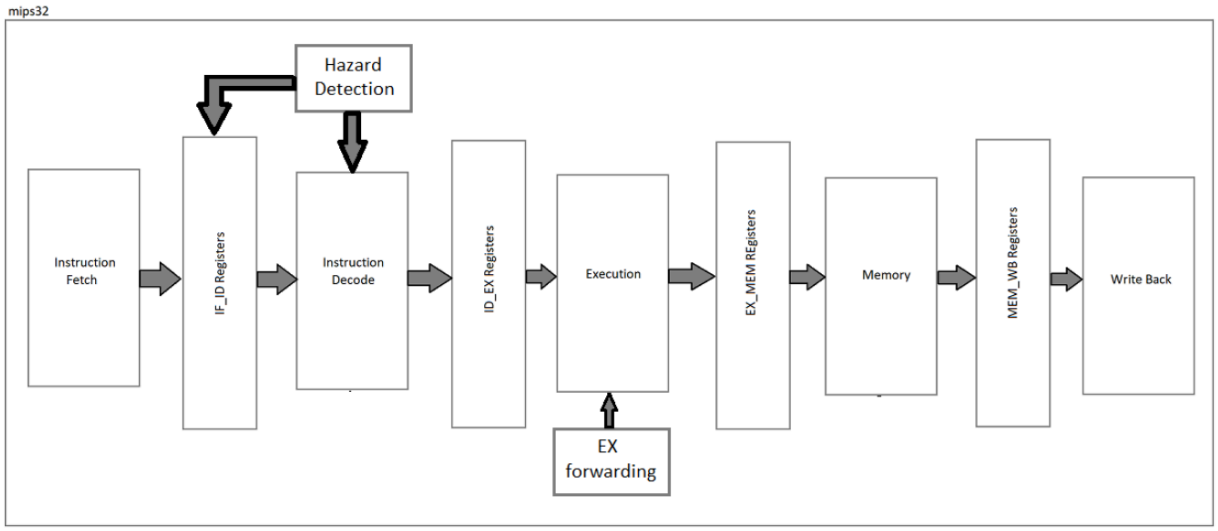


Figure 1. *Pipelined processor.*

The pipeline processor has a big advantage compared to a single-cycle processor, like the one designed in Lab 2. Pipelining is a technique where multiple instructions are overlapped in execution. The pipeline processor breaks down MIPS instructions into five steps:

- IF (Instruction Fetch)
- ID (Instruction Decode)
- EXE (Execution)
- MEM (Memory)
- WB (Write Back)

Procedure

Instruction Fetch

The first stage of a pipeline processor, instruction fetch, reads instructions from memory using the address in the PC. It also takes into consideration jump/branch (*Figure 2*).

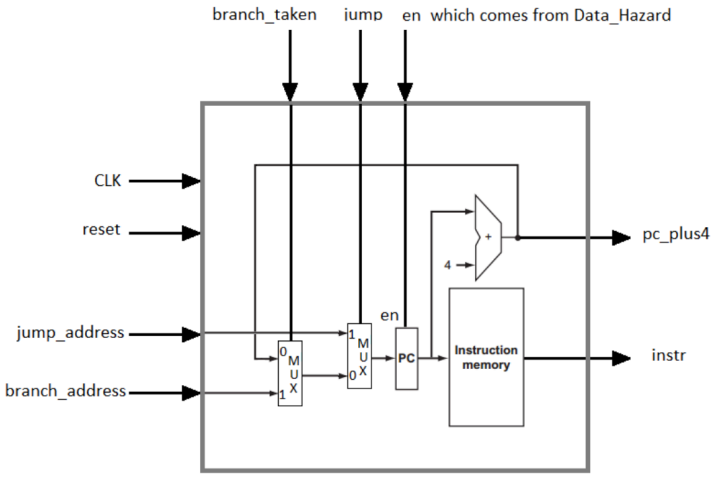


Figure 2. *Instruction fetch diagram.*

Instruction Decode

The next stage of a pipeline processor, instruction decode. (*Figure 3*).

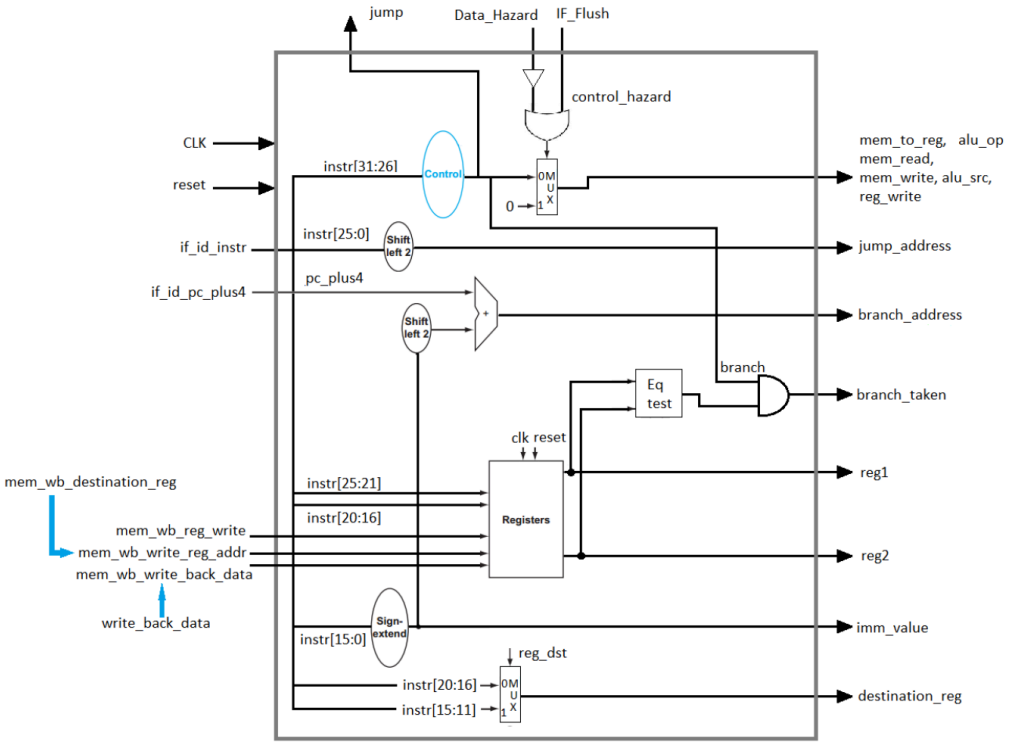


Figure 3. *Instruction decode diagram.*

Execution

The next stage of a pipeline processor, execution, uses the ALU to perform logical or arithmetic operations. Which operation is determined by the ALU Control that gets its inputs from the ID/EX Registers (*Figure 4*).

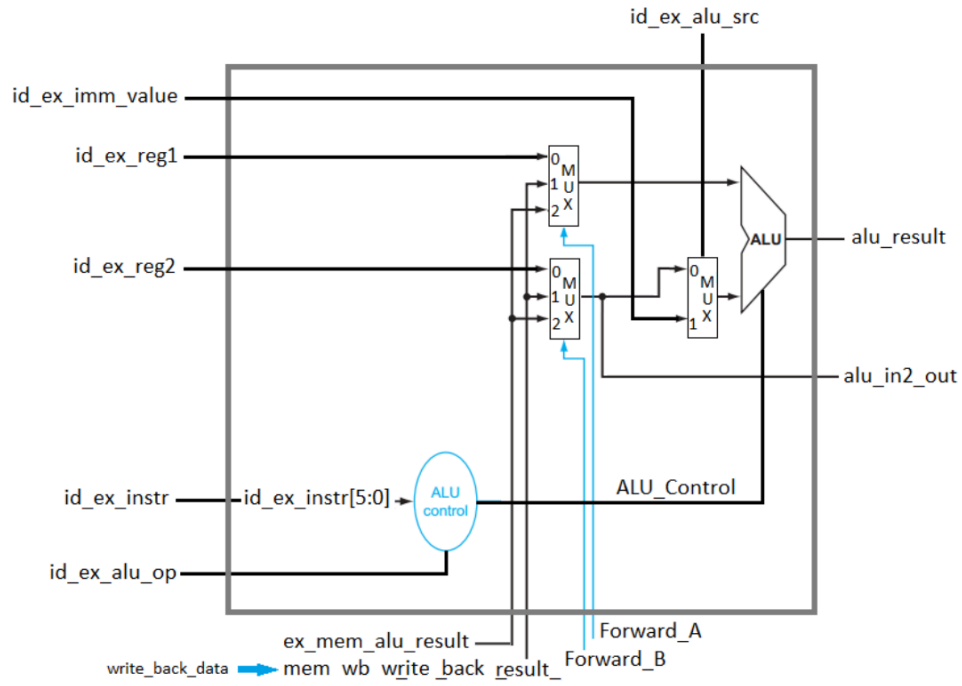


Figure 4. *Execution diagram.*

MIPS_32

The MIPS_32.v file connects together all of our individual modules. It includes all of the previous stages, necessary registers, memory, and write back.

Simulation Results

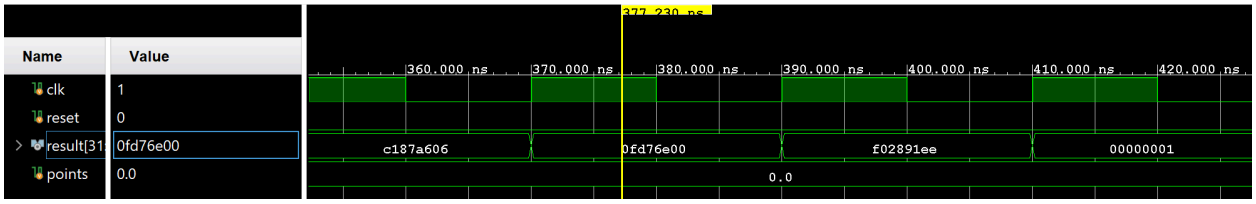
Using the included testbench and instruction memory, we get the following:

```
NO DEPENDENCY ANDI  success!

NO DEPENDENCY NOR   success!

NO DEPENDENCY SLT   success!

NO DEPENDENCY SLL   success!

NO DEPENDENCY SRL   success!

NO DEPENDENCY SRA   success!

NO DEPENDENCY XOR   success!

NO DEPENDENCY MULT  success!

NO DEPENDENCY DIV   success!

ANDI No Forwarding       success!

Forward EX/MEM to EX B   success!

Forward MEM/WB to EX A  success!

SLL  No Forwarding      success!

Forward EX/MEM to EX A  success!

Forward MEM/WB to EX A  success!

XOR  No Forwarding      success!

MULT No Forwarding      success!

Forward MEM/WB to EX B  success!

DATA HAZARD RS DEPENDENCY    success!

DATA HAZARD RT DEPENDENCY   success!

CONTROL HAZARD BRANCH   success!

CONTROL HAZARD JUMP success!
```

No dependency test, no jump

These are basic arithmetic operations, so there is not much to show of the waveform. The console report shown in the last page proves that all of these operations are performing as intended, but I will show a couple of examples as highlighted below.

```
// no dependency test, no jump
rom[10] = 32'b00110000011010111111111101100011; // andi r11,r3,#ff63
rom[11] = 32'b00000000001000100110000000100111; // nor  r12,r1,r2
rom[12] = 32'b00000000001000100110100000101010; // slt  r13,r1,r2
rom[13] = 32'b11000000010000000111000011000000; // sll  r14,r2,#3
rom[14] = 32'b11000000001000000111100101000010; // srl  r15,r1,#5
rom[15] = 32'b11000000110000001000000110000011; // sra  r16,r6,#6
rom[16] = 32'b00000000010000111000100000100110; // xor  r17,r2,r3
rom[17] = 32'b00000000001000101001000000011000; // mult r17,r1,r2
rom[18] = 32'b00000000010000011001100000011010; // div  r19,r2,r1
```



From this waveform, result[31:0] shows the result of our operations andi, nor, and slt. You can see that they match the intended results below.

```
// no dependency test, no jump
rom[10] = 32'b00110000011010111111111101100011; // andi r11,r3,#ff63
rom[11] = 32'b00000000001000100110000000100111; // nor  r12,r1,r2
rom[12] = 32'b00000000001000100110100000101010; // slt  r13,r1,r2
```

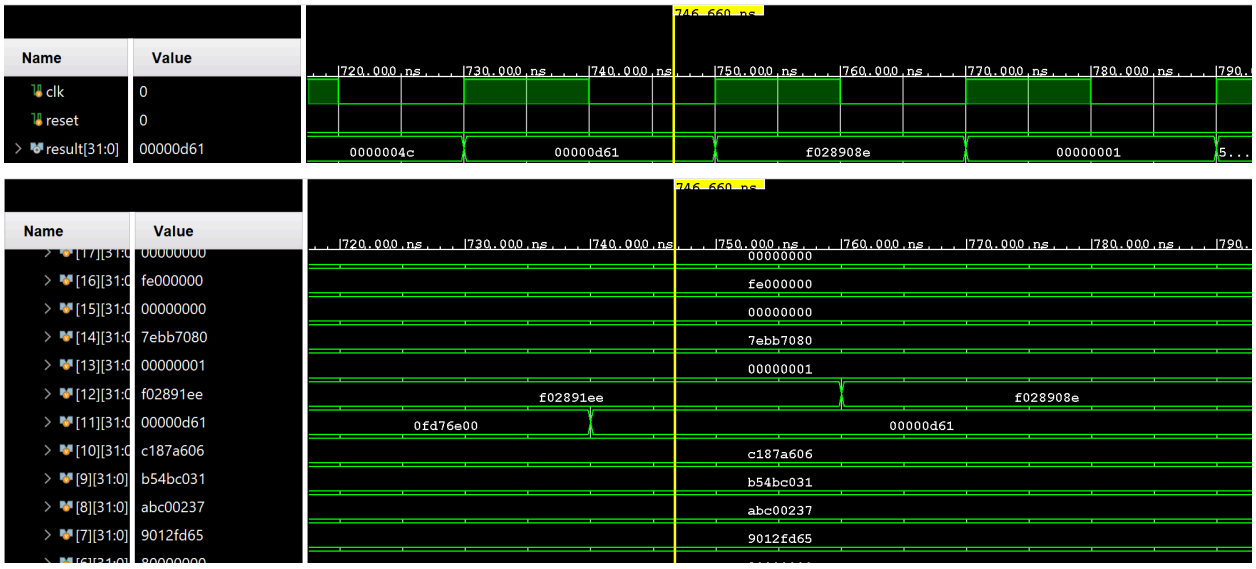
```
0fd76e00
f02891ee
1
```

Forwarding test

Forwarding reduces data hazards by passing information to the next instruction so that it does not have to wait for it to go through write back. Below, the testbench performs some forwarding tests.

```
// forwarding test
rom[28] = 32'b00110000111010110000111101100011; // andi r11,r7,#f63
rom[29] = 32'b00000000010010110110000000100111; // nor  r12,r2,r11
rom[30] = 32'b00000001011000100110100000101010; // slt  r13,r11,r2
rom[31] = 32'b11000000111000000111001101000000; // sll  r14,r2,#13
rom[32] = 32'b11000001110000000111100111000010; // srl  r15,r14,#7
rom[33] = 32'b11000001110000001000000010000011; // sra  r16,r14,#2
rom[34] = 32'b00000000010001111000100000100110; // xor  r17,r2,r7
rom[35] = 32'b00000000010001111001000000011000; // mult r18,r2,r7
rom[36] = 32'b00000000111100011001100000011010; // div  r19,r7,r17
```

The highlighted instructions show one example of forwarding. rom[28] wants to produce a value in r11, but the next instruction rom[29] also needs r11. In his case, forwarding is needed.



In this waveform, we can see that it performs correctly andi, saving the result in r11 = 00000d61. We know that forwarding occurred because r12 was updated on the very next cycle clock. The output also aligns with their intended results.

```
rom[28] = 32'b00110000111010110000111101100011; // andi r11,r7,#f63
rom[29] = 32'b00000000010010110110000000100111; // nor  r12,r2,r11
```

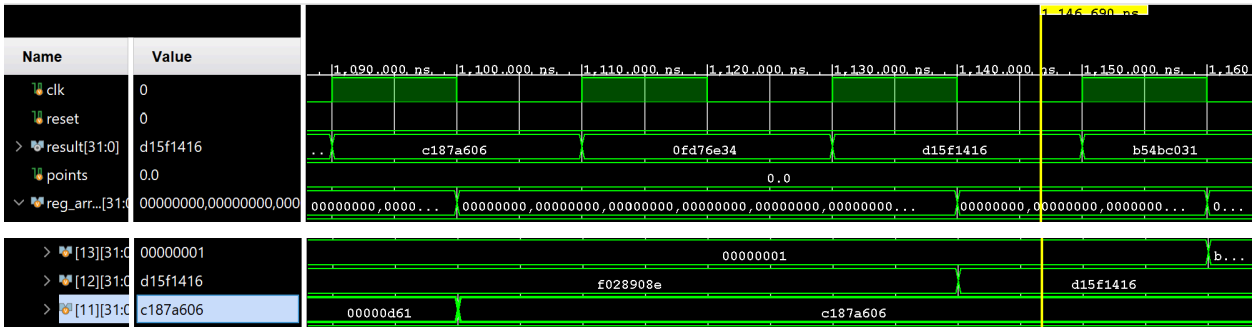
```
00000d61
f028908e
```

Data hazard test

In the data hazard tests in our testbench, the instructions will force a stall. This is because of a dependency between a LW instruction's destination register and one of the next instruction's source register.

```
rom[46] = 32'b10001100000010110000000000100100; // r11 = mem[36]
rom[47] = 32'b00000001011000100110000000100000; // add r12,r11,r2
rom[48] = 32'b10001100000011010000000000100000; // r13 = mem[32]
rom[49] = 32'b00000000001011010111000000100000; // add r14,r1,r13
```

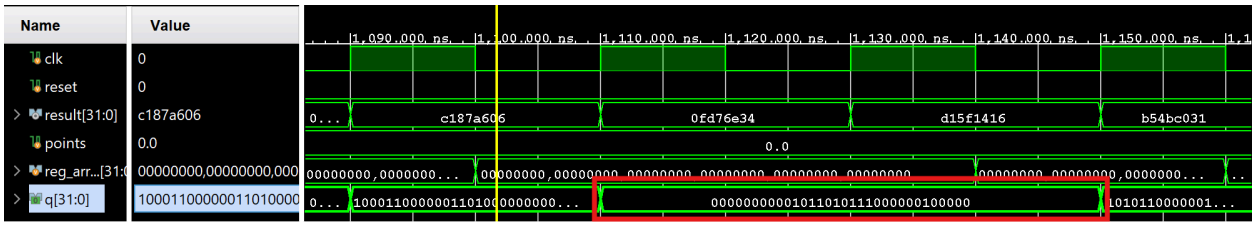
The highlighted above show two examples in our testbench where the processor needs a stall. As you can see, rom[47] requires r11 from the previous load instruction, and rom[49] has a dependency on its previous instruction.



Here, we can see the first example of stalling. rom[46] loads r11= c187a606, which we can see in the waveform result. However, we see that the next instruction is not performed on the next clock cycle, instead, it is the next NEXT clock cycle where rom[47] is performed. We can see on the registers that r11 was set, but the result of add r12, r11, r2 was not saved into register r12 until 2 clock cycles later.

```
rom[46] = 32'b10001100000010110000000000100100; // r11 = mem[36]          9              r11= c187a606
rom[47] = 32'b00000001011000100110000000100000; // add r12,r11,r2        d15f1416        r12= d15f1416
```

We see that the registers contain their intended results, but because rom[47] needed to stall, we see that it took 2 clock cycles to complete.

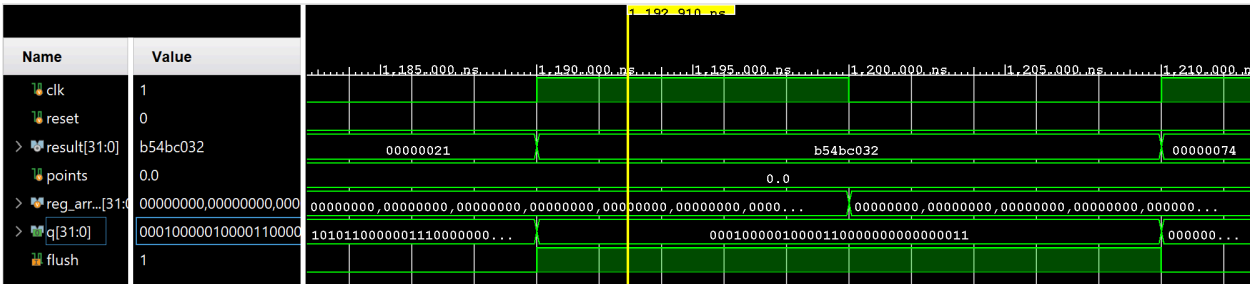


Here we see that it held the instruction rom[47] = 32’b00000001011000100110000000100000.

Control hazard test branch

In this test, it checks to see if the processor produces a flush when executing a branch instruction. The flush ensures that subsequent instructions are not executed when there is a branch.

```
rom[52] = 32'b00010000010000110000000000000011; // beq r2,r3,#3
rom[53] = 32'b00000000001000100101000000100000; // add r10,r1,r2
rom[54] = 32'b00000000001001000101000000100000; // add r10,r1,r4
rom[55] = 32'b00000000001001010101000000100000; // add r10,r1,r5
```



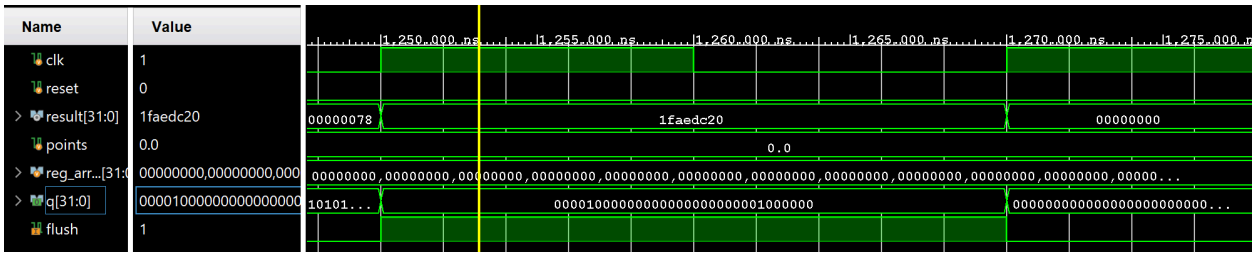
In this waveform, we see that at instruction rom[52], the processor produces a flush signal. It skips the subsequent instructions and branches to rom[56].

```
rom[56] = 32'b10101100000010100000000001111100; // sw mem[r0+31] <= r10          7c
```

Control hazard test jump

Similarly to the branch hazard, a flush will be produced when the processor reads a jump instruction. The flush ensures that subsequent instructions are not executed when there is a jump.

```
rom[57] = 32'b00001000000000000000000001000000; // j #40
rom[58] = 32'b00000000001000100100100000100000; // add r9,r1,r2
rom[59] = 32'b00000000001001000100100000100000; // add r9,r1,r4
rom[60] = 32'b00000000001001010100100000100000; // add r9,r1,r5
rom[61] = 32'b00000000001001100100100000100000; // add r9,r1,r6
rom[62] = 32'b00000000001001110100100000100000; // add r9,r1,r7
rom[63] = 32'b00000000001010000100100000100000; // add r9,r1,r8
```



In this waveform, we see that the processor produces a flush signal when it reads the rom[57] jump instruction.