Organization of Digital Computers Lab
EECS 112L

Lab 4: MIPS Pipeline Processor

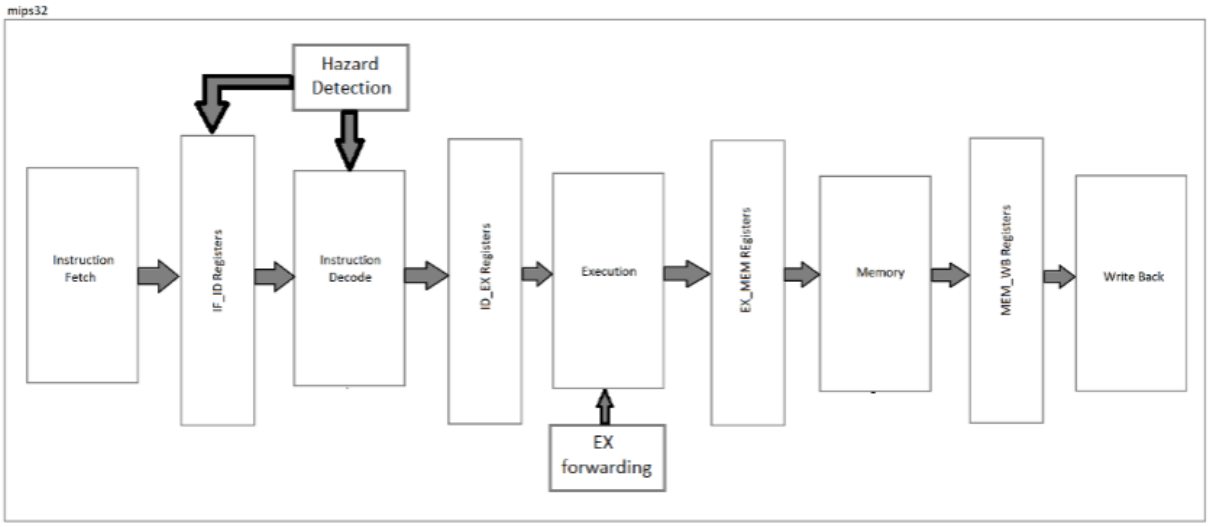
Kunal Paode

71402651

Mar 9, 2023

MIPS Pipeline Processor

1 Objective



The pipeline processor works by having 5 pipelining stages with registers in between each stage with the exception of the WB/IF transition. In order to effectively pipeline, the process also contains a forwarding unit which allows for subsequent arithmetic operations. The processor also includes hazard detection for data hazards such as when the processor is ordered to load from memory, then immediately do an arithmetic operation using the loaded register. In this situation, the processor stalls for a single cycle. The processor should be able to also handle control hazards, where the processor makes a change to its PC other than adding 4 to it.

2 Procedure

Many of the modules were already defined. Instruction fetch used the included mux instantiations along with a reg as a pc. The included instruction memory was used inside of it. The instruction decode stage again followed the schematic using logical assign statements and internal wires. The included register file instantiation was also used inside. The hazard detection module was made using and if-else statement inside of a always block. Similarly to the instruction fetch and instruction decode, in that it contains mux instantiations and internal wires, along with an ALU and ALU Control instantiations. The data memory module is simply instantiated. The writeback stage is represented by a single mux.

All of the modules described above then exist within the mips_32.v file, which contains plenty of wires hooking up each module. The transition registers for each stage are loaded into an internal wire, then run though the pipe_reg or pipe_reg_en module, then split up into individual wires later.

3 Simulation Results

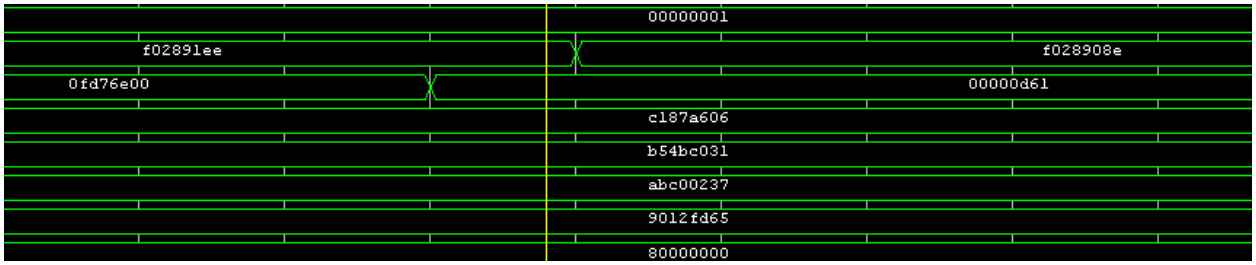
Passing of all tests:

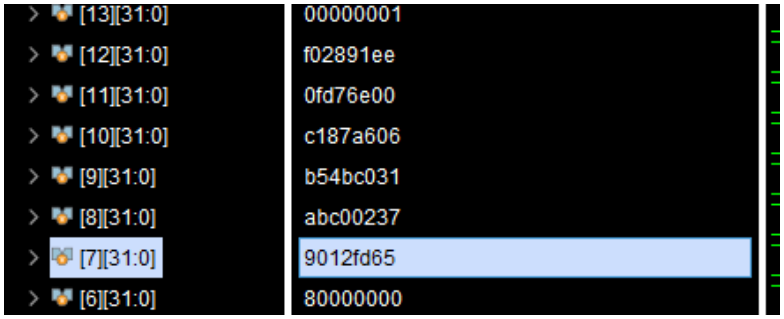
```
NO DEPENDENCY ANDI  success!

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!
```

Demonstrating Forwarding functionality:

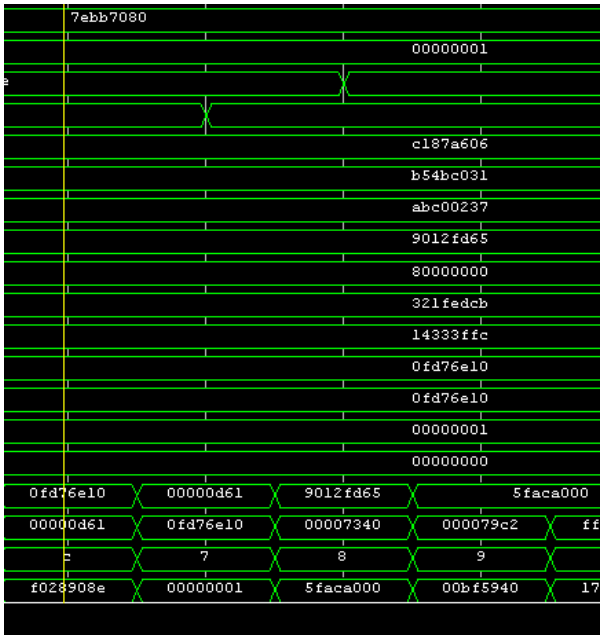
```
// forwarding test
rom[28] = 32'b00110000111010110000111101100011; // andi r11,r7,#f63          00000d61          r11= 00000d61
rom[29] = 32'b00000000010010110110000000100111; // nor  r12,r2,r11           f028908e          r12= f028908e
```

The code above shows 2 ALU operations, with r11 being calculated in the first instruction, then used in the second. Thus forwarding will occur.





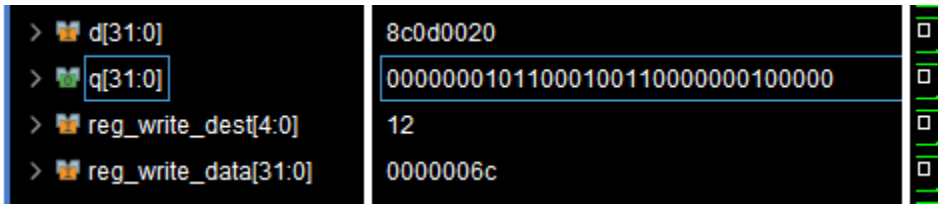
The initial value for r7 is 0x9012fd65. That bitwise anded with 0xf63 is equal to 0xd61, which we eventually see r11 eventually equal.



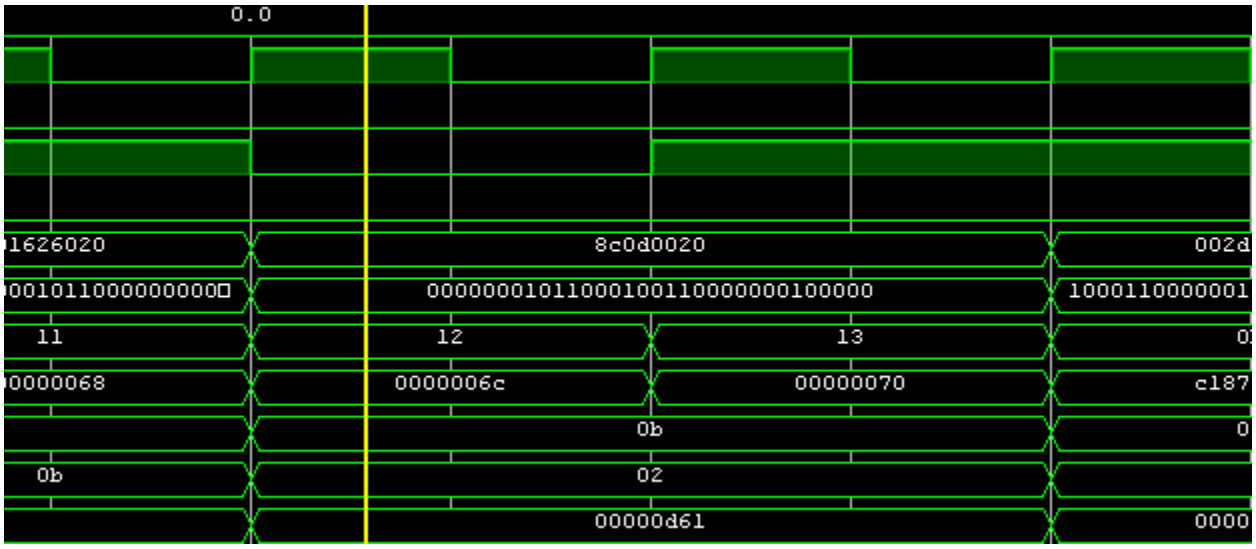
We can also see that r12 is updated on the next clock cycle, thus proving that forwarding did occur.

Demonstrating Data Hazards:

```
rom[46] = 32'b10001100000001011000000000100100; // r11 = mem[36]          9          r11= c187a606
rom[47] = 32'b00000000101100010011000000100000; // add r12,r11,r2       d15f1416       r12= d15f1416
rom[48] = 32'b10001100000011010000000000100000; // r13 = mem[32]          9          r13= b54bc031
rom[49] = 32'b00000000001011010111000000100000; // add r14,r1,r13       b54bc032       r14= b54bc032
// store the result in memory
```



Looking at the above screenshot, we can see that instruction 47 is executing at the instruction decode stage.

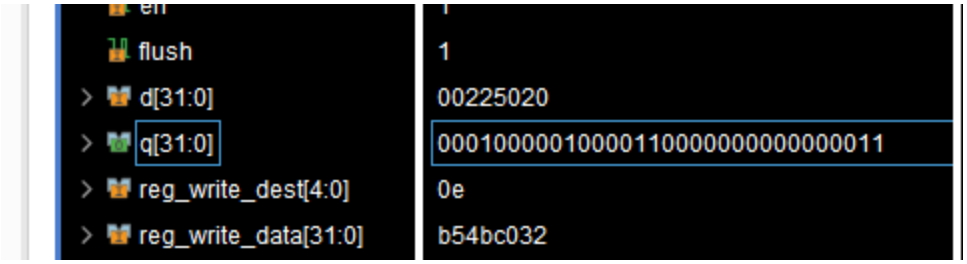


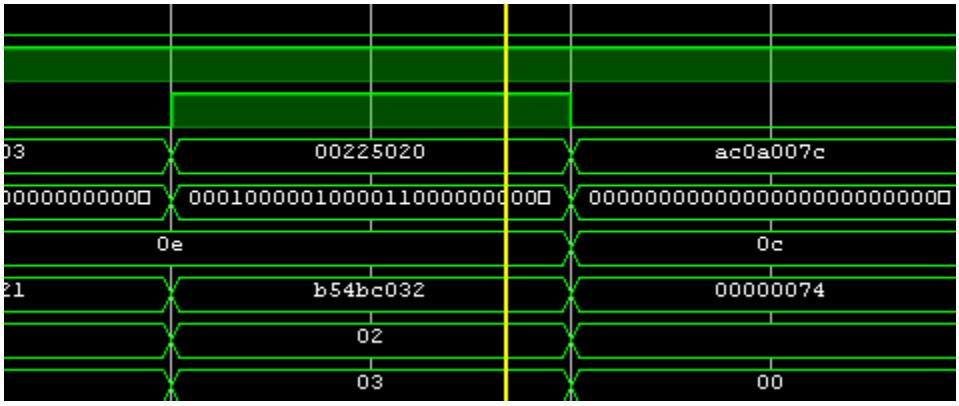
Looking at the waveform, we can see that the processor stalled itself due to the possible data hazard.

Demonstrating Branch Control Hazards:

```
// Control Hazard test Branch
rom[52] = 32'b00010000010000110000000000000011; // beq r2,r3,#3          0        branch to instruction rom[56]
rom[53] = 32'b00000000001000100101000000100000; // add r10,r1,r2      0fd76e11          r10= 0fd76e11          -
rom[54] = 32'b00000000001001000101000000100000; // add r10,r1,r4      14333ffd          r10= 14333ffd          -
rom[55] = 32'b00000000001001010101000000100000; // add r10,r1,r5      321fedcc          r10= 321fedcc          -
// store the result in memory
rom[56] = 32'b10101100000001010000000000011111100; // sw mem[r0+31] <= r10      7c                -                mem
```

Looking at the commands, when the processor executes instruction 52, it will branch, and so the flush signal should turn on.



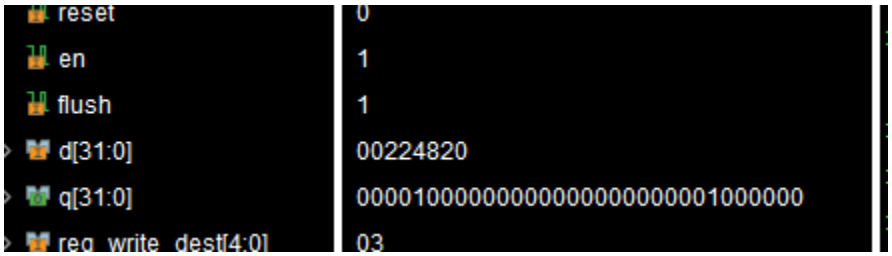


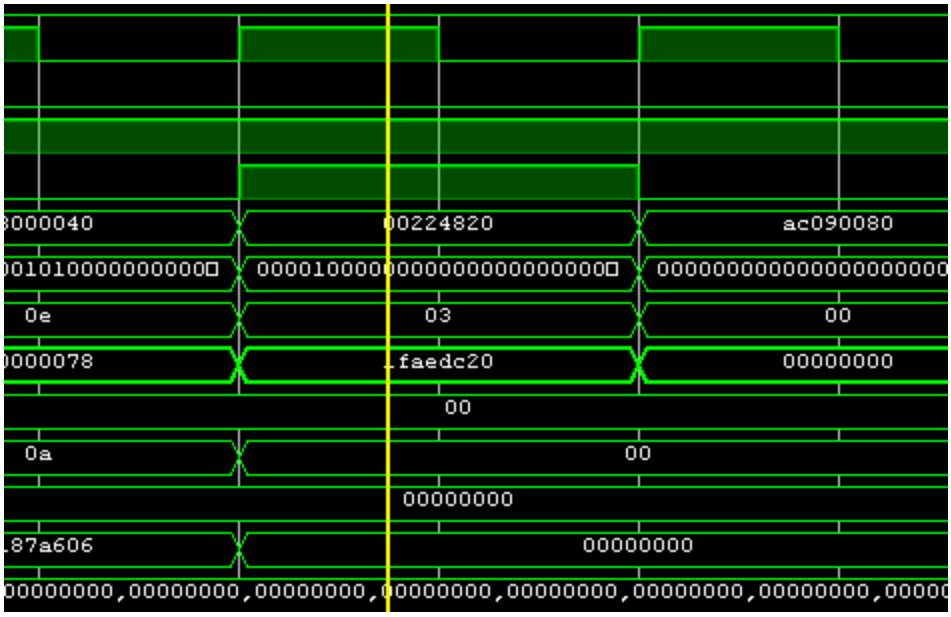
Looking at the 2 images above, we can see that the flush does turn on for the duration of the stage.

Demonstrating Jump Control Hazards:

```
// Control Hazard test 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
```

Looking at the adobe instruction we should expect the flush signal to turn on at instruction 57 as the processor will make a jump.





Looking at the 2 images above, we can see the process does have the flush signal on.