Computer Architecture Lab 4: Pipelined CPU

Lauren Anfenson
Anusha Datar
William Fairman

November 2018

1 Background

A single-cycle CPU executes one instruction in each clock cycle, so its speed of computation is limited by the slowest possible instruction. By breaking each instruction the CPU carries out into several discrete steps and then allowing the CPU to perform these steps for different instructions in parallel, we can lower our overall computation time by increasing the amount of time during which various resources are in use. This principle is the basic idea of a Pipelined CPU. In this lab we have modified a single-cycle CPU to include pipelining functionality – our new pipelined CPU conforms to the MIPS architecture and should be able to complete the instructions ADD, ADDI, SUB, SLT, Store Word, Load Word, XORI, and Jump.

2 Our Architecture

To implement basic pipelining, we added registers between the different stages of instruction execution of a single-cycle CPU to store necessary intermediate flags and values. The stages of instruction implementation were: Instruction Fetch (IF), where the program counter is incremented and the instruction is fetched from instruction memory, Instruction Decode (ID), where the control flags for the instruction are set and the relevant data is pulled from the register file, Execution (EX), where any ALU operations are performed, Memory (MEM), where data is written to or read from the data memory, and Write Back (WB), where values are written back to be stored in the register file. The block diagram of the architecture we used to implement this behavior can be seen below.

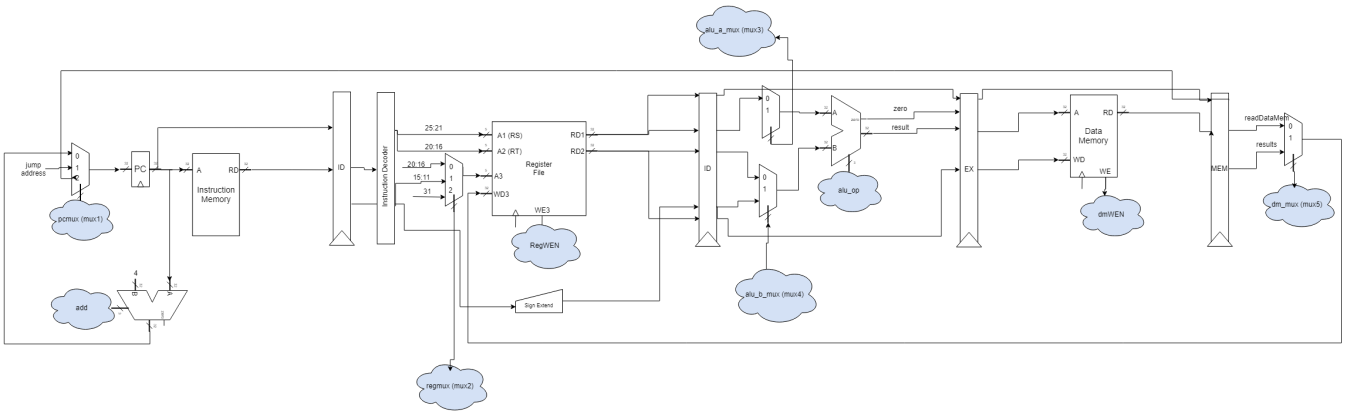

Figure 1: The block diagram of our pipelined CPU without hazard control.

To support a variety of instructions and modes, we created a set of flags in our instruction decoder to dictate the behavior of the rest of the CPU. The flags are defined as following:

Flag	**Definition**	I0	1	2
pc_mux	Program Counter Mux	PC + 4	Jump Address	32
reg_mux	Register Write Address Mux	rt	rd	32
alu_a_mux	ALU Input 1	rs	imm	N/A
alu_b_mux	ALU Input 2	rt	imm	N/A
dm_mux	Result Read Mux	DM read data	ALU result	N/A
reg_we	Register Write Enable	False	True	N/A
dm_we	Data Memory Write Enable	False	True	N/A
alu_op	ALU Operation	N/A	N/A	N/A

Then, we set these flags for each of the potential operations.

Instruction	**pc_mux**	**reg_mux**	**alu_a_mux**	**alu_b_mux**	**dm_mux**	**reg_we**	**dm_we**	**alu_op**
LW	1	1	0	0	1	0	0	x
SW	0	1	0	0	1	1	0	sub
J	1	0	0	0	0	0	0	x
XORI	0	0	0	1	1	1	0	xor
ADDI	0	0	0	1	1	1	0	add
ADD	0	1	0	0	1	1	0	add
SUB	0	1	0	0	1	1	0	sub
SLT	0	1	0	0	1	1	0	slt

3 Considering Hazards

When running assembly programs on the pipelined CPU, we quickly ran into hazards, or problems associated with program operation due to the pipelining behavior that can lead to decreased performance and/or incorrect results. The main issue that we addressed was a data hazard, where an instruction writes to a register but an instruction after it tries to use that same register to perform an operation. This is a problem because the second register will not be able to get the most updated value of that register until the first instruction reaches the Write Back stage, well after the second instruction has read from the register file. This is demonstrated in the table below that shows the execution of a simple assembly program.

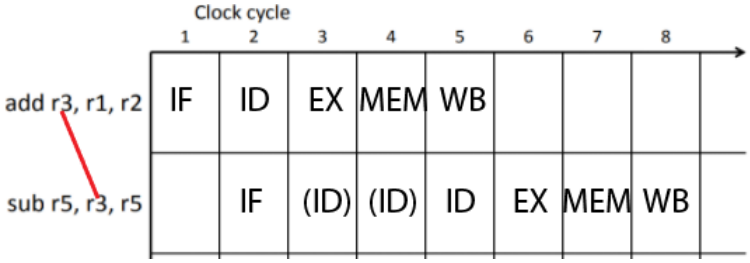

Figure 2: A table showing the instruction execution flow for our pipelined CPU when handling a data hazard. The instructions in parentheses represent a stall at that stage.

We are able to Write Back the register value and read it out in Instruction Decode within the same clock cycle because the CPU writes to the register file during the first half of the clock cycle and reads from the register file during the second half. This was implemented by writing on the positive edge of the clock and reading on the negative edge.

In order for our CPU to be able to recognize that there is a data hazard we created another module of control logic to detect hazards and set flags to stall the instruction if a hazard is detected. To figure out

whether there is a data hazard we store the register addresses that the last three instructions wrote to – Rd for R-Type, and Rt for I-Type. If the current instruction has an operand that conflicts with one of our past three writes, we stall for one clock cycle.

A stall is done by setting flags to disable the necessary registers to stop the next instruction from propagating through the CPU and send a no-op instruction in its place. This involves disabling the IF register so no new instruction can be passed into the ID stage, disabling the program counter so it stops incrementing the instruction temporarily, and passing all zeroes through the ID register to implement a no-op instruction instead of the instruction that caused the hazard.

The next type of hazard that we considered is a jump hazard, where the jump address cannot be passed to the program counter until it's in the Write Back stage, but by then the program counter already has passed the next incremented instruction and so failed to jump. Our solution was to automatically stall for three clock cycles when our hazard control unit detected a J-type instruction. Jump is the only J-type instruction we attempted to implement so this was a valid generalization.

The block diagram that we used to include hazard control functionality in our pipelined CPU can be seen below.

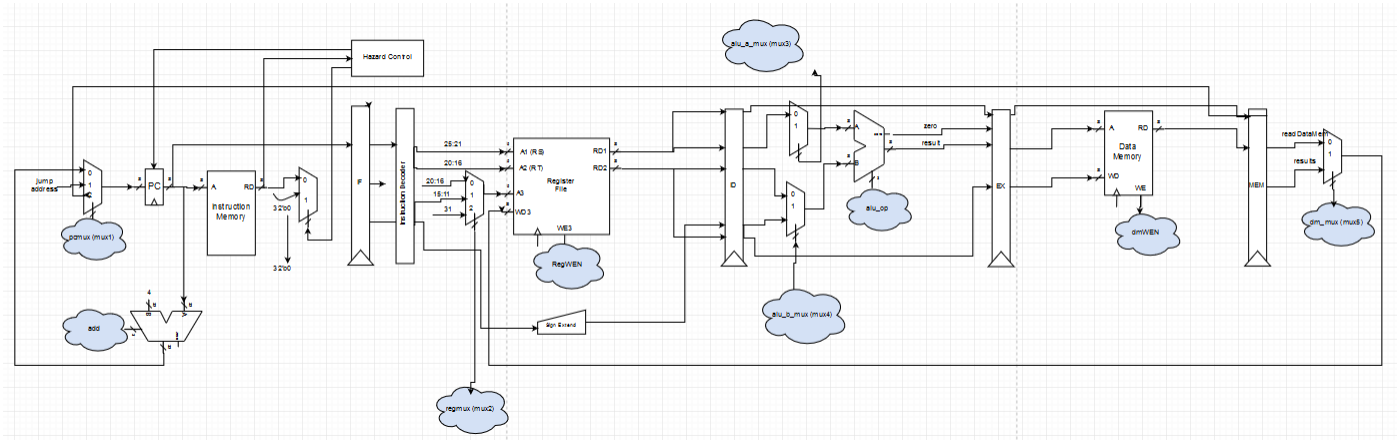

Figure 3: The block diagram of our pipelined CPU with hazard control.

4 Testing and Results

The first test we attempted to pass after adding basic pipeline functionality was a simple assembly program that stored `100` in registers $t1 and $t2. This tested our ADDI functionality, as well as our ability to read and write to registers.

After our CPU was able to complete that successfully, we moved on to running assembly programs that used the ADD, SUB, SLT, and XORI functionalities. These assembly programs had no-ops written in to prevent hazards instead of relying on our built-in hazard detection.

In keeping with the incremental testing of our CPU's functionality, we then ran a program that included a Jump. Once again, we included the no-ops in the assembly code rather than generating them in the CPU for this test.

After we confirmed that the majority of our basic functionality could be implemented and pipelined correctly without hazards we began testing our hazard control. The code-equivalent of our hazard control logic can be seen in the Appendix.

When we implemented this in verilog we ran into some confusing errors. We were able to enter a stall at the appropriate times and observe our flags change to indicate we were exiting a stall when expected. However, our program counter would continue to have a disconnected output even after we exited the stall, meaning we never stopped passing no-op as an instruction. While we believe that part of the problem that we see when we try to do hazard control on a Jump is that the jump address is never fully driven, we also observe this stuck-in-stall issue with data hazards. So, the root cause of the issue remains unclear.

5 Conclusion

Though we were not able to create as refined a hazard-free CPU as we would have liked, we were still able to implement a pipelined design. Our hazard control correctly identifies jump hazards and mostly correctly identifies data hazards, successfully creating a stall in response. However, our CPU is not able to resume correct operation after a stall is initiated.

6 Reflection

Before implementing hazard control, we managed to stay on schedule. We ran into no significant errors while implementing the new registers and spent a moderate amount of time debugging our hazard-free CPU. Our main issues stemmed from hazard control. In an attempt to simplify our process, we started by trying to implement a hardware stall function that would replaced our manual need to input no-ops into our code. Unfortunately, we were never able to get this completely working and did not have time to implement other forms of hazard control.

Our reasons for failing to implement hazard control most likely stem from a late-stage plan of implementing hazard control and the increased complexity of wiring and components that accompany a pipeline CPU. The breadth of our gtkwave file was too much to comprehend in one evening.

7 Appendix : Hazard Control Logic

```
always @(posedge clk) begin

      itype = (noopOut[31:26] == `ADDI || noopOut[31:26] == `XORI ||
        noopOut[31:26] == `LW || noopOut[31:26] == `SW);
      rtype = (noopOut[31:26] == `RTYPE);
      jtype = (noopOut[31:26] == `JUMP);

      write_hist4 <= write_hist3;
      write_hist3 <= write_hist2;
      write_hist2 <= write_hist1;

      if(itype) begin
        write_hist1 <= noopOut[20:16];
        count3 <= count2;
        count2 <= count1;
        count1 <= 1;
        if(noopOut[25:21] == write_hist2 ||
            noopOut[25:21] == write_hist3 || noopOut[25:21] == write_hist4) begin
          regIF_en <= 0;
```

```
20          regID_en <= 0;
21          nopMux <= 1;
22          pcEnable <= 0;
23        end
24        else begin
25          regIF_en <= 1;
26          regID_en <= 1;
27          nopMux <= 0;
28          pcEnable <= 1;
29        end
30      end
31      else if(rtype && noopOut != 32'b0) begin
32        write_hist1 <= noopOut[15:11];
33        count3 <= count2;
34        count2 <= count1;
35        count1 <= 1;
36        if(noopOut[25:21] == write_hist2 || noopOut[25:21] == write_hist3
37            || noopOut[25:21] == write_hist4) begin
38          regIF_en <= 0;
39          regID_en <= 0;
40          nopMux <= 1;
41          pcEnable <= 0;
42        end
43        else if(noopOut[20:16] == write_hist2 || noopOut[20:16] == write_hist3
44            || noopOut[20:16] == write_hist4) begin
45          regIF_en <= 0;
46          regID_en <= 0;
47          nopMux <= 1;
48          pcEnable <= 0;
49        end
50        else begin
51          regIF_en <= 1;
52          regID_en <= 1;
53          nopMux <= 0;
54          pcEnable <= 1;
55        end
56      end
57      else if(jtype) begin
58        write_hist1 <= 6'b0;
59        count3 <= 0;
60        count2 <= 0;
61        count1 <= 1;
62        regIF_en <= 0;
63        regID_en <= 0;
64        nopMux <= 1;
```

```
65          pcEnable <= 0;
66
67        end
68        else begin
69          if(count3 == 0) begin
70            regIF_en <= 0;
71            regID_en <= 0;
72            nopMux <= 1;
73            pcEnable <= 0;
74            count3 <= count2;
75            count2 <= count1;
76            count1 <= 1;
77        end
78        else begin
79            regIF_en <= 1;
80            regID_en <= 1;
81            nopMux <= 0;
82            pcEnable <= 1;
83        end
84      end
85    end
86  endmodule
```