Lab 4: Pipelined CPU

Annie Kroo, Adam Selker, Emma Westerhoff

November 16, 2018

Introduction

Our goal for this project was to create a functional and well-documented pipelined CPU. A pipelined CPU breaks the processing into multiple stages with faster clock cycles. Our design breaks the processing stream into five stages: instruction fetch (IF), instruction decode (ID), execute (EX), save to memory (MEM), and write back register (WB). A pipelined CPU has increased functionality over a multi-cycle CPU because a different instruction can be processed in each stage of the stream. For example, the IF cycle of instruction 3 can happen at the same time that instruction 2 is going through ID. Refer to the visual below.

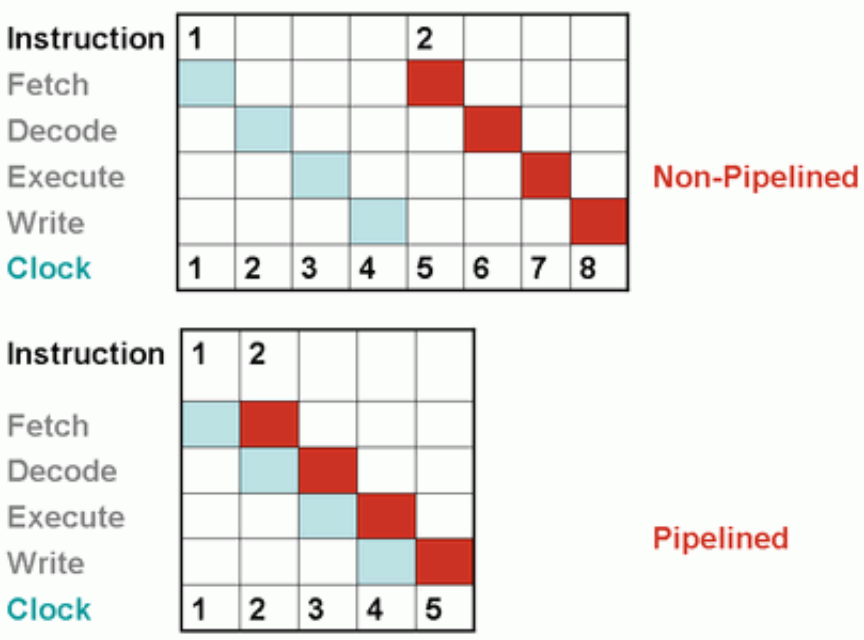

Figure 1: Pipelining Graphic: credits to stackpointer.io

It is easy to compare a pipelined CPU to a multi-cycle CPU, stating that a pipelined CPU is clearly better because instructions can be fed through 5x as quickly. However, there are some other differences. One is that a pipelined CPU cannot reuse hardware like a multi-cycle CPU. In addition, a pipelined CPU deals with special problems called control hazards.
There are two types of control hazards. Data hazards apply when instruction n-1 has information required to execute instruction n. If instruction n-1 writes to register $t3, then $t3 is not written until 3 stages after instruction n (which uses $t3 as an input) loaded its value from the register file. So, instruction n will receive an outdated value of $t3 if the problem is not addressed. We present a solution to this later, by forwarding our data from one stage to the next. Structural hazards apply when two instructions are trying to use one block of hardware at the same time.

Functionality

Our pipelined CPU has a basic instruction set, with the added functionality of jump and branch instructions to allow for function calls. We have reduced the number of restrictions we have on the assembly code specifics, allowing for any series of branches, jumps, and simple add subtract slt functions. The only restriction that we have found on the assembly code is that one should not use a jump register to create an infinite loop. If a jump register loops back to itself, it will continue on. For infinite loops, jump is the preferred instruction which is already common practice so does not reduce the overall function of our CPU.

Instruction Set

Below is the listed RTL of each supported instruction.

- LW: R[rt]=M[R[rs]+SignExtImm]
- SW: M[R[rs]+SignExtImm] = R[rt]
- J: PC=JumpAddr
- JR: PC=R[rs]
- JAL: R[31]=PC+4; PC=JumpAddr
- JumpAddr= PC+4[31:28], address, 2'b0
- XORI: R[rt]=R[rs] XOR SignExtImm
- ADDI: R[rt]=R[rs]+SignExtImm
- ADD: R[rd]=R[rs]+R[rt]
- SUB: R[rd]=R[rs]-R[rt]
- SLT: R[rd]=(R[rs]¡R[rt])?1:0

Block Diagram

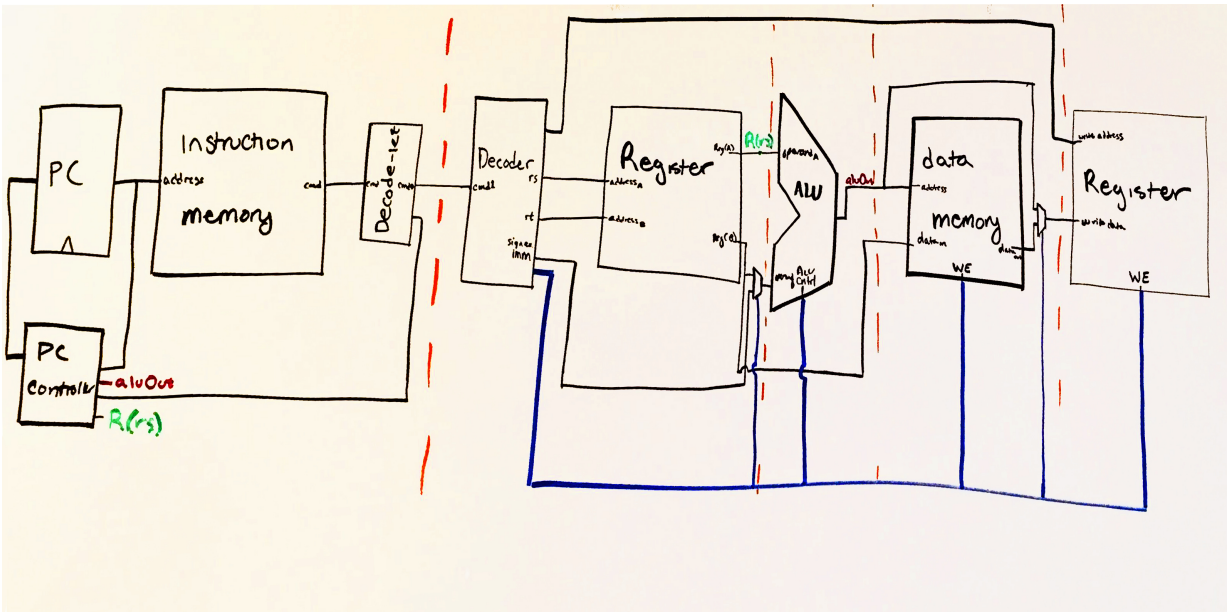

Figure 2: Block Diagram

Testing

We implemented a rigorous series of tests to ensure that our CPU was performing as desired, and met the specs that we sent out to meet at the start of the project. We used exhastively tested submodules from our previous single-cycle CPU design. This includes our register file, memory, PC, and ALU. We did this to ensure that the parts of our machine were not broken before we started stitching them together. We also did this to fix any gaps in knowledge between team members left over from previous labs, so that we were all on equal footing moving forward. All team members worked on different single-cycle CPUs, so we discussed implementation strategies for that project, and selected one best suited to our needs moving forward.

Standard Instructions

Our first test was designed to observe the functionality of standard I-type instructions, including xori, addi, subi, and slt. In conjunction with this, we also tested basic R-type operations like add and sub.
Our first version of this test was created with four no-ops between each instruction. We did this to ensure that the basic CPU functionality was working before we checked that the pipeline worked, alleviating our system from potential structural hazards. We eventually removed these.
After some minor connectivity and wiring problems, basic CPU functionality was established. We observed the passing of instruction from section to section, along with all of the control signals.
When we removed the no-ops and created the pipeline functionality, we ran into some difficulty getting the values to pass from one section to the next. Through our testing, we observed that it would somewhat sporadically pass through the values that we intended on the clock edge. Some investigation in gtkwave led us to find the cause of the error lay in the blocking definition of our registers as we passed the values from one section to the next. By setting our registers in always blocks with non-blocking equal signs, we were able to effectively pass our values from one section to the next.
After this, we quickly ran into our second set of hazards.

```
1 addi  $t1, $zero, 30
2 add $t2, $t0, $t1
```

Here, we notice that the second instruction depends on the value of $t1 calculated one instruction earlier. We fix this problem with forwarding (the process of which is described later on).

Memory

The next set of tests focused on the functionality of the load/store word operations. Our forwarding system was set up to forward the executed value (the ALU output) to the ALU inputs in order to use the most current value for that register. This system although effective for almost all scenarios neglects the use of load word followed by an operation using the output of load word. This is problematic because the value of the register to be used in the second operation is not available until the data memory stage, at which time it is too late for the value to be forwarded and have the next operation complete in time. As such we used a bubble to prevent such a data hazard. Every time we have a load word, we have implemented our PC controller and decode-let to trigger a bubble. This ensures that any load word is called, no matter what follows it, no data hazard will arise. Here we have optimized this hazard mitigation for a reduction in complexity and thus space, while ensuring functionality of our CPU. In this optimization, timing has been somewhat compromised, as we use a bubble even when a data hazard would not have occurred (e.g. if the value loaded was not used in the very next instruction).

Jumps

Our third suite of tests checked the J-type instructions. Jump and jump and link didn't require much extra hardware, aside from the enabling of an extra control signal and the hardware needed to determine its jump address from the instruction.
JR proved much more complex and required an extra selection of hardware that would prevent structural hazards from occurring. We chose to bubble whenever we detected a jump register. This means that when our decode-let comes across a jump register, it sends it through the pipeline but then stops the PC from incrementing

at the next cycle and instead sends no-ops for 1 cycle and then again sends the jump register to trigger the jump. During the second pass through of the jump register command the pc calculates that address to jump to, which is then available at the output of the register as R(rs), and we successfully jump to the appropriate command.

Hazard Mitigation: Implementation

A general overview of the above hazard mitigation is as follows.

Forwarding

In order to resolve data hazards, we make available the data that we will write to the registers early to the next command. Specifically, the data is available at the end of a program's execute stage, but written back to the register two cycles later. By forwarding the data available at the end of instruction n's execute phase to the beginning of instruction n+1's, then there is no data hazard. Extra hardware had to be added to see if this value was needed. This means that we have no restrictions on the sequence of commands and one instruction can set the value of a register, and the next can use that same updated value in its operation.

Bubbling

In the few cases in which forwarding does not fix our hazards such as in load word followed by the use of the loaded word or structural hazards such as in jump register, we utilize bubbling. We use bubbling as a somewhat last resort and only in these few circumstances to minimize the number of wasted cycles we spend. In these cases however, we keep track of what kind of command is currently being executed (and in the case of JR what command has been executed before), and pass no-ops into the system (add $zero, $zero, $zero) until the hazard is resolved, which is dependent on the instruction being executed. We still utilize the concept of forwarding, which is taking the data as soon as it is available in the pipeline, not waiting until it gets written back to the register file.

Next Steps

Future iterations of this project would include branch-prediction feature to save time. In addition, we could add functionality to the data memory to be able to pre-store arrays without writing them to memory each time using the register file.

Work Plan Reflection

Simple Pipeline Design (No Flow Control)

Block Diagram: 45 minutes
Comments: This took about as long as expected. Changes were made to the block diagram later in the project as we learned more and added functionality, but we started off with a good foundation.

Submodules: 30 minutes for decoder and 20 minutes for all others. We have all other modules already coded and working.
Comments: Took us about this amount of time to do this, but we spent an extra half hour discussing our previous project and implementation strategies.

Submodules Testing: 45 minutes for decoder, the only submodule that cannot be pulled from a previous lab.
Comments: This testing didn't happen until later, when the CPU was built. Still only took about 30 minutes at that point in the project though, since it was always a likely culprit of things not working.

Integration and Connections: 20 minutes

Comments: Took more around an hour and a half, between all the extra multiplexors we needed to add and making sure everything was connected.

Verilog Debugging: 1 hour
Comments: The wiring was the only debugging that we needed to do.

Assembly Testing and Debugging: 6 hours
Comments: We were able to reuse much of the assembly written for the previous lab. This total process probably took around 2 hours.

Total: 11.6 hours
Comments: 7.6 hours

Add Hazard Prevention (No Flow Control)

Planning Hazard Unit: 1 hour
Comments: We skipped this step all together, we had a pretty good idea of where to start due to lecture.

Coding and Implementation: 2 hours
Comments: Sounds about right

Testing and Integration: 3 hours
Comments: Only took about an hour.

Total: 6 hours
Comments: 3 hours

First Report:

We are aiming for a very well-documented project: 4 hours
Comments: We pushed this to our final project.

Total: 19.4 hours
Comments: 11.6

Second Section of Project

This section of the project contains our stretch goals: we'd really like to be able to implement these if our basic design takes less time than intended. The time estimations for this section of the project are very rough, and don't have much logic behind them aside from our previous experiences dealing with black box design.

Jumps: 4 hours
Comments: Took us about an hour

Branch, Jal, JR: 6 hours

Comments: Added Jal, Branch, and JR. Took us about 2.5 hours

Total time spent: 17