Processor Prototyping Lab

Midterm Report

Course: ECE437

Name: Xinyuan Cai, Xiangyu Guo

Lab Section: 4

Teaching Assistant: Bhakti Vora, Jimmy Jin

October 16, 2022

1 Executive Overview

So far, we have designed two types of processors without caches: SingleCycle and Pipeline. The Pipeline prototype is based on SingleCycle since the most significant change is four extra latches. However, four latches will cause hazards. The typical hazards we have are structural hazards, data hazards, and control hazards. The memory control is designed to handle the structural hazard. For Pipeline, the request unit will be adapted into the latch between the execution stage and the memory stage. The other structural hazard is caused by reading and loading from the registers simultaneously. Here comes another difference between SingleCycle and Pipeline: data will be written into the registers file on a negative clock edge. The second significant difference between SingleCycle and Pipeline is the forwarding unit. The forwarding unit is to handle the data hazard basically. The final difference is the hazard detection unit, which handles the branch hazard (control hazard).

Compared to the Pipeline, the SingleCycle doesn't have too many benefits, but it is the easiest to design in terms of hardware requirements. Unfortunately, the SingleCycle prototype has low clock frequency even though the CPI is only one. As a result, the performance is still bad. The Pipeline has a higher clock frequency according to the stages we have in the pipeline. The clock rate for this specific pipeline we designed is five times faster ideally since we have only five stages: instruction fetch, decode, execution, memory, and write back. Therefore, the basic improvement in the pipeline is the throughput or performance in other words.

2 Processor Design

(a) SingleCycle:

Our design of SingleCycle could complete the execution in one clock cycle when the memory latency is 0. We design the branch select logic that will combine the branch signal from the control unit and the zero signal to generate the final branchSel signal. To implement

a branch or jump address updates, we used several muxes in series. The first branch will determine the output as "npc" or the branch address based on the branchSel signal. The second mux will choose from the address from the output of the previous mux, the jump address, or the value read from register 31. The final address will forward to the program counter as the next address.

We also have several special designs for memory latency greater than 0. First, we extend the dmemload value until the next ihit signal. This will help the dmemload signal stay the same value until the next instruction fetch is finished. Second, the register-write signal combines the regWrite signal from the control unit and the ihit signal. It will be enabled only if both signals are high. This makes sure that each instruction will only write to the register once. Without doing this, when waiting for memory access, the value of the previous instruction will write to register repeatedly, and the data written will be zero when memory is not in ACCESS state.

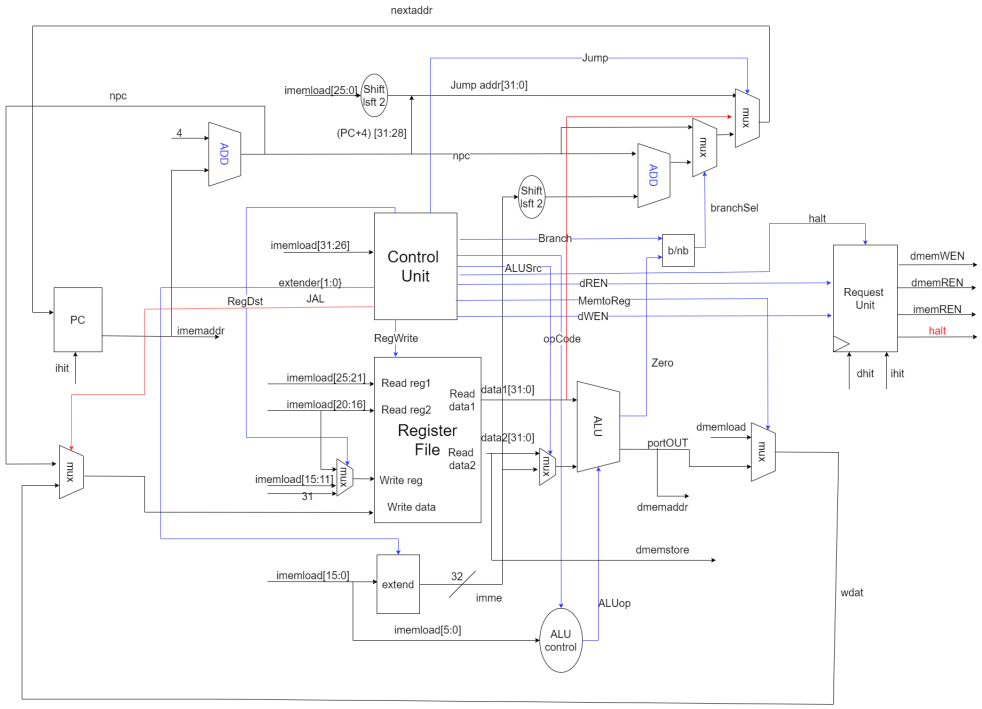

Figure 1: Single Cycle Block Diagram

(b) Pipeline:

Our pipeline design includes 5 individual stages. They are IF, ID, EXE, MEM, and WB stages from left to right (according to figure 2). Each of these stages is separated by corresponding latches. In the source files, we created a separate module and interface for each stage. It will make modification and debugging much more manageable. From the SingleCycle design, we removed the request unit; instead, we integrated it in the latch between the EXE stage and the MEM stage.

Three kinds of hazards are solved with different designs. The structural hazards are solved in memory control. It takes a higher priority of data access than instruction fetch. It will finish the data access first and then fetch the next instruction.

Most parts of data hazards are solved by forwarding unit. After detecting that a register will be used as input (in the EXE stage) and output (in MEM or WB stage), which means the data is not ready in the register, the forwarding unit will send the data from later stages to the corresponding ALU input port in the EXE stage. When MEM and WB stages write data to the same register, the forwarding unit will use the data from closer stage (MEM stage).

Although the forwarding unit could forward unready data to the EXE stage, there is still one situation that it cannot resolve. If one data needs to be used right after a load instruction, the data is likely to be unready in memory. Therefore, we have to add a NOP to wait for the memory to prepare the data.
At this point, we introduced the hazard detection unit. It will detect the hazard at the ID stage. When the RAW hazard is detected, it will stall the pc and flush the IF/ID latch. This will add a bubble to the pipeline. As a result, the RAW hazard penalty is 1 cycle.

The final hazard is the control hazard. We used static prediction which is the NON-TAKEN method to solve this hazard. Based on our design, similar to Singlecycle above, we treated jump and branch with the same method. When a jump or branch instruction is decoded, we just ignore it and update the program counter as usual. Then, when the jump address is ready or the branch prediction is incorrect in the MEM stage, the hazard

detection unit will flush all latches except the MEM/WB latch. This will cause 3 bubbles in the pipeline. As a result, the penalty for a jump or incorrect prediction is 3 cycles. If the prediction of the branch is correct, the program will keep going without any stall.

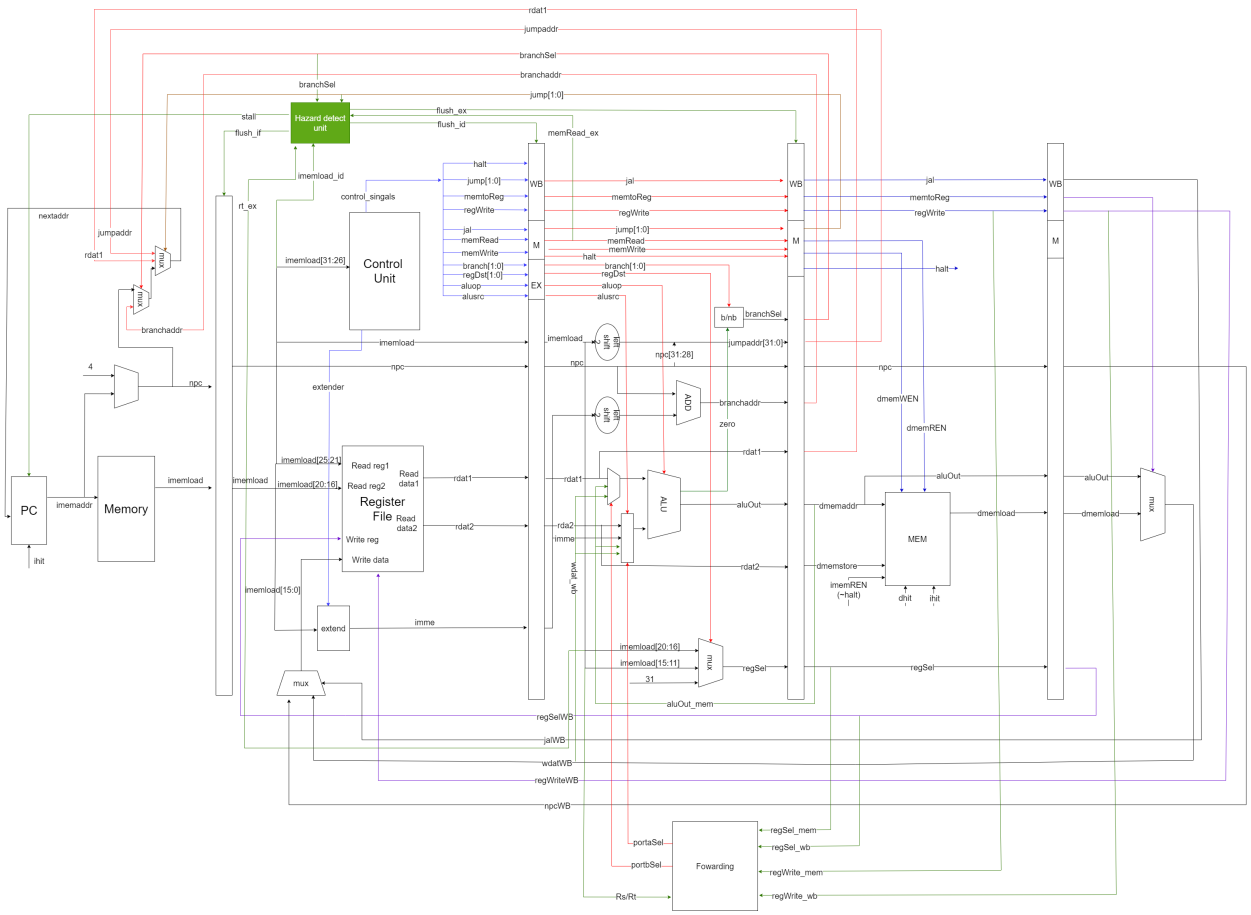

Figure 2: Pipeline Block Diagram

3 Results

We gather the data when memory latency is 0 and run with mergesort.asm. The frequency, number of cycles, and number of instructions are generated from the simulation result.

Criteria	pipeline	singlecycle
Frequency(CPU)Hz	$7.31 * 10^7$	$4.11 * 10^7$
# of cycles	10152	6900
# of instructions	5404	5404
CPI	1.879	1.277
MIPS	$3.89 * 10^1$	$3.22 * 10^1$
latency(ns)	$6.84 * 10^{-8}$	$2.44 * 10^{-8}$
execution time	$1.39 * 10^{-4}$	$1.68 * 10^{-4}$

FPGA resources:

Criteria	pipeline	singlecycle
Total logic element	3889/114480 (3%)	3242/114480 (3%)
Total combinational functions	3534/114480 (3%)	2962/114480 (3%)
Dedicated logic registers	1787/114480 (2%)	1312/114480 (1%)

Table 1: Processor Specs

$$\text{CPI} = \frac{cycles}{instructions}$$
$$\text{execution time} = \frac{1}{Freq} * CPI * numberOfInstruction$$
$$\text{MIPS} = \frac{numberOfInstructions}{executiontime} * 1 * 10^{-6}$$
$$\text{latency(Pipeline)} = \frac{5}{Freq}$$
$$\text{latency(SingleCycle)} = \frac{1}{Freq}$$

The number of cycles, CPI, and latency is increased because of the 5 stages of pipeline design. On the other hand, the frequency of the pipeline design is increased due to the shorter critical path. As a result, the MIPS increased. Thus, the execution time decreased for the specific assembly file.

The result shows that pipeline design uses more resources in FPGA on all aspects. This is

caused by the higher complicity of the pipeline. It includes more latches, a hazard detection unit, and a forwarding unit.

4 Conclusion

The performance, clock frequency, and critical path have huge improvement after adapting the Pipeline design. Although the circuit becomes much more complicated since it needs to handle three types of hazard and data forwarding, the MIPS (million instructions per second) improves a lot compared to the SingleCycle. What's more, to reduce the times of NOP to improve the performance, we use static branch prediction, which means that every branch will be predicted NOT-TAKEN. If the prediction is correct, we don't need to insert three times NOPs. If the prediction is incorrect, we can just flush the three wrong instructions right after the branch instruction and branch to the correct location after three NOPs. Therefore, the CPI will be closer to 1.

However, the increasing of complexity is huge. From the FPGA result part, you can see that much more resources are used in FPGA. The reason is obvious since there are three more latches, hazard detection unit, and forwarding unit. More resources corresponds with more cost. This makes sense since high performance always means more money.

5 Contributions

SingleCycle(Before lab5): Individually

Lab5 - Lab7

Xinyuan Cai:

Lab5: Write the source sv codes for each stages and modify datapath

Lab6: Write the source sv codes for hazard detection unit

Lab7: Write the testbench for forwarding unit

Xiangyu Guo:

Lab5: Modify singlecycle code to 5 separated interfaces to 5 stages.

Lab6: Write the testbench of hazard detection unit and test the sv code from teammate.

Lab7: Write the sv code for forwarding unit

Draw and modify each RTL diagram

Together: Design and Debug the overall process