ECE 437L Midterm Report

TA: Adam Hendrickson

Huayi Guo and Mingfei Huang

October 11 2013

EXECUTIVE SUMMARY

For past 7 ECE 437 labs, my lab partner and we designed and builded a single-cycle processor individually and a pipeline processor together to implement the MIPS instruction architecture set. The single-cycle processors can successfully handle basic MIPS instructions and execute assembly programs. In order to increase the throughput of the processor, my lab partner Huayi and we builded a pipeline based on the single cycle processor. The pipeline processor can successfully and properly handle all control and data hazards observed in the pipeline. In order to further increase the processor throughput, we attempted to add a branch prediction unit to the pipeline processor as a bonus feature but did not successfully make the unit fully functional before the lab deadline. From designing the processor, we further learned how a basic single-cycle processor works, how the CPU and RAM interface is, how the 5 stage processor works, and how to detect and handle read-after-write control and data hazards exsist in the pipeline processor design.

The rest part of this report mainly focuses on 1. Single cycle processor designs and how to test and verify them of both Mingfei Huang's and Huayi Guo's. 2. The design of our pipeline processor and how to test and verify it. 3. The testing, result and performance of the two single cycle processors and the performance result and testing of the pipeline processor design, including how the results are achieved.

PROCESSOR DESIGN

Single Cycle Processor - Huayi Guo

During verifying and testing the design of my single-cycle processor, hierarchy testing methodology is used. I have tested each single function block before starting combining any of them together. I started at testing the functionality of ALU, which has to be verified first to ensure other parts of the testing to be working. In this case, exhaustive test cases are used to ensure every single ALU operation is functionally correct.

Control unit signals are the secondary part to test using the cases that involves most of the R-type instructions, which doesn't require inputs of immediate value, to make the test as simple as possible.

Program counter and all the multiplexers are the next steps. As I have written all the muxes in a single module, it is slightly hard to connect in the top level but easier to write testbench for this block. Since at this stage there's still no memory control unit or hazard unit been added, memory associated instructions are omitted at this time. But all other instructions are tested.

After all the above been fully operated, hazard unit and memory control are incorporated into the design and a testbench for the top level can be written and tested.

Single Cycle Processor - Mingfei Huang

In the process of testing my single cycle processor, I splitted the design into several steps in order to modelize the entire design. To test the arithmetic logic unit, I wrote a testbench to provide different ALU opcode and observe if all calculations, flags agrees with expectation. The test cases include all operations with zero, small number and large, close to overflow number and overflow test cases.

To test the control unit, I used similar method by writing a testbench to provide different instructions to the control unit and observe the output control signals. Test cases include all types of instructions.

I then designed and tested the datapath with the similar method but also used a "fake memory unit", which is a simple pure combinational data array with PC as index and provide the 32 bits instructions, this unit helps me detect any logic bugs exist in the previous design without considering any timing factors from RAM unit. Test cases are also all types of instructions including loops, sw/lw involved programs.

I then added the hazard unit and memory control unit to connect the whole system together. Testing for this part is mainly done by using the system level testbench provided by TA, and simply looking at waveform and tracing signals since this is the last part of the design. This is the part of the entire design that caused most problems because it started to involve timing factors and also is lack of documentation of RAM, hazard, memory units involved. Test cases include all types of simple instructions, instructions with same data sources (r2=r1+r1), instructions with destination register same as source register (r1=r1+r2), branch taken/not taken, and lw/sw followed by/before branch instructions.

Pipeline Processor

At highest level, pipeline hazard we need to handle for our design can be categorized into two kind: 1. data hazard and 2. control flow hazard. Furthermore, the kind of data hazard that our design has is only Read After Write hazard, so it has the subsets of 1a. those that don't need stalls and 1b. those that need stalls. Since those that doesn't need stalls are the simplest and most basic case, we verified this part first.

1a. The behavior of program segments that have data hazard but don't need stalls is that the data that a later instruction needs to read from register file is modified by a previous instruction but has not go through the Write Back stage when the later instruction reads in the instruction decode stage, but the data is already correctly computed by the previous instruction, so only a forward from another pipeline latch is needed. For example, data might need to be forwarded from ID/EX latch to ID stage, a test case can be:lui r1,0x1; addu r1,r1,r3; jr r1; data might need to be forwarded from EX/MEM latch to EX stage, a test case can be:ori r1,r0,0x1; or r2,r1; data might need to be forwarded from MEM/WB latch to MEM stage, a test case can be:lw r1,r2; sw r1,r3.

1b. Program segments that need stalls, which are: EX followed dependent ID, for example: ori r1,r0,0x1; jr r1; which need one bubble to be turned to ori r1,r0,0x1; nop; jr r1. load word followed by dependent EX instruction for example: lw r1, r2; or r1, r2, r0, which needs one bubble to be turned to lw r1, r2; nop; or r1, r2, r0. load word followed by dependent ID instruction which needs two bubble: lw r1,r2;jr r1; need to be turned to lw r1,r2;nop;nop;jr r1.

2. control flow hazard: our design assumes branch always not taken, with branch handled in ID stage PC load new value in EX stage so no matter what instruction is loaded, IF stage assumes next PC is current PC + 4, so when ID stage decides value of the correct next PC value, the IF stage has already loaded a new instruction. Hence when a branch is taken, instruction load in IF stage is from an incorrect memory location and need to be flushed. To verify this mechanism is correctly working, test cases of different branch/jump with either branch taken or not are used.

For other corner cases, those that we had considered include:

register write followed by dependent branch jr or j type

jal followed by jr/branch

source and destination register are same

lw followed by sw

sw followed by lw

alu operation’s destination register same as lw/sw’s address register

jal/lui’s destination register same as lw/sw’s address register

RESULT

Single-cycle processor

The critical path results from the synthesize log file meets our expectation towards the critical path of the single-cycle processor design. Since it's 37.566 ns which is almost twice as the longest timing constraints we were supposed to test on, no doubt our design falling the simulation of target frequency of even 50MHz. After analysing the system.log file, we have tried running simulation targeting 25MHz and it turned out to be properly working. This is reasonable before we've already known the critical path for register file alone is around 20ns.

The metrics for the simulation targeting 25MHz is listed below for the purpose of comparison with pipelined processor:

Single-cycle metrics

Average IPC	0.152
Average latency of each instruction(ns)	261.75
performance (MIPS)	3.82
FPGA resources required(area)	1284
Total time consumed(ns)	41880

Pipelined processor

After successfully synthesizing our design, the worst-case critical path turns out to be 16.694 ns, starting from syif.tbCTRL signal to syif.load[0] signal, which is the ram delay.
That is acceptable as it was expected in our design.

We have also run the simulation to meet the timing constraints in the table below. As our design can successfully handle the case when the clock period goes down to 10ns, the Average latency of each instructions goes down as well because of the shorter clock time. Thus under same condition it will achieve approximately as twice the performance as using 20ns clock period. Also as indicated below, the clock period does not affect the average instructions per cycle, as it's a measurement of performance that independent from the clock rate.

However our pipelined design can't handle when the clock period goes down to 2ns,
as it will not possible for it to wait for the critical path finish so most of the instructions will be lost as clock flies.

pipelined processor metrics:

Clock period	20ns	10ns	2ns
Estimated frequency	50Mhz	100Mhz	500Mhz
Average IPC	0.133	0.133	n/a
Average latency of each instruction(ns)	156.5	78.25	n/a
performance (MIPS)	6.34	12.79	n/a
FPGA resources required(area)	1781 registers	1781 registers	n/a

Total time consumed(ns)	25020	12510	n/a

After summarizing the specs of the two processors, we can derive several interesting fact from the comparison. First of all, our pipelined processor used more registers and space then the single-cycled processor does. This makes sense since the pipelined processor contains all the logic of single-cycle processors we designed, and it also has additional pipeline registers, forwarding unit and data hazard handling logic. The portion of the additional hardware is significant, which at the end also speed up the performance at a comparable amount. More specifically, the resources consumption for the pipelined processor is 38.7% more than single-cycle processor of our design.

But the critical path is decreased 55.6%, which allows half of the clock period of the single-cycled processor can be tolerated by the pipelined processor. This result introduce that fact that it's very hard to achieve absolute improvement but a lot easier to improve one aspect, say the clock speed, in the cost of other aspects, such as area and presumably the power.

Another result comes to our attention is that our single-cycle processor has a slightly higher IPC, which means it is slightly more efficient than pipelined processor in terms of processing an instruction within one cycle. Our explanation for this is that since pipelined processor is using bubbles to solve data hazard, as well as blank operations at the very beginning of the program will cause the IPC to decrease a small amount. And this ratio of IPC(or inverse of CPI) may vary depending on the fraction of consecutive dependent instructions in the program.

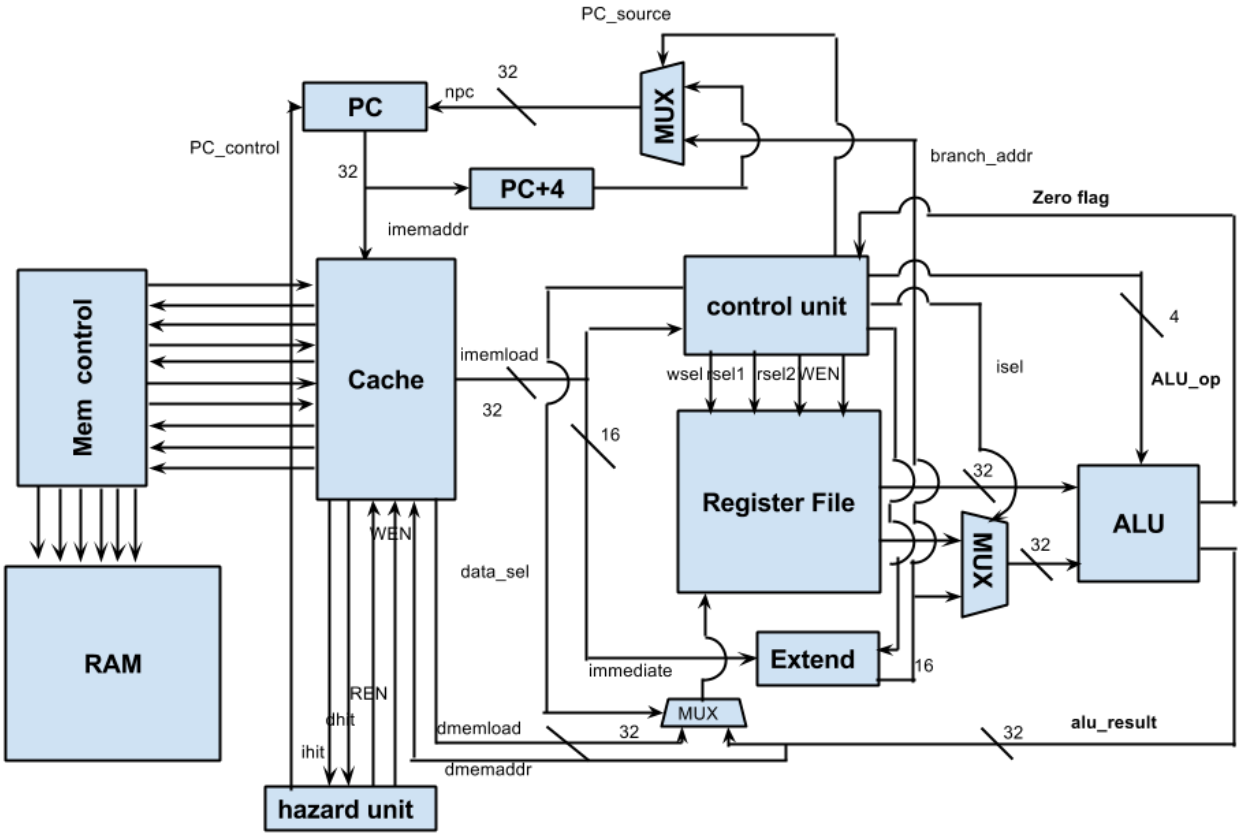
CONCLUSION

Our design of single-cycle and pipeline processor functions perfectly, they can handle all instructions in design requirement and work fine for any case of instructions. The flaw in our design is that the CPI values of our processors (both the single cycle of Mingfei's and Huayi's and pipeline processor) are much larger than ideal, which is supposed to be equals to latency of RAM + 1. This flaw causes our CPI to be about 5 instead of RAM's latency 2 + 1 = 3.

What we learned from this lab is mostly details of how a single/pipeline processor works, how to handle RAW hazards in a pipeline processor, how to collaborate with others using git and we gained a lot of experience in debugging HDL language.

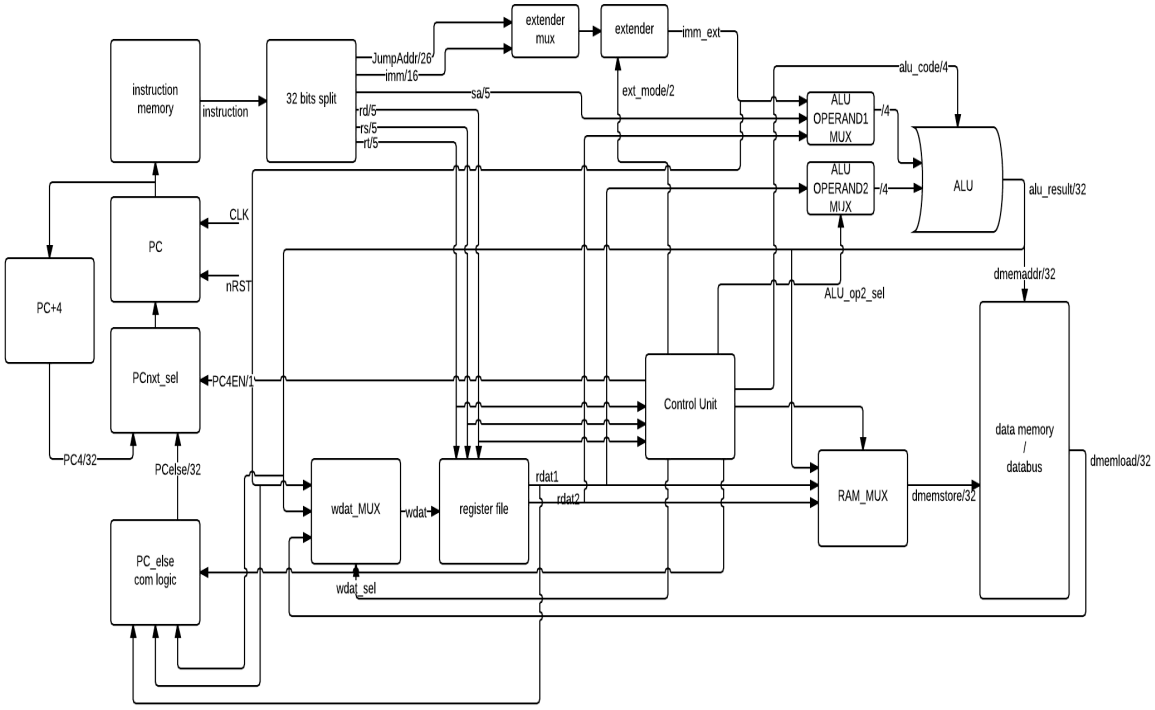
Appendix A

Block diagram for Huayi's single-cycle processor



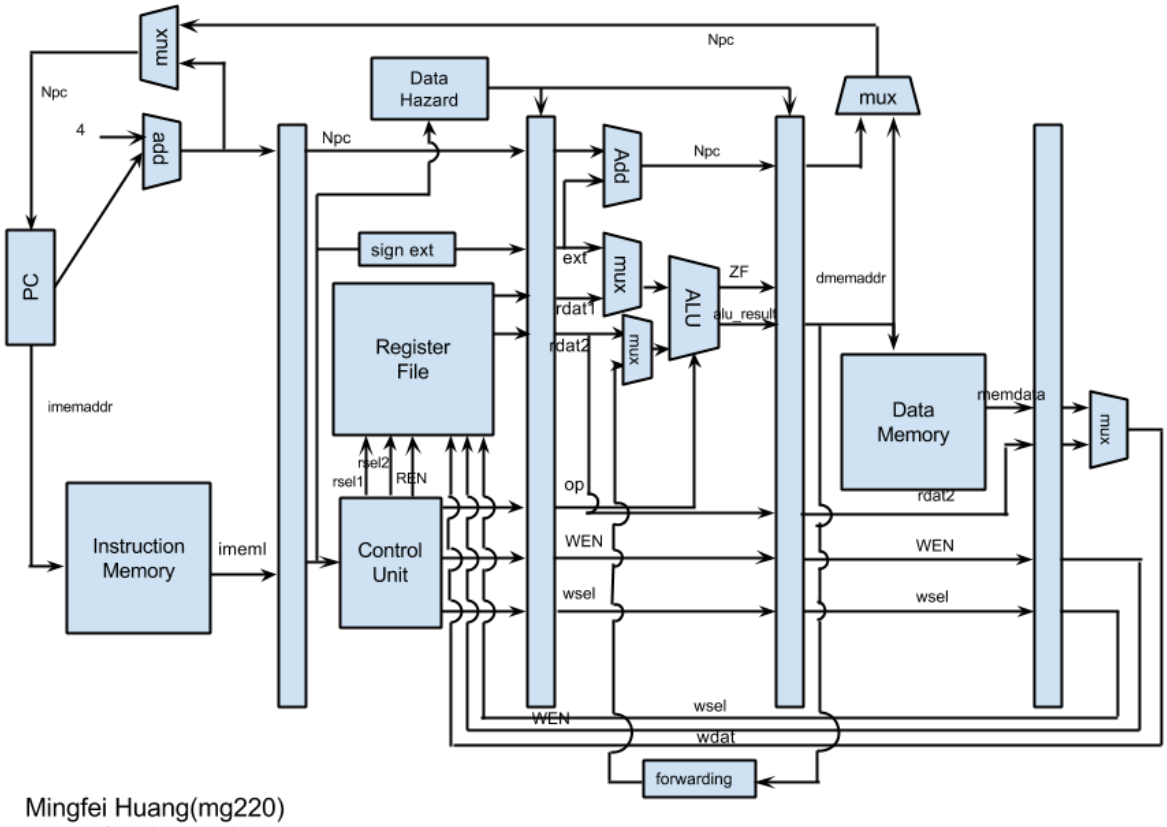
Appendix B

Block diagram for Mingfei's single-cycle processor



Appendix C

Block diagram for pipelined processor



Mingfei Huang(mg220)
Huayi Guo(mg224)

Section 06