Computer Science M151B, Homework 4

Michael Wu
UID: 404751542

April 30th, 2018

Problem 1

a) The first program has

$$\frac{1.1 \text{ s} \times \frac{1 \text{ cycle}}{10^{-9} \text{ s}}}{10^9 \text{ instructions}} \approx 1.1 \text{ CPI}$$

and the second has

$$\frac{1.5 \text{ s} \times \frac{1 \text{ cycle}}{10^{-9} \text{ s}}}{1.2 \times 10^9 \text{ instructions}} \approx 1.25 \text{ CPI}$$

b) Changing the processor means that both the cycles per instruction could change and the clock speed could change, so we cannot make any conclusions about the relative clock speeds of the different processors.

c) The new compiler will generate an executable with the execution time

$$\frac{6 \times 10^8 \text{ instructions} \times 1.1 \text{ CPI}}{10^9 \text{ cycles/s}} = 0.66 \text{ s}$$

which represents a

$$\frac{1.1 \text{ s}}{0.66 \text{ s}} \approx 1.667$$

times speedup over compiler A's program, and a

$$\frac{1.5 \text{ s}}{0.66 \text{ s}} \approx 2.273$$

times speedup over compiler B's program.

Problem 2

$$\begin{aligned}\frac{1}{10.1} &= \frac{1}{x} \times \frac{1}{2.45}\\ x &= \frac{10.1}{2.45}\\ x &\approx 4.122\end{aligned}$$

The programmer's instruction count is improved by a factor of approximately 4.122 relative to the compiler's.

Problem 3

We need to find the total number of cycles over the total number of instructions. Let r be the clock rate in cycles per second. Then the total number of instructions i is given by

$$i = r\left(\frac{3.8\text{ s}}{3\text{ CPI}} + \frac{8.5\text{ s}}{3.5\text{ CPI}}\right) = r\frac{388}{105}\text{ instructions}$$

and the total number of cycles c is given by

$$c = r(3.8\text{ s} + 8.5\text{ s}) = r\frac{123}{10}\text{ cycles}$$

Then we have that the new cycles per instruction will be

$$\frac{r\frac{123}{10}\text{ cycles}}{r\frac{388}{105}\text{ instructions}} \approx 3.329\text{ CPI}$$

Problem 4

a) Let i be the number of instructions in the program. The cycles per instruction x of non floating point operations is given by

$$\begin{aligned}\frac{0.82i \times x + 0.18i \times 5.3\text{ CPI}}{i} &= 3.6\text{ CPI}\\ x &\approx 3.227\text{ CPI}\end{aligned}$$

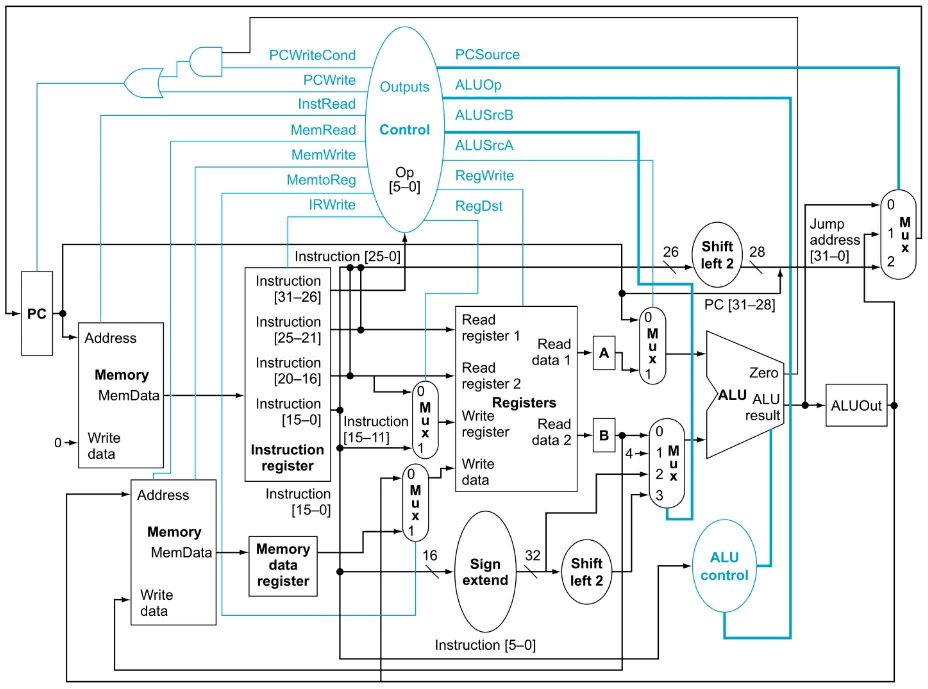
Then the overall cycles per instruction for the improved processor is

$$\frac{0.82i \times 3.227\text{ CPI} + 0.18i \times 3\text{ CPI}}{i} \approx 3.186\text{ CPI}$$

b) It would take 194.7 s to execute the program on the improved processor.

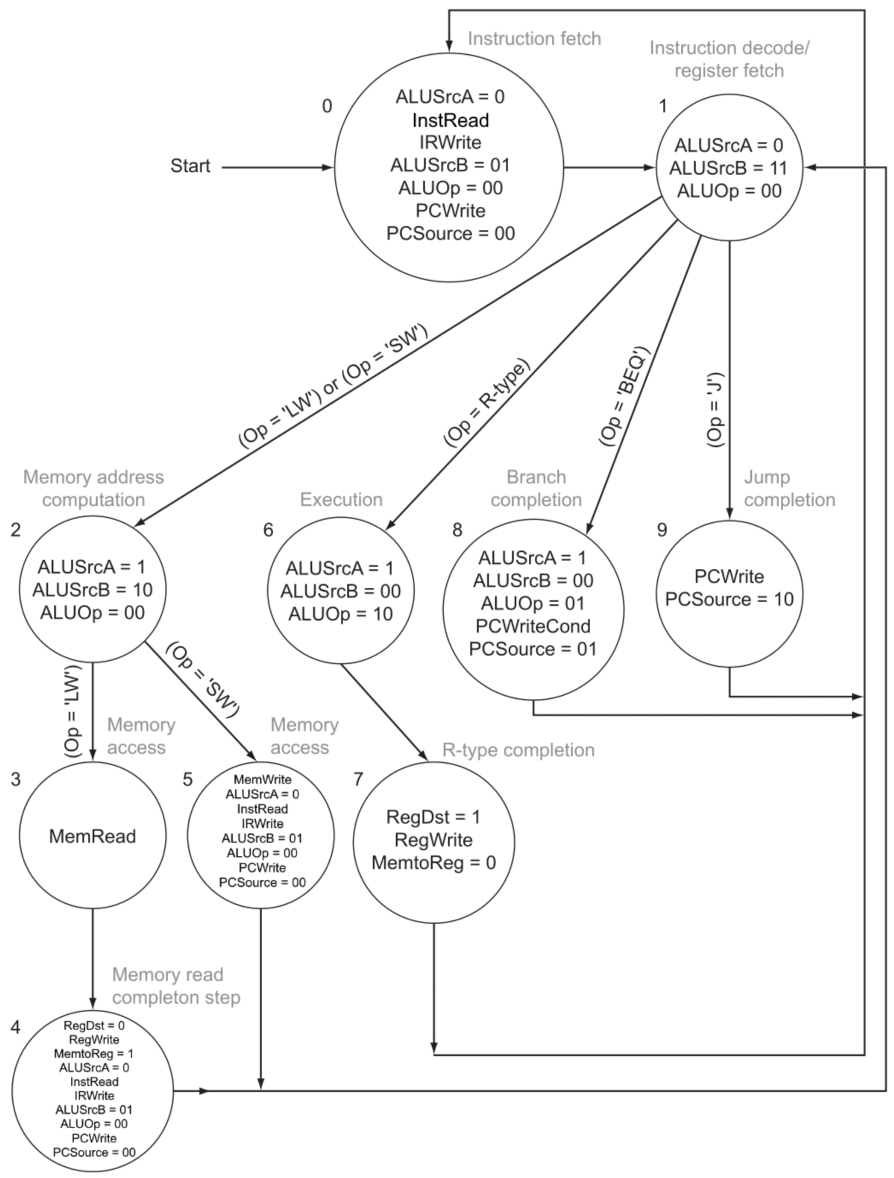
Problem 5

a) I would add an instruction memory unit so that the final step of load and store can be done in parallel with the instruction fetch. The last state of both the load and store instructions would point to the instruction decode step, reducing the cycles per instruction by 1.



b) The datapath changes above show the addition of the instruction memory unit. I input a zero to the `write data` signal of the instruction memory unit since it should not be doing any writing, so this value will never be used.

c) The control signal `IorD` is renamed to `InstRead`, since there is no need for a multiplexer to choose between reading instruction memory or regular memory. This signal controls whether to read from the address stored in the program counter or not.



d) The state changes are shown above. I have removed `IorD` and have allowed for the instruction fetch to happen at the same time as the last states of the load and store instructions. This allows them to immediately go to the instruction decode step, reducing the number of cycles required by 1.

Problem 6

Since `lw` uses every stage of the pipeline, its latency will be 1750 ps in the pipelined processor. This is because the clock cycle in a pipelined processor will be equivalent to the latency of the longest stage, which is 350 ps. So `lw` will execute over 5 clock cycles. In the equivalent non-pipelined processor that uses a single cycle implementation, the clock cycle must accommodate the longest instruction. This would mean that the clock cycle length is the sum of all the latencies of the instruction stages, and so it would be 1250 ps. So in the non-pipelined processor `lw` would have a latency of 1250 ps.

Problem 7

I would split the `ID` stage of the pipelined datapath into two, as this will decrease the latency of the longest stage to 300 ps. So the new clock cycle time of the processor would be 300 ps.

Problem 8

The instruction i_1

```
or r1, r2, r3
```

has no data dependencies. The instruction i_2

```
or r2, r1, r4
```

has a data dependency on the first instruction i_1. There is a Read After Write (RAW) data dependency, and it occurs due to i_1 writing to `r1` before i_2 reads from `r1`. There is also a Write After Read (WAR) dependency due to i_1 reading from `r2` before i_2 writes to `r2`. The instruction i_3

```
or r1, r1, r2
```

has data dependencies on both i_1 and i_2. There is a RAW dependency due to i_i writing to `r1` before i_3 reads from `r1`. There is a Write After Write (WAW) dependency due to i_1 writing to `r1` before i_3 writes to `r1`. There is a RAW dependency due to i_2 writing to `r2` before i_3 reads from `r2`. Finally, there is a WAR dependency due to i_2 reading from `r1` before i_3 writes to `r1`. Note that in our simple pipelined processor we only really need to be concerned about RAW dependencies, since we do not do out-of-order execution. So WAR and WAW dependencies will be taken care of and not change the execution in any way.

Problem 9

I will use

```
add $zero, $zero, $zero
```

as my `nop`. Then we need to ensure that all dependencies are taken care of. As noted before, WAR and WAW will be taken care of already because our processor does not do out of order execution. They are not a hazard, but the RAW dependencies represent a data hazard. To ensure that the RAW dependencies are taken care of, we must ensure that the `ID` stage of an instruction occurs after the `WB` stage of the previous instruction. This means that we need three `nop` instructions between each original instruction, because `WB` is the fifth stage and `ID` is the second stage. So our final program will be the code shown below.

```
or  r1,    r2,    r3
add $zero, $zero, $zero
add $zero, $zero, $zero
add $zero, $zero, $zero
or  r2,    r1,    r4
add $zero, $zero, $zero
add $zero, $zero, $zero
add $zero, $zero, $zero
or  r1,    r1,    r2
```

Problem 10

As before, data hazards occur due to the RAW dependencies. But with full forwarding, they should not slow down execution and no `nop` instructions are required. The instruction i_2 will receive its ALU input through forwarding the contents of the `EX/MEM` register from i_1, and the instruction i_3 will receive its ALU input through forwarding the contents of the `EX/MEM` register from i_2 and the contents of the `MEM/WB` register from i_1.

Problem 11

a) For the `sw` instruction, no control signals need to be asserted for the `IF` and `ID` stages. During the `EX` stage, the signals `ALUOp = 00` and `ALUSrc = 1` are asserted. In this stage `MemWrite` is asserted, but it does not do anything and is passed along to the next stage of execution. In the `MEM` stage, the signal `MemWrite` is asserted and performs the writing of data to memory. In the `WB` stage, no control signals are asserted.

b) For the `sub` instruction, no control signals need to be asserted for the `IF` and `ID` stages. During the `EX` stage, the signals `RegDst = 1`, `ALUOp = 10`, and `ALUSrc = 0` are asserted. In this stage `RegWrite` and `MemtoReg = 0` are asserted, but they not do anything and are passed along to the next stage of execution. In the `MEM` stage, again `RegWrite` and `MemtoReg = 0` are asserted, but they do not do anything and are passed along to the next stage of execution. In the `WB` stage, the signals `RegWrite` and `MemtoReg = 0` are asserted, storing the results of the ALU operation into a register.