Part 1: MIPS ISA and Memory: The Basics

Problem 1: Hands-on MIPS Code

Please express the following register expressions as MIPS code fragments, using as few instructions as possible. The registers in each expression below have been loaded with the correct data, and no other integer registers are in use.

1. \( R1 \leftarrow 164000 \)
2. \( R2 \leftarrow 2(-15 + R3 - R2 + R5) \)
3. \( R3 \leftarrow R3' + 1 \)
4. \( R4 \leftarrow (R5 \oplus R2)' \)
5. if \( R2 == 0 \) then \( R4 \leftarrow R4 - 24 + R2 \) else \( R4 \leftarrow 24 - R4 - R2 \)

Problem 2: Hands OFF the ISA questions

Briefly answer the following questions, explaining your answer for full credit, of course.

2.1 True or False: The ISA uniquely determines the specifications of the hardware on which it is to be executed.

2.2 Why is the encoding of the ISA so important in designing a computer architecture? That is, speculate as to why so much of chapter 2 is spent justifying the decisions made in designing MIPS.

2.3 Why is byte alignment an issue? That is, why might an ISA forbid one to load a word from a specific address? For example, in each pair of MIPS instructions below, the first instruction results in a byte alignment error, while the second doesn’t. To see this, compute the effective address for each operation—that is, assuming you know what one is.

Also, why might allowing the first instruction below cause problems? And, are there any load/store instructions that are immune from byte alignment errors in MIPS?
LD R2, 7(R0) ; byte alignment error
LD R4, 128(R0) ; legal instruction

DADDI R8, R0, #99 ;
SD R3, 17(R8) ; byte alignment error
SD R5, 29(R8) ; legal instruction
 ; check store syntax!

DADDI R9, R0, #1050;
L.D F2,-1 (R9) ; byte alignment error
L.D F4,-26(R9) ; legal instruction
S.D F11,-998(R9) ; legal or not?
1 Part2: Pipelining and the Pipe-Like Pipe

Problem 3: Hands ON the Pipe-Like Pipe

It would help to have read the beginning of Appendix A, and/or

3.1 What do I mean by “pipe-like pipe”? That is, why do I constantly refer to it that way, instead of merely “the pipeline”?

3.2 Compute the speedup due to forwarding and bypassing for the extremely strange code fragment below, assuming that the 5-stage pipe-like pipe is used. By raw, I mean the naive version that we discussed in class, before we'd ever heard of structural hazards. There is no split instruction and data memory, and register reads and writes conflict before and after bypassing and forwarding are added.

*Don’t panic, see the steps below the code fragment.*

```
LD R1, 100(R1);
LD R2, 200(R1);
DADDI R1, R2, #3000;
DSUB R2, R4, R1;
AND R2, R3, R2;
SD R2,400(R2);
LD R3,100(R0);
XOR R3, R3, R3;
SD 0(R3), R3;
LD R3, 0(R2);
SD R3,100(R3);
```

**Interpretation:**

1. Compute $\text{CPI}_{\text{old}}$ by tracking execution of the fragment without bypassing and forwarding.
2. Compute $\text{CPI}_{\text{new}}$ by tracking execution of the fragment assuming bypassing and forwarding.
3. Speedup is $\text{CPI}_{\text{old}}$ divided by $\text{CPI}_{\text{new}}$. Why? Make sure you know!
4.1 Repeat the last part of problem 3, except this time, assume that split instruction and data memory are used, and that registers are written in the first half of the clock cycle, and read during the second throughout the entire problem. That is, what is the speedup due to bypassing and forwarding when all the structural hazards have been removed?

LD R1, 100(R1);
LD R2, 200(R1);
DADDI R1, R2, #3000;
DSUB R2, R4, R1;
AND R2, R0, R2;
SD R2, 400(R2);
LD R3, 100(R0);
XOR R3, R3, R3;
SD R3, 0(R3);
LD R3, 0(R2);
SD R3, 100(R3);

4.2 Okay, now, just for grins and giggles, see if you can play optimizing compiler and accomplish whatever it is our friendly code fragment with fewer instructions, if possible, and try to minimize the CPI, essentially repeat 3.2 and 4.1 on your modified code fragment. The point is for you to become comfortable with the way code executes on the pipeline. Why? Because we’ll be doing a lot of that in the next two weeks.

4.3 Bored yet? Well, here’s my favorite question of all. There’s this formula on page A-13, on your cheat sheet, and in Bill’s Pipelining Handout.

It says that the speedup from pipelining is

\[
\frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \text{Pipeline depth.}
\]

Why do I just adore this formula? NOT! Go back and look at your problem 3.1 and 4.1 execution profiles (my term for mapping out what happens while executing on the pipe-like pipe) for the unoptimized code fragment. Then, apply this formula. And, then, tell me. I have a real problem applying this one. Why? Can you compute the number of stalls per instruction, independent of instruction order?