Instructions:

- Print your name in the space provided below.
- This examination is closed book and closed notes, aside from the permitted one-page formula sheet.
- No calculators or other computing devices may be used. The use of any such device will be interpreted as an indication that you are finished with the test and your test form will be collected immediately.
- Answer each question in the space provided. If you need to continue an answer onto the back of a page, clearly indicate that and label the continuation with the question number.
- If you want partial credit, justify your answers, even when justification is not explicitly required.
- There are 9 questions, some with multiple parts, priced as marked. The maximum score is 100.
- When you have completed the test, sign the pledge at the bottom of this page, sign your fact sheet, and turn in the test and fact sheet.
- Note that failing to return this test and discussing its content with a student who has not taken it are violations of the Honor Code.

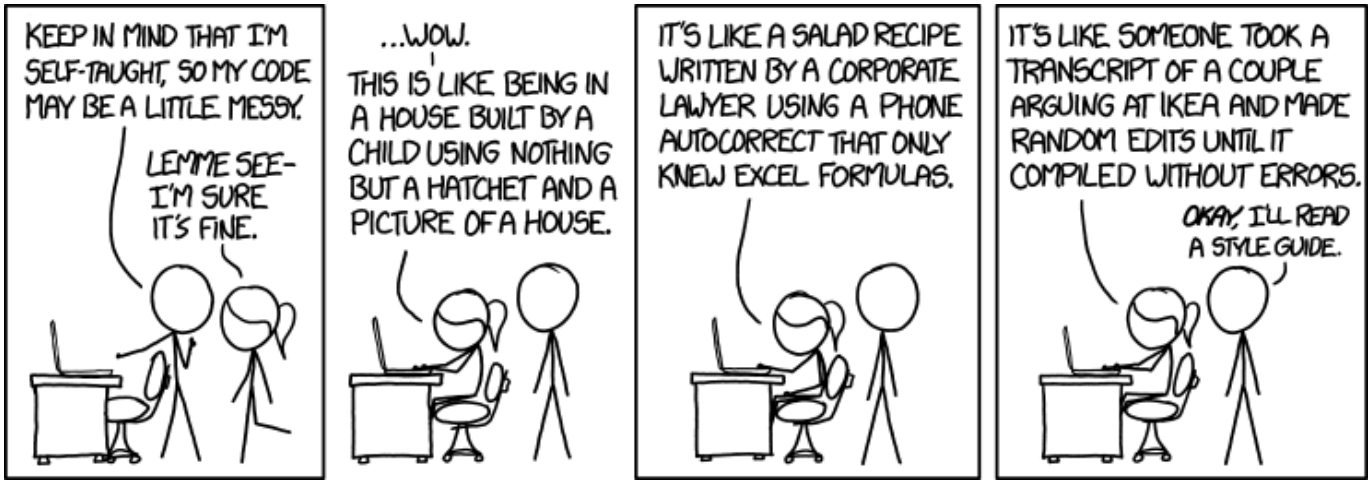
Do not start the test until instructed to do so!

Name Solution
printed

Answers are formatted in dark blue; commentary is in green.

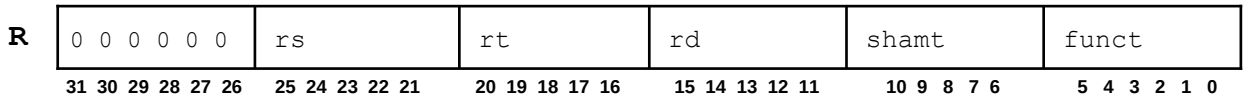
Pledge: On my honor, I have neither given nor received unauthorized aid on this examination.

signed


KEEP IN MIND THAT I'M SELF-TAUGHT, SO MY CODE MAY BE A LITTLE MESSY.
LEMME SEE- I'M SURE IT'S FINE.
...WOW.
THIS IS LIKE BEING IN A HOUSE BUILT BY A CHILD USING NOTHING BUT A HATCHET AND A PICTURE OF A HOUSE.
IT'S LIKE A SALAD RECIPE WRITTEN BY A CORPORATE LAWYER USING A PHONE AUTOCORRECT THAT ONLY KNEW EXCEL FORMULAS.
IT'S LIKE SOMEONE TOOK A TRANSCRIPT OF A COUPLE ARGUING AT IKEA AND MADE RANDOM EDITS UNTIL IT COMPILED WITHOUT ERRORS.
OKAY, I'LL READ A STYLE GUIDE.

xkcd.com

1. Recall the format for the R-type MIPS32 machine instructions:



Consider both the instruction format and the datapath hardware design we've been studying in class.

a) [5 points] Explain why 5 bits are used to represent the register numbers. Be precise.

Answer:
The datapath includes 32 programmable registers (in the register file) and we need 5 bits to number them uniquely from 0 to 31.

b) [5 points] Explain why 5 bits are used to represent the shift amount (`shamt`) field. Be precise.

Answer:
The data registers store 32-bit values, so it only makes sense to perform shifts of 0 to 31 bits, and it takes 5 bits to specify those possible shift amounts.

---

2. [10 points] The RegDst control signal is a don't-care for `sw` instructions. Explain why. Be precise.

Answer:
RegDst determines whether the number specified for the write-to register comes from bits 20:16 or bits 15:11 of the instruction.

Since RegWrite is 0 for sw instructions (so no register is written to), it does not matter which bits we choose to specify that register.

For full credit, you needed to explicitly say "no side effect" (no Reg updates) as RegWrite is 0.

3. Consider the following MIPS32 assembly code:

```
        slt   $t4,   $t1,     $t2
        beq   $t4,   $zero,   ELSE
        sub   $t3,   $t3,     $t1
        j     DONE
ELSE:
        sub   $t3,   $t3,     $t2
DONE:
```

a) **[6 points]** Assume that, immediately before the code given above is executed, `$t1` holds the value 1, `$t2` holds the value 2, and `$t3` holds the value 3. What is the value of `$t3` after the above assembly code finishes?

Answer:

#1: $t4 = $t1 < $t2 ? 1 : 0 = 1
#2: $t4 != 0, so don't jump to ELSE
#3: %t3 = $t3 - $t1 = 3 - 1 = 2

b) **[10 points]** Disassemble the above assembly code and write equivalent C code. Suppose C program variables `A`, `B`, `C` and `D` are mapped to `$t1`, `$t2`, `$t3` and `$t4`, respectively.

Answer:

```
if ( A < B ) {
   C = C - A;
}
else {
   C = C - B;
}
```

In grading, I deduct 3 points for each mistake.

---

4. **[10 points]** Suppose the MIPS32 hardware supports only the instructions add, sub, and, or, slt, lw, sw, beq, and j instructions. Pseudo-instructions give MIPS a richer set of assembly language instructions than those implemented by the hardware. Explain how the following pseudo-instruction can be implemented using only the supported instructions shown above. If you need to use an "extra" register, that is what `$at` is for.

```
addfm $rd, ($rs)     # GPR[rd] = GPR[rd] + Mem[GPR[rs]]
```

Answer:

Since the MIPS architecture requires a value be loaded to a register in order to be operated on, we need to take that into account when translating the addfm pseudo-instruction:

```
lw    $at, ($rs)      # load the value from RAM to $at
add   $rd, $rd, $at   # compute the specified value and place in $rd
```

In grading, I gave 5 points for partially correct answers.

5. **[10points]** Suppose that when a `beq` instruction is executed the control signals RegDst, MemRead, MemtoReg, MemWrite, RegWrite , ALUSrc and Jump are set to values that would be correct if an R-format instruction were being executed. And, suppose that the Branch and ALUOp signals are set properly (for a `beq` instruction). Explain what could go wrong. Be precise and complete.

Answer:
To be correct when executing beq, the control signals should be set as follows:

RegDst = D/C	**this is 1 for R-type instructions, but it doesn't matter**
MemRead = 0	**this is the same for R-type instructions**
MemtoReg = D/C	**this is 0 for R-type instructions, but it doesn't matter**
MemWrite = 0	**this is the same for R-type instructions**
RegWrite = 0	**this is different for R-type instructions**
ALUSrc = 0	**this is the same for R-type instructions**
Jump = 0	**this is the same for R-type instructions**

So, the only concern is the effect of the RegWrite signal being turned on.

If RegWrite is turned on when beq is executed, then the value computed by the ALU (since MemtoReg will be 0) will be written into a register, determined by the high 5 bits of the immediate field of the beq instruction.

So, some unpredictable register will be modified, which is very bad.

---

6. **[12 points]** Suppose we want to support the I-type `addi` instruction:

```
addi $rt, $rs, imm     # GPR[rt] = GPR[rs] + imm
```

Specify the values that should be set for the control signals so the `addi` instruction will execute correctly. (Note: the only change we need to make to the existing SCD hardware is to make the Control unit recognize the opcode for `addi` and set the control signals accordingly.)

Signal	Value	
RegDst	**0**	**need to write to rt, not rd**
Jump	**0**	**same as add**
Branch	**0**	**same as add**
MemRead	**0**	**same as add**
MemtoReg	**0**	**same as add, send ALU result to register file**
MemWrite	**0**	**same as add**
ALUSrc	**1**	**need to add immediate to register value**
RegWrite	**1**	**same as add, write result to register**

None of the settings are don't-cares. For example, memory reads and writes are never safe unless we are sure we are computing an address that is (allegedly) valid, and if either Branch or Jump were set to 1 then we would choose the wrong update for the PC. In grading, I gave more weight to the settings that differed from those for add.

7. **[12 points]** Suppose you want to make certain programs run faster by modifying the integer addition hardware. More precisely, if a program currently spends 30% of its time executing integer addition instructions (on the existing design), you want the execution of that program to take 80% as long on the new design. By what factor must you speed up the integer addition hardware to achieve that goal? (Note: the arithmetic does work out cleanly if you set this up correctly.)

Answer:
Applying Amdahl's Law, letting S be the needed speedup factor, we get:

$$\text{Time}_{\text{after}} = \text{Time}_{\text{unaffected}} + \frac{\text{Time}_{\text{affected}}}{S}$$

Substituting, we get:

$$0.85 \times \text{Time}_{\text{before}} = 0.70 \times \text{Time}_{\text{before}} + \frac{0.30 \times \text{Time}_{\text{before}}}{S}$$

Solving, we get S = 2.0.

The most common error was to not make use of Amdahl's Law, which is the relevant concept.

---

8. **[12 points]** Explain why the preliminary pipeline, as derived in lecture, cannot be guaranteed to execute the following sequence of MIPS32 assembly instructions correctly:

```
and   $t1, $t2, $t3   # 1
lw    $t4, 0($t1)     # 2
lw    $t5, 0($t2)     # 3
sub   $t1, $t2, $t4   # 4
```

Be precise and complete!

Answer:
Instruction #1 writes to $t1, which is read by instruction #2; however, #1 does not actually write to the register until #1 reaches the WB stage of the pipeline. Since #2 reads $t1 while #2 is in the decode stage, and #1 is in the EX stage at that time, #2 will read a stale value from $t1.

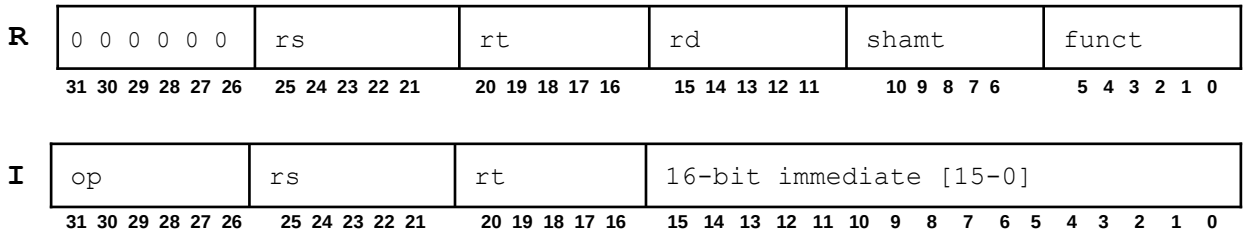
A similar difficulty arises because #2 writes to $t4 and #4 reads from $t4, before the register is updated.

The problems here were all read-after-write data hazards; that is, the issue was that one instruction computed a value that a later instruction needed for input, but the reading instruction was "too close" to the writing instruction.

Since register reads occur in the second half of the ID stage, but register writes occur in the first half of the WB stage, the reading instruction must be at least 3 stages behind the writing instruction in order for the relevant register to be updated in time.

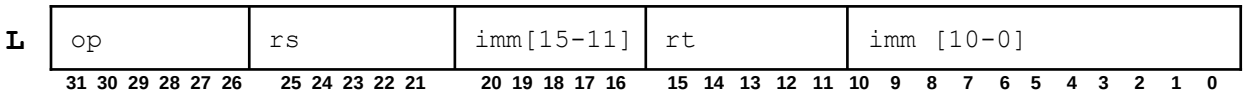
I gave majority credit for identifying and explaining one such hazard; I gave full credit only if you correctly identified and explained all of them.

9. [8 points] Recall the formats for the R-type and I-type MIPS machine instructions:



To specify Write register number, R-type instructions use bits 15-11 (`rd` field), whereas `lw` instruction (which are I-type) uses bits 20-16 (`rt` field). This is the reason why the MUX on the left of Registers is needed in the original SCD design.

To remove this MUX and make the existing hardware simpler, Prof. Hokie proposes a new instruction format (called L-type) for the `lw` instruction as follows, while other instructions in I-type remain the same.



Of course, in order to accommodate this new instruction format, there might have to be some additional changes to the hardware. Explain whether Prof. Hokie's proposal can be applied to simplify the existing hardware design.

Answer:
This change would allow us to eliminate the MUX in question.

But... there would now be two cases for what's sent to the Sign-extend unit:

- **for lw, we need to send bits 20:16 concatenated with bits 10:0**
- **for other I-format instructions, we need to send bits 15:11 (as before)**

Therefore, we would now need a new MUX to select the input to the Sign-extend unit.

So, the change would not really lead to a simpler datapath design.

In fact, it's arguably worse, since if the new MUX selects between 16-bit inputs, that would require more hardware than the old MUX, with 5-bit inputs, needed.

For full credit, you needed to say that the original MUX can be eliminated, say that a new MUX would be necessary, and make at least one additional valid point:

- **the new MUX would be more complex than the old one, unless you used the new MUX to decide what to concatenate with bits 10:0 (either bits 20:16 or bits 15:11)**
- **the new MUX would need a control signal, replacing the control signal for the old MUX**
- **the concatenation of the bits is trivial**

Some answers incorrectly claimed that new shift units and/or adders would be needed; that's simply wrong.