Instructions:

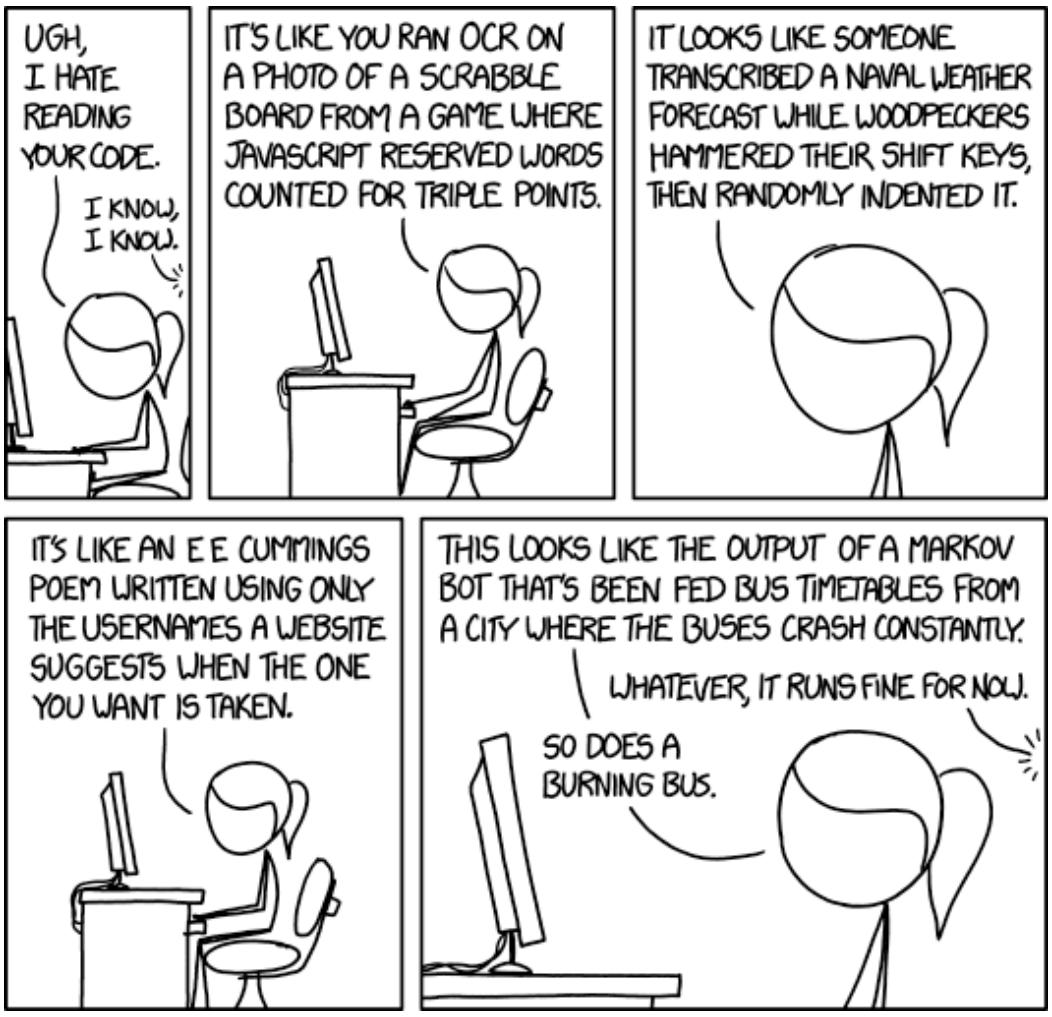
- Print your name in the space provided below.
- This examination is closed book and closed notes, aside from the permitted one-page formula sheet.
- No calculators or other computing devices may be used. The use of any such device will be interpreted as an indication that you are finished with the test and your test form will be collected immediately.
- Answer each question in the space provided. If you need more space to answer a question, you are probably thinking about it the wrong way.
- If you want partial credit, justify your answers, even when justification is not explicitly required.
- There are 6 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

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

_____
signed


UGH, I HATE READING YOUR CODE.
I KNOW, I KNOW.
IT'S LIKE YOU RAN OCR ON A PHOTO OF A SCRABBLE BOARD FROM A GAME WHERE JAVASCRIPT RESERVED WORDS COUNTED FOR TRIPLE POINTS.
IT LOOKS LIKE SOMEONE TRANSCRIBED A NAVAL WEATHER FORECAST WHILE WOODPECKERS HAMMERED THEIR SHIFT KEYS, THEN RANDOMLY INDENTED IT.
IT'S LIKE AN E E CUMMINGS POEM WRITTEN USING ONLY THE USERNAMES A WEBSITE SUGGESTS WHEN THE ONE YOU WANT IS TAKEN.
THIS LOOKS LIKE THE OUTPUT OF A MARKOV BOT THAT'S BEEN FED BUS TIMETABLES FROM A CITY WHERE THE BUSES CRASH CONSTANTLY.
WHATEVER, IT RUNS FINE FOR NOW.
SO DOES A BURNING BUS.

xkcd.com

1. The single-cycle MIPS datapath design used one stage with a clock cycle of 800 ps. The MIPS pipeline had five stages, with a clock cycle of 200 ps. The pipelined version achieved better performance than the single-cycle version by improving instruction throughput. Let's consider some alternative outcomes.

Suppose that the MIPS designers managed (somehow) to produce a feasible pipeline design had resulted in eight stages, with a clock cycle of 150 ps.

a) **[6 points]** How does the latency for a single instruction on this new design compare to the latency on the five-stage pipeline discussed in class?

The latency for the new design is 8 * 150 = 1200 ps, versus 1000 ps for the five-stage design.

Latency is the time from the beginning of the fetch until the completion of execution.

b) **[6 points]** Assuming an infinite sequence of instructions, what speedup will this design achieve when compared to the original, single-cycle design? Show calculations to support your conclusion.

The single-cycle design completes one instruction per 800 ps; the eight-stage design completes one instruction per 150 ps. So, the speedup is

800 ps / 150 ps = 16 / 3 or about 5.33

More precisely:

$$\lim_{n\to\infty} \frac{800n}{1050+150n} = \lim_{n\to\infty} \frac{800}{150} = \frac{16}{3}$$

c) **[6 points]** Assuming an infinite sequence of instructions, what speedup will this design achieve when compared to the five-stage pipeline design discussed in class? Show calculations to support your conclusion.

Versus the five-stage pipeline, the speedup will be 200 ps / 150 ps = 4 / 3 or about 1.33.

More precisely:

$$\lim_{n\to\infty} \frac{800+200n}{1050+150n} = \lim_{n\to\infty} \frac{200}{150} = \frac{4}{3}$$

2. Use Amdahl's Law to justify your answers to the following questions.

a) **[8 points]** Suppose that a program spends 60% of its execution time performing integer arithmetic operations, 10% performing floating point operations, and 30% performing I/O operations. Given the advanced state of hardware for arithmetic computations, it is not likely we can make much improvement with respect to that. However, we may be able to speed up the I/O operations. Theoretically, what is the limit on the speedup that could be achieved by improving the I/O hardware? (Not that we are claiming that limit can be achieved.)

We aren't given the actual execution time, but we can assume any convenient value, so let's say it was 100 seconds. So, the program would spend 70 seconds on operations that are not affected by our improvements, and 30 seconds on operations that are affected:

$$T_{after} = 70 + 30 / (I/O \text{ improvement})$$

The theoretical limit would be if we produced an infinite speedup for I/O operations (so, no, we won't actually achieve the limit). In that case, the time after the improvement is 70 seconds, and so the limiting speedup would be 100 / 70 or about 1.43.

b) **[8 points]** On certain hardware, integer multiplication takes 5 times as many cycles as integer addition, which takes 3 times as many cycles as simple integer operations like shifting and incrementing. A particular program spends 60 seconds on integer multiplication, 90 seconds on integer addition, 20 seconds on simple integer operations, and 30 seconds on other operations.

A code review reveals that each of the integer multiplication operations can actually be replaced with a sequence of two shift operations and a single addition operation. What speedup would the program achieve if the integer multiplications were replaced as described?

Each integer multiplication will be replaced by two simple operations and one addition. If we say that one simple operation takes 1 time unit, then an addition takes 3 time units and a multiplication takes 15 units. The substitution would replace each multiplication instruction with a sequence taking 5 time units, 1/3 as long.

So, the substitution would effectively speed multiplication operations up by a factor of 3. By Amdahl's Law, the execution time of the revised program would be:

T_after = T_unaffected + T_affected / SpeedupFactor = 140 + 60 / 3 = 160 seconds

The speedup would be T_old / T_new = 200 / 160 = 1.25

The key was to determine what improvement factor would be achieved if we made the instruction substitutions that were described; it resulted in a factor of 3 improvement for multiplication instructions.

3. [14 points] Pseudo-instructions give MIPS a richer set of assembly language instructions than those supported directly by the hardware. Write an implementation for the following pseudo-instruction, using only the MIPS32 instructions listed at the bottom of this page (those are sufficient). You must include comments explaining the logic of your solution. If you need to use any "extra" registers besides those used in the instruction itself, you may use the the t-registers `$t0`, `$t1`, etc.

```
blt_m  ($rs), ($rt), Label
               # if ( Mem[GPR[rs]] < Mem[GPR[rt]])
               #    branch to Label
               # else
               #    don't branch
               #
               # $rs and $rt should not be modified
```

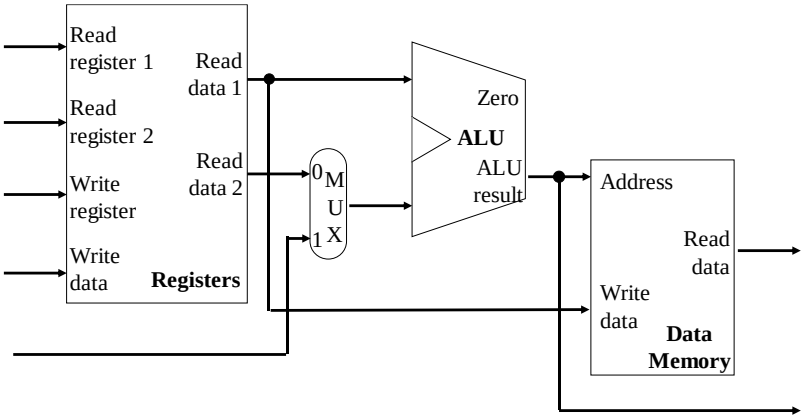
The instruction requires fetching two values from memory, seeing if the first is less than the second, and jumping to Label if it is:

```
lw   $t0, ($rs)          # load left operand to $t0
lw   $t1, ($rt)          # load right operand to $t1

slt  $t2, $t0, $t1       # $t2 = 1 iff $t0 < $t1

bne  $t2, $zero, Label   # jump iff $t0 < $t1
```

```
add   rd, rs, rt       # GPR[rd] <-- GPR[rs] + GPR[rt]
addi  rd, rs, imm16    # GPR[rd] <-- GPR[rs] + imm16     (sign extended)
and   rd, rs, rt       # GPR[rd] <-- GPR[rs] AND GPR[rt] (bitwise)
beq   rs, rt, label    # if GPR[rs] == GPR[rt], jump to label
bne   rs, rt, label    # if GPR[rs] != GPR[rt], jump to label
lw    rt, imm16(rs)    # GPR[rt] = Mem[GPR[rt] + imm16]
sw    rt, imm16(rs)    # Mem[GPR[rt] + imm16] GPR[rt] =
or    rd, rs, rt       # GPR[rd] <-- GPR[rs] OR GPR[rt]  (bitwise)
sub   rd, rs, rt       # GPR[rd] <-- GPR[rs] - GPR[rt]
sll   rd, rt, sa       # GPR[rd] <-- GPR[rs] << sa       (logical shift)
slt   rd, rs, rt       # GPR[rd] <-- (GPR[rs] < GPR[rt] ? 1 : 0)
```

(**imm16** denotes a 16-bit immediate value)

4. **[12 points]** Suppose that a buggy implementation of the MIPS data path miswires the selection of the source for the Write data input to the Data Memory unit as shown:



Everything else in the datapath is implemented as shown on the diagram in the Supplement.

Suppose that, initially, `$t0=0x1000`, `$t1=0x2000`, and `$t2=0x3000`. Consider the execution of the following code in this buggy datapath. Determine the final values of the `$t0`, `$t1`, and `$t2` registers.

```
sw    $t1, 0x0($t0)          # Mem[0x1000] = 0x1000
sw    $t2, 0x1000($t0)       # Mem[0x2000] = 0x1000
addi  $t0, $t0, 0x1000       # $t0 = 0x2000
lw    $t1, 0x0($t0)          # $t1 = Mem[0x2000] = 0x1000
add   $t2, $t1, $t2          # $t2 = 0x1000 + 0x3000 = 0x4000
```

Register	Final Value
`$t0`	**`0x2000`**
`$t1`	**`0x1000`**
`$t2`	**`0x4000`**

5. [15 points] Suppose the following sequence of instructions was executed on the preliminary pipeline design shown on the Supplement, which has no hardware to detect or handle hazards:

```
sub   $t4, $t1, $t2   # 1

slt   $s0, $t4, $t2   # 2

sw    $t4, 0($t2)     # 3

add   $t2, $t4, $s0   # 4

or    $t1, $t2, $t4   # 5

lw    $t2, 0($t2)     # 6
```

Due to the limitations of the preliminary datapath design, a number of different logical errors are likely to occur. These may include possibly incorrect values being used as input to an instruction, known as *read-after-write* hazards.

Identify each of the read-after-write hazards that occur in the given sequence of instructions. For each, clearly identify the writing instruction, the reading instruction, and the register that is involved in the hazard.

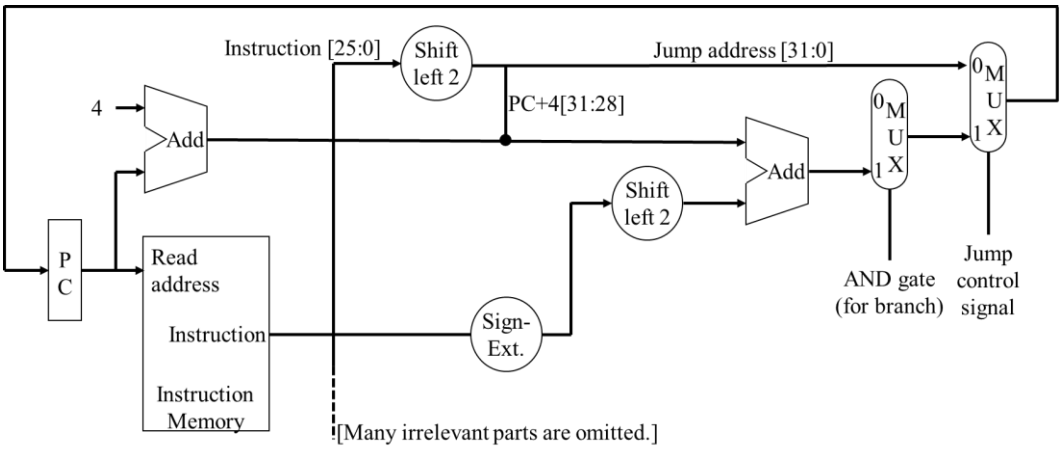
Writer#	Reader#	Register(s)	
#3	#5	$t2	<-- example, may be wrong
#1	**#2**	**$t4**	
#1	**#3**	**$t4**	
#2	**#4**	**$s0**	
#4	**#5**	**$t2**	
#4	**#6**	**$t2**	

6. Suppose we would like to design the new MIPS64 architecture using 64-bit addresses, 64-bit instructions, and 64 (general-purpose integer) registers. We will keep the same number of instructions as in the original MIPS32 architecture, so that we can use 6-bit opcodes, and we still require that machine instructions be aligned on 8-byte multiples.

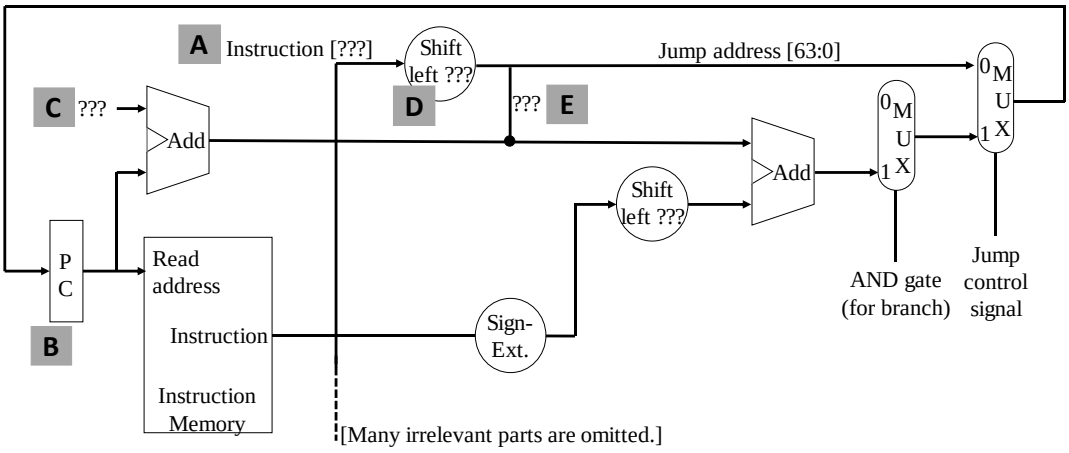
a) **[5 points]** For I-format instructions, how many bits could we use for the immediate field? Explain.

Register numbers take 6 bits (2^6 registers now), opcodes still take 6 bits, so with 64-bit instructions, we have 46 bits for the immediate field.

The existing MIPS32 datapath contains the following hardware elements to support the `beq` instruction:



The new MIPS64 datapath will involve essentially similar hardware, from a logical point of view, but some details will be different. For example, the change to the item labelled **A** in the diagram below was explained in your answer to part **a)**.



b) **[5 points]** Explain how item **B**, in the new design, differs from the corresponding element in the original MIPS32 design, and why. (The MIPS32 diagram may provide hints.)

The PC holds the instruction that was just fetched, so it must now be a 64-bit register.

c) **[5 points]** Explain how item **C**, in the new design, differs from the corresponding element in the original MIPS32 design, and why.

This specifies the value to be added to the PC if we don't branch. Since instructions are now 8 bytes wide, this must now be 8.

d) **[5 points]** What is the logical connection between **C** and **D?** (The answer would be the same for both the old and the new designs.)

Since instructions are 8 bytes long, they are stored at addresses that are multiples of 8.

D shifts the immediate to compensate for the bits that would have been dropped from the offset between PC + 8 and the branch target address. Since instruction addresses always have 000 for their low bits, so we will drop 3 bits from the offset, and the shifter must now shift left by 3 bits.

e) **[5 points]** Explain what item **E** should be, in the new design, and why. (The MIPS32 diagram may provide hints.)

The immediate field for jump is now 58 bits wide (64 bits minus the opcode width). The shifter produces a 61-bit value.

So, E completes the 64-bit address by supplying bits 63:61 from PC + 8.