Instructions:

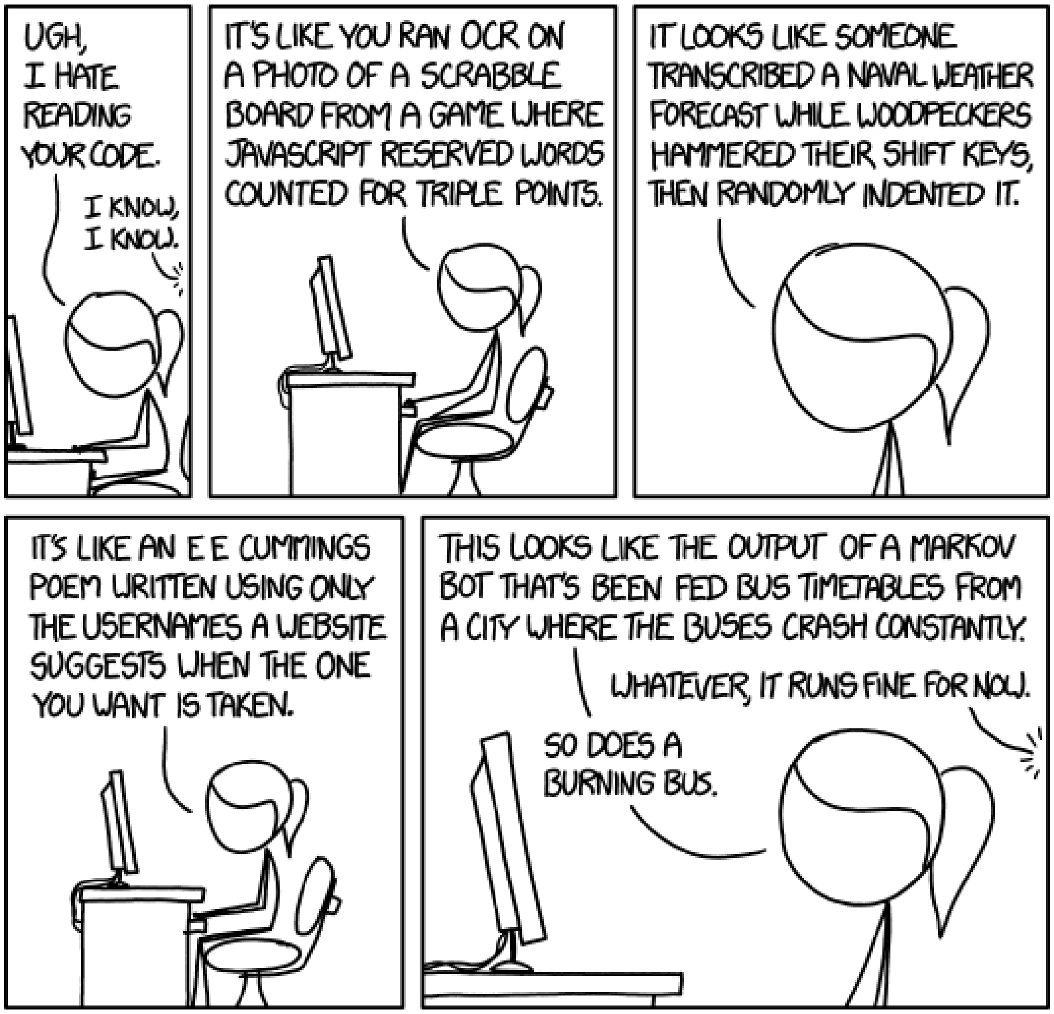
- Print your name in the space provided below.
- This examination is closed book and closed notes, aside from the permitted one-page formula sheet.
- No calculators or other computing devices may be used. The use of any such device will be interpreted as an indication that you are finished with the test 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 pipeline achieved better performance by improving instruction throughput. Let's consider some alternative outcomes.

Suppose that the MIPS pipeline design had resulted in six 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 original, single-cycle design and on the five-stage pipeline discussed in class?

Answer:
With the original single-cycle design, the clock cycle was 800ps and an instruction spent one clock cycle in the datapath. With the five-stage pipeline, the clock cycle was 200ps, and an instruction spent 5 cycles in the pipeline (ignoring stalls), for a latency of 1000ps.

With this design, the clock cycle is 150ps and an instruction spends six cycles in the pipeline (ignoring stalls), for a latency of 900ps.

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.

Answer:
With both designs, an instruction will complete at the end of each clock cycle, ideally. But now the cycle is only 150ps, so the speedup is

$$\text{Speedup} = T_{old}/T_{new} = 800\text{ps}/150\text{ps} = 16/3$$

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.

Answer:
Now the speedup is

$$\text{Speedup} = T_{old}/T4 = 200\text{ps}/150\text{ps} = 4/3$$

2. Suppose a program spends 20% of its time executing integer addition instructions and 30% of its time executing integer multiplication instructions; you want to improve the performance of that program by modifying the hardware for those integer operations. Justify your answers to the following questions by using Amdahl's Law. The parts of the question are independent. For each part, show computations to support your conclusion.

a) **[10 points]** How much faster would the program execute if you made integer addition 4 times faster and integer multiplication 3 times faster?

Answer:

Let $T_{before} = 1$ (without specifying any particular time unit). Then:

$$\begin{aligned} T_{after} &= 0.50 * T_{before} + 0.20 * T_{before} / 4 + 0.30 * T_{before} / 3 \\ &= (0.50 + 0.05 + 0.10) * T_{before} \\ &= 0.65 * T_{before} \end{aligned}$$

So, after the improvement, the program takes 65% as long as before, and the speedup is 1/.65 or 20/13 (about 1.55).

b) **[8 points]** Suppose the program currently executes in 150 seconds, with a given input set, and your goal is to reduce its execution time on the same input set to 120 seconds. Assuming your only option is to speed up the execution of integer multiplication instructions, how much faster would they have to execute in order to achieve your goal?

Answer:

$$T_{after} = 0.70 * 150 + 0.30 * 150 / x = 120$$

and that implies that $x = 3$. So you'd need to make integer multiplication 3 times faster.

3. Pseudo-instructions give MIPS a richer set of assembly language instructions than those supported directly by the hardware. Explain how each of the following pseudo-instruction can be implemented using only the MIPS32 instructions listed at the bottom of this page (those are sufficient). If you need to use any "extra" registers, you may use the t-registers, `$t0`, `$t1`, and so forth.

a) **[8 points]** `sge  $rd, $rs, $rt`

```
        # GPR[rd] = ( GPR[rs] >= GPR[rt] ? 1 : 0 )
```

Answer:

There are many solutions. Mine uses slt to set $rd to the wrong value, and then flips the low bit to correct it:

```
slt    $rd, $rs, $rt      # $rd = $rs < $rt ? 1 : 0
ori    $t0, $zero, 1      # $t0 = 1
sub    $rd, $t0, $rd      # $rd = 1 - $rd
```

b) **[8 points]** `swapifevensum  $rs, $rt`

```
        # if ( GPR[rs] + GPR[rt] is even ) swap GPR[rs] and GPR[rt]
```

Answer:

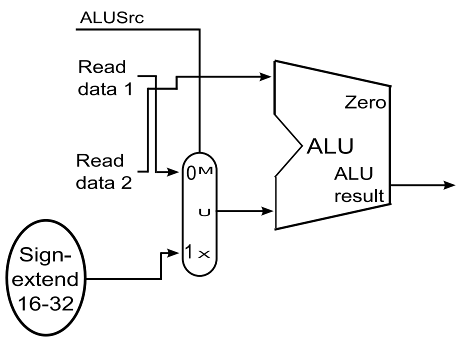
Again, there are many solutions. Here's one:

```
    add    $t0, $rs, $rt      # $t0 = $rs + $rt
    andi   $t0, $t0, 1        # set bits 31:1 of sum to 0; result shows low bit of sum

    bne    $t0, zero, skip    # sum is odd, so no need to swap

    or     $t0, $zero, $rs    # $t0 = $rs
    or     $rs, $zero, $rt    # $rs = $rt
    or     $rt, $zero, $t0    # $rt = $rs
skip:
```

```
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)
andi  rd, rs, imm16    # GPR[rd] <-- GPR[rs] OR 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
or    rd, rs, rt       # GPR[rd] <-- GPR[rs] OR GPR[rt]  (bitwise)
ori   rd, rs, imm16    # GPR[rd] <-- GPR[rs] OR imm16    (bitwise, sign extended)
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. Suppose that a buggy implementation of the MIPS datapath switches the use of Read data 1 and Read data 2:



Everything else in the datapath operates normally.

a) **[8 points]** Describe, in detail, how this error would affect the execution of the following instruction:

```
lw  $t0, 0($t1)
```

Answer:
This change means that the ALU will calculate the address as $t0 (instead of the intended $t1). That results in reading a value from the wrong address (and possibly a segfault).

However, this does not affect the choice of the destination register, so that's still $t0.

So, the instruction reads from the wrong address, but writes to the correct register.

A common error was to say that the wrong register would be written to; but the write-to register number is set by other hardware, and is not altered by this change.

b) **[8 points]** Describe, in detail, how this error would affect the execution of the following instruction:

```
beq  $s0, $s1, target
```

Answer:
This change simply swaps the left and right operands to the ALU. Since beq only depends on whether the difference of the two values is zero, that has no effect on beq at all.

5. **[15 points]** Suppose the following sequence of instructions was executed on the preliminary pipeline design derived in class, and shown on the supplement:

```
add   $t2, $t1, $t4   # 1
lw    $t3, 0($t2)     # 2
sub   $t2, $t1, $t3   # 3
lw    $t1, 0($t3)     # 4
sw    $t2, 0($t1)     # 5
```

Describe all of the logical errors that would occur. For each logical error, clearly identify the instruction whose execution would be incorrect, and identify the register(s), if any, that are involved in the error.

```
Instruction    Description of error
-------------------------------------------------------------------
```

#2 reads $t2 before it's updated by #1;
accesses the wrong address

#3 reads $t3 before it's updated by #2;
computes the wrong difference

#4 reads $t3 before it's updated (probably incorrectly) by #3;
accesses the wrong address

#5 reads $t2 before it's updated by #3
reads $t1 before it's updated by #4

The almost-universal error was to overlook the fact that there are two error conditions associated with instruction #5.

6. For this problem, you will design a modification of the single-cycle datapath to add support for the following instruction:

```
ble   rs, rt, offset    # conditional branch if rs <= rt
                        # PC <-- (rs <= rt ? PC + 4 + offset << 2)
                        #                  : PC + 4)
```

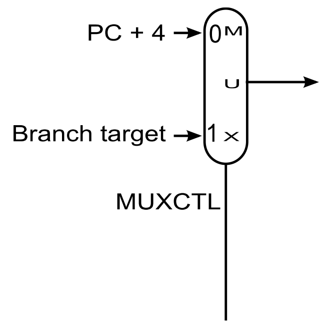
We will rename the Branch control signal BEQ, and add a new control signal BLE (equals 1 iff the current instruction is `ble`). We will retain the Zero control signal unchanged, and add a new Positive signal from the ALU that is set to 1 if and only if the last result computed was positive (greater than zero). We will call the control line for the MUX that selects the address for the next instruction MUXCTL.

a) [12 points] Write a truth table showing the relationship between BEQ, BLE, Zero, Positive and MUXCTL.

BEQ	BLE	Zero	Positive	MUXCTL
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	▨
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
0	1	1	1	▨
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	▨
1	1	0	0	▨
1	1	0	1	▨
1	1	1	0	▨
1	1	1	1	▨

The shaded boxes indicate don't care conditions, corresponding to forbidden input combinations.

BEQ and BLE can never both be 1, nor can Zero and Positive.



b) [8 points] Draw a circuit showing the control logic for the MUX that selects the address for the next instruction. Gate diagrams are shown on the supplement.

BEQ

BLE

Zero

Positive

There are several ways to accomplish this. The simplest depends on realizing that whether a ble is taken really just depends on BLE and ~Positive, and not on Zero at all.