Instructions:

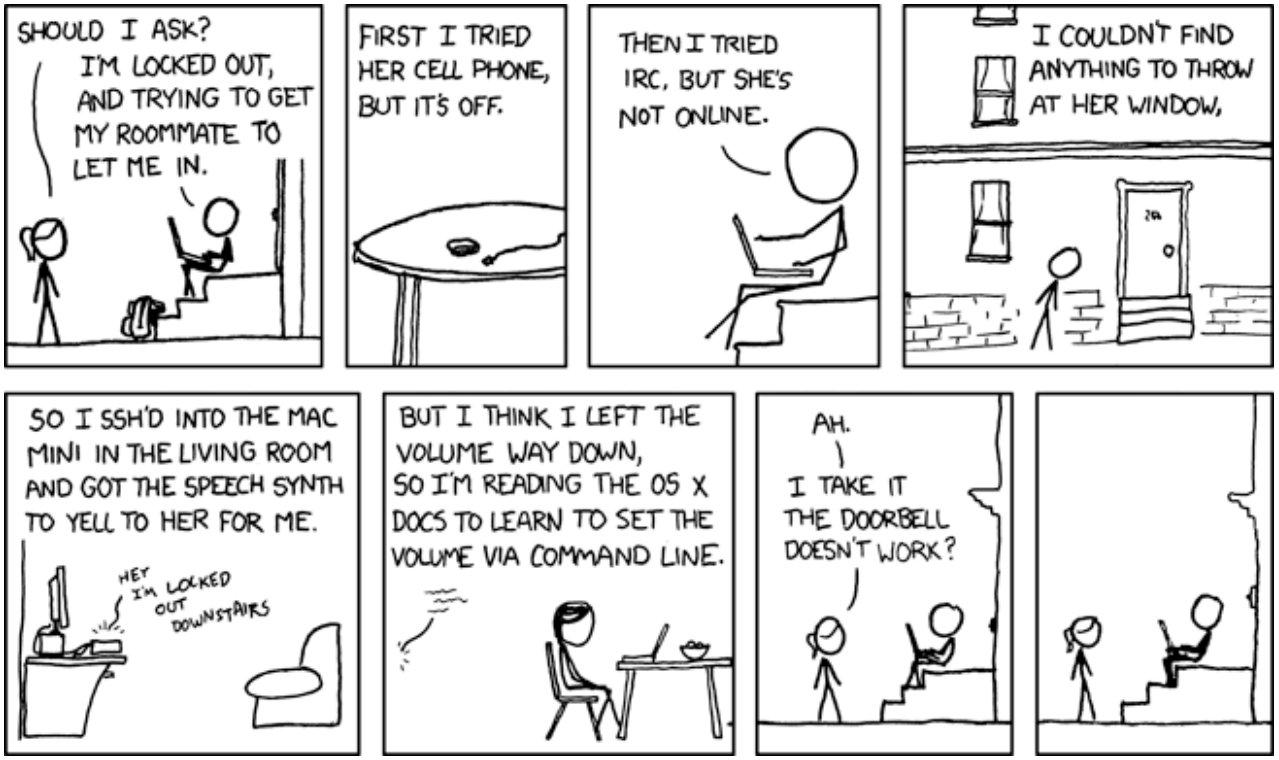
- Print your name in the space provided below.
- This examination is closed book and closed notes, aside from the permitted one-page formula sheet.
- No calculators or other computing devices may be used. The use of any such device will be interpreted as an indication that you are finished with the test 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



xkcd.com

Answers to questions are shown in blue.

Commentary on the questions/answers is shown in green.

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 160 ps.

a) **[8 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.

Key fact: once n - 1 stages of the pipeline are full, an instruction will be completed on each clock cycle (ignoring stalls).

So, for the pipeline described here, the time to execute a sequence of N instructions would be given by:

$$\text{Time} = 800 + 160N$$

And so, the speedup versus the single-cycle design would be:

$$\text{Speedup} = \frac{800N}{800 + 160N}$$

And, as N goes to infinity, we see the limiting speedup will be 800/160 = 5.

b) **[8 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.

And so, the speedup versus the textbook pipeline design would be:

$$\text{Speedup} = \frac{800 + 200N}{800 + 160N}$$

And, as N goes to infinity, we see the limiting speedup will be 200/160 = 5/4 = 1.25.

2. Suppose a program spends 30% of its time executing integer addition instructions and 40% 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) **[8 points]** How much faster would the program execute if you made integer addition 3 times faster and integer multiplication 2 times faster?

Applying Amdahl's Law, and letting the current time taken be T:

$$T_{after} = 0.30 * T + \frac{0.30T}{3} + \frac{0.40T}{2}$$
$$= 0.30 * T + 0.10T + 0.20T = 0.60T$$

So, after the revision, the program would take 60% as long (speedup would be 100/60).

b) **[8 points]** Suppose the program currently executes in 300 seconds, with a given input set, and your goal is to reduce its execution time on the same input set to 220 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?

Applying Amdahl's Law:

$$220 = 0.60 * 300 + \frac{0.40 * 300}{X}$$

Solving for X, we get 3. So, we would have to make multiplication three times as fast.

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] `movnz  $rd, $rs, $rt`
`# if GPR[rs] != 0 then GPR[rd] = GPR[rt]`

```
        beq   $rs, $zero, skip
        add   $rd, $rt, $zero
skip:
```

b) [8 points] `modby  $rd, $rs, exp`
`# GPR[rd] = GPR[rs] % 2^exp (2 to the power exp)`

N % 2^k yields the bits k-1 to 0 of N; so the task is to create a mask that would zero out the bits 31:k of N, and then apply that mask to N.

You also need to note that you only have a left-shift instruction, and that shifting to the left will shift in 0s at the bottom.

```
nor   $t0, $zero, $zero    # put FFFFFFFF into $t0

sll   $t0, $t0, exp        # zero out exp low bits of $t0

nor   $t0, $t0, $t0        # flip the bits of $t0, so we have 0s for the
                           # high bits and 1s for the low bits

and   $rd, $rs, $t0        # apply mask to zero high bits of $rd
```

There are a number of ways to create the mask; here's one alternative:

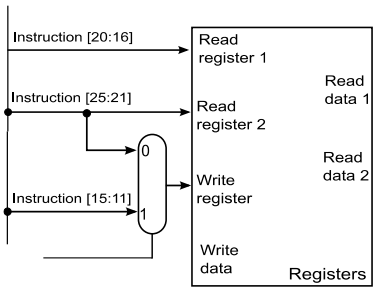
```
lui   $t0, 0xFFFF           # t0 = 0xFFFF0000
addi  $t0, $t0, 0xFFFF      # t0 = 0xFFFFFFFF
```

Then proceed as shown above.

```
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
lui   rt, imm16         # GPR[rt] <-- imm16 << 16          (load upper 2 bytes)
or    rd, rs, rt        # GPR[rd] <-- GPR[rs] OR GPR[rt]   (bitwise)
nor   rd, rs, rt        # GPR[rd] <-- GPR[rs] NOR GPR[rt]  (bitwise)
sub   rd, rs, rt        # GPR[rd] <-- GPR[rs] - GPR[rt]
sll   rd, rt, sa        # GPR[rd] <-- GPR[rs] << sa        (logical shift)
```

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

4. Suppose that a buggy implementation of the MIPS datapath switches the use of bits 25:21 and 20:16:



Everything else in the datapath operates normally.

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

```
sub  $t0, $t1, $t2
```

The error effectively swaps the rs and rt registers for the instruction, and doesn't change anything else. Therefore, this will perform the computation:

$t0 = $t2 - $t1

So, this will set $t0 to the negative of the expected result.

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

```
sw   $s0, 0($s1)
```

Effectively, this will perform the computation:

sw $s1, 0($s0)

Therefore, the instruction will write to $s1 to memory instead of $s0, and (attempt to) write to the address $s0 instead of to $s1.

So, there may well be a segfault on the memory access, and if not, the wrong register will be written to memory.

5. [12 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
sub   $t1, $t2, $t4   # 2
slt   $t3, $t3, $t1   # 3
sw    $t3, ($t1)      # 4
lw    $t3, 0($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	Registers
#2 sub	**$t2 (read before #1 actually updates it)**
#3 slt	**$t1 (read before #2 actually updates it; sets $t1 incorrectly)**
#4 sw	**$t3 (read before #3 actually updates it; sets memory incorrectly)**
	$t1 (read before #2 actually updates it)
#5 lw	**$t1 (was set incorrectly by the slt instruction)**

There are other dependencies, but in all cases the reading instruction is at least 3 cycles (stages) behind the writing instruction, so there's no read-after-write hazard, nor are there any other logical issues.

A common point of confusion was over the fact that sw READS from $t3 and that lw WRITES to $t3. That's why lw has no problem with respect to $t3; it's overwriting what's there anyway, and never accesses the previous value.

And, lw has an issue with $t1 because it was set incorrectly by an earlier instruction; note the question asked you to describe all the logical errors that would occur, not just identify data hazards.

6. For this problem, you will design a modification of the single-cycle datapath to add support for the branch-on-not-equal (`bne`) instruction:

```
bne   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 BNE (equals 1 iff the current instruction is `bne`). We will retain the Zero control signal unchanged, and call the control line for the MUX that selects the address for the next instruction MUXCTL.

a) **[8 points]** Write a truth table showing the relationship between BEQ, BNE, Zero and MUXCTL.

BEQ	BNE	Zero	MUXCTL
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	?
1	1	1	?

All six rows shown here in blue matter, since they all represent valid input states. The two rows shown in green represent invalid states, since BEQ and BNE can never both be 1. It's OK to include those rows, but not to indicate that MUXCTL should be a specific value then.

b) **[8 points]** Derive a simplified Boolean expression for MUXCTL.

```
MUXCTL = ~BEQ * BNE * ~Zero  +  BEQ * ~BNE * Zero
```

I graded this on consistency with part a). The expression shown here does not simplify unless you take the two invalid input states into account, in which case it simplifies to:

```
MUXCTL = BNE * ~Zero  +  BEQ * Zero
```

c) **[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.

I graded this on consistency with part b). I'll insert a circuit diagram later.