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 5 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 and turn in the test.
- If you brought a fact sheet to the test, write your name on it and turn it in with the test.
- Note that either failing to return this test, or discussing its content with a student who has not taken it is a violation of the Honor Code.

Do not start the test until instructed to do so!

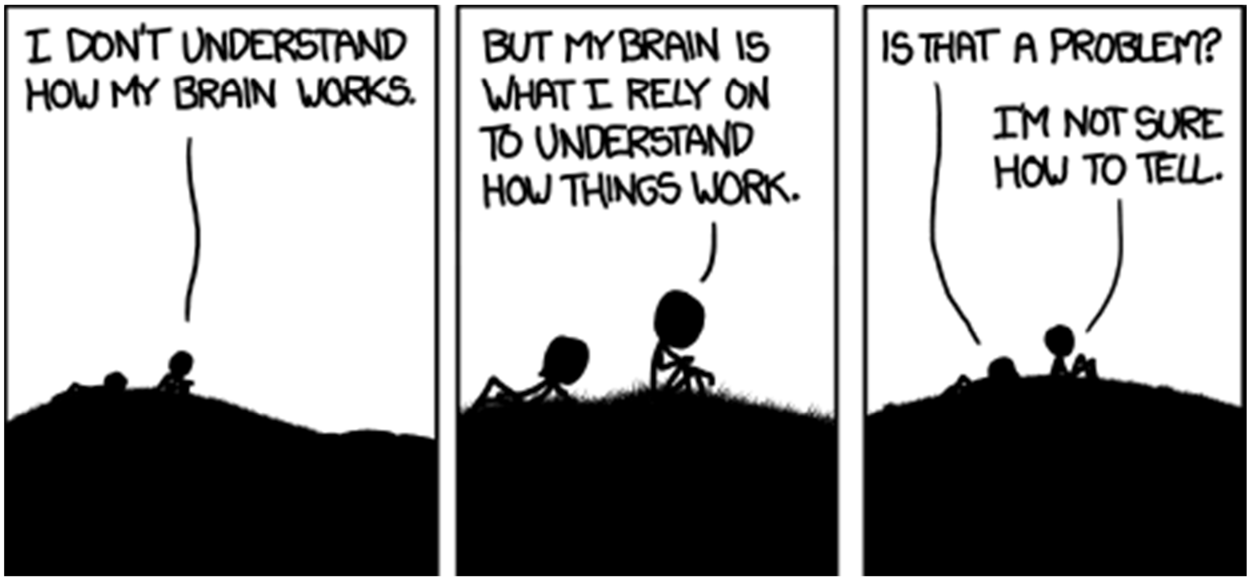
Name Solution

printed

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

____

signed


I DON'T UNDERSTAND HOW MY BRAIN WORKS.
BUT MY BRAIN IS WHAT I RELY ON TO UNDERSTAND HOW THINGS WORK.
IS THAT A PROBLEM?
I'M NOT SURE HOW TO TELL.

xkcd.com

1. [12 points] An ISA includes three classes of instructions: type I taking 1 clock cycle, type II taking 2 clock cycles and type III taking 5 clock cycles. The execution of a particular program, running on a machine with a clock rate of 2.4 GHz, will require 30% type I instructions, 50% type II instructions, and 20% type III instructions. What is the average CPI for this program?

Avg CPI = .30 * 1 + .50 * 2 + .20 * 5 = .30 + 1.0 + 1.0 = 2.3

2. Suppose a program consists of 30% floating point multiply instructions, 20% floating point divide instructions, and the remaining 50% are other instructions. Floating point division, floating point multiplication, and each of the other instructions have the same CPI.

a) [8 points] What speedup can be achieved, for that program, if the execution of a floating point multiplication instruction is made 3/2 times faster?

```
Amdahl's Law says T_after = T_unaffected + T_affected / speedup
                          = 0.70 * T_before + 0.30 * T_before / (3/2)
                          = (.70 + .20) * T_before
                          = 0.90 * T_before
```

So, speedup = T_before / T_after = 1 / 0.90 = 10/9.

b) [10 points] What speedup can be achieved, for the same program, if the execution of a floating point multiplication is made 5/4 faster, and the execution of a floating point division instruction is made 4/3 faster?

Applying Amdahl's Law again:

```
T_after = 0.50 * T_before + 0.30 * T_before / (5/4) + 0.20 * T_before / (4/3)
        = (.50 + .24 + .15) * T_before
        = .89 * T_before
```

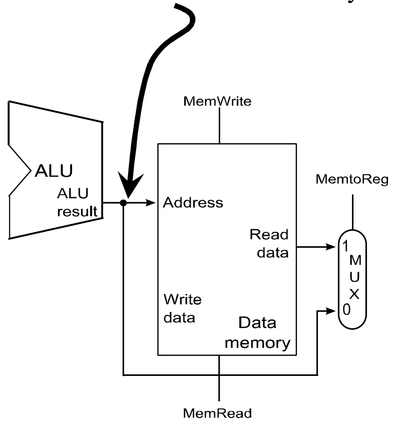
So, speedup = 1 / .89 = 100 / 89.

Aside: it's incorrect to say that because the execution time was reduced by 11% (100 – 89) that the speedup is 11%. (Although it's close to that.) Suppose we reduced the execution time from 100 to 50; then the speedup would be 2, not 50%.

3. These questions refer to the single-cycle MIPS32 datapath (full diagram supplied with the test). Recall that this datapath supports the following instructions: `add`, `sub`, `and`, `or`, `slt`, `lw`, `sw`, `beq` and `j`.

a) [10 points] For which supported instruction(s) would the data line from ALU result to Address below be necessary? Give a precise explanation why.

This line passes a value computed in the ALU to the Address port on the Data memory unit; the only instructions that use the Address port on the Data memory unit are `lw` and `sw`.



Aside: note that the question clearly asks about the line from ALU result to Address; the other branch from ALU result to the MUX is irrelevant.

b) [8 points] The MemRead control signal must be set to 0 unless a `lw` instruction is being executed. Explain why, for example, an `add` instruction could lead to a runtime error if MemRead was set to 1. (Hint: think segfault!)

Recall that MemRead == 1 iff the value at Address is read.

The key is thing to remember is that processes are not allowed to read from just any address; we've all seen enough C code that produces segmentation faults for just that reason.

Consider executing an add instruction that computes a sum of 0. If MemRead == 1, the Data memory will attempt to read the value at address 0 --- a NULL pointer dereference.

Since we cannot anticipate what value will be emitted by the ALU when executing other instructions, we cannot take the chance of setting MemRead to 1 in that situation.

That said, setting MemRead to 1 during a sw instruction will not cause a segmentation fault, but it could affect performance.

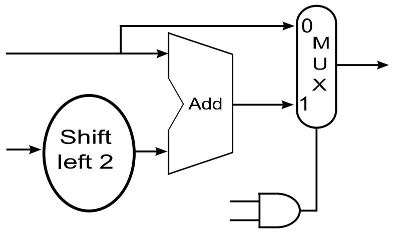
Aside: for full credit, I expected your answer to address precisely how an segfault could be triggered. Just saying that the add instruction would produce a value that's in some sense random doesn't make the problem clear. The simplest way to explain this clearly is to show how the ALU could yield an address that's guaranteed to produce a segfault, namely 0.

4. The hardware shown below was inserted when the datapath was extended for a specific instruction.

a) [8 points] Identify the instruction.

This hardware is used to compute the branch target address, and determine which address to pass back to the PC.

That's relevant only to `beq`.



When executing that instruction, a portion of the instruction is sign-extended and then shifted left 2 bits. Shifting left 2 bits inserts two 0 bits at the low end of the value and chops two bits from the high end of the value.

b) [8 points] What is gained by applying the left shift to the value? Be precise.

The left shift is applied to the (sign-extended) immediate from the beq instruction; that effectively multiplies the value by 4. Since that value is added to PC + 4, this effectively makes it possible to branch to a target instruction that's farther from the beq instruction.

Aside: 2 points for saying the shift multiplies bye immediate by 4; 4 points for saying that the shift means we effectively have an 18-bit immediate instead of a 16-bit immediate; but neither of those statements describes what's GAINED.

c) [8 points] Why should we not be concerned about the two bits that are chopped from the high end of the sign-extended value? Be precise.

When the 16-bit immediate is sign-extended, we tack 16 copies of its (formerly) high bit onto the front of the immediate. That doesn't actually change the value that's represented (recall how 2's complement representation works); so we have an extended value of the following form:

$$b_{15}\ldots b_{15}b_{15}b_{14}b_{13}\ldots b_1b_0$$

where b_{15} is either 0 or 1. Now, if b_{15} = 0, we have a nonnegative value, which is just being represented in base-2, and chopping leading 0s obviously doesn't change the value. If b_{15} = 1, we have a negative value, in fact it's the negative of the value represented by flipping all the bits after the rightmost 1. If we chop off two leading 1s (from the representation of the negative value), we still have a representation of the same negative value (consider the effect of the bit-flipping described above).

When we chop off 2 high bits, we're just chopping off redundant copies of the (former) high bit, and that doesn't change the value that's represented.

Aside: there were a lot of confused (or confusing) answers to this question. Because of that, I was picky about awarding full credit. I expected a clear explanation of why removing the two high bits doesn't really alter the value that's represented; and that required saying something clear about the 2's complement representation that's used.

d) [8 points] In order to correctly execute that instruction, the control signals Zero and Branch were added to the design of the datapath. Explain why neither of those control signals would be sufficient by itself.

The Zero signal tells us whether the ALU just computed a result equaling zero; that has nothing to do with what the current instruction is.

The Branch signal tells us whether the current instruction is beq, but nothing about whether the two relevant registers are equal.

We jump to the branch target instruction iff the current is beq AND the two registers involved are equal. So, we need both signals.

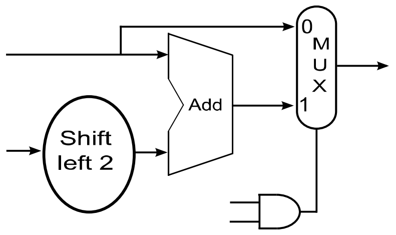
Aside: the question was about why neither of the two signals was sufficient by itself. A lot of answers explained how the datapath design uses the two bits. That doesn't answer the question of why both are necessary. A correct answer needed to explain what necessary information was unavailable if we had only the Zero signal or only the Branch signal.

5. [20 points] The MIPS32 instruction set includes the branch-on-not-equal (I-format) instruction:

```
bne   $rs, $rt, Imm      # if GPR[rt] != GPR[rs] then
                         #    PC = PC + 4 + ( Imm << 2 )
```

The given single-cycle datapath can be extended to support the branch-on-not-equal instruction, but that requires:

- replacing the Branch control signal with two signals
- modifying the hardware shown below to make proper use of those signals



Explain in detail how the changes described above would be made to provide support for branch-on-not-equal.

Explain how the control signals will be used, and explain any changes you would make to the hardware shown above.

You may alter, replace, or remove any parts of the hardware shown above. The best way to indicate your design decisions would be to redraw the diagram given above, showing your changes with appropriate labels and explanations.

When executing a bne instruction, we want to jump to the branch target instruction iff the two registers in question are not equal.

More precisely, we need two new signals (replacing Branch):

- BEQ set to 1 iff the current instruction is beq
- BNE set to 1 iff the current instruction is bne

The existing Zero signal is sufficient to tell us whether the two registers were equal.

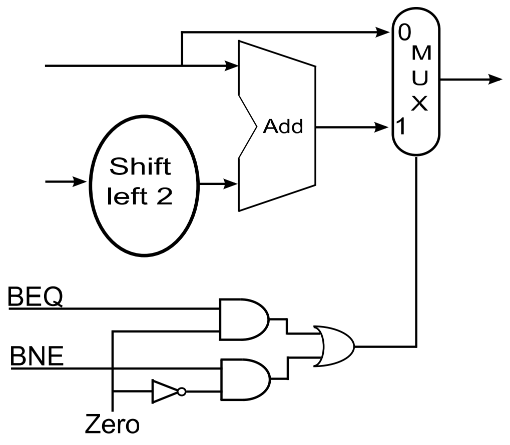
The branch target address is calculated the same way for beq and bne.

So, we need to set the control signal for the existing MUX using the following Boolean logic:

(BEQ && Zero) || (BNE && ~Zero)

Aside: you must have added a new control signal; otherwise there's no way to tell a beq from a bne. That's why the old Branch signal isn't adequate.

There are a number of alternate solutions. Some make use of a second MUX, some of a MUX with three inputs, some with a more complex arrangement of logic gates.



0
M
U
X
1
Add
Shift
left 2
BEQ
BNE
Zero