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.

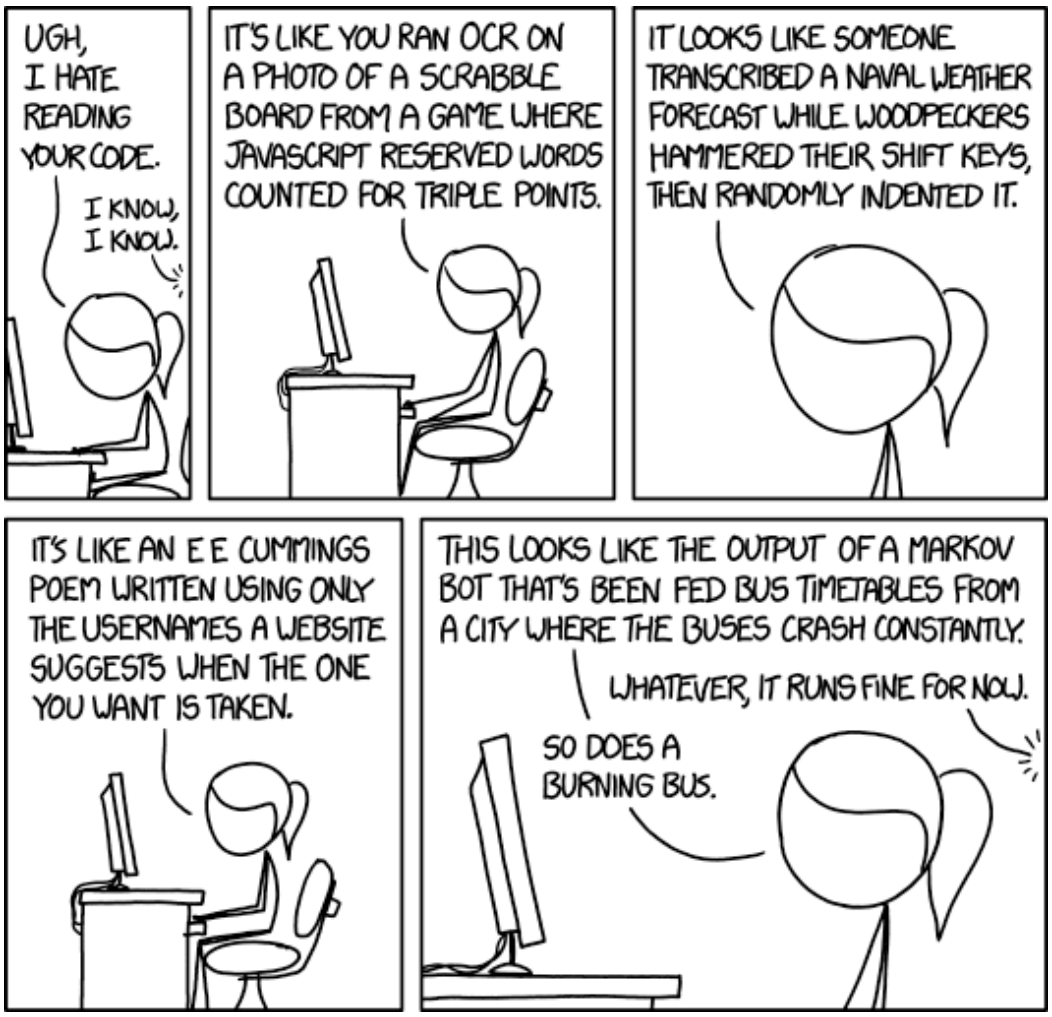
Do not start the test until instructed to do so!

Solutions are in blue. Commentary and explanations are in green.

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. The pipelined version achieved better performance than the single-cycle version by improving instruction throughput, but it did not achieve the theoretically-best speedup of 5.

a) **[8 points]** Explain what the designers of the MIPS pipeline would have needed to accomplish in order to achieve that theoretically-best speedup. (This is not a question about detailed hardware issues.)

The inherent pipeline throughput is one instruction per clock cycle, so they needed a design that would have a clock cycle 1/5 as long as that of the SCD design, or 160ps.

With both designs, one instruction is completed per clock cycle (once the pipeline is "hot"). So, the speedup is determined by the ratio of the cycle length for the SCD design and the cycle length for the pipeline design.

b) **[8 points]** What information do we have about the actual hardware the MIPS pipeline designers used that indicates why they could not achieve the result you described in the previous part.

We are told that the memory access stages and the ALU operation require 200ps to stabilize, so it was not possible to lower the clock cycle further.

This is clearly about how the hardware limited the ability to reduce the clock cycle.

---

2. **[12 points]** Suppose that a program spends part of its execution time performing integer arithmetic operations, part performing floating point operations, and part 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.

If it were possible to speed up I/O operations by a factor of 3, what percentage of the program's original execution time must it have spent on I/O operations in order to reduce the program's execution time to 2/3 of its original time?

Use Amdahl's Law to justify your answer.

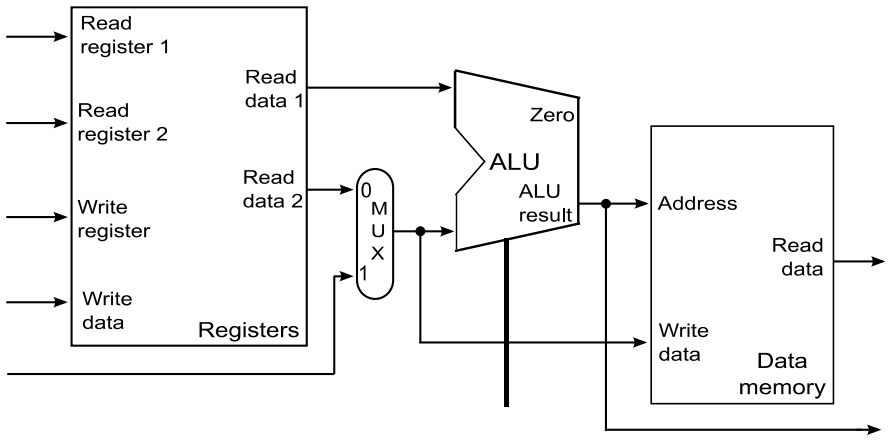
Suppose that the original execution time was T, and the time spent on I/O operations was x; then the time spent on other operations was T - x. Using Amdahl's Law, assuming we can speed up I/O operations by a factor of 3, we need to satisfy the equation:

$$\frac{2T}{3} = T - x + \frac{x}{3}$$

Solving for x, we get that x must be T/2, so the program must spend 50% of its time on I/O.

There are other ways to set up a formula from Amdahl's Law; many solutions used separate variables for integer arithmetic, floating-point arithmetic, and I/O, which is perfectly valid. OTOH, everything but I/O comes under the heading of "not affected" by the change.

3. 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 below:



Register	value
$t0	1000
$t1	2000
$t2	3000

Address	value
0	segfault
1000	10
2000	20
3000	30
4000	40
5000	50
6000	60
7000	70

Everything else in the datapath is implemented as shown on the diagram in the Supplement. All the control signals would be set as discussed in class. For each question below, we will assume that registers and memory locations hold the values shown in the tables above.

a) **[8 points]** What value would be stored in the Data memory, and at what address, if the following instruction was executed? Justify your answer.

```
sw    $t2, 1000($t1)
```

The error means that the sign-extended immediate from the instruction (1000) will be written into data memory, at the usual address ($t1 + 1000) for sw.

So, the value 1000 will be written at the address 3000.

The hardware error affects nothing except what is sent to the Write date port on the Data memory unit (sign-extended immediate instead of the value from $t2). The address calculation proceeds normally, and the question states that all control signals are set normally as well.

b) **[8 points]** What value would be stored in the register `$t1` if the following instruction was executed?

```
lw    $t1, 1000($t0)
```

The error only affects sw instructions, since it only affects what value is sent to the Write data port on the Data memory unit.

So, the lw instruction will proceed normally, and store the value Mem[$t0 + 1000] == Mem[2000] == 20 into the register $t1.

4. In our single-cycle processor design, the instruction with the longest latency is the `lw` instruction, which takes 800ps. This causes the overall clock cycle time to also become 800ps. To reduce the latency of the `lw` instruction and, therefore, reduce the cycle time, one option is to remove the *register+immediate* addressing mode from all memory-access instructions (including the `sw` instruction).

That is, we would have these instructions

```
lw   $rt, ($rs)          # GPR[rt] = Mem[GPR[rs]]
sw   $rt, ($rs)          # Mem[GPR[rs]] = GPR[rt]
```

but not these instructions

```
lw   $rt, imm($rs)       # GPR[rt] = Mem[GPR[rs] + imm]
sw   $rt, imm($rs)       # Mem[GPR[rs] + imm] = GPR[rt]
```

By eliminating the address computation step from memory-access instructions, we would reduce the latency for `lw` and `sw`, and so we could reduce clock cycle time to 600ps. However, this would require using an additional instruction `addi` to compute the memory address before each `lw` or `sw` instruction that previously specified an *immediate* value that wasn't zero (i.e., we don't need to add if the *immediate* is zero). The current datapath design can actually execute the `addi` instruction without any further modifications, so the need for that instruction would not, in itself, be a problem.

a) **[10 points]** Assume that in a typical program, memory-access instructions account for 40% of all executed instructions. For simplicity, also assume that all memory-access instructions specify *immediate* values that are not zero.

Quantitatively explain why removing the *register+immediate* addressing mode would be a bad idea. (That is, make a precise argument with numeric content.)

A program that required executing I instructions before will now require executing 1.4I instructions, due to the additional addi instructions needed. For that program:

$$\text{Old Time} = 800 \times \text{Ips}$$

$$\text{New Time} = 1.4 \times 600 \times \text{Ips} = 840 \times \text{Ips}$$

Since the new time is greater, the change would degrade performance.

There were many ways to set this up. You could use Amdahl's Law to get the execution time after the change. Effectively, the change speeds up non-memory-access instructions from 800ps to 600ps (or by a factor of 4/3), and it slows down memory-access instructions from 800ps to 1200ps (or by a factor of 2/3). So:

$$\begin{aligned} T_{after} &= \frac{T_{non\text{-}memory\text{-}access}}{4/3} + \frac{T_{memory\text{-}access}}{2/3} \\ &= \frac{3}{4}(0.6T_{before}) + \frac{3}{2}(0.4T_{before}) \\ &= \frac{3}{4}(0.6)(800I) + \frac{3}{2}(0.4)(800I) \\ &= \frac{3}{4}(480I) + \frac{3}{2}(320I) = (360 + 480)I = 840I \end{aligned}$$

b) **[10 points]** What is the maximum percentage of executed instructions that could be memory-access instructions, in order for removing the *register+immediate* addressing mode to be a good idea?

Let the percentage of memory-access instructions be x. To achieve a better execution time, we would need to satisfy:

$$800I > (1+x)600I$$

Solving this yields x < 1/3, so there could not be more than 33% memory-access instructions.

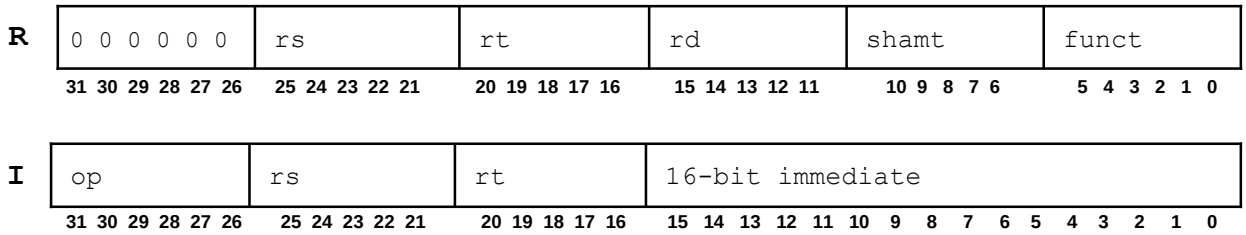
Again, this can be set up in other ways. Taking the Amdahl's Law setup shown earlier, let x be the percentage of memory-access instructions in a program. Then we have:

$$\begin{aligned} T_{after} &= \frac{T_{non\text{-}memory\text{-}access}}{4/3} + \frac{T_{memory\text{-}access}}{2/3} \\ &= \frac{3}{4}(1-x)(T_{before}) + \frac{3}{2}(x)(T_{before}) \\ &= (1-x)(600I) + (x)(1200I) \\ &= (600 - 600x + 1200x)I = 600(1+x)I \end{aligned}$$

We want this to be less than the original execution time, so:

$$600(1+x)I < 800I$$

5. Recall the formats for the R-type and I-type MIPS machine instructions:



a) **[8 points]** The design of these MIPS machine instructions makes the hardware for handling register reads simpler than the hardware for handling register writes. Explain why. Be specific.

For R-type instructions, the destination register is specified by bits 15:11, but for I-type instructions it's specified by bits 20:16.

On the other hand, for both formats the first read register specified by bits 25:21 and the second read register (if applicable) is specified by bits 20:16.

So a choice has to be made to choose the correct bits to specify the destination register, but not for either read register.

The question was clearly about how the design of the machine instructions related to the handling of register reads and register writes, NOT about the difference between executing register reads and register writes in the broader sense. Many answers ignored the issue of the machine instruction formats, and discussed irrelevant issues. Other answers discussed efficiency, which was also not related to the machine instruction format.

b) **[8 points]** What hardware component and control signal in the single cycle datapath design are needed because of the fact described in part **a)**? Be specific.

The difference noted above is why we must have the RegDst signal and the MUX it controls.

Nothing else is relevant to the given question.

We DO need RegWrite for register writes, but not for register reads, but that has nothing to do with the instruction formats.

We DO need other hardware for register writes (ALU to compute values, Data memory for lw to read data values, MemtoReg and its MUX, etc) that is not needed for register reads, but that's not a side-effect of the machine instruction format; it's a side-effect of the logic of execution.

6. 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). Suppose the following instruction was executed:

```
beq   $t3, $t1, btarg  # btarg is the label for some instruction
```

a) **[6 points]** Would the decision of whether to take the branch be made correctly? Explain why or why not.

RegDst and MemtoReg are don't-care for beq; MemRead, MemWrite and Jump should be 0 for both R-type and beq. ALUSrc should be 1 for R-type and beq. So those are all OK.

Branch will be set to 1, and ALUop will tell the ALU to subtract. Therefore, the AND gate will receive the proper signals, and the decision whether to take the branch will be made correctly.

The question talks about the settings of control signals, and so should your answer. One common flaw was to not say anything explicit about the don't-care signals, or the ones that are the same for both beq and R-type instructions.

b) **[5 points]** Could any unintended changes be made to value(s) in Data memory? If so, describe what could happen. If not, explain why not.

MemWrite will be set to 0, so no changes will be made to Data memory.

You need to explain WHY Data memory cannot be changed in this case, and that is due to exactly one thing: MemWrite == 0.

c) **[5 points]** Could any unintended changes be made to value(s) in registers? If so, describe what could happen. If not, explain why not.

Unfortunately, RegWrite will be set to 1.

Since MemtoReg will be set to 0, the ALUresult ($t3 - $t1) will be passed back to the Write data input on the register file.

Since RegDst will be set to 1, the Write register number will be specified by the high 5 bits of the immediate field of the beq instruction… and we don't know what those are.

So, the value $t3 - $t1 will be written to SOME register.

I wanted to see three things in you answer: why a register will be written to, what will be written there, and what register will be written to (as much as you can identify that).