Instructions:

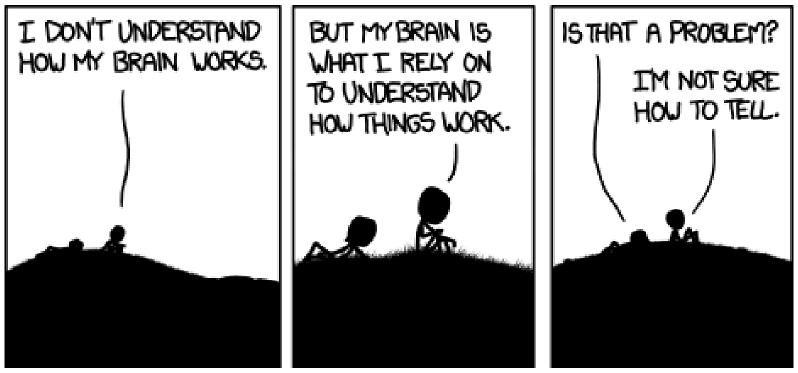
- Print your name in the space provided below.
- This examination is open book, but no other resources are allowed. No calculators or other computing devices may be used. The use of any such device will be interpreted as an indication that you are finished with the test and your test form will be collected 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 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 and turn in 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. [6 points] A processor with a clock rate of 2.0 GHz executes 4 billion machine instructions in 3 seconds. What is the average CPI of the instructions that were executed? Justify your conclusion.

From the given information, there are 3 x 2.0 x 10^9, or 6.0 x 10^9 clock cycles in 3 seconds.

So, the average CPI for this sequence of instructions would be

6.0 x 10^9 / 4.0 x 10^9 = 1.5 cycles per instruction

---

2. Suppose that running a program on a system requires executing *I* instructions, consisting of 30% integer add instructions, 20% integer multiply instructions, and 50% other instructions. With the current hardware, integer add instructions take 4 clock cycles, integer multiply instructions take 2 clock cycles, and each of the other instructions take 1 clock cycle.

a) [10 points] What is the total time (in clock cycles) needed to execute this program (in terms of *I*)? Justify your conclusion.

#ClockCycles = 0.30I x 4 + 0.20I x 2 + 0.50I x 1 = (1.2 + 0.4 + 0.5)I = 2.1I

b) [6 points] When this program is executed, what fraction* of the execution time is spent performing integer addition instructions? Justify your conclusion.

The number of clock cycles spent on integer multiplication is 1.2I, and the total number of clock cycles is 2.1I, so the fraction is

1.2I / 2.1I = 12/21 = 4/7

(which is about 57%).

* Use fractions (rational numbers), not decimal representation when you work this out.

c) [10 points] Suppose it's possible to speed up the execution of integer multiply instructions by 50%, without altering the number of cycles required for any other instructions. If that improvement is made and we executed the same program on the improved hardware, what would be the speedup*? Justify your conclusion.

Applying Amdahl's Law, the number of cycles required would now be

$$\text{Cycle}_{\text{safter}} = \text{Cycles}_{\text{unaffected}} + \text{Cycles}_{\text{affected}} / \text{Speedup} = 0.9I + 1.2I / 1.5 = 1.7I$$

Now, there's a question… the problem didn't say that the clock cycle length would not change as a result of the improvement in the integer multiplication hardware. If it does, then we don't have enough information to answer this question numerically, but we can still give an answer.

Suppose the old clock rate is $\text{Rate}_{\text{before}}$ and the new clock rate is $\text{Rate}_{\text{after}}$. Then we know that

$$\text{ExecutionTime} = \#\text{cycles} * \text{CycleLength} = \#\text{cycles} / \text{ClockRate}$$

So we'd have:

$$\text{ExecutionTime}_{\text{before}} = 2.1I / \text{Rate}_{\text{before}}$$
$$\text{ExecutionTime}_{\text{after}} = 1.7I / \text{Rate}_{\text{after}}$$

And, so the speedup would be

$$\text{ExecutionTime}_{\text{before}} / \text{ExecutionTime}_{\text{after}} = (2.1I / \text{Rate}_{\text{before}}) / (1.7I / \text{Rate}_{\text{after}})$$

or

$$\text{Speedup} = 7 * \text{Rate}_{\text{before}} / 5 * \text{Rate}_{\text{after}}$$

And, if the clock rate doesn't change, that would reduce to 7/5 (or 1.4).

3. Suppose a C programmer writes a C program with three functions `foo()`, `bar()` and `zoo()`; and a prototype MIPS compiler generates the following assembly code for each function.

C code (pseudo-code): Assembly code (with addresses and instructions)

C code	Assembly code

`int foo() {`
`   . . .`
`   x = bar (a, b);`
`   printf("x: %d\n", x);`
`   . . .`
`}`
	`. . .`
	`        foo:`
	`. . .`
	`0x4000  3a  move  $a0, $t0`
	`0x4004      move  $a1, $t1`
	`0x4008      jal   bar`
	`0x400c      move  $a0, $v0`
	`0x4010      li    $v0, 1`
	`0x4014      syscall`

`int bar(int a, int b) {`
`   x = zoo(b, a);`
`     // note params`
`   return x;`
`}`
	`. . .`
	`0x5000 bar: move  $t0, $a0`
	`0x5004      move  $a0, $a1`
	`0x5008      move  $a1, $t0`
	`0x500c  3b  jal   zoo`
	`0x5010      jr    $ra`

`int zoo(int a, int b) {`
`   x = a + b;`
`   return x;`
`}`
	`. . .`
	`0x6000 zoo: add   $v0, $a0, $a1`
	`. . .`
	`0x600c      jr    $ra`

a) [6 points] Suppose the initial register states were as follows (the second column) before the program executed the instruction 3a (`move $a0, $t0`) in `foo()`. Then, the program makes progress, and now it is about to execute the instruction 3b (`jal zoo`) in `bar()`. Please write down the register states before the instruction 3b executes.

	Before 3a	Before 3b
`$t0`	0x01	0x01
`$t1`	0x02	0x02
`$a0`	0x0	0x02
`$a1`	0x0	0x01
`$v0`	0x0	0x0
`$ra`	0x3200	0x400c
… (ignore the rest) …	…	…

b) [8 points] After running the assembly code, a C programmer found that the program did not write anything. Please find what was wrong in the assembly code and how to fix it. (You do not need to write down new assembly code. A detailed explanation of how to fix it is sufficient).

When jal zoo is executed, that resets $ra to 5010 for the return from zoo to bar. When zoo executes jr $ra, execution returns to jr $ra in bar. But executing that yields an infinite loop. We need to:

- back up the old $ra (400c) to the stack before bar executes jal zoo
- restore the old $ra (400c) from the stack to $ra before bar executes jr $ra

4. These questions refer to the simplified 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] Suppose the Branch control signal, labeled 4a on the datapath diagram, was stuck-at-1. Assume the rest of the hardware operates as designed. Which of the supported instructions would be affected, under what circumstances, and why?

The hardware would "think" that every instruction was beq. So, no matter what the current instruction was, if the ALU computed 0 the Zero signal would be set to 1, so the AND gate would output 1, and the branch target address would be passed to the (upper) right-most MUX.

Effectively, any R-type instruction or lw or sw for which the ALU computed 0 would result in a branch being taken. However, that would not affect the execution of that instruction itself, just produce a side-effect.

The j instruction would be entirely unaffected by this. When executing a j, the Jump signal is set to 1, and so the (upper) right-most MUX will pass the jump target address to the PC (and so it doesn't matter what the other input to that MUX might be).

b) [10 points] Suppose the MemtoReg control signal, labeled 4b on the datapath diagram, was stuck-at-1. Assume the rest of the hardware operates as designed. Which of the supported instructions would be affected, under what circumstances, and why?

The MUX to the right of the Data memory unit will always pass the Read data value from the Data memory unit to the Write data input on the Register file.

So, none of the R-type instructions would operate correctly (aside from the unlikely case that the memory read was allowed and the data read from memory matched the ALU's output).

But lw would operate exactly as intended, since it needs MemtoReg to be 1.

And since RegWrite is 0 for sw, beq and j, none of them would be affected at all.

c) [10 points] Consider the Add unit labeled 4c on the datapath diagram. Are there any supported instruction(s), for which this is unit <u>not</u> needed? (That is, are there instructions that would execute correctly, in all cases, even if this unit was removed.) If yes, identify those instructions and explain why they do not need this unit.

The value PC + 4, computed by this Add unit, is used in computing the address of the next instruction to be executed, no matter what the current instruction is.

So, if we didn't have the Add unit, we could never compute the address of the next instruction.

If you think that's just a side-effect, you would say the current instruction is still completed correctly.

But you could also argue that the address computation is required to logically complete the current instruction.

5. [12 points] Suppose the following instruction is being executed: `lw  $t1, 12($s7)`

When that instruction is executed, the datapath hardware will perform some actions that are logically unnecessary for that instruction (although they would be necessary for some other instructions). Identify two such actions, and for each explain why the fact that the hardware performs that (unnecessary) action does not cause any difficulties.

There are a number of irrelevant actions, including:

Computation of the branch target address (shifter, adder, zero signal)
shifting the sign-extended immediate bits
adding that to PC + 4
setting the Zero signal

Computation of the jump target address (shifter, concatenation)
shifting Instr[25:0]
concatenating that with PC+4[31:28]

Sending the output from Read data 2 to the Write data port on the Data memory unit

Sending the ALU output to the MUX to the right of the Data memory unit

In each case, the irrelevant action is harmless because it has no subsequent effect. For example:

- the branch target address is never used, because Branch will be set to 0 whenever we're executing a lw.
- the jump target address is never used, because Jump will be set to 0 whenever we're executing a lw.
- the value on the Write data input of the Data memory unit is never used, because MemWrite will be set to 0 whenever we are executing a lw.
- the MUX to the right of the Data memory unit never sends the ALU result anywhere, because MemtoReg will be set to 1 whenever we're executing lw.

6. [12 points] Consider the following proposed instruction for the simplified single-cycle MIPS32 datapath discussed in class:

```
addm    ($rd), $rs, $rt        # Mem[rd] = GPR[rs] + GPR[rt]
```

000000	rs	rt	rd	00000	funct

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

The instruction adds the values in registers `$rs` and `$rt`, and stores the result at the address in register `$rd`. Of course, this could be accomplished by a sequence consisting of an `add` instruction and a `sw` instruction, but that would require two clock cycles and an extra register for temporary storage. Supporting this new instruction would also require modifying the internals of the Control unit to recognize the `funct` field for the new instruction, but assume that's easily accomplished.

Haskell Hoo IV, who proposed the new instruction, insists that it can be added to the current datapath design (as shown on the datapath diagram) with no changes other than to the internals of the Control unit. That is, there will be no need to add any new hardware to the datapath, nor will there be a need for any new control signals.

If Haskell Hoo IV is correct, explain how the eight existing control signals (excluding ALUop) would need to be set.

If Haskell Hoo IV is incorrect, describe at least one hardware modification that must be made to the existing datapath in order to support `addm`, and explain why that modification is necessary.

Haskell Hoo IV is incorrect. Many changes would be needed, including:

- ***the ability to read three register values at once***
  - ***a Read register 3 input to the register file***
  - ***a Read data 3 output from the register file***

- ***the ability to send the third read register value to the Write data input on Data memory***
  - ***a MUX to choose whether Read data 3 or Read data 2 goes to the Write data input***
  - ***a control signal for that MUX***