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 9 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.

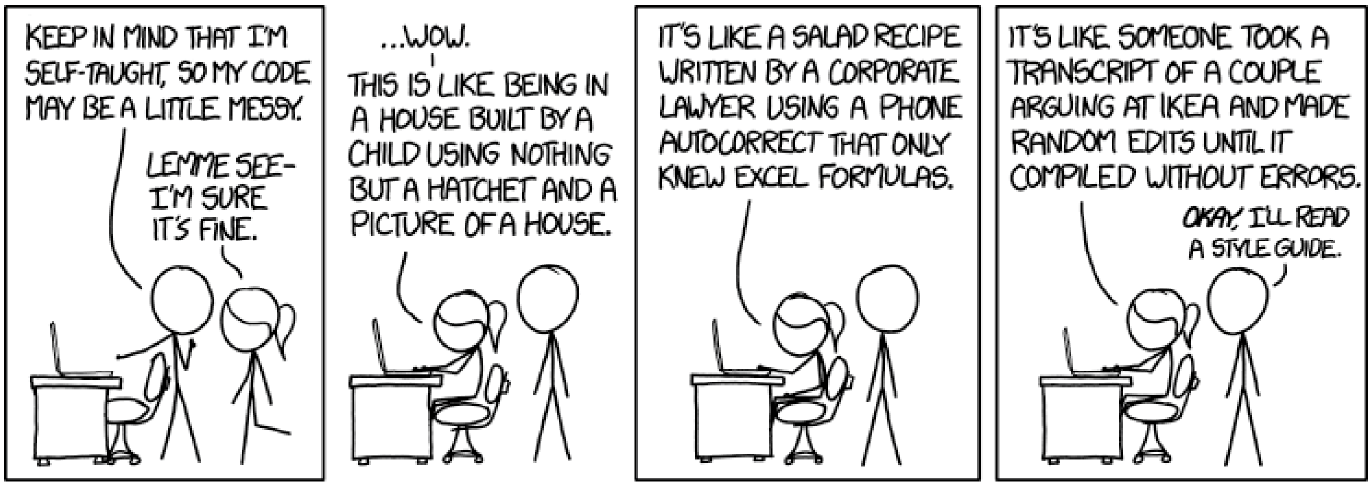
Do not start the test until instructed to do so!

Answers are formatted in blue; commentary is in green.

Name ______Solution______________
printed

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

____
signed


KEEP IN MIND THAT I'M SELF-TAUGHT, SO MY CODE MAY BE A LITTLE MESSY.
LEMME SEE– I'M SURE IT'S FINE.
...WOW.
THIS IS LIKE BEING IN A HOUSE BUILT BY A CHILD USING NOTHING BUT A HATCHET AND A PICTURE OF A HOUSE.
IT'S LIKE A SALAD RECIPE WRITTEN BY A CORPORATE LAWYER USING A PHONE AUTOCORRECT THAT ONLY KNEW EXCEL FORMULAS.
IT'S LIKE SOMEONE TOOK A TRANSCRIPT OF A COUPLE ARGUING AT IKEA AND MADE RANDOM EDITS UNTIL IT COMPILED WITHOUT ERRORS.
OKAY, I'LL READ A STYLE GUIDE.

xkcd.com

1. **[10 points]** The MIPS32 machine language includes a 6-bit `opcode` field to specify the particular instruction. However, this would only provide for 64 different machine instructions, so for R-type instructions (e.g., add, sub, or, etc.), the `opcode` is set to zero and an additional `funct` field (also 6-bit) is used to differentiate them. An alternative design would have widened the `opcode` field, say to 8 bits (for all machine instructions), and eliminated the `funct` field. Explain the downside of this alterative design.

Extending the opcode field may allow us to differentiate all the machine instructions in a unified manner. For R-type instruction, this new design would not affect the other fields in the machine instruction because the unused funct field will free out some space.

However, this is not the case for I-type and J-type instructions. In detail, the immediate field for I-type and J-type instructions should be reduced by two bits: i.e., from 16-bits to 14-bits for I-type, from 26-bits to 24-bits for J-type instructions.

The downside of this reduced immediate field is that branch or jump instructions are now only able to branch/jump out a smaller distance than the original design with larger immediate field.

---

2. **[10 points]** When compared to the single-cycle design, the pipelined MIPS datapath design achieves greater performance when executing a sequence of instructions because it improves instruction throughput. On the other hand, the execution time (latency) of each individual instruction actually increases. Explain why the latency increases.

The pipelined datapath is divided into multiple stages, which execute at the same clock cycle time. The implication is that to ensure that all stages are finished before moving to the next stage, the clock cycle needs to be as long as the longest stage. Even though one stage originally took less time, it still has to wait on the longest step to finish. Thus, faster stages (e.g, ID, WB) will actually have to wait the whole clock cycle.

3. Consider the following snippet of MIPS32 assembly code:

```
         . . .
         j    L02
L01:
         add  $t2,   $t2,   $t1
L02:     blt  $t2,   $t3,   L01
         . . .
```

a) **[4 points]** Assume that, immediately before the code given above is executed, `$t1` holds the value 2, `$t2` holds the value 1, and `$t3` holds the value 4. What is the value of `$t2` after the above assembly code finishes?

Initially, $t1 == 2, $t2 == 1, and $t3 == 4

Code execution goes like this:

```
j   L02
blt  $t2, $t3, L01    # $t2 == 1 and $t3 == 4, so we branch to L01
add $t2, $t2, $t1     # $t2 += $t1, so $t2 == 3
blt  $t2, $t3, L01    # $t2 == 3 and $t3 == 4, so we branch to L01
add $t2, $t2, $t1     # $t2 += $t1, so $t2 == 5
blt  $t2, $t3, L01    # $t2 == 5, so we do not branch and execution of this code stops
```

So, at the end $t2 == 5.

(In the absence of work showing your logic, very little partial credit was given.)

b) **[8 points]** Disassemble the above assembly code and write equivalent C code. Suppose the C program uses variables `X`, `Y`, and `Z`, which are mapped to `$t1`, `$t2`, and `$t3`, respectively.

From the analysis in part a), it's clear that this is a while loop (loop due to the backwards branch, while due to the jump to the loop test before entering the loop body).

So, in terms of the registers we have:

```
while ( $t2 < $t3 ) {
   $t2 = $t2 + $t1
}
```

Substituting in the given variable names, we have:

```
while ( Y < Z ) {
   Y = Y + X;
}
```

4. [10 points] Suppose the MIPS32 hardware supports only the instructions add, sub, and, or, slt, lw, sw, beq, and j. Pseudo-instructions give MIPS a richer set of assembly language instructions than those implemented by the hardware. Explain how the following pseudo-instruction can be implemented using only the supported instructions shown above. If you need to use any "extra" registers, you may use the t-registers, `$t0`, `$t1`, and so forth.

```
minfrom  $rd, ($rs), ($rt)
    # rd = Mem[GPR[rs]] < Mem[GPR[rt]] ? Mem[GPR[rs]] : Mem[GPR[rt]]
```

Executing this instruction requires loading two values from memory to registers, comparing them, and then conditionally assigning a value to a third register:

```
        lw    $t0, ($rs)         # load first value to $t0
        lw    $t1, ($rt)         # load second value to $t1
        slt   $t2, $t0, $t1      # $t2 = $t0 < $t1
        beq   $t2, $zero, ELSE   # if $t0 < $t1, $rd = $t0
        add   $rd, $t0, $zero
        j DONE
ELSE:   add   $rd, $t1, $zero    # else $rd = $t1
DONE:
```

The most common error was to fail to copy the values for the second and third operands from memory into registers, so they could be operated on.

5. [10 points] The MIPS32 architecture reserves the $ra register (register number 31) to hold the return address (not the return value) when a function call completes. Despite this, many MIPS programs use the runtime stack in memory to store the return address. Explain why having the $ra is not sufficient, and a program may use the runtime stack in memory to hold a return address. Illustrate your explanation by giving the outline of a relevant, short C program.

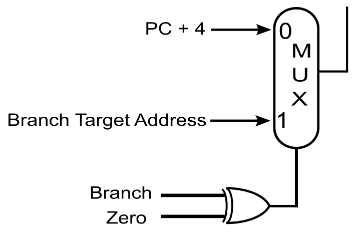
$ra registers can hold one return address at a time, so if there is a multiple (nested) function calls in sequence the return address in $ra would be overwritten and lost. Therefore, the return addresses should be stored somewhere (i.e., in stack) per each function call to be safe.

```
foo(){                        bar(){                        zoo(){
   bar(); // foo calls bar       zoo(){ // bar calls zoo        …
}                             }                             }
```

When foo() calls bar(), $ra would hold the address next to the call instruction in foo();
When bar() calls zoo(), $ra would be overwritten to the instruction address next to the call instruction in bar();
Therefore, zoo() can safely return to bar(); but, bar() cannot return to foo().

(Many people provided a recursive function as an example.)

6. Suppose that the AND gate for the beq decision was replaced with an XOR gate:



a) **[6 points]** Describe exactly how that (erroneous) change would affect the execution of beq instructions.

Recall that A xor B == 1 iff A and B are different.

So, if we are executing a beq instruction, we know that Branch == 1. Therefore, the XOR gate will output 1 iff Zero == 0, which means that the registers were NOT equal.

Hence a beq instruction will branch iff it should not branch!

The most common issue here was that the answer is an if-and-only-if statement; many answers only specified half of that… a number of other answers ignored the fact the question was specifically about the execution of beq instructions, and nothing else.

The second most common issue was to seemingly not recall the definition of XOR.

b) **[6 points]** Describe exactly how that (erroneous) change would affect the execution of sub instructions.

Now, if a sub instruction is being executed, Branch == 0. Hence, the XOR gate will output 1 iff Zero == 1, which means the result of the subtraction IS zero.

Therefore, if the result of a subtraction not zero, the sub instruction will execute correctly, with no undesirable side-effects.

But, if the result of a subtraction IS zero, the sub instruction will correctly compute the answer, and store it into the correct register, and also trigger a branch to an address that is computed using the bits in the RD, SHAMT, and FUNCT fields of the instruction.

(And that will most likely result in a segfault.)

Again, the answer requires addressing two cases, and the question is specifically about the execution of sub instructions, and nothing else.

7. Suppose a program currently spends 20% of its time executing integer addition instructions and 40% of its time executing integer multiplication instructions (on the existing design), 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.

a) **[8 points]** How much faster would the program execute if you made integer addition 2 times faster and integer multiplication 4 times faster?

According to Amdahl's Law, the execution time after the improvement would be given by:

$$\begin{aligned} Time_{after} &= 0.40 * Time_{before} + \frac{0.20 * Time_{before}}{2} + \frac{0.40 * Time_{before}}{4} \\ &= 0.60 * Time_{before} \end{aligned}$$

So, the program would take 60% as long to execute; the speedup would be 100/60 or 1.667.

The instructions were to justify your conclusion "by using Amdahl's Law"; failure to explicitly use Amdahl's Law cost 3 points here (and 2 points in part b).

b) **[6 points]** Suppose you need to reduce the running time of the program by 20%, but you are restricted to making changes to either the addition hardware or the multiplication hardware, but not both. Which would you alter, and by how much would you have to improve that hardware to achieve your goal? Or is it impossible to do so?

Since the program only spends 20% of its time running addition instructions, we cannot reduce the running time by 20% by speeding up addition operations (unless we make them take NO time, which is impossible).

Therefore, we must focus on speeding up multiplication operations. Applying Amdahl's Law again, and letting S be the factor by which we will speedup multiplication operations:

$$0.80 * Time_{before} = 0.60 * Time_{before} + \frac{0.40 * Time_{before}}{S}$$

Solving, we get that S == 2, so we must make multiplication twice as fast.

8. **[12 points]** The preliminary pipeline, as derived in lecture and shown in the supplement, does not have support for forwarding operands or introducing stalls. Suppose the following sequence of instructions was executed on that pipeline design:

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

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.

Recall that no instruction actually writes a value to a register until that instruction reaches the WB stage in the pipeline. Therefore, if an instruction is to read a value from a register, and an earlier instruction has written that value, then the reading instruction must be at least 3 cycles (stages) behind the writing instruction (unless we have forwarding, which we do not).

So, the code above would lead to the following logical errors:

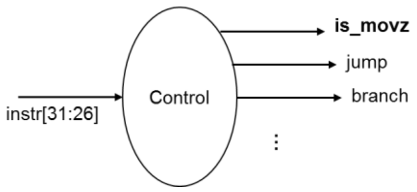
- **Instruction #2 (and) will read from register $t4 two cycles before instruction #1 (lw) writes to $t4.**
- **Instruction #3 (add) will read from register $t4 one cycle before instruction #1 (lw) writes to $t4.**
- **Instruction #3 (add) will read from register $t1 two cycles before instruction #2 (and) writes to $t1.**
- **Instruction #4 (sub) will read from register $t2 two cycles before instruction #3 writes to $t2.**

Instruction #3 (add) reads from register $t4 during the second half of the cycle on which instruction #1 (lw) writes to $t4, so that does not result in an error.

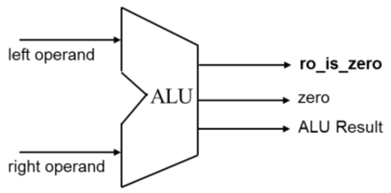
9. **[10 points]** Suppose we want to support a new instruction, *movz* (Move Conditional on Zero). The *movz* instruction is a R-type instruction that takes three register numbers, and conditionally moves a value from one register to another register after testing the third register value. Specifically, if the value in GPR[rt] is equal to zero, then the contents of GPR[rs] are placed into GPR[rd]. Here is the formal specification.

Format:	MOVZ rd, rs, rt
Description:	if (GRP[rt] == 0) then GRP[rd] ← GRP[rs]

Suppose we have already extended the Control Unit so that when a *movz* instruction is decoded, it sets the new control signal, called *is_movz*, to be true (otherwise, false); and sets the ALUop control signal accordingly (the specific value doesn't matter) to make the extended ALU Unit behave as follows. First, the ALU Unit introduces the new control signal, named *ro_is_zero*, and sets it true if the right operand (where the output of the MUX is fed) is zero. Second, the ALU Unit simply forwards the value from the left operand (where GRP[rs] is fed) to its output, ALU Result, regardless of the value of the right operand.



Control Unit

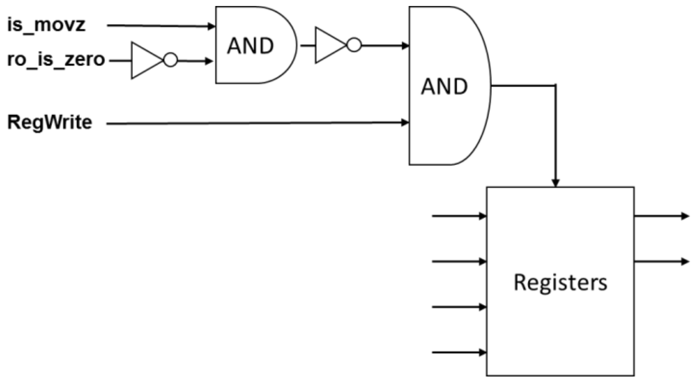


ALU Unit

Given the above extended Control Unit and ALU Unit, explain how the datapath should be further modified to support *movz* instruction.

For R-type instructions, by default, the output of ALU unit (ALU Result) is written back to the destination register (GRP[rd]) because the control signals MemtoReg is 0; and RegWrite is 1. For movz instruction, we know that ALU Result would hold GRP[rs] value (the left operand of ALU unit). If both is_movz and ro_is_zero are true, then we don't need to do anything special. However, we need to disable the write-back to destination register GRP[rd] if is_movz is true and ro_is_zero is false.

One way to implement this is to have two AND and NOT gates as follows. In this specific implementation, for non-movz instructions, because is_movz is set to 0, the register write-back will be solely controlled by RegWrite as usual. For movz instructions, because RegWrite is 1 (as in R-type instruction), the register write-back will be controlled by ro_is_zero signal.



(There are many different but valid answers to this question).