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 and the MIPS reference card. No calculators or other computing devices may be used.
- 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, 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

Questions 1 through 3 refer to Figure 4.24 from P&H that was distributed with the test.

1. Consider the set of MIPS instructions that are supported by the given datapath.

a) [10 points] Which of the supported instructions benefit from the fact that the data register implementation allows reading two register values simultaneously?

Any instruction that requires using values from two different registers as input; in this case, that would include:

`add`, `sub`, `and`, `or`, `slt`, `sw`, and `beq`

b) [10 points] For which of the supported instructions must the RegWrite control signal be set to:

0: **instructions that do not store a result to a register: `sw`, `beq` and `j`**

1: **instructions that do store a result to a register: `add`, `sub`, `and`, `or`, `slt` and `lw`**

---

2. [20 points] Consider executing the following assembly language code on this datapath. The values shown at the left are the addresses at which the corresponding machine instructions are stored. Do not worry about whether any of the given assembly instructions would expand to two or more machine instructions. All numeric values, including addresses are given in base-10.

```
                .data
00000100  X:    .word
. . .
00001000  Y:    .word
. . .
                .text
. . .
00004092        lw    $s1, 100($s0)   # assume $s0 == 4000 and $s1 == 100
00004096        sub   $s4, $s0, $s1
00004100        lw    $s3, 0($s0)
00004104        lw    $s4, 0($s1)
. . .
```

Consider what would happen if this code were executed if the Branch control line was stuck at 1. Describe the results in detail up to the point that the first unintended consequence would occur and describe that consequence in detail.

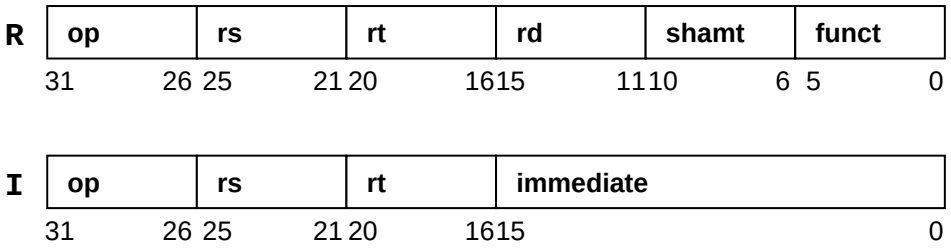
If the Branch signal is stuck at 1, then the datapath will act, in part, as if the current instruction is `beq`, whether or not it actually is.

In this example, the `lw` instruction will execute and complete normally, loading the value from address 4100 into register `$s1`; i.e., `$s1 <-- Mem[4100]` (which is the machine code for the second `lw` instruction).

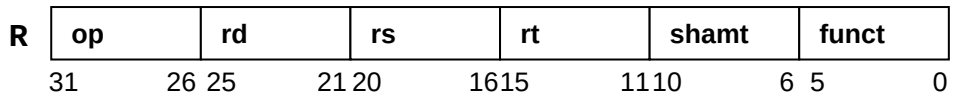
Since the ALU result is 4100, the Zero control signal will be 0, and so the branch will be not be taken at this point, and the next instruction to be executed will be the `sub` instruction. If the code for the second `lw` instruction happens to equal the value of `$s0` (4000) then the `sub` instruction will complete and then a branch will be taken, to the instruction at an address computed from the low 16 bits of the `sub` instruction.

However, we cannot say what the next instruction will be without analyzing the binary representation of the `sub` and `lw` instructions (and knowing what instruction, if any is at the corresponding address).

3. [20 points] The R-format and I-format instructions use the following machine language formats:



Recall that **rs** is the left operand, **rt** is the right operand and **rd** is the destination register. Suppose the R-type format was changed so that the destination register was listed before the source registers:



The I-format instructions are not changed. Describe how the given datapath would have to be changed in order to accommodate this change… or explain why it would not have to be changed.

The actual MIPS machine language format means that the two source register numbers for the R-type instructions and the two register numbers for the I-type instructions always come from the same bits of the machine instructions (25:21 and 20:16).

If the change shown above were made, the first register to be fetched for an R-type instruction would come from bits 20:16 but the first register to be fetched for an `lw` or `sw` instruction (used in the address calculation) would come from bits 25:21.

This would mean we would have to add a multiplexor to the datapath to select which bits were used to specify Read register 1, depending on the opcode bits for the current instruction.

Moreover, the second register to be fetched for an R-type instruction would come from bits 15:11 but the second register to be fetched for a `sw` operation would come from bits 20:16, so we would also need a new multiplexor to determine which bits were used for that as well.

And, the inputs to the current multiplexor used to select the bits for the write register number would have to be changed so that it selected between bits 25:21 (for R-type instructions) and 20:16 (for `lw`).

For questions 4 and 5, feel free to refer to the list of basic (native) MIPS instructions that was distributed with the test.

4. [20 points] The native MIPS assembly language does not include a memory-to-memory data transfer instruction:

```
    mcopy   ($rd), ($rs), $rt       # copy $rt consecutive bytes, starting at
                                    #  address $rs to consecutive locations
                                    #  starting at address $rd
                                    #
                                    # Mem[$rd:$rd+$rt] <-- Mem[$rs:$rs+$rt]
```

Show how an assembler might replace the pseudo-instruction above with a sequence of basic (native) instructions to achieve the same effect. Include comments to explain the logic of your design. For simplicity, it is acceptable to use temp registers if they are backed up and restored properly. You should not modify any of the registers used in the instruction itself.

```
        addi  $sp, $sp, -16          # make room to back up t-regs used below
        sw    $t0, 12($sp)           # back up t-regs
        sw    $t1,  8($sp)
        sw    $t2,  4($sp)
        sw    $t3,  0($sp)

        or    $t0, $rs, $zero        # set pointer to source data
        or    $t1, $rd, $zero        # set pointer to target region
        or    $t2, $t2, $rt          # set counter for bytes copied

mcloop: beq   $t2, $zero, mcdone     # exit copy loop when $rt bytes are copied
        lb    $t3, 0($t0)            # get next byte to be copied
        sb    $t3, 0($t1)            # write byte to destination address
        addi  $t0, $t0, 1            # step to next source byte
        addi  $t1, $t1, 1            # step to next destination address
        addi  $t2, $t2, -1           # decrement byte counter
        j     mcloop                 # check for loop reentry
mcdone:

        lw    $t0, 12($sp)           # restore used t-regs used above
        lw    $t1,  8($sp)
        lw    $t2,  4($sp)
        lw    $t3,  0($sp)
        addi  $sp, $sp, 16           # restore stack pointer
```

5. [20 points] Assume that a MIPS procedure F takes three parameters via the runtime stack, and places its return value onto the stack. The procedure is designed so that it expects the stack to be organized as shown below when it begins to execute:

```
        | unknown            | <-- $sp was here before setting up for call
        +--------------------+
        | address of array   |
        | dimension of array |
        | target value 1     |
        | target value 2     |
$sp --> | return value       | <-- $sp should be here when the call occurs
        +--------------------+
```

Given the data segment below for the calling procedure, write the code the calling procedure would need in order to set up the stack before the call and the code the calling procedure would need to retrieve the 4-byte return value to `$v0` and tear down the stack setup after the call:

```
          .data
Size:     .word  100
Array:    .word  400
Targ1:    .word   42
Targ2:    .word   20

# code to set up stack goes here:

        addi   $sp, $sp, -20        # make room for specified values on stack

        la     $t0, Array           # get address of array
        sw     $t0, 16($sp)         # put it on the stack
        lw     $t0, Size            # get size of array
        sw     $t0, 12($sp)         # put it on the stack
        lw     $t0, Targ1           # get target 1
        sw     $t0,  8($sp)         # put it on the stack
        lw     $t0, Targ2           # get target 1
        sw     $t0,  4($sp)         # put it on the stack









           jal  F
# code to retrieve return value and tear down stack goes here:

        lw    $v0, 0($sp)             # fetch return value from stack

        addi  $sp, $sp, 20            # restore stack pointer
```