University of Calgary
Department of Electrical and Computer Engineering
ENCM 369: Computer Organization
Lecture Instructors: Steve Norman and Norm Bartley

Winter 2023 PRACTICE MIDTERM TEST #2A
Duration: 90 minutes

General Instructions

- If you use a **calculator**, it must be one of the following models sanctioned by the Schulich School of Engineering: Casio FX-260, Casio FX-300MS, TI-30XIIS.
- The test is **closed-book**. You may not refer to books or notes during the test, with one exception: you may refer to the *Reference Material* page that accompanies this test paper.
- You are not required to add **comments** to assembly language code you write, but you are strongly encouraged to do so, because writing good comments will improve the probability that your code is correct and will help you to check your code after it is finished.
- Some problems are relatively **easy** and some are relatively **difficult**. Go after the easy marks first.
- To reduce distraction to other students, you are not allowed to leave during the last **ten minutes** of the test.
- Write all answers on the question paper and hand in the question paper when you are done.
- Please print or write your answers **legibly**. What cannot be read cannot be marked.
- If you write anything you do not want marked, put a large X through it and write "rough work" beside it.
- You may use the backs of pages for rough work.

Problem	Mark
1	/ 11
2	/ 14
3	/ 12
4	/ 6
5	/ 6
TOTAL	/ 49

PROBLEM 1 *(11 marks)*

Consider the C code listed to the right. Translate the function `squish` into RARS assembly language. Follow the usual calling conventions from lectures and labs, and use only instructions from the Midterm 2 Instruction Subset described on the *Reference Material* page.

```
void squish(char *str)
{
  char *p;
  int space, c;
  p = str;
  space = 32;
  c = *str;
  while (1) {
    *p = c;
    if (c == '\0')
      break;
    p++;
    if (c != space) {
      str++;
      c = *str;
    }
    else {
      do {
        str++;
        c = *str;
      } while(c == space);
    }
  }
}
```

PROBLEM 2 (*total of 14 marks*) Questions on integer arithmetic and logical instructions.

Part a. *(4 marks.)* Suppose `t0` contains `0x8000_0002`, and the RISC-V instruction `addi t1, t0, -8` is attempted. Determine the 32-bit result that will be produced by the adder that handles this instruction. Show your work, and use hexadecimal notation for your answer.

Part b. *(1 mark.)* Did signed overflow occur in the addition of **part a**? Give a precise reason for your answer.

Part c. *(4 marks.)* Suppose `s0` contains `0x9000_0000`, `s1` contains `0x2fff_ffff`, and the MIPS instruction `subu s2, s0, s1` is attempted. Showing your work, determine the value that `s2` will receive. Use hexadecimal notation for your answer.

Part d. *(1 mark.)* Did signed overflow occur in the subtraction of **part c**? Give a precise reason for your answer.

Part e. *(4 marks.)* Suppose you had to write an assembler for RISC-V assembly language. A very small fragment of your code might need to give a value to the `s2` register as follows:

> Bits 6–0 of `s2` should be 0110111.
> Bits 11–7 `s2` should equal bits 4–0 of `s0`.
> Bits 31–20 of `s2` should equal bits 19–0 of `s1`.

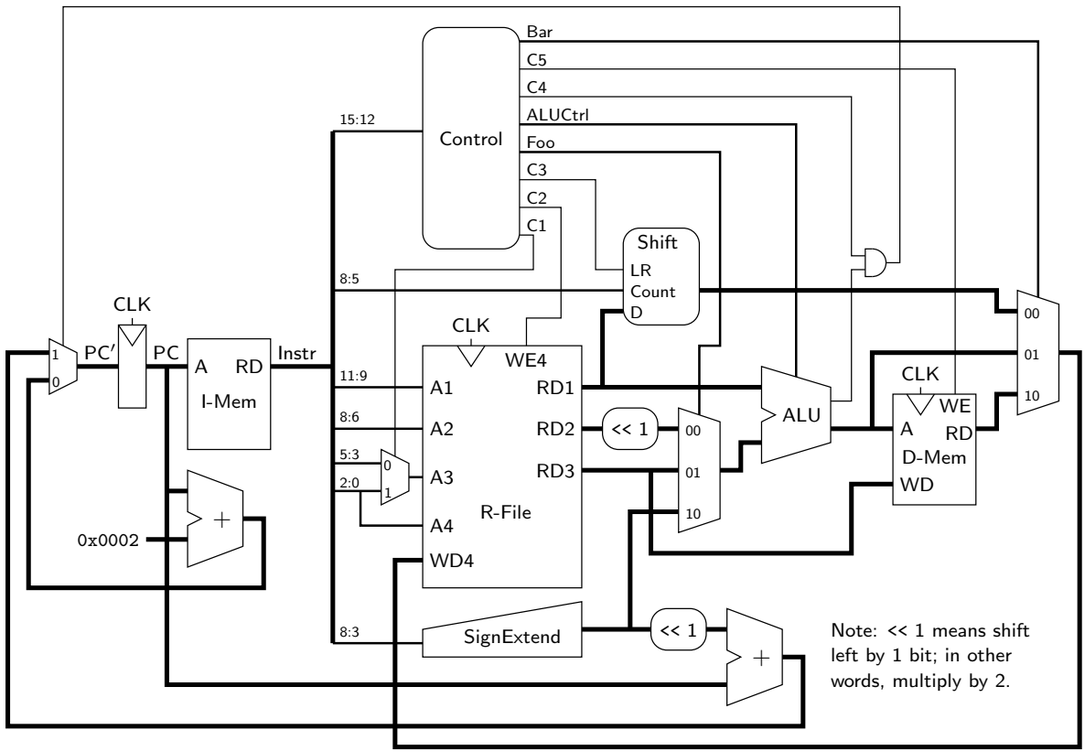
Write RISC-V instructions for the above operation. (Code just the above steps, *not a complete procedure*). You may use whatever t-registers you like for intermediate results.

PROBLEM 3 *(12 marks)* In the QuuxMT2 instruction set architecture, GPRs, memory addresses, memory words and instructions are all 16 bits wide. The number of GPRs is 8.

The table below specifies the machine code for eleven instructions. In the table, 3-bit register encodings are given as `ddd` for destination, `sss` for first source, and `ttt` for second source. For example, the machine code for `or x2, x4, x5` would be `0010_100_000_101_010`.

instruction	machine code	notes
`sll`	`0000_sss_cccc_00_ddd`	`cccc` is a shift count
`srl`	`0001_sss_cccc_00_ddd`	`cccc` is a shift count
`or`	`0010_sss_000_ttt_ddd`	
`and`	`0011_sss_000_ttt_ddd`	
`add`	`0100_sss_000_ttt_ddd`	
`sub`	`0101_sss_000_ttt_ddd`	
`slt`	`0110_sss_000_ttt_ddd`	
`addi`	`0111_sss_iiiiii_ddd`	`iiiiii` is a 6-bit 2's complement constant
`beq`	`1000_sss_iiiiii_ttt`	`iiiiii` is a 6-bit 2's complement constant
`lw`	`1001_bbb_iii_000_ddd`	see notes below about `lw` and `sw`
`sw`	`1010_bbb_iii_sss_000`	see notes below about `lw` and `sw`

Data address computation for `lw` and `sw` works this way, which is convenient for array element access: address = *base* + 2 × *index*. *base* comes from the register specified by `bbb` and *index* comes from the register specified by `iii`.



For the above single-cycle QuuxMT2 microarchitecture, fill in control signals in the table below. You must use X to indicate "don't-care" wherever appropriate.

Instruction	C1	C2	C3	Foo	ALUCtrl	C4	C5	Bar
`sll`								
`srl`								
`or`								
`and`								
`add`								
`sub`								
`slt`								
`addi`								
`beq`								
`lw`								
`sw`								

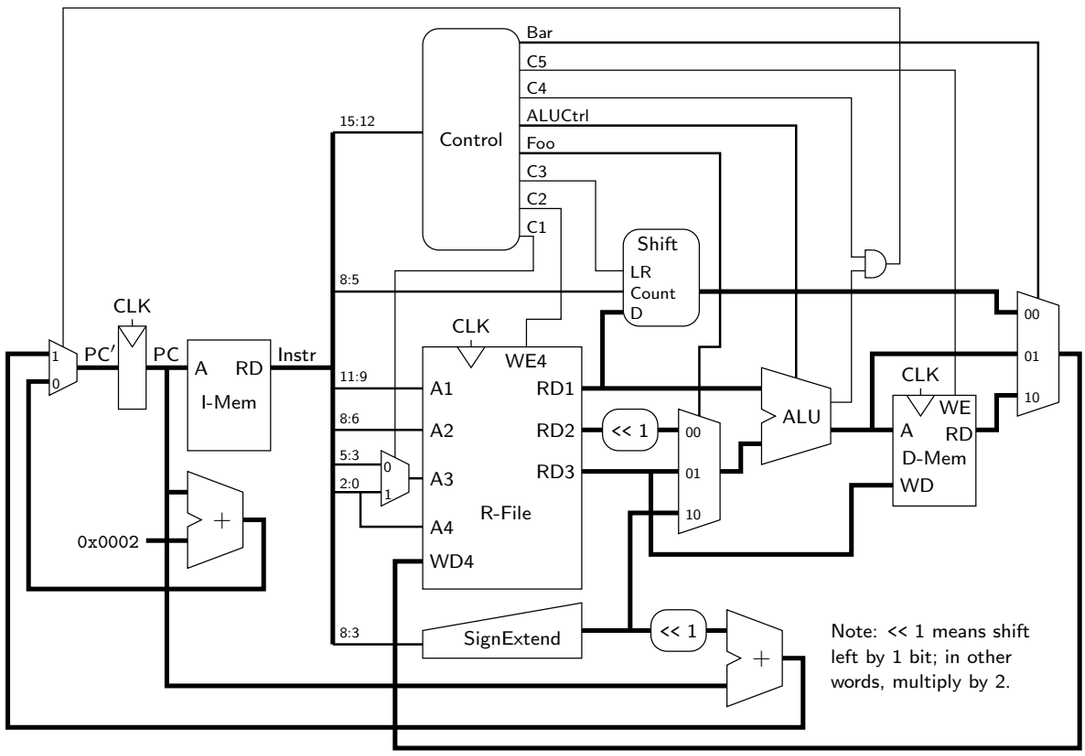
For the Shift unit, LR should be 1 for a left shift and 0 for a right shift.

These 3-bit patterns tell the ALU what to do:

bits	operation
000	bitwise AND
001	bitwise OR
010	addition
110	subtraction
111	SLT

PROBLEM 4 (*total of 6 marks*) This problem concerns the same machine as was used in Problem 3. For convenience, the table of machine code and the schematic are repeated here ...

instruction	machine code	notes
`sll`	`0000_sss_cccc_00_ddd`	`cccc` is a shift count
`srl`	`0001_sss_cccc_00_ddd`	`cccc` is a shift count
`or`	`0010_sss_000_ttt_ddd`	
`and`	`0011_sss_000_ttt_ddd`	
`add`	`0100_sss_000_ttt_ddd`	
`sub`	`0101_sss_000_ttt_ddd`	
`slt`	`0110_sss_000_ttt_ddd`	
`addi`	`0111_sss_iiiiii_ddd`	`iiiiii` is a 6-bit 2's complement constant
`beq`	`1000_sss_iiiiii_ttt`	`iiiiii` is a 6-bit 2's complement constant
`lw`	`1001_bbb_iii_000_ddd`	see notes below about `lw` and `sw`
`sw`	`1010_bbb_iii_sss_000`	see notes below about `lw` and `sw`



Part a. *(3 marks.)* Consider this QuuxMT2 assembly language fragment:

```
L1:  beq  x3, x6, L2
     lw   x2, x3(x4)  # x4 is base, x3 is index
     add  x5, x5, x2
     addi x3, x3, 1
     beq  x0, x0, L1
L2:  sw   x5, x0(x1)  # x1 is base, x0 is index
```

The machine code for the first `beq` instruction is `1000_011_000101_110`. Find the machine code for the `addi` instruction and the second `beq` instruction.

Part b. *(3 marks.)* [This part would not be a fair question for Midterm #2 in 2023. It would be a fair question for the final exam.]

Assume that t_{pd} for Control and t_{pd} for Sign Extend are both less than t_{pd} for the R-File. Timing analysis for the `sll` instruction includes summing t_{pd} values for I-Mem, R-File, Shift, and 3:1 bus mux.

Which t_{pd} values should be added together for timing analysis of the `addi` instruction?

PROBLEM 5 (*total of 6 marks*). Questions about miscellaneous topics.

Part a. *(3 marks.)* Consider a C toolchain for RISC-V processors, with a preprocessor, compiler, assembler and linker. Suppose that `bar.h` and `foo.c` are two of the many source files for a C project and that the compiler generates the given assembly language translation of `foo`. Explain precisely (1) why the assembler will have no trouble determining the machine code for the branch and `j` instructions, and (2) why the linker will have to help determine machine code for the `jal` instruction.

foo.c

```
#include "bar.h"

int foo(int fa)
{
  int x = bar(fa);
  if (x < -12)
    x = -12;
  else if (x > 12)
    x = 12;
  return x;
}
```

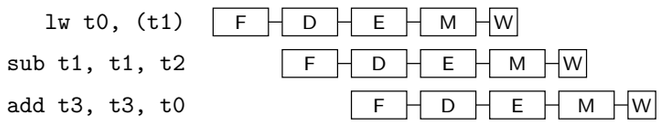
bar.h

```
int bar(int ba);
```

foo.s

```
        .text
        .globl  foo
foo:    addi    sp, sp, -16
        sw      ra, 12(sp)
        jal     bar
        li      t0, -12
        bge     a0, t0, L1
        li      a0, -12
        j       L2
L1:     li      t1, 12
        ble     a0, t1, L2
        li      a0, 12
L2:     lw      ra, 12(sp)
        addi    sp, sp, 16
        jr      ra
```

Part b. *(3 marks.)* This diagram outlines how a sequence of RISC-V instructions would be processed by the 5-stage pipeline introduced in lectures:



Explain precisely why there is a *data hazard* in the given sequence of instructions. (Reminder: In this 5-stage pipeline, GPR values are read from the Register File in the Decode stage.)