CIS 351 Practice Final Solutions

April 1, 2022

1. Complete Homework 5 and be comfortable answering similar questions.
2. What are the four elements that hold the Single Cycle CPU state?

```
PC, Instruction Memory, Register File, Data Memory
```

3. Why are the inputs (B, C) to the register file 5-bits while the outputs (I, G) are 32-bits? (Figure 1)

```
B and C are addresses for the registers that the register file will get values
from while the outputs I and G hold the 32-bit values contained within the
registers.
```

4. Is line F active for all instructions? If so, why does it have data on it for R-type or other instructions where it isn't used? If not, why is it not used and how is it kept empty?

```
Yes, data will always flow through the lines, but the multiplexer between G, I,
and J decides whether or not the output of the line is used.
```

5. Explain the function of the MemWrite flag going into the Data memory component.

```
The flag controls whether we are reading or writing data to memory.
```

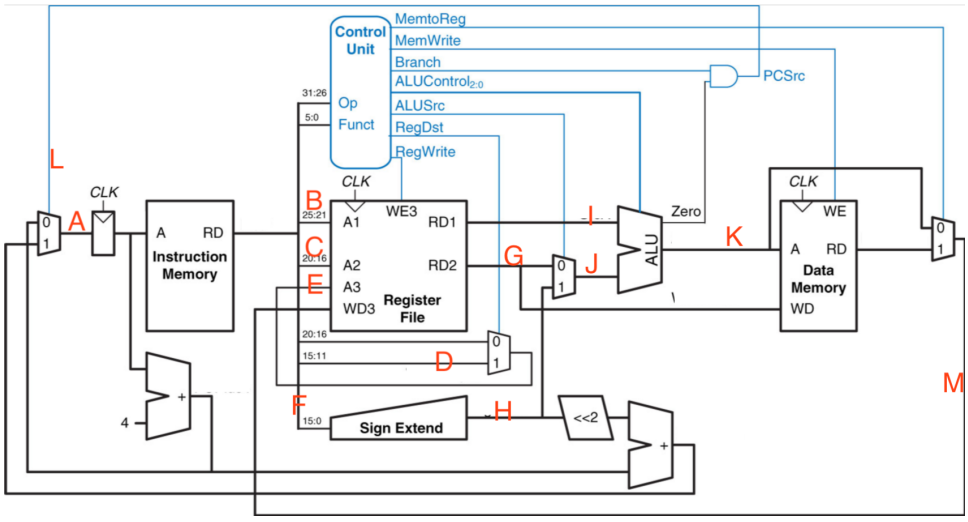

Figure 1: Single Cycle CPU with labeled points

Problems 6-8 are similar to a problem on the homework. However, in this case, you are not given which instruction to use.

6. Write a single assembly language statement that will set bits 1, 2, and 6 of register `$t0` to 1 and leave all other bits unchanged. `ori $t0, $t0, 0x46`

7. Write two or three assembly language statements that will set bits 4, 5, 7, and 8 of register `$t0` to 0 and leave all other bits unchanged.

```
lui $t1, 0xFFFF
ori $t1, $t1, 0xFE4F
and $t0, $t0, $t1
```

8. Write a single assembly language statement that will flip bits 0 and 7 of register `$t0` and leave all other bits unchanged. `xori $t0, $t0, 0x81`

9. Write one or two assembly language statements that will move bits 2 through 5 of register `$t1` into bits 0 through 3 of register `$t0`. Register `$t1` should remain unchanged. Bits 4 - 31 of register `$t0` should be 0.

```
srl $t0, $t1, 2
andi $t0, $t0, 0x 0F
```

Any coding questions on the exam will require strict adherence to assembly conventions, including preserved regsisters. They are intentionally designed to give few or no points to an almost-correct solution that does not correctly preserve registers or makes incorrect assumptions about which registers will be preserved by called functions.

10. Convert the following Java code to assembly. Use standard procedure-calling conventions — including preserving registers where appropriate.

```
int p1(int[] a0) {
  int x = 0;
  while (check(array[x], array[x + 2]) != 0) {
    array[x] = array[x + 2];
    x++;
  }
  return x;
}
```

```
p1:   addi $sp, $sp, -16
      sw $s0, 0($sp)
      sw $s1, 4($sp)
      sw $s2, 8($sp)
      sw $ra, 12($sp)

      addi $s0, $0, 0      # $s0 is the variable x
      move $s1, $a0  # $s1 is a pointer to the current position in the array

top:  lw $a0, 0($s1) # load array[x] into $a0
      lw $a1, 8($s1) # load array[x+2] into $a1
      move $s2, $a1  # save array[x+2] in a "safe" register for later use
      jal check
      beq $v0, $zero, end # exit the loop if check() returns 0
      sw $s2, 0($s1)      # save $s2 in array[x]
      addi $s1, $s1, 4    # increment the array pointer
      addi $s0, $s0, 1    # increment x
      j top

end:  move $v0, $s0       # place return value in $s0
      lw $s0, 0($sp)      # restore s registers
      lw $s1, 4($sp)
      lw $s2, 8($sp)
      lw $ra, 12($sp)
      addi $sp, $sp, 16   # restore the stack
      jr $ra
```

11. Implement the procedure `countX(int val, int[] array, int size)` recursively. You may find the C implementation below helpful. Note: I will not ask you to write a recursive function on the

exam, but I may ask you to read and understand recursive code. Also, recursive functions are good practice for calling functions correctly.

```
int countX(int val, int array[], int size)
{
  int count = 0;
  if (size == 0) { return 0;}

  if (array[0] == val) { count = 1; }

  /* The expression "array + 1" is legal in C because,
     just as it is in assembly, an array variable is simply
     the memory address of the first element in the array.
     In C, however, the "+1" means to add the *size* of one
     element, which, in this case, is 4 bytes.
   */
  return count + countX(val, array + 1, size -1);
}
```

```
# a0 is the value
# a1 is the array
# a2 is the size of the array
countX:
bne $a2, $zero, cont1
addi $v0, $0, 0
jr $ra

cont1:
addi $sp, $sp, -8
sw $s0, 0($sp)
sw $ra, 4($sp)

addi $s0, $0, 0
lw $t0, 0($a1)
bne $t0, $a0, cont2
addi $s0, $0, 1

cont2:
# a0 is unchanged
addi $a1, $a1, 4
addi $a2, $a2, -1
jal countX
add $v0, $v0, $s0

lw $s0, 0($sp)
lw $ra, 4($sp)
addi $sp, $sp, 8
jr $ra
```

12. Convert the following Java code to assembly. Use standard calling and register usage conventions. You may assume that `method1`, `method2`, and `method3` all exist. (In other words, call them but don't write them.)

```
int composition(int a, int b) {
   return method1(a, b, method2(method3(a) + method3(b)));
}
```

```
composition:

addi $sp, $sp, -16
sw $ra, 0($sp)
sw $s0, 4($sp)
sw $s1, 8($sp)
sw $s2, 12($sp)

move $s0, $a0  # save a
move $s1, $a1  # save b

jal method3    # call method3(a)
move $s2, $v0  # save result

move $a0, $s1  # set up function call
jal method3    # call method3(b)

add $a0, $s2, $v0 # compute method3(a) + method3(b)
jal method2

move $a0, $s0
move $a1, $s1
move $a2, $v0  # put the return value into $a2 for method1
jal method1    # call method1

lw $ra, 0($sp)
lw $s0, 4($sp)
lw $s1, 8($sp)
lw $s2, 12($sp)
addi $sp, $sp, 16

jr $ra
```