Introduction to Computer Architecture - Midterm Exam (2023 Spring Semester)

Total 8 questions, 80 pts, Total 4 pages

IMPORTANT RULE: When writing a numeric value, you must explicitly indicate its base. Otherwise, it will be assumed that you are using the decimal representation. For example, hexadecimal numbers can be written with the prefix "0x" or the postfix subscript "$_{16}$". It is not necessary to add leading zeros to fill in the digits. (e.g., "0x00000041" can be written as "0x41", "41_{16}", "65", or "65_{10}". "41" is not "0x41")

1. True/False (10 pts, 2pts each)
   a. CPUs that use the little-endian utilize memory space more efficiently than the CPUs using the big-endian. **F**
   b. If you can place your computer in an extremely cold location, such as an arctic region, you may be able to reduce your electricity consumption when operating your computer system. **T**
   c. By utilizing stack memory, you can make recursive function calls without any limitation. **F**
   d. It is possible to execute any program using a single-cycle CPU, given that we have enough memory. **T**
   e. When returning from a procedure call, the `jalr` instruction should use the `x1` register as `rs1`. **F**

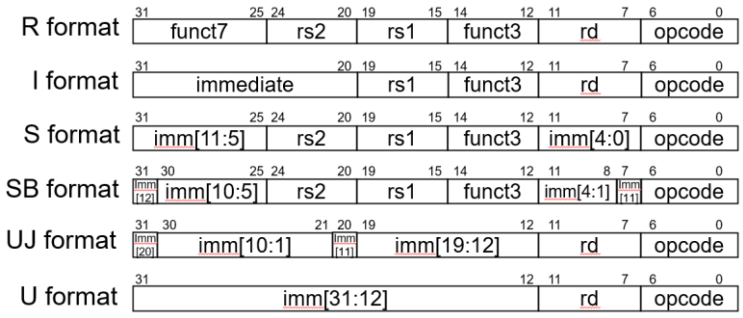
2. Your CPU has three types of instructions, **A**, **B**, and **C**. On average, these instructions are utilized 50%, 30%, and 20% of the total instructions, respectively. In the original design, the CPI for all instructions is 16 cycles.
   You are now designing a new CPU and have access to a *magic* tool that can reduce the CPI of an instruction by half. (e.g., 16 → 8, 8 → 4, …).
   a. If you can only use this magic tool once, which instruction type's CPI should you reduce? (5 pts) **A**
   b. If you can use this tool five times, what is the lowest average CPI you can achieve? (5 pts) **4.8**

3. Find the results of the following RISC-V program execution. Write the **hexadecimal values of the destination register** after executing each instruction one at a time (not the final value). Assume that you can freely read/write from any memory address. (10 pts, 1pts each)

`ori x10, x0, 0x200`	**x10: 0x200**
`ori x11, x0, -200`	**x11: 0xFFFFFF38**
`and x12, x10, x11`	**x12: 0x200**
`slli x13, x12, 1`	**x13: 0x400**
`lui x14, 0x8`	**x14: 0x8000**
`bge x12, x13, L1`	
`addi x14, x14, 0x7FF`	
`L1: addi x15, x14, 1`	**x15: 0x8800**
`slli x16, x15, 16`	**x16: 0x88000000**
`srai x17, x16, 24`	**x17: 0xFFFFFF88**
`andi x18, x17, 0xF`	**x18: 0x8**
`mul x19, x18, x11`	**x19: 0xFFFFF9C0**

4. Binary representation (10 pts, 5 pts each). Please use the following opcode/funct table and the RISC-V binary encoding formats. The codes are written in binary (base 2)

Instruction	Opcode	Funct3	Funct7
add	0110011	000	0000000
sub	0110011	000	0100000
sll	0110011	001	0000000
slt	0110011	010	0000000
lw	0000011	010	-
sw	0100011	010	-
addi	0010011	000	-
slti	0010011	010	-
slli	0010011	001	-
beq	1100011	000	
bne	1100011	001	
blt	1100011	100	
jal	1101111		
jalr	1100111	000	



a. Convert the following RISC-V assembly instruction "**bne**" into machine code format. Do not need to convert other instructions. Write the 32-bit machine code in **hexadecimal** number.

```
    add x1, x2, x3
TT:
    and x4, x5, x6
    or x7, x8, x9
    bne x10, x11, TT
```

→ **0xFEB51CE3**

b. Convert the following machine code written in hexadecimal number **into RISC-V assembly instruction**.

0x09B1A723 → **sw x27, 142(x3)**

5. Indicate whether the following five instructions are valid RISC-V RV32I instructions. If an instruction is not supported by the RISC-V RV32I CPU (such as pseudoinstructions), please mark it as invalid. (10 pts, 2pts each)

a. `xor x0, x0, x0` → **valid**

b. `lwu x10, 0(ra)` → **invalid**

c. `srai x31, x30, 0` → **valid**

d. `addi x10, x20, 5000` → **invalid**

e. `L1: beq x0, x0, L1` → **valid**

6. When the CPU is processing the following `jal` RISC-V instruction loaded from memory address `0xABC`, **write the values at the indicated datapath locations** in the figure (you need to write a total of 5 values and write in hexadecimal). FYI, `jal` instruction uses the "UJ" format in question 4. (10 pts, 2pts each)

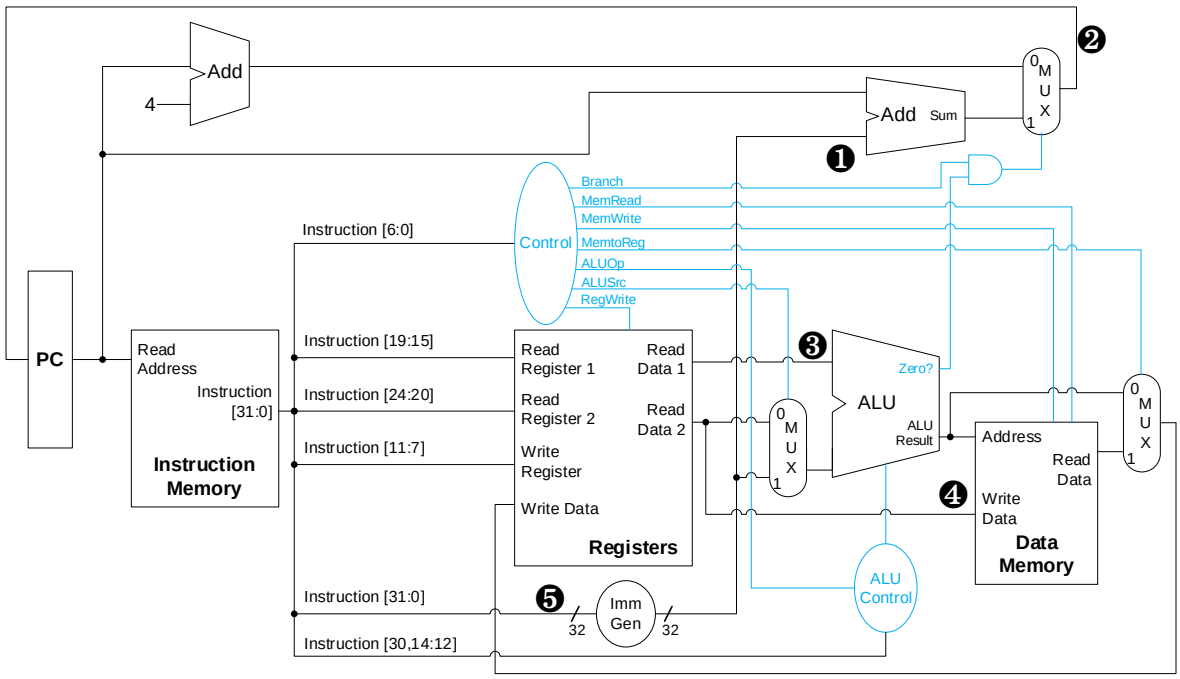
```
jal x10, F1                 ← Address: 0xABC

…

F1: add x1, x2, x3          ← Address: 0x1000
```

Currently, the register file has the following values:

x1:	0x12345678	x5:	0x0FF1CE	x15:	0x5678	x19:	0x11111111
x2:	0xFFFFFFFF	x6:	0xCAFE	x16:	0x90AB	x20:	0x0
x3:	0xFACEB00C	x7:	0xFEED	x17:	0xCDEF	x21:	0x87654321
x4:	0xDEADBEEF	x8:	0xF00D	x18:	0xF0F0	x22:	0xC0DE



1) 0x544

2) 0x1000

3) 0x0

4) 0xDEADBEEF

5) 0x5440056F

7. Large constant values

   a. Suppose you want to perform the following calculation: A = A + 10000_{10}.
   10000_{10} is 0x2710 in hexadecimal, which requires more than 12 bits to represent the constant value.

   Write RISC-V assembly code that performs the calculation in 3 instructions. Assume "A" is in x9, and you can freely use one of the temporary registers (`t0-t6`, or `x5-x7, x28-x31`) (5 pts).

```
lui x5, 0x2
addi x5, x5, 0x710
add x9, x9, x5
```

   b. Now, you want to perform the same calculation as above, but with a different constant value 20000_{10}. 20000_{10} is 0x4E20 in hexadecimal. Write the RISC-V assembly code for this case, again in 3 instructions as before (5 pts).

```
lui x5, 0x5
addi x5, x5, -0x1E0
add x9, x9, x5
```

8. The RISC-V datapath we studied in class includes three ALU-like structures: the main ALU, an ADD unit for PC+4, and an ADD unit for PC+immediate.

   Suppose we want to fix the architecture to remove the third ALU, the ADD unit that performs PC + immediate. However, to process a normal beq instruction, you need all three ALUs. Therefore, in the modified CPU, we will introduce a new conditional branch instruction called 'bz.'

   bz will branch if the lowest bit of the `rs1` register is zero. It does not use `rs2` or other bits of the `rs1` register. What modifications are required to implement this change in the CPU architecture? Draw the modified datapath or explain (10 pts). (You may use English or Korean)