CS2200 Spr24 Exam 1: Released

Question #	Topic	Total
1	Switch Statement / Jump Table	10
2	Datapath	12
3	*Hidden*	10
4	Interrupts 2	8
5	Performance Metrics 1	10
6	Performance Metrics 2	10
7	Structs and Endianness	10
8	Datapath Delay	10
9	*Hidden*	10
10	*Hidden*	10
Total		100

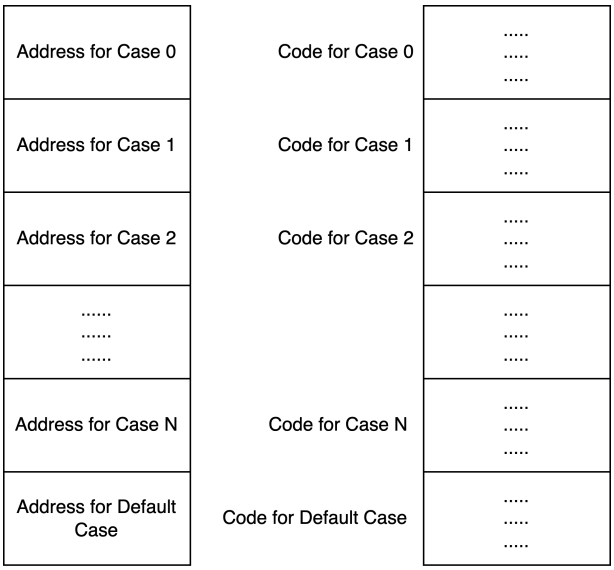
Note: Released questions will not change qualitatively, but can change quantitatively. For example, numbers, instructions, order of events, etc. may be different in the exam. The exam uses the **LC-2200** instruction set, with the addition of the LEA and BGT instructions. See the appendix for more details.

Question 1 (10 points):

High-level languages provide a “switch” statement that looks as follows.

```
switch(k) {
      case 0:
      case 1:
      case 2:
      ...
      case N:
      default:
}
```

The compiler writer Kaylia knows that “k” can take non-negative contiguous integer values from 0 to N during execution, with any value greater than N going to the default case. She decides to use a jump table data structure (implemented as an array indexed by the value contained in k) to hold the start address for the code for each of the case values as shown below:



Jump Table

Assume we are using the **modified LC-2200 instruction set architecture**, which can be found in the appendix.

Q1.1 (4 points)

Assume the **base address of the jump table** is stored in the register **$t0**, the value of **N** is stored in register **$a0**, and the **value of k** is stored in the register **$t1**. Help Kaylia out by writing a series of LC-2200 instructions for any positive value of k.

Q1.2 (6 points)

Instead of using a jump table, her friend suggests using a series of conditional branch instructions to compile the switch statement. List one reason why conditional branches could be better, and one reason why switch statements could be better. Explain your answer, clearly stating your assumptions.

Question 2 (12 points):

You are given 23 clock cycles of control signals which is a subsection of the execution macro states of some assembly code. Fill out the **4** lines of LC-2200 assembly instructions that correspond to the control signals. Note that the clock cycles corresponding to the fetch macrostate have been omitted.

_____
_____
_____
_____

```
.fill Label_X 0x1111
```

Cycle 4: DrREG, LdA, RegSel = $t0
Cycle 5: DrREG, LdB, RegSel = $t0
Cycle 6: ALU=add, DrALU, WrREG, RegSel=$t1

Cycle 10: DrREG, LdA, RegSel = $t1
Cycle 11: DrREG, LdA, RegSel = $t1
Cycle 12: ALU=add, DrALU, WrReg, RegSel=$t1

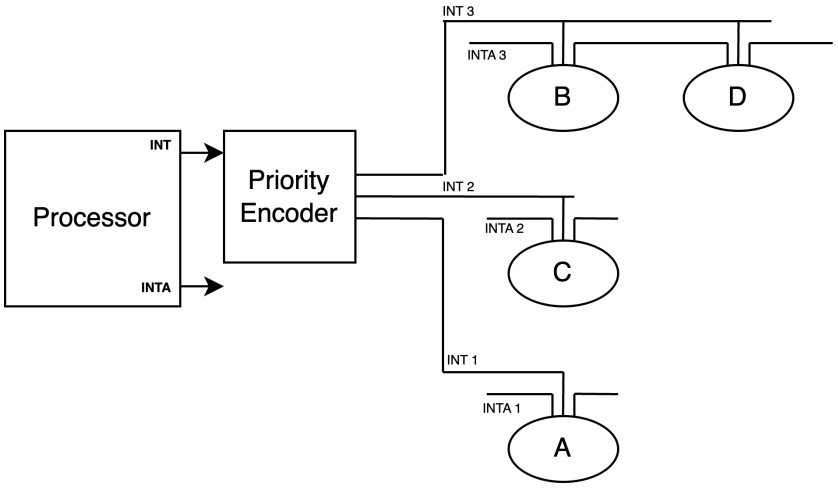
Cycle 16: DrREG, LdA, RegSel = $t1
Cycle 17: DrOFF, LdB [Offset = 1]
Cycle 18: ALU=add, DrALU, WrREG, RegSel = $t2

Cycle 22: DrPC, WrREG, RegSel=$at
Cycle 23: DrREG, LdPC

Question 4 (8 points):

Consider an architecture that has 3 interrupt priority levels. The interrupt handler for every device enables interrupts before executing the device-specific code.

The below diagram shows four devices. **Smaller numbers represent a higher priority level.** B is electrically closer to the processor than D. The INTA line is chained through devices B and D.



A schedule of signals is shown below. Each handler takes 1000 cycles.

Time	**Interrupted By**	**INTA**
0	C	
100	D	
200		INTA
300	B	
400	A	
500		INTA
600		INTA
...		
900		INTA

Q4.1 (4 points)
In what order are the three devices acknowledged? Explain your answer.

Q4.2 (4 points)
Assuming that these are the only interrupts, in what order do the interrupt handlers complete? Explain your answer.

Question 5 (10 points):

Kaylia has a program that consists of the following **3** lines, but repeats **220** times.

```
add $t0, $t0, $t0
sw $t0, 0($sp)
addi $sp, $sp, -1
```

In this implementation, the **add** instruction takes **4** clock cycles. The **sw** instruction takes **5** clock cycles. The **addi** instruction takes **3** clock cycles. Each clock cycle takes **7** nanoseconds.

Q5.1 (2 points)

What is the execution time of this program?

Q5.2 (2 points)

If Kaylia changes her implementation so the **sw** instruction now takes **3** clock cycles, what is the new execution time? Round to the nearest nanosecond.

Q5.3 (3 points)

What is the **speedup** achieved after this change? Leave your answer as a fraction.

Q5.4 (3 points)

What is the **improvement** in execution time after this change? Leave your answer as a fraction.

Question 6 (10 points):

Consider the following program that contains **600** instructions:

```
I1:
I2:
I3:
…
I330:
I331: LEA
I332:
…
I343:
I344: COND BR to I330
I345:
I346:
……..
I599: HALT
I600: LEA
```

LEA instruction occurs exactly twice in the program as shown. Instructions 330-344 constitute a loop that gets executed 150 times. All other instructions execute exactly once. Leave your answer as a fraction.

Q6.1 (5 points)

What is the **static** frequency of LEA instruction? Show your work for credit.

Q6.2 (5 points)

What is the **dynamic** frequency of LEA instruction? Show your work for credit.

Question 7 (10 points):

For the struct defined below, show how a smart compiler might pack the data to minimize wasted space and follow alignment restrictions. Pack in such a way that each member is **naturally aligned** based on its data type. Assume the compiler will not reorder fields of the struct in memory. Assume a char is 1 byte, an int is 4 bytes, and a short is 2 bytes, and an int64_t is 8 bytes. Moreover, assume the CPU architecture is **big-endian** and its granularity supports load word (LW), load byte (LB), and load half-word (LHW), where a memory word is 4 bytes.

```
struct x {
    int a;
    int64_t b;
    char d;
    short e[1];
}

data = {
     0xBEEF2D6F,
     0x021420241324B9CF,
     0x24,
     {0x0C67}
};
```

Q7.1 (10 points)
Assume struct **data** is saved at **0x1000**. Fill out this table with the hex values of the data stored at each memory address

+0	**+1**	**+2**	**+3**	**Starting Address**
				0x1000
				0x1004
				0x1008
				0x100C
				0x1010

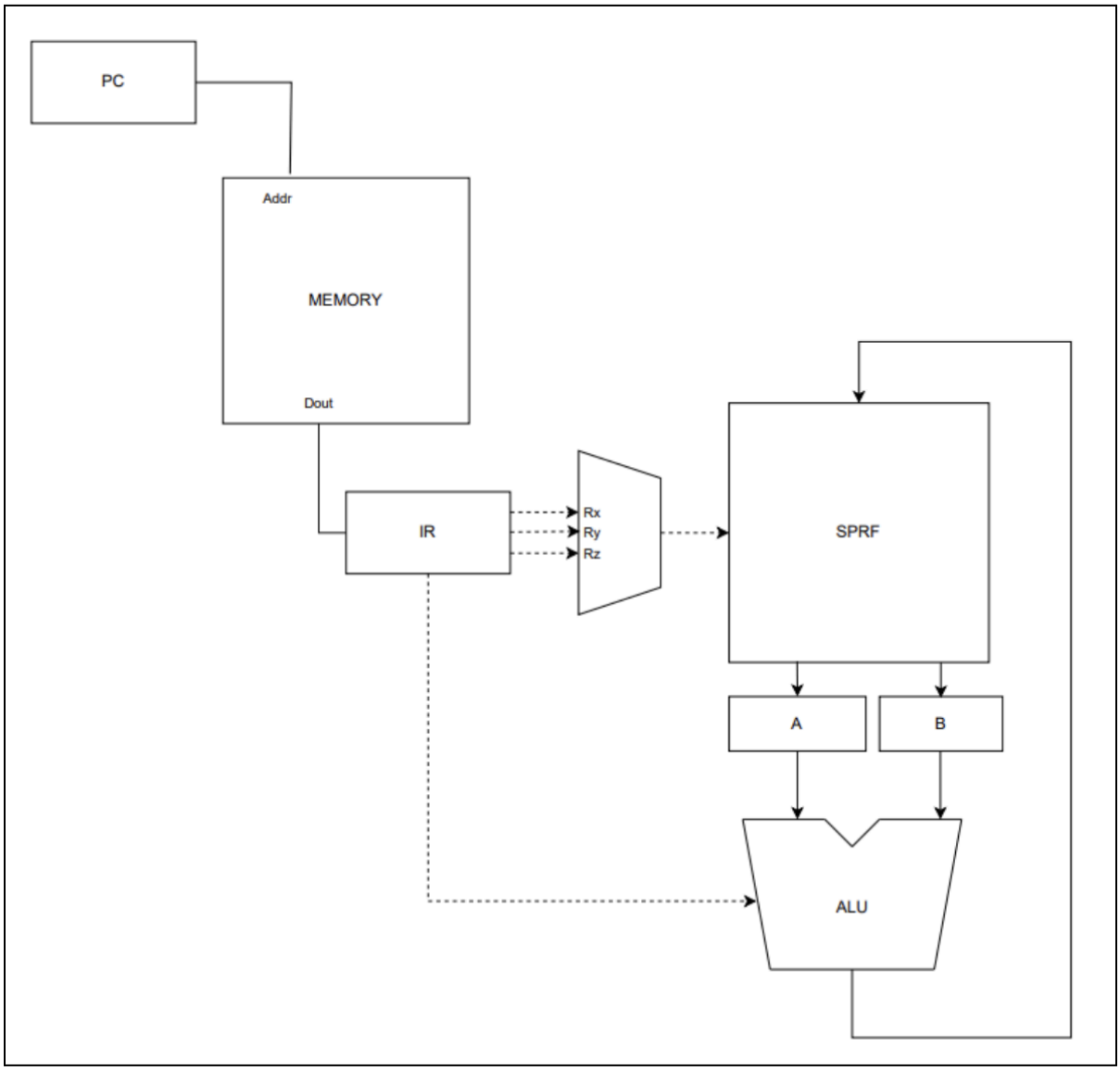
Q7.2 (2 points)
CPU issues the following instruction: `LW R1, MEM[0x1008].` Show the content of **R1**.

Q7.3 (1 point)
You are a clever programmer who knows the architectural details and how the compiler will pack the variables to optimize on space. Reorder the elements of struct **data** such that it will result in optimal space usage.

Q7.4 (1 point)
How many bytes do you save by reordering the elements of struct **data**?

Question 8 (10 points):

Consider the datapath with the delays shown below. Solid lines are data lines; dashed lines are control lines. All times are given in picoseconds - e.g., reading from or writing to the register file takes 2 picoseconds (2x10^-12s). What is the minimum clock cycle in ps?



Component	Delay (ps)	Notes
Wire	2	
ALU	3	Combinational
Register hold + setup	4	For IR,PC,A,B
Register output-stable	2	For IR,PC,A,B
Memory	200	Level Logic
SPRF Access (read or write)	3	
MUX	2	

Appendix:

LC 2200 ISA with an additional LEA and BGT instruction		
Mnemonic Example	**Opcode (Binary)**	**Action Register Transfer Language**
add add $v0, $a0, $a1	0000	Add contents of reg Y with contents of reg Z, store results in reg X. RTL: $v0 ← $a0 + $a1
nand nand $v0, $a0, $a1	0001	Nand contents of reg Y with contents of reg Z, store results in reg X. RTL: $v0 ← ~($a0 && $a1)
addi addi $v0, $a0, 25	0010	Add Immediate value to the contents of reg Y and store the result in reg X. RTL: $v0←$a0+25
lw lw $v0, 0x42($fp)	0011	Load reg X from memory. The memory address is formed by adding OFFSET to the contents of reg Y. RTL: $v0 ← MEM[$fp + 0x42]
sw sw $a0, 0x42($fp)	0100	Store reg X into memory. The memory address is formed by adding OFFSET to the contents of reg Y. RTL: MEM[$fp + 0x42] ← $a0
beq beq $a0, $a1, done	0101	Compare the contents of reg X and reg Y. If they are the same, then branch to the address PC+1+OFFSET, where PC is the address of the beq instruction. RTL: if($a0 == $a1) PC ← PC+1+OFFSET
jalr jalr $at, $ra	0110	First store PC+1 into reg Y, where PC is the address of the jalr instruction. Then branch to the address now contained in reg X. Note that if reg X is the same as reg Y, the processor will first store PC+1 into that register, then end up branching to PC+1. RTL: $ra ← PC+1; PC ← $at Note that an **unconditional jump** can be realized using **jalr $ra, $t0,** and discarding the value stored in $t0 by the instruction. This is why there is no separate jump instruction in LC-2200.
halt	0111	Halt the machine

bgt bgt $a0, $a1, done	1000	Compare the contents of reg X and reg Y. If the value in reg X is greater than the value in reg Y, then branch to the address PC+1+OFFSET, where PC is the address of the bgt instruction. RTL: if($a0 > $a1) PC ← PC+1+OFFSET
lea lea $a0, stack	1001	An address is computed by sign-extending bits [19:0] to 32 bits and adding this result to the incremented PC (address of instruction + 1). It then stores the computed address into register DR. RTL: $a0 = MEM[stack]