Test 1

Due Feb 21, 2022 at 11:59pm **Points** 100 **Questions** 11
Available Feb 18, 2022 at 11:59pm - Feb 21, 2022 at 11:59pm **Time Limit** 60 Minutes

Instructions

Instructions

The exam will be available as a canvas quiz and proctored using Honorlock. The exam time is **60 minutes** and **closed everything, except for a blank sheet of scratch paper and a calculator.** The exam will be over either after the 60 minutes or 11:59 PM on Monday, whatever comes first. Make sure to start the exam at least an hour before the deadline.

This quiz is no longer available as the course has been concluded.

Attempt History

	Attempt	Time	Score
LATEST	Attempt 1	60 minutes	67 out of 100

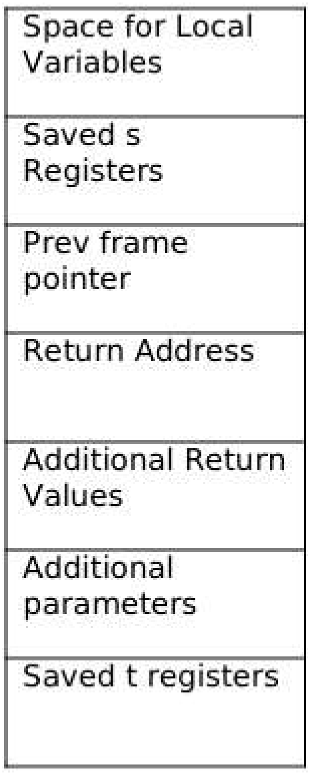
Score for this quiz: **67** out of 100
Submitted Feb 21, 2022 at 11:58pm
This attempt took 60 minutes.

Problem	**(Points, Minutes)**
Question 1: Calling Convention	12, 7
Question 2: Switch Statement and Jump Table	10, 5
Question 3: Datapath	12, 7
Question 4: Halt Instruction	12, 7
Question 5: Interrupts 1	6, 3
Question 6: Interrupts 2	6, 4
Question 7: Performance Metrics 1	6, 3
Question 8: Performance Metrics 2	6, 5
Question 9: Performance Metrics 3	10, 7
Question 10: Hidden	10, 7
Question 11: Hidden	10, 5
Total	100, 60

Question 1

12 / 12 pts

Recall that the stack is used for communication between the caller and the callee. It stores information as shown below:



Space for Local Variables

Saved s Registers

Prev frame pointer

Return Address

Additional Return Values

Additional parameters

Saved t registers

(i) (4 points) Before executing JALR, the caller saves **$ra** (return address) on the stack. Explain why.

(ii) (4 points) Explain the benefit of dividing the register save/restore chore between the caller and the callee.

(iii) (4 points) Explain the benefit of having a **frame pointer**. Who saves the "prev frame pointer" on the stack? In the above picture, show where the frame pointer is currently pointing to on the stack.

Your Answer:

(i) When a program wants to execute an instruction in another part of memory, it needs to know two things 1.) Where to jump to and 2.) where to return to. Register $ra solves the second problem. It is a location to store the address to return to once we finish executing instructions at the target address

As for why it is stored on the stack, if when we jump we run into another jalr instruction, the contents of $ra will be overwritten, we need to store $ra on the stack so we don't lose our original return address.

(ii) It is for efficiency. By dividing up the labor between caller and callee, we can choose to save registers that are only pertinent to either party. For example, if the caller only uses one t register, then it does not matter what the contents of t1 and t2 are. Therefore, it would be a waste of time and space for the callee to save all t registers.

(iii) The framer pointer acts as a constant pointer to the start of the activation record. It is not to be confused with the stack pointer that points to the top of the stack (the "space for local variables box " in the above diagram). We need the frame pointer because there is too much overhead in trying to keep track of offsets to certain values in the stack when values are allocated and deallocated onto the stack. The frame pointer is saved by the callee and points to the start of the activation record (the saved t registers box in the above diagram)

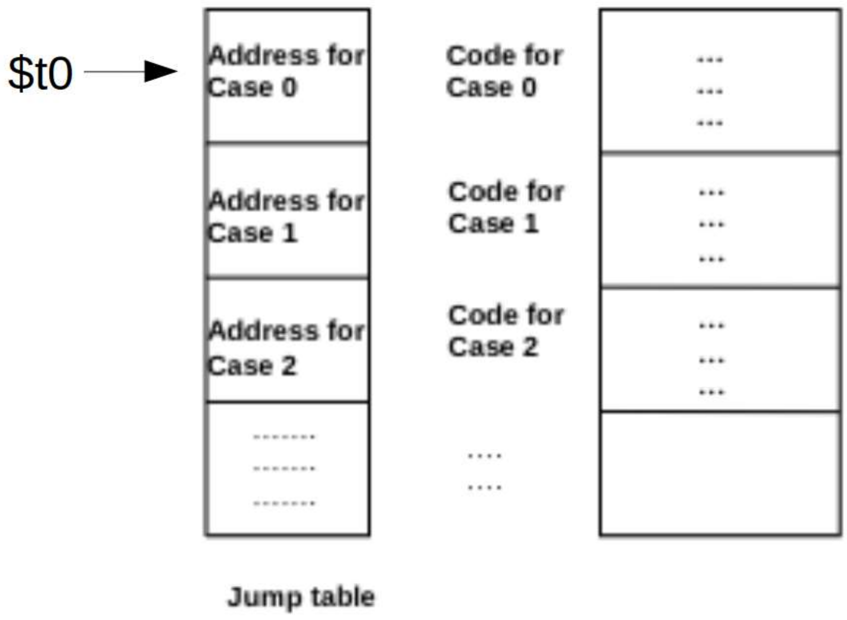
Question 2

10 / 10 pts

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

```
switch (k) {
        case 0:
        case 1:
        case 2:
        case 3:
        ......
}
```

The compiler writer knows that “k” can take contiguous integer values from 0 to 9 during execution. 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:



Assuming we are using the LC-2200 instruction set architecture (See Appendix) The architecture is word addressable.

(i) (5 points) Assume the **base address of the jump table** is stored in the register **$t0**, the value of k is stored in register $t1. Write a series of instructions that reaches the code for **case k.**

(ii) (5 points) Can this implementation of a switch statement be simulated by a series of conditional branch instructions without accessing memory? If so, which approach is more time-efficient? Why? (Hint: think about space and time complexity)

Your Answer:

(i)

add $at, $t1, $t0

lw $at, 0($at)

jalr $at, $ra

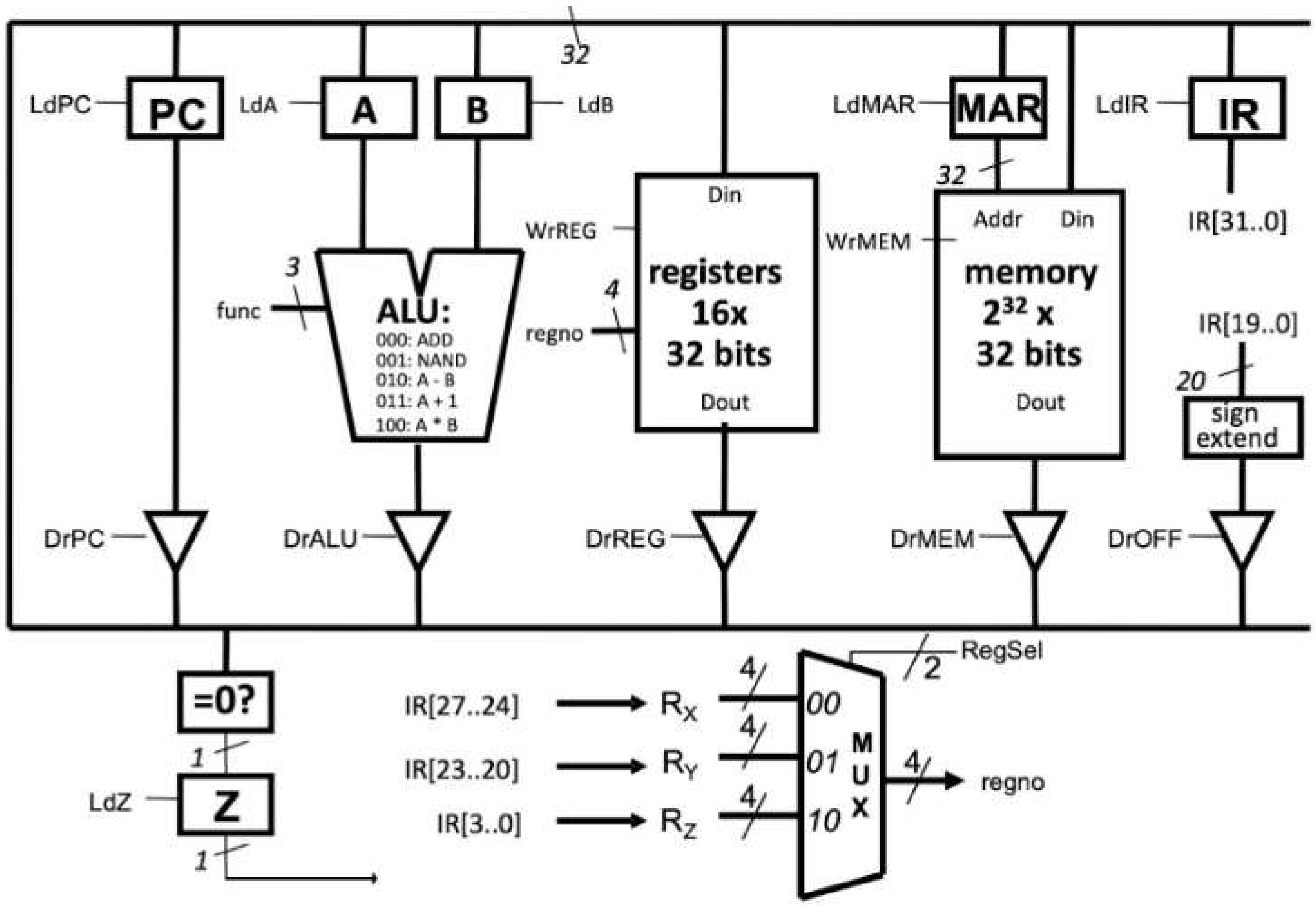
(ii)

Yes, it can be implemented using a series of conditional branch instructions. However, this would be more time-inefficient. The time complexity of going through each case is O(n) where n is the k_max (in case (i) it is O(1) so this implementation of using branches takes longer). Space efficiency-wise, both part (i) and part (ii) are the O(1).

Question 3

11 / 12 pts

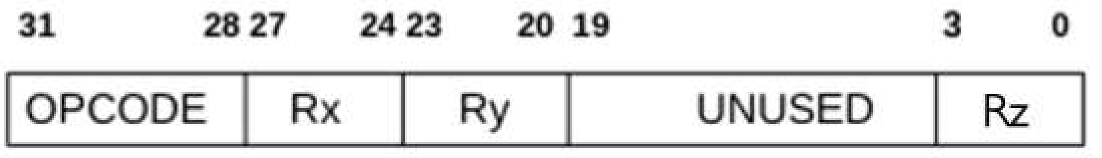
The datapath shown below corresponds to the familiar LC-2200, 32-bit word addressable architecture, with the difference that the ALU also supports a multiply operation, selected by setting the ALU’s *func* control signal to 100.



We are introducing a new instruction MULTADD Rx, Ry, Rz to the LC-2200 ISA. The semantics of the instruction is as follows:

Rx <- Rx + (Ry * Rz)

The instruction's format is as shown below:



31 28	27 24	23 20	19	3 0
OPCODE	Rx	Ry	UNUSED	Rz

Given the above datapath, write the sequence of micro-states and signals within that are required to implement the **MULTADD** instruction (you **ONLY** need to write the sequence for the **execution macrostate of the instruction**). For each microstate, show the datapath action (in register transfer format such as A ← Ry), followed by the set of control signals you need to enable for the datapath action (such as DrALU). **NOTE: In each microstate, state the value for all control signals. Signals you don't explicitly specify will be assumed to be taking the value 0.**

Your Answer:

We don't have to do ifetch1, ifetch2, or ifetch3

process:

1.) store Ry into A

2.) store Rz into B

3.) compute A * B and store in A

4.) store Rx into B

5.) Compute A + B and store into REG FILE

multadd1: DrREG, LdA, RegSel = 01   (A ← Ry)

multadd2: DrREG, LdB, RegSel=11   (B ← Rz)

multadd3: DrALU, LdA, func=100,   (A ← Ry * Rz = A * B)

multadd4: DrREG, LdB, RegSel=00   (B ← Rx)

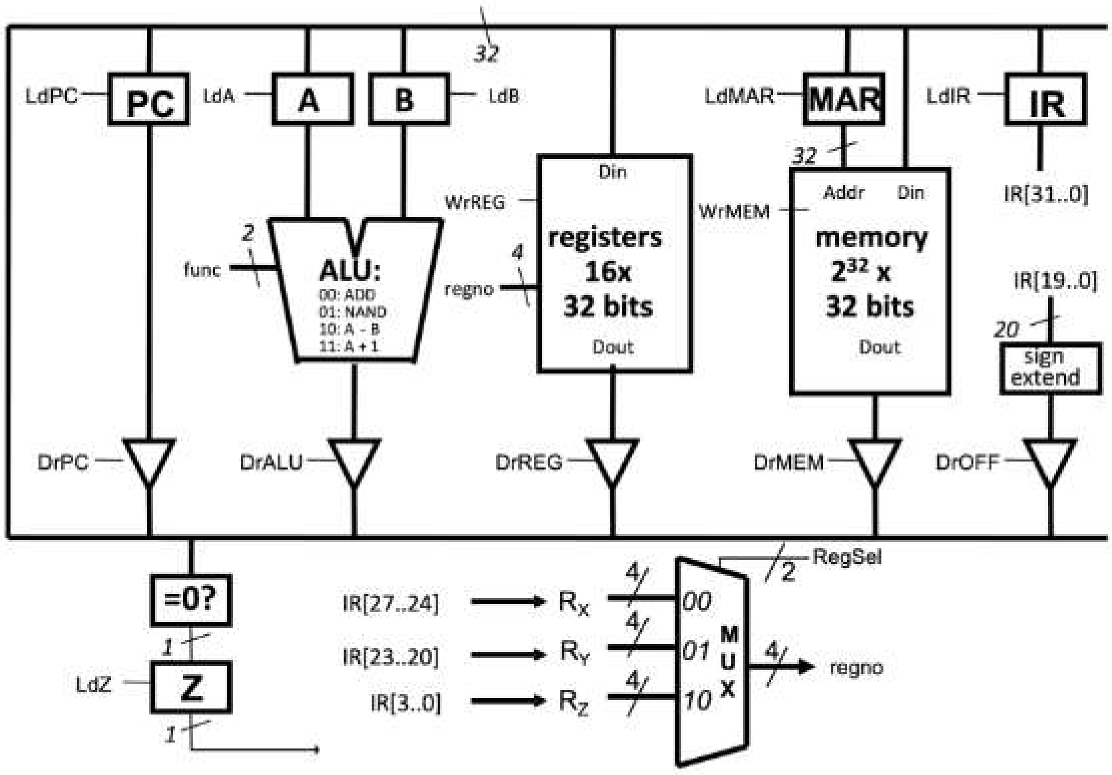
multadd5: DrALU, WrREG, RegSel=00, func=000,   (Rx ← Rx + (Ry * Rz) = A + B)

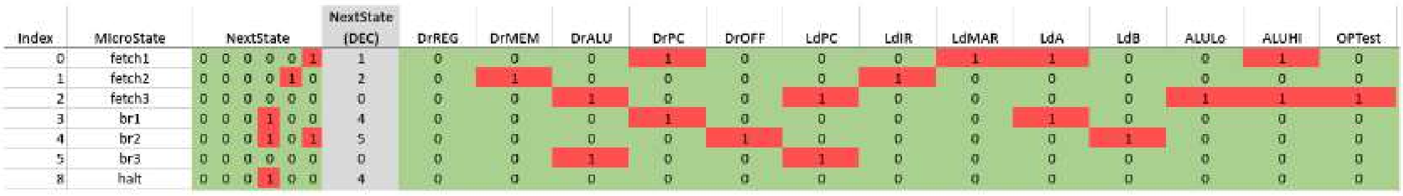
Register Rz is RegSel=10.

Question 4

12 / 12 pts

Assume the following datapath of LC-2200 and part of microcode. Take a look at the “NextState” of the halt instruction.





Index	MicroState	NextState	NextState (DEC)	DrREG	DrMEM	DrALU	DrPC	DrOFF	LdPC	LdIR	LdMAR	LdA	LdB	ALULo	ALUHI	OPTest
0	fetch1	0 0 0 0 0 1	1	0	0	0	1	0	0	0	1	1	0	0	1	0
1	fetch2	0 0 0 0 1 0	2	0	1	0	0	0	0	1	0	0	0	0	0	0
2	fetch3	0 0 0 0 0 0	0	0	0	1	0	0	1	0	0	0	0	1	1	1
3	br1	0 0 0 1 0 0	4	0	0	0	1	0	0	0	0	1	0	0	0	0
4	br2	0 0 0 1 0 1	5	0	0	0	0	1	0	0	0	0	1	0	0	0
5	br3	0 0 0 0 0 0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
8	halt	0 0 0 1 0 0	4	0	0	0	0	0	0	0	0	0	0	0	0	0

(i) (6 points) Would this microcode achieve the intended semantics of halt? Describe in detail. (Hint: the instruction hex for halt is always 0x70000000.)

(ii) (6 points) Would the microcode achieve the intended semantics of halt if the halt micro-state’s “NextState” value was **5** (i.e., *br3*) instead of 4 (i.e., *br2*)? Describe why or why not.

Your Answer:

(i) Yes. Before the execution macrostate of halt, the fetch macrostate of halt will load the current value of the PC into register A. Currently, halt's next state is br2. br2 will store the 20-bit offset of the halt instruction (which is zero since the last 20-bits of 0x70000000) into register B and then proceed to br3. br3 will add the contents of A and B and (A=PC and B=0 therefore A+B=PC) then store it back in the PC (thus, we arrive at our original value of the PC). Before we overwrote the PC, the value of PC+1 was in the PC from the first halt macrostate, but we just overwrote it with the value of only PC (thus we have no incremented the PC). From here, the fetch macrostate of the next instruction will begin and we will load the value of the PC back into A. ifetch2 begins and we then get the instruction from MEMORY and place it in the IR. Then, we increment the PC and move on, again, to the halt instruction, however, since we never changed the original value of the PC out of register A, we will effectively loop through the fetch macrostate, halt, and br2 and br3 forever. The PC will never get incremented and we will never execute another instruction. Therefore we have achieved the semantics of halt.

(ii) No. br2 is the instruction that sets the value of register B to zero by taking the last 20-bits of halt (which is always zero) and setting it to the contents of register B. If we skip br2 and go right to br3 then we don't set register B to zero and we do not know the value of register B. If it is anything other than zero, we will not arrive back at the original value of the PC after br3.

Question 5

3 / 6 pts

Consider an architecture that has *8 priority levels*. The interrupt handler for every device enables interrupts for **higher priority** levels before executing the device-specific code.

Two devices -- timer and keyboard are the **same** lowest priority level. The timer device is electrically **closer** to the processor. The INTA line is chained through the two devices.

(i) (3 points) Both devices simultaneously assert the interrupt line. Whose interrupt will be serviced first?

(ii) (3 points) When will the second device get serviced?

Your Answer:

(i) Because the Timer device is closer, the timer is serviced first.

(ii) First higher-prioritied interrupt interrupts that occur during the timer interrupt will be serviced (if any). Once all those are over, any other device that is on the same lowest priority level will be serviced (if any). Then the second device will be get serviced.

The second device will only be serviced after the timer interrupt handler code finishes, it cannot interrupt during the timer interrupt.

Question 6

2 / 6 pts

In LC-2200, **$k0** contains the address to which the processor has to **return** from an interrupt handler. Before returning from the handler, the interrupts have to be **enabled**. Thus we can return from the interrupt by executing the following two instructions:

- Enable interrupt
- Jump via $k0

What is the problem with this idea?

Your Answer:

If multiple interrupts had occurred, we lose the original value of $k0.

Please elaborate on the reason why we lose the original $k0

Question 7

6 / 6 pts

Consider the following program that contains 1000 instructions:

```
I1:
I2:
I3:
…
…
I110:
I111: NAND
I112:
…
…
I143                            loop (I110 to I144)
I144: COND BR to I110
I145
…
…
I1000
```

NAND instruction occurs exactly **once** in the program as shown. Instructions I110–I144 constitute a loop that gets executed **250** times. All other instructions execute exactly once.

(i) (2 points) What is the static frequency of NAND instruction?

(ii) (4 points) What is the dynamic frequency of NAND instruction?

Note: Show your work in detail for credit

Your Answer:

(i) instruction occurs 1 time in 1000 instructions

1/1000 = 0.001 or 01%

(ii) instruction occurs 250 times over the course of the program

total instructions executed = (1000 - 35) + (250*35) = 965 + 8750 = 9715

250/9715 = 0.0257 or 2.57%

250*35 since that's how many instructions are executed inside the loop (144-110 + 1 (add the one since both instructions 110 and 144 are executed so this is an inclusive range)

1000 - 35 since there are 1000 static instructions total and 35 of which are inside a loop (that we've already accounted for)

Question 8

6 / 6 pts

An architecture has three types of instructions that have the following CPI:

Type	CPI
A	4
B	2
C	6

An architect determines that she can reduce the CPI for C to **4**, with no change to the CPIs of the other two instruction types, but with an increase in the clock cycle time of the processor. What **maximum permissible** increase in clock cycle time will make this architectural change worthwhile? Assume that all the workloads executing on this processor use 40% of A, 30% of B, and 30% of C types of instructions.

Note: Show your work in detail for credit

Your Answer:

old clock cycle time: 4(0.4) + 2(0.3) + 6(0.3) = 1.6 + 0.6 + 1.8 = 2.2 + 1.8 = 4

new clock cycle time: 4(0.4) + 2(0.3) + 4(0.3) = 2.2 + 1.2 = 3.4

$4C_{old} = 3.4C_{new}$

$$\boxed{\frac{4}{3.4} = \frac{C_{new}}{C_{old}} = 1.1765}$$

max increase in the clock cycle is 17.65%

Question 9

0 / 10 pts

A program spends **25%** of its runtime in multiplications. LC-2200 simulates the multiplication instructions through a sequence of additions. A student in CS 2200 decides to add a MULT instruction to LC-2200, which does **not affect** the execution time of any other instructions in LC-2200. How much faster should MULT instruction be compared to the simulated code sequence for the program's overall speedup of 1.25?

Note: Show your work in detail for credit

Your Answer:

rate, use Harmonic Mean

1/(weight/rate)

$$\boxed{\frac{1}{\frac{0.25}{x+0.75}} = 1 \, + \, 0.25}$$

$$\boxed{\frac{x+0.75}{0.25} = 1.25}$$

$$\boxed{x + \frac{3}{4} = \frac{1}{4}\left(\frac{5}{4}\right)}$$

[-10] Need to use Amdahl's law

Question 10

0 / 10 pts

The struct defined below shows 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, int is 4 bytes, and a short is 2 bytes. Moreover, assume the CPU architecture is **little-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;      // a = 0x11001100
    char b;     // b = 0x41
    int c;      // c = 0x12345678
    short d;    // d = 0x6164
    short e[2]; // e = {0x8765, 0x4321}
}
```

(i)

+3	+2	+1	+0	Starting Address

0x11	0x00	0x11	0x00	0x1000
n/a	n/a	n/a	0x41	0x1004
0x12	0x34	0x56	0x78	0x1008
0x87	0x65	0x61	0x64	0x100C
n/a	n/a	0x43	0x21	0x1010

(i) CPU issues: **LW R1, MEM{0x100C}**. Show the content of **R1:** 0x64

Answer 1:

You Answered 0x11

Correct Answer 0

Answer 2:

You Answered 0x00

Correct Answer 0

Answer 3:

You Answered 0x11

Correct Answer 0

Answer 4:

You Answered 0x00

Correct Answer 0

Answer 5:

You Answered n/a

Correct Answer 0

Answer 6:

You Answered n/a

Correct Answer 0

Answer 7:

You Answered n/a

Correct Answer 0

Answer 8:

You Answered 0x41

Correct Answer 0

Answer 9:

You Answered 0x12

Correct Answer 0

Answer 10:

You Answered 0x34

Correct Answer 0

Answer 11:

You Answered 0x56

Correct Answer 0

Answer 12:

You Answered 0x78

Correct Answer 0

Answer 13:

You Answered 0x87

Correct Answer 0

Answer 14:

You Answered 0x65

Correct Answer 0

Answer 15:

You Answered 0x61

Correct Answer 0

Answer 16:

You Answered 0x64

Correct Answer 0

Answer 17:

You Answered n/a

Correct Answer 0

Answer 18:

You Answered n/a

Correct Answer 0

Answer 19:

You Answered 0x43

Correct Answer 0

Answer 20:

You Answered 0x21

Correct Answer 0

Answer 21:

You Answered 0x64

Correct Answer 0

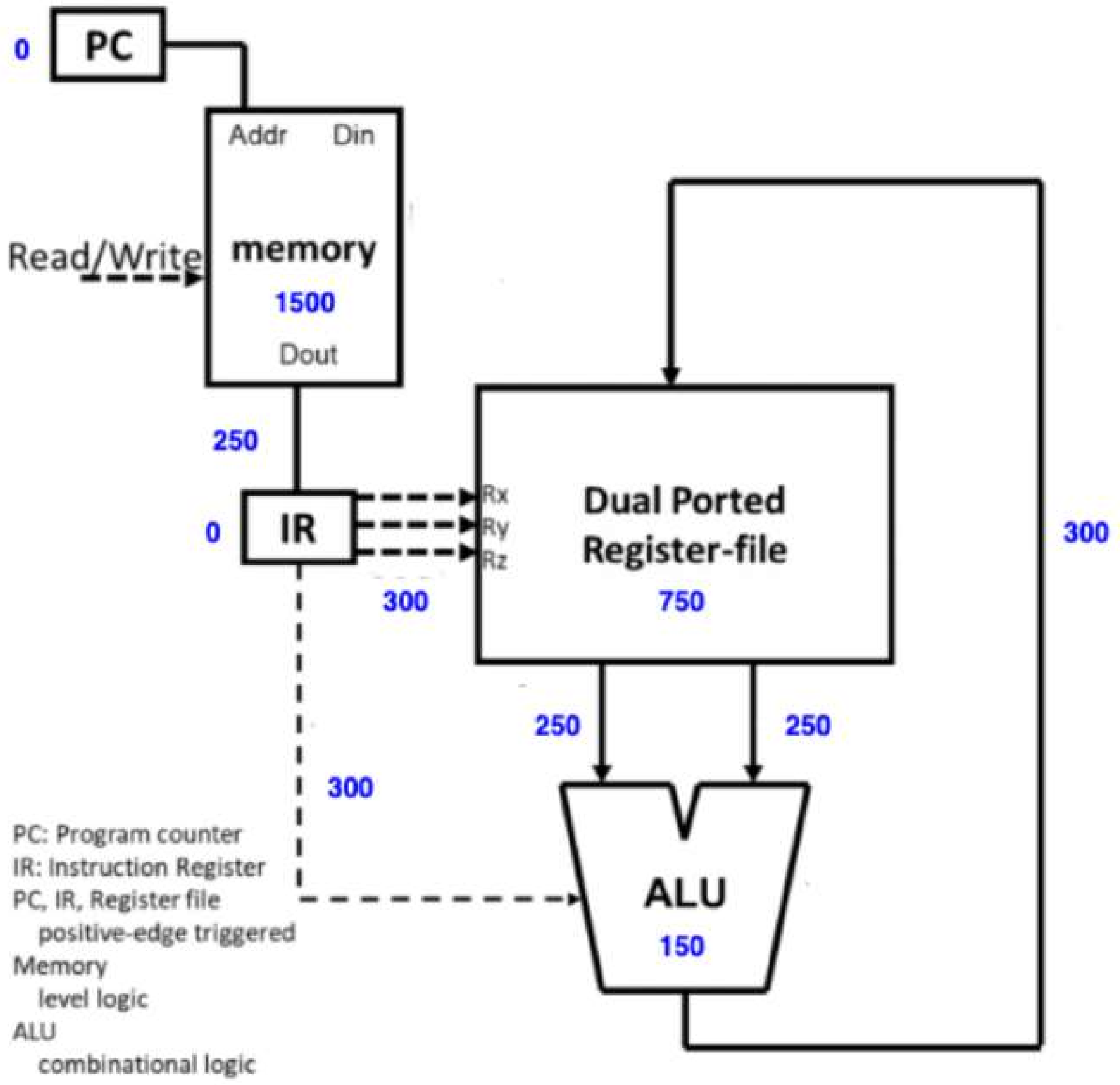
Correct Answer 0

loads byte instead load word

Question 11

5 / 10 pts

Consider the datapath with the delays shown below. Solid lines are data lines; dashed lines are control lines. All times are given in **nanoseconds** - e.g., reading from or writing to the register file takes 750 nanoseconds (1x10-9s).



What is the datapath's maximum possible clock frequency in **kHz**?

Note: Show your work in detail for credit

Your Answer:

i1: 1500 + 250 = 1750

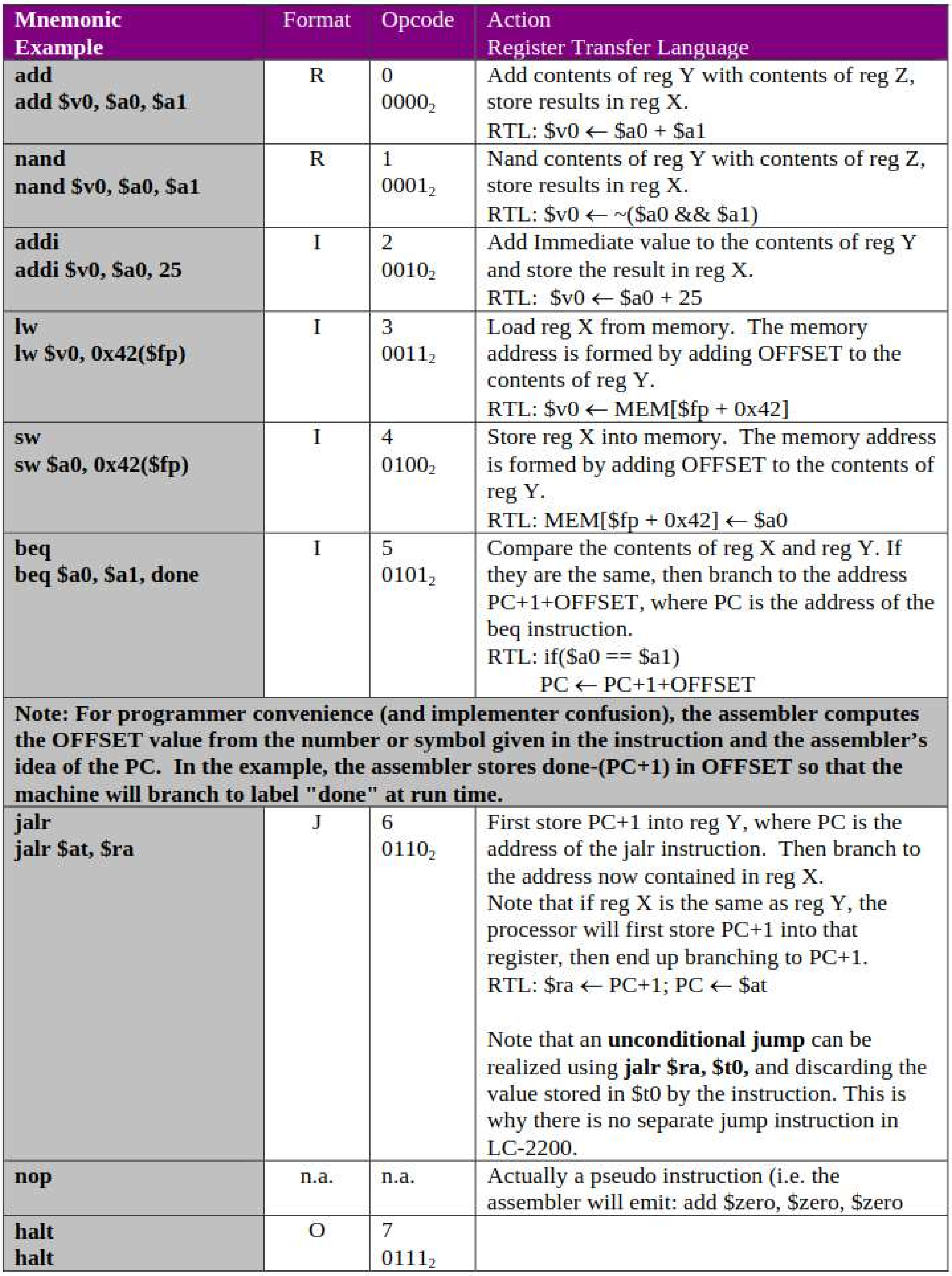
i2: 300 + 750 + 250 + 150 + 300 = 1450

i3: 300 + 750 = 1050

max of all instructions is i1

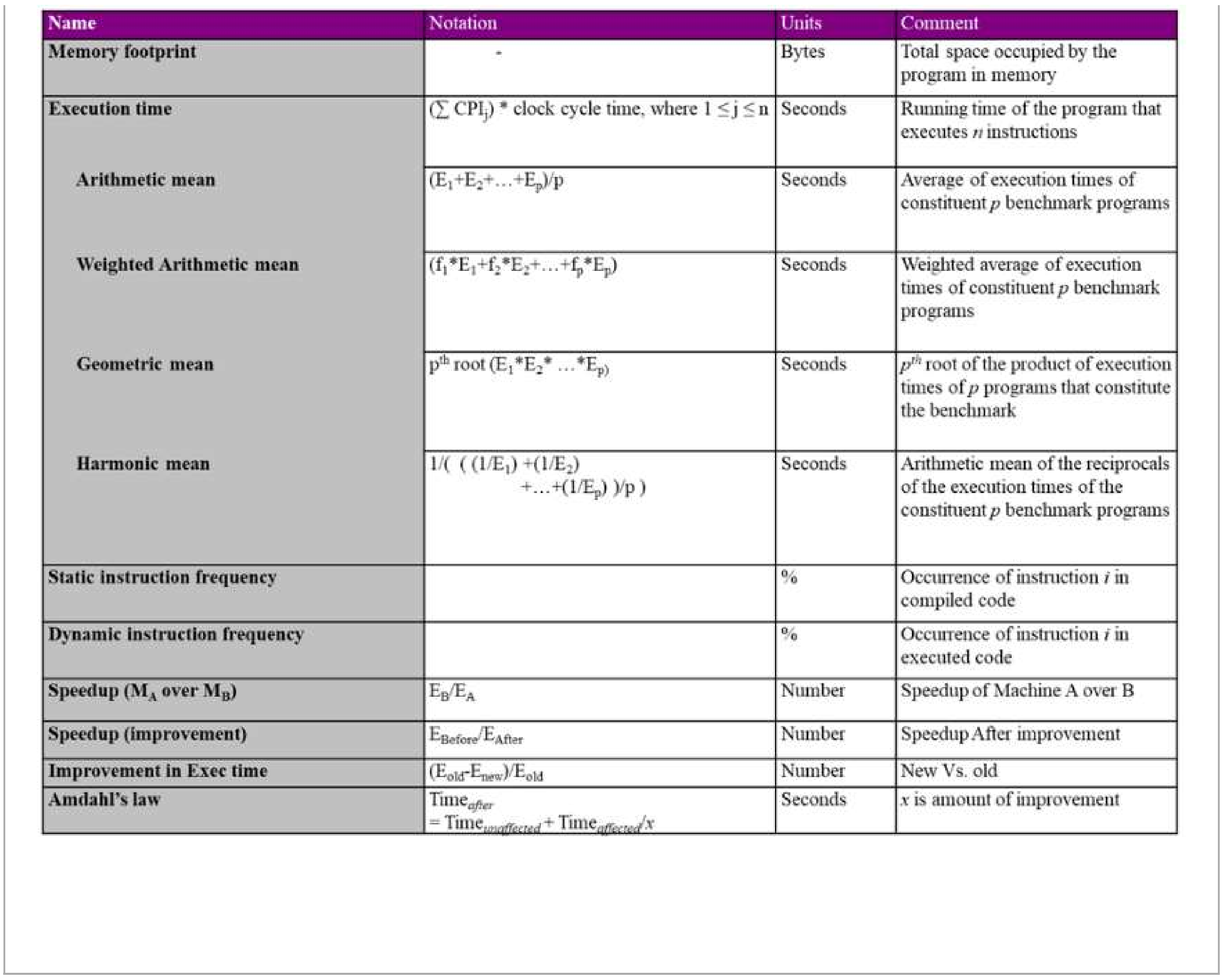
1/(1750 ns) = 0.00057143 GHz = 571.428571 kHz

LC2200 ISA



Mnemonic Example	Format	Opcode	Action Register Transfer Language
add **add $v0, $a0, $a1**	R	0 0000_2	Add contents of reg Y with contents of reg Z, store results in reg X. RTL: $v0 ← $a0 + $a1
nand **nand $v0, $a0, $a1**	R	1 0001_2	Nand contents of reg Y with contents of reg Z, store results in reg X. RTL: $v0 ← ~($a0 && $a1)
addi **addi $v0, $a0, 25**	I	2 0010_2	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)**	I	3 0011_2	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)**	I	4 0100_2	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, $a1, done**	I	5 0101_2	Compare the contents of reg X and reg Y. If they are the same, then branch to the address PC+1+OFFSET, where PC is the address of the beq instruction. RTL: if($a0 == $a1) PC ← PC+1+OFFSET
Note: For programmer convenience (and implementer confusion), the assembler computes the OFFSET value from the number or symbol given in the instruction and the assembler's idea of the PC. In the example, the assembler stores done-(PC+1) in OFFSET so that the machine will branch to label "done" at run time.			
jalr **jalr $at, $ra**	J	6 0110_2	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.
nop	n.a.	n.a.	Actually a pseudo instruction (i.e. the assembler will emit: add $zero, $zero, $zero
halt **halt**	O	7 0111_2	

Performance Formulas



Name	Notation	Units	Comment
Memory footprint	-	Bytes	Total space occupied by the program in memory
Execution time	$(\sum CPI_j)$ * clock cycle time, where $1 \leq j \leq n$	Seconds	Running time of the program that executes *n* instructions
Arithmetic mean	$(E_1+E_2+\ldots+E_p)/p$	Seconds	Average of execution times of constituent *p* benchmark programs
Weighted Arithmetic mean	$(f_1*E_1+f_2*E_2+\ldots+f_p*E_p)$	Seconds	Weighted average of execution times of constituent *p* benchmark programs
Geometric mean	p^{th} root $(E_1*E_2*\ldots*E_{p)}$	Seconds	*pth* root of the product of execution times of *p* programs that constitute the benchmark
Harmonic mean	$1/(\ ((1/E_1)+(1/E_2)$ $+\ldots+(1/E_p))/p\ )$	Seconds	Arithmetic mean of the reciprocals of the execution times of the constituent *p* benchmark programs
Static instruction frequency		%	Occurrence of instruction *i* in compiled code
Dynamic instruction frequency		%	Occurrence of instruction *i* in executed code
Speedup (M_A over M_B)	E_B/E_A	Number	Speedup of Machine A over B
Speedup (improvement)	E_{Before}/E_{After}	Number	Speedup After improvement
Improvement in Exec time	$(E_{old}-E_{new})/E_{old}$	Number	New Vs. old
Amdahl's law	$Time_{after}$ $= Time_{unaffected} + Time_{affected}/x$	Seconds	*x* is amount of improvement

Quiz Score: **67** out of 100