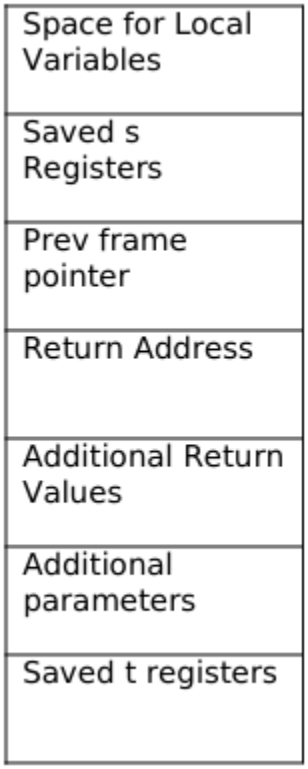
Question 1: Calling Convention (12 points) (7 minutes)

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



Q1.1 (4 Points)
Before executing JALR, the caller saves **$ra** on the stack. Explain why.

Q1.2 (4 Points)
Explain the benefit of dividing the register save/restore chore between the caller and the callee.

Q1.3 (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.

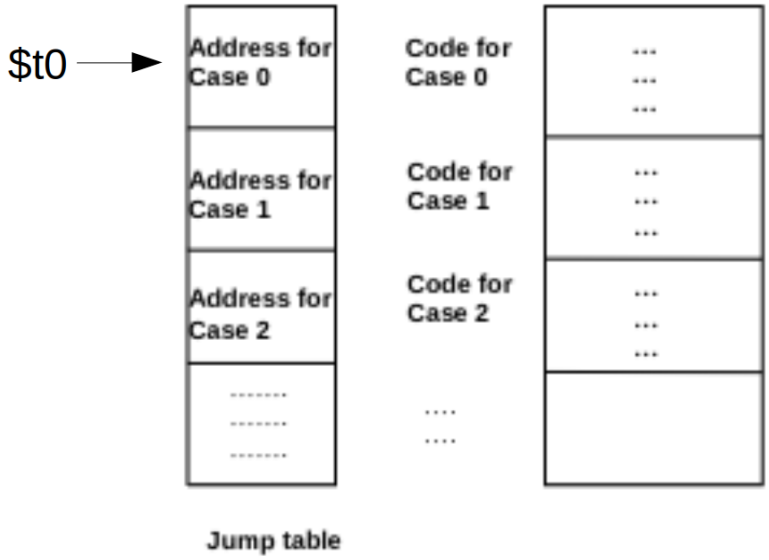
Question 2: Switch Statement and Jump Table (10 points) (5 minutes)

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:



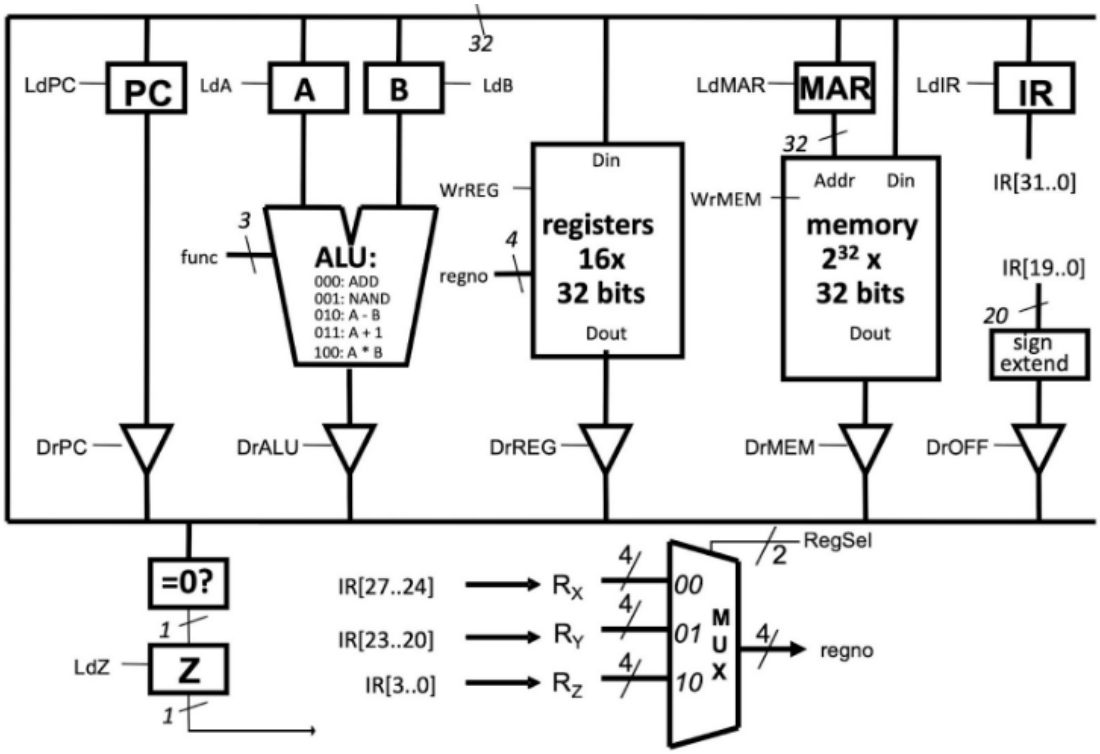
Assuming we are using the LC-2200 instruction set architecture (See Appendix).
The architecture is word addressable.

Q2.1 (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**.

Q2.2 (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)

Question 3: Datapath (12 points) (7 minutes)

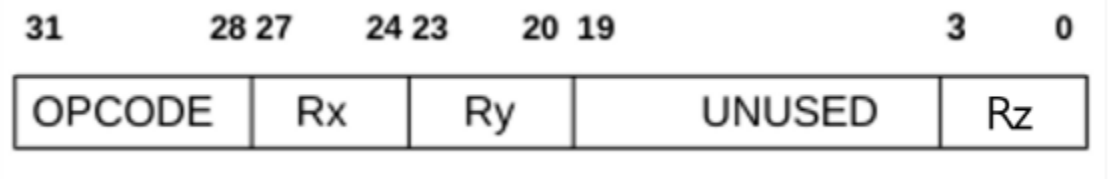
The datapath shown below corresponds to the familiar LC-2200, 32-bit word addressable architecture, with the difference that the ALU also supports a multiply operation, selected by setting the ALU’s func control signal to 100.



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)

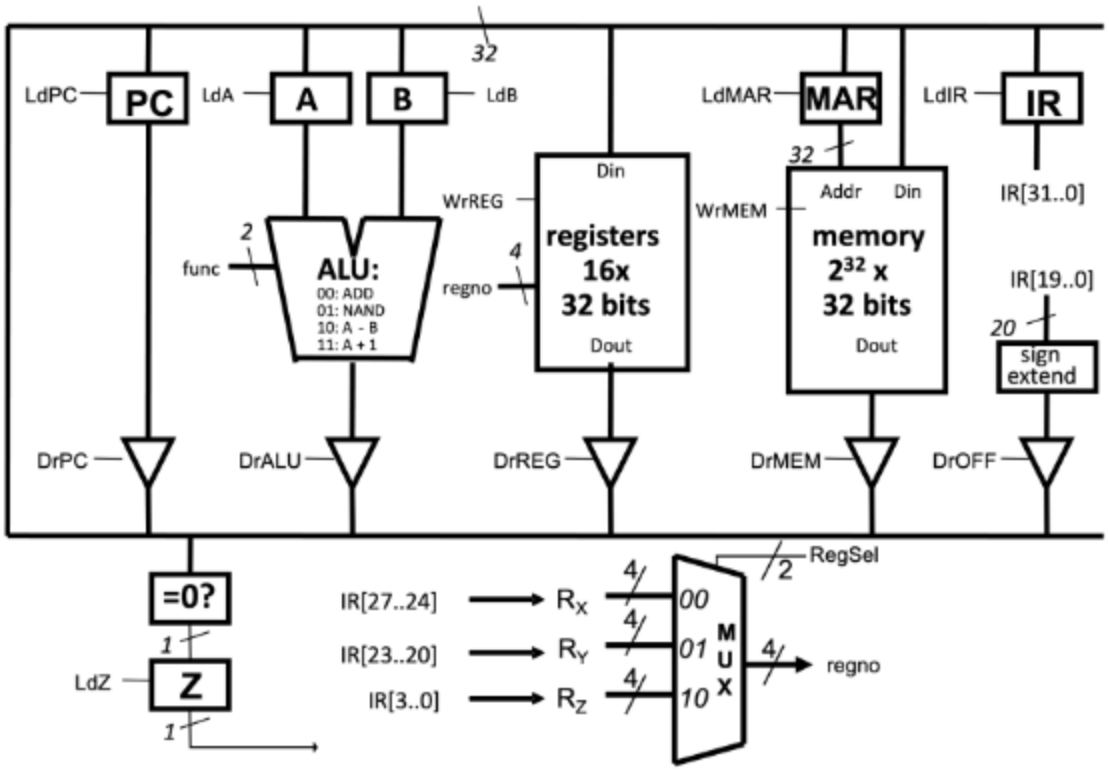
The instruction's format is as shown below:



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.

Question 4: Halt Instruction (12 points) (7 minutes)

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

Q4.1 (6 Points)
Would this microcode achieve the intended semantics of halt? Describe in detail.
(Hint: the instruction hex for halt is always 0x70000000.)

Q4.2 (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.

Question 5: Interrupts 1 (6 points) (3 minutes)

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.

Q5.1 (3 Points)
Both devices simultaneously assert the interrupt line. Whose interrupt will be serviced first?

Q5.2 (3 Points)
When will the second device get serviced?

Question 6: Interrupts 2 (6 points) (4 minutes)

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?

Question 7: Performance Metrics 1 (6 points) (3 minutes)

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

Q7.1 (2 Points)
What is the static frequency of NAND instruction?

Q7.2 (4 Points)
What is the dynamic frequency of NAND instruction?

Note: Show your work in detail for credit

Question 8: Performance Metrics 2 (6 points) (5 minutes)

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

Question 9: Performance Metrics 3 (10 points) (7 minutes)

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

Question 10: Hidden (10 points) (7 minutes)

Question 11: Hidden (10 points) (5 minutes)

Appendix A

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	

Appendix B

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	p^{th} 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