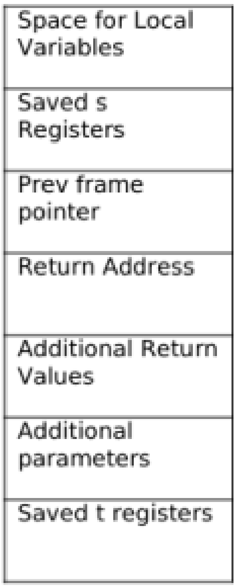
Exam 1

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

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

Q1.1 (4 Points)

Before executing JALR, the caller saves $ra on the stack. Explain why.

If the callee loses the return address to its caller, there is no way to return to the caller. Consider the case of calling subroutines within other subroutines, then the address stored in $ra will get overwritten with no way of restoring its previous values unless we save them in memory, specifically on the stack frame.

Q1.2 (4 Points)

Explain the benefit of dividing the register save/restore chore between the caller and the callee.

Because not all registers need to be saved each time, by splitting it, each can save fewer registers based on what they actually need to use. This division of responsibility saves memory access.

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.

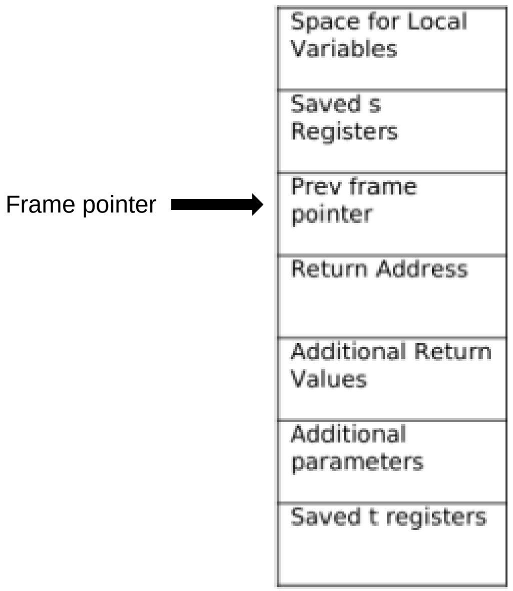
Importance of frame pointer

Commented [1]: Gradescope: "The register is a fixed point in the activation record of a procedure call. The frame pointer allows variables to be accessed with fixed offset (opposed to stack pointer that moves)"

- During execution of given module/subroutine it is possible for the stack pointer to move
- Since the location of all of the items in a stack frame is based on the stack pointer, it is useful to define a fixed point in each stack frame and maintain the address of this fixed point in a register $fp

Who saves the "prev frame pointer"

- Callee saves the frame pointer



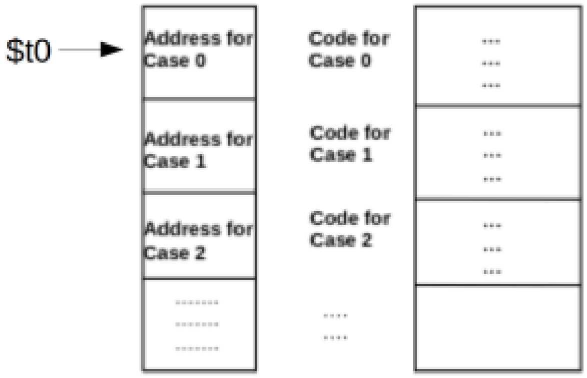
The frame pointer of the current stack frame stores the address at which the caller’s (previous) frame pointer is saved.

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:



Jump table

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**.

```
add $t2, $t0, $t1
lw $t2, 0($t2)
jalr $t2, $t0
```

—--------------------------------------please correct me if I am wrong in RED

- Could you also do this?
  - add $at, $t0, $t1
  - lw $at, 0($at)
  - jalr $at, $ra (Can be jalr $at, $zero since it doesn't specify we need to return)

Lea $a0, next
Beq $zero, $zero, case 0
Next:
Case0: Beq case 1
Case1: Beq case 2……. Br case k

Q2.2 (5 Points)

Can this implementation of a switch statement be simulated by a series of conditional branch

Commented [2]: I don't think this is correct. The second parameter should be $ra. Jalr will save the PC into this register. If we use $t0, we lose the pointer to our jump table

Commented [3]: we don't need the $t0, or we don't need to go back as it is switch statement.

Commented [4]: That's why we have $t0, or even $zero. We don't care the value stored in there. It is an unconditional jump

Commented [5]: Switches automatically fall through. The cases below can't execute if $t0 is destroyed.

Commented [6]: In the appendix at the end of the document, check part called "unconditional jump". It stated clearly that you could use $t0 in unconditional jump...'

Commented [7]: I think we want to save the address of $t0 so $zero register may be better

Commented [8]: WHY IS $T2 FIRST AND $T0 SECOND WHEN JALR IS $RA, $AT???????????

Commented [9]: I am pretty sure JALR is at, ra. Also, can the last line of code be "JALR $t2, $zero"?

Commented [10]: I think you can, as the $ra part should be ignored

Commented [11]: Why do you say that ra should be ignored? We need to have a place to return to once the switch has been executed. Even if we don't use the specific register $ra, we still need a return address

Commented [12]: We are not calling a subroutine in this scenario, so I dont think we need to save $ra on the stack.

Commented [13]: Any word on the ordering of ra and at in JALR?

Commented [14]: pretty much the same to the top one by substituting $ta to $at

Commented [15]: Would this be $t0? We never set $ra to $t0

Commented [16]: I don't think we would return back to $t0 since the switch statement would already be executed right? So I do think due to ambiguity that $ra would be the correct register.

Commented [17]: ^^

Commented [18]: Wouldn't this be jalr $ra, $at since we would want to jump to the address stored in $at and then return to whatever is deemed by $ra according to the instruction set?

Commented [19]: no, look at the appendix on the test

Commented [20]: Yeah you're right

Commented [21]: What is next? I can't find this label in the question, not quite sure what it represents instructions without accessing memory? If so, which approach is more time-efficient? Why?

(Hint: think

about space and time complexity)
Yes it is able to, and it would make it faster becausTime it doesn't have to access ram, which is slower(?)

Yes. It may be more time efficient depending on how long access to memory takes. Replacing the switch statement with conditional statements, however, would increase the amount of instructions when compiled. Foor all 9 possible values of k, specific branch statements would have to be defined. If k = 9, 9 comparisons would have to be made in the worst case before the actual code is executed. In this scenario, conditional statements would be slower.

- Revised Answer: Yes it can be completed via a series of conditional branch instructions. However, the time complexity of this would be O(n) where n is the number of branches that need to be checked. However, using memory is O(1). For the space complexity both are O(1). From this we can infer that in the worst case and average case accessing memory is quicker. In the best case where k = 0, the time it takes to access memory may be longer than simply doing a quick comparison.

Question: Is it worst case? If best case, conditional would be faster bc you only do one comparison.

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.

Commented [22]: confused whats the answer to this

Commented [23]: Wouldn't the memory-accessing approach *almost always* be faster, as it takes 9 clock cycles to get to our switch code (3 for add, 4 for lw, 2 for jalr) giving us a consistent O(1), while the BEQ approach would only be faster for the case k = 0? Each BEQ that actually branches requires 7 clock cycles while those that don't branch still need 4 since the LC2200 has a Z register that requires a dummy cycle to get Z into ROM, thus giving us O(n). From a time-efficiency standpoint, wouldn't it be better to use the memory-accessing approach that consistently achieves our goal in a timely manner rather than the BEQ approach that occasionally will be faster by 2 clock cycles in 1 case but will take more time in every other case?

Commented [24]: I'm confused as to what this means

Commented [25]: I actually kinda disagree with this. Yes, you're accessing ram, but you're only doing it once, regardless of how many cases there are. With the multiple beq approach, you're potentially accessing dozens of registers in dozens of instructions depending on how many cases you have.

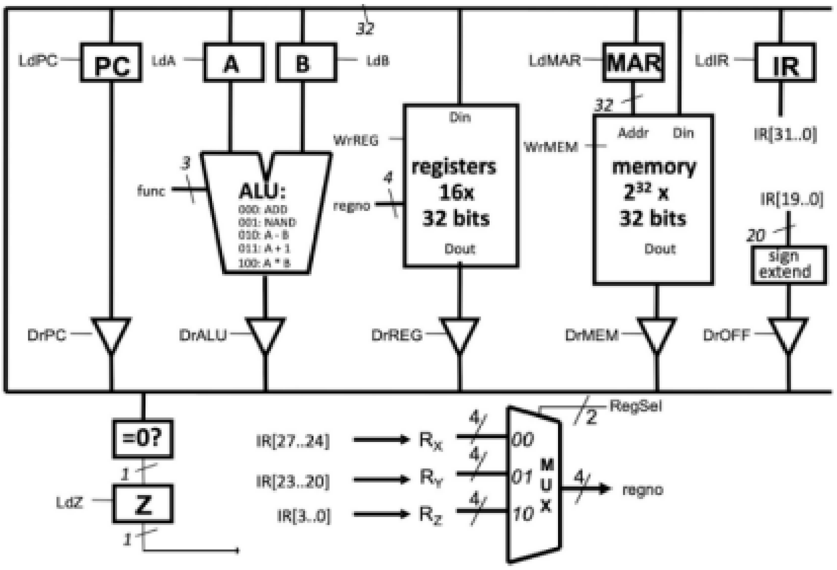
Commented [26]: Then again, if the case size was small enough you could save ram space by storing in registers so it would be more space-efficient but htat is highly dependent on architecture choices? IDK this q seems pretty open-ended to me

Commented [27]: I think accessing RAM is significantly worse than accessing registers, even if we are accessing a multitude of them because I think the time difference between accessing RAM and accessing registers is a factor of some number large enough to make the number of registers accessed irrelevant.

Commented [28]: Accessing memory takes 2 clock cycles in the LC2200 and accessing registers takes 1, so it's better to use memory if you would have to access registers more than twice as often as memory. Best-case scenario, one comparison, is better with registers, but the other nine cases are worse since you have to access the registers 2-10 times (2-10 clock cycles) for what could have been one memory access (2 clock cycles).

Commented [29]: I think we are supposed to consider time and space complexity. Using conditional branches is O(n) time complexity where n is the number of branches. Memeory is O(1). For space complexity they are both O(1) because for the conditional branches you can reuse the same registers to make the comparisons.

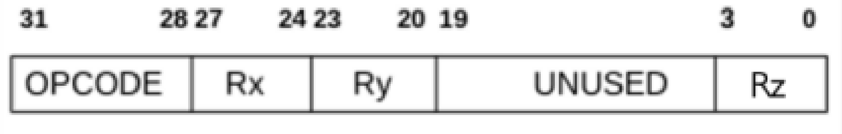
Commented [30]: memory itself may be O(1) but we also need to take into account the clock cycles it takes to take the value from memory and put it into a register.



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.

DrReg, RegSel = 01 (RegSelLo = 1), LdA
DrReg, RegSel = 10 (RegSelHi = 1), LdB
DrALU, func = 100, WrReg, RegSel = 01
DrReg, RegSel = 01 (RegSelLo = 1), LdA
DrReg, RegSel = 00, LdB

Commented [31]: This cycle is unnecessary but the question is not asking for *the optimal* solution so i guess it is fine

Commented [32]: _Re-opened_

Commented [33]: This line will overwrite the value in Rx before we can add rx value to Ry*Rz

Commented [34]: how is this cycle unnecessary? Edit: oh because you can directly load the multiplication back into A or B?

Commented [35]: Yup i think so. There's no need to write the product back into the reg file when you can just put it directly into A or B.

DrALU, func = 000, WrReg

- Mine is slightly different – after the first mmultiply of Ry and Rz, could you just load that result back into LdA or would that be incorrect?
  - A ← Ry; LdA = 1, DrReg = 1, RegSel = 01, RegSelLo = 1
  - B ← Rz; LdB = 1, DrReg = 1, RegSel = 10, RegSelHi = 1
  - A ← (A * B); DrALU = 1, func = 100, LdA = 1 (different from other solution)
  - B ← Rx; LdB = 1, DrReg = 1, RegSel = 00
  - Rx ← (A + B); DrALU = 1, func = 000, WrREG = 1, RegSel = 00

Commented [36]: Please feel free to correct this in a different color

Commented [37]: My answer was the same -- it looks good. You should be using RegSel instead of regno though

Commented [38]: ty!!

Commented [39]: It looks good, I tested the middle line A<-A*B, and it doesn't give any feedback loops on circuitsim.

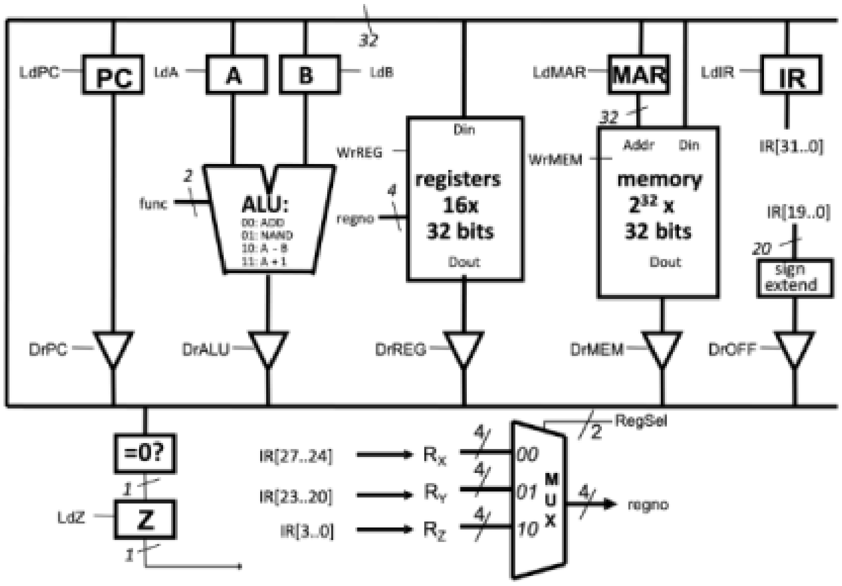
Commented [40]: 👍

Commented [41]: Why is there a RegSelLo and RegSelHi here on top of RegSel? RegSel is the concatenation of those two bits?

Commented [42]: Also I did pretty much this method exactly except I loaded the A * B into B and Rx into A (flipped the A and B registers in the ALU). I don't think this matters though

Commented [43]: yup this solution looks good

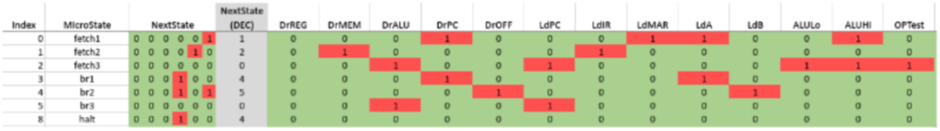
Commented [44]: Thanks



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

Assume the following datapath of LC-2200 and part of the 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.)

I think that this could work. Since we know that fetch is a precursor to any instruction, we know that A<-PC from as it is left there after the ALU operation to increment the PC. So, this means that the instruction x70000000 with an immediate value field of 0(last 19 bits are 0) would be passed into the B register for BR2 and BR3 would load PC <- PC+0. Which would stall the instruction at the same address, as it should.

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.

This would not work because we are unsure of the value of register B. We know A is the PC, but based on the previous non-fetch instruction, B could be anything. Therefore, when we set PC to A+B therefore there is no guarantee that we are branching back to the halt state and causing an infinite loop like we want

B is unknown so when calc PC + offset (A + B), you can jump to an unknown location which is not halting.h

New Answer: No - HALT’s next index should point to itself (infinite loop to ensure no other instructions after execute). If it goes to br3, that then points to fetch3 after.

NOTE ON NEW ANSWER: If you set the br instruction parameters to jump back to halt, that would also result in the infinite loop which is what I think it’s trying to do.

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?

Timer will get serviced first because it is closer.

Commented [45]: doesn't fetch increment pc by 1 though each time? so it would break out of the loop? or am I wrong

Commented [46]: Fetch does increment PC by 1, but remember at the beginning of fetch we store the original PC value in A. Then in branch2, in the IR the last 20 bits of HALT are all 0, so that gets sign-extended to 32-bits as 0, and that 0 gets stored in B. Then in branch3, A is the original PC, B is 0, and we add those together. Original PC + 0 is Original PC, then we store that into PC, and because of that, our PC is back at the same address as it was when we started fetching for the HALT instruction. Let me know if that makes sense

Commented [47]: lit ty

Commented [48]: I plugged this microcode into circuitsim and both versions work. Not sure why I second one works though.

Commented [49]: I think it only works if the value in B is 0. However, this is not guaranteed to be the case. ...

Commented [50]: I think it'll work no matter what. I tested both versions with the project 1 pow file and they both executed correctly. All the registers lined up and ...

Commented [51]: I don't know why this is happening though.

Commented [52]: Try it with Hanoi. I think we got kinda lucky that pow.s sets the B register to 0 before halt.

Commented [53]: can anyone please clarify this answer

Commented [54]: I think I said something similar - I thought that if the nextstate was br3, then br3 would go to fetch and the next instruction would be a halt so th ...

Commented [55]: but branch is jumping into an unspecified address of A + B. We don't know what A and B are and it is most likely not going to generate a ...

Commented [56]: We don't know what the registers contain, so it's probably best to not assume br is configured to go back to halt's address

Commented [57]: Would work if we know A

Commented [58]: This is true only if it is the first in the INTA chain, which I assume it is.

Commented [59]: I think the question is asking which of these two would run first. There could be something before the timer in the INTA chain, but the question is ...

Commented [60]: Yeah, but can we assume that between those two devices that the timer is first in the chaining?

Commented [61]: Yes, that is what is implied with timer device is "electrically closer" to the processor.

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

If there aren't any higher priority interrupts that were asserted after the timer or any devices closer to the process, it will get serviced second, but otherwise after the other interrupts.

Has anyone figured this out? Couldn't find the answer in slides but seem to remember them saying that we handle higher-priority interrupts within our initial interrupt handler but otherwise resolve the second interrupt once we've exited the initial handler.

Commented [62]: Is it after the other interrupts or immediately when interrupts are re-enabled in the timer interrupt handler (if the interrupt handler does re-enable interrupts)?

Commented [63]: I think it would be when interrupts are re-enabled in the timer interrupt handler

Commented [64]: But doesn't the interrupt handler only allow interrupts before executing device-specific code for higher priority levels?

Commented [65]: yeah I think it would be after it returns from the interrupt, since they have the same priority level

Commented [66]: But isn't the INT line reset immediately after the INTA line acknowledges the interrupt? This means that any other device (even at the same priority level) can raise an interrupt after the first interrupt handler code is branched to and interrupts are re-enabled

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:

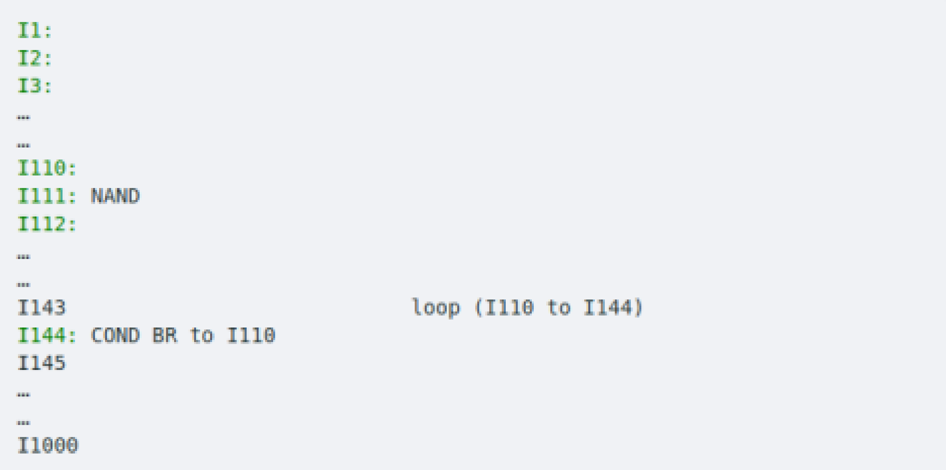
- Enable interrupt
- Jump via $k0

What is the problem with this idea?

If there are multiple interrupts, by the time the second interrupt happens, we will have lost our original return address in $k0.

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?

1/1000 = 0.1% (instruction occurs in compiled code)

Q7.2 (4 Points)

What is the dynamic frequency of NAND instruction?

c = 2.57% (particular instruction is executed, 35 instructions * 250 + 965)

Number of NANDs executed: 250

Total number executed: 35*250+(1000-35)

Note: Show your work in detail for credit

Commented [67]: Could someone please explain this result a bit more? Thank you!

Commented [68]: i have this instead: 250/(5*250+995) = 0.111 = 11.1%

Commented [69]: did you do a loop size 5 by mistake? it should be 35 - (110 to 144)

Commented [70]: i read it wrong oops

Commented [71]: how did you get that?

Commented [72]: basically, loop starts at instruction 110 (decimal) and ends at 144. The number of instructions in that loop is thus 144-110 + 1 (to account for the start itself). That means there is 35 instructions in the loop, including the NAND, and 1000-35 instructions (965) not in the loop.

to get the total dynamic amount of instructions run, we need to multiply the loop's size by 250 and then add it with the 965 other instructions, which gives us 8750+965=9715 total instructions dynamically run. But since we only care about the NAND, that means 250 of those run instructions are nand, so 250/9715 is its dynamic frequency

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.

C1: original clock cycle time
Original workload time: ((.4)(4)+(.3)(2)+(.3)(6))C1=4(C1)

C2: new longer clock cycle time
New workload time: ((.4)(4)+(.3)(2)+(.3)(4))C2=3.4(C2)

Permissible if: 3.4(C2) < 4(C1) [new time is less than old time]
C2<(4/3.4)C1

Since it’s asking for max permissible increase and C1 < C2 < (4/ 3.4)C1, would the exact answer be (0.6/3.4)C1 for the max possible increase?

Commented [73]: This is a problem from the Textbook

Commented [74]: page 166 (5-10)

4/3.4 = 1.1765, therefore the maximum increase is 17.65%
Yes, max possible increase = 17.65%

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

Did anyone else get 5x faster? Yup!, yessir!

Let x be the speed up value: 1/(0.25/x + 0.75) = 1.25

solve x = 5. same as answer above

Alternate formula: , x (seconds) is the amount of improvement (Amdahls)

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

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

Commented [75]: what formula does this use?

Commented [76]: amadhls law?

Commented [77]: Isn't amadhls this though:

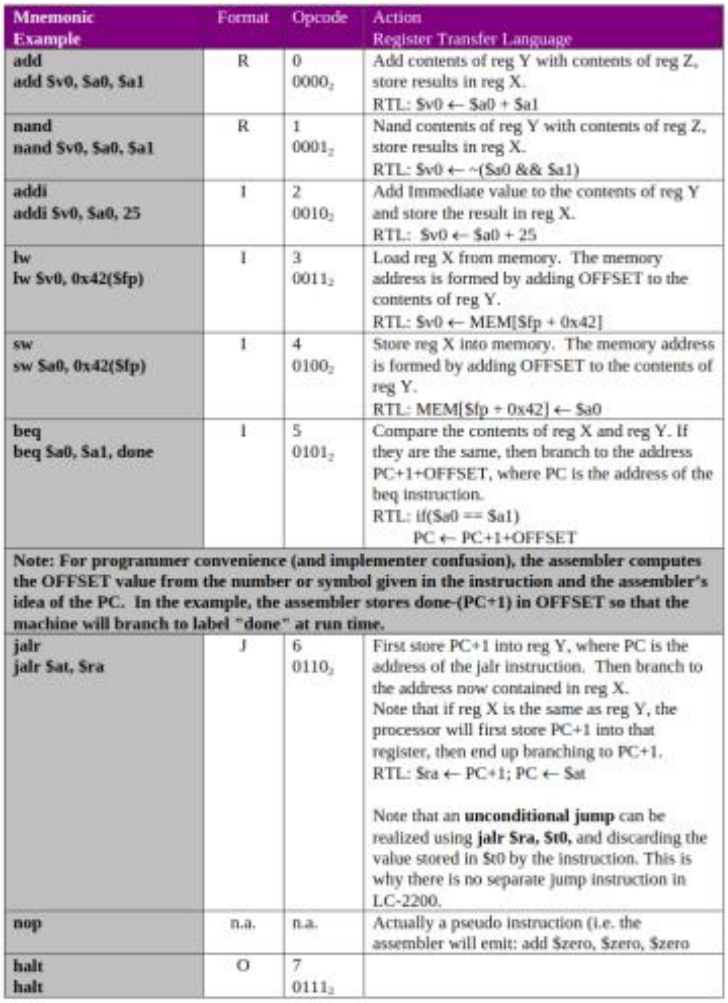
1.25 = 1 + 1.25/x

Commented [78]: Overall speedup = Old execution time / new execution time = 1/ ((1- 0.25) + (0.25/x)) = 1.25

Commented [79]: and x = 5 if you solve the equation

Appendix A

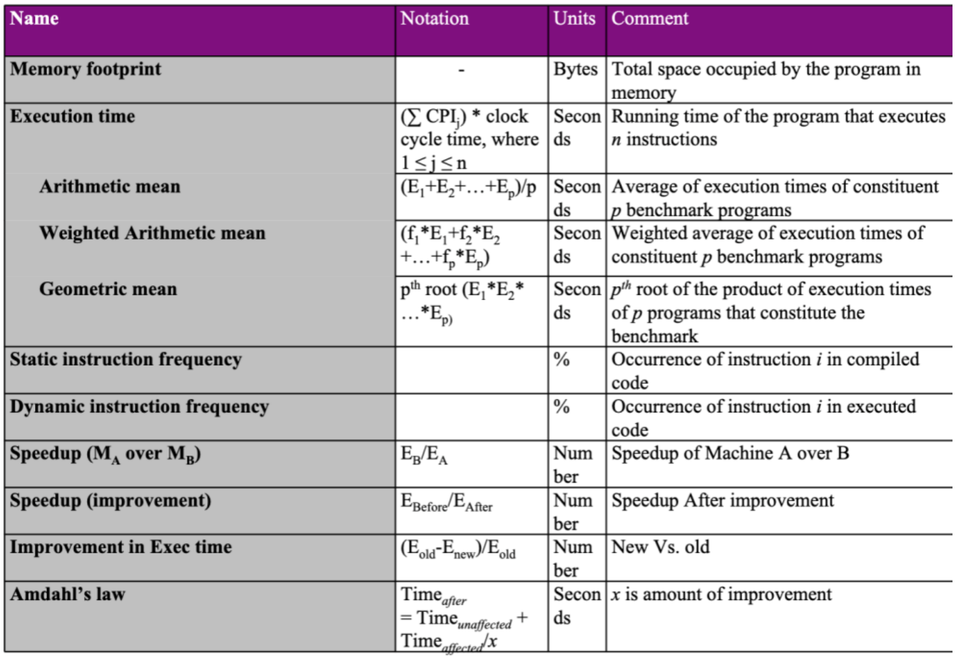
LC2200 ISA



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 $1 \le j \le n$	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	pth root $(E_1*E_2*\ldots*E_{p)}$	Seconds	*pth* root of the product of execution times of *p* programs that constitute the benchmark
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	$\text{Time}_{after} = \text{Time}_{unaffected} + \text{Time}_{affected}/x$	Seconds	*x* is amount of improvement