# 

[**Q1:**](#_fryt2mxo8dm) **4**

[Q 1.1](#_brcgou37fs20) 4

[Solution #1](#_gb0x0cu2v56p) 4

[Solution #2](#_6x21eadu66b1) 5

[Solution #3 5](#_6hc9ntkwg986)

[Q 1.2](#_p58vp4wrvay3) 5

[**Q2:**](#_wur0kn1u1z8b) **8**

**Q3: Hidden**

[**Q4:**](#_v2n753xfywi0) **8**

[Q4.1](#_svx5yzxmehl6) 8

[Q4.2](#_svx5yzxmehl6) 9

[**Q5:**](#_k8zal38ps0i) **11**

[Q5.1](#_t9lt825z8db4) 12

[Q5.2](#_9hduq6gk60fn) 12

[Q5.3](#_gmhl97r3hd6d) 12

[Q5.4](#_olkion8u0tyx) 12

[**Q6:**](#_lkmgujkgp2ls) **12**

[Q6.1](#_4xuoegk7i0k) 12

[Q6.2](#_r4gp1ktz23vq) 12

[**Q7:**](#_5crw6l2vtwzd) **12**

[Q7.1](#_tqgwmdje70zz) 13

[If we assume that int64\_t must be aligned on 4 bytes:](#_jvcbq4yom4n4) 13

[If we assume that int64\_t must be aligned on 8 bytes:](#_pk5qnb9ob2o1) 13

[Q7.2](#_wf7nji814qs3) 13

[If we assume that int64\_t must be aligned on 4 bytes:](#_7nt7ujh130tt) 13

[If we assume that int64\_t must be aligned on 8 bytes:](#_o0awxppfbuik) 13

[Q7.3](#_pku4fjg2gge6) 13

[If we assume that int64\_t must be aligned on 4 bytes:](#_rq25m6jd13ze) 13

[If we assume that int64\_t must be aligned on 8 bytes:](#_1f2d4ik243jx) 13

[Q7.4](#_47sjcypgazbp) 14

[If we assume that int64\_t must be aligned on 4 bytes:](#_jkibxbdjhzv3) 14

[If we assume that int64\_t must be aligned on 8 bytes:](#_w7nmky9fy7r) 16

[**Q8:**](#_fy7aazy3ow64) **16**

**Q9: Hidden**

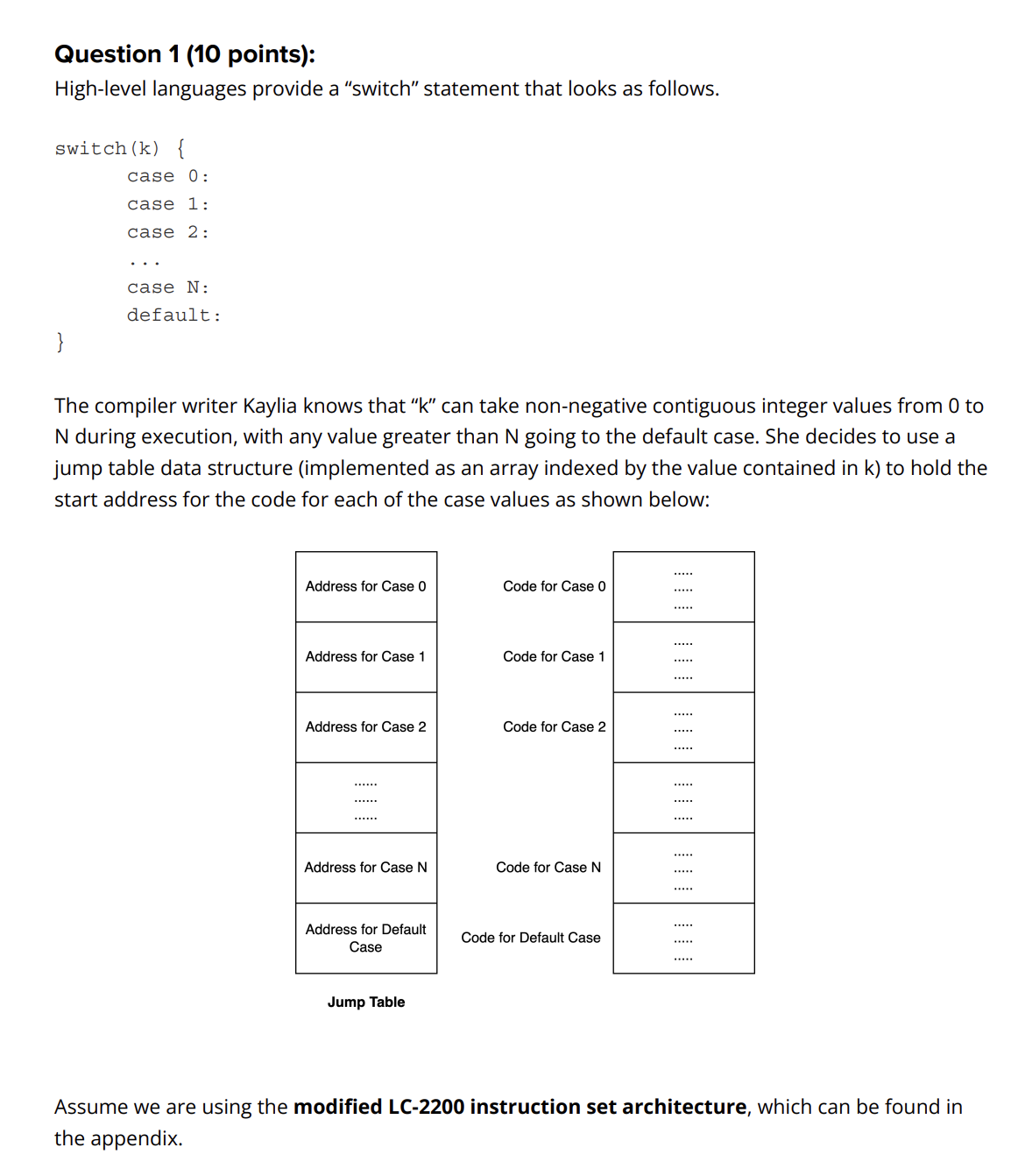
**Q10: Hidden**

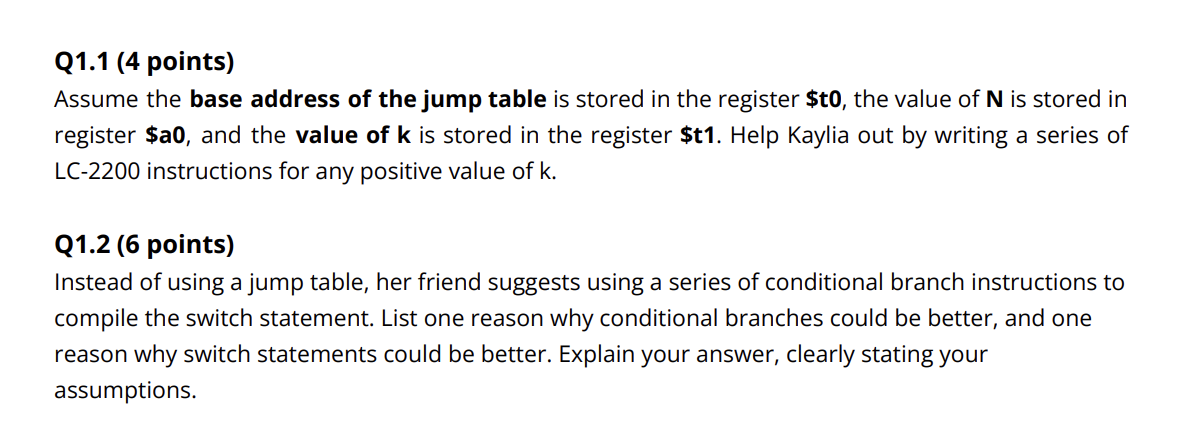
[**Appendix**](#_ltmlgugyztmp) **17**

# 

# 

# Q1:





## Q 1.1

## 

**Note: we don’t need to write out the cases, but I did it for clarity as people seem confused about this.**

### Solution #1

! if k > N, set k to N+1

bgt $t1, $a0, 1

beq $zero, $zero, 1

addi $t1, $a0, 1

! load address from k(base)

add $at, $t0, $t1

lw $at, 0($at)

! jump to address

jalr $at, $zero

case1:

! code for case 1

beq $zero, $zero, end

case2:

! code for case 2

beq $zero, $zero, end

case3:

! code for case 3

beq $zero, $zero, end

caseN:

! code for case N

beq $zero, $zero, end

default:

! code for default case

end:

! code that runs after the switch here

### Solution #2

! if k > N, set k to N+1

bgt $t1, $a0, 1

beq $zero, $zero, 1

addi $t1, $a0, 1

! load address from k(base)

add $at, $t0, $t1

lw $at, 0($at)

! store $ra and jump

addi $sp, $sp, -1

sw $ra, 0($sp)

jalr $at, $ra

! restore $ra

lw $ra, 0($sp)

addi $sp, $sp, 1

! code that comes after the switch goes here

halt

case1:

! code for case 1

jalr $ra, $zero

case2:

! code for case 2

jalr $ra, $zero

case3:

! code for case 3

jalr $ra, $zero

caseN:

! code for case N

jalr $ra, $zero

default:

! code for default case

jalr $ra, $zero

### Solution #3

bgt $t1, $a0, greater

beq $zero, $zero, less

greater:

addi $at, $a0, 1

add $at, $at, $t0

beq $zero, $zero, jump

less:

add $at, $t0, $t1

jump:

lw $at, 0($at)

jalr $at, $zero

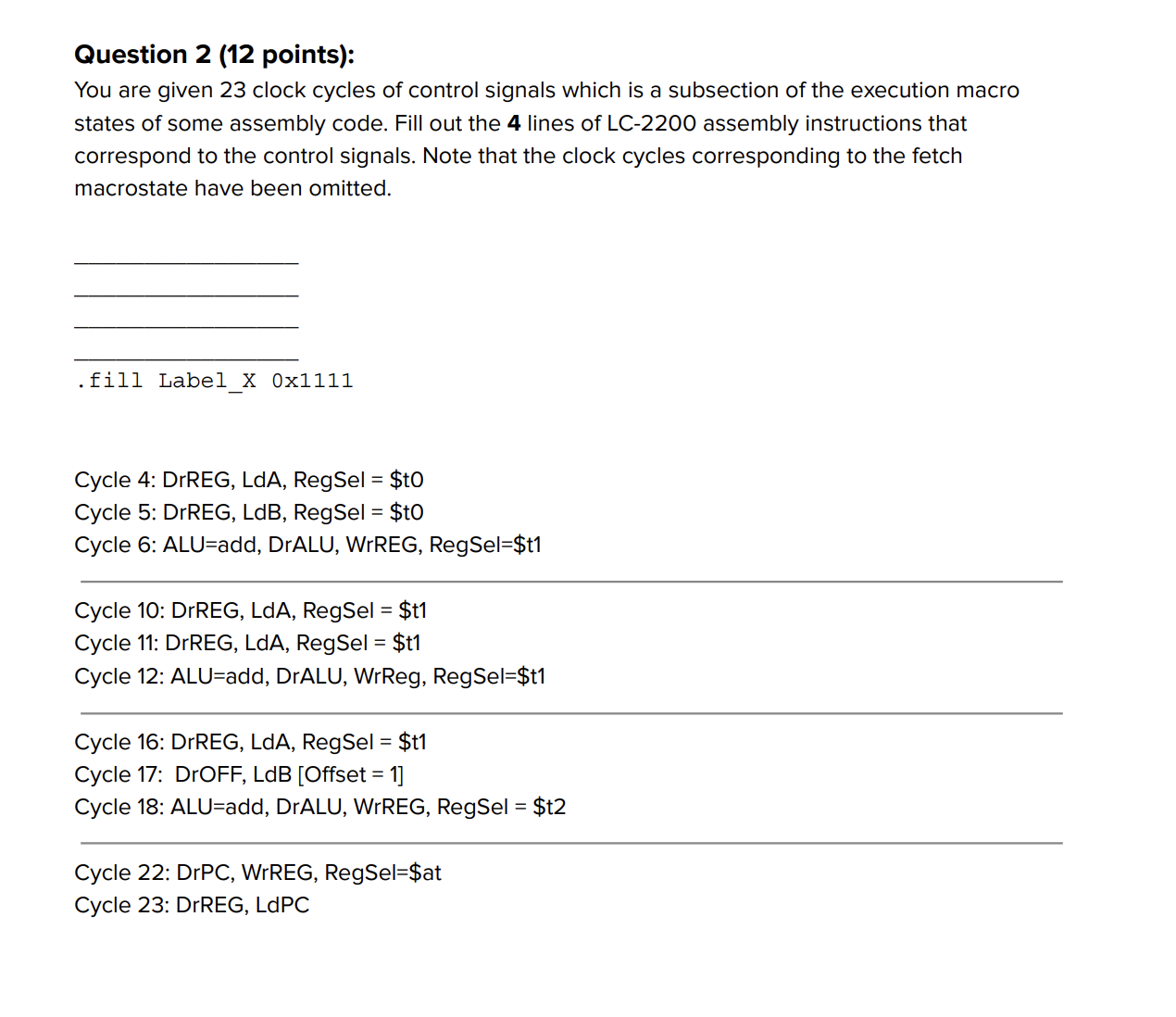
## Q 1.2

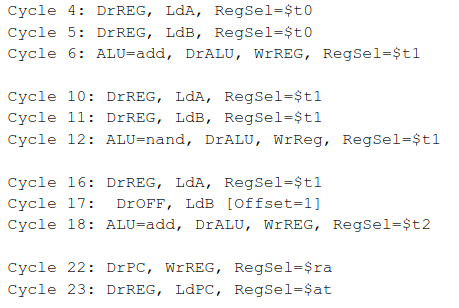
Each conditional branch can be implemented using only a single instruction, compared with the several instructions necessary for indexing into a jump table. Thus, conditional branches could be better when N is small due to the overhead of a jump table. On the other hand, when N is large, then the constant speed of the jump table will easily be faster than running through an increasingly long list of conditional branches.

Another time that conditional branches may be better is when we have non-contiguous cases, which could make the jump table very sparse. THIS IS NOT THE CASE FOR THIS PROBLEM; however, in that case, the additional speed may not be worth the extra memory required to store the jump table, even for large N. This problem specifies that the cases are known to be contiguous (perhaps we are switching over an Enum?), so this does not apply for this question.

Also, conditional branches can be faster in cases where earlier branch statement are much more likely to be used compared to the later cases which can lead to a lower weighted average runtime

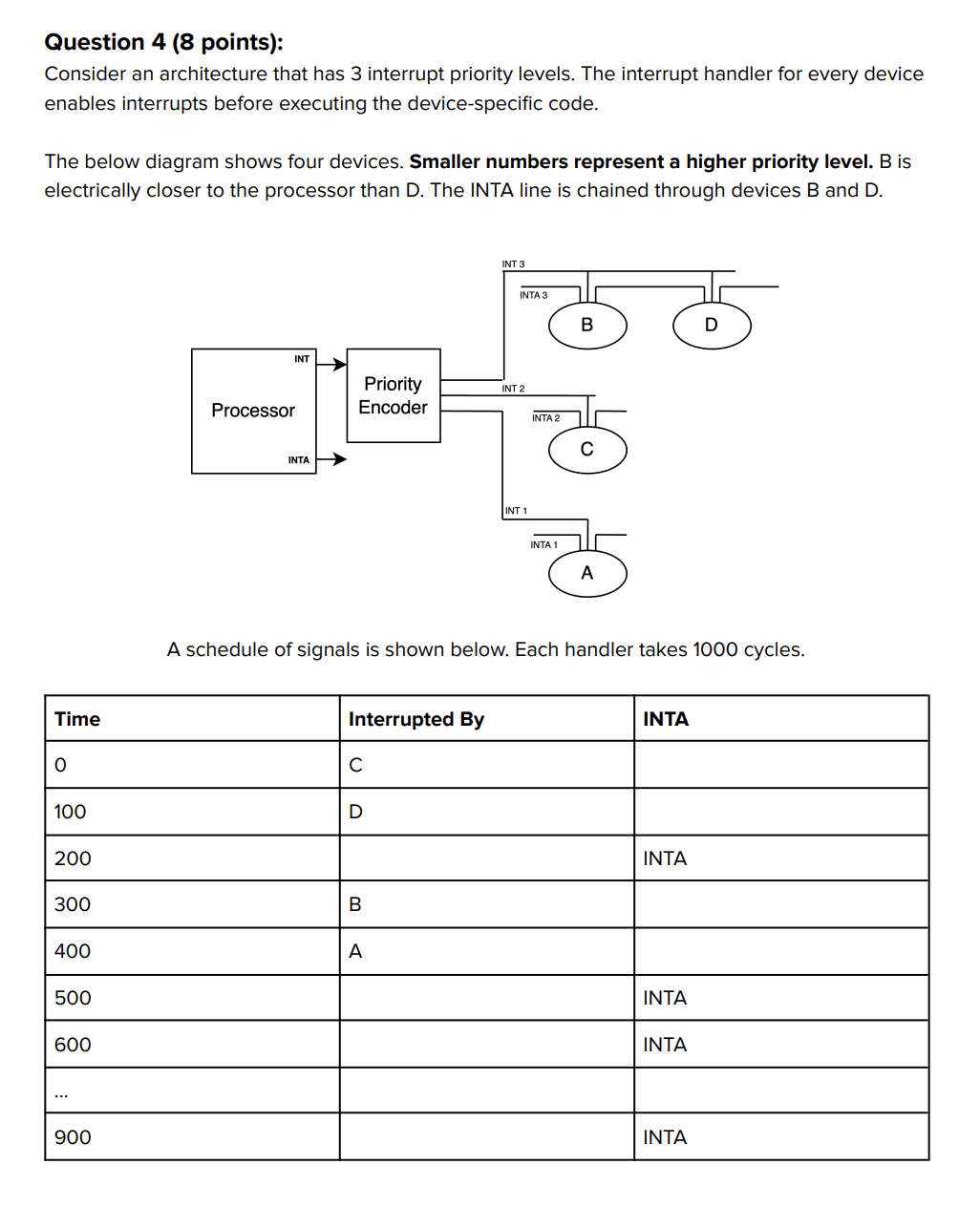
# Q2:

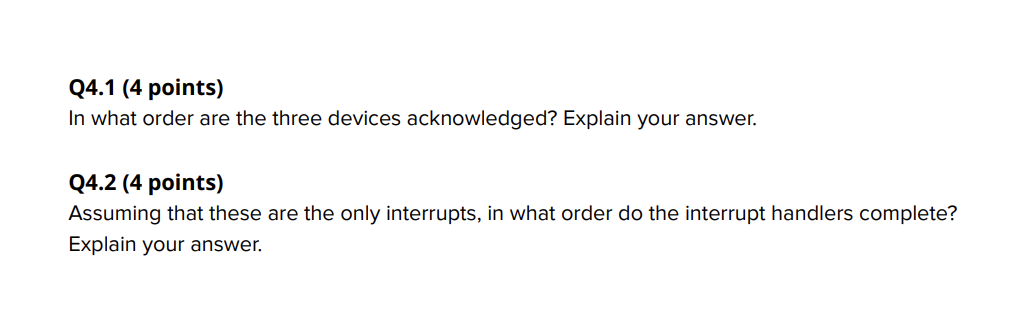




1. ADD t1, t0, t0
2. NAND t1, t1, t1
3. ADDI $t2, $t1, 1
4. JALR $at, $ra

# Q4:





## Q4.1

The order the devices are acknowledged in are C, A, B, D.

Before the first INTA, both INT2 and INT3 are enabled. Thus, the INTA goes to INT2, which has a higher priority, and C is acknowledged.

By the next INTA, only INT1 and INT3 are enabled. Thus, the INTA goes to INT1, which has the highest priority, and A is acknowledged.

On the next INTA, only INT3 is enabled. The acknowledge goes to B, since it is first in the daisy chain. On the last INTA, D is acknowledged, as it is the only device that has enabled INT.

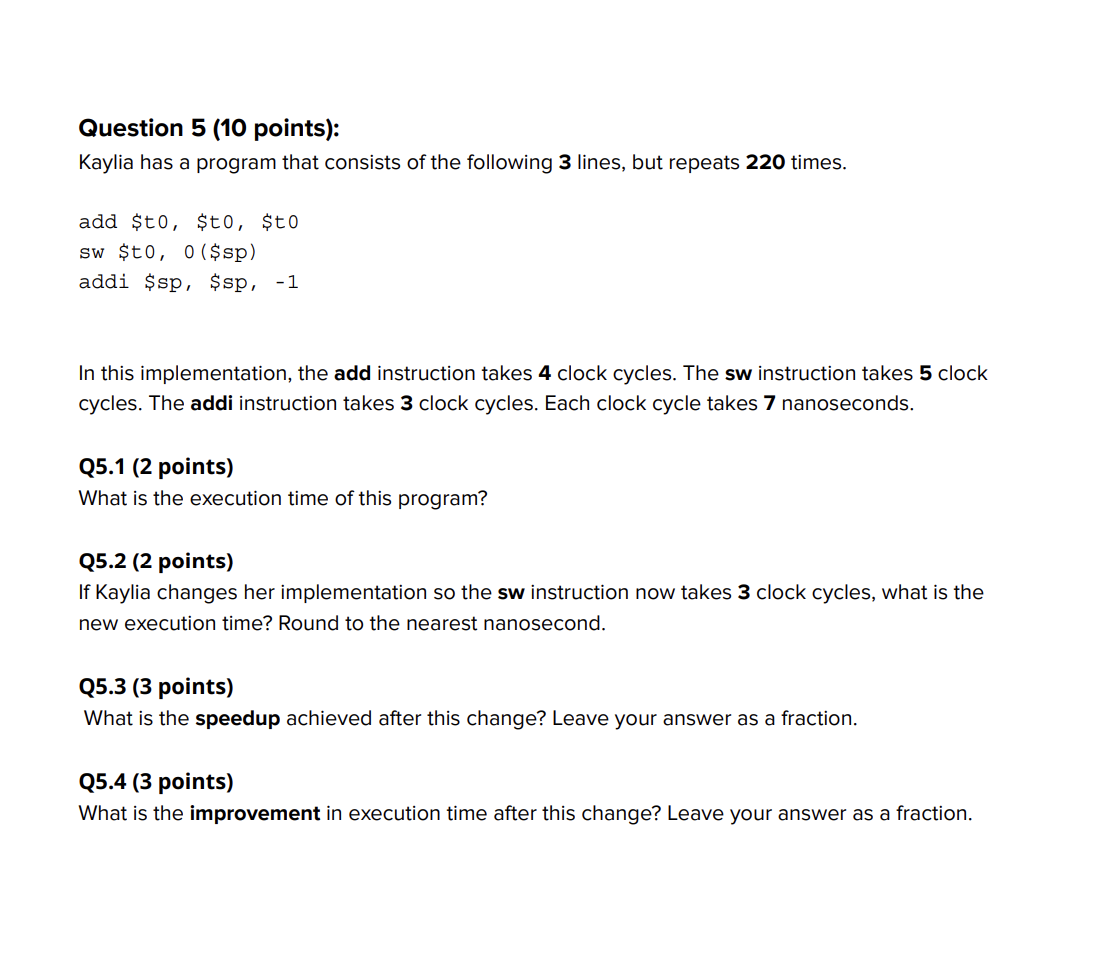
## Q4.2

The interrupt handlers finish in the reverse order of how they were acknowledged, since none of the handlers were allowed to finish executing before being interrupted. Thus, they finish in the order D, B, A, C.

Order of execution (Post Canvas Announcement.):

* Device C acknowledged, execute handler C
  + Device A acknowledged, jump to handler A and execute
  + Finish executing handler A
  + Return to handler C
* Finish executing handler C
* Device B acknowledged, execute handler B
  + Finish executing handler B
* Device D acknowledged, jump to handler D
  + Finish executing handler D

# Q5:



## Q5.1

(# add × 4 + # sw × 5 + # addi × 3) cycles × 7 ns/cycle

220 × 12 cycles × 7 ns/cycle = 18480 ns

## Q5.2

220 × 10 cycles × 7 ns/cycle = 15400 ns

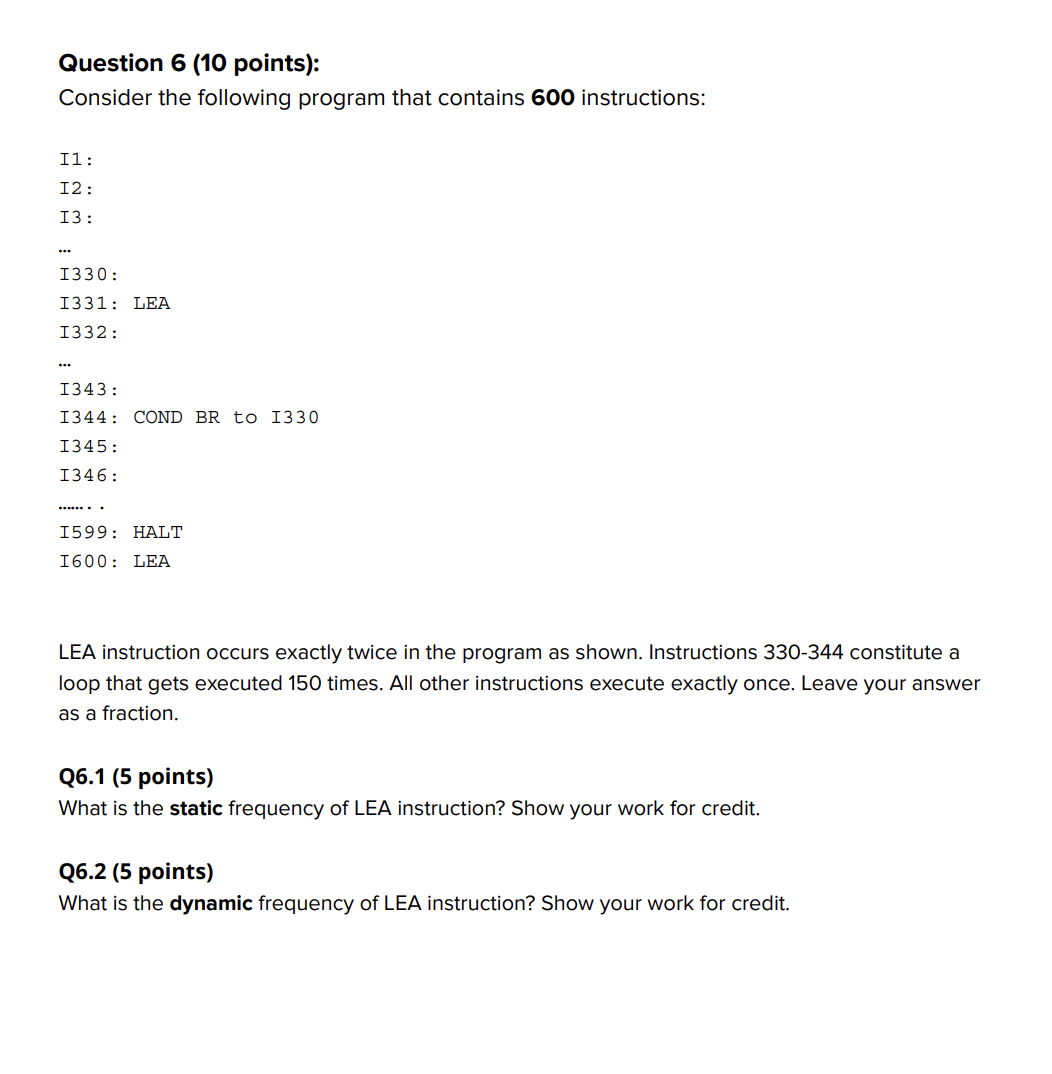
## Q5.3

= 6/5

## Q5.4

= 1/6

# Q6:



## Q6.1

Static Instruction Frequency of LEA = 2/600

F

## Q6.2

(599 - 15) = 584 instructions get executed once since no instructions execute after HALT,

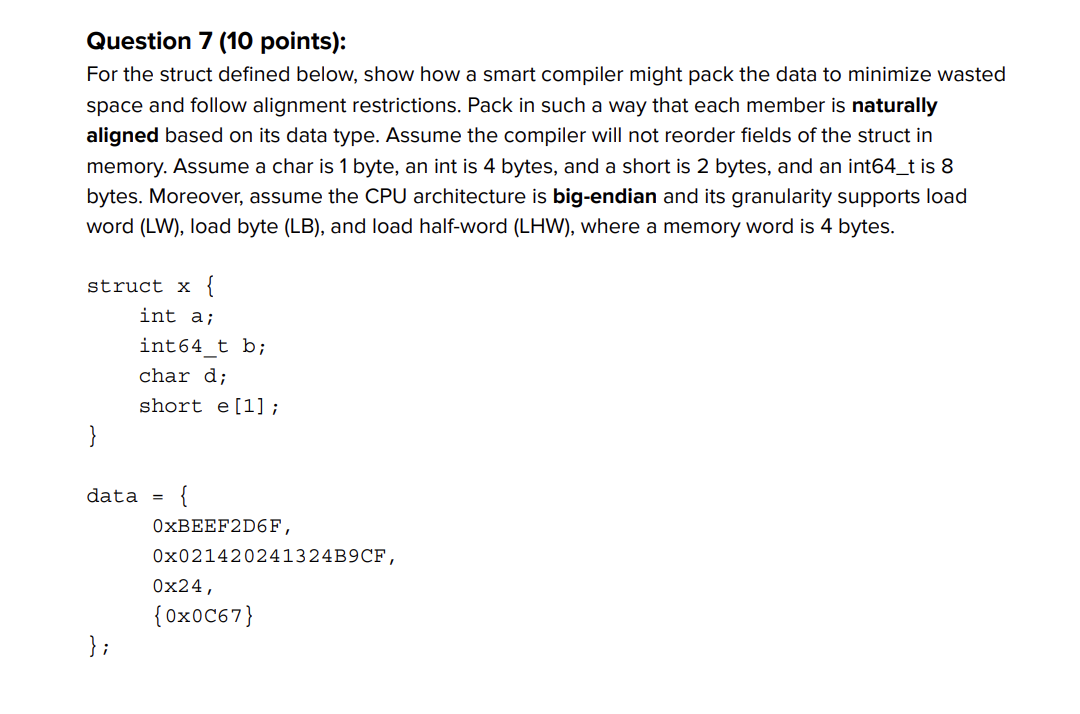
15 instructions happen 150 times = 2250,

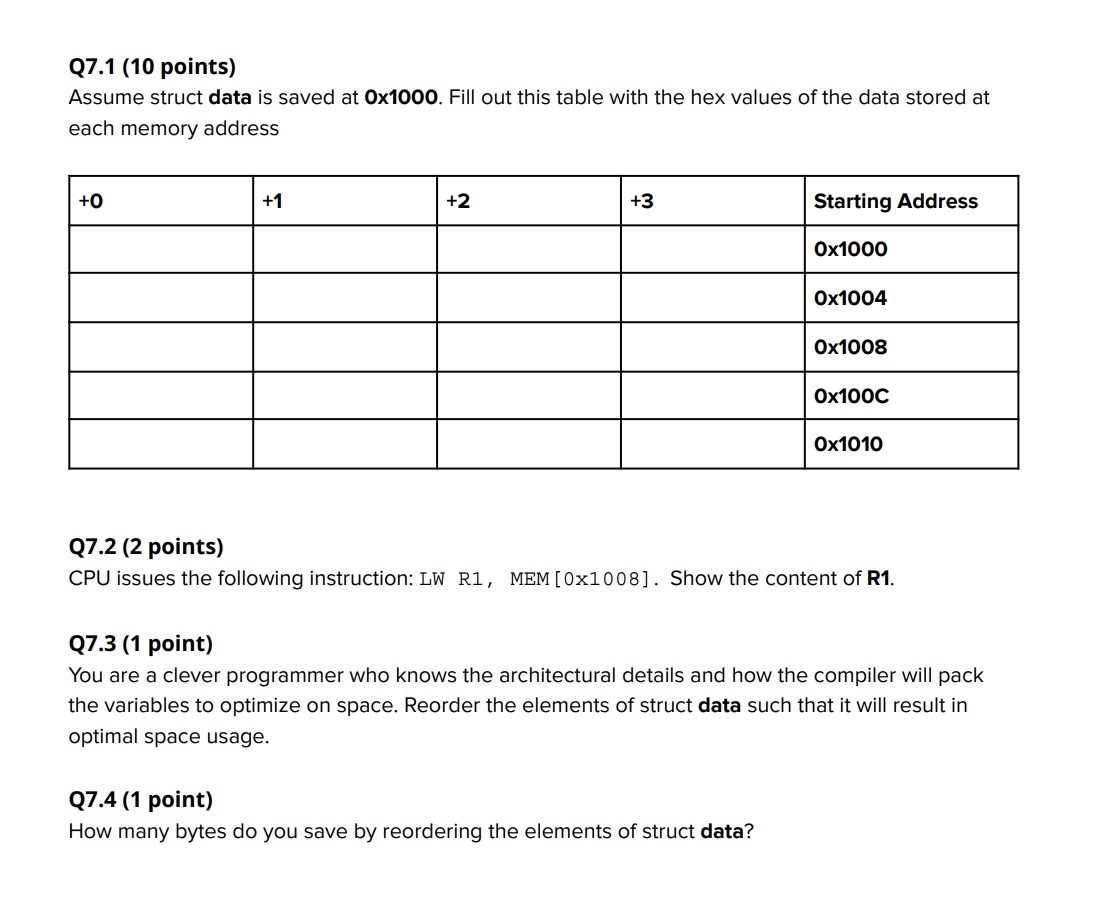
so the total number of instructions is 584 + 2250 = 2834.

LEA occurs 150 times in the loop,

so the Dynamic Instruction Frequency of LEA is 150/2834.

# Q7:





Question 7:

Important Note: 7.1 has a different answer depending on if int64\_t aligns on 8 bytes, or 4 bytes. I believe the intended answer is that it aligns on 4 bytes, since there is no load double word instruction; thus, it would not make sense to align on anything more than 4 bytes. However, the lab slides say that structs should be aligned with the size of the largest element, which in this case is 8 bytes. Furthermore, if int64\_t aligns on 8 bytes, the provided table is one row too short to fit all the padding.

## Q7.1

### If we assume that int64\_t must be aligned on 4 bytes:

| **+0** | **+1** | **+2** | **+3** | **Starting Address** |
| --- | --- | --- | --- | --- |
| 0xBE | 0xEF | 0x2D | 0x6F | **0x1000** |
| 0x02 | 0x14 | 0x20 | 0x24 | **0x1004** |
| 0x13 | 0x24 | 0xB9 | 0xCF | **0x1008** |
| 0x24 |  | 0x0C | 0x67 | **0x100C** |
|  |  |  |  | **0x1010** |

### If we assume that int64\_t must be aligned on 8 bytes:

| **+0** | **+1** | **+2** | **+3** | **Starting Address** |
| --- | --- | --- | --- | --- |
| 0xBE | 0xEF | 0x2D | 0x6F | **0x1000** |
|  |  |  |  | **0x1004** |
| 0x02 | 0x14 | 0x20 | 0x24 | **0x1008** |
| 0x13 | 0x24 | 0xB9 | 0xCF | **0x100C** |
| 0x24 |  | 0x0C | 0x67 | **0x1010** |
|  |  |  |  | **0x1014** |

## 

## 

### Q7.2

### If we assume that int64\_t must be aligned on 4 bytes:

R1 = 0x1324B9CF

### If we assume that int64\_t must be aligned on 8 bytes:

R1 = 0x02142024

## Q7.3

### If we assume that int64\_t must be aligned on 4 bytes:

There is no change.

### If we assume that int64\_t must be aligned on 8 bytes:

struct x {

int64\_t b;

int a;

short e[1];

char d;

};

Doing this way will yield the following table:

| +0 | +1 | +2 | +3 | Starting Address |
| --- | --- | --- | --- | --- |
| 0x02 | 0x14 | 0x20 | 0x24 | 0x1000 |
| 0x13 | 0x24 | 0xB9 | 0xCF | 0x1004 |
| 0xBE | 0xEF | 0x2D | 0x6F | 0x1008 |
| 0x24 |  | 0x0C | 0x67 | 0x100C |
|  |  |  |  |  |

## Q7.4

## 

### If we assume that int64\_t must be aligned on 4 bytes:

Rearranging the struct saves 0 bytes, since the compiler must add padding.

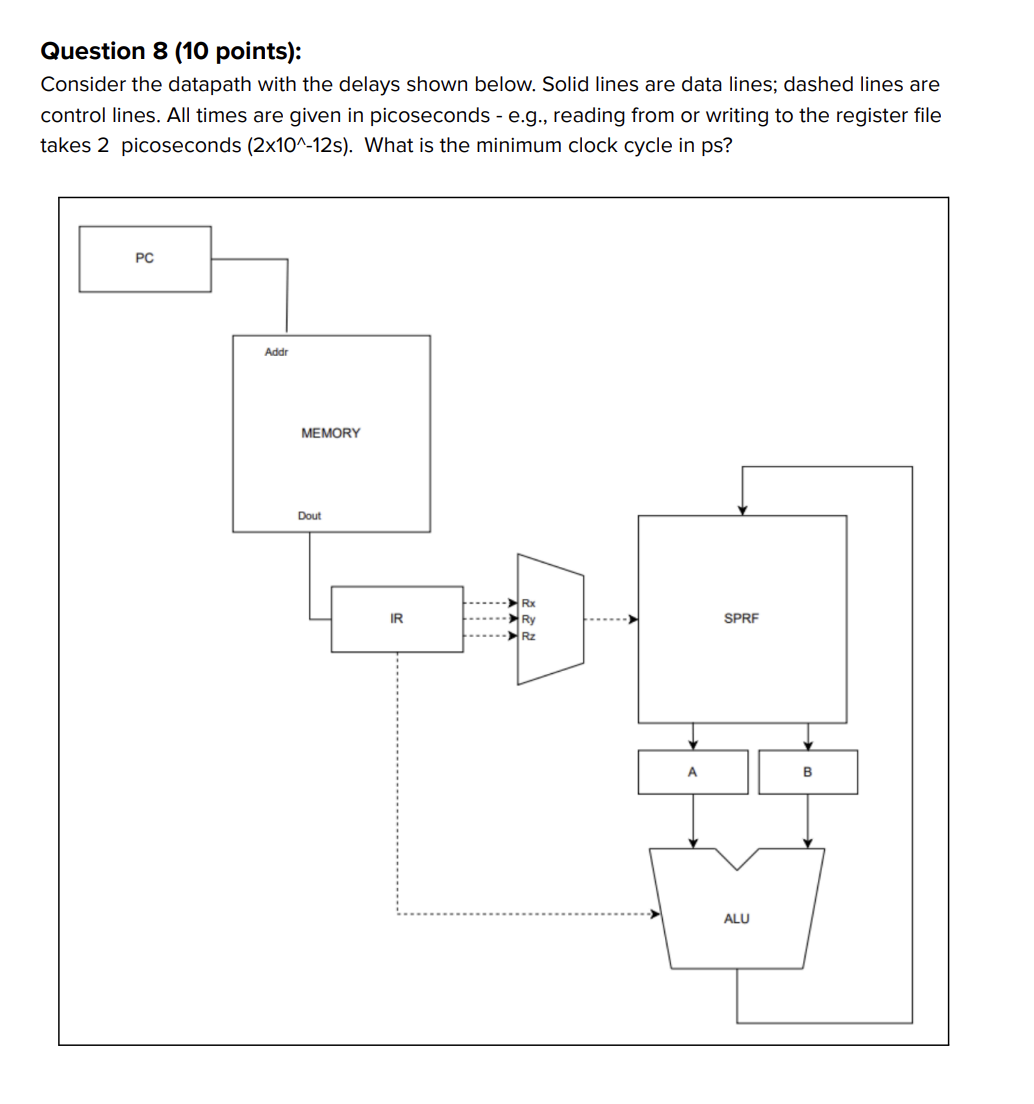
You could argue that this saves 1 byte. However, since padding is necessary to make sure any following data is word-aligned, this is not the case. Due to the padding, the given struct is already optimal.

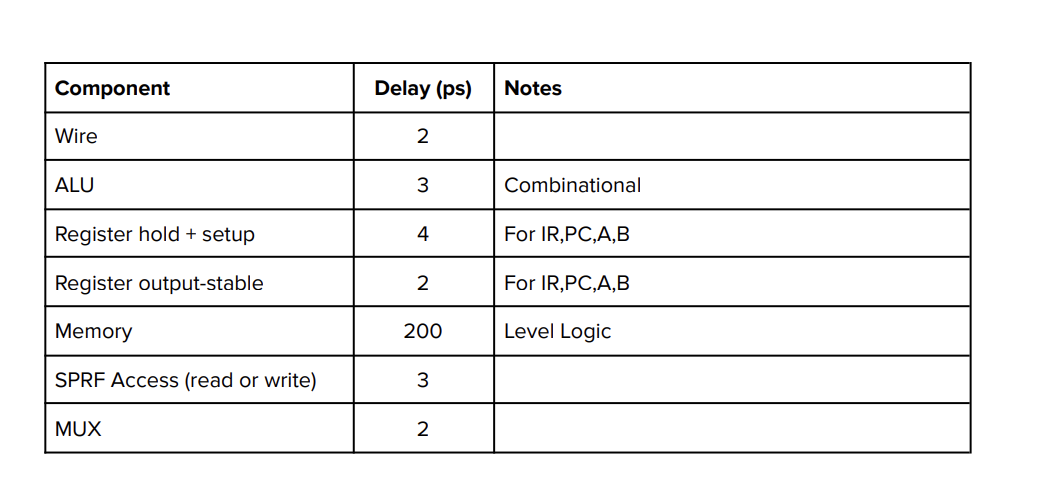
There is a case to be made that the given struct not only has optimal space usage, but also has the best readability/ease of debugging. By putting the array at the end, out-of-bounds errors are more likely to cause segmentation faults, allowing errors to be found and fixed faster.

### If we assume that int64\_t must be aligned on 8 bytes:

Rearranging the struct saves 8 bytes. Since structs must be aligned with the largest data type inside (in this case, int64\_t with an alignment of 8 bytes), the size of the struct is 24 bytes (9 bytes of padding). By rearranging the int, short[], and char to be in the same 8 byte “chunk,” we can get rid of 8 bytes of padding, bringing the total size down to 16 bytes.

# Q8:





210 ps

Trace a pretend instruction that takes the max possible time. (There’s also a rising edge 5, but I made a whoopsie the first time around and we don’t have room for it.) 5 rising edges means we have 4 cycles.

| Rising edge 1 | | Rising edge 2 | | Rising edge 3 | | Rising edge 4 | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| PC updates |  | IR updates |  | A updates |  | B updates, stabilizes | 2 |
| PC stable | 2 | IR stable | 2 | MUX, A stabilizes | 2 | Wire from B to ALU | 2 |
| Wire from PC to Memory | 2 | Wire from IR to MUX | 2 | Wire from MUX to SPRF | 2 | ALU | 3 |
| Memory access | 200 | MUX | 2 | Read SPRF | 3 | Wire from ALU to SPRF | 2 |
| Wire from Memory to IR | 2 | Wire from MUX to SPRF | 2 | Wire from SPRF to B | 2 | Write SPRF | 3 |
| Hold and setup IR | 4 | Read SPRF | 3 | Hold and setup B | 4 |  |  |
|  |  | Wire from SPRF to A | 2 |  |  |  |  |
|  |  | Hold and setup A | 4 |  |  |  |  |
|  |  |  |  |  |  |  |  |
| Total: 210 ps | | Total: 17 ps | | Total: 13 ps | | Total: 12(+4) ps | |
|  | |  | |  | |  | |

# 

# 

# 

# 

# 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 =[stack] |