11 BONUS: VLIW [50 points]

Consider a VLIW (very long instruction word) CPU that uses the long instruction format shown in Table 4. Each long instruction is composed of four short instructions, but there are restrictions on which type of instruction may go in which of the four slots.

MEMORY	INTEGER	CONTROL	FLOAT

Table 4: VLIW instruction format.

Table 5 provides a detailed description of the available short instructions and the total execution latency of each type of short instruction. Each short instruction execution unit is fully pipelined, and its result is available *on the cycle* given by the latency, e.g., a `CONTROL` instruction's results (if any) are available for other instructions to use *in the next cycle*.

Category	**Latency (cycles)**	**Instruction(s)**	**Description**	**Functionality**
`CONTROL`	1	`BEQ LABEL, Rs1, Rs2`	Branch IF equal	`IF Rs1 == Rs2: PC = LABEL`
		`NOP`	No operation	`PC = Next PC`
`MEMORY`	3	`LD Rd, [Rs]`	Memory load	`Rd = MEM[Rs]`
`INTEGER`	2	`IADD Rd, Rs1, Rs2`	Integer add	`Rd = Rs1 + Rs2`
`FLOAT`	4	`FADD Rd, Rs1, Rs2`	Floating-point add	`Rd = Rs1 + Rs2`

Table 5: Instruction latencies and descriptions.

Consider the piece of code given in Table 6. Unfortunately, it is written in terms of short instructions that cannot be directly input to the VLIW CPU.

	Instruction	**Notes**
	`< Initialize R0-R2 >`	`R0-R2 point to valid memory`
	`LOOP:`	
1	`LD R0, [R0]`	`R0 <- MEM[R0]`
2	`LD R1, [R1]`	`R1 <- MEM[R1]`
3	`IADD R4, R0, R1`	`R4 <- R0 + R1`
4	`FADD R5, R0, R4`	`R5 <- R0 + R4`
5	`LD R6, [R2]`	`R6 <- MEM[R2]`
6	`LD R2, [R0]`	`R2 <- MEM[R0]`
7	`FADD R3, R1, R6`	`R3 <- R1 + R6`
8	`IADD R4, R2, R4`	`R4 <- R2 + R4`
9	`IADD R5, R5, R4`	`R5 <- R5 + R4`
10	`IADD R0, R6, R2`	`R0 <- R6 + R2`
11	`IADD R0, R0, R3`	`R0 <- R0 + R3`
12	`BEQ LOOP, R0, R5`	`GOTO LOOP if R0 == R5`

Table 6: Proposed code for calculating the results of the next Swiss referendum.

(a) [10 points] Warm-up: which of the following are goals of VLIW CPU design (circle all that apply)?

(i) Simplify code compilation.

(ii) Simplify application development.

(iii) Reduce overall hardware complexity.

(iv) Simplify hardware inter-instruction dependence checking.

(v) Reduce processor fetch width.

(b) [25 points] Your task is to determine the optimal VLIW scheduling of the short instructions by hand. Fill in the following table with the highest performance (i.e., fewest number of execution

cycles) instruction sequence that may be directly input into the VLIW CPU and have the same functionality as the code in Table 6. Where possible, you may write instruction IDs corresponding to the numbers given in Table 6 and leave any `NOP` instructions as blank slots.

Consider **only one loop iteration** (including the `BEQ` instruction), **ignore initialization** and any cross-iteration optimizations (e.g., loop unrolling), and **do not** optimize the code by removing or changing existing instructions.

Cycle	MEMORY	INTEGER	CONTROL	FLOAT
1	1 (`LD R0, [R0]`)			
2	2 (`LD R1, [R1]`)			
3	5 (`LD R6, [R2]`)			
4	6 (`LD R2, [R0]`)			
5		3 (`IADD R4, R0, R1`)		
6				7 (`FADD R3, R1, R6`)
7		8 (`IADD R4, R2, R4`)		4 (`FADD R5, R0, R4`)
8		10 (`IADD R0, R6, R2`)		
9		8 (`IADD R4, R2, R4`)		
10		11 (`IADD R0, R0, R3`)		
11		9 (`IADD R5, R5, R4`)		
12				
13			12 (`BEQ LOOP, R0, R5`)	
14				
15				
16				
17				
18				
19				
20				

Hint: you should not require more than 20 cycles.

Note: Instruction 8 may go in EITHER of the red slots.

(c) [5 points] How many total cycles are required to complete execution of all instructions in the previous question? Ignore pipeline fill overheads and assume the instruction latencies given in Table 5.

13

(d) [10 points] What is the utilization of the instruction scheduling slots (computed as the ratio of utilized slots to total execution slots throughout execution)?

(12 slots used) / (13 cycles * 4 slots/cycle) = $\frac{3}{13} \approx 23.1\%$