ILP

Q1

a

		additional cycles	
fld	f2, 0(Rx)	3	√
fmul.d	f2. f0, f2	4	√
fdiv.d	f8, f2, f0	10	√
fld	f4, 0(Ry)	3	√
fadd.d	f4, f0, f4	2	√
fadd.d	f10, f8, f2	2	√
fsd	f4, 0(Ry)	1	√
addi	Rx,Rx,8		√
addi	Ry,Ry,8		√
sub	x20,x4,Rx		√
bnz	x20,Loop	1	√

reorder can be

	current cycles for issue	complete
fld f2, 0(Rx)	1	4
fld f4, 0(Ry)	2	5
addi Rx,Rx,8	3	3
sub x20,x4,Rx	4	4
fmul.d f2. f0, f2	5	9
fadd.d f4, f0, f4	6	8
fsd f4, 0(Ry)	9	10
fdiv.d f8, f2, f0	10	20
addi Ry,Ry,8	11	11
fadd.d f10, f8, f2	21	23
bnz x20,Loop	22	23

it need 23 cycles (ignore the WAR hazard and `fadd.d f4, f0, f4` can be calculate in next loop)

b

first cycle second cycle

	current cycles for issue	complete
fld f2, 0(Rx)	1	4
fld f4, 0(Ry)	2	5
addi Rx,Rx,8	3	3
sub x20,x4,Rx	4	4
fmul.d f2. f0, f2	5	9
fadd.d f4, f0, f4	6	8
fsd f4, 0(Ry)	9	10
fdiv.d f8, f2, f0	10	20
bnz x20,Loop	11	12
addi Ry,Ry,8	12	12
fld f4, 0(Ry)	13	16
fadd.d f4, f0, f4	14	16
fsd f4, 0(Ry)	17	18
addi Ry,Ry,8	19	19
fadd.d f10, f8, f2	21	23
fld f2, 0(Rx)	22	25
addi Rx,Rx,8	23	23
sub x20,x4,Rx	24	24
fmul.d f2. f0, f2	25	29
fdiv.d f8, f2, f0	30	40
fadd.d f10, f8, f2	41	43
bnz x20,Loop	42	43

so, it needs 43 cycles

Q2

The two types of hazards, Write-After-Write (WAW) and Write-After-Read (WAR), are both false dependencies. Only Read-After-Write (RAW) represents a true data dependency, as it involves the flow of data.

WAW and WAR can be resolved using register renaming. However, in processors with in-order dispatch and out-of-order execution, there is a possibility of encountering the two false dependencies mentioned earlier. In such cases, a subsequent instruction may write back its result before a preceding instruction, causing the earlier instruction to read incorrect data. Alternatively, the preceding instruction may write back its result later, resulting in an incorrect final outcome.

To prevent errors in out-of-order execution, the processor "must" be able to recognize these two false dependencies and execute the instructions in the correct order when hazards occur. However, this approach can lead to out-of-order execution degrading into in-order execution, particularly in program segments where false data hazards occur frequently, such as in a for-loop accumulation program. In such cases, the processor will execute all instructions sequentially, which ultimately wastes processor performance.

For example, consider the following instructions:

```
div x10, x11, x12
sw x10, 8(x0)
add x10, x11, x12
sw x10, 4(x0)
```

When register renaming is used, the execution time of the `div` instruction is often much longer than that of the `add` instruction. As a result, the result of the `add` instruction may require stalling for several cycles.

However, if the second occurrence of `x10` is renamed as `x13`, there is no need for stalling.

```
div x10, x11, x12
sw x10, 8(x0)
add x13, x11, x12
sw x13, 4(x0)
```

DLP

a

6 chimes

```
mulvv.s lv # a_re * b_re
lv mulvv.s # a_im * b_im
subvv.s sv # c_re
mulvv.s lv # a_re * b_im
mulvv.s lv # a_im & b_re
addvv.s sv # c_im
```

total cycles per iteration = 6 * 64 + 15 (l/s) *6 + 8(mul) *4 +5 (add/sub) *2 = 516

clock cycles are required per complex result value = $516/128 = 4$

b

same cycles per result still is **4** as adding additional load/store units did not improve performance

TLP

a.	P0: read 120	→	P0.B0: (S, 120, 0020) returns 0020
b.	P0: write 120 ← 80	→	P0.B0: (M, 120, 0080)
			P3.B0: (I, 120, 0020)
c.	P3: write 120 ← 80	→	P3.B0: (M, 120, 0080)
d.	P1: read 110	→	P1.B2: (S, 110, 0010) returns 0010

Step d changed to "returns 30."↵