3.1 The baseline performance (in cycles, per loop iteration) of the code sequence in Figure 3.48, if no new instruction's execution could be initiated until the previous instruction's execution had completed, is 40. See Figure S.2. Each instruction requires one clock cycle of execution (a clock cycle in which that instruction, and only that instruction, is occupying the execution units; since every instruction must execute, the loop will take at least that many clock cycles). To that base number, we add the extra latency cycles. Don't forget the branch shadow cycle.

```
Loop:  LD      F2,0(Rx)      1 + 4
       DIVD    F8,F2,F0      1 + 12
       MULTD   F2,F6,F2      1 + 5
       LD      F4,0(Ry)      1 + 4
       ADDD    F4,F0,F4      1 + 1
       ADDD    F10,F8,F2     1 + 1
       ADDI    Rx,Rx,#8      1
       ADDI    Ry,Ry,#8      1
       SD      F4,0(Ry)      1 + 1
       SUB     R20,R4,Rx     1
       BNZ     R20,Loop      1 + 1
                             ----
       cycles per loop iter  40
```

Figure S.2 Baseline performance (in cycles, per loop iteration) of the code sequence in Figure 3.48.

3.2 How many cycles would the loop body in the code sequence in Figure 3.48 require if the pipeline detected true data dependencies and only stalled on those, rather than blindly stalling everything just because one functional unit is busy? The answer is 25, as shown in Figure S.3. Remember, the point of the extra latency cycles is to allow an instruction to complete whatever actions it needs, in order to produce its correct output. Until that output is ready, no dependent instructions can be executed. So the first LD must stall the next instruction for three clock cycles. The MULTD produces a result for its successor, and therefore must stall 4 more clocks, and so on.

```
Loop:  LD       F2,0(Rx)                1 + 4
       <stall>
       <stall>
       <stall>
       <stall>
       DIVD     F8,F2,F0                1 + 12
       MULTD    F2,F6,F2                1 + 5
       LD       F4,0(Ry)                1 + 4
       <stall due to LD latency>
       <stall due to LD latency>
       <stall due to LD latency>
       <stall due to LD latency>
       ADDD     F4,F0,F4                1 + 1
       <stall due to ADDD latency>
       <stall due to DIVD latency>
       <stall due to DIVD latency>
       <stall due to DIVD latency>
       <stall due to DIVD latency>
       ADDD     F10,F8,F2               1 + 1
       ADDI     Rx,Rx,#8                1
       ADDI     Ry,Ry,#8                1
       SD       F4,0(Ry)                1 + 1
       SUB      R20,R4,Rx               1
       BNZ      R20,Loop                1 + 1
       <stall branch delay slot>
                                        ------
       cycles per loop iter             25
```

Figure S.3

3.3 Consider a multiple-issue design. Suppose you have two execution pipelines, each capable of beginning execution of one instruction per cycle, and enough fetch/decode bandwidth in the front end so that it will not stall your execution. Assume results can be immediately forwarded from one execution unit to another, or to itself. Further assume that the only reason an execution pipeline would stall is to observe a true data dependency. Now how many cycles does the loop require? The answer is 22, as shown in Figure S.4. The LD goes first, as before, and the DIVD must wait for it through 4 extra latency cycles. After the DIVD comes the MULTD, which can run in the second pipe along with the DIVD, since there's no dependency between them. (Note that they both need the same input, F2, and they must both wait on F2's readiness, but there is no constraint between them.) The LD following the MULTD does not depend on the DIVD nor the MULTD, so had this been a superscalar-order-3 machine,

	Execution pipe 0		Execution pipe 1
Loop:	**LD F2,0(Rx)**	;	<nop>
	<stall for LD latency>	;	<nop>
	<stall for LD latency>	;	<nop>
	<stall for LD latency>	;	<nop>
	<stall for LD latency>	;	<nop>
	DIVD F8,F2,F0	;	**MULTD F2,F6,F2**
	LD F4,0(Ry)	;	<nop>
	<stall for LD latency>	;	<nop>
	<stall for LD latency>	;	<nop>
	<stall for LD latency>	;	<nop>
	<stall for LD latency>	;	<nop>
	ADD F4,F0,F4	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	ADDD F10,F8,F2	;	**ADDI Rx,Rx,#8**
	ADDI Ry,Ry,#8	;	**SD F4,0(Ry)**
	SUB R20,R4,Rx	;	**BNZ R20,Loop**
	<nop>	;	**<stall due to BNZ>**
	cycles per loop iter 22		

Figure S.4 Number of cycles required per loop.

that LD could conceivably have been executed concurrently with the DIVD and the MULTD. Since this problem posited a two-execution-pipe machine, the LD executes in the cycle following the DIVD/MULTD. The loop overhead instructions at the loop's bottom also exhibit some potential for concurrency because they do not depend on any long-latency instructions.

3.4 Possible answers:

1. If an interrupt occurs between N and $N + 1$, then $N + 1$ must not have been allowed to write its results to any permanent architectural state. Alternatively, it might be permissible to delay the interrupt until $N + 1$ completes.
2. If N and $N + 1$ happen to target the same register or architectural state (say, memory), then allowing N to overwrite what $N + 1$ wrote would be wrong.
3. N might be a long floating-point op that eventually traps. $N + 1$ cannot be allowed to change arch state in case N is to be retried.

Long-latency ops are at highest risk of being passed by a subsequent op. The `DIVD` instr will complete long after the `LD F4,0(Ry)`, for example.

3.5 Figure S.5 demonstrates one possible way to reorder the instructions to improve the performance of the code in Figure 3.48. The number of cycles that this reordered code takes is 20.

```
      Execution pipe 0                    Execution pipe 1
Loop: LD       F2,0(Rx)                ;  LD       F4,0(Ry)
      <stall for LD latency>           ;  <stall for LD latency>
      <stall for LD latency>           ;  <stall for LD latency>
      <stall for LD latency>           ;  <stall for LD latency>
      <stall for LD latency>           ;  <stall for LD latency>
      DIVD     F8,F2,F0                ;  ADDD     F4,F0,F4
      MULTD    F2,F6,F2                ;  <stall due to ADDD latency>
      <stall due to DIVD latency>      ;  SD       F4,0(Ry)
      <stall due to DIVD latency>      ;  <nop>                         #ops:   11
      <stall due to DIVD latency>      ;  <nop>                         #nops:  (20 × 2) – 11 = 29
      <stall due to DIVD latency>      ;  ADDI     Rx,Rx,#8
      <stall due to DIVD latency>      ;  ADDI     Ry,Ry,#8
      <stall due to DIVD latency>      ;  <nop>
      <stall due to DIVD latency>      ;  <nop>
      <stall due to DIVD latency>      ;  <nop>
      <stall due to DIVD latency>      ;  <nop>
      <stall due to DIVD latency>      ;  <nop>
      <stall due to DIVD latency>      ;  SUB      R20,R4,Rx
      ADDD     F10,F8,F2               ;  BNZ      R20,Loop
      <nop>                            ;  <stall due to BNZ>

      cycles per loop iter 20
```

Figure S.5 Number of cycles taken by reordered code.

3.6 a. Fraction of all cycles, counting both pipes, wasted in the reordered code shown in Figure S.5:

11 ops out of 2x20 opportunities.

$1 - 11/40 = 1 - 0.275$
$= 0.725$

b. Results of hand-unrolling two iterations of the loop from code shown in Figure S.6:

c. $\text{Speedup} = \frac{\text{exec time w/o enhancement}}{\text{exec time with enhancement}}$

$\text{Speedup} = 20 / (22/2)$
$= 1.82$

	Execution pipe 0		;	Execution pipe 1	
Loop:	LD	F2,0(Rx)	;	LD	F4,0(Ry)
	LD	F2,0(Rx)	;	LD	F4,0(Ry)
	<stall for LD latency>		;	<stall for LD latency>	
	<stall for LD latency>		;	<stall for LD latency>	
	<stall for LD latency>		;	<stall for LD latency>	
	DIVD	F8,F2,F0	;	ADDD	F4,F0,F4
	DIVD	F8,F2,F0	;	ADDD	F4,F0,F4
	MULTD	F2,F0,F2	;	SD	F4,0(Ry)
	MULTD	F2,F6,F2	;	SD	F4,0(Ry)
	<stall due to DIVD latency>		;	<nop>	
	<stall due to DIVD latency>		;	ADDI	Rx,Rx,#16
	<stall due to DIVD latency>		;	ADDI	Ry,Ry,#16
	<stall due to DIVD latency>		;	<nop>	
	<stall due to DIVD latency>		;	<nop>	
	<stall due to DIVD latency>		;	<nop>	
	<stall due to DIVD latency>		;	<nop>	
	<stall due to DIVD latency>		;	<nop>	
	<stall due to DIVD latency>		;	<nop>	
	<stall due to DIVD latency>		;	<nop>	
	ADDD	F10,F8,F2	;	SUB	R20,R4,Rx
	ADDD	F10,F8,F2	;	BNZ	R20,Loop
	<nop>		;	<stall due to BNZ>	
	cycles per loop iter 22				

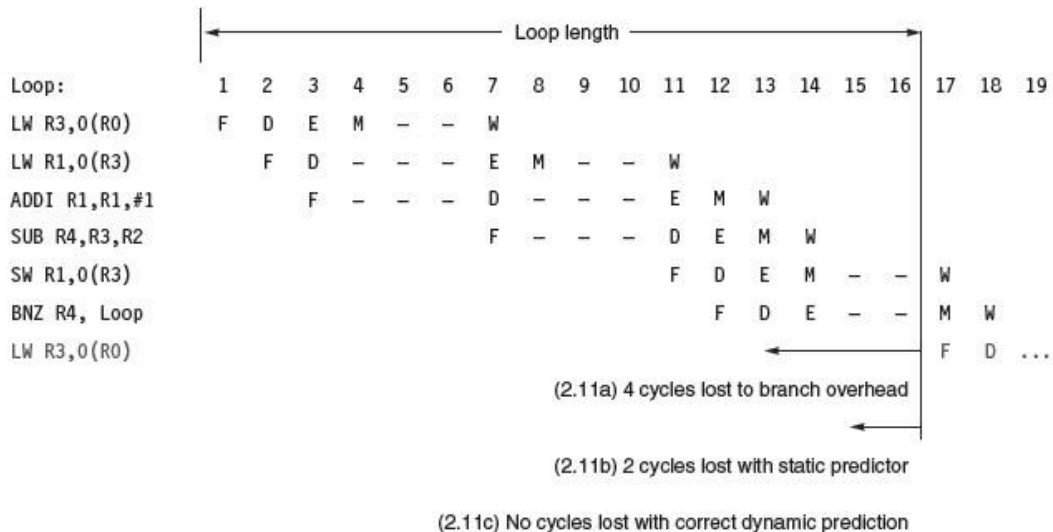
Figure S.6 Hand-unrolling two iterations of the loop from code shown in Figure S.5.

3.10 An example of an event that, in the presence of self-draining pipelines, could disrupt the pipelining and yield wrong results is shown in Figure S.10.

Clock cycle	alu0	alu1	ld/st	ld/st	br
1	ADDI R11, R3, #2		LW R4, 0(R0)		
2	ADDI R2, R2, #16	ADDI R20, R0, #2	LW R4, 0(R0)	LW R5, 8(R1)	
3				LW R5, 8(R1)	
4	ADDI R10, R4, #1				
5	ADDI R10, R4, #1		SW R7, 0(R6)	SW R9, 8(R8)	
6		SUB R4, R3, R2	SW R7, 0(R6)	SW R9, 8(R8)	
7					BNZ R4, Loop

Figure S.10 Example of an event that yields wrong results. What could go wrong with this? If an interrupt is taken between clock cycles 1 and 4, then the results of the LW at cycle 2 will end up in R1, instead of the LW at cycle 1. Bank stalls and ECC stalls will cause the same effect—pipes will drain, and the last writer wins, a classic WAW hazard. All other "intermediate" results are lost.

3.11 See Figure S.11. The convention is that an instruction does not enter the execution phase until all of its operands are ready. So the first instruction, `LW R3,0(R0)`, marches through its first three stages (F, D, E) but that M stage that comes next requires the usual cycle plus two more for latency. Until the data from a `LD` is available at the execution unit, any subsequent instructions (especially that `ADDI R1, R1, #1`, which depends on the 2nd `LW`) cannot enter the E stage, and must therefore stall at the D stage.



Loop:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
LW R3,0(R0)	F	D	E	M	–	–	W												
LW R1,0(R3)		F	D	–	–	–	E	M	–	–	W								
ADDI R1,R1,#1			F	–	–	–	D	–	–	–	E	M	W						
SUB R4,R3,R2							F	–	–	–	D	E	M	W					
SW R1,0(R3)											F	D	E	M	–	–	W		
BNZ R4, Loop												F	D	E	–	–	M	W	
LW R3,0(R0)																	F	D	...

Figure S.11 Phases of each instruction per clock cycle for one iteration of the loop.

a. 4 cycles lost to branch overhead. Without bypassing, the results of the `SUB` instruction are not available until the `SUB`'s W stage. That tacks on an extra 4 clock cycles at the end of the loop, because the next loop's `LW R1` can't begin until the branch has completed.

b. 2 cycles lost w/ static predictor. A static branch predictor may have a heuristic like "if branch target is a negative offset, assume it's a loop edge, and loops are usually taken branches." But we still had to fetch and decode the branch to see that, so we still lose 2 clock cycles here.

c. No cycles lost w/ correct dynamic prediction. A dynamic branch predictor remembers that when the branch instruction was fetched in the past, it eventually turned out to be a branch, and this branch was taken. So a "predicted taken" will occur in the same cycle as the branch is fetched, and the next fetch after that will be to the presumed target. If correct, we've saved all of the latency cycles seen in 3.11 (a) and 3.11 (b). If not, we have some cleaning up to do.

B小題寫1 or 2都給對

3.14

(a)

Clock cycle	Unscheduled code		Clock cycle	Scheduled code	
1	DADDIU	R4,R1,#800	1	DADDIU	R4,R1,#800
2	L.D	F2,0(R1)	2	L.D	F2,0(R1)
3	stall		3	L.D	F6,0(R2)
4	MUL.D	F4,F2,F0	4	MUL.D	F4,F2,F0
5	L.D	F6,0(R2)	5	DADDIU	R1,R1,#8
6	stall(4)		6	DADDIU	R2,R2,#8
10	ADD.D	F6,F4,F6	7	DSLTU	R3,R1,R4
11	stall(3)		8	stall(2)	
14	S.D	F6,0(R2)	10	ADD.D	F6,F4,F6
15	DADDIU	R1,R1,#8	11	stall(2)	
16	DADDIU	R2,R2,#8	13	BNEZ	R3,foo
17	DSLTU	R3,R1,R4	14	S.D	F6,-8(R2)
18	stall				
19	BNEZ	R3,foo			
20	stall				

The unscheduled code uses 20 cycles to finish the execution while the scheduled one uses 14. The scheduled one is 42.9% faster.

3.14

(b)

b. See Figure S.19.

Clock cycle	Scheduled code	
1	DADDIU	R4,R1,#800
2	L.D	F2,0(R1)
3	L.D	F6,0(R2)
4	MUL.D	F4,F2,F0

Figure S.19 **The code must be unrolled three times to eliminate stalls after scheduling.**

Clock cycle	Scheduled code	
5	L.D	F2,8(R1)
6	L.D	F10,8(R2)
7	MUL.D	F8,F2,F0
8	L.D	F2,8(R1)
9	L.D	F14,8(R2)
10	MUL.D	F12,F2,F0
11	ADD.D	F6,F4,F6
12	DADDIU	R1,R1,#24
13	ADD.D	F10,F8,F10
14	DADDIU	R2,R2,#24
15	DSLTU	R3,R1,R4
16	ADD.D	F14,F12,F14
17	S.D	F6,-24(R2)
18	S.D	F10,-16(R2)
19	BNEZ	R3,foo
20	S.D	F14,-8(R2)

Figure S.19 *Continued*

3.14

(c)

c. See Figures S.20 and S.21.

Unrolled 6 times:

Cycle	Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
1	L.D F1,0(R1)	L.D F2,8(R1)			
2	L.D F3,16(R1)	L.D F4,24(R1)			
3	L.D F5,32(R1)	L.D F6,40(R1)	MUL.D F1,F1,F0	MUL.D F2,F2,F0	
4	L.D F7,0(R2)	L.D F8,8(R2)	MUL.D F3,F3,F0	MUL.D F4,F4,F0	
5	L.D F9,16(R2)	L.D F10,24(R2)	MUL.D F5,F5,F0	MUL.D F6,F6,F0	
6	L.D F11,32(R2)	L.D F12,40(R2)			
7					DADDIU R1,R1,48
8					DADDIU R2,R2,48
9			ADD.D F7,F7,F1	ADD.D F8,F8,F2	
10			ADD.D F9,F9,F3	ADD.D F10,F10,F4	
11			ADD.D F11,F11,F5	ADD.D F12,F12,F6	
12					DSLTU R3,R1,R4
13	S.D F7,-48(R2)	S.D F8,-40(R2)			
14	S.D F9,-32(R2)	S.D F10,-24(R2)			
15	S.D F11,-16(R2)	S.D F12,-8(R2)			BNEZ R3,foo

Figure S.20 15 cycles for 34 operations, yielding 2.67 issues per clock, with a VLIW efficiency of 34 operations for 75 slots = 45.3%. This schedule requires 12 floating-point registers.

Figure S.20 *Continued*

Unrolled 10 times:

Cycle	Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
1	L.D F1,0(R1)	L.D F2,8(R1)			
2	L.D F3,16(R1)	L.D F4,24(R1)			
3	L.D F5,32(R1)	L.D F6,40(R1)	MUL.D F1,F1,F0	MUL.D F2,F2,F0	
4	L.D F7,48(R1)	L.D F8,56(R1)	MUL.D F3,F3,F0	MUL.D F4,F4,F0	
5	L.D F9,64(R1)	L.D F10,72(R1)	MUL.D F5,F5,F0	MUL.D F6,F6,F0	
6	L.D F11,0(R2)	L.D F12,8(R2)	MUL.D F7,F7,F0	MUL.D F8,F8,F0	
7	L.D F13,16(R2)	L.D F14,24(R2)	MUL.D F9,F9,F0	MUL.D F10,F10,F0	DADDIU R1,R1,48
8	L.D F15,32(R2)	L.D F16,40(R2)			DADDIU R2,R2,48
9	L.D F17,48(R2)	L.D F18,56(R2)	ADD.D F11,F11,F1	ADD.D F12,F12,F2	
10	L.D F19,64(R2)	L.D F20,72(R2)	ADD.D F13,F13,F3	ADD.D F14,F14,F4	
11			ADD.D F15,F15,F5	ADD.D F16,F16,F6	
12			ADD.D F17,F17,F7	ADD.D F18,F18,F8	DSLTU R3,R1,R4
13	S.D F11,-80(R2)	S.D F12,-72(R2)	ADD.D F19,F19,F9	ADD.D F20,F20,F10	
14	S.D F13,-64(R2)	S.D F14,-56(R2)			
15	S.D F15,-48(R2)	S.D F16,-40(R2)			
16	S.D F17,-32(R2)	S.D F18,-24(R2)			
17	S.D F19,-16(R2)	S.D F20,-8(R2)			BNEZ R3,foo

Figure S.21 17 cycles for 54 operations, yielding 3.18 issues per clock, with a VLIW efficiency of 54 operations for 85 slots = 63.5%. This schedule requires 20 floating-point registers.