instead. This way all dependent instructions (in this case I3) will get the right value and will become ready and issue. Of course this solution makes the pipeline more complex and one can wonder whether it is justified. May be you have another, better solution???

Problem 3.22

a.

Table 35: Latency of operation under various forwarding assumptions

Source	Destination	No Forwarding	Register Forwarding	Full Forwarding
L.W	INT	3	2	1
L.W	S.W (on the memory operand and the address register)	3	2	1
INT	L.W or S.W (on the address register)	2	1	0
INT	Branch (on register)	2	1	1
L.W	Branch (on register)	3	2	2

If there is no forwarding at all, the dependent instruction can be in ID stage in the cycle right after the parent instruction has left the WB stage.
With register forwarding, a dependent instruction can be in the decode stage in the same cycle when its parent instruction is in the WB stage.
With full forwarding, a dependent instruction can enter execution after the cycle when its parent instruction has completed execution.
The one exception is when the dependent instruction is a conditional branch, because branches are executed in the ID stage and therefore cannot take advantage of the forwarding paths, they can only take advantage of register forwarding. Thus the latency TO a branch is the same for full forwarding as it is for register forwarding.

With full forwarding, the forwarding paths are indicated below:
from ME1/WB1 to EX1, EX2, EX3, and EX4
from ME2/WB2 to EX1, EX2, EX3, and EX4
from EX3/WB3 to EX1, EX2, EX3, and EX4
from EX4/WB4 to EX1, EX2, EX3, and EX4

b. Because the result of CMOVZ instruction is used in the next iteration, there is a data dependency on R1 register even though we unroll the loop three times. Therefore, R1 is not renamed.
In the scheduled code below, the last CMOVZ instruction is executed whether the branch is taken or not because the branch is delayed by one instruction.

Table 36: unrolling the loop three times

Unroll 3 times	Rename	Schedule
LW R5,0(R3)	LW R5,0(R3)	LW R5,0(R3)
SUB R6,R5,R2	SUB R6,R5,R2	LW R8,4(R3)
ADDI R7,R1,#1	ADDI R7,R1,#1	LW R11,8(R3)
CMOVZ R1,R7,R6	CMOVZ R1,R7,R6	ADDI R7,R1,#1
LW R5,0(R3)	LW R8,4(R3)	SUB R6,R5,R2
SUB R6,R5,R2	SUB R9,R8,R2	SUB R9,R8,R2
ADDI R7,R1,#1	ADDI R10,R1,#1	SUB R12,R11,R2
CMOVZ R1,R7,R6	CMOVZ R1,R10,R9	CMOVZ R1,R7,R6
LW R5,0(R3)	LW R11,8(R3)	ADDI R10,R1,#1
SUB R6,R5,R2	SUB R12,R11,R2	ADDI R3,R3,#12
ADDI R7,R1,#1	ADDI R13,R1,#1	CMOVZ R1,R10,R9
CMOVZ R1,R7,R6	CMOVZ R1,R13,R12	ADDI R13,R1,#1
ADDI R3,R3,#12	ADDI R3,R3,#12	BNE R4,R3,SEARCH
BNE R4,R3,SEARCH	BNE R4,R3,SEARCH	CMOVZ R1,R13,R12

No forwarding at all:

Table 37: VLIW program after unrolling the loop 3 times w/o forwarding

	LD/ST 1	LD/ST 2	INT	INT/BRANCH
Search:	LW R5,0(R3)	LW R8,4(R3)	NOOP	NOOP
2	LW R11,8(R3)	NOOP	NOOP	NOOP
3	NOOP	NOOP	ADDI R7,R1,#1	NOOP
4	NOOP	NOOP	NOOP	NOOP
5	NOOP	NOOP	SUB R6,R5,R2	SUB R9,R8,R2
6	NOOP	NOOP	SUB R12,R11,R2	NOOP
7	NOOP	NOOP	NOOP	NOOP
8	NOOP	NOOP	CMOVZ R1,R7,R6	NOOP
9	NOOP	NOOP	NOOP	NOOP
10	NOOP	NOOP	NOOP	NOOP
11	NOOP	NOOP	ADDI R10,R1,#1	ADDI R3,R3,#12
12	NOOP	NOOP	NOOP	NOOP
13	NOOP	NOOP	NOOP	NOOP
14	NOOP	NOOP	CMOVZ R1,R10,R9	NOOP
15	NOOP	NOOP	NOOP	NOOP
16	NOOP	NOOP	NOOP	NOOP
17	NOOP	NOOP	ADDI R13,R1,#1	NOOP
18	NOOP	NOOP	NOOP	NOOP

Table 37: VLIW program after unrolling the loop 3 times w/o forwarding

	LD/ST 1	LD/ST 2	INT	INT/BRANCH
19	NOOP	NOOP	NOOP	BNE R4,R3,SEARCH
20	NOOP	NOOP	CMOVZ R1,R13,R12	NOOP

Register Forwarding only:

Table 38: VLIW program after unrolling the loop 3 times w/ Register Forwarding

	LD/ST 1	LD/ST 2	INT	INT/BRANCH
Search:	LW R5,0(R3)	LW R8,4(R3)	NOOP	NOOP
2	LW R11,8(R3)	NOOP	NOOP	NOOP
3	NOOP	NOOP	ADDI R7,R1,#1	NOOP
4	NOOP	NOOP	SUB R6,R5,R2	SUB R9,R8,R2
5	NOOP	NOOP	SUB R12,R11,R2	NOOP
6	NOOP	NOOP	CMOVZ R1,R7,R6	NOOP
7	NOOP	NOOP	NOOP	NOOP
8	NOOP	NOOP	ADDI R10,R1,#1	NOOP
9	NOOP	NOOP	NOOP	NOOP
10	NOOP	NOOP	CMOVZ R1,R10,R9	NOOP
11	NOOP	NOOP	NOOP	ADDI R3,R3,#12
12	NOOP	NOOP	ADDI R13,R1,#1	NOOP
13	NOOP	NOOP	NOOP	BNE R4,R3,SEARCH
14	NOOP	NOOP	CMOVZ R1,R13,R12	NOOP

Full forwarding:

Table 39: VLIW program after unrolling the loop 3 times w/ Full Forwarding

	LD/ST 1	LD/ST 2	INT	INT/BRANCH
Search:	LW R5,0(R3)	LW R8,4(R3)	NOOP	NOOP
2	LW R11,8(R3)	NOOP	ADDI R7,R1,#1	NOOP
3	NOOP	NOOP	SUB R6,R5,R2	SUB R9,R8,R2
4	NOOP	NOOP	SUB R12,R11,R2	CMOVZ R1,R7,R6
5	NOOP	NOOP	ADDI R10,R1,#1	ADDI R3,R3,#12
6	NOOP	NOOP	CMOVZ R1,R10,R9	NOOP
7	NOOP	NOOP	ADDI R13,R1,#1	BNE R4,R3,SEARCH

Table 39: VLIW program after unrolling the loop 3 times w/ Full Forwarding

	LD/ST 1	LD/ST 2	INT	INT/BRANCH
8	NOOP	NOOP	CMOVZ R1,R13,R12	NOOP

c. The major limitation of loop unrolling is the number of available architectural registers. We have 32 general purpose registers. Five registers are reserved for various reasons and cannot be used for renaming: R0, R1, R2, R3, and R4. Thus 27 registers remain for renaming. Each loop unroll consumes 3 rename registers. Thus the loop can be unrolled 9 times. The problem is: is it useful?
Looking at the 3 pieces of code, we observe that the dependency chain between the CMOVZ and the increment of R1, limits the possible code compression. The minimum execution time of the loop unrolled N times is:
latency_of_Load_to_SUB + latency_of_SUB_to_CMOVZ
+(N-1) x (latency_CMOVZ_to_ADDI + latency_of_ADDI_to_CMOVZ)
Because unrolling the loop increases the code size, it is not advisable to unroll the loop more than 3 times.

Problem 3.23

In this problem, instruction scheduling is speculative.
a. Conservative policy without store-to-load forwarding

Table 40: Tomasulo algorithm with speculation and speculative scheduling

		Dispatch	Issue	Register Fetch	Exec start	Exec complete	Cache	CDB	Retire	Comments
I1	L.D F0,0(R1)	1	2	3	(4)	4	(5)	(6)	7	
I2	L.D F2,-8(R1)	2	3	4	(5)	5	(6)	(7)	8	
I3	ADD.D F0,F2,F0	3	5	6	(7)	11	--	(12)	13	wait for F2
I4	S.D-A F0,-8(R1)	4	5	6	(7)	7	--	--	--	
I5	S.D-C F0,-8(R1)	5	10	11	(12)	12	(13)	--	14	wait for F0
I6	SUBI R1,R1,#8	6	7	8	(9)	9	--	(10)	15	
I7	BNEZ R1,R2,	7	8	9	(10)	10	--	(11)	16	
I8	L.D F2,-8(R1)	8	9	10	(11)	11	(12)	(13)	17	
I9	ADD.D F0,F2,F0	9	11	12	(13)	18	--	(19)	20	wait for F2
I10	S.D-A F0,-8(R1)	10	11	12	(13)	13	--	--	--	
I11	S.D-C F0,-8(R1	11	17	18	(19)	19	(20)	--	21	wait for F0
I12	SUBI R1,R1,#8	12	13	14	(15)	15	--	(16)	22	
I13	BNEZ R1,R2,	13	14	15	(16)	16	--	(17)	23	

The addresses of loads I1 and I2 are both in the L/S queue after clock 5. Store I5 reaches the L/S queue at the end of clock 12. By that time all previous memory access addresses are known and load I2 with the same address has retired. Moreover store I5 reaches the top of the ROB at clock 13 when I3 retires. Therefore all conditions are met for store I5 to access the cache at clock 13.
The address of load I8 is known at the end of clock 11. It is different from the address of store I5 known by the end of clock 7. Thus load i8 can access the cache at clock 12, ahead of store I5.
Store I11 reaches the L/S queue at the end of clock 19. By that time all previous memory accesses have retired and store I11 is at the top of the ROB in clock 20, when I9 retires. Thus store I11 can access the cache in clock 20.