b. Store-to-load forwarding cannot improve the execution of the first two iterations of the loop in this problem. The only opportunity for forwarding would be between store I5 and load I8 (a store followed by a load). However these two memory instructions have different addresses.

c. In the optimistic policy, a load can issue to cache as soon as its address is known, even if the address of prior stores are still unknown. In the schedule of Table 40 all addresses of prior stores are always known when a load reaches the L/S queue. Therefore an optimistic policy for memory disambiguation would not improve the schedule.

Problem 3.24

```
I1              LW R5,0(R1)               /load A
I2              LW R6,0(R2)               /load B
I3              LW R7,0(R3)               /load C
I4              SLT R8,R5,R7              /A<C?
I5              SLT R9,R5,R6              /A<B?
I6              OR R9,R8,R9               /A>=C&&A>=B?
I7      (~R9)   ADD R10,R6,R7             /Yes. Compute B+C in R10.
I8      (~R9)   SW R10,0(R1)              /execute if clause
I9      (~R9)   ADDI R17,R0,#1            /set R17 to skip I17 and I18
I10     (R9)    LW R11,0(R4)              /No. Load D
111     (R9)    ADD R12,R11,R7            /C+D in R12
I12     (R9)    SLT R13,R11,R6            /B>D?
I13     (R9)    SLT R14,R5,R12            /A<C+D?
I14     (R9)    SLT R15,R12,R5            /C+D<A?
I15     (R9)    OR R16,R14,R15            /C+D!=A?
I16     (R9)    AND R17,R13,R16           /B>D&&A!=C+D??
I17     (~R17)  SUB R18,R5,R7             /NO. Execute then clause. A-C in R18
I18     (~R17)  SW R18,0(R2)
I19     exit
```

I9 is inserted to make sure that I17 and I18 are not executed when R9 is equal to 0.

Problem 3.25

a.
BB1: I1~ I5
BB2: I6 ~ I7
BB3: I8 ~ I10
BB4: I11 ~ I13
BB5: I14 ~ I15
BB6: I16 ~ I17

b. LOCAL SCHEDULING
There are 7 execution traces depending on the conditions:
Trace 1: BB1 -> BB2 -> BB3: (A>=C&&A>=B)
Trace 2: BB1 -> BB2 -> BB4 -> BB5 -> BB6: (A>=C&&A<B) and (B>D&&A=C+D)
Trace 3: BB1 -> BB2 -> BB4 -> BB5: (A>=C&&A<B) and (B>D&&A!=C+D)
Trace 4: BB1 -> BB2 -> BB4 -> BB6: (A>=C&&A<B) and (B<=D)
Trace 5: BB1 -> BB4 -> BB5 -> BB6: (A<C) and (B>D&&A=C+D)
Trace 6: BB1 -> BB4 -> BB5: (A<C) and (B>D&&A!=C+D)
Trace 7: BB1 -> BB4 -> BB6: (A<C) and (B<=D)

To compute the speedup, we first need to get the execution time without local scheduling. The orig-

inal code on the VLIW machine is given below. In this VLIW program, no instruction of the original code can be scheduled before a prior instruction in process order. The VLIW program is shown in Table 41.

Table 41: VLIW program with no scheduling

BB	INSTR	LABEL	LD/ST 1	LD/ST 2	INT	INT/BRANCH
1	I1		LW R5,0(R1)	LW R7,0(R3)	NOOP	NOOP
1	I2		NOOP	NOOP	NOOP	NOOP
1	I3		LW R6,0(R2)	NOOP	SLT R8,R5,R7	NOOP
1	I4		NOOP	NOOP	NOOP	NOOP
1	I5		NOOP	NOOP	NOOP	BNEZ R8,else
1	I6		NOOP	NOOP	NOOP	NOOP
2	I7		NOOP	NOOP	SLT R9,R5,R6	NOOP
2	I8		NOOP	NOOP	NOOP	NOOP
2	I9		NOOP	NOOP	NOOP	BNEZ R9,else
2	I10		NOOP	NOOP	NOOP	NOOP
3	I11		NOOP	NOOP	ADD R10,R6,R7	NOOP
3	I12		SW R10,0(R1)	NOOP	NOOP	NOOP
3	I13		NOOP	NOOP	NOOP	J exit
3	I14		NOOP	NOOP	NOOP	NOOP
4	I15	else	LW R11,0(R4)	NOOP	NOOP	NOOP
4	I16		NOOP	NOOP	NOOP	NOOP
4	I17		NOOP	NOOP	SLT R12,R11,R6	NOOP
4	I18		NOOP	NOOP	NOOP	NOOP
4	I19		NOOP	NOOP	NOOP	BEQZ R12,else1
4	I20		NOOP	NOOP	NOOP	NOOP
5	I21		NOOP	NOOP	ADD R13,R7,R11	NOOP
5	I22		NOOP	NOOP	NOOP	NOOP
5	I23		NOOP	NOOP	NOOP	BNE R5,R13,exit
5	I24		NOOP	NOOP	NOOP	NOOP
6	I25	else1	NOOP	NOOP	SUB R14,R5,R7	NOOP
6	I26		SW R14,0(R2)	NOOP	NOOP	NOOP
	I27	exit				

The execution times of basic blocks are:

BB1: 6 clocks (I1-I6)
BB2: 4 clocks (I7-I10)
BB3: 4 clocks (I11-I14)
BB4: 6 clocks (I15-I20)
BB5: 4 clocks (I21-I24)
BB6: 2 clocks (I25-I26)
Thus the execution times of each trace is:
Trace 1: BB1 -> BB2 -> BB3: (A>=C&&A>=B): 6+4+4 = 14
Trace 2: BB1 -> BB2 -> BB4 -> BB5 -> BB6: (A>=C&&A<B) and (B>D&&A=C+D): 6+4+6+4+2 = 22
Trace 3: BB1 -> BB2 -> BB4 -> BB5: (A>=C&&A<B) and (B>D&&A!=C+D): 6+4+6+4 = 20
Trace 4: BB1 -> BB2 -> BB4 -> BB6: (A>=C&&A<B) and (B<=D): 6+4+6+2 = 18
Trace 5: BB1 -> BB4 -> BB5 -> BB6: (A<C) and (B>D&&A=C+D): 6+6+4+2 = 18
Trace 6: BB1 -> BB4 -> BB5: (A<C) and (B>D&&A!=C+D): 6+6+4 = 16
Trace 7: BB1 -> BB4 -> BB6: (A<C) and (B<=D): 6+6+2 = 14

Table 42 shows the VLIW program after local scheduling. Here instructions may be moved within basic blocks but not across branches or jumps.

Table 42: VLIW program with local scheduling

BB	INSTR	LABEL	LD/ST 1	LD/ST 2	INT	INT/BRANCH
1	I1		LW R5,0(R1)	LW R7,0(R3)	NOOP	NOOP
1	I2		NOOP	NOOP	NOOP	NOOP
1	I3		LW R6,0(R2)	NOOP	SLT R8,R5,R7	NOOP
1	I4		NOOP	NOOP	NOOP	NOOP
1	I5		NOOP	NOOP	NOOP	BNEZ R8,else
1	I6		NOOP	NOOP	NOOP	NOOP
2	I7		NOOP	NOOP	SLT R9,R5,R6	NOOP
2	I8		NOOP	NOOP	NOOP	NOOP
2	I9		NOOP	NOOP	NOOP	BNEZ R9,else
2	I10		NOOP	NOOP	NOOP	NOOP
3	I11		NOOP	NOOP	ADD R10,R6,R7	J exit
3	I12		SW R10,0(R1)	NOOP	NOOP	NOOP
4	I13	else	LW R11,0(R4)	NOOP	NOOP	NOOP
4	I14		NOOP	NOOP	NOOP	NOOP
4	I15		NOOP	NOOP	SLT R12,R11,R6	NOOP
4	I16		NOOP	NOOP	NOOP	NOOP
4	I17		NOOP	NOOP	NOOP	BEQZ R12,else1
4	I18		NOOP	NOOP	NOOP	NOOP
5	I19		NOOP	NOOP	ADD R13,R7,R11	NOOP

Table 42: VLIW program with local scheduling

BB	INSTR	LABEL	LD/ST 1	LD/ST 2	INT	INT/BRANCH
5	I20		NOOP	NOOP	NOOP	NOOP
5	I21		NOOP	NOOP	NOOP	BNE R5,R13,exit
5	I22		NOOP	NOOP	NOOP	NOOP
6	I23	else1	NOOP	NOOP	SUB R14,R5,R7	NOOP
6	I24		SW R14,0(R2)	NOOP	NOOP	NOOP
	I25	exit				

After applying local scheduling with delayed branch, the execution time of BB3 could be reduced by two clocks. Other BBs have the same execution time. So only Trace 1 benefits from local scheduling and its execution time becomes 14-2 = 12.

Because of data dependencies, we cannot take advantage of local scheduling, and there is no proper instruction that can be placed after the branch in basic blocks 1 and 3. The speedup for Trace 1 is 14/12=1.17. No speedup is achieved for the other traces.

c.
The student decided not to move stores up across branches. Why is this a good thing?
Since store instructions modify the value in memory, they cannot be executed speculatively. Hence, store instructions cannot be moved up across a branch. The code with a check instruction is given below (w/o branch/jump delay slots):

```
I1              LW R5,0(R1)                 /load A
I2              LW R7,0(R3)                 /load C
I3              SLT R8,R5,R7
I4              LW R6,0(R2)                 /Load B
I11             LW.s R11,0(R4)              /Load D
I12             SLT R12,R11,R6
I14             ADD R13,R7,R11
I6              SLT R9,R5,R6
I8              ADD R10,R6,R7
I16             SUB R14,R5,R7
I5              BNEZ R8,then    /test A<C
I7              BNEZ R9,then                /test A<B
I9              SW R10,0(R1)                /execute if clause
I10             J exit
        then    check.s R11,repair          /inserted where I11 was
I13             BEQZ R12,then1              /test D<B
I15             BNE R5,R13,exit             /test A==C+D
I17     then1   SW R14,0(R2)                /execute then clause
I18     exit
```

The LW instruction is hoisted across a branch and a jump. Thus a check.s instruction is inserted at the original location of the LW to guard against control hazards (unwanted exceptions). Although it would seem that the LW is elevated across a SW, the flowchart reveals that the LW is never executed when the SW is executed (and vice-versa). Thus there is no data hazard. The scheduled code

with the delayed branch (1 instruction) is shown in Table 43.

Table 43: VLIW program with local scheduling

BB	INST	LABEL	LD/ST 1	LD/ST 2	INT	INT/BRANCH
1	I1		LW R5,0(R1)	LW R7,0(R3)	NOOP	NOOP
1	I2		LW R6,0(R2)	LW.s R11,0(R4)	NOOP	NOOP
1	I3		NOOP	NOOP	SLT R8,R5,R7	SUB R14,R5,R7
1	I4		NOOP	NOOP	SLT R12,R11,R6	ADD R13,R7,R11
1	I5		NOOP	NOOP	SLT R9,R5,R6	BNEZ R8,then
1	I6		NOOP	NOOP	ADD R10,R6,R7	NOOP
2	I7		NOOP	NOOP	NOOP	BNEZ R9,then
2	I8		NOOP	NOOP	NOOP	NOOP
3	I9		SW R10,0(R1)	NOOP	NOOP	J exit
3	I10		NOOP	NOOP	NOOP	NOOP
4	I11	then	NOOP	NOOP	**check.s R11,repair**	BEQZ R12,then1
4	I12		NOOP	NOOP	NOOP	NOOP
5	I13		NOOP	NOOP	NOOP	BNE R5,R13,exit
5	I14		NOOP	NOOP	NOOP	NOOP
6	I15	then1	SW R14,0(R2)	NOOP	NOOP	NOOP
	I16	exit				

The execution times of basic blocks are:
BB1: 6 clocks (I1-I6)
BB2: 2 clocks (I7-I8)
BB3: 2 clocks (I9-I10)
BB4: 2 clocks (I11-I12)
BB5: 2 clocks (I13-I14)
BB6: 1 clocks (I15)
Thus the execution times of each trace are:
Trace 1: BB1 -> BB2 -> BB3: (A>=C&&A>=B): 6+2+2 = 10
Trace 2: BB1 -> BB2 -> BB4 -> BB5 -> BB6: (A>=C&&A<B) and (B>D&&A=C+D): 6+2+2+2+1 = 13
Trace 3: BB1 -> BB2 -> BB4 -> BB5: (A>=C&&A<B) and (B>D&&A!=C+D): 6+2+2+2 = 12
Trace 4: BB1 -> BB2 -> BB4 -> BB6: (A>=C&&A<B) and (B<=D): 6+2+2+1 = 11
Trace 5: BB1 -> BB4 -> BB5 -> BB6: (A<C) and (B>D&&A=C+D): 6+2+2+1 = 11
Trace 6: BB1 -> BB4 -> BB5: (A<C) and (B>D&&A!=C+D): 6+2+2 = 10
Trace 7: BB1 -> BB4 -> BB6: (A<C) and (B<=D): 6+2+1 = 9

When is the new code with speculative loads better?

The code with the speculative load performs always better than the original code without load speculation. The speedups are as follows:

Trace 1: 14/10 = 1.4
Trace 2: 22/13 = 1.69
Trace 3: 20/12 =1.67
Trace 4: 18/11 = 1.64
Trace 5: 18/11 = 1.64
Trace 6: 16/10 = 1.6
Trace 7: 14/9 = 1.56

Problem 3.26

a. The code for processing each strip of 64 components is given below:

```
LOOP:   L.V     V1,0(R2),R6             /load X; R6 contains the stride of 1.
        L.V     V2,0(R3),R6             /load Y; 0(R3) is the base address of Y
        MUL.V   V3,V2,V1                /Multiply two vector registers
        ADD.V   V4,V4,V3                /Partial sums accumulate in V4
        ADDI    R2,R2,#64               /This assumes that memory
        ADDI    R3,R3,#64               /addresses point to vector elements
        ADDI    R4,R4,#1
        BNE     R4,R5,LOOP
```

Partial vector sums accumulate in V4. At the end we simply have to add the components of V4. To simplify we have assumed that memory addresses point to vector elements. If vector components have 8 bytes (double precision) and memory is byte-addressable, then the increments on the address registers should be 512.
Given that vector operations are chained, one iteration of the loop takes:
Tite = latency(L.V) + latency(MUL.V)+ latency(ADD.V) +(V.L)-1 = 30+10+5+64-1 = 108 cycles.
Since the loop has to iterate 16 times (16=1024/64), the total number of cycles taken by a dot-product is 108*16 =1728 cycles (ignoring the scalar phase at the end).

b.
For the multiplication of two matrices, a component of the result matrix is obtained by a dot-product of two vectors. Therefore, we need 1024*1024= 1048576 dot-products to multiply two 1024*1024 matrices. The matrix multiply takes $1728*1048576 = 1.812*10^9$ cycles.

c. Unrolling the vector loop twice (after register renaming):

```
LOOP:   L.V     V1,0(R2),R6             /load X
        L.V     V2,0(R3),R6             /load Y
        MUL.V   V3,V2,V1                /Multiply two vector registers
        ADD.V   V4,V4,V3                /Partial-sum
        ADDI    R2,R2, #64
        ADDI    R3,R3, #64
        L.V     V5, 0(R2), R6           /load X
        L.V     V6, 0(R3), R6           /load Y
        MUL.V   V7,V5,V6                /Multiply two vector registers
        ADD.V   V8,V8,V7                / partial-sum
        ADDI    R2,R2,#64
        ADDI    R3,R3,#64
        ADDI    R4,R4,#2
        BNE     R4,R5,LOOP
```

Load and multiplication on each strip of 64 components of the vector are chained and run in parallel. Partial sums are written in two different vector registers, V4 and V8. The scalar processor accumulates the components of the partial sum vectors at the end. Ignoring the scalar phases, the one iteration of the unrolled loop takes 108 cycles. Hence we can compute 128 elements of the vector in each iteration, and the number of iterations for the vector with size of 1024 is halved and it is 8(=1024/128).Therefore, the total number of cycle to execute the dot-product is 108 * 8 = 864