Last updated:
11/15/2022

Problem M14.1: Microprogramming and Bus-Based Architectures

Problem M14.1.A — Memory-to-Memory Add

Worksheet M14.1-1 shows one way to implement ADDm in microcode.

Note that to maintain "clean" behavior of your microcode, no registers in the register file should change their value during execution (unless they are written to). This does not refer to the registers in the datapath (IR, A, B, MA). Thus, using asterisks for the load signals (ldIR, ldA, ldB, and ldMA) is acceptable as long as the correctness of your microcode is not affected.

Problem M14.1.B — Implementing DBNEZ Instruction

The question asked to jump to PC+4+offset. This ignores that the immediate value needs to be shifted left by 2 before it can be added to PC+4, to make sure we don't run into alignment problems. We did this because the data path given doesn't really have facilities for shifting.

Worksheet M14.1-2 shows one way to implement DBNEZ in microcode.

Problem M14.1.C — Implementing RETZ Instruction

Worksheet M14.1-3 shows one way to implement RETZ in microcode.

Problem M14.1.D — Implementing CALL Instruction

Worksheet M14.1-4 shows one way to implement CALL in microcode.

Problem M14.1.E — Instruction Execution Times

Instruction	Cycles
`SUB  R3,R2,R1`	3 + 3 = 6
`SUBI R2,R1,#4`	3 + 3 = 6
`SW   R1,0(R2)`	3 + 5 = 8
`BNEZ R1,label  # (R1 == 0)`	3 + 2 = 5
`BNEZ R1,label  # (R1 != 0)`	3 + 5 = 8
`BEQZ R1,label  # (R1 == 0)`	3 + 5 = 8
`BEQZ R1,label  # (R1 != 0)`	3 + 2 = 5
`J    label`	3 + 3 = 6
`JR   R1`	3 + 2 = 5
`JAL  label`	3 + 4 = 7
`JALR R1`	3 + 4 = 7

As discussed in Lecture 21, instruction execution includes the number of cycles needed to fetch the instruction. The lecture notes used 4 cycles for the fetch phase, while Worksheet 1 shows that this phase can actually be implemented in 3 cycles —either answer is fine. The above table uses 3 cycles for the fetch phase. Overall, SW, BNEZ (for a taken branch), and BEQZ (for a taken branch) take the most cycles to execute (8), while BNEZ (for a not-taken branch), BEQZ (for a not-taken branch) and JR take the fewest cycles (5).

State	PseudoCode	Ld IR	Reg Sel	Reg W	en Reg	ld A	ld B	ALUOp	en ALU	ld MA	Mem W	en Mem	Ex Sel	en Imm	μBr	Next State
FETCH0:	MA <- PC; A <- PC	0	PC	0	1	1	*	*	0	1	*	0	*	0	N	*
	IR <- Mem	1	*	*	0	0	*	*	0	*	0	1	*	0	N	*
	PC <- A+4; dispatch	0	PC	1	1	*	*	INC_A_4	1	*	*	0	*	0	D	*
. . .																
NOP0:	microbranch Back to FETCH0	0	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH0
ADDm0:	**MA <- R[rs]**	**0**	**rs**	**0**	**1**	*****	*****	*****	**0**	**1**	*****	**0**	*****	**0**	**N**	*****
	A <- Mem	**0**	*****	*****	**0**	**1**	*****	*****	**0**	*****	**0**	**1**	*****	**0**	**N**	*****
	MA <- R[rt]	**0**	**rt**	**0**	**1**	**0**	*****	*****	**0**	**1**	*****	**0**	*****	**0**	**N**	*****
	B <- Mem	**0**	*****	*****	**0**	**0**	**1**	*****	**0**	*****	**0**	**1**	*****	**0**	**N**	*****
	MA <- R[rd]	*****	**rd**	**0**	**1**	**0**	**0**	*****	**0**	**1**	*****	**0**	*****	**0**	**N**	*****
	Mem <- A+B; fetch	*****	*****	*****	**0**	*****	*****	**ADD**	**1**	*****	**1**	**1**	*****	**0**	**J**	**FETCH0**

Worksheet M14.1-1: Implementation of the ADDm instruction

State	PseudoCode	ld IR	Reg Sel	Reg W	en Reg	ld A	ld B	ALUOp	en ALU	Ld MA	Mem W	en Mem	Ex Sel	en Imm	μBr	Next State
FETCH0:	MA <- PC; A <- PC	*	PC	0	1	1	*	*	0	1	*	0	*	0	N	*
	IR <- Mem	1	*	*	0	0	*	*	0	*	0	1	*	0	N	*
	PC <- A+4; B <- A+4	0	PC	1	1	*	1	INC_A_4	1	*	*	0	*	0	D	*
. . .																
NOP0:	microbranch back to FETCH0	*	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH0
DBNEZ:	**A <- rs**	**0**	**rs**	**0**	**1**	**1**	**0**	*****	**0**	*****	*****	**0**	*****	**0**	**N**	*****
	rs <- A – 1 μB to FETCH0 if zero	**0**	**rs**	**1**	**1**	*****	**0**	**DEC_A_1**	**1**	*****	*****	**0**	*****	**0**	**Z**	**FETCH0**
	A <- sExt16(IR)	*****	*****	*****	**0**	**1**	**0**	*****	**0**	*****	*****	**0**	**sExt16**	**1**	**N**	*****
	PC <- A+B jump to FETCH0	*****	**PC**	**1**	**1**	*****	*****	**ADD**	**1**	*****	*****	**0**	*****	**0**	**J**	**FETCH0**

Worksheet M14.1-2: Implementation of the DBNEZ Instruction

State	PseudoCode	Ld IR	Reg Sel	Reg W	en Reg	ld A	ld B	ALUOp	en ALU	Ld MA	Mem W	en Mem	Ex Sel	en Imm	μBr	Next State
FETCH0:	MA <- PC; A <- PC	*	PC	0	1	1	*	*	0	1	*	0	*	0	N	*
	IR <- Mem	1	*	*	0	0	*	*	0	*	0	1	*	0	N	*
	PC <- A+4; B <- A+4	0	PC	1	1	*	1	INC_A_4	1	*	*	0	*	0	D	*
. . .																
NOP0:	microbranch back to FETCH0	*	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH0
retz0	**A <- Reg[Rs]**	**0**	**Rs**	**0**	**1**	**1**	*****	*****	**0**	*****	*****	**0**	*****	**0**	**N**	*****
retz1	**A <- Reg[Rt] MA <- Reg[Rt] uBr to retz3 if zero**	**0**	**Rt**	**0**	**1**	**1**	*****	**COPY_A**	**0**	**1**	*****	**0**	*****	**0**	**Z**	**retz3**
retz2		*****	*****	*****	**0**	*****	*****	*****	**0**	*****	*****	**0**	*****	**0**	**J**	**FETCH0**
retz3	**PC <- MEM**	**0**	**PC**	**1**	**1**	**0**	*****	*****	**0**	*****	**0**	**1**	*****	**0**	**N**	*****
retz4	**Reg[Rt] < A+4**	*****	**Rt**	**1**	**1**	*****	*****	**INC_A_4**	**1**	*****	*****	**0**	*****	**0**	**J**	**FETCH0**

Worksheet M14.1-3: Implementation of the RETZ Instruction

State	PseudoCode	ld IR	Reg Sel	Reg W	en Reg	ld A	ld B	ALUOp	en ALU	Ld MA	Mem W	en Mem	Ex Sel	en Imm	μBr	Next State
FETCH0:	MA <- PC; A <- PC	*	PC	0	1	1	*	*	0	1	*	0	*	0	N	*
	IR <- Mem	1	*	*	0	0	*	*	0	*	0	1	*	0	N	*
	PC <- A+4; B <- A+4	0	PC	1	1	*	1	INC_A_4	1	*	*	0	*	0	D	*
. . .																
NOP0:	microbranch back to FETCH0	*	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH0
CALL:	**MA <- R[ra]; A <- R[ra]**	**0**	**ra**	**0**	**1**	**1**	**0**	*****	**0**	**1**	*****	**0**	*****	**0**	**N**	*****
	Mem <- B	**0**	*****	*****	**0**	**0**	**0**	**COPY_B**	**1**	*****	**1**	**1**	*****	**0**	**N**	*****
	R[ra] <- A - 4	**0**	**ra**	**1**	**1**	*****	**0**	**DEC_A_4**	**1**	*****	*****	**0**	*****	**0**	**N**	*****
	A <- sExt16(IR)	*****	*****	*****	**0**	**1**	**0**	*****	**0**	*****	*****	**0**	**sExt16**	**1**	**N**	*****
	PC <- A+B; jump to FETCH0	*****	**PC**	**1**	**1**	*****	*****	**ADD**	**1**	*****	*****	**0**	*****	**0**	**J**	**FETCH0**

Worksheet M14.1-4: Implementation of the CALL Instruction

Problem M14.1.F Exponentiation

In the given code, 'm' and 'n' are always nonnegative integers. Therefore, we don't have to worry about the cases where 'i' is larger than 'n' or 'j' is larger than 'm'. Also, for this problem, 0 raised to any power is just 0, while any nonzero value raised to the 0^{th} power is 1. Note that the pseudo code that is given returns a value of 0 when 0 is raised to the 0^{th} power. However, the actual `pow()` function in the standard C library returns a value of 1 for this case. We present the solution that implements the pseudo code given in the problem rather than C's `pow()` function.

```
#
# R5: temp, R6: j
#

            ADD   R3, R0, R0        ; put 0 in result
            BEQZ  R1, _END_I        ; if m is 0, end
            ADDI  R3, R0, #1        ; put 1 in result
            BEQZ  R2, _END_I        ; if n is 0, the loop is over; we set
                                    ; i equal to n and count down to 0—since
                                    ; R2 does not have to be preserved, we
                                    ; use it for i
            SUBI  R5, R1, #1        ; temp = m – 1
            BEQZ  R5, _END_I        ; if m is 1, the result will be 1,
                                    ; so end the program
_START_I:
            ADD   R5, R0, R3        ; temp = result
            SUBI  R6, R1, #1        ; j = m – 1 (the number of times to
                                    ; execute the second loop)
_START_J:
            ADD   R3, R3, R5        ; result += temp
            SUBI  R6, R6, #1        ; j--
            BNEZ  R6, _START_J      ; Re-execute loop until j reaches 0
_END_J:
            SUBI  R2, R2, #1        ; i--
            BNEZ  R2, _START_I      ; Re-execute loop until i reaches 0
_END_I:
```

To compute the number of instructions and cycles to execute this code, let us consider subsets of the code.

Code	# of instructions	# of cycles
`ADD R3, R0, R0` `BEQZ R1, _END_I`	2	6×1 + 8×1 = 14 (m = 0) 6×1 + 5×1 = 11 (m > 0)
`ADDI R3, R0, #1` `BEQZ R2, _END_I`	2 (if m > 0)	6×1 + 8×1 = 14 (n = 0) 6×1 + 5×1 = 11 (n > 0)
`SUBI R5, R1, #1` `BEQZ R5, _END_I`	2 (if m > 0 and n > 0)	6×1 + 8×1 = 14 (m =1) 6×1 + 5×1 = 11 (m > 1)

`_START_I:`
`ADD R5, R0, R3`
`SUBI R6, R1, #1`
	2n (if m > 1 and n > 0)	(6×2)×n = 12n

`_START_J:`
`ADD R3, R3, R5`
`SUBI R6, R6, #1`
`BNEZ R6, _START_J`
	3n(m-1)
	(if m > 1 and n > 0)
		(6×2 + 5×1)×n + (6×2 + 8×1) ×(m-2)×n = 17n + 20n(m-2)

`_END_J:`
`SUBI R2, R2, #1`
`BNEZ R2, _START_I`
	2n (if m > 1 and n > 0)	(6 + 8)×n – 3 = 14n-3

From the above table, we can complete the table given in the problem.

m,n	Instructions	Cycles
0, 1	2	14
1, 0	4	25
2, 2	20	116
3, 4	46	282
M, N (M = 0)	2	14
M, N (M > 0, N = 0)	4	25
M, N (M = 1, N > 0)	6	36
M, N (M > 1, N > 0)	3N(M-1)+4N+6	20N(M-2)+43N+30

Problem M14.1.G — Microcontroller Jump Logic

One way to start designing the microcontroller jump logic is to write out a table of the input signals and the output bits. For clarity, the bits that encode the μJumpTypes are labeled A, B and C, from left to right. The output bits are labeled H and L, also from left to right. So the table we need to implement is the following (where asterisks are for the input bits that we don't care about).

Input bits					Output bits	
A	B	C	Zero	Busy	H	L
0	0	0	*	*	0	0
0	0	1	*	0	0	0
0	0	1	*	1	0	1
0	1	0	*	*	1	0
1	0	0	*	*	1	1
1	1	0	0	*	0	0
1	1	0	1	*	1	0
1	1	1	0	*	1	0
1	1	1	1	*	0	0

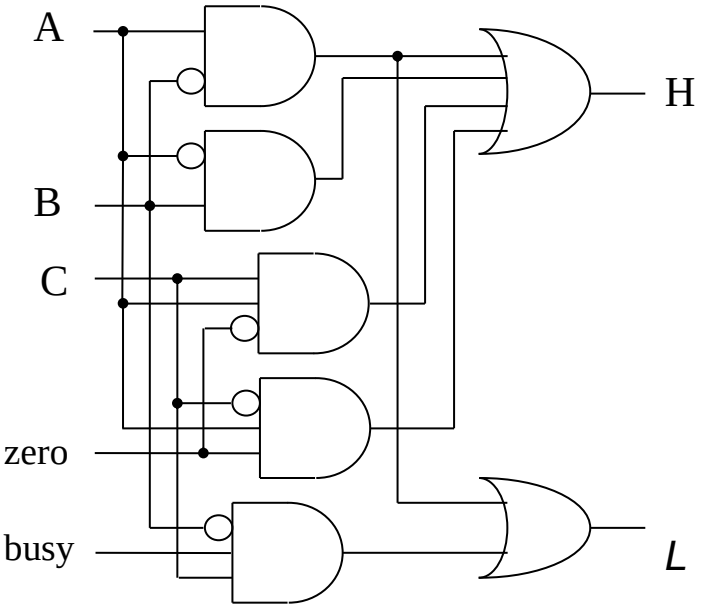
Writing out boolean equations for the H and L output bits (by directly recognizing only the lines which have logical ones as output) we find

$$H = A\overline{B}\,\overline{C} + \overline{A}B\overline{C} + AB\overline{C}\cdot zero + ABC\cdot\overline{zero}$$
$$L = \overline{A}\,\overline{B}C\cdot busy + A\overline{B}\,\overline{C}$$

Also, we do not care about the output when the μJump type is 011 or 101, since those are invalid encodings. Thus we can simplify the equations to

$$H = A\overline{B} + \overline{A}B + A\overline{C}\cdot zero + AC\cdot\overline{zero}$$
$$L = \overline{B}C\cdot busy + A\overline{B}$$

Drawing this out as gates we get



Problem M14.2: VLIW Programming

Problem M14.2.A

To get 1 cycle per vector element performance, we need to use loop unrolling and software pipelining. The original loop is unrolled four times and software pipelined. Two registers (**`F3`** and **`F7`**) are used for saving partial sums, which are summed at the end.

At the start of the program `n` may be any value. By making successive checks and providing fix-up code, `n` can be guaranteed to be positive and a multiple of 4 by the prolog.

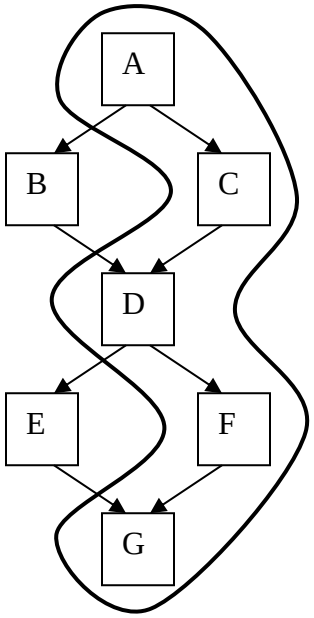
```
// R1 - points to X
// R2 - points to Y
// R5 - n
// F7 – result

    // clear partial sum registers
    MOVI2FP F3,R0
    MOVI2FP F7,R0

    // clear temporary registers used for multiply results
    MOVI2FP F2,R0
    MOVI2FP F6,R0
    MOVI2FP F10,R0
    MOVI2FP F14,R0

    // n must be greater than 0
    SGT     R3,R5,R0
    BEQZ    R3,end     // if !(n>0) goto end

    // n must be greater than 0
    ANDI  R3,R5,#3
    BEQZ  R3,prolog

    // (n>0) && ((n%4)!=0)
    SUB   R5,R5,R3
L1:
    L.S   F3,0(R1); L.S  F4,0(R2); SUBI R3,R3,#1
    MUL.S F3,F3,F4; ADDI R1,R1,#4;
    ADD.S F7,F7,F3; ADDI R2,R2,#4; BNEZ R3,L1

    BEQZ  R5,end

    // (n>=4) && ((n%4)==0)
prolog:
    L.S  F0, 0(R1); L.S  F1, 0(R2); SUBI R5,R5,#4
    L.S  F4, 4(R1); L.S  F5, 4(R2); ADDI R1,R1,#16
    L.S  F8,-8(R1); L.S  F9, 8(R2); ADDI R2,R2,#16
    L.S F12,-4(R1); L.S F13,-4(R2); BEQZ R5,epilog

    L.S  F0, 0(R1); L.S  F1, 0(R2); MUL.S  F2, F0, F1; SUBI R5,R5,#4
    L.S  F4, 4(R1); L.S  F5, 4(R2); MUL.S  F6, F4, F5; ADDI R1,R1,#16
    L.S  F8,-8(R1); L.S  F9, 8(R2); MUL.S F10, F8, F9; ADDI R2,R2,#16
    L.S F12,-4(R1); L.S F13,-4(R2); MUL.S F14,F12,F13; BEQZ R5,epilog
```

```
loop:
    L.S  F0, 0(R1); L.S  F1, 0(R2); MUL.S  F2, F0, F1; ADD.S F3,F3, F2; SUBI R5,R5,#4
    L.S  F4, 4(R1); L.S  F5, 4(R2); MUL.S  F6, F4, F5; ADD.S F7,F7, F6; ADDI R1,R1,#16
    L.S  F8,-8(R1); L.S  F9, 8(R2); MUL.S F10, F8, F9; ADD.S F3,F3,F10; ADDI R2,R2,#16
    L.S F12,-4(R1); L.S F13,-4(R2); MUL.S F14,F12,F13; ADD.S F7,F7,F14; BNEZ R5,loop

epilog:
    MUL.S  F2, F0, F1; ADD.S F3,F3, F2
    MUL.S  F6, F4, F5; ADD.S F7,F7, F6
    MUL.S F10, F8, F9; ADD.S F3,F3,F10
    MUL.S F14,F12,F13; ADD.S F7,F7,F14

    ADD.S F3,F3, F2
    ADD.S F7,F7, F6
    ADD.S F3,F3,F10
    ADD.S F7,F7,F14

    ADD.S F7,F7,F3

end:
```

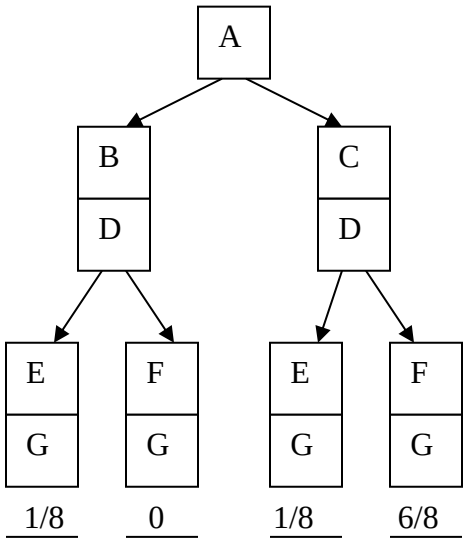
Problem M14.3: Trace Scheduling

Problem M14.3.A

Program's control flow graph



Decision tree



Problem M14.3.B

```
ACF: ld   r1, data
          div  r3, r6, r7 ;; X <- V2/V3
          mul  r8, r6, r7 ;; Y <- V2*V3
D:        andi r2, r1, 3  ;; r2 <- r1%4
          bnez r2, G
A:        andi r2, r1, 7  ;; r2 <- r1%8
          bnez r2, E
B:        div  r3, r4, r5 ;; X <- V0/V1
E:        mul  r8, r4, r5 ;; Y <- V0*V1
G:
```

Problem M14.3.C

Assume that the load takes x cycles, divide takes y cycles, and multiply takes z cycles. Approximately how many cycles does the original code take? (ignore small constants)
x+max(y,z)

Approximately how many cycles does the new code take in the best case? **max(x,y,z)**

Problem M14.4: Scalar vs. VLIW Processors

Problem M14.4.A

(a) No stalls for the I1->I3 or I2->I5 load-use dependencies, as the compiler has scheduled these instructions far enough apart.
I3->I5 and I5->I7 each require one stall to wait for a floating-point operation.
Total: 2 stalls

(b) 8 instructions + 2 stalls = 10 cycles per iteration, so with 2 floating point operations this gives: 2/10 **= 1/5 = 0.2 FLOPs/cycle**

Problem M14.4.B

Unrolling by 2 is sufficient. We could simply interleave instructions from two iterations, which works since the original loop never required two stalls for a single instruction. Although not required, for this particular loop, we can also reduce the bookkeeping instructions and an unrolling factor of 2 is still sufficient. The following code with only two additions (which is the minimum possible) still has zero stalls:

```
I1: loop: ld f0, 0(r2)        ;; Load B[i]
I2:       ld f5, 4(r2)        ;; Load B[i+1]
I3:       ld f2, 0(r1)        ;; Load A[i]
I4:       ld f6, 4(r1)        ;; Load A[i+1]
I5:       fadd f3, f0, f1
I6:       fadd f7, f5, f1
I7:       addi r1, r1, 8
I8:       fmul f4, f2, f3
I9:       fmul f8, f6, f7
I10:      addi r2, r2, 8
I11:      st f4, -8(r1)       ;; Store A[i]
I12:      st f8, -4(r1)       ;; Store A[i+1]
I13:      bne r1, r3, loop
```

Problem M14.4.C

(a) Unrolling by **3** is sufficient. The most critical dependencies are fadd->fmul->st, due to the long-latency floating-point ops which take 3 cycles. Unrolling by a factor of 2 does not give enough ops to fill the cycles (since we aren't satisfied to fill a cycle with just an integer ALU op). Once you unroll by 3, matching the longest latency, you can interleave ops from three

fadd->fmul->st chains to put at least one op in each cycle. For example, you could have extended and filled in the table on the previous page to get this schedule for unrolling by 3:

Inst.	ALU/Branch Unit	Memory Unit	Floating Point Unit
1		`ld f0, 0(r2)`	
2		`ld f5, 4(r2)`	
3		`ld f9, 8(r2)`	`fadd f3, f0, f1`
4	`addi r2, r2, 12`	`ld f2, 0(r1)`	`fadd f7, f5, f1`
5		`ld f6, 4(r1)`	`fadd f11, f9, f1`
6		`ld f10, 8(r1)`	`fmul f4, f2, f3`
7	`addi r1, r1, 12`		`fmul f8, f6, f7`
8			`fmul f12, f10, f11`
9		`st f4, -12(r1)`	
10		`st f8, -8(r1)`	
11	`bne r1, r3, loop`	`st f12, -4(r1)`	

(b) The loop performs 3 memory operations for every 2 floating point operations. We can issue at most 1 memory op per cycle, so the peak throughput is **2/3 = 0.67 FLOPs/cycle.** (Addendum: Software pipelining and its associated hardware support is not actually needed to achieve this ideal throughput: simply unrolling by a larger factor can result in enough memory ops to keep the memory unit fully utilized.)

Problem M14.5: VLIW & Vector Coding

Ben Bitdiddle has the following C loop, which takes the absolute value of elements within a vector.

```
for (i = 0; i < N; i++) {
    if (A[i] < 0)
        A[i] = -A[i];
}
```

Problem M14.5.A

```
; Initial Conditions:
;     R1 = N
;     R2 = &A[0]


      SGT R3, R1, R0
      BEQZ R3, end                                  ; R3 = (N > 0) | special case N ≤ 0
loop: LW R4, 0(R2)       | SUBI R1, R1, #1        ; R4 = A[i] | N--
      SLT R5, R4, R0     | ADDI R2, R2, #4        ; R5 = (A[i] < 0) | R2 = &A[i+1]
      BEQZ R5, next      |                        ; skip if (A[i]≥0)
      SUB R4, R0, R4     |                        ; A[i] = -A[i]
      SW R4, -4(R2)      |                        ; store updated value of A[i]
next: BNEZ R1, loop      |                        ; continue if N > 0
end:
```

Average Number of Cycles: ½ × (6 + 4) = 5

```
; SOLUTION #2

      SGT R3, R1, R0
      BNEZ R3, end                                  ; R3 = (N > 0) | special case N ≤ 0
loop: LW R4, 0(R2)       | SUBI R1, R1, #1        ; R4 = A[i] | N--
      SLT R5, R4, R0     | ADDI R2, R2, #4        ; R5 = (A[i] < 0) | R2 = &A[i+1]
      BNEZ R5, next      | SUB R4, R0, R4         ; skip if (A[i]≥0) | A[i] = -A[i]
      SW R4, -4(R2)      |                        ; store updated value of A[i]
next: BNEZ R1, loop      |                        ; continue if N > 0
end:
```

Average Number of Cycles: ½ × (5 + 4) = 4.5

NOTE: Although this solution minimizes code size and average number of cycles per element for this loop, it causes extra work because it subtracts regardless of whether it has to or not.

Problem M14.5.B

```
      SGT R3, R1, R0
      BNEZ R3, end                                ; R3 = (N > 0) | if N ≤ 0
loop: LW R4, 0(R2)            | SUBI R1, R1, #1 ; R4 = A[i] | N--
      CMPLTZ P0, R4           | ADDI R2, R2, #4 ; P0 = (A[i]<0) | R2 = &A[i+1]
      (P0) SUB R4, R0, R4     |                 ; A[i] = -A[i]
      (P0) SW R4, -4(R2)      | BNEZ R1, loop   ; store updated value of A[i]
end:                                            ; continue if N > 0
```

Average Number of Cycles: ½ × (4 + 4) = 4 Cycles

Problem M14.5.C

```
; Initial Conditions:
;     R1 = N
;     R2 = &A[i]

R3 = N > 0
R4 = A[i]
R5 = N odd
R6 = A[i+1]

      SGT R3, R1, R0
      BEQZ R3, end            | ANDI R5, R1, #1
      BEQZ R5, loop           | LW R4, 0(R2)
      CMPLTZ P0, R4           | SUBI R1, R1, #1
      ADDI R2, R2, #4         | (P0) SUB R4, R0, R4
      (P0) SW R4, -4(R2)      | BEZ R1, end

loop: LW R4, 0(R2)            | SUBI R1, R1, #2
      CMPLTZ P0, R4           | LW R6, 4(R2)
      (P0) SUB R4, R0, R4     | CMPLTZ P1, R6
      (P0) SW R4, 0(R2)       | (P1) SUB R6 R0, R6
                              | (P1) SW R6 4(R2)
      ADDI R2, R2, #8         | BNEZ R1, loop
end:
```

Average Number of Cycles: 6 for 2 elements = 3 cycles per element

Problem M14.5.D

```
; Initial Conditions:
;     R1 = N
;     R2 = &A[i]

      L.D F0, #0
      MTC1 VLR R1             # operate on all N elements
      CVM
      LV V1, R2               # load A
      SLTVS.D V1, F0          # setup the mask vector
      SUBSV.D V1, F0, V1      # negate appropriate elements
      SV R2, V1               # store back changes
```

Average Number of Cycles: ≈ (N/2 + N/2) / N ≈ 1 cycle per element (assuming chaining)

Note: Because there is only one ALU per lane, only the load and the SLT (Set-Less-Than) can be chained together, while the subtract and the store can be chained together. Execution time (per element) of the other instructions is negligible when N is large.

Problem M14.5.E

```
; assume m = known vector length
; Initial Conditions:
;     R1 = N
;     R2 = &A[i]

      L.D F0, #0
      ANDI R3, R1, (m-1)      # get N%m - assume m is a power of 2
      MTC1 VLR R3             # operate on first N%m elements
      LV V1, R2               # load A
      SLTVS.D V1, F0          # setup the mask vector
      SUBSV.D V1, F0, V1      # negate appropriate elements
      SV R2, V1               # store back changes
      SUB R1, R1, R3          # decrease i by N%m (i is divisible by m now)
      SLLI R3, R3, #2         # (we're counting i down)
      ADDI R2, R2, R3         # advance A pointer
      BEQZ R1, end            # i == 0 -> done
      ADDI R3, R0, m
      MTC1 VLR R3             # operate on all elements

loop:
      CVM
      LV V1, R2               # load A
      SLTVS.D V1, F0          # setup the mask vector
      SUBSV.D V1, F0, V1      # negate appropriate elements
      SV R2, V1               # store back changes
      ADDI R2, R2, (m*4)      # advance A pointer
      SUBI R1, R1, m          # decrease i by m
      BNEZ R1, loop           # done?
```

```
end:
      CVM
```

Problem M14.6: Predication and VLIW

Problem M14.6.A

```
      l.s   f1, 0(r1)     ; f1 = *r1
      seq.s r5, f10, f1   ; r5 = (f10==f1)
      cmpnez p1, r5       ; p1 = (r5!=0)
(p1)  add.s f2, f1, f11   ; if (p1) f2 = f1+f11
(!p1) add.s f2, f1, f12   ; if(!p1) f2 = f1+f12
      s.s   f2, 0(r2)     ; *r2 = f2
```

Problem M14.6.B

See the next page (**Table M14.6-2**).

Label	integer op	floating point add	memory op	branch
`loop:`			`l.s f1,0(r1)`	
			`l.s f3,4(r1)`	
	`addi r1, r1, #8`	`cmpnez p1, f1`		
		`cmpnez p3, f3`		
		`(p1) add.s f2, f1, f1`		
		`(p3) add.s f4, f3, f3`		
			`(p1) s.s f2, -8(r1)`	
			`(p3) s.s f4, -4(r1)`	`bneq r1, r2, loop`

Table M14.6-1

label	integer op	floating point add	memory op	branch
			`l.s f1,0(r1)`	
			`l.s f3,4(r1)`	
	`addi r1, r1, #8`	`cmpnez p1, f1`		
		`cmpnez p3, f3`		`beq r1, r2, epilog`
`loop:`		**`(p1) add.s f2, f1, f1`**	**`l.s f1,0(r1)`**	
		`(p3) add.s f4, f3, f3`	**`l.s f3,4(r1)`**	
	`addi r1, r1, #8`	**`cmpnez p1, f1`**	**`(p1) s.s f2, -8(r1)`**	
		`cmpnez p3, f3`	**`(p3) s.s f4, -12(r1)`**	`bneq r1, r2,loop`
`epilog:`		`(p1) add.s f2, f1, f1`		
		`(p3) add.s f4, f3, f3`		
			`(p1) s.s f2, -8(r1)`	
			`(p3) s.s f2, -4(r1)`	

Table M14.6-2

Problem M14.7: Vector Machines

Problem M14.7.A

Consider the implementation of the C-code on the vector machine that executes in a minimum number of cycles. Assuming the following initial values, insert vector instructions to complete the implementation.

- R1 points to A[0]
- R2 points to B[0]
- R3 points to C[0]
- R4 contains the value 328

```
     ANDI R5, R4, 31       # 328 mod 32
     MTC1 VLR, R5          # set VLR to remainder
loop:
     LV   V1, R1           # load A
     LV   V2, R2           # load B
     LV   V3, R3           # load C
     MULV V4, V2, V1       # A * B
     ADDV V5, V3, V4       # C + A
     SV   V4, R1           # store A
     SV   V5, R3           # store C
     SLL  R7, R5, 2
     ADD  R1, R1, R7       # increment A ptr
     ADD  R2, R2, R7       # increment B ptr
     ADD  R3, R3, R7       # increment C ptr
     SUB  R4, R4, R5       # update loop counter
     LI   R5, 32           # reset VLR to max
     MTC1 VLR, R5
     BGTZ R4, loop
```

Problem M14.7.B

The following **supplementary information** explains the diagram.

Scalar instructions execute in 5 cycles: fetch (**F**), decode (**D**), execute (**X**), memory (**M**), and writeback (**W**). A vector instruction is also fetched (**F**) and decoded (**D**). Then, it stalls (—) until its required vector functional unit is available. With no chaining, a dependent vector instruction stalls until the previous instruction finishes writing back ALL of its elements. A vector instruction is pipelined across all the lanes in parallel. For each element, the operands are read (**R**) from the vector register file, the operation executes on the load/store unit (**M**) or the ALU (**X**) or the MUL (**Y**), and the result is written back (**W**) to the vector register file. Assume that there is no structural conflict on the writeback port. A stalled vector instruction does not block a scalar instruction from executing.

LV_1 and LV_2 refer to the first and second LV instructions in the loop.

instr.	cycle																																															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40							
LV_1	F	D	R	M1	M2	M3	M4	W																																								
LV_1				R	M1	M2	M3	M4	W																																							
LV_1					R	M1	M2	M3	M4	W																																						
LV_1						R	M1	M2	M3	M4	W																																					
LV_2		F	D	—	—	—	R	M1	M2	M3	M4	W																																			
LV_2								R	M1	M2	M3	M4	W																																		
LV_2									R	M1	M2	M3	M4	W																																	
LV_2										R	M1	M2	M3	M4	W																															
LV_3			F	D	—	—	—	—	—	—	R	M1	M2	M3	M4	W																													
LV_3												R	M1	M2	M3	M4	W																											
LV_3													R	M1	M2	M3	M4	W																											
LV_3														R	M1	M2	M3	M4	W																											
MULV				F	D	—	—	—	—	—	—	—	—	—	—	R	Y1	Y2	W																											
MULV																	R	Y1	Y2	W																									
MULV																		R	Y1	Y2	W																									
MULV																			R	Y1	Y2	W																									
ADDV					F	D	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	R	X1	W																					
ADDV																								R	X1	W																			
ADDV																									R	X1	W																		
ADDV																										R	X1	W																	
SV_1						F	D	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	R	M1	M2	M3	M4	W																		
SV_1																							R	M1	M2	M3	M4	W																	
SV_1																								R	M1	M2	M3	M4	W																
SV_1																									R	M1	M2	M3	M4	W															
SV_2							F	D	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	R	M1	M2	M3	M4	W												
SV_2																													R	M1	M2	M3	M4	W									
SV_2																														R	M1	M2	M3	M4	W								
SV_2																															R	M1	M2	M3	M4	W						

Problem M14.7.C

instr.	cycle																																						
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
LV_1	F	D	R	M1	M2	M3	M4	W																																
LV_1				R	M1	M2	M3	M4	W																															
LV_1					R	M1	M2	M3	M4	W																														
LV_1						R	M1	M2	M3	M4	W																													
LV_2		F	D	—	—	—	R	M1	M2	M3	M4	W																												
LV_2								R	M1	M2	M3	M4	W																											
LV_2									R	M1	M2	M3	M4	W																										
LV_2										R	M1	M2	M3	M4	W																									
LV_3			F	D	—	—	—	—	—	—	R	M1	M2	M3	M4	W																								
LV_3												R	M1	M2	M3	M4	W																							
LV_3													R	M1	M2	M3	M4	W																						
LV_3														R	M1	M2	M3	M4	W																					
MULV				F	D	—	—	—	—	—	—	R	Y1	Y2	W																									
MULV													R	Y1	Y2	W																								
MULV														R	Y1	Y2	W																							
MULV															R	Y1	Y2	W																						
ADDV					F	D	—	—	—	—	—	—	—	—	—	R	X1	W																						
ADDV																	R	X1	W																					
ADDV																		R	X1	W																				
ADDV																			R	X1	W																			
SV_1						F	D	—	—	—	—	—	—	—	R	M1	M2	M3	M4	W																				
SV_1																R	M1	M2	M3	M4	W																			
SV_1																	R	M1	M2	M3	M4	W																		
SV_1																		R	M1	M2	M3	M4	W																	
SV_1							F	D	—	—	—	—	—	—	—	—	—	—	R	M1	M2	M3	M4	W																
SV_2																				R	M1	M2	M3	M4	W															
SV_2																					R	M1	M2	M3	M4	W														
SV_2																						R	M1	M2	M3	M4	W													
				1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19																		

Problem M14.7.D

What is the performance (flops/cycle) of the program with chaining?

2*32/19

Problem M14.7.E

Would loop unrolling of the assembly code improve performance without chaining? Explain. (You may rearrange the instructions when performing loop unrolling.)

Yes. We can overlap some of the vector memory instructions from different loops.

Problem M14.8: Vector Machines

Problem M14.8.A

The following **supplementary information** explains the diagram:

Scalar instructions execute in 5 cycles: fetch (**F**), decode (**D**), execute (**X**), memory (**M**), and writeback (**W**).
A vector instruction is also fetched (**F**) and decoded (**D**). Then, it stalls (—) until its required vector functional unit is available. With no chaining, a dependent vector instruction stalls until the previous instruction finishes writing back all of its elements. A vector instruction is pipelined across all the lanes in parallel. For each element, the operands are read (**R**) from the vector register file, the operation executes on the load/store unit (**M**) or the ALU (**X**), and the result is written back (**W**) to the vector register file.
A stalled vector instruction does not block a scalar instruction from executing.
LV_1 and LV_2 refer to the first and second LV instructions in the loop.

instr.	cycle																																										
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
LV_1	F	D	R	M1	M2	M3	M4	W																																		
LV_1				R	M1	M2	M3	M4	W																																	
LV_1					R	M1	M2	M3	M4	W																																
LV_1						R	M1	M2	M3	M4	W																															
LV_2		F	D	—	—	—	R	M1	M2	M3	M4	W																														
LV_2								R	M1	M2	M3	M4	W																												
LV_2									R	M1	M2	M3	M4	W																											
LV_2										R	M1	M2	M3	M4	W																										
ADDV			F	D	—	—	—	—	—	—	—	—	—	—	—	R	X1	W																							
ADDV																	R	X1	W																					
ADDV																		R	X1	W																				
ADDV																			R	X1	W																			
SUBVS				F	D	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	R	X1	W																	
SUBVS																							R	X1	W															
SUBVS																								R	X1	W														
SUBVS																									R	X1	W													
SV					F	D	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	R	M1	M2	M3	M4									
SV																													R	M1	M2	M3	M4							
SV																														R	M1	M2	M3	M4						
SV																															R	M1	M2	M3	M4					
ADDI						F	D	X	M	W																															
ADDI							F	D	X	M	W																													
ADDI								F	D	X	M	W																												
SUBI									F	D	X	M	W																											
BNEZ										F	D	X	M	W																										
LV_1											F	D	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	R	M1	M2	M3	M4	W				
LV_1																																R	M1	M2	M3	M4	W			
LV_1																																	R	M1	M2	M3	M4	W		
LV_1																																		R	M1	M2	M3	M4	W	

Problem M14.8.B

Vector processor configuration	Number of cycles between successive vector instructions					Total cycles per vector loop iter.
	LV_1, LV_2	LV_2, ADDV	ADDV, SUBVS	SUBVS, SV	SV, LV_1	
8 lanes, no chaining	4	9	**6**	**6**	**4**	**29**
8 lanes, chaining	**4**	**5**	**4**	**2**	**4**	**19**
16 lanes, chaining	**2**	**5**	**2**	**2**	**2**	**13**
32 lanes, chaining	**1**	**5**	**2**	**2**	**1**	**11**

Note, with 8 lanes and chaining, the SUBVS can not issue 2 cycles after the ADDV because there is only one ALU per lane. Also, since chaining is done through the register file, 2 cycles are required between the ADDV and SUBVS and between the SUBVS and SV even with 32 lanes (if bypassing was provided, only 1 cycle would be necessary).

Problem M14.8.C

Instr. Number	Instruction
I1	LV V1, R1
I2	LV V2, R2
I6	**ADDI R1, R1, 128**
I7	**ADDI R2, R2, 128**
I10	**LV V5, R1**
I11	**LV V6, R2**
I3	**ADDV V3, V1, V2**
I4	**SUBVS V4, V3, R4**
I5	**SV R3, V4**
I12	**ADDV V7, V5, V6**
I13	**SUBVS V8, V7, R4**
I8	**ADDI R3, R3, 128**
I14	**SV R3, V8**
I15	ADDI R1, R1, 128
I16	ADDI R2, R2, 128
I17	ADDI R3, R3, 128
I9	SUBI R5, R5, 32
I18	SUBI R5, R5, 32
I19	BNEZ R5, loop

This is only one possible solution. Scheduling the second iteration's LV's (I10 and I11) before the first iteration's SV (I5) allows the LV's to execute while the load/store unit would otherwise be idle. Interleaving instructions from the two iterations (for example, if I12 were placed between I3 and I4) could hide the functional unit latency seen with no chaining. However, doing so would delay the first SV (I5), and hence, increase the overall latency. This tension makes the optimal solution very tricky to find. Note that to preserve the instruction dependencies, I6 and I7 must execute before I10 and I11, and I8 must execute after I5 and before I14.

Problem M14.9: Vectorizing `memcpy` and `strcpy`

Problem M14.9.A

Because there is only one load/store unit, `SV` instruction should wait at least till the last element of the `LV` instruction is issued. Since there is only one lane, each `SV` and `LV` instruction takes 32 cycles to issue. In steady state, it takes 32 (`LV`) + 10 (dead time) + 32 (`SV`) + 10 (dead time) cycles per 32 elements, and 2.62 cycles per element. All scalar instructions can be overlapped with `SV`.

Problem M14.9.B

We can vectorize `strcpy` using `SEQSV` and `CLZM`. The algorithm is as follows. First, we load 32 elements. Second, we use `SEQSV` to check whether each element has `'\0'` or not. Third, we use `CLZM` to count the number of the elements before the first `'\0'` in the vector and set the vector length to that number. Then, we do a vector store. If no element has `'\0'` (i.e. the number is 32), we go back to the first step and load the next 32 elements. If a vector has `'\0'`, `strcpy` ends. As discussed in the function definition, our `strcpy` copies one word at a time, and assumes that the string is word-aligned with the terminating character of 32-bit `'\0'`.

```
    ADD     R5,R1,R0     ; store destination address in R5
    ADD     R4,R2,R0     ; store source address in R4
    ADDI    R6,R0,#32
    MTC1    VLR,R6       ; set vector length to 32
    CVM
    MOVI2FP F0,R0
loop:
    LV      V1,R4
    ADDI    R4,R4,#128   ; bump source pointer
    SEQSV   F0,V1        ; setup the mask register
    CLZM    R6,VM        ; number elements before '\0'
    MTC1    VLR,R6
    SV      R5,V1
    ADDI    R5,R5,#128   ; bump destination pointer
    SUBI    R7,R6,#32    ;
    BEQZ    R7,loop      ; if no element has '\0' goto loop
    SLLI    R6,R6,#2     ; move destination pointer to
    SUBI    R5,R5,#128   ; the end of the string
    ADD     R5,R5,R6     ; copy '\0'
```

Problem M14.9.C

Without vector chaining, `strcpy` takes more cycles per element than `memcpy` since it has one additional vector instruction, `SEQSV`. It takes 32+10 (`LV`) + 32 (`SEQSV`) + 1 (`CLZM`) + 1 (`MTC1`) + 32 (`SV`) + 10 (dead time) = 118 cycles per 32 elements or 3.69 cycles per element.

With vector chaining, the first element of V1 can be bypassed to `SEQSV` instruction after 10 cycles. Store can be executed only after we get the value of VLR, that is, after `SEQSV`, `CLZM`, and MTC1. Therefore, it takes 10 (`LV`) + 32 (`SEQSV`) + 1 (`CLZM`) + 1 (`MTC1`) + 32 (`SV`) + 10 (dead time) = 86 cycles per 32 elements or 2.69 cycles per element.

In `memcpy`, both vector instructions (`SV` and `LV`) use the same functional unit. Therefore, the execution of two instructions cannot be overlapped even with vector chaining. Copying each element takes 2.62 cycles as in M14.9.A. With vector chaining, the performance of `strcpy` is comparable to that of `memcpy`.

Problem M14.10: Performance of Vector Machines

Problem M14.10.A

With 8 lanes, a 2-cycle dead time and no vector chaining, we get the following pipeline diagram.

	Cycle																								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**
I1	F	D	R	X1	X2	W																	
I1				R	X1	X2	W																
I1					R	X1	X2	W															
I1						R	X1	X2	W														
I2		F	D	D	D	D			R	X1	X2	W											
I2										R	X1	X2	W										
I2											R	X1	X2	W									
I2												R	X1	X2	W								
I3			F	D	D	D	D	D	D	D	D	D	D	D	D	R	X1	X2	X3	W			
I3																	R	X1	X2	X3	W		
I3																		R	X1	X2	X3	W	
I3																			R	X1	X2	X3	W

Since each vector has 32 elements, and there are 8 lanes, the vector register file needs to be read 4 times for each instruction. Although I2 does not need the results of I1, both instructions use the vector add unit, so I2 must wait until after I1 completes its last read, plus an additional 2 cycles for dead time before beginning its first read. And because there is no chaining, I3, which is dependent on I2, needs to wait until I2 has finished its last write back before beginning its first read.

The execution time is 18 cycles (from cycle 6 to cycle 23, inclusive).

Problem M14.10.B

With 8 lanes, no dead time and flexible chaining, we get the following pipeline diagram.

	Cycle																
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
I1	F	D	R	X1	X2	W											
I1				R	X1	X2	W										
I1					R	X1	X2	W									
I1						R	X1	X2	W								
I2		F	D	D	D	D			X2	W							
I2								R	X1	X2	W						
I2									R	X1	X2	W					
I2										R	X1	X2	W				
I3			F	D	D	D	D	D	D	R	X1	X2	X3	W			
I3											R	X1	X2	X3	W		
I3												R	X1	X2	X3	W	
I3													R	X1	X2	X3	W

With no dead time, I2 can issue its first read after the last read of I1. And with flexible chaining, I3 can begin its first read in the same cycle as the first write of I2.

The execution time is 12 cycles (from cycle 6 to cycle 17, inclusive).

Problem M14.10.C

With 16 lanes, no dead time and flexible chaining, we get the following pipeline diagram.

	Cycle												
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
I1	F	D	R	X1	X2	W							
I1				R	X1	X2	W						
I2		F	D	D	R	X1							
I2						R	X1	X2	W				
I3			F	D	D	D	D	R	X1	X2	X3	W	
I3									R	X1	X2	X3	W

Since each vector has 32 elements, and there are 16 lanes, the vector register file needs to be read 2 times for each instruction.

The execution time is 8 cycles (from cycle 6 to cycle 13, inclusive).

Problem M14.11: Let's Talk About Loads (Spring 2014 Quiz 3, Part A)

Consider the following code sequence:

```
…
I1:  DIV R3, R1, 8
I2:  BNEZ R9, Somewhere
I3:  ST R2, 0(R3)
I4:  LD R1, 8(R4)
I5:  ADD R5, R1, 8
I6:  SUB R10, R6, R7
I7:  MUL R8, R9, R10
I8:  BEQZ R8, Somewhere else
…
```

We will explore how this program behaves on different architectural styles. In all cases, assume the following execution latencies:

- `ADD, SUB`: 2 cycles
- `BNEZ, BEQZ`: 2 cycles
- `LD`: 2 cycles if cache hit, 8 cycles if miss
- `MUL`: 5 cycles
- `DIV`: 10 cycles

Additionally, the `LD (I4)` in this sequence *misses* in the data cache and therefore has a long latency of 8 cycles.

Assume that the branch at `I2` is not taken and fetch and decode never stall (e.g., by missing on the instruction cache or the BTB). Also assume that there are no structural hazards.

Problem M14.11.A

Loads are often a bottleneck in processor performance, and as such compilers will try to move the loads as early as possible in the program to "hide" their latency. However, in the preceding code sequence, an optimizing compiler *cannot* move the load earlier in the program. Explain why in one or two sentences.

We need to explain why the LD can't be moved before the ST. (Otherwise, it *could* be moved earlier, even if not to the very beginning.) The reason is that there could be a RAW hazard through memory—maybe 0(R3)==8(R4).

Answers that there is a control hazard at I2 or a WAW hazard with I1 do not explain the difficulty of moving the LD earlier.

Problem M14.11.B

Show how this program would work on a single-issue in-order pipeline that tracks dependencies with a simple scoreboard. Instructions are issued (i.e., dispatched for execution) in order, but can complete out of order. Assume infinite functional units and full bypassing. Fill in the remainder of the table below.

Instruction	Issue Cycle	Completion Cycle
`I1: DIV R3, R1, 8`	1	11
`I2: BNEZ R9`	2	4
`I3: ST R2, 0(R3)`	11	n/a
`I4: LD R1, 8(R4)`	12	20
`I5: ADD R5, R1, 8`	20	22
`I6: SUB R10, R6, R7`	21	23
`I7: MUL R8, R9, R10`	23	28
`I8: BEQZ R8`	28	30

There is no hazard preventing issue of I6, so it can issue at 21. It can't issue earlier because the processor is in-order. Following I6 is a string of RAW dependencies, so the latency of I6, I7, and I8 determine the code sequence's completion time.

Problem M14.11.C

Assuming a single-issue out-of-order processor, show at which cycles instructions are issued (i.e., dispatched for execution) and complete. Assume that instructions are dispatched in program order if multiple are ready in the same cycle, and *do not speculate on data dependencies.* Again assume infinite functional units and full bypassing.

Instruction	Issue Cycle	Completion Cycle
`I1: DIV R3, R1, 8`	1	11
`I2: BNEZ R9`	2	4
`I3: ST R2, 0(R3)`	11	n/a
`I4: LD R1, 8(R4)`	12	20
`I5: ADD R5, R1, 8`	20	22
`I6: SUB R10, R6, R7`	3	5
`I7: MUL R8, R9, R10`	5	10
`I8: BEQZ R8`	10	12

Because we are not speculating on data dependencies, we cannot issue the LD before we know the ST address. So the earliest that the LD can issue is when I1 completes. Since the ST appears earlier in program order, it is issued first, and the LD is delayed until cycle 12. We can, however, begin issuing I6 at cycle 3 while waiting for I1 to complete.

In one or two sentences, what is the advantage of an out-of-order architecture vs. the in-order pipeline for this code sequence?

We are able to execute I6, I7, and I8 while the processor is waiting on memory, shortening the completion time.

Problem M14.11.D

Suppose the out-of-order processor chose to execute the load first, *before all other instructions in the code sequence*. What events could cause the load to be aborted, and what mechanisms are required to detect mis-speculation and roll back? Ignore exceptions in your answer.

Two events are relevant: the ST writes the address read by the LD, or the branch at I2 is mispredicted.

The former requires a speculative load buffer to detect RAW memory hazards. The latter requires detection of mis-speculation and redirecting fetch to the right address. Both require flushing the ROB for mis-speculated instructions.

Problem M14.11.E

Write VLIW code for this instruction sequence, assuming that the VLIW format is:

Memory operation	ALU operation	ALU operation / Branch

Try to make your VLIW code as efficient as possible, including re-ordering any instructions that do not have dependencies. For this VLIW code just use standard MIPS instructions to fill slots without predication or new, VLIW-specific instructions. (That is, simply schedule the instructions already provided.) Assume that the VLIW architecture has a scoreboard that stalls when a result is used before it is ready (e.g., on a cache miss).

	`DIV R3, R1, 8`	`BNEZ R9`
`ST R2, 0(R3)`	`SUB R10, R6, R7`	
`LD R1, 8(R4)`		
	`MUL R8, R9, R10`	
	`ADD R5, R1, 8`	`BEQZ R8`

This code schedule is effectively what the OOO processor does, with some independent operations scheduled in parallel. I6 is moved earlier in the program, and I7 & I8 execute while the LD is waiting. The one subtlety of this code is that the MUL is delayed one instruction so that the LD is not delayed. This is important because the critical path of this computation is DIV→ST→LD→ADD (issued).

In one or two sentences, what is the advantage/disadvantage of a VLIW architecture for this code sequence vs. the out-of-order pipeline?

For this code sequence, the VLIW code can achieve similar performance to an OOO processor with much simpler hardware logic. This is possible because it pushes the scheduling complexity into the compiler.

The disadvantage is similar—for VLIW to work well, the compiler must be able to schedule instructions effectively. Often this is not possible in practice.

Josh Fisher points out that if it has a scoreboard, it's not a *true* VLIW. How would the code sequence change if we didn't have a scoreboard?

We would need to schedule NOPs explicitly to handle the latency of each operation. This becomes complicated with variable latency operations, like LDs with a cache.

Problem M14.11.F

VLIW architectures rely heavily on the compiler to expose instruction-level parallelism in the program, so hiding load latency is a major challenge. VLIW compilers developed a technique called *trace scheduling* that merges multiple basic blocks into a single code sequence with software checks to ensure correctness. We profile our program and find that the first branch (`I2`) is almost never taken, so merging both basic blocks is a good idea.

If we use trace scheduling to move the load (`I4`) to be the *first* instruction, what conditions must software check to ensure correctness of the load for this code sequence? Ignore exceptions in your answer.

The answer is: "Same as OOO, except in software." We must check that there was no RAW hazard between ST→LD. We also must check R9 to make sure that the I2 branch was not taken.

Problem M14.11.G

To mitigate load latency, you decide to implement a prefetch instruction. `PREFETCH Imm(rs)` takes a single argument, an address, and *hints* to the processor that the given address may be used soon. Crucially, `PREFETCH` is side-effect free—the processor can choose to ignore `PREFETCH`'s without affecting program behavior.

Now consider the following simplified code sequence:

```
DIV R3, R1, 8
ST R2, 0(R3)
LD R1, 8(R4)
ADD R5, R1, 8
```

The diagram below shows how this code executes on an in-order issue processor with scoreboarding. Show how performance can be improved using `PREFETCH`.

Cycle	In-order	In-order w/ Prefetch
1	DIV	DIV
2		PREFETCH
3		
4		
5		
6		
7		
8		
9		
10		
11	ST	ST
12	LD	LD
13		
14		ADD
15		
16		Complete
17		
18		
19		
20	ADD	
21		
22	Complete	

Scheduling the PREFETCH before the DIV is correct but wastes a cycle unnecessarily.

Problem M14.11.H

In lecture we discussed an alternative instruction, "load-speculate":

```
LD.S rt, Imm(rs)
```

Load-speculate will fetch the value from memory but if the access faults it instead returns zero and does not cause an exception. Unlike prefetch, it gives not just the address but the source address *and* the destination register, which receives a value from memory. A load-speculate is followed in the program by a "load-check":

```
CHK.S rt, cleanup
```

Load-check checks if the register was written by a `LD.S` that should have caused an exception (e.g., due to a page fault). If it was, then `CHK.S` branches to somewhere else to service the exception and handle any necessary cleanup. `CHK.S` executes in 1 cycle.

Show how to use `LD.S/CHK.S` to speed up the code even further than was possible with `PREFETCH`. Assume scoreboarding and infinite functional units. Assume that in this case the compiler knows that the load (`I4`) can be scheduled before the store (`I3`) safely. Do not show cleanup code.

Cycle	In-order	In-order+LD.S+CHK.S
1	DIV	DIV
2		LD.S
3		
4		
5		
6		
7		
8		
9		
10		ADD
11	ST	ST
12	LD	CHK.S
13		Complete
14		
15		
16		
17		
18		
19		
20	ADD	
21		
22	Complete	

The benefit of LD.S is that it allows for speculative computation on data before the check occurs. This can lead to significant performance gains.