Problem M14.1: Microprogramming and Bus-Based Architectures

In this problem, we explore microprogramming by writing microcode for the bus-based implementation of the MIPS machine described in Handout (Bus-Based MIPS Implementation). Read the instruction fetch microcode in Table H14-3 which has been reproduced at the end of this problem (Worksheet M14.1-1) for the readers' convenience. Make sure that you understand how different types of data and control transfers are achieved by setting the appropriate control signals before attempting this problem.

In order to further simplify this problem, ignore the busy signal and assume that the memory is as fast as the register file.

The final solution should be elegant and efficient (e.g. number of new states needed, amount of new hardware added).

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

For this problem, you are to implement a new memory-memory add operation. The new instruction has the following format.

ADDm r_d, r_s, r_t

ADDm performs the following operation.

$$M[r_d] \leftarrow M[r_s] + M[r_t]$$

Fill in Worksheet M14.1-1 with the microcode for ADDm. Use *don't cares* (*) for fields where it is safe to use don't cares. Study the hardware description well, and make sure all your microinstructions are legal.

Please comment your code clearly. If the pseudo-code for a line does not fit in the space provided, or if you have additional comments, you may write in the margins as long as you do it neatly. Your code should exhibit "clean" behavior and not modify any registers (except r_d) in the course of executing the instruction.

Finally, make sure that the instruction fetches the next instruction (by doing a microbranch to FETCH0 as discussed above).

Problem M14.1.B Implementing DBNEZ Instruction

DBNEZ stands for Decrease Branch Not Equal Zero. This instruction uses the same encoding as conditional branch instructions on MIPS.

6	5	5	16
opcode	**rs**		**Offset**

DBNEZ decrements register **rs** by 1, writes the result back to **rs** and branches to (**PC**+4)+**offset**, if result in **rs** is not equal to 0. Offset is sign extended to allow for backward branches. This instruction can be used for efficiently implementing loops.

Your task is to fill out Worksheet M14.1-2 for DBNEZ instruction. You should try to optimize your implementation for minimum number of cycles necessary and for maximum number of don't-care signals. You do not have to worry about the busy signal.

(Note that the microcode for the fetch stage has changed slightly from the one in Problem M14.1.A, to allow for a more efficient implementation of some instructions.)

Problem M14.1.C Implementing RETZ Instruction

In this question we ask you to implement a special return instruction, *return on zero* (**`retz`**), which uses the same encoding as a conditional branch instruction on MIPS.

`retz Rs, Rt`

6	5	5	16
`Retz`	**Rs**	**Rt**	**Unused**

`retz` instruction provides fast return from a subroutine call using **Rt** as the stack pointer. The instruction first tests the value of register **Rs**. If it is **not** zero, simply proceed to the next instruction at **PC+4**. If it is zero, the instruction does the following: (1) it reads the return address from memory at the address in register **Rt,** (2) increments **Rt** by 4 and (3) jumps to the return address.

Fill out Worksheet M14.1-3 for the `retz` instruction. You should try to optimize your implementation for minimum number of cycles necessary and for maximum number of don't-care signals. You do not have to worry about the busy signal. You may not need all the lines in the table for your solution.

You are allowed to introduce *at most* one new μBr target (Next State) for J (Jump) or Z (branch-if-Zero) other than FETCH0.

Problem M14.1.D — Implementing CALL Instruction

In this question you will implement a new complex CALL instruction, which uses the same encoding as a conditional branch instruction on MIPS.

6	5	5	16
opcode	**ra**		**Offset**

CALL stores the return address, **PC**+4, to memory at the address in register **ra** (i.e., in **M[ra]**), decrements **ra** by 4, saves the new value back to **ra** and branches to (**PC**+4)+**offset**. This instruction provides fast subroutine calls, using register **ra** as the stack pointer.

Your task is to fill out Worksheet M14.1-4 for the CALL instruction. You should optimize your implementation to execute in the minimum number of cycles and to have the most signals set to don't care. You do not have to worry about the busy signal from memory. You may not need all the lines in the table for your solution.

Problem M14.1.E — Instruction Execution Times

How many cycles does it take to execute the following instructions in the microcoded MIPS machine? Use the states and control points from the MIPS microcontroller in Lecture 20 and assume Memory will not assert its busy signal.

Instruction	Cycles
`SUB  R3,R2,R1`	
`SUBI R2,R1,#4`	
`SW   R1,0(R2)`	
`BEQZ R1,label  # (R1 == 0)`	
`BNEZ R1,label  # (R1 != 0)`	
`J    label`	
`JR   R1`	
`JAL  label`	
`JALR R1`	

Which instruction takes the most cycles to execute? Which instruction takes the fewest cycles to execute?

Problem M14.1.F **Exponentiation**

Ben Bitdiddle needs to compute the power function for small numbers. Realizing there is no multiply instruction in the microcoded MIPS machine, he uses the following code to calculate the result when an unsigned number `m` is raised to the `n`th power, where `n` is another unsigned number.

```
if (m == 0) {
    result = 0;
}
else {
    result = 1;
    i = 0;

    while (i < n) {
        temp = result;
        j = 1;
        while (j < m) {
            result += temp;
            j++;
        }
        i++;
    }
}
```

The variables `i`, `j`, `m`, `n`, `temp` and `result` are unsigned 32-bit values.

Write the MIPS assembly that implements Ben's code. Use only the MIPS instructions that can be executed on the microcoded MIPS machine (ALU, ALUi, LW, SW, J, JAL, JR, JALR, BEQZ and BNEZ). The microcoded MIPS machine does not have branch delay slots. Use `R1` for `m`, `R2` for `n` and `R3` for `result`. At the end of your code only `R3` must have the correct value. The values of all other registers do not have to be preserved.

How many MIPS instructions are executed to calculate the power function? How many cycles does it take to calculate the power function? Again, use the states and control points from the MIPS microcontroller in Lecture 20 and assume Memory will not assert its busy signal.

m, n	Instructions	Cycles
0, 1		
1, 0		
2, 2		
3, 4		
M, N		

Problem M14.1.G **Microcontroller Jump Logic**

Now we will fill in a gap in the microcontroller implementation. In the lecture on microprogramming, we did not explain the implementation of the jump logic of the microcontroller. Your task in this problem is to implement that logic. Use AND gates, OR gates and inverters to implement the combinational logic that realizes the control equations for the jump logic of the MIPS microcontroller below. The control equations for the jump logic are

μPCSrc = *Case* μJumpTypes

next	**=>**	**μPC+1**
spin	**=>**	**μPC.busy + (μPC+1).~busy**
fetch	**=>**	**absolute**
dispatch	**=>**	**op-group**
feqz	**=>**	**absolute.zero + (μPC+1).~zero**
fnez	**=>**	**absolute.~zero + (μPC+1).zero**

The selection bits for each input of the μPCSrc mux, as well as the μJumpTypes encoding are given in the tables below. Your task is to create combinational logic that translates between them, according to the control equations. Assume that the busy and zero signals follow positive logic (so they are true if the wire is carrying a 1 and false if the wire is carrying a 0). Your design will be judged for its correctness, clarity and organization. These factors are more important than the efficiency of your design.

μJumpTypes	Encoding
next	000
spin	001
feqz	110
fnez	111
fetch	010
dispatch	100

Table M14.1-2: μJumpTypes Encoding

μPCSrc	Selection bits
μPC+1	00
μPC	01
absolute	10
op-group	11

Table M14.1-1: μPCSrc Selection bits

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	μB r	Next State
FETCH0:	MA <- PC; A <- PC	0	PC	0	1	1	*	*	0	1	*	0	*	0	N	*
	IR <- Mem	1	*	*	0	0	*	*	0	0	0	1	*	0	N	*
	PC <- A+4	0	PC	1	1	0	*	INC_A_4	1	*	*	0	*	0	D	*
. . .																
NOP0:	microbranch back to FETCH0	0	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH0
ADDM0:																

Worksheet M14.1-1

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	μB r	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:																

Worksheet M14.1-2

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`																

Worksheet M14.1-3

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	μB r	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:																

Worksheet M14.1-4

Problem M14.2: VLIW Programming

Ben Bitdiddle and Louis Reasoner have started a new company called Transbeta® and are designing a new processor named Titanium™. The Titanium processor is a single-issue in-order VLIW processor with:

- 2 load/store units. There is no cache and a load has a latency of 4 cycles but is fully pipelined.
- 1 integer ALU: single cycle
- 1 floating-point multiplier: 3 cycles, fully pipelined
- 1 floating-point adder: 2 cycles, fully pipelined
- 1 branch unit with no delay slots and 100% branch prediction accuracy
- 128 GPRs and 128 FPRs

A single Titanium instruction can issue to all the above units simultaneously. By definition, the operations in a Titanium instruction are independent. Every operation in a Titanium instruction reads the operands and issues simultaneously. Thus, if one operation is waiting for a result of a previous Titanium instruction, the entire Titanium instruction is stalled in the decode stage.

Everything is fully bypassed. Each functional unit has a dedicated writeback port, so there is never any contention. Writing to the same register multiple times in the same instruction is disallowed in the Titanium ISA. WAW hazards will also cause stalls. The Titanium ISA resembles MIPS, except that there can be up to 6 instructions on each line separated by semicolons.

You have been hired to work on some hand-optimized math libraries. The most important of these is the dot-product, given by $\Sigma(X_n \times Y_n)$.

Problem M14.2.A

Ben has translated dot-product from MIPS to the Titanium ISA

```
// R1 – pointer to X
// R2 – pointer to Y
// R5 - n
// R3 - temp
// F4 - temp
// F6 – result
      MOVI2FP F6,R0
loop:
      L.S   F3,0(R1); L.S  F4,0(R2); ADDI R5,R5,#-1
      MUL.S F3,F3,F4; ADDI R1,R1,#4
      ADD.S F6,F6,F3; ADDI R2,R2,#4; BNEZ R5,loop
```

Each iteration takes 9 cycles but the program averages 8 cycles per vector element. Alyssa P. Hacker says that it can be done in 1 cycle per vector element for long vectors. Show Ben and Louis what the code should be. Louis isn't too bright so make sure your code is well commented.

Problem M14.3: Trace Scheduling

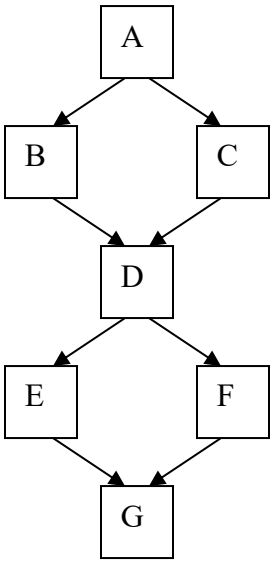
Trace scheduling is a compiler technique that increases ILP by removing control dependencies, allowing operations following branches to be moved up and speculatively executed in parallel with operations before the branch. It was originally developed for statically scheduled VLIW machines, but it is a general technique that can be used in different types of machines and in this question we apply it to a single-issue MIPS processor.

Consider the following piece of C code (% is modulus) with basic blocks labeled.

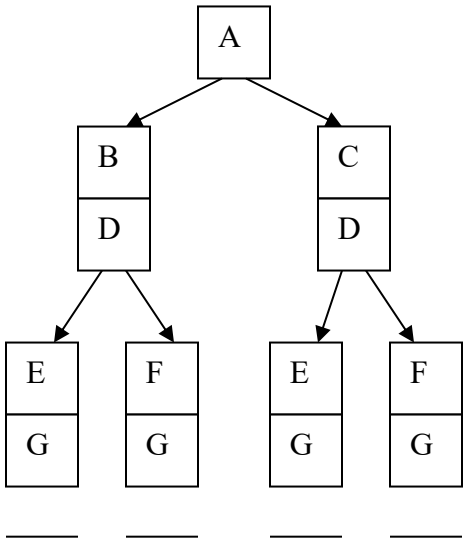
```
A:    if (data % 8 == 0)
B:      X = V0 / V1;
      else
C:      X = V2 / V3;
D:    if (data % 4 == 0)
E:      Y = V0 * V1;
      else
F:      Y = V2 * V3;
G:
```

Assume that **data** is a uniformly distributed integer random variable that is set sometime before executing this code.

Program's control flow graph



Decision tree



Path probabilities for 5.A: _____ _____ _____ _____

The control flow graph and the decision tree both show the possible flows of execution through basic blocks. However, the control flow graph captures the static structure of the program, while the decision tree captures the dynamic execution (history) of the program.

Problem M14.3.A

On the decision tree, label each path with the probability of traversing that path. For example, the leftmost block will be labeled with the total probability of executing the path ABDEG. (Hint: you might want to write out the cases). Circle the path that is most likely to be executed.

Problem M14.3.B

This is the MIPS code (no delay slots):

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

This code is to be executed on a single-issue processor **without** branch speculation. Assume that the memory, divider, and multiplier are all separate, long latency, **unpipelined** units that can run in parallel. Rewrite the above code using trace scheduling. Optimize only for the most common path. Just get the other paths to work. Don't spend your time performing any other optimizations. Ignore the possibility of exceptions. (Hint: Write the most common path first and then add fix-up code.)

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.) Approximately how many cycles does the new code take in the best case?

Problem M14.4: VLIW Machines

The program we will use for this problem is listed below. (In all questions, you should assume that arrays **A**, **B** and **C** do not overlap in memory.)

C code

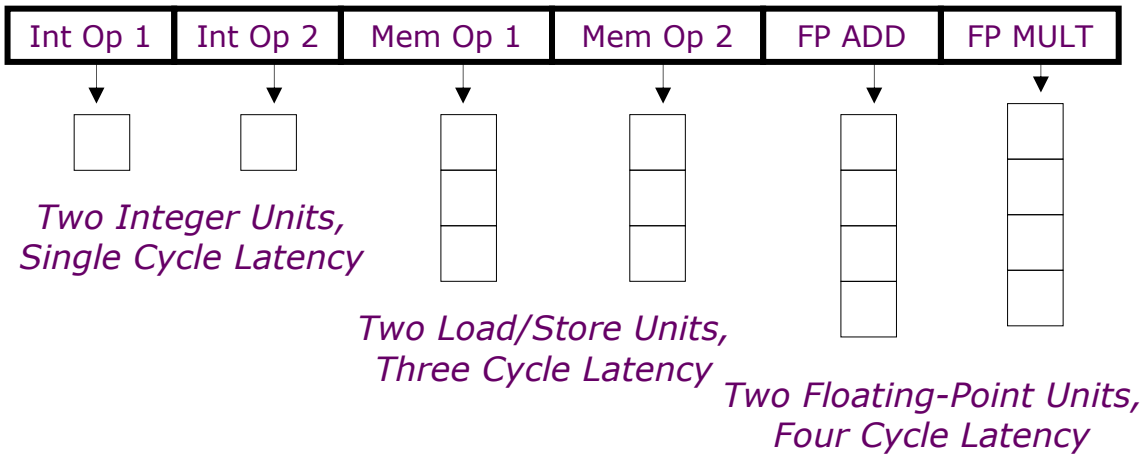
```
for (i=0; i<328; i++) {
    A[i] = A[i] * B[i];
    C[i] = C[i] + A[i];
}
```

In this problem, we will deal with the code sample on a VLIW machine. Our machine will have six execution units.

- two ALU units: latency one cycle, also used for branch operations
- two memory units: latency three cycles, fully pipelined, each unit can perform either a store or a load
- two FPU units: latency four cycles, fully pipelined, one unit can perform **fadd** operations, the other **fmul** operations.

Our machine has no interlocks. The result of an operation is written to the register file immediately after it has gone through the corresponding execution unit: one cycle after issue for ALU operations, three cycles for memory operations and four cycles for FPU operations. The old values can be read from the registers until they have been overwritten.

Below is a diagram of our VLIW machine.



The program for this problem translates to the following VLIW operations:

```
loop:   1.  ld f1, 0(r1)      ; f1 = A[i]
        2.  ld f2, 0(r2)      ; f2 = B[i]
        3.  fmul f4, f2, f1   ; f4 = f1 * f2
        4.  st f4, 0(r1)      ; A[i] = f4
        5.  ld f3, 0(r3)      ; f3 = C[i]
        6.  fadd f5, f4, f3   ; f5 = f4 + f3
        7.  st f5, 0(r3)      ; C[i] = f5
        8.  add r1, r1, 4     ; i++
        9.  add r2, r2, 4
       10.  add r3, r3, 4
       11.  add r4, r4, -1
       12.  bnez r4, loop     ; loop
```

Problem M14.4.A

Table M14.4-1, on the next page, shows our program rewritten for our VLIW machine, with some operations missing (instructions `2`, `6` and `7`). We have rearranged the instructions to execute as soon as they possibly can, but ensuring program correctness. Please fill in the missing operations. (Note, you may not need all the rows.)

Problem M14.4.B

How many cycles are required to complete one iteration of the loop in steady state? What is the performance (flops/cycle) of the program?

Problem M14.4.C

How many VLIW instructions would the smallest software pipelined loop require? Explain briefly. Ignore the prologue and the epilogue. Note: You do not need to write the software pipelined version. (You may consult **Table M14.4-1** for help.)

Problem M14.4.D

What would be the performance (flops/cycle) of the program? How many iterations of the loop would we have executing at the same time?

ALU1	ALU2	MU1	MU2	FADD	FMUL
`Add r1, r1, 4`	`add r2, r2, 4`	`ld f1, 0(r1)`			
`Add r3, r3, 4`	`add r4, r4, -1`	`ld f3, 0(r3)`			
					`fmul f4, f2, f1`
			`st f4, -4(r1)`		
	`bnez r4, loop`				

Table M14.4-1: VLIW Program

Problem M14.4.E

If we unrolled the loop once, would that give us better performance? How many VLIW instructions would we need for optimal performance? How many flops/cycle would we get? Explain.

Problem M14.4.F

What is the optimal performance in flops/cycle for this program on this architecture? Explain.

Problem M14.4.G

If our machine had a rotating register file, could we use fewer instructions than in *Problem M14.4.F* and still achieve optimal performance? Explain.

Problem M14.4.H

Imagine that memory latency has just increased to 100 cycles. How many instructions (approximately) an optimal loop would require? (There is no rotating register file, and ignore prologue/epilogue). Explain briefly.

5 50 100 200

Problem M14.4.I

Now our processor still has a memory latency of up to 100 cycles when it needs to retrieve data from main memory, but only 3 cycles if the data comes from the cache. Thus a memory operation can complete and write its result to a register anywhere between 3 and 100 cycles after being issued. Since our processor has no interlocks, other instructions will continue being issued. Thus, given two instructions, it is possible for the instruction issued second to complete and write back its result first. Circle how many instructions (approximately) are required for an optimal loop. Explain briefly.

5 50 100 200

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

Ben is working with an in-order VLIW processor, which issues two MIPS-like operations per instruction cycle. Assume a five-stage pipeline with two single-cycle ALUs, memory with one read and one write port, and a register file with four read ports and two write ports. Also assume that there are no branch delay slots, and loads and stores only take one cycle to complete. Turn Ben's loop into VLIW code. A[i's] and N are 32-bit signed integers. Initially, `R1` contains N and `R2` points to `A[0]`. You do not have to preserve the register values. Optimize your code to improve performance but do not use loop unrolling or software pipelining. What is the average number of cycles per element for this loop, assuming data elements are equally likely to be negative and non-negative?

Problem M14.5.B

Ben wants to remove the data-dependent branches in the assembly code by using predication. He proposes a new set of predicated instructions as follows.

1) Augment the ISA with a set of 32 predicate bits `P0-P31`.
2) Every standard non-control instruction now has a predicated counterpart, with the following syntax:

```
(pbit1) OPERATION1  ; (pbit2) OPERATION2
```

(Execute the first operation of the VLIW instruction if pbit1 is set and execute the second operation of the VLIW instruction if pbit2 is set.)

3) Include a set of compare operations that conditionally set a predicate bit.

```
CMPLTZ pbit,reg     ; set pbit if reg < 0
CMPGEZ pbit,reg     ; set pbit if reg >= 0
CMPEQZ pbit,reg     ; set pbit if reg == 0
CMPNEZ pbit,reg     ; set pbit if reg != 0
```

Eliminate all forward branches from Question M14.5.A with the new predicated operations. Try to optimize your code but do not use software pipelining or loop unrolling.

What is the average number of cycles per element for this new loop? Assume that the predicate-set compare instructions have a single cycle latency (i.e., they behave similarly to a regular ALU instruction including, full bypassing of the predicate bit).

Problem M14.5.C

Unroll the predicated VLIW code to perform two iterations of the original loop before each backward branch. You should use software pipelining to optimize the code for both performance and code density. What is the average number of cycles per element for a large value of N?

Problem M14.5.D

Now Ben wants to work with a vector processor with two lanes, each of which has a single-cycle ALU and a vector load-store unit. Write-back to the vector register file takes a single cycle. Assume for this part that each vector register has at least N elements.

Ben can also eliminate branches from his code by using vector masks. He wants to introduce a vector mask register as follows.

1) Augment the ISA with a vector mask register, `VM`.
2) Every vector instruction now executes each element operation only if the corresponding bit in the mask register is set.
3) Include compare operations that conditionally set the mask register.

`S--V`	`V1,V2`	Compare the elements (`EQ`,`NE`,`GT`,`LT`,`GE`,`LE`) in `V1` and `V2`. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put the resulting bit vector in a vector-mask register (`VM`). The instruction `S--SV` performs the same compare but using a scalar value as one operand.
`S--SV`	`F0,V1`	

Vectorize Ben's C loop, and replace all branches using vector masks. What is the average number of cycles per element for this loop in steady state for a very large value of N?

Problem M14.5.E

Modify the code from Part M14.5.D to handle the case when each vector register has *m* elements, where *m* may be less than N and is not necessarily a factor of N.

Problem M14.6: Predication and VLIW

Problem M14.6.A

Consider the following code.

```
            l.s   f1, 0(r1)      ; f1 = *r1
            seq.s r5, f10, f1    ;
            bneq  f1, f10, else  ; if f1==f10
            add.s f2, f1, f11    ;    f2 = f1 + f11
            b     if_end         ; else
else:       add.s f2, f1, f12    ;    f2 = f1 + f12
if_end:     s.s   f2, 0(r2)      ; *r2 = f2
```

Convert the code above to use predication rather than conditional branches. You should use the `CMPLTZ`, `CMPGEZ`, `CMPEQZ` or `CMPNEZ` instruction from Problem M5.8.B for predication. You may use negative predication for instructions, e.g.

```
(p1)  add r1, r2, r3    ; if (p1) r1 = r2 + r3
(!p1) add r1, r2, r3    ; if (!p1) r1 = r2 + r3
```

Problem M14.6.B

Our VLIW processor, called Adamantium, is very similar to the Titanium processor from Problem M14.2. Below are the details of our machine. Bold parts are different from Titanium.

- **1 load/store unit**: There is no cache and a **load has a latency of 2 cycles** and is fully pipelined.
- 1 integer ALU: Single cycle latency
- **no floating-point multiplier unit**
- 1 floating-point adder: 2 cycles, fully pipelined
- 1 branch unit with no delay slots and 100% branch prediction accuracy
- 128 GPRs, 128 FPRs and **128 predicate registers**

Consider the following simple loop written in predicated MIPS assembler.

```
loop:       l.s    f1, 0(r1)    ; f1 = *r1
            cmpnez p1, f1       ; p1 = (f1 != 0)
      (p1)  add.s  f2, f1, f1   ; if (p1) f2 = f1+f1
      (p1)  s.s    f2, 0(r1)    ; if (p1) *r1 = f2
            addi   r1, r1, #4   ; r1 += 4
            bneq   r1, r2, loop ; if (r1!=r2) goto loop
end:
```

On the next page, in Table M14.6-1, we have converted the code above into Adamantium code and unrolled it twice. Complete a software pipelined version of this loop for Adamantium below in Table M14.6-2. You should assume that the number of times the loop needs to execute is divisible by the unrolling factor, thus the loop doesn't need any fix-up code.

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:`				
				`bneq ,loop`
`epilog:`		`(p1) add.s`		
		`(p3) add.s`		
			`(p1) s.s`	
			`(p3) s.s`	

Table M14.6-2

Problem M14.7: Vector Machines

In this problem, we analyze the performance of vector machines. We start with a baseline vector processor with the following features.

- **32 elements per vector register**
- **8 lanes**
- **One ALU per lane: 1 cycle latency**
- **One MULT per lane: 2 cycle latency, fully pipelined**
- **One LOAD/STORE unit per lane: 4 cycle latency, fully pipelined**
- **No dead time**
- **No support for chaining**
- **Scalar instructions execute on a separate 5-stage fully-bypassed pipeline**

To simplify the analysis, we assume a **magic memory system** with no bank conflicts and no cache misses. Also, scalar operands of vector instructions are read in the Decode stage.

The program we will use for this problem is listed below. (In all questions, you should assume that arrays **A**, **B** and **C** do not overlap in memory.)

C code

```
for (i=0; i<328; i++) {
    A[i] = A[i] * B[i];
    C[i] = C[i] + A[i];
}
```

Problem M14.7.A

Consider the implementation of the C-code on the vector machine that executes it in the least 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
     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

Complete the pipeline diagram below with the loop code from Question M14.7.A on the baseline vector processor for one loop iteration. Do not fill in scalar instructions. Assume the scalar registers are available immediately, whenever needed. You may not require the entire length of the table.

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																									

Problem M14.7.C

In this question, we analyze the performance benefits of chaining.

Vector chaining is done through the register file. An element can be read (**R**) on the same cycle in which it is written back (**W**), or it can be read on any later cycle (chaining is *flexible*).

Complete the pipeline diagram below, with loop code from Question M14.7.A on a chained vector processor for one loop iteration. Do not fill in scalar instructions. Assume the scalar registers are available immediately, whenever needed. You may not require the entire length of the table.

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																									

Problem M14.7.D

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

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.)

Problem M14.8: Vector Machines

In this problem, we analyze the performance of vector machines. We start with a baseline vector processor with the following features.

- **32 elements per vector register**
- **8 lanes**
- **One ALU per lane: 1 cycle latency**
- **One load/store unit per lane: 4 cycle latency, fully pipelined**
- **No dead time**
- **No support for chaining**
- **Scalar instructions execute on a separate 5-stage pipeline**

To simplify the analysis, we assume a magic memory system with no bank conflicts and no cache misses.

We consider the execution of the following loop.

C code

```
for (i=0; i<320; i++) {
    C[i] = A[i] + B[i] – 1;
}
```

assembly code

```
# initial conditions:
#   R1 points to A[0]
#   R2 points to B[0]
#   R3 points to C[0]
#   R4 = 1
#   R5 = 320

loop:
  LV    V1, R1      # load A
  LV    V2, R2      # load B
  ADDV  V3, V1, V2  # add A+B
  SUBVS V4, V3, R4  # subtract 1
  SV    R3, V4      # store C
  ADDI  R1, R1, 128 # incr. A pointer
  ADDI  R2, R2, 128 # incr. B pointer
  ADDI  R3, R3, 128 # incr. C pointer
  SUBI  R5, R5, 32  # decr. count
  BNEZ  R5, loop    # loop until done
```

Problem M14.8.A

Complete the pipeline diagram of the baseline vector processor running the given code.

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	—																															
SUBVS																																					
SUBVS																																					
SUBVS																																					
SV					F	D	—																															
SV																																					
SV																																					
SV																																					
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	—																									
LV_1																																						
LV_1																																						
LV_1																																						

Problem M14.8.B

In this question, we analyze the performance benefits of chaining and additional lanes. Vector chaining is done through the register file and an element can be read (**R**) on the same cycle in which it is written back (**W**), or it can be read on any later cycle (chaining is *flexible*). For this question, we always assume 32 elements per vector register, so there are 2 elements per lane with 16 lanes, and 1 element per lane with 32 lanes.

To analyze performance, we calculate the total number of cycles per vector loop iteration by summing the number of cycles between the issuing of successive vector instructions. For example, in Question M14.8.A, LV_1 begins execution in cycle 3, LV_2 in cycle 7 and ADDV in cycle 16. Therefore, there are 4 cycles between LV_1 and LV_2, and 9 cycles between LV_2 and ADDV.

Complete the following table. The first row corresponds to the baseline 8-lane vector processor with no chaining. The second row adds flexible chaining to the baseline processor, and the last two rows increase the number of lanes to 16 and 32.
(Hint: You should consider each pair of vector instructions independently, and you can ignore the scalar instructions.)

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				
8 lanes, chaining						
16 lanes, chaining						
32 lanes, chaining						

Problem M14.8.C

Even with the baseline 8-lane vector processor with no chaining (used in Question M14.8.A), we can improve performance using software loop-unrolling and instruction scheduling. As a first step, we unroll two iterations of the loop and rename the vector registers in the second iteration.

```
loop:
I1:    LV    V1, R1       # load A
I2:    LV    V2, R2       # load B
I3:    ADDV  V3, V1, V2   # add A+B
I4:    SUBVS V4, V3, R4   # subtract 1
I5:    SV    R3, V4       # store C
I6:    ADDI  R1, R1, 128  # incr. A pointer
I7:    ADDI  R2, R2, 128  # incr. B pointer
I8:    ADDI  R3, R3, 128  # incr. C pointer
I9:    SUBI  R5, R5, 32   # decr. count
I10:   LV    V5, R1       # load A
I11:   LV    V6, R2       # load B
I12:   ADDV  V7, V5, V6   # add A+B
I13:   SUBVS V8, V7, R4   # subtract 1
I14:   SV    R3, V8       # store C
I15:   ADDI  R1, R1, 128  # incr. A pointer
I16:   ADDI  R2, R2, 128  # incr. B pointer
I17:   ADDI  R3, R3, 128  # incr. C pointer
I18:   SUBI  R5, R5, 32   # decr. count
I19:   BNEZ  R5, loop     # loop until done
```

Reorder the instructions in the unrolled loop to improve performance on the baseline vector processor (your solution does not need to be optimal).
Provide a valid ordering by listing the instructions below (a few have already been filled in for you). You may assume that the A, B and C arrays do not overlap.

Instr. Number	Instruction
I1	LV V1, R1
I2	LV V2, R2
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**

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

Ben Bitdiddle has bought a state-of-the-art vector machine, the Zirconium™, which has vector registers holding up to 32 elements, and has decided to vectorize his C library functions. As a starting point, he vectorizes the C function `memcpy`. The specification for `memcpy` is given as

```
/* copy n words from ct to s, and return s.       */
/* The actual C code copies one byte at a time.   */
/* Our version copies one word at a time.         */
void *memcpy(void *s, void *ct, size_t n)
```

Ben implements `memcpy` in the following fashion, assuming `s`, `ct`, and `n` are in registers `R1`, `R2`, and `R3` respectively. Assume that there are no delay slots.

```
    ADD    R5,R1,R0      ; store destination address in R5
    ADD    R4,R2,R0      ; store source address in R4
    ANDI   R6,R3,#31     ; N % 32
    MTC1   VLR,R6        ; put length in vector length register
loop:
    LV     V1,R4
    SV     R5,V1
    SUB    R3,R3,R6      ; subtract elements
    SLLI   R6,R6,#2
    ADD    R4,R4,R6      ; bump source pointer
    ADD    R5,R5,R6      ; bump destination pointer
    ADDI   R6,R0,#32
    MTC1   VLR,R6        ; reset to full length
    BNEZ   R3,loop       ; any more to do?
```

Problem M14.9.A

The Zirconium processor has one load/store unit with a single lane that is fully pipelined with a latency of 10 cycles and a dead time of 10 cycles. Instructions do not need to spend an extra cycle writing back values. All scalar instructions are executed on a separate 5-stage pipelined fully-bypassed datapath. Therefore, the execution of scalar instructions and vector instructions maybe overlapped. How many cycles are required to copy each element when a very long memory vector is copied, i.e., in steady state?

Problem M14.9.B

Ben's next target is `strcpy`, defined as follows:

```
/* copy string ct to string s, including '\0' and return s */
/* The actual C code copies one byte at time.               */
/* Our version copies one word at a time.                   */
void *strcpy(void *s, void *ct)
```

The difference between `strcpy` and `memcpy` is that `strcpy` terminates when it sees the string terminating character `'\0'` while `memcpy` copies a given length.

Ben makes several attempts to vectorize the code, but gives up deciding that it is not vectorizable. Alyssa, however, informs Ben that this function can be vectorized using some additional vector instructions listed below.

`CLZM`	`R1,VM`	Counts the number of leading 0s in the vector-mask register `VM` and puts the result in `R1`. For example, if the contents of `VM` are `0001010...000`, `clzm R1,VM` puts 3 into `R1`.
`S--V` `S--SV`	`V1,V2` `F0,V1`	Compare the elements (`EQ`,`NE`,`GT`,`LT`,`GE`,`LE`) in `V1` and `V2`. If the condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put the resulting bit vector in the vector-mask register (`VM`). The instruction `S--SV` performs the same compare but using a scalar value as one operand.

Given the additional instructions, help Ben write vectorized code for the Zirconium processor. Assume `s` and `ct` are in register `R1` and `R2`, respectively. The Zirconium processor does not have virtual memory and does not trap on memory protection violations on vector memory loads. Also, assume that a string must be word-aligned. The terminating character must start at a word boundary and the remaining 3 bytes after the terminating character must be 0x0. (Hint: The ASCII value of `'\0'` is 0.)

Problem M14.9.C

Compare the performance of vectorized `memcpy` and vectorized `strcpy` with and without vector chaining. Specifically, how many cycles are required to transfer one element in steady state? Assume that there is one vector compare unit with one lane and one cycle latency that compares whether two values are equal.

Problem M14.10: Performance of Vector Machines

The vector processor Germanium™ has a vector addition and a vector multiply unit with the following attributes.

1) Vector registers have 32 elements. The vector register file supports 2 read ports and 1 write port for each addition unit and multiplication unit.

2) The vector addition unit has a 2-cycle latency and is fully pipelined.

3) The vector multiplication unit has a 3-cycle latency and is fully pipelined.

You are now given the following code.

```
I1: ADDV  V3,V2,V1
I2: ADDV  V4,V2,V1
I3: MULTV V5,V4,V3
```

Note: All vectors are 32 elements in length.

Problem M14.10.A

Draw a pipeline diagram of the Germanium processor running the given code, assuming it has 8 lanes, a 2-cycle dead time, and no vector chaining. Instruction fetch takes one cycle, so does instruction decode (unless the instruction is stalled). Reading data from the register file also takes one cycle. Use F for fetch, D for Decode, R for Vector register read and W for write back.

How many cycles does the given code take to execute? Count execution time as the number of cycles from when the first result is written to when the last result is written (inclusive).

Pipeline diagram for `ADDV V3,V2,V1` and vector lengths of 24 elements, is shown below. Because we need to do 24 operations using 8 lanes, the vector register file should be read three times. X1 is the first stage of the addition unit and X2 is the second. In cycle 6, the results of the first 8 operations are written back. This instruction takes 3 cycles to execute.

Time ⟶

Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8
F	D	R	X1	X2	W		
			R	X1	X2	W	
				R	X1	X2	W

Problem M14.10.B

Draw a pipeline diagram of the Germanium processor running the given code, assuming it has 8 lanes, no dead time, and vector chaining. Vectoring chaining is done through the register file. A vector unit can read an element from the register file in the same cycle it is being written back. How many cycles does the given code take to execute?

Problem M14.10.C

Draw a pipeline diagram of the Germanium processor running the given code, assuming it has 16 lanes, no dead time, and vector chaining. How many cycles does the given code take to execute?

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.

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	
`I6: SUB R10, R6, R7`		
`I7: MUL R8, R9, R10`		
`I8: BEQZ R8`		

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`		
`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`		

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

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.

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).

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

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?

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.

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	
2		
3		
4		
5		
6		
7		
8		
9		
10		
11	ST	
12	LD	
13		
14		
15		
16		
17		
18		
19		
20	ADD	
21		
22	Complete	

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	
2		
3		
4		
5		
6		
7		
8		
9		
10		
11	ST	
12	LD	
13		
14		
15		
16		
17		
18		
19		
20	ADD	
21		
22	Complete	