Problem M3.11: VLIW Programming [?? Hours]

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 M3.11.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 M3.12: Trace Scheduling

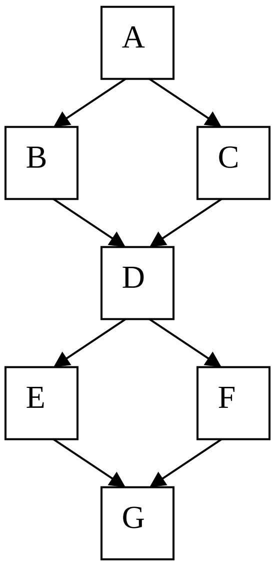
Trace scheduling is a compiler technique that increases ILP by removing control dependencies, allowing operations following branches to be moved up and speculatively executed in parallel with operations before the branch. It was originally developed for statically scheduled VLIW machines, but it is a 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:
```

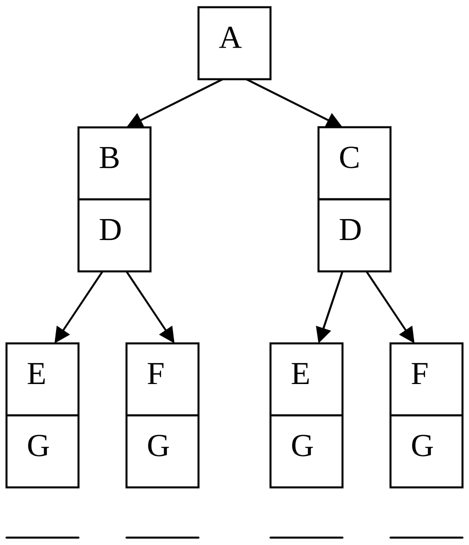
Assume that **data** is a uniformly distributed integer random variable that is set sometime before executing this code.

Program's control flow graph



$$\begin{tikzcd} & A \arrow[dl] \arrow[dr] & \\ B \arrow[dr] & & C \arrow[dl] \\ & D \arrow[dl] \arrow[dr] & \\ E \arrow[dr] & & F \arrow[dl] \\ & G & \end{tikzcd}$$

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 M3.12.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 M3.12.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 M3.12.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 M3.13: VLIW Machines [?? Hours]

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

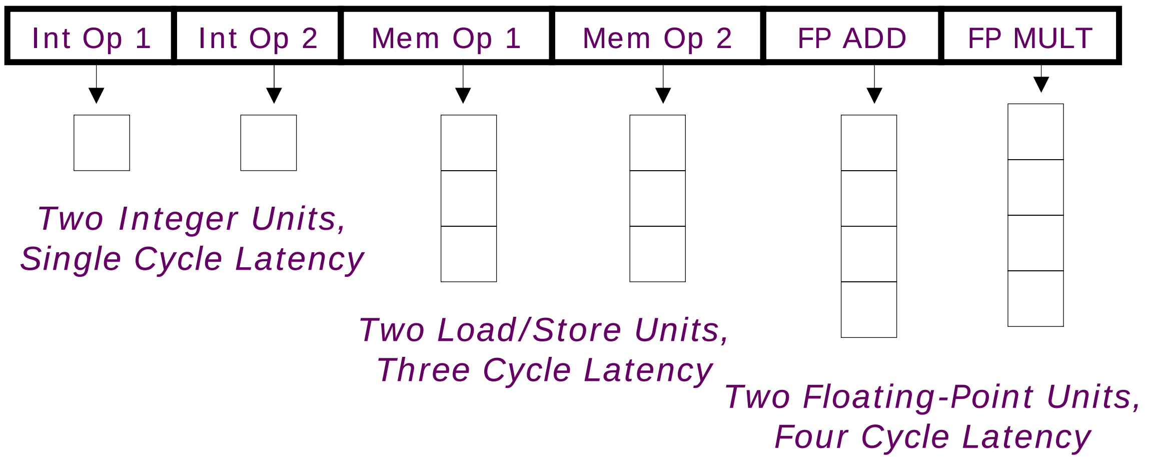
```
for (i=0; i<328; i++) {
    A[i] = A[i] * B[i];
    C[i] = C[i] + A[i];
}
```

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.


Int Op 1
Int Op 2
Mem Op 1
Mem Op 2
FP ADD
FP MULT
Two Integer Units, Single Cycle Latency
Two Load/Store Units, Three Cycle Latency
Two Floating-Point Units, Four Cycle Latency

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 M3.13.A

Table M3.13-1, 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 M3.13.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 M3.13.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 M3.13-1** for help.)

Problem M3.13.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 M3.13-1: VLIW Program

Problem M3.13.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 M3.13.F

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

Problem M3.13.G

If our machine had a rotating register file, could we use fewer instructions than in *Problem M3.13.F* and still achieve optimal performance? Explain.

Problem M3.13.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 M3.13.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 M3.14: VLIW & Vector Coding [?? Hours]

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 M3.14.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 M3.14.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 M314.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 M3.14.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 M3.14.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 M3.14.E

Modify the code from Part M3.14.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 M3.15: Predication and VLIW [?? Hours]

Problem M3.15.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 M3.14.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 M3.15.B

Our VLIW processor, called Adamantium, is very similar to the Titanium processor from Problem M3.11. 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 M3.15-1, 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 M3.15-2. 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 M3.15-1

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 M3.15-2