CS152
Computer Architecture and Engineering
VLIW, Vector, and
Multithreaded Machines

Assigned March 30 — Problem Set #4 — *Due April 8*

http://inst.eecs.berkeley.edu/~cs152/sp10

The problem sets are intended to help you learn the material, and we encourage you to collaborate with other students and to ask questions in discussion sections and office hours to understand the problems. However, each student must turn in their own solutions to the problems.

The problem sets also provide essential background material for the quizzes. The problem sets will be graded primarily on an effort basis, but if you do not work through the problem sets you are unlikely to succeed at the quizzes! We will distribute solutions to the problem sets on the day the problem sets are due to give you feedback. Homework assignments are due at the beginning of class on the due date. Homework will not be accepted once solutions are handed out.

Problem P4.1: Trace Scheduling

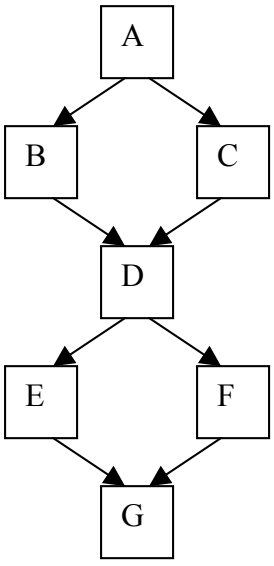
Trace scheduling is a compiler technique that increases ILP by removing control dependencies, allowing operations following branches to be moved up and speculatively executed in parallel with operations before the branch. It was originally developed for statically scheduled VLIW machines, but it is a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

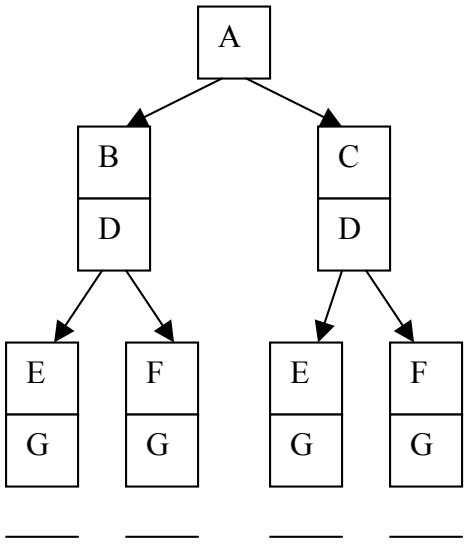
```
A     if (data % 8 == 0)
B       X = V0 / V1;
      else
C       X = V2 / V3;
D     if (data % 4 == 0)
E       Y = V0 * V1;
      else
F       Y = V2 * V3;
G
```

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

The program's control flow graph is



The decision tree is



Path probabilities for 5.A: _____ _____ _____ _____

A control flow graph and the decision tree both show the possible flow 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 P4.1.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 P4.1.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 be 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 then add fix-up code.)

Problem P4.1.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 P4.2: 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

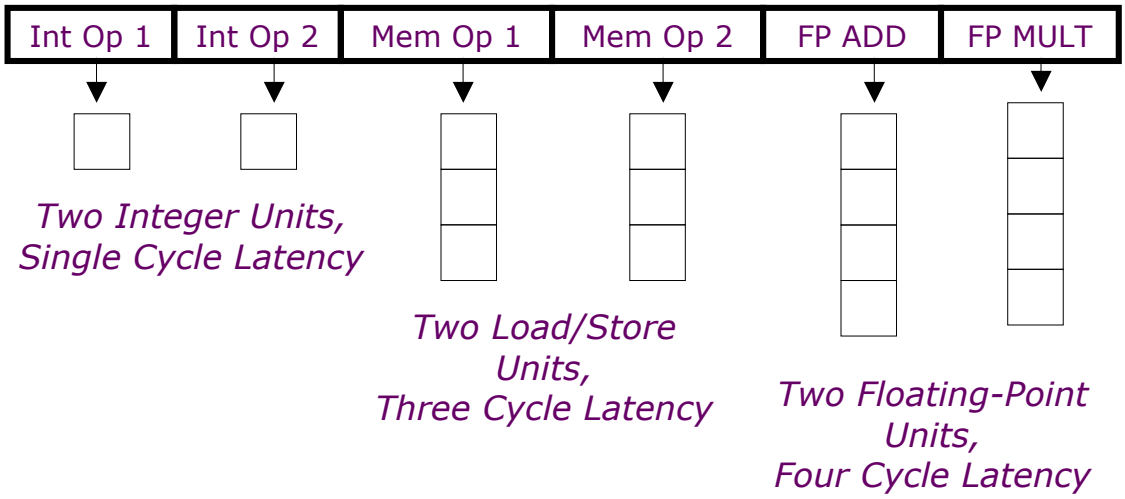
```
for (i=0; i<328; i++) {
    A[i] = A[i] * B[i];
    C[i] = C[i] + A[i];
}
```

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 P4.2.A

Table P4.2-1, 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 missing operations. (Note, you may not need all the rows)

Problem P4.2.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 P4.2.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 P4.2-1** for help)

Problem P4.2.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 P4.2-1: VLIW Program

Problem P4.2.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 P4.2.F

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

Problem P4.2.G

If our machine had a rotating register file, could we use fewer instructions than in *Question P4.2.F* and still achieve optimal performance? Explain.

Problem P4.2.H

Imagine that memory latency has just increased to 100 cycles. Circle how many instructions (approximately) an optimal loop would require. (no rotating register file, ignoring prologue/epilogue). Explain briefly.

5 50 100 200

Problem P4.2.I

Now our processor still has 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 P4.3: 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 P4.3.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 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 P4.3.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 P4.3.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-setting compares have single cycle latency (i.e., behave similarly to a regular ALU instruction including full bypassing of the predicate bit).

Problem P4.3.C

Unroll the predicated VLIW code to perform two iterations of the original loop before each backwards 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 large N?

Problem P4.4: 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 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 P4.4.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 P4.4.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 (the 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 P4.4.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 P4.4.C

Even with the baseline 8-lane vector processor with no chaining (used in Question P4.4.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 instruction numbers below (a few have already been filled in for you). Filling in the "Instruction" field is optional. 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 P4.5: Multithreading

This problem evaluates the effectiveness of multithreading using a simple database benchmark. The benchmark searches for an entry in a linked list built from the following structure, which contains a key, a pointer to the next node in the linked list, and a pointer to the data entry.

```
struct node {
     int key;
     struct node *next;
     struct data *ptr;
}
```

The following MIPS code shows the core of the benchmark, which traverses the linked list and finds an entry with a particular key. Assume MIPS has no delay slots.

```
      ;
      ; R1: a pointer to the linked list
      ; R2: the key to find
      ;
loop: LW       R3, 0(R1)  ; load a key
      LW       R4, 4(R1)  ; load the next pointer
      SEQ      R3, R3, R2 ; set R3 if R3 == R2
      BNEZ     R3, End    ; found the entry
      ADD      R1, R0, R4
      BNEZ     R1, Loop   ; check the next node
End:
      ; R1 contains a pointer to the matching entry or zero
      if    ; not found
```

We run this benchmark on a single-issue in-order processor. The processor can fetch and issue (dispatch) one instruction per cycle. If an instruction cannot be issued due to a data dependency, the processor stalls. Integer instructions take one cycle to execute and the result can be used in the next cycle. For example, if `SEQ` is executed in cycle 1, `BNEZ` can be executed in cycle 2. We also assume that the processor has a perfect branch predictor with no penalty for both taken and not-taken branches.

Problem P4.5.A

Assume that our system does not have a cache. Each memory operation directly accesses main memory and takes 100 CPU cycles. The load/store unit is fully pipelined, and non-blocking. After the processor issues a memory operation, it can continue executing instructions until it reaches an instruction that is dependent on an outstanding memory operation. How many cycles does it take to execute one iteration of the loop in steady state?

Problem P4.5.B

Now we add zero-overhead multithreading to our pipeline. A processor executes multiple threads, each of which performs an independent search. Hardware mechanisms schedule a thread to execute each cycle.

In our first implementation, the processor switches to a different thread every cycle using fixed round robin scheduling (similar to CDC 6600 PPUs). Each of the N threads executes one instruction every N cycles. What is the **minimum** number of threads that we need to fully utilize the processor, i.e., execute one instruction per cycle?

Problem P4.5.C

How does multithreading affect throughput (number of keys the processor can find within a given time) and latency (time processor takes to find an entry with a specific key)? Assume the processor switches to a different thread every cycle and is fully utilized. Check the correct boxes.

	Throughput	**Latency**
Better		
Same		
Worse		

Problem P4.5.D

We change the processor to only switch to a different thread when an instruction cannot execute due to data dependency. What is the minimum number of threads to fully utilize the processor now? Note that the processor issues instructions in-order in each thread.

Problem P4.6: Multithreading

Consider a single-issue in-order multithreading processor that is similar to the one described in Problem P4.5.

Each cycle, the processor can fetch and issue one instruction that performs any of the following operations:

- **load/store, 12-cycle latency (fully pipelined)**
- **integer add, 1-cycle latency**
- **floating-point add, 5-cycle latency (fully pipelined)**
- **branch, no delay slots, 1-cycle latency**

The processor **does not have a cache**. Each memory operation directly accesses main memory. If an instruction cannot be issued due to a data dependency, the processor stalls. We also assume that the processor has a perfect branch predictor with no penalty for both taken and not-taken branches.

You job is to analyze the processor utilizations for the following two thread-switching implementations:

Fixed Switching: the processor switches to a different thread every cycle using fixed round robin scheduling. Each of the N threads executes an instruction every N cycles.

Data-dependent Switching: the processor only switches to a different thread when an instruction cannot execute due to a data dependency.

Each thread executes the following MIPS code:

```
loop:   L.D     F2, 0(R1)       ; load data into F2
        ADDI    R1, R1, 4       ; bump source pointer
        FADD    F3, F3, F2      ; F3 = F3 + F2
        BNE     F2, F4, loop    ; continue if F2 != F4
```

Problem P4.6.A

What is the minimum number of threads that we need to fully utilize the processor for each implementation?

Fixed Switching: _____ Thread(s)

Data-dependent Switching: _____ Thread(s)

Problem P4.6.B

What is the minimum number of threads that we need to fully utilize the processor for each implementation if we change the **load/store latency to 1-cycle (but keep the 5-cycle floating-point add)**?

Fixed Switching: _____ Thread(s)

Data-dependent Switching: _____ Thread(s)

Problem P4.6.C

Consider a **Simultaneous Multithreading (SMT)** machine with limited hardware resources. **Circle** the following hardware constraints that can limit the total number of threads that the machine can support. For the item(s) that you circle, **briefly describe** the minimum requirement to support **N** threads.

(A) Number of Functional Unit:

(B) Number of Physical Registers:

(C) Data Cache Size:

(D) Data Cache Associatively: