

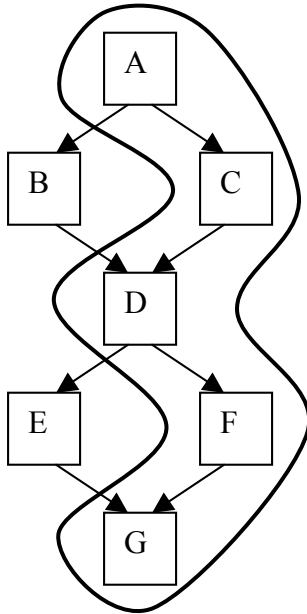
# Problem Set 5 Solutions

## Problem P5.1: Trace Scheduling

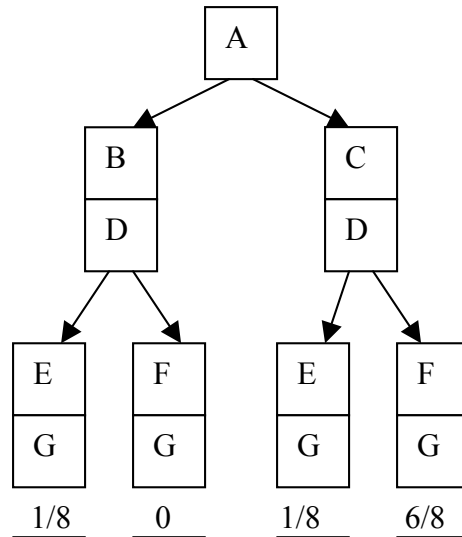
### Problem P5.1.A

---

The program's control flow graph is



The decision tree is



### Problem P5.1.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 P5.1.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)

$$x + \max(y, z)$$

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

## Problem P5.2: VLIW machines

### Problem P5.2.A

---

See the next page (Table P5.2-1).

### Problem P5.2.B

---

12 cycles,  $2/12=0.17$  flops per cycle

### Problem P5.2.C

---

3 instructions, because there are 5 memory ops and 5 ALU ops, and we can only issue 2 of those per instruction. The answer of is not 4 (longest latency operation FMUL), because the memory unit will become the bottleneck first.

Here is the resulting code:

add r1, r1, 4	add r2, r2, 4	ld f1, 0(r1)	ld f2, 0(r2)		fmul f4, f2, f1
add r3, r3, 4	add r4, r4, -1	ld f3, -4(r3)	st f4, -8(r1)	fadd f5, f4, f3	
	bnez r4, loop		st f5, -12(r3)		

for a particular  $i$ , white background corresponds to first iteration of the loop, grey background to the second iteration, yellow background to third, and blue to fourth. Note, one does not need to write the code to get an answer, because it's just a question of how many instructions are needed to express all the operations.

### Problem P5.2.D

---

$(2 \text{ flops}) / (3 \text{ cycles per iteration}) = 0.67 \text{ flops per cycle}$ , 4 iterations at a time (total iteration length) / (min cycles per iteration) =  $12/3=4$

ALU1	ALU2	MU1	MU2	FADD	FMUL
<code>add r1, r1, 4</code>	<code>add r2, r2, 4</code>	<code>ld f1, 0(r1)</code>	<code>ld f2, 0(r2)</code>		
<code>add r3, r3, 4</code>	<code>add r4, r4, -1</code>	<code>ld f3, 0(r3)</code>			
					<code>fmul f4, f2, f1</code>
			<code>st f4, -4(r1)</code>	<code>fadd f5, f4, f3</code>	
	<code>bnez r4, loop</code>	<code>st f5, -4(r3)</code>			

**Table P5.2-1: VLIW Program**

### **Problem P5.2.E**

---

If we unrolled once, while still software pipelining, we need 5 instructions to execute two iterations – we get  $4/5=0.8$  flops/cycle.

### **Problem P5.2.F**

---

Same as above: 0.8 flops/cycle. We are fully utilizing the memory units (they are the bottleneck), so we can't execute more loops/cycle.

### **Problem P5.2.G**

---

No. We need to unroll the loop once to have an even number of memory ops. Using rotating registers would not allow us to squeeze more memory ops per iteration, so we'd still need 5 instructions per iteration of the program's flow. The code size in memory might be made smaller.

### **Problem P5.2.H**

---

This is actually rather tricky. The correct answer is 5, because increasing the latency of one unit doesn't hurt its throughput, so it can still sustain the same throughput as before. The only complication is if we have enough registers (in this case we do). Without interlocks, we can use the registers just as values come in for them, using the execution units to "store" the loops (inside pipeline registers).

### **Problem P5.2.I**

---

With many registers (an unreasonable amount), this still could be 5, and we could use code similar to P5.2.H. Because of the uncertainty, we can't use the same pipelining trick as P5.2.H, and without an unreasonable amount of registers we are forced to be conservative. There are approximately 100 instructions required, because maximal latency will be 100. (Could also argue for 50)

## Problem P5.3: 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 P5.3.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]                             |                          ; skip if (A[i] ≥ 0)
      BEQZ R5, next                  |                          ; A[i] = -A[i]
      SUB R4, R0, R4                 |                          ; store updated value of A[i]
      SW R4, -4(R2)                  |                          ; continue if N > 0
next:  BNEZ R1, loop
end:
```

Average Number of Cycles:  $\frac{1}{2} \times (6 + 4) = 5$

; 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]                             |                          ; skip if (A[i] ≥ 0) | A[i] = -
A[i]                                  |                          ; store updated value of A[i]
      BNEZ R5, next                  |                          ; continue if N > 0
      SW R4, -4(R2)
next:  BNEZ R1, loop
end:
```

Average Number of Cycles:  $\frac{1}{2} \times (5 + 4) = 4.5$

*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 P5.3.B

---

```
SGT R3, R1, R0
BNEZ R3, end
loop: LW R4, 0(R2)          | SUBI R1, R1, #1 ; R3 = (N > 0) | if N ≤ 0
    CMPLTZ P0, R4          | ADDI R2, R2, #4 ; R4 = A[i] | N--
    &A[i+1]                |                   ; P0 = (A[i]<0) | R2 =
    (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:  $\frac{1}{2} \times (4 + 4) = 4$  Cycles

### Problem P5.3.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
BNEZ R3, end
BEQZ R5, loop
CMPLTZ P0, R4
ADDI R2, R2, #4
(P0) SW R4, -4(R2)
loop: LW R4, 0(R2)          | SUBI R1, R1, #1
    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
    ADDI R2, R2, #8       | (P1) SW R6 4(R2)
end:                       | BNEZ R1, loop
```

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

## Problem P5.4: Vector Machines

### Problem P5.4.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<sub>1</sub> and LV<sub>2</sub> 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 <sub>1</sub>	F	D	R	M1	M2	M3	M4	W																																														
LV <sub>1</sub>				R	M1	M2	M3	M4	W																																													
LV <sub>1</sub>					R	M1	M2	M3	M4	W																																												
LV <sub>1</sub>						R	M1	M2	M3	M4	W																																											
LV <sub>2</sub>	F	D	—	—	—	R	M1	M2	M3	M4	W																																											
LV <sub>2</sub>							R	M1	M2	M3	M4	W																																										
LV <sub>2</sub>								R	M1	M2	M3	M4	W																																									
LV <sub>2</sub>									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 <sub>1</sub>										F	D	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	R	M1	M2	M3	M4	W				
LV <sub>1</sub>																																																R	M1	M2	M3	M4	W	
LV <sub>1</sub>																																																	R	M1	M2	M3	M4	W
LV <sub>1</sub>																																																	R	M1	M2	M3	M4	W

Note, although it should be the default assumption, the problem did not explicitly state that vector instructions targeting the same unit must execute in-order. Therefore, solutions which executed the final LV before the SV are accepted.

**Problem P5.4.B**

---

vector processor configuration	number of cycles between successive vector instructions					total cycles per vector loop iter.
	LV <sub>1</sub> , LV <sub>2</sub>	LV <sub>2</sub> , ADDV	ADDV, SUBVS	SUBVS, SV	SV, LV <sub>1</sub>	
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 P5.4.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 them 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 thereby 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 P5.5: Multithreading

### Problem P5.5.A

---

Since there is no penalty for conditional branches, instructions take one cycle to execute unless there is a dependency problem. The following table summarizes the execution time for each instruction. From the table, the loop takes **104 cycles** to execute.

	Instruction	Start Cycle	End Cycle
LW	R3, 0 (R1)	1	100
LW	R4, 4 (R1)	2	101
SEQ	R3, R3, R2	101	101
BNEZ	R3, End	102	102
ADD	R1, R0, R4	103	103
BNEZ	R1, Loop	104	104

### Problem P5.5.B

---

If we have  $N$  threads and the first load executes at cycle 1, SEQ, which depends on the load, executes at cycle  $2 \cdot N + 1$ . To fully utilize the processor, we need to hide 100-cycle memory latency,  $2 \cdot N + 1 \geq 101$ . The minimum number of thread needed is **50**.

### Problem P5.5.C

---

	Throughput	Latency
<b>Better</b>	✓	
<b>Same</b>		
<b>Worse</b>		✓

### Problem P5.5.D

---

In steady state, each thread can execute 6 instructions (SEQ, BNEZ, ADD, BNEZ, LW, LW). Therefore, to hide 98 (104-6) cycles between the second LW and SEQ, a processor needs  $\lceil 98/6 \rceil + 1 = \mathbf{18}$  threads.

## Problem P5.6: Multithreading

### Problem P5.6.A

---

Fixed Switching:         6         Thread(s)

If we have  $N$  threads and L.D. executes at cycle 1, FADD, which depends on the load executes at cycle  $2N + 1$ . To fully utilize the processor, we need to hide 12-cycle memory latency,  $2N + 1 \geq 13$ . The minimum number of thread needed is 6.

Data-dependent Switching:         4         Thread(s)

In steady state, each thread can execute 4 instructions (FADD, BNE, LD, ADDI). Therefore, to hide 11 cycles between ADDI and FADD, a processor needs  $\lceil 11/4 \rceil + 1 = 4$  threads.

### Problem P5.6.B

---

Fixed Switching:         2         Thread(s)

Each FADD depends on the previous iteration's FADD. If we have  $N$  threads and the first FADD executes at cycle 1, the second FADD executes at cycle  $4N + 1$ . To fully utilize the processor, we need to hide 5-cycle latency,  $4N + 1 \geq 6$ . The minimum number of thread needed is 2.

Data-dependent Switching:         2         Thread(s)

In steady state, each thread can execute 4 instructions (FADD, BNE, LD, ADDI). Therefore, to hide 1 cycle between ADDI and FADD, a processor needs  $\lceil 1/4 \rceil + 1 = 2$  threads.

### Problem P5.6.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: Since not all the threads are executed each cycle, the number of functional unit is not a constraint that limits the total number of threads that the machine can support.

(B) Number of Physical Registers: We need at least  $[N \times (\text{number of architecture registers})]$  physical registers for an in-order system. Since it is SMT, it is actually least  $[N \times (\text{number of architecture registers}) + 1]$  physical registers, because you can't free a physical register until the next instruction commits to that same architectural register.

(C) Data Cache Size: This is for performance reasons.

(D) Data Cache Associativity: This is for performance reasons.