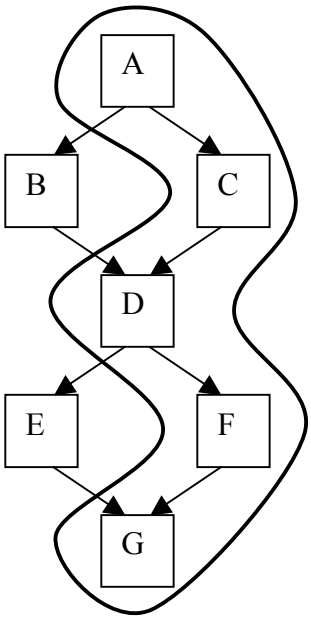
Problem Set 4 Solutions

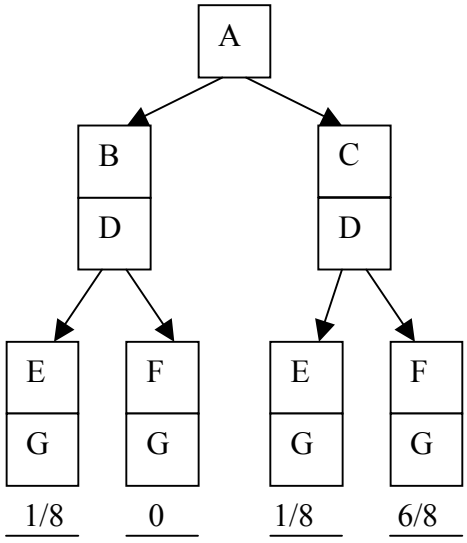
Problem P4.1: Trace Scheduling

Problem P4.1.A

The program's control flow graph is



The decision tree is



Problem P4.1.B

```
ACF:    ld   x1, data
   div  x3, x6, x7 ;; X <- V2/V3
   mul  x8, x6, x7 ;; Y <- V2*V3
D: andi x2, x1, 3  ;; x2 <- x1%4
   bnez x2, G
A: andi x2, x1, 7  ;; x2 <- x1%8
   bnez x2, E
B: div  x3, x4, x5 ;; X <- V0/V1
E: mul  x8, x4, x5 ;; Y <- V0*V1
G:
```

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)

x+max(y,z)

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

Problem P4.2: VLIW machines

Problem P4.2.A

See the next page (**Table P4.2-1**).

Problem P4.2.B

12 cycles, 2/12=0.17 flops per cycle

Problem P4.2.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 x1, x1, 4	add x2, x2, 4	ld f1, 0(x1)	ld f2, 0(x2)		fmul f4, f2, f1
add x3, x3, 4	add x4, x4, -1	ld f3, -4(x3)	sw f4, -8(x1)	fadd f5, f4, f3	
	bnez x4, loop		sw f5, -12(x3)		

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 P4.2.D

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

ALU1	ALU2	MU1	MU2	FADD	FMUL
`add x1, x1, 4`	`add x2, x2, 4`	`ld f1, 0(x1)`	**`ld f2, 0(x2)`**		
`add x3, x3, 4`	`add x4, x4, -1`	`ld f3, 0(x3)`			
					`fmul f4, f2, f1`
			`sw f4, -4(x1)`	**`fadd f5, f4, f3`**	
	`bnez x4, loop`	**`sw f5, -4(x3)`**			

Table P4.2-1: VLIW Program

Problem P4.2.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 P4.2.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 P4.2.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 P4.2.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 P4.2.I

With many registers (an unreasonable amount), this still could be 5, and we could use code similar to P4.2.H. Because of the uncertainty, we can't use the same pipelining trick as P4.2.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 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

```
; Initial Conditions:
;     x1 = N
;     x2 = &A[0]


      SGT x3, x1, x0
      BEQZ x3, end                                  ; x3 = (N > 0) | special case
N ≤ 0
loop: LW x4, 0(x2)       | SUBI x1, x1, #1        ; x4 = A[i] | N--
      SLT x5, x4, x0     | ADDI x2, x2, #4        ; x5 = (A[i] < 0) | x2 =
&A[i+1]
      BEQZ x5, next      |                        ; skip if (A[i]≥0)
      SUB x4, x0, x4     |                        ; A[i] = -A[i]
      SW x4, -4(x2)      |                        ; store updated value of A[i]
next: BNEZ x1, loop      |                        ; continue if N > 0
end:
```

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

```
; SOLUTION #2

      SGT x3, x1, x0
      BNEZ x3, end                                  ; x3 = (N > 0) | special case
N ≤ 0
loop: LW x4, 0(x2)       | SUBI x1, x1, #1        ; x4 = A[i] | N--
      SLT x5, x4, x0     | ADDI x2, x2, #4        ; x5 = (A[i] < 0) | x2 =
&A[i+1]
      BNEZ x5, next      | SUB x4, x0, x4         ; skip if (A[i]≥0) | A[i] = -
A[i]
      SW x4, -4(x2)      |                        ; store updated value of A[i]
next: BNEZ x1, loop      |                        ; continue if N > 0
end:
```

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

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

Problem P4.3.B

```
      SGT x3, x1, x0
      BNEZ x3, end                                ; x3 = (N > 0) | if N ≤ 0
loop: LW x4, 0(x2)            | SUBI x1, x1, #1  ; x4 = A[i] | N--
      CMPLTZ P0, x4           | ADDI x2, x2, #4  ; P0 = (A[i]<0) | x2 =
&A[i+1]
      (P0) SUB x4, x0, x4     |                  ; A[i] = -A[i]
      (P0) SW x4, -4(x2)      | BNEZ x1, loop    ; store updated value of A[i]
end:                                              ; continue if N > 0
```

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

Problem P4.3.C

```
; Initial Conditions:
;     x1 = N
;     x2 = &A[i]

x3 = N > 0
x4 = A[i]
x5 = N odd
x6 = A[i+1]

      SGT x3, x1, x0
      BNEZ x3, end            | ANDI x5, x1, #1
      BEQZ x5, loop           | LW x4, 0(x2)
      CMPLTZ P0, x4           | SUBI x1, x1, #1
      ADDI x2, x2, #4         | (P0) SUB x4, x0, x4
      (P0) SW x4, -4(x2)      | BEZ x1, end

loop: LW x4, 0(x2)            | SUBI x1, x1, #2
      CMPLTZ P0, x4           | LW x6, 4(x2)
      (P0) SUB x4, x0, x4     | CMPLTZ P1, x6
      (P0) SW x4, 0(x2)       | (P1) SUB x6 x0, x6
                              | (P1) SW x6 4(x2)
      ADDI x2, x2, #8         | BNEZ x1, loop
end:
```

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

Problem P4.4: Vector Machines

Problem P4.4.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_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
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
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																				

instr. \ cycle	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
LV_1																				
LV_1																				
LV_1																				
LV_1																				
LV_2																				
LV_2																				
LV_2																				
LV_2																				
ADDV																				
ADDV																				
ADDV																				
ADDV	W																			
SUBVS	—	R	X1	W																
SUBVS			R	X1	W															
SUBVS				R	X1	W														
SUBVS					R	X1	W													
SV	—	—	—	—	—	—	—	R	M1	M2	M3	M4								
SV									R	M1	M2	M3	M4							
SV										R	M1	M2	M3	M4						
SV											R	M1	M2	M3	M4					
ADDI																				
ADDI																				
ADDI																				
SUBI																				
BNEZ																				
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_1															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 P4.4.B

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

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

Problem P4.4.C

Instr. Number	Instruction
I1	LV V1, x1
I2	LV V2, x2
I6	**ADDI x1, x1, 128**
I7	**ADDI x2, x2, 128**
I10	**LV V5, x1**
I11	**LV V6, x2**
I3	**ADDV V3, V1, V2**
I4	**SUBVS V4, V3, x4**
I5	**SV x3, V4**
I12	**ADDV V7, V5, V6**
I13	**SUBVS V8, V7, x4**
I8	**ADDI x3, x3, 128**
I14	**SV x3, V8**
I15	ADDI x1, x1, 128
I16	ADDI x2, x2, 128
I17	ADDI x3, x3, 128
I9	SUBI x5, x5, 32
I18	SUBI x5, x5, 32
I19	BNEZ x5, 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 P4.5: Multithreading

Problem P4.5.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 x3, 0(x1)`	1	100
`LW x4, 4(x1)`	2	101
`SEQ x3, x3, x2`	101	101
`BNEZ x3, End`	102	102
`ADD x1, x0, x4`	103	103
`BNEZ x1, Loop`	104	104

Problem P4.5.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 P4.5.C

	Throughput	**Latency**
Better	√	
Same		
Worse		√

Problem P4.5.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 $\lceil 98/6 \rceil + 1 = 18$ threads.

Problem P4.6: Multithreading

Problem P4.6.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 P4.6.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 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: Since not all the treads 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 × (number of architecture registers)] physical registers for an in-order system. Since it is SMT, it is actually least [N × (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 Associatively: This is for performance reasons.