Name ____________________________

Computer Architecture and Engineering
CS152 Quiz #4

April 11th, 2012

Professor Krste Asanović

Name:__ANSWER SOLUTIONS____

This is a closed book, closed notes exam.

80 Minutes

17 Pages

Notes:

- Not all questions are of equal difficulty, so look over the entire exam and budget your time carefully.
- Please carefully state any assumptions you make.
- Please write your name on every page in the quiz.
- You must not discuss a quiz's contents with other students who have not yet taken the quiz. If you have inadvertently been exposed to a quiz prior to taking it, you must tell the instructor or TA.
- You will get **no** credit for selecting multiple-choice answers without giving explanations if the instruction ask you to explain your choice.

Writing name on each sheet	________	1 Points
Question 1	________	24 Points
Question 2	________	20 Points
Question 3	________	17 Points
Question 4	________	18 Points
TOTAL	________	**80 Points**

Question 1: Scheduling for VLIW Machines (24 points)

The following questions concern the scheduling of floating-point code on a VLIW machine.

Written in C, the code is as follows (in all questions, you should assume that all arrays do not overlap in memory):

```
#define N 1024
float A[N],B[N],C[N],D[N];

... arrays are initialized ...

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

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

```
  addi $n, $0, 1024
  addi $i, $0, 0
loop:
  flw  $a,  A($i)     # floating point load word
  flw  $b,  B($i)
  fadd $c,  $a, $b    # floating point add word
  fmul $d,  $a, $c    # floating point multiply word
  fsw  $c,  C($i)     # floating point store word
  fsw  $d,  D($i)
  addi $i,  $i, 4
  addi $n,  $n, -1
  bnez $n,  loop
```

A, **B**, **C**, and **D** are immediates set by the compiler to point to the beginning of the **A**, **B**, **C**, and **D** arrays. Register $i is used to index the arrays. For this ISA, register $0 is read-only and always returns the value zero.

Name ______________________________

This code will run on the VLIW machine that was presented in lecture and used in PSet #4, shown here:

Int Op 1	Int Op 2	Mem Op 1	Mem Op 2	FP Addition	FP Multiply

Figure 1. The VLIW machine

Our machine has six execution units:

- **two** ALU units, latency **one-cycle** (i.e., dependent ALU ops can be issued back-to-back), also used for branches.
- **two** memory units, latency **three cycles**, fully pipelined, each unit can perform either a load or a store.
- **two** FPU units, latency **four cycles**, fully pipelined, one unit can only perform **fadd** operations, the other can only perform **fmul** operations.

Our machine has no interlocks. All register values are read at the start of the instruction before any writes from the same instruction take effect (i.e., no WAR hazards between operations within a single VLIW instruction).

The naive scheduling of the above assembly code (one operation per instruction) turns into the following schedule (another copy of this is provided in *Appendix A*. Feel free to remove *Appendix A* from the test to help in answering Questions Q.1A and Q1.B):

Inst	Int Op 1	Int Op 2	Mem Op 1	Mem Op 2	FP Add	FP Mul
1	addi $n,$0,1024	--	--	--	--	--
2	addi $i, $0, 0	--	--	--	--	--
loop: 3	--	--	flw $a, A($i)	--	--	--
4	--	--	flw $b, B($i)	--	--	--
5	--	--	--	--	--	--
6	--	--	--	--	--	--
7	--	--	--	--	fadd $c, $a, $b	--
8	--	--	--	--	--	--
9	--	--	--	--	--	--
10	--	--	--	--	--	--
11	--	--	--	--	--	fmul $d, $a, $c
12	--	--	--	--	--	--
13	--	--	--	--	--	--
14	--	--	--	--	--	--
15	--	--	fsw $c, C($i)	--	--	--
16	--	--	fsw $d, D($i)	--	--	--
17	addi $i, $i, 4	--	--	--	--	--
18	addi $n, $n, -1	--	--	--	--	--
19	bnez $n, loop (Inst. 3)	--	--	--	--	--

Name ______________________________

Q1.A: Loop Unrolling & General Optimizations (12 points)

Loop unrolling will enable higher throughput over the naive implementation shown on the previous page. Unroll the above code **once**, to get two iterations inflight for every loop in the VLIW code. You should also consider other performance optimizations to improve throughput (i.e., re-ordering operations, adding or removing operations, and packing operations into a single VLIW instruction). However, do *not* do software pipelining. That is for the next part. To receive full credit, your code should demonstrate good throughput.

Note: the array length for this program is statically declared as 1024, which will help you make simplifying assumptions in the start-up and exit code. You may not need all entries in the following table. For your convenience, an empty entry will be interpreted as a NOP.

Hint: when indexing arrays, an operation such as `ld $a, A+8($i)` is perfectly valid (remember that $i is a register, and **A** is the literal set by the compiler to point to the beginning of the **A** array).

Extra Hint: For this problem, it is recommended that you name your registers $a0, $a1, $b0, $b1, etc.

What is the resulting throughput of the code, in "floating point operations per cycle"? Only consider the steady-state operation of the loop.

First loop starts on cycle 2, second iteration of the loop starts on cycle 15, so:

4 FLOPS / (15-2) cycles in the loop =

Throughput (FLOPS/cycle) ___4/13__

Name ______________________________

Inst	Int Op 1	Int Op 2	Mem Op 1	Mem Op 2	FP Add	FP Mul
1	addi $n,$0,1024	addi $i, $0, 0	--	--	--	--
loop: 2	--	--	flw $a0, A($i)	flw $b0, B($i)	--	--
3	--	--	flw $a1, A+4($i)	flw $b1, B+4($i)	--	--
4	--	--	--	--	--	--
5	--	--	--	--	fadd $c0, $a0, $b0	--
6	--	--	--	--	fadd $c1, $a1, $b1	--
7	--	--	--	--	--	--
8	--	--	--	--	--	--
9	--	--	fsw $c0, C($i)	--	--	fmul $d0, $a0, $c0
10	--	--	fsw $c1, C+4($i)	--	--	fmul $d1, $a1, $c1
11	--	--	--	--	--	--
12	--	--	--	--	--	--
13	addi $n, $n, -2	--	fsw $d0, D($i)	--	--	--
14	addi $i, $i, 8	bnez $n, loop (Inst. 3)	fsw $d1, D+4($i)	--	--	--

bnez

11 points for table, 1 point for FLOP calculation.

-1/2 points for incrementing $i by 4 (instead of 8)
-1/2 points for decrementing $n by 1 (instead of -2)
-1/2 points for each error in immediates
-1 point for issuing an operation too early
-1/2 points for suboptimal scheduling of an instruction/operation
-1 points for getting throughput calculation incorrect
-3 points for not condensing the two iterations together

Note that some ops, such as "addi $i, $i, 8" are not on the critical path, and can be correctly placed in many of the instructions (so long as immediates, etc. are handled correctly).

Name ______________________________

Q1.B: Software Pipelining (12 points)

An other optimization technique is software pipelining. Rewrite the assembly code to leverage software pipelining (do ***not*** also use loop unrolling). **You *only* need to write the inner loop**: you do ***not*** need to write the prologue or epilogue. You may not need to use all entries in the table. To receive full credit, your code should demonstrate good throughput.

Hint: you can move floating point data to another register by using the instruction **`fmove fd,fs1`**. This may be helpful if you need to copy data to another register. It can use either the FP Add slot or the FP Mul slot, but it only takes a single cycle to execute (it also does *not* count as a FLOP).

Inst	Int Op 1	Int Op 2	Mem Op 1	Mem Op 2	FP Add	FP Mul
1	--	--	flw $a, A+16 ($i)	flw $b, B+16 ($i)	fadd $c, $a, $b	--
2	addi $n, $n, -1	--	--	fsw $c, C+8 ($i)	fmove $e, $a	fmul $d, $e, $c
3	addi $i, $i, 4	bnez $n, loop (Inst. 3)	fsw $d, D($i)	--	--	--

-1 point if done in 4 cycles
-2 points if done in 5 cycles
-3 points if done in 6 cycles (or more points, this is only two iterations in flight...)

check for addresses....

roughly:
-3 for incorrect addresses
-5 for not honoring dependencies
-2 overwriting $a (needs to be copied to keep it around for fmul)
-2 throughput incorrect
-2 for unrolling loop

What is the resulting throughput of the code, in "floating point operations per cycle"? Only consider the steady-state operation of the loop.

Throughput (FLOPS/cycle) ___ 2/3__

Question 2: Vector Processors (20 points)

We will now look at the code segment from **Question 1** and how it would be scheduled on a vector processor.

Here the code is again, written in C:

```
#define N 1024
float A[N],B[N],C[N],D[N];

... arrays are initialized ...

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

In *traditional* vector assembly code:

```
  addi $n, $0, 1024
  addi $i, $0, 0
loop:
  vsetvl $vlen, $n        # set vector length
  vflw   $va,  A($i)
  vflw   $vb,  B($i)
  vvfadd $vc,  $va,$vb    # perform vector-vector fadd
  vvfmul $vd,  $va,$vc    # perform vector-vector fmul
  vfsw   $vc,  C($i)
  vfsw   $vd,  D($i)
  slli   $temp, $vlen, 2
  add    $i,  $i, $temp
  sub    $n,  $n, $vlen
  bnez   $n,  loop
```

The baseline vector processor in this question has the following features:

- 32 elements per vector register
- 8 lanes
- one ALU per lane: 1 cycle latency
- one LD/ST unit per lane: 3 cycle latency
- no dead time
- no support for chaining
- scalar instructions execute separately on a control processor (5-stage, in-order)

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

Q2.A: Scheduling Vector Code, No Chaining (14 points)

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

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.
- VFLW1 and VFLW2 refer to the first and second VFLW instructions in the loop, etc.

Name ______________________________

Cycle Inst	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
vflw1	F	D	R	M1	M2	M3	W																																	
vflw1				R	M1	M2	M3	W																																
vflw1					R	M1	M2	M3	W																															
vflw1						R	M1	M2	M3	W																														
vflw2		F	D	-	-	-	R	M1	M2	M3	W																													
vflw2								R	M1	M2	M3	W																												
vflw2									R	M1	M2	M3	W																											
vflw2										R	M1	M2	M3	W																										
vvfadd			F	D	-	-	-	-	-	-	-	-	-	-	R	X	W																							
vvfadd																R	X	W																						
vvfadd																	R	X	W																					
vvfadd																		R	X	W																				
vvfmul				F	D	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	R	X	W																	
vvfmul																						R	X	W																
vvfmul																							R	X	W															
vvfmul																								R	X	W														
vfsw1					F	D	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	R	M1	M2	M3	W											
vfsw1																										R	M1	M2	M3	W										
vfsw1																											R	M1	M2	M3	W									
vfsw1																												R	M1	M2	M3	W								
vfsw2						F	D	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	R	M1	M2	M3	W							
vfsw2																														R	M1	M2	M3	W						
vfsw2																															R	M1	M2	M3	W					
vfsw2																																R	M1	M2	M3	W				
slli							F	D	X	M	W																													
add								F	D	X	M	W																												
sub									F	D	X	M	W																											
bnez										F	D	X	M	W																										
vsetvl											F	D	X	M	W																									
vflw1												F	D	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	R	M1	M2	M3	W			
vflw1																																		R	M1	M2	M3	W		
vflw1																																			R	M1	M2	M3	W	
vflw1																																				R	M1	M2	M3	W

-2 for structural hazard of vvfmul and vfsw1 on read ports (can't dual issue instructions!)
-3 for trying to issue out-of-order. -3 for trying to issue too early. -1 for non-optimal mistakes. Note that vfsw1 doesn't depend on vvfmul, so it can issue immediately (once the vvfmul is out of the way that is).

Q2.B: With Chaining (6 points)

In this part, we analyze the performance benefits of chaining. 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).

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 Q2.A, VFLW1 begins execution in cycle 3 and VFLW2 in cycle 7. Therefore, there are 4 cycles between VFLW1 and VFLW2.

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.

Name ______________________________

Hint: You should consider each pair of vector instructions independently, and you can ignore the scalar instructions.

Now the chaining diagram:

Cycle Inst	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
vflw1	F	D	R	M1	M2	M3	W																																	
vflw1				R	M1	M2	M3	W																																
vflw1					R	M1	M2	M3	W																															
vflw1						R	M1	M2	M3	W																														
vflw2		F	D	-	-	-	R	M1	M2	M3	W																													
vflw2								R	M1	M2	M3	W																												
vflw2									R	M1	M2	M3	W																											
vflw2										R	M1	M2	M3	W																										
vvfadd			F	D	-	-	-	-	-	-	R	X	W																											
vvfadd												R	X	W																										
vvfadd													R	X	W																									
vvfadd														R	X	W																								
vvfmul				F	D	-	-	-	-	-	-	-	-	-	R	X	W																							
vvfmul																R	X	W																						
vvfmul																	R	X	W																					
vvfmul																		R	X	W																				
vfsw1					F	D	-	-	-	-	-	-	-	-	-	-	-	-	R	M1	M2	M3	W																	
vfsw1																				R	M1	M2	M3	W																
vfsw1																					R	M1	M2	M3	W															
vfsw1																						R	M1	M2	M3	W														
vfsw2						F	D	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	R	M1	M2	M3	W													
vfsw2																								R	M1	M2	M3	W												
vfsw2																									R	M1	M2	M3	W											
vfsw2																										R	M1	M2	M3	W										
slli							F	D	X	M	W																													
add								F	D	X	M	W																												
sub									F	D	X	M	W																											
bnez										F	D	X	M	W																										
vsetvl											F	D	X	M	W																									
vflw1												F	D	-	-	-	-	-	-	-	-	-	-	-	-	-	R	M1	M2	M3	W									
vflw1																												R	M1	M2	M3	W								
vflw1																													R	M1	M2	M3	W							
vflw1																														R	M1	M2	M3	W						

vector processor configuration	number of cycles between successive vector instructions					total cycles per vector loop iteration
	vflw1, vflw2	vflw2, vvfadd	vvfadd, vvfmul	vvfmul1, vfsw2	vfsw2, vflw1	
8 lanes, no-chaining	4	8	6	8	4	30
8 lanes, chaining	4	4	4	8	4	24

vvfmul can't go sooner than 4 cycles, due to waiting for vvfadd to finishing reading the RF. Same with vfsw1. Notice that there are some cycles where two instructions are writing back to the regfile. This is okay though because the regfile is banked and different elements are writing back (i.e., micro-threads 0-7 are writing back at the same time that micro-threads 16-23 are also writing back, thus no structural hazard).

Question 3: Multithreading (17 points)

For this problem, we are interested in evaluating the effectiveness of multithreading using the code from **Question 1** and **Question 2**:

```
#define N 1024
float A[N],B[N],C[N],D[N];

... arrays are initialized ...

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

The corresponding assembly code is:

```
  addi $n, $0, 1024
  addi $i, $0, 0
loop:
  flw  $a,  A($i)
  flw  $b,  B($i)
  fadd $c,  $a, $b
  fmul $d,  $a, $c
  fsw  $c,  C($i)
  fsw  $d,  D($i)
  addi $i,  $i, 4
  addi $n,  $n, -1
  bnez $n,  loop
```

Assume the following:
- Our system does not have a cache.
- Each memory operation directly accesses main memory and has a latency of 20 CPU cycles (if a `load` is issued on cycle X, a dependent instruction can issue on cycle X+20).
- The load/store unit is fully pipelined.
- 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.
- The `fmul` and `fadd` instructions both have a use-delay of 4 cycles (there are 4 cycles of stalling before an instruction dependent on the FP instruction can issue).

Name ______________________________

Q3.A: In-order Single-threaded Processor (5 points)

How many cycles does it take to execute one iteration of the loop in steady-state for a single-threaded processor? Do not re-order the instructions or modify the code in any way.

```
addi $n, $0, 1024
  addi $i, $0, 0
loop:                    1st iter   / 2nd iter
  flw  $a,  A($i)          0           35
  flw  $b,  B($i)          1           etc.
  fadd $c,  $a, $b         21
  fmul $d,  $a, $c         26
  fsw  $c,  C($i)          27
  fsw  $d,  D($i)          31
  addi $i,  $i, 4          32
  addi $n,  $n, -1         33
  bnez $n,  loop           34
```

-1 getting steady state part wrong, -1 off by one error, -3 for missing a dependence, -3 for considering st end time
note: we do NOT care when the stores finish... we are concerned with the steady state of the loop. Once a store is sent off to memory, we are finished, as far as the core is concerned.

cycles __35__

Q3.B: In-order Multi-threaded Processor, Round Robin (6 points)

Now consider multithreading the pipeline. Threads are **switched every cycle** using a fixed round-robin schedule. If the thread is not ready to run on its turn, a bubble is inserted into the pipeline.

Each thread executes the above algorithm, and is calculating its own independent piece of the **C** and **D** arrays (i.e., there is no communication required between threads). Assuming an infinite number of registers, what is the **minimum** number of threads we need to fully utilize the processor? You are free to re-schedule the assembly as necessary to minimize the number of threads required.

```
loop:                    ST version  / MT version (round-robin)
  flw  $a,  A($i)          0           0
  flw  $b,  B($i)          1           n
  addi $i,  $i, 4          2           2n
  addi $n,  $n, -1         3           3n
  fadd $c,  $a, $b         21          4n
  fmul $d,  $a, $c         26          5n, etc.
  fsw  $c,  C-4($i)        27
  fsw  $d,  D-4($i)        31
  bnez $n,  loop           32
```

-2 for not re-ordering instructions (saying 20 threads, which is correct, but not optimal)
-3 for getting dependency wrong (4 instructions instead of 3)

need to hide latency of fload to fadd (20 cycles). we can move in two addi instructions to help hide it:

4n-n >= 20 cycles
3n>= 20 cycles
n >= 6.66 threads <u>**7 threads is the answer**</u>

A better instruction ordering though arises from using "software pipelining"! You can schedule the stores *from the previous iteration* before the FP ops of the current iteration. So its flw,flw,addi,addi,fsw,fsw,fadd,fmul,bnez! This gets you <u>**20/5 = 4 threads**</u>.

Q3.B: *In-order Multi-threaded Processor, Dynamic Scheduling (6 points)*

Now consider an in-order, multi-threaded pipeline in which threads are **dynamically scheduled as needed** (i.e., the pipeline can pick the next instruction from any thread that is ready to execute). If the thread is not ready to run on its turn, a new thread is switched in.

What is the **minimum** number of threads we need to achieve peak performance? Again, you are free to re-schedule the assembly as necessary to minimize the number of threads required.

Acceptable answers was 4 or 5 threads.

The problem statement stated "threads are dynamically scheduled as needed (i.e., the pipeline can pick the next instruction ***from any thread***...)". This means that we can be a bit "loose" with scheduling instructions "perfectly" (most quiz questions will ask about fine-grain "round-robin" scheduling and data-dependent course-grain scheduling that only switch on a thread stall).

Thus, since we are allowed to be creative with the scheduling of instructions, and allow the processor to magically pick the best instructions to issue (basically prioritize the oldest loads over all other instructions), we can get the following scheduling which only takes ***4 threads:***

```
bnez $n,  loop
flw  $a,  A($i)
flw  $b,  B($i)

<switch threads, executes 3 threads * 3 instructions>

addi $i,  $i, 4
addi $n,  $n, -1
fsw  $c,  C-8($i)  (notice, this is from the last iteration)

<switch threads, executes 3 threads * 3 instructions>

fadd $c,  $a, $b
fsw  $d,  D-4($i)

<switch threads>

fmul $d,  $a, $c   (Thread 1)

<switch threads>
```

Many students however took the problem statement to be discussing coarse-grain "data dependent" scheduling, in which threads are only switched out on what would otherwise be a data dependent stall. This is a perfectly valid interpretation of the problem, but makes it much, much harder to accurately solve.

From that point of view, here is one possible scheduling of threads (assume perfect branch prediction). We can get 7 instructions issued before having to switch threads (I will refer to this chunk of instructions as "Block A"). The difficulty introduced by this problem however, is that we must switch threads after each FP op as well ("Block B" and "Block C", respectively).

```
Block A
  fsw  $c,  C-4($i)    (notice we don't switch threads between loop iterations)
  fsw  $d,  D-4($i)
  bnez $n,  loop
  flw  $a,  A($i)
  flw  $b,  B($i)      (we need to hide this load's 19 stall cycles)
  addi $i,  $i, 4
  addi $n,  $n, -1

<switch threads, hide 17 cycles>

Block B
  fadd $c,  $a, $b

<switch threads, hide 4 cycles>

Block C
  fmul $d,  $a, $c

<switch threads, hide 4 cycles>
```

To get a ball park guess, we can say that we have 25 stall cycles to hide, and with 9 instructions, that would require 3+1 threads. That wouldn't actually work out though, because the FP ops would have to be scheduled 18 cycles after the end of Block A (it *might* work if the processor "just knew" how best to dynamically schedule the instructions in which it prioritized getting the loads issued as early as possible, and then evenly issued book-keeping instructions to hide the stall cycles).

But if we only consider having to hide the 25 stall cycles with Block A instructions (which are ready to fire after only 4 cycles of stalling from Block C), then we get ciel(25/7)+1, or 5 threads. If you draw this scheduling out by hand though (in which threads only switch once they must stall), you will still have to occasionally stall.

Both of these estimates though still garnered full credit.

It is also possible to reschedule the code using software pipelining to only get a single stall.
(fadd c, fsw d, addi n, addi i, flw b, fmul d, flw a, fsw c, bne)
This requires a lot of cleverness to make sure you aren't violating any hazards! (since a non-VLIW pipeline will interlock if there is a hazard).

In this way, 17 cycles of stall exist between the bne, and the fadd c. ciel(17 / 9)+1 = 3 threads!

Question 4: Iron Law (18 points)

For each of the following questions, describe how you expect each proposed change will affect **instructions/program** and **cycles/instruction** (CPI), and **seconds/cycle**.

Mark whether the following modifications will cause each parameter to **increase, decrease**, or whether the change will have **no effect**. **Explain your reasoning** to receive credit.

Q4.A: Doubling the Width of a VLIW Macine (6 points)

What is the effect on the following parameters from doubling the width of a VLIW machine? This means that we are double the width of the VLIW instruction, which includes doubling the number of functional units to match.

Assume the machine has data caches and uses interlocks to handle data hazards on cache misses, and that the code has a lot of ILP to exploit.

instructions/program:

cycles/instruction (CPI)**:**

seconds/cycle:

Q4.B: Doubling the Number of Lanes in a Vector Machine (6 points)

What is the effect on the following parameters from doubling the number of lanes in a vector machine? Assume the vector register length stays the same, and that the code has a lot of DLP to exploit.

instructions/program:

cycles/instruction (CPI)**:**

seconds/cycle:

Name ____________________________

Q4.C: Doubling the Number of Threads for a Multi-threaded Processor (6 points)

What is the effect on the following parameters from doubling the number of threads available to run on a multi-threaded processor? You can assume the code has been written to take advantage of as many threads as the machine can provide.

instructions/program:

cycles/instruction (CPI)**:**

seconds/cycle:

END OF QUIZ