Name _____(answer key)_____

Computer Architecture and Engineering
CS152 Quiz #4
April 11th, 2011
Professor Krste Asanović

Name:___<ANSWER KEY>___

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	_____	2 Points
Question 1	_____	24 Points
Question 2	_____	22 Points
Question 3	_____	14 Points
Question 4	_____	18 Points
TOTAL	_____	**80 Points**

Question 1: Scheduling for VLIW Machines (24 points)

The following questions concerns 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
double S[N],A[N],B[N],Y[N];

... arrays are initialized ...

for(int i = 0; i < N; i++)
  S[i] = A[i] * B[i] + Y[i];
```

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

```
  addi $n, $0, 1024
  addi $i, $0, 0
loop:
  ld   $a,  A($i)
  ld   $b,  B($i)
  fmul $t,  $a, $b
  ld   $y,  Y($i)
  fadd $s,  $t, $y
  st   $s,  S($i)
  addi $i,  $i, 8
  addi $n,  $n, -1
  bnez $n,  loop
```

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

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	--	--	ld $a, A($i)	--	--	--
4	--	--	ld $b, B($i)	--	--	--
5	--	--	--	--	--	--
6	--	--	--	--	--	--
7	--	--	--	--	--	fmul $t, $a, $b
8	--	--	ld $y, Y($i)	--	--	--
9	--	--	--	--	--	--
10	--	--	--	--	--	--
11	--	--	--	--	fadd $s, $t, $y	--
12	--	--	--	--	--	--
13	--	--	--	--	--	--
14	--	--	--	--	--	--
15	--	--	st $s, S($i)	--	--	--
16	addi $i, $i, 8	--	--	--	--	--
17	addi $n, $n, -1	--	--	--	--	--
18	bnez $n, loop (Inst. 3)	--	--	--	--	--

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.

4 FLOPS / 13 cycles in the loop =

Throughput (FLOPS/cycle) ___4/13__

Name _____(answer key)_____

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			ld $a0, A($i)	ld $b0, B($i)		
3			ld $a1, A+8($i)	ld $b1, B+8($i)		
4						
5						fmul $t0, $a0, $b0
6			ld $y0, Y($i)			fmul $t1, $a1, $b1
7			ld $1y, Y+8($i)			
8						
9					fadd $s0, $t0, $y0	
10					fadd $s1, $t1, $y1	
11						
12						
13		addi $n, $n, -2	st $s0, S($i)			
14	addi $i, $i, 16	bnez $n, loop (Inst. 2)	st $s1, S+8($i)			
15						
16						
17						
18						
19						
20						
21						
22						

Table Q1.A (Loop unrolling)

-1/2 points for incrementing $i by 8 (instead of 16)
-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 “ld $y, Y($i)” and “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).

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 should show the loop prologue and epilogue to initiate and drain the software pipeline. You may not need all entries in the following table. However, it is okay if you run out of entries writing the epilog code.

Note: the array for this program is statically allocated to 1024 elements, which allows the compiler (i.e., you) to make simplifying assumptions about the prologue and epilog code.

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

Depends on the student's implementation. For the following table:

2 FLOPS / 4 cycles in the loop =

Throughput (FLOPS/cycle) ___1/2__

Name _____(answer key)_____

Inst	Int Op 1	Int Op 2	Mem Op 1	Mem Op 2	FP Add	FP Mul
1	addi $n,$0,1021	addi $i, $0, 0				
prologue: 2			ld $a, A($i)	ld $b, B($i)		
3						
4						
5	addi $i, $i, 8		ld $y, Y($i)			fmul $t, $a, $b
6			ld $a, A($i)	ld $b, B($i)		
7						
8						
9	addi $I, $i, 8		ld $y, Y($i)		fadd $s, $t, $y	fmul $t, $a, $b
10			ld $a, A($i)	ld $b, B($i)		
11						
12						
loop: 13	addi $i, $i, 8		ld $y, Y($i)	st $s, S-16($i)	fadd $s, $t, $y	fmul $t, $a, $b
14			ld $a, A($i)	ld $b, B($i)		
15		addi $n, $n, -1				
16		bnez $n, loop				
epilogue: 17			ld $y, Y($i)	st $s, S-16($i)	fadd $s, $t, $y	fmul $t, $a, $b
18						
19						
20						
21				st $s, S-8($i)	fadd $s, $t, $y	
23/24/25				....		
25				st $s, S($i) (cycle 25)		

Note this assumes that you can't read from a register after an instruction has been issued that writes to that same register (RAW) until the write has finished. If this constraint is relaxed, a more optimal solution can be used.
-1/2 points: $n is initialized to 1021, because we need to leave the loop early and let the epilogue finish the last three iterations as the loop drains.
-1/2 points: for each wrong immediate
-1 for throughput calculation
-3 points for epilogue
-3 for prologue
-5 for loop
-2 for missing an important operation something
-3-5 points for being very suboptimal

"ld $y" is tricky, because the load can happen very early, but we have to be careful for it not to be overwritten by later "$ld y"'s before $y is used.

Question 2: Vector Processors (22 points)

We will now look at three different code segments. For each, comment on whether it can be vectorized, and then write the strip-mined vector assembly code if it can be vectorized.

Use (`mtc1 vlr, RI`) to set the machine's vector length to `RI`. However, this will throw an exception if `RI` is greater than the machine's maximum vector length. Note that the machine's maximum vector length is held in a register called `VLENMAX`.

Q2.A: (8 points)

Written in C, the first code piece is as follows:

```
// N is the array size
double A[N],B[N];

... arrays are initialized ...

for(int i = 0; i < N; i++)
  A[i] = A[i] + B[i];
```

```
; initial conditions
; RA <= A[0]
; RB <= B[0]
; RN <= N
; VLENMAX <= max vlen

; inner-most loop
   LV   VA, RA
   LV   VA, RB
   ADDV VA, VA, VB
   SV   VA, RA
```

The initial conditions are shown above, along with some example vector assembly code to describe the code found in the inner-most loop (lacking the strip-mining).

Can this code be vectorized? *Explain* why or why not. If it can be vectorized, write the vector assembly code. Make sure to add a strip-mine loop to deal with the case where N is greater than the machine's vector length. However, you can assume that N will be a multiple of the vector length.

(Write your answer on the following page).

Q2.A continued ...

```
for(int i = 0; i < N; i++)
  A[i] = A[i] + B[i];
```

```
; initial conditions
; RA = &(A[0])
; RB = &(B[0])
; RN = N
; VLENMAX = max vlen
```

YES.

```
 BEQZ RN, DONE         ; handle N == 0 case
LOOP:
 MTC1 vlr, VLENMAX     ; set vector length

 LV   VA, RA
 LV   VB, RB
 ADDV VA, VA, VB
 SV   VA, RA

 SLLI RK, RI, 3        ; convert # of elements to # of bytes
 ADD  RA, RA, RK       ; RA = &(A[RA+RI])
 ADD  RB, RB, RK       ; RB = &(B[RB+RI])
 SUB  RN, RN, VLENMAX
 BNEZ RN, LOOP
DONE:
```

Because we made the assumption that N will be a multiple of VLENMAX, you can just directly set vlr to VLENMAX.

Also, no points were taken off for not checking for the N==0 corner case.

- 3 points for not updating memory addresses (RA, RB).

-2 points for not incrementing addresses correctly (-1 for each mistake)
-1 point for not updating RN correctly
-1 for not setting mtc1 vlr

-6 for only saying yes, but providing no code

Q2.B: (7 points)

Written in C, the second code piece is as follows:

```
// N is the array size
double A[N+1],B[N];

... arrays are initialized ...

for(int i = 0; i < N; i++)
  A[i] = A[i+1] + B[i];
```

Can this code be vectorized? *Explain* why or why not. If it can be vectorized, write the vector assembly code. Make sure to add a strip-mine loop to deal with the case where N is greater than the machine's vector length. However, you can assume that N will be a multiple of the vector length.

YES. While there is an interdependency between iterations, it is still possible to vectorize.

```
LOOP:
 OR   RI, N, VLENMAX  ; set vlen = n % vlenmax
 MTC1 vlr, RI

 ADD  RC, RA, 8       ; RC = &(A[i+1]), if you will

 LV   VC, RC
 LV   VB, RB
 ADDV VA, VC, VB
 SV   VA, RA

 SLLI RK, RI, 3       ; convert # of elements to # of bytes
 ADD  RA, RA, RK      ; RA = &(A[RA+RI])
 ADD  RB, RB, RK      ; RB = &(B[RB+RI])
 ADDI RN, RN, -1
 BNEZ RN, LOOP
```

-1 for VC/RC being +1, and not +8

Q2.C: (7 points)

Written in C, the third code piece is as follows:

```
// N is the array size
double A[N+1],B[N+1];

... arrays are initialized ...

for(int i = 1; i < N+1; i++)
  A[i] = A[i-1] + B[i];
```

Can this code be vectorized? *Explain* why or why not. If it can be vectorized, write the vector assembly code. Make sure to add a strip-mine loop to deal with the case where N is greater than the machine's vector length. However, you can assume that N will be a multiple of the vector length.

-7 for saying 'yes'

NO. Computing A[i] in iteration "i" requires using the previously computed A[i-1] from iteration "i-1", which forces a serialization (you must compute the elements one at a time, and in-order).

Notice that the following code will get the **wrong** answer:

```
ADD  RC, RA, -8     ; RC = &(A[i-1]), if you will

LV   VC, RC
LV   VB, RB
ADDV VA, VC, VB     ; A[i] = A[i-1] + B[i] ????
SV   VA, RA
```

Again, the above vector assembly gets the **wrong** answer. Consider the following arrays:

```
A = {0, 1, 2, 3, 4, 5};
B = {0, 0, 0, 0, 0, 0};
```

Running the above C code, the correct and final solution is
`A = {0, 0, 0, 0, 0, 0}`
(because A[0] = 0, and A[i] = A[i-1] starting from iteration "i=1").

But running the above vector assembly, for a VLEN of 6, we incorrectly get:
`A = {0, 0, 1, 2, 3, 4};`

Question 3: Multithreading (14 points)

For this problem, we are interested in evaluating the effectiveness of multithreading using the following numerical code.

```
#define N 1024
double S[N],A[N],B[N],Y[N];

... arrays are initialized ...

for(int i = 0; i < N; i++)
  S[i] = A[N] * B[N] + Y[N];
```

Using the disassembler we obtain:

```
  addi $n, $0, 1024
  addi $i, $0, 0
loop:
  ld   $a,  A($i)
  ld   $b,  B($i)
  fmul $t,  $a, $b
  ld   $y,  Y($i)
  fadd $s,  $t, $y
  st   $s,  S($i)
  addi $i,  $i, 8
  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 takes 50 CPU cycles.
- 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 5 cycles.

Q3.A: In-order Single-threaded Processor (4 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 assembly code.

```
loop:                    start / end
  ld   $a,  A($i)          1 / 50
  ld   $b,  B($i)          2 / 51
  fmul $t,  $a, $b        52 / 57
  ld   $y,  Y($i)         53 / 102
  fadd $s,  $t, $y       103 / 108
  st   $s,  S($i)        109
  addi $i,  $i, 8        110 / 110
  addi $n,  $n, -1       111 / 111
  bnez $n,  loop         112 / 112
  LD   $a,  A($i)        113 ....
```

112 cycles. (113 - 1).

Notice because this is an in-order pipeline, the "ld $y" must wait for the FMUL to be executed first before we can start the load.

A lot of students had "off by one" errors. Memory operations were stated as "takes 50 CPU cycles". This means that a LD that starts in cycle 1 will end in cycle 50 (think about an ALU instruction that takes 1 CPU cycle. If it starts in cycle 1, it will end in cycle 1. Likewise, an instruction that takes 2 CPU cycles that starts in cycle 1 will end in cycle 2, etc.).

On the other hand, FADD/FMUL both have a "use-delay of 5 cycles". This means an FADD that starts in cycle 103 will end in cycle 108, and the dependent ST can begin in cycle 109 (e.g., for the standard 5-stage pipeline LD instructions have a use-delay of 1 cycle, which means if a LD started in cycle 1, it finished* in cycle 2, and a dependent ADD could start in cycle 3).

*I say "finished" because it still has to be committed/written-back, but from the point of view of when the LD value can be used, it finishes in cycle 2.

- 0 points for saying 4 versus 5 should be accepted as correct (fadd -> st)
-1/2 points for each off-by-one error (ld->fadd, ld->fmul)
-3 points for waiting for the store to finish for the store to complete
-3 points for counting latency of store, instead of steady-state of loop (throughput)

Q3.B: In-order Multi-threaded Processor (5 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 code, and is calculating its own independent piece of the **S** array (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:
  ld   $a,  A($i)      ;       1
  ld   $b,  B($i)      ;   N + 1
  ld   $y,  Y($i)      ;  2N + 1
  addi $i,  $i, 8      ;  3N + 1
  addi $n,  $n, -1     ;  4N + 1
  fmul $t,  $a, $b     ;  5N + 1
  fadd $s,  $t, $y     ;  6N + 1
  st   $s,  S-8($i)    ;  7N + 1
  bnez $n,  loop       ;  8N + 1
```

critical path: is either LD $y -> FADD, or LD $b -> FMUL (so long as N is greater than 5 to hide the FMUL latency).

For LD $y -> FADD:
(6N+1) - (2N+1) >= 50 cycles

4N = 50, N = 12.5

So **13 threads** will keep the machine fully utilized.

-1.5 points for correct analysis, not ideal (answers 25 threads by moving LD Y)
-2.5 for not rescheduling at all (answers 50 threads) between LD Y -> FADD
-3 points for wrong analysis, optimal rescheduling
-5 for getting non-optimal scheduling, wrong analysis

-3 for not using round-robin

Q3.C: Out-of-Order SMT Processor (5 points)

Now consider a four-wide, out-of-order SMT (simultaneous multithreading) processor. It can issue any ready instruction from any thread.

Assume all resources have infinite size (i.e., physical registers, ROB, instruction-window). There is still no cache (50 cycles to access memory) and the use-delay on floating-point arithmetic is still 5 cycles. A maximum of four instructions can be issued in a given cycle.

How many threads are required to saturate this machine, if every thread is running the above code?

This is actually a trick question. If an OoO processor has infinite resources, because each iteration of the loop is independent, the processor will dynamically unroll the loop in hardware and use instruction-level-parallelism to saturate the machine and hide the 50 cycle memory latency.

Thus, only *one* thread is required to saturate the machine.

Question 4: Iron Law (18 points)

Describe how you expect switching to each of the following architectures will affect instructions/program and cycles/instruction (CPI) relative to a baseline 5-stage, in-order processor.

Mark whether the following modifications will cause instruction/program and CPI to **increase, decrease**, or whether the change will have **no effect**. **Explain your reasoning** to receive credit.

Q4.A: VLIW (6 points)

How do instructions/program and CPI change when moving from a 5-stage-pipeline in-order processor to a traditional VLIW processor.

Inst/Program: DECREASES. The same number of operations need to be executed, but some of these operations can now be combined and scheduled to run in a single instruction, therefore, I/P goes down (however, if very few instructions could be scheduled together, and lots of NOP instructions had to be added to hide latencies in software, then Inst/Program could go up, but then you've done something horribly wrong by switching to VLIW).

Of course, the number of operations will increase, because NOPs will have to be added to to hide latencies and for when some operations couldn't be scheduled for a given functional unit. Thus, the static instruction bytes / program will likely increase.

Cycles/Instruction:

Depends on your answer to inst/program.

If the student said that inst/program decreases, then CPI must be the same or *increase*, as the machine is more likely to see a stall.

If the student said that inst/program increases because of added NOP-only instructions, then CPI *decreases* as it will now stall less.

-1 point if student said that inst/program decreases *and* CPI decreases

Q4.B: Vector Processors (6 points)

How do instructions/program and CPI change when moving from a 5-stage-pipeline in-order processor to a single-lane vector processor.

Inst/Program: DECREASES. Instructions will decrease because many operations can now be expressed in a single instruction. Also, a vector processor can express a number of loop iterations in small number of vector instructions, eliminating the loop-overhead expended every iteration (versus once per vector).

Cycles/Instruction: INCREASES. Vector instructions take multiple cycles to execute, though chaining can help overlap execution of multiple vector instructions to mitigate this issue.

Q4.C: Multithreaded Processor (6 points)

How do instructions/program and CPI change when moving from a 5-stage-pipeline in-order processor to a multithreaded processor?

Assume that the new processor is still an in-order, 5-stage-pipeline processor, but that it has been modified to switch between two threads every clock cycle (fine-grain multithreading). If a thread is not ready to be issued (e.g., a cache miss), a bubble is inserted in the pipeline.

Inst/Program: UNCHANGED. The ISA needs not change to enable multithreading.

Cycles/Instruction:

The answer depends on your point of view.

From the POV of one thread, CPI *increases*, as multithreading adds contention of pipeline resources, and thus structural hazards can now occur that will stall the thread that is otherwise ready to execute.

From the POV of multiple threads, the aggregate CPI *decreases*, as they are less likely to encounter a stall since independent instructions can be interleaved.

END OF QUIZ