Computer Architecture and Engineering
CS152 Quiz #4

April 11th, 2016

Professor George Michelogiannakis

Name:______<ANSWER KEY>_____

This is a closed book, closed notes exam.

80 Minutes

21 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 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 instructions ask you to explain your choice.

Writing name on each sheet	________________	1 Point
Question 1	________________	30 Points
Question 2	________________	30 Points
Question 3	________________	23 Points
Question 4	________________	16 Points
TOTAL	________________	100 Points

Question 1: VLIW Machines [30 points]

In this question, we will consider the execution of the following code segment on a VLIW processor.

```
loop:
  flw  f1, 0(x1)
  lw   x9, 0(x2)
  fmul f3, f1, f1
  add  x7, x5, x7
  sw   x7, 0(x1)
  fsw  f3, 0(x2)
  addi x1, x1, 4
  add  x2, x2, x9
  bne  x2, x5, loop
```

This code will run on a VLIW machine with the following instructions format:

Int Op	Branch	Mem Op	FP or Int Op	FP Add	FP Mul

Our machine has six execution units (in order from left to right in the instruction above). All execution units are fully pipelined and latch their operands in the first stage.

- One integer unit, latency two cycles.
- One branch unit, latency one cycle.
- One memory unit, latency three cycles, each unit can perform both loads and stores. You can assume a 100% cache hit rate (i.e., no variability).
- One functional unit that can deal with integer or floating point operations of any kind (additions and multiplications). Latency five cycles.
- One floating point add unit. Latency three cycles.
- One floating point multiply unit. Latency four cycles.

This 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). Functional units write the register file at the end of their last pipeline stage, and there are no data forwarding or stalls. I.e., a functional unit that requires 4 cycles and starts an operation in cycle 1 will have its result be visible at the beginning of cycle 5 (writes at the end of cycle 4).

Q1.A Scheduling VLIW Code, Naïve scheduling [6 points]
Schedule operations into VLIW instructions in the following table. Show only one iteration of the loop. Schedule the code efficiently (try to use the least number of cycles), but do not use software pipelining or loop unrolling. You don't need to write in NOPs.

Inst	Int Op	Branch	Mem Op	FP/Int Op	FP Add	FP Mul
1			flw f1, 0(x1)			
2	add x7, x5, x7		lw x9, 0(x2)			
3	addi x1, x1, 4					
4			sw x7, 0(x1)			fmul f3, f1, f1
5						
6						
7	add x2, x2, x9					
8			fsw f3, 0(x2)			
9		bne x2, x5, loop				
10						
11						
12						
13						
14						
15						
16						
17						
18						
19						
20						
21						
22						
23						

Also: What is the resulting throughput of the code in "floating-point operations per cycle"? Don't count flw and fsw as floating-point operations.

1/9

Q1.B Scheduling VLIW Code, Software pipelining [11 points]
Rewrite the assembly code to leverage software pipelining. Do not loop unroll. Schedule VLIW instructions in the following table. You should show the loop prologue and epilogue in addition to the body. Use a clear method to distinguish between instructions of different loops (iterations). You can use a different ink color, underlining, a number in parenthesis, or anything else that is clear.
The below only shows two loops but a more complete solution has more. In that case, the body contains 9 instructions.

Inst	Int Op	Branch	Mem Op	FP/Int Op	FP Add	FP Mul
1	add x7, x5, x7		flw f1, 0(x1)			
2			lw x9, 0(x2)			
3	add x7, x5, x7			addi x1, x1, 4		
4			sw x7, 0(x1)			fmul f3, f1, f1
5			flw f1, 0(x1)	addi x1, x1, 4		
6						
7			lw x9, 0(x2)			
8	add x2, x2, x9		fsw f3, 0(x2)			fmul f3, f1, f1
9		bne x2, x5, loop				
10						
11						
12	add x2, x2, x9		fsw f3, 0(x2)			
13		bne x2, x5, loop				
14						
15						
16						
17						
18						
19						
20						
21						

Also: What is the resulting throughput of the code in “floating-point operations per cycle”?
2/13

Q1.C More Aggressive Loop Unrolling [3 points]
If we were to unroll the loop to four times, could this be done efficiently in this processor? If you could add a resource such as a functional unit, what would it be?

We can’t make the body smaller because the multiply unit is four cycles. The bottleneck remains the memory unit. Unrolling to four times would stress the one memory functional unit we have eve more. We should add at least one more.

Q1.D Cache Misses [3 points]
We made the assumption that loads and stores are predictable because we always hit in the cache. This is unrealistic. How can a VLIW processor such as the one we describe in this question respond to a cache miss?

It would have to stall until the cache miss is serviced and re-execute the same instruction. Stall and abort (with re-execute) are two ways of dealing with a miss.

Q1.E Variable Latency Functional Units [3 points]
In the processor of this question, assume that the floating point or integer functional unit has a variable latency depending on if it executes a functional point or integer operation. Ignoring structural hazards in the writeback stage, what problem does this create?

Since there is no hazard resolution in the hardware, the VLIW compiler needs to know the latency of each functional unit to know when each instruction will write to the register file, and thus what version of the register (old or new) subsequent instructions will see.

Q1.F Branches [4 points]
Assume that the branch resolution unit in this processor has a two-cycle latency, instead of one cycle as originally stated. Without prediction and without a branch delay slot, can we schedule an instruction the cycle after the branch (i.e., if the branch is in cycle 13, the question is for cycle 14)? With branch prediction, what is the challenge in case of mispeculation and what do we need to do about it in this processor?

No because we have not resolved the branch, therefore we do not know what the next instruction is. If we have branch prediction the challenge is not allowing instructions that were fetched as part of the mispeculation to change the state of the processor. In this processor, since all functional units have a latency of two cycles or more, it suffices to kill the mispredicted operations from the appropriate pipeline stages in the functional units. No operation would have changed the state of the processor by the time we resolve the branch.

Question 2: Vector Machines [30 points]

In this question, we will consider the following code written in C:

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

You can assume VLMAX = 32 (vector registers contain 32 elements). Also, unless otherwise specified, N = 32. The base address of array A is contained in scalar register rA, B in rB, and C in rC. For the rest of the machine, assume the following:

- 16 lanes
- one ALU per lane: 2 cycle latency
- one LD/ST unit per lane: 2 cycle latency. Fully pipelined that latches operands at the first pipeline stage.
- all functional units have dedicated read/write ports into the vector register file
- no dead time
- no support for chaining
- scalar instructions execute separately on a control processor (5-stage, in-order)

Q2.A Vector Memory Memory and Register [6 points]

Write the vector assembly code for the above C code first in a vector memory-memory code, and then vector register code. You can use as many registers as you need, as well as temporary memory locations (e.g., rT). Remember to set VLR. You can use any pseudo-assembly language that makes sense (‟add″, ‟mul″, etc).

Vector memory-memory

```
LI VLR 32
Mul rT, rB, rC
Add rA, rA, rT
```

Vector register

```
LI VLR 32
LV v1 0(rB)
LV v2 0(rC)
Mul v3, v2, v1
LV v4 0(rA)
Add v5, v3, v4
SV v5 0(rA)
```

Q2.B Compare [2 points]
Which of the two versions of the code from Q2.A is more stressful to memory (causes more memory accesses)? Also, which of the two versions provides more opportunities to exploit instruction-level parallelism?

Vector memory memory causes 6 memory accesses whereas register causes 4. Vector register provides more opportunities to exploit ILP because it has more independent instructions that can be re-ordered.

Q2.C Lane Tradeoffs [3 points]

Suppose we want to add two vector registers (add v1, v2, v3), followed by another addition to different registers (add v4, v5, v6). The next instruction after that uses a different functional unit. VLR=VLRMAX=32. What would you choose between an ALU with 8 lanes and 2 cycles dead time, and an ALU with 16 lanes and 8 cycles dead time?

The first case will need (32/8)*2 + 2 = 10 cycles. The second (32/16)*2 + 4 + 8 = 12 cycles. So the first option is faster.

Q2.D Vectorize [7 points]

How can the following code be vectorized? You can assume N=VLMAX. Clearly state any assumptions that you used for your answer for what the architecture provides, such as specialized instructions, registers, etc.

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

The indirection is not the issue here because we can load C to a register and use a load indirect instruction ("LVI") to load B, as long as the architecture supports that. For the if condition, we have to rely on a special instruction and predicate registers. We first load &(A[i]) into a register, and then &(A[i+1]) to a different register. We use the special instruction to set the predicate registers of A based on the values of A[i+i]. Then we perform the addition.

Q2.E Scheduling Vector Code, No Chaining [8 points]

Complete the pipeline diagram of the baseline vector processor running the following code. Assume no chaining.

```
LI VLR 32
LV v1 0(rA)
LV v2 0(rB)
LV v3 0(rC)
Mul v3, v3, v1
Add v3, v3, v2
LV v4 0(rD)
Add v4, v4, v3
Add v4, v4, v4
saddi r1, r1, 8 // Scalar add
SV v4 0(rA)
```

The following supplementary information explains the diagram:

- Scalar instructions execute in 5 cycles: fetch (**F**), decode (**D**), execute (**X**), memory (**M**), and writeback (**W**).
- Vector instructions should write back in program order.
- 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.

Name

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	41
LI	F	D	X	M	W																																				
LV		F	D	R	M1	M2	W																																		
					R	M1	M2	W																																	
LV			F	D	-	R	M1	M2	W																																
							R	M1	M2	W																															
LV				F	D	-	-	R	M1	M2	W																														
									R	M1	M2	W																													
MUL					F	D	-	-	-	-	-	-	R	X1	X2	W																									
														R	X1	X2	W																								
ADD						F	D	-	-	-	-	-	-	-	-	-	-	R	X1	X	W																				
																			R	X1	X2	W																			
LV							F	D	-	-	-	-	-	-	-	-	-	-	-	R	M	M	W																		
																					R	M	M	W																	
ADD								F	D	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	R	M	M	W													
																										R	M	M	W												
ADD									F	D	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	R	X1	X2	W								
																															R	X1	X2	W							
saddi											F	D	X	M	W																										
SV												F	D	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	R	M	M	W			

Name ___________________________________

Q2.F With Chaining [4 points]
In the same code, how many fewer cycles would be required if we added chaining?

In a RAW hazard, chaining lets the instruction that writes forward to the instruction that reads after the functional unit produces the result but before it writes to the register file. The above code, there are 5 RAW hazards between vector instructions: LV (v3) -> mul -> add -> add -> add -> SV. For each of those, chaining would save 2 cycles (the first R of the second instruction lines up with the first W of the first instruction). So 10 cycles total.

Name ________________________________

Question 3: Multithreading [23 points]

In this question, we will consider multithreading executing in a single-issue, in-order, multithreaded processor that supports a variable number of threads (as individual questions specify). The code we will be working with is:

```
loop:
  lw x1, 0(x3)
  lw x2, 0(x4)
  add x1, x2, x1
  sw x1, 0(x3)
  addi x3, x3, 4
  addi x4, x4, 4
  bne x1, x2, loop
```

The processor has the following functional units:

- Memory operation (load/store), 4 cycles latency (fully pipelined).
- integer add, 1 cycle latency.
- Floating point add and multiply unit, 3 cycles latency (fully pipelined).
- branch, 1 cycle latency.

The processor has a cache which has a 100% hit rate (4 cycle latency is with a cache hit). 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. Throughout this question there is no communication between threads, and the loads and stores of each thread are to different addresses.

Name ______________________________________

Q3.A Scheduling [5 points]

First consider that the processor can handle two threads at the same time. You have a choice between round-robin scheduling where if the thread that was going to be selected in not ready a bubble is inserted instead, and dynamic hardware scheduling where the processor at every cycle picks a thread that is ready (in a round-robin manner if both threads are ready).

With the code and assumptions given, will this make a different in performance (i.e., how quickly each thread completes a pre-defined large number of iterations)?

Does your answer change if we have an ideal cache and with a 10% chance a load/store misses at the cache at imposes a 50-cycle penalty?

With the assumptions given, each thread has the same hazards and goes through the same instructions with the same latencies, so when one stalls (e.g., due to RAW) the other stalls too. So the two scheduling policies will not make a difference. However, if we now risk having a cache miss, one thread may miss in the cache and the other may not. In that case, the ideal scheduler will provide more cycles to the thread that is ready. With round-robin, the time for one thread to complete does not depend on other threads, except for cache misses and communication.

Name ______________________________

Q3.C Number of Threads [10 points]

Now lets consider the same single-issue in-order processor as originally described, with round-robin scheduling and a 100% cache hit rate, but a variable number of threads that it can support. What is the minimum number of threads needed to fully utilize the processor, assuming you reschedule the assembly as necessary to minimize the number of threads? Also, assume an infinite number of registers.
With the same instruction sequence that you used, what is the number of threads if all adds are floating point adds.

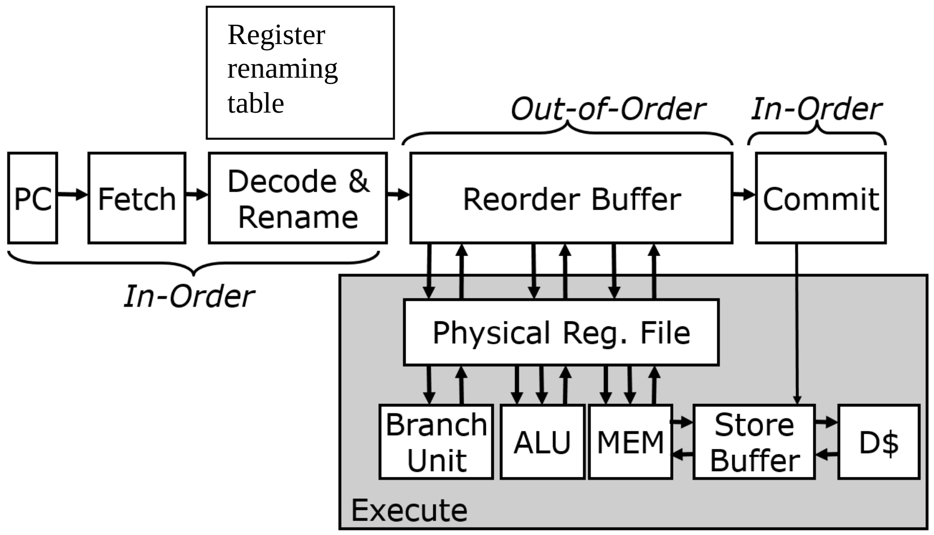
```
loop:
   lw x1, 0(x3)
  lw x2, 0(x4)
  addi x4, x4, 4  // This instruction was reordered
  add x1, x2, x1
  sw x1, 0(x3)
  addi x3, x3, 4
  bne x1, x2, loop
```

The first load is pushed to the memory in cycle 1. It writes back the result at the end of cycle 4. In between, instructions 2 and 3 issued and are able to execute, but in cycle 4 there was a bubble because instruction 4 issued but can't execute. With the latencies and instructions provided, there are no more bubbles. Therefore, two threads are needed to keep this processor busy.

If adds are floating point adds (only one add will be affected), their latency is now 3 cycles instead of 1. Now there are two bubbles between instructions 4 and 5 (RAW). In the first bubble, instruction 5 issues and in the next bubble instruction 6 issues. Neither can execute. Instruction 6 can't execute because of the WAR hazard. To cover these two bubbles we need three threads.

Name ________________________________

Q3.C Pipeline [7 points]



Above you can see a diagram of an in-order issue, out-of-order execute processor with a physical register file that performs register renaming. This diagram is for a single-threaded processor. If we wanted to make this processor two-way multithreaded with round-robin scheduling, which block(s) do we *have to* duplicate, and which block(s) we *should* probably make larger in order to avoid having them be a performance bottleneck? Do not make the processor superscalar. Explain why for each block.

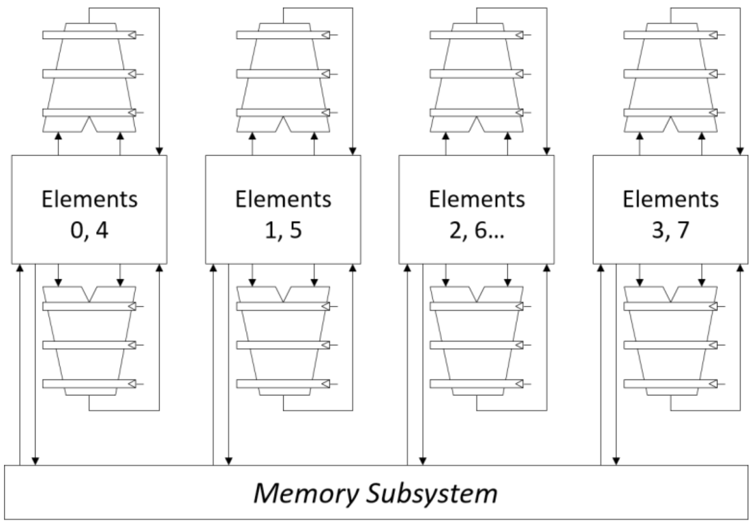
We have to duplicate the PC register to two PC registers, one for each thread. We also have to duplicate the register renaming table because each thread has its own set of architectural registers. Those are the only blocks we have to duplicate to ensure functional correctness in this processor. We should make the physical register file larger because now we have two threads looking to store data in registers. We should also make the reorder buffer (ROB) larger because now we have instructions from two threads that are independent. Therefore, we need a larger ROB to find instruction-level parallelism. Finally, we can also use a deeper store buffer because there will be more pending (completed but not committed) stores, and more functional units.

Name ______________________________

Question 4: Potpourri [16 points]

Q4.A Vector Processors [5 points]

Assume a register vector processor (no vector memory-memory operations) with vector registers that can hold 8 elements and has 4 lanes in its integer functional unit (shown below). The alternative processor is a similar vector processor that has vector registers that can hold 16 elements but only 2 lanes in its integer functional unit. Both architectures have an infinite number of registers. If we want to add 64 array elements (code shown below), which of the two will complete faster? Assume loads and stores take one cycle, and load instructions have to be before adds, and adds have to be before stores.



```
# C code
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];
```

The first processor needs 64/8 = 8 loads and 8 stores, so 16 cycles. The addition takes 64/4 =16 cycles. So 32 cycles overall. The second processor needs 64/16 = 8 (4 loads and 4 stores). The addition takes 64/2 = 32 cycles. So the first processor is faster.

Name ___________________________________

Q4.B Multithreaded Processors [4 points]

Assume a multithreaded processor with a cache, 64 integer registers, and in-order issue and commit. We have four threads, and we want to compare statically interleaving them cycle-by-cycle by having thread 1 issue an instruction in cycle 1, 5, 9, etc (option 1), against option 2 where we statically execute 512 cycles of thread 1, then 512 of thread 2, etc. Name one advantage and one disadvantage of option 2 compared to option 1.

One disadvantage is that if thread 1 blocks, the rest of its cycles before the next thread executes are wasted. One advantage is that a thread running for a number of cycles consecutively warms up the cache and brings its own data in, before other threads have the change to evict them due to capacity or conflict. Other answers are also possible.

Q4.C VLIW Processors [3 points]

Both VLIW and out-of-order superscalar processors exploit instruction-level parallelism. What is the motivation to choose VLIW processors instead of out-of-order superscalar? How does this affect hardware and compiler complexity?

VLIW can issue multiple operations per cycle with simple hardware. Out-of-order superscalar need complex control logic. Independent operations are scheduled statically by the compiler into the same VLIW instruction. The compiler’s job becomes more complex.

Name ____________________________________

Q4.D Multithreaded with VLIW and Vector [4 points]
Suppose we have two threads (which may have different instructions), and we want to merge their instructions into one (mega-)thread. We have the option of running this (mega-)thread on a VLIW and a vector processor. Give one reason why we would choose VLIW (and what assumption or condition that depends on), versus a reason we would choose the vector processor (and what assumption or condition that depends on).

This has to do with how different the instruction streams are. The advantage of VLIW is that it provides more flexibility such that if threads have different operations each cycle, VLIW can mix different kinds of operations in the same cycle. On the other hand, if threads are perfectly synchronized (unlikely but possible), the vector multithreaded processor will increase throughput because it can do more operations of the same kind in the same cycle.

Appendix

This is the code and other information for question 1. You may detach the appendix.

```
loop:
  flw  f1, 0(x1)
  lw   x9, 0(x2)
  fmul f3, f1, f1
  add  x7, x5, x7
  sw   x7, 0(x1)
  fsw  f3, 0(x2)
  addi x1, x1, 4
  add  x2, x2, x9
  bne  x2, x5, loop
```

This code will run on a VLIW machine with the following instructions format:

Int Op	Branch	Mem Op	FP or Int Op	FP Add	FP Mul

Our machine has six execution units (in order from left to right in the instruction above). All execution units are fully pipelined and latch their operands in the first stage.

- One integer unit, latency two cycles.
- One branch unit, latency one cycle.
- One memory unit, latency three cycles, each unit can perform both loads and stores. You can assume a 100% cache hit rate (i.e., no variability).
- One functional unit that can deal with integer or floating point operations of any kind (additions and multiplications). Latency five cycles.
- One floating point add unit. Latency three cycles.
- One floating point multiply unit. Latency four cycles.

This 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). Functional units write the register file at the end of their last pipeline stage, and there are no data forwarding or stalls. I.e., a functional unit that requires 4 cycles and starts an operation in cycle 1 will have its result be visible at the beginning of cycle 6 (writes at the end of cycle 5).