Computer System Architecture
6.823 Quiz #4
May 20, 2021

Name: _____

90 Minutes
18 Pages

Notes:

- Not all questions are equally hard. Look over the whole quiz and budget your time carefully.
- Please state any assumptions you make, and show your work.
- Please write your answers by hand, on paper or a tablet.
- Please email all 17 pages of questions with your answers, including this cover page. Alternatively, you may email scans (or photographs) of separate sheets of paper. Emails should be sent to 6823-staff@csail.mit.edu
- Do not discuss a quiz's contents with students who have not yet taken the quiz.
- Please sign the following statement before starting the quiz. If you are emailing separate sheets of paper, copy the statement onto the first page and sign it.

I certify that I will start and finish the quiz on time, and that
I will not give or receive unauthorized help on this quiz.

Sign here: _____

Part A	_____	30 Points
Part B	_____	25 Points
Part C	_____	30 Points
Part D	_____	15 Points
TOTAL	**_____**	**100 Points**

Part A: VLIW (30 points)

Consider the following C code, which operates on two arrays `A` and `B` that contain 32-bit floating-point elements:

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

The following is the equivalent MIPS assembly code, where:

- `f3` contains the value of variable `x`
- `r1` and `r2` are initialized to the addresses of `A[0]`, `B[0]` respectively at the beginning of the loop
- `r3` contains the address of `A[N]`

```
loop:
     ld f0, 0(r1)
     fadd f1, f0, f3
     fmul f2, f0, f1
     st f2, 0(r2)
     addi r1, r1, 4
     addi r2, r2, 4
     bne r1, r3, loop
```

We want to schedule this code on a VLIW processor that issues one instruction per cycle. Each VLIW instruction encodes operations for 6 functional units:

- Two integer ALU units (also used for branches) with a 1-instruction latency (i.e., the result of the operation can be used by a dependent instruction 1 cycle later).
- Two memory units that can be used for both loads and stores. The processor does not have a data cache, so all memory accesses have a fixed 2-instruction latency.
- One floating point adder with 3-instruction latency.
- One floating point multiplier with 3-instruction latency.

Question 1 (5 points)

Schedule operations into VLIW instructions by filling in the following table. Your solution should only contain one iteration of the loop (do not unroll). Your schedule take as few cycles as possible to earn full credit. You can leave NOPs as blanks.

(For those writing on a separate piece of paper: Simply indicate any VLIW instruction with no operations with a solid line across the instruction. For instance, if there is are two empty VLIW instructions, you would write the following)

Label	Integer	Integer	Memory	Memory	FP add	FP mul
			`ld rx, 0(ry)`			
					`fadd ra,rb,rc`	

Write down your solution here:

Label	Integer	Integer	Memory	Memory	FP add	FP mul

Question 2 (1 points)

What number of floating point operations per cycle (FLOPs/cycle) does your schedule in Question 1 achieve?

Question 3 (3 points)

Ben now considers loop unrolling to improve performance. What is the minimum factor by which the loop must be unrolled so that, in steady state, every instruction performs at least one memory or floating point operation? Whatever degree of unrolling you choose, assume that it divides the number of loop iterations exactly.

Question 4 (6 points)

Ben now wants to apply software pipelining to this loop. With proper application of unrolling and software pipelining, Ben achieves the ideal peak throughput of 2 FLOPs/cycle. How many VLIW *instructions* should the body of the steady-state software-pipelined loop contain to achieve this throughput (excluding the prologue and epilogue)? You need not write down the whole loop body, but explain your reasoning for the number of instructions.

Hint: Note that the `fmul` *instruction has a data dependence on both the* `ld` *and the* `fadd`*.*

Ben now introduces a *direct-mapped data cache with 8 sets and 64 bytes per line* to his processor. This cache makes loads have variable latency: 1 cycle if it hits the cache, and 2 cycles otherwise. Since VLIW processors expose fixed instructions latencies to software, benefitting from the lower latency on cache hits requires some software changes.

To this end, Ben adds a **memory latency register (MLR)** to his processor. As we saw in lecture, the MLR (featured in Cydrome's Cydra-5) is a programmatically writable register that contains the desired latency of loads, in VLIW instructions. The programmer sets the MLR with following instruction:

```
setmlr rs  ;; Set the MLR to the value of Reg[rs]
```

The processor is modified to ensure that loads always take the latency specified by the MLR. If the load produces the result earlier than the MLR latency (e.g., if MLR is set to 2 instructions but the cache replies in 1 cycle), the processor temporarily buffers the data to match the longer latency. If the operation produces it later than expected (e.g., if the MLR is set to 1 instruction but memory replies in 2 cycles), the processor is stalled until the data is available.

Question 5 (5 points)

Recall that array `A` contains 32-bit floating point elements. Assume that the data cache is initially empty. To what value should Ben set the MLR to maintain the peak throughput in Question 4 with software pipelining and loop unrolling? Explain briefly.

Question 6 (5 points)

Ben now adds a direct-mapped *instruction cache* with 2 sets and 48 bytes per line. To what value should Ben set the MLR such that the body of the software-pipelined loop can fit in the instruction cache? Assume instructions are aligned properly. Explain briefly.

Hint: Each VLIW instruction is 32bits× 6 = 24 bytes.

Question 7 (4 points)

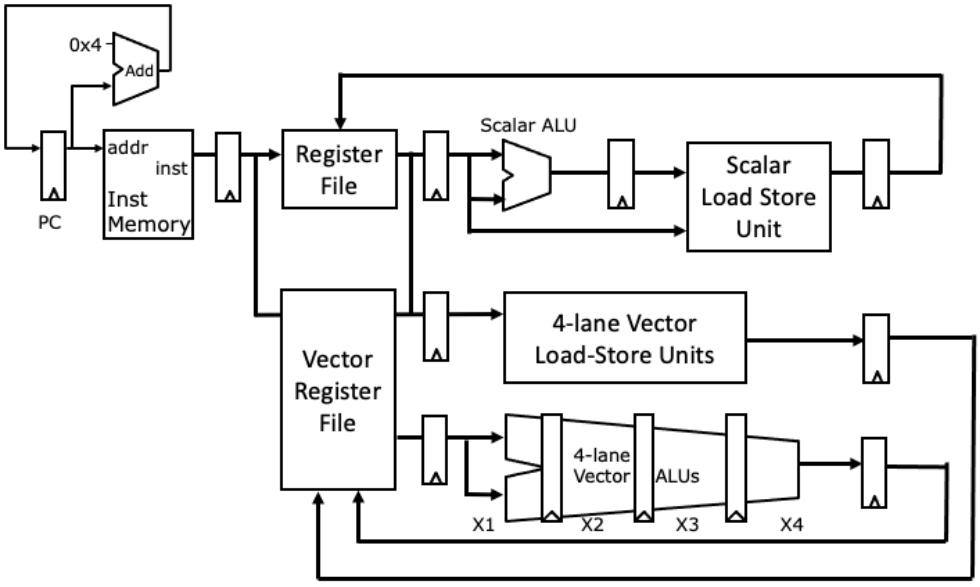
So far we have ignored the performance impact of instruction cache misses. Suppose that instruction cache misses occur a 1-cycle additional latency in fetching the instruction. If Ben uses the instruction cache with the same parameters given in Question 6, to what value should he set the MLR to maximize the performance (FLOPs/cycle) of his code?

Part B: Vector Processors and GPUs (25 points)

For the following questions, we will explore how different vector architecture features can affect the performance of various vectorized codes. The following describes the baseline vector processor with support for vector masks:

- Single-issue, in-order execution.
- Scalar instructions execute on a 5-stage, fully-bypassed pipeline.
- 32 vector registers named v0 through v31. Each vector register holds **16 elements**. The register files have enough ports to keep all lanes busy.
- **Four** vector lanes, each with one ALU and one load-store unit. Both units are fully-pipelined and can process vector elements from independent instructions.
- No support for vector chaining.
- The ALUs have a **4-cycle latency** (4 for FP add/mul and 1 for writeback).
- The vector memory system has no cache and consists of 16 banks, with 4-byte word interleaving (`0x0` maps to bank 0, `0x4` to bank 1, etc.). Memory access latency is **4 cycles** (4 cycles for access, 1 for writeback) with a 2-cycle bank busy time (additional cycles between accesses to the same bank). A vector lane's load-store unit stalls if its required bank is busy.
- Vector instructions are maskable, but each lane always processes all its vector elements and turns off writeback for the masked ones.

This schematic shows a simplified view of the processor:



The processor can issue a single (scalar or vector) instruction per cycle. Once it issues, a vector instruction uses either all lanes' ALUs or all lanes' load-store units for as many cycles as needed to produce all of its results. A vector load or store can execute in parallel with independent operations that use the vector ALUs, and vector operations can execute in parallel with scalar operations. If a vector instruction depends on the result of a prior instruction, it stalls until the prior instruction finishes writing back **all** of its results. The processor implements MIPS plus the following vector instructions.

Instruction	Meaning
`setvlr Rs`	Set vector length register (VLR) to the value in Rs
`lv Vt, Rs, Stride`	Load vector register Vt starting at address in Rs, with stride immediate
`sv Vt, Rs, Stride`	Store vector register Vt starting at address in Rs, with stride immediate
`add.vv Vd, Vs, Vt`	Add elements in Vs, Vt, and store result in Vd
`mul.vv Vd, Vs, Vt`	Multiply elements in Vs, Vt, and store result in Vd
`add.vs Vd, Vs, Rt`	Add Rt to each element in Vs, and store result in Vd
`mul.vs Vd, Vs, Rt`	Multiply each element in Vs by Rt, and store result in Vd
`s--.vs Vd, Rs`	Compare the elements (eq, ne, gt, lt, ge, le) in Vd and Rs. *For each element, if the condition is true, set the corresponding bit of the vector mask register to 1. If the condition is false, set the corresponding bit of the vector mask register to 0.*
`cvm`	Set all elements in vector mask register to 1.

The following set of questions give a short vector assembly sequence and ask the performance impact of various microarchitectural feature changes. Assume that the vector length register is set to 16 before the code segment, and all register values used by the vector instructions are initially available. For full credit, **explain your answer** and clearly state your assumptions.

Question 1 (5 points)

Consider the following vector assembly:

```
lv v0, r1, 1       ;; Load with stride of 1
mul.vv v1, v1, v2
add.vv v2, v2, v3
mul.vv v3, v3, v4
add.vv v4, v4, v0
```

a) Does doubling the number of vector lanes increase the performance of this code?

b) Suppose we now add support for chaining. With chaining, a vector instruction that depends on a previous instruction can start execution if the first set of elements it processes is already written to the vector register file (assume there's no bypassing from the writeback stage). Does chaining help increase the performance of this code?

Question 2 (5 points)

Consider the following vector assembly with a load that has a stride of 16:

```
lv v0, r1, 16       ;; Load with stride of 16
mul.vv v1, v0, v1
```

a) Does doubling the number of vector lanes help decrease the latency of the `lv` instruction?

b) Does adding support for chaining (with the same implementation as described in Question 1) improve the performance of this code?

Question 3 (5 points)

Consider the following vector assembly that uses vector masks:

```
lv v0, r1, 1       ;; Load with stride of 1
sgt.vs v0, r0
add.vv v1, v1, v0
```

Does the performance of this code differ based on the values loaded by `lv`?

Ben Bitdiddle wants to run his C code on a GPU with the following features:

- 4 threads per warp that share the same PC and thus execute the same instruction in lockstep.
- The GPU has 4 lanes, with each lane having one ALU and one load-store unit that are fully pipelined, with a latency of 16 cycles.
- Each warp has a stack of masks to handle branch divergence. Each mask has a bit for each thread. Each thread looks at its corresponding bit of the mask at the top of the stack. If a mask bit is zero, the corresponding thread does not execute the current instruction.

The following is Ben's code:

```
    for (int i = 0; i < N; i++) {
I1      if (A[i] > 0) {
            C[i] = A[i] - B[i];
I2      } else {
            C[i] = A[i] + B[i];
I3      }
    }
```

Assume that `A[i]` is positive if and only if `i` is divisible by `4` (i.e., `A[0]`, `A[4]`, ...).

Question 3 (5 points)

Ben translates the code to run on the GPU. His code uses N threads (grouped in N/4 warps), and each thread executes the a single loop iteration (shown in grey background). Thread `i` executes iteration `i`. Assume that the warp mask is initially all set, and that N is a multiple of 4. Describe the state of the per-warp mask stack after each point `I1`, `I2`, and `I3` as specified in the above code.

Question 4 (5 points)

What is the minimum number of warps needed to achieve the highest pipeline utilization? Note that the load-to-use latency is 16 cycles (i.e., you can issue a dependent operation 16 cycles later), and different warps execute in lockstep.

Part C: Transactional Memory (30 points)

In this part we will analyze the operation of different hardware TM (HTM) designs, and the concurrency they achieve for different transaction schedules on a multicore system. For any HTM design, the memory system dynamically tracks the set of addresses read or written by each transaction (i.e., its read set and write set) as accesses are performed.

Consider the following two HTM designs:

- **Eager & Pessimistic HTM** uses eager version management and pessimistic conflict detection. For every transactional load, the memory system checks whether this load reads an address in the write set of any other transaction, and declares a conflict if so. For every transactional store, the memory system checks whether this store writes an address in the read set or write set of any other transaction, and declares a conflict if so. Upon a conflict, the *requester stalls* and waits until all conflicting transactions abort or commit. Assume that the requester immediately resumes execution once all conflicting transactions have aborted or committed.

- **Lazy & Optimistic HTM** uses lazy version management and optimistic conflict detection. Conflicts are detected when a transaction attempts to commit. The finished transaction validates its write-set with coherence actions. If any of its writes appear in the read- or write-set of other transactions in the system, a conflict is declared, and the *committer wins*, aborting any other conflicting transactions. Assume that the aborted transaction immediately re-executes from the beginning at the same cycle.

In the following questions, for timing, assume conflict detection and coherence happen in the same cycle a memory access executes. Note that we denote a transaction reading from or writing to a memory location `A` by `Rd A` and `Wr A`, respectively.

Question 1 (10 points)

Consider the following scenario, where two transactions X and Y begin at cycles 5 and 0. The following table shows how the transactions would proceed **in the absence of conflict detection**:

Cycle	0	5	10	15	20	25	30	35	40	45
Transaction X		Begin		Wr A			Rd B	Wr B		End
Transaction Y	Begin	Rd A			Wr B	Rd A		End		

a) Is the above execution schedule serializable in the absence of conflict detection? If so, what is the serialization order?

b) At which cycle is a conflict detected between the two transactions for the two HTM systems? If you think that no conflict detection will occur, write "No Conflict" as your answer.

- Eager & Pessimistic:

- Lazy & Optimistic:

c) At which cycle will both transactions have finished execution with the two HTM systems?

- Eager & Pessimistic:

- Lazy & Optimistic:

Question 2 (10 points)

Consider the following scenari,o where two transactions X and Y begin at cycles 0 and 5. The following table shows how the transactions would proceed **in the absence of conflict detection**:

Cycle	0	5	10	15	20	25	30	35	40	45
Transaction X	Begin		Rd B			Wr B			Rd A	End
Transaction Y		Begin		Rd B	Rd A		Wr A	End		

a) Is the above execution schedule serializable in the absence of conflict detection? If so, what is the serialization order?

b) At which cycle is a conflict detected between the two transactions for the two HTM systems? If you think that no conflict detection will occur, write "No Conflict" as your answer.

- Eager & Pessimistic:

- Lazy & Optimistic:

c) At which cycle will both transactions have finished execution with the two HTM systems?

- Eager & Pessimistic:

- Lazy & Optimistic:

Question 3 (10 points)

Consider the following scenario, where three transactions X, Y, and Z begin at cycle 0. The following table shows how the transactions would proceed **in the absence of conflict detection**:

Cycle	0	5	10	15	20	25	30	35	40	45
Transaction X	Begin	Rd A	Wr A					End		
Transaction Y	Begin			Rd A	Wr A				End	
Transaction Z	Begin					Rd A		Wr A		End

a) Is the above execution schedule serializable in the absence of conflict detection? If so, what is the serialization order?

b) At which cycles is the *first* conflict detected between any two transactions for the two HTM systems? If you think that no conflict detection will occur, write "No Conflict" as your answer.

- Eager & Pessimistic:

- Lazy & Optimistic:

c) At which cycle will all transactions have finished execution with the two HTM systems?

- Eager & Pessimistic:

- Lazy & Optimistic:

Part D: Security & Virtualization (15 points)

Question 1 (6 points)

For the following questions, answer True or False. (2 points each)

a) Hardware support is necessary to implement virtual machines.

b) Suppose we have a virtual machine running on top of our Virtual Machine Monitor (VMM) with no support for shadow page tables. If the guest OS uses M-level page tables and the host OS uses N-level page tables, a TLB miss will incur (M+1)(N+1)-1 page table entry accesses.

c) In an out-of-order processor that supports simultaneous multithreading, the physical register file can be a potential side channel.

Question 2 (4 points)

Recall, for an architecture to be effectively virtualizable (by Popek and Goldberg's rules), all sensitive instructions should be privileged so the VMM can emulate them through traps. This is called *classical virtualization*.

Are the following instructions classically virtualizable? Briefly explain why or why not.

```
a) sptbr rs:
      if in supervisor mode:
            # Move the content of GPR rs to register ptbr,
            # which holds the physical address of the
            # root (level-1) page table
            ptbr ← Reg[rs]
      else:
            set supervisor bit to 1, jump to exception handler
```

```
b) mret rs:
      if in supervisor mode:
            set supervisor bit to 0, enable interrupts
            pc ← Reg[rs]
      else:
            # Treat mret as a normal jump
            pc ← Reg[rs]
```

```
c) invlpg rs1, rs2:
      invalidate the TLB entry for the virtual address = Reg[rs1]
      and the address space id = Reg[rs2]
      # The TLB uses address space ids to avoid flushing on context
      # switches. Since the TLB is microarchitectural state, the ISA
      # designers made invlpg work identically in user and supervisor
      # mode
```

Question 3 (5 points)

Consider an out-of-order processor with support for simultaneous multithreading. The processor has a single, unpipelined floating point divider that takes N cycles when the numerator has N bits. The processor also has support for very precise timers.

Consider the following kernel C code:

```
float secret, x, a, b;
...

if (x < 1024)
      secret = secret / x;
float a = a / b;
```

Imagine a scenario where the attacker invokes this kernel code (e.g., through a system call) with small values of `x` to prime the branch to be not taken, and provides a value of `x` larger than 1024 when he wishes to extract information about `secret`. Can this code be used as a transmitter to leak information about the contents of `secret` to the attacker under speculative execution? If so, how much detail could the attacker reveal about `secret`? Explain your reasoning.