# Computer System Architecture
# 6.823 Quiz #4
# May 12, 2020

## SOLUTIONS

Name: _____

90 Minutes
13 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 13 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 | _____ | 19 Points |
| Part B | _____ | 32 Points |
| Part C | _____ | 24 Points |
| **TOTAL** | _____ | **75 Points** |

# Part A: Scalar vs. VLIW Processors (19 points)

Ben Bitdiddle wants to examine the execution of the following C loop on different machines. This code operates on two arrays of length N, containing 32-bit floating point numbers:

```
for (i = 0; i < N; i++)
   A[i] = A[i] * (B[i] + 1.0);
```

Ben starts by compiling the loop to run on a scalar machine. The compiler generates the following instructions for the body of the loop:

```
      ;; Initial values:
      ;;  f1 := 1.0
      ;;  r1 := &A[0] and r2 = &B[0]
      ;;  r3 := &A[N] (first address after vector A)
I1:  loop: ld f0, 0(r2)       ;; Load B[i]
I2:        ld f2, 0(r1)       ;; Load A[i]
I3:        fadd f3, f0, f1
I4:        addi r1, r1, 4
I5:        fmul f4, f2, f3
I6:        addi r2, r2, 4
I7:        st f4, -4(r1)      ;; Store A[i]
I8:        bne r1, r3, loop
```

## *Question 1 (6 points)*

The code above runs on an in-order, pipelined, single-issue scalar processor with perfect branch prediction and full bypassing. ALU (integer) operations have a 1-cycle latency (so, thanks to bypassing, consecutive dependent ALU operations execute without stalling), loads have a 2-cycle latency, and floating-point operations have a 3-cycle latency.

(a) How many cycles will the processor stall per loop iteration? Briefly explain your answer.

No stalls for the I1→I3 or I2→I5 load-use dependencies, as the compiler has scheduled these instructions far enough apart.
I3→I5 and I5→I7 each require one stall to wait for a floating-point operation.
Total: 2 stalls

(b) How many floating-point arithmetic operations per cycle will the processor perform on average in steady state?

8 instructions + 2 stalls = 10 cycles per iteration, so with 2 floating point operations this gives: 2/10 = **1/5 = 0.2 FLOPs/cycle**

## *Question 2 (4 points)*

If you applied unrolling to the loop, what is the minimum unrolling factor needed to remove all stalls in steady-state execution? The unrolling factor is the total number of copies of code you end up with for the computation in the loop. Briefly explain your answer.

Unrolling by **2** is sufficient. We could simply interleave instructions from two iterations, which works since the original loop never required two stalls for a single instruction. Although not required, for this particular loop, we can also reduce the bookkeeping instructions and an unrolling factor of 2 is still sufficient. The following code with only two additions (which is the minimum possible) still has zero stalls:

```
I1:  loop: ld f0, 0(r2)       ;; Load B[i]
I2:        ld f5, 4(r2)       ;; Load B[i+1]
I3:        ld f2, 0(r1)       ;; Load A[i]
I4:        ld f6, 4(r1)       ;; Load A[i+1]
I5:        fadd f3, f0, f1
I6:        fadd f7, f5, f1
I7:        addi r1, r1, 8
I8:        fmul f4, f2, f3
I9:        fmul f8, f6, f7
I10:       addi r2, r2, 8
I11:       st f4, -8(r1)      ;; Store A[i]
I12:       st f8, -4(r1)      ;; Store A[i+1]
I13:       bne r1, r3, loop
```

## *Question 3 (9 points)*

Ben now considers a VLIW machine, where each instruction has slots for up to three operations:

- One (integer) ALU or branch operation (which always completes in a single cycle)
- One memory operation (which takes 2 cycles, as in the in-order processor)
- One floating point operation (which takes 3 cycles, as in the in-order processor)

The compiler must generate no-ops to stall the machine for data dependencies. Ben observes a compiler for this VLIW machine generates the following VLIW code when loop unrolling and software pipelining are disabled:

| Inst. | ALU/Branch Unit | Memory Unit | Floating Point Unit |
|---|---|---|---|
| 1 loop: | `addi r1, r1, 4` | `ld f0, 0(r2)` | nop |
| 2 | `addi r2, r2, 4` | `ld f2, -4(r1)` | nop |
| 3 | nop | nop | `fadd f3, f0, f1` |
| 4 | nop | nop | nop |
| 5 | nop | nop | nop |
| 6 | nop | nop | `fmul f4, f2, f3` |
| 7 | nop | nop | nop |
| 8 | nop | nop | nop |
| 9 | `bne r1, r3, loop` | `st f4, -4(r1)` | nop |

SOLUTIONS

(a) Ben is disappointed that this VLIW code takes nine cycles per iteration and has some instructions consisting entirely of no-ops. To address this, Ben considers loop unrolling. What is the minimum factor by which the loop must be unrolled so every instruction in steady state performs at least one **memory or floating point** operation? Whatever degree of unrolling you choose, assume it divides the number of loop iterations exactly. Identify which data dependencies are most critical in determining your answer.

Unrolling by **3** is sufficient. The most critical dependencies are fadd→fmul→st, due to the long-latency floating-point ops which take 3 cycles. Unrolling by a factor of 2 does not give enough ops to fill the cycles (since we aren't satisfied to fill a cycle with just an integer ALU op). Once you unroll by 3, matching the longest latency, you can interleave ops from three fadd→fmul→st chains to put at least one op in each cycle. For example, you could have extended and filled in the table on the previous page to get this schedule for unrolling by 3:

| Inst. | ALU/Branch Unit | Memory Unit | Floating Point Unit |
|---|---|---|---|
| 1 | | `ld f0, 0(r2)` | |
| 2 | | `ld f5, 4(r2)` | |
| 3 | | `ld f9, 8(r2)` | `fadd f3, f0, f1` |
| 4 | `addi r2, r2, 12` | `ld f2, 0(r1)` | `fadd f7, f5, f1` |
| 5 | | `ld f6, 4(r1)` | `fadd f11, f9, f1` |
| 6 | | `ld f10, 8(r1)` | `fmul f4, f2, f3` |
| 7 | `addi r1, r1, 12` | | `fmul f8, f6, f7` |
| 8 | | | `fmul f12, f10, f11` |
| 9 | | `st f4, -12(r1)` | |
| 10 | | `st f8, -8(r1)` | |
| 11 | `bne r1, r3, loop` | `st f12, -4(r1)` | |

(b) Assume the VLIW processor has appropriate support for software pipelining (e.g., a rotating register file). If you can overlap any number of iterations and apply software pipelining, what is the maximum achievable throughput, in floating-point arithmetic operations per cycle?

The loop performs 3 memory operations for every 2 floating point operations. We can issue at most 1 memory op per cycle, so the peak throughput is **2/3 = 0.67 FLOPs/cycle**.

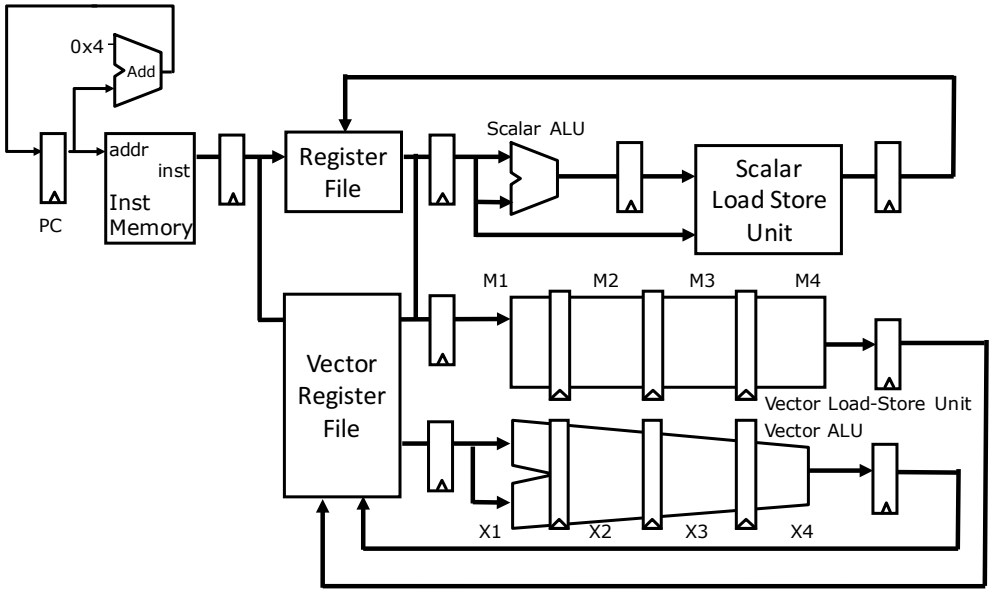
(Addendum: Software pipelining and its associated hardware support is not actually needed to achieve this ideal throughput: simply unrolling by a larger factor can result in enough memory ops to keep the memory unit fully utilized.)

# Part B: Vector Processors and GPUs (32 points)

Ben moves on to consider using a vector machine. Ben's vector processor has these features:

- 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 **32 floating-point elements**. The register files have enough ports to keep all lanes busy.
- **Four** vector lanes, each with one floating-point ALU and one load-store unit. Vector loads and arithmetic take **four** cycles to produce results and **one** cycle for writeback.
- No support for vector chaining.

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. Vector units are pipelined, so independent operations can be issued in sequence such that each stage in each vector unit operates on different values every cycle. 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` | Load vector register Vt starting at address in rs |
| `sv Vt, rs` | Store vector register Vt starting at address in rs |
| `fadd.vv Vd, Vs, Vt` | Add elements in Vs, Vt, and store result in Vd |
| `fmul.vv Vd, Vs, Vt` | Multiply elements in Vs, Vt, and store result in Vd |
| `fadd.vs Vd, Vs, ft` | Add floating-point scalar ft to each element in Vs, store result in Vd |
| `fmul.vs Vd, Vs, ft` | Multiply each element in Vs by scalar ft, and store result in Vd |

## *Question 1 (8 points)*

Ben wants to analyze the performance of this vector processor on the same loop as in Part A:

```
for (i = 0; i < N; i++)
   A[i] = A[i] * (B[i] + 1.0);
```

For this part, assume that N is a multiple of 32. For your convenience, we've reproduced the original scalar assembly code for this loop:

```
      ;; Initial values:
      ;;  f1 := 1.0
      ;;  r1 := &A[0] and r2 = &B[0]
      ;;  r3 := &A[N] (first address after vector A)
I1:  loop: ld f0, 0(r2)       ;; Load B[i]
I2:        ld f2, 0(r1)       ;; Load A[i]
I3:        fadd f3, f0, f1
I4:        addi r1, r1, 4
I5:        fmul f4, f2, f3
I6:        addi r2, r2, 4
I7:        st f4, -4(r1)      ;; Store A[i]
I8:        bne r1, r3, loop
```

(a) Provide equivalent vector code. For full credit, your code should execute as quickly as possible while using no more than two vector ALU instructions.

```
      addi r20, r0, 32      ;; set r20 to 32
      setvlr r20            ;; use all 32 vector elements
loop: lv v0, r2
      lv v1, r1
      fadd.vs v2, v0, f1
      fmul.vv v3, v1, v2
      sv v3, r1
      addi r1, r1, 128
      addi r2, r2, 128
      bne r1, r3 loop
```

(If you assume that the sv instruction allows an immediate offset/displacement, then the two addi instructions could also be placed earlier. In this case, they can be placed almost anywhere without affecting performance, since they will be issued while vector operations are being executed in parallel on the vector lanes.)

(b) How many cycles are required in steady state for each vectorized loop iteration (which corresponds to 32 iterations of the original loop)?

47 cycles

Each vector instruction takes 8 cycles (32 elements / 4 lanes) to get all its operands into the lanes. There's a 13-cycle latency between dependent vector instructions (8 cycles to get all elements into lanes + 4 cycles for FU latency + 1 cycle for writeback). The following table shows how the latencies add up to 47 cycles per iteration. The scalar instructions can be issued while the vector lanes are busy and are not shown.

| Instruction | Iteration i | Iteration (i+1) |
|---|---|---|
| `lv` | 0 | 47 (must wait for `sv` to finish using load/store units) |
| `lv` | 8 | |
| `fadd.vs` | 13 (depends on first `lv`) | |
| `fmul.vv` | 26 (depends on `fadd.vs`) | |
| `sv` | 39 (depends on `fmul.vv`) | |

## *Question 2 (8 points)*

Suppose we add chaining support to the processor. With chaining, a vector instruction that depends on a previous instruction can start execution if the first set of elements it processes is either already written to the vector register file or is available in the writeback stage (we add the required bypass paths).

(a) How would you reorder the vector instructions in your vectorized loop from the previous question to improve performance with chaining?

Put the `fadd.vs` between the two `lv` instructions (because the `fadd.vs` can use the vector ALUs in parallel with the use of the vector load/store units by the loads).

(b) What is the resulting throughput in floating-point arithmetic operations per cycle in steady state?

Dependent vector instruction issue 4 cycles after the previous one, and now the vector load/store units are maximally utilized, with a vector load or store issuing every 8 cycles.

So, each iteration of the vectorized loop takes 24 cycles and performs 64 floating-point operations, so the throughput is 64/24 = **8/3 = 2.67 FLOPs/cycle**

This table shows how the latencies add up for one loop iteration:

| Instruction | Iteration i | Iteration (i+1) |
|---|---|---|
| `lv` | 0 | 24 (waits for `sv` to finish) |
| `fadd.vs` | 4 (dependant on 1st `lv`) | |
| `lv` | 8 (waits for 1st `lv` to finish) | |
| `fmul.vv` | 12 (dependant on 2nd `lv`) | |
| `sv` | 16 (dependant on `fmul.vv`) | |

## *Question 3 (4 points)*

So far, we have assumed that loads can execute within a few cycles, which is reasonable if the data can be served by an L1 cache. Now consider if we are accessing arrays whose size far exceeds the capacity of any of our caches. Explain how a GPU would enable us to obtain high performance in this case.

A GPU executes many threads concurrently, overlapping many long-latency memory accesses from independent threads (by switching warps).

## *Question 4 (12 points)*

State whether each of the loops below can be vectorized on our vector processor. If the code would require the vector processor to have additional features to be vectorizable, specify those features. If the code cannot be vectorized regardless of what additional features the processor were to implement, state your reasoning.

The loops operate on floating-point arrays `A[N]`, `B[N]`, and `C[N]`. These arrays do not overlap.

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

The code can be vectorized with additional support for vector masking.

b)
```
for (int i = 0; i < N; i++)
  A[i] = A[i+1] * B[i] - 1.0;
```

Yes, the code can be vectorized (without any additional hardware support).

(Depending on the implementation, you could make a copy of some parts of array A, and that is easily done without requiring new hardware features.)

c)
```
for (int i = 0; i < N; i++)
  A[i] = 1.0 + A[i-1] * B[i];
```

No, the code cannot be vectorized. Each iteration depends on the value A[i-1] computed in the previous iteration, forcing serialized execution.

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

The code can be vectorized with additional support for vector gather.

# Part C: Transactional Memory (24 points)

Ben Bitdiddle is designing a hardware transactional memory (HTM) system. He is concerned about three potential issues arising in his system:

1. *Deadlock*: Some transactions stay stalled indefinitely on a cyclic waiting pattern, so they neither commit nor abort.

2. *Livelock*: Some transactions can execute, but no transaction ever commits (e.g., due to repetitive aborts and re-execution). Thus, the system does not make forward progress.

3. *Starvation*: Some transactions can commit, but at least one other transaction is prevented from committing indefinitely. Thus, one or a subset of transactions does not make forward progress.

Ben wants to classify each of the 4 HTM systems in Questions 1 to 4 as one of four types, according to the forward progress guarantees they provide:

**A.** May deadlock
**B.** May livelock, but cannot deadlock
**C.** May starve, but cannot deadlock or livelock
**D.** Cannot deadlock, livelock, or starve

For **Questions 1 to 4**, **write down the letter A, B, C, or D** and **explain your choice**. You can either explain intuitively why an issue cannot arise, or use an example to show that the system suffers from the issue.

When you choose a particular option, you only need to explain the issues it differentiates between. For example, if you choose B, you should explain why the system cannot deadlock, and describe an example of how it may livelock.

SOLUTIONS

## *Question 1 (4 points)*

**HTM 1**: Optimistic conflict detection, lazy versioning, and *committer-wins* resolution policy. Assume there is enough capacity for versioning (so transactions do not overflow speculative buffers, e.g., the L1 cache).

The committer-wins resolution policy works as follows. Upon a conflict, the committing transaction wins and any conflicting transactions are aborted. After aborting, transactions immediately restart execution.

C.

This system cannot livelock, because transactions can only be aborted by committing transactions, so forward progress for at least one transaction is guaranteed.

But it suffers from starvation. Example:
A long transaction can be repetitively aborted by many short transactions that update the data it reads.

## *Question 2 (4 points)*

**HTM 2**: Pessimistic conflict detection, lazy versioning, and *requester-wins* resolution policy. Assume there is enough capacity for versioning.

The *requester-wins* resolution policy works as follows. Upon a conflict, the transaction that triggers the detection (the requester) wins and any conflicting transactions are aborted. After aborting, transactions immediately restart execution.

B.

This system cannot deadlock because no transaction is stalled due to conflicts.

But it may livelock. Example:
Two conflicting transactions that write the same data can repetitively abort each other.

## *Question 3 (4 points)*

**HTM 3:** Pessimistic conflict detection, eager versioning, and *requester-stalls* resolution policy. Assume there is enough capacity for versioning.

The requester-stalls resolution policy works as follows. Upon a conflict, the transaction that triggers the detection (the requester) stalls until the conflicting transactions abort or commit. After aborting, transactions immediately restart execution.

A.

This system can deadlock. Example:

TX0: LD A ST B (TX0 stalls, waiting for TX1's commit/abort)
TX1: LD B ST A (TX1 stalls, waiting for TX0's commit/abort)

## *Question 4 (4 points)*

**HTM 4:** Pessimistic conflict detection, lazy versioning, and *oldest-wins* resolution policy. Assume there is enough capacity for versioning.

The oldest-wins resolution policy works as follows. Each transaction is assigned a unique, monotonically increasing timestamp when it first begins execution. Upon a conflict, if the requesting transaction's timestamp is lower than the timestamps of all other conflicting transactions, the requesting transaction commits and other conflicting transactions are aborted. Otherwise, the requesting transaction is aborted. After aborting, transactions immediately restart execution. Aborted transactions retain their original timestamp when they restart execution.

D.

This system is free from deadlocks, livelocks, and starvations.
The transaction with the lowest/oldest timestamp is guaranteed to commit. Because timestamps are assigned when beginning execution, starvation cannot happen: each transaction is guaranteed to eventually become the oldest one in the system.

## *Question 5 (4 points)*

Consider HTM 1, HTM 2, HTM 3, and HTM 4.
Which HTM(s) may suffer from serialization bottlenecks when running only non-conflicting transactions concurrently? Point out such bottlenecks for each machine.

HTM 1: Potentially serialized commits
HTM 4: Potentially serialized timestamp assignment
Lazy versioning is slow during commits but it happens locally and does not involve serialization among multiple transactions.

## *Question 6 (4 points)*

Consider an HTM system. Suppose **N** identical transactions are running simultaneously on N cores. All the transactions read and write to the same memory location, causing conflicts between any pair of them.

Does the HTM design guarantee that all the transactions can commit in the end? If so, what is the maximum number of aborts?

Answer the questions above for each of HTM 1, HTM 2, and HTM 3, respectively.

HTM 1: Yes, (N-1)N/2
HTM 2: No, due to livelocks
HTM 3: No, due to deadlocks

EDIT: The question is not clear about how many memory locations receive conflicted accesses. If there is only one such location, deadlock won't happen on HTM 3. We've adjusted the scores for students who reason in this way.