# Computer System Architecture
# 6.823 Quiz #3
# December 8th, 2021

Name: _____SOLUTIONS_____

This is a closed book, closed notes exam.
80 Minutes
16 Pages (+2 Scratch)

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.
- Show your work to receive full credit.
- 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.
- Pages 17 and 18 are scratch pages. Use them if you need more space to answer one of the questions, or for rough work.

| | | |
|---|---|---|
| Part A | _____ | 30 Points |
| Part B | _____ | 20 Points |
| Part C | _____ | 24 Points |
| Part D | _____ | 12 Points |
| Part E | _____ | 14 Points |
| **TOTAL** | _____ | **100 Points** |

# Part A: VLIW with Predication (30 points)

For this part, refer to the predication handout for details on the VLIW processor and its support for predication. Ben Bitdiddle is running the following code on his VLIW machine:

```
int A[N];
int B[N];
int C[N];
int D[N];
int x, y;
...

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

The following is the corresponding MIPS assembly code:

```
;; Initial values:
;;  R1 := &A[0], R2 := &B[0], R3 := &C[0], R4 := &D[0]
;;  R5 := x, R6 := y
;;  R20 := &A[N] (first address after array A)

loop: LW  R7, 0(R1)
      LW  R8, 0(R2)
      BLT R7, R0, else ; A[i] >= 0?
      MUL R7, R7, R5
      MUL R8, R8, R6
      ADD R8, R7, R8
      SW  R8, 0(R3)    ; C[i] = A[i] * x + B[i] * y
      J   next
else: MUL R8, R7, R8
      SW  R8, 0(R4)    ; D[i] = A[i] * B[i]
next: ADDI R1, R1, 4
      ADDI R2, R2, 4
      ADDI R3, R3, 4
      ADDI R4, R4, 4
      BNE  R1, R20, loop
```

Here we have converted the assembly code above into VLIW code:

| Label | Simple Int. Op | Simple Int. Op | Mul/Div Op | Memory Op |
|---|---|---|---|---|
| loop | ADDI R1, R1, 4 | ADDI R2, R2, 4 | | LW R7, 0(R1) |
| | ADDI R3, R3, 4 | ADDI R4, R4, 4 | | LW R8, -4(R2) |
| | | | | |
| | BLT R7, R0, else | | | |
| if | | | MUL R7, R7, R5 | |
| | | | MUL R8, R8, R6 | |
| | | | | |
| | ADD R8, R7, R8 | | | |
| | J next | | | SW R8, -4(R3) |
| else | | | MUL R9, R7, R8 | |
| | | | | |
| | | | | SW R9, -4(R4) |
| next | BNE R1, R20, loop | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

## *Question 1 (10 points)*

Rewrite the VLIW code by using predication to eliminate the branch within the loop as described in the handout. Your code should use as few VLIW instructions as possible.

| Label | Simple Int. Op | Simple Int. Op | Mul/Div Op | Memory Op |
|---|---|---|---|---|
| loop | ADDI R1, R1, 4 | ADDI R2, R2, 4 | | LW R7, 0(R1) |
| | ADDI R3, R3, 4 | ADDI R4, R4, 4 | | LW R8, -4(R2) |
| | | | | |
| | SETPGE p1, R7, R0 | | | |
| | | | (p1)MUL R11, R7, R5 | |
| | | | (p1)MUL R12, R8, R6 | |
| | | | (!p1)MUL R13, R7, R8 | |
| | (p1)ADD R11, R11, R12 | | | |
| | | | | (p1)SW R11, -4(R3) |
| | BNE R1, R20, loop | | | (!p1)SW R12, -4(R4) |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

There is a solution that completes in 9 cycles if you only predicate the stores, exploiting the fact that computing the register-destined operations both ways does not matter since the effect is not visible to the rest of the program. We gave full points for both solutions.

## *Question 2 (6 points)*

Assume that you are able to unroll or software-pipeline the predicated code as much as possible. What is the lowest number of cycles/iteration you can achieve? *Hint: You should be able to answer this question without writing down the loop-unrolled or software-pipelined code.*

With full unrolling and software pipelining, we need to look at which functional unit will be the bottleneck. With 4 memory operations and a single memory operation slot, each iteration must take at least 4 VLIW instructions, so it's 4 cycles/iteration.

## *Question 3 (7 points)*

Ben wants to unroll the original code *without any predication*. Since each iteration can take different paths of the if-else branch, his initial approach is to simply compute one iteration after another in the unrolled loop body. However, Alyssa P. Hacker points out that this implementation does not help performance much, as the unrolled loop only computes one iteration at a time.

Alyssa P. Hacker thinks that she can unroll the original code *and* overlap the computation from several iterations within the loop body to significantly improve execution time. Do you think this is possible? If so, describe at a high level how you can resolve different iterations of the unrolled loop taking different paths of the if-else statement. If not, briefly describe why it is impossible to unroll this loop without using Ben's inefficient approach.

One way is to create a branch target for each possible combination of branching paths the code can take. For a factor of N unrolling this will be 2^N targets. Then, for each target you unroll the loop given the branch resolution for each element.

Another way is to use trace scheduling like in Question 4 -- we profile the program and create a basic block consisting of the most commonly taken path. Then, you add fixup code for each of the branches (so there will be a total of N fixup code targets for N-level unrolling). Note that since the branches are data-dependent this may not work as well as expected if the values of array A are not regular and/or change between runs of the program.

Lastly, you can simply unroll the multiplies and the adds and only compute the stores in each branch, as suggested in the alternate solution to Question 1.

## *Question 4 (7 points)*

Ben profiles the code and finds that most elements of A will be greater than or equal to zero. Thus, he optimizes the above code sequence via trace scheduling by merging the not-taken path into a single basic block. He also adds a compensation code at location `comp` to jump to in case the branch is taken:

| Label | Simple Int. Op | Simple Int. Op | Mul/Div Op | Memory Op |
|---|---|---|---|---|
| loop | ADDI R1, R1, 4 | ADDI R2, R2, 4 | | LW R7, 0(R1) |
| | ADDI R3, R3, 4 | ADDI R4, R4, 4 | | LW R8, -4(R2) |
| | MOV R17, R7 | MOV R18, R8 | | |
| | | | MUL R7, R7, R5 | |
| | | | MUL R8, R8, R6 | |
| | | | | |
| | ADD R8, R7, R8 | BLT R17, R0, comp | | |
| | | | | SW R8, -4(R3) |
| next | BNE R1, R20, loop | | | |

Write down what the compensation code should be for this assembly code to be functionally correct. You should use as few VLIW instructions as possible.

No need to undo any action from the trace-scheduled block since it never stores to memory, so we can simply do the operations in the taken path.

| Label | Simple Int. Op | Simple Int. Op | Mul/Div Op | Memory Op |
|---|---|---|---|---|
| comp | | | MUL R9, R17, R18 | |
| | | | | |
| | J next | | | SW R9, -4(R4) |
| | | | | |
| | | | | |

# Part B: Vector Processors (20 points)

We will explore several different options for implementing vector masks. Recall that a vector mask allows a vector processor to selectively operate on elements whose corresponding masks are set. We explore three different design options which differ in terms of *when* the mask is read, and what action to perform after reading the mask.

Consider a vector processor with 4-elements long vectors. The machine has a *single* vector lane with an ALU that takes 4 cycles to process each element, including writeback. The ALU unit is also fully-pipelined such that it is able to process one vector element after another in consecutive cycles (i.e., if the first element is issued at cycle X, then the first result will be written back at cycle X+4, the next result at cycle X+5, etc).

To illustrate the different design options, we will look at the following piece of vectorized code:

```
SGE.VS  V1, R0
MUL.VV  V4, V2, V3
```

We provide the relevant descriptions for these vector instructions in the table below:

| Instruction | Meaning |
|---|---|
| `SGE.VS Vd, Rs` | Compare the elements in Vd and Rs. For each element in Vd, if the element is greater than or equal to Rs, set the corresponding bit of the vector mask register to 1, and 0 otherwise. |
| `MUL.VV Vd, Vs, Vt` | Multiply elements in Vs and Vt, and store result in Vd. This instruction reads the vector mask register to selectively operate on elements whose corresponding masks are set. |

## *Question 1 (6 points)*

First consider an *issue-time masking* implementation. This works by having the vector instruction read the vector mask register at issue, and sending the corresponding mask bit with the element to the vector lane. The lane will either perform the operation (if the mask is set) or do nothing for the element (if the mask is not set).

Suppose `SGE.VS` finishes writing to the vector mask register at cycle 8, and can be read the cycle after. What is the earliest cycle at which the first element of `MUL.VV` can be issued? Given your issue cycle, at what cycle is the last result written back?

Issue cycle: 9
Last written back: 9 + 4 (4 cycle latency) + 3 (3 more elements) = 16

## *Question 2 (7 points)*

Now consider a *writeback-time masking* implementation. This works by having the vector lane read the vector mask register at the writeback stage, and squashing the write if the corresponding mask bit is not set.

Again, suppose `SGE.VS` finishes writing to the vector mask register at cycle 8, and can be read the cycle after. What is the earliest cycle at which the first element of `MUL.VV` can be issued? Given your issue cycle, at what cycle is the last result written back?

Issue cycle: 5, since it can now be read 4 cycles later (the amount of ALU latency)
Last written back: 5 + 4 (4 cycle latency) + 3 (3 more elements) = 12

## *Question 3 (7 points)*

Now consider an *issue-time masking with skipping* implementation. Just like issue-time masking, each vector instruction reads the vector mask at issue. The difference here is that it *only sends the masked vector elements* to the (single-lane) ALU.

Again, suppose `SGE.VS` finishes writing to the vector mask register at cycle 8, and can be read the cycle after. Assume only half of `V1`'s elements are non-negative. What is the earliest cycle at which the first element of `MUL.VV` can be issued? Given your issue cycle, at what cycle is the last result written back?

Issue cycle: 9
Last written back: 9 + 4 (4 cycle latency) + 1 (1 more element due to skipping) = 14

# Part C: Transactional Memory (24 points)

In this part you will analyze the operation of different hardware TM (HTM) designs, and the concurrency they achieve for different transaction schedules on a 2-core 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 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 transaction receiving an invalidation or downgrade aborts, i.e. the *requester wins*.

- **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. Analogous to pessimistic requester-wins, the *committer wins*.

The system runs a program consisting of the following two transactions.

| Txn X |
|---|
| `Begin` |
| `Read A` |
| `Write B` |
| `Read C` |
| `End` |

| Txn Y |
|---|
| `Begin` |
| `Read B` |
| `Write A` |
| `Write C` |
| `End` |

In the following questions, for timing, assume conflict detection and coherence happen in the same cycle a memory access executes.

## *Question 1 (8 points)*

Suppose transaction X starts at cycle 0 and transaction Y starts at cycle 5, and they would produce the following schedule in the absence of conflict detection:

| Cycle | 0 | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 |
|---|---|---|---|---|---|---|---|---|---|---|
| Txn X | Begin | | Rd A | | Wr B | | Rd C | | End | |
| Txn Y | | Begin | | Rd B | | Wr A | | Wr C | | End |

a) In the absence of conflict detection (i.e. no HTM), if the memory operations interleaved in the given order, would the transactions be serializable in general? If so, circle what would be the apparent commit order of the transactions, or circle "Not serializable".

If your answer is "Not serializable", provide the *earliest cycle after 5* at which transaction Y can start such that the two transactions become serializable.

**X before Y** **Y before X** **Not serializable**

Txn Y needs to be issued at cycle 11 at the earliest for Rd B to be ordered after Wr B, such that Txn X is ordered before Txn Y.

b) Given the two mentioned HTM designs, indicate in the following table at what cycle a conflict is detected, if any, and which transaction aborts (or neither).

| | **Conflict cycle** | **Aborted Transaction (X, Y, or Neither)** |
|---|---|---|
| **Eager & Pessimistic** | **20** | **Y** |
| **Lazy & Optimistic** | **40** | **Y** |

## *Question 2 (8 points)*

Now suppose transaction X starts at cycle 0 and transaction Y starts at cycle 5, and they would produce the following schedule in the absence of conflict detection:

| Cycle | 0 | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 | 55 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Txn X | Begin | | Rd A | | Wr B | | Rd C | End | | | | |
| Txn Y | | Begin | | | | Rd B | | | Wr A | | Wr C | End |

a) In the absence of conflict detection (i.e. no HTM), if the memory operations interleaved in the given order, would the transactions be serializable? If so, circle what would be the apparent commit order of the transactions, or circle "Not serializable".

If your answer is "Not serializable", provide the *earliest cycle after 5* at which transaction Y can start such that the two transactions become serializable.

**X before Y** **Y before X** **Not serializable**

b) Given the two mentioned HTM designs, indicate in the following table at what cycle a conflict is detected, if any, and which transaction aborts (or neither).

| | **Conflict cycle** | **Aborted Transaction (X, Y, or Neither)** |
|---|---|---|
| **Eager & Pessimistic** | **25** | **X** |
| **Lazy & Optimistic** | **35** | **Y** |

## *Question 3 (8 points)*

Consider a different program consisting of the following three transactions:

| Txn V |
|---|
| Begin |
| Read A |
| Write A |
| Write B |
| End |

| Txn W |
|---|
| Begin |
| Read A |
| Write A |
| Read B |
| End |

| Txn Z |
|---|
| Begin |
| Read A |
| Write A |
| Write B |
| End |

Suppose transaction V starts at cycle 0, transaction W starts at cycle 5, transaction Z starts at cycle 10, resulting in the following schedule.

| Cycle | 0 | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 | 55 | 60 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Txn V | Begin | Rd A | Wr A | | | | | Wr B | | | | End | |
| Txn W | | Begin | | Rd A | Wr A | | | | | Rd B | | | End |
| Txn Z | | | Begin | | | Rd A | Wr A | | Wr B | | End | | |

a) In the absence of conflict detection (i.e. no HTM), if the memory operations interleaved in the given order, would the transactions be serializable? If so, what is the serialization order? (e.g., V before W before Z).

Not serializable.

b) Given the two mentioned HTM designs, indicate in the following table at what cycle the *first* conflict is detected, if any, and which transaction aborts (or none).

| | **Conflict cycle** | **Aborted Transactions (V, W, Z, or None)** |
|---|---|---|
| **Eager & Pessimistic** | **15** | **V** |
| **Lazy & Optimistic** | **50** | **V, W** |

# Part D: Security (12 points)

Consider an out-of-order processor. The processor has very precise timers, and has a simple branch predictor consisting of a table of 1-bit counters indexed by the lower bits of the PC. The processor is running an OS that has the following piece of kernel code:

```
      kernel_syscall(int X) {
        // X is allocated to register R1
        int secret;       // Allocated to register R2
        int counter;      // Allocated to register R3

        // long sequence of instructions
I0:     bool condition = secret > X;     // SLT  R4, R1, R2
I1:     if (condition)                   // BEQZ R4, I3
I2:       counter++;                     // ADDI R3, R3, 1
I3:     ...                              // ...
        // long sequence of instructions
      }
```

There exists an exploit for this syscall that relies on the *branch predictor* we described above to leak the `secret`, where the attacker has control over the variable `X`. Answer the following series of questions to relate the concepts we have learned to this exploit (2 points each):

1) What is the microarchitectural side-channel for transmitting the secret in this code? Be specific in your answer (i.e., don't just answer "the branch predictor").

   The branch predictor entry that I1 aliases into.

2) Is an active receiver needed? If so, how does the receiver precondition the channel?

   Yes. The receiver needs to precondition the branch predictor entry to be in a known state such that it can detect when the branch is taken

3) Which instruction(s) are the transmitter? You should answer in terms of their labels provided in the code above (e.g., `I0` and `I2`).

   I1 is the transmitter since it modulates the channel based on the value of secret. Also accepted the answer that I0 and I1 and both part of the transmitter since I0 is needed to set up the transmission.

4) How does the transmitter modulate the channel?

   It sets the branch predictor entry to 1 if the secret is greater than X, and 0 otherwise.

5) How does the receiver decode the modulation on the channel?

   The receiver first primes the branch predictor entry to 0 by evaluating a not-taken branch that aliases into the same branch table entry, and runs the syscall. After the transmission, it probes the channel by again evaluating the not-taken branch again. If the branch resolution is slow, secret is greater than X because the branch predictor will mispredict.

6) What information is contained in one transmission?

   Whether secret is greater than X.

   Bonus: Given this "bandwidth", we can figure out all the bits in the secret within 32 transmissions (since the secret is 32 bits) assuming no noise.

# Part E: Accelerators (14 points)

For a given compute kernel, we define a tensor's **reuse interval (RI)** as the number of different elements of that tensor that have been referenced between each re-reference of the same element. For example, consider the following:

```
for m in [0, M)
  for n in [0, N)
      Z[m, n] = A[m] * B[n]
```

Since A's element is used at every iteration of the inner loop, its RI is 1. Each element of `B` is re-referenced after N references, so its RI is N. `Z` has "infinite" reuse interval (i.e., no data reuse) since no element is re-referenced throughout the computation:

RI of A = 1
RI of B = N
RI of Z = infinite / no reuse

## *Question 1 (4 points)*

Consider the following Matrix-Matrix multiply pseudocode, which multiplies two dense matrices `A` and `B` to produce `Z`:

```
;; Multiply two matrices A and B to produce Z
;; First matrix A is MxK
;; Second matrix B is KxN
;; Thus, resulting matrix Z is MxN

for m in [0, M)
  for n in [0, N)
    for k in [0, K)
      Z[m, n] += A[m, k] * B[k, n]
```

What are the reuse intervals for the three matrices? Provide your answers in terms of M, N, and K.

RI of A = K
RI of B = N*K
RI of Z = 1

## *Question 2 (5 points)*

Now consider the following where we re-order the loop nest:

```
for k in [0, K)
  for m in [0, M)
    for n in [0, N)
      Z[m, n] += A[m, k] * B[k, n]
```

What are the reuse intervals for the three matrices? Provide your answers in terms of M, N, and K.

RI of A = 1
RI of B = N
RI of Z = M*N

## *Question 3 (5 points)*

Now consider the following scenario:

- M = 500, N = 1000, and K = 30
- `A`, `B`, and `Z` are dense matrices.
- The matrices are resident in DRAM at the beginning of the computation kernel.

Your wish to design a matrix-matrix multiply accelerator given the above assumptions. Your goal is to maximize **compute intensity**, defined as the average number of computations you perform per DRAM access. You are given the following hardware budget:

- A multiply-accumulate (MAC) unit with a flip-flop on each input and the output.
- An SRAM array large enough to store 32 elements.

Which order of loop nest would you choose, and given the order how would you partition the SRAM budget among the different elements to maximize compute intensity? You can reorder the loop nests from Questions 1 and 2, but do not add more loops (e.g., no tiling).

Use the loop nest from Question 1 (output stationary), and allocate 30 elements for the entire row of A. This row will be used for the entire computation of a single element of Z, then never reused again. The other two elements can be used to store anything else and will not impact computational intensity much.

## *Scratch Space*

Use these extra pages if you run out of space or for your own personal notes. We will not grade these unless you tell us explicitly in the earlier pages.

## *Scratch Space*

Use the table below if you run out of space when answering questions in Problem 1.

| **Label** | **Simple Int. Op** | **Simple Int. Op** | **Mul/Div Op** | **Memory Op** |
|---|---|---|---|---|
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |