University of California at Berkeley
College of Engineering
Department of Electrical Engineering and Computer Sciences

EECS152/252A
Spring 2022

J. Wawrzynek
5/11/22

Final Exam Solutions

Name: _____

Student ID number: _____

You have until 2:30PM to take the exam.

This is a *closed-book, closed-notes* exam, except for two handwritten sheets. Also no calculators, phones, pads, or laptops allowed.

Each question is marked with its number of points (one point per expected minute of time). Start by answering the easier questions then move on to the more difficult ones. You can use the backs of the pages to work out your answers. Neatly copy your answer to the allocated places. **Neatness counts.** We will deduct points if we need to work hard to understand your answer.

Write the student ID numbers of the person to your left and to your right on this page. If you are sitting on an aisle, just indicate "aisle" for left or right.

Left student ID number: _____

Right student ID number: _____

Before you turn in your exam, write your student ID number on all pages.

Monday 16th May, 2022 15:57

1. MEMORY HIERARCHY [24 pts]

In this question, you are asked to design a memory hierarchy with a 32-bit virtual and physical address and 8-word (32 byte) cache lines. Consider the two loops below:

Loop A:

```
int sum = 0;
for (int i = 0; i < 256; i++) {
   for (int j = 0; j < 64; j++) {
      sum += Y[i][j] * Z[i][j];
   }
 }
```

Loop B:

```
int sum = 0;
for (int j = 0; j < 64; j++) {
   for (int i = 0; i < 256; i++) {
      sum += Y[i][j] * Z[i][j];
   }
}
```

Assume `Y[0][0]` is stored at address 0x0. Further assume that matrices Y and Z are stored contiguously and that both matrices are stored in row-major order, i.e. `Y[i][j]` is next to `Y[i][j+1]` in memory and `Y[256][64]` is next to `Z[0][0]`. `sum` is not stored in the cache. Also assume that caches are initially empty. All elements of the matrices are 32-bit integers.

(a) Consider a 32KiB 4-way set-associative data cache with FIFO replacement policy. Calculate the number of compulsory, conflict, and capacity misses that will occur when running each of Loop A and B. Show your work below and fill out the table.

	Compulsory	Conflict and Capacity	Total Misses
Loop A	$2^{12} = 4096$	0	4096
Loop B	4096	$2^{15} - 4096 = 28672$	$2^{15} = 32768$

- Loop A accesses the two matrices in row-major order (linear access). Only misses are compulsory misses that occur at the first access to each line. This happens every 8 accesses since cache lines are 8-words long. Therefore $256 \times 64 \times 2/8 = 2^{12} = 4096$ compulsory and total misses.
- Loop B accesses each matrix with a stride of 64 words or 256 Bytes. The first iteration of the inner loop will access sets with indices 0, 8, 16, ..., 127, and after 128/8*4 = 64 column acceses, all four-ways of these sets will be full. The following accesses during the remainder of the inner loop will thus need to evict earlier lines, which prevents any reuse of cached lines. Effectively, this access pattern uses only 1/8 of the available sets in the cache. Therefore, all memory accesses will result in a cache miss with the same number of compulsory misses (4096) and the rest being capacity or conflict misses.

Grading: total 8 pts. [-4] for each incorrect answer for Loop A and B, and partial credit assigned if parts of the table entry are correct.

(b) Suppose you implement the cache as Virtually-Indexed Physically-Tagged (VIPT). Assuming a 4 KiB page size, is it possible to have virtual address aliasing with this cache? If so, what is the minimum number of set-associativity that is required to avoid aliasing given that the cache size is fixed at 32 KiB? Explain your reasoning.

Aliasing can occur. Minimum associativity to not have virtual address aliasing in a VIPT cache: (Num Ways) x (Num Sets) x (Line Size) ≤ (Page Size). Therefore associtvity has to be at least 8.

Grading: total 4 pts.

- Correctly mention that there can be aliasing [+2]
- Correctly answered minimum 8-way associativity for no alasing [+2].
- Partial credit for correctly setting up numerical relationship between cache parameters and min. associativity. [+1]
- Partial credit for correct answers with insufficient justification. [+0.5]

(c) Now, consider implementing a L2 cache in addition to the VIPT L1. The L2 cache starts off empty. With no more than three sentences, briefly explain how an inclusive L2 cache can be used to resolve aliasing.

An inclusive, physically-indexed (e.g. PIPT) L2 cache can resolve virtual address aliasing by tracking which virtual address is currently mapped to the given physical address. Suppose two different addresses VA1 and VA2 map to the same physical address and that VA1 is already in the L1. By the inclusion property, L2 also holds the line. When VA2 is accessed, it will miss in the L1, get translated to PA, and then detected in the L2 that VA2 is different from VA1. For more detail, refer to the lecture slides on virtual memory.

Grading: total 2 pts. Full credit only if the answer:

- Mentions that the L2 needs to be physically-indexed in order to correctly detect aliasing [if not, -0.25]
- Describes clearly and exactly how the inclusive L2 allows aliasing resolution [if not, -0.5].

(d) Suppose you implement a 256 KiB 8-way set-associative L2 cache with FIFO replacement policy. Assume the L1 data cache is still 32 KiB 4-way set-associative and the L2 is inclusive of the L1 cache.

When you execute Loops A and B, how many cache misses occur at the L2?

- Loop A still experiences the same number of compulsory misses as in the L1, since these misses are unavoiadable. Total of 4096 misses.
- Now the L2 is large enough with $2^{18}/(2^5 * 2^3) = 2^{10}$ sets such that a single iteration of the inner loop that goes through 256 columns in 64-word strides for each matrix can all fit in the 1024 sets of 8 ways. In the next iteration of the inner loop, we then access the next word in each of the cached lines, resulting in cache hits. Therefore, the only misses are compulsory misses that occur 4096 times.

Grading: total 4 pts, 2 pts for each Loop A and B. Partial credit for correctly answering that there are only compulsory misses.

(e) Based on extensive profiling and performance analysis, you measure that the L1 hit time is 4 cycles, the L2 hit time is 12 cycles, and main memory (DRAM) access time is 50 cycles. What is the average memory access time (AMAT) while executing Loop A and B? You can leave your answer in the form of an expression.

(Hint: what are the miss rates at each level of memory hierarchy when executing the loops?)

$$\text{AMAT} = t_{L1} + LMR_{L1} \cdot (t_{L2} + LMR_{L2} \cdot t_{DRAM})$$

where t_{L1}, t_{L2} and t_{DRAM} are L1, L2 and DRAM hit times, and the LMR's are local miss rates.

- Loop A: AMAT $= 4 + 1/8 \cdot (12 + 1 \cdot 50) = 11.75$ cycles
- Loop B: AMAT $= 4 + 1 \cdot (12 + 1/8 \cdot 50) = 22.25$ cycles

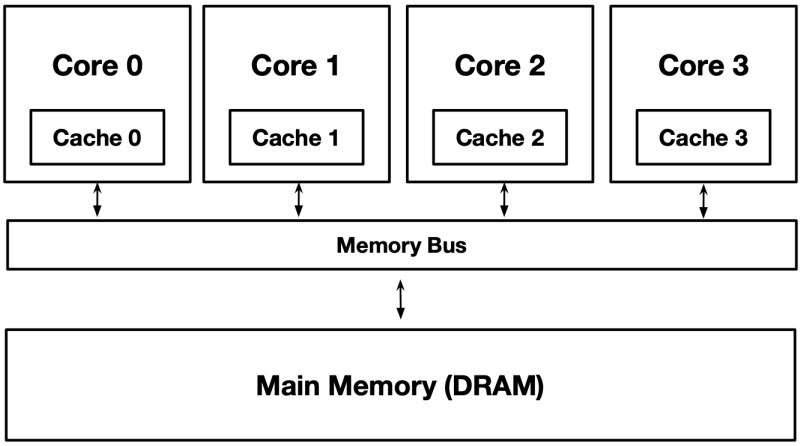
Note that the local miss rates for the L2 cache is 1 (always miss) for Loop A. For details and review on this topic, refer to Henessey and Patterson Appendix B.3.

Grading: total 6 pts.

- Correctly showed or identified equation to calculate AMAT [+3]
- Correctly calculated AMAT for each loop [+1.5 each]
- Partial credit for answers that correctly setup equations but have incorrect values for miss rates or other errors.

2. Memory Cache Coherency [12 pts]

We are given a multi-core system where each core has its own private cache, and the caches snoop each other and communicate with main memory over a shared memory bus as shown below. Consider the baseline snoopy cache-coherence protocol discussed in Handout 7 and in Problem Set 5.



Recall that it implements the MOESI protocol with five possible cache states.

Invalid (I): Block is not present in the cache.

Clean exclusive (CE): The cached data is consistent with memory, and no other cache has it.

Owned exclusive (OE): The cached data is different from memory, and no other cache has it. This cache is responsible for supplying this data instead of memory when other caches request copies of this data.

Clean shared (CS): The data has not been modified by the corresponding CPU since cached. Multiple CS copies and at most one OS copy of the same data could exist.

Owned shared (OS): The data is different from memory. Other CS copies of the same data could exist. This cache is responsible for supplying this data instead of memory when other caches request copies of this data.

The following memory bus transactions were supported by the protocol.

Coherent Read (CR): issued by a cache on a read miss to load a cache line.

Coherent Read and Invalidate (CRI): issued by a cache on a write-allocate after a write miss.

Coherent Invalidate (CI): issued by a cache on a write hit to a block that is in one of the shared states.

Block Write (WR): issued by a cache on the write-back of a cache block.

Coherent Write and Invalidate (CWI): issued by an I/O processor (DMA) on a block write (a full block at a time).

Cache to Cache Intervention (CCI): used by a cache to satisfy other caches' read transactions when appropriate. A CCI intervenes and overrides the answers normally supplied by memory. Data should be supplied using CCI whenever possible for faster response relative to the memory.

(a) Consider the case where each of the private caches are write-back, write-allocate. Assume that all caches are initially empty and that words A and B are in the same cache line.

Complete the following table that shows a sequence of memory events that occur.
For each event, determine:

- which caches send out what messages on the bus
- the subsequent states of each of the caches *after the event*.
- whether main memory will have the most up-to-date value of the cache block *after the event*.

ID	Event	Message Shared	Cache0 State	Cache1 State	Cache2 State	Main Memory Up-to-Date?
0	CPU0: read A	0:CR	CE	I	I	Yes
1	CPU2: write B	2:CRI	I	I	OE	No
2	CPU1: read B	1:CR; 2:CCI	I	CS	OS	No
3	CPU1: write B	1:CI	I	OE	I	No
4	CPU2: write A	2:CRI; 1:CCI	I	I	OE	No
5	CPU0: write B	0:CRI; 2:CCI	OE	I	I	No

Grading:

- each fully correct line in the table [+2]
- each partially correct line with all cache states correct [+1.5]
- each partially correct line with some cache states incorrect [+1]

(b) Which of the events in the table correspond to false sharing misses? Answer in terms of the event IDs.
None of the events are false sharing misses. Even if A and B were in separate lines, the invalidations and misses would occur as described in the table.

Grading:

- Correctly answered that none correspond to false sharing misses [+2]
- Partial credit for answers that list only events 1 or 2; these invalidates are true sharing misses only because the caches were initially empty (invalid) [+1]

3. OUT-OF-ORDER [22 pts]

Consider the loop below, performing a saturating vector sum; `result`, `X[i]`, and `MAX` are all of type `int32`.

```
for (int i = 0; i < N; i++) {
    result += X[i];

    if (result > MAX)
        result = MAX;
    }
}
```

In assembly, the loop is as follow:

```
# x1 is a pointer to the beginning of "X"
# x2 is a pointer to the end of "X"
# x3 has "MAX"
# Return "result" in x4

          beq x1, x2, end

   loop:  ld x5, 0(x1) # x5 has "X[i]"
          add x4, x4, x5

          blt x4, x3, skip_sat
          mv x4, x3  # Equivalent to "addi x4, x3, 0"

skip_sat: addi x1, x1, 4
          bne x1, x2, loop
     end:
```

(a) **Register Renaming** [14 pts]

Now, suppose we run *one iteration* of this loop on an out-of-order, superscalar core with the following characteristics:

- Up to two instructions can be fetched, decoded, dispatched, issued, written-back, and committed every cycle.
- The ROB has 4 entries.
- There are 32 physical registers.
- There is one fully-pipelined load/store unit.
- There is one fully-pipelined ALU. The ALU also executes branches.
- Adds and branches take 1 cycle to execute.
- Loads and stores take 3 cycles to execute.
- All instructions must spend one cycle in the write-back stage before their result can be used by a dependent instruction.
- An ROB entry becomes available one cycle after the instruction it is holding commits.
- The scheduler always selects the oldest ready instructions to issue.

- The CPU can execute all instructions speculatively in case of a branch.

Below is the register rename table at the beginning of execution:

Architectural Register	**Physical Register**
x1	P1
x2	P2
x3	P3
x4	P4
x5	P5

The free list is a FIFO with the initial state below. The leftmost element is the head of the free list, and the rightmost element is the tail.

P7	P11	P9	P14	P10	P18	P20

For this section, assume that the CPU correctly predicts all branches (as well as branch targets).
In the table below, fill in the cycle number for when each instruction enters the ROB, issues, writes-back, and commits. Also, fill in the new register names for each instruction, where applicable. Use only physical register names for the src entries.
Assume that the ROB is initially empty. Part of the table has been filled in for you.

Time				OP	Dest	Src1	Src2
Enter ROB	Issue	WB	Commit				
0	1	4	5	`ld x5, 0(x1)`	P7	P1	-
0	5	6	7	`add x4, x4, x5`	P11	P4	P7
1	7	8	9	`blt x4, x3, skip_sat`	-	P11	P3
1	2	3	9	`mv x4, x3`	P9	P3	-
6	8	9	10	`addi x1, x1, #4`	P14	P1	-
8	10	11	12	`bne x1, x2, loop`	-	P14	P2

Also, fill in the values in the free list when the last instruction in the table above commits:

P10	P18	P20	P5	P4	P11	P1

Grading:

- Cycle times not completely correct [-1]
- Register names not completely correct [-1]
- Free list not completely correct [-1]

- Does not delay entry of some instructions into ROB due to ROB size [-1]
- Delays issuing the `mv` instruction even though it has no RAW hazards [-1]

- Issues multiple instructions into the ALU at the same time [-1]
- Commits instructions out-of-order, or enters them into the ROB out-of-order [-2]

- Never dispatches/writes-back/commits two instructions in the same cycle, even when possible [-1]

- Issues two or more instructions earlier than RAW dependencies permit. [-1]
  - We don't take away this particular point if you issue in the same cycle as a writeback, even though this is not permitted.

- Incorrect number of source registers for one or two instructions. [-1]
- Incorrect number of source registers for three or more instructions. [-2]

- Incorrect number of destination registers for one or two instructions. [-1]
- Incorrect number of destination registers for three or more instructions. [-2]

(b) **Short Answers** [8 pts]

i. Suppose that the `blt x4, x3, skip_sat` instruction in part 3a was mispredicted. After the following instructions were flushed, what would be the state of the free list?

P9	P14	P10	P18	P20	P5	P4

(Because this question did not specify exactly how the free-list was recovered (e.g. with a rollback or with a snapshot), we also accepted other orderings of the free list.)

Grading:

- One or two entries incorrect [-1]
- Three or more entries incorrect [-2]

ii. Which of the following optimizations might improve the performance of this single iteration of the loop shown above? Assume that N is very large and branch prediction is perfect. Check off all that apply.

- ☐ Adding more load-store units, but otherwise keeping the CPU unchanged.
  (Only one instruction in the trace of the iteration above needed the load-store unit.)
- ☐ ✓ Adding more ALUs, but otherwise keeping the CPU unchanged.
  (Adding an extra ALU would allow the `addi x1` instruction to be issued one cycle earlier. This would cause the final `bne` to be issued and committed one cycle earlier.)
- ☐ ✓ Adding more ROB entries, but otherwise keeping the CPU unchanged.
  (If we add even just one more ROB entry, then the final `bne` can be committed two cycles earlier.)

Grading:

- ☐ One or two checkmarks are incorrect [-1]
- ☐ All three checkmarks are incorrect [-2]

iii. Data-in-ROB designs store the actual values of their source and destination registers in the ROB itself. But this leads to expensive data duplication.
One tempting optimization would be to store only tags for the source registers, where these tags can point to the architectural register file or to other rows in the ROB.
Would this optimization result in correct behavior? Explain your answer.

Yes. (This similar to a design with a unified physical register file except that the physical registers can be ROB entries).
If instruction A relies upon the result of instruction B which writes to register R, then no other instruction writing to R will be able to commit until A commits. So A's source operands will always be in either the arch regfile or in the ROB.

Grading:

- ☐ Argues that the optimization would cause incorrect behaviour [-1]
- ☐ Gives an insufficiently thought-out reason/explanation [-1]

(This question was also a bit unspecified. E.g., we did not specify that the tags would be updated whenever an earlier instruction commits. So we also gave partial (or even full) credit to students who answered "No", as long as their explanation was sufficiently thought-out, plausible, and detailed.)

iv. A mispredicted branch can greatly reduce the performance of a loop! To reduce that penalty, an engineer in your team suggests that you replace these two instructions:

```
blt x4, x3, skip_sat
mv x4, x3
```

with the single instruction below, which does the same thing:

```
min x4, x4, x3   # x4 < x3 ? x4 : x3
```

What conditions would the compiler need to check for to correctly perform this replacement?

- ☐ The branch offset must be +4
- ☐ The `blt` and `mv` must be consecutive
- ☐ The first source operand of the `blt` must be the destination register of the `mv`
- ☐ The second source operand of the `blt` must be the source register of the `mv`

Grading:

- ☐ Does not mention that the branch offset must be +4 [-1]
- ☐ Does not mention any relevant conditions that a compiler would have to check for [-1]
  - Information which would not be available to a compiler, like the exact timing of instructions or the presence of structural hazards, is not sufficient to gain points for this question.

4. VLIW [9 pts]

(a) **Software Pipelining** [8 pts] Consider the loop below, performing a saturating vector sum; `result`, `X[i]`, and `MAX` are all 32-bit floats.

```
for (int i = 0; i < N; i++) {
    result += X[i];
    result = min(result, MAX);
}
```

In assembly the loop is as follow:

```
# x1 is a pointer to the beginning of "X"
# x2 is a pointer to the end of "X"
# f0 has "MAX"
# We return "result" in f1

loop: fld f2, 0(x1) # f2 has "X[i]"
      fadd f3, f1, f2
      fmin f1, f3, f0

      addi x1, x1, #4
      blt x1, x2, loop
 end:
```

The exam had a typo where `blt` was written as `bgtz` instead.

Suppose that we run this loop on a VLIW architecture with the following fully-pipelined functional units:

- One integer ALU unit which also perform branches; 1 cycle delay.
- One FPU which performs "fadd" and "fmin"; 2 cycle delay.
- One load/store unit; 3 cycle delay.

Instructions are statically scheduled with no interlocks; all latencies are exposed in the ISA. All register operands are read before any writes from the same instruction take effect (i.e., no WAR hazards between operations within a single VLIW instruction).

Schedule the loop using software pipelining (but without unrolling the loop) in the table below. You can just write the opcode and destination register of the instructions in the table.

Minimize the number of cycles taken. You can re-order instructions.

label	ALU	FPU	MEM
	addi x1		fld x2
		fadd f3	
loop	addi x1		fld f2
		fmin f1	
	blt loop	fadd f3	
		fmin f1	

Any other solution which is functionally correct, and achieves the same or better performance, is also acceptable.

Grading:

- Does not perform software pipelining [-3]
- Solution is not functionally correct [-2]
- Solution is functionally correct, but each loop iteration take more cycles than the reference solution [-1]
- Does not place the `fmin` of iteration `i` before the `fadd` of iteration `i+1` [-1]
- Branch targets not clearly labelled [-2]

A common mistake was to neglect the fact that the `fmin` command has a two-cycle latency.

(b) **Short Answer** [1 pt] Does trace-scheduling become more useful, or less useful, as your static branch prediction accuracy increases? Check off the correct answer.

- ☐ ✓ More useful
- ☐ Less useful

Trace-scheduling statically optimizes code-paths which are very likely to be taken. If you can't statically predict which code-paths will be taken, then there is little point in optimizing them.

Grading:

- ☐ Checked "less useful" [-1]

5. MULTITHREADING [8 pts]

(a) **Thread Scheduling** [6 pts]

Consider the loop below, performing a saturating addition of two vectors; `Y[i]`, `X[i]`, and `MAX` are all of type `int32`.

```
for (int i = 0; i < N; i++) {
    Y[i] += X[i];

    if (Y[i] > MAX)
        Y[i] = MAX;
    }
}
```

In assembly, the loop is as follow:

```
# x1 is a pointer to the beginning of "X"
# x2 is a pointer to the beginning of "Y"
# x3 is "MAX"
# x4 is "i"

    loop: ld x5, 0(x1) # x5 has "X[i]"
          ld x6, 0(x2) # x6 has "Y[i]"

          addi x1, x1, 4 # increment X pointer
          addi x2, x2, 4 # increment Y pointer
          addi x4, x4, -1 # decrement "i"

          add x7, x5, x6

          blt x7, x3, skip_sat
          mv x7, x3

skip_sat: sw x7, -4(x2)

          bnez x4, loop
     end:
```

The exam had a typo where the integer offset to the `sw` command was `0` instead of `-4`.

Suppose that we run this loop on a multi-threaded 5-stage classic RISC processor where:

- Loads and stores have a latency of 20 cycles
- Taken branches have a latency of two cycles
- All other instructions (including non-taken branches) have a latency of one cycle
- There is perfect branch prediction

Do not reorder the instructions in the loop.

i. How many threads are needed to guarantee that we will never stall with fixed round-robin scheduling?

The longest stall is caused by the dependency between the `ld x6` and the `add x7`.

$5N - N \geq 20$

$4N \geq 20$

$N \geq 5$

At least 5 threads are required. This is also sufficient to hide the latency of the taken branches.

To illustrate:

ld x6 ld x6 ld x6 ld x6 ld x6 addi x1 addi x1 addi x1 addi x1 addi x1 addi x2 addi x2 addi x2 addi x2 addi x2 addi x4 addi x4 addi x4 addi x4 addi x4 add x7

Grading:

- Answer is not 5 threads [-1]
- Does not identify the stall caused by the `ld x6` to `add x7` dependency as the primary performance limiter [-1]
- No attempt to divide latency by number of threads [-1]

ii. Suppose that we instead use a coarse-grained thread scheduling policy, which switches threads whenever a stall would occur due to a RAW hazard or due to a taken branch. (WAR and WAW hazards do not cause stalls).
How many threads are needed to guarantee that would never stall with this scheduling policy? Consider only the steady-state execution.

In the worst case, we must assume that both branches are always taken, each causing a stall. Therefore, the longest sequence of instructions that we can execute without any stalls is: `ld x5`, `ld x6`, `addi x1`, `addi x2`, `addi x4`

There is still a 16-cycle stall caused by the `ld x6` to `add x7` dependency. To hide this latency in the steady-state, we need the following number of threads:

$\lceil \frac{16}{5} \rceil + 1 = 4 + 1 = 5$

Five threads is also sufficient to hide the latency of the taken branches.

To illustrate:

ld x5, ld x6, addi x1, addi x2, addi x4, ld x5, ld x6, addi x1, addi x2, addi x4, ld x5, ld x6, addi x1, addi x2, addi x4, ld x5, ld x6, addi x1, addi x2, addi x4, ld x5, ld x6, addi x1, addi x2, addi x4, add x7

Grading:

- Answer is not 5 threads [-1]
- No attempt to find longest sequence of unstalling instructions [-1]
- Does not identify the stall caused by the `ld x6` to `add x7` dependency as the primary performance limiter [-1]

(b) **Short Questions** [2 pts]

Suppose that we have two machines, A and B. They are both used to run a wide variety of multithreaded workloads.

A has one CPU. The CPU is an OoO 8-wide superscalar with SMT. Two threads can run simultaneously on the CPU.

B has two CPUs. Each CPU is an OoO 4-wide superscalar without multithreading.

Check off the correct answers below:

i. A would be expected to have:
- ☐ the same clock cycle time as B?
- ☐ a lower clock cycle time than B?
- ☐ ✓ a higher clock cycle time than B?

ii. A would be expected to have:
- ☐ the same CPI as B?
- ☐ ✓ a lower CPI than B?
- ☐ a higher CPI than B?

Grading:
- ☐ i is not "higher" [-1]
- ☐ ii is not "lower" [-1]

6. Vector Processors [16 pts]

(a) **Vector Chaining** [10 pts]

Consider the loop below, performing a saturating addition of two vectors; `Y[i]`, `X[i]`, `Z[i]`, and `MAX` are all of type `double`.

```
for (int i = 0; i < N; i++) {
    X[i] = Y[i] + Z[i];
    X[i] = min(X[i], MAX);
}
```

In vectorized assembly, the loop is as follow:

```
      # x1 is a pointer to the beginning of "X"
      # x2 is a pointer to the beginning of "Y"
      # x3 is a pointer to the beginning of "Z"
      # x4 is "N"
      # f0 is "MAX"

loop: vsetvli x5, x4, e64 # x5 is the vector length

      vle.v v0, 0(x2) # ld "Y"
      vle.v v1, 0(x3) # ld "Z"
      vfadd.vv v2, v0, v1 # add Y and Z
      vfmin.vv v3, v2, f0 # min(X, MAX)
      vse.v v3, 0(x1) # st "X"

      sub x4, x4, x5
      slli x5, x5, 3
      add x1, x1, x5
      add x2, x2, x5
      add x3, x3, x5

      bnez x4, loop
```

The exam had a typo where `x1`, `x2`, and `x3` were not labelled properly.

Suppose we run this loop on a vector machine with the following characteristics:

- 8 elements per vector register
- 4 vector lanes
- 1 load-store unit per lane, 3-cycle latency
- 1 FPU per lane, 2-cycle latency
  - This performs both additions and `vfmin` operations
- All functional units are fully pipelined
- All functional units have dedicated read/write ports into the vector register file
- No dead time
- Vector instructions execute in order
- Scalar instructions execute separately on a decoupled scalar processor

We will compare the performance of the vector processor with and without chaining. Vector chaining is performed through the vector register file. An element can be read on the same cycle that it is written back, or it can be read on any later cycle—the chaining is flexible.

However, with no chaining, a dependent vector instruction must stall until the previous vector instruction finishes writing back all elements. As an example, the pipeline timing would proceed as follows for two dependent `vadd` instructions if not using chaining:

Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**
`vfadd v2, v0, v1`	R	X1	X2	W			
		R	X1	X2	W		
`vfadd v4, v2, v3`						R	W

Complete the following table for one stripmine iteration, which shows the cycle numbers at which each vector instruction begins execution (starting from the vector register read). The first column corresponds to the baseline vector design with no chaining. The second column adds flexible chaining to the processor. Assume that vl is set to the maximum vector length, and the first vector instruction executes in cycle 1. Ignore scalar instructions.

Instruction	Cycle Number	
	without chaining	with chaining
`vle.v v0, 0(x1)`	1	1
`vle.v v1, 0(x2)`	3	3
`vfadd.vv v2, v0, v1`	9	7
`vfmin.vv v3, v2, f0`	14	10
`vse.v v3, 0(x2)`	19	13

To illustrate for the case without vector chaining:

Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
vle v0	R	M1	M2	M3	W														
		R	M1	M2	M3	W													
vle v1			R	M1	M2	M3	W												
				R	M1	M2	M3	W											
vfadd v2									R	X1	X2	W							
										R	X1	X2	W						
vfmin v3														R	X1	X2	W		
															R	X1	X2	W	
vse v3																			R

To illustrate for the case with vector chaining:

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13
vle v0	R	M1	M2	M3	W								
		R	M1	M2	M3	W							
vle v1			R	M1	M2	M3	W						
				R	M1	M2	M3	W					
vfadd v2							R	X1	X2	W			
								R	X1	X2	W		
vfmin v3										R	X1	X2	W
											R	X1	X2
vse v3													R

Grading:

- Without chaining column is not completely correct [-1]
- With chaining column is not completely correct [-1]
- Without chaining: `vle v1` does not begin 1-3 cycles after `vle v0` [-1]
- With chaining: `vle v1` does not begin 1-3 cycles after `vle v0` [-1]
- Without chaining: `vfadd` does not begin 5-7 cycles after `vle v1` [-1]
- With chaining: `vfadd` does not begin 3-5 cycles after `vle v1` [-1]
- Without chaining: `vfmin` does not begin 4-6 cycles after `vfadd` [-1]
- With chaining: `vfmin` does not begin 2-4 cycles after `vfadd` [-1]
- Without chaining: `vse` does not begin 4-6 cycles after `vfmin` [-1]
- With chaining: `vse` does not begin 2-4 cycles after `vfmin` [-1]

(b) **Short Questions** [4 pts]

i. Suppose we are running a program with extremely long vector lengths. Why might we want to have fewer vector lanes than the vector register length? Assume that cycle time is not affected by the number of lanes, and that hardware complexity costs are not a concern.

There are many potential answers:

- If we had as many vector lanes as the vector register length, then a new vector command could theoretically complete every cycle. However, our scalar processor might not be able to keep up and issue vector commands at such a high throughput, causing low functional unit utilization.
- Completing vector commands so quickly will make it more difficult to hide the impact of long latency instructions (such as loads) through vector-chaining.
- Having more vector lanes increases the negative performance impact of dead-time.

Grading:

- Partially correct reason why we would want fewer vector lanes [-1]
- Significantly incorrect or incomplete why we would want fewer vector lanes [-2]
- An explanation which focuses only on cycle time or hardware complexity costs, e.g. "we want fewer vector lanes to save area" [-2]

Note: we specify that the program has extremeley long vector lengths. So answers that focus on small vectors, or the "remainder" of a stripmined loop are not sufficient, as their negative impact becomes more and more insignificant as vector lengths increase.

ii. Why are precise exceptions difficult to support with vector processors?

You may have already written many elements back into the vector register file before you come across an element towards the end of your vector that causes an exception. Maintaining precise exceptions requires you to preserve the original value of a full vector register, which can be a lot of state to maintain. Alternatively, it may require you to check that no elements cause an exception before you begin committing individual elements of a vector, which may add latency.

Grading:

- Partially correct reason [-1]
- Significantly incorrect or incomplete [-2]

7. Memory Consistency and Synchronization [13 pts]

Consider the following code for implementing producer–consumer communication using FIFOs on a pair of cores in a shared memory system. One core is the producer, and the other core is the consumer. The cores support a weak memory model. Assume that the FIFOs can be infinitely long.

```
      # Producer
      # x1 has the address of the tail pointer
      # x2 has the data we are appending to the FIFO

      lw x3, 0(x1)      # Load the tail pointer
      sw x2, 0(x3)      # Append new data to the tail
      addi x3, x3, 1
      sw x3, 0(x1)      # Store the new tail pointer

      ##########################

      # Consumer
      # x1 has the address of the head pointer
      # x2 has the address of the tail pointer

      lw x3, 0(x1)      # Load the head pointer
spin: lw x4, 0(x2)      # Load the tail pointer
      beq x3, x4, spin  # Spin if FIFO is empty
      lw x5, 0(x3)      # Load data from FIFO
      addi x3, x3, 1
      sw x3, (x1)       # Store new head pointer
      # then process x5
```

The exam had a typo where a comment said that the consumer stored a new *tail pointer* rather than the *head pointer*. The typo was in the comment, rather than in the code.

(a) Briefly explain what could go wrong that would prevent the communication from happening correctly.

(b) What changes (additions, deletions, rearrangments) could you make to fix the code to guarantee correct behavior? (Modify the above code to demonstration your approach.)

Now assume that we have *multiple consumers*.

(c) Briefly explain what could go wrong that would prevent the communication from happening correctly with multiple consumers.

(d) Describe in a detailed way how you would modify the code to guarantee correct behavior with multiple consumers, using an atomic compare-and-swap instruction. You do not have to show exact code.

8. Warehouse Scale Computing (WSC) [10 pts]

(a) Suppose you run a WCS and one of your workloads has 1000 threads and you run those threads on 1000 seperate servers. You measure the *reliability* of this computation as 0.99. Assume the reliability is the probably of sucessful completion, i.e. no fault occurs during execution.

You would like to improve the reliability by using more servers and decide to redundantly run each thread on two seperate servers. Assuming all faults are i.i.d. (independent and identically distributed), what would be the new reliability?

Without a calculator, it is difficult to reduce this to a single number, so leave your answer in the form of an expression. Explain your work.

Hint: Think about the reliability of a single server that would lead to the reliability for the entire collection of threads, and then how doubling up the thread execution effects the reliability of each thread.

(b) Your California WSC has a total of 1000 servers along with a collection of switches and other IT equipment such as load balancers. A server requires 100 Watts. Assume that besides the servers all other IT equipment has insignificant power draw.

Your average Power Utilization Efficiency (PUE) = Total facilty power / IT equipment power = 1.1.

Pacific Gas and Electric charges you \$0.1 per Kilowatt-hour.

In expression form, how much does it cost per week to run your WSC (assuming all equipment runs 24 hours per day and 7 days per week)?

(c) Suppose you have a large physics simulation, such as solving a system of partial differential equations (PDEs). You would normally run such problems on a supercomputer, but are considering running it on a WSC. What are the factors that might limit performance?

9. FPGAs [6 pts]

Suppose you are the VP of Engineering at a electric scooter company and you need to decide which implementation technology to use for your on-board controller chip for the scooter. You've done the logic/gate level design and now are deciding between designing an ASIC or using an off-the-shelf FPGA. The appropriate FPGA parts are available for $10 each. The same function on an ASIC will cost you $1 per chip, however designing the ASIC will incur significant up-front engineering costs (NREs). For the lifetime of this product you expect to build and sell 100K units. From experience you know that you can map your design to an FPGA for $100,000. What would be the most that you could spend on ASIC NREs to justify designing an ASIC? Show your work.

Besides this cost analysis, what other factors might come into your decision?