Name: SOLUTIONS

<table>
<thead>
<tr>
<th>Problem</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem I</td>
<td>190</td>
<td></td>
</tr>
<tr>
<td>Problem II</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>Problem III</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>Problem IV</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>Problem V</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>Problem VI</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>Legibility and Name</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>447</td>
<td></td>
</tr>
</tbody>
</table>

Instructions:

1. This is a closed book exam. You are allowed to have two letter-sized cheat sheets.
2. No electronic devices may be used.
3. This exam lasts 3 hours.
4. Clearly indicate your final answer.
5. Please show your work when needed.
6. Please write your initials on every odd page.
7. Please make sure that your answers to all questions (and all supporting work that is required) are contained in the space provided.

- **Be cognizant of time.** Do not spend too much time on one question.
- **Be concise.** You will be penalized for verbosity.
- **Show work when needed.** You will receive partial credit at our discretion.
- **Write legibly.** Show your final answer.
1. Potpourri (190 Points)

1) Superscalar and Out-of-Order Processing (10 Points)

Please explain the following concepts, each in no more than 20 words:

Superscalar processing: A superscalar processor can fetch, execute and retire more than one instruction per cycle.

Out-of-order execution: An out-of-order processor executes instructions out of program order to tolerate long or variable instruction latencies.

Please answer the questions below:

(a) Are the concepts of superscalar processing and out-of-order execution orthogonal? **YES**

Why or why not? Explain and justify in no more than 20 words.

Execution order and execution width are independent.

More detail: an out-of-order processor can be built with a machine width of 1. A superscalar processor can also be built that executes in-order.

(b) Assume your friend at a processor design company designed a 1-instruction-wide processor with out-of-order execution. Every instruction in this machine takes a single cycle.

What would you suggest to your friend to simplify the design of the above processor?

Execute in-order. Out-of-order execution will not improve performance because instructions have fixed latency and never stall.

2) Out-of-Order Execution (10 points)

Assume the execution latency of the longest-latency instruction in a 4-wide superscalar, out-of-order machine implementing Tomasulo’s algorithm is 1000 cycles.

How large should the instruction window be such that the decode of instructions does not stall in the presence of this longest-latency instruction?

4000 instructions.

More detail: 1000 cycles/stall × 4 instructions to buffer every cycle ⇒ Need a 4000-instruction-entry window.

Assume we would like to sustain 4 executed instructions per cycle, each with two source registers at most, and we have 4 general-purpose functional units. How many total tag comparators are needed in the reservation stations to support this in this machine? (Each tag comparator can perform only one comparison per cycle.)

32000 (8 tag comparators per reservation station × 4000 reservation stations).

More detail: there will be four tag broadcast buses (one from each functional unit) and each reservation station must have one tag comparator per source register per result bus.
3) Register versus Memory Dependences (10 Points)

What is the fundamental difference between dependencies through registers and dependencies through memory? Explain in no more than 20 words.

Register dependences are statically known. Memory dependences are dynamically determined.

More detail: register dependences are statically known because register numbers are fixed in each instruction, and so can be known at decode time. Memory dependences are dynamically determined because load/store addresses are computed at runtime, hence are not known until the load/store is executed.

4) DRAM Chips (5 Points)

In class we discussed that the row address and the column address are sent at different times to the DRAM chip. Why is this so?

To save pins.

More detail: Sending the entire address at once requires number of pins as many as the total number of row+column address bits. Sending row address separately form the column address reduces the number of pins required by multiplexing the same pins for different portions (row and column) of the address.

5) Cache Replacement (15 Points)

In class, we discussed a cache replacement algorithm called Belady’s OPT, published by Belady in 1966. Belady’s OPT chooses the victim block such that it is the block that is going to be referenced furthest in the future by the program.

(a) Suppose you want to design a processor that implements this algorithm. How would you design this processor?

Cannot design it: a processor cannot know the future perfectly.

(b) Is this algorithm optimal in minimizing cache miss rate? Circle one: **YES**  **NO**

Explain briefly why or why not:

OPT chooses a victim block which is the least likely to be referenced in the near future.

(c) Is this algorithm optimal in minimizing the execution time of a program? Circle one: **YES**  **NO**

Explain briefly why or why not:

It is unaware of cache miss latency, which can vary for each block: (i) some misses can be overlapped, and (ii) remote/local caches may have varying access latencies.

See the following paper for a demonstration of why Belady’s OPT is not optimal in minimizing the execution time of a program:

6) VLIW vs. Superscalar vs. Array Processor (20 Points)

(a) What do VLIW, superscalar execution, and array processing concepts have in common?

All three execute multiple operations per cycle.

(b) Provide two reasons why a VLIW microarchitecture is simpler than a “same-width” superscalar microarchitecture:

No need for dependency-checking logic.

Simpler fetch (instructions packed in fixed-length blocks).

Simpler steering of concurrently-fetched instructions to corresponding functional units.

(c) Provide two reasons why a superscalar microarchitecture could provide higher performance than a “same-width” VLIW microarchitecture:

Better packing of instructions in the I-cache, as there is no need for NOPs as in VLIW.

No restriction on alignment of concurrently-fetched instructions to functional units (a restriction that is present in a VLIW instruction).

(d) Provide a reason why a VLIW processor is more flexible than an array processor:

Can execute different operations concurrently (as opposed to the same operation concurrently on different data elements).

(e) Provide a reason why an array processor is simpler than a “same-width” VLIW processor. Explain.

Only decodes one instruction for all concurrently-fetched operations (as the operations are the same).

7) CRAY-1 (5 Points)

In lecture we discussed that, CRAY-1, a fast vector processor, had the fastest scalar unit of its time (circa early 1970s). Why so? Explain in less than 20 words.

Serial bottleneck.

More detail: if the scalar unit is not fast, then serial portions of application code become the bottleneck, and Amdahl’s law limits the speedup that the parallel vector architecture can provide.
8) Bloom Filters (15 Points)

Consider a Bloom filter with a bit vector $v$ consisting of $m$ bits. This Bloom filter uses $k$ hash functions $h_1, \ldots, h_k$ that map any element $e$ to a bit index between 0 and $m - 1$. To represent that an element $e$ is a member of the set, the Bloom filter sets the $k$ bits at indices $h_1(e), \ldots, h_k(e)$ to 1. An element is considered part of the set if and only if all $k$ bits to which the hash functions map the element are set to 1.

(a) Let $m = 8$ and $k = 2$. Define hash functions $h_1(x) = x \mod 8$, and $h_2(x) = \left\lfloor \frac{x \mod 16}{2} \right\rfloor$. Show the contents of the Bloom filter’s bit vector below after we add the elements 8, 9 and 4.

<table>
<thead>
<tr>
<th>Bit Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>

(b) Find an element $e, e \not\in \{4, 8, 9\}$, such that the Bloom filter with the hash functions and bit vector above indicates that $e$ is in the set. Write $e$ below:

17 or 24 (for example). Many answers are possible here.

9) Update vs. Invalidate Based Coherence (5 Points)

In a multiprocessor system with private L1 caches and a shared L2 cache, if shared data is read by all cores every cycle, but written once per 1000 cycles by a single core, what kind of coherence protocol would you use?

UPDATE INVALIDATE

Why?

The data changes rarely and is read often. Hence updates are cheap and invalidations are expensive.

10) Directory Based Coherence (5 Points)

Assume we use the directory based coherence mechanism we discussed in class. Say, for cache block A, the bit vector stored in the directory is all zeros. What does this tell you about the cache block?

The cache block is not present in any cache in the system.

11) Coherence Protocols (20 Points)

Suppose we have a multiprocessor system with 512 processors. Each processor has a 1 Megabyte private writeback cache with 64-byte cache blocks. The main memory size is 1 Gigabyte.

(a) If we implement a snoopy bus based MESI cache coherence protocol, how many bits of state do we need in the entire system for the coherence protocol implementation?
$2^{24}$ bits.

More detail: there are $2^{23}$ cache blocks in total ($2^{20}$ bytes per cache, $2^9$ caches, hence $2^{29}$ bytes in private caches; divided by $2^6$ bytes per cache block). Each cache block requires two bits of state (M, E, S, or I). Hence, $2^{24}$ bits.

Where do these bits reside?

In the private cache tag stores.

(b) If we instead implement a directory based cache coherence protocol *as we discussed in class*, how many bits of state do we need in the entire system for the coherence protocol implementation?

$2^{24} \times 513 + 2^{24}$ bits.

More detail: We still need the MESI state bits in private caches. Then, we must compute the directory storage space. There are $2^{24}$ cache blocks in main memory, and each cache block needs one bit per processor and one exclusive bit (513 bits), hence $2^{24} \times 513$ bits in the directory in total.

Where do these bits reside?


Which of the above two protocols would you choose for this system?

**Snoopy**

**Directory**

Why?

A bus is not scalable to 512 processors, but a directory is.

More detail: Although the snoopy bus-based system has much lower storage requirements, a bus cannot provide enough bandwidth to sustain the demand of 512 processors. Hence, a directory-based system (built using a scalable interconnect) is more appropriate.

12) Multiprocessing (10 Points)

What differentiates a tightly coupled multiprocessor from a loosely coupled one?

The existence of shared memory.

Assume you are designing a loosely coupled multiprocessor. What type of cache coherence protocol would you implement?

None.
Why?

There is no shared memory in a loosely coupled multiprocessor, so the cache coherence problem does not exist.

13) Programmer versus (Micro)architect (15 Points)

Choices made by a computer architect significantly affect the difficulty the programmer experiences in writing programs or the microarchitect experiences in designing hardware. Over the past 27 lectures this semester, we explored many concepts where this tradeoff was evident.

We discussed many techniques which made programmer’s job easier while making hardware designer’s job more difficult. Please list five such techniques below (no more than 5 words for each technique’s name):

- Virtual memory
- Unaligned access
- Hardware cache coherence
- Accelerated critical sections
- Sequential consistency
- Sequential instruction semantics
- Hardware-managed interlocks

14) Topologies (10 Points)

Suppose you would like to connect 625 processors, and you are considering three different topologies: bus, point-to-point network, and mesh.

Describe one disadvantage of each:

A Single Bus: Very high bus contention (with 625 processors)

A Point-to-Point Network: Many individual wires (between every pair of nodes)

A 25x25 Mesh: High complexity

Which one would you choose? Why?

A 25x25 Mesh: it is performance-scalable to 625 processors and not cost-prohibitive.

15) Hot Potato Routing (10 Points)

In class, we discussed the idea of hot potato routing, introduced by Paul Baran in 1962 as a means to handle contention when two packets need to use the same output link. This idea is also called misrouting or deflection routing.
(a) What is the main benefit of hot potato routing?

No buffers are necessary in the routers.

(b) What is the main disadvantage of hot potato routing?

Deflections can increase latency and hence reduce performance.

(c) Why do you think this idea is also called “misrouting”?

A deflection sends a packet further from its destination. Routing’s goal is to send a packet closer to its destination. Hence, a deflection “misroutes” the packet.

16) INV Bits in Runahead Execution (20 points)

(a) What is the purpose of the INV bits in a runahead execution processor?

To identify L2-miss-dependent instructions and remove them from the window without executing them.

(b) Suppose we did not have the hardware budget to have INV bits in a runahead processor. Describe how you would design a runahead processor to deal with the problem INV bits are otherwise used to solve.

Use value prediction for L2-miss-dependent values.

(c) Give one reason why the runahead processor design you propose would have higher performance (if possible) than the runahead processor that has INV bits. If this is not possible, write “impossible”.

With accurate prediction, more load/store addresses might be computed accurately, yielding more useful prefetches.

(d) Give one reason why the runahead processor design you propose would have lower performance (if possible) than the runahead processor that has INV bits. If this is not possible, write “impossible”.

With inaccurate prediction, useless prefetches could result (wasting memory bandwidth), or data-dependent branches could be resolved incorrectly (whereas the branch predictor might have been correct).

17) Runahead Execution (5 Points)

A 1-wide runahead execution processor that incurs 100 cycles on an L2 miss, executes the following code segment:

```
LOOP:
    lw $t1 ← 0($s0) //L2 Miss
    lw $t2 ← 8($t1)
    add $t3 ← $t1, $t2
    lw $s0 ← 0($t3)
    jmp LOOP
```
The first load (lw $t1 ← 0($s0)) is an L2 miss. When this miss becomes the oldest instruction in the reorder buffer, the processor enters runahead mode. How many L2 misses are generated in runahead mode?

Zero.

More detail: all subsequent instructions are dependent on the first load.
II. Parallel Speedup (60 Points)

You are a programmer at a large corporation, and you have been asked to parallelize an old program so that it runs faster on modern multicore processors.

(a) You parallelize the program and discover that its speedup over the single-threaded version of the same program is significantly less than the number of processors. You find that many cache invalidations are occurring in each core’s data cache. What program behavior could be causing these invalidations (in 20 words or less)?

Cache ping-ponging due to (inefficient or poorly-designed) data sharing.

(b) You modify the program to fix this first performance issue. However, now you find that the program is slowed down by a global state update that must happen in only a single thread after every parallel computation. In particular, your program performs 90% of its work (measured as processor-seconds) in the parallel portion and 10% of its work in this serial portion. The parallel portion is perfectly parallelizable. What is the maximum speedup of the program if the multicore processor had an infinite number of cores?

10.

Use Amdahl’s Law: for $n$ processors, $\text{Speedup}(n) = \frac{1}{0.1 + \frac{0.9}{n}}$.
As $n \to \infty$, $\text{Speedup}(n) \to 10$.

(c) How many processors would be required to attain a speedup of 4?

6.

Let $\text{Speedup}(n) = 4$ (from above) and solve:

\[ 4 = \frac{1}{0.1 + \frac{0.9}{n}} \]
\[ 0.25 = 0.1 + \frac{0.9}{n} \]
\[ 0.15 = \frac{0.9}{n} \]
\[ n = 6 \]

(d) In order to execute your program with parallel and serial portions more efficiently, your corporation decides to design a custom heterogeneous processor.

- This processor will have one large core (which executes code more quickly but also takes greater die area on-chip) and multiple small cores (which execute code more slowly but also consume less area), all sharing one processor die.
- When your program is in its parallel portion, all of its threads execute only on small cores.
- When your program is in its serial portion, the one active thread executes on the large core.
- Performance (execution speed) of a core is proportional to the square root of its area.
- Assume that there are 16 units of die area available. A small core must take 1 unit of die area. The large core may take any number of units of die area $n^2$, where $n$ is a positive integer.
- Assume that any area not used by the large core will be filled with small cores.
How large would you make the large core for the fastest possible execution of your program?

4 units.

For a given large core size of $n^2$, then the large core yields a speedup of $n$ on the serial section, and there are $16 - n^2$ small cores to parallelize the parallel section. Speedup is thus $1/(\frac{0.1}{n} + \frac{0.9}{16-n^2})$. To maximize speedup, minimize the denominator. One can find that for $n = 1$, the denominator is 0.16. For $n = 2$, the denominator is 0.125. For $n = 3$, the denominator is 0.1619 (this can be approximated without a calculator: 0.0333 plus 0.90/7 > 0.12 is greater than 0.15, thus worse than $n = 2$.) Hence, $n = 2$ is optimal, for a large core of $n^2 = 4$ units.

What would the same program’s speedup be if all 16 units of die area were used to build a homogeneous system with 16 small cores, the serial portion ran on one of the small cores, and the parallel portion ran on all 16 small cores?

6.4

Speedup is $1/(0.10 + \frac{0.96}{16}) = 6.4$.

Does it make sense to use a heterogeneous system for this program which has 10% of its work in serial sections?

[YES] [NO]

Why or why not?

The serial portion of the program is large enough that speedup of the serial portion with the large core speedup outweighs loss in parallel throughput due to the large core.

(c) Now you optimize the serial portion of your program and it becomes only 4% of total work (the parallel portion is the remaining 96%). What is the best choice for the size of the large core in this case?

A large core of 4 units.

Similar to above, check speedup for a large core of execution speed $n$ units and $16 - n^2$ small cores. Speedup is $1/(\frac{0.04}{n^2} + \frac{0.96}{16-n^2})$. Minimize the denominator to maximize speedup. For $n = 1$, the denominator is 0.104. For $n = 2$, the denominator is 0.1, and for $n = 3$ it is 0.1504. Thus, as before, use a large core of 4 units.

What is the program’s speedup for this choice of large core size?

10

Speedup is $1/(0.04 + \frac{0.96}{12}) = 10$.

What would the same program’s speedup be for this 4%/96% serial/parallel split if all 16 units of die area were used to build a homogeneous system with 16 small cores, the serial portion ran on one of the small cores, and the parallel portion ran on all 16 small cores?

10

Speedup is $1/(0.04 + \frac{0.96}{16}) = 10$. 

Does it make sense to use a heterogeneous system for this program which has 4% of its work in serial sections?

**YES**  **NO**

Why or why not?

Heterogeneous system provides no performance benefit over the homogeneous system but is more complex to design.
III. VLIW (60 Points)

You are using a tool that transforms machine code that is written for the MIPS ISA to code in a VLIW ISA. The VLIW ISA is identical to MIPS except that multiple instructions can be grouped together into one VLIW instruction. Up to $N$ MIPS instructions can be grouped together ($N$ is the machine width, which depends on the particular machine). The transformation tool can reorder MIPS instructions to fill VLIW instructions, as long as loads and stores are not reordered relative to each other (however, independent loads and stores can be placed in the same VLIW instruction). You give the tool the following MIPS program (we have numbered the instructions for reference below):

(01) lw $t0 \leftarrow 0(a0)
(02) lw $t2 \leftarrow 8(a0)
(03) lw $t1 \leftarrow 4(a0)
(04) add $t6 \leftarrow t0, t1
(05) lw $t3 \leftarrow 12(a0)
(06) sub $t7 \leftarrow t1, t2
(07) lw $t4 \leftarrow 16(a0)
(08) lw $t5 \leftarrow 20(a0)
(09) srlv $s2 \leftarrow t6, t7
(10) sub $s1 \leftarrow t4, t5
(11) add $s0 \leftarrow t3, t4
(12) sllv $s4 \leftarrow t7, s1
(13) srlv $s3 \leftarrow t6, s0
(14) sllv $s5 \leftarrow s0, s1
(15) add $s6 \leftarrow s3, s4
(16) add $s7 \leftarrow s4, s6
(17) srlv $t0 \leftarrow s6, s7
(18) srlv $t1 \leftarrow t0, s7

(a) Draw the dataflow graph of the program in the space below. Represent instructions as numbered nodes (01 – 18), and flow dependences as directed edges (arrows).
(b) When you run the tool with its settings targeted for a particular VLIW machine, you find that the resulting VLIW code has 9 VLIW instructions. What minimum value of $N$ must the target VLIW machine have?

$$N = 3$$ (see VLIW program below). If $N = 2$, then the VLIW program must have at least 11 MIPS instructions, and the number of VLIW instructions either stays the same or decreases as width is increased by one MIPS instruction.

(c) Write the MIPS instruction numbers (from the code above) corresponding to each VLIW instruction, for this value of $N$. When there is more than one MIPS instruction that could be placed into a VLIW instruction, choose the instruction that comes earliest in the original MIPS program.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VLIW Instruction 1:</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 2:</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 3:</td>
<td>07</td>
<td>08</td>
<td>09</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 4:</td>
<td>10</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 5:</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 6:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 7:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 8:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 9:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(d) You find that the code is still not fast enough when it runs on the VLIW machine, so you contact the VLIW machine vendor to buy a machine with a larger machine width $N$. What minimum value of $N$ would yield the maximum possible performance (i.e., the fewest VLIW instructions), assuming that all MIPS instructions (and thus VLIW instructions) complete with the same fixed latency and assuming no cache misses?

$$N = 6$$. This is the maximum width of the dataflow graph and results in 7 VLIW instructions (see below). If $N = 5$, then the VLIW program will instead have 8 VLIW instructions. Increasing $N$ further does not allow any more MIPS instructions to be parallelized in wider VLIW instructions.

(e) Write the MIPS instruction numbers corresponding to each VLIW instruction, for this optimal value of $N$. Again, as in part (c) above, pack instructions such that when more than one instruction can be placed in a given VLIW instruction, the instruction that comes first in the original MIPS code is chosen.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VLIW Instruction 1:</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>05</td>
<td>07</td>
<td>08</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 2:</td>
<td>04</td>
<td>06</td>
<td>10</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 3:</td>
<td>09</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 4:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 5:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 6:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 7:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 8:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 9:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(f) A competing processor design company builds an in-order superscalar processor with the same machine width \( N \) as the width you found in part (b) above. The machine has the same clock frequency as the VLIW processor. When you run the original MIPS program on this machine, you find that it executes slower than the corresponding VLIW program on the VLIW machine in part (b). Why could this be the case?

Concurrently fetched instructions can be dependent in a superscalar processor, requiring bubbles in the pipeline to be processed. A VLIW code translator can reorder instructions to minimize such bubbles.

*Note that the superscalar processor is in-order in this question.*

(g) When you run some other program on this superscalar machine, you find it runs faster than the corresponding VLIW program on the VLIW machine. Why could this be the case?

VLIW code must have explicit NOPs; the superscalar processor does not require these NOPs. Higher code density results in a higher I-cache hit rate and lower required fetch bandwidth.
IV. Memory System (30 Points)

A machine with a 4 GB DRAM main memory system has 4 channels, 1 rank per channel and 4 banks per rank. The cache block size is 64 bytes.

(a) You are given the following byte addresses and the channel and bank to which they are mapped:
- Byte: 0x0000 ⇒ Channel 0, Bank 0
- Byte: 0x0100 ⇒ Channel 0, Bank 0
- Byte: 0x0200 ⇒ Channel 0, Bank 0
- Byte: 0x0400 ⇒ Channel 1, Bank 0
- Byte: 0x0800 ⇒ Channel 2, Bank 0
- Byte: 0x0C00 ⇒ Channel 3, Bank 0
- Byte: 0x1000 ⇒ Channel 0, Bank 1
- Byte: 0x2000 ⇒ Channel 0, Bank 2
- Byte: 0x3000 ⇒ Channel 0, Bank 3

Determine which bits of the address are used for each of the following address components. Assume row bits are higher order than column bits:

- Byte on bus
  Addr [ 2 : 0 ]

- Channel bits (channel bits are contiguous)
  Addr [ 11 : 10 ]

- Bank bits (bank bits are contiguous)
  Addr [ 13 : 12 ]

- Column bits (column bits are contiguous)
  Addr [ 9 : 3 ]

- Row bits (row bits are contiguous)
  Addr [ 31 : 14 ]

Note: We did not ask for the cache block offset bits; however, the cache block offset is in bits [5:0].

(b) Two applications App 1 and App 2 share this memory system (using the address mapping scheme you determined in part (a)). The memory scheduling policy employed is FR-FCFS. The following requests are queued at the memory controller request buffer at time t. Assume the first request (A) is the oldest and the last one (A + 15) is the youngest.


These are cache block addresses, not byte addresses. Note that requests to A + x are from App 1, while requests to B + x are from App 2. Addresses A and B are row-aligned (i.e., they are at the start of a row) and are at the same bank but are in different rows.

Assuming row-buffer hits take T time units to service and row-buffer conflicts/misses take 2T time units to service, what is the slowdown (compared to when run alone on the same system) of
i) App 1?

1.

All requests $A + x$ map to one row, and all requests $B + x$ map to another row (both rows are in the same bank), because there are 16 cache blocks/row and in all requests above, $x < 16$. Since the request for cache block address $A$ comes first, the row containing all requested cache blocks $A + x$ will be opened and all of these requests, which are row-buffer hits come first (and will complete in time $1 \times 2T + 15 \times 1T = 17T$). Thus, none of App 1’s requests are ever delayed by requests from App 2, and so they all execute at exactly the same time as they would if App 2 were not running. This results in a slowdown of 1.

ii) App 2?

21/4.

When running alone, App 2’s three requests are a row buffer-closed access (time $2T$) and two row buffer hits (each taking time $T$); thus they complete in $1 \times 2T + 2 \times 1T = 4T$ time. When running with App 1, all of App 1’s requests come first, in time $17T$ (see above). Then App 2's requests execute as in the alone case (row conflict, row hit, row hit) in $4T$ time. Hence App 2’s requests are completed at time $21T$. Slowdown is thus $21T/4T = 21/4$.

(c) Which application slows down more?

App 2.

Why?

The high row-buffer locality of App 1 causes its requests to occupy the bank for a long period of time with the FR-FCFS scheduling policy, denying App 2 of service during that period.

(d) In class, we discussed memory channel partitioning and memory request scheduling as two solutions to mitigate interference and application slowdowns in multicore systems. Propose another solution to reduce the slowdown of the more-slowed-down application, without increasing the slowdown of the other application? Be concrete.

Interleaving data at a sub-row or cache line granularity could reduce the slowdown of App 2 by reducing the row-buffer locality of App 1 which causes the interference.

One possible interleaving scheme that achieves this is shown below:

- Byte on bus
  - Addr [ 2 : 0 ]
- Lower Column bits
  - Addr [ 7 : 3 ]
- Channel bits
  - Addr [ 9 : 8 ]
- Bank bits
  - Addr [ 11 : 10 ]
- Higher Column bits
  - Addr [ 13 : 12 ]
- Row bits
  - Addr [ 31 : 14 ]
This address interleaving scheme interleaves 256 KB chunks across channels. Thus, the longest row hit streak would be 4, as compared to 16 in the original interleaving scheme in part (a), preventing App 2’s requests from being queued behind 16 of App 1’s requests.
V. Transient Faults (45 Points)

In class and in labs, we have implicitly assumed that the circuits in the processor are always correct. However, in the real world, this is not always the case. One common type of physical problem in a processor is called a transient fault. A transient fault simply means that a bit in a flip-flop or register incorrectly changes (i.e., flips) to the opposite state (e.g., from 0 to 1 or 1 to 0) temporarily.

Consider how a transient fault could affect each of the following processor structures. For each part of this question, answer

(i) does a transient fault affect the correctness of the processor (will the processor still give correct results for any program if such a fault occurs in this structure)?

(ii) if the fault does not affect correctness, does it affect the performance of the processor?

Valid answers for each question are “can affect” or “cannot affect”. Answer for performance in a particular situation only if correctness is not affected. When in doubt, state your assumptions.

(a) A bit in the global history register of the global branch predictor:

**Flipped from 0 to 1:**
Correctness: CAN AFFECT  CANNOT AFFECT

If correctness is affected, why?

N/A

Performance: CAN AFFECT  CANNOT AFFECT

**Flipped from 1 to 0:**
Correctness: CAN AFFECT  CANNOT AFFECT

If correctness is affected, why?

N/A

Performance: CAN AFFECT  CANNOT AFFECT

(b) A bit in a pipeline register (prior to the stage where branches are resolved) that when set to ‘1’ indicates that a branch should trigger a flush due to misprediction, when that pipeline register is currently holding a branch instruction:

**Flipped from 0 to 1:**
Correctness: CAN AFFECT  CANNOT AFFECT

If correctness is affected, why?

N/A

Performance: CAN AFFECT  CANNOT AFFECT
Flipped from 1 to 0:
Correctness: CAN AFFECT CANNOT AFFECT

If correctness is affected, why?

A flip to 0 eliminates a flush that was necessary to recover from a misprediction, hence causes incorrect instructions to write back their results. Note that a flip to 1 does not impact correctness because it only causes an extra (unnecessary) flush.

Performance: CAN AFFECT CANNOT AFFECT

(c) A bit in the state register of the microsequencer in a microcoded design:
Flipped from 0 to 1:
Correctness: CAN AFFECT CANNOT AFFECT

If correctness is affected, why?

Such a bit-flip causes the microcode to jump to another state, which will likely cause random/unexpected behavior.

Performance: CAN AFFECT CANNOT AFFECT

Flipped from 1 to 0:
Correctness: CAN AFFECT CANNOT AFFECT

If correctness is affected, why?

See above (same reason for both flips).

Performance: CAN AFFECT CANNOT AFFECT

(d) A bit in the tag field of a register alias table (RAT) entry in an out-of-order design:
Flipped from 0 to 1:
Correctness: CAN AFFECT CANNOT AFFECT

If correctness is affected, why?

Such a flip could result in a random (undesired) result being captured in the register file.

Performance: CAN AFFECT CANNOT AFFECT

Flipped from 1 to 0:
Correctness: CAN AFFECT CANNOT AFFECT

If correctness is affected, why?

See above (same reason for both flips).
Performance: CAN AFFECT  CANNOT AFFECT

(e) The dirty bit of a cache block in a write-back cache:

**Flipped from 0 to 1:**

Correctness: CAN AFFECT  CANNOT AFFECT

If correctness is affected, why?

N/A

Performance: CAN AFFECT  CANNOT AFFECT

**Flipped from 1 to 0:**

Correctness: CAN AFFECT  CANNOT AFFECT

If correctness is affected, why?

Such a flip causes a modified data block to not be written back to memory, thus losing data.

Performance: CAN AFFECT  CANNOT AFFECT

(f) A bit in the application id field of an application-aware memory scheduler’s request buffer entry:

**Flipped from 0 to 1:**

Correctness: CAN AFFECT  CANNOT AFFECT

If correctness is affected, why?

N/A

Performance: CAN AFFECT  CANNOT AFFECT

**Flipped from 1 to 0:**

Correctness: CAN AFFECT  CANNOT AFFECT

If correctness is affected, why?

N/A

Performance: CAN AFFECT  CANNOT AFFECT

(g) Reference bit in a page table entry:

**Flipped from 0 to 1:**

Correctness: CAN AFFECT  CANNOT AFFECT

If correctness is affected, why?
(h) A bit indicating the processor is in runahead mode:

**Flipped from 0 to 1:**
Correctness: CAN AFFECT CANNOT AFFECT

If correctness is affected, why?

A flip to 1 causes the processor to spuriously enter runahead mode. Depending on how the logic to exit runahead mode is implemented, the processor may never exit runahead mode (hence never making forward progress), and even if it does, it will not restart from the correct checkpoint afterward.

Performance: CAN AFFECT CANNOT AFFECT

**Flipped from 1 to 0:**
Correctness: CAN AFFECT CANNOT AFFECT

If correctness is affected, why?

A flip to 0 causes the processor to behave as if it is in normal mode when it should really be in runahead mode, which will result in speculatively executed instructions to update the architectural state (when they should not).

Performance: CAN AFFECT CANNOT AFFECT

(i) A bit in the stride field of a stride prefetcher:

**Flipped from 0 to 1:**
Correctness: CAN AFFECT CANNOT AFFECT

If correctness is affected, why?

N/A

Performance: CAN AFFECT CANNOT AFFECT

**Flipped from 1 to 0:**
Correctness: CAN AFFECT  CANNOT AFFECT

If correctness is affected, why?

N/A

Performance: CAN AFFECT  CANNOT AFFECT
VI. Prefetching (60 Points)

You and your colleague are tasked with designing the prefetcher of a machine your company is designing. The machine has a single core, L1 and L2 caches and a DRAM memory system.

We will examine different prefetcher designs and analyze the trade-offs involved.

- For all parts of this question, we want to compute prefetch accuracy, coverage and bandwidth overhead after the prefetcher is trained and is in steady state. Therefore, exclude the first six requests from all computations.
- If there is a request already outstanding to a cache block, a new request for the same cache block will not be generated. The new request will be merged with the already outstanding request in the MSHRs.

(a) You first design a stride prefetcher that observes the last three cache block requests. If there is a constant stride between the last three requests, it prefetches the next cache block using that stride.

You run an application that has the following access pattern to memory (these are cache block addresses):

A A+1 A+2 A+7 A+8 A+9 A+14 A+15 A+16 A+21 A+22 A+23 A+28 A+29 A+30...

Assume this pattern continues for a long time.

Compute the coverage of your stride prefetcher for this application.

0%

After each group of three requests, a prefetch is triggered due to a detected stride, but the prefetched block is always useless; none of the demand requests are covered by this prefetch.

Compute the accuracy of your stride prefetcher for this application.

0% (see above).

(b) Your colleague designs a new prefetcher that, on a cache block access, prefetches the next N cache blocks.

The coverage and accuracy of this prefetcher are 66.67% and 50% respectively for the above application. What is the value of N?

N = 2.

After (for example) the access to block 14, the prefetcher prefetches blocks 15 and 16. After 15, it prefetches 16 (merged with the prefetch that was already issued) and 17. After 16, it prefetches 17 and 18. Hence, two out of every three demand accesses are covered (66.7%), and half of prefetches are useful (50%).

We define the bandwidth overhead of a prefetcher as

\[
\text{Bandwidth overhead} = \frac{\text{Total number of cache block requests with the prefetcher}}{\text{Total number of cache block requests without the prefetcher}}
\]

What is the bandwidth overhead of this next-N-block prefetcher for the above application?
For every group of accesses to three consecutive cache blocks, two extra blocks are prefetched. For example, cache blocks 14, 15 and 16 are fetched. In addition to these, blocks 17 and 18 are prefetched.

(c) Your colleague wants to improve the coverage of her next-N-block prefetcher further for the above application, but is willing to tolerate a bandwidth overhead of at most 2x.
Is this possible? **YES**  **NO**
Why or why not?

To get better coverage, the prefetcher must prefetch into the next group of 3 strided requests from the previous group, because the full group of 3 is already prefetched by the first. For instance, on an access to A+14, A+15 and A+16 are already prefetched. To improve coverage, A+21 (which is the first of the next group of 3 strided requests) should be prefetched. However, this would require prefetching the four cache blocks in between A+16 and A+21 (A+17, A+18, A+19, A+20). This increases the bandwidth overhead beyond 2x.

(d) What is the minimum value of N required to achieve a coverage of 100% for the above application? Remember that you should exclude the first six requests from your computations.

\[ N = 5 \] (so that A+16 prefetches A+21, then A+21 prefetches A+22, A+23, etc.)

What is the bandwidth overhead at this value of N?

\[ \frac{7}{3} \]

(e) You are not happy with the large bandwidth overhead required to achieve a prefetch coverage of 100% with a next-N-block prefetcher. You aim to design a prefetcher that achieves a coverage of 100% with a 1x bandwidth overhead. Propose a prefetcher design that accomplishes this goal. Be concrete and clear.

1. A prefetcher that learns the pattern of strides: 1, 1, 5, in this case. This can be accomplished by keeping the last three strides and a confidence counter for each pattern of last three strides.

2. A two-delta stride prefetcher could record up to two different strides \( \Delta_1 \) and \( \Delta_2 \), and the number of strides \( \Delta_1 \) that are traversed before a stride of \( \Delta_2 \) is traversed. For this sequence, the prefetcher would learn that \( \Delta_1 = 1, \Delta_2 = 5 \), and that two strides of \( \Delta_1 \) occur followed by one stride of \( \Delta_2 \).