# CS 351 Final Exam Solutions

## Notes:

- You must explain your answers to receive partial credit.
- You will lose points for incorrect extraneous information, even if the answer is otherwise correct.

## Question 1: Short answer [25 points].

a. Define the compiler terms *front end* and *back end*.

*Front end:* The part of the compiler that translates source code to an intermediate internal representation.

*Back end:* The part of the compiler that translates the intermediate internal representation to machine code.

Which one is responsible for architecture-specific optimizations, and why?

Back end: The front end is machine-independent, while the back end translates the machine-independent IR to the architecture-specific machine code.

b. What is a SIMD instruction? (Just stating what it stands for is not sufficient for full credit.)

SIMD ("single instruction, multiple data") has two meanings, either of which earns full credit:

1. An instruction that specifies a single operation to be carried out multiple times on different data, like a vector instruction
2. A specific type of #1 that treats a normal register as a vector of smaller operands, often seen in the media extensions to general-purpose instruction sets.

What kind of parallelism does it exploit?

DLP

c. What is VLIW? (Just stating what it stands for is not sufficient for full credit.)

A VLIW ("very long instruction word") instruction set packages multiple instructions that can execute at the same time into a single word.

What kind of parallelism does it exploit?

ILP

Does it do so statically (at compile time) or dynamically (at runtime)? Explain.

Statically. The compiler is responsible for finding instructions that can execute at the same time; that is, instructions that are independent and which the machine has sufficient hardware to execute at once.

d. What are the sources of overhead for context switching (switching between threads) on a uniprocessor?

The big ones are saving and restoring register state, as well as raising and returning from the exception.

e. What model for inter-processor communication is typically used in clusters? Why?

Message-passing; it matches the physical reality of processors in clusters (connected over a network) better than shared memory.

(Talking about specific distributed programming platforms that abstract this away is also OK.)

## Question 2: Compiler optimizations [20 points].

(a) In MIPS or a higher-level language, show an example of dead code elimination; that is, show the code before and after the optimization is applied.

Before:

```
	if (false) cout << "False!";
	a = 6;
```

After:

```
	a = 6;
```

(b) In MIPS or a higher-level language, show an example of constant propagation; that is, show the code before and after the optimization is applied.

Before:

```
	x = 5;
	y = x + 3;
	z = y * 2;
```

After:

```
	x = 5;
	y = 8;
	z = 16;
```

(c) For the following C code:

```
for (int i=0; i<N; i++) {
     a[i] = b[i] + c[i];
     d[i] = e[i] + f[i];
}
```

Assume the following:

- The arrays are integer arrays, where an integer is 1 word long.
- N is very large.
- The loop counter *i*, the array size *N*, and the base addresses of the six arrays are kept in registers at all times.
- The code runs on a machine whose L1 cache is fully associative and consists of four 4-word blocks.

Name *one* optimization that would improve both the cache hit rate *and* the loop overhead of this code.

Loop unrolling. (Loop fission improves the hit rate but does not improve the loop overhead.)

Show the code after applying this optimization to the extent that cache misses are minimized. You can abbreviate at your own risk – if it's completely clear what you mean, you'll get full credit.

To maximize the hit rate, we can unroll the loop 4 times. This way, when we miss on array[i] and bring in a 4-word block, we will immediately access array[i+1], array[i+2], and array[i+3].

```
for (int i=0; i<N; i+=4) {
        a[i] = b[i] + c[i];
        a[i+1] = b[i+1] + c[i+1];
        a[i+2] = b[i+2] + c[i+2];
        a[i+3] = b[i+3] + c[i+3];
        d[i] = e[i] + f[i];
        d[i+1] = e[i+1] + f[i+1];
        d[i+2] = e[i+2] + f[i+2];
        d[i+3] = e[i+3] + f[i+3];
}
```

## Question 3: Exploiting ILP [25 points].

(a) In MIPS assembly code, show an example of….

…a WAW dependency:

```
ADD $t0, $t1, $t2
ADD $t0, $t3, $t4
(on $t0)
```

…a WAR dependency:

```
ADD $t0, $t1, $t2
ADD $t1, $t3, $t4
(on $t1)
```

…a RAW dependency:

```
ADD $t0, $t1, $t2
ADD $t3, $t0, $t4
(on $t0)
```

(b) Which of the above are true dependencies? Why?

RAW – in a RAW dependency, the first instruction produces a value that is consumed by the dependent instruction. The other "dependencies" are artifacts of multiple distinct values using the same register name. They could be avoided by using a different register mapping.

(c) For the following MIPS code:

```
1) LW $t2, 0($t3)
2) ADD $t2, $t0, $t2
3) SUB $t8, $t2, $t0
4) LW $t4, 0($t5)
5) ADD $t4, $t0, $t4
```

Rename these instructions to physical registers \$p0 through \$p63.

1) LW \$p2, 0(\$p3)
2) ADD \$p12, \$p0, \$p2
   We rename \$p12 because it is a new value, but not \$p2 because we actually need the value produced by instruction (1)
3) SUB \$p8, \$p12, \$p0
   \$t2 gets renamed to \$p12 because we are looking for the value produced by instruction (2)
4) LW \$p4, 0(\$p5)
5) ADD \$p14, \$p0, \$p4
   See note on instruction (2)

(d) Assume an out-of-order processor with a 5-stage pipeline that works as follows:

- Stage 1: Fetch 2 instructions from memory.
- Stage 2: Decode and rename the instructions; choose up to 2 ready instructions to issue to the execution units.
- Stage 3: (identical to MIPS EX stage, except that 2 instructions can be processed at once)
- Stage 4: (identical to MIPS MEM stage, except that 2 instructions can be processed at once. We unrealistically assume a 1-cycle memory access time in all cases.)
- Stage 5: Write back to the physical register file; update the reorder buffer; "officially" complete up to 2 instructions by removing them from the reorder buffer.

Assume that an instruction can start Stage 3 in the cycle immediately after its operands are produced. That is, assume that the issue logic can figure out that an instruction's operands will be ready in time for the next cycle.

In what order will the instructions from Part (c) be issued? (That is, in what order will they reach the EX stage?) List the instructions that will be issued earliest first, and put instructions that will issue during the same cycle on the same line.

Tracing out the execution below (which goes farther than you would have to go to solve this problem):

1 (reaches EX in Cycle 3)
4 (reaches EX in Cycle 4)
2 (reaches EX in Cycle 5)
3, 5 (reach EX in Cycle 6)

Cycle 1: 1 and 2 are fetched
Cycle 2: 3 and 4 are fetched; 1 and 2 are decoded; 1 is ready, 2 waits on \$p2
Cycle 3: 5 is fetched; 3 and 4 are decoded; 4 is ready, 2 waits on \$p2, 3 waits on \$p12; 1 is in the EX stage
Cycle 4: 5 is decoded; 2 will be ready next cycle, 3 waits on \$p12, 5 waits on \$p4; 4 is in EX; 1 is in MEM (and produces its result)
Cycle 5: 3 will be ready next cycle, 5 will be ready next cycle; 2 is in EX (and produces its result); 4 is in MEM (and produces its result); 1 completes and commits
Cycle 6: 3 and 5 are in EX, 2 is in MEM, 4 completes but can't commit
Cycle 7: 3 and 5 are in MEM, 2 completes and commits
Cycle 8: 3 and 5 complete; 3 and 4 commit (only 2 at a time can commit in order)
Cycle 9: 5 commits

## Question 4: Cache coherence [20 points].

For this problem, see the state machine on the final page of this test, which is identical to Figure 9.3.4 in your textbook.

(a)Fill out the following chart for a two-processor system executing the following sequence of instructions and adhering to the cache protocol in the diagram. Assume the following:

- The data from addresses 100 and 104 are initially stored in both processors' caches and marked as clean/shared.
- The cache block size is 1 word.

"Before" and "After" refer to the block's state in cache before the instruction is executed and after it is executed, respectively.

| **Processor: Instruction** | **Hit or miss?** | **Block's state in P0 cache (before)** | **Block's state in P0 cache (after)** | **Block's state in P1 cache (before)** | **Block's state in P1 cache (after)** |
|---|---|---|---|---|---|
| P0: read 100 | Hit | Shared | Shared | Shared | Shared |
| P1: write 104 | Hit | Shared | Invalid | Shared | Modified |
| P0: read 100 | Hit | Shared | Shared | Shared | Shared |
| P1: read 100 | Hit | Shared | Shared | Shared | Shared |
| P0: write 100 | Hit | Shared | Modified | Shared | Invalid |
| P1: read 104 | Hit | Invalid | Invalid | Modified | Modified |

(b) Repeat the problem for a cache block size of 8 words.

| **Processor: Instruction** | **Hit or miss?** | **Block's state in P0 cache (before)** | **Block's state in P0 cache (after)** | **Block's state in P1 cache (before)** | **Block's state in P1 cache (after)** |
|---|---|---|---|---|---|
| P0: read 100 | Hit | Shared | Shared | Shared | Shared |
| P1: write 104 | Hit | Shared | Invalid | Shared | Modified |
| P0: read 100 | Miss | Invalid | Shared | Modified | Invalid |
| P1: read 100 | Miss | Shared | Shared | Invalid | Shared |
| P0: write 100 | Hit | Shared | Modified | Shared | Invalid |
| P1: read 104 | Miss | Modified | Invalid | Invalid | Shared |

## Question 5: Parallel decomposition [10 points].

A message-passing multiprocessor will be used to count the number of zeroes in a very large array (size N). The array is initially in P0's memory, and there are 8 processors in total. It takes X cycles to send/receive A array elements, and it takes Y cycles to do a comparison. Assume that:

- N is a multiple of A
- Sending fewer than A elements still takes X cycles
- The time to increment the counter is negligible.

(a) What is the algorithm for parallelizing this computation? Number each step.

(1) Send N/8 elements from P0 to each of P1…P7

(2) In parallel, have each processor count the number of elements that are zero

(3) Reduction tree:

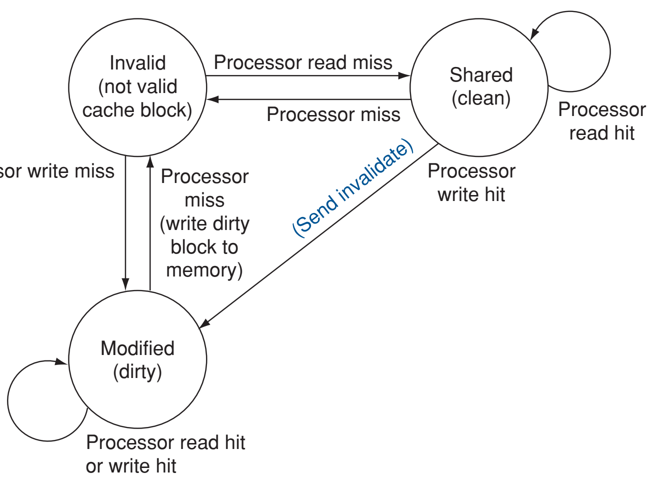
- In parallel, have each odd-numbered processor Px send its count to its neighbor P(x-1), and have the neighbor compute a subtotal
- In parallel, have P2 send its count to P0 and P6 to P4; have P0 and P4 compute subtotals.
- Have P4 send its count to P0, and have P0 compute the final total.

(b) How much time will each step take?

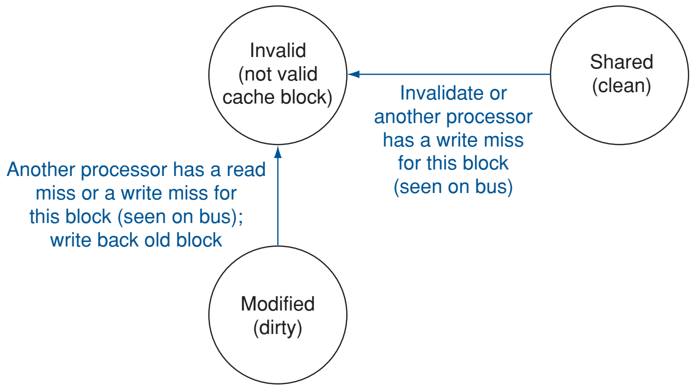
(1) We need to send (N/8) elements from P0 to each of 7 processors. It takes X cycles to send A elements:
7 * X * ceil(N/8A)

(2) Each processor must compare each of its N/8 elements to zero. As stated in the problem, we do not need to worry about the time to increment the counter. All processors do this step in parallel, so we only need the time for a single processor to do this step:
Y * (N/8)

(3) For the reduction tree, the amount of time to compute a subtotal is not specified; full credit was given for either Y or zero cycles. In any event, each of the 3 steps of the reduction tree requires one value to be communicated, which takes X cycles (= X*ceil(1/A)), so the answer is either
3X or 3(X+Y), depending on your assumptions.



a. Cache state transitions using signals from the processor


Invalid (not valid cache block)
Shared (clean)
Invalidate or another processor has a write miss for this block (seen on bus)
Another processor has a read miss or a write miss for this block (seen on bus); write back old block
Modified (dirty)

b. Cache state transitions using signals from the bus

**FIGURE 9.3.4 A write-invalidate cache coherence protocol.** Part a of the diagram shows state transitions based on actions of the processor associated with this cache; part b shows transitions based on actions of other processors seen as operations on the bus. There is really only one state machine in a cache block, although there are two represented here to clarify when a transition occurs. The black arrows and actions specified in black text would be found in caches without coherency; the colored arrows and actions are added to achieve cache coherency.