

Department of Computer Science
Computer Architecture Qualifying Exam
Spring, 2009

The following exam is open book and open notes. You may feel free to use whatever additional reference material you wish, but **no electronic aids** are allowed. Please note the following:

- show your work whenever appropriate. There can be no partial credit unless you show how you derived your answers
- be succinct. You may lose points for facts that, while true, are not relevant to the question at hand

1. (25 points) Some multiprocessor computer system uses a snoopy cache protocol for memory consistency.

The computer's word size, and the line size in its data cache, is 32 bits. Processors P1 and P2 share an array in shared memory declared as follows:

```
int a[100];
```

The data in the array is word-aligned, and the cache is much larger than the array (so conflict misses will not occur). As usual, we follow the convention that a variable has the value 0 the first time it is read.

Processor P1 executes the following code:

```
for (i=0; i < 100; i++) {
    while (a[i] == 0) { /*nothing*/ }
    a[i] = 0;
}
```

Processor P2 executes the following code:

```
for (j=0; j < 100; j++) {
    while (a[j] == 1) { /*nothing*/ }
    a[j] = 1;
}
```

Variables *i* and *j* are private.

- (a) Perhaps surprisingly (given the apparent race condition in the code), it is possible to predict the final values of all the elements of the shared array *a* []. What are they? Explain why.
- (b) Compare the bus traffic from data cache misses you can expect for the following snoopy cache protocols:
 - i. MSI
 - ii. MESI, and reading a cache line which is held in the Modified state by another processor results in the cache line being held in the Shared states in both processors
 - iii. MESI, and reading a cache line which is held in the Modified state by another processor results in the cache line being held in the Exclusive state in the new processor

2. (25 points) Some computer system uses a two-level adaptive branch prediction scheme, as follows:

There is a per-address branch history table (BHT), keeping track of the last eight executions of every branch instruction in the program, with a 1 representing a branch-taken and a 0 representing a branch-not-taken. All of the BHT entries are initialized to 11111111.

The current contents of an address's BHT are used to index a per-address pattern history table (PPHT).

A two-bit saturating counter in each PPHT entry is used for the actual branch prediction: the counter is incremented on a branch-taken, and decremented on a branch-not-taken. A counter value of 10 or 11 predicts taken; a counter value of 00 or 01 predicts not-taken. All the counters are initialized to 10.

This is, of course, the scheme identified as PAp, combined with automaton A2, in the Yeh&Patt branch prediction paper.

The following code is executed:

```
for (i = 0; i < 16; i++)
    if ((i % 2) == 0)
        a[i] = a[i+1];
```

Trace the behavior of the `if` branch predictor on the 16 iterations of the loop. For each iteration, state the contents of the BHT and the relevant PPHT before and after the `if` is executed, and what the prediction is. Assume that **not** executing the body of the `if` requires a branch, while **executing** it does not (while counterintuitive, this is the code that most compilers would generate).

*Note that you are only being asked about the `if` statement. You **don't** need to trace the behavior of the predictor for the `for` loop.*

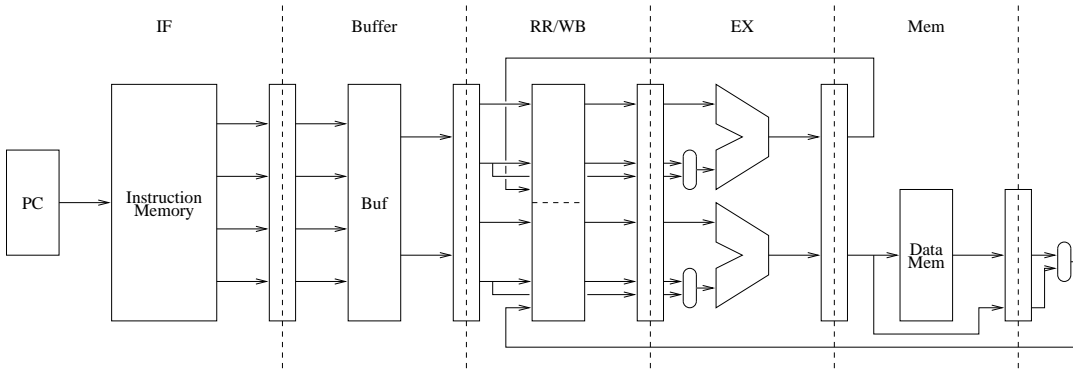
3. (10 points) Suppose you have the following eight bits of data, with four error correction bits and an overall parity bit:

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀	C ₈	C ₄	C ₂	C ₁	P
1	0	0	1	0	1	1	1	0	1	0	1	0

Of course, the bits D_n are the data bits, C_n are the correction bits, and P is the overall parity bit.

If we are using SECDED (single error correction, double error detection), does this data have an error? If so, how many bits? If there is a correctable error, correct it.

4. (40 points) Consider a processor with the following pipeline structure:



The CPU has two pipelines. The top pipeline in the figure can handle any instruction other than memory load/store instructions; the bottom pipeline can handle any instruction. Notice that the top pipeline only has five stages, while the bottom has six. For the sake of simplicity, this problem doesn't consider branches. There is no forwarding between instructions.

The instruction set is a standard three-operand, load-store RISC-style instruction set with no delayed instructions. An arithmetic instruction has the assembly language syntax

```
op $dest, $src1, $src2
op $dest, $src1, immed
```

and a memory instruction has the syntax

```
op $r1, offset($index)
```

where *op* is the operation to be performed, *\$dest* is an instruction's destination register, *\$src1* and *\$src2* are source registers, *immmed* is an immediate operand, *\$r1* is the destination of a load instruction or the source of store instruction, *offset* is a constant offset in a load or store instruction, and *\$index* is an index register in a load or store instruction. You can refer to the registers as \$1, \$2, \$3 (etc) and you don't have to worry about any limits on the number of available registers.

The pipeline stages operate as follows:

- The IF (Instruction Fetch) stage fetches up to four instructions at a time, starting from the address in the PC. The number of instructions fetched is determined by the number of open slots in the buffer (see Buffer stage, next).
- The Buffer stage is able to hold four instructions. Whenever an instruction's data dependencies are satisfied, it can be sent down the pipelines to the RR stage. As many as two instructions can be issued from this stage at a time; when a slot is made available in the Buffer stage, the IF stage can simultaneously fetch an instruction to fill it.

Note that this description implies out-of-order execution: an instruction is sent down the pipeline whenever its dependencies are satisfied, even if another instruction, which occurs first in program order, is still pending.

- The RR (Read Registers) stage decodes instructions and reads registers.
- The EX (Execute) stage performs arithmetic.
- The MEM (Memory) stage performs data memory reads and writes. This stage only exists in the bottom pipe.
- The WB (Writeback) stage writes results back to registers.

Draw a Gantt (timing) chart showing the execution of the following program through this pipeline:

```
load $1, 100($0)
load $2, 200($0)
add $3, $1, 12
add $4, $1, $2
add $5, $3, $4
add $6, $3, $4
store $5, 300($0)
```

The following page of this exam is provided so you can draw the chart clearly. *You are expected to draw the chart clearly enough that timing relationships between instructions are clear and unambiguous. Several copies of the graph are provided so you can start over if necessary.*

