Fall 2008 Computer Architecture Ph.D. Qualifying Exam

[15pts] 1. Performance Analysis

Suppose we have a load-store-architecture computer with a multi-issue CPU that has the following parameters:

- "Perfect" no-stall average CPI: 0.5 (i.e. 2 instructions/cycle)
- Branch misprediction penalty: 8 cycles
- Instruction cache miss penalty: 6 cycles
- Data cache read/write miss penalty with no write-back: 5 cycles
- Data cache read/write miss penalty with write-back: 10 cycles

And an instruction mix of:

Instruction class	Mix %
Arithmetic + Logical	50
Load	30
Store	10
Branch/Jump	10

Note that being a load-store architecture, the only instructions that can perform memory data accesses are the load and store instructions.

With a branch prediction accuracy of 95% (over all conditional and unconditional branches/jumps), an I-cache hit rate of 98%, a D-cache hit rate of 90%, and with 10% of the D-cache misses causing a write-back, what is the actual achieved average CPI?

[35pts] 2. Branching and Pipelining

The main loop for counting the number of values in an integer array that fall between a min and max value might look like (in C/C++):

```
// p is int pointer to beginning of array
// assume array ends with int value of 0
while (*p != 0)
{
   if (*p >= MIN && *p <= MAX)
      count++;
   p++;
}
```

We have a CPU with a 5-stage pipelined architecture whose datapath is shown in the figure following the text of this question. Its features are:

- The stages are: (1) instruction fetch; (2) instruction decode and register access; (3) execute; (4) data memory access; (5) register write back.
- For a memory read instruction, stage 3 computes the effective address, the actual read (4 bytes) is completed in stage 4, and stage 5 writes the value to the destination register.
- Forwarding of values can be done from either of the last two stages to the execute stage (as shown in the datapath diagram).
- Branch decisions are made in stage 3, and can be acted upon in the next cycle. Branches use the ALU for equality comparison and the branch target computation block to compute the target address that will be used if the branch is to be taken. Operands for the branch decision can be forwarded just like all other forwarding (i.e., to the execute stage).
- A static prediction of branch **not taken** (i.e., fall through) is always applied for all branches.
- Registers $t0 to $t15 are available for general use.
- The register $zero is a synonym for the constant 0 (zero).
- The instructions we will use to implement the above loop are:

add r1, r2, r3	r1 = r2 + r3
slt r1, r2, r3	if (r2 < r3) r1 = 1 else r1 = 0
beq r1, r2, label	if (r1 == r2) goto label
bne r1, r2, label	if (r1 != r2) goto label
lw r1, const(r2)	r1 = MEM[r2+const : r2+const+3]

(note that the last instruction loads a word (4 bytes) from memory)

The translation of the C program into this assembly language is:

```
      # assume $t5 has MIN value, $t6 has MAX+1 value
      # assume $t1 initially has address of beginning of array
      # assume $t4 is initially 0 (holds count)
      # $t0 always contains current element (see L0)

      jmp  L0    # start down at loop condition, label L2 is top of loop

L2:   slt  $t2, $t0, $t5
      bne  $t2, $zero, L1
      slt  $t2, $t0, $t6
      beq  $t2, $zero, L1
      add  $t4, $t4, 1
L1:   add  $t1, $t1, 4
L0:   lw   $t0, 0($t1)
      bne  $t0, $zero, L2

      # rest of program below loop continues here
```

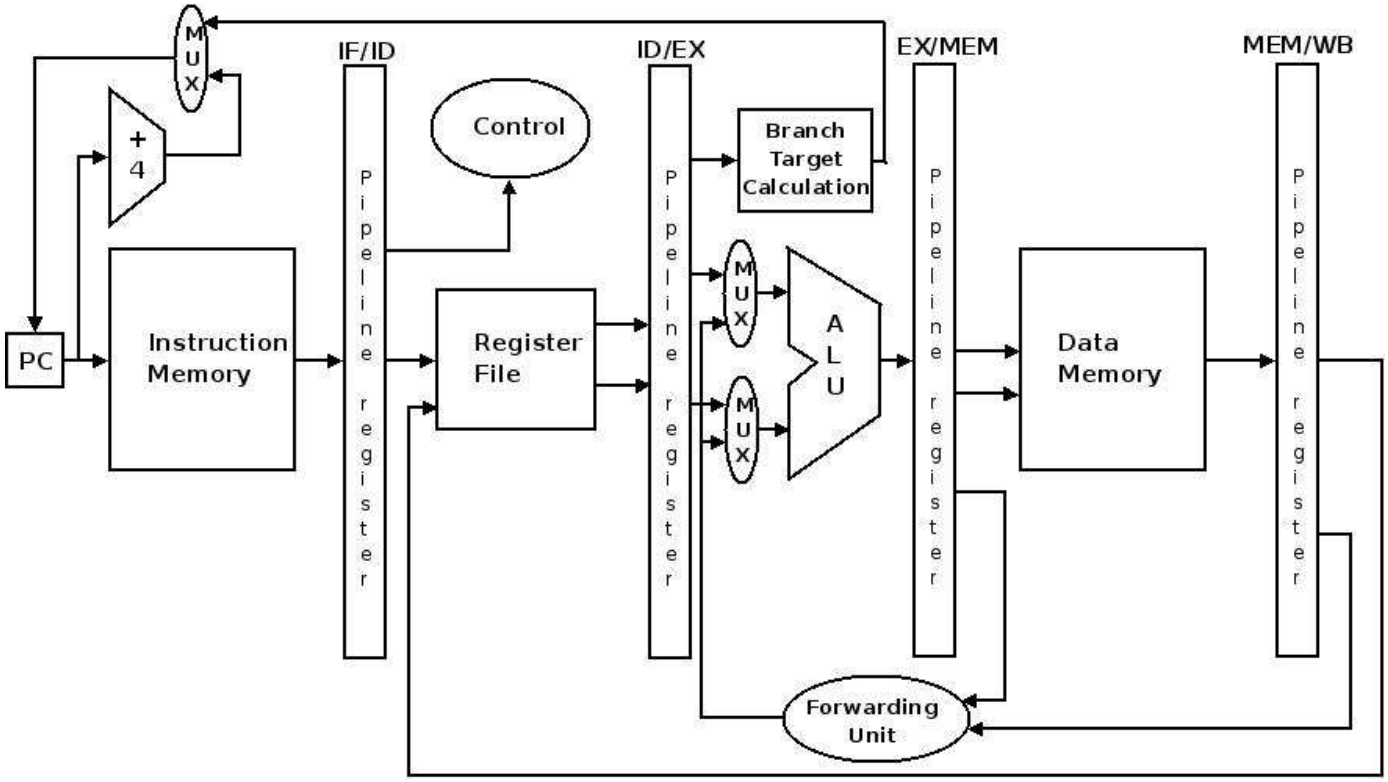
Instructions	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14
L2: slt $t2, $t0, $t5														
bne $t2, $zero, L1														
slt $t2, $t0, $t6														
beq $t2, $zero, L1														
addi $t4, $t4, 1														
L1: addi $t1, $t1, 4														
L0: lw $t0, 0($t1)														
bne $t0, $zero, L2														

[15pts] A. Fill in the pipelined execution of the loop body, cycle by cycle, in the table above. Assume that the loop has iterated at least once before, and that the array value being used for this iteration is **greater** than MAX. Do not erase, but just X out so it is visible, any speculative execution that needs to be thrown away due to an incorrect branch prediction.

[7pts] B. Describe all instances of forwarding that occur in your answer for part A.

[7pts] C. If we had a very large array, and 80% of the values fell between MIN and MAX, 10% were below MIN, and 10% were above MAX, what would be the branch prediction accuracy for predicting not taken on all branches? (answer with a decimal fraction with two significant digits)

[6pts] D. What would the prediction accuracy be if only 20% fell between MIN and MAX, 40% were below MIN, and 40% were above MAX? (again predicting not taken on all branches) (answer with a decimal fraction with two significant digits)


MUX
+
4
PC
Instruction Memory
IF/ID
Pipeline register
Control
Register File
ID/EX
Pipeline register
Branch Target Calculation
MUX
MUX
ALU
EX/MEM
Pipeline register
Data Memory
MEM/WB
Pipeline register
Forwarding Unit

[35pts] 3. Cache and VM Access Simulation

For this problem we have a cache and virtual memory system of the following configuration:

- Addressing: 20-bit virtual and physical byte addresses (20 bits == 5 hex digits)
- Cache configuration:
  - Number of entries (indices): 256
  - Associativity: 2-way
  - Word size: 4 bytes
  - Block size: 4 words
  - Write policy: write back
  - Miss penalty: 10 cycles if no write needed, 12 if write needed
  - Replacement algorithm: LRU
- Virtual memory configuration:
  - Page size: 4 Kilobytes (KB)
  - TLB: 16 entries, fully associative
- VM/Cache interaction: cache is virtually indexed but physically tagged

[5pts] A. What is the total data storage capacity of the cache, in bytes?

[5pts] B. Draw the layout of the caching fields that the address gets divided into, labeling each and indicating their size in bits. Your drawing should have the most significant bit to the left.

[5pts] C. Draw the layout of the VM paging fields that the address gets divided into, labeling each and indicating their size in bits. Your drawing should have the most significant bit to the left.

[20pts] D. A partial TLB and cache are given below, showing their current contents, along with a memory access trace. All address and data values are in hex. The data values in the cache are for each word: assume the upper 3 bytes of each word value are 0x000000. For the cache index 43, the block with the 1a tag is least recently used, and for the index 6b, the block with the 8b tag is least recently used. For the memory access sequence given, fill in whether or not the access results in a cache hit or a cache miss. Update the cache as needed, and whereever a cache block is being evicted, make a note of this off to the right side of the cache table. Fill in the miss penalty where needed, and fill in the data value that was read, where appropriate.

TLB:

Virtual PN	Physical PN
1a	8b
23	9c
7f	34

Memory access trace:

Operation	Address	Value	Hit or Miss	Miss Penalty
write	0x1a430	11		
read	0x1a6b4			
read	0x23438	22		
read	0x1a6bc			
write	0x7f6b4	55		
read	0x2343c	33		
read	0x7f434	44		

Cache:

Index	Tag	Data	Tag	Data
43	8b	12 34 56 78	1a	52 61 73 84
6b	8b	9a 8b 7c 6d	1a	a1 b2 c3 d4

[15pts] 4. Cache Coherence

Assume you have a quad-core CPU whose level-1 caches use a MESI (cache states are Modified-exclusive, Exclusive-unmodified, Shared-unmodified, and Invalid) protocol, as described in the Papamarcos and Patel paper. Further, assume that when one core reads a cache block that another currently has in the Modified-exclusive state, the new processor gets the block in the Exclusive-unmodified state.

Suppose variables x, y are in the same cache block but variable z is in a different one (with a different cache index). Initially, all cores' level 1 caches are empty, and all variables are initialized to 0.

Further, suppose three cores in the CPU execute the following lines of C code, in the order shown (assume no reordering or optimization):

```
P1: x = z + 1;
P2: y = 3 + z;
P3: z = x;
```

Show the sequence of cache contents and their states for the three cores as this code is executed.