Spring 2020 Computer Architecture Ph.D. Qualifying Exam

[30pts] 1. Performance Analysis

Many modern out-of-order processor cores can issue up to 4 instructions per cycle to their functional core units, which gives a theoretical peak CPI (cycles per instruction) of 0.25 (4 inst/cycle). In this problem we have such a CPU core, with the following parameters:

- "Perfect" no-stall CPI: 0.25 (i.e. 4 instructions/cycle)
- Branch misprediction penalty: 6 cycles
- Instruction cache miss penalty: 8 cycles
- Data cache read miss penalty: 8 cycles
- Data cache write miss penalty: 12 cycles

Our program's dynamic instruction mix is:

Instruction class	Mix %
Arithmetic + Logical	60
Load	15
Store	5
Branch/Jump	20

Note that the only instructions that can perform memory data accesses are the load (read) and store (write) instructions.

[20pts] A. 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 20% of the D-cache misses being write operations, what is the actual achieved average CPI?

> *Base CPI is 0.25*
> *I-cache miss penalty is (1.0-0.98)*8 = .02*8 = .16 CPI for all instructions*
> *Branch mispredict penalty is 0.05*.20*6 = .06 CPI*
> *D-cache loads miss penalty is .1*.8*.15*8 = .096 CPI*
> *D-cache stores miss penalty is .1*.2*.05*12 = .012 CPI*
>
> *So total actual CPI is 0.5+.16+.03+.128+.024 = 0.578 CPI*

[10pts] B. Suppose that when we actually run and measure our program's performance, the CPI that is achieved ends up being 1.5X what you calculated in part A. What would your explanation for this be? (Hint: consider potential hazards)

> *(Not sure if I should give the hint) Apart from the basic architectural delays for cache misses and branch mispredicts, the instructions themselves have data dependencies between them (data hazards), and with a pretty wide issue width (4 instructions), it is highly unlikely that forwarding could alleviate all of the data hazards. So instructions have to wait in queues for their data to be available, and this slows down the program.*

[40pts] 2. Branching and Pipelining

The main loop for a not-very-secure encryption algorithm might look like (in C/C++):

```
// p is int pointer, initialized to beginning of array (data to be encrypted)
// end is int pointer, initialized to array end plus 4 (value to stop p at)
// seed is the encryption code (hash value of the password)
// block is block size (also derived from password)
lastv = seed;
count = 0;
while (p != end)
{
   *p = *p ^ lastv;
   count++;
   lastv = *p;
   if (count % block == 0)
      lastv = seed;
   p++;
}
```

The algorithm uses exclusive-or chaining for a block size, then restarts the chaining from the seed again. This gives two parameters to make it at least a tiny bit secure.

We have a CPU with a 5-stage pipelined architecture whose basic 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).
- A static prediction of branch **taken** (i.e., branch to label) is always applied for all branches. The branch target address is calculated in stage 2 (as shown on the datapath) and can be used to fetch the target instruction without any stalls. The actual branch decision for conditional branches is not made until stage three (using the ALU for comparison).
- 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, const	r1 = r2 + const
xor r1, r2, r3	r1 = r2 ^ r3
mod r1, r2, r3	r1 = r2 % r3
mov r1, r2	r1 = r2
bne r1, r2, label	if (r1 != r2) goto label
sw const(r1), r2	MEM[r1+const] = r2
lw r1, const(r2)	r1 = MEM[r2+const]

(note that the 'sw' and 'lw' instructions store and load words (4 bytes)

The translation of the C program (loop only) into assembly language is:

```
      # assume $t1 holds p, $t2 holds end, $t3 holds seed,
      # assume $t4 holds lastv, $t5 holds count, $t6 holds block size
      # assume $t1-$t5 are already initialized
      # $t0 will be used to hold temp vals: *p and mod result
      jmp  L0             # start down at loop condition
L2:   lw   $t0, 0($t1)     # label L2 is top of loop body
      xor  $t0, $t0, $t4
      sw   0($t1), $t0
      add  $t5, $t5, 1
      mov  $t4, $t0
      mod  $t0, $t5, $t6
      bne  $t0, $zero, L1
      mov  $t4, $t3
L1:   add  $t1, $t1, 4
L0:   bne  $t1, $t2, L2
      # rest of program below loop continues here
```

Instructions	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17
L2: lw $t0, 0($t1)	FE	REG	EX	MEM	WB												
xor $t0, $t0, $t4		FE	REG	(bub)	EX	MEM	WB										
sw 0($t1), $t0			FE	(bub)	REG	EX	MEM	WB									
add $t5, $t5, 1					FE	REG	EX	MEM	WB								
mov $t4, $t0						FE	REG	EX	MEM	WB							
mod $t0, $t5, $t6							FE	REG	EX	MEM	WB						
bne $t0, $zero, L1								FE	REG	EX	MEM	WB					
mov $t4, $t3											FE	REG	EX	MEM	WB		
L1: add $t1, $t1, 4									(FE)	(REG)		FE	REG	EX	MEM	WB	
L0: bne $t1, $t2, L2													FE	REG	EX	MEM	WB

[20pts] A. Fill in the pipelined execution of one iteration of the loop body, cycle by cycle, in the table above. Assume that the loop has iterated at least once before, and that for this iteration the modulus result is equal to zero (a block has ended). 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.

Table is filled in; the three entries in parentheses are speculative work that is thrown away once the misprediction on the branch is realized. Bottom-of-loop branch will predict taken and so will not stall.

[10pts] B. Describe all instances of forwarding that occur in your answer for part A.

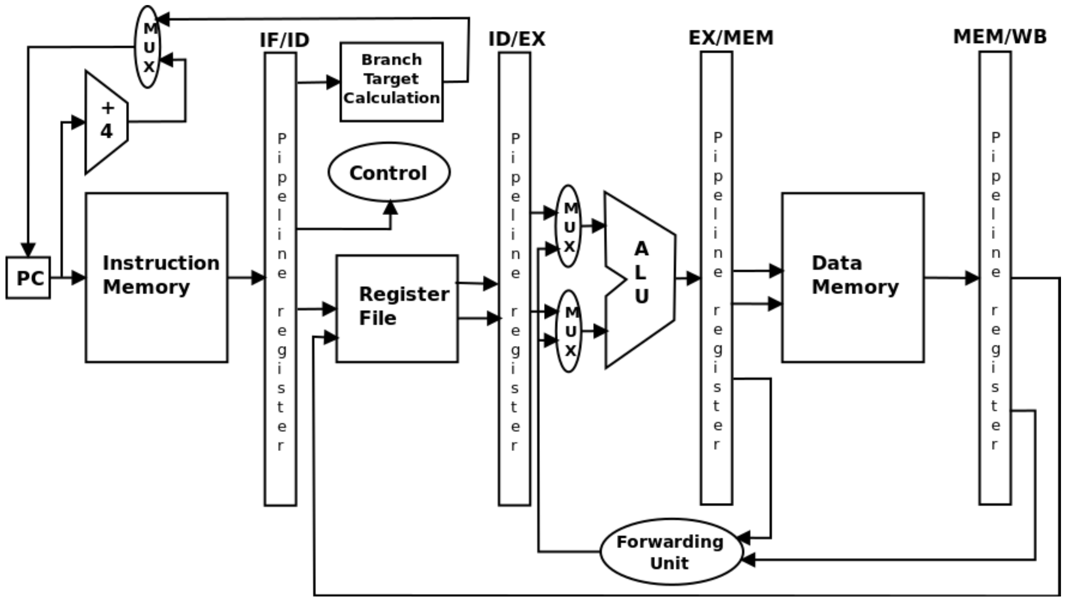
Five: lw->xor (even with stall); xor->sw; add->mod; mod->bne; add->bne

[5pts] C. Describe how a compiler could reorder the instructions to improve this loop's cycle time.

The first add could be moved above the xor to remove the stall bubble.

[5pts] D. If we have 1,000,000 words of data and our block size is 100, what is the average number of cycles per iteration of this loop, for that data?

*In our table, the next iteration of the loop will start in C14, so our iteration is 13 cycles long. But iterations where the modulus is nonzero will only be 10 cycles long. so 99*10+13 = 1003 cycles for 100 iterations, which is an average of 10.03 cycles per iteration.*


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

[30pts] 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 sets (indices): 256
  - Associativity: 2-way
  - Word size: 4 bytes
  - Block (line) size: 4 words
  - 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, with page coloring if needed

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

> *The block size is 16 bytes (4 words, 4 bytes each), and there are two blocks for each of the 256 entries because it is 2-way set associative. So 16*2*256 = 8KB.*

[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.

8 bits	8 bits	4 bits
Tag	Index	Block+word offset

[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.

8 bits	12 bits
Page Number	Page offset

[15pts] D. Assuming the cache is empty (all entries are invalid), provide a sequence of three memory reads (of words) that will cause at least one eviction to occur. All addresses you choose must cause hits in the TLB below. Show the cache set(s) accessed (by index) and the tags that are stored for each block (line).

TLB:

Virtual PN	Physical PN
A5	81
6D	B9
91	2D

> *Answer will have to just pick three addresses whose virtual page numbers are in the TLB and whose index bits select the same cache set, and that are not part of the same cache block. The third one will cause an eviction.*