EECS 470 W15 Final Exam Answer Key

Answers in red. Notes/Explanations in blue.

1. Multiple Choice Questions

a. In a hash cache and skew cache we use some hashing function to determine the index to use when access the cache. Those accesses typically use an XOR. Why don't we generally use some other gate such as a NAND?

- **Because the XOR gate is typically faster than a NAND gate.**
  An XOR gate requires more transistors in its implementation making it slower due to increased capacitance.
- **Because a NAND gate would make some sets more likely to be accessed than others.**
  Unlike an XOR gate, a NAND gate's output is non-uniform. It is more likely to produce a 1 than a 0.
- **Because the XOR gate typically uses less power than a NAND gate.**
  NAND gates consume less power than XOR gates.
- **It's just what people typically use – either XOR or NAND should work equally well.**

b. Say you have a single application on an embedded system that runs every second for 0.9 seconds and is otherwise idle. Assuming it's an option, to best reduce power you should?

- **Turn the processor off (or nearly off) for the 0.1 s when idle.**
  Turning the processor off for the 0.1 seconds will result in a 10% power saving.
- **Scale the voltage and frequency down to about 0.9 times what it is.**
  Scaling the frequency down to 90%, the application will be continuously running to complete its work. Since $P \sim V^2f$ and $V \sim f$, then $P \sim f^3$. So, power will be scaled by a factor of $0.9^3 = 0.729$. This is a 27% power saving.
- **Scale the voltage and frequency down to about 0.729 times what it is.**
  The processor will be unable to complete its work in time under this scaling.
- **Scale the current down to about 0.729 times what it is**

c. You would expect a direct-mapped 128-byte cache with 32-byte cache lines to get a hit about **50%** of the time on a memory access with stack distance of 2.

There are 128 bytes / (32 bytes per cache block) = 4 cache blocks

Stack distance refers to the number of unique memory references made between two successive accesses to the same memory location.

The access pattern (A, B, C, A) has a stack distance of two. For arbitrary A to be evicted and cause a miss on the subsequent access of A, either B or C must alias to the same

cache block as A. The probability that the subsequent A is a miss: P(B evicts A) + P(C evicts A) = 1/4 + 1/4 = 1/2. Therefore, the hit rate is 1 – 1/2 = 50%.

d. The **BTB / gshare / RAS / tournament** predictor has problems correctly predicting the address of **load / store / call / return** instructions.

The BTB struggles with return instructions since return addresses are dynamic and stack-based making them harder to predict than direct branches. A RAS (return address stack) is used to accurately predict returns.

2. Caches and virtual memory

a. Consider the following C-code segment:

```
int X [4096];
for (i=0; i<10000000; i++)
     for (j=0; j<A; j=j+B)
          X[j]++;
```

Assume that all values other than the data in array "X" are kept in registers during this loop. For this code, what would be the expected hit-rate for the various values of A and B if the data cache were 4 KB with 32-byte lines and was two-way associative?

Without loss of generality, let's assume that array X starts at address 0. With a two-way associative 4 KB size cache, we know that for any arbitrary address A, (A + 2048*k) will alias to the same cache set (think about the set index bits of the address to see why this is the case).

(A = 2048, B = 2)

The access pattern of the code will be: 0, 8, 16, 24, …, 8184 (with a read and write at each address). Let's consider the block containing address 0 (we can pick any address). The addresses within the pattern that will alias to the same cache set as 0 are 2048, 4096, and 6144. Since the pattern accesses 4 blocks that all alias to the same 2-way associative set, the block containing address 0 is always a miss on the first access to it in the inner loop. After this initial miss, we get hits for subsequent access to the same 32-byte cache block. For example, although the initial read of address 0 will be a miss, the write to address 0 will be a hit, and so will accesses to address 8, 16, and 24. This results in a overall hit rate of 7/8.

(A = 2048, B = 6)

The access pattern of the code will be: 0, 24, 48, 72, …, 8184 (with a read and write at each address). For the same reason as explained for (A = 2048, B = 2), the first time a block is accessed in the inner loop will always be a miss. After this initial miss, there are three subsequent hits. For example, although the initial read of address 0 will be a miss,

the write to address 0 will be a hit, and so will accesses to address 24. This results in a overall hit rate of ¾.

(A = 1024, B = 2) and (A = 1024, B = 6):

The entire range of the array that is being accessed can fit within our 4 KB cache. Therefore, all accesses will be a hit.

	B=2	B=6
A=2048	7/8	3/4
A=1024	1	1

If you did not consider that each iteration of the inner for-loop does a read and a write, you should get the following:

	B=2	B=6
A=2048	3/4	1/2
A=1024	1	1

b. Consider a memory system with 32 KB pages where the virtual memory size is 4 GB and physical memory addresses are 32 bits. If you have a 32-entry fully-associative TLB, how many bits are:

- Used for the entire TLB tag store (just the tags, not status bits)?
  544 bits

  The tags of a TLB are virtual page numbers. The size of a virtual page number is calculated as (size of virtual address) – (size of page offset).
  Size of virtual address = log2(virtual memory size) = log2(4 GB) = 32 bits
  Size of page offset = log2(page size) = log2(32 KB) = 15 bits
  VPN size = 32 bits – 15 bits = 17 bits
  Total tag store size = (tag size) * (# of TLB lines) = 17 bits * 32 lines = 544 bits

- Used for the entire TLB “data” store?
  544 bits

The "data" of a TLB are physical page numbers. The total data store size is the same as the tag store size since the size of a physical page number equals the size of a virtual page number.

c. Say you are working on a processor that has a 256 KB write-back data cache with 32-byte cache lines that is virtually addressed. In order to avoid problems associated with a virtually addressed cache, you have proposed invalidating the cache on every context switch. Clearly that will make the next process run slower for a while as the cache warms up. But your boss is worried it might make even very short high-priority context switches (a dozen instructions say which need to be done quickly) be very slow in the worst case (more than just due to cache misses!). Why could that happen?

On a context switch, the TLB is also invalidated. Since initial accesses to the virtually addressed cache are misses after the context switch, the processor needs to translate the virtual address to a physical address. Without a valid TLB entry, the processor needs to perform a costly page table walk, greatly increasing latency.

3. Power

a. If we had a processor that used 200W and ran at 2 GIPS about what performance and power consumption would you expect if we moved the voltage and frequency up by 4%?

Given that $P \sim V^2 f$, increasing frequency and voltage by 4% (or a factor of 1.04) means that power will increase by a factor of $1.04^3 = 1.125$. So, P = 1.125 * 200 W = 225 W.

With a 4% faster clock, we can perform 4% more work. Performance = 1.04 * 2 GIPS = 2.08 GIPS.

Power:	225 W
Performance:	2.08 GIPS

b. Consider an ALU and a cache both of which draw 2 Watts of power. If you scale the frequency and voltage up by 5%, which of the two devices would be likely to draw more power? Explain your answer.

The higher switching activity of an ALU compared to a cache makes it more susceptible to increases in dynamic power consumption as voltage and frequency are scaled.

4. Efficient SMT Branch Predictor

An academic paper entitled "Exploring Efficient SMT Branch Predictor Design" concluded that:

> "… branch prediction accuracy is less important in an SMT system than in a traditional superscalar processor. Misprediction ratios that increased by factors of two or three only caused slowdowns of as little as 2%."

If a single-thread running on the same processor would see a slowdown of 30%+ given the same misprediction rates on the same benchmarks, what could explain why the multi-threaded applications were largely unaffected by the very poor misprediction rates?

When a branch misprediction occurs on a processor running a single thread, the pipeline must be flushed, and the correct instruction path must be fetched and executed. Flushing the pipeline causes significant slowdowns as resources idle with no instructions to execute. On a multithreaded system, when one thread incurs a branch misprediction and stalls, other threads can continue to execute and utilize processor resources.

5. VIPT Caches

You work at "Cool Computers Inc" and have been tasked with evaluating replacements for your company's current 4 KB direct-mapped VIPT L1 data cache. The virtual memory system you use has 32-bit physical and virtual addresses and uses a 4 KB page size. Assuming you must stay with the same virtual memory system and continue to use a VIPT cache, label each of the following options as being "viable" or "non-viable". For those you label as non-viable, provide a brief explanation as to why it isn't viable.

When determining viability, we need to consider VIPT cache aliasing. That is, if we have two different virtual addresses map to the same physical address, we must ensure that the index bits extracted from the virtual addresses are the same. If they weren't the same, then we could have the same physical data in two different sets of the cache (sadness). We can guarantee that the index bits will be the same by ensuring that the page offset bits encompass the index bits and block offset bits. Mathematically:

Page offset bits >= block offset bits + set index bits

Doing some rearranging and algebra:

log2(page size) >= log2(block size) + log2(number of sets)
log2(page size) >= log2(block size) + log2(cache size / (associativity * block size))
log2(page size) >= log2(block size * cache size / (associativity * block size))
log2(page size) >= log2(cache size / associativity)
Page size >= cache size / associativity
Page size * associativity >= cache size

a. 4-way associative 16 KB cache.
   Viable

b. 2-way associative 4 KB cache.
   Viable

c. Direct-mapped 4 KB hash cache.
   Not Viable

   A hash cache will apply a hash function on the address bits to determine what set to index to. Since the virtual page number is used to determine the set, aliasing is a problem regardless of the cache size.

d. Direct-mapped 8 KB cache.
   Not viable

   8 KB cache size is not less than or equal to (4 KB page size * 1-way associativity).

6. Cache Behavior

You've been working on a high-end embedded-systems design (1GHz processor, 1GB memory). During development the data cache has been kept turned off (to ease debugging). When the cache is turned on, you find that performance drops by a factor of 3. You know that the cache works correctly and greatly improves performance of other applications running on the same system. You know that you've not made any fundamental error. Why can adding a data cache slow down an application by this much? Your solution must be reasonable given the scenario provided.

Since the cache improves performance on other programs, the performance reduction is a consequence of the access pattern of the target program. The target program has very little spatial and temporal locality, so adding the cache just adds cache lookup latency on top of every memory access.

7. Verilog

Write a Verilog module to arbitrate the output of three functional units to two CDB buses. If a functional unit requests a CDB but none is available then, and only then, you should stall it. Your code must be synthesizeable. You may pick any prioritization that you wish but if two or more functional units want to use the CDB, at least 2 must be able to.

(One of many solutions)

This solution assigns the lowest priority to functional unit 2. Given that there are 2 CDB buses, it is only ever possible for functional unit 2 to stall.

In the case that both functional unit 0 and 1 are not requesting, functional unit 2 is valid on bus 0.

```
module cdb_arb (
       input  logic [2:0]          fu_request_i,       // Functional unit x wants CDB
       input logic [2:0][4:0]      fu_prf_i,           // Which PRF entry unit x has
                                                       // as its destination
       input logic [2:0][63:0]     fu_data_i,          // FU x's result
       output logic [1:0]          cdb_valid_o,        // Data on CDB x is valid
       output logic [1:0][4:0]     cdb_prf_o,          // CDB y's dest. PR entry.
       output logic [1:0][63:0]    cdb_data_o,         // CDB y's result
       output logic [2:0]          fu_stall_o          // Stall the functional unit if
                                                       // it didn't get the CDB the
                                                       // cycle
);

       assign fu_stall_o = {fu_request_i[0] & fu_request_i[1], 2'b0};
       assign cdb_valid_o[0] = fu_request_i[0] | fu_request_i[2];
       assign cdb_valid_o[1] = fu_request_i[1] | (fu_request_i[0] & fu_request_i[2]);


       always_comb begin
              for (int i = 0; i < 2; ++i) begin : cdb_mux
                     cdb_data_o[i] = fu_request_i[i] ? fu_data_i[i] : fu_data_i[2];
                     cdb_prf_o[i] = fu_request_i[i]? fu_prf_i[i] : fu_prf_i[2];
              end
       end

endmodule
```

8. Bus-based MESI [12 points]

Consider a processor that is part of a multi-processor system. Its instruction and data caches are each 16 KB direct-mapped caches with 32-byte cache lines. The data cache is write-back, the instruction cache is write-through. It supports the MESI protocol as described in class. The word size (for instructions and data) is 4 bytes.

Now say you have a program running on that processor. All you know about the program is that it performs 100,000 loads, 20,000 stores and fetches 400,000 instructions (5,000 of which are unique). Answer the following questions. Show your work.

With a 16 KB cache and 32-byte blocks, our cache has 16 KB / 32 bytes = 512 cache blocks. ***(I am unsure about my answers here as I am confused about how to correctly interpret the nature of the loads and stores and whether self-modifying code is allowed)***

Answers to the following can be much less verbose

a. Assuming both caches start with all lines marked invalid and all other processors are idle, what is the smallest number of read lines (BRLs) that could be generated when this program runs?

   In the best case, all 5000 unique instructions are contiguous, and once an instruction is evicted from the cache, it is never fetched again. These 5000 instructions can fit in (5000 instructions) * (4 bytes / instruction) * (1 block / 32 bytes) = 625 blocks. Each block requires 1 BRL.

   For the loads and stores, in the best case, all loads and stores are to the same cache block. So, there is only 1 BRL for all loads/stores.

   Total = 625 (fetches) + 1 (loads/stores) = **626 BRLs**

b. Assuming both caches start with (potentially useful) instructions/data and all other processors are idle, what is the smallest number of read lines (BRLs) that could be generated when this program runs?

   In the best case, all instructions/data currently in the caches are useful.

   With the 512 instruction cache lines already populated, we only need 625 – 512 = 113 additional cache blocks to reach 5000 unique instructions fetched.

   Again, with the load and stores, in the best case they are all to the same cache block. Assuming that cache block is already populated in the data cache, there are 0 BRLs for load/stores.

   Total = 113 (fetches) + 0 (load/stores) = **113 BRLs**

c. Assuming both caches start with (potentially useful) instructions/data and all other processors are idle, what is the largest number of read lines (BRLs) that could be generated when this program runs?

   In the worst case, all instructions/data currently in the caches are not useful.

   Assuming each instruction belongs to its own cache block, 5000 BRLs are a result of fetches.

Assuming each load is to a unique cache block, we have 100,000 BRLs for loads.

Store instructions do not cause a BRL bus transaction. They either are silently upgraded (E→M), trigger a BIL (S→M), or a BRIL (I→M).

Total = 5000 (fetches) + 100,000 (loads) = **105,000 BRLs**

9. Directory vs Bus-Based Systems [12 points]

a. Say we have 4 processors, each of which has a 1MB private writeback L1 cache with 64-byte cache blocks. The main memory size is 4GB (total) and the system uses the MESI coherence protocol as described in class. How many state-bits do we need to maintain the coherence protocol? Where do those state-bits reside? Show your work and ignore any transient states.

There are 2^{17} state bits, distributed amongst the tag stores of each processor's private L1 cache.

Each cache block needs to know its coherence state. With an MESI protocol and ignoring transient states, each block can be in one of four states (requiring 2 bits to encode).

# of cache blocks per processor = 1 MB / 64-byte blocks = 2^{20} B / 2^{6} B = 2^{14} blocks
Total number of cache blocks = 2^{14} blocks per proc * 4 procs = 2^{16} blocks
Number of state bits = 2 state bits per block * 2^{16} blocks = **2^{17} state bits**

b. Now say we are using a directory-based system rather than a bus-based one with the same caches as above. Each of the 4 processors has 1GB of memory (so still 4GB total and the system uses the MESI coherence protocol as described in class. How many state-bits do we need to maintain the coherence protocol? Where do those state-bits reside? Show your work and ignore any transient states (This problem is quite hard).

(2^{26} * 6) total state bits. These state bits are distributed in the directories of the four processors. Directories can be placed in a reserved range of memory addresses.

For the number of state bits, we will assume that we are using a full-map directory. Other approaches including "sparse directories" reduce total overhead.

The problem statement specifies that the 4 GB total of system memory is partitioned equally amongst the 4 processors (1 GB each). For each block of memory in its

partition, a processor must track the block's coherence state and the processor(s) that have a copy of it.

The coherence state of a block is tracked using 2 bits. Keeping track of the processors that have a copy of a particular block can be accomplished using a bit vector with one bit per processor (so 4 bits in this case). So, the total number of bits of state information for each block is 6 bits.

# of memory blocks = 4 GB / 64 bytes per block = 2^{32} B / 2^{6} B per block = 2^{26} blocks
# of state bits = # of memory blocks * state bits per block = **2^{26} * 6 bits**

10. R10K [15 points]

Please look at other exams for solutions to similar problems. Sry :(