



The University of Michigan - Department of EECS

EECS370 – Introduction to Computer Organization

Midterm Exam 2  
November 27, 2007

---

Name: SOLUTION KEY

University of Michigan unickname: \_\_\_\_\_  
(NOT your student ID number!)

Open book, open notes. No laptops, PDAs, cell phones, etc. (calculators are ok). This exam has **8** questions, **11** pages, and **100** points. Questions vary in difficulty; it is strongly recommended that you do not spend too much time on any one question. For questions where a blank line is provided, please write your final answer on the blank line. Follow instructions carefully; some of the questions specify that you ***MUST SHOW YOUR WORK TO RECEIVE CREDIT.***

The rules of the Honor Code of the University of Michigan - College of Engineering apply for this exam.

Honor code pledge: I have neither given nor received aid on this examination, nor have I concealed any violations of the Honor Code.

Signature: \_\_\_\_\_  
(Exams without a signed pledge will not be graded)

### 1. Cache block size (6 points)

Suppose for a given cache, a designer wishes to increase the block size leaving other parameters like associativity and cache size same, which of the following will be true? (circle **ALL** that apply)

- (a) Compulsory misses will reduce or stay the same
- (b) Conflict misses will reduce
- (c) In case of miss, transfer time to get the block will not decrease
- (d) Hit-time will increase

### 2. Cache address bits (9 points) *(Show your work and write final answers on the blanks)*

Consider a byte addressable architecture with a 48 bit address. A 16-way set associative cache with a 128 byte block size and a total cache data size of 4 megabytes is used. The cache is write back and write allocate.

How many address bits are used for the tag? 30

How many address bits are used for the set index? 11

How many address bits are used for the block offset? 7

$$\text{Block offset} = \log_2(\text{block size}) = \log_2(128) = 7$$

$$\# \text{ of sets} = \text{cache size} / (\text{block size} * \text{associativity}) = 2^{11}$$

so 11 index bits

$$\text{tag bits} = \text{total address bits} - (\text{index} + \text{block offset}) = 30$$

**3. Cache performance (10 points)** (*Show your work to receive credit*)

(a) Suppose that accessing a cache takes 10ns while accessing main memory in case of cache-miss takes 100ns. What is the average memory access time if the cache hit rate is 97%?

$$\text{AMAT} = \text{cache access time} + \text{miss-rate} * \text{main memory access}$$

Or

$$\text{AMAT} = \text{hit-rate} * \text{cache-access time} + \text{miss-rate}(\text{cache} + \text{memory access time})$$

$$\text{AMAT} = 10 + .03 * 100 = 13 \text{ ns}$$

(b) To improve performance, the cache size is increased. It is determined that this will increase the hit rate by 1%, but it will also increase the time for accessing the cache by 2ns. Will this improve the overall average memory access time?

$$\text{AMAT} = 12 + .02 * 100 = 14 \text{ ns}$$

Not a good trade-off

Point allocation:

8-9 points if calculation mistake, depending on mistake at what stage

6-7 points if not addressed both increase in hit rate and access time correctly

5 points if not accounted for cache accesses for missed accesses.

**4. Pipelined datapath performance (15 points)** (*Show your work to receive credit*)

(a) Calculate the CPI for a 5-stage in-order pipelined processor as described in class. Branches are resolved in the MEM stage, and the branch prediction accuracy is 60%. Assume perfect instruction and data caching behavior, and that 20% of the lw instructions are followed by an instruction dependent on the result of that load. The instruction mix is as follows:

- 40% R-type instructions
- 25% lw instructions
- 15% sw instructions
- 20% branching instructions

**3 points for branch mispredict penalty**  
**3 points for lw followed by dependent instruction penalty**  
**6 points total**

CPI:   1.29  

(b) Instruction and data caches are added to the processor described in part (a) above. The instruction cache has a hit rate of 95% and the data cache has a hit rate of 60%. The caches can be accessed in a single cycle. Main memory takes 25 clock cycles to access (this is in addition to the cache access time). The processor must stall for all cache misses. Calculate the CPI for this processor (with the same instruction mix as described in part (a)).

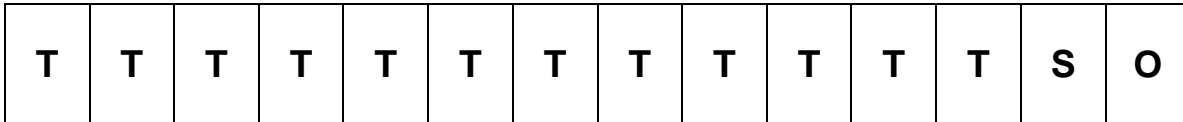
**3 points for instruction cache miss penalty**  
**3 points for data cache miss penalty (lw)**  
**3 points for data cache miss penalty (sw)**  
**9 points total**

CPI:   6.54

### 5. Cache simulation (15 points)

Consider a 14-bit byte-addressable architecture with a 16-byte cache. The cache is a write-back, 4-way set associative cache with a block size of 2 bytes.

a) In the diagram below, mark each bit position of an address to indicate how the address will be partitioned when accessing the cache: for each bit, mark T for tag, S for set index and O for block offset.



b) Now that you have determined how to extract tag, set index and offset from a memory address, let's work on the organization of our cache. We have 16-bytes available, shown in the schematic below. We want the bytes within a block to be adjacent, and the blocks within a same set to be also adjacent.

Please assign a distinct number ID (0, 1, 2, ... ) to each **block** of the cache, and mark each byte in the cache with the block ID it belongs to. Then assign a distinct character ID (a, b, c, ... ) to each **set** of the cache, and mark each block with the set ID it belongs to. To get you started, we filled some of the answers for you.

Byte	Block ID	Set ID
0	0	a
1	0	a
2	1	a
3	1	a
4	2	a
5	2	a
6	3	a
7	3	a
8	4	b
9	4	b
10	5	b
11	5	b
12	6	b
13	6	b
14	7	b
15	7	b

c) We just came up with a new replacement policy for caches that is going to change cache design forever. In our replacement policy, we want to exploit spatial locality instead of temporal locality. Thus, instead of using an LRU policy, we will implement the new **MFA (most far away) policy**. With an MFA policy, the block that gets evicted from a cache is the one whose address is the most far away from the new block that is being inserted in the cache.

Consider again the cache that we just designed (with the same structure that you determined in part b) and indicate the entries that are modified after each of the load operations indicated below, by writing the new address value for each modified entry in the corresponding column of the table below. Assume that values already in the cache had been entered in index order: that is, the entry at index 0 is the oldest, and the entry at location 15 is the most recent.

- 1) lb \$2, M[0x1c12]                      2) lb \$1, M[0x3211]                      3) lb \$3, M[0x1b1b]  
 4) lb \$4, M[0x1c36]                      5) lb \$2, M[0x1b1a] (**HIT**)

			After instr. 1	After instr. 2	After instr. 3	After instr. 4	After instr. 5
Byte	V	Address	Address	Address	Address	Address	Address
0	1	0x34ac					
1	1	0x34ad					
2	1	0x0000		<b>0x3210</b>			
3	1	0x0001		<b>0x3211</b>			
4	1	0x2a00					
5	1	0x2a01					
6	1	0x03d8					
7	1	0x03d9					
8	1	0x1a32					
9	1	0x1a33					
10	1	0x35aa			<b>0x1b1a</b>		
11	1	0x35ab			<b>0x1b1b</b>		
12	0		<b>0x1c12</b>				
13	0		<b>0x1c13</b>				
14	1	0x08de				<b>0x1c36</b>	
15	1	0x08df				<b>0x1c37</b>	

## 6. Cache misses (15 points)

Consider three data caches as follows: cache A is direct mapped, cache B is two way set associative, and cache C is four way set associative. For all three caches, the total cache size is 4 bytes, the block size is 1 byte, LRU replacement is used, and the cache is initially empty. A program runs which does loads from the following sequence of memory addresses:

3, 1, 4, 1, 5, 9, 1, 9, 1, 3

Show what happens on each memory access for each of the three caches running the specified address sequence. For each blank box in the table, if the corresponding reference is a cache hit, write in "HIT". If the reference is a miss, write in one of the three words "CAPACITY", "COMPULSORY", or "CONFLICT" to indicate what type of miss it is. You may use these abbreviations: "CAP" for "CAPACITY", "COMP" for "COMPULSORY", and "CONF" for "CONFLICT". At the bottom of the column for each of the three caches, write in the hit rate percentage for that cache with this sequence of addresses.

Address	Cache A direct mapped	Cache B 2 way set associative	Cache C 4 way set associative
3	<b>COMPULSORY</b>	<b>COMPULSORY</b>	<b>COMPULSORY</b>
1	<b>COMPULSORY</b>	<b>COMPULSORY</b>	<b>COMPULSORY</b>
4	<b>COMPULSORY</b>	<b>COMPULSORY</b>	<b>COMPULSORY</b>
1	<b>HIT</b>	<b>HIT</b>	<b>HIT</b>
5	<b>COMPULSORY</b>	<b>COMPULSORY</b>	<b>COMPULSORY</b>
9	<b>COMPULSORY</b>	<b>COMPULSORY</b>	<b>COMPULSORY</b>
1	<b>CONFLICT</b>	<b>CONFLICT</b>	<b>HIT</b>
9	<b>CONFLICT</b>	<b>HIT</b>	<b>HIT</b>
1	<b>CONFLICT</b>	<b>HIT</b>	<b>HIT</b>
3	<b>HIT</b>	<b>CAPACITY</b>	<b>CAPACITY</b>

Hit rate (%)	<b>20%</b>	<b>30%</b>	<b>40%</b>
--------------	------------	------------	------------

## 7. Branch prediction (15 points):

The following C code computes the sum of the first two odd numbers:

```
int sum = 0, i = 0; /* LC2 register assignments: r2 = sum, r3 = i */
while (i != 4) { /* i in range 0 through 3 */
    if ((i & 1) != 0) { /* if i is odd */
        sum = sum + i; /* i is odd so add it to sum */
    }
    i = i + 1; /* next i in range */
}
```

It is compiled to this LC2K7 assembly language code:

lw	0	1	one	constant 1 in register 1	
lw	0	4	four	constant 4 in register 4	
while	beq	3	4	done	exit loop if i == 4
	nand	3	1	5	r5 = ~(i & 1)
	nand	5	5	5	r5 = i & 1
if	beq	5	0	endi f	skip if block if i & 1 == 0
	add	2	3	2	sum = sum + i
endi f	add	3	1	3	i = i + i
end	beq	0	0	while	back to top of loop
done	halt				
one	.fill	1			
four	.fill	4			

Note that this code has three conditional branches, at the labels "while", "if", and "end". When the code runs, a total of 13 beq instructions are executed.

Two different branch prediction schemes are tried:

- predict never taken
- predict same as previous

For "predict same as previous", a given branch is assumed to do whatever it did on its most recent previous execution of that particular branch; for the first execution of a given branch, it will be predicted not taken.

There is a table on the next page provided. Each row of the table corresponds (in execution order) to one of the 13 beq instructions that is executed. Please write all of your answers for this question in the table.

- (a) In the "Taken?" column, fill in "Y" or "N" next to each beq to indicate whether the particular branch is taken.
- (b) In the "Predict never taken correct?" column in the table, fill in "Y" or "N" next to each beq to indicate whether a "predict never taken" strategy correctly predicts the branch.
- (c) In the "Predict same as previous correct?" column in the table, fill in "Y" or "N" next to each beq to indicate whether a "predict same as previous" strategy correctly predicts the branch.
- (d) Fill in the bottom row with the prediction accuracy percentage for each of the two branch prediction strategies.

<b>Branch</b>	<b>Taken?</b>	<b>Predict never taken correct?</b>	<b>Predict same as previous correct?</b>
while	<b>N</b>	<b>Y</b>	<b>Y</b>
if	<b>Y</b>	<b>N</b>	<b>N</b>
end	<b>Y</b>	<b>N</b>	<b>N</b>
while	<b>N</b>	<b>Y</b>	<b>Y</b>
if	<b>N</b>	<b>Y</b>	<b>N</b>
end	<b>Y</b>	<b>N</b>	<b>Y</b>
while	<b>N</b>	<b>Y</b>	<b>Y</b>
if	<b>Y</b>	<b>N</b>	<b>N</b>
end	<b>Y</b>	<b>N</b>	<b>Y</b>
while	<b>N</b>	<b>Y</b>	<b>Y</b>
if	<b>N</b>	<b>Y</b>	<b>N</b>
end	<b>Y</b>	<b>N</b>	<b>Y</b>
while	<b>Y</b>	<b>N</b>	<b>N</b>
<b>Branch prediction accuracy (%):</b>		<b>6/13 = 46.2%</b>	<b>7/13 = 53.8%</b>

## 8. Single-Cycle Datapath Extension (15 Points)

We want to modify the LC2K7 single-cycle datapath to support a different type of branch instruction. This new branch instruction is BNEZ and it adds to the PC the value of register B if the value of register A is not equal to zero. The BNEZ will replace the BEQ instruction and it will have the opcode 100. It has the same format as the BEQ instruction, with bits 15-0 being unused (0). The BNEZ instruction has the following semantics:

$$\text{if } (\text{regA} \neq 0) \text{ then } \text{PC} = \text{PC} + \text{regB} \text{ else } \text{PC} = \text{PC} + 1$$

Note that if the condition is satisfied, the instruction causes a jump to PC+regB, (not PC+1+regB).

(a) Add all the required hardware extensions that are needed to implement the BNEZ instruction to the datapath in the next page. To do that, fill all the double-border white boxes with the wires or hardware components listed in the two tables in the next page. The only place in the datapath that you can add wires or hardware components is in the provided boxes. If any of the boxes are not needed for implementing this instruction, fill the box with the NN (Not Needed) symbol. Assume that you have an unlimited amount of hardware components available. As an example, the top hardware component box is filled for you with a 2-to-1 MUX and one of its input wire boxes is connected with the additional control signal CTRL1.

(b) Write the values of the control signals in order to execute the BNEZ instruction on the modified datapath. Fill the values in the single-border white boxes at the bottom of the datapath. If any of the control signals is not used, fill the box with the NN (Not Needed) symbol. Make sure that you use “don’t cares” (X) whenever is possible.

