

# EECS 370 Final (Fall 2000, Lecture 1)

## ANSWERS

You will have 1 hour, 50 minutes to work on this exam, which is closed book. You may use a calculator.

There are 4 problems on 11 pages. You are to abide by the University of Michigan/Engineering honor code. Please sign below to indicate that you have abided by the honor code on this exam.

Honor code pledge: I have neither given nor received aid on this exam.

Signature: \_\_\_\_\_

Name: \_\_\_\_\_

Uniqname: \_\_\_\_\_

Problem 1	_____ out of 25
Problem 2	_____ out of 20
Problem 3	_____ out of 15
Problem 4	_____ out of 40
<b>Total:</b>	<b>_____ out of 100</b>

**1. Short Answer (approx. 25 minutes)**

A. Recall that the error correction code (ECC) scheme described in class was capable of correcting a single bit error. Consider an ECC scheme where ECC bits  $x, y, z$  are computed over data bits  $a, b, c, d$  as follows:

$$x = \text{parity}(a, b, c)$$

$$y = \text{parity}(b, c, d)$$

$$z = \text{parity}(a, b, d)$$

If a single bit-flip occurs and results in  $a=1, b=1, c=1, d=1; x=0, y=0, z=1$ , what were the correct values for  $a, b, c, d$ ?

parity bits  $x$  and  $y$  are wrong. This implies that data bit  $c$  is wrong (original data is 1101)

B. Consider a byte-addressed machine with a 32-bit physical address space and a physically addressed cache. The cache is 16 KB in size, 4-way set associative, has 8 byte words, and uses 64 byte blocks. Which bits of the address specify the set number (i.e. the index bits), the tag, and the offset? The least significant bit of the address is bit 0.

bits 5-0 are offset. 16 KB cache ==> 256 blocks ==> 64 sets. bits 11 to 6 make up the index. bits 31-12 are the tag.

C. Consider a computer which is byte addressed, has a 36-bit address space, and has a page size of 4 MB. Each page-table entry occupies 16 bytes. How much memory space would it take to store the maximum-size page table?

$2^{14}$  pages, each  $2^4$  bytes ==> page table is  $2^{18}$  bytes (256 KB)

D. Complete the following analogy:

(TLB is to the page table) as (memory is to the \_\_\_\_\_ Disk \_\_\_\_\_)?

E. Consider a disk that rotates at 12,000 revolutions per minute, has an average seek time of 11 milliseconds, and transfers data off the disk surface at 16 MB/second. What is the average time to read a randomly located 512 byte sector? Your answer should be in milliseconds, and should be accurate to 3 decimal places.

rotation time =  $60 \text{ s} / 12000 = 5 \text{ ms}$  ==> average rotation = 2.5 ms

transfer time =  $512 \text{ bytes} / 16 \text{ MB/s} = .031 \text{ ms}$

total time =  $11 + 2.5 + .032 = 13.531 \text{ ms}$

## 2. Cache Analysis (approx. 20 minutes)

Your EECS 380 program is generating accesses on a processor with a CPU cache. The CPU cache is 2048 bytes in size. The only data structure in your program is an array with 4096 elements. Each array element is 1 byte. There is one input parameter (**stride**) to your program. Your program reads the array according to the parameter stride in the following pattern (yes, this is an infinite loop).

```
byteAddress = 0;
while (1) {
    read array[byteAddress % 4096];
    byteAddress += stride;
}
```

For example, with stride = 4, the access pattern would be: byte 0, 4, 8, 12, 16, ... , 4088, 4092, 0, 4, 8... Ignore the accesses during the first time through the array (i.e. give the miss rate after the program has been running for a very long time and has gone through the entire array many times).

Assume the CPU cache is **direct-mapped**. Fill in each entry of the following table with the CPU cache's **miss rate** for a given combination of stride and number of bytes per block..

Stride	Number of bytes per block		
	1	8	32
1	100%	1/8	1/32
4	100%	1/2	1/8
16	100%	100%	1/2

Now assume the CPU cache is **fully associative and LRU**. Fill in each entry of the following table with the **miss rate** for a given combination of stride and cache size in bytes.

Stride	Number of bytes per block		
	1	8	32
1	100%	1/8	1/32
4	0	1/2	1/8
16	0	0	1/2

### 3. Virtual Memory (15 minutes)

Consider a byte-addressed computer with a 32-bit virtual address. The size of a virtual page is 1 KB. The computer has 16 MB of physical memory. The size of the CPU cache is 128 KB, and the CPU cache is direct-mapped and physical addressed. The TLB has 4 entries and is direct-mapped. The size of a page-table entry is 4 bytes. Your task is to describe the steps needed on this computer to load a byte of data from virtual address ADDR. Assume that no writebacks occur in the TLB, memory, or cache.

Below are four incomplete and **out-of-order** steps involved in getting data from virtual address ADDR to the processor. Label the first step as “START”. When the entire process is done, goto the step “DONE”. Multiple choices are specified as {choice1 | choice2 | choice3}; circle the correct choice. Fill in the blanks (\_\_\_\_\_) with an appropriate equation (you may specify either bit fields or use arithmetic operations).

**Step A.** In the page table entry, look for information that specifies the {virtual | physical} page number. If the page is {invalid | dirty}, {stall the pipeline | trap to the operating system} and load the page from {CPU cache | disk}. Otherwise construct the {virtual | physical} address: {VA | PA} = \_\_\_\_\_. Goto step \_\_\_\_\_.

**Step B.** Read the page table entry from memory for {virtual | physical} page number \_\_\_\_\_. Load {TLB | CPU cache | memory} with the contents of this PTE. Goto step \_\_\_\_\_.

**Step C.** Look in TLB slot \_\_\_\_\_ for {virtual | physical} page number \_\_\_\_\_. If it is in the TLB, read the {page table entry | virtual address | user data} from the TLB and goto step \_\_\_\_\_. If not in TLB, goto step \_\_\_\_\_.

**Step D.** Look in the CPU cache for {virtual | physical} address \_\_\_\_\_. If in the CPU cache, read the {page table entry | user data} from the CPU cache, transfer it to the {memory | processor | TLB}, and goto step \_\_\_\_\_. If not in the CPU cache, load the CPU cache with data from the {TLB | memory | disk} and goto step \_\_\_\_\_.

ANSWER:

Step A: In the page table entry, look for information that specifies the {virtual | physical} page number. If the page is {invalid | dirty}, {stall the pipeline | trap to the operating system} and load the page from {CPU cache | disk}. Otherwise construct the {virtual | physical} address: {VA | PA} = Physical page number of ADDR\*1024 + ADDR%1024. Goto step D.

Step B: Read the page table entry from memory for {virtual | physical} page number ADDR/1024. Load {TLB | CPU cache | memory} with the contents of this PTE. Goto step A.

(START) Step C: Look in TLB slot (ADDR/1024)%4 for {virtual | physical} page number ADDR/1024. If it is in the TLB, read the {page table entry | virtual address | user data} from the TLB and goto step A. If not in TLB, goto step B.

Step D: Look in the CPU cache for {virtual | physical} address PA. If in the CPU cache, read the {page table entry | user data} from the CPU cache, transfer it to the {memory | processor | TLB} and goto step done. If not in the CPU cache, load the CPU cache from the {TLB | memory | disk} and goto step D.

**4. Pipeline Design (approx. 40 minutes)**

A. Consider the following LC-900 assembly-language program (the Project 1 LC-900 description is on page 11 if you need it):

Assembly-language program <b>test1</b>			
(0)	start	lw 0 1	end
(1)		sw 0 1	test
(2)		noop	
(3)		noop	
(4)		noop	
(5)		noop	
(6)		noop	
(7)	test	noop	
(8)		noop	
(9)		noop	
(10)	end	halt	

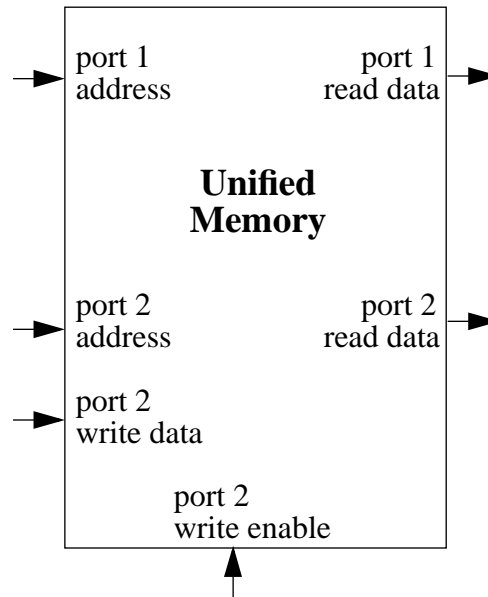
**Describe what this assembly-language program does (according to the LC-900 architecture specified in Project 1). What is the address of the last instruction executed?**

The `lw` loads the machine-code value for the `halt` instruction into register 1, then writes that value into address 7. When it executes address 7, it will execute the newly stored “halt” instruction. The address of the last instruction executed is 7.

**B. The Project 3 pipeline simulator will not execute this program correctly (i.e. its results violate the specification of the LC-900 architecture). Briefly describe what assembly-language `test1` would do when run on the Project 3 pipeline simulator, and explain what causes this erroneous execution.**

`test1` would do the `lw` and `sw`, then halt at address 10. The problem is that the project 3 pipeline simulator had separate memories for instructions and data. The instruction memory is not updated by `sw` instructions, so the instruction at address 7 doesn't get updated.

C. For questions **4C** and **4D**, you will replace the separate instruction and data memories used in Project 3 with a single, unified memory shown below:



Note that the unified memory has two ports, and hence can do two memory operations per cycle. Port 1 can read (it cannot write), and is to be used for fetching instructions during the IF stage. Port 2 can read or write, and is to be used for loading and storing data during the MEM stage. Your pipelines should be as similar as possible to Project 3's pipeline: 5 stages (IF, ID, EX, MEM, WB), forwarding to resolve most data hazards, assume branch-not-taken to resolve branch hazards.

You first try replacing the instruction and data memories from Project 3 with the above unified memory (you make no other changes). **Does this new pipeline correctly execute assembly-language program test1? Justify your answer.**

Yes, this new pipeline correctly executes test1. The sw writes the halt instruction to the unified memory, and this unified memory provides the correct value (ie. the halt instruction) when address 7 is fetched.

**D.** Replacing the instruction and data memory with the unified memory exposes a new class of hazards for assembly-language programs like the one below (test2). Question 4D is about how to modify the pipeline to resolve this new class of hazards.

Assembly-language program <b>test2</b>				
(0)	start	lw	0	1 end
(1)		sw	0	1 test
(2)	test	noop		
(3)		noop		
(4)		noop		
(5)	end	halt		

**Describe the new class of hazards and the circumstances in general that lead to the hazard. In particular, describe the dependency relationship that leads to the hazard.**

The hazard is between the sw data and the subsequent instruction read. When the instruction read is too close to the sw, the fetch reads stale data.

Describe at a high-level how to fix the unified-memory pipeline simulator to **detect** and **resolve** the new class of hazards. Your solution should not slow down programs that have none of the new hazards.

**Which pipeline stage of what instruction detects the new hazard, and what specific condition indicates the new hazard?**

In the sw's MEM (or EX) stage, compare the sw's effective address with the instruction addresses of the instructions behind the sw in the pipeline. If the sw's effective address matches the PC of one of the instructions behind the sw, then there's a hazard.

**What does your solution do when the new hazard is detected?**

Uniqname: \_\_\_\_\_

Cancel the instructions behind the sw (starting at the instruction whose PC matched the sw's effective address), and set the PC to be equal to the sw's PC + 1.

If detect in the EX stage, another solution could be to turn the following instructions into an ID\* and IF\*, and forward the new instruction (generated by the sw) to the input of the ID stage in the next cycle (this is shown in the 2nd pipeline timeline).

Fill out the pipeline timeline below to show how your fixed pipeline would execute assembly-language program test2. Use IF\* and ID\* to indicate stall cycles (as was done in class). If a stage executes but the results (that would have gone to the next pipeline register) are turned into a NOOP, write the stage name and cross it out (e.g. ~~EX~~). Recall that a halt instruction stops the machine when halt reaches the MEM/WB register.

We've provided an extra pipeline timeline as scratch space. Circle the timeline you want us to grade.

Assembly Program	0	1	2	3	4	5	6	7	8	9	10	11
(0) lw 0 1 end	IF	ID	EX	MEM	WB							
(1) sw 0 1 test		IF	ID*	ID	EX	MEM	WB					
(2) test noop			IF*	IF	ID	EX	IF	ID	EX	MEM		
(3) noop					IF	<del>ID</del>		IF	ID	EX		
(4) noop						<del>IF</del>			IF	ID		
(5) end halt										IF		

Assembly Program	0	1	2	3	4	5	6	7	8	9	10	11
(0) lw 0 1 end	IF	ID	EX	MEM	WB							
(1) sw 0 1 test		IF	ID*	ID	EX	MEM	WB					
(2) test noop			IF*	IF	ID*	ID	EX	MEM				
(3) noop					IF*	IF	ID	EX				
(4) noop							IF	ID				
(5) end halt								IF				

For each occurrence of forwarding, write the following line (filling in the 3 specific items between the angle brackets <>):

<pipeline stage name> in cycle <cycle number> gets data from <pipeline register name>

(both solutions) EX in cycle 4 gets data from MEM/WB

(second solution only) ID in cycle 5 gets data from EX/MEM

The LC-900 is an 8-register, 32-bit computer. All addresses are word-addresses. The LC-900 has 65536 words of memory. By assembly-language convention, register 0 will always contain 0 (i.e. the machine will not enforce this, but no assembly-language program should ever change register 0 from its initial value of 0).

There are 4 instruction formats (bit 0 is the least-significant bit). Bits 31-25 are unused for all instructions, and should always be 0.

R-type instructions (add, nand):  
 bits 24-22: opcode  
 bits 21-19: reg A  
 bits 18-16: reg B  
 bits 15-3: unused (should all be 0)  
 bits 2-0: destReg

I-type instructions (lw, sw, beq):  
 bits 24-22: opcode  
 bits 21-19: reg A  
 bits 18-16: reg B  
 bits 15-0: offsetField (a 16-bit, 2's complement number with a range of -32768 to 32767)

J-type instructions (jalr):  
 bits 24-22: opcode  
 bits 21-19: reg A  
 bits 18-16: reg B  
 bits 15-0: unused (should all be 0)

O-type instructions (halt, noop):  
 bits 24-22: opcode  
 bits 21-0: unused (should all be 0)

Instruction	Opcode	Action
add (R-type)	000	add contents of regA with contents of regB, store results in destReg.
nand (R-type)	001	nand contents of regA with contents of regB, store results in destReg
lw (I-type)	010	load regB from memory. Memory address is formed by adding offsetField with the contents of regA.
sw (I-type)	011	store regB into memory. Memory address is formed by adding offsetField with the contents of regA.
beq (I-type)	100	if the contents of regA and regB are the same, then branch to the address PC+1+offsetField, where PC is the address of the beq instruction.
jalr (J-type)	101	First store PC+1 into regB, where PC is the address of the jalr instruction. Then branch to the address now contained in regA. Note that if regA is the same as regB, the processor will first store PC+1 into that register, then end up branching to PC+1.
halt (O-type)	110	Increment the PC (as with all instructions), then halt the machine (let the simulator notice that the machine halted).
noop (O-type)	111	do nothing.