

Sample EECS 370 Final—ANSWERS

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

There are 6 problems on 13 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: _____

Username: _____

Problem 1	_____ out of 25
Problem 2	_____ out of 25
Problem 3	_____ out of 10
Problem 4	_____ out of 15
Problem 5	_____ out of 15
Problem 6	_____ out of 10
Total:	_____ out of 100

1. Pipelining (approx. 25 minutes)

The Project 3 implementation of the LC-900 did not include the jalr instruction. Your job is to modify the Project 3 pipeline to include jalr. Assume the PC is written in the **EX** stage of jalr, and that jalr works much like a branch: instructions are fetched after the jalr, then turned into noops when the PC is written. The PC+1 value is written to regB in the writeback stage, as with most instructions.

Following this question is the Project 1 description of the LC-900 in case you need it.

A. Fill in the pipeline timing diagram below. Assume the jalr jumps to the instruction after the shaded region. Use IF* and ID* in the same way they were used 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~~).

cycle number	0	1	2	3	4	5	6	7	8
jalr 1 2	IF	ID	EX	MEM	WB				
add 3 3 3		IF	ID						
sw 0 4 100			IF						
nand 5 5 5									
add 2 2 2				IF	ID	EX	MEM	WB	

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>

EX in cycle 5 gets data from WB/END

Uniqname: _____

B. Fill in the pipeline timing diagram below. Assume the jalr jumps to the instruction after the shaded region. Use IF* and ID* in the same way they were used 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~~).

cycle number	0	1	2	3	4	5	6	7	8	9	10
lw 0 1 100	IF	ID	EX	MEM	WB						
jalr 1 3		IF	ID*	ID	EX	MEM	WB				
add 2 2 2			IF*	IF	ID						
sw 0 4 100					IF						
nand 5 5 5											
add 6 6 6						IF	ID	EX	MEM	WB	

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>

EX in cycle 4 gets data from MEM/WB

C. During the EX stage of jalr, the PC is written with the target address. Here are the Project 3 typedefs for the pipeline registers. Circle all fields that could be written to the PC during the EX stage of jalr.

```
typedef struct IFIDStruct {
    int instr;
    int pcPlus1;
} IFIDType;
typedef struct IDEXStruct {
    int instr;
    int pcPlus1;
    int readRegA;
    int readRegB;
    int offset;
} IDEXType;
typedef struct EXMEMStruct {
    int instr;
    int branchTarget;
    int aluResult;
    int readRegB;
} EXMEMType;
typedef struct MEMWBStruct {
    int instr;
    int writeData;
} MEMWBType;
typedef struct WBENDStruct {
    int instr;
    int writeData;
} WBENDType;
```

[IDEX.readRegA](#)
[IDEX.pcPlus1](#)
[EXMEM.aluResult](#)
[MEMWB.writeData](#)
[WBEND.writeData](#)

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.

2. Cache Analysis (approx. 25 minutes)

Your EECS 380 program is generating accesses on a processor with a CPU cache. The CPU cache has a block size of 4 bytes. 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 cache size in bytes.

Stride	Cache Size in Bytes				
	512	1024	2048	4096	8192
1	<u>25%</u>	<u>25%</u>	<u>25%</u>	<u>0%</u>	<u>0%</u>
4	<u>100%</u>	<u>100%</u>	<u>100%</u>	<u>0%</u>	<u>0%</u>
16	<u>100%</u>	<u>100%</u>	<u>100%</u>	<u>0%</u>	<u>0%</u>

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	Cache Size in Bytes				
	512	1024	2048	4096	8192
1	<u>25%</u>	<u>25%</u>	<u>25%</u>	<u>0%</u>	<u>0%</u>
4	<u>100%</u>	<u>100%</u>	<u>100%</u>	<u>0%</u>	<u>0%</u>
16	<u>100%</u>	<u>0%</u>	<u>0%</u>	<u>0%</u>	<u>0%</u>

3. Cache Performance (approx. 10 minutes)

Your job is to compute the CPI of a non-pipelined computer with split instruction and data caches (both of which are write-back caches). Cache and memory accesses cannot be overlapped. You are given the following information (not all of which is needed). **Compute the CPI (to 3 decimal places), then check the data in the tables below that was needed to compute CPI.** We will grade on getting the correct CPI and on checking only that information that is needed.

CPI for each type of instruction

check if needed	Type of Instruction	CPI
	R-type	4
	lw	5
	sw	4
	beq	3
	jalr	2

Workload information

check if needed	Type of Instruction	Percent of Workload
	R-type	40%
	lw	25%
	sw	20%
	beq	15%
	jalr	0%

Cache information

check if needed		
	time to read a cache block from memory	11 cycles
	time to write a cache block to memory	12 cycles
	I-cache size	128 KB
	I-cache block size	32 bytes
	I-cache associativity	2
	I-cache replacement policy	FIFO
	I-cache miss rate	5%
	D-cache size	1 MB
	D-cache block size	128 bytes
	D-cache associativity	4
	D-cache replacement policy	LRU
	D-cache miss rate	10%
	% of D-cache accesses that cause write-backs	2%

Uniqname: _____

$$\text{CPI with perfect cache} = 40\% * 4 + 25\% * 5 + 20\% * 4 + 15\% * 3 = 4.1$$

$$\text{CPI penalty due to imperfect I-cache} = 100\% * 5\% * 11 = .55$$

$$\text{CPI penalty due to imperfect D-cache} = 45\% * 10\% * 11 + 45\% * 2\% * 12 = .603$$

$$\text{total CPI} = 5.253$$

4. Virtual Memory Analysis (approx. 15 minutes)

Consider the following computer with a CPU cache and virtual memory system:

- The size of a virtual page is 256 (2^8) bytes. The computer has 1024 bytes of memory.
- The size of the CPU cache is 64 bytes, with a block size of 16 bytes. The CPU cache is direct-mapped and physically addressed.
- The TLB has 2 entries, is fully associative, and uses LRU replacement.
- Page faults are handled as follows: the operating system loads the page into physical memory without changing the CPU cache, then restarts the faulting instruction. If there is more than one free physical page, the page-fault handler uses the lowest-numbered page.

Your job is to analyze the following trace of virtual addresses (all addresses are from lw instructions and are **byte addresses**): 0x4ce0, 0x8ef4, 0x3e38, 0x4ce8. After each memory reference is completed, show the contents of the page table, the TLB, and the CPU cache (only show valid entries). For each memory reference, also state if it is a page fault or not, TLB hit/miss, and CPU cache hit/miss. All answers should be in hexadecimal.

Space for your answers is given on the next 2 pages of the exam.

After address 0x4ce0

page table	Virtual Page #	Physical Page #
	<u>4c</u>	<u>0</u>
TLB	Virtual Page #	Physical Page #
	<u>4c</u>	<u>0</u>
CPU cache	Cache Block #	Memory Block #
	0	
	1	
	2	<u>0e</u>
	3	

Circle the types of misses that were caused by 0x4ce0: page fault, TLB miss, CPU cache miss. Page fault, TLB miss, CPU cache miss

After address 0x8ef4

page table	Virtual Page #	Physical Page #
	<u>4c</u> <u>8e</u>	<u>0</u> <u>1</u>
TLB	Virtual Page #	Physical Page #
	<u>4c</u>	<u>0</u>
	<u>8e</u>	<u>1</u>
CPU cache	Cache Block #	Memory Block #
	0	
	1	
	2	<u>0e</u>
	3	<u>1f</u>

Circle the types of misses that were caused by 0x8ef4: page fault, TLB miss, CPU cache miss. Page fault, TLB miss, CPU cache miss

After address 0x3e38

page table	Virtual Page #	Physical Page #
	<u>3e</u>	<u>2</u>
	<u>4c</u>	<u>0</u>
	<u>8e</u>	<u>1</u>
TLB	Virtual Page #	Physical Page #
	<u>3e</u>	<u>2</u>
	<u>8e</u>	<u>1</u>
CPU cache	Cache Block #	Memory Block #
	0	
	1	
	2	<u>0e</u>
	3	<u>23</u>

Circle the types of misses that were caused by 0x3e38: page fault, TLB miss, CPU cache miss. Page fault, TLB miss, CPU cache miss

After address 0x4ce8

page table	Virtual Page #	Physical Page #
	<u>3e</u>	<u>2</u>
	<u>4c</u>	<u>0</u>
	<u>8e</u>	<u>1</u>
TLB	Virtual Page #	Physical Page #
	<u>3e</u>	<u>2</u>
	<u>4c</u>	<u>0</u>
CPU cache	Cache Block #	Memory Block #
	0	
	1	
	2	<u>0e</u>
	3	<u>23</u>

Circle the types of misses that were caused by 0x4ce8: page fault, TLB miss, CPU cache miss. TLB miss

5. Virtual Memory and TLB (approx. 15 minutes)

Consider a virtual memory system with an 8-page, fully associative physical memory. The system uses a 4-entry, direct-mapped TLB to speed up translation. The TLB contains only translation information (no dirty bits, LRU bits, or protection bits).

Below is an incomplete picture of the current state of the memory and TLB. All page numbers are given in hexadecimal. Fill in the missing entries. If there is more than one possibility, write down all possible entries. If there is not enough information to fill out an entry, note this in the blank. Assume that each TLB entry points to a page that is currently in physical memory.

physical memory page #	virtual page #
0	31
1	5c
2	17
3	
4	22
5	
6	3
7	24

TLB entry #	virtual page #	physical page #
0		
1	29	
2	82	3
3		6

TLB entry 2 ==> physical memory page 3 has virtual page 82
physical page 6 ==> TLB entry 3 has virtual page 3 (because TLB entry 3 has physical page 6)
TLB entry 1 ==> any physical page (but not physical page 3) could have virtual page 29 ==> physical page 5 has virtual page 29
physical page 1 ==> TLB entry 0 could have mapping of (5c, 1)
physical page 7 ==> TLB entry 0 could have mapping of (24, 7)
note: virtual pages 31, 22, and 17 cannot be in TLB entry 0 (since it's direct mapped)

6. I/O Performance (approx. 10 minutes)

You are analyzing the memory performance of a single program running on a processor with a virtual memory system (page size is 4 KB). The disk used by the virtual memory system has the following characteristics:

- constant seek time of 10 milliseconds (i.e. all seeks take 10 milliseconds)
- rotates at 7200 revolutions per minute
- has 2000 cylinders
- stores 80 sectors per track
- has a sector size of 512 bytes

The program issues 100 memory instructions and 1000 R-type instructions. 40 of the 100 memory instructions cause page faults. 15 of the 40 page faults replace a page of data that has been stored to since it was brought into memory.

How much time does this program spend waiting for disk I/O?

$$\text{Total number of disk I/Os} = 40 + 15 = 55$$

$$\text{Rotation time} = 60/7200 = 8.3 \text{ ms} \implies \text{average rotation} = 4.17 \text{ ms}$$

$$\text{Transfer time} = 4 \text{ KB} / 40 \text{ KB} * 8.3 \text{ ms} = 0.83 \text{ ms}$$

$$\text{Time for each disk I/O} = 10 \text{ ms} + 4.17 \text{ ms} + 0.83 = 15 \text{ ms}$$

$$\text{total time waiting for disk I/O} = 55 * 15 \text{ ms} = 825 \text{ ms}$$