



The University of Michigan - Department of EECS

EECS370 – Introduction to Computer Organization

Midterm Exam 1  
October 14, 2008

---

Name: \_\_\_\_\_ **SOLUTIONS** \_\_\_\_\_

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 **5** questions, **12** 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 box is provided, please put your final answer in the box.

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. Short and True/False Questions [32 points]

(a) [4 pts] Convert the following MIPS assembly instruction into binary code:

MOV \$t1, \$s2  
Write your answer in hexadecimal format here:

Many possible answers – here are some examples:

add \$9,\$18, 0                   0x0240 4820  
and \$9, \$18, \$18               0x0252 4824  
addi \$9, \$18, 0                0x2132 0000

Note: it is not possible to use mov.s or mov.d, because floating point move can only operate on floating point registers.

(b) [5 pts] Consider the following C code snippet.

```
void myalignfunct( int param)
{
    char a; //byte address //0-0
    struct { //strb must be aligned as int
        short b; //4-5
        struct { //stra must be aligned as int
            int c; //8-11
            char d; //12-12
        } stra; //13-15 padding
    } strb; //no padding needed
    double e; //16-23

    ... <more C code here> ...
}
```

Note: param is not a local variable.

How many bytes does the allocation of the local variables in this code take in memory?  
Assume that the first variable can be allocated starting at address 0 and that the target processor is a MIPS 32-bit architecture.

24

Bytes

(c) [3 pts] We have crafted an extremely compact floating point representation which uses only 8 bits. The new representation works exactly as the IEEE floating point format that we studied in class. However, the bit allocation is as follows:

- sign – 1 bit
- exponent – 3 bits with bias 3
- mantissa – 4 bits

Which of the following is a base 10 representation of **1100 0000**? Circle all correct answers:

Sign: -    Exponent:  $4-3 = 1$     Mantissa:  $1.0000 = 1$     Answer:  $- 1 \times 2^1$

-192    -65    -12    **-2**    -0    2    12    65    192

(d) [6 pts] The C function below is a recursive implementation of the “McCarthy91” function:

```
int mccarthy(int n)
{
    int tmp;
    if (n>100) return (n-10);
    else {
        tmp = mccarthy(n+11);
        return mccarthy(tmp);
    }
}
```

This program is compiled and executed on a MIPS processor with input **n = 98**. What is the maximum number of stack frames that can be found on the stack during this execution of the function? [Assume that the first stack frame is created with the call to `mccarthy(98)`]

Stack history:

Level 1	Mc(98)	Mc(98)	Mc(98)	Mc(98)	Mc(98)	Mc(98)	Mc(98)
Level 2		Mc(109)	Mc(99)	Mc(99)	Mc(99)	Mc(99)	Mc(99)
Level 3				Mc(110)	Mc(100)	Mc(100)	Mc(100)
Level 4						Mc(111)	Mc(101)

**4**

frames

(e) [3 pts] Indicate whether the following statement is True or False.

Any caller-save register may be freely modified by the callee, even if it has not been saved by the caller.

True

False

(f) [3 pts] Indicate whether the following statement is True or False.

In the case of self-recursive calls, the callee-save registers do not need to be saved and restored across function calls.

True

False

(g) [2 pts] Consider the following C statement.

```
printf("the sum is %d - input %d\n", sum, input);
```

Where is the string "the sum is %d - input %d\n" stored? Circle all correct answers.

heap

stack

global segment

text segment

(see an example in slide 29 – lecture 4)

## 2. LC-2K Assembly [18 points]

Consider the following LC-2K assembly program.

```
0          lw    0    1    foo
1          lw    0    2    n
2          lw    0    3    neg1
3          lw    0    4    stAdd
4  start   beq   0    2    end
5          add   1    1    1
6          add   2    3    2
7  here    jalr  4    5
8          sw    0    1    baz
9          halt
10 end     jalr  5    6
11 foo     .fill  1
12 n       .fill  4
13 neg1    .fill -1
14 stAdd   .fill  start
15 baz     .fill  0
```

(a) [5 pts] List the PC values of the first 10 instructions to be executed in the program, assuming that the program is stored starting at location 0.

PC: 

0	1	2	3	4	5	6	7	4	5
---	---	---	---	---	---	---	---	---	---

(b) [5 pts] How many times is the line labeled **here** executed during execution of this code?

4
---

 times

(c) [4 pts] What value is in memory address 15 when this code is done executing?

Answer:

16

(d) [4 pts] Provide a brief one or two sentence explanation of what the code is doing.

$\text{Mem}[15] = \text{foo} * 2^n$  for  $n > 0$

For full credit, you only need to say that the code computes  $2^n$ .

### 3. Memory Addressing [12 points]

Consider the following MIPS assembly program:

```

0:  lui $1, 1
1:  addi $1, $1, -2
2:  sh $1, 0($2)
3:  lb $2, 1($2)
4:  add $2, $1, $2
5:  lhu $1, 1002($0)

```

Recall that Register \$0 always contains the value zero. The memory is organized in big-endian format, the same format we studied in class. Fill in the following tables based on the instruction sequence above. When a register or memory location does not change write a dash (--) in the corresponding space of the table.

The initial value for Register \$1 is **0x0000AAAA** and for Register \$2 is **0x00001000**.

Reg	After inst 0	After inst 1	After inst 2	After inst 3	After inst 4	After inst 5
\$1	0x0001 0000	0x0000 FFFE	--	--	--	0x0000 2680
\$2	--	--	--	0xFFFF FFFE	0x0000 FFFC	--

Memory Address	Initial value	After inst 0	After inst 1	After inst 2	After inst 3	After inst 4	After inst 5
1000 <sub>10</sub>	0xFF	--	--	0xFF	--	--	--
1001 <sub>10</sub>	0x11	--	--	0xFE	--	--	--
1002 <sub>10</sub>	0x26	--	--	--	--	--	--
1003 <sub>10</sub>	0x80	--	--	--	--	--	--

#### 4. ISA Design [22 points]

You are the Chief Architect at Urban Stackfitters Inc., a company specializing in the production of small stack-based processors. Stack-based processors contain a hardware stack instead of a register file and can only access data in the first and/or second position from the top of the stack. The ISA for the next generation StackAttack processor is documented in the tables below. Data, instructions and memory addresses are all 8-bits long on this processor. The processor has the following instructions:

##### R-type Instructions

bits 7-5: opcode

bits 4-0: unused

Instruction	Opcode	Action
pop	001	Remove the [top] value from the stack.
noop	010	Do nothing to the stack.
halt	011	Halt the processor.

##### I-type Instructions

bits 7-5: opcode

bits 4-0: immediate value (IMM) in 2's complement form

Instruction	Opcode	Action
pushi	000	Push the sign-extended immediate value onto the stack.
beqz	100	Pop the [top] entry from the stack. If it is zero, start executing at PC+1+IMM, where IMM is the sign-extended immediate value. Otherwise, execute the next instruction at PC+1.
load	101	Form a memory address by popping the [top] stack entry and adding it to the sign-extended immediate value IMM. Push the word loaded from memory onto the stack.
store	110	Form a memory address by popping the [top] stack entry and adding it to the sign-extended immediate value IMM. Pop the [top-1] stack entry and store it to that address in memory.

##### A-type Instructions

bits 7-5: opcode

bits 4-3: unused

bits 2-0: ALUOP encoding

**NOTE:** All A-type instructions pop the top two stack entries to use as inputs and push the result onto the stack.

Instruction	Opcode	ALUOP	Action
alu.add	111	000	Add [top] to [top-1].
alu.sub	111	001	Subtract [top] from [top-1].
alu.mult	111	010	Multiply [top] and [top-1].
alu.nand	111	011	Bitwise nand [top] and [top-1].
alu.nor	111	100	Bitwise nor [top] and [top-1].
alu.xor	111	101	Bitwise xor [top] and [top-1].
alu.sll	111	110	Logical left shift of [top-1] by [top] bits.
alu.slr	111	111	Logical right shift of [top-1] by [top] bits.

(a) [4 pts] Encode the following assembly instructions.

Instruction	Hexadecimal
load -4	1011 1100 = 0xBC
alu.mult	1110 0010 = 0xE2
halt	0110 0000 = 0x60

(b) [5 pts] The **load** instruction uses “**base + displacement**” memory addressing mode. You are asked to design a new pseudo-instruction **loadd**, which uses a “**direct**” memory addressing mode. The memory address for **loadd** is specified using the 5-bit IMM field.

(i) What is the range of values that the immediate field (IMM) can encode for the **load** instruction?

Answer: -16 to 15  
[min value] [max value]

**The IMM field for load uses 2’s complement notation to encode the offset.**

(ii) What is the range of values that the immediate field (IMM) can encode for the new **loadd** pseudo-instruction?

Answer: 0 to 31  
[min value] [max value]

**Only positive numbers make sense for direct memory addressing, so loadd uses unsigned notation to encode the address.**

(c) [5 pts] Assembly programmers request also that you add a “**dupe**” pseudo-instruction, which pops the top value from the stack and pushes it back onto the stack **twice**. The ISA reserves memory location 0 for pseudo-instruction temporary use. Write the assembly code for this pseudo-instruction using as few instructions as possible. Note: you may use the **loadd** pseudo-instruction of part (b). Write your code **\*neatly\*** below:

```
pushi 0
store 0
loadd 0
loadd 0
```

(d) [5 pts] Reorder the scrambled assembly instructions provided below to perform the following computation:

```
// compute the absolute value of x (2's complement)
if (x < 0)    result = -x;
else         result =  x;
```

Assume x is initially stored at the top of the stack. For full credit, clearly write the reordered instructions in the boxes provided on the right-hand side (no drawing arrows or the line numbers of the instructions).

halt	<b>dupe</b>
pushi -1	<b>pushi 7</b>
pushi 1	<b>alu.slr</b>
pushi 7	<b>beqz 4</b>
alu.add	<b>pushi -1</b>
alu.slr	<b>alu.xor</b>
alu.xor	<b>pushi 1</b>
beqz 4	<b>alu.add</b>
dupe	<b>halt</b>

(e) [3 pts] A new intern's summer task is to design a new TurboALU that can perform **more** ALUOPs than the current ALU in StackAttack (such as "not", "shift right arithmetic" and "divide"). He comes to you to ask how many **additional** ALUOPs **can be added** to the current ISA. What is your answer?

**There are two unused bits for A-type instructions, so the ALUOP field can be expanded from 3 bits to 5 bits.**

**$2^5$  possible –  $2^3$  existing =  $32 - 8 = 24$  additional ALUOPs.**

Answer: **24**

## 5. Single-Cycle Datapath [14 points]

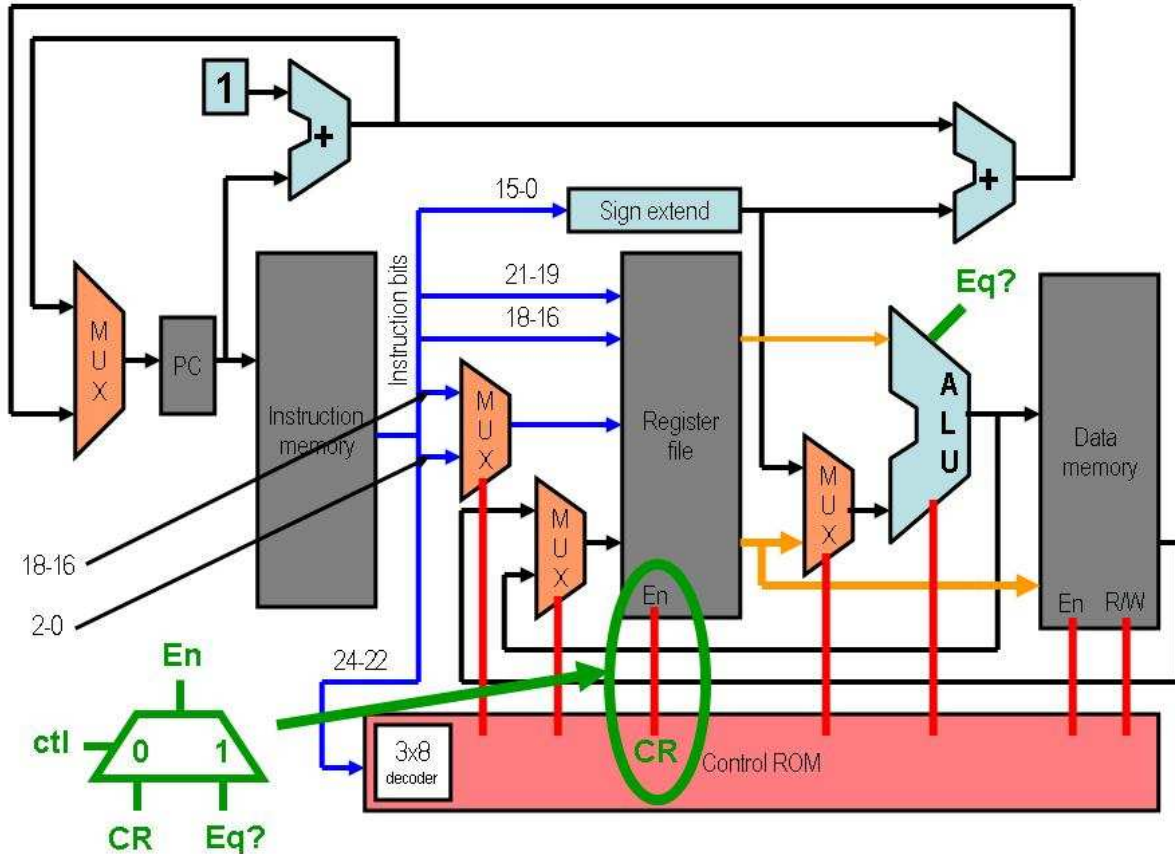


Figure 1: Single-cycle datapath from Lectures 9 and 10

Consider the single-cycle datapath shown in the figure above. (This is the datapath covered in Lectures 9 and 10). We want to introduce the instruction

```
condi_add  rs1  rs2  rd
```

In this instruction, the opcode is stored in bits 24-22,  $rs1$  is stored in bits 21-19,  $rs2$  is stored in bits 18-16, and  $rd$  is stored in bits 2-0. The following pseudo-code describes how the instruction operates (Note:  $[rs]$  denotes the contents of register  $rs$ ) :

```
if    [rs1] == [rs2]
then [rd] ← [rs1] + [rs2]
```

(a) [4 pts] List all the new hardware components (e.g., adders, ALUs, MUXs, registers, register files, memories, constants, etc.) that must be introduced to support `condi_add`. List type and quantity for each component you add in the table below:

Component type	quantity
<b>MUX</b>	<b>1</b>

(b) [5 pts] Using Figure 1, show where in the datapath your new hardware from part (a) is introduced. Make sure you clearly indicate any new wires that are introduced or that need to be re-routed.

**See Figure 1**

(c) [5 pts] List any new control signals that must be introduced to implement this instruction and indicate which components they control. For the various instructions in the LC-2K ISA, give the values provided by the control ROM for these signals.

New control signal - name	Controlled component	Value to use for <code>condi_add</code>	Value for other instructions
<b>ctl</b>	<b>MUX</b>	<b>1</b>	<b>0</b>