



The University of Michigan - Department of EECS

EECS370 – Introduction to Computer Organization

Midterm Exam 1  
October 11, 2007

---

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

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 **7** 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: KEY  
(Exams without a signed pledge will not be graded)

## 1. Short questions [15 points]

(a) [1 pts] Is the LC-2K ISA studied in class a load-store or a memory architecture?  
Circle the correct answer:

- Load-store architecture
- Memory architecture
- Neither
- Both

Ans: Load-store architecture

(b) [2 pts] We want to earn more money from our coke machine of lecture 9, so we extend it to accept quarters and **dollar coins** (which you usually get as change from post office machines). Each drink still costs 75c, and the machine lets you select a drink as soon as it receives at least 75c (this is the same as in lecture). However, if you pay more, the coke machine still does not return any change, and you will lose the extra money you paid. (That's exactly how we hope to make more money). How many states does the new FSM for the coke machine have? Circle the correct answer:

- 3 states
- 4 states
- 5 states
- 7 states
- 8 states

Since the question didn't ask for optimum solution we accepted all 4, 5 or 7 states

(c) [4 pts] What information is listed in the relocation table? Circle all that apply.

- Variable names in the C source file which the linker must locate
- Line numbers of the C source file that should be modified by the linker
- Function names in the C source file which the compiler cannot locate
- Line numbers in the compiled assembly file that must be modified by the linker

Ans: Line numbers in the compiled assembly file that must be modified by the linker.  
Relocation table and symbol table do not deal with C source file.

(d) [4 pts] Which of the following addressing modes is used in the MIPS ISA for control transfer instructions such as jump and branch? Circle all that apply.

- Base + Displacement
- Direct addressing
- PC-Relative
- Register Indirect

Direct addressing, PC – Relative and Register indirect are used for control transfer instructions

(e) [4 pts] Which of the following statements are true for an N-bit adder? Circle all that apply.

- For a ripple carry adder, the output delay is proportional to N.
- For a carry look-ahead adder, the output delay is proportional to  $\sqrt{N}$
- A carry look-ahead adder speeds up the calculation of the lowest-order bits
- A carry look-ahead adder speeds up the calculation of the highest-order bits

Options 1 and 4 are true statements. Output delay for a carry look ahead adder is proportional to  $\log N$

## 2. MIPS assembly [18 points]

The following MIPS code (with portions missing) is an implementation of the `mystery` function in assembly. The `mystery` function is a recursive function that takes as input parameter an integer (`$a0`) and returns an integer (`$v0`).

```
mystery addi    $sp,$sp,-12      // allocate memory to stack
        sw     $ra,0($sp)         //
        sw     $s0,4($sp)        // store callee-save registers
        sw     $s1,8($sp)       //
        addiu  $s0,$zero,1       // set $s0 to 1
add     $s1,$zero,$a0          // move input argument ($a0) to $s1
        beq   $s1,$zero,return    // check if $s1 == 0
        beq   $s1,$s0,return    // check if $s1 == 1
        subi  $s1,$s1,1       // decrease $s1 by 1
        add   $a0,$zero,$s1      // pass $s1 as the function parameter
        jal   mystery            // recursive call to mystery
add     $s0,$zero,$v0          // move the return value to $s0
        subi  $s1,$s1,1          // decrease $s1 by 1
        add   $a0,$zero,$s1     // pass $s1 as the function parameter
        jal   mystery            // recursive call to mystery
add     $s0,$s0,$v0           // add the return value to $s0
return  add   $v0,$zero,$s0      // set the function return value
        lw   $s1,8($sp)       //
        lw   $s0,4($sp)        // restore callee-save registers
        lw   $ra,0($sp)        //
        addi  $sp,$sp,12       // deallocate memory from stack
        j    $ra                // return
```

(a) [9 pts] Complete the assembly code for the `mystery` function by filling in the blanks. (hint: this part can be done by just reading the comments and not necessarily understanding what the function computes).

### Point allocation:

- Small blanks = 0.5 pts

- Big blanks = 1pts

**Big blanks that were slightly wrong (e.g., misplaced destination register) were getting half credit.**

**Notice that the stack pointer needs to get decremented at the beginning of the function and incremented at the end of the function. Half credit was given if the amount of bytes was correct (12), but the stack pointer was first increment and then decremented.**

(b) [7 pts] What does the `mystery` function compute? Below is an implementation of the `mystery` function in C with some portions missing. Complete the code by filling in the blanks.

```
int mystery(int arg)
{
    if ( __arg == 0 || arg == 1__ )
        return 1;
    return mystery(__arg - 1__) + mystery(__arg - 2__);
}
```

**Point allocation:**

**- 1.5 pts for each argument in the if statement**

**- 2 pts for each parameter passed into the recursive function calls.**

(c) [2 pts] What is the output of the C statement below?  
(show your work for partial credit)

```
printf("X=%d\n", mystery(4));
```

Answer: 

<b>X=5</b>
------------

**Partial credit was given if the correct function call tree was shown.**

### 3. Memory addressing [12 points]

Consider the following MIPS assembly program:

```

lbu  $4, 100($0)
sh   $4, -42($4)
lh   $3, 2($1)
sw   $3, 101($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. The initial value for register 1 is: \$1 = 0x0000 0064

Show the state of the memory and of the two 32-bit registers (\$3 and \$4) after executing each instruction of the snippet above. The initial values for register file and memory are provided. Fill in the following tables to show your answers (Leave a box empty if its value is the same as the previous one):

Register File:

Reg	Initial value	After inst 1	After inst 2	After inst 3	Final values
\$3	0x0000 0020			0xFFFFC700	
\$4	0x0000 0074	0x00000091			

Memory:

Address	Initial	After inst 1	After inst 2	After inst 3	Final values
100 <sub>10</sub>	0x91				
101 <sub>10</sub>	0x34				0xFF
102 <sub>10</sub>	0xC7				0xFF
103 <sub>10</sub>	0x84		0x00		0xC7
104 <sub>10</sub>	0x57		0x91		0X00
105 <sub>10</sub>	0xB3				

#### 4. Heaps, stacks and other parts of memory [10 points]

The following C code is compiled into MIPS assembly language:

```
int a, z;
int foo(int b) {
    static int sizeOfArray = 100;
    int e = 7;
    int f = 3;
    int *d[100];

    double n = bar(e, f);

    for (a = 0; a < sizeOfArray; a++) {
        d[a] = (int *) malloc(sizeof(int) * 100);
    }
}
```

Which variables are allocated on the stack? Which on the heap? Which on the static memory segment? For each variable entry in the table, circle the correct answer. Note: *malloc()* is the C's equivalent of C++'s *new()*.

Variable			
<b>a</b>	Heap	Stack	Static
<b>b</b>	Heap	Stack	Static
<b>d</b>	Heap	Stack	Static
<b>e</b>	Heap	Stack	Static
<b>f</b>	Heap	Stack	Static

Variable			
<b>n</b>	Heap	Stack	Static
<b>d[23]</b>	Heap	Stack	Static
<b>*d[56]</b>	Heap	Stack	Static
<b>sizeOfArray</b>	Heap	Stack	Static
<b>z</b>	Heap	Stack	Static

All global variables on Static : a, z  
 Variables declared as static on Static : sizeOfArray  
 Local variables on Stack : e, f, n, d  
 Input parameters on Stack : b  
 Value pointed by pointer on heap : \*d

## 5. Caller-save or callee-save? That is the question. [16 points]

You were given the gruesome task of compiling the program below. Your target ABI (application binary interface) recommends to use 3 registers as callee-saved and 3 as caller-saved. These 6 registers are the only registers you can use to perform general computation, all other registers are needed for other tasks (i.e., stack pointer, global pointer, return value, etc.).

*Make sure to show your work on this problem to receive partial credit for your answer.*

```
int mymain(int a) {
    int b, c, d ;
    b = 5;

    for (c = 0; c < 10; c++) {
        d = loopy(a,c);
        b = b + d;
    }
    return 0;
}

int loopy(int ina, int inb) {
    int e, f;
    sw e, sw f
    if (ina < inb) {
        if (ina <= 0) lw e, lw f return inb;
        f = inb - ina;
        e = loopy (ina-1, inb-2);
        e = e-f;
    }
    else e = ina-inb;
    lw e, lw f
    return e;
}
```

(a) [8 pts] In the **loopy** function, assume that variables `ina` and `inb` are mapped to caller-saved registers, while `e` and `f` are mapped to callee-saved registers. How many load and store instructions must be included in the assembly code for `loopy` for the purpose of storing and restoring the values of these variables to/from registers?

**The variables that must be stored/restored are marked on the C code of `loopy` in the proper location. `e` and `f` are callee-saved and `loopy` must write on them, hence it stores before starting to execute the function, and restores them before any return instruction.**

**`ina` and `inb` are caller-saved, however their value need not be preserved ACROSS the call to `loopy`, once their value is passed as an argument, the registers holding them can be freely overwritten. Hence these two variables do not require any store/restore.**

Number of lw instructions:

4

Number of sw instructions:

2

**Point allocation for wrong answers explaining the reasoning leading to the answer:**

- **4pts for storing/restoring both e and f in at least one restore point.**
- **1 pts additional for correctly restoring e and f in both return points**
- **3 pts for detecting that ina and inb do not require any store/restore**
- **0.5 is subtracted if the analysis takes into account how many times loopy is executed**

(b) [8 pts] For the **mymain** function, what is the best mapping from variables to registers that you can come up with? Your goal is to minimize the number of loads and stores that must be executed to save/retrieve the register values from/to the stack in order to preserve the necessary values across function calls.

For each variable, place an X on the appropriate column in the table on the next page indicating whether you want to use a caller-saved or a callee-saved register.

Function	Variable	CALLER-SAVED register	CALLEE-SAVED register
<b>mymain()</b>	<b>a</b>		<b>X</b>
	<b>b</b>		<b>X</b>
	<b>c</b>		<b>X</b>
	<b>d</b>	<b>X</b>	

**a, b and c's values must be preserved across the function call to loopy. For all of them a callee-saved register is best, so that we only add 1 lw/sw in mymain, instead of 10.**

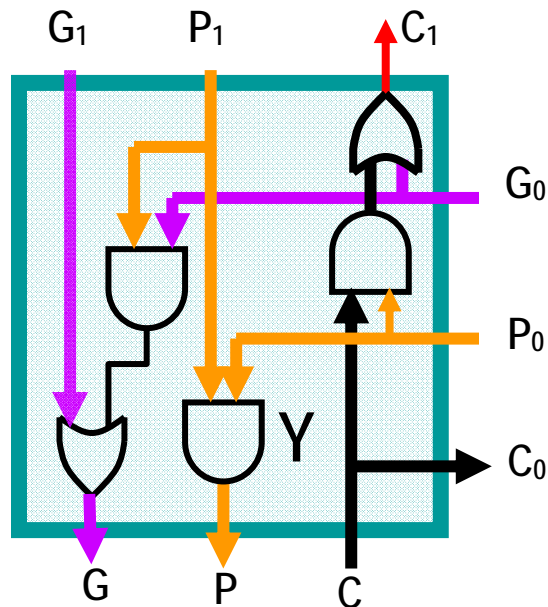
**d does not need to be preserved across the call to loopy. Hence a caller-saved register is best, so that we can avoid adding any lw/sw for the corresponding register.**

## 6. Adders and delays [11 points]

Consider the Y block of a carry look-ahead adder (shown below), and assume that we implemented this circuit using gates with the following propagation delays (delays are from any of the inputs to the output of the gate):

AND: 1 ns

OR: 1 ns



(a) [3 pts] What is the worst case propagation delay for the CarryOut ( $C_1$ ), Propagate ( $P$ ) and Generate ( $G$ ) signals?

$$P = P_0 P_1$$

One AND gate so delay is 1 ns

$$G = G_1 + P_1 G_0$$

1 AND gate and 1 OR gate so total delay is 2 ns

$$C_1 = G_0 + P_0 C$$

Again 1 AND gate and 1 OR gate do total delay is 2 ns

(b) [8 pts] For a 16-bit carry look-ahead adder, how many nanoseconds – in the worst case – does it take before the result is available?

For simplicity, assume that the delay through the X block is trivial (0 ns) – we have found a very clever implementation for the X block circuit which can compute its outputs instantaneously. Also, please assume that the propagation delay across a Y block is the

worst propagation delay that you computed in part (a), independently of which path through the Y block is actually used.

To jog your memory, we reproduced below a schematic of the 8-bit carry look-ahead adder that we studied in class – Note, however, that your answer should be the delay on a 16-bit adder. *Show your work to receive partial credit on this problem.*

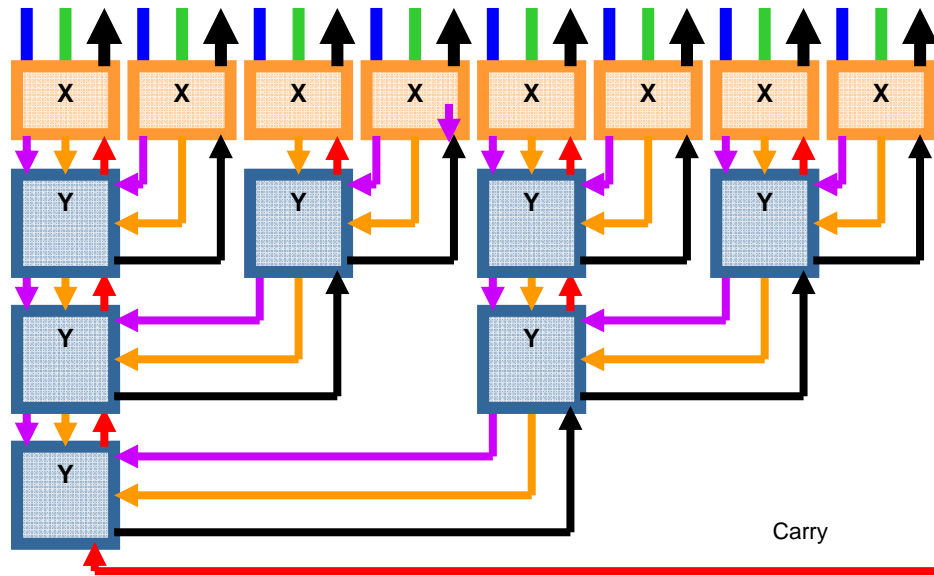
For a 16-bit adder the worst case path will have  $7Y$  ( $3Y + 4Y$ ) blocks. So the worst case propagation delay will be  $14ns$ .

**Regarding grading:**

- whosoever did the calculation for 8 bit adder instead of 16 bit adder got 6 points.
- students with who came up with 4 Y blocks instead of 7Y got 5 points
- student with a decent understanding of what is asked and came up with something close got 4/8
- students who calculated all Y blocks got 2/8
- students who scribbled something which kind of made sense that they were talking about adder got 1/8

Delay for a 16-bit adder:

Schematic of an 8-bit carry look-ahead adder: (your answer must be for a 16-bit adder)



## 7. Floating point arithmetic [18 points]

AMD's new SSE5 extension to the x86 ISA includes a new 16 bits floating point data type. The 16 bits of a number stored in this format are allocated as follows:

- 1 bit sign
- 5 bit exponent with a bias of 15
- 10 bit mantissa (a.k.a. significand)

All other aspects of this format are exactly the same as the standard IEEE floating point studied in class.

*You must show your work on this problem to be eligible for partial credit. If you need more space, attach another sheet, but label it clearly.*

(a) [3 pts] Circle the numbers listed below that can be represented **exactly** using the SSE5 floating point format:

$2^{18}$

 $\pi$ 

$2^{-6}$

$-2^{12}$

$-2^{-12}$

 $1/3$ 

$2^{18}$  is too big (exponent + bias would be  $18+15 = 33$ , which does not fit in the 5 bit exponent field)

$\pi$  is irrational and can never be represented exactly

$1/3$  is a fraction that infinitely repeats in binary (0.01010101...)

(b) [5 pts] What decimal number is represented by the following?

S	E	E	E	E	E	M	M	M	M	M	M	M	M	M	M
1	1	0	1	1	1	1	0	0	1	0	0	1	0	1	0

Sign: 1, so number is negative. Exponent = 10111 binary = 23 decimal, subtract bias of 15, actual exponent is  $23 - 15 = 8$ .  
 Mantissa: insert implied leading 1, 1.1001001010, shift the radix point right 8 places (since exponent is  $2^8$ ), so the actual number represented is -110010010.10 binary. Convert this from binary to decimal, get -402.5

Answer: 

-402.5
--------

(c) [5 pts] Show the representation of 370 (decimal) in this format (fill in the bits in the slots below).

370 decimal is 101110010 binary =  $1.01110010 * 2^8$   
 Sign bit is 0 (positive)  
 Exponent: add bias:  $8 + 15 = 23 = 10111$  binary  
 Mantissa: suppress leading 1, store 0111001000

Answer:

S	E	E	E	E	E	M	M	M	M	M	M	M	M	M	M
0	1	0	1	1	1	0	1	1	1	0	0	1	0	0	0

(d) [5 pts] Add together the floating point numbers in parts (b) and (c) above and show the sum in SSE5 format:

Exponents already match (both are  $2^{+8}$ ), so no need to shift to align radix points.

Add mantissa (do not forget leading 1s)

$$-1.1001001010 + 1.0111001000 = -0.0010000010 (* 2^8)$$

Must shift the radix point 3 places right (and decrease exponent by 3) to renormalize:  $-1.000001 * 2^5$

Number is negative so sign = -1

Add bias of 15 to exponent of 5, get 20 decimal, binary 10100

Mantissa: suppress leading 1, store 0000010000

Answer:

S	E	E	E	E	E	M	M	M	M	M	M	M	M	M	M
<b>1</b>	<b>1</b>	<b>0</b>	<b>1</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>1</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>