

# Computer Architecture I Midterm I

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

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

E-Mail ... @shanghaitech.edu.cn: \_\_\_\_\_

Question	Points	Score
1	11	
2	6	
3	16	
4	19	
5	48	
Total:	100	

- This test contains 9 numbered pages, including the cover page. The back of each page is blank and can be used for scratch-work, but will not be looked at for grading.
- Put your pinyin name on the top of every page.
- Please turn **off** all cell phones, smartwatches, and other mobile devices. Remove all hats and headphones. Put everything in your backpack. Place your backpacks, laptops and jackets under your seat.
- You have 85 minutes to complete this exam. The exam is closed book; no computers, phones, or calculators are allowed. You may use one A4 page (front and back) of notes in addition to the provided green sheet.
- The estimated time needed for each of the 5 topics is given in parenthesis. The total estimated time is 75 minutes.
- There may be partial credit for incomplete answers; write as much of the solution as you can. We will deduct points if your solution is far more complicated than necessary. When we provide a blank, please fit your answer within the space provided.

## 1. Various Questions (12 min)

- 3 (a) Name 6 Great Ideas in Computer Architecture.

**Solution:** 1. Abstraction (Layers of Representation/Interpretation)  
 2. Moores Law (Designing through trends)  
 3. Principle of Locality (Memory Hierarchy)  
 4. Parallelism  
 5. Performance Measurement and Improvement  
 6. Dependability via Redundancy

- 2 (b) How would J-type instructions be affected (in terms of their "reach") if we relaxed the requirement that instructions be placed on word boundaries, and instead allow them to be placed anywhere?

**Solution:** The range over which we can jump would be only 1/4th.

- 2 (c) You have a program that can achieve almost a 20x speedup with millions of processors, so what is the percent of the parallel portion of its workload? Show how you got to your result!

(c) 95%

- 4 (d) Put the following (not exhaustive list) in chronological order. We've started it for you.

6	Code and Data from various places are stitched together.
8	Static, code, and global space are reserved/initialized in memory.
1	A student is assigned a homework that implements a prime search.
9	Execution begins at main.
4	MAL is translated into TAL.
2	The student writes his or her code in C.
5	Link tables are produced.
3	The student's C code is translated into MIPS.
7	Links are "edited"

2. Suppose for questions 2 (a)-(c) that we were to modify the MIPS ISA so that it exposed 50 registers instead of 32, and adjusted the field widths of our R, I and J instruction formats to be able to address all the registers, but did not change the size of the opcode or shamt fields. Registers and instructions will remain 4 bytes wide. (5 min)

2

- (a) At most how many instructions can a single beq instruction now reach?

(a)  $2^{14}$

2

- (b) How many more addresses can now be reached from a jal instruction?

(b) **none**

2

- (c) How many different R-type instructions can we now have?

(c) **8**

### 3. Number Representation (12min)

4

- (a) Suppose  $a$  is an 8-bit signed integer represented as  $a_{hex} = 0xBC$ , then its binary

representation is  $a_{two} =$  \_\_\_\_\_, its decimal representation is  $a_{ten} =$  \_\_\_\_\_.

**Solution: Answer:** 1011110, -60

4

- (b) For an 12-bit value, two's complement integer, what are the largest AND smallest value you can represent in decimal?

**Solution: Answer:** 2047, -2048

4

- (c) Assume an 8-bit two's complement machine on which all operators are performed on 8-bit registers. Answer the results of the following operations in hexadecimal. Assume that subtraction is done with SUBU and addition is done with ADDU.

$$\begin{array}{rcl} \text{a} & 6A & (\text{hex}) \\ - & 89 & (\text{hex}) \\ \hline \end{array}$$

$$\begin{array}{rcl} \text{b} & CA & (\text{hex}) \\ + & 16 & (\text{hex}) \\ \hline \end{array}$$

**Solution: Answer:** E1, E0

4

- (d) Fill in the blank so that the function `mod32` will return the remainder of `x` when divided by 32. The first blank should be a **bitwise** operator, and the second blank should be a single **decimal** number:

```
unsigned int mod32( unsigned int x ){  
  
    return x & 31;  
}
```

## 4. C Programming (15 min)

We wish to implement a *nibble* (4-bit) array, where we can read and write a particular *nibble*. Normally for read/write array access, we would just use bracket notation (e.g.,  $x = A[5]$ ;  $A[5] = y$ ), but since a nibble is smaller than the smallest datatype in C, we have to design our own **make\_nibble()**, **set\_nibble()** and **get\_nibble()** functions. To bring you to the *C of Madness*, we require you to **complete all the blanks below in C in one line**. There is a 25% point deduction of a sub-question where you use **if/else** or the **?:** format. We'll use the following typedefs to make our job easier.

```
/* If it is a single nibble, value is in least significant
   nibble. */
typedef uint8_t nibble_t;
```

E.g., imagine a nibble array **A** with 2 nibbles (1 byte):

```
A[0] = 0x3f; // initialization
get_nibble(A, 0); // return 0xf
get_nibble(A, 1); // return 0x3
set_nibble(A, 0, 0x34) // set the 0st nibble to 0x4
get_nibble(A, 0); // return 0x4
set_nibble(A, 1, 0xed) // set the 1st nibble to 0xd
get_nibble(A, 1); // return 0xd
```

2

- (a) To make things easier, we require that the length of a nibble array is always a power of 2. So you may complete the macro definition to facilitate the multiplication by 2. e.g., **TWICE(3)** gives 6.

```
/* Calculate the multiplication by 2 below */

#define TWICE(X) ( (X) * 2 )
```

2

- (b) Suppose we have declared a pointer **nibble** of nibble array.

```
nibble_t *nibble; /* pointer of nibble array. */
```

Now we need to allocate 4 nibbles for the declared pointer. You may 1) complete the parameter list, 2) write **make\_nibble()** in C in one line (you don't need to handle exceptions), 3) complete the function call.

```
void make_nibble(nibble_t **head, uint32_t length) {
    /* one line of code below */
    (*head) = (nibble_t*)malloc(length / 2);
}
/* somewhere in main... */
make_nibble(&nibble, TWICE(2));
```

10

- (c) You may write **set\_nibble()** in C in as one command. When we say write item into head[index], we are to write the four bits on the right end of the item into head[index].

```
void set_nibble(nibble_t *head, uint32_t index, uint8_t item) {
    head[index / 2] = (head[index / 2] & (0xf0 >> 4 * (index %
        2))) | ((item & 0x0f) << 4 * (index % 2));
}
```

5

- (d) You may write **get\_nibble()** in C in one line.

```
uint8_t get_nibble(nibble_t *head, uint32_t index) {
    return ((head[index / 2] >> 4 * (index % 2)) & 0x0f);
}
```

5. Question: Binary Search in MIPS: ( (a) 4 min; (b) 8 min; (c-i) 10 min; (j) 8 min; total: 30 min)

6

- (a) First of all, let's review the binary search algorithm. The binary search algorithm finds the position of a target value within a sorted array (smallest to biggest). It is efficient for sorted datastructures that are stored continuously in memory. We've provided you with a C function in the space below. Fill in the code at places marked with (1) and (2).

```
int binary_search(int *data, int n, int value)
{
    int left = 0;
    int right = n-1;
    int mid ;
    while (left < right)
    {
        mid = (left + right) >>1;
        if(value > data[mid])
        {
            left = mid+1;
        } else
        {
            right = mid;
        }
    }
    if (data[left]==value) return left;
    return -1;
}
```

12

- (b) Now that you've warmed up on the C version of this code, let's convert it into MIPS code! In the following code, each element in array is 32-bit integer. The argument register \$a0 is the pointer to the first element in memory. The argument register \$a1 a 32-bit integer which indicates the number of elements in array. The value you need to find out from array is in \$a2. You need to provide the index in \$v0 if the value of \$a2 is in the array, otherwise set it to -1. Fill in the code at (3) - (8).

```
binary_search:
li $t0, 0
addu $t1, $a1, -1
LOOP:
slt $t1, $t0, $t1 # (3)
beq $t2, $0, ENDLOOP
add $t3, $t0, $t1

sra $t4, $t3, 1 # (4)

mul $t3, $t4, 4 # (5)
add $a3, $t3, $a0
lw $t3, 0($a3)
slt $t2, $t3, $a2

beq $t2, $0, ELSE # (6)
add $t0, $t4, 1
j NEXT
ELSE:
add $t1, $t4, $0
NEXT:
j LOOP
ENDLOOP:
mul $t3, $t0, 4

add $a3, $t3, $a0 # (7)
lw $t3, 0($a3)
beq $t3, $a2, SUCCESS
li $v0, -1
j DONE
SUCCESS:

add $v0, $t0, $0 # (8)
DONE:
jr $ra
```

- 2 (c) Towards the end of the above listing there is the instruction "beq \$t3, \$a2, SUCCESS". When will the goal address be determined and, if you can tell now, what is the value?

**Solution:** During assembly. The value is 2.

- 2 (d) Towards the end of the above listing there is the instruction "j DONE". When will the goal address be determined and, if you can tell now, what is the value?

**Solution:** During linking. The value is unknown now.

- 2 (e) Towards the end of the above listing there is the instruction "jr \$ra". When will the goal address be determined and, if you can tell now, what is the value?

**Solution:** During runtime. The value is unknown now.

- 2 (f) The Assembly code above can be used as a function. How would you call the function and which registers would you need to fill with proper values before calling the function?

**Solution:** Call: "jal binary\_search". Setup the argument registers \$a0, \$a1 and \$a2.

- 4 (g) The Assembly code above can be used as a function. Why is it not using the stack? Explain in detail.

**Solution:** It is a leaf function that does not call any other function so we do not need to save \$ra. It is not using any saved registers so we also do not need to save those. So we do not need to make use of the stack.

- 4 (h) The Assembly code above can be used as a function. It is not using the stack. Assuming we would have used 8 bytes of the stack space, add the instruction that is needed at the end of the function:

```
DONE:
addi $sp, $sp, 8           # (9)
jr $ra
```



- 6 (i) The Assembly code above contains pseudoinstructions. List all of the pseudoinstructions that were used and also which instructions they are generally replaced with (without registers).

**Solution:** move => addi

li => lui + ori

mul => mult + mflo

- 8 (j) Instruction Format: Translate the assembly into machine code and vice versa.

Instruction	Code
mult \$t1, \$t2	0x012a0018
xor \$v0, \$a0, \$a1	0x00851026
lw \$t3, -8(\$s4)	0x8e8bfff8
ori \$v0, \$t0, 24	0x35020018