# CIS 600

Advanced Computer Architecture

HW# 1

Due: 2/8 (Thursday)

Turn in the answer to Blackboard.

**Name:** ***Yi Dong 2663734***

Question 1 (13pts)

1. (2pts) What three steps occur to execute MIPS instructions?
   [ Fetch ], [ **Decode** ], [ **Execute** ]

2. (2pts) The C compiler is actually two separate programs: the [ **preprocessor** ] and the [ **translator** ].

3. "True"or "False" to the following statements. (All numbers are **decimal** numbers.)

| | | |
|---|---|---|
| **True** | (a) "sw $1, $3(100)" | is a legal MIPS instruction. |
| **False** | (b) "sw $1, 100($3)" | is a legal MIPS instruction. |
| **False** | (c) "add $4, $5, 100($3)" | is a legal MIPS instruction. |
| **False** | (d) "add $4, 200($4), 100($3)" | is a legal MIPS instruction. |
| **True** | (e) "lw $1, 102($3)" | is a legal MIPS instruction. |

3. The following diagram is to explain two multiplexor control signals: "RegDst" and "MemtoReg". They are used to select destination register number and write data to the destination register. Answer with TRUE or FALSE on the following five statements.

*add $8, $17, $18*

| 000000 | 10001 | 10010 | 01000 | 00000 | 100000 |
|---|---|---|---|---|---|
| op | rs | rt | rd | shamt | funct |

*lw $1,*

| 100011 | 00010 | 00001 | 0000 0000 0110 0100 |
|---|---|---|---|
| op | rs | rt | 16 bit offset |

**True** (i) "RegDst" control signal selects "rt" field ($18) when executing the "add" instruction.
**True** (ii) "RegDst" selects "rt" field ($1) when executing the "lw" instruction.
**False** (iii) "MemtoReg" selects ALU output ($17+$18) when executing the "add" instruction.
**False** (iv) "MemtoReg" selects ALU output ($2+100) when executing the "lw" instruction.

Question 2 (17pts)

The following variables are allocated in the memory beginning at address 2000 on a **64-bit** operating system. Note that the allocation goes towards the **upper end** of the address space.

1. (3 pts): How many bytes does the following data type occupy?

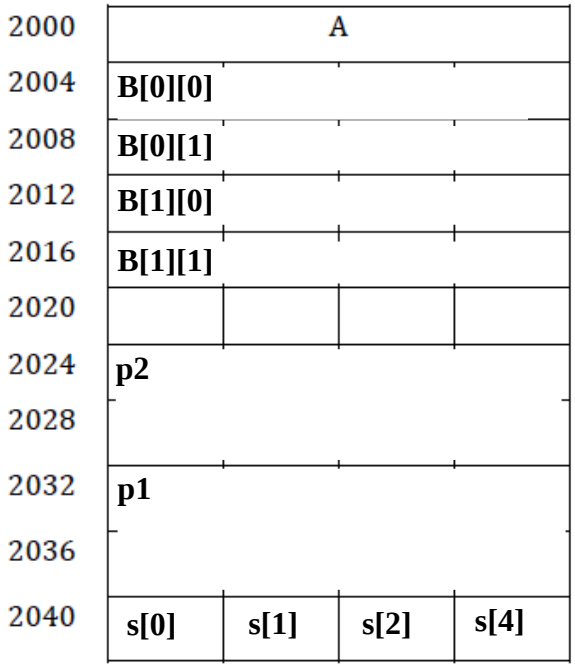
1. char – 1 byte
2. int – 4 bytes
3. float – 4 bytes
4. char* - **8 bytes**
5. int* - **8 bytes**
6. float* **- 8 bytes**

2. (6pts): Complete the memory map below with the variable names at each word or byte in memory. Do not include the variable's value. For B[2][2], you need to specify the mapping for each of the 3 elements in array B.

```
Int A = 200;
float B[2][2] = {{1.0,2.0},
{3.0,4.0}};
float *p2 = &(B[1][1]);
int *p1 = &A;
char s[  ] = "foo";
```

**p2 is 8 bytes long, need allocated at an address that is a multiple of 8**

**s[4] is '\0'**




The variables are allocated in the order that they are defined. The allocation scheme tries not to leave gaps in the memory unless necessary (e.g no gaps between char data types).
Hint: **Watch for alignment**: a data type of 4 bytes is always allocated at an address that is a multiple of 4. A data type of 8 bytes is always allocated at an address that is a multiple of 8.

3. (8 pts): using the values in part B, give the value of each expression. Answers may be characters

p1 **2000** *p1 **200** &p1 **2032** p2 **2016**

*p2 **4.0** &p2 **2024** &(f[1]) **2041** *(f+3) **'\0'**

**here f should be s**

Question 3 (30pts)

1. (8 pts) Write a MIPS code fragment that adds the value 0x4FA2C3 to register $1 and puts the result in $2. Modify only registers $1 and $2. (You may use hexadecimal immediate values.) *For maximum credit, include comments.*

   **An integer is 4 bytes long, that is double-word length. Because MIPS can only load up to one word length in one instruction, 0x4FA2C3 need to load into $2 separately, first load to upper 16bits (one word length), then load the rest to lower 16bits, the second load need to use OR operation to avoid upper bit overwritten by 0.**

| Label | Instruction | Comment |
|---|---|---|
| 1 | lui $2, 0x4F | $2 = 0x4F * $2^{16}$ |
| 2 | ori $2, $2, 0xA2C3 | $2 = $2 \| 0xA2C3 |
| 3 | add $2, $2, $1 | $2 = $2 + $1 |

2. (10pts) Write a C code fragment that loops through an array of integers A until it its one that is zero. It should compute and store in the variable p the running product of the nonzero integers that occur in the array before the zero. Assume that A contains at least one zero and it contains at least one nonzero integer before the first zero. *For maximum credit, choose the most appropriate loop construct and declare and initialize variables as needed.*

```
int A[ ] = {..., 0, ...};     // Array A
int p = 1;                    // product initialize as 1
int i = 0;
do {                          // in loop body, iterate
     p = p*A[i];              // through array A, until
     i++;                     // A[i] == 0, store running
} while(A[i]);                // product in p
return p;
```

3. (12pts) Suppose A is stored in memory location 1020 and B is stored in memory location 1024. Write a MIPS program fragment that computes "$256 \cdot (A + B/16)$" and stores the result at memory location 1028. Use a minimum number of instructions and registers.

| Instruction | Comment |
|---|---|
| lw $1, 1020($0) | $1 = mem[$0+1020], load A into $1 |
| lw $2, 1024($0) | $2 = mem[$0+1024], load B into $2 |
| sra $2, $2, 4 | $2 = $2 >> 4 (arithmetic), now B = $B/2^4$ |
| add $2, $2, $1 | $2 = $2 +$1, now $2 = A + B/16 |
| sll $2, $2, 8 | $2 = $2 << 8 (logical), now $2 = $2 * 256 |
| sw $2, 1028($0) | Mem[$0+1028] = $2, store $2 to memory location 1024 |

Question 4. (20pts)

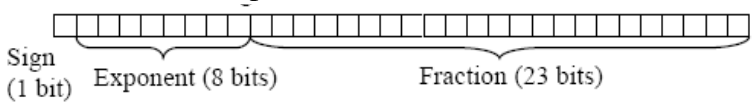
1. (20pts) Write a MIPS code fragment that corresponds to this C fragment. Assume `$1` holds `N`, and `$2` holds `Sum`. Feel free to use additional registers, but use a minimum number of instructions and registers.

```
Sum = 0;
while (N!=0){
     Sum += N&1;
     N = N >> 1;
}
```

| Label | Instruction | Comment |
|---|---|---|
| | addi $2, $0, 0 | $2 = $0 + 0, add immediate 0 to $2, Sum = 0 |
| Loop: | beq $1, $0, Exit | branch if($1=$0), go to label Exit |
| | andi $3, $1, 1 | $3 = $1 & 1, and immediate 1 with $1, store to $3 |
| | add $2, $2, $3 | $2 = $2 + $3, add $3 to $2, Sum += N&1 |
| | srl $1, $1, 1 | $1 = $1 >> 1 (logical), $1 shift right one bit, N = N>>1 |
| | j Loop | jump back to label Loop, continue loop body |
| Exit: | | |

Question 5. (20pts)

Number conversion. IEEE 754 floating-point standard represents a 32-bit single-precision normalized decimal number into floating-point number to be the following. Please convert the given IEEE 754 FP representation (in hex). Describe the intermediates.




$$= (-1)^{sign} * (1+fraction) * 2^{Exponent-127}$$

1 **negative number, set Sign bit to 1**

2 **the integer part, 30, convert to binary, $(30)_{10} = (11110)_2$**

3 **the fraction part, 0.375, keep multiply 2, until fraction part equals zero, record result in integer part each time in binary bits as following,**

- **3.1 0.375 * 2 = 0.75 ==> $b^{-1} = 0$**
- **3.2 0.75 * 2 = 1.5 ==> $b^{-2} = 1$**
- **3.3 0.5 * 2 = 1.0 ==> $b^{-3} = 1$, since fraction part is 0, stop**
- **3.4 now we get $(0.375)_{10} = (0.011)_2$**

4 **put binary form from step 2 and step 3.4 together, we get $(11110.011)_2$**

5 **since IEEE 754 single-precision requires format as $(1.xxxxxx)_2 * 2^e$, convert the binary from step 4 as $(1.1111011)_2 * 2^4$**

6 **the fraction part (right side of the binary point) is 1111011**

7 **the exponential is 4, need to add 127 to get bias form 131, convert to binary, $(131)_{10} = (10000011)_2$**

8 **now put sign bit, exponent binary, fraction binary together, fill the following bits with 0**

**1-10000011-11110110000000000000000**

9 **convert binary to Hex, we get** $(-30.375)_{10} = ($ **C1FB0000** $)_{16}$

# MIPS Instruction Set (core)

| *instruction* | *example* | *meaning* |
|---|---|---|
| **arithmetic** | | |
| add | add $1,$2,$3 | $1 = $2 + $3 |
| subtract | sub $1,$2,$3 | $1 = $2 - $3 |
| add immediate | addi $1,$2,100 | $1 = $2 + 100 |
| add unsigned | addu $1,$2,$3 | $1 = $2 + $3 |
| subtract unsigned | subu $1,$2,$3 | $1 = $2 - $3 |
| add immediate unsigned | addiu $1,$2,100 | $1 = $2 + 100 |
| set if less than | slt $1, $2, $3 | if ($2 < $3), $1 = 1 else $1 = 0 |
| set if less than immediate | slti $1, $2, 100 | if ($2 < 100), $1 = 1 else $1 = 0 |
| set if less than unsigned | sltu $1, $2, $3 | if ($2 < $3), $1 = 1 else $1 = 0 |
| set if < immediate unsigned | sltui $1, $2, 100 | if ($2 < 100), $1 = 1 else $1 = 0 |
| multiply | mult $2,$3 | Hi, Lo = $2 * $3, 64-bit signed product |
| multiply unsigned | multu $2,$3 | Hi, Lo = $2 * $3, 64-bit unsigned product |
| divide | div $2,$3 | Lo = $2 / $3, Hi = $2 mod $3 |
| divide unsigned | divu $2,$3 | Lo = $2 / $3, Hi = $2 mod $3, unsigned |
| **transfer** | | |
| move from Hi | mfhi $1 | $1 = Hi |
| move from Lo | mflo $1 | $1 = Lo |
| load upper immediate | lui $1,100 | $1 = 100 x $2^{16}$ |
| **logic** | | |
| and | and $1,$2,$3 | $1 = $2 & $3 |
| or | or $1,$2,$3 | $1 = $2 \| $3 |
| and immediate | andi $1,$2,100 | $1 = $2 & 100 |
| or immediate | ori $1,$2,100 | $1 = $2 \| 100 |
| nor | nor $1,$2,$3 | $1 = not($2 \| $3) |
| xor | xor $1, $2, $3 | $1 = $2 ⊕ $3 |
| xor immediate | xori $1, $2, 255 | $1 = $2 ⊕ 255 |
| **shift** | | |
| shift left logical | sll $1,$2,5 | $1 = $2 << 5 (logical) |
| shift left logical variable | sllv $1,$2,$3 | $1 = $2 << $3 (logical), variable shift amt |
| shift right logical | srl $1,$2,5 | $1 = $2 >> 5 (logical) |
| shift right logical variable | srlv $1,$2,$3 | $1 = $2 >> $3 (logical), variable shift amt |
| shift right arithmetic | sra $1,$2,5 | $1 = $2 >> 5 (arithmetic) |
| shift right arithmetic variable | srav $1,$2,$3 | $1 = $2 >> $3 (arithmetic), variable shift amt |
| **memory** | | |
| load word | lw $1, 1000($2) | $1 = memory [$2+1000] |
| store word | sw $1, 1000($2) | memory [$2+1000] = $1 |
| load byte | lb $1, 1002($2) | $1 = memory[$2+1002] in least sig. byte |
| load byte unsigned | lbu $1, 1002($2) | $1 = memory[$2+1002] in least sig. byte |
| store byte | sb $1, 1002($2) | memory[$2+1002] = $1 (byte modified only) |
| **branch** | | |
| branch if equal | beq $1,$2,100 | if ($1 = $2), PC = PC + 4 + (100*4) |
| branch if not equal | bne $1,$2,100 | if ($1 ≠ $2), PC = PC + 4 + (100*4) |
| **jump** | | |
| jump | j 10000 | PC = 10000*4 |
| jump register | jr $31 | PC = $31 |
| jump and link | jal 10000 | $31 = PC + 4; PC = 10000*4 |

# CIS 600

## Advanced Computer Architecture

## HW# 1

Due: 2/16 (Friday)

Turn in the answer to Blackboard.

**Name:**

Question 1 (30pts) The instruction encodings for the 6-instruction processor are as shown in the table below:

| Instruction | Opcode | 16-bit encoding | | | | Function |
|---|---|---|---|---|---|---|
| Mov Ra, d | 0000 | Opcode **(4 bits)** | Destination Register **(4 bits)** | Address **(8 bits)** | | RF[a]←M[d] |
| Mov d, Ra | 0001 | Opcode **(4 bits)** | Source Register **(4 bits)** | Address **(8 bits)** | | M[d]←RF[a] |
| Add Ra,Rb,Rc | 0010 | Opcode **(4 bits)** | Destination Register **(4 bits)** | Source Register **(4 bits)** | Source Register **(4 bits)** | RF[a]←RF[b] + RF[c] |
| Mov Ra, #C | 0011 | Opcode **(4 bits)** | Destination Register **(4 bits)** | Constant **(8 bits)** | | RF[a] ← c |
| Sub Ra,Rb,Rc | 0100 | Opcode **(4 bits)** | Destination Register **(4 bits)** | Source Register **(4 bits)** | Source Register **(4 bits)** | RF[a]←RF[b] - RF[c] |
| Jumpz Ra, X | 0101 | Opcode **(4 bits)** | Source Register **(4 bits)** | Offset **(8 bits)** | | If RF[a] == 0, PC←PC+offset |

a) Assume that you want to augment this ISA to support 20 additional and unique instructions (e.g. Mult, And, Or, etc.), while still keeping the instruction encoding as 16 bits. How will the execution and encoding of the Add instruction be affected? (Other instructions could be affected too, but you just need to comment on how the Add instruction will be impacted.)

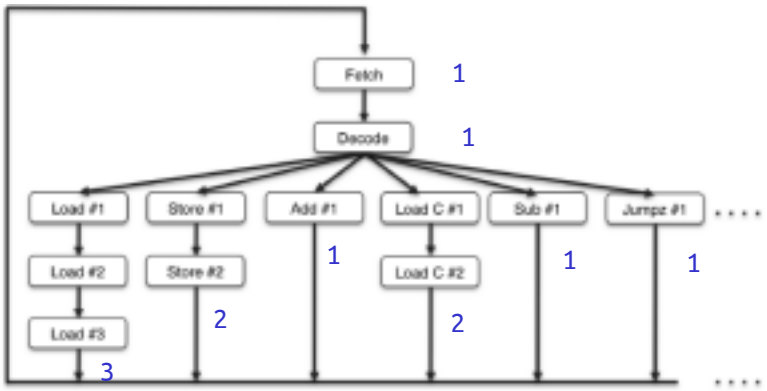
Since there will be 20 additional and unique instructions, the opcode will range from 0 to 25, we need at least 5 bits to encode all 26 opcode, instead of 4 bits.
Since we need to keep instruction encoding as 16 bits, the bits for other parts encoding need to be one bit less.

For Add instruction, only 0-7 register allowed for either Destination register, or Source register

b) Now, assume that the 20 new instructions have all been added. Their addition has resulted in changes to the state diagram. A portion of the new finite state diagram is shown below. Given this new state diagram, how many clock cycles will the code (that is also shown below) take to run?

Code:

```
        Mov     R1, #1
        Mov     R2, #2
        Mov     R4, #4
        Sub     R5, R4, R2
        Sub     R5, R5, R2
        Jumpz   R5, X
        Add     R1, R1, R1
X:      Add     R1, R2, R4
        Mov     10, R1
```




```
        Mov     R1, #1          4       <---load 1 into R1 takes 4 CCs
        Mov     R2, #2          4       <---load 2 into R2 takes 4 CCs
        Mov     R4, #4          4       <---load 4 into R4 takes 4 CCs
        Sub     R5, R4, R2      3       <---R5=R4-R2=2, Sub takes 3 CCs
        Sub     R5, R5, R2      3       <---R5=R5-R2=0, Sub takes 3 CCs
        Jumpz   R5,X            3       <---R5=0, jump on zero, to label X, jumpz takes 3 CCs
        Add     R1, R1, R1              <---skip, no fetch, no decode
X:      Add     R1, R2, R4      3       <---R1=R2+R4=6, Add takes 3 CCs
        Mov     10, R1          4       <---M(10)=R1, Store takes 4 CCs
```

Total = 28 CCs

Question 2 (20pts), Review:True or False (fill in T or F)

1. (__F__) Register is implemented inside the main memory on computer board
2. (__T__) von Neumann architecture has data and instructions in the same memory space.
3. (__T__) Static RAM is typically used to implement Cache
4. (__F__) Dynamic RAM is faster than Static RAM in access time.
5. (__T__) In modern computers, CPU and Memory are connected by BUS
6. (__F__) Direct Mapping in Cache system is more complex in implementation than Set Associative Mapping
7. (__F__) Floating point data points are uniformly represented on the real line.
8. (__T__) unsigned integer data points are uniformly represented on the real line.
9. (__T__) Read and write operations can be performed on EPROM devices
10. (__F__) The density of data on the hard disk does not affect data access speed.

Question 3 (20pts) Assume that to spell check a large file, 820,000,000 instructions are needed. The instructions in the program are broken down into 4 different classes, and each class requires N clock cycles to execute. Specific information is given in the table below.

| Instruction Class | Clock Cycles per Instruction | Number of Instructions |
|---|---|---|
| Branch | 3 | 150,000,000 |
| Store | 4 | 185,000,000 |
| Load | 5 | 260,000,000 |
| ALU / R-type | 4 | 225,000,000 |

If the total execution time for this program is found to be 1.57 seconds, what is the clock cycle time of the computer on which it was run?

```
total CCs = 3*1.5*10^8 + 4*1.85*10^8 + 5*2.6*10^8 + 4*2.25*10^8
          = 3.39*10^9

time for each CC = 1.57 seconds / 3.39*10^9
                 = 0.463 * 10^(-9) seconds
                 = 0.463 nano-seconds      The machine is a 2.16 GHz machine
```

Question 4 (30pts) Write the MIPS assembly for the following C-code:

a)

```
int f, g, h, i, j;
f = (g + h) - (i + j);
```

```
suppose f, g, h, i, j are saved
in memory address starting from
1000 consecutively, that is
M(1000) = f
M(1004) = g
M(1008) = h
M(1012) = i
M(1016) = j

the code will be:
```

```
MIPS Code
-------------------
lw  $1, 1004($0)
lw  $2, 1008($0)
add $1, $1, $2
lw  $2, 1012($0)
lw  $3, 1016($0)
add $2, $2, $3
sub $1, $1, $3
sw  $1, 1000($0)
```

b)

```
if ( i < j ) {
     k++ ;
     i = i * 2 ;
     goto L1 ;
}
```

```
suppose i, j, k are loaded
in register $1, $2, $3,
L1 is loaded in $31

the code will be:
```

```
MIPS Code
-----------------------
        sub  $2, $1, $2 <---- $2=j-i
        blez $2, X      <---- branch to X if j-i<=0
        addi $3, $3, 1  <---- $3=$3+1
        sll  $1, $1, 1  <---- $1=i*2
        jr   $31        <---- jump to L1
L1:     ....
        ....
X :     ....            <---- if i>=j, will branch here
        ....
```

c)

```
i = 0;
While (i != 5)
i = i + 5;
```

```
MIPS Code
-----------------------
        and  $1, $1, $0         <----$1=0
        addi $2, $0, 5          <----$2=5
Loop:                           <----jump here until $1=5
        beq  $1, $2, End        <----if $1=$2=5, goto End
        addi $1, $1, 5          <----$1=$1+5
        j    L1                 <----jump to L1
End:                            <----end of loop
```