# CSED211 Lab 02
## Floating Point& Assembly

2018. 09. 12.

# Floating Point

- Fractional binary numbers

- IEEE floating point standard

# Fractional Binary Numbers

- Representation

| $2^{i}$ | $2^{i-1}$ | | $4$ | $2$ | $1$ | $2^{-1}$ | $2^{-2}$ | | $2^{-j}$ |
|---|---|---|---|---|---|---|---|---|---|
| $b_i$ | $b_i$ | … | $b_2$ | $b_1$ | $b_0$ | $b_{-1}$ | $b_{-2}$ | … | $b_{-j}$ |

- Limitation
  - Limitation1: how can we represent $\frac{1}{3}, \frac{1}{7} \ldots$ ?
  - Limitation2: how can we represent $2^{i+1} + 2^{-j}$?

# IEEE floating point standard

- Representation

$$(-1)^s \, M \, 2^E$$

  - s determines whether number is negative or positive.
  - M normally a fractional value in range [1.0, 2.0).
  - E weights value by power of two.

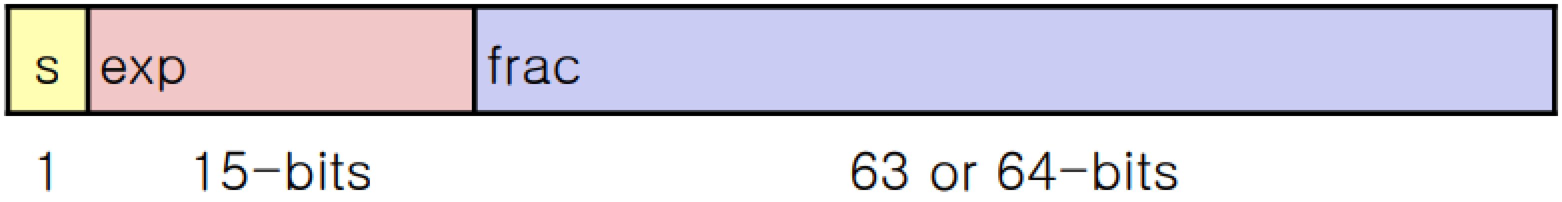
- Encoding
  - MSB s is sign bit s
  - exp field encodes E(but is not equal to E)
  - frac field encodes M(but is not equal to M)

# Precisions

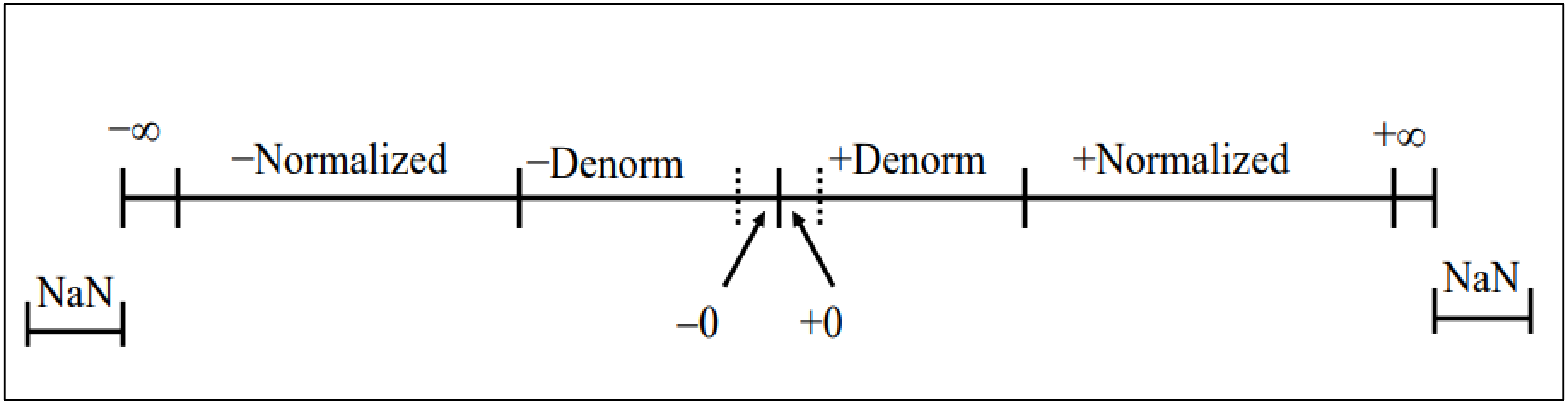
- Single precision: 32 bits


- Double precision: 64 bits


- Extended precision: 80 bits (Intel only)



# Visualization: Floating point encoding




- Normalized value
- Denormalized value
- Special value

# Normalized Value

- <u>When exp ≠ 000…0 and exp ≠ 111…1</u>

**E = Exp – Bias**

  - Exp: unsinged value exp
  - Bias: $2^{k-1} - 1$ , where k is number of exponent bits
    - Single precision: 127
    - Double precision: 1023
  - M= 1+f(fraction)
    - Minimum when 000…0 (M = 1.0)
    - Maximum when 111…1 (M = 2.0 – ϵ)

# Normalized Encoding Example

$$v = (-1)^s \, M \, 2^E$$
$$E = Exp - Bias$$

- Value: `Float F = 15213.0;`
  - $15213_{10} = 11101101101101_2$
    $= 1.1101101101101_2 \times 2^{13}$

- Significand
  - $M = $ $\mathtt{1.\underline{1101101101101}_2}$
  - **`frac=`** $\mathtt{\underline{1101101101101}0000000000_2}$

- Exponent
  - $E = 13$
  - $Bias = 127$
  - $Exp = 140 = \mathtt{10001100_2}$

- Result:

| s | exp | frac |
|---|---|---|
| 0 | 10001100 | 11011011011010000000000 |

# Denomalized Value

- <u>When exp = 000…0</u>

  **E = 1 – Bias**

  - M = 0 + f(fraction)

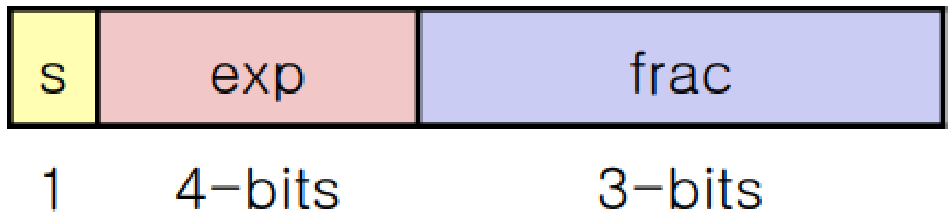
  - Case1) exp = 000…0, frac = 000…0
    - Represent 0
    - There exist  +0 & -0
  - Case2)
    - Very small number closes to 0.0

# Special Values

- Condition: exp = 111…1
  - Case 1) exp = 111…1, frac = 000…0
    - infinity
    - Operation that overflow
    - Both positive and negative
  - Case 2) exp = 111…1, frac ≠ 000…0
    - Not-a-Number (NaN)
    - Represents case when no numeric value can be determined

# Small example:
# 8bits Floating Point

# Tiny Floating Point Example




- 8-bit Floating Point Representation
  - the sign bit is in the most significant bit
  - the next four bits are the exponent, with a bias of 7
  - the last three bits are the **`frac`**

- Same general form as IEEE Format
  - normalized, denormalized
  - representation of 0, NaN, infinity

# Dynamic Range (Positive Only)

| | s | exp | frac | E | Value | |
|---|---|---|---|---|---|---|
| | 0 | 0000 | 000 | -6 | 0 | |
| | 0 | 0000 | 001 | -6 | 1/8*1/64 = 1/512 | closest to zero |
| **Denormalized numbers** | 0 | 0000 | 010 | -6 | 2/8*1/64 = 2/512 | |
| | ... | | | | | |
| | 0 | 0000 | 110 | -6 | 6/8*1/64 = 6/512 | |
| | 0 | 0000 | 111 | -6 | 7/8*1/64 = 7/512 | largest denorm |
| | 0 | 0001 | 000 | -6 | 8/8*1/64 = 8/512 | smallest norm |
| | 0 | 0001 | 001 | -6 | 9/8*1/64 = 9/512 | |
| | ... | | | | | |
| | 0 | 0110 | 110 | -1 | 14/8*1/2 = 14/16 | |
| **Normalized numbers** | 0 | 0110 | 111 | -1 | 15/8*1/2 = 15/16 | closest to 1 below |
| | 0 | 0111 | 000 | 0 | 8/8*1 = 1 | |
| | 0 | 0111 | 001 | 0 | 9/8*1 = 9/8 | closest to 1 above |
| | 0 | 0111 | 010 | 0 | 10/8*1 = 10/8 | |
| | ... | | | | | |
| | 0 | 1110 | 110 | 7 | 14/8*128 = 224 | |
| | 0 | 1110 | 111 | 7 | 15/8*128 = 240 | largest norm |
| | 0 | 1111 | 000 | n/a | inf | |

# Assembly Language

# Programming Language

- Human-friendly language
- C, JAVA, Python, …
- What you program and see

```
int add(int a, int b){
    return a + b;
}
```

# Assembly Language

- Machine-friendly language
- Result of ‘compile’
- What machine really sees and runs

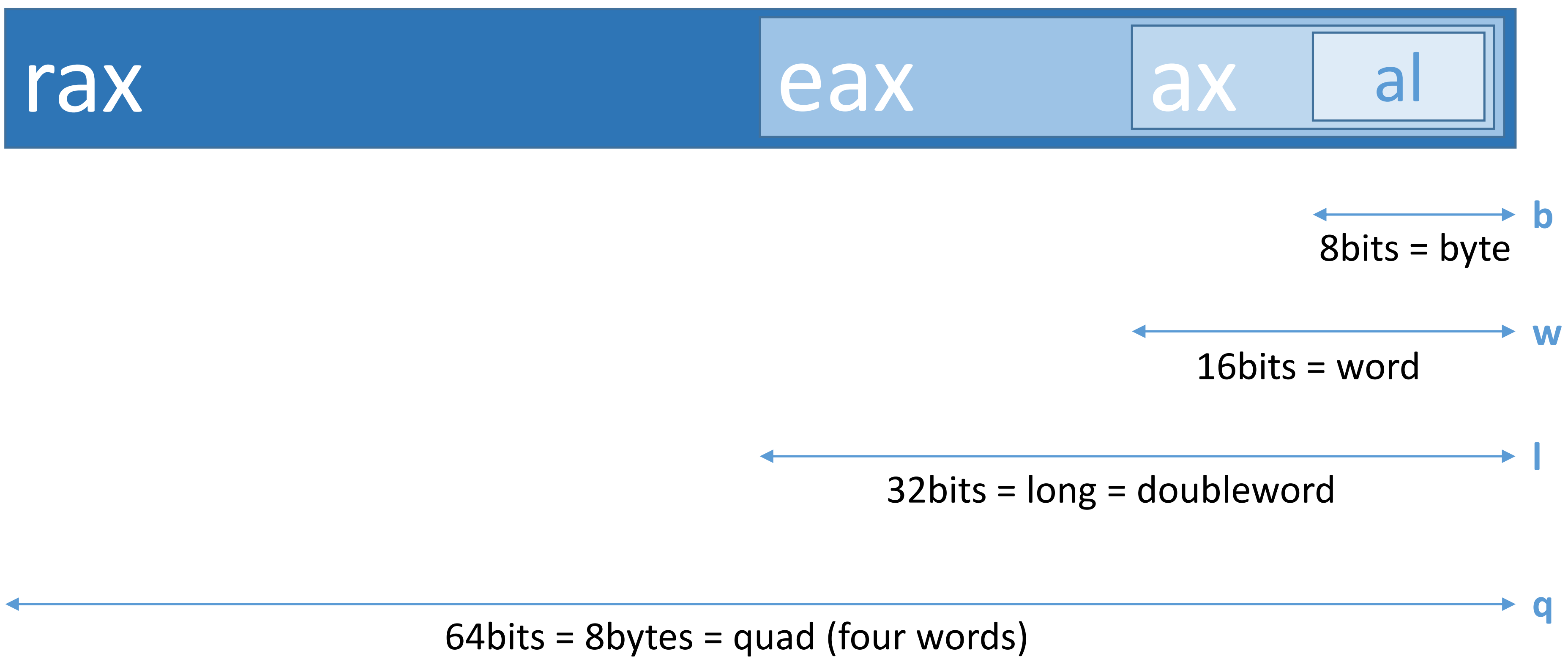
```
(gdb) disas add
Dump of assembler code for function add:
   0x00000000004004ed <+0>:     push   %rbp
   0x00000000004004ee <+1>:     mov    %rsp,%rbp
   0x00000000004004f1 <+4>:     mov    %edi,-0x4(%rbp)
   0x00000000004004f4 <+7>:     mov    %esi,-0x8(%rbp)
   0x00000000004004f7 <+10>:    mov    -0x8(%rbp),%eax
   0x00000000004004fa <+13>:    mov    -0x4(%rbp),%edx
   0x00000000004004fd <+16>:    add    %edx,%eax
   0x00000000004004ff <+18>:    pop    %rbp
   0x0000000000400500 <+19>:    retq
End of assembler dump.
```

# Registers

- A quickly accessible storage available to CPU
- General purpose registers:
  - rax (return value) → eax in 32bit
  - rbx (calle saved) → ebx in 32 bit
  - rcx (4th argument) → …
  - rdx (3rd argument)
  - rsi (2nd argument)
  - rdi (1st argument)
- Special purpose registers:
  - rsp (stack pointer)

Reference ***Figure 3.2***

# Size of Registers



# Assembly Instruction

```
(gdb) disas
Dump of assembler code for function phase_1:
=> 0x0000000000400f00 <+0>:     sub    $0x8,%rsp
   0x0000000000400f04 <+4>:     mov    $0x402470,%esi
   0x0000000000400f09 <+9>:     callq  0x401398 <strings_not_equal>
   0x0000000000400f0e <+14>:    test   %eax,%eax
   0x0000000000400f10 <+16>:    je     0x400f17 <phase_1+23>
   0x0000000000400f12 <+18>:    callq  0x40149a <explode_bomb>
   0x0000000000400f17 <+23>:    add    $0x8,%rsp
   0x0000000000400f1b <+27>:    retq
End of assembler dump.
```

# Assembly Op Code

- Unit of operation
- Calculate and save the result to register/memory
- Binary operators:

  ***OP*** SRC, DST ⇔ DST = ***OP***(SRC, DST)

  - mov, sub, add, cmp, …
  - e.g., **add** %eax, %ecx → ecx = eax **+** ecx
- Unary operators:

  ***OP*** DST ⇔ DST = ***OP***(DST)

  - inc, dec, neg, not, …

# Assembly Operands

Operands

**add** $\boxed{\text{\%eax, \%ecx}}$

- Immediate
  - $0x8, $5, $-1
- Register
  - %rsp, %esi, %eax, %r14
- Memory
  - D(Rb, Ri, S)
  - 8(%ebx), 12(%ebx, %ecx, 4)

Reference ***Figure 3.3***

# Normal Operation

| Instruction | Effect | Description |
|---|---|---|
| inc D | D ← D + 1 | Increment |
| add S, D | D ← D + S | Add |
| sub S, D | D ← D – S | Subtract |
| mov S, D | D ← S | Move |
| sal k, D | D ← D<<k | Left shift |
| … | … | … |

# Special Operations

- CLTQ (may see it in bomblab):
  - Convert long to quad : %rax = SignExtend(%eax)
  - eax = 0x05 → rax = 0x05
  - eax = 0x80000000 → rax = 0xffffffff 80000000
- LEA: load effective address
  - Does not dereference memory
  - Used as an arithmetic operator to perform memory address calculations
    - Commonly used for calculating offsets into an array in a loop
  - lea 7(%rdx, %rdx, 4), %rax ⇔ rax = 5 * rdx + 7

# Control: Compare and Flags

- Condition codes:
  - CMP S1, S2 → executes R = S2 – S1, and set ***Flags***
  - TEST S1, S2 → executes R = S1 & S2, and set ***Flags***

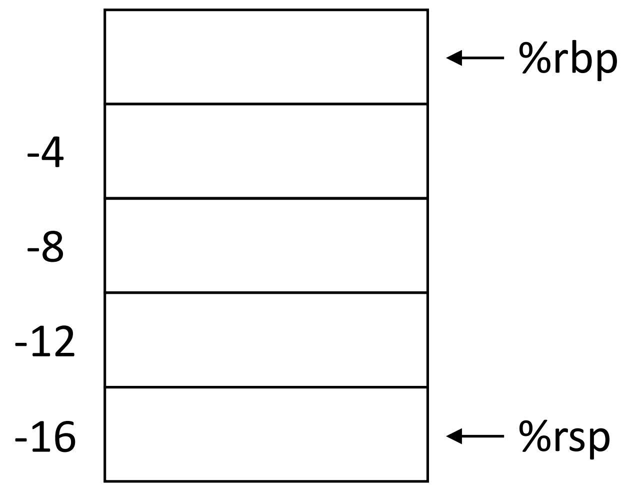
- Flags (about R = S1 + S2)
  - **CF** (Carry Flag) is set if **(unsigned) R < (unsigned) S1**
  - **ZF** (Zero Flag) is set if **R == 0**
  - **SF** (Sign Flag) is set if **R < 0**
  - **OF** (Overflow flag) is set if **(S1 < 0 == S2 < 0) && ( R < 0 != S1 < 0)**

# Control: Jumps

| Instruction | Jump condition | Description |
|---|---|---|
| jmp *Label* | 1 | Direct jump |
| jmp **Operand* | 1 | Indirect jump |
| je *Label* | ZF | Equal / zero |
| jne *Label* | ~ZF | Not equal / not zero |
| jg *Label* | ~(SF^OF)&~ZF | Greater (>) |
| jge *Label* | ~(SF^OF) | Greater or equal(>=) |
| jl *Label* | SF^OF | Less (<) |
| jle *Label* | (SF^OF)\|ZF | Less or equal (<=) |

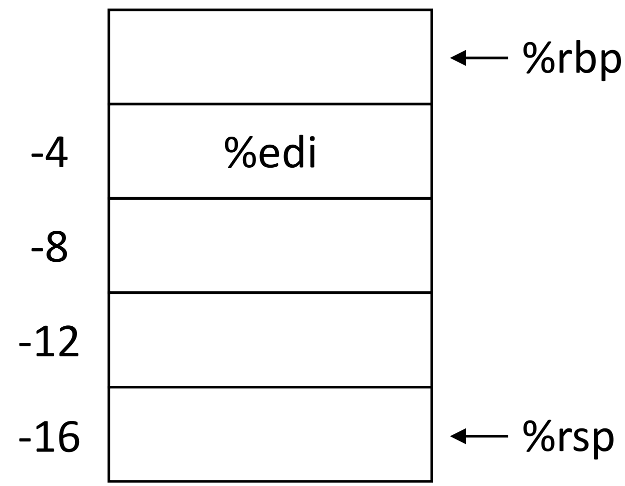
# Example of Control

```
0x000000000040059d <+0>:     push   %rbp
0x000000000040059e <+1>:     mov    %rsp,%rbp
0x00000000004005a1 <+4>:     sub    $0x10,%rsp
0x00000000004005a5 <+8>:     mov    %edi,-0x4(%rbp)
0x00000000004005a8 <+11>:    cmpl   $0xd3,-0x4(%rbp)
0x00000000004005af <+18>:    jne    0x4005bb <print+30>
0x00000000004005b1 <+20>:    mov    $0x400674,%edi
0x00000000004005b6 <+25>:    callq  0x400470 <puts@plt>
0x00000000004005bb <+30>:    nop
0x00000000004005bc <+31>:    leaveq
0x00000000004005bd <+32>:    retq
```



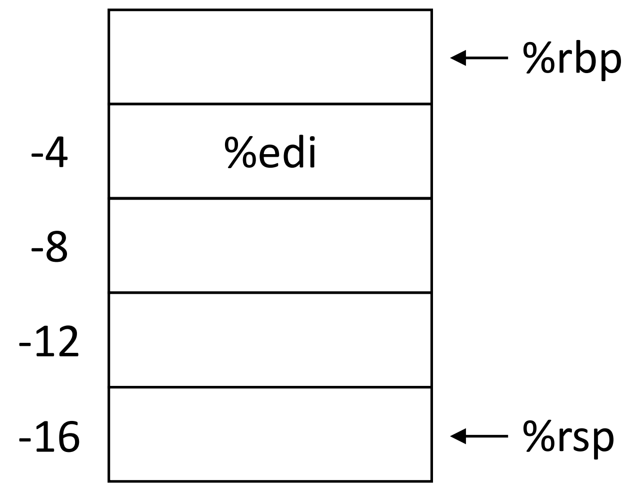
# Example of Control

```
0x000000000040059d <+0>:     push   %rbp
0x000000000040059e <+1>:     mov    %rsp,%rbp
0x00000000004005a1 <+4>:     sub    $0x10,%rsp
0x00000000004005a5 <+8>:     mov    %edi,-0x4(%rbp)
0x00000000004005a8 <+11>:    cmpl   $0xd3,-0x4(%rbp)
0x00000000004005af <+18>:    jne    0x4005bb <print+30>
0x00000000004005b1 <+20>:    mov    $0x400674,%edi
0x00000000004005b6 <+25>:    callq  0x400470 <puts@plt>
0x00000000004005bb <+30>:    nop
0x00000000004005bc <+31>:    leaveq
0x00000000004005bd <+32>:    retq
```



# Example of Control

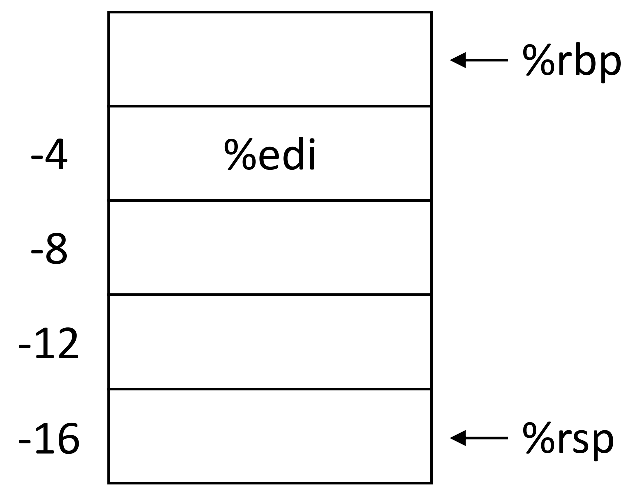
```
0x000000000040059d <+0>:     push   %rbp
0x000000000040059e <+1>:     mov    %rsp,%rbp
0x00000000004005a1 <+4>:     sub    $0x10,%rsp
0x00000000004005a5 <+8>:     mov    %edi,-0x4(%rbp)
0x00000000004005a8 <+11>:    cmpl   $0xd3,-0x4(%rbp)
0x00000000004005af <+18>:    jne    0x4005bb <print+30>
0x00000000004005b1 <+20>:    mov    $0x400674,%edi
0x00000000004005b6 <+25>:    callq  0x400470 <puts@plt>
0x00000000004005bb <+30>:    nop
0x00000000004005bc <+31>:    leaveq
0x00000000004005bd <+32>:    retq
```




**Compare 0xd3 and %edi**

# Example of Control

```
0x000000000040059d <+0>:    push   %rbp
0x000000000040059e <+1>:    mov    %rsp,%rbp
0x00000000004005a1 <+4>:    sub    $0x10,%rsp
0x00000000004005a5 <+8>:    mov    %edi,-0x4(%rbp)
0x00000000004005a8 <+11>:   cmpl   $0xd3,-0x4(%rbp)
0x00000000004005af <+18>:   jne    0x4005bb <print+30>
0x00000000004005b1 <+20>:   mov    $0x400674,%edi
0x00000000004005b6 <+25>:   callq  0x400470 <puts@plt>
0x00000000004005bb <+30>:   nop
0x00000000004005bc <+31>:   leaveq
0x00000000004005bd <+32>:   retq
```




**Compare 0xd3 and %edi**

**→ if not equal, jump to <print+30>**

# Homework

- Due: 09/19 00:00 Wed. (midnight)
- Upload your source file and report to LMS
  - Explain your answer in the final report
  - File name format(again): [student_#]_[your name].c / .doc
  - Only upload **bits.c and report**
- Notice
  - TA only check lab-assn2
  - You don’t have to worry about lab1 assignment

# Homework Instructions (1/4)

- Use the minimum number of operator as you can

- You are allowed to use only the following:
    1. Integer constants 0 through 255(0xFF)
    2. Function arguments and local variables
    3. Unary integer operations ! ~
    4. Binary integer operations & ^ | + << >>

# Homework Instructions (2/4)

- You are expressly forbidden to:
  1. Use any control constructs such as if, do, while, for, switch
  2. Define or use any macros
  3. Call any functions
  4. Use any other operations, such as &&, ||, -, or ?:
  5. Use any data type other than int. This implies that you cann ot use arrays, structs, or unions

# Homework Instructions (3/4)

- You may assume that your machine:
    1. Uses 2s complement, 32-bit representations of integers
    2. Performs right shifts arithmetically
    3. Has unpredictable behavior when shifting an integer by more than the word size

# Problem1

```
/*
 * isLessOrEqual :  x <= y?
 * Example 1) isLessOrEqual(8,7) =  0(False)
 * Example 2) isLessOrEqual(5,7) =  1
 * Legal Ops: ! ~ & ^ | + << >>
 * Max ops: 24
 */
int isLessOrEqual(int x, int y){
 // to be implemented
}
```

# Homework Instruction (4/4)

- For this part of the assignment, you will implement some common single-precision floating point operation.
- You are allowed to use standard control structures
  - conditional, loops
- You also use both int and unsigned data types
- You may not use unions, structs, or arrays
- You may not use any floating point data types, operation, or constants. Instead, any floating-point operand will be passed to the function as having type unsigned, and any returned floating-point value will be of type unsigned

# Problem2

```
/*
 * float_neg(uf): Compute -f
 *
 * Max ops: 10
 */
unsigned float_neg(unsigned uf){
 // to be implemented
}
```

# Problem3

```
/*
 * float_i2f(x): Compute (float) x
 *
 * Max ops: 30
 */
unsigned float_i2f(int x){
  // to be implemented
 }
```

# Problem4

```
/*
 * float_twice(uf): Compute 2*f
 *
 * Max ops: 40
 */
unsigned float_twice(unsigned uf){
 // to be implemented
 }
```

# How to Compile

- After write bits.c file, typing **make**

```
[pcy8201@programming lab2]$ make
gcc -O -Wall -m32 -lm -o btest bits.c btest.c decl.c tests.c
gcc -O -Wall -m32 -o fshow fshow.c
gcc -O -Wall -m32 -o ishow ishow.c
```

- If you want to see your score, typing **./btest**

```
[pcy8201@programming lab2]$ ./btest
Score   Rating  Errors  Function
 1      1       0       bitOr
 3      3       0       logicalShift
 4      4       0       bitCount
 2      2       0       negate
 3      3       0       addOK
 3      3       0       isLessOrEqual
 2      2       0       float_neg
 4      4       0       float_i2f
 4      4       0       float_twice
Total points: 26/26
```

# How to Compile

- If you change bits.c file, you should recompile the codes.
- Type **make clean** and **make**

```
[pcy8201@programming lab2]$ make clean
rm -f *.o btest fshow ishow *~
[pcy8201@programming lab2]$ make
gcc -O -Wall -m32 -lm -o btest bits.c btest.c decl.c tests.c
gcc -O -Wall -m32 -o fshow fshow.c
gcc -O -Wall -m32 -o ishow ishow.c
[pcy8201@programming lab2]$
```

- You can use **./fshow** and **./ishow**

```
[pcy8201@programming lab2]$ ./fshow 203032948

Floating point value 1.186657315e-31
Bit Representation 0x0c1a0974, sign = 0, exponent = 0x18, fraction = 0x1a0974
Normalized.  +1.2034134865 X 2^(-103)
[pcy8201@programming lab2]$
```

```
[pcy8201@programming lab2]$ ./ishow 12345
Hex = 0x00003039,       Signed = 12345, Unsigned = 12345
[pcy8201@programming lab2]$
```