CSO-Recitation 03

CSCI-UA 0201-007

R03: Assessment-01 & Debugging with gdb

Today’s Topics

- Weekly assessment-02
- Weekly assessment-03
- Debugging with gdb

Assessment 02

Answers and explanations

Q1 64-bit processor

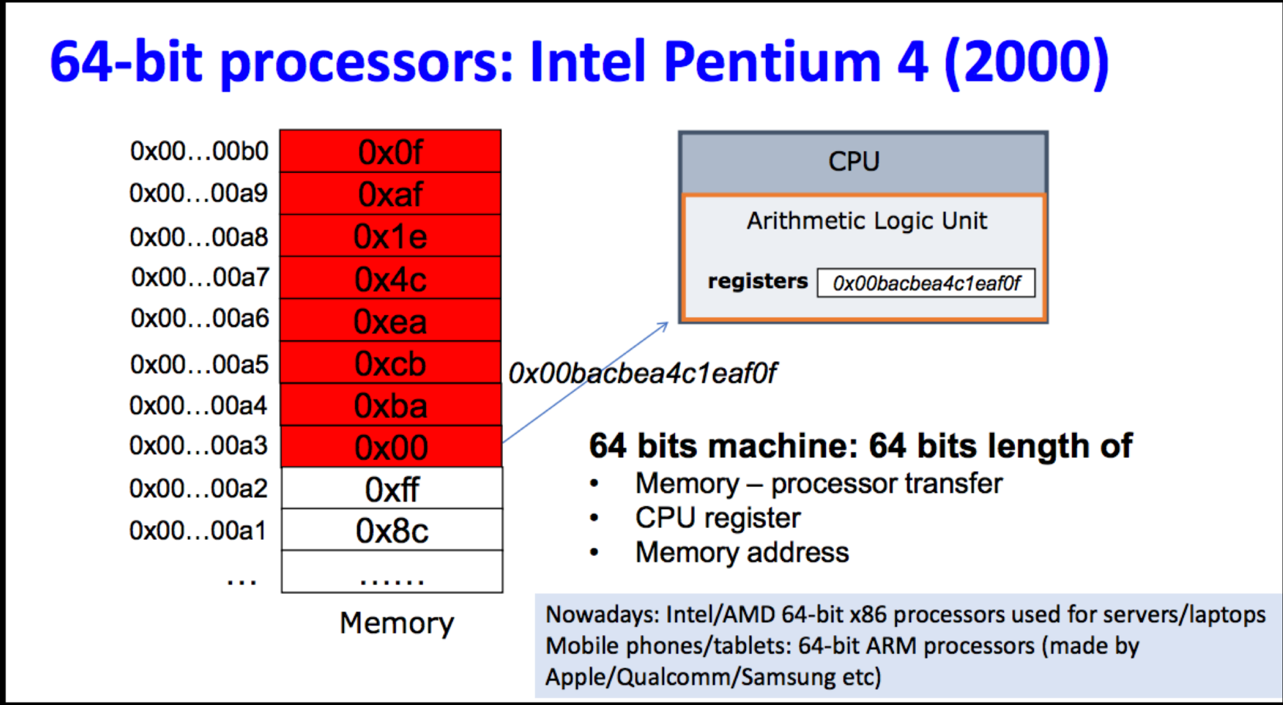
Which of the following statements are true for a 64-bit processor?

[✓ A. its registers are 64-bit in length.]

B. it only supports signed and unsigned integers of exactly 64-bit in length.

C. each memory address stores 64-bit of data.

[✓ D. each memory address is represented by a 64-bit unsigned int.]



64-bit processors: Intel Pentium 4 (2000)

0x00…00b0	0x0f
0x00…00a9	0xaf
0x00…00a8	0x1e
0x00…00a7	0x4c
0x00…00a6	0xea
0x00…00a5	0xcb
0x00…00a4	0xba
0x00…00a3	0x00
0x00…00a2	0xff
0x00…00a1	0x8c
…	……

Memory

CPU

Arithmetic Logic Unit

registers *0x00bacbea4c1eaf0f*

0x00bacbea4c1eaf0f

64 bits machine: 64 bits length of

- Memory – processor transfer
- CPU register
- Memory address

Nowadays: Intel/AMD 64-bit x86 processors used for servers/laptops
Mobile phones/tablets: 64-bit ARM processors (made by Apple/Qualcomm/Samsung etc)

Q2 Normalized Exponential Representation

Which of the following is a **normalized** exponential representation in either binary or decimal?

A. $(0.11)_2*2^1$

[✓ B.] $(1.00)_2*2^{-10}$

C. $(10.11)_2$

D. $(78.5)_{10}*2^{10}$

[✓ E.] $(7.85)_{10}*10^1$

Binary:
Normalized exponential representation:
$\pm M * 2^E$, where $1<=M<2$, $M=(1.F)_2$

Decimal:
Normalized Scientific notation:
$\pm M * 10^E$, where $1<=M<10$

Q3 IEEE Floating Point

What's the value of the 32-bit IEEE floating point with bit pattern 0xc0600000? (Give your answer in the form of regular decimal fractional notation xxx.yyy with no leading nor trailing zeros)

- -3.5

  - 0xc0600000
  - 1100 0000 0110 0000 0000 0000 0000 0000
  - S=1 -> $-M*2^E$
  - $exp=(10000000)_2 = 2^7 = 128$
  - E= exp-bias = exp-127 =1
  - $M=(1.1100...000)_2 = 2^0 + 2^{-1} + 2^{-2} = 1.75$
  - $-M*2^E = -1.75*2^1 = -3.5$

Q4 Signed/Unsigned int

Given a 32-bit bit pattern 0xffffffff, what is the value if we are to interpret the bit pattern as an unsigned int **or** signed int?

A. 2^{31}
B. 2^{32}
C. 2^{31} -1
[✓ D. 2^{32} -1]
[✓ E. -1]
F. - 2^{31}
G. - 2^{32}
H. - 2^{31} +1
I. - 2^{-32} +1
J. None of the above

Unsigned: $-b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i$

Signed: $\sum_{i=0}^{n-1} b_i 2^i$

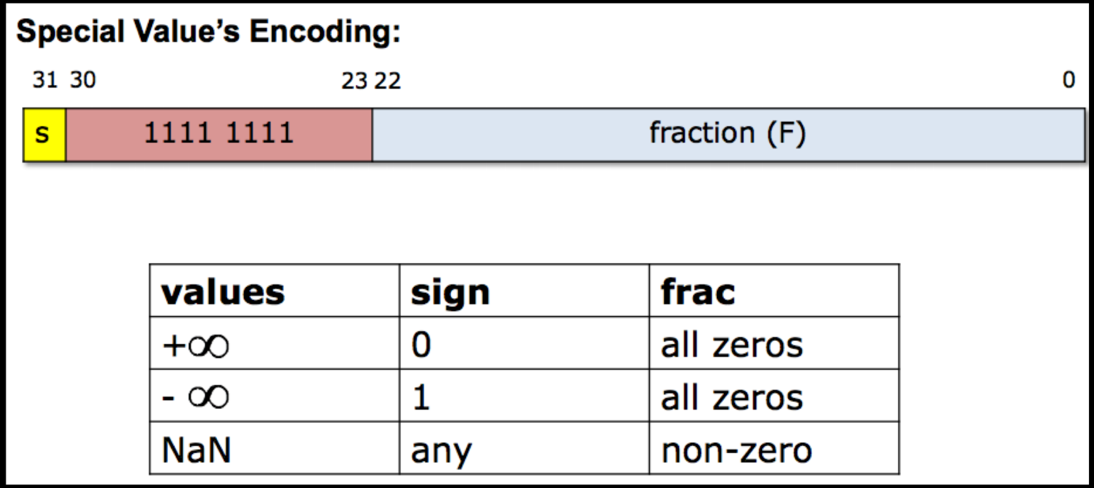
0xffffffff: all $b_i = 1, n = 32$

Q5 IEEE Floating Point

Given a 32-bit bit pattern 0xffffffff, what is the value if we are to interpret the bit pattern as an IEEE 32-bit floating point number.

[✓ A. NaN]
B. ∞
C. $-\infty$
D. 0
E. $\approx 2^{129}$
F. $\approx -2^{129}$
G. None of the above

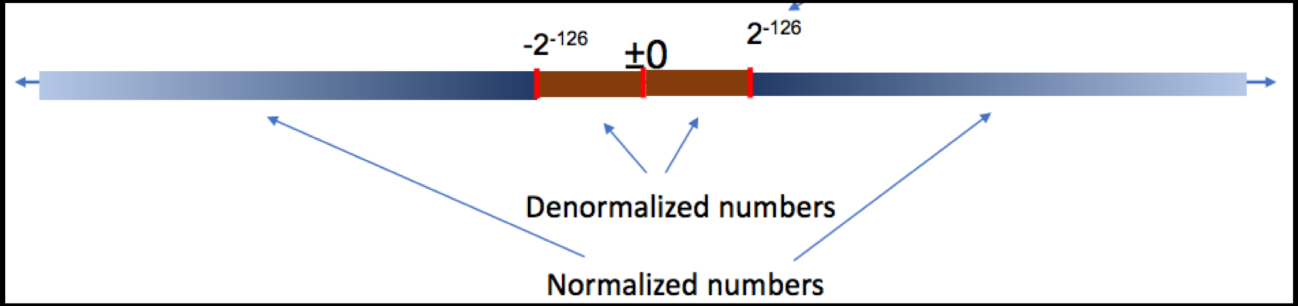
- 0xffffffff
- 1111 1111 1111 1111 11….11
- Special values



Special Value's Encoding:

values	sign	frac
$+\infty$	0	all zeros
$-\infty$	1	all zeros
NaN	any	non-zero

Q6 IEEE floating point



Which of the following statements are true about IEEE floating points?

A. The number zero is represented ~~in normalized encoding~~

[✓ B.] The number zero is represented in denormalized encoding

[✓ C.] All denormalized numbers are closer to zero than normalized numbers

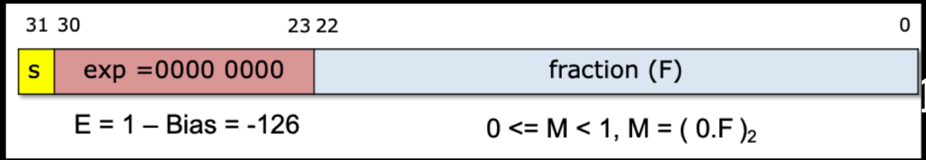
D. Some but not all denormalized numbers are closer to zero than normalized numbers.

[✓ E.] The exponent value (E) in denormalized encoding is 1-127 = -126.

F. The exponent value (E) in denormalized encoding is 0-127 = -127.

Normalized:

Denormalized:



Q7 FP (smallest positve)

What's the bit-pattern of the smallest positive single precision IEEE floating point number?

A. 0x70000001

B. 0x80000001

C. [✓ 0x00000001]

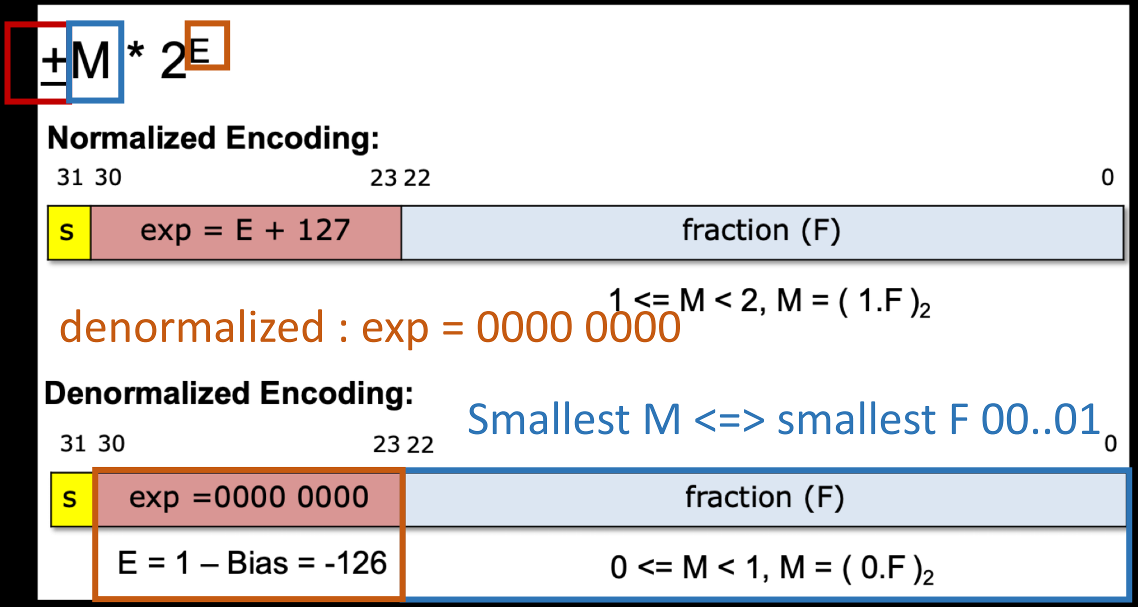
D. 0x0007ffff

E. 0x7f800000

F. 0x7f7fffff

Smallest: denormalized & smallest M

Sign = 0
(positive)



Q8 FP (largest positve)

What's the bit-pattern of the largest positive single precision IEEE floating point number? (\infty∞ does not count)

A. 0x70000001

B. 0x80000001

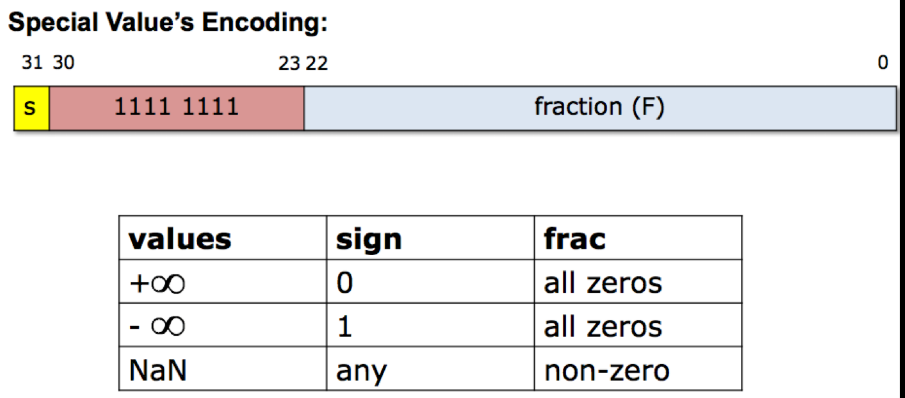
C.

D.

E.

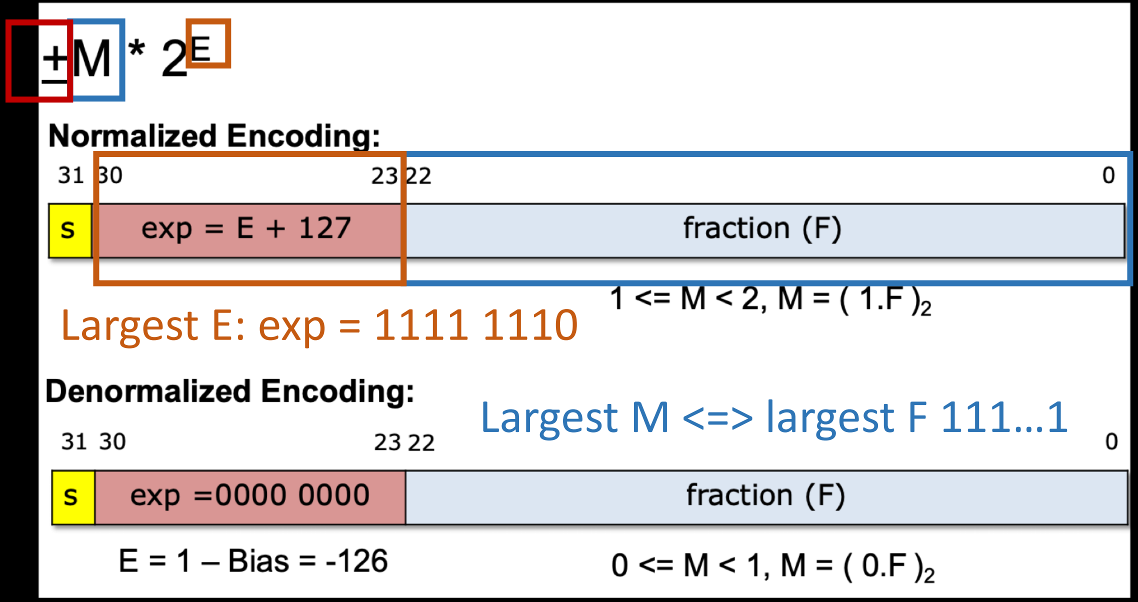
F.

Sign = 0 (positive)



Special Value's Encoding:

values	sign	frac
$+\infty$	0	all zeros
$-\infty$	1	all zeros
NaN	any	non-zero



$\pm M * 2^E$

Normalized Encoding:

$1 <= M < 2, M = (1.F)_2$

Largest E: exp = 1111 1110

Denormalized Encoding:

Largest M <=> largest F 111…1

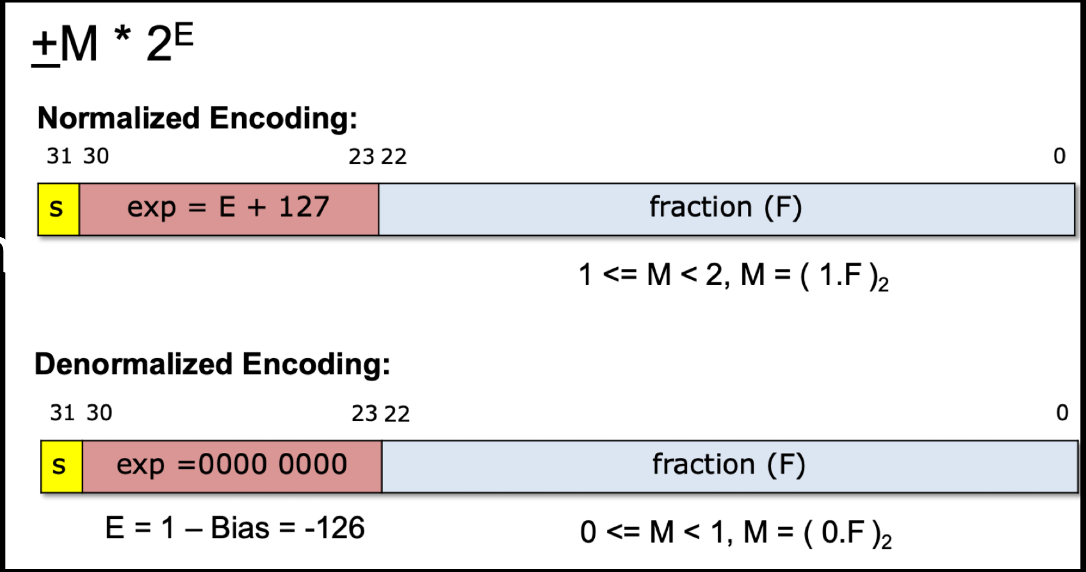
E = 1 – Bias = -126

$0 <= M < 1, M = (0.F)_2$

Q9 FP (precision)

What the highest and lowest precision
points?

A. 2^{–149} and 2^{105}
B. 2^{-150} and 2^{104}
[✓ C. 2^{-149} and 2^{104}]
D. 2^{-150} and 2^{105}
E. 2^{-23} and 2^{23}
F. 2^{-126} and 2^{127}
G. 2^{-127} and 2^{127}



$$(x.b_1 \ldots b_{23})_2 * 2^E = x * 2^E + \sum_{i=1}^{23} b_i * 2^{E-i}$$

It's precision is E-23 (the smallest change you can make is flipping b_{23} which gives 2^{E-23} of difference)

Highest precision -> smallest E -> E=-126 -> precison=-126-23=-149

Lowest precision -> largest E -> E=127 -> precison=127-23=104

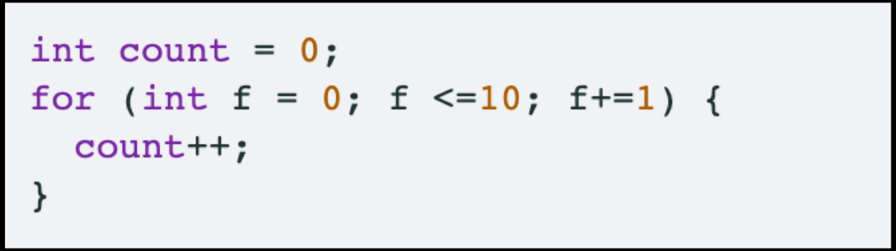
Q10 Counting using FP (NO!!!!)

On Snappy1, the “f”s look like this:

```
0.000000000000000000000000000000
0.100000001490116119384765625000
0.200000002980232238769531250000
0.300000011920928955078125000000
0.400000005960464477539062500000
0.500000000000000000000000000000
0.600000023841857910156250000000
0.700000047683715820312500000000
0.800000071525573730468750000000
0.900000095367431640625000000000
1.000000119209289550781250000000
```

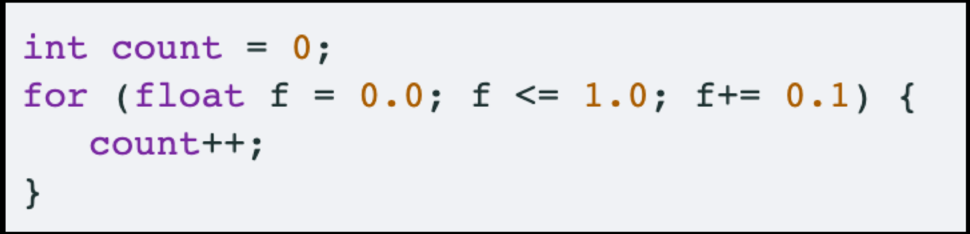
Below are two code fragments:

Code snippet 1:



```
int count = 0;
for (int f = 0; f <=10; f+=1) {
  count++;
}
```

Code snippet 2:



```
int count = 0;
for (float f = 0.0; f <= 1.0; f+= 0.1) {
   count++;
}
```

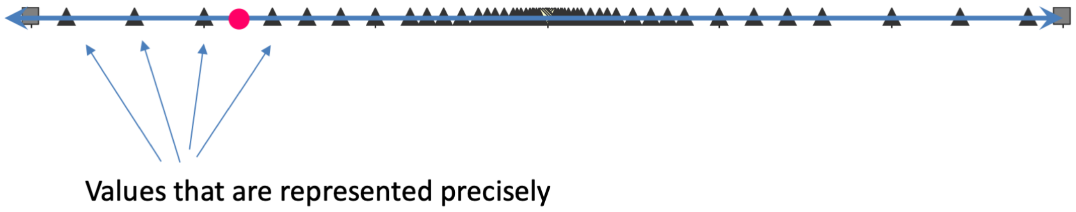
A. 11 vs. 11

[✓ B. 11 vs. 10]

C. 10 vs. 10

D. 10 vs. 11

The computation on floats may not be precise:

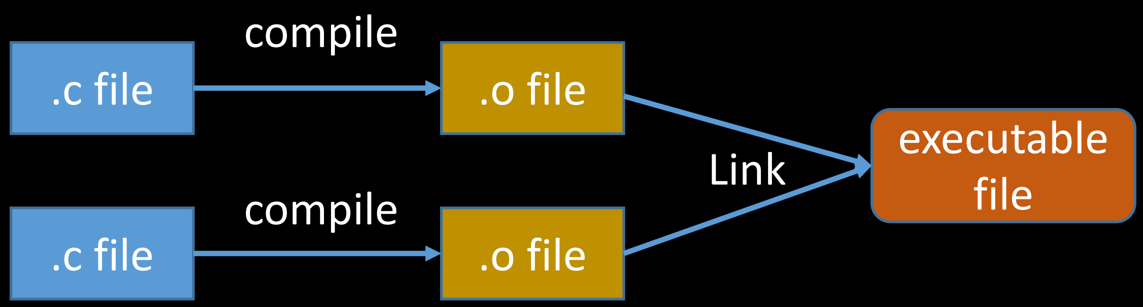


What if the result of computation is at ● ?

Rounding: Use the “closest” representable value *x’* for x.

Assessment 03

Q1 Make



compile
.c file
.o file
compile
.c file
.o file
Link
executable file

What does this make rule do?

```
prog: main.o util.o
        gcc main.o util.o -o prog
```

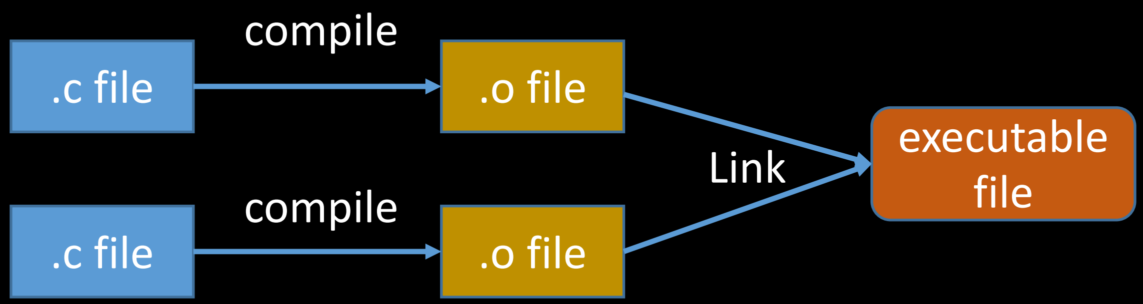
A. It compiles object files main.o and util.o and generates the object file prog

[✓ B.] It links object files main.o and util.o and generates the executable file prog

C. It compiles and links object files main.o and util.o and generates the executable file prog

D. It links object files main.c and util.c and generates the object file prog

Q2 Make



.c file
compile
.o file

.c file
compile
.o file

Link
executable file

Which of the following statements are **true** about the following make rule?

```
main.o: main.c
        gcc -c main.c
```

[✓ A.] It compiles C source file main.c and generates the object file main.o.

B. It links object files main.o and generates the executable file a.out.

C. It compiles and links C source file main.c and generates the executable file a.out.

[✓ D.] Whenever file main.c has changed, running make will invoke gcc -c main.c according to this rule.

E. Running make will always invoke gcc -c main.c according to this rule even if main.c has not changed.

Q3 C program organization

Which of the following statements are **true** about C program?

A. A header file (*.h) includes the implementation of functions to be used in other source files.

[✓ B.] A header file (*.h) includes the signature (aka declaration) of functions to be used in other source files.

C. Every source file (*.c) must contain a main function.

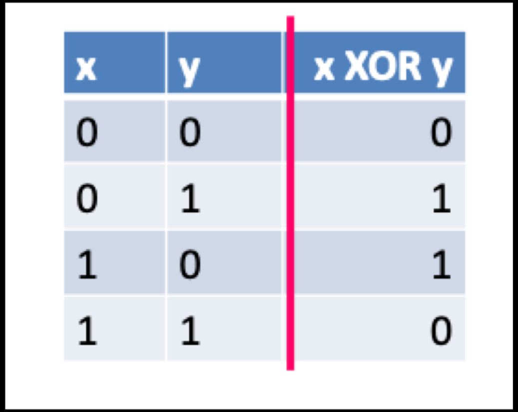
D. Each C binary executable file is compiled from exactly one file.

E. One can execute an object file, e.g. test.o by typing ./test.o

Q4 Bitwise op

What is the value of x^x? (Assuming variable x is unsigned int)

A. It depends on x
[✓ B. Always 0]
C. Always 1
D. Always 0xffffffff



x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

Q5 Bitwise op

What is the value of x^(~x)? (Assuming variable x is unsigned int)

A. It depends on x

B. Always 0

C. Always 1

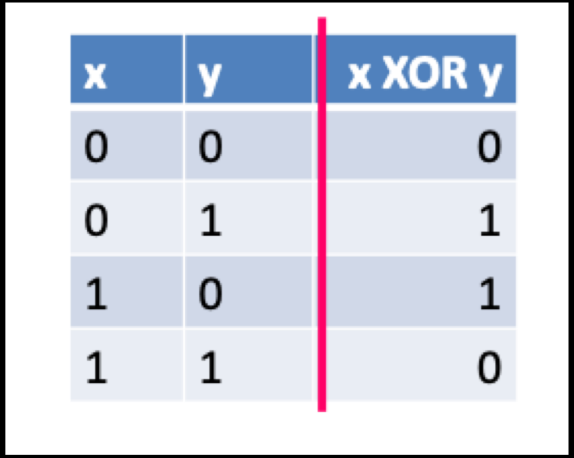
[✓ D. Always 0xffffffff]

~: flips all bits

For each bit in x:

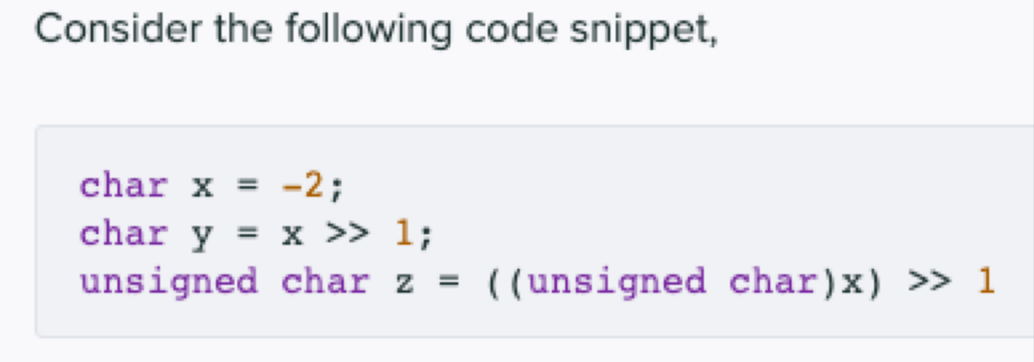
if it is 0: in ~x it will be 1

if it is 1: in ~x it will be 0



x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

Q6 Shift



Consider the following code snippet,

```
char x = -2;
char y = x >> 1;
unsigned char z = ((unsigned char)x) >> 1
```

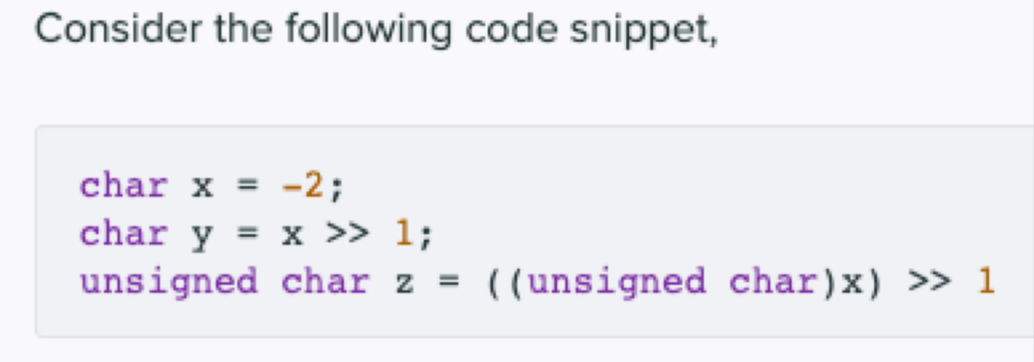
Q6.1 After executing the code snippet, what is the value of y (please write a decimal number as your answer)

Answer: -1

Q6.2 After executing the code snippet, what is the value of z (please write a decimal number as your answer)

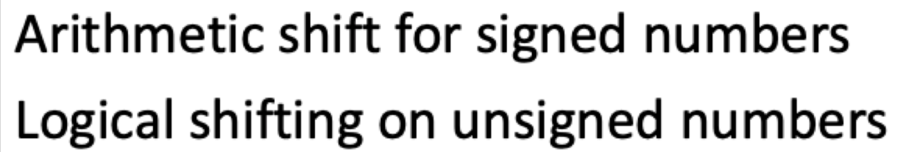
Answer: 127

Q6 Shift



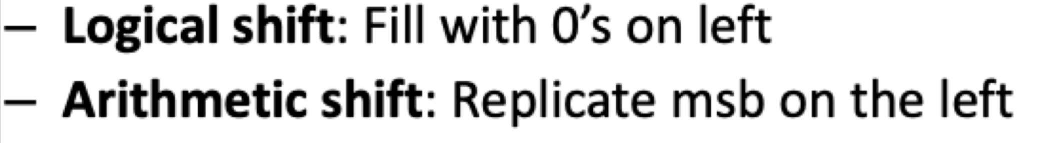
Consider the following code snippet,

```
char x = -2;
char y = x >> 1;
unsigned char z = ((unsigned char)x) >> 1
```



Arithmetic shift for signed numbers

Logical shifting on unsigned numbers



- **Logical shift**: Fill with 0's on left
- **Arithmetic shift**: Replicate msb on the left

X: 111…10

Y (arithmetic shift): 111…11 => signed char => -1

Z (logical shift): 011…11 => unsigned char => 127

Q7 Shift

Which value is the closest to 1<<20

A. 1000

B. [✓ 1 million]

C. 1 billion

D. 2000

E. 2 million

F. 2 billion

- 1<<20
- $0..01\underbrace{00..000}_{}$
- $2^{20} = (2^{10})^2 = 1024^2 \approx (10^3)^2 = 10^6$

Q8 Bit-wise ops

Variable x is of type unsigned int. Which of the following statements returns the most significant byte of x?

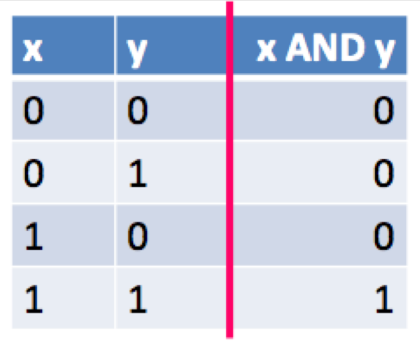
A. (char)x least significant byte

[✓ B. (char)(x >> 24)]

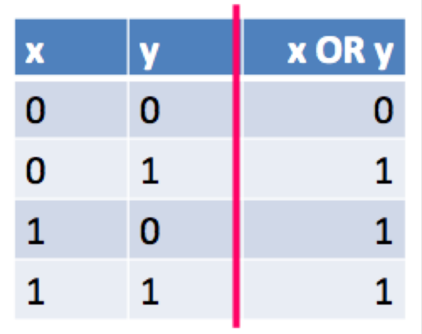
C. (char)(x | 0xff000000) least significant byte

D. (char)(x & 0xff000000) 0x00

E. None of the above



x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1



x	y	x OR y
0	0	0
0	1	1
1	0	1
1	1	1

```
  0xXXXXXXXX          0xXXXXXXXX
| 0xFF000000        & 0xFF000000
------------        ------------
  0xFFXXXXXX          0xXX000000
```

Q9 Floating point (clear exp)

& is often used to mask off bits: b&0 = 0 but b&1 = b

| can be used to turn some bits on: b|1=1

Suppose fi is an unsigned int whose bit pattern represents a single-precision floating point number, which of the following statements clears the exponent field of corresponding floating point number?

A. fi = fi & 0x100fffff

[✓ B. fi = fi & 0x807fffff]

C. fi = fi & 0x80ffffff

D. fi = fi & (0xff<<23)

fi & 0111 1111 1000 00.. 00

E. fi = fi | (0xff<<23)

fi | 0111 1111 1000 00.. 00

[✓ F. fi = fi & (~(0xff<<23))]

fi & 1000 0000 0111 11.. 11

G. fi = fi & (~(1<<23))

fi & 1111 1111 0111 11.. 11

- clear the exponent field
- fi & mask
- mask = 1000 0000 0111 1..1
  - mask = 0x807fffff
  - mask = ~(0xff<<23)
    = ~0x7f800000
    = 0x807fffff

Q10 Local variable

- Consider the following code snippet,
  Which of the following statements are true:

[✓ A.] Running test() will result in assertion failure.

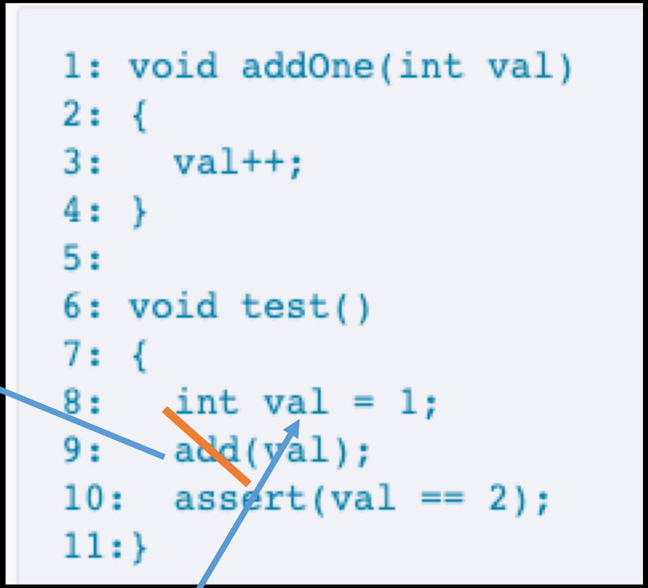
B. Running test() will pass the assertion correctly.

C. The addOne function argument val and the local variable val refer to the same variable

[✓ D.] The addOne function argument val and the local variable val are unrelated.

E. The program will pass test correctly after moving line 8 out of the test function, making val a global variable.

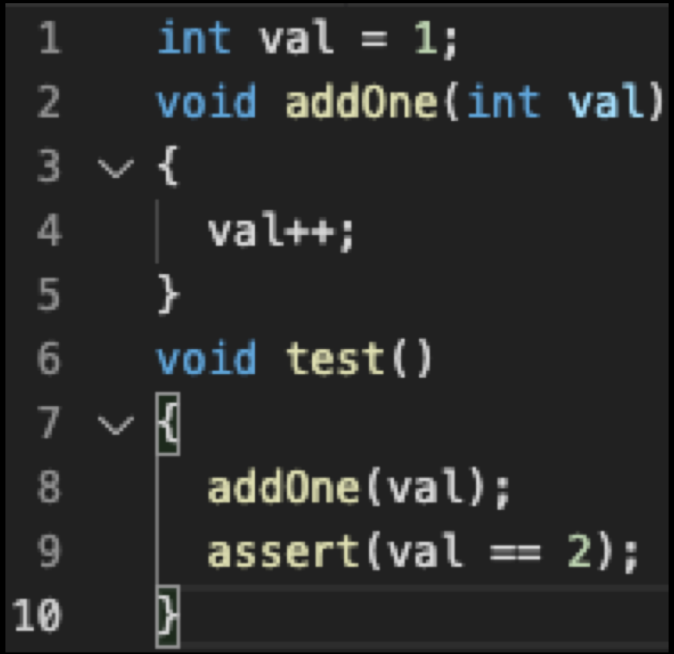
[✓ F.] The program will fail the assertion after moving line 8 out of the test function, making val a global variable.



```
1: void addOne(int val)
2: {
3:    val++;
4: }
5:
6: void test()
7: {
8:    int val = 1;
9:    add(val);
10:   assert(val == 2);
11:}
```

addOne

Q10 Local variable



```
int val = 1;
void addOne(int val)
{
    val++;
}
void test()
{
    addOne(val);
    assert(val == 2);
}
```

- E&F ask what happen if val is globally defined:
- Nothing changes: the global val is shadowed by the definition in function argument
    - The two vals are still unrelated

Getting started with GDB

How to use it and why you should

What is debugging?

- Just because your code compiles doesn’t mean it does what you want
  - It could loop forever, crash, or otherwise just not work correctly
  - Writing tests helps you find out that your code doesn’t work correctly, but you might need more help figuring out why your code doesn’t correctly
- A debugger can help you by providing a number of helpful tools
  - In this class we will use gdb, the GNU debugger

What is debugging?

- GDB lets you
    - Run your program
    - Stop your program at a certain point
    - Print out the values of certain variables at that point
    - Examine what your program is doing
    - Change things within your program to see if it helps

How do you use GDB?

- Add the -g flag when you compile with gcc
  - This flag tells gcc to include debugging information that gdb can use
  - gcc -g main.c -o myprogram
- Run your program with gdb
  - Run gdb ./myprogram
  - You will then be given an interactive shell where you can issue commands to gdb
    - Run your program, look at variables, etc., using the commands
  - To exit the program just type quit (or just q)

Some common gdb commands

Demo: wget https://raw.githubusercontent.com/DingDTest/Recitation-examples/main/main.c

- help
  - Gdb provides online documentation. Just typing *help* will give you a list of topics. Or just type *help command* and get information about any other command.

Short Name	Long Name	What do it do?
r	run	Begins executing the program – you can specify arguments after the word run
s	step	Execute the current source line and stop before the next source line, going inside functions and running their code too
n	next	Continue until the next source line, counting called functions as a single line
p	print	Prints the value of an expression or variable
l	list	Prints out source code
q	quit	Exit gdb

step through the program one line at a time

Some more advanced gdb commands

Set the breakpoint at the beginning of the function

Short Name	**Long Name**	**What do it do?**
b	break	Sets a breakpoint at a specified location (either a *function* name or *line number*)
c	continue	Continues executing after being stopped by a breakpoint
bt	backtrace	Prints out information on the call stack, i.e. where in the program's execution it is being stopped at
f	frame	Prints information on the current frame / allows you to change frames
i	info	Prints out helpful information (e.g. info args and info locals)

Debugging an infinite loop

- Set a breakpoint inside the loop
  - Or just run it and hit control-c (signal)
- *list* the code
  - This is so you can see the loop condition
- *step* over the code
- Check the values involved in the loop condition
  - Are they changing the right way? Are the variables changing at all?

Debugging a crash

- *run* your program
- Use *bt* to see the call stack
  - You can also use *where* to see where you were last running
- Use *frame* to go to where your code was last running
- Use *list* to see the code that ran
- Check the locals and args to see if they are bad