CSO-Recitation 04

CSCI-UA 0201-007

R04: Assessment-02 & Debugging with gdb

Today’s Topics

- Assessment 02
- Breakout exercise
- Debugging with gdb

Assessment 02

Q1 2's complement

What's the bit pattern (2's complement) of 32-bit signed integer -130 in hex format? (Please prefix your answer with 0x)

- 0xffffff7e

- $(130)_{10}$: 00..000 1000 0010
- Step 1-> flip all bits: 11..111 0111 1101
- Step 2-> add 1: 11..111 0111 1110
- 0xffffff7e

Q2 64-bit processor

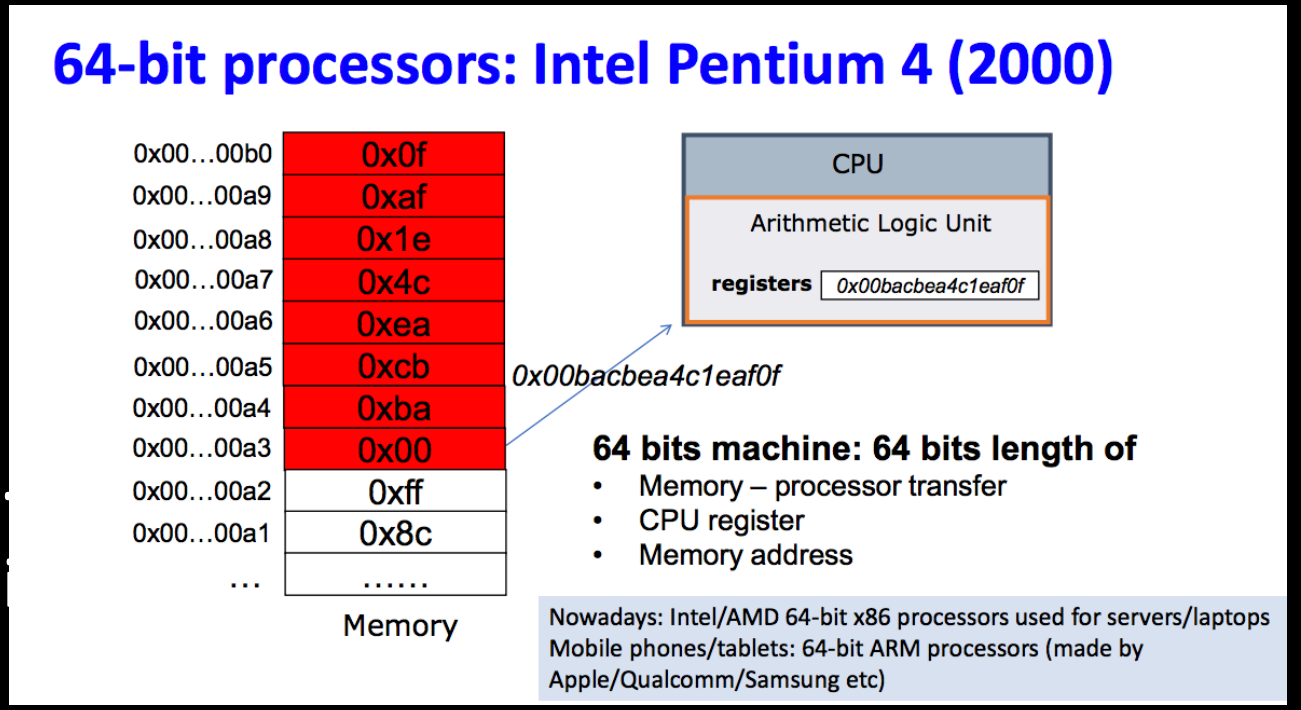
Which of the following statements

A. its registers are 64-bit in length.

B. it only supports signed and unsi
length.

C. each memory address corresponds to 64-bit of data.

D. the CSO lab's virtual machine "emulates" a 64-bit processor.



Q3 Byte ordering

Suppose the byte values stored at memory address a, a+1, a+2, a+3, a+4, a+5, a+6, a+7 are 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08 respectively. If a Little-Endian processor is to load a 4-byte integer from memory at address a into a 4-byte register, what's the 4-byte register value after the load? (Please write your answer in hex, and prefix it with 0x)

- 0x04030201

Little Endian:

- Least significant byte stored at smallest address

Big Endian:

- Most significant byte stored at smallest address

Q4 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

Q5 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

Q6 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

Q7 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

$B.\ \infty$

$C.\ -\infty$

D. 0

$E.\ \approx 2^{129}$

$F.\ \approx -2^{129}$

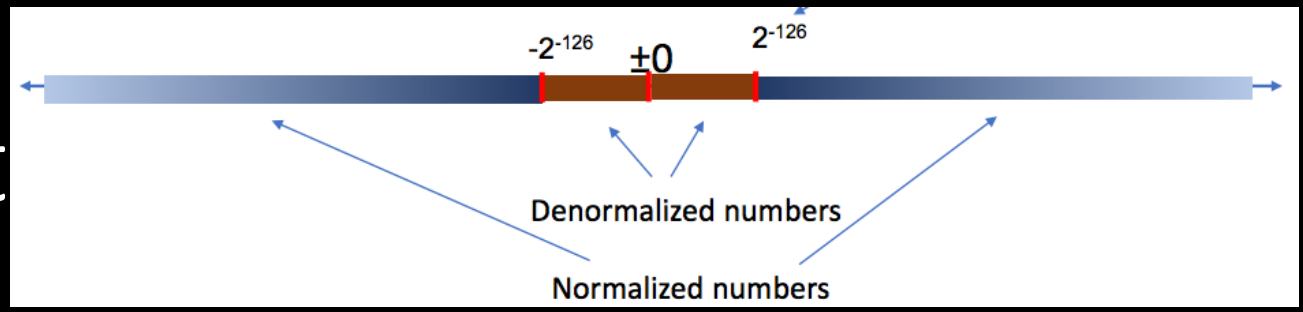
G. None of the above

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

Special Value's Encoding:

values	sign	frac
+∞	0	all zeros
- ∞	1	all zeros
NaN	any	non-zero

Q8 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.

Q9 GDB

For debugging with GDB, the commands you need to do

A. gcc -g main.c -o myprogram
B. gcc main.c
C. gcc main.c -o myprogram
D. gdb ./myprogram

Q10 GDB

To print the value of a variable while debugging with GDB, what command can be used.

A. printf
B. print
C. show
D. p

Breakout exercise

Breakout exercise

Suppose one clears the fraction field (least significant 23 bits) of float f1 = 10.0 and float f2=0.2, what's the resulting floats f1' and f2'?

- f1=10.0
- $M*2^E$ -> f1 = $(1.25)_{10} * 2^3 = (1.F)_2 * 2^3$
- 1.25=2^0+2^-2 -> $(1.01)_2$ -> f1=$(1.0100..00)_2 * 2^3$
- clear the fraction field: f1’ = $(1.00…00)_2 * 2^3 = 8.0$

- f2=0.2 = $(1.6)_{10}*2^{-3} = (1.F)_2 * 2^{-3}$
- clear: f2’ = $(1.00..00)_2 * 2^{-3} = 0.125$

Breakout exercise

Suppose int variable fi stores the bit-pattern for a single-precision floating point, write the line of code that clears the fraction field of the floating point.

- & is often used to mask off bits
- b & 0 = 0, b & 1 = b
- clear the fraction field -> clear the least 23 bits
- fi & mask
- mask: 1111 1111 1000 0000 0000 0000 0000 0000
- mask: 0xff800000
- fi & 0xff800000

Truth table (of boolean

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

Getting started with GDB

How to use it and why you should

Debugging with gdb

- 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

- 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)

Segmentation fault
(core dumped)

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 (*print*) 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 (*info locals*) and args (*info args*) to see if they are bad