Poll: Why is multiplying numbers by 100 easier than multiplying by 128?

EECS 370 - Lecture 2

Binary and

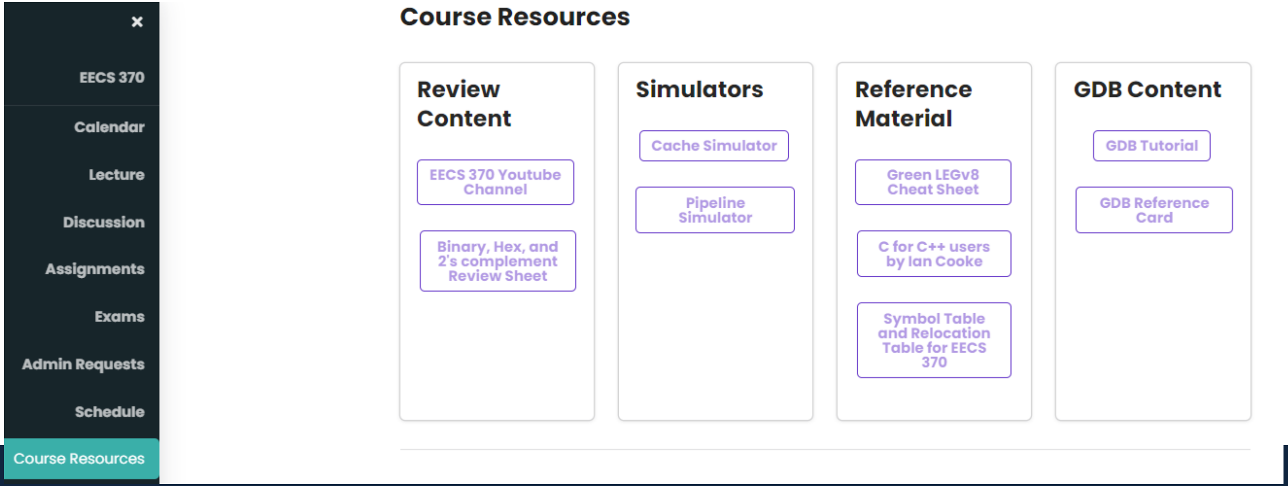
Instruction Set Architecture (ISA)

Announcements

- Project 1 will be posted next week
- Office hours will be in full swing next week as well
- We’ll make an announcement for both
- No lab on Monday (Labor Day)

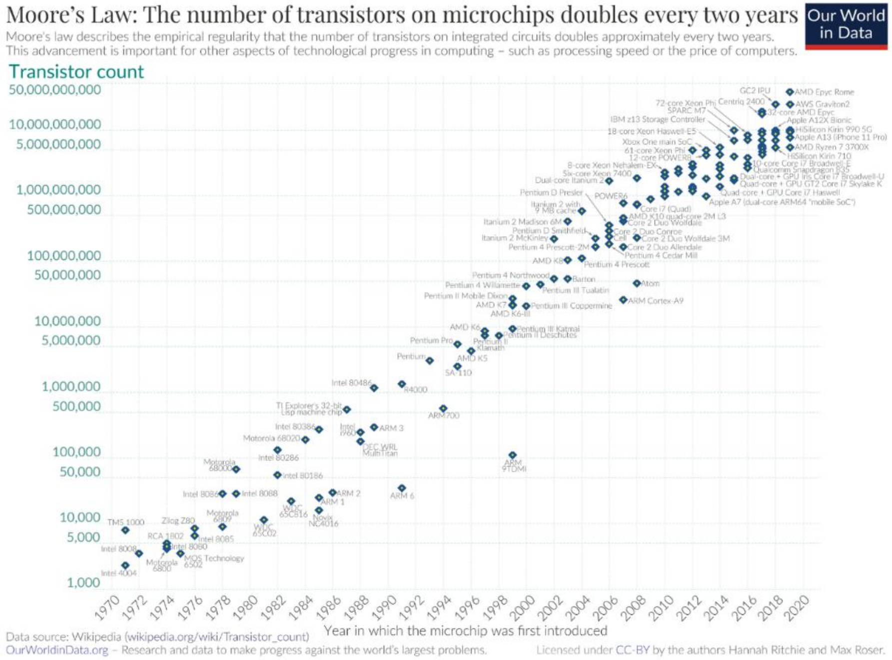
Extra Resources

- Want more examples on binary? Two's complement?
  - See "resources tab" on website
  - Extra videos, review sheets



The Trend of Computing

- Moore's Law



The End of Moore's Law?: Dennard Scaling

- Dennard Scaling: as transistors get smaller their power density stays constant
- Translation: as the number of transistors on a chip grows (Moore’s Law), the power stays roughly constant
- Mid-2000’s Dennard Scaling broke. Why? Transistors got so small that they began to leak a lot of power. Leaking lots of power caused a chip heat up a lot.
- Conclusion: you can put lots of transistors on a chip, but you can’t use them all at full power at the same time.
    - You'll melt the processor!
- This is why newest processors focus on having multiple **cores**

Instruction Set Architecture (ISA) Design Lectures

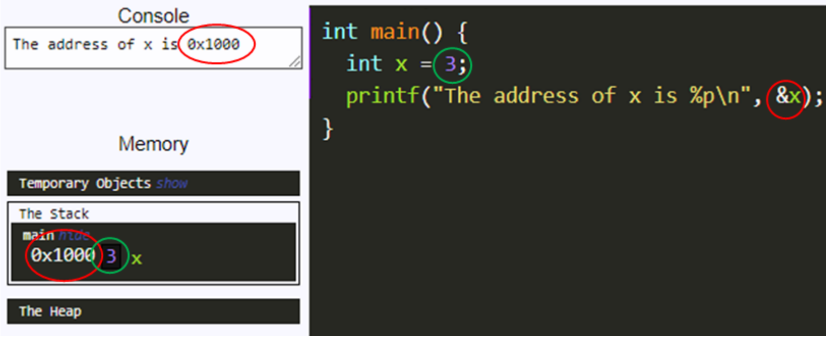
- **Lecture 2: ISA - storage types, binary and addressing modes**
- Lecture 3 : LC2K
- Lecture 4 : ARM
- Lecture 5 : Converting C to assembly – basic blocks
- Lecture 6 : Converting C to assembly – functions
- Lecture 7 : Translation software; libraries, memory layout

Agenda

- **Computer Model and Binary**
- ISAs
  - Registers
  - Control Flow
  - Representing Different Values

Basic Computer Model

- You know from 280 that computers have "memory"
    - Abstractly, a long array that holds values
- Every piece of data in a running program lives at a numerical address in memory
    - You can see the address in C by using the "&" operator



- Most programs work by loading values from memory to the processor, operating on those values, and writing values back into memory

Basic Memory Model

- 1st question in understanding how programs run on computers:
  - How are values actually represented in memory?
- Answer: binary*

Aside: Decimal and Binary

- Humans often represent numbers in base-10 (decimal) because we have 10 fingers (or "digits")
- The n^{th} digit corresponds to 10^n

$$\begin{aligned} &1407 \\ &= 1 \cdot 10^3 + 4 \cdot 10^2 + 0 \cdot 10^1 + 7 \cdot 10^0 \\ &= 1000 + 400 + 00 + 7 \end{aligned}$$

Collection of 8 bits is called a **byte**

- Computers are made of wires with either high or low voltages
- Internally represents values in base-2 (binary) since it has "binary digits"
  - (or bits for short)
- In binary, the n^{th} bit corresponds to 2^n

$$\begin{aligned} &1101 \\ &= 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= 8 + 4 + 0 + 1 \\ &= 13 \end{aligned}$$

Does Bart Simpson count in octal?

Aside: Hexadecimal

- A bunch of 0s and 1s is hard to read for humans
  - But translating to decimal and back is tricky
- Solution: Bases that are a power of 2 are easy to translate between, since a fixed group of bits corresponds to one digit
- In practice, base-16 or **hexadecimal** is used
  - Digits 0-9, plus letters A-F to represent 10-15

Poll: What is 52 in binary? Hex?

Poll: What does your neighbor think of pineapple pizza?

Aside: Hexadecimal

Represent binary using 0b. Hex using 0x. If not specified, it's decimal

- Every 4 bits corresponds to 1 hex digit (since 2^4=16)

```
(binary)     0b 0010 0101 1010 1011
(hexadecimal)0x    2    5    A    B
```

0x25AB

Other Units in this Class

Unit	Number of Bytes
word	4 (in this class)
Kilobyte (KB)	$2^{10} = 1{,}024$
Megabyte (MB)	$2^{20} = 1{,}048{,}576$
Gigabyte (GB)	2^{30} = About a billion
Terabyte (TB)	2^{40} = About a trillion

Agenda

- Computer Model and Binary
- **ISAs**
  - **Registers**
  - Control Flow
  - Representing Different Values

Where do ISAs come into the game ?

Application software

Compilers

ARCHITECTURE – a.k.a. ISA (Instruction Set Architecture)
- Platform-specific
- a limited set of assembly language commands "understood" by hardware (e.g. ADD, LOAD, STORE, RET)

The hw/sw divide

MICROARCHITECTURE (Hardware implementation of the ISA)
- Intel Core i9/i7/i5 implements the x86 ISA (desktop laptop)
- Apple A9 implements the ARM v8-A ISA (iPhone)

Circuits

Devices

How is Assembly Different from C/C++?

- C/C++ instructions operate on **variables**
  - e.g.

    x = i+j;

  - Practically unlimited
- We might guess that assembly instructions act on addresses, e.g.

  0x10000100 = 0x10000200 + 0x10000300
- Problems:
  1. For large memories, this makes the instructions really long
  2. As we'll see later in the course, memory is slow
     - We don't want to access multiple times for every instruction

How is Assembly Different from C/C++?

- Modern ISAs define **registers**
  - Basically a small number (~8-32) of fixed-length, hardware variables that have simple names like "r5"
- In a **load-store architecture** (what we'll use in this class):
  - **load** instructions bring values from memory into a register
  - Other instructions specify register indices (compact and fast)
  - **store** instructions send them back to memory

Example Assembly Code

We'll talk more about how loads / stores work later
(The way we specify addresses is more complicated)

Example ISA (simplified)

```
int a, b, c;
main()
{
  a =  a + b + c;
}
```

C program

Compile →

```
r1 ← load(0x1000)
r2 ← load(0x1004)
r3 ← load(0x1008)
r1 ← add(r1, r2)
r1 ← add(r1,r3)
r1 → store(0x1000)
```

Assembly code

Example Architectures

- ARMv8—LEGv8 subset from P+H text book
  - 32 registers (X0 – X31)
  - 64 bits in each register
  - Some have special uses e.g. X31 is always 0—XZR

- Intel x86 (not discussed much in this class)
  - 4 general purpose registers (eax, ebx, ecx, edx) 32 bits
  - Special registers: 3 pointer registers (si,di,ip), 4 segment (cs,ds,ss,es), 2 stack (sp, bp), status register (flags)

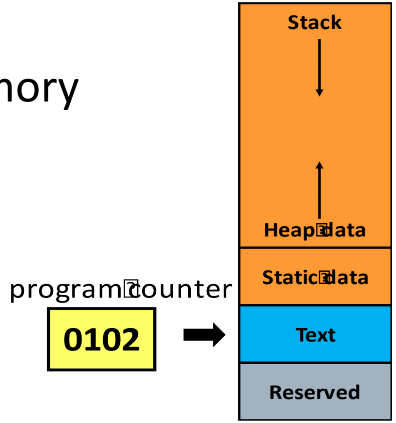
- LC2K (simple architecture made up for this class)
  - 8 registers, 32 bits each

Agenda

- Computer Model and Binary
- ISAs
  - Registers
  - **Control Flow**
  - Representing Different Values

How is Assembly Different from C/C++?

- C/C++: next statement is executed until you get to:
  - function call
  - return statement
  - if statement or for/while loop
  - etc
- Assembly: a program counter (PC) keeps track of which memory address has the next instruction, gets incremented until
  - a "branch" or "jump" instruction
    - Used to change control flow (more later)
  - This model is called a **von Neumann Architecture**



Traditional (von Neumann) Architecture

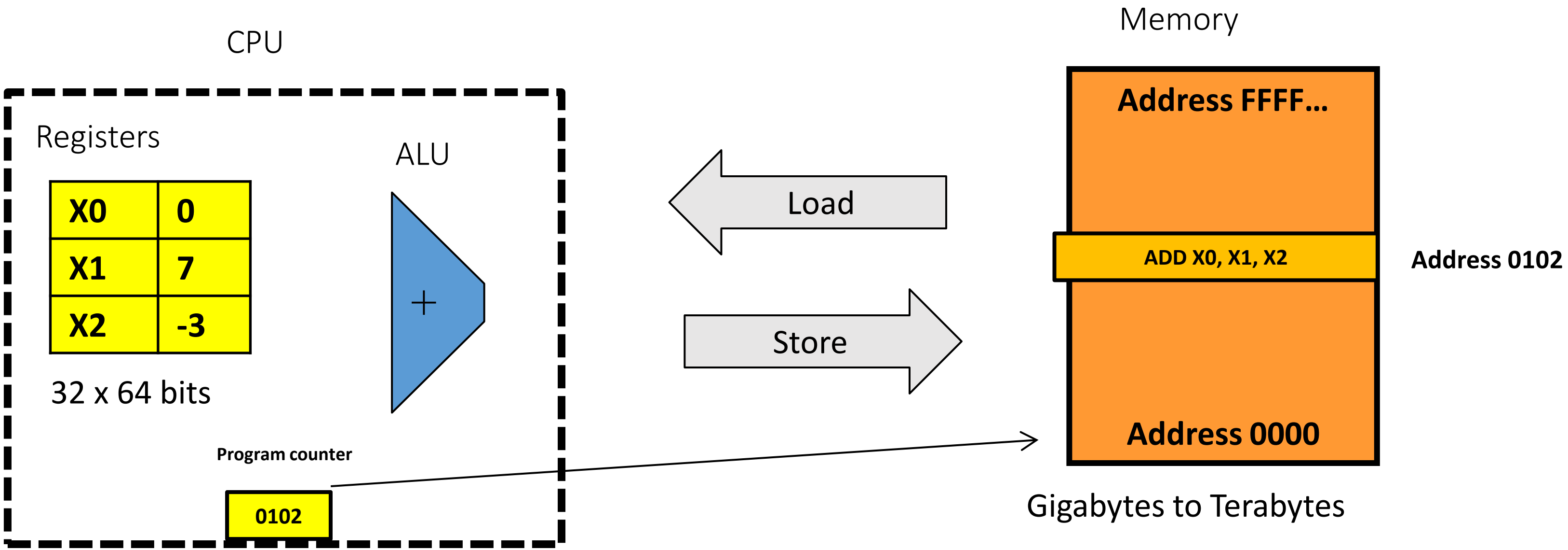
Here’s the (endless) loop that hardware repeats forever:

1. Fetch—get next instruction–use PC to find where it is in memory and place it in instruction register (IR)
    - PC is changed to “point” to the next instruction in the program
2. Decode—control logic examines the contents of the IR to decide what instruction it should perform
3. Execute—the outcome of the decoding process dictates
    - an arithmetic or logical operation on data
    - an access to data in the same memory as the instructions
    - OR a change to the contents of the PC

(Simplified) System Organization

Let's execute this short program (destination register listed first):

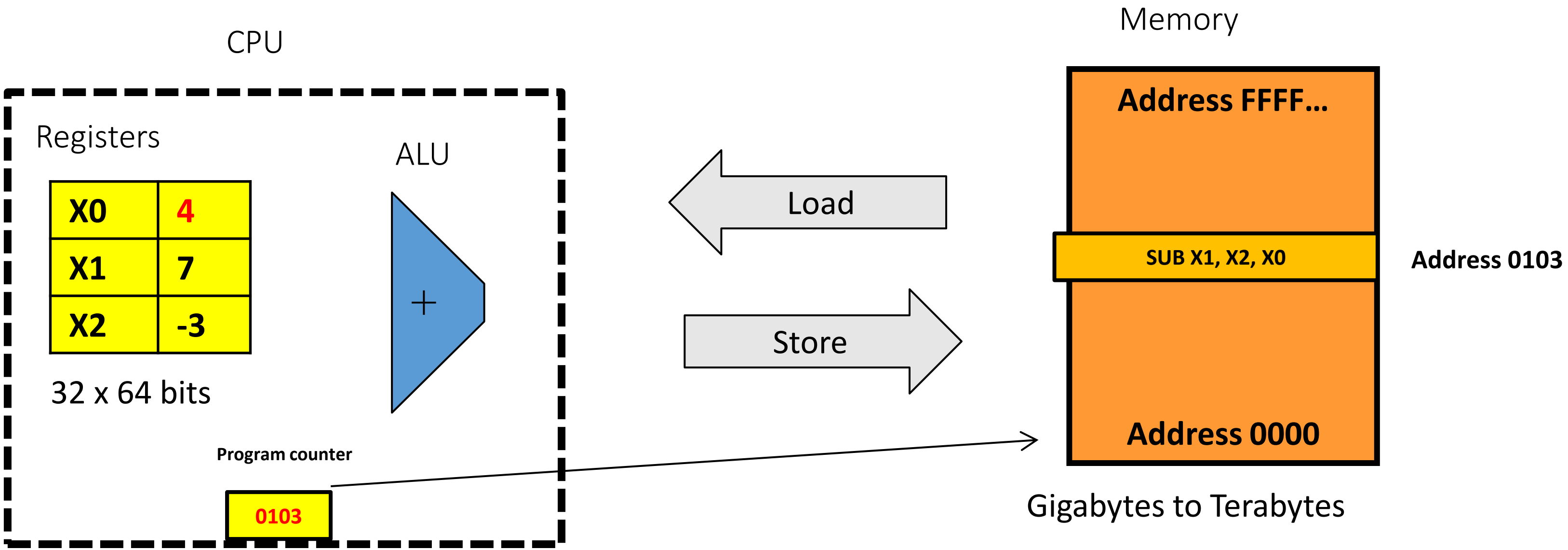
ADD X0, X1, X2
SUB X1, X2, X0



Let's execute this short program (destination register listed first):

ADD X0, X1, X2
SUB X1, X2, X0

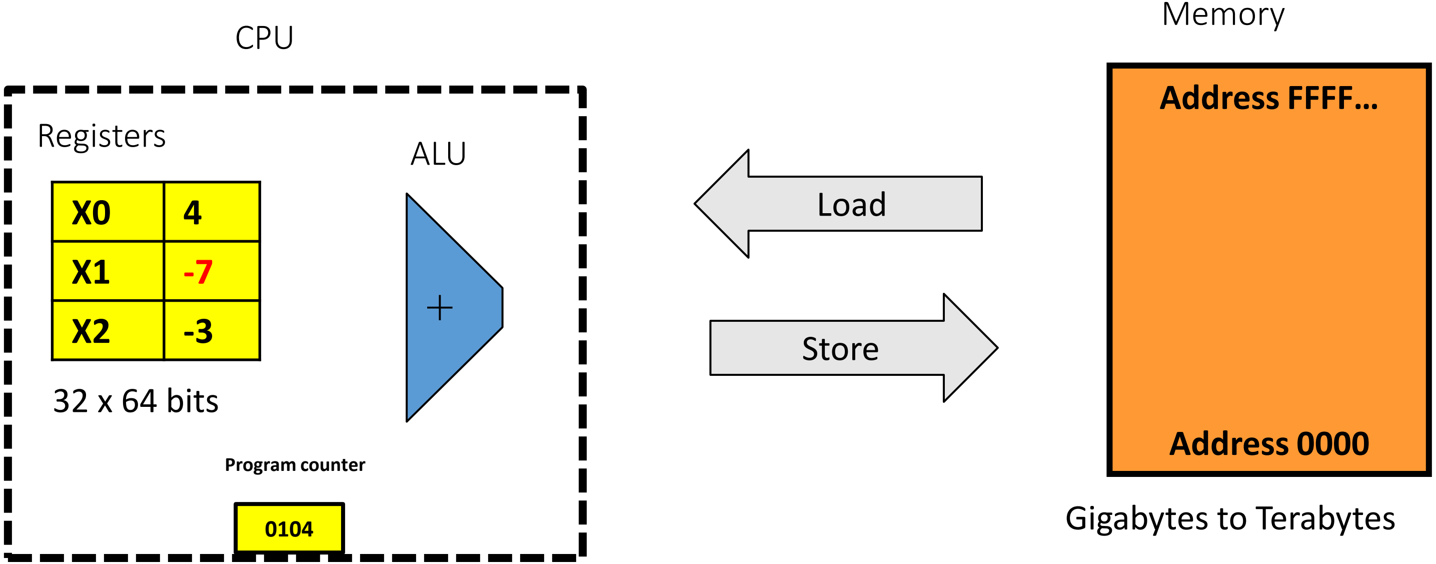
(Simplified) System Organization



Let's execute this short program (destination register listed first):

```
ADD X0, X1, X2
SUB X1, X2, X0
```

(Simplified) System Organization



Assembly Code – ARM Example

ARM v8 ISA

Poll: What are the final contents of X1,X2, and X3?

- What are the contents of the registers after executing the given assembly code (destination register is listed first in ARM)?

	opcode	*d s1 s2imm*
Program:	**ADD**	**X3, X1, X2**
	ADDI	**X3, X3, #3**
	SUB	**X2, X3, X1**

ADDI means "add immediate", the last field is a literal value, not a register index

Initial register file:

X1	25
X2	-4
X3	57

	Initial	(1) ADD X3, X1, X2	(2) ADDI X3, X3, #3	(3) SUB X2, X3, X1
X1	25	25	25	25
X2	-4	-4	-4	-1
X3	57	21	24	24

Agenda

- Computer Model and Binary
- ISAs
    - Registers
    - Control Flow
    - **Representing Different Values**

Different Data Types

- How does memory distinguish between different data types?
  - E.g. int, int *, char, float, double
- It doesn't! It's all just 0s and 1s!
- We'll see how to encode each of these later
- Exact length depends on architectures

How is Assembly Different from C/C++?

- No data types in assembly
- Everything is 0s and 1s: up to the programmer to interpret whether these bits should be interpreted as ints, bools, chars… or even instructions themselves!

```
char c = 'a';
c++; // c is now 'b'

// results in the same assembly as

int x = 97;
x++; // c is now 98

x = (int) c; // this instruction has no effect... why?
```

Minimum Datatype Sizzes

Type	Minimum size (bits)
char	8
int	16
long int	32
float	32
double	64

Representing Values in Hardware

- Unsigned integers represented as we've seen
- Chars are represented as ASCII values
  - e.g. 'a' -> 97, 'b' -> 98, '#' -> 35
- What about negative numbers?
- Fractional numbers?

Negative Numbers

- There are many ways we could represent negative numbers
- Because it will eventually make our hardware simpler, the most common representation is 2's complement

No, not 2's *compliment!*

Two's Complement Representation

- Recall that 1101 in binary is 13 in decimal.

$$\underset{2^3}{1}\quad\underset{2^2}{1}\quad\underset{2^1}{0}\quad\underset{2^0}{1} = 8 + 4 + 1 = 13$$

- 2's complement numbers are very similar to unsigned binary numbers.
  - The only difference is that the first number is now negative.

$$\underset{-2^3}{1}\quad\underset{2^2}{1}\quad\underset{2^1}{0}\quad\underset{2^0}{1} = -8 + 4 + 1 = -3$$

Fun with 2's Complement Numbers

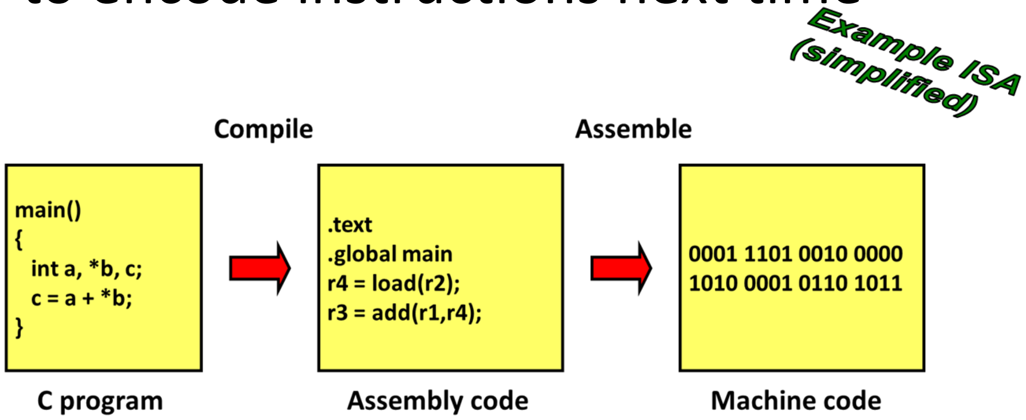
- What is the range of representation of a 4-bit 2’s complement number?
  - [-8, 7]   (corresponding to 1000 and 0111)
- What is the range of representation of an n-bit 2’s complement number?
  - $[-2^{(n-1)}, 2^{(n-1)} - 1]$
- Useful trick: You can negate a 2’s complement number by inverting all the bits and adding 1.
  - 5 is represented as `0101`
  - Negate each bit: `1010`
  - Add 1: `1011` = -8 + 2 + 1 = -5

What about fractional numbers?

- One idea: fixed point notation
  - Have some bits represent numbers before decimal point, some bits represent numbers after decimal point
- Better idea: floating point notation
  - Inspired by scientific notation (e.g. 1.3*10e-3)
  - Allows for larger range of numbers
  - We'll come back to this in a few lectures

Representing Instructions?

- Instructions, not just data, are stored in memory
- So, they must be expressible as numbers
- We'll look at how to encode instructions next time



Next Time

- Finish Up ISAs
- LC2K details