<u>Poll:</u> Why is multiplying numbers by 100 easier than multiplying by 128?

EECS 370 - Lecture 2

ISA and Binary

Unresolved questions from last time

- *"Is there a reason that most programming languages are open source but architectures IP?"*
  - Designing hardware is expensive!
  - Anyone can write software: incentive to develop toolchains and share them with people
  - Actually implementing a processor design takes millions of dollars. Mostly only specialized companies develop toolchains, and sell them to other companies for big $$$
  - But that's gradually changing!
- Remember, you can post questions on Slido, or fill out this "end of lecture" form:

Announcements

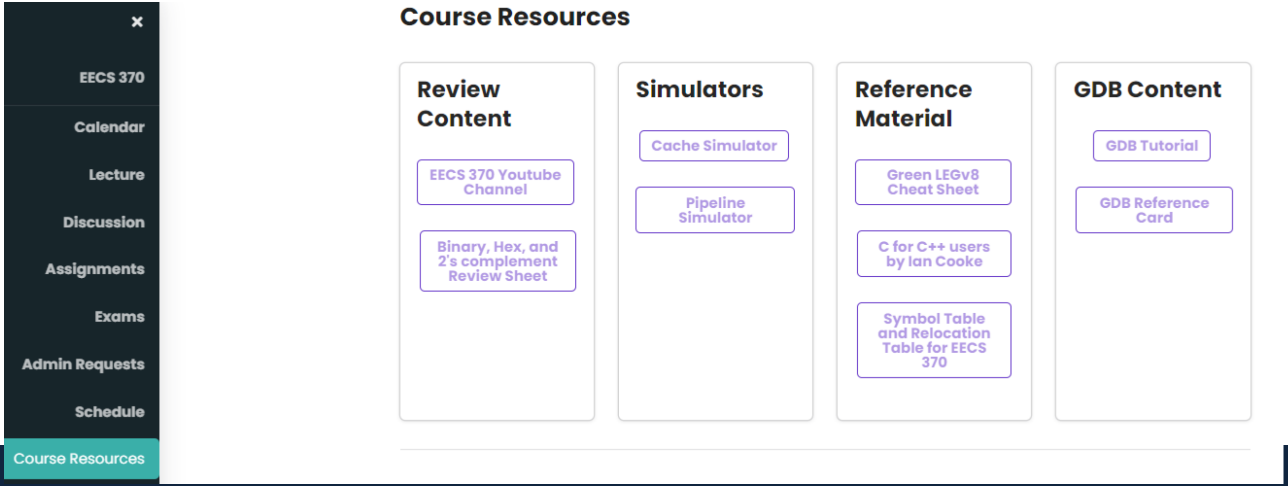
- HW 0
  - Posted on website, due Wednesday
- P1
  - Released soon
- Setup clinic
  - Friday
  - Need help getting your debugger setup?
- OH
  - Going on now: see website->Google Calendar->Office Hours

My Office Hours

- 2 types:
- Group:
  - In-person
  - T/Th - 30 minutes right after class (3901 BBB)
  - Prioritize group questions over individual debugging
  - Starting this Thursday
- Individual:
  - Wednesday – 4:30-5:30
  - Over Zoom (for now) – see website for link
  - One-on-one: any questions welcome

Extra Resources

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

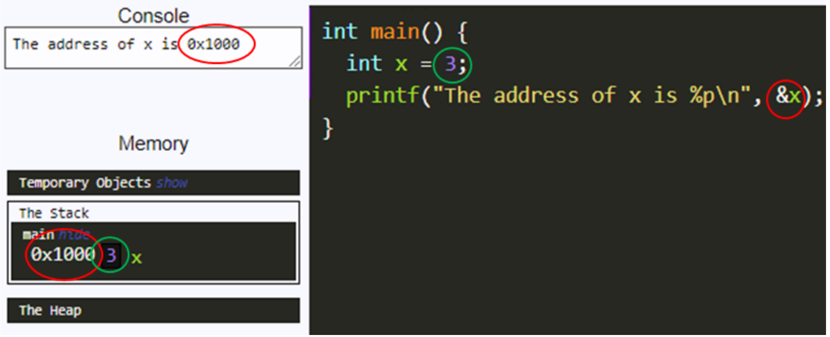


Instruction Set Architecture (ISA) Design Lectures

- **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

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 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 10-16

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
```

Operating on Binary Values

- All values are stored in binary, even when you specify the number in decimal
- It is often convenient to treat values as sequences of bits, rather than values
  - You will need to do this in P1a
- C provides "bitwise operators" to do this
  - Shift ("<<" and ">>")
  - Bitwise boolean ("&", "|", "^", and "~")

Shift Operators

- Shift a value x bits to the left via "<<"
- Inserts "x" zeros to the right (least significant)
- E.g.

```
int a = 60;
int s = a << 2;
```

Shift Operators

- Shift a value x bits to the left via "<<"
- Inserts "x" zeros to the right (least significant)
- E.g.

```
int a = 60;      // 0b0011_1100
int s = a << 2; // 0b1111_0000
```

- "a" is still 60, "s" is 240
- Same idea for ">>", but to the right

shifting x to the left in decimal → multiplying by 10^x

shifting x to the left in binary → multiplying by 2^x

Bitwise operations

- Bitwise operations apply a Boolean operation on each bit of a value (or each pair of bits across two values)

```
int a = 60;    // 0b0011_1100
int b = 13;    // 0b0000_1101

int o = a | b;
```

Bitwise operations

- Bitwise operations apply a Boolean operation on each bit of a value (or each pair of bits across two values)

```
int a = 60;    // 0b0011_1100
int b = 13;    // 0b0000_1101

int o = a | b; // 0b0011_1101
```

- "a" and "b" are the same, "o" is 61
- **&** – and **|** – or **^** – xor **~** – not
- **Very different** from Boolean &&, ||, etc
  - Why?

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

Minimum Datatype Sizzes

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

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++: string multiple operations & operands in one statement:

  x = a + b + c - d

- In assembly, you can
  - only specify one of a fixed number of operations per statement, and
  - have a fixed number of operands per statement

    ADD r1, r2, r3 // equivalent to r1 = r2 + r3

- Complex Instruction Set Computing (CISC) ISAs focus on having many, complex instructions to make programming easier
  - E.g. x86, not discussed in this class
- Reduced Instruction Set Computing (RISC) ISAs focus on having fewer, simpler instructions to ease hardware design
  - E.g. LC2K, ARM (sorta), primary focus of this class

How is Assembly Different from C/C++?

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

    x = i+j;
  - Practically unlimited
- Assembly instructions operate on **memory locations** or **registers**
  - e.g.

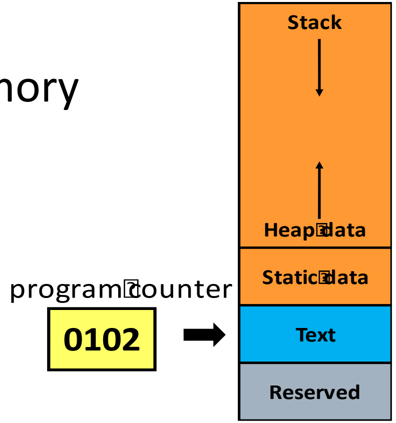
    ```
    add r1, r2, r3
    ld r1, 0x1000
    ```
  - Registers are basically a small number (~8-32) of fixed-length, hardware variables that have simple names like "r5"

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

How is Assembly Different from C/C++?

- C/C++: next line of code 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)



Traditional (von Neumann) Architecture

- We're used to thinking about program code and data being separate
- But in practice, both are stored in the same memory
  - That's why things like "function pointers" in C/C++ work
- This is the basis of the **von Neumann** Architecture

Traditional (von Neumann) Architecture

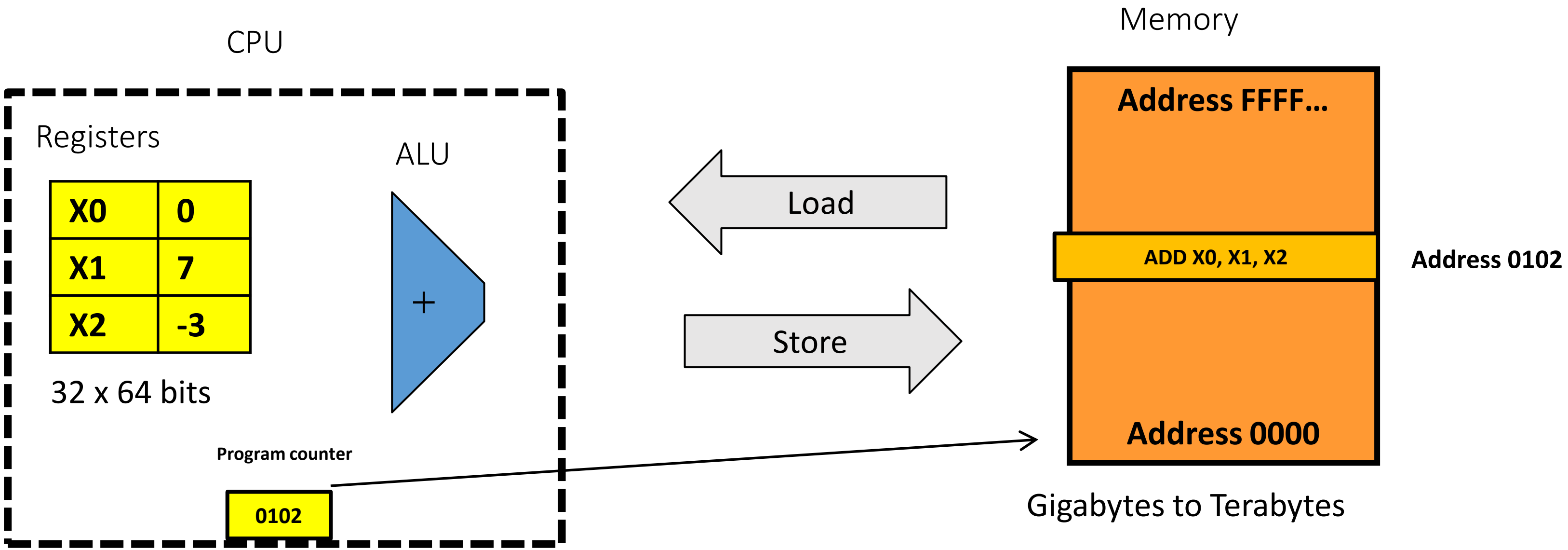
Here's the (endless) loop that hardware repeats forever:

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

Let's execute this short program:

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

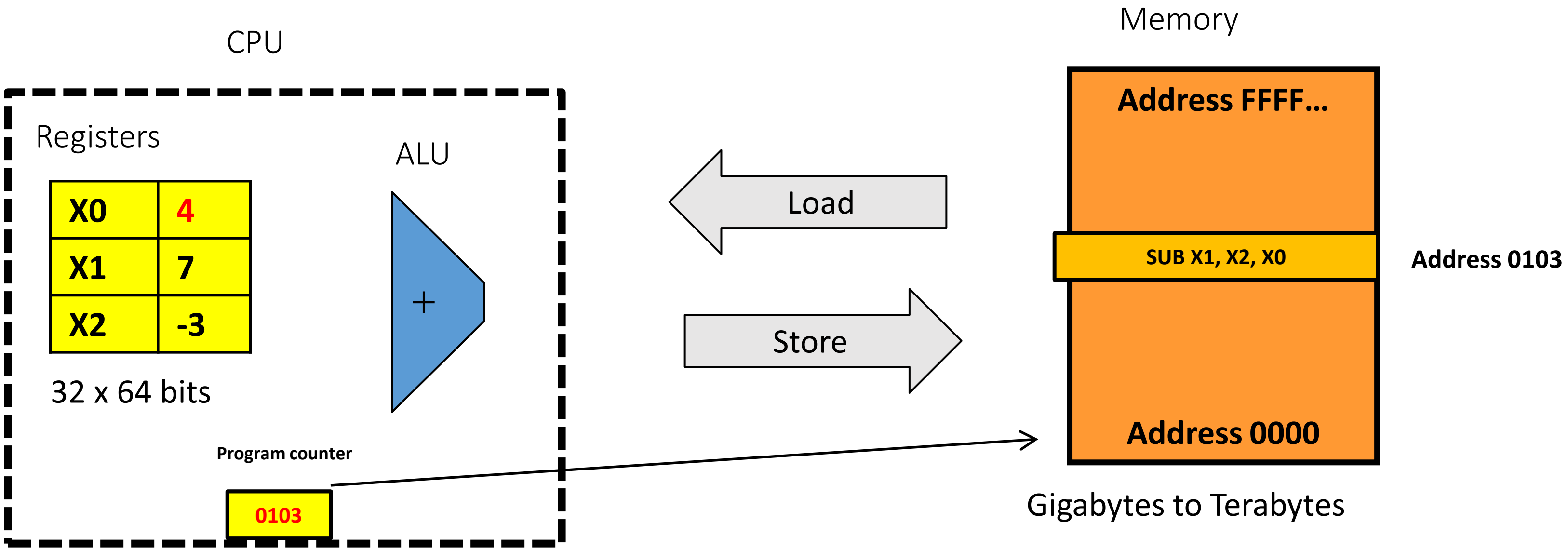
(Simplified) System Organization



Let's execute this short program:

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

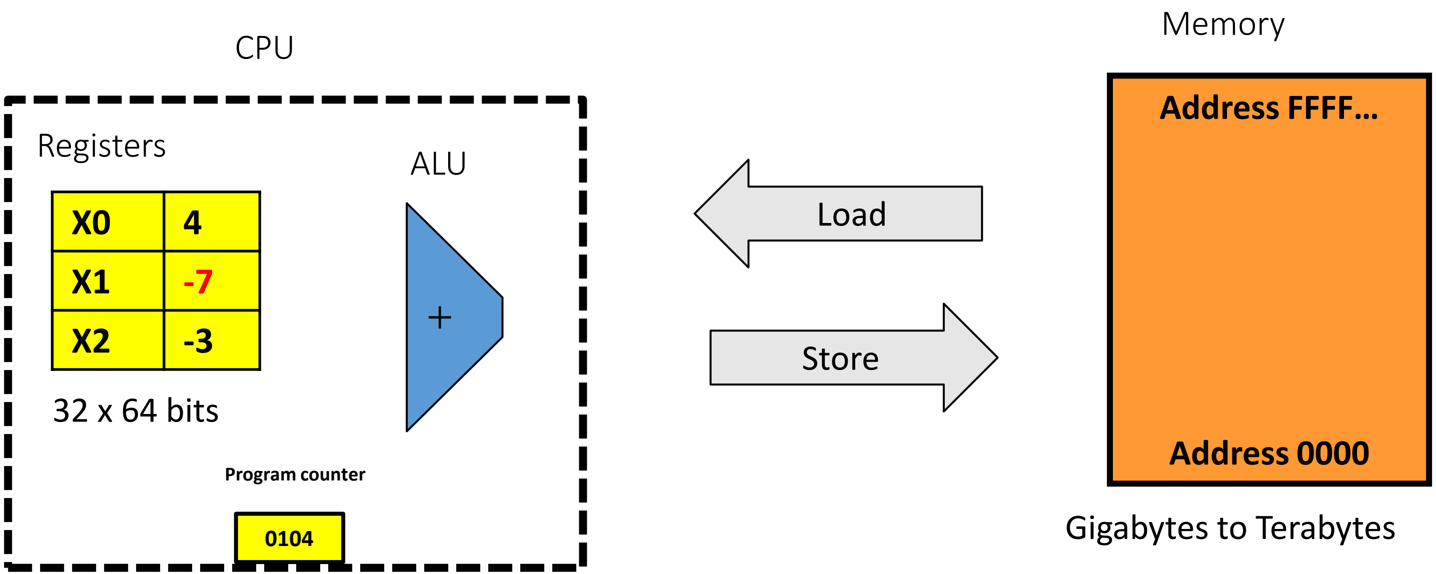
(Simplified) System Organization



Let's execute this short program:

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

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
```

How is Assembly Different from C/C++?

- 1000011 could be interpreted as:
  - Unsigned number: 67
  - A character: 'C'
  - Signed number: -61
  - A float: 9.38e-44
  - LC2K Instruction: 'add 0 0 3'

How did I get these?

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.

$$\begin{array}{cccc} 1 & 1 & 0 & 1 = 8 + 4 + 1 = 13 \\ 2^3 & 2^2 & 2^1 & 2^0 \end{array}$$

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

$$\begin{array}{cccc} 1 & 1 & 0 & 1 = -8 + 4 + 1 = -3 \\ -2^3 & 2^2 & 2^1 & 2^0 \end{array}$$

Poll

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 weeks

Encoding Instructions

- So binary numbers can represent signed and unsigned numbers, chars, and fractional numbers
- But they must also represent instructions themselves!
  - After all, memory is just a collection of 1s and 0s
- We need a way of ***encoding*** instructions in order to store them in memory

Software program to machine code

Example ISA (simplified)

Compile → **Assemble**

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

C program

```
.text
.global main
r4 = load(r2);
r3 = add(r1,r4);
```

Assembly code

```
0001 1101 0010 0000
1010 0001 0110 1011
```

Machine code

Assembly Instruction Encoding

- Since the EDSAC (1949) almost all computers stored program instructions the same way they store data.
- Each instruction is encoded as a number

Example ISA (simplified)

add	R2	R3	R1
011011	010	011	001

$$011011010011001 = 2^0 + 2^3 + 2^4 + 2^7 + 2^9 + 2^{10} + 2^{12} + 2^{13}$$
$$= \mathit{13977}$$

- This is the number stored in memory (in binary)!

Poll: How many different "operation codes" could be supported by this ISA? How many registers?

Next Time

- Finish Up ISAs
- LC2K details
- Lingering questions / feedback? I'll include an anonymous form at the end of every lecture: https://bit.ly/3oXr4Ah