To think about: Why is multiplying numbers by 100 easier than multiplying by 128?

EECS 370 - Lecture 2

Binary and

Instruction Set Architecture (ISA)

Announcements

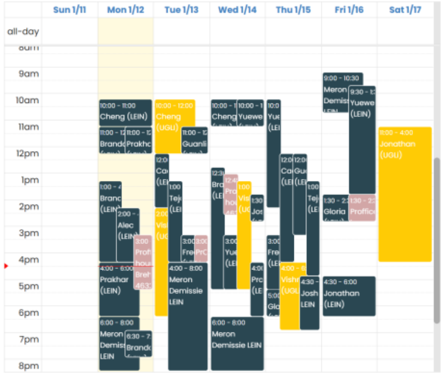
- Project 1
  - P1a due Thursday 1/29
  - P1s due Thursday 2/5
  - P1m due Thursday 2/5
- HW1
  - Posted, due Monday 2/2
- OH
  - Started this week.
  - Schedule posted on Google calendar

Will have what you need by the end of Lecture 3

Can start now, each problem is labelled by what lecture you need.

How to succeed on 370 projects

- Start as soon as you can.
  - For P1 you will know enough to start after Thursday's lecture.
- Fully understand the sample output
  - Don't just look it over, be able to explain every single line, every single value!
- Use office hours
  - We have ~100 hours of office hours/week.
    - Central: Basement of the UGLi
    - North: 2421 Leinweber
    - Proffice hours: varies, see website calendar
  - Asking coding questions is fine, but also specification questions and anything about the sample output too!

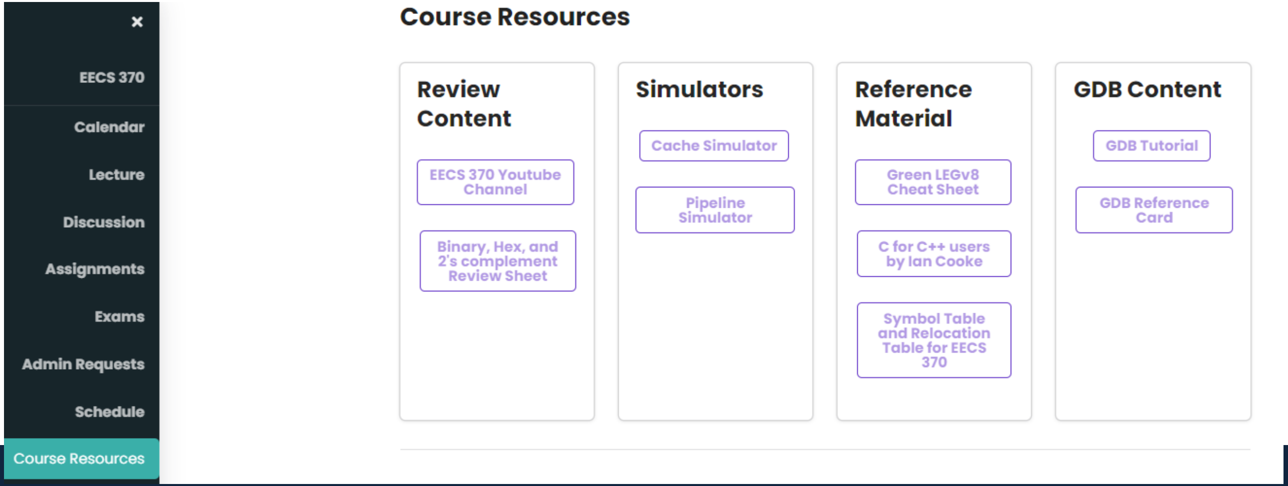


No lab this week

- So no prelab/lab quiz, etc.
- Friday and Tuesday lab sections will be used to give people a chance to get their laptops set up, debuggers working, etc. etc.
  - You can go to any lab section for help (or skip it if you don’t need help) this next Friday and Tuesday.

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

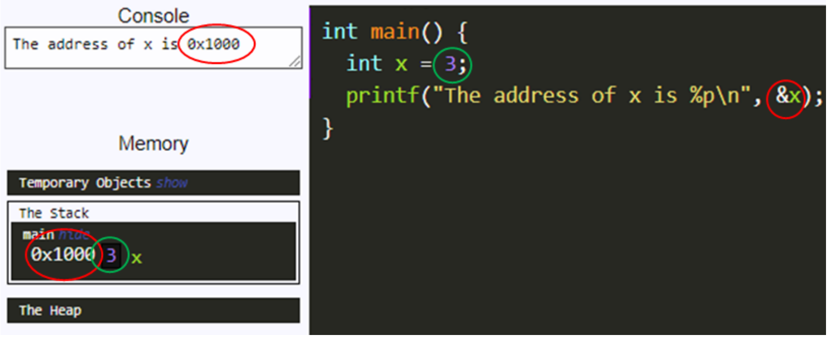
- **Lecture 2: ISA - storage types, binary and addressing modes**
- Lecture 3 : LC2K
- Lecture 4 : ARM
- Lecture 5 : Converting C to assembly – basic blocks
- Lecture 6 : Converting C to assembly – functions
- Lecture 7 : Translation software; libraries, memory layout

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

$$1407$$
$$= 1 \cdot 10^3 + 4 \cdot 10^2 + 0 \cdot 10^1 + 7 \cdot 10^0$$
$$= 1000 + 400 + 00 + 7$$

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

$$1101$$
$$= 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$
$$= 8 + 4 + 0 + 1$$
$$= 13$$

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

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
```

What is 52 in binary? Hex?

`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 P1
- 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;      // 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;     // 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

"Bits is bits"

- 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

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

Agenda

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

Where do ISAs come into the game ?

Application software

Compilers

The application

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 interface

MICROARCHITECTURE (Hardware implementation of the ISA)
- Core i7 implements the x86 ISA
- OMAP 4430 implements the ARM ISA

The implementation

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. This makes the instructions really long
  2. As we'll see later in the course, memory is slow
     - We don't want to go 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 assume 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

Which Instructions Should Be Defined?

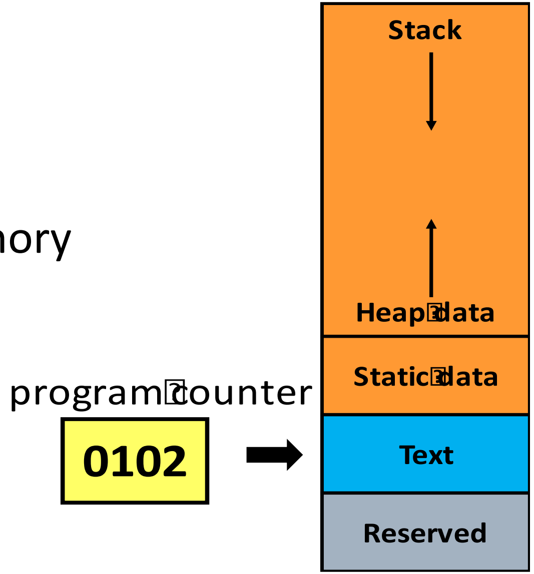
- Complex Instruction Set Computing (CISC) ISAs focus on having many, complex instructions to make programming easier
  - E.g. x86, not discussed in this class
- Reduced Instruction Set Computing (RISC) ISAs focus on having fewer, simpler instructions to ease hardware design
  - E.g. LC2K, ARM, primary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

Agenda

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

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)
  - This model is called a **von Neumann Architecture**



Traditional (von Neumann) Architecture

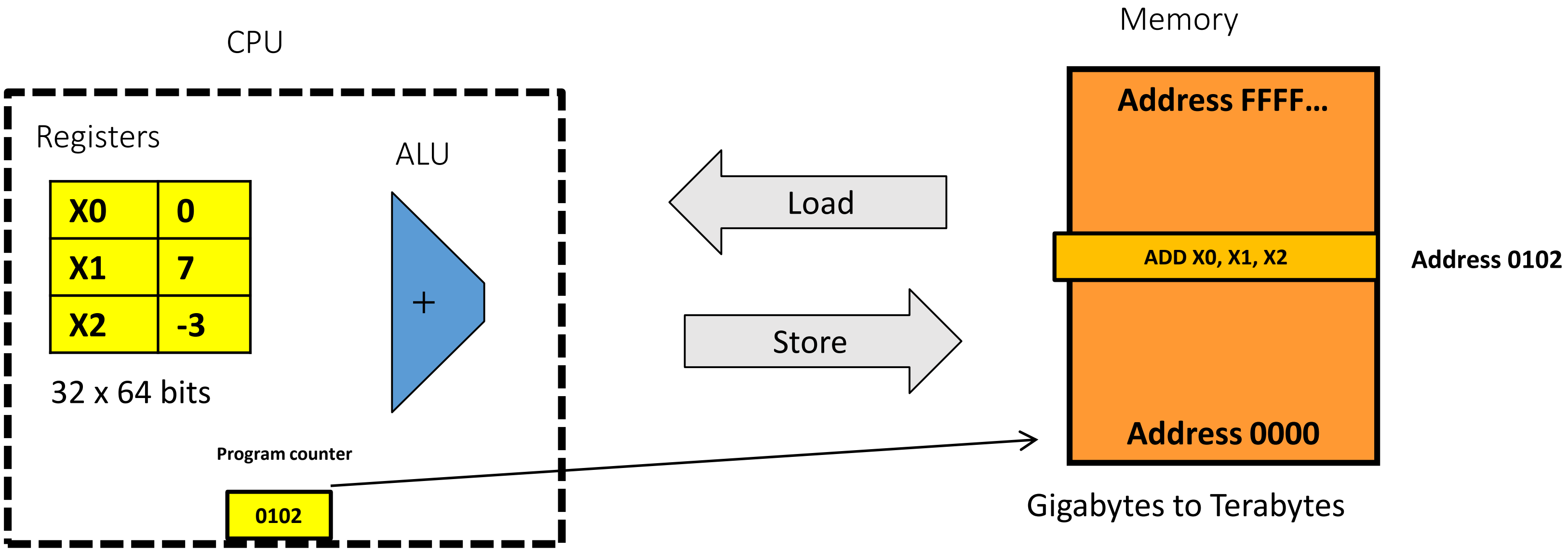
Here's the (endless) loop that hardware repeats forever:

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

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

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

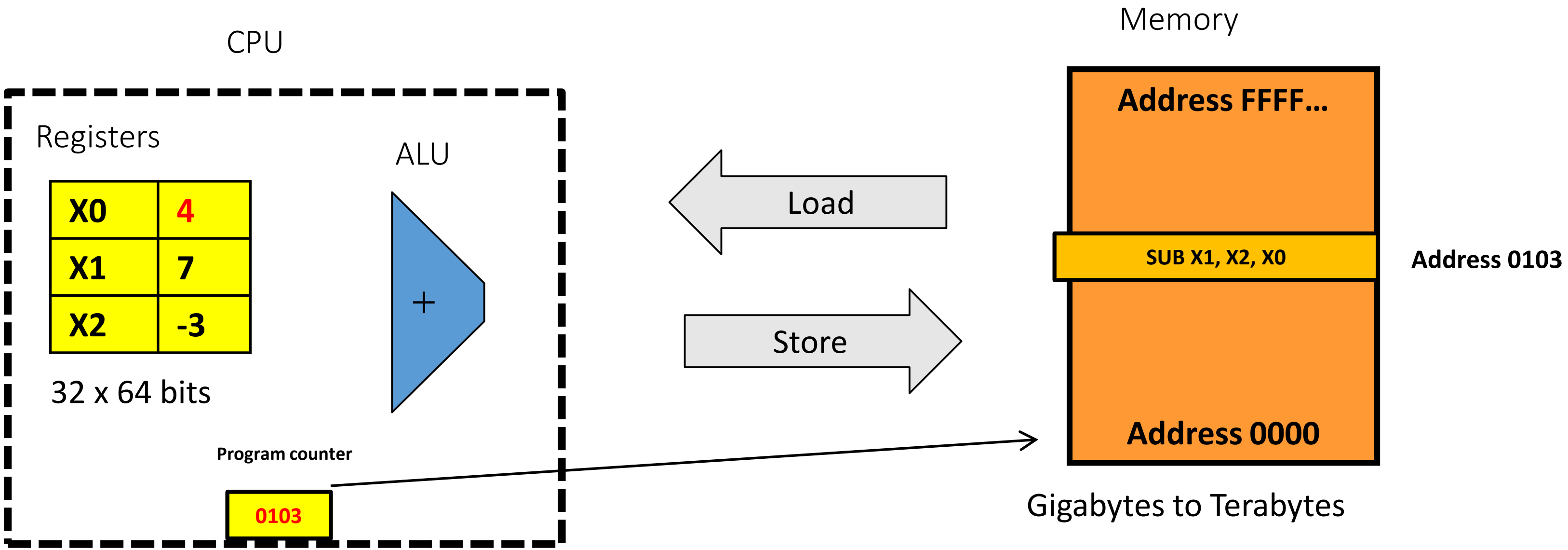
(Simplified) System Organization



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

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

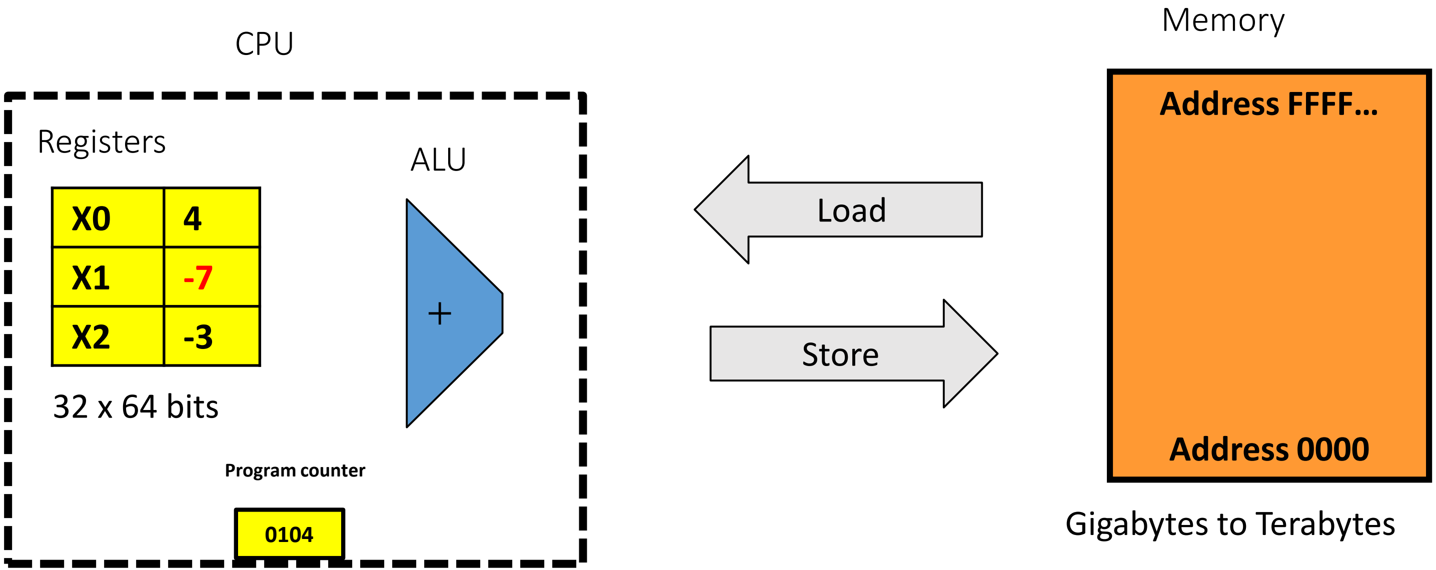
(Simplified) System Organization



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++; // x 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

Hey, Looking good!

2

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

Sign Extension

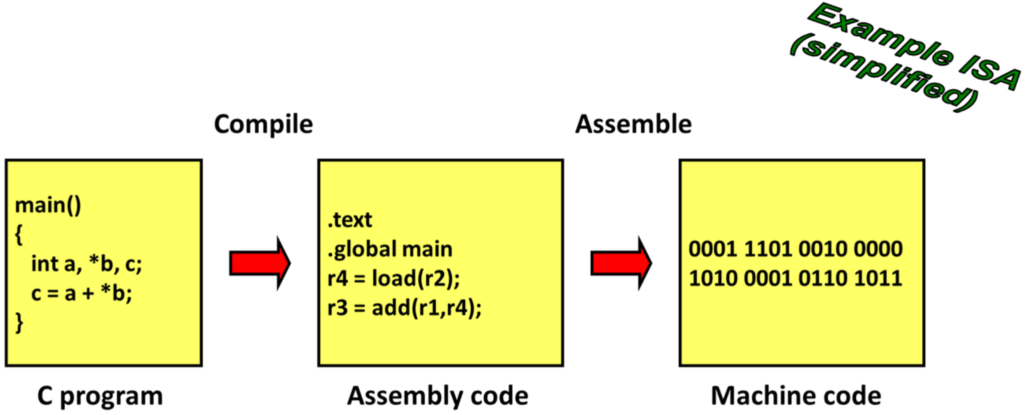
- If we want to represent a unsigned 5-bit binary number using 8 bits, we'd add 3 zeros before it
- But what about signed numbers in 2's complement?
  - If it's a positive number, its first bit should be 0 => pad it with 0s
  - If it's a negative number, its first digit should be 1

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

Extra Slides

Addressing Modes

- Direct addressing
- Register indirect
- Base + displacement
- PC-relative

Direct Addressing

Example ISA (simplified)

- Consider this code:

```
const double PI = 3.14;

double two_pi() {
  return 2*PI;
}
```

Not practical in modern ISAs… if we have 32 bit instructions and 32 bit addresses, the entire instruction is the address!

- When we load PI, it's ALWAYS the same address
  - If the ISA supports it, we can just hardcode that address in the instruction
- Like register addressing
  - Specify address as immediate constant

```
load r1,   mem[1500]   ; r1 ← contents of location 1500
jump       mem[3000]   ; jump to address 3000
```

- Useful for addressing locations that don’t change during execution
  - Branch target addresses
  - Global/static variable locations

Register indirect

Example ISA (simplified)

- Consider this code:

```
int my_arr[2] = {6666, 7777};
int* ptr = &my_arr[0];
for(int i=0; i<2; i++) {
  int x = *ptr;
  ptr++;
}
```

- Everytime we load into x, it's a different address
- But the address is always stored in another variable
- If ISA supports it, we could use a load like this
  - **`load r1, mem[r2]`**

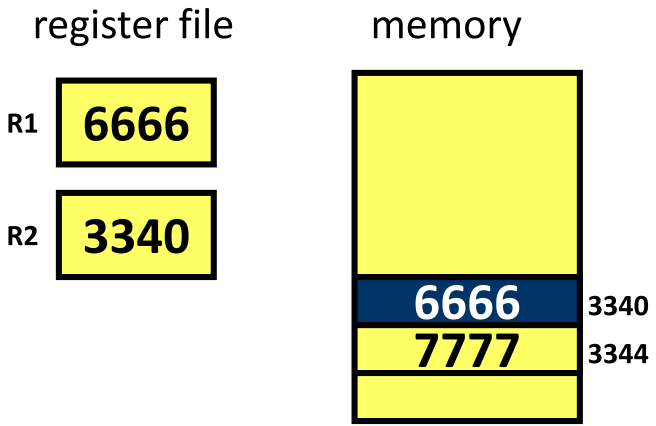
Register indirect

Example ISA (simplified)

- Consider this code:

```
int my_arr[2] = {6666, 7777};
int* ptr = &my_arr[0];
for(int i=0; i<2; i++) {
  int x = *ptr;
  ptr++;
}
```

```
load r1, mem[ r2 ]
add  r2, r2, #4
load r1, mem[ r2 ]
```



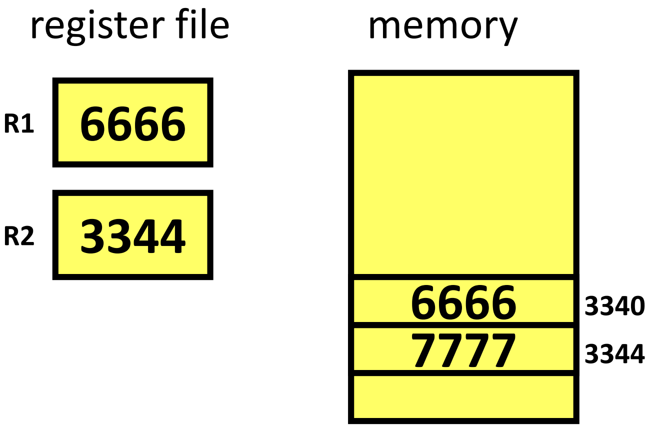
Register indirect

Example ISA (simplified)

- Consider this code:

```
int my_arr[2] = {6666, 7777};
int* ptr = &my_arr[0];
for(int i=0; i<2; i++) {
  int x = *ptr;
  ptr++;
}
```

```
load r1, mem[ r2 ]
add  r2, r2, #4
load r1, mem[ r2 ]
```



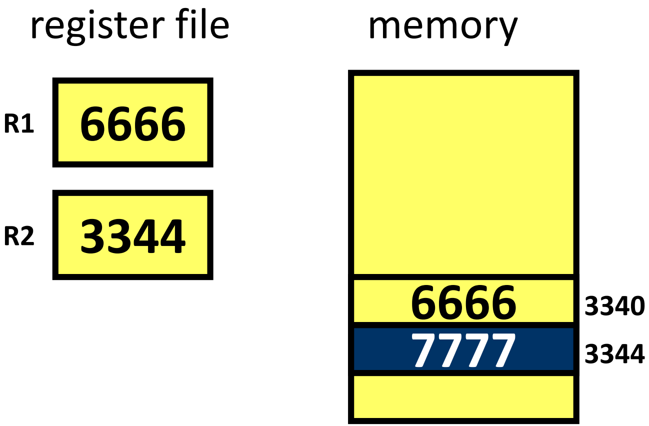
Example ISA (simplified)

Register indirect

- Consider this code:

```
int my_arr[2] = {6666, 7777};
int* ptr = &my_arr[0];
for(int i=0; i<2; i++) {
  int x = *ptr;
  ptr++;
}
```

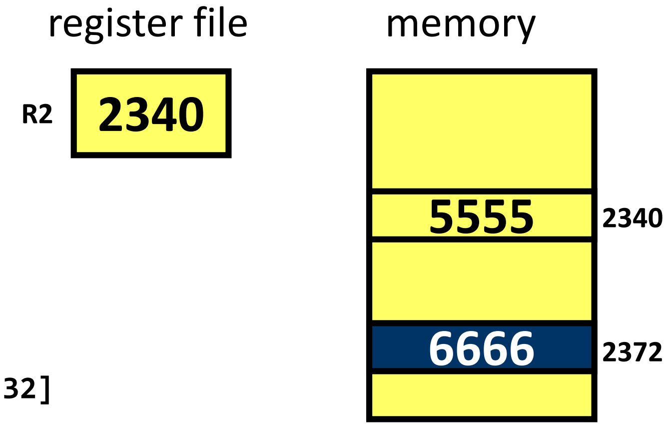
```
load r1, mem[ r2 ]
add  r2, r2, #4
load r1, mem[ r2 ]
```



This is better, but we can be more general

Base + Displacement

- Consider this code:




```
struct My_Struct {
  int tot;
  //...
  int val;
};

My_Struct a;
//...
a.tot += a.val;
```

```
load r1, mem[r2 + 32]
```

- If a register holds the starting address of "a"...
  - Then the specific values needed are a slight **offset**
- **Base + Displacement**
  - reg value + immed

Very general, most common addressing mode today

Class Problem

a. What are the contents of register/memory after executing the following instructions

```
r2 =  load mem[r3]
r3 =  load mem[r2+4]
store mem[r2+8], r3
```

Poll: What are the contents of register / memory?

register file

R1	0
R2	10
R3	108

memory

108	100
-1	104
100	108

PC-relative addressing

- **Relevant for P1.a!**

- Variant on base + displacement
- Remember PC is "Program Counter", keeps track of which line (memory address) of the program we're at
- PC register is base, longer displacement possible since PC is assumed implicitly (more bits available)
  - Used for branch instructions
    - jump [- 8] ; jump back 2 instructions (32-bit instructions)

ISA Types

Reduced Instruction Set Computing (RISC)

- Fewer, simpler instructions
- Encoding of instructions are usually the same size
- Simpler hardware
- Program is larger, more tedious to write by hand
- E.g. LC2K, RISC-V, ARM (kinda)
- More popular now

Complex Instruction Set Computing (CISC)

- More, complex instructions
- Encoding of instructions are different sizes
- More complex hardware
- Short, expressive programs, easier to write by hand
- E.g. x86
- Less popular now

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?

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;  // 0b0011_1101
```

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