

2. Instruction Set Architecture – Storage types and addressing modes

EECS 370 – Introduction to Computer Organization – Winter 2023

**EECS Department
University of Michigan in Ann Arbor, USA**

Announcements

- ❑ Homework 0 due Wednesday 1/11
- ❑ Project 1 assigned
 - Project 1.a due Thursday 1/26
 - Project 1.s and 1.m due Thursday 2/2
- ❑ All due dates on calendar on web page.

Last time

❑ Course logistics

- Website is <https://eecs370.github.io/>
- Office hours, due dates, etc.
 - Has projects, homework, schedule, reading list, etc.

❑ Topics:

- Why you need to know this stuff
 - Sorted array and branch prediction example
- Instruction set architecture (ISA) as an interface.
- Machine code and assembly briefly discussed
- Different ISAs
 - ARM, x86, RISC-V
- Did a bit with binary and hex.
 - Let's start with that

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

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

- ❑ Computers are made of parts 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

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



Collection of 8
bits is called a
byte



*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

- ❑ 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

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

What is 136 in binary? Hex?

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 \rightarrow multiplying
by 10^x

shifting x to the left in
binary \rightarrow 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
- ❑ 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 Sizes

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

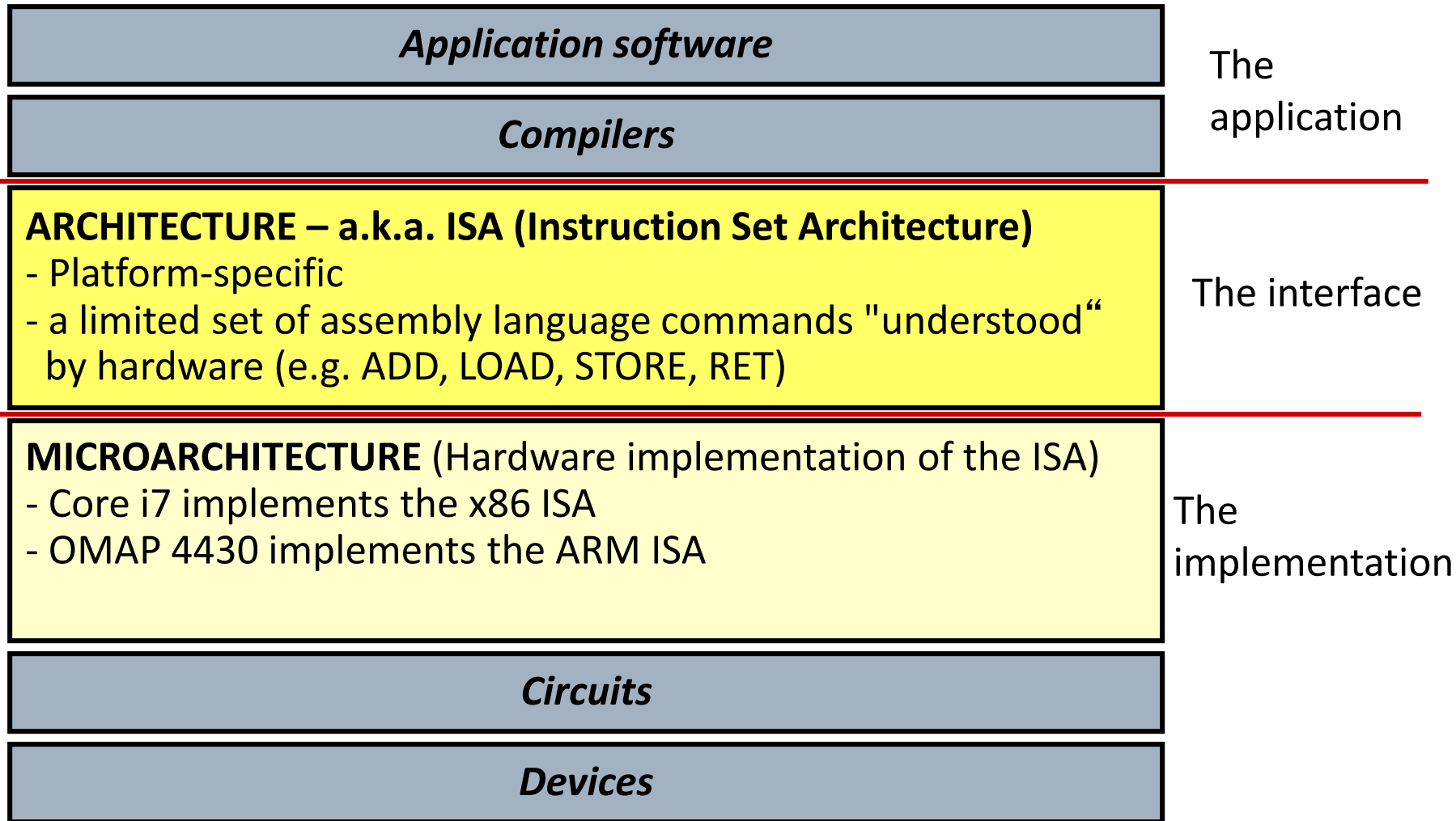
Instruction Set Architecture (ISA) Design Lectures

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

Today: Instruction Set Architectures

- ❑ What is an ISA?

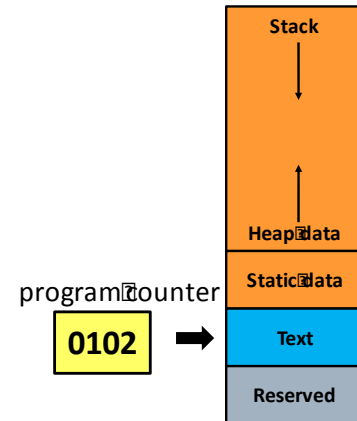
Where do ISAs come into the game ?



Basic von Neumann Architecture

Two Key Ideas

1. Data and instructions are in the same memory
 - Programs (instructions) can be viewed as data and vice versa!
2. Instructions are stored sequentially in memory
 - Accessed by the program counter (PC) —it contains the address/location of the instruction the hardware is executing
 - The PC is simply incremented to “point to” the next instruction
 - “jumps” / “branches” override fetching the sequential next insn
 - Terminology: Jumps are usually unconditional and branches are conditional on a flag being checked
 - there are conditional jumps....




How do we get the data?

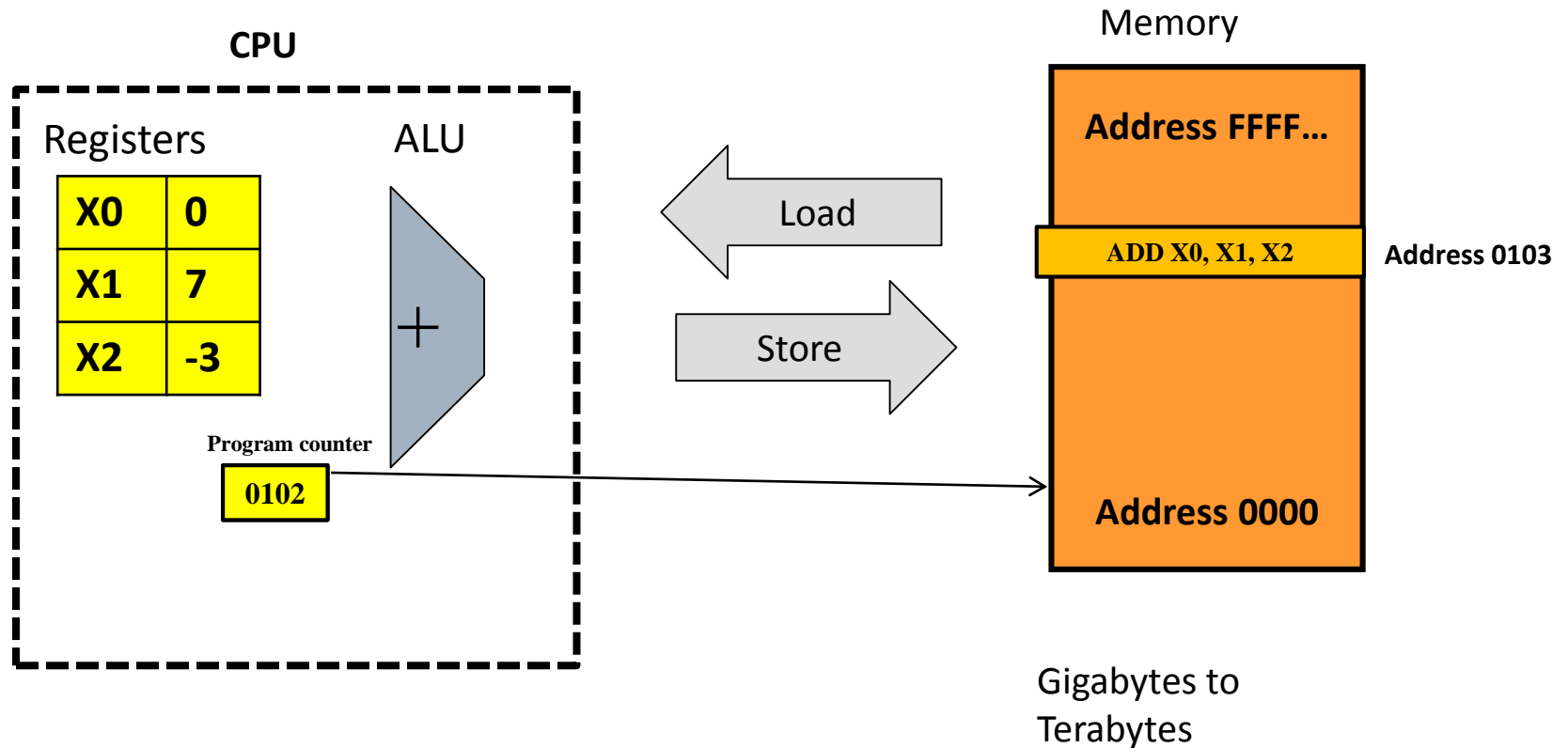
- May be in registers or memory—addressing modes

Basic von Neumann Architecture

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

- 
1. **Fetch**—get the next instruction—use the PC to find where it is in memory and place it in the instruction register
 - The PC is changed to “point” to the next instruction in the program
 - To make things simple, the von Neumann assumes the next instruction is placed sequentially in the memory
 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
 - the kind of access to data in the same memory as the instructions
 - OR the outcome is a change in the contents of the PC

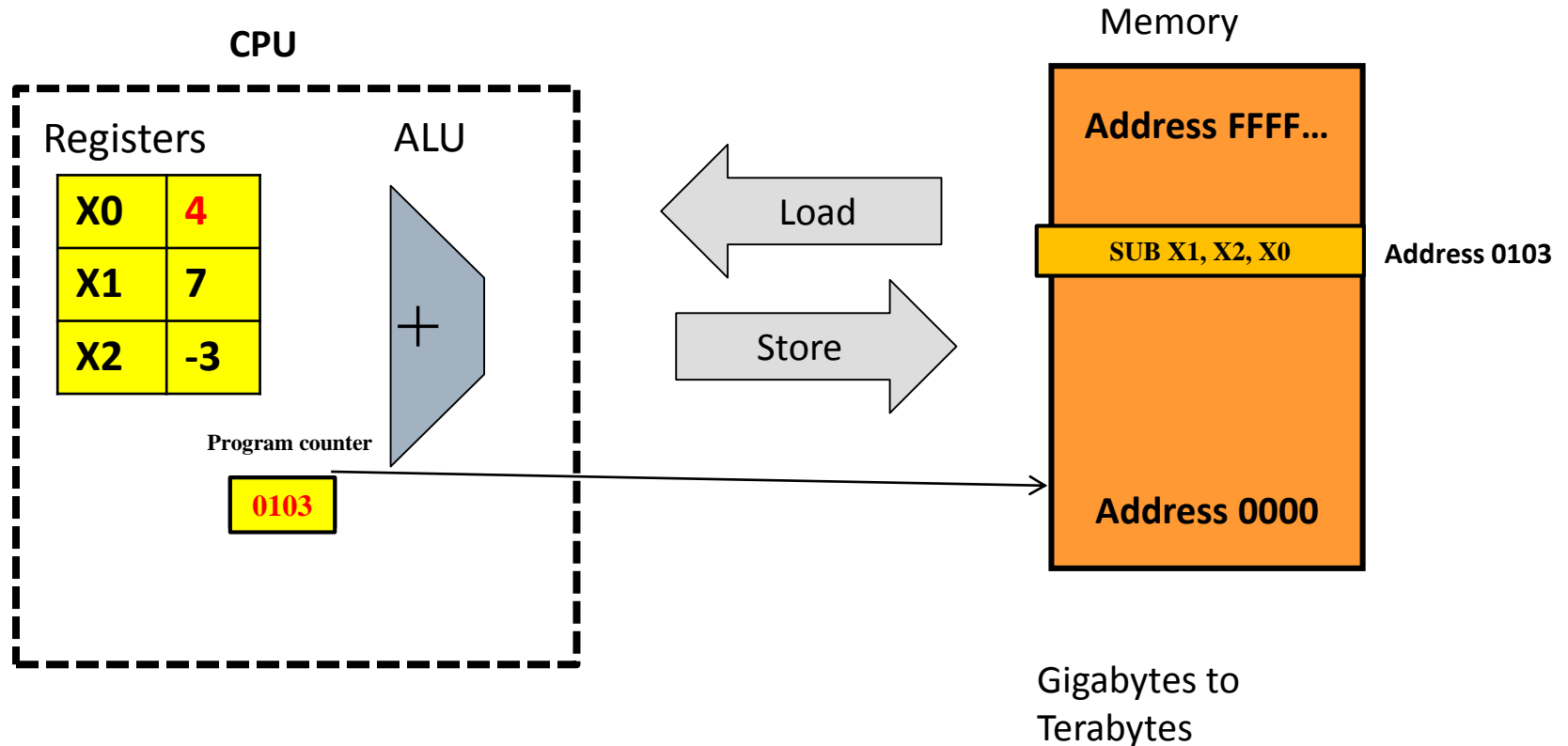
(Simplified) System Organization



(Simplified) System Organization

Let's execute this short program:

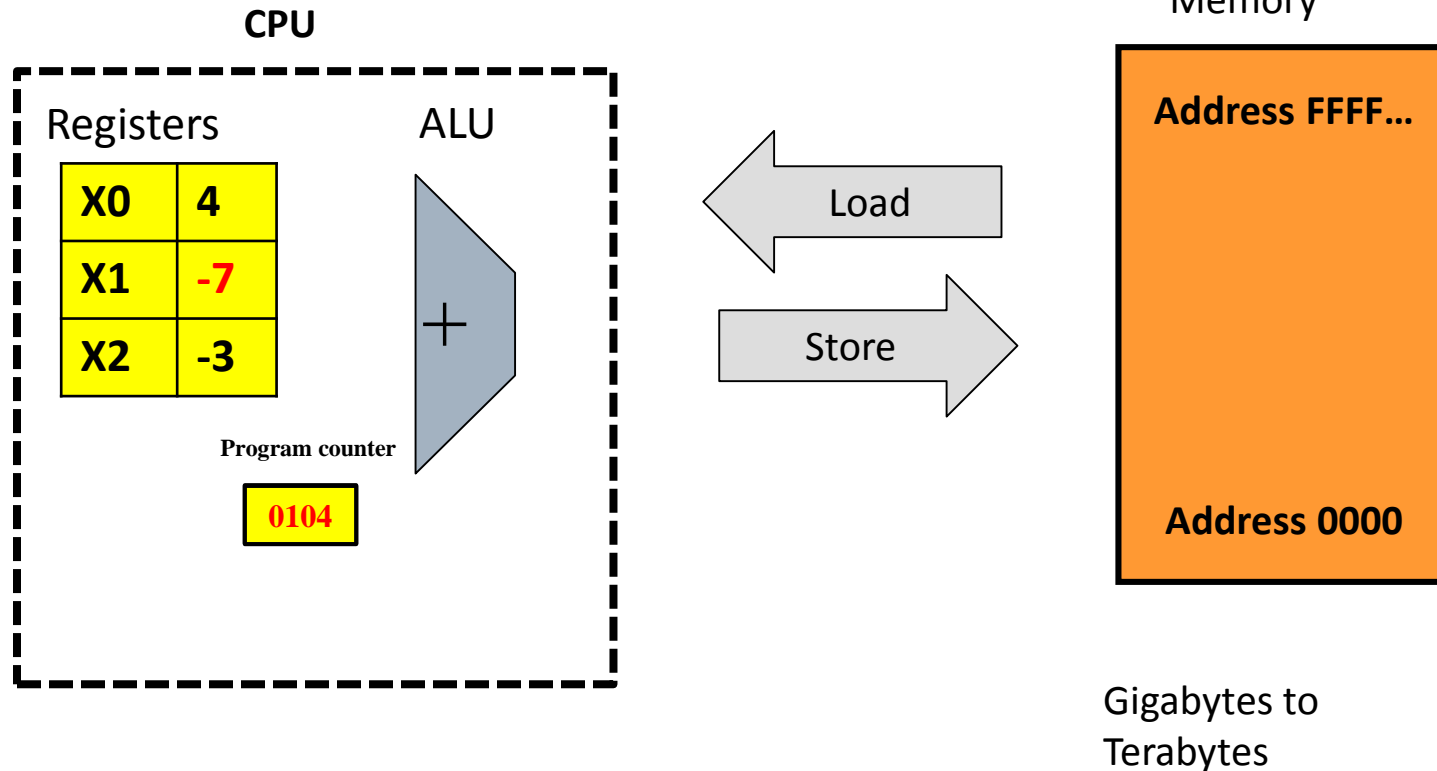
```
ADD X0, X1, X2  
SUB X1, X2, X0
```



(Simplified) System Organization

Let's execute this short program:

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



Assembly Code – ARM Example

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

What are the contents of the registers after executing the given assembly code?

Program:

<i>opcode</i>	<i>d</i>	<i>s1</i>	<i>s2imm</i>
ADD	X3	X1	X2
ADDI	X3	X3	#3
SUB	X2	X3	X1

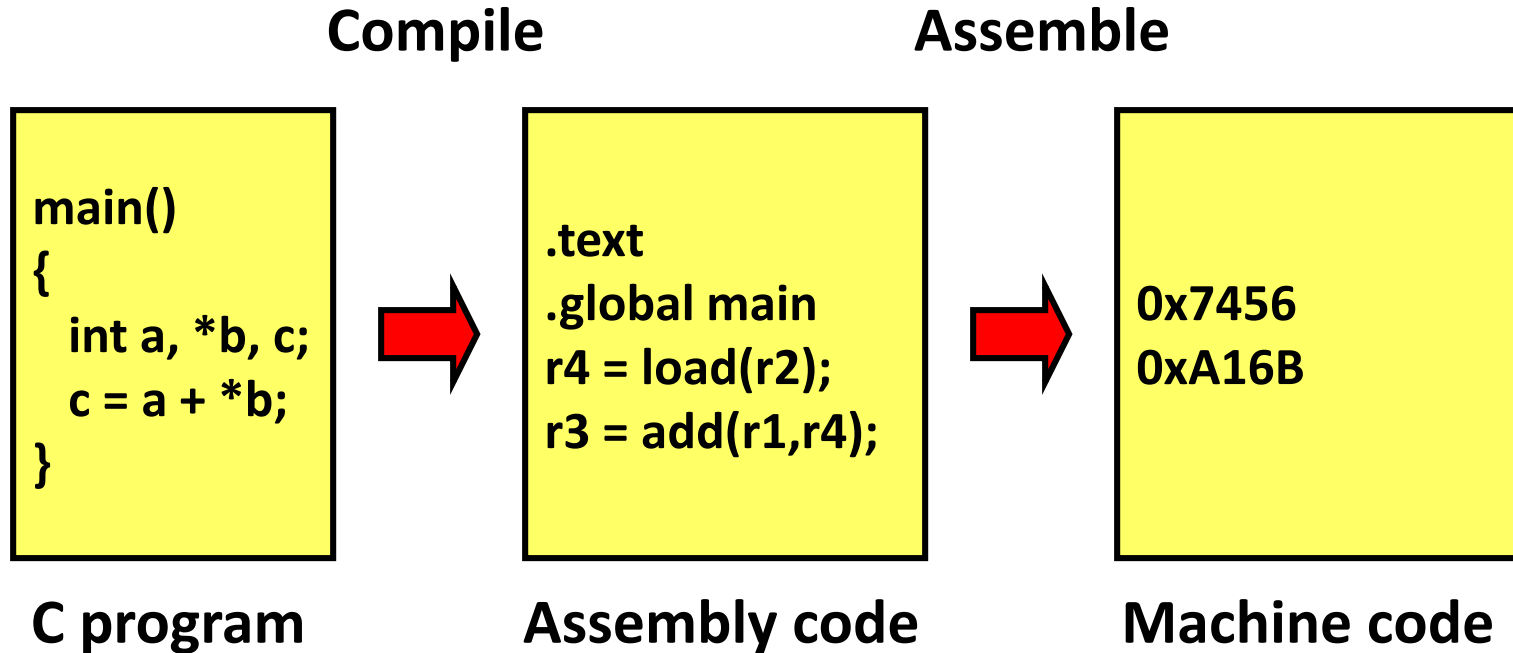
Initial
register file:

X1	25
X2	-4
X3	57

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

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

Software program to machine code



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?
- ❑ Instructions?

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.

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

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

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

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. 1.3×10^{-3})
 - Allows for larger range of numbers
 - We'll come back to this in a few weeks

Representing Instructions: Machine Code

□ Fields

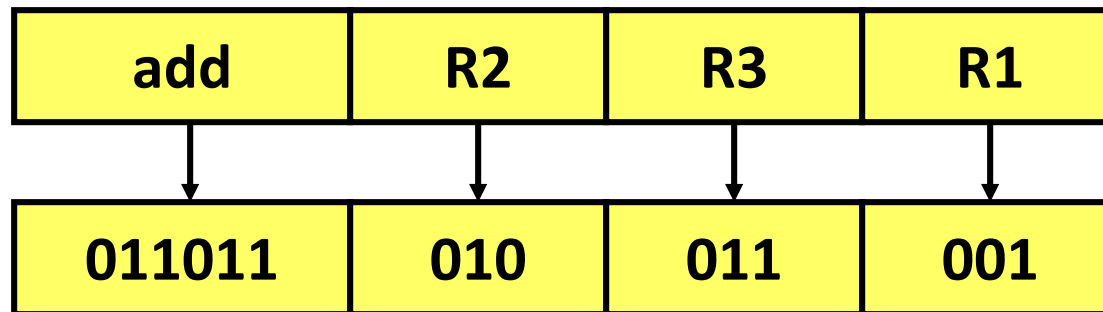
- *Opcode* – What instruction to perform
- *Source* (input) operand specifier(s)
- *Destination* (output) operand specifier(s)
 - What data to perform operation on

opcode	src1	src2	dest
add	R2	100	R1

Translation: **value in r1 = contents of r2 + constant 100**

Machine Instruction Encoding

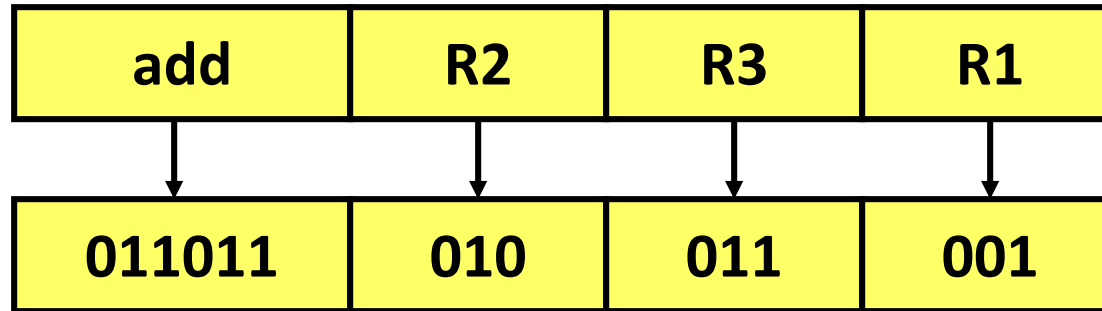
- ❑ Since the EDSAC (1949) almost all computers stored program instructions the same way they store data.
- ❑ Each instruction is encoded as a number



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

$$= 13977$$

Instruction Encoding (cont' d)

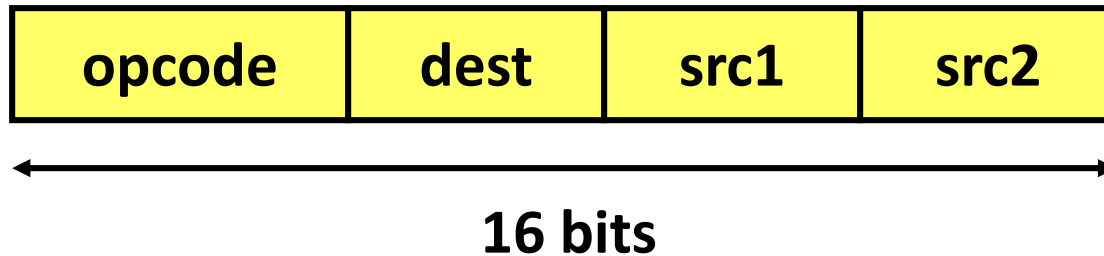


- ❑ m bits can encode 2^m different values
- ❑ n values can be encoded in $\lceil \log_2(n) \rceil$ bits
- ❑ For above
 - Can encode $2^6 = 64$ opcodes
 - Can encode $2^3 = 8$ src/destination registers
- ❑ How can we encode immediate operations, e.g., “add r2, r3, #4”?

Instruction Encoding - Example

What is the max number of registers that can be designed in a machine given:

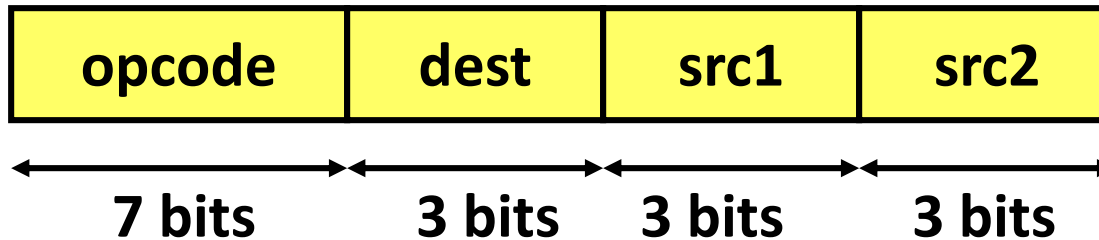
- * 16-bit instructions
- * Num. opcodes = 100
- * All instructions are (reg, reg) \rightarrow reg
(i.e., 2 source operands, 1 destination operand)



1. num opcode bits = $\lceil \log_2(100) \rceil = 7$
2. num bits for operands = $16 - 7 = 9$
3. num bits per operand = $9 / 3 = 3$
4. maximum number of registers = $2^3 = 8$

Class Problem

Given the following:



- **add** opcode is encoded as the number 53
- registers encoded with their register number

What is the encoding for **add R2, R3, R4** ?

(R2 is dest, R3 is src1, and R4 is src2)

- specify answer in binary
- specify answer in hex
- specify answer as an integer—*likely need a computer/calculator*

Instruction Set Design – design space (1/2)

❑ What instructions should be included?

- add, multiply, divide, sqrt [functions]
- branch [flow control]
- load/store [storage management]

❑ What storage locations?

- How many registers?
- How much memory?
- Any other “architected” storage?

❑ How should instructions be formatted?

- 0, 1, 2 or more operands?
- Immediate operands

Instruction Set Design – design space (2/2)

□ How to encode instructions?

- **RISC** (Reduced Instruction Set Computer):
all instructions are same length (e.g. ARM, LC2K)
- **CISC** (Complex Instruction Set Computer):
instructions can vary in size (Digital Equipment's VAX, x86)

□ What instructions can access memory?

- For ARM and LC2K, only loads and stores can access memory (called a “**load-store architecture**”)
- Intel x86 is a “**register-memory architecture**”, that is, other instructions beyond ld/st can access memory

Why study Instruction Set Design?

❑ Isn't there only one?

- No, and even if there were, it would be too messy for a first course in computer architecture

❑ How often are new architectures created?

- Embedded processors are designed all the time
- Even the Intel x86 ISA changes (MMX, MMX2, SSE, SSE2, AVX, ...)

❑ Will I ever get to (have to) design one?

- Very possible...

Storage Architecture

1. Immediate Values
 - Specifying constants in instructions
2. Registers
 - Fast and small (and useful)
3. Memory
 - Big and complex (and useful)
4. Memory-mapped I/O

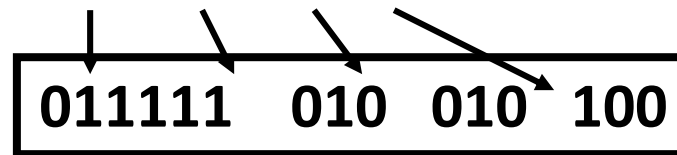
1. Immediate Values

= small constant values placed in instructions

❑ They are stored in memory only because all instructions are in memory

❑ Useful for loading small constants

- Example: `ptr++;` → `addi R2, R2, #4`



- Very useful for branch instructions

→ target address is often immediate – in the instruction

❑ Size of the immediate is usually determined by how many bits are left in the instruction format.

2. Register Storage

- ❑ First came the **accumulator**, a single register architecture
 - Example: add #5
 - You don't need to specify which register when you only have one!

- ❑ Register File
 - Small array of **memory-like** storage
 - Register access is faster than memory because register file arrays are small and can be put right next to the functional units in the processor.

- ❑ Each register in the register file has a specific size
 - e.g. 32-bit registers

- ❑ Also called “**register addressing**”

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
 - 4 general purpose registers (eax, ebx, ecx, edx) 32 bits
 - You can treat them as two 8 or one 16-bits as well (ah, al, and ax)
 - Special registers: 3 pointer registers (si, di, ip), 4 segment (cs, ds, ss, es),
2 stack (sp, bp), status register (flags)

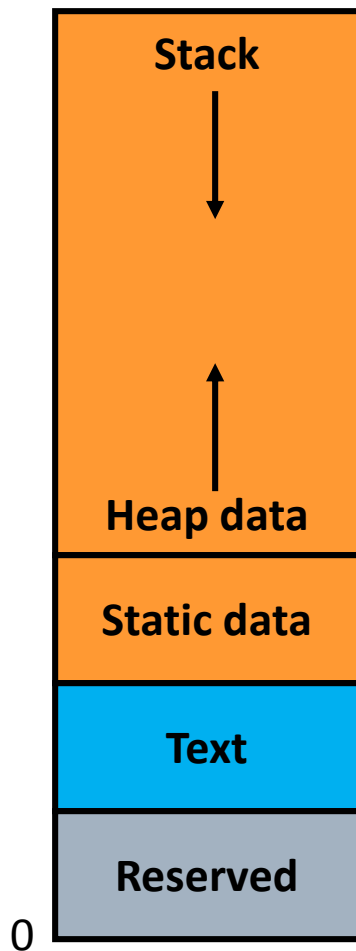
- ❑ LC2K (the architecture you will be simulating)
 - 8 registers, 32 bits each

3. Memory Storage

- ❑ Large array of storage accessed using memory addresses
 - A machine with a 32 bit address can reference memory locations 0 to $2^{32}-1$ (or 4,294,967,295).
 - A machine with a 64 bit address can reference memory locations 0 to $2^{64}-1$ (or 18,446,744,073,709,551,615—18 exa-locations)

- ❑ Lots of different ways to calculate the address.

Memory architecture: The ARM (Linux) Memory Image



Activation records: local variables, parameters, etc.

Dynamically allocated data—new or malloc()

Global data and static local data

Machine code instructions (and some constants)

Reserved for operating system

Addressing Modes

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

Direct Addressing

❑ Like register addressing

- Specify address as immediate constant
- `load r1, M[1500] ; r1 ← contents of location 1500`
- `jump M[3000] ; jump to address 3000`

Not practical in modern ISAs...

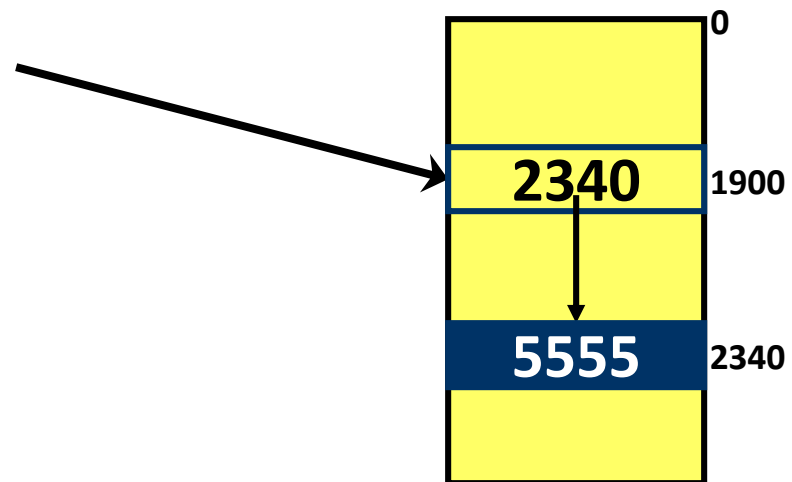
if we have 32 bit instructions
and 32 bit addresses, the entire
instruction is the address!

❑ Useful for addressing locations that don't change during execution

- Branch target addresses
- Global/static variable locations

Indirect Addressing

- ❑ Compute the reference address by
 1. Specifying an immediate address
 2. Loading the value at that immediate address
 3. Using that value as the reference address
- ❑ `load r1, M[M[1900]]`



Register indirect

- Specify which register has the reference address

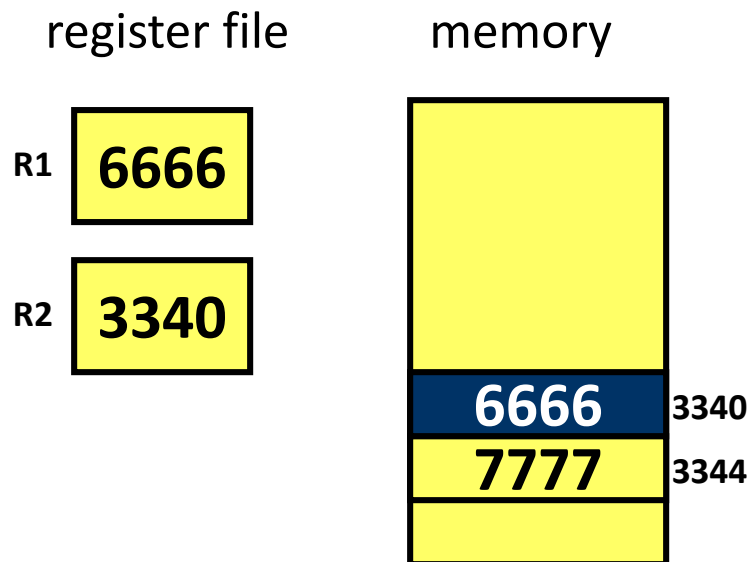
- Very useful for pointers



load r1, M[r2]

add r2, r2, #4

load r1, M[r2]



Register indirect

- Specify which register has the reference address

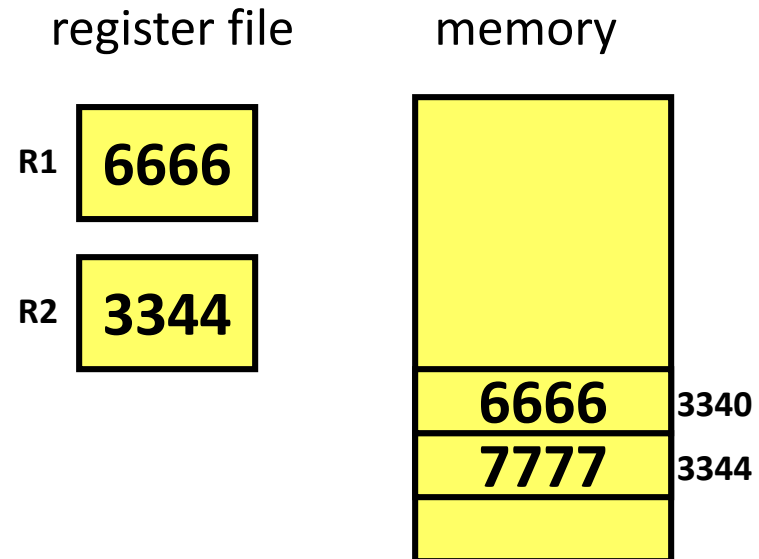
- Very useful for pointers

load r1, M[r2]



add r2, r2, #4

load r1, M[r2]



Register indirect

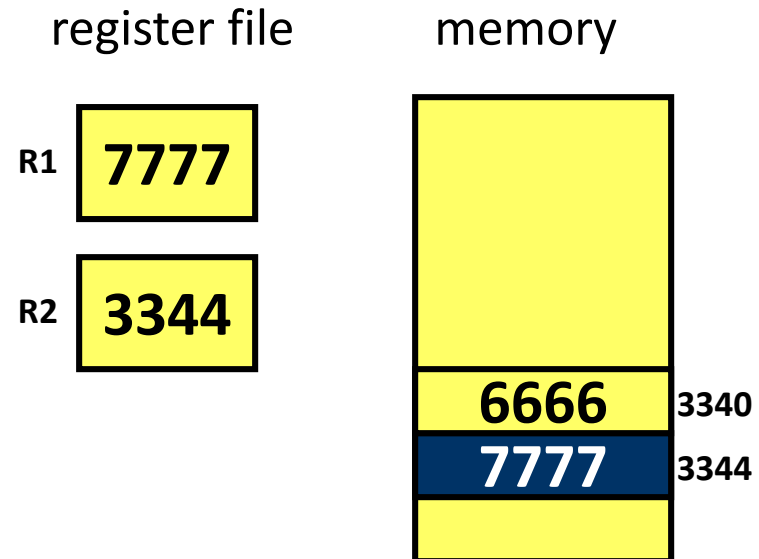
□ Specify which register has the reference address

- Very useful for pointers

load r1, M[r2]

add r2, r2, #4

➡ *load r1, M[r2]*

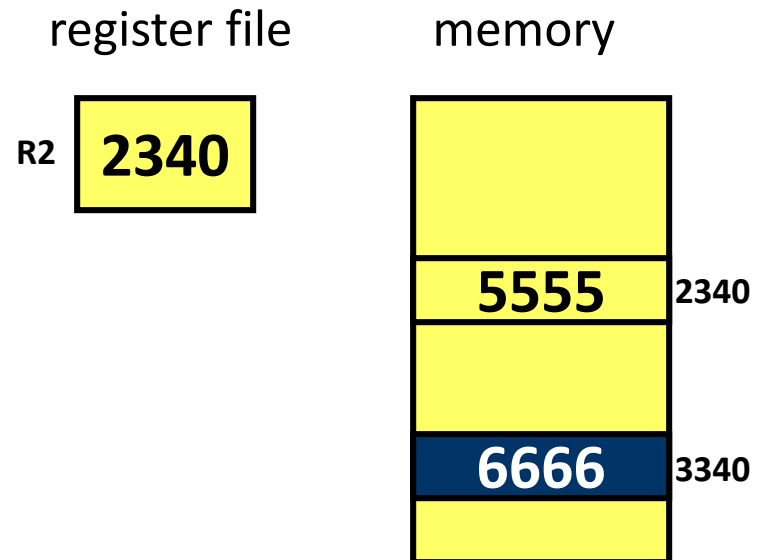


Base + Displacement

- ❑ Most common addressing mode today
- ❑ Compute address as: reg value + immed
- ❑ load r1, M [r2 + 1000]
- ❑ Good for accessing class objects/structures. Why?

a.tot += a.val;

Very general, most
common addressing
mode today!



Class Problem

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

`r2 = load M[r3]`

`r3 = load M[r2+4]`

`store M[r2+8], r3`

register file		memory	
R1	0	108	100
R2	10	-1	104
R3	108	100	108

- b. How can base + displacement be made to simulate indirect addressing??
(Hint: requires 2 instructions)

PC-relative addressing

- ❑ Variant on base + displacement
- ❑ PC register is base, longer displacement possible since PC is assumed implicitly
 - Used for branch instructions
 - `jump [- 8] ; jump back 2 instructions (32-bit instructions)`
- ❑ Humans use labels and leave it to the assembler to determine the immediate value. Why?

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

Programming Assignment #1

- ❑ Write an assembler to convert input (assembly language program) to output (machine code version of program)—due Thursday 1/26
 - “1a”

- ❑ Write a behavioral simulator to run the machine code version of the program (printing the contents of the registers and memory after each instruction executes)—due Thursday 2/2
 - “1s”

- ❑ Write an efficient LC2K assembly language program to multiply two numbers—due Thursday 2/2
 - “1m”

Programming Assignment #1

□ Where to start...

- Write some test cases to check your code
 - Program 1: halt
 - Program 2: noop
 halt
 - Program 3: add 1 1 1
 halt
 - Program 4: nand 1 1 1
 halt