Memory Concepts

- Memory is divided into "addressable" units, each with an address (like an array with indices)
- Addressable units are usually larger than a bit, typically 8, 16, 32, or 64 bits
- Each address has variable "contents"
- Memory contents might be:
  - Integers in 2's complement
  - Floats in IEEE format
  - Strings in ASCII or Unicode
  - Data structure de jour
  - ADDRESSES
  - Nothing distinguishes the difference

Address	Contents
0	0x0000002a
1	0x40490fd8
2	0x2065654C
3	0x74726148
4	0x20647542
5	0x6976654c
6	0x0020656c
7	0x00000002
8	0xe3a00000
9	0xe3a0100a
10	0xe0800001
11	0xe2511001
12	0x1afffffc
13	0xeafffffe
14	0x00004020
15	0x20090001

One More Thing

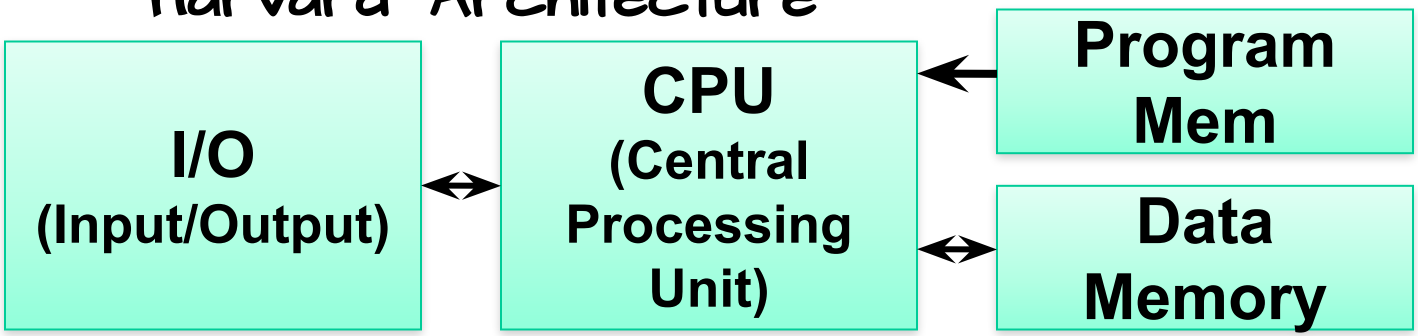
- INSTRUCTIONS for the CPU are stored in memory along with data
- CPU fetches instructions, decodes them and then performs their implied operation
- Mechanism inside the CPU directs which instruction to get next.
- They appear in memory as a string of bits that are typically uniform in size
- Their encoding as "bits" is called "machine language." ex: 0c3c1d7fff
- We assign "mnemonics" to particular bit patterns to indicate meanings.
- These mnemonics are called **Assembly language**. ex: mov r1, #10

Address	Contents
0	42
1	3.141592
2	"Lee "
3	"Hart"
4	"Bud "
5	"Levi"
6	"le "
7	2
8	mov r0, #0
9	mov r1, #10
10	add r0, r0, r1
11	subs r1, r1, #1
12	bne .-2
13	b .
14	0x00004020
15	0x20090001

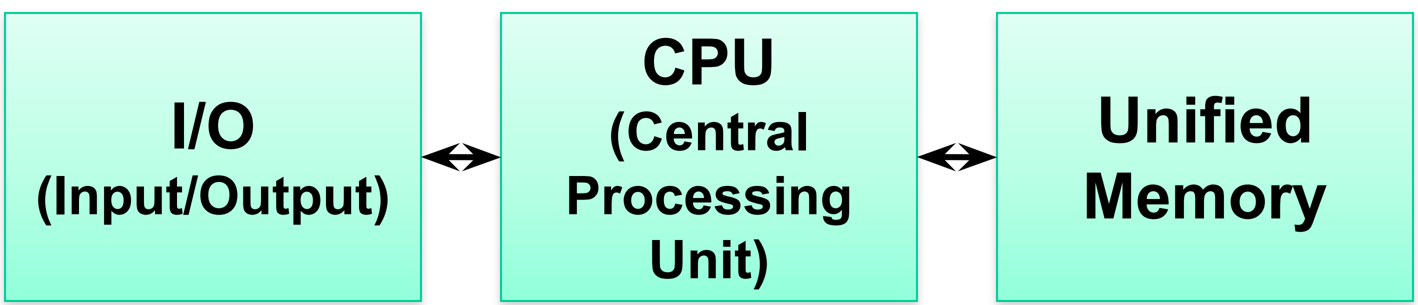
A Bit of History

There is a commonly recurring debate over whether "data" and "instructions" should be mixed. Leads to two common flavors of computer architectures

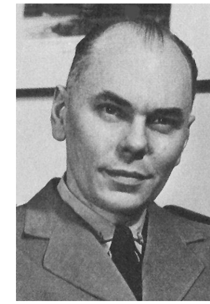
"Harvard" Architecture



"Von Neumann" Architecture



Harvard Architecture



Howard Aiken: Architect of the Harvard Mark 1

Instructions and data do not/should not interact. They can have different "word sizes" and exist in different "address spaces"

- **Advantages**:
  - No self-modifying code (a common hacker trick)
  - Optimize word-lengths of instructions for control and data for applications
  - Higher Throughput (i.e. you can fetch data and instructions from their memories simultaneously)
- **Disadvantages**:
  - The H/W designer decides the trade-off between how big of a program and how large are data
  - Hard to write "Native" programs that generate new programs (i.e. assemblers, compilers, etc.)
  - Hard to write "Operating Systems" which are programs that at various points treat other programs as data (i.e. loading them from disk into memory, swapping out processes that are idle)

Von Neumann Architecture

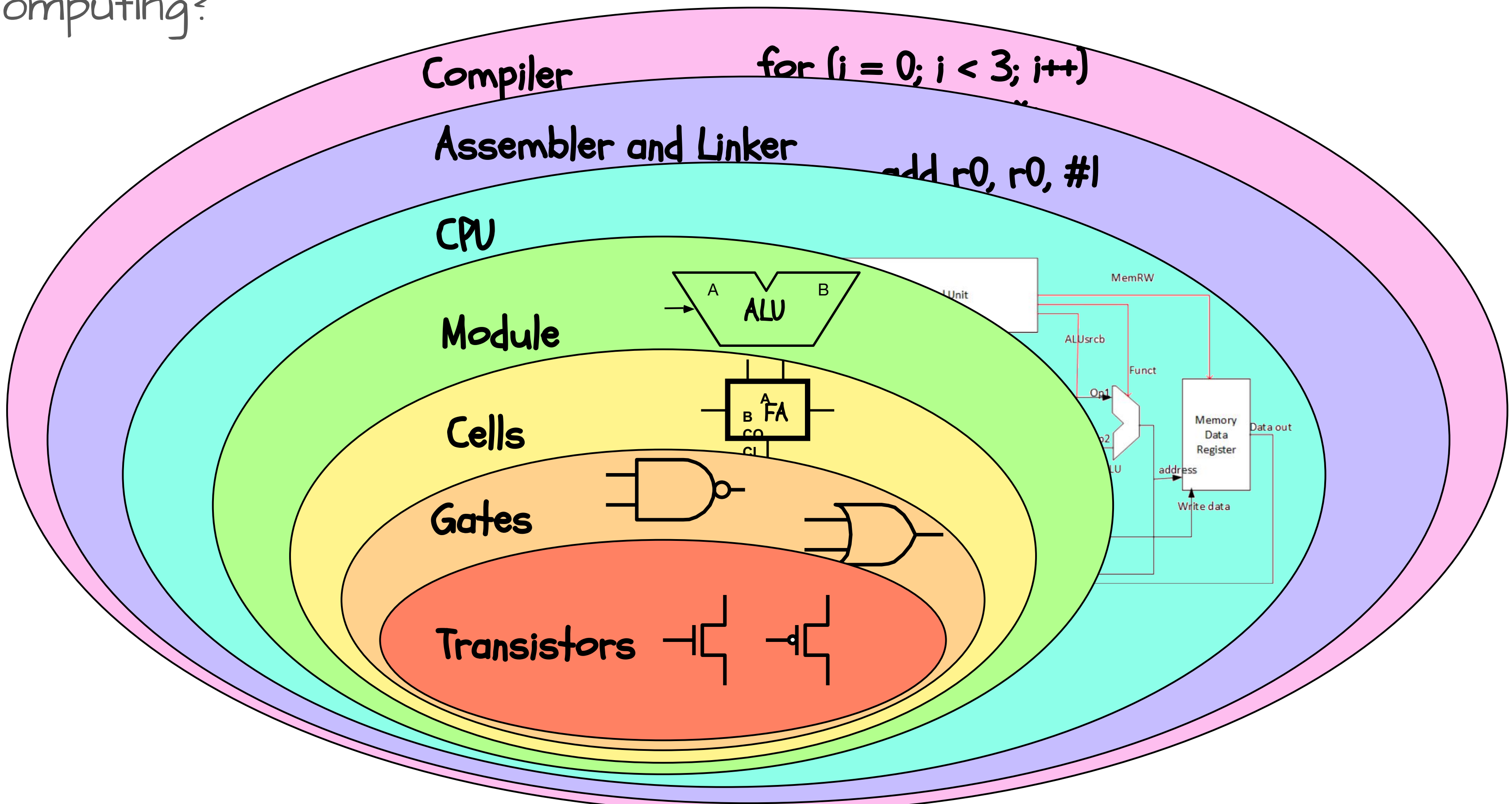
John Von Neumann: Proponent of unified memory architecture

Instructions are just a type of data that share a common "word size" and "address space" with other types.

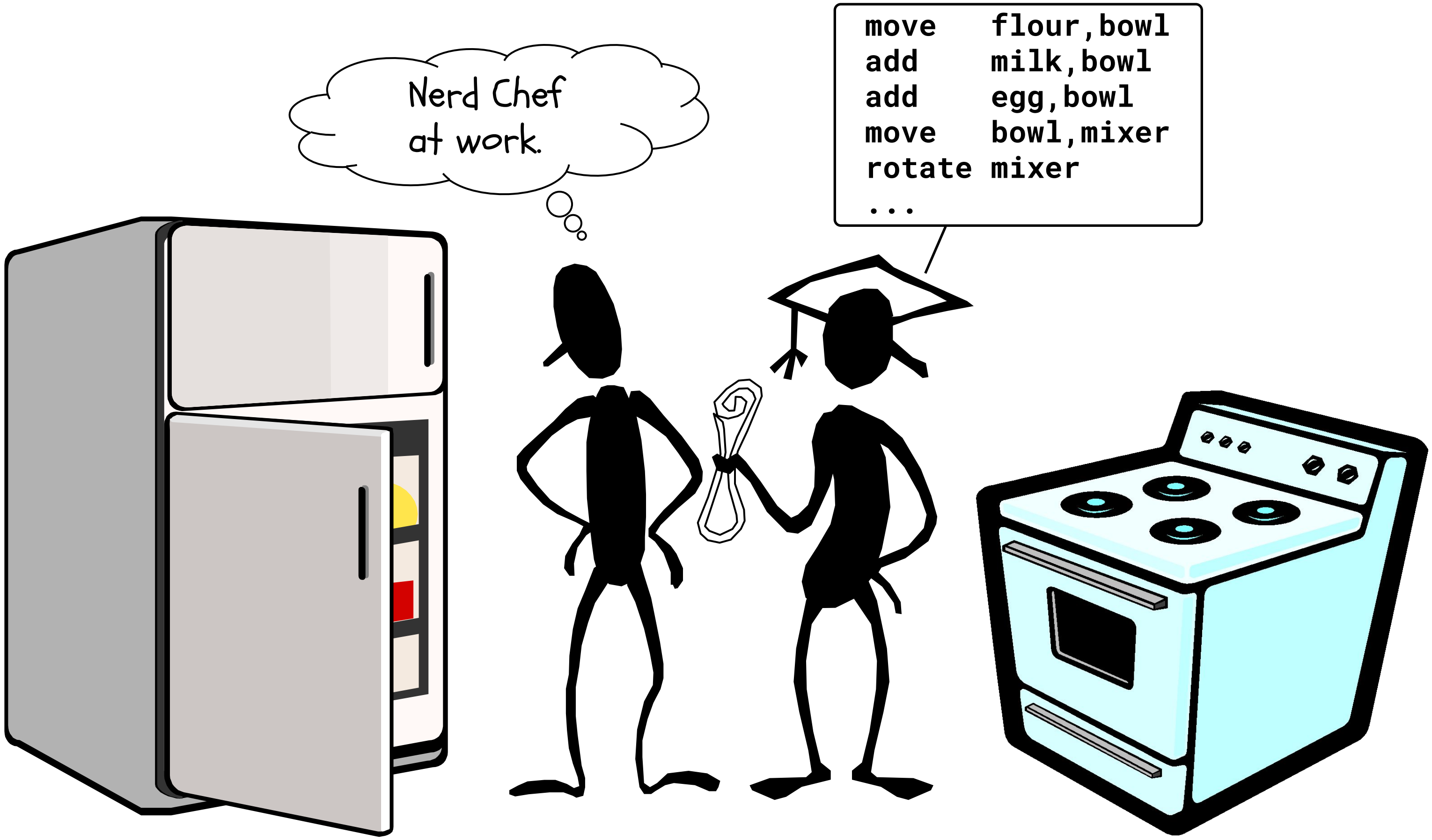
- Most common model used today, and what we assume in 411
- **Advantages**:
  - S/W designer decides how to allocate memory between data and programs
  - Can write programs to create new programs (assemblers and compilers)
  - Programs and subroutines can be loaded, relocated, and modified by other programs (dangerous, but powerful)
- **Disadvantages**:
  - Word size must suit both common data types and instructions
  - Slightly lower performance due to memory bottleneck (mediated in modern computers by the use of separate program and data caches)
  - We need to be very careful when treading on memory. Folks have taken advantage of the program-data unification to introduce viruses.

Computer Systems

At what level of abstraction can we understand a computer? Computing?



Concocting an Instruction Set



Your first problem set is posted

Instructions are Simple

- Computers interpret "programs" by translating them from the high-level language where into "low-level" simple instructions that it understands
- High-Level Languages
  - Compilers
  - Interpreters
- Assembly Language

```
int x, y;
y = (x-3)*(y+123456)
```

→

```
x:    .word 0
y:    .word 0
c:    .word 123456

      ...
      LDR     R0, [R10, #0]    ; get x
      SUB     R0, R0, #3
      LDR     R1, [R10, #4]    ; get y
      LDR     R2, [R10, #8]    ; get c
      ADD     R1, R1, R2
      MUL     R0, R0, R1
      STR     R0, [R10, #4]    ; save y
```

Instructions are Binary

- Computers interpret "assembly programs" by translating them from their mnemonic simple instructions into strings of bits
- Assembly Language
- Machine Language
  - Note the one-to-one correspondence between lines of assembly code and Lines of machine code

```
x:   .word 0
y:   .word 0
c:   .word 123456
     ...
     LDR     R0, [R10, #0]    ; get x
     SUB     R0, R0, #3
     LDR     R1, [R10, #4]    ; get y
     LDR     R2, [R10, #8]    ; get c
     ADD     R1, R1, R2
     MUL     R0, R0, R1
     STR     R0, [R10, #4]    ; save y
```

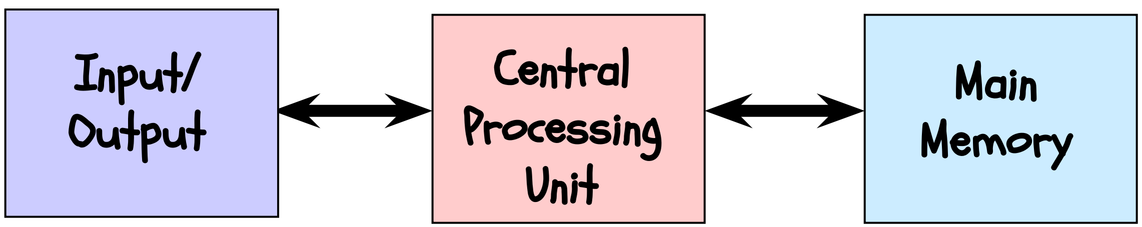
→

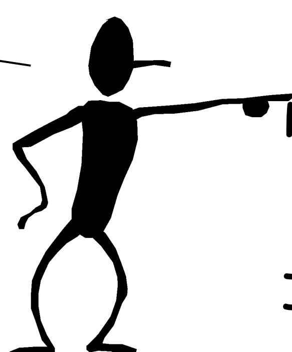
```
0x00000000
0x00000000
0x0001E240

...

0xE59A0000
0xE2400003
0xE59A1004
0xE59A2008
0xE0811002
0xE0000190
0xE58A0004
```

A General-Purpose Computer

The von Neumann Model

Many architectural approaches to the general purpose computer have been explored. The one upon which nearly all modern computers is based was proposed by John von Neumann in the late 1940s. Its major components are:





Central Processing Unit (CPU): A device which fetches, interprets, and executes a specified set of operations called Instructions.

Memory: storage of N *words* of W bits each, where W is a fixed architectural parameter, and N can be expanded to meet needs.

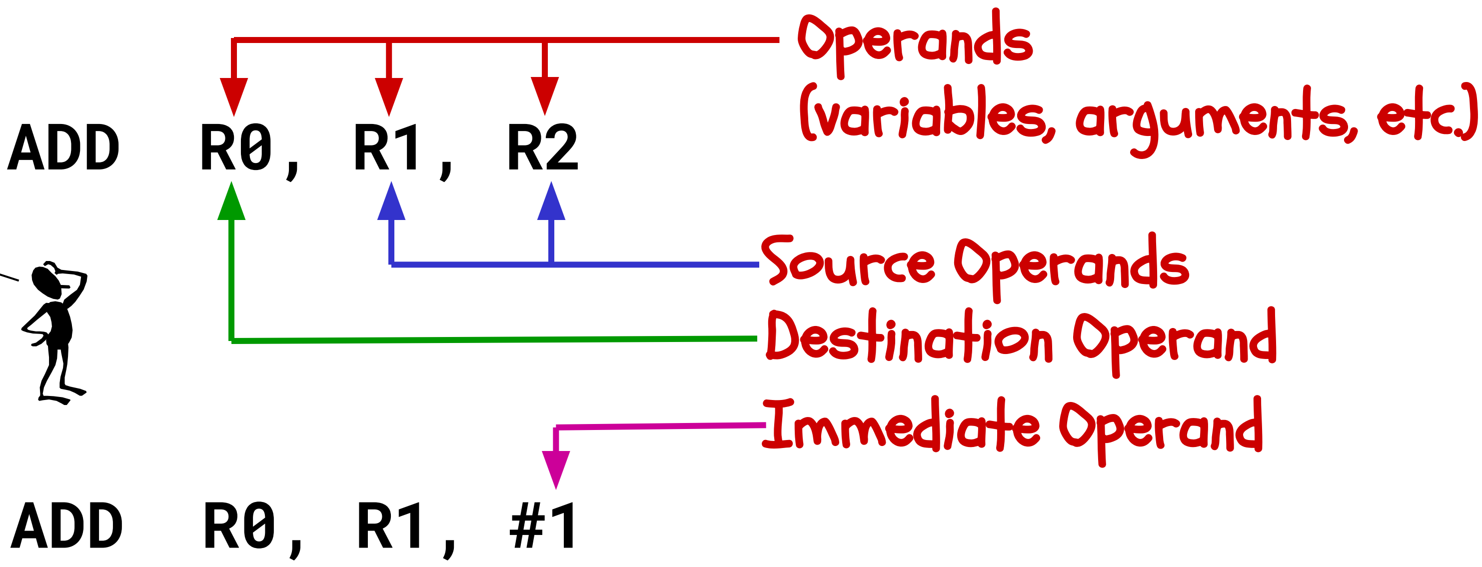
I/O: Devices for communicating with the outside world.

Anatomy of an Instruction

- Computers execute a set of primitive operations called **instructions**
- Instructions specify an **operation** and its **operands** (arguments of the operation)
- Types of operands: **destination**, **source**, and **immediate**

Why do all of the variables start with "R"?

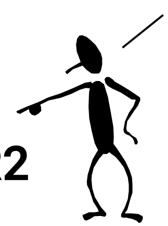
CPU's have a small number (16-32) of registers that are used for a small number of holding variables



Meaning of an Instruction

- Operations are abbreviated into **opcodes** (1-4 letters)
- Instructions are specified with a very regular syntax
    - Opcodes are followed by arguments
    - Usually the destination is next, then one or more source arguments (This is not strictly the case, but it is generally true)
- Why this order?
  Analogy to high-level language like Java or C

```
add  R0, R1, R2
```



The instruction syntax provides operands in the same order as you would expect in a statement from a high level language.

```
int r0, r1, r2;
r0 = r1 + r2;
```

Instead of:

```
r1 + r2 = r0;
```

A Series of Instructions

- Generally...
  - Instructions are retrieved sequentially from memory
  - An instruction executes to completion before the next instruction is started
  - But, there are exceptions to these rules

Instructions
`ADD R0, R1, R1`
`ADD R0, R0, R0`
`ADD R0, R0, R0`
`SUB R1, R0, R1`

What does this program do?

Variables
R0: ~~0~~ ~~12~~ ~~24~~ 48
R1: ~~6~~ 42
R2: 8
R3: 10

Program Analysis

- Repeat the process treating the variables as unknowns or "formal variables"
- Knowing what the program does allows us to write down its specification, and give it a meaningful name
- The instruction sequence then becomes a general-purpose tool

Instructions

ADD R0, R1, R1
ADD R0, R0, R0
ADD R0, R0, R0
SUB R1, R0, R1

Variables

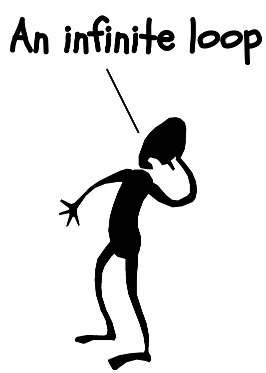
R0: ~~x~~ ~~2x~~ ~~4x~~ 8x
R1: ~~x~~ 7x
R2: y
R3: z

Looping the Flow

- Repeat the process treating the variables as unknowns or "formal variables"
- Knowing what the program does allows us to write down its specification, and give it a meaningful name
- The instruction sequence then becomes a general-purpose tool

Instructions

```
times7: ADD R0, R1, R1
        ADD R0, R0, R0
        ADD R0, R0, R0
        SUB R1, R0, R1
        B   times7
```

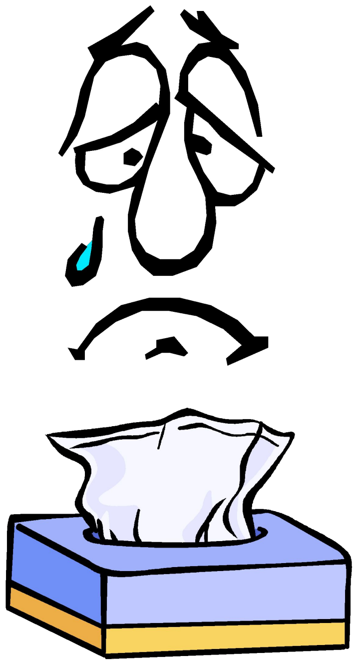


Variables

```
R0:w 8x 56x 392x
R1:x 7x 49x 343x
R2:y
R3:z
```

Open Issues in our Simple Model

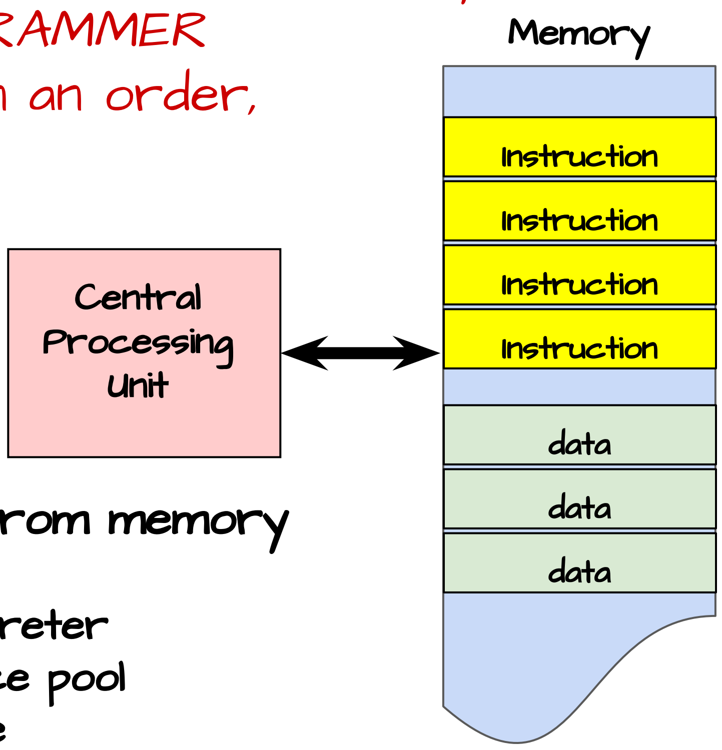
- WHERE in memory are INSTRUCTIONS stored?
- HOW are instructions represented?
- WHERE are VARIABLES stored?
- What are LABELs? How do they relate to where instructions are stored?
- How about more complicated data types?
  - Arrays?
  - Data Structures?
  - Objects?
- Where does a program start executing?
- When does it stop?



The Stored-Program Computer

- The von Neumann architecture addresses these issues as follows:
- Instructions and Data are stored in a common memory
- Sequential semantics: To the *PROGRAMMER* all instructions appear to execute in an order, or sequentially

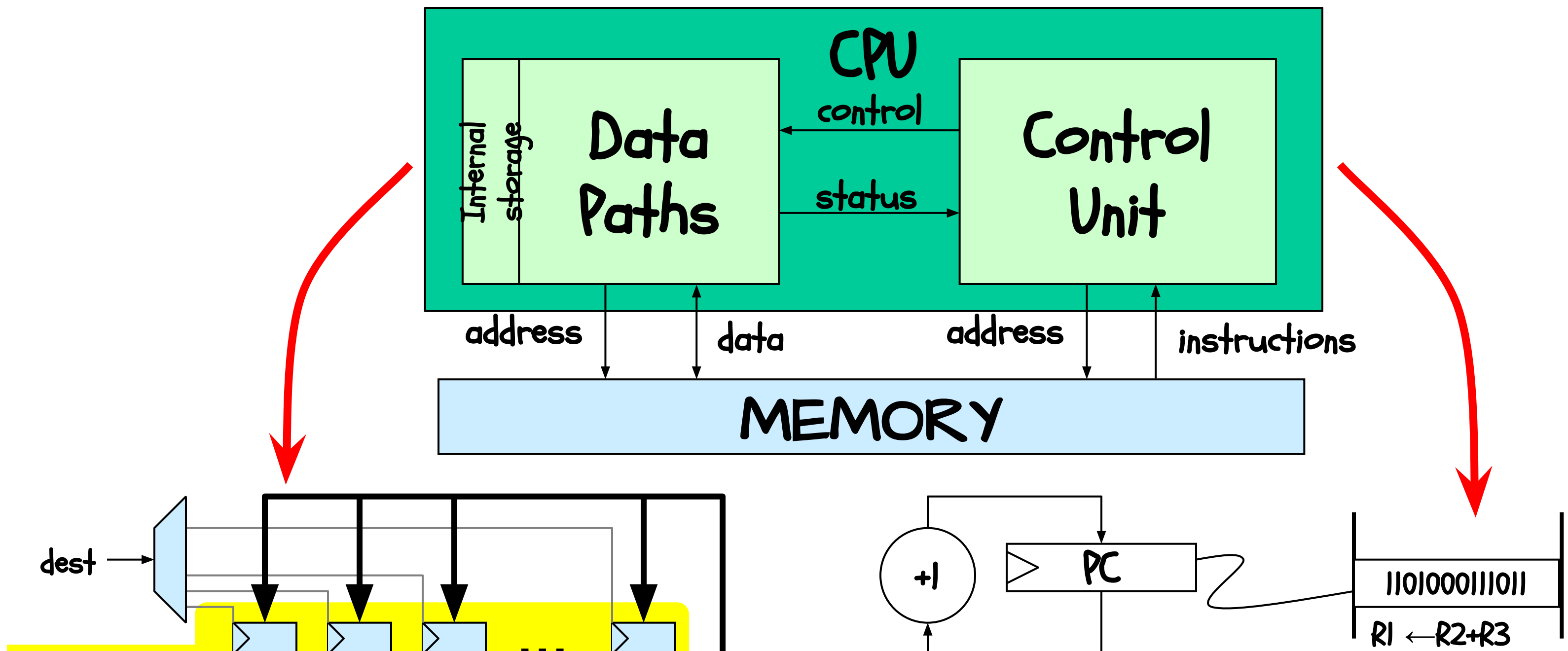
Key idea: Memory holds not only data, but *coded instructions* that make up a *program*.

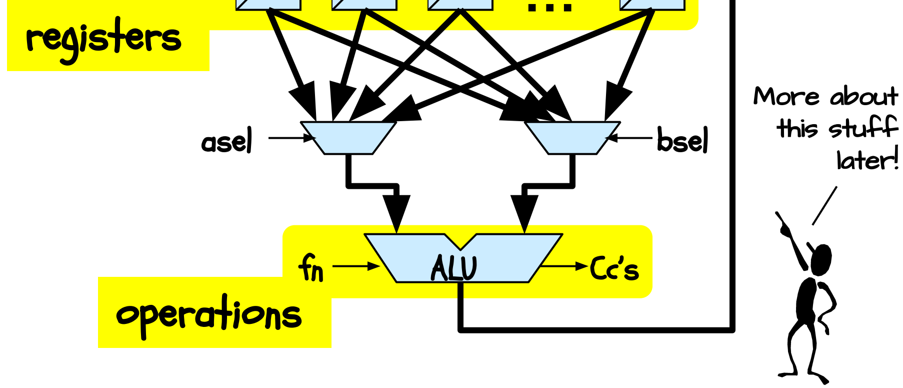


CPU fetches and executes instructions from memory

- The CPU is a H/W interpreter
- Program IS simply DATA for this interpreter
- Main memory: Single expandable resource pool
  - constrains both data and program size
  - don't need to make separate decisions of how large of a program or data memory to buy

Anatomy of a von Neumann Computer





- INSTRUCTIONS coded as binary data
- PROGRAM COUNTER or PC:
  Address of next instruction to execute
- logic to translate instructions into control signals for data path

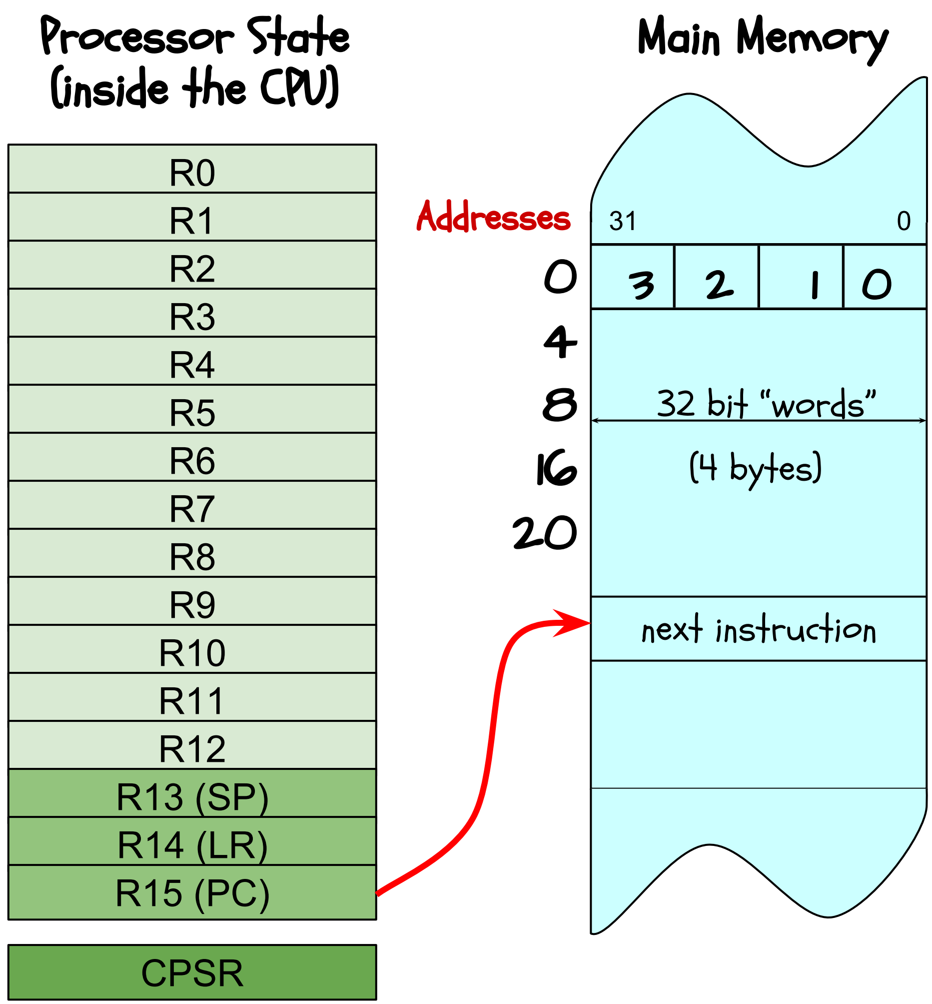
Instruction Set Architecture (ISA)

Encoding of instructions raises some interesting choices…

- Tradeoffs: performance, compactness, programmability
- Uniformity. Should different instructions
  - Be the same size (number of bits)?
  - Take the same amount of time to execute?
  - Trend: Uniformity. Affords simplicity, speed, pipelining.
- Complexity. How many different instructions? What level operations?
  - Level of support for particular software operations: array indexing, procedure calls, "polynomial evaluate", etc
  - "Reduced Instruction Set Computer" (RISC) philosophy: simple instructions, optimized for speed
- Mix of Engineering & Art…

ARM7 Programming Model

A representative RISC machine



In Comp 411 we'll use a subset of the ARM7 core Instruction set as an example ISA.

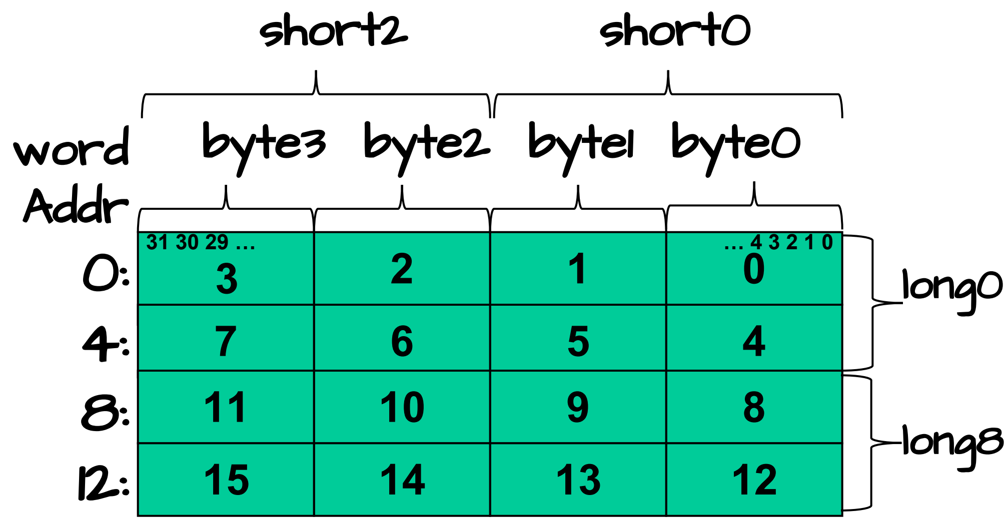
Fetch/Execute loop:

- fetch Mem[PC]
- PC = PC + 4[†]
- execute fetched instruction (may change PC!)
- repeat!

ARM7 uses byte memory addresses. However, each instruction is 32-bits wide, and *must* be aligned on a multiple of 4 (word) address. Each word contains four 8-bit bytes. Addresses of consecutive instructions (words) differ by 4.

ARM Memory Nits

- Memory locations are addressable in different sized chunks
  - 8-bit chunks (bytes)
  - 16-bit chunks (shorts)
  - 32-bit chunks (words)
  - 64-bit chunks (longs/doubles)
- We also frequently need access to individual bits! (Instructions help with this)
- Every BYTE has a unique address (ARM is a byte-addressable machine)
- Most instructions are one word



ARM Register Nits

- There are 16 named registers [R0, R1, ... R15]
- The operands of most instructions are registers
- This means to operate on a variables in memory you must:
    - Load the value/values from memory into a register
    - Perform the instruction
    - Store the result back into memory
- Going to and from memory can be expensive
  (4x to 20x slower than operating on a register)
- Net effect: Keep variables in registers as much as possible!
- 3 registers are dedicated to specific tasks (SP, LR, PC)
  13 are available for general use

Basic ARM Instructions

- Instructions include various "fields" that encode combinations of OPCODES and arguments
- special fields enable extended functions
- several 4-bit OPERAND fields, for specifying the sources and destination of the operation, usually one of the 16 registers
- Embedded constants ("immediate" values) of various sizes,

The basic data-processing instruction formats:

	4	3	4	1	4	4	8	4
R type:	1110	00**0**	Opcode	0	Rn	Rd	00000000	Rm

	4	3	4	1	4	4	4	8
I type:	1110	00**1**	Opcode	0	Rn	Rd	Shift	Imm

R-type Data Processing

Instructions that process three-register arguments:

	4	3	4	1	4	4	8	4
R type:	1110	00**0**	Opcode	0	Rn	Rd	00000000	Rm

0000 - AND
0001 - EOR
0010 - SUB
0011 - RSB
0100 - ADD
0101 - ADC
0110 - SBC
0111 - RSC
1000 - TST
1001 - TEQ
1010 - CMP
1011 - CMN
1100 - ORR
1101 - MOV
1110 - BIC
1111 - MVN

Simple R-type instructions follow the following template:

`OP Rd, Rn, Rm`

Later on we'll introduce more complex variants of these 'simple' R-type instructions.

`ADD R0, R1, R3`

Is encoded as:

1110 0000 **1000 0001 0000** 0000 0000 **0011**

`0xE0810003`

I-type Data Processing

Instructions that process one register and a constant:

	4	3	4	1	4	4	4	8
R type:	1110	001	Opcode	0	Rn	Rd	Rotate	Imm8

- 0000 - AND
- 0001 - EOR
- 0010 - SUB
- 0011 - RSB
- 0100 - ADD
- 0101 - ADC
- 0110 - SBC
- 0111 - RSC
- 1000 - TST
- 1001 - TEQ
- 1010 - CMP
- 1011 - CMN
- 1100 - ORR
- 1101 - MOV
- 1110 - BIC
- 1111 - MVN

Simple I-type instructions follow the following template:

```
OP    Rd, Rn, #constant
```

In the I-type instructions the second register operand is replaced by a constant that is encoded in the instruction

```
RSB    R7, R10, #49
```

Is encoded as:

1110 0010 0110 1010 0111 0000 0011 0001

0xE26A7031

I-type constants

ARM7 provides only 8-bits for specifying an immediate constant value. Given that ARM7 is a 32-bit architecture, this may appear to be a severe limitation. However, by allowing for a rotating shift to be applied to the constant.

```
imm32 = (imm8 >> (2 * rotate)) | (imm8 << (32 - (2 * rotate)))
```

Example: 1920 is encoded as:

Rotate	Imm8
1101	00011110

= (30 >> ~~(2*13))~~ | (30 << (32 - (2*23)))

= 0 | 30 * 128 = 1920

Next Time

- We will examine more instruction types and capabilities
  - Branching
  - Loading from and storing to memory
  - Special instructions
- Result flags
- Processor Status Registers