University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 220: Computer Systems & Programming

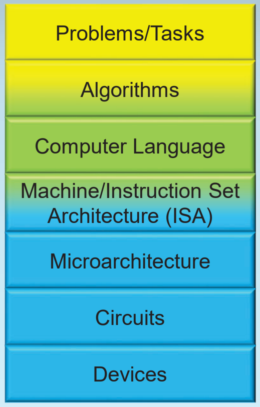
Introduction and Overview

Fall 2019

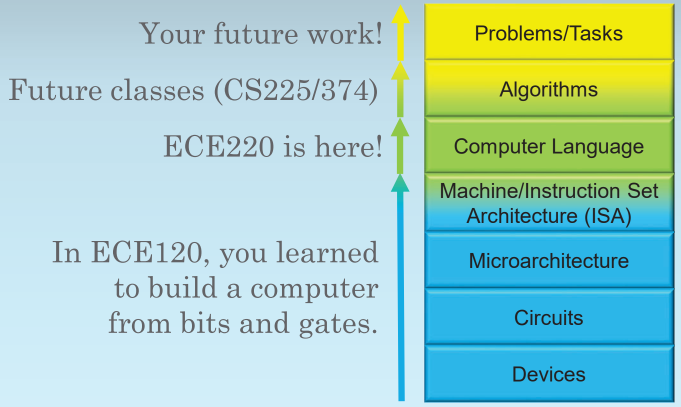
Digital Systems are Comprised of Seven Layers

The colors indicate the typical basis for each layer
- human language / theory
- software
- digital hardware

(figure based on Patt & Patel Ch. 1)



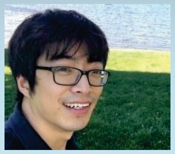
Our Class Builds Upwards from ECE120


Your future work!
Future classes (CS225/374)
ECE220 is here!
In ECE120, you learned to build a computer from bits and gates.
Problems/Tasks
Algorithms
Computer Language
Machine/Instruction Set Architecture (ISA)
Microarchitecture
Circuits
Devices

What is ECE220?

- Teach a systems perspective that includes both hardware and software (and math!)
- ECE culture and goals
- Expectations of engineers
- Lifelong learning necessary
- Understand and identify tradeoffs
- International group—leverage it!
- Academic reality and grade philosophy

Our Staff



Prof. Yuting Chen (Course Coordinator)

ywchen@illinois.edu
Phone: (217) 300-2813
Office: 3064 ECEB

Prof. Volodymyr Kindratenko

kindrtnk@illinois.edu
Phone: (217) 265-0209
Office: 3044 ECEB and 3050E NCSA

Prof. Ujjal Bhowmik

ubhowmik@illinois.edu
Phone: (217) 300-4257
Office: 2054 ECEB

Prof. Thomas Moon

tmoon@illinois.edu

Graduate Teaching Assistants
Iou-Jen Liu (Head TA), Xingkai Zhou, Andrew Chen, Xiaohao Wang, Liz Li, Emily Moog, Huizi Hu

Undergraduate Course Aides

Where to Find Information

Start with the web page!

One way: remember this link
https://wiki.illinois.edu/wiki/display/ece220

Another way:
- type “ECE 220” into Google
- follow first link that comes up

Read Web Page and Piazza Every Day

On the web page:
- announcements from course staff
- course information and timing
- assignments, exams, and due dates
- reference materials

On Piazza:
- ask any non-personal questions here
- do not post answers

Workload Includes Machine Problems

Machine Problems (MPs) every week
- programming assignments
- usually due **Thursdays**
  - **FIRST MP: Thursday, September 5th**
- submit via GitHub at UofI
  - **you are responsible for testing**!

Workload Also Includes Exams

6 programming quizzes (at CBTF)
- each **designed to take 50 minutes**
- **Dates are posted on the course schedule page**

2 midterm exams
- each **designed to take 1.5 hours**
  - **Thursday, October 3**
  - **Thursday, November 7**

Final exam
- **Expect to be here until Thursday, December 19**

Let Us Know About Conflicts Early

University has clear rules for conflicts (online)
- Midterms: **Section 3-202 of Student Code**
- Finals: **Section 3-201 of Student Code**

Finals rules
- depend on class sizes;
- if you can't tell, ask someone in advising.

If you have a conflict, **let us know early**!
(at least one week before the exam)

And Workload Includes Labs

In discussion section (aka lab), you will…

- solve programming problems
- related to concepts from lecture
- and somewhat relevant to your MPs.

How Will We Grade?

MPs: 15%

Quizzes: 20%

Midterm 1: 20%

Midterm 2: 20%

Final Exam: 25%

Late Policy for MPs: -2 pts per hour until Saturday night. We will grade ONLY your last submission.

Get to Know Your Fellow Students

Say "hi" to the person next to you in lecture, discussion, lab.
Go ahead, try it now. Really!

Don't Cheat!

See **Section 1-402 of the UIUC Academic code**.

In lab and MP assignments in our class, **you can work with other students**.

- It's ok to talk and help each other understand, but it's not ok to give/share/lend/copy/allow someone to copy code/answers.

However, all quizzes and exams however are **individual assignments!**

Your Guide to the Slides

The title gives the main point.

Definitions and **key messages** in bold blue.

Parameters and **variables** in bold green.

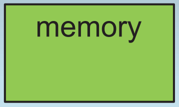
Other colors used on a per-slide basis.

University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 220: Computer Systems & Programming

Von Neumann model

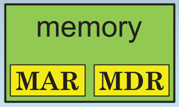
The von Neumann Model Includes a Memory



In 1946, John von Neumann invented
- a **model for computer organization**
- in which a **computer comprises five parts**.

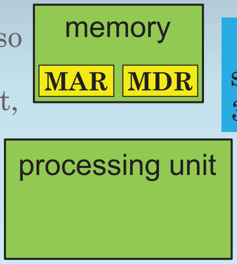
One part is memory:
- The same as you've seen.
- One operation per cycle: read or write.
- The computer's **instructions (program) are stored in the memory**.

The Memory Uses Two Registers: MAR and MDR



The memory uses two registers to manage data:
- The **Memory Address Register (MAR) holds the address** on which the memory operates (to read or to write).
- The **Memory Data Register (MDR) holds the bits** read from the memory, or the bits to write to the memory.

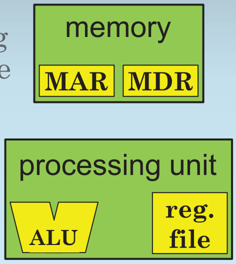
A Computer Contains a Processing Unit



Typical word sizes today are 32 and 64 bits.

A computer also contains a processing unit, which **performs all operations.** and **defines the word size** for the computer, the number of bits used in most computations.

The Processing Unit Includes an ALU and a Register File



The register file is faster but smaller than memory.

The processing unit makes use of an **Arithmetic Logic Unit (ALU)** to handle the operations.

The processing unit also contains a **register file** for temporary storage of values.

The Register File is Fast but Small

The register file contains (surprise!) registers.

Registers use flip-flops and share a clock with the ALU.

Registers are thus

- **faster than SRAM**, and
- **much faster than DRAM** (usually off-chip).

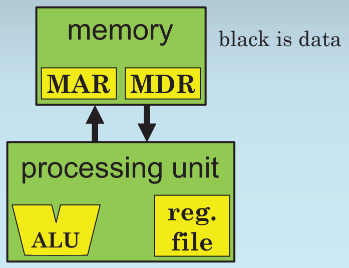
But usually there are **only tens or a hundred registers** (again for speed reasons).

As you might expect, the registers in the register file are named with bits.

Black Arrows Represent Data Moving Amongst Elements

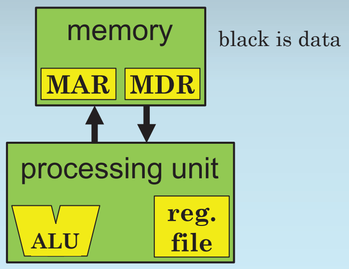
Data moves between the processing unit and memory.

The black arrows represent data.



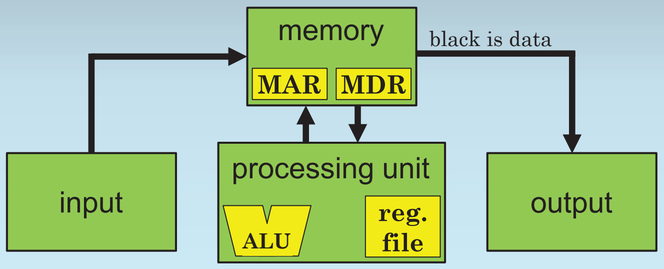
A Computer Needs Methods for Input and Output

A computer also needs the ability to **get input** from outside, and to **deliver results** to the external world.



For example, a keyboard, monitor, mouse, disk, printer, network, and so forth.

The von Neumann Model Includes Input and Output



The **von Neumann model includes both input and output**.

The Fifth Element of the von Neumann Model

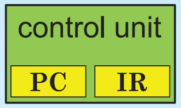
I said five components, right?

What's missing?

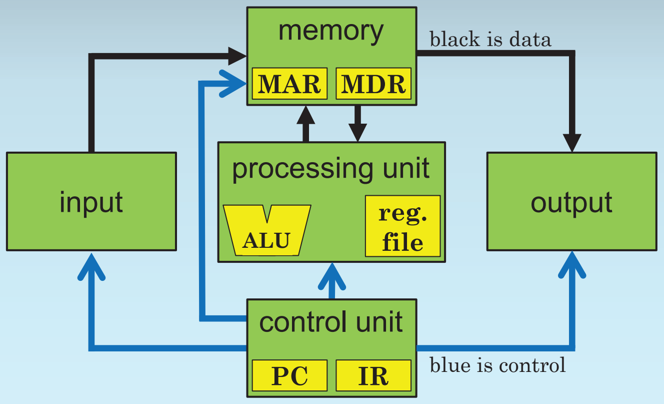
One control unit to rule them all!

The **control unit (an FSM)** uses two registers:

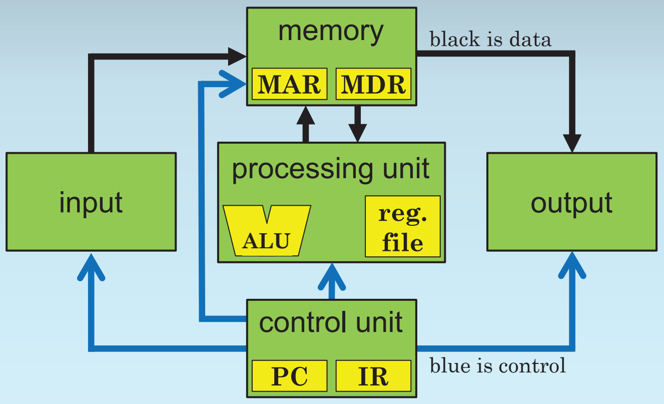
- The **Program Counter (PC) holds the address of the next instruction**.
- The **Instruction Register (IR) holds the bits** of the current instruction.



The Control Unit (FSM) Controls All Other Elements



The von Neumann Model Consists of Five Parts



University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 220: Computer Systems & Programming

LC-3

Build an LC-3 Processor as a von Neumann Machine

Let's talk about a specific von Neumann machine.

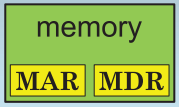
The **Little Computer-3 (LC-3) ISA**

- was developed by Patt & Patel
- as **an educational tool**.

As Yale (Patt) says, it took them **three tries to get it right**, hence LC-3.

In our class, we will build up towards the LC-3 microarchitecture in Appendix C of P&P.

The LC-3 Memory is 2^{16}×16-Bit



Let's start again with the memory.

In the LC-3 ISA,

- memory has **2^{16} addresses**, and
- **16-bit addressability**.

Call this number X.

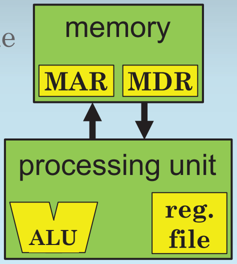
And call this Y.

How many bits in the MAR, X or Y? **X (16)**

How many bits in the MDR? **Y (also 16)**

The LC-3 ALU Supports ADD, AND, and NOT Operations

The ALU in the LC-3 supports three operations: **ADD, AND, and NOT**.



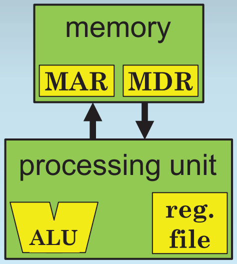
And remember DeMorgan's Law: A+B = [A'B']'

What if someone wants another function?

Apply logical completeness.

The LC-3 Register File Has Eight Registers

The register file contains **eight registers**.



The "R" is just for humans, of course.

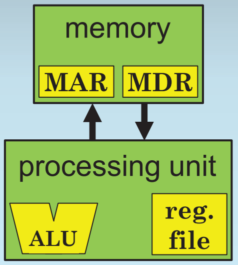
Guess what their names are…

~~**Donner, Blitzen**~~**…** **No. R0 through R7.**

The LC-3's Word Size is 16 Bits

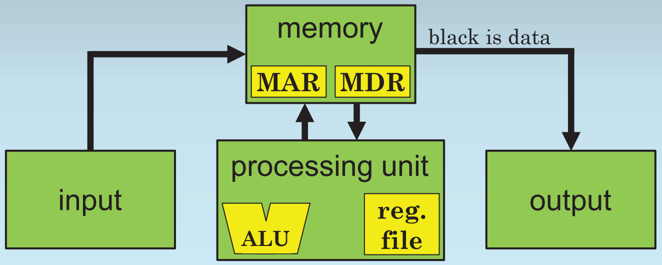
The LC-3's **word size is 16 bits**.

The **ALU operates on 16-bit operands**.



Each register in the register file **stores 16 bits**.

The LC-3 Includes a Keyboard and a Display



The LC-3 has one **input** device: **a keyboard**.

And one **output** device: **a monitor/display**.

* *

Details of the LC-3 Input Device

keyboard input uses two registers

- **Keyboard Status Register (KBSR)**
  used to handshake when a key arrives
- **Keyboard Data Register (KBDR)**
  used to delivers keystrokes, coded as **ASCII**

Using these registers is a topic for next lecture (see Ch. 8-9 of Patt & Patel).

* *

Details of the LC-3 Output Device

display output also uses two registers

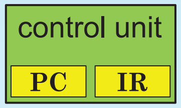
- **Display Status Register (DSR)**
  used to handshake (processor must wait for the display!)
- **Display Data Register (DDR)**
  used to send characters to print, coded as **ASCII**

Again, using these is a topic for next lecture (see Ch. 8-9 of Patt & Patel).

The LC-3 Also Has a Control Unit

Recall that
- the **program counter (PC)** stores the address of the next instruction, and
- LC-3 memory is **$2^X \times Y$-bit**, where **X** and **Y** are both 16.

How many bits in the PC, X or Y? X (16)

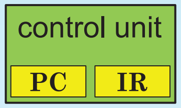


What About the IR?

Recall that the **instruction register (IR)** stores the encoded bits of the instruction being executed.

How many bits in the IR?

16

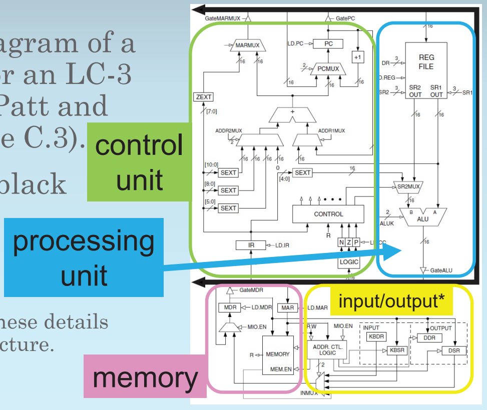


A Datapath for an LC-3 Processor

Here's a diagram of a datapath for an LC-3 processor (Patt and Patel Figure C.3).

The heavy black line is a 16-bit bus.

*We will study these details of I/O in next lecture.



University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 220: Computer Systems & Programming

LC-3 ISA

What About the IR?

How do we encode instructions?

The ISA defines a representation.

Instructions may require a variable number of bits (as in x86).

However, **in the LC-3 ISA, every instruction requires 16 bits**.

This design choice is **deliberately equal to the addressability of the memory** so that each memory location holds one instruction.

So, yes, the **IR requires 16 bits**.

Encoding Instructions

How do we represent instructions?

With bits, of course!

Instructions are encoded using a representation defined by the ISA.

The LC-3 ISA uses 16 bits to encode instructions.

That's a big representation!

The LC-3 ISA Has Three Kinds of Instructions

The **LC-3 ISA** has **three kinds of instructions**:

1. **operations** (with the ALU)
2. **data movement** (registers to/from memory)
3. **control flow** (conditionally change the PC)

Let's look at each kind in turn.

The LC-3 Supports Three Operate Instructions

The LC-3 ALU is capable of **three operations**.

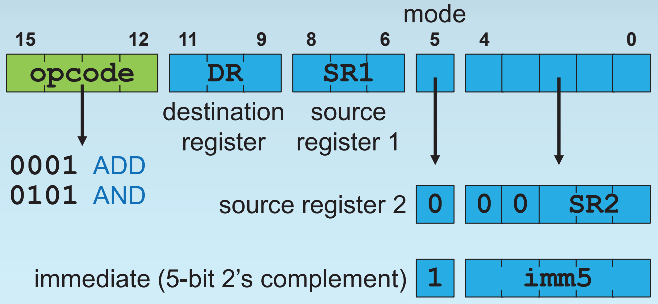
The ISA includes **one opcode for each** operation: **ADD**, **AND**, and **NOT**.

Each operation uses
- **one source register** and
- **one destination register**.

The second operand (for **ADD** and **AND** only) allows a choice of **addressing modes**:
- **register** (another register) or
- **immediate** (a number stored in the instruction).

ADD and AND Have Two Addressing Modes

IR[5] is the mode bit for the second operand.



What Can We Do with Immediate Mode?

Add small numbers
- +1
- -1
- (anything from -16 to +15).

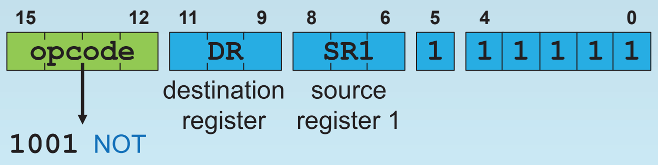
Mask out high bits
- AND 1
- AND 3

Mask out low bits
- AND -2 (xFFFE)
- AND -4 (xFFFC)

Put 0 in a register! (AND 0)

Second Operand Bits are All 1 for the NOT Instruction

For **NOT**, **IR[5:0] must be 1**.

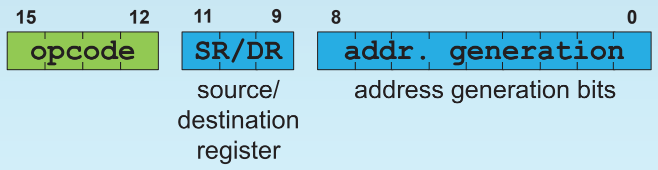


(NOT only has one input operand.)

Loads and Stores Have Four Addressing Modes

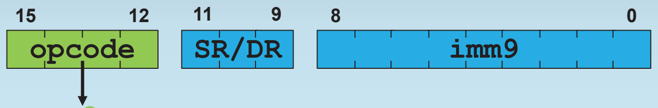
Data movement instructions are of two types:
- **loads** (from memory to register)
- **stores** (from register to memory)

The LC-3 ISA supports **four addressing modes** for data movement instructions.



PC-Relative Addressing Adds an Offset to the PC

The first addressing mode is **PC-relative**.



`0010` LD `DR ← M[PC + SEXT16(imm9)]`

`0011` ST `M[PC + SEXT16(imm9)] ← SR`

memory read/write control

Important: **PC** is the **value after FETCH**
(the address of the LD/ST + 1).

Example of PC-Relative Addressing

What does this instruction
(at memory address x1480) do?

`x1480 LD R3,x09`

In **RTL**: `R3 ← M[PC + SEXT16(x09)]`

What is PC?

Again, since execution occurs after **FETCH**,
PC in this case is x1481. So…

`R3 ← M[x148A]` **(NOT x1489!)**

Time for a Quiz!

What's stored at a memory address?

Bits. (16 of them, yes.)

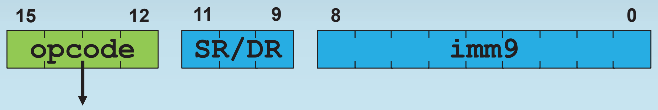
How do we name a memory address?

Bits. (Again 16 of them.)

So we can
use the bits at a memory address
to name another memory address!

Indirect Addressing Accesses Memory Twice

The second addressing mode is **indirect**.



`1010` LDI `DR ← M[M[PC + SEXT16(imm9)]]`

`1011` STI `M[M[PC + SEXT16(imm9)]] ← SR`

Indirect mode reads the address for the
load/store from a PC-relative address.

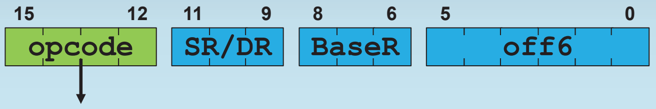
What Purpose Does Indirect Mode Serve?

Primarily to make you realize that
- the bits at a memory address
- can **point to** a memory address.

The concept of a **pointer**
- (just another word for a memory address)
- is **critical for understanding more complex representations** in many programming languages (such as C).

Base + Offset Mode Uses Another Register

The third addressing mode is **base + offset**.



0110 LDR **DR ← M[BaseR + SEXT16(off6)]**

0111 STR **M[BaseR + SEXT16(off6)] ← SR**

Base + offset mode **uses another register** to generate the address for the memory access.

Base + Offset Mode Enables Wider Memory Access

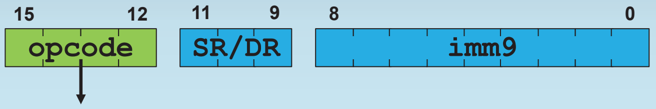
PC-relative and **indirect** addressing
- can only generate
- addresses within a 9-bit offset
- of the instruction address (-256 to +255)
- (Indirect addressing can access any location, but the address has to be stored near the **LDI/STI**.)

Base+Offset enables access to any address by using another register.

But how do we get an address into a register?

Immediate Addressing Loads a Nearby Address

The fourth addressing mode is **immediate**.



1110 LEA **DR ← PC + SEXT16(imm9)**

LEA stands for "load effective address":
- **address generation is PC-relative**, but
- **memory is not accessed** (so not really data movement).

Control Flow Instructions Conditionally Change PC

After executing an instruction at address **A**, the **LC-3** next executes the instruction at address **A + 1**, then **A + 2**, and so forth.

So far, we have seen **operations** and **data movement** instructions.

But how can we do things like **`if`** statements and loops?

We need another kind of instruction to manage **control flow**.

Control flow instructions **conditionally change the PC**.

LC-3 ISA Provides Three Condition Codes: N, Z, and P

The **LC-3** maintains **three 1-bit registers** called **condition codes**.

These are based on the **last value written to the register file** (by operations or by loads).

N: the last value was **negative**

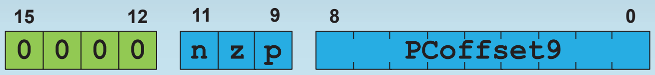
Z: the last value was **zero**

P: the last value was **positive**

Obviously, **exactly one of these three bits is 1** in any cycle.

Conditional Branch BR* Conditionally Changes PC

Let's start with conditional branch, BR.

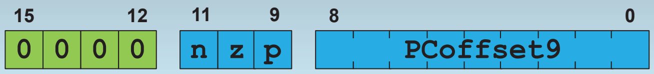


`BEN: PC ← PC + SEXT16(PCoffset9)`

The calculation of **BEN**, the **branch enable condition**, is specified in the opcode's name.

For example, **BRnp** has the **`n`** and **`p`** bits set in the instruction, while the **`z`** bit is zero.

Calculation of the Branch Enable Condition



`BEN: PC ← PC + SEXT16(PCoffset9)`

The **BEN** condition is calculated

- in the **DECODE** state
- based on instruction bits **`n`**, **`z`**, **`p`** and
- condition codes **N**, **Z**, and **P**:

`BEN ← nN + zZ + pP`

Examples of Branch Condition Names

Let's consider a few examples…

BRnz	**branch if not positive**
BRnzp	**branch always**
BRnp	**branch if not zero**

Note: by convention, **BR** means **BRnzp** (unconditional branch), not "do nothing."

JMP Instruction Copies Any Register to the PC

Branch target addresses are limited.

The **BR** instruction only has a **9-bit offset**:
- **PC** is the **BR address + 1**, and
- add another -256 to +255.

What if we want to change **PC** to something farther from the current instruction?

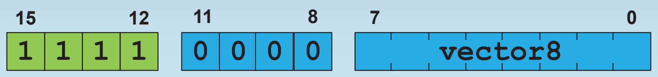
Use a JMP (jump) instruction!* The **RTL** is

$$\texttt{PC} \leftarrow \texttt{BaseR}$$

*Look up the encoding if you need it.

Use TRAP to Invoke Operating System Services

One more instruction: TRAP.



The **TRAP** instruction **invokes operating system services**.*

The specific service depends on **vector8**.

*For a detailed explanation of the mechanism, read Ch. 8 & 9 of Patt and Patel, or wait for ECE220.

TRAP Functions Available in the LC-3 OS

TRAP vec #	mnemonic	effect
x20	GETC	read one ASCII character from keyboard into R0
x21	OUT	write one ASCII character from R0 to display
x25	HALT	end program (return control to the "operating system")

For more detail, see p. 543 of Patt and Patel.

Do NOT Use R7 in Your ECE220 LC-3 Programs

Obviously,
- if you invoke the **GETC** trap,
- whatever bits were in **R0** are lost.

Not so obviously,
- any **TRAP** will change **R7**.
- **Do NOT use R7 for our class**.

(Again, see Ch. 8 and 9 of Patt and Patel if you want to know why.)

University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 220: Computer Systems & Programming

LC-3 Assembly Language

Review Our Process for Programming

Step 1: Figure out the instruction sequence.

Step 2: Map instructions and data to memory addresses.

Step 3: Calculate and fill in relative offsets.

Step 1 is hard.

Steps 2 and 3 are … counting.

That's the fun part!

But maybe some of us might get bored.

Can a Computer Help Us Program?

Step 1: Figure out the instruction sequence.

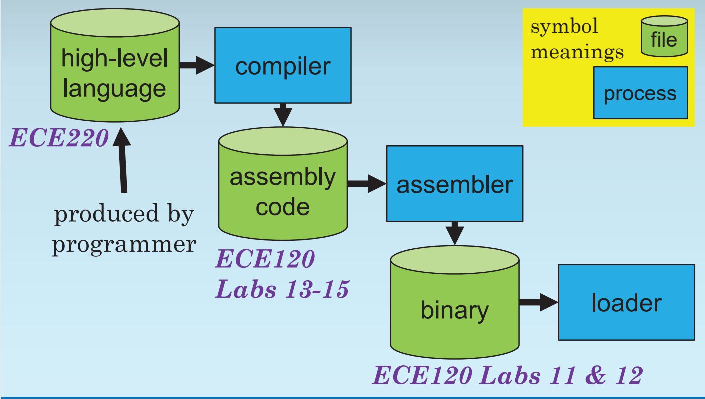
We have do this part (computers are dumb).

Step 2: Map instructions and data to memory addresses.

Step 3: Calculate and fill in relative offsets.

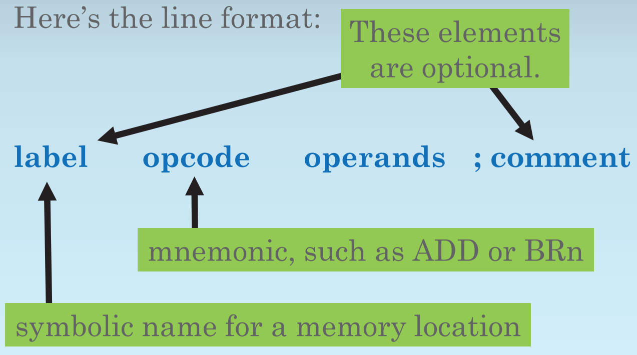
We can program a computer to do these.

A Typical Programming Process



Assembly Language is Written One Line at a Time

Here's the line format:


These elements are optional.
label
opcode
operands
; comment
mnemonic, such as ADD or BRn
symbolic name for a memory location

Examples of LC-3 Assembly Language

Here are a couple of examples…

```
INFLOOP BRnzp INFLOOP ; get it?

LD R3,INFLOOP
```

Labels name memory locations.

Assembly Language Supports Directives and Pseudo-Ops

Assembly language also supports*

- **directives**, which provide information to the assembler, and
- **pseudo-ops**, which are shortcut notation for various types of bits.

*Most people do not distinguish between these two elements of assembly language.

LC-3 .ORIG Directive Must Appear Once at Start

The **.ORIG** directive tells the assembler **where to start writing bits in memory**.

For example:

```
.ORIG x3000
```

This directive

- **must appear exactly once** in any assembly file, and
- must **appear before any lines that generate bits** (only comments can precede **.ORIG**).

LC-3 .END Directive Ends the File

The **.END** directive tells the assembler **to stop reading the file**.

For example:

```
.END
```

Any lines after the .END directive are ignored by the assembler.

Generally, one should always **put it at the end of the file** to avoid confusion.

Note that **.END** is NOT a **HALT** (**TRAP x25**).

LC-3 .BLKW Directive Skips Over Memory Locations

The **.BLKW** directive tells the assembler **to leave blank words in memory.**

For example:

```
.BLKW #30
```

skips 30 memory locations.

Do not assume that these locations are filled with 0s.

(Although they will be by the **LC-3** assembler, not all assemblers do so.)

When Would One Use .BLKW?

Remember when we wrote code

- to read a number from the keyboard
- and store the typed value in memory?

That's one case in which we use **.BLKW**:

- We need a place in memory.
- But we don't need it initialized.

LC-3 .FILL Pseudo-Op Allows Us to Write Specific Bits

What if we want to write data bits into memory?

The `.FILL` pseudo-op tells the assembler **to write a specific 16-bit value** into the next memory location.

For example:

```
.FILL xFFD0
```

writes the bits 1111 1111 1101 0000 into the next location.

LC-3 .STRINGZ Pseudo-Op Allows Us to Write Strings

The `.STRINGZ` pseudo-op tells the assembler **to write a NUL-terminated ASCII string** into memory.

For example:

```
.STRINGZ "Hello!"
```

ASCII characters (zero-extended to 16 bits) are **written into consecutive memory locations**, and followed by a **NUL** (x00) in another memory location.

.STRINGZ Always Writes a NUL

Don't forget that `.STRINGZ` always writes a **NUL** after the **ASCII** characters in the string.

So **the number of memory locations needed is the number of characters + 1**.

How many memory locations for …

```
.STRINGZ "One..."   ? 7
.STRINGZ "Two?"     ? 5
.STRINGZ "3"        ? 2
```

Use Traps by Name in LC-3 Assembly Language

The **LC-3** assembler also supports pseudo-ops for **TRAP** instructions.

The ones that you have seen* are…

```
GETC    ; TRAP x20
OUT     ; TRAP x21
HALT    ; TRAP x25
```

*Patt & Patel p. 543 has a couple more.

University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 220: Computer Systems & Programming

MP1

Time to Write A Program

Let's say that we want to do the following:
- given an **ASCII** string (a sequence of characters terminated by a **NUL**, **ASCII x00**),
- count the occurrences of each letter (regardless of case), and
- count the number of non-alphabetic characters.
- And print this as a histogram

Let's Develop a Flow Chart

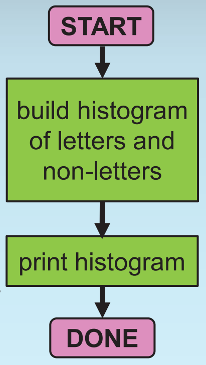
Ready?

My work here is done.

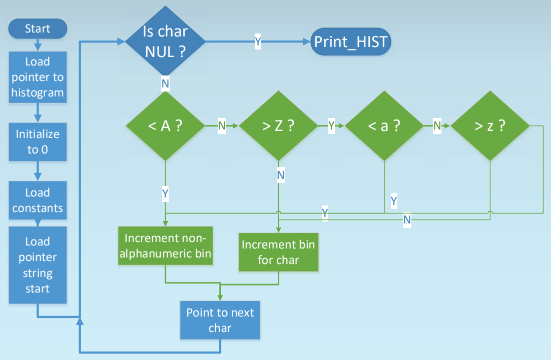
Now you can apply **systematic decomposition**.

What's a histogram?

A function on a set of categories.



Letter Frequency Counting Flowchart



Writing the Letter Frequency Program in Assembly

Let's get started…

Let's start our code here.

```
            .ORIG x3000
; I don't feel like writing
; initialization yet.  In assembly,
; I can come back later with no
; worries.  The assembler will
; recalculate offsets.
```

Labels Make Programming Much Easier

Let's write the counting part…

Make up a name: we'll need to come back.

```
COUNTLOOP    LDR R2,R1,#0
             BRz DONE

DONE         HALT
```

Just make up a name!

We can even write that code first!

Found the end of the string. Where do we go?

Assembler will calculate the BRz offset for us!

What is a Label, Exactly? A Memory Address!

A label represents an address.

This instruction

```
COUNTLOOP    LDR R2,R1,#0
             BRz DONE

DONE         HALT
```

… is at this address.

When BRz is taken, PC changes to this address.

Next, Compare with Capital A

What's next? Compare with capital A.

```
COUNTLOOP    LDR R2,R1,#0
             BRz DONE
             ADD R2,R2,R3
             BRp AT_LEAST_A
AT_LEAST_A   ; placeholder for later
```

Again, just make up a name!

Found a character >= 'A'. Where do we go?

Increment the Non-Alphabetic Bin

What's next? Compare with capital A.

We could add this name now or later.

```
NON_ALPHA     LDR R6,R0,#0
              ADD R6,R6,#1
              STR R6,R0,#0
              BRnzp GETNEXT
```

Done with this character. Where to?

Again, just make up a name!

Place Data After the Code (But Before .END!)

What about data? After the code…

```
NUM_BINS     .FILL #27
NEG_AT       .FILL xFFC0
STR_START    .FILL STRING
HIST         .BLKW #27
STRING       .STRINGZ "Example."
```

Now, we can easily place the histogram and string behind the code.