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

ECE 120: Introduction to Computing

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.

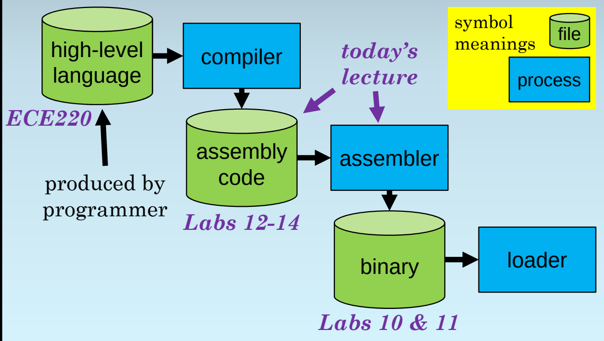
We have do this part (computers are dumb).

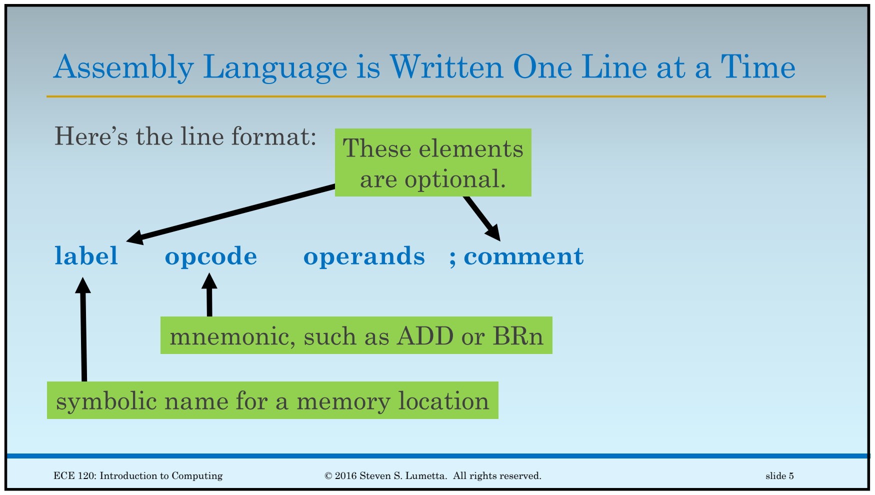
Step 2: Map instructions and data
to memory addresses.

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

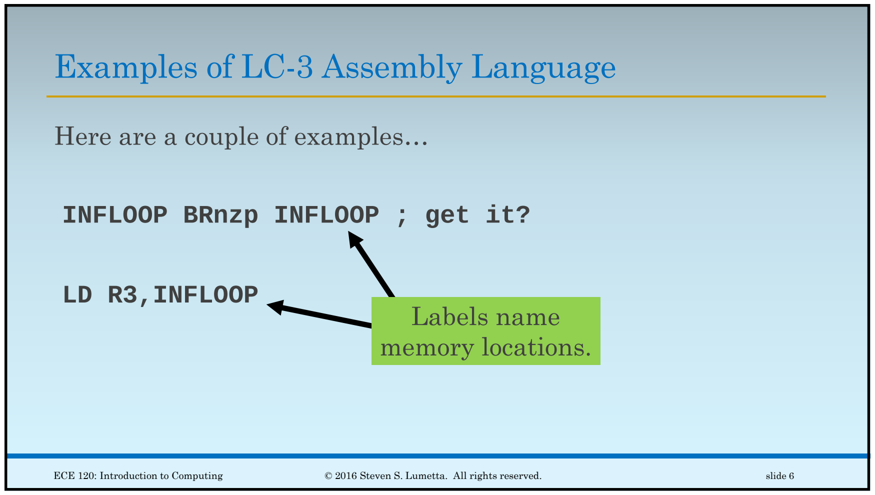
Here's the line format:

These elements are optional.

label **opcode** **operands** **; comment**

mnemonic, such as ADD or BRn

symbolic name for a memory location

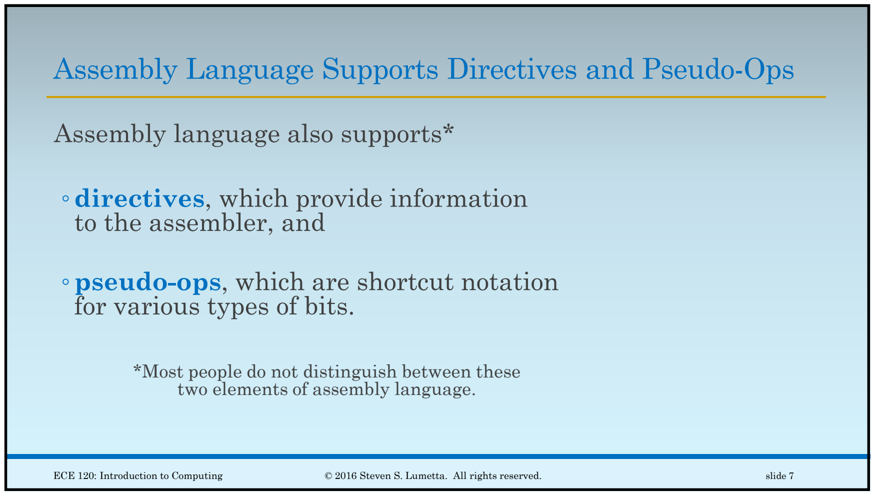


Examples of LC-3 Assembly Language

Here are a couple of examples…

```
INFLOOP BRnzp INFLOOP ; get it?

LD R3,INFLOOP
```

Labels name memory locations.

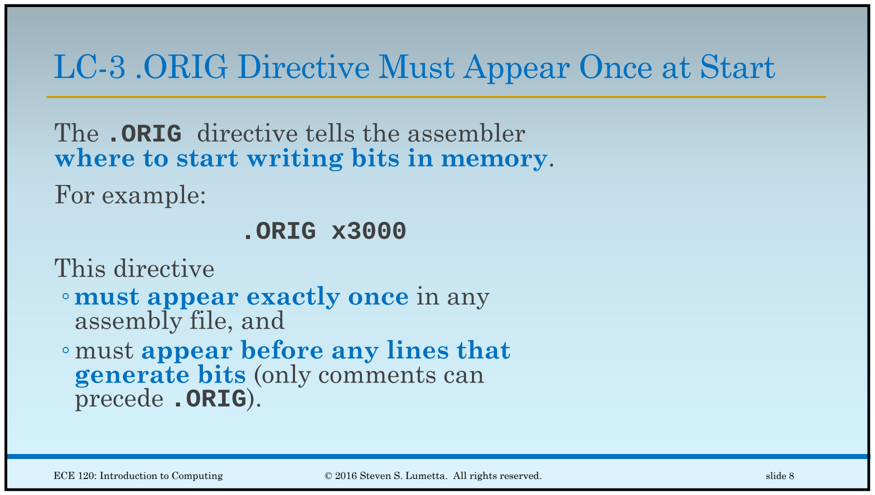


Assembly Language Supports Directives and Pseudo-Ops

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.

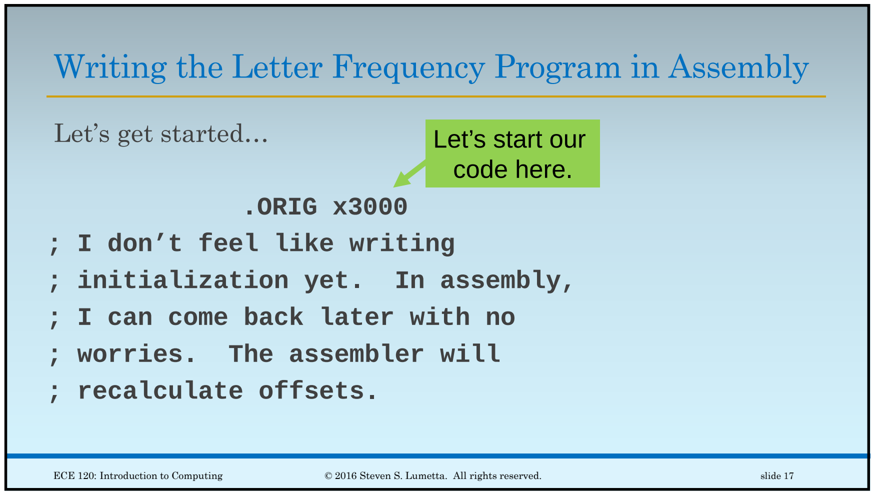
What's the Advantage of Assembly?

Let's pretend that we're writing our letter frequency program in assembly.

You can read and get the code (in both forms) on our web page.

But here I want to pretend

- that we are writing it
- in order to highlight the advantages of assembly.

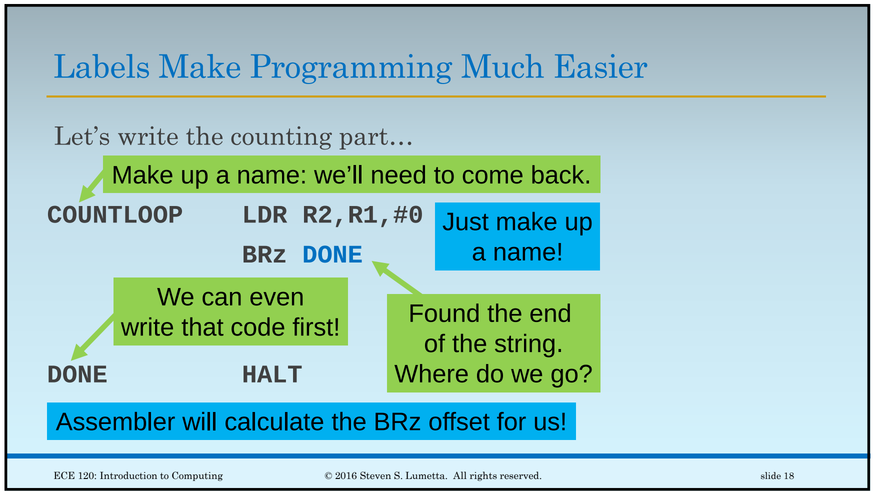


Writing the Letter Frequency Program in Assembly

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.

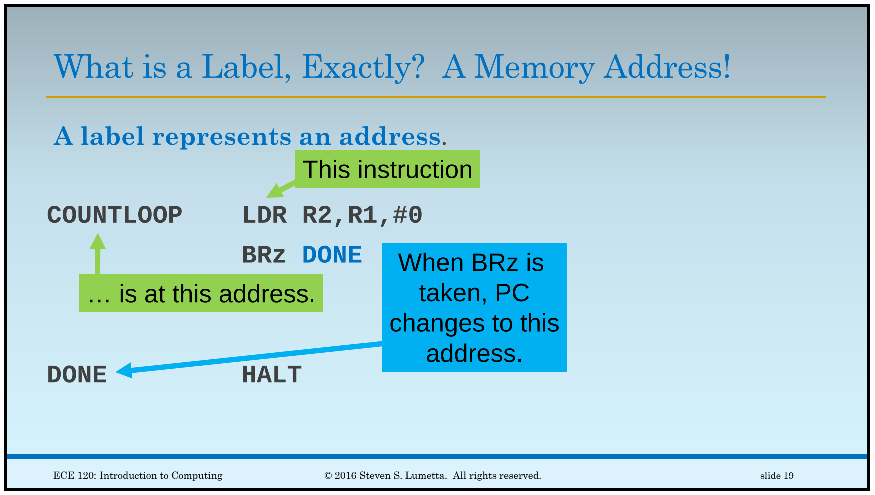
```
COUNTLOOP    LDR R2,R1,#0
             BRz DONE

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!

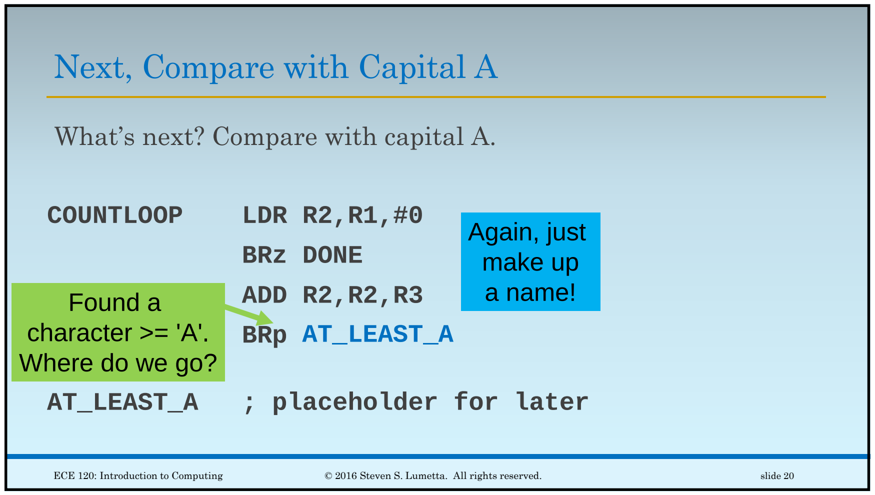
A label represents an address.

This instruction

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

DONE         HALT
```

… is at this address.

When BRz is taken, PC changes to this address.



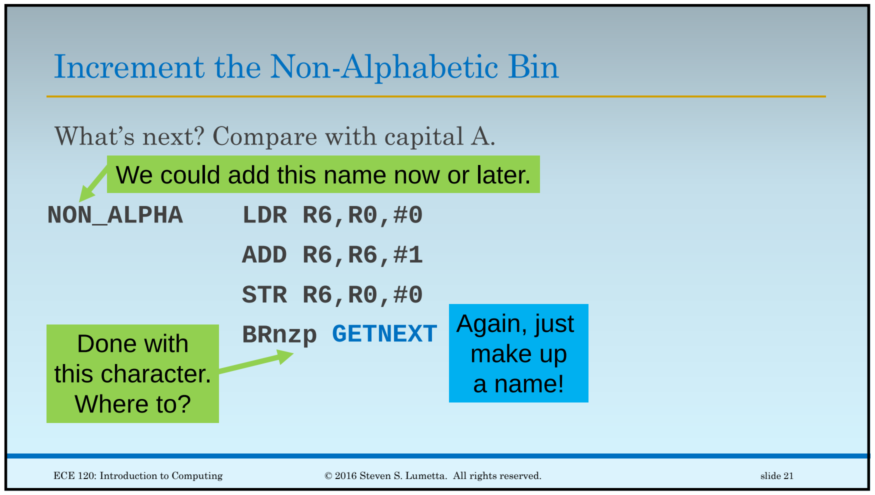
Next, Compare with Capital A

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

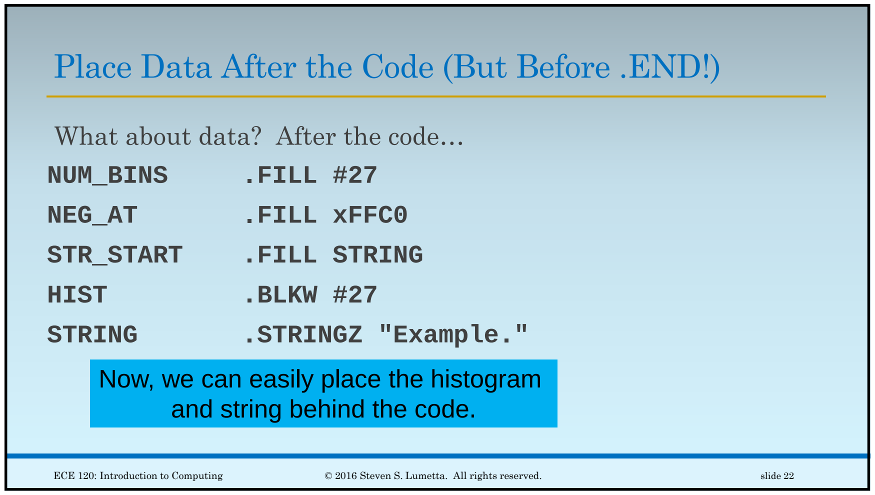
What's next? Compare with capital A.

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.