

From Bits and Gates to C and Beyond

Welcome Aboard

Chapter 1

Introduction to the World of Computing

There is no **magic** to computing!

Deterministic system – behaves the same way every time.

Does exactly what we tell it to do: no more, no less.

Complex system made of very simple parts.

Even recent advances in AI come from our ability to do many (billions!) simple computations very fast!

[Access the text alternative for slide images.](#)

Two Recurring Themes

Abstraction: Productivity Enhancer

You don't need to worry about the details...

- You can drive a car without knowing how an internal combustion engine works.

... until something goes wrong!

- Where's the dipstick? What's a spark plug? Where's that smoke coming from?

Important to understand the components and how they work together.
But thinking at higher levels of abstraction is more efficient.

Hardware and Software

It's not either/or – both are essential components of a computer system.

Even if you specialize in one,
you must understand the capabilities and limitations of the other.

Big Idea #1: Universal Computing Device

All computers, given enough time and memory, are capable of computing exactly the same things.

- Smartphone, laptop, supercomputer... limited only by time and memory.

Anything that can be computed, can be computed by a computer.

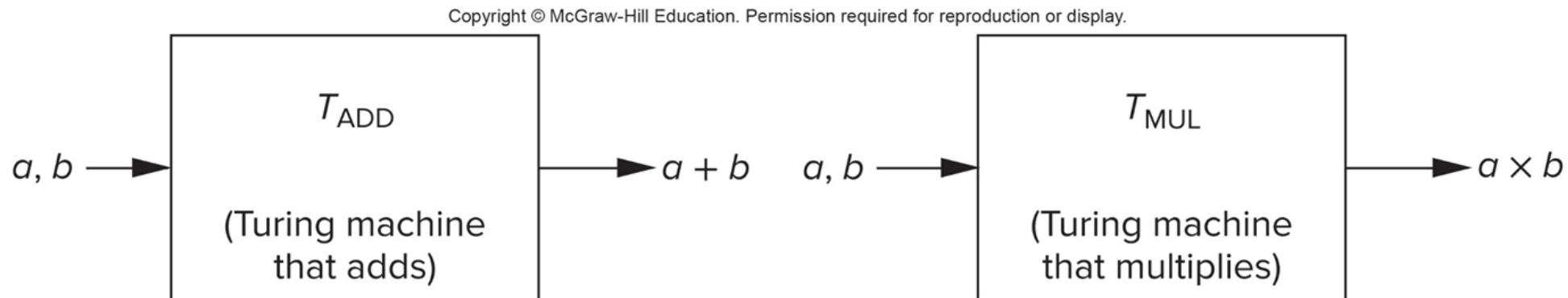
- If you can describe something in terms of computation, it can be done by a computer... again, given enough time and memory.

Turing Machine

Mathematical model of a device that can perform any computation – Alan Turing (1937)

- ability to read/write symbols on an infinite “tape”
- state transitions, based on current state and symbol

Every computation can be performed by some Turing machine. (*Turing's thesis*)



For more info about Turing machines, see
http://www.wikipedia.org/wiki/Turing_machine

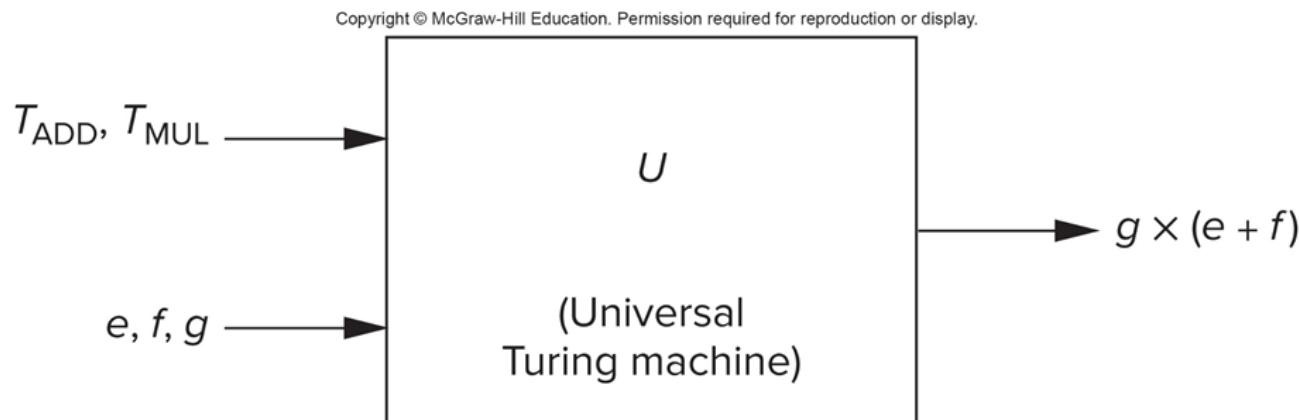
For more about Alan Turing, see
<http://www.turing.org.uk/>

[Access the text alternative for slide images.](#)

Universal Turing Machine

A machine that can implement all Turing machines
-- this is also a Turing machine!

- inputs: data, plus a description of computation (other TMs).



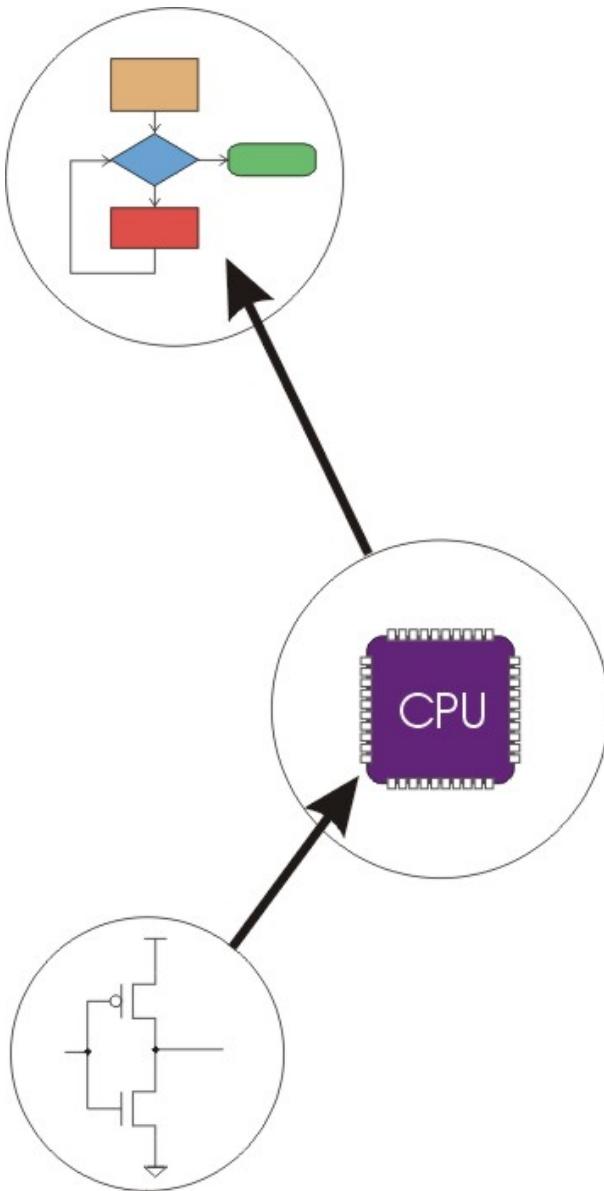
U is programmable – so is a computer!

- Instructions are part of the input data.
- A computer can emulate a Universal Turing Machine.

A computer is a universal computing device.

[Access the text alternative for slide images.](#)

Big Idea #2: Transformations Between Layers



Problems

Algorithms

Language

Instruction Set Architecture

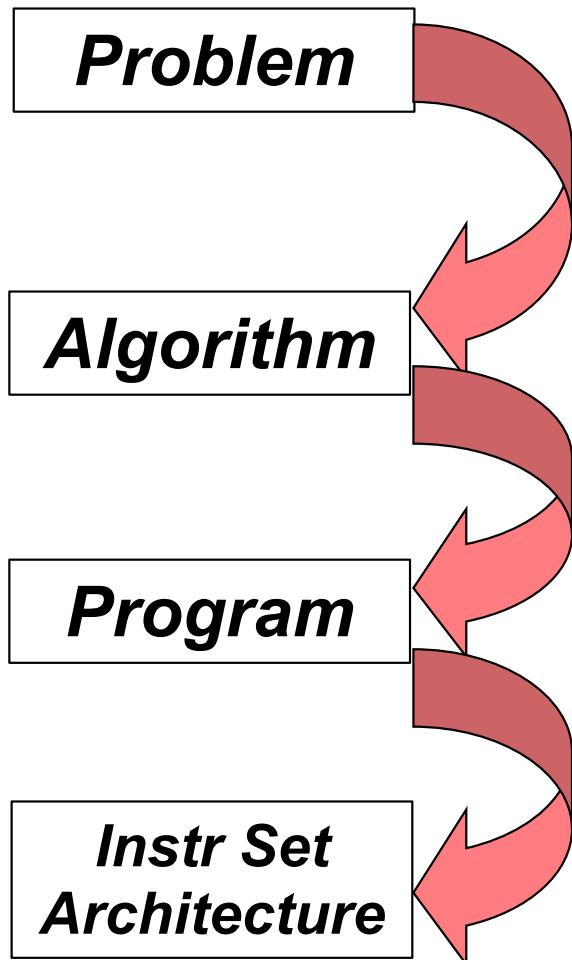
Microarchitecture

Circuits

Devices

How do we solve a problem using a computer?

A systematic sequence of transformations between abstraction layers.



Software Design:

choose algorithms and data structures

Programming:

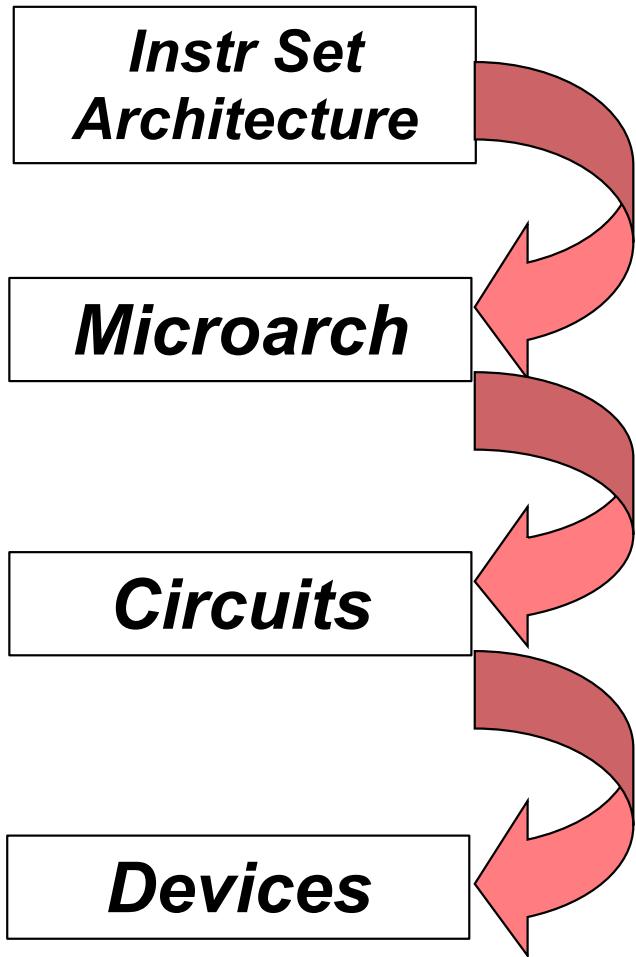
use language to express design

Compiling/Interpreting:

convert language to machine instructions

[Access the text alternative for slide images.](#)

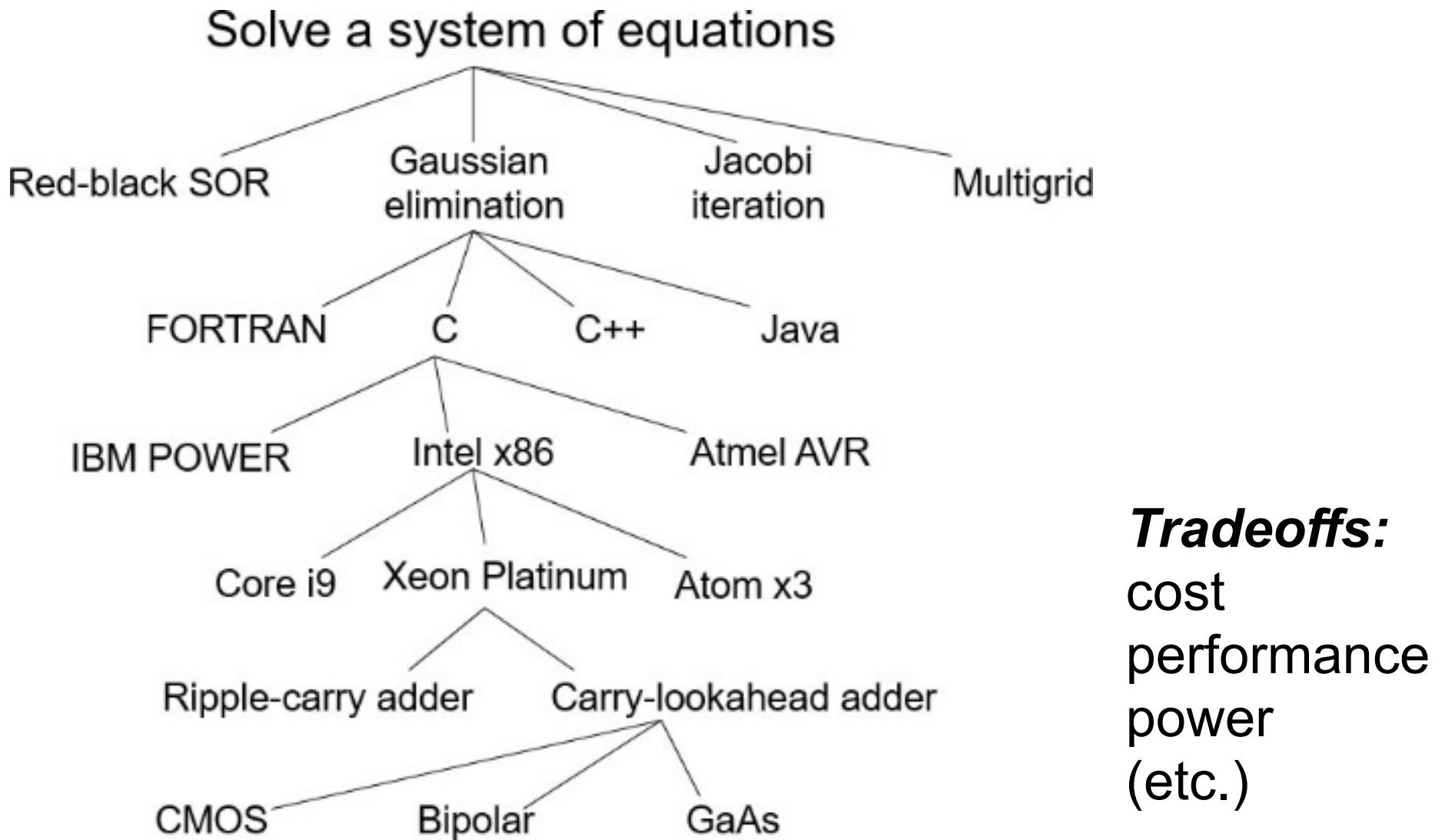
...and even more layers...



- Processor Design:***
choose structures to implement ISA
- Logic/Circuit Design:***
gates and circuits to implement components
- Process Engineering & Fabrication:***
develop and manufacture transistors, wires, etc.

[Access the text alternative for slide images.](#)

Many choices at each layer



[Access the text alternative for slide images.](#)

Course Outline

Bits and Bytes

- How do we represent information using electrical signals?

Digital Logic

- Leave for next course

Processor and Instruction Set

- How do we build a processor out of logic elements?
- What operations (instructions) will we implement?

Assembly Language Programming

- How do we use processor instructions to implement algorithms?
- How do we write modular, reusable code? (subroutines)

I/O, Traps, and Interrupts

- How does processor communicate with outside world?

C Programming

- How do we write programs in C?
- How do we implement high-level programming constructs?



Because learning changes everything.[®]

www.mheducation.com

From Bits and Gates to C and Beyond

Bits, Data Types, and Operations

Chapter 2

How do we represent data in a computer?

At the lowest level, a computer is an electronic machine.

- Works by controlling the flow of **electrons**.

Easy to recognize two conditions:

1. Presence of a voltage – we'll call this state “1.”
2. Absence of a voltage – we'll call this state “0.”

We could base the state on the value of the voltage, but control and detection circuits are more complex.

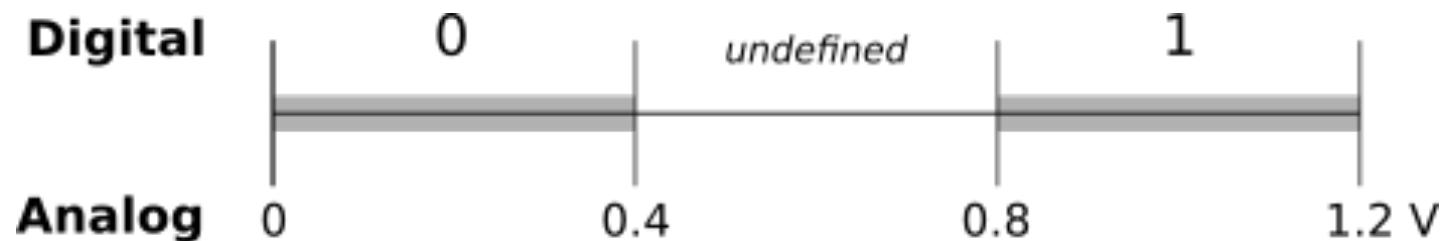
- Compare sticking a lightbulb (or your finger) in a socket versus using a voltmeter.

Computer is a Binary Digital system

Digital = finite number of values (compared to “analog” = infinite values)

Binary = only two values: 0 and 1

Unit of information = binary digit, or “bit”



Circuits (see Chapter 3) will pull voltage down towards zero, or will pull up towards the highest voltage.

Grey areas represent noise margin -- allowable deviation due to resistance, capacitance, interference from other circuits, temperature, etc. More reliable than analog.

[Access the text alternative for slide images.](#)

More than two values...

Each "wire" in a logic circuit represents one bit = 0 or 1.

Values with more than two states require multiple wires (bits).

With two bits, can represent four different values:

00, 01, 10, 11

With three bits, can represent eight different values:

000, 001, 010, 011, 100, 101, 110, 111

With **n** bits, can represent **2^n** different values.

What kinds of values do we want to represent?

Numbers - integer, fraction, real, complex, ...

Text - characters, strings...

Images - pixels, colors, shapes...

Sound

Video

Logical - true, false

Instructions

All data is represented as a collection of bits.

We encode a value by assigning a bit pattern to represent that value.

We perform operations (transformations) on bits, and we interpret the results according to how the data is encoded.

Data Type

In a computer system, we need a **representation** of data and **operations** that can be performed on the data by the machine instructions or the computer language.

This combination of *representation + operations* is known as a **data type**.

Type	Representation	Operations
Unsigned integers	binary	add, multiply, etc.
Signed integers	2's complement binary	add, multiply, etc.
Real numbers	IEEE floating-point	add, multiply, etc.
Text characters	ASCII	input, output, compare

Unsigned Integers

Non-positional notation (unary)

- Could represent a number (“5”) with a string of ones (“1111”).
- Problems?

Weighted positional notation (binary)

- Like decimal numbers: “329.”
- “3” is worth 300, because of its position, while “9” is only worth 9.
- Since only 0 and 1 digits, use a binary (base-two) number system.

$$\begin{array}{r} 329 \\ 10^2 \quad | \quad 10^1 \quad 10^0 \end{array}$$

$$3 \times 100 + 2 \times 10 + 9 \times 1 = 329$$

$$\begin{array}{r} \text{most} \dots \text{least} \\ \text{significant} \qquad \qquad \qquad \text{significant} \\ 101 \\ 2^2 \quad | \quad 2^1 \quad 2^0 \end{array}$$

$$1 \times 4 + 0 \times 2 + 1 \times 1 = 5$$

[Access the text alternative for slide images.](#)

Unsigned Integers 2

Encode a decimal integer using base-two binary number

Use **n** bits to represent values from 0 to $2^n - 1$

2^2	2^1	2^0	value
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

Unsigned Binary Arithmetic

Base-2 addition -- just like base-10

- Add from right to left, propagating carry.

$$\begin{array}{r} 10010 \\ + 1001 \\ \hline 11011 \end{array}$$

$$\begin{array}{r} 10010 \\ + 1011 \\ \hline 11101 \end{array}$$

$$\begin{array}{r} 1111 \\ + 1 \\ \hline 10000 \end{array}$$

$$\begin{array}{r} 10111 \\ + 111 \\ \hline \end{array}$$

Can also do subtraction, multiplication, etc., using base-2.

[Access the text alternative for slide images.](#)

Signed Integers

How to represent positive (+) and negative (-)?

With n bits, we have 2^n encodings.

- Use half of the patterns (the ones starting with zero) to represent positive numbers.
- Use the other half (the ones starting with one) to represent negative numbers.

Three different encoding schemes

- Signed-magnitude.
- 1's complement.
- 2's complement.

Signed-Magnitude

Leftmost bit represents the **sign** of the number.

0 = positive

1 = negative

Remaining bits represent the **magnitude** of the number using the binary notation used for unsigned integers.

Examples of 5-bit signed-magnitude integers:

00101 = +5 (sign = 0, magnitude = 0101)

10101 = -5 (sign = 1, magnitude = 0101)

01101 = +13 (sign = 0, magnitude = 1101)

10010 = -2 (sign = 1, magnitude = 0010)

1's complement

To get a negative number, start with a positive number (with zero as the leftmost bit) and flip all the bits -- from 0 to 1, from 1 to 0.

Examples -- 5-bit 1's complement integers:

 00101 (5)
11010 (-5)

 01001 (9)
10110 (-9)

[Access the text alternative for slide images.](#)

Disadvantages of Signed-Magnitude and 1's Complement

In both representations, two different representations of zero

- Signed-magnitude: $00000 = 0$ and $10000 = 0$.
- 1's complement: $00000 = 0$ and $11111 = 0$.

Operations are not simple.

- Think about how to add $+2$ and -3 .
- Actions are different for two positive integers, two negative integers, one positive and one negative.

A simpler scheme: 2's complement

2's Complement

To simplify circuits, we want all operations on integers to use binary arithmetic, regardless of whether the integers are positive or negative.

When we add $+X$ to $-X$ we want to get zero.

Therefore, we use "normal" unsigned binary to represent $+X$.

And we assign to $-X$ the pattern of bits that will make $X + (-X) = 0$.

$$\begin{array}{r} 00101 \quad (5) \\ + 11011 \quad (-5) \\ \hline 00000 \quad (0) \end{array}$$

$$\begin{array}{r} 01001 \quad (9) \\ + 10111 \quad (-9) \\ \hline 00000 \quad (0) \end{array}$$

NOTE: When we do this, we get a carry-out bit of 1, which is ignored.

We only have 5 bits, so the carry-out bit is ignored and we still have 5 bits.

[Access the text alternative for slide images.](#)

2's Complement Integers

4-bit 2's complement	value	4-bit 2's complement	value
0000	0		
0001	1	1111	-1
0010	2	1110	-2
0011	3	1101	-3
0100	4	1100	-4
0101	5	1011	-5
0110	6	1010	-6
0111	7	1001	-7
		1000	-8

With n bits, represent values from -2^{n-1} to $+2^{n-1} - 1$

NOTE: All positive number start with 0, all negative numbers start with 1.

Converting X to -X

1. Flip all the bits. (Same as 1's complement.)
2. Add +1 to the result.

$$\begin{array}{rcl} \text{00101} & (5) \\ \text{11010} & (1\text{'s comp}) \\ + \quad \quad \quad 1 \\ \hline \text{11011} & (-5) \end{array} \qquad \begin{array}{rcl} \text{10111} & (-9) \\ \text{01000} & (1\text{'s comp}) \\ + \quad \quad \quad 1 \\ \hline \text{01001} & (9) \end{array}$$

A shortcut method:

Copy bits from right to left up to (and including) the first '1'.
Then flip remaining bits to the left.

[Access the text alternative for slide images.](#)

Converting Binary (2's C) to Decimal

1. If leading bit is one, take two's complement to get a positive number.
2. Add powers of 2 that have “1” in the corresponding bit positions.
3. If original number was negative, add a minus sign.

$$X = 01101000_{\text{two}}$$

$$= 2^6 + 2^5 + 2^3 = 64 + 32 + 8$$

$$= 104_{\text{ten}}$$

Examples use 8-bit 2's complement numbers.

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

More Examples

$$X = 00100111_{\text{two}}$$

$$= 2^5 + 2^2 + 2^1 + 2^0 = 32 + 4 + 2 + 1$$

$$= 39_{\text{ten}}$$

$$X = 11100110_{\text{two}}$$

$$-X = 00011010$$

$$= 2^4 + 2^3 + 2^1 = 16 + 8 + 2$$

$$= 26_{\text{ten}}$$

$$X = -26_{\text{ten}}$$

<i>n</i>	<i>2ⁿ</i>
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

Examples use 8-bit 2's complement numbers.

Converting Decimal to Binary (2's C)¹

First Method: *Division*

1. Find magnitude of decimal number. (Always positive.)
2. Divide by two – remainder is least significant bit.
3. Keep dividing by two until answer is zero, writing remainders from right to left.
4. Append a zero as the MS bit; if original number was negative, flip bits and add +1.

$$X = 104_{\text{ten}}$$

$$X = 01101000_{\text{two}}$$

$$104 / 2 = 52 \ r0 \quad \text{bit 0}$$

$$52 / 2 = 26 \ r0 \quad \text{bit 1}$$

$$26 / 2 = 13 \ r0 \quad \text{bit 2}$$

$$13 / 2 = 6 \ r1 \quad \text{bit 3}$$

$$6 / 2 = 3 \ r0 \quad \text{bit 4}$$

$$3 / 2 = 1 \ r1 \quad \text{bit 5}$$

$$1 / 2 = 0 \ r1 \quad \text{bit 6}$$

Converting Decimal to Binary (2's C)₂

Second Method: *Subtract Powers of Two*

1. Find magnitude of decimal number.
2. Subtract largest power of two less than or equal to number.
3. Put a one in the corresponding bit position.
4. Keep subtracting until result is zero.
5. Append a zero as MS bit; if original was negative, flip bits and add +1.

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

$$X = 104_{\text{ten}}$$

$$104 - 64 = 40$$

bit 6

$$40 - 32 = 8$$

bit 5

$$8 - 8 = 0$$

bit 3

$$X = 01101000_{\text{two}}$$

Operation: Addition

As discussed, 2's complement addition is just binary addition.

- Assume operands have the same number of bits.
- Ignore carry-out.

$$\begin{array}{r} 01101000 \text{ (104)} \\ + 11110000 \text{ (-16)} \\ \hline 01011000 \text{ (98)} \end{array} \quad \begin{array}{r} 11110110 \text{ (-10)} \\ + 11110111 \text{ (-9)} \\ \hline 11101101 \text{ (-19)} \end{array}$$

[Access the text alternative for slide images.](#)

Operation: Subtraction

You can, of course, do subtraction in base-2, but easier to negate the second operand and add.

Again, assume same number of bits, and ignore carry-out.

$$\begin{array}{rcl} 01101000 & (104) & 11110110 & (-10) \\ - 00010000 & (16) & - \underline{11110111} & (-9) \\ \hline 01101000 & (104) & \underline{11110110} & (-10) \\ + 11110000 & (-16) & + \underline{00001001} & (9) \\ \hline 01011000 & (88) & \underline{11111111} & (-1) \end{array}$$

[Access the text alternative for slide images.](#)

Operation: Sign Extension

To add/subtract, we need both numbers to have the same number of bits.
What if one number is smaller?

Padding on the left with zero does not work:

4-bit

0100 (+4)

1100 (-4)

8-bit

00000100 (still +4)

00001100 (12, not -4)

Instead, replicate the most significant bit (the sign bit):

4-bit

0100 (+4)

1100 (-4)

8-bit

00000100 (still +4)

11111100 (still -4)

[Access the text alternative for slide images.](#)

Overflow

If the numbers are two big, then we cannot represent the sum using the same number of bits.

For 2's complement, this can only happen if both numbers are positive or both numbers are negative.

$$\begin{array}{r} 01000 \quad (8) \\ + 01001 \quad (9) \\ \hline 10001 \quad (-15) \end{array} \qquad \begin{array}{r} 11000 \quad (-8) \\ + 10111 \quad (-9) \\ \hline 01111 \quad (+15) \end{array}$$

How to test for overflow:

1. Signs of both operands are the same, AND.
2. Sign of sum is different.

Another test (easier to perform in hardware): Carry-in to most significant bit position is different than carry-out.

[Access the text alternative for slide images.](#)

Logical Operations: AND, OR, NOT

If we treat each bit as TRUE (1) or FALSE (0), then we define these logical operations, also known as Boolean operations, on a bit.

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

A	NOT A
0	1
1	0

View n -bit number as a collection of n logical values (*bit vector*)

- operation applied to each bit independently.

Examples of Logical Operations

AND

useful for clearing bits

- AND with zero = 0.
- AND with one = no change.

$$\begin{array}{r} \text{AND} \\ \begin{array}{r} 11000101 \\ 00001111 \\ \hline 00000101 \end{array} \end{array}$$

OR

useful for setting bits

- OR with zero = no change.
- OR with one = 1.

$$\begin{array}{r} \text{OR} \\ \begin{array}{r} 11000101 \\ 00001111 \\ \hline 11001111 \end{array} \end{array}$$

NOT

unary operation -- one argument

- flips every bit.

$$\begin{array}{r} \text{NOT} \\ \begin{array}{r} 11000101 \\ \hline 00111010 \end{array} \end{array}$$

[Access the text alternative for slide images.](#)

Logical Operation: Exclusive-OR (XOR)

Another useful logical operation is the XOR.
True if one of the bits is 1, but not both.

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

XOR

useful for selectively flipping bits

- XOR with zero = no change.
- XOR with one = flip.

$$\begin{array}{r} 11000101 \\ \text{XOR } 0000\textcolor{red}{1111} \\ \hline 11001010 \end{array}$$

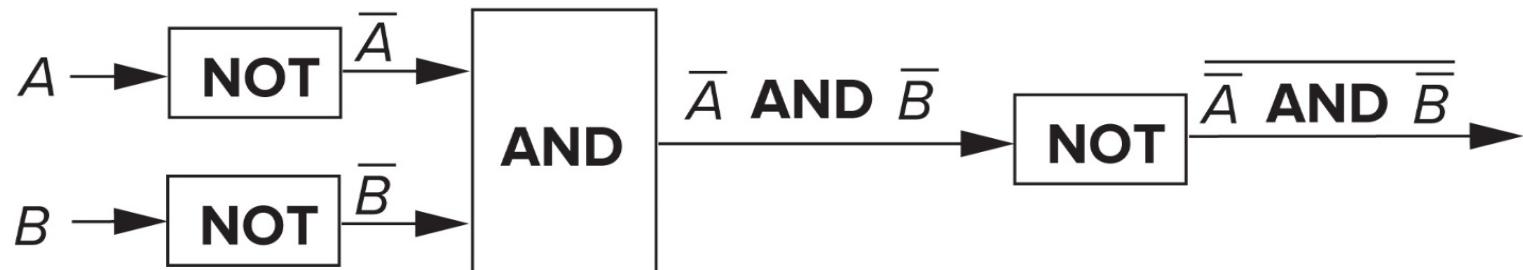
[Access the text alternative for slide images.](#)

DeMorgan's Laws

There's an interesting relationship between AND and OR.

If we NOT two values (A and B), AND them, and then NOT the result, we get the same result as an OR operation. (In the figure below, an overbar denotes the NOT operation.)

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



Here's the truth table to convince you:

Copyright © McGraw Hill Education. Permission required for display

A	B	\bar{A}	\bar{B}	$\bar{A} \text{ AND } \bar{B}$	$\overline{\bar{A} \text{ AND } \bar{B}}$
0	0	1	1	1	0
0	1	1	0	0	1
1	0	0	1	0	1
1	1	0	0	0	1

[Access the text alternative for slide images.](#)

DeMorgan's Laws 2

This means that any OR operation can be written as a combination of AND and NOT.

$$\begin{array}{r} \text{00111010} \\ \text{AND } \underline{\text{11110000}} \\ \text{00110000} \end{array}$$
$$\begin{array}{r} \text{11000101} \\ \text{OR } \underline{\text{00001111}} \\ \text{11001111} \end{array}$$
$$\begin{array}{r} \text{NOT } \underline{\text{00110000}} \\ \text{11001111} \end{array}$$

This is interesting, but is it useful? We'll come back to this in later chapters...

Also, convince yourself that a similar "trick" works to perform AND using only OR and NOT.

[Access the text alternative for slide images.](#)

Hexadecimal Notation: Binary Shorthand

To avoid writing long (error-prone) binary values, group four bits together to make a base-16 digit. Use A to F to represent values 10 to 15.

binary (base 2)	hexadecimal (base 16)	decimal (base 10)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7

binary (base 2)	hexadecimal (base 16)	decimal (base 10)
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	15
1111	F	15

Ex: Hex number **3D6E** easier to communicate than binary
0011110101101110.

Converting from binary to hex

Starting from the right, group every four bits together into a hex digit.
Sign-extend as needed.

0111 0101 0001 1110 1001 1010 111
↓ ↓ ↓ ↓ ↓ ↓ ↓
3 A 8 F 4 D 7

*This is not a new machine representation or data type,
just a convenient way to write the number.*

[Access the text alternative for slide images.](#)

Text: ASCII Characters

ASCII: Maps 128 characters to 7-bit code.

- both printable and non-printable (ESC, DEL, ...) characters

00	nul	10	dle	20	sp	30	0	40	@	50	P	60	`	70	p
01	soh	11	dc1	21	!	31	1	41	A	51	Q	61	a	71	q
02	stx	12	dc2	22	"	32	2	42	B	52	R	62	b	72	r
03	etx	13	dc3	23	#	33	3	43	C	53	S	63	c	73	s
04	eot	14	dc4	24	\$	34	4	44	D	54	T	64	d	74	t
05	enq	15	nak	25	%	35	5	45	E	55	U	65	e	75	u
06	ack	16	syn	26	&	36	6	46	F	56	V	66	f	76	v
07	bel	17	etb	27	'	37	7	47	G	57	W	67	g	77	w
08	bs	18	can	28	(38	8	48	H	58	X	68	h	78	x
09	ht	19	em	29)	39	9	49	I	59	Y	69	i	79	y
0a	nl	1a	sub	2a	*	3a	:	4a	J	5a	Z	6a	j	7a	z
0b	vt	1b	esc	2b	+	3b	;	4b	K	5b	[6b	k	7b	{
0c	np	1c	fs	2c	,	3c	<	4c	L	5c	\	6c	l	7c	
0d	cr	1d	gs	2d	-	3d	=	4d	M	5d]	6d	m	7d	}
0e	so	1e	rs	2e	.	3e	>	4e	N	5e	^	6e	n	7e	~
0f	si	1f	us	2f	/	3f	?	4f	O	5f	_	6f	o	7f	del

Interesting Properties of ASCII Code

What is relationship between a decimal digit ('0', '1', ...) and its ASCII code?

What is the difference between an upper-case letter ('A', 'B', ...) and its lower-case equivalent ('a', 'b', ...)?

Given two ASCII characters, how do we tell which comes first in alphabetical order?

**Are 128 characters enough?
(<http://www.unicode.org/>)**

No new operations -- integer arithmetic and logic.

Fractions: Fixed-Point Binary

We use a "binary point" to separate integer bits from fractional bits, just like we use the "decimal point" for decimal numbers.

Two's complement arithmetic still works the same.

$$\begin{array}{r} & \begin{array}{l} 2^{-1} = 0.5 \\ 2^{-2} = 0.25 \\ 2^{-3} = 0.125 \end{array} \\ & \downarrow \quad \downarrow \quad \downarrow \\ 00101000.101 & (40.625) \\ + \underline{11111110.110} & (-1.25) \\ \hline 00100111.011 & (39.375) \end{array}$$

NOTE: In a computer, the binary point is implicit -- it is not explicitly represented. We just interpret the values appropriately.

[Access the text alternative for slide images.](#)

Converting Decimal Fraction to Binary (2's C)

1. Multiply fraction by 2.
2. Record one's digit of result, then subtract it from the decimal number.
3. Continue until you get 0.0000... or you have used the desired number of bits (in which case you have approximated the desired value).

$$X = 0.421_{\text{ten}}$$

$0.421 \times 2 = 0.842$	bit -1 = 0
$0.842 \times 2 = 1.684$	bit -2 = 1
$0.684 \times 2 = 1.368$	bit -3 = 1
$0.368 \times 2 = 0.736$	bit -4 = 0
until 0, or until no more bits...	

$$X = 0.0110_{\text{two}} \text{ (if limited to 4 fractional bits)}$$

Very Large or Very Small Numbers: Floating-Point

Large values: 6.023×10^{23} -- requires 79 bits

Small values: 6.626×10^{-34} -- requires >110 bits

- Range requires lots of bits, but only four decimal digits of precision.

Use a different encoding for the equivalent of “scientific notation”: $F \times 2^E$

Need to represent F (*fraction*), E (*exponent*), and sign.

IEEE 754 Floating-Point Standard (32-bits):



$$N = (-1)^S \times 1.\text{fraction} \times 2^{\text{exponent}-127}, \quad 1 \leq \text{exponent} \leq 254$$

$$N = (-1)^S \times 0.\text{fraction} \times 2^{-126}, \quad \text{exponent} = 0$$

[Access the text alternative for slide images.](#)

Floating-point Example

Single-precision IEEE floating point number:

- Sign is 1 – number is negative.
 - Exponent field is 01111110 = **126** (decimal).
 - Fraction is 0.10000000000... = **0.5** (decimal).

$$\text{Value} = -1.5 \times 2^{(126-127)} = -1.5 \times 2^{-1} = -0.75.$$

[Access the text alternative for slide images.](#)

Converting from Decimal to Floating-Point Binary

How do we represent $-6 \frac{5}{8}$ as a floating-point binary number?

Ignoring sign, represent as a binary fraction: 0110.101

Next, normalize the number, by moving the binary point to the right of the most significant bit: $0110.101 = 1.10101 \times 2^2$

Exponent field = $127 + 2 = 129 = 10000001$

Fraction field = everything to the right of the binary point: 101010000....

Sign field is 1 (because the number is negative).

Answer:

11000000110101000000000000000000

Special Values

Infinity

- If exponent field is 11111111, the number represents **infinity**.
- Can be positive or negative (per sign bit).

Subnormal

- If exponent field is 00000000, the number is not normalized. This means that the implicit 1 is not present before the binary point. The exponent is -126.
- $N = (-1)^s \times 0.\text{fraction} \times 2^{-126}$.
- Extends the range into very small numbers.
- Example: 00000000000010000000000000000000 .

$$0.00001_{\text{two}} \times 2^{-126} = 2^{-131}$$



Because learning changes everything.[®]

www.mheducation.com

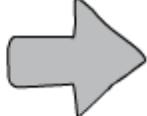
From Bits and Gates to C and Beyond

The von Neumann Model

Chapter 4

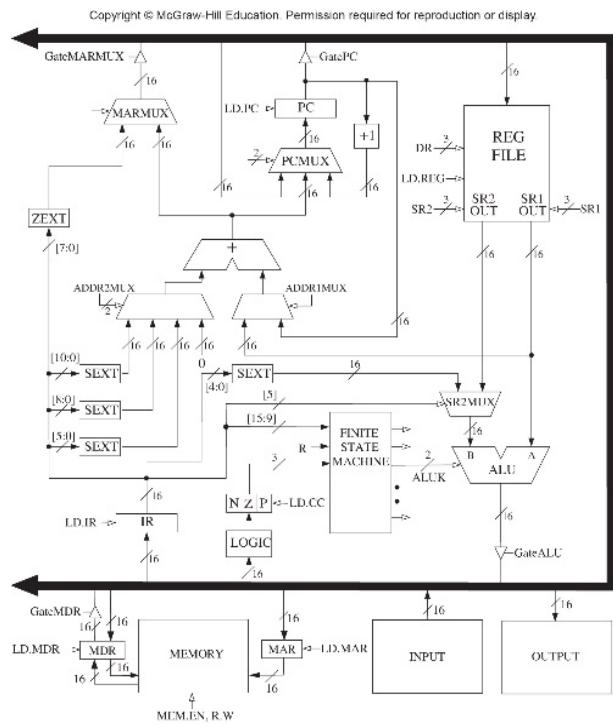
Solving a Problem using a Computer

Compiler translates
into a sequence
of instructions (binary)



```
0010100110110111  
1010011101011001  
0100111010001010  
0111111100101001  
1101100010110110  
0100111011010101
```

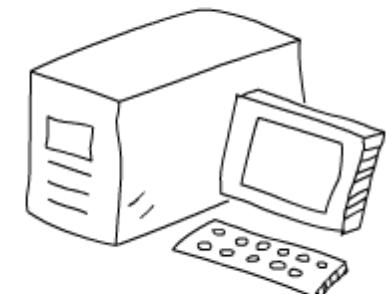
Programmer writes
a program



Instructions stored
in memory of computer

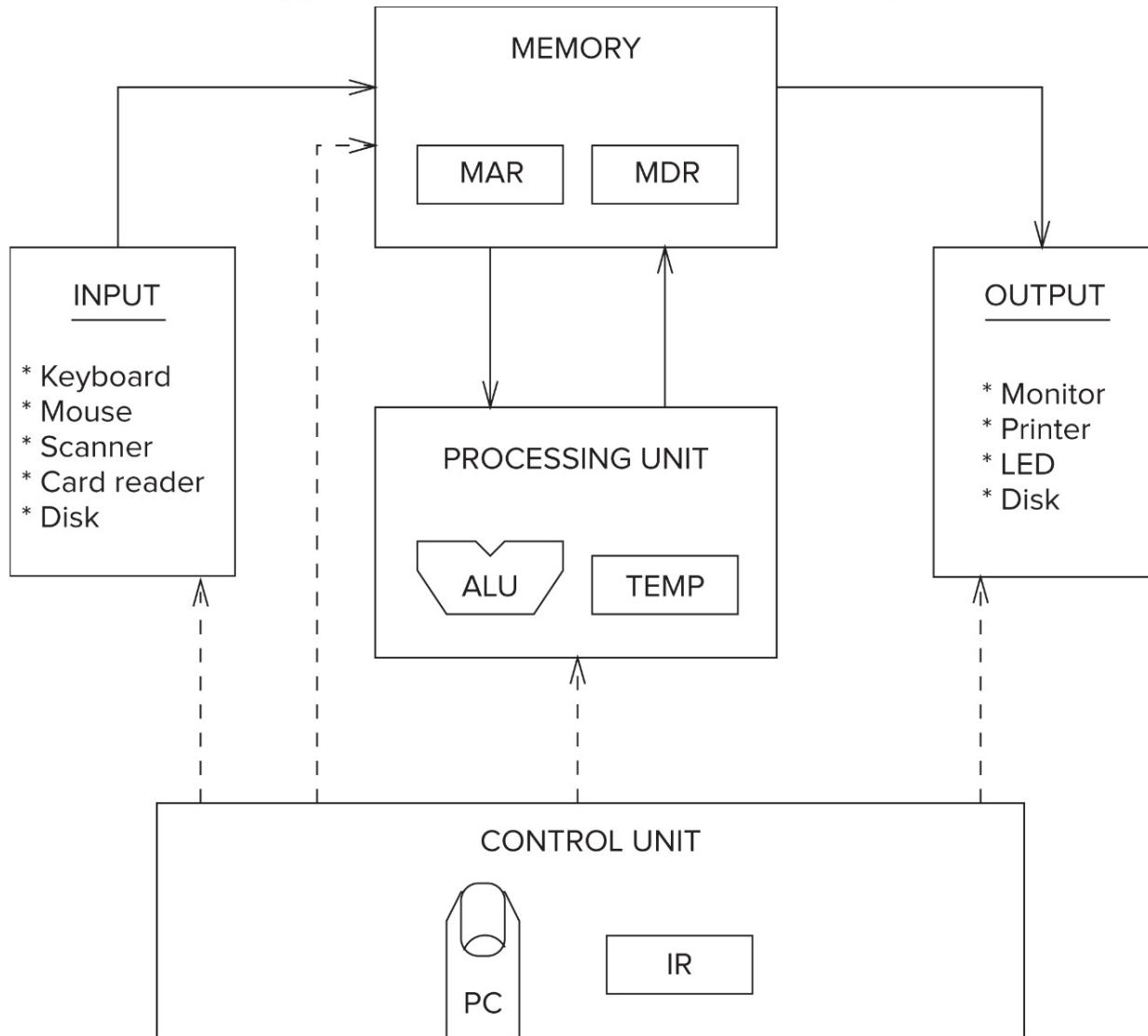


CPU hardware
executes instructions



Computer Organization: von Neumann Model

Copyright © McGraw-Hill Education. Permission required for reproduction or display.

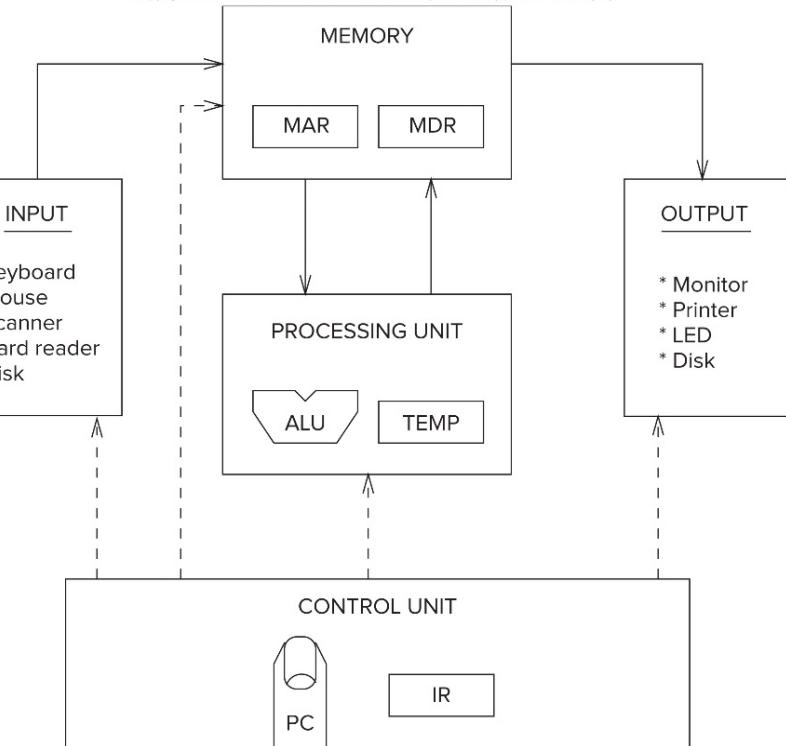


Fundamental model
of a computer
proposed by
John von Neumann
in 1946

Computer Organization: von Neumann Model

- **Memory**: stores data and instructions during program execution
- **Processing Unit**: transforms data -- logic circuits, temporary storage
- **Control Unit**: orchestrates fetching of instructions (from memory) and execution of instructions (in processing unit)
- **Input**: receives data from external environment
- **Output**: sends data to external environment

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



Processing Unit + Control Unit = **CPU**: Central Processing Unit.

Memory

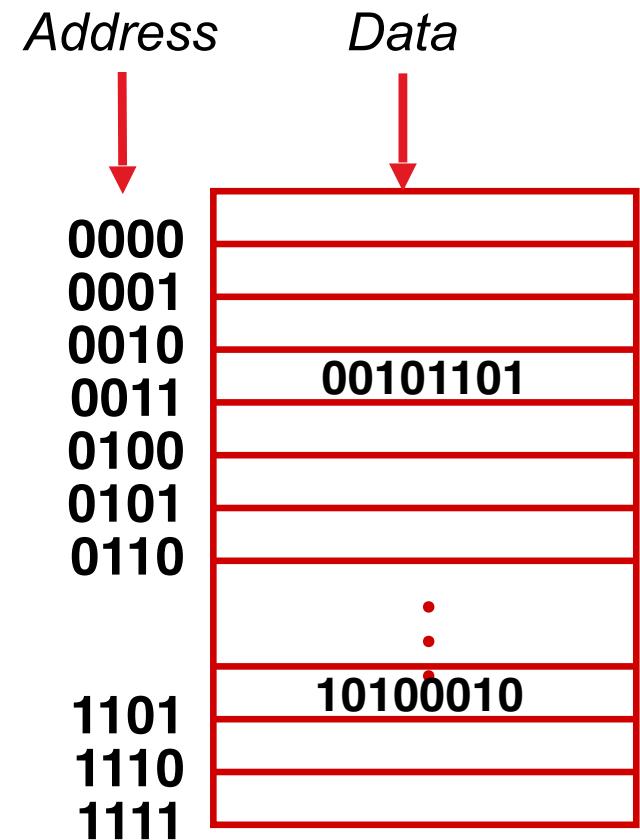
Address space = number of locations

2^n locations means n -bit address.

Addressability = number of bits in each location

Basic operations

- LOAD data from memory to CPU
- STORE data from CPU to memory



Interface to Memory

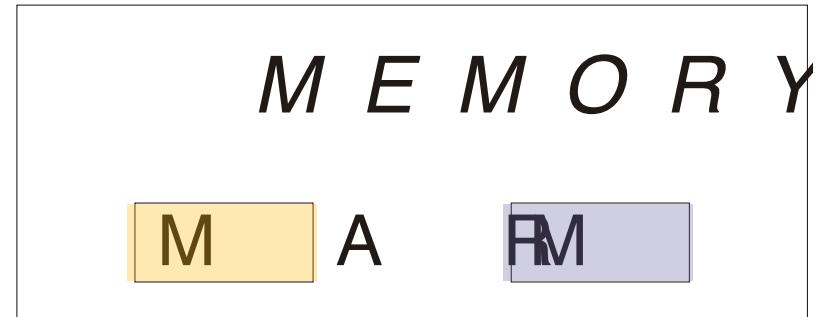
Registers used to move data into and out of memory

MAR = Memory Address Register

MDR = Memory Data Register

To **LOAD** data from memory:

1. Write the **address** into the **MAR**
2. Send a "read" signal to memory
3. Read data from **MDR**



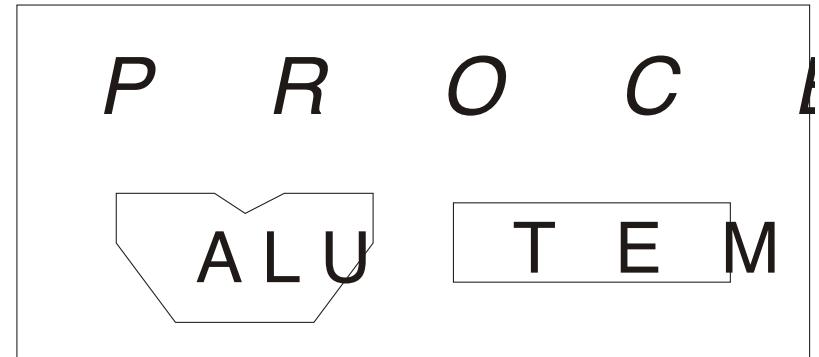
To **STORE** data to memory:

1. Write the **address** into the **MAR**
2. Write the **data** into **MDR**
3. Send a "write" signal to memory

Processing Unit

Functional Units

- Perform data transformations
- ALU = Arithmetic / Logic Unit
add, subtract, AND, ...



Registers

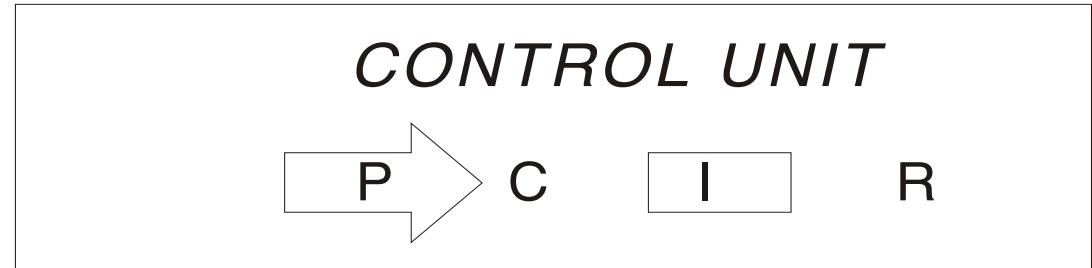
- Temporary storage of data

Word Length

- ALU operations are usually performed on words
- Typical word length:
64 bits (Intel Core), 32-bit (Intel Atom), 16 bits (LC-3)

Control Unit

Orchestrates the rest of the computer to carry out program instructions



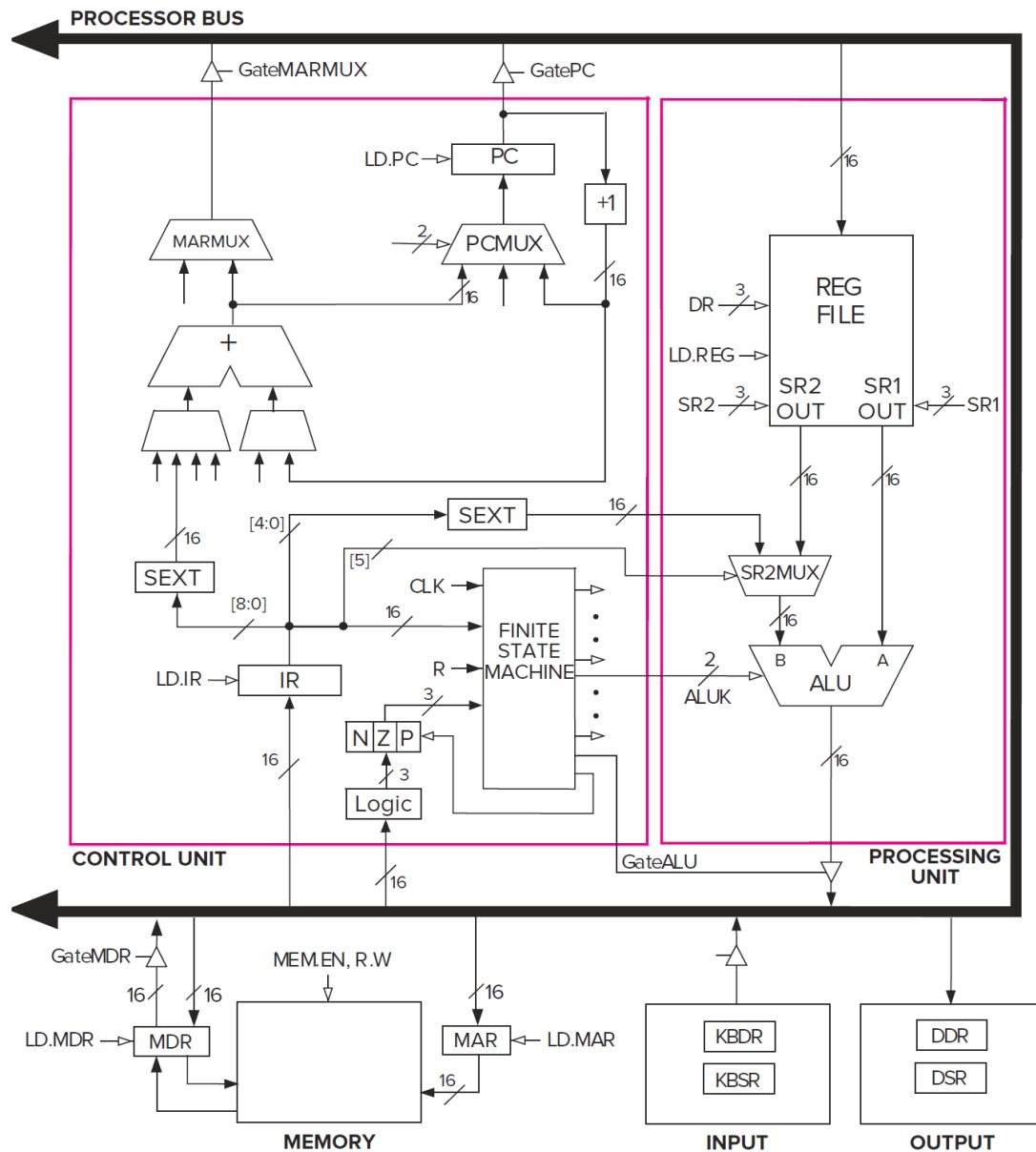
Instruction Register (IR) holds the current instruction.

Program Counter (PC) holds the memory address of the next instruction to be executed. It is known as a "pointer" because it points to a memory location.

Control unit is a state machine that does the following:

1. Read (fetch) the next instruction from memory. (Address is in the PC.)
2. Sends control signals to other parts of processor to move / transform data according to the instruction.
3. Go to step 1, repeat forever...

Example Processor: LC-3



Memory = 2^{16} locations,
16 bits in each location

ALU: performs ADD, AND, NOT
word length is 16 bits

Instruction length is 16 bits

Instruction

A computer **instruction** is a binary value that specifies one operation to be carried out by the hardware. It's binary, so it looks just like the data in Chapter 2 -- a collection of bits. The components of the instruction are *encoded* as various bit fields of the binary value.

Instruction is the **fundamental unit of work**. Once an instruction begins, it will complete its job.

Two components of an instruction:

opcode specifies the operation to be performed (e.g., ADD)

operands specifies the data to be used during the operation

The set of instructions and their formats is called the **Instruction Set Architecture (ISA)** of a computer.

LC-3 Instruction

LC-3 has a four-bit opcode, bits [15:12] -- up to 16 different operations.

LC-3 has eight registers in the CPU. Registers are used as operands for instructions -- each operand requires 3 bits to specify which register.

Example: ADD instruction, opcode = 0001

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD		Dst		Src1		0	0	0		Src2					

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	0	1	0	0	0	0	1	1	0

“Add the contents of R2 (010) to the contents of R6 (110), and store the result in R6 (110).”

Types of Instructions

Instructions are classified into different types, depending on what sort of operation is specified.

Operate

Will perform some sort of data transformation

Examples: add, AND, NOT, multiply, shift

Data Movement

Move data from memory to a register in the CPU, or

Move data from a register to a memory location

Control

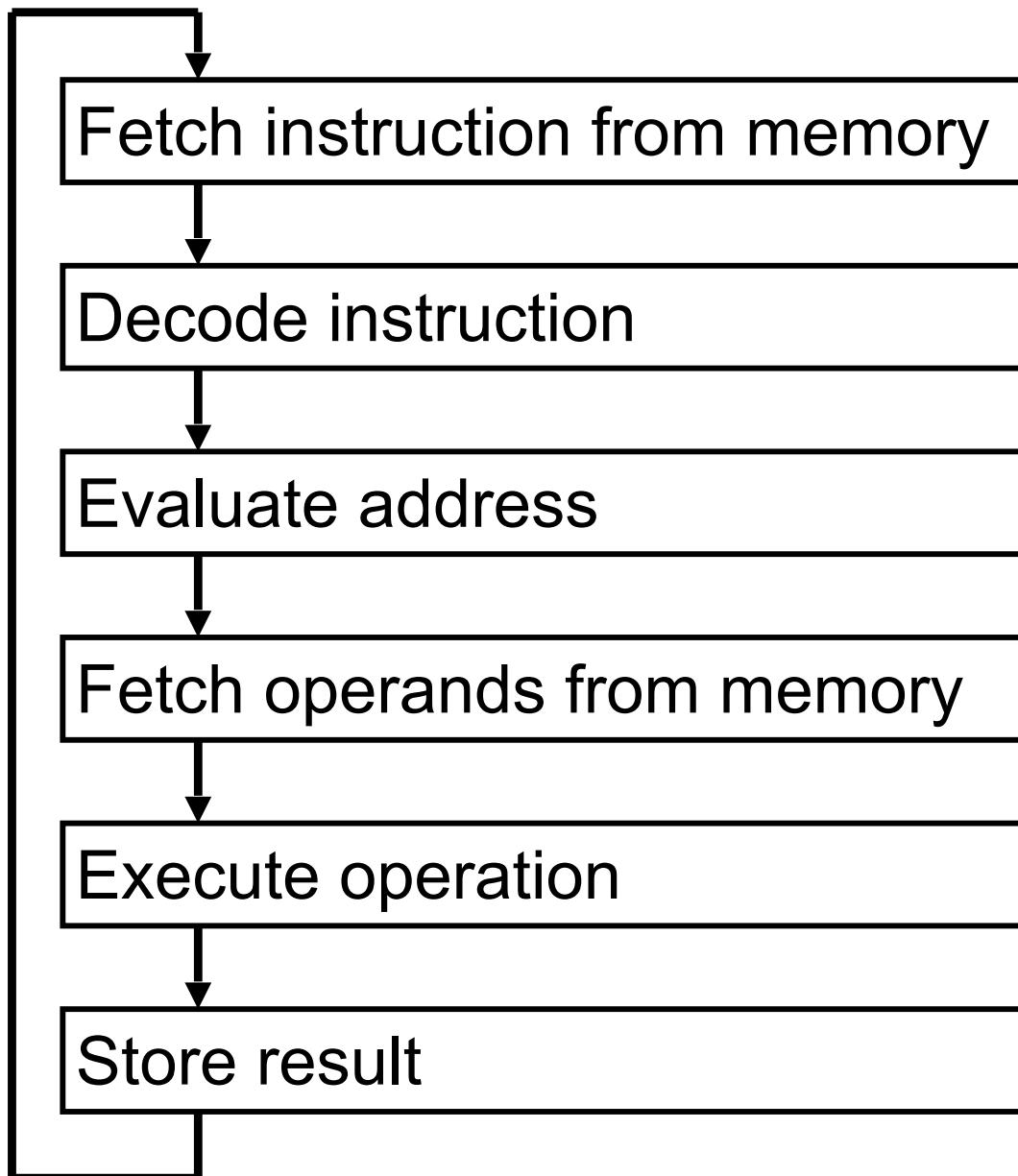
Determine the next instruction to be executed

LC-3 Instruction Set

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD ⁺	0001			DR			SR1		0	00			SR2			
ADD ⁺	0001			DR			SR1		1				imm5			
AND ⁺	0101			DR			SR1		0	00			SR2			
AND ⁺	0101			DR			SR1		1				imm5			
BR	0000	n	z	p									PCoffset9			
JMP	1100		000		BaseR								0000000			
JSR	0100	1					PCoffset11									
JSRR	0100	0	00		BaseR								0000000			
LD ⁺	0010		DR										PCoffset9			
LDI ⁺	1010		DR										PCoffset9			

LDR ⁺	0110	DR	BaseR	offset6
LEA	1110	DR		PCoffset9
NOT ⁺	1001	DR	SR	111111
RET	1100	000	111	000000
RTI	1000			0000000000000000
ST	0011	SR		PCoffset9
STI	1011	SR		PCoffset9
STR	0111	SR	BaseR	offset6
TRAP	1111	0000		trapvect8
reserved	1101			

Processing an Instruction



The diagram shows the **instruction cycle** -- the set of steps that must be performed to execute each instruction.

The Control Unit is responsible for performing this sequence of actions.

Each step may require one or more clock cycles.

Instruction Processing: Fetch

Fetch

- PC contains the address of the instruction to be fetched.
- The control unit writes the PC value into the MAR to read the instruction stored at that location.
- It also **increments the PC** to point to the next consecutive memory location. (It is assumed that the next instruction is stored in the next location in memory. This could be changed when the instruction is executed -- see "control instructions" later in this chapter.)
- When the instruction is retrieved from memory (via the MDR), it is copied into the instruction register (IR).

Instruction Processing: Decode

Decode

- Once the instruction is in the IR, the control unit "looks at" the opcode to determine what operation should be performed.
- One possibility: a 4-bit decoder sets a particular control signal to 1, corresponding to the desired operation.
- The opcode also determines how the other instruction bits should be interpreted.
 - E.g., for ADD, bits [8:6] specify a register to be used as the first operand of the addition.

Instruction Processing: Executing the Instruction

Evaluate Address

For memory instructions, determine which address to load/store.

Fetch Operands

Get data values from registers and/or memory.

Execute

Perform the necessary operations (e.g., add).

Store Result

Write the result of the computation to a register or to memory.

Not every step is required for every instruction. For example, if there is no memory access, then the Evaluate Address step is not performed.

Changing the Sequence of Instructions

In the Fetch step, we incremented the PC to point to the next instruction. Sometimes, we want to execute a different instruction, not the next one in memory. This is the job of a **control** instruction.

A control instruction may change the PC value. In other words, instead of pointing to the next place in memory, a new address can be calculated and stored in the PC register.

This change may be conditional -- e.g., if the last value calculated was zero, write a new value to the PC. These are typically called **branch instructions**.

The change may be unconditional -- it always happens. These instructions are usually called **jump instructions**.

LC-3 control instructions will be explained in Chapter 5.

Control of the Instruction Cycle

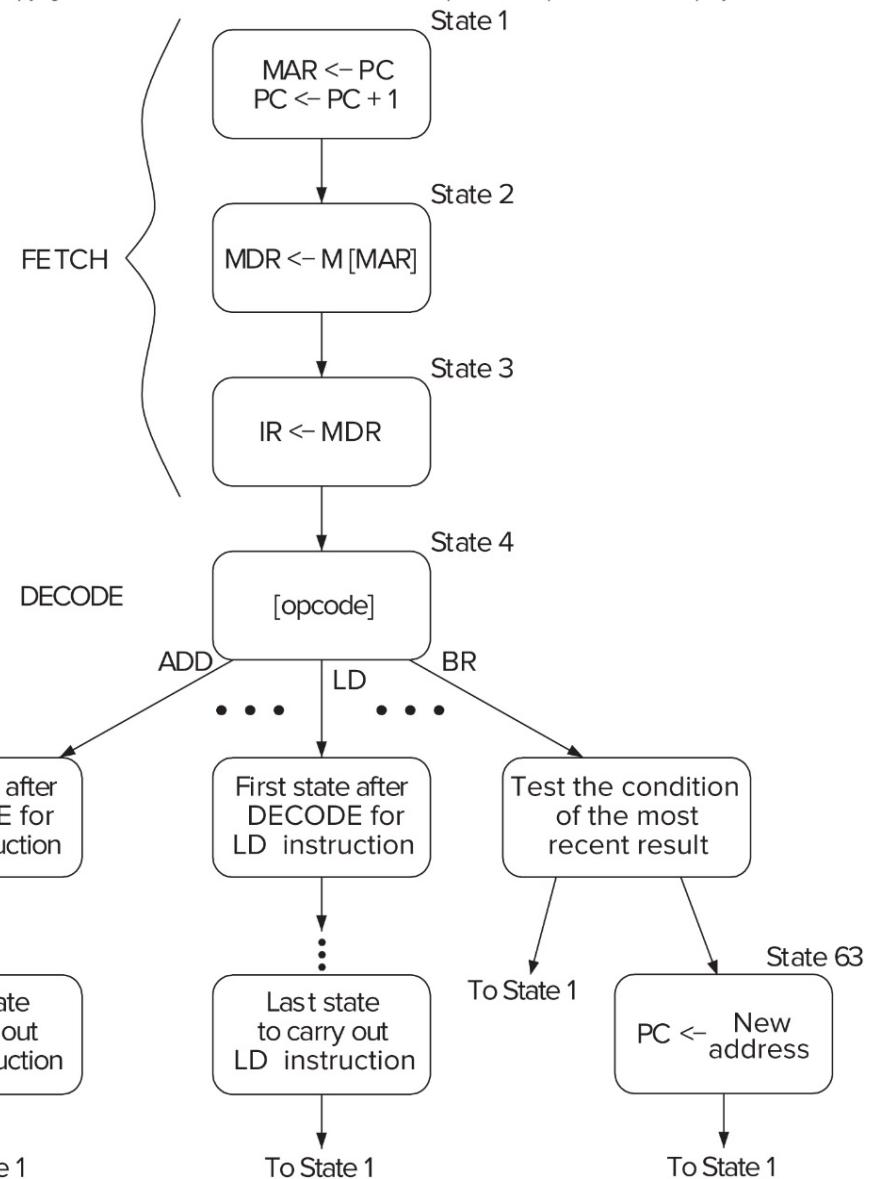
Copyright © McGraw-Hill Education. Permission required for reproduction or display.

The control unit is a state machine.
A partial state diagram is shown below.

The instruction cycle starts with State 1, which initiates the Fetch phase. This is done for every instruction.

After the Decode phase (State 4), the states will depend on the specific opcode.

Appendix C has a complete description of the LC-3 state machine.

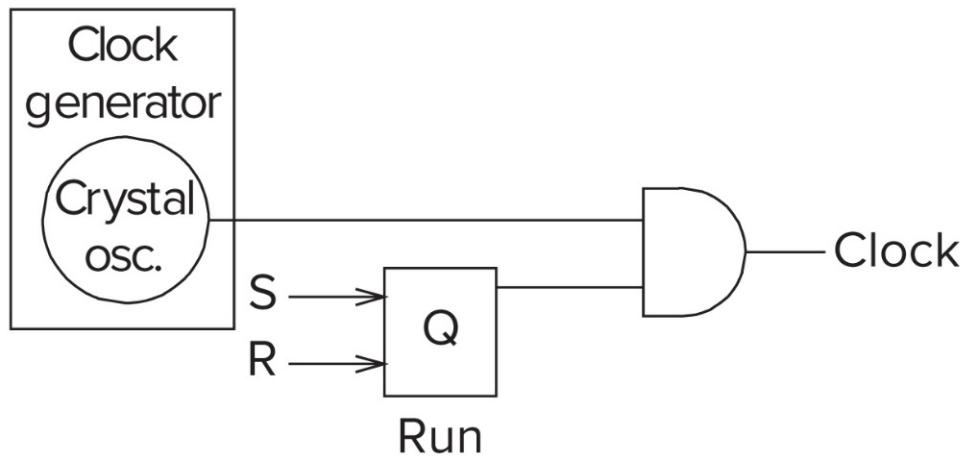


Stopping Execution

The control unit's state machines keeps running, as long as the clock signal keeps going up and down.

The program can halt execution by turning off the clock signal.

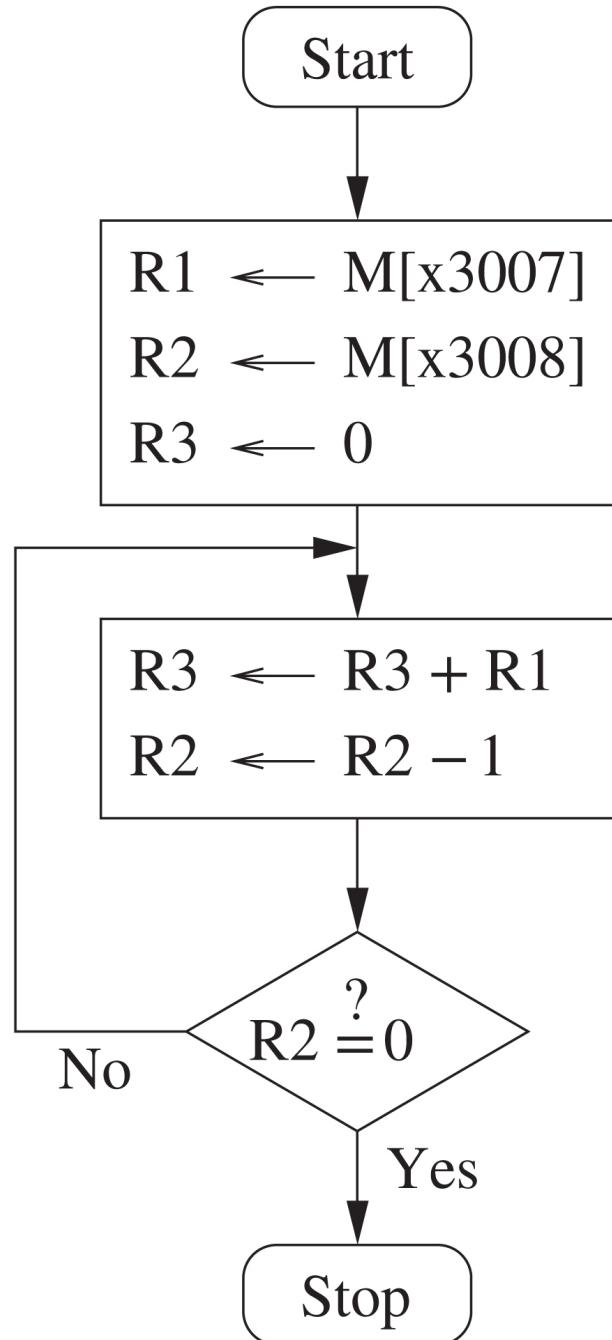
Copyright © McGraw-Hill Education. Permission required for reproduction or display.



In the circuit above, we "gate" the clock signal using a bit stored in the latch. If the control unit sets that bit to zero, the clock will stop.

In some machines, this is done by a HALT instruction. In LC-3 (and others), the latch is part of a special register controlled by the operating system.

Multiplication Algorithm



Multiplication Algorithm

Copyright © McGraw-Hill Education. Permission required for reproduction or display.

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x3000	0	0	1	0	0	0	1	0	0	0	0	0	0	1	1	0
x3001	0	0	1	0	0	1	0	0	0	0	0	0	0	1	1	0
x3002	0	1	0	1	0	1	1	0	1	1	1	0	0	0	0	0
x3003	0	0	0	1	0	1	1	0	1	1	0	0	0	0	1	
x3004	0	0	0	1	0	1	0	0	1	0	1	1	1	1	1	1
x3005	0	0	0	0	1	0	1	1	1	1	1	1	1	1	0	1
x3006	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1
x3007	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
x3008	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

R1 <- M[x3007]
R2 <- M[x3008]
R3 <- 0
R3 <- R3+R1
R2 <- R2-1
BR not-zero M[x3003]
HALT
The value 5
The value 4

Table for Question 4.5

Address	Data
0000	0001 1110 0100 0011
0001	1111 0000 0010 0101
0010	0110 1111 0000 0001
0011	0000 0000 0000 0000
0100	0000 0000 0110 0101
0101	0000 0000 0000 0110
0110	1111 1110 1101 0011
0111	0000 0110 1101 1001

Chapter Summary

Computer program is specified by a sequence of instructions, stored in the computer's memory.

Each instruction is a bit pattern that specifies an operation to be performed. Different processors encode this information in different ways.

Combinational and sequential logic is used to retrieve instructions and data from memory, transform data according to the instructions, and write the results back to memory.

The control unit is a state machine, driven by a clock signal. It uses the Program Counter (PC) to step through the instructions in memory, fetching and executing each instruction.

All of this machinery is implemented using the digital logic structures discussed in Chapter 3.



Because learning changes everything.[®]

www.mheducation.com

From Bits and Gates to C and Beyond

The LC-3

Chapter 5

LC-3 Instruction Set Architecture

The Instruction Set Architecture (ISA) specifies everything about the computer that is visible to the programmer.

Memory Organization

- 2^{16} memory locations, with 16 bits in each location.

Registers

- 8 general purpose registers, each 16 bits, called R0 - R7.
- Used as source and destination locations for instruction operands.

Instructions

- 15 opcodes, each specified by a 4-bit code.
- Only one data type: 16-bit 2's complement integer.
- Addressing modes: various encodings to specify location of operands.

Condition Codes

- Three 1-bit registers (N, Z, P) that are set whenever a value is written to a register. These are used for conditional branch instructions. N = negative, Z = zero, P = positive.

LC-3 Instruction Set

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD ⁺	0001		DR		SR1	0	00			SR2						
ADD ⁺	0001		DR		SR1	1				imm5						
AND ⁺	0101		DR		SR1	0	00			SR2						
AND ⁺	0101		DR		SR1	1				imm5						
BR	0000	n	z	p						PCoffset9						
JMP	1100		000		BaseR					000000						
JSR	0100	1				PCoffset11										
JSRR	0100	0	00		BaseR					000000						
LD ⁺	0010		DR			PCoffset9										
LDI ⁺	1010		DR				1	PCoffset9								
LDR ⁺	0110		DR		BaseR					offset6						
LEA	1110		DR							PCoffset9						
NOT ⁺	1001		DR		SR					111111						
RET	1100		000		111					000000						
RTI	1000									00000000000000						
ST	0011		SR			PCoffset9										
STI	1011		SR			PCoffset9										
STR	0111		SR		BaseR					offset6						
TRAP	1111		0000							trapvect8						
reserved	1101															

[Access the text alternative for slide images.](#)

LC-3 Instruction Set: Notation

Each opcode has a 4-bit encoding and one or more mnemonic symbols. For example, the ADD instruction has opcode 0001, but we typically just say "ADD". The mnemonic symbols are used also used when we program in assembly language (Chapter 7).

Some opcodes appear twice, because they have different addressing modes -- different ways to specify the operands. (Sometimes, the different addressing modes use different mnemonic symbols.)

Some abbreviations are defined below, and will be explained when we discuss the relevant instructions:

SR = source register

DR = destination register

BaseR = base register

immN = immediate value, N bits

offsetN, PCoffsetN = offset to be added to the BaseR or PC, N bits

trapvector8 = index into the TRAP table, 8 bits

Operate Instructions

ADD

Add two integers and write the result to a register.

Set condition codes.

AND

Perform a bitwise AND operation on two integers, and write the result to a register.

Set condition codes.

NOT

Perform a bitwise NOT on an integer, and write the result to a register.

Set condition codes.

LEA

Add an offset to the PC and write the result to a register.

Addressing Modes

Register

First operand is always specified by a register.

Second operand is a register if IR[5] = 0.

Immediate

If IR[5] = 1, the second operand is specified in the instruction.

IR[4:0] is a 5-bit 2's complement integer, sign-extended to 16 bits.

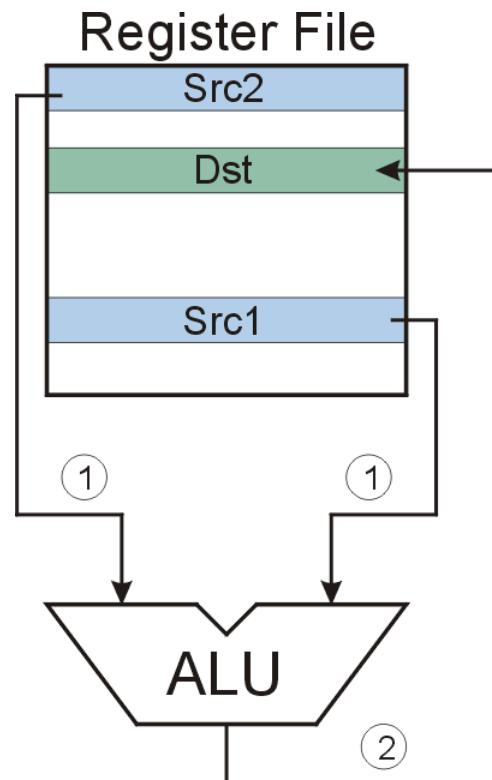
ADD/AND (Register mode)

this zero means “register mode”

ADD	0 0 0 1	Dst	Src1	0 0 0	Src2

AND	0 1 0 1	Dst	Src1	0 0 0	Src2

Sets condition codes.



[Access the text alternative for slide images.](#)

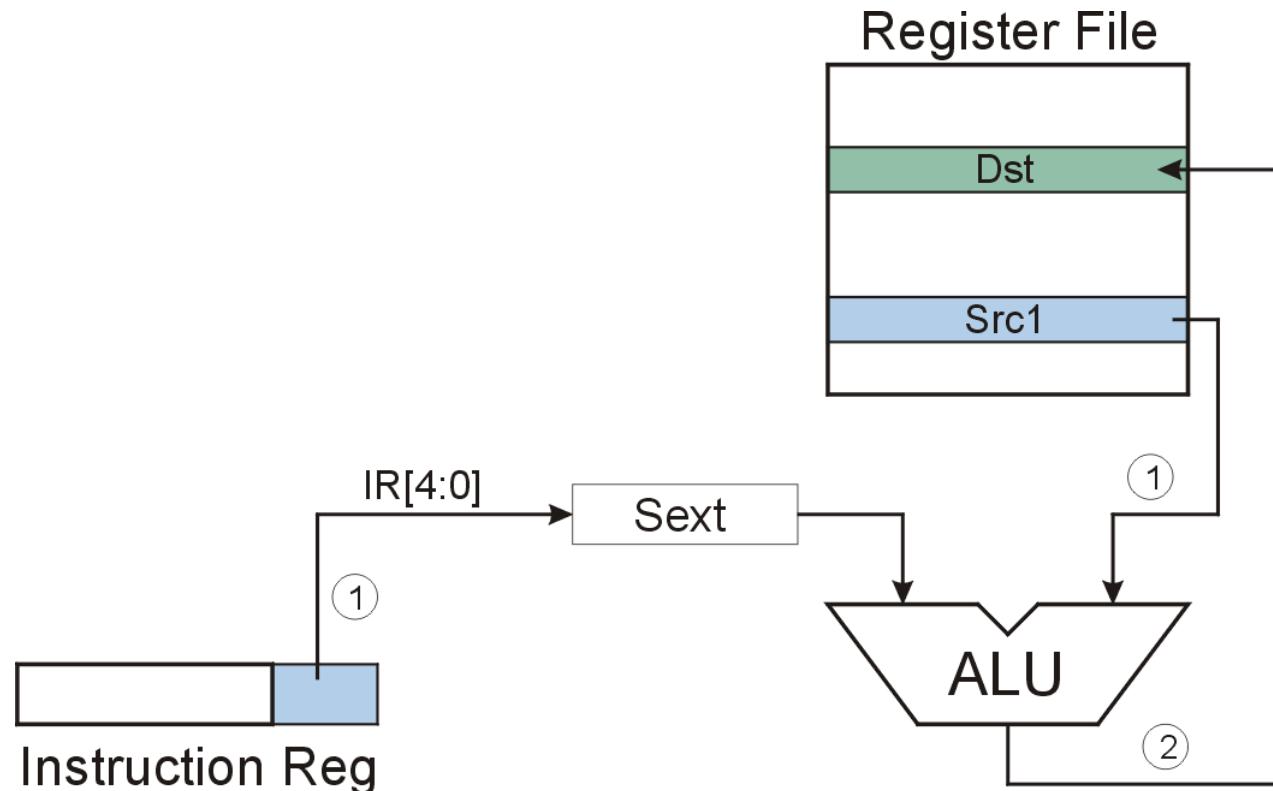
ADD/AND (Immediate mode)

this one means “immediate mode”

ADD	0 0 0 1	Dst	Src1	1	Imm5
-----	---------	-----	------	---	------

AND	0 1 0 1	Dst	Src1	1	Imm5
-----	---------	-----	------	---	------

Sets condition codes.

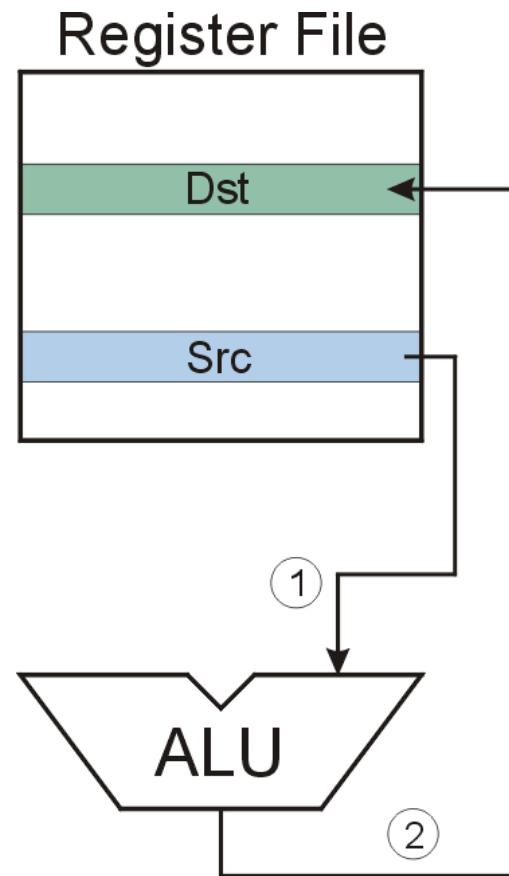


[Access the text alternative for slide images.](#)

NOT (Register mode)



Sets condition codes.



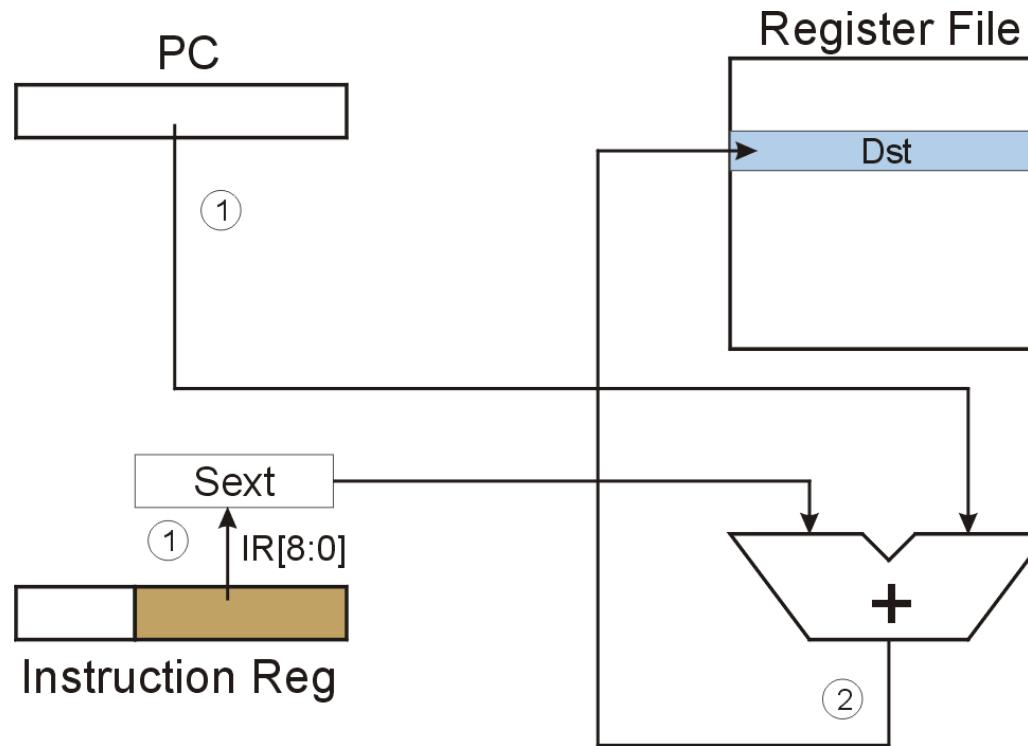
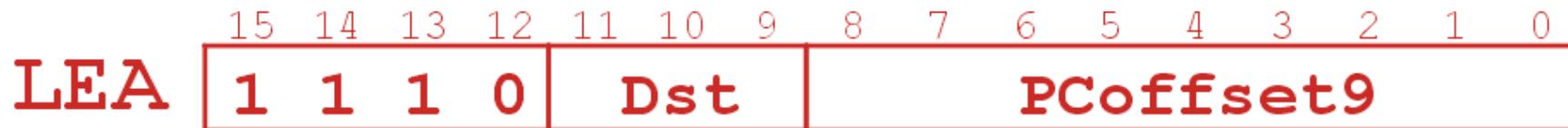
Only ADD, AND, and NOT?

It might seem very limiting to only provide these three operations, but anything else can be built from these.

- Negate -- Flip bits with NOT, then ADD +1.
- Subtract -- Negate and ADD.
- Multiply -- Repeated additions.
- OR -- Use DeMorgan's Law (see Chapter 2).
- Clear -- AND with 0 to set register to zero.
- Copy -- ADD 0 to copy from one register (src) to another (dst).
- Others: XOR, divide, shift left, ...

LEA

Computes an address by adding sign-extended IR[8:0] to PC, and write the result to a register. (Does not set condition codes.)



[Access the text alternative for slide images.](#)

Data Movement Instructions: Load and Store

Used to move data between memory and registers.

LOAD = from memory to register (and set condition codes)

STORE = from register to memory

Three variants of each, with different addressing modes:

PC-Relative	Compute address by adding an offset IR[8:0] to the PC.
Indirect	Load an address from a memory location.
Base+Offset	Compute address by adding an offset IR[5:0] to a register.

PC-Relative Addressing Mode

Want to specify address directly in the instruction.

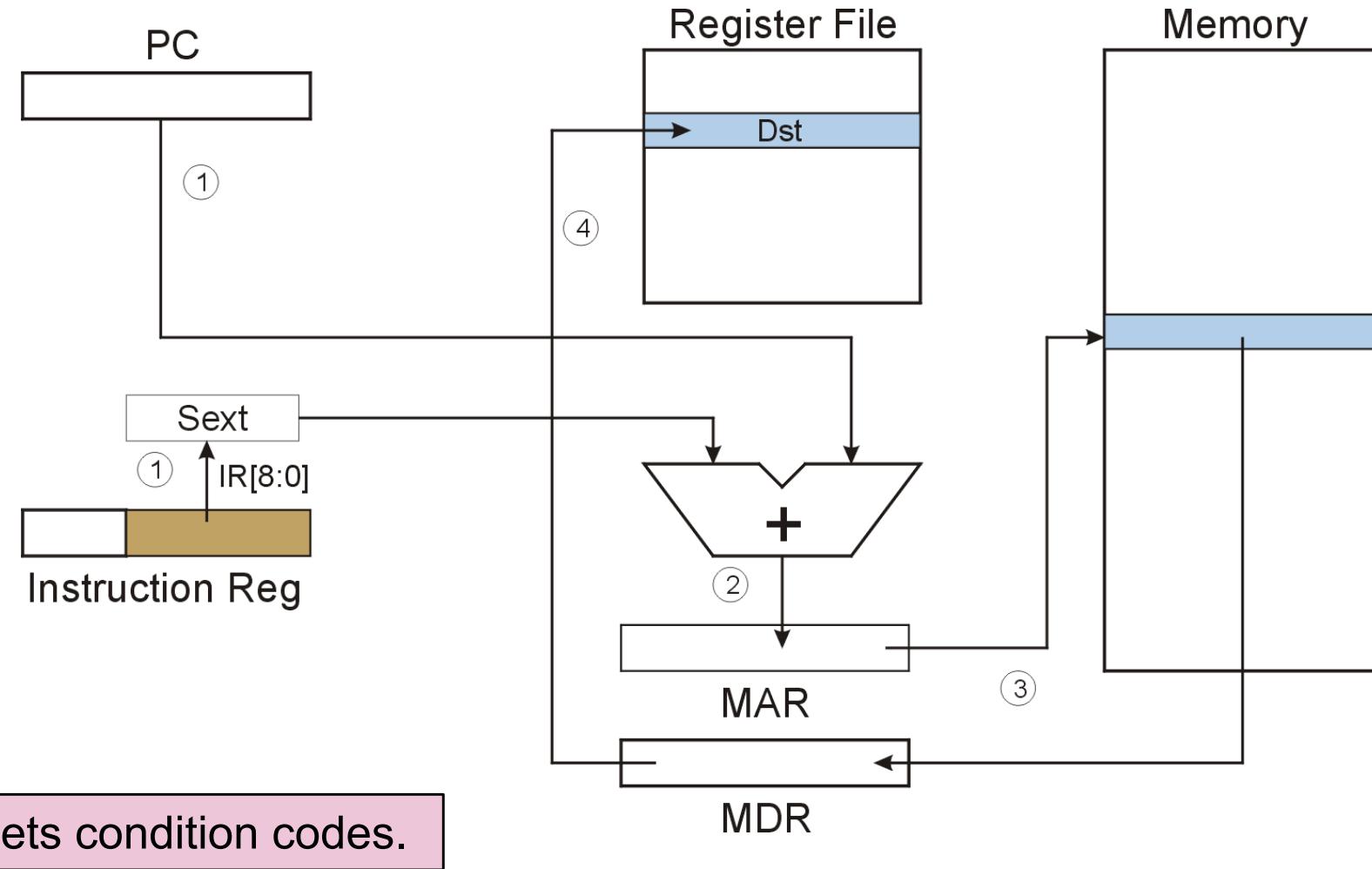
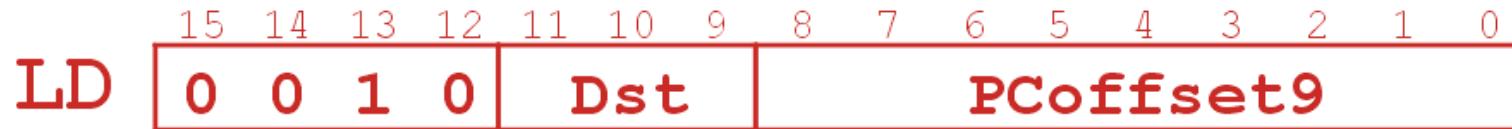
- But an address is 16 bits, and so is an instruction!
- After subtracting 4 bits for opcode and 3 bits for register, we have 9 bits available for address.

Solution: Use the 9 bits as a *signed offset* from the current PC.

Can form any address X, such that: $PC - 256 \leq X \leq PC + 255$

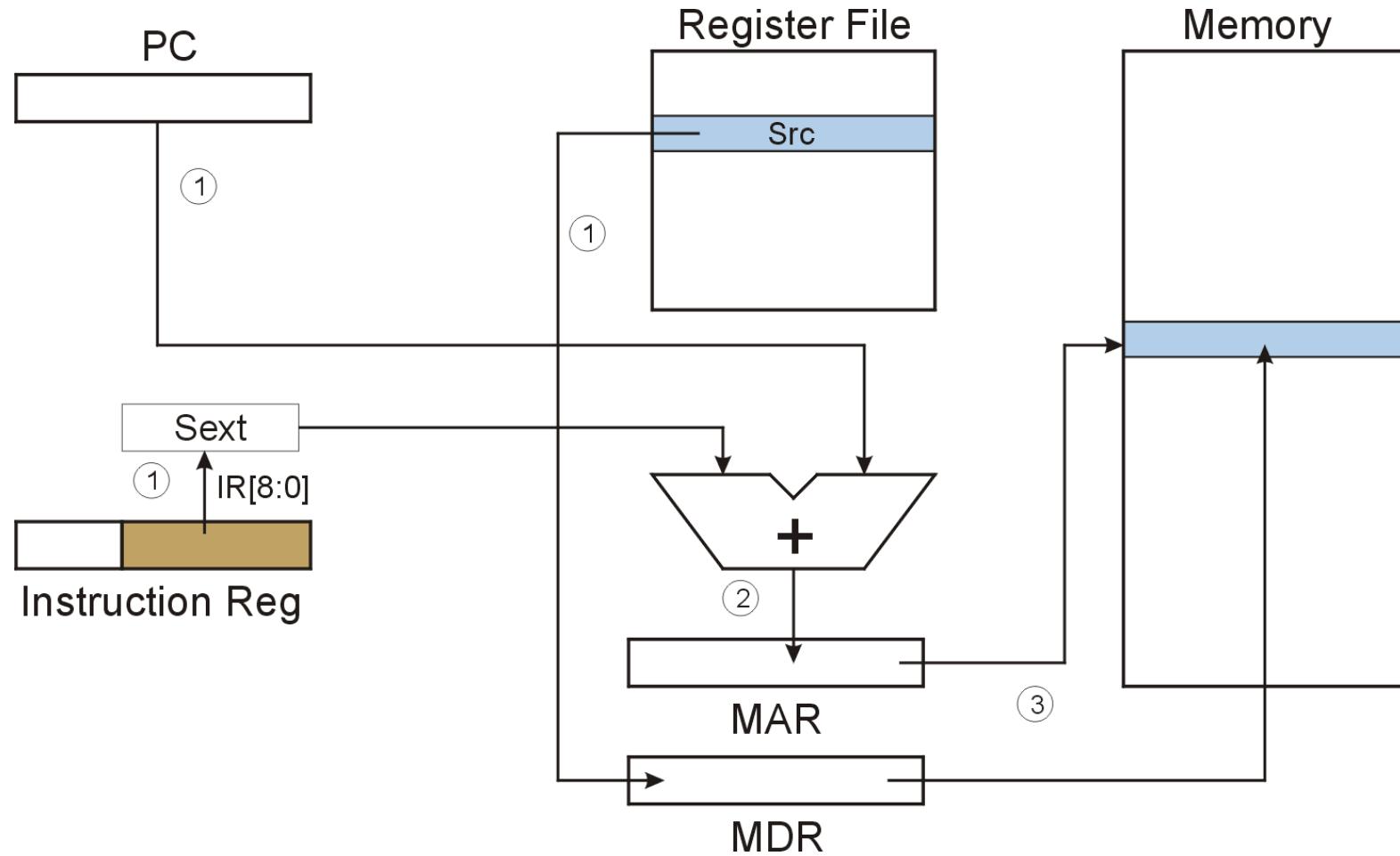
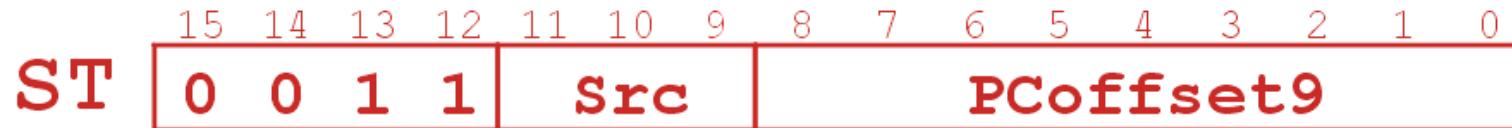
Remember that PC is incremented as part of the FETCH phase;
This is done before the EVALUATE ADDRESS stage.

LD (PC-Relative mode)



[Access the text alternative for slide images.](#)

ST (PC-Relative mode)



[Access the text alternative for slide images.](#)

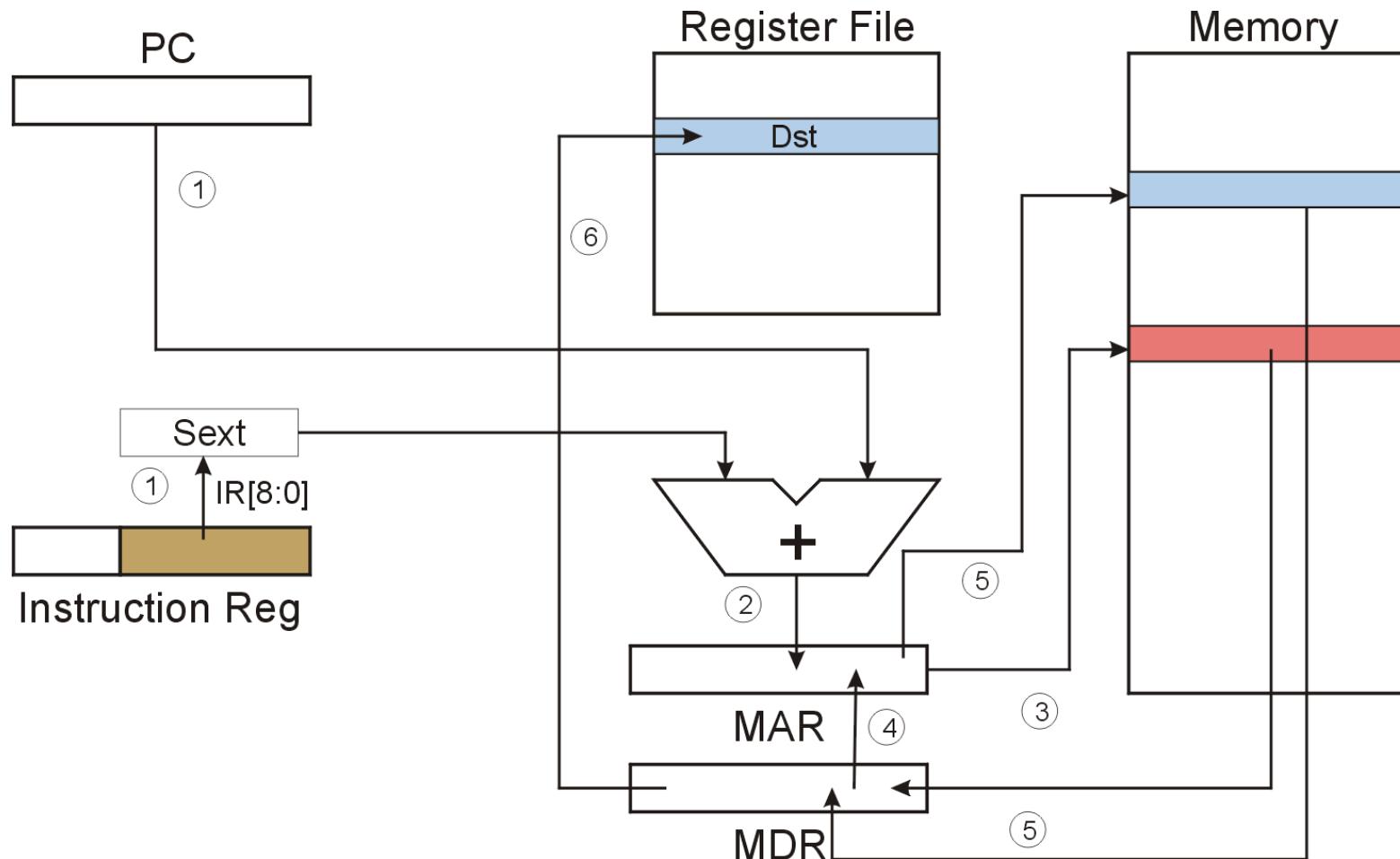
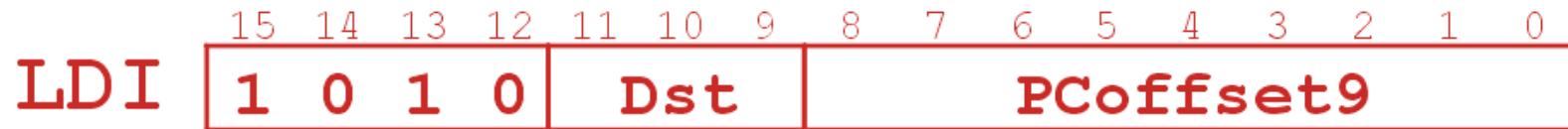
Indirect Addressing Mode

With PC-relative mode, can only address data within 256 words of the instruction. What about the rest of memory?

Solution #1: Read address from memory location, then load/store to that address.

- (1) Compute $PC + IR[8:0]$ (just like PC-Relative), put in MAR.
- (2) Load from that address into MDR.
- (3) Move data from MDR into MAR.
- (4) Perform load/store.

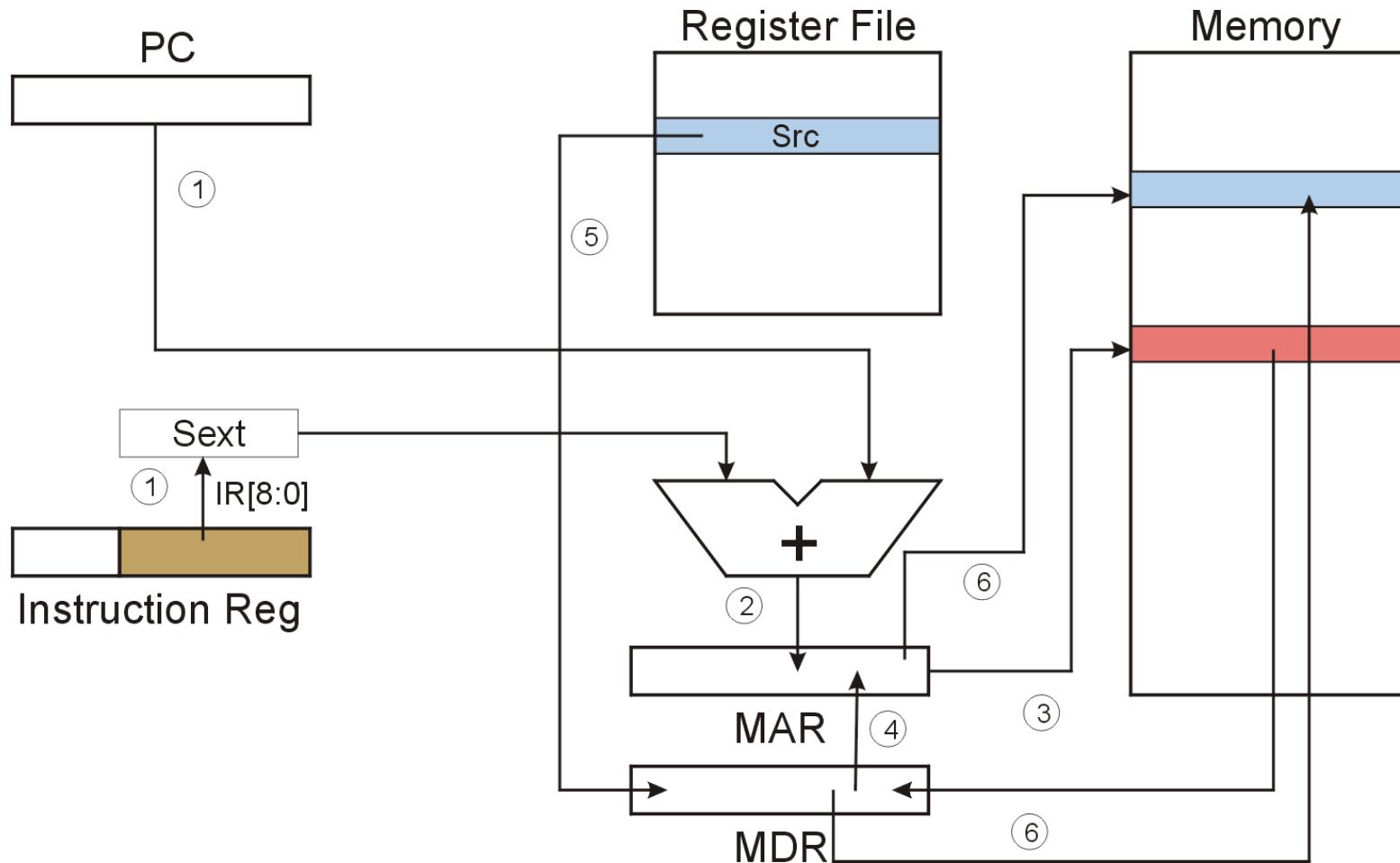
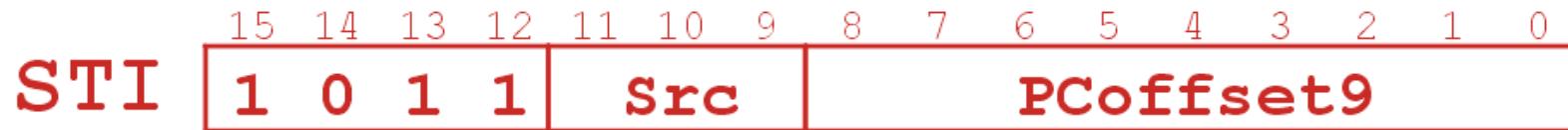
LDI (Indirect mode)



Sets condition codes.

[Access the text alternative for slide images.](#)

STI (Indirect mode)



[Access the text alternative for slide images.](#)

Base + Offset Addressing Mode

With PC-relative mode, can only address data within 256 words of the instruction. What about the rest of memory?

Solution #2: Use a register to generate a full 16-bit address.

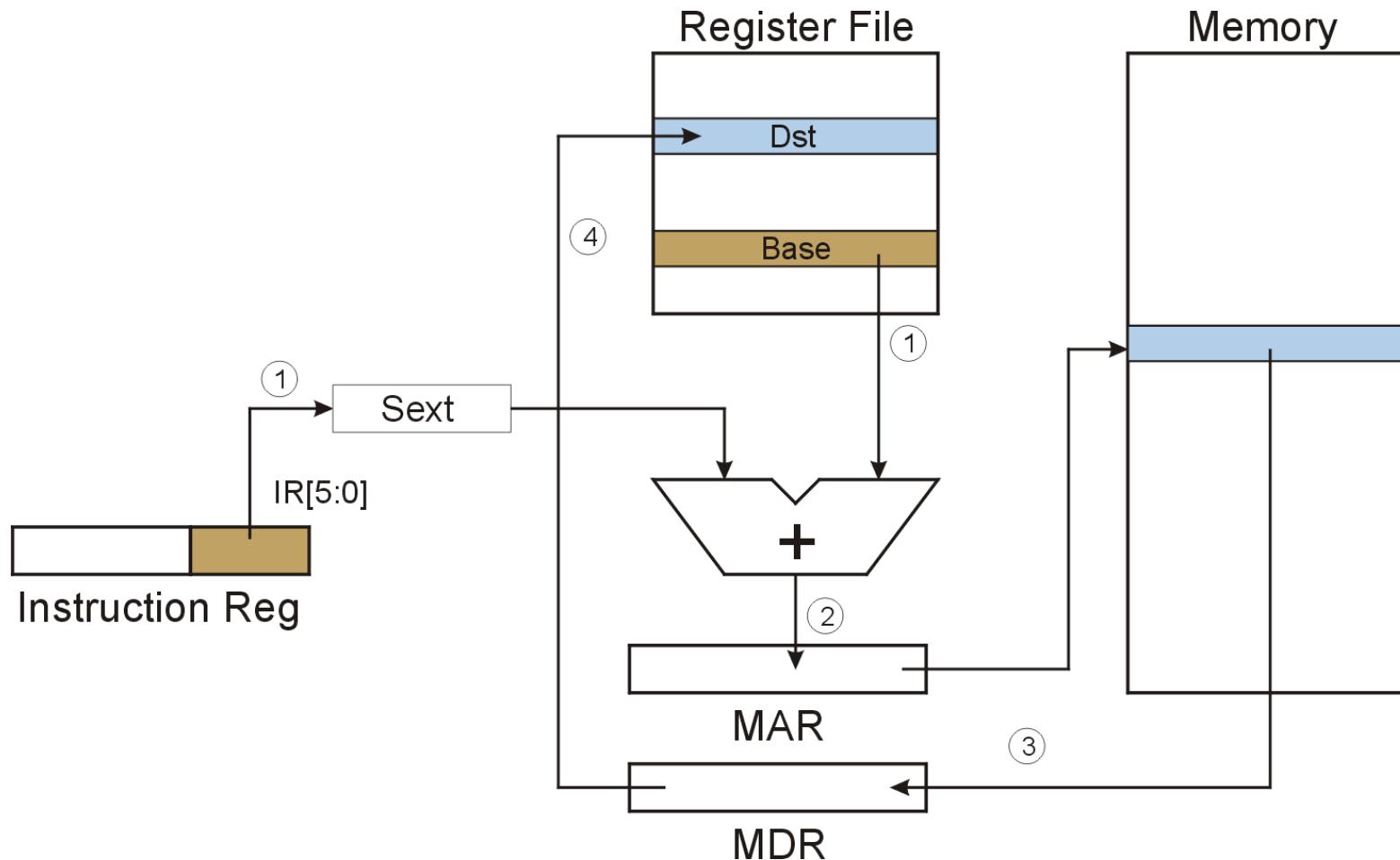
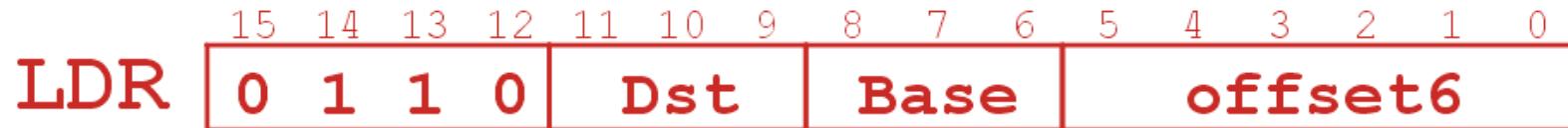
4 bits for opcode, 3 for src/dest register,

3 bits for *base* register -- the one to be used as an address.

Remaining 6 bits are used as a *signed offset*.

- Offset is *sign-extended* before adding to base register.
Offset is often zero, but a non-zero offset can be useful at times.

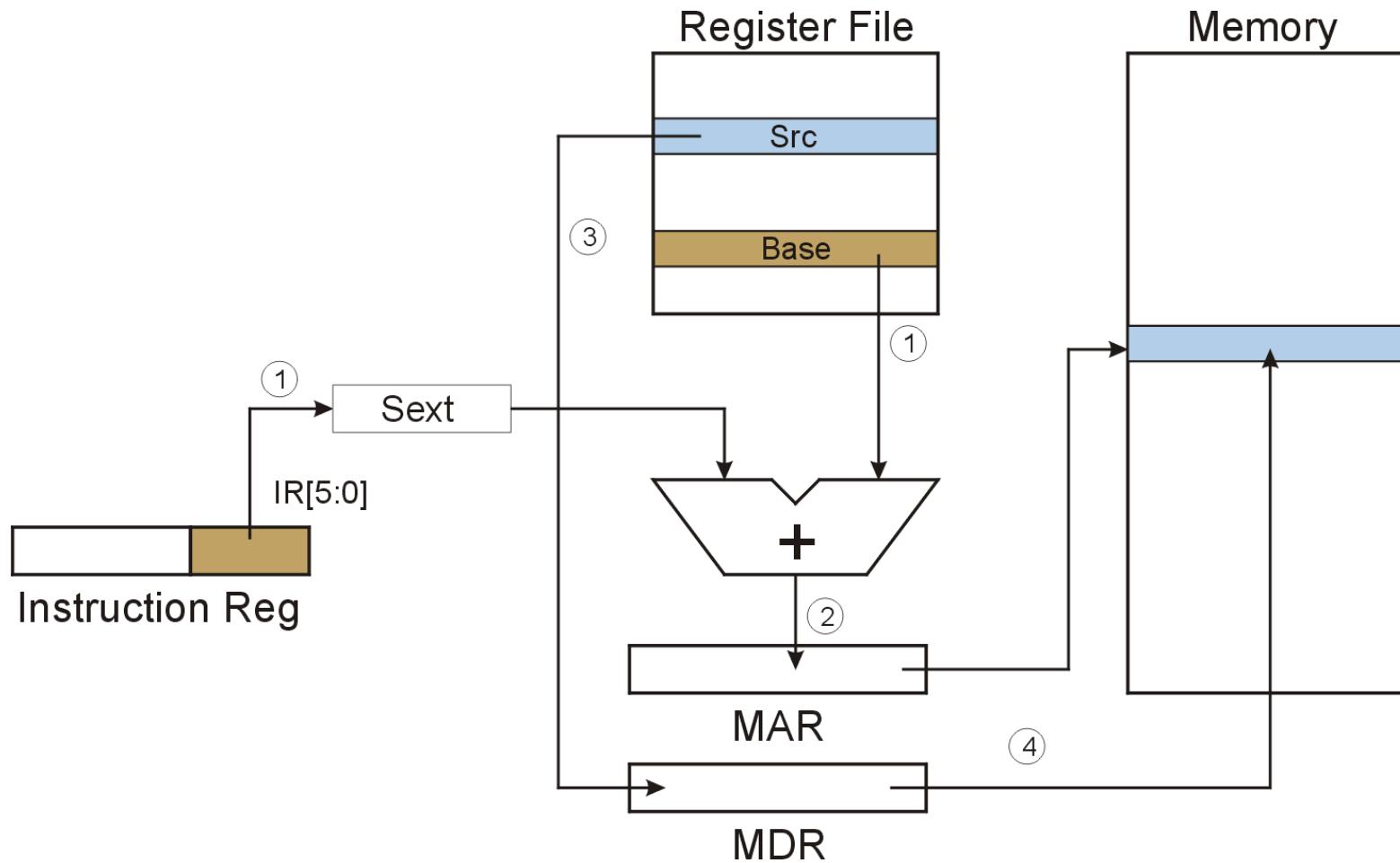
LDR (Base + Offset mode)



Sets condition codes.

Access the text alternative for slide images.

STR (Base + Offset mode)



[Access the text alternative for slide images.](#)

Example using Load/Store Addressing Modes

Address	Instruction										Comments
x30F6	1 1 1 0	0 0 1	1 1 1 1	1 1 1 0 1							$R1 \leftarrow PC - 3 = x30F4$
x30F7	0 0 0 1	0 1 0	0 0 1	1 0 1 1 1 0							$R2 \leftarrow R1 + 14 = x3102$
x30F8	0 0 1 1	0 1 0	1 1 1 1 1 1 0 1 1								$M[PC - 5] \leftarrow R2$ $M[x30F4] \leftarrow x3102$
x30F9	0 1 0 1	0 1 0	0 1 0 1 0 1 0 0 0 0 0								$R2 \leftarrow 0$
x30FA	0 0 0 1	0 1 0	0 1 0 1 0 1 0 0 1 0 1								$R2 \leftarrow R2 + 5 = 5$
x30FB	0 1 1 1	0 1 0	0 0 1 0 0 0 1 0 0 1 1 1 0								$M[R1+14] \leftarrow R2$ $M[x3102] \leftarrow 5$
x30FC	1 0 1 0	0 1 1	1 1 1 1 1 1 0 1 1 1								$R3 \leftarrow M[M[x30F4]]$ $R3 \leftarrow M[x3102]$ $R3 \leftarrow 5$

opcode

[Access the text alternative for slide images.](#)

Control Instructions

Changes the PC to redirect program execution.

Conditional Branch: BR

Unconditional Branch: JMP

Subroutines and System Calls: JSR, JSRR, TRAP, RET, RTI

(Discussed in Chapter 8 and Chapter 9)

Condition Codes

LC-3 has three condition code registers.

N = negative (less than zero)

Z = zero

P = positive (greater than zero)

Set by instructions that compute or load a value into a register.

ADD, AND, NOT, LD, LDI, LDR

Based on result of computation / load. Other instructions do not change the condition codes.

Exactly one of these will be set (1). Others will be zero.

Conditional Branch (BR)

BR specifies two things:

- Condition --
if true, branch is **taken** (PC changed to target address)
if false, branch is **not taken** (PC not changed)
- Target address.

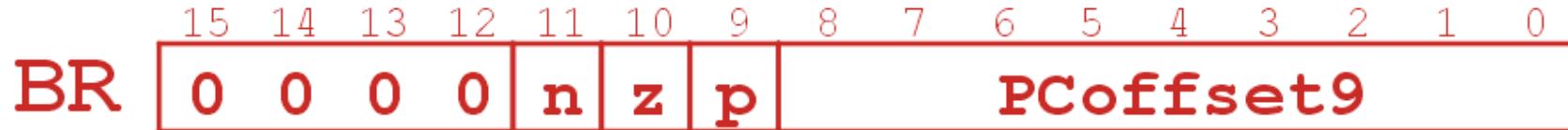
Condition specifies which condition code bits are relevant for this branch.
If any specified bit is set, branch is taken.

Example: BRn = branch if N bit is set

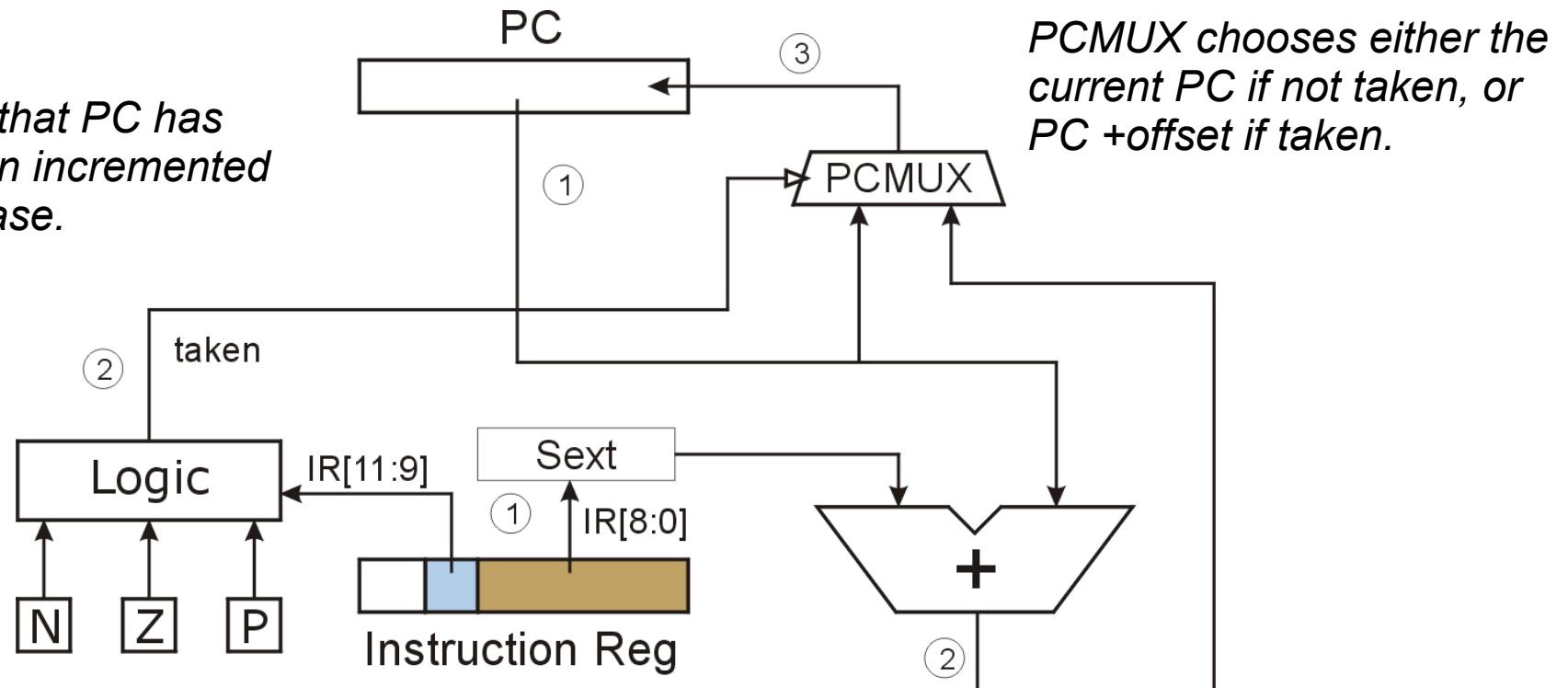
Example: BRnp = branch if N or P bit is set

Target is specified using PC-Relative mode.

BR



Remember that PC has already been incremented in Fetch phase.



Logic: $(IR[11] \text{ and } N) \text{ or } (IR[10] \text{ and } Z) \text{ or } (IR[9] \text{ and } P)$

[Access the text alternative for slide images.](#)

Example BR instruction

Suppose the following instruction is fetched from address x4027

Copyright © McGraw-Hill Education. Permission required for reproduction or display.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	1	1	0	1	1	0	0	1

BR n z p x0D9

Condition: z

Branch will be taken if Z bit is set.

Target: x4028 + x00D9 = x4101

Current PC is x4028, because it was incremented when fetching.

If branch is taken PC will be x4101.

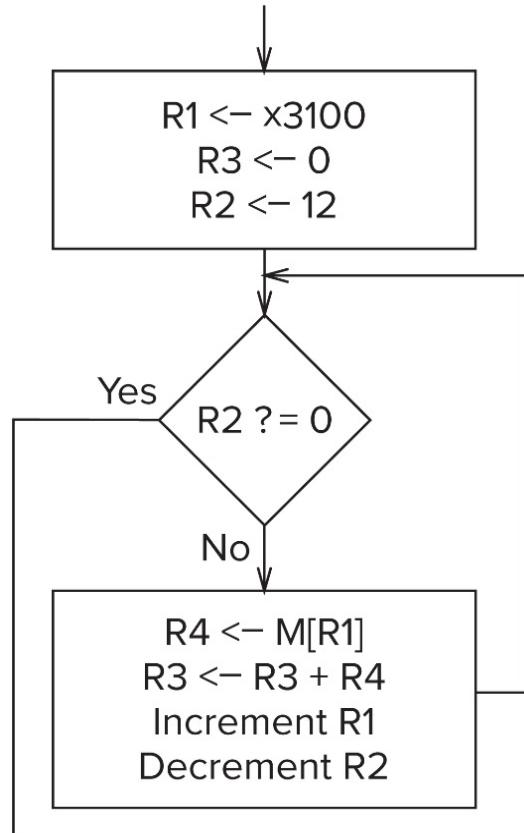
If branch is not taken PC will remain x4028.

[Access the text alternative for slide images.](#)

Example: Loop with a Counter

Compute the sum of 12 integers stored in x3100 - x310B. Instructions begin at address x3000.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



Copyright © McGraw-Hill Education. Permission required for reproduction or display.

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x3000	1	1	1	0	0	0	1	0	1	1	1	1	1	1	1	1
x3001	0	1	0	1	0	1	1	0	1	1	1	0	0	0	0	0
x3002	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0
x3003	0	0	0	1	0	1	0	0	1	0	1	0	1	1	0	0
x3004	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0
x3005	0	1	1	0	1	0	0	0	0	1	0	0	0	0	0	0
x3006	0	0	0	1	0	1	1	0	1	1	0	0	0	1	0	0
x3007	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1
x3008	0	0	0	1	0	1	0	0	1	0	1	1	1	1	1	1
x3009	0	0	0	0	1	1	1	1	1	1	1	1	1	0	1	0

R1 <- 3100
R3 <- 0
R2 <- 0
R2 <- 12
BRz x300A
R4 <- M[R1]
R3 <- R3 + R4
R1 <- R1 + 1
R2 <- R2 - 1
BRnzp x3004

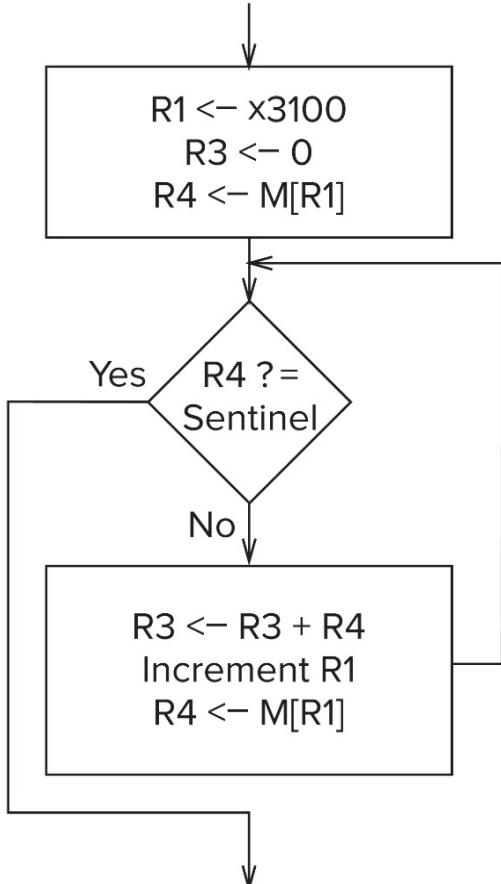
R2 counts down from 12. Branch out of the loop when R2 becomes zero.

[Access the text alternative for slide images.](#)

Example: Loop with Sentinel

Compute the sum of integers starting at x3100, until a negative integer is found. Instructions start at x3000.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x3000	1	1	1	0	0	0	1	0	1	1	1	1	1	1	1	1
x3001	0	1	0	1	0	1	1	0	1	1	1	0	0	0	0	0
x3002	0	1	1	0	1	0	0	0	0	1	0	0	0	0	0	0
x3003	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0
x3004	0	0	0	1	0	1	1	0	1	1	0	0	0	1	0	0
x3005	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1
x3006	0	1	1	0	1	0	0	0	0	1	0	0	0	0	0	0
x3007	0	0	0	0	1	1	1	1	1	1	1	1	1	0	1	1

Memory dump:

R1 <- x3100	R3 <- 0	R4 <- M[R1]	BRn x3008	R3 <- R3+R4	R1 <- R1+1	R4 <- M[R1]	BRnzp x3003
-------------	---------	-------------	-----------	-------------	------------	-------------	-------------

In this case, it's the data that tells when to exit the loop. When the value loaded into R4 is negative, take branch to x3008.

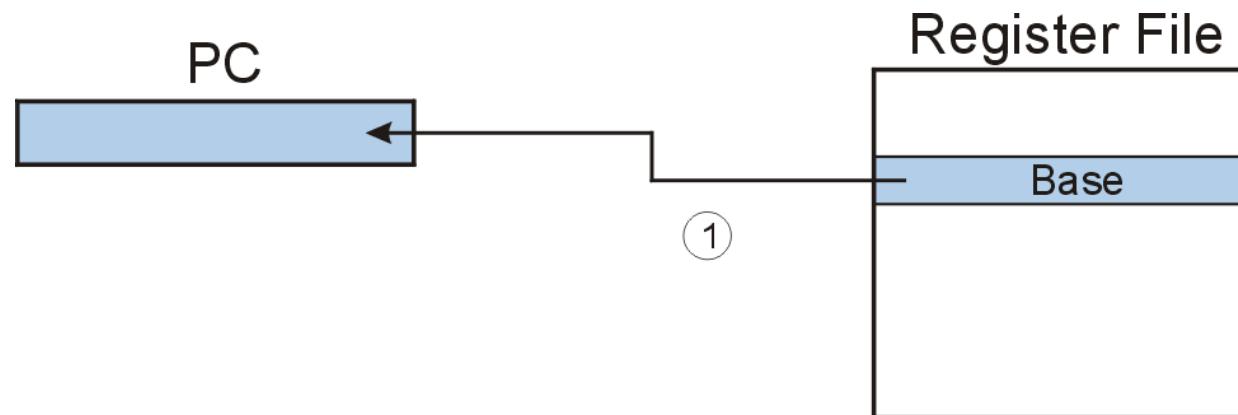
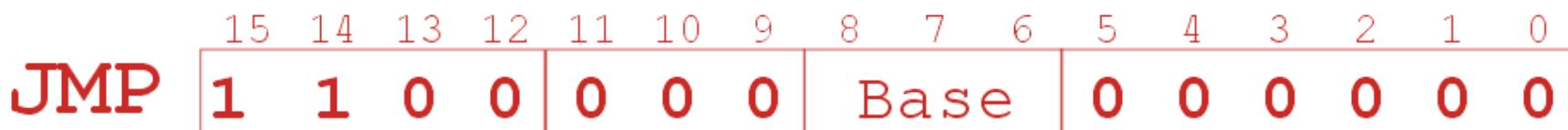
[Access the text alternative for slide images.](#)

Unconditional Branch (JMP)

We can create a branch that is always taken by setting n, z, and p to 1.

However, the target address of BR is limited by the 9-bit PC offset.

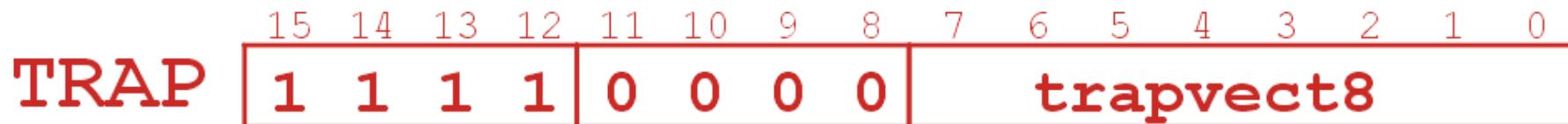
The JMP instruction provides an unconditional branch to any target location by using the contents of a register as the target address. It simply copies the register into the PC.



[Access the text alternative for slide images.](#)

TRAP: Invoke a System Service Routine

The TRAP instruction is used to give control to the operating system to perform a task that user code is not allowed to do. The details will be explained in Chapter 9.



For now, you just need to know that bits [7:0] hold a "trap vector" -- a unique code that specifies the service routine. The service routines used in this part of the course are:

trapvector	service routine
x23	Input a character from the keyboard.
x21	Output a character to the monitor
x25	Halt the processor.

[Access the text alternative for slide images.](#)

Input / Output Service Routines

Getting character input from the keyboard

- TRAP x23 is used to invoke the keyboard input service routine.
- When the OS returns control to our program, the ASCII code for the key pressed by the user will be in R0.

Sending character output to the monitor

- TRAP x21 is used to invoke the monitor output service routine.
- Before invoking the routine, put the ASCII character to be output into R0.

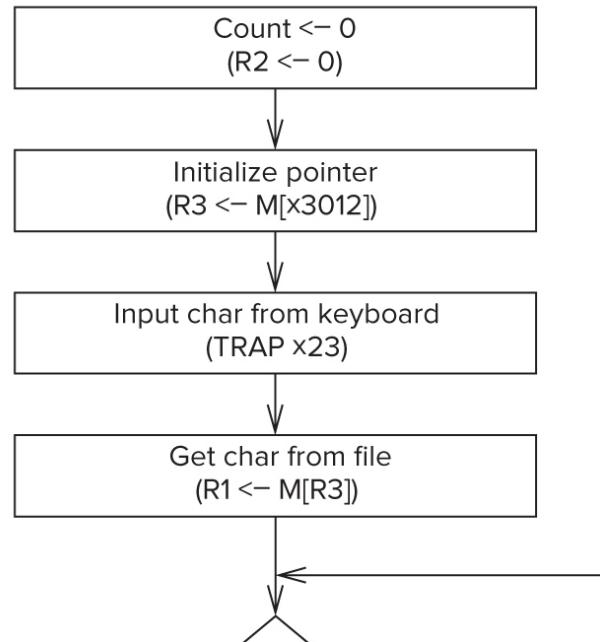
Example Program: Count Occurrences of a Character

We want to count the number of times a user-specified character appears in a text file, and then print the count to the monitor.

- The text file is stored in memory as a sequence of ASCII characters.
- The end of the file is denoted with the EOT character (x04).
NOTE: We do not know how many characters are in the file; EOT is the sentinel value that signals when we are done.
- A pointer to the file will be stored at the end of the program. (A "pointer" is a memory address; it will be the address of the first character in the file.)
- We will assume that the character will appear no more than 9 times.
- Program instructions will start at x3000. The file data can be anywhere in memory.

Part 1: Initializing Registers

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



Copyright © McGraw-Hill Education. Permission required for reproduction or display.

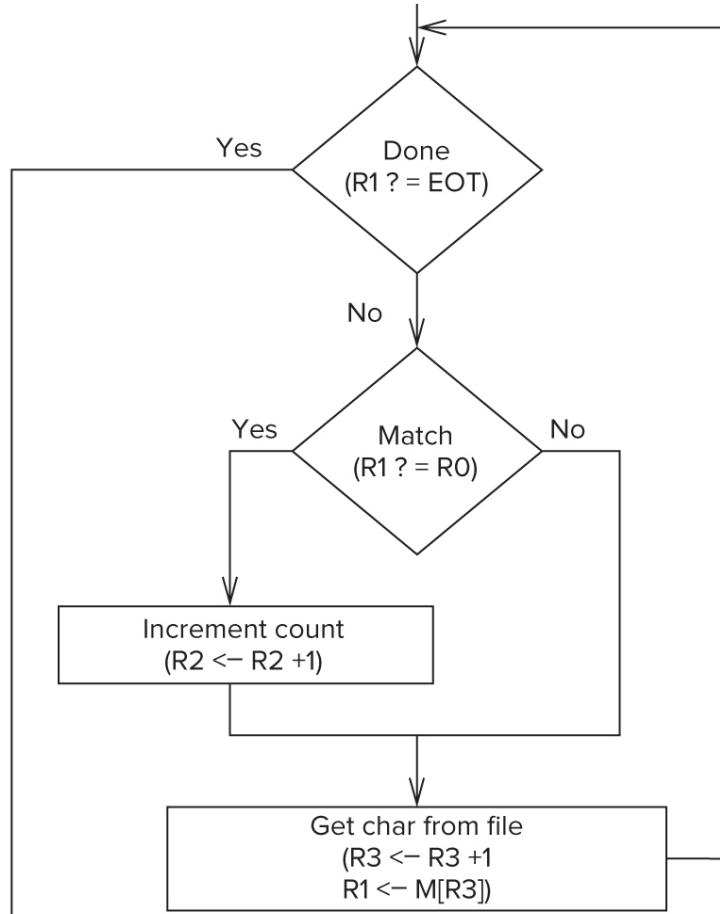
Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3000	0	1	0	1	0	0	1	0	1	0	0	0	0	0	0	0	R2 <- 0
x3001	0	0	1	0	0	1	1	0	0	0	0	1	0	0	0	0	R3 <- M[x3012]
x3002	1	1	1	1	0	0	0	0	0	0	1	0	0	0	1	1	TRAP x23
x3003	0	1	1	0	0	0	1	0	1	1	0	0	0	0	0	0	R1 <- M[R3]
x3004	0	0	0	1	1	0	0	0	0	1	1	1	1	1	0	0	R4 <- R1-4
x3005	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	BRz x300E
x3006	1	0	0	1	0	0	1	0	0	1	1	1	1	1	1	1	R1 <- NOT R1
x3007	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1	R1 <- R1 + 1
x3008	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	R1 <- R1 + R0
x3009	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	1	BRnp x300B
x300A	0	0	0	1	0	1	0	0	1	0	1	0	0	0	0	1	R2 <- R2 + 1
x300B	0	0	0	1	0	1	1	0	1	1	1	0	0	0	0	1	R3 <- R3 + 1
x300C	0	1	1	0	0	0	1	0	1	1	0	0	0	0	0	0	R1 <- M[R3]
x300D	0	0	0	0	1	1	1	1	1	1	1	0	1	1	0	0	BRnzp x3004
x300E	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	R0 <- M[x3013]
x300F	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	R0 <- R0 + R2
x3010	1	1	1	1	0	0	0	0	0	0	1	0	0	0	0	1	TRAP x21
x3011	1	1	1	1	0	0	0	0	0	0	0	1	0	0	1	0	TRAP x25
x3012	Starting address of file																ASCII TEMPLATE
x3013	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	

R2 is counter. R3 is address of first character to read from file.
R0 is character from keyboard. R1 is first character from file.

[Access the text alternative for slide images.](#)

Part 2: Read Characters and Count

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



Copyright © McGraw-Hill Education. Permission required for reproduction or display.

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x3000	0	1	0	1	0	0	1	0	1	0	0	0	0	0	0	0
x3001	0	0	1	0	0	1	1	0	0	0	0	1	0	0	0	0
x3002	1	1	1	1	0	0	0	0	0	1	0	0	0	1	1	
x3003	0	1	1	0	0	0	1	0	1	1	0	0	0	0	0	0
x3004	0	0	0	1	1	0	0	0	0	1	1	1	1	0	0	0
x3005	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0
x3006	1	0	0	1	0	0	1	0	0	1	1	1	1	1	1	
x3007	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1
x3008	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0
x3009	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	1
x300A	0	0	0	1	0	1	0	0	1	0	1	0	0	0	0	1
x300B	0	0	0	1	0	1	1	0	1	1	1	0	0	0	0	1
x300C	0	1	1	0	0	0	1	0	1	1	0	0	0	0	0	0
x300D	0	0	0	0	1	1	1	1	1	1	1	0	1	1	0	0
x300E	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0
x300F	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0
x3010	1	1	1	1	0	0	0	0	0	1	0	0	0	0	1	
x3011	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1
x3012	Starting address of file															
x3013	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0

R2 <- 0
R3 <- M[x3012]
TRAP x23
R1 <- M[R3]
R4 <- R1-4
BRz x300E
R1 <- NOT R1
R1 <- R1 + 1
R1 <- R1 + R0
BRnp x300B
R2 <- R2 + 1
R3 <- R3 + 1
R1 <- M[R3]
BRnzp x3004
R0 <- M[x3013]
R0 <- R0 + R2
TRAP x21
TRAP x25

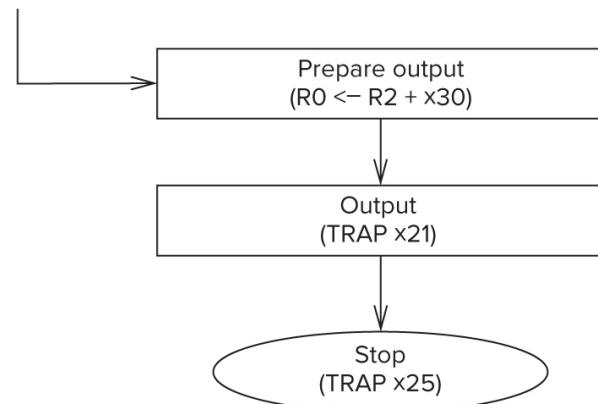
ASCII TEMPLATE

Compare character to x04 (EOT). If equal, exit loop. Otherwise, count if matches user input and read the next character.

[Access the text alternative for slide images.](#)

Part 3: Output Count and HALT

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



Copyright © McGraw-Hill Education. Permission required for reproduction or display.

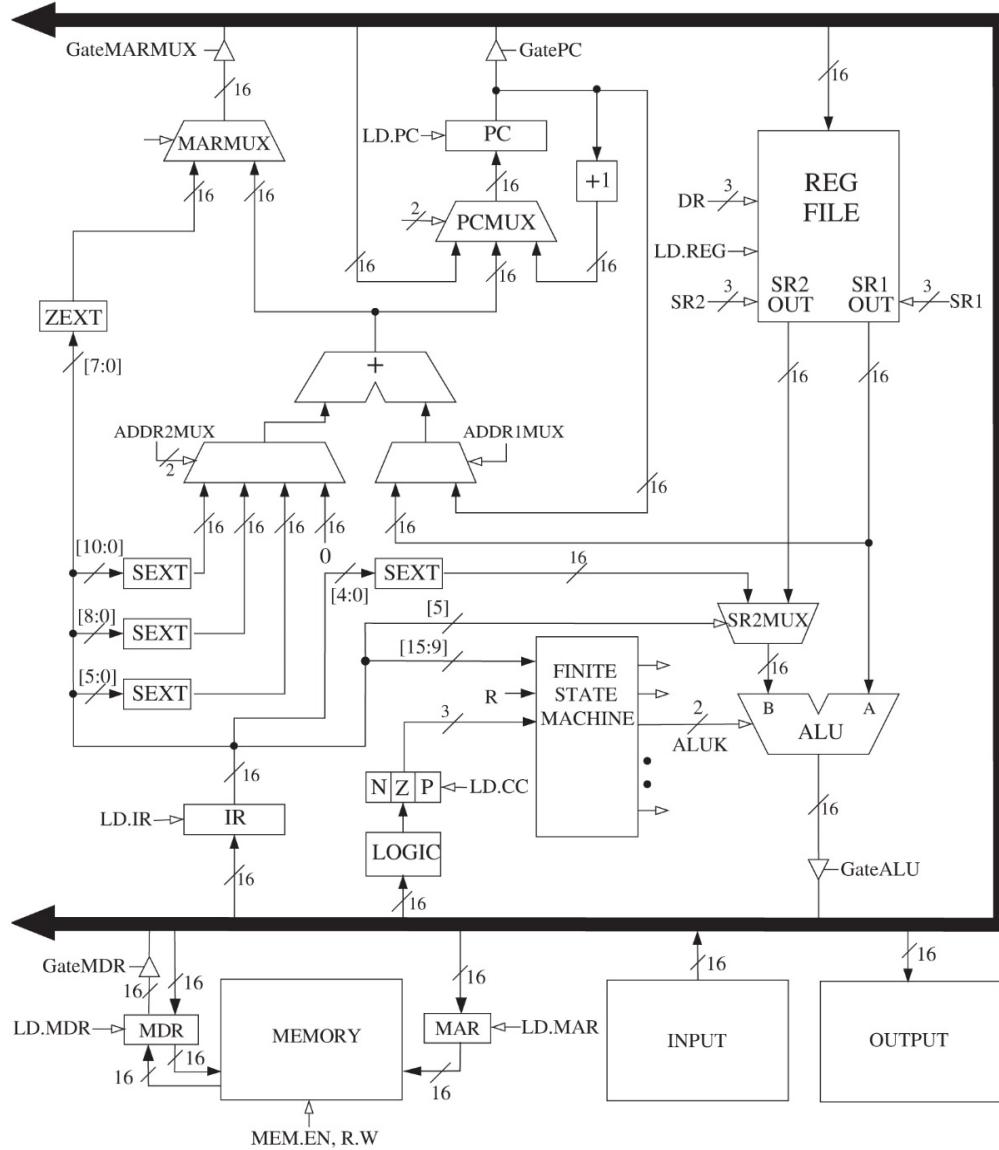
Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x3000	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0
x3001	0	0	1	0	0	1	1	0	0	0	0	1	0	0	0	0
x3002	1	1	1	1	0	0	0	0	0	0	1	0	0	0	1	1
x3003	0	1	1	0	0	0	1	0	1	1	0	0	0	0	0	0
x3004	0	0	0	1	1	0	0	0	1	1	1	1	1	0	0	0
x3005	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0
x3006	1	0	0	1	0	0	1	0	0	1	1	1	1	1	1	1
x3007	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1
x3008	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0
x3009	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	1
x300A	0	0	0	1	0	1	0	0	1	0	1	0	0	0	0	1
x300B	0	0	0	1	0	1	1	0	1	1	1	0	0	0	0	1
x300C	0	1	1	0	0	0	1	0	1	1	0	0	0	0	0	0
x300D	0	0	0	0	1	1	1	1	1	1	1	1	0	1	1	0
x300E	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0
x300F	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0
x3010	1	1	1	1	0	0	0	0	0	0	1	0	0	0	0	1
x3011	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1
x3012	Starting address of file															
x3013	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0

When loop is finished, convert count (R2) to the corresponding ASCII character by adding '0' (x30). Output the character and halt the program.

[Access the text alternative for slide images.](#)

LC-3 Data Path

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



Data path is used to execute LC-3 programs.

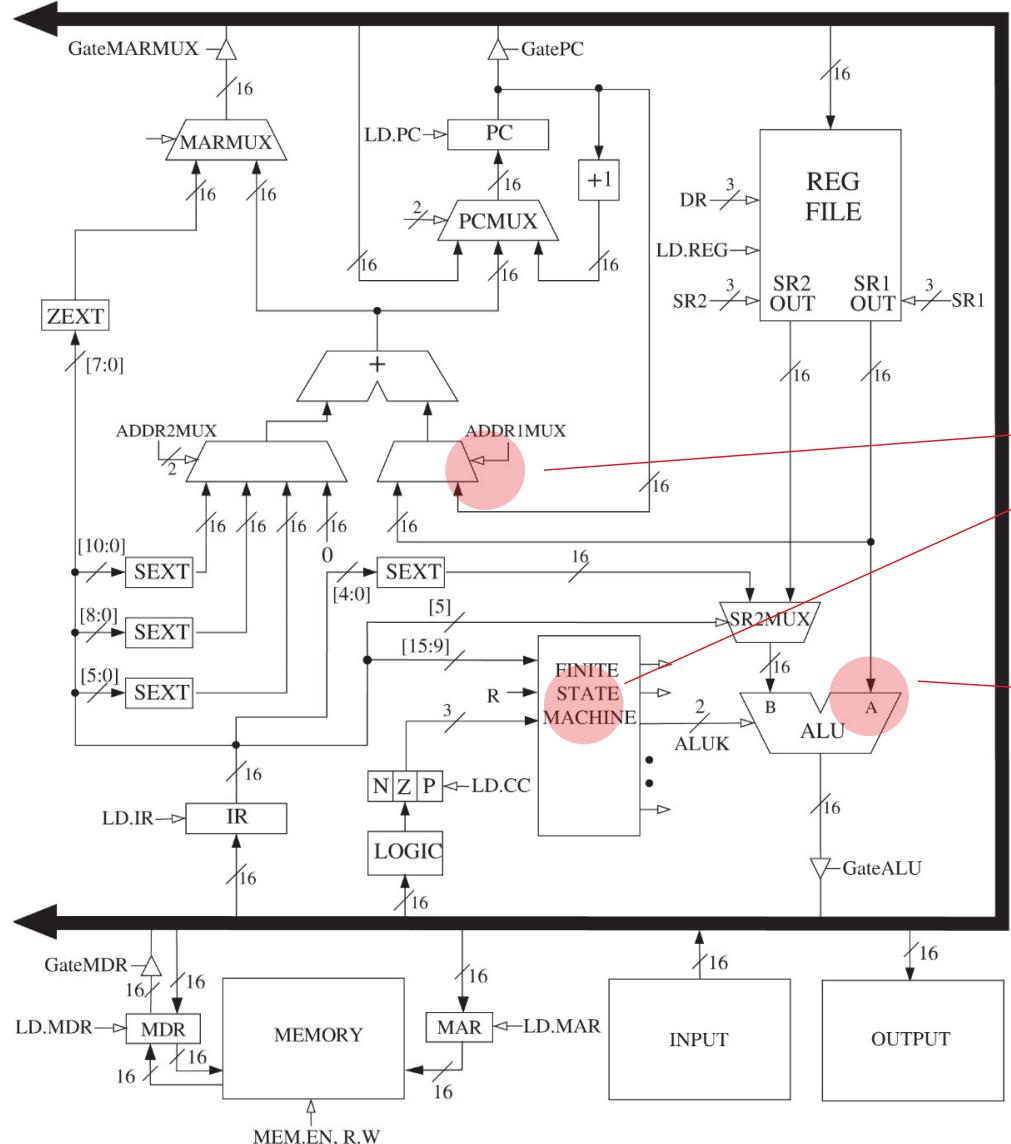
PC is initialized to point to the first instruction. Clock is enabled, and the control unit takes over.

Next slides will give a little more detail on various components.

[Access the text alternative for slide images](#)

LC-3 Data Path ₂

Copyright © McGraw-Hill Education. Permission required for reproduction or display.

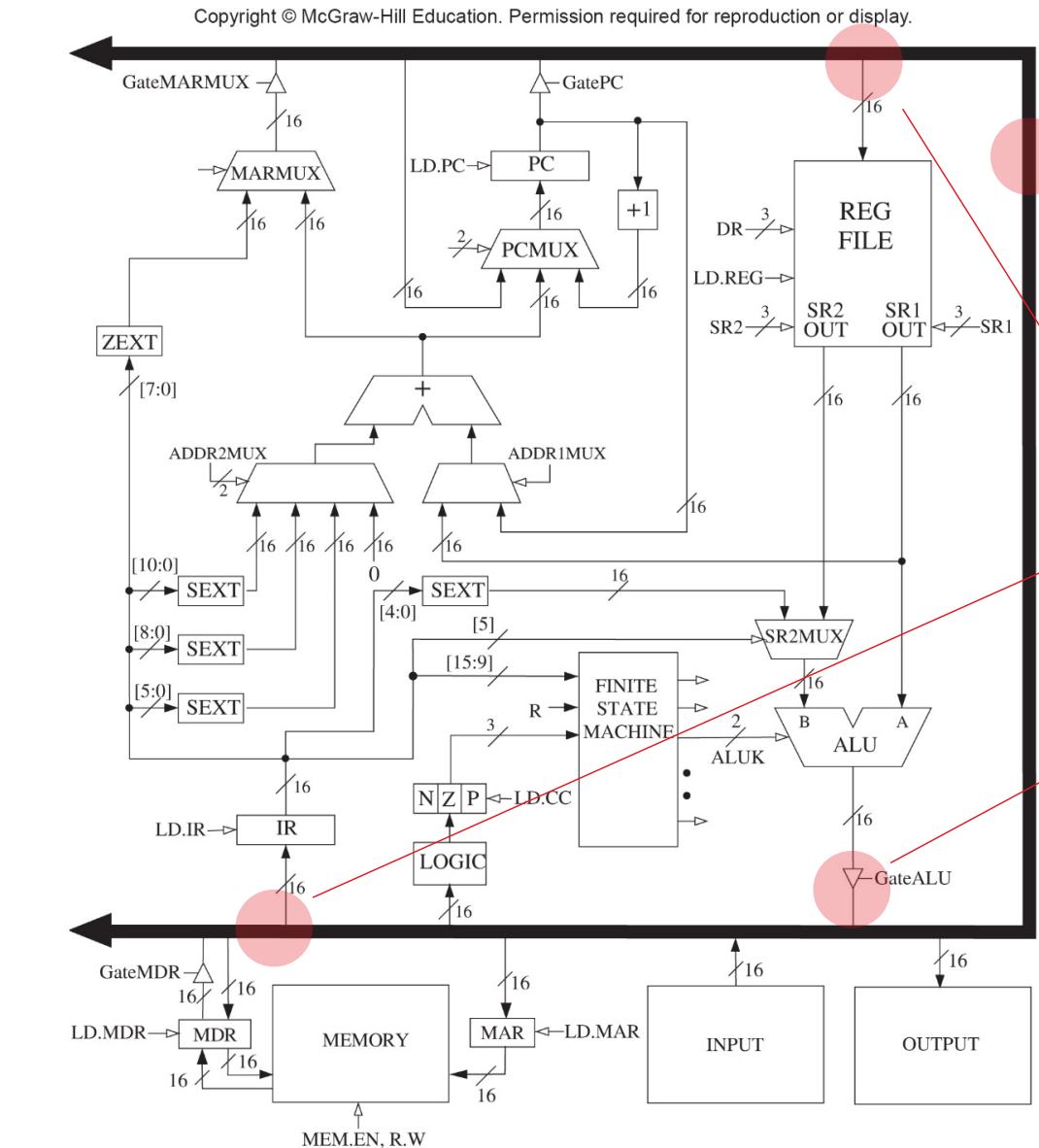


Arrows with open heads represent control signals from FSM.

Arrows with filled heads represent data that is processed.

Access the text alternative for slide images

LC-3 Data Path



Global bus is a set of wires that allow various components to transfer 16-bit data to other components.

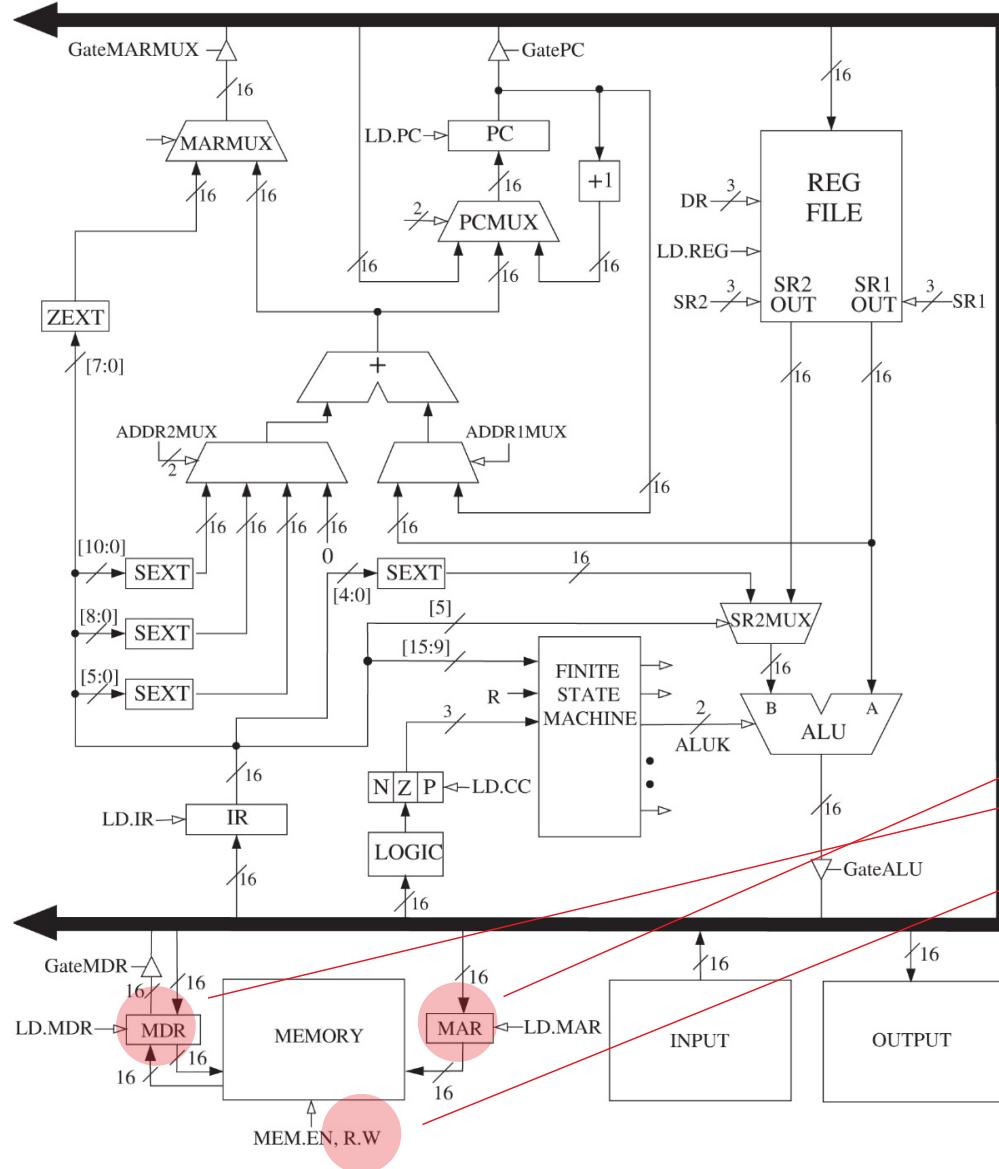
One or more components may read data from the bus on any cycle.

Tri-state device determines which component puts data on the bus. Only one source of data at any time.

Access the text alternative for slide images

LC-3 Data Path 4

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



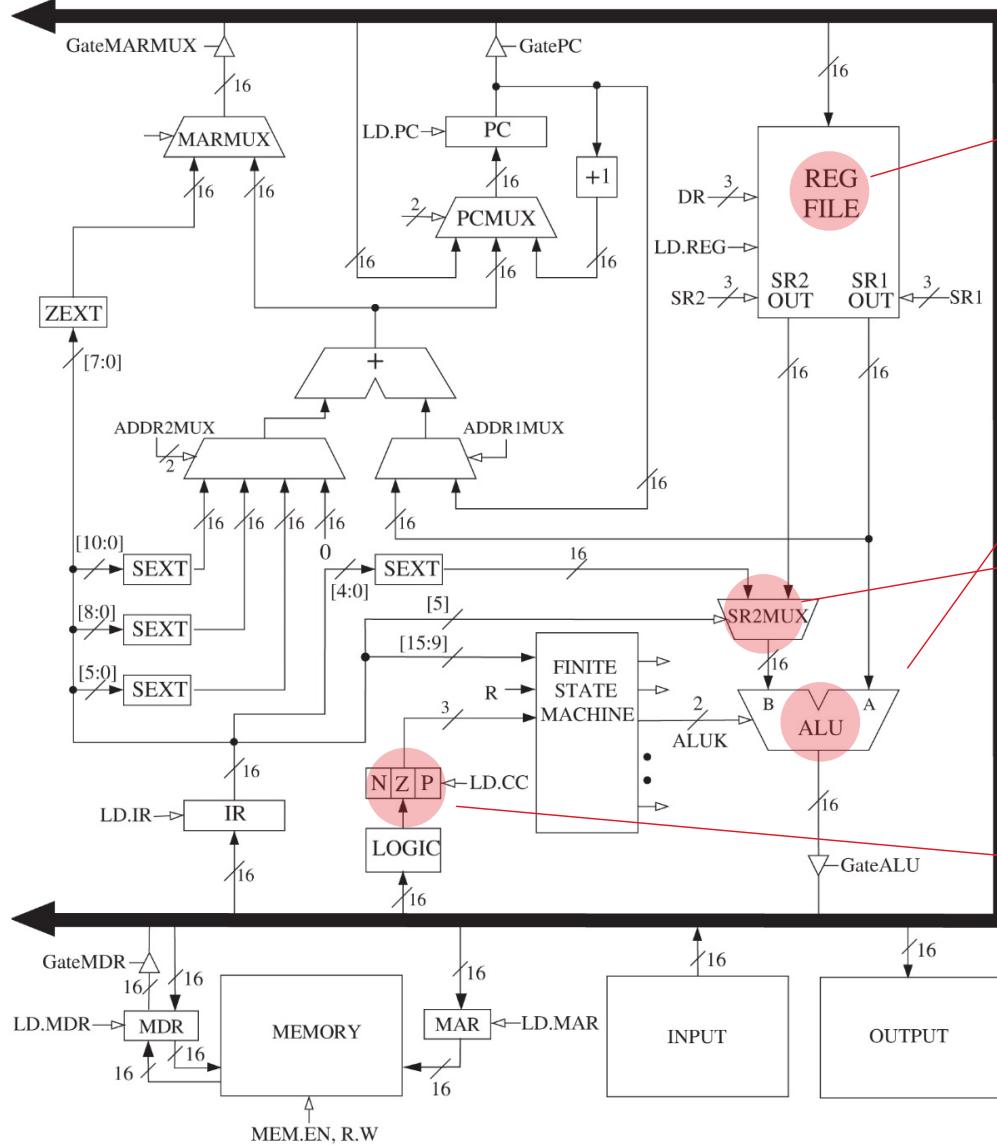
Memory interface:
MAR
MDR
Read/Write control

Access the text alternative for slide images

LC-3 Data Path

5

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



Register File (R0-R7)

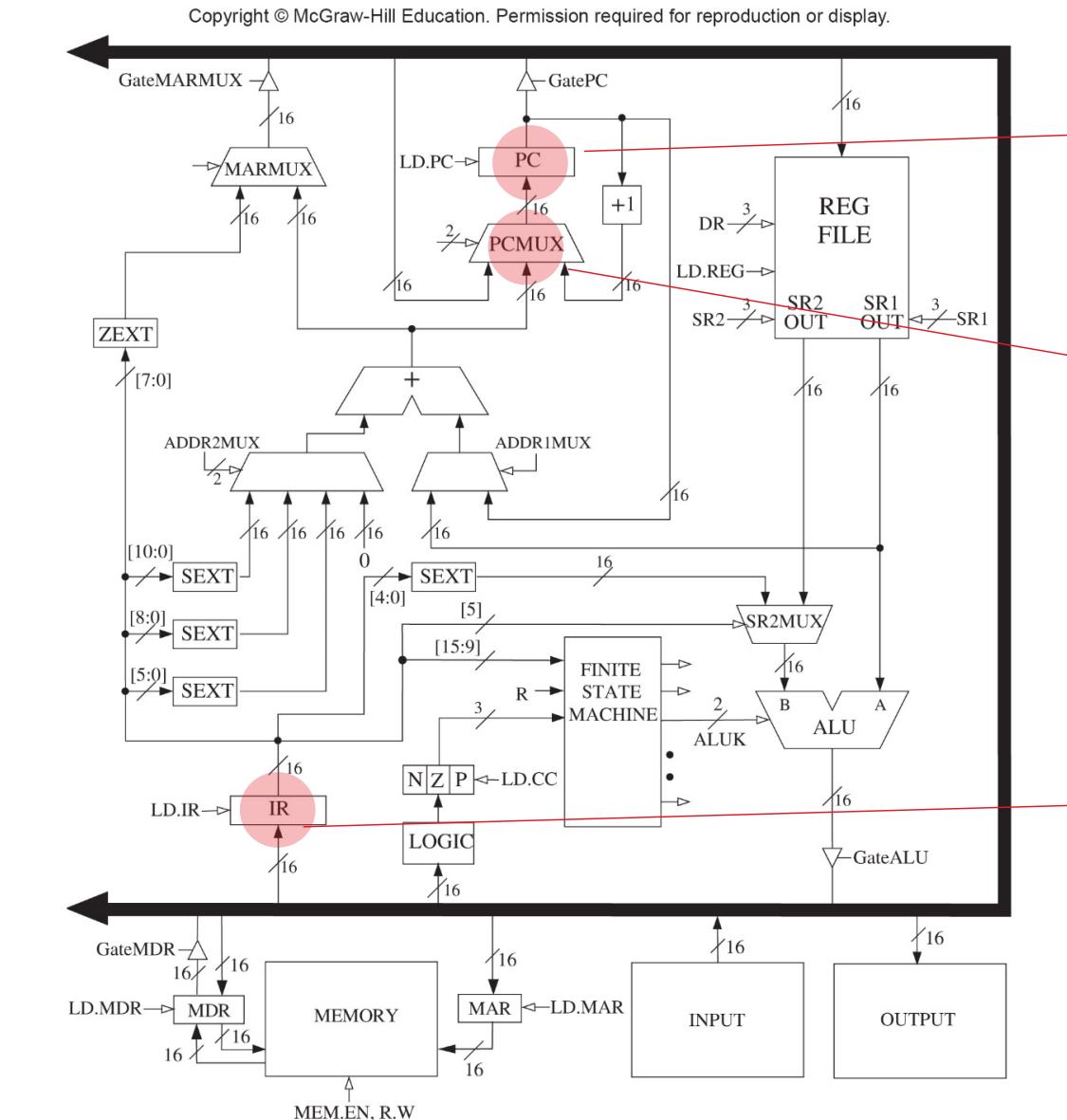
Control signals specify two source register (SR1, SR2) and one destination (DR).

ALU performs ADD, AND, NOT.

Operand A always comes from register file. Operand B is from register file or IR. Output goes to bus, to be written into register file.

Condition codes are set by looking at data placed on the bus by ALU or memory (MDR).

LC-3 Data Path



PC puts address on bus.
Placed in MAR during Fetch.

PCMUX allows various
values to be written to PC:
incremented PC (Fetch),
computed address (BR), or
register data from bus (JMP).

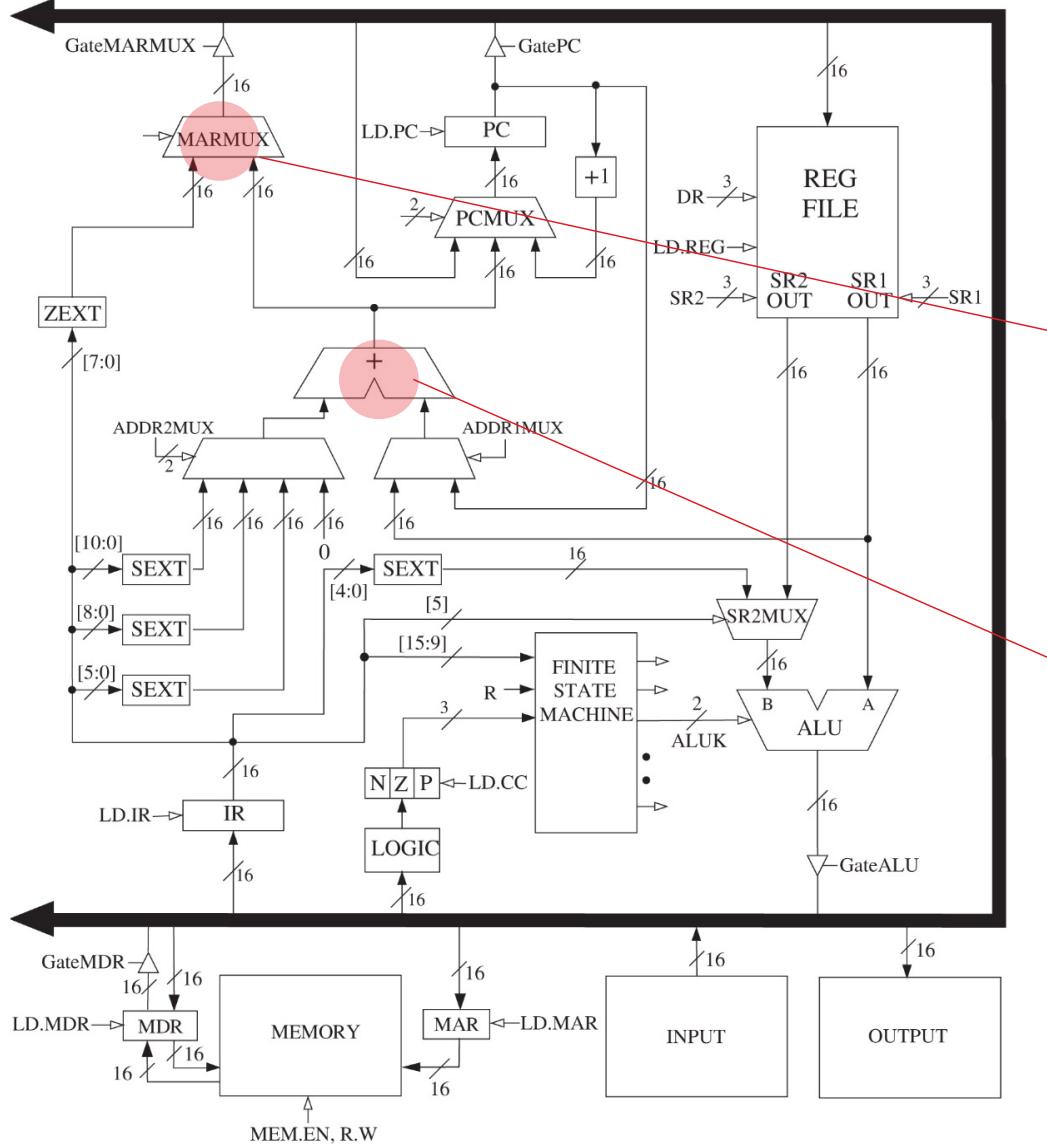
IR gets data from bus (MDR)
during Fetch.

Access the text alternative for slide images

LC-3 Data Path

7

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



MARMUX chooses value to be written to MAR during load, store, or TRAP.

Evaluate Address phase adds offset to PC or register for load, store, BR.

Access the text alternative for slide images

Chapter Summary

LC-3 is a simple, but complete, computer architecture.

Remember Chapter 1? LC-3 is capable of computing anything, given enough time and memory. Memory is limited to 64K words, but that's enough to solve many interesting problems.

We have presented the details of 12 opcodes -- the other 3 will be covered in Chapters 8 and 9.

We will now focus more on **software**:

- Writing algorithms to solve computational problems.
- Expressing those algorithms in machine instructions (Ch 6), assembly language (Ch 7 to 10), and higher-level languages (Ch 11 to 20).
- Using data structures -- arrangements of data in memory -- and subroutines/functions to write modular, reusable, and maintainable programs.



Because learning changes everything.[®]

www.mheducation.com

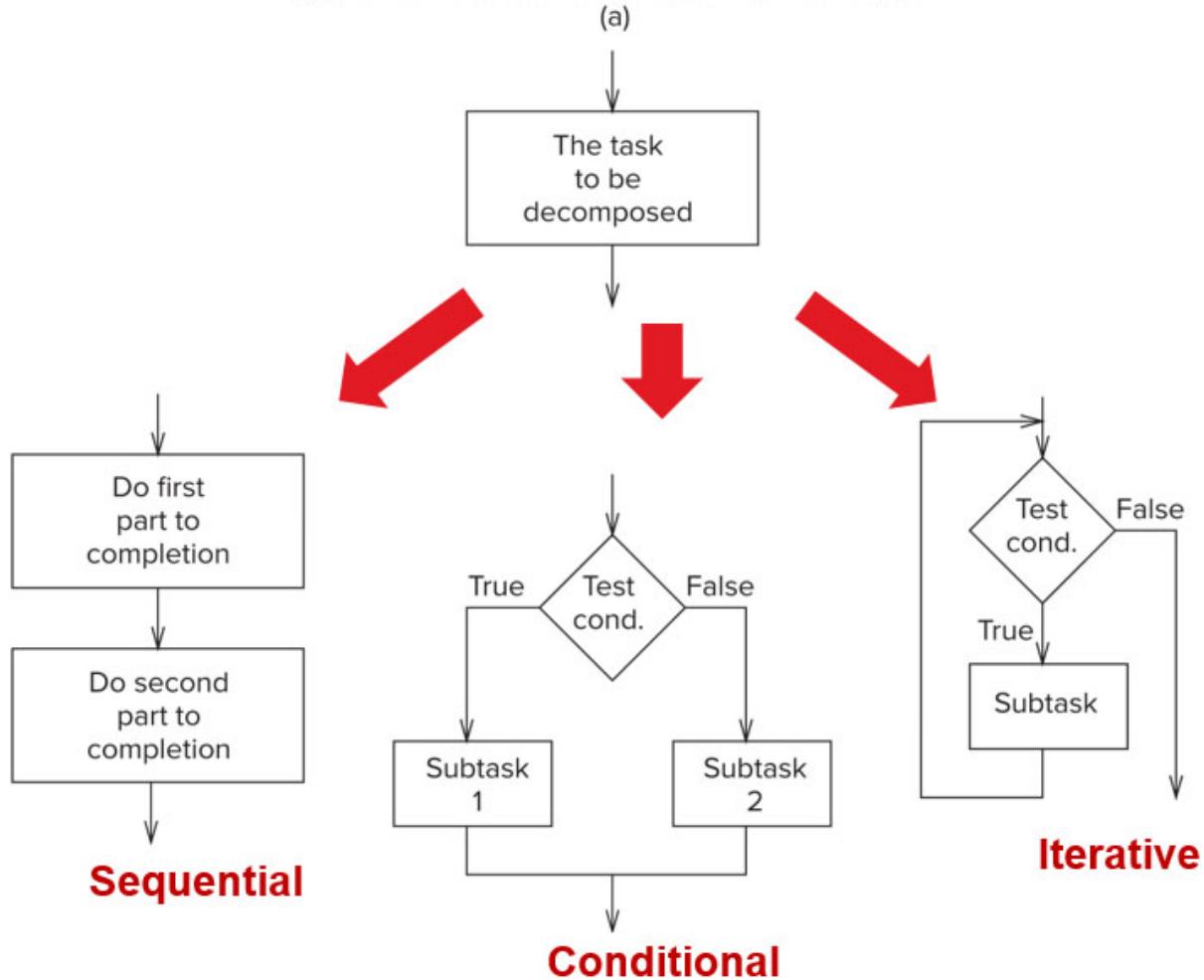
From Bits and Gates to C and Beyond

Programming

Chapter 6

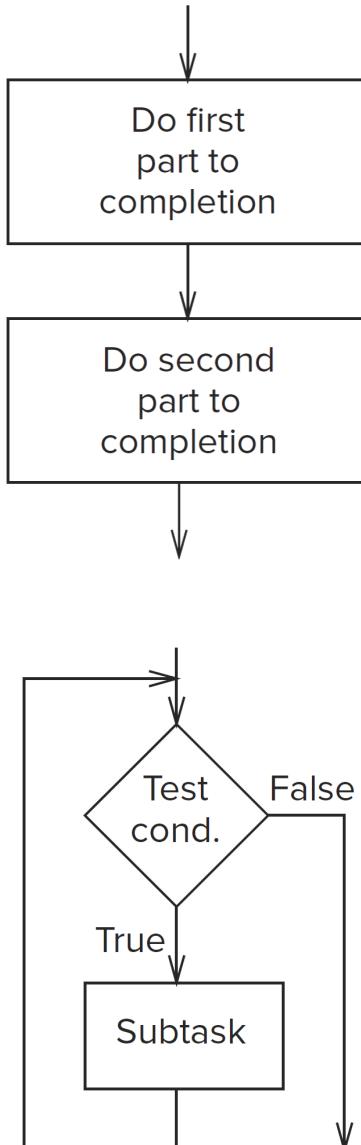
Structured Programming: Three Basic Constructs

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



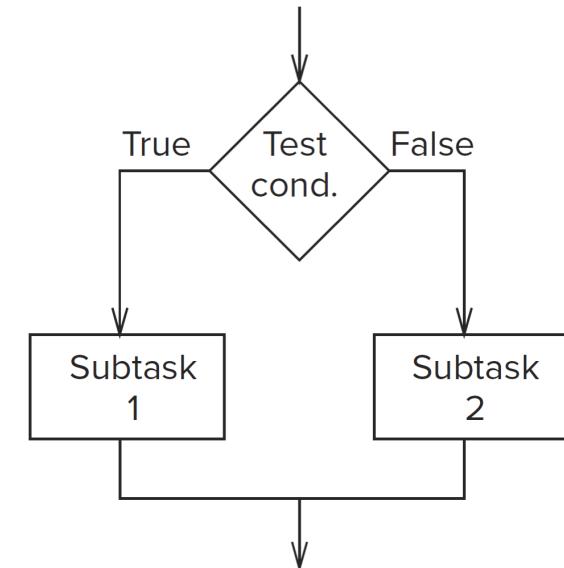
[Access the text alternative for slide images.](#)

Three Basic Constructs



Sequential: Break into two subtasks, both of which must be performed. Do the first task to completion, then the second.

Conditional: Task consists of two subtasks, one of which should be performed and the other not performed. Test condition to choose, then perform subtask to completion. (Either subtask may be empty.)



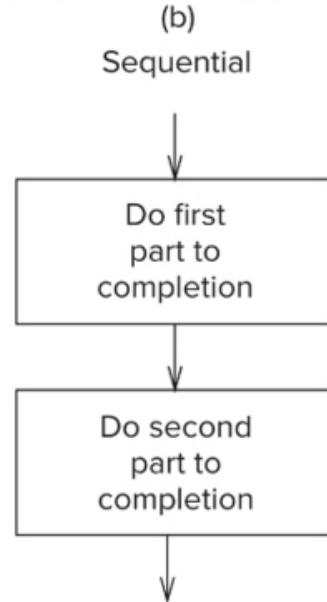
Iterative: Perform a single subtask multiple times, but only if condition is true. When task is done, go back to retest the condition. The first time condition is not true, program proceeds.

[Access the text alternative for slide images.](#)

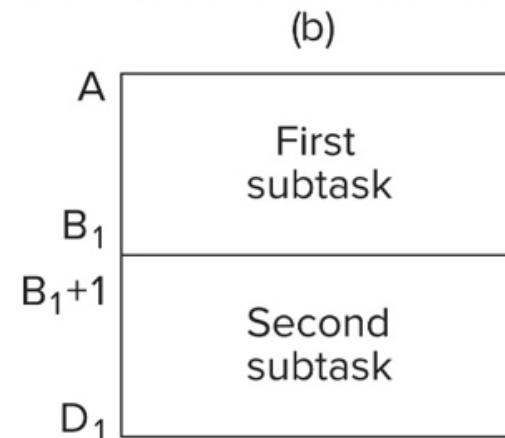
Implementing Constructs with LC-3 Instructions 1

Sequential

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



Copyright © McGraw-Hill Education. Permission required for reproduction or display.



No control instructions needed. Execution flows sequentially from one subtask to the next.

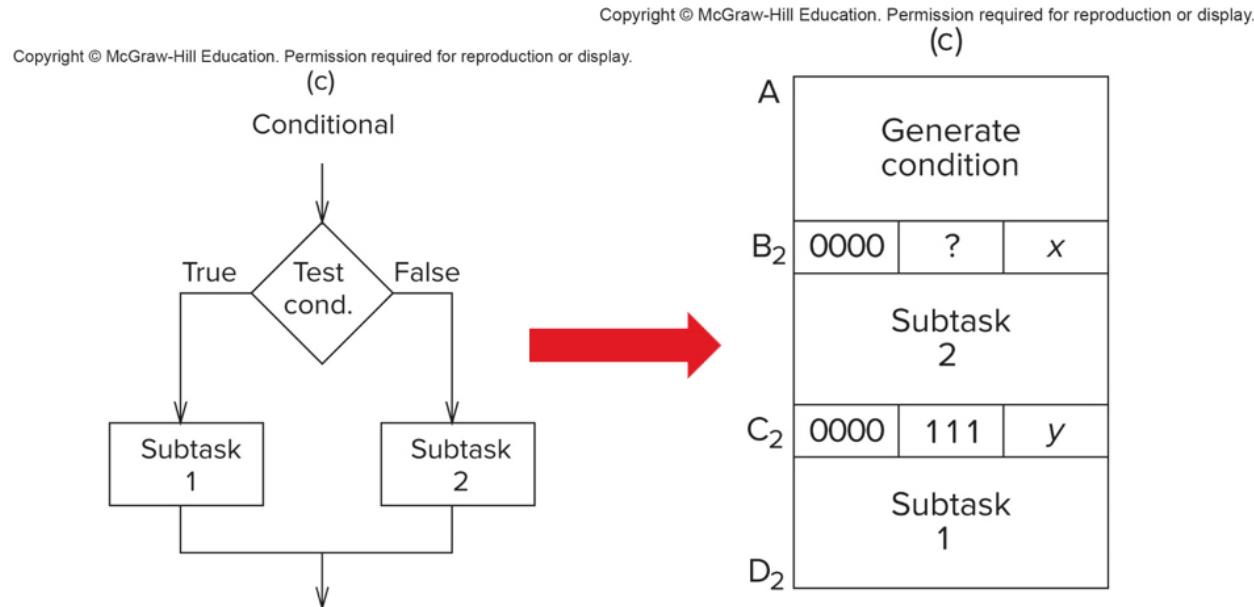
Instructions from A to B₁ for first subtask.

Instructions from B₁+1 to D₁ for second subtask.

[Access the text alternative for slide images.](#)

Implementing Constructs with LC-3 Instructions ₂

Conditional



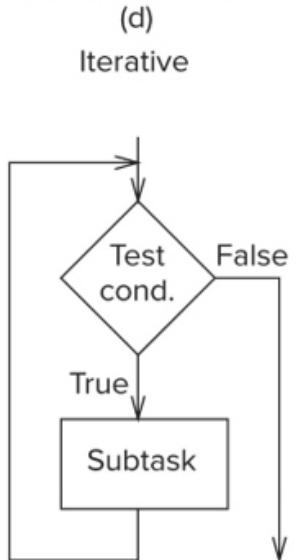
First (A), instructions that will set condition codes according to condition.
Branch at B₂ -- taken if condition is true, target = C₂+1 (subtask 1).
If condition is not true, branch is not taken, and subtask 2 (B₂+1...) is executed. After subtask 2, C₂ is unconditional branch to D₂+1, so that subtask 1 is not executed.

[Access the text alternative for slide images.](#)

Implementing Constructs with LC-3 Instructions 3

Iterative

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



Copyright © McGraw-Hill Education. Permission required for reproduction or display.

(d)

A	Generate condition		
B ₃	0000	?	z
Subtask			
D ₃	0000	111	w

First (A), instructions that will set condition codes according to condition.

Branch at B₃ -- taken if condition is false, target = D₃+1.

If condition is true, branch is not taken, and subtask (B₃+1...) is executed.
After subtask 2, D₃ is unconditional branch to A, to check the condition again.

[Access the text alternative for slide images.](#)

Debugging

Traditionally, a computer error is known as a "bug," and the process of finding and removing errors is known as "debugging."

How do you know there's an error?

- Run the program with a set of inputs for which the desired output is known or can be confirmed. (This is called "testing.") If you don't get the expected answer, there's an error. Testing with a variety of inputs is essential, because the execution of the program may change with different data -- big numbers, small numbers, negative numbers...
- Sometimes, the behavior of the program itself shows that there's an error. For example, the program runs "forever" without halting. Or a desired output is never displayed on the monitor.

Real-World Analogy

Suppose you want to drive from Raleigh, NC to Washington, DC.

Someone tells you to take I-40 East to I-95, and then go South on I-95, and that it should take about 4.5 hours.

You follow the directions, drive for 4.5 hours, and find yourself in Savannah, GA.

- Error detected -- Savannah is not Washington.
- What went wrong? Retrace your steps. Since there's only one major turn, chances are good that's where the problem is. Should have turned North on I-95, rather than South.
- Note that you (the computer) behaved perfectly. You did exactly what you were told to do. It was the instructions (the program) that was incorrect.

Steps for Debugging

Trace the program

Record the **sequence** of events as the program executes

Check the **results** of each sequence of events -- are they consistent with what you expect?

When you find a result that does not match your expectation (Why does that sign say "Welcome to South Carolina"?), you can narrow down the location of the bug.

Of course, once the bug is fixed, you have to test again.
There might be multiple bugs!!

Debugging Tools

A **debugger** is an environment for running your program that allows you to do the following:

1. Write values into registers and memory locations.
2. Execute a sequence of instructions.
3. Stop execution when desired (that is, before the program ends).
4. Examine values in memory and register at any point in the program.

For now, we will focus on tools that operate at the instruction level. Other tools will operate at higher levels of abstraction -- program statements and variables in a high-level language.

The **LC-3 simulator** provided for this class allows you to run interactively in debugging mode, providing the capabilities shown above.

Examining and Changing Values

The data for your program is stored in two locations:
memory and **registers**.

The simulator **displays** the values of all registers and provides a way to look at any memory location. You can also **change** the values stored in any of those locations.

In the context of the character counting program:

- Set the value of R0 to be an ASCII character of your choice, ignoring the user's input.
- Look at the characters stored in the file. Make sure there's an EOT somewhere that designates the end of file.
- Set the PC to start execution at a specific part of the program.

NOTE: Instructions are stored in memory, too, do you can change an instruction during the debugging session!

Executing a Sequence of Instructions

To debug, you want to (1) set memory and registers as desired, (2) run an "interesting" sequence of instructions, and (3) check the values in memory/registers. Then repeat (2) and (3) until you find something interesting.

There are two fundamental ways of controlling execution:

single-stepping allows you to execute one instruction at a time.

breakpoints allow you to specify a particular instruction where you want to stop execution.

Stepping vs. Breakpoints

Using breakpoints is much faster, because a sequence of instructions can be executed quickly by the simulator. It's a good way to quickly narrow down the likely location of a problem.

Set breakpoints between **modules** of your program -- for example: at the end of each task that you created when you designed the program.

Once you've identified a sequence of interest:

1. **Set a breakpoint** at the beginning of that sequence.
2. **Run** the program from the beginning, quickly executing instructions and stopping when the breakpoint is reached.
3. **Single-step** through one instruction at a time to see exactly what the processor is doing.

(Instead of 1 and 2, you can set the PC to the start of your sequence and set up registers/memory with known initial values.)

Debugging Examples

The best way to learn how to debug is to do it. You'll get plenty of practice, because it's rare to write a program correctly the first time!

We will show four examples of faulty programs, and show how debugging can find the errors. The examples illustrate several common types of errors:

1. Executing a loop the wrong number of times.
2. Using the wrong opcode.
3. Testing the wrong condition code.
4. Not considering all types of input values.

Example 1: Multiply

Program: Registers R4 and R5 contain positive integers. Multiply them and put the result in R2.

Code:

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x3200	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0
x3201	0	0	0	1	0	1	0	0	1	0	0	0	0	1	0	0
x3202	0	0	0	1	1	0	1	1	0	1	1	1	1	1	1	1
x3203	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	1
x3204	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1

Copyright © McGraw-Hill Education. Permission required for reproduction or display.

R2 <- 0
R2 <- R2 + R4
R5 <- R5 - 1
BRzp x3201
HALT

Each instruction has an annotation (comment) that tells what the instruction is supposed to do. These comments are not part of the program, but are extremely helpful as you are trying to interpret the execution.

Result: When R4 = 10 and R5 = 3, R2 gets a result of 40.

[Access the text alternative for slide images.](#)

Example 1: Multiply²

First step: Stare at the code for awhile to see if it makes sense.

Warning: Don't do this too long. It's generally much easier to debug by executing the code and watching its actual behavior. But a careful look to better understand the code is a good practice.

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3200	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	R2 <- 0
x3201	0	0	0	1	0	1	0	0	1	0	0	0	0	1	0	0	R2 <- R2 + R4
x3202	0	0	0	1	1	0	1	1	0	1	1	1	1	1	1	1	R5 <- R5 - 1
x3203	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	1	BRzp x3201
x3204	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	HALT

R4 gets added to itself multiple times, and R5 gets decreased each time. The loop ends when R5 is zero, so

$$R2 = 0 + R4 + R4 + R4 = 0 + 10 + 10 + 10 = 30$$

Looks reasonable -- why doesn't it work? Why do we get 40?

[Access the text alternative for slide images.](#)

Example 1: Multiply³

Next step: Trace execution of the code. We'll using **single-stepping**. The following table shows the PC and the state of the interesting registers after each instruction is executed.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.
(a)

PC	R2	R4	R5
x3201	0	10	3
x3202	10	10	3
x3203	10	10	2
x3201	10	10	2
x3202	20	10	2
x3203	20	10	1
x3201	20	10	1
x3202	30	10	1
x3203	30	10	0
x3201	30	10	0
x3202	40	10	0
x3203	40	10	-1
x3204	40	10	-1
	40	10	-1

When R5 = 3, the loop (x3201 to x3203) is executed four times. It should only execute three times to add R4 the correct number of times.

Error: We shouldn't re-execute the loop when R5 becomes zero. We should exit the loop.

Fix: Change BRzp to BRp.

Access the text alternative for slide images.

Example 1: Multiply⁴

Using Breakpoints

Because loop errors are very common, it's often enough to stop execution once per iteration, rather than stepping through every instruction. Here's the trace when we set a breakpoint at the BR instruction.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.

(b)

PC	R2	R4	R5
x3203	10	10	2
x3203	20	10	1
x3203	30	10	0
x3203	40	10	-1

Fewer stopping points means faster debugging and less chance to get lost in the details of the other instructions.

This allows us to quickly check: (a) how many times is the BR being taken? (b) what is the cause of the BR being taken / not taken?

Example 2: Add a Sequence of Numbers

Program: Add ten numbers starting at x3100, putting the result in R1.

Code:

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x3000	0	1	0	1	0	0	1	0	0	1	1	0	0	0	0	0
x3001	0	1	0	1	1	0	0	1	0	0	1	0	0	0	0	0
x3002	0	0	0	1	1	0	0	1	0	0	1	0	1	0	1	0
x3003	0	0	1	0	0	1	0	0	1	1	1	1	1	0	0	0
x3004	0	1	1	0	0	1	1	0	1	0	0	0	0	0	0	0
x3005	0	0	0	1	0	1	0	0	1	0	1	0	0	0	0	1
x3006	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1
x3007	0	0	0	1	1	0	0	1	0	0	1	1	1	1	1	1
x3008	0	0	0	0	0	0	1	1	1	1	1	1	1	0	1	1
x3009	1	1	1	1	0	0	0	0	0	0	0	1	0	0	1	0

R1 <- 0
R4 <- 0
R4 <- R4 + 10
R2 <- M[x3100]
R3 <- M[R2]
R2 <- R2 + 1
R1 <- R1 + R3
R4 <- R4 - 1
BRp x3004
HALT

Address	Contents
x3100	x3107
x3101	x2819
x3102	x0110
x3103	x0310
x3104	x0110
x3105	x1110
x3106	x11B1
x3107	x0019
x3108	x0007
x3109	x0004
x310A	x0000
x310B	x0000
x310C	x0000
x310D	x0000
x310E	x0000
x310F	x0000
x3110	x0000
x3111	x0000
x3112	x0000
x3113	x0000

Result:

When memory contains the values to the right,
result is x0024, but should be x8135.

[Access the text alternative for slide images.](#)

Example 2: Add a Sequence of Numbers 2

Step 1: Examine code -- what is the algorithm?

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3000	0	1	0	1	0	0	1	0	0	1	1	0	0	0	0	0	R1 <- 0
x3001	0	1	0	1	1	0	0	1	0	0	1	0	0	0	0	0	R4 <- 0
x3002	0	0	0	1	1	0	0	1	0	0	1	0	1	0	1	0	R4 <- R4 + 10
x3003	0	0	1	0	0	1	0	0	1	1	1	1	1	1	0	0	R2 <- M[x3100]
x3004	0	1	1	0	0	1	1	0	1	0	0	0	0	0	0	0	R3 <- M[R2]
x3005	0	0	0	1	0	1	0	0	1	0	1	0	0	0	0	1	R2 <- R2 + 1
x3006	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	R1 <- R1 + R3
x3007	0	0	0	1	1	0	0	1	0	0	1	1	1	1	1	1	R4 <- R4 - 1
x3008	0	0	0	0	0	0	1	1	1	1	1	1	1	0	1	1	BRp x3004
x3009	1	1	1	1	0	0	0	0	0	0	0	1	0	0	1	0	HALT

R1 (sum) initialized to zero. R4 (counter) initialized to 10.

R2 (address) initialized to x3100.

R3 gets value from M[R2], added to R1. Increment R2 for next address.
Decrement R4 and exit when counter is zero.

[Access the text alternative for slide images.](#)

Example 2: Add a Sequence of Numbers 3

Stepping through the code, we see the following:

Copyright © McGraw-Hill Education. Permission required for reproduction or display.

PC	R1	R2	R4
x3001	0	x	x
x3002	0	x	0
x3003	0	x	#10
x3004	0	x3107	#10

After x3004, R2 should contain x3100, the starting address of the numbers. Why does it contain x3107?

Opcode at x3004 is **LD**. Therefore, it loads the content of memory location x3100, which is x3107.

Instead, we need to use **LEA** to put the **address** x3100 into R2, not the **data** at M[x3100].

Example 3: Look for a Value in a Sequence 1

Program: Look for the value 5 in the sequence of ten numbers starting at address x3100. If the value is there, put 1 in R0; else put 0 in R0.

Code:

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x3000	0	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0
x3001	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	1
x3002	0	1	0	1	0	0	1	0	0	1	1	0	0	0	0	0
x3003	0	0	0	1	0	0	1	0	0	1	1	1	1	0	1	1
x3004	0	1	0	1	0	1	1	0	1	1	1	0	0	0	0	0
x3005	0	0	0	1	0	1	1	0	1	1	1	0	0	1	0	0
x3006	0	0	1	0	1	0	0	0	0	0	0	0	0	1	0	1
x3007	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0
x3008	0	0	0	1	0	1	0	0	1	0	0	0	0	0	0	1
x3009	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0
x300A	0	0	0	1	1	0	0	1	0	0	1	0	0	0	0	1
x300B	0	0	0	1	0	1	1	0	1	1	1	1	1	1	1	1
x300C	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0
x300D	0	0	0	0	0	0	1	1	1	1	1	1	1	0	1	0
x300E	0	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0
x300F	1	1	1	1	0	0	0	0	0	0	0	1	0	0	1	0
x3100	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0

Copyright © McGraw-Hill Education. Permission required for reproduction or display.

R0 <- 0
R0 <- R0 + 1
R1 <- 0
R1 <- R1 - 5
R3 <- 0
R3 <- R3 + 10
R4 <- M[x3010]
R2 <- M[R4]
R2 <- R2 + R1
BRz x300F
R4 <- R4 + 1
R3 <- R3 - 1
R2 <- M[R4]
BRp x3008
R0 <- 0
HALT
x3100

[Access the text alternative for slide images.](#)

Example 3: Look for a Value in a Sequence 2

Result:

When tested with data that includes one or more 5's, R0 set to 0.

Algorithm:

R0 = 1 -- assuming that we will see a 5

R1 = -5 -- used to test if value = 5

R3 = 10 -- counter

R4 = x3100 -- starting address

Load M[R4] into R2.

LOOP:

Test if R2 equals 5. If so, branch to x300F -- halt (with R1=1).

Increment R4 to next address. Decrement loop counter (R3).

Load M[R4] into R2. Branch to back to LOOP if R3 not zero.

Set R0=0 -- counter hit zero before seeing any 5's.

Example 3: Look for a Value in a Sequence 3

Learning from our earlier loop bug, we set a breakpoint at the bottom of the loop, to see how many times it executes.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.

PC	R1	R2	R3	R4
x300D	-5	7	9	3101
x300D	-5	32	8	3102
x300D	-5	0	7	3103

After three breakpoints, the execution does not hit this BR again, even though R3 is 7.

Why?? R2 shows that we did not load 5.

Looking more closely, the simulator shows that the Z bit is set at the last breakpoint. Why? Because the LD at x300C loaded a zero into R2. We were thinking the BR would be based on the previous ADD instruction (decrementing R3) and forgot that LD also sets the condition codes.

Fix: Move the LD before the decrement of R3, so that the condition code is set by the relevant instruction.

Example 4: Find the First 1 in a Word

Program: Load a data word from a far-away memory location (x3400). From left to right, look for the first one bit in the value, and put the bit position of that one in R1. If no 1's, make R1 = -1.

Code:

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x3000	0	1	0	1	0	0	1	0	0	1	1	0	0	0	0	0
x3001	0	0	0	1	0	0	1	0	0	1	1	0	1	1	1	1
x3002	1	0	1	0	0	1	0	0	0	0	0	0	0	1	1	0
x3003	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0
x3004	0	0	0	1	0	0	1	0	0	1	1	1	1	1	1	1
x3005	0	0	0	1	0	1	0	0	1	0	0	0	0	1	0	0
x3006	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1
x3007	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0
x3008	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1
x3009	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0

Copyright © McGraw-Hill Education. Permission required for reproduction or display.

R1 <- 0
R1 <- R1 + 15
R2 <- M[M[x3009]]
BRn x3008
R1 <- R1 - 1
R2 <- R2 + R2
BRn x3008
BRnzp x3004
HALT
x3400

Result: Program usually works, but sometimes fails to terminate.

[Access the text alternative for slide images.](#)

Example 4: Find the First 1 in a Word

Copyright © McGraw-Hill Education. Permission required for reproduction or display.

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x3000	0	1	0	1	0	0	1	0	0	1	1	0	0	0	0	0
x3001	0	0	0	1	0	0	1	0	0	1	1	0	1	1	1	1
x3002	1	0	1	0	0	1	0	0	0	0	0	0	0	1	1	0
x3003	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0
x3004	0	0	0	1	0	0	1	0	0	1	1	1	1	1	1	1
x3005	0	0	0	1	0	1	0	0	1	0	0	0	0	0	1	0
x3006	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1
x3007	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0
x3008	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1
x3009	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0

```
R1 <- 0
R1 <- R1 + 15
R2 <- M[M[x3009]]
BRn x3008
R1 <- R1 - 1
R2 <- R2 + R2
BRn x3008
BRnzp x3004
HALT
x3400
```

Algorithm:

R1 = 15, leftmost bit position

R2 = data value (memory address is in x3009, using LDI to load M[x3400])

If value is negative, then 15 is the first 1 --> branch to HALT.

LOOP:

Decrement R1 (next bit position). Shift value (R2) to the left.

If negative, then a 1 was shifted into position 15 --> exit loop.

[Access the text alternative for slide images.](#)

Example 4: Find the First 1 in a Word 3

Set breakpoint at bottom of loop, to see how many times it's executed.

Loop should execute at most 16 times. Why does it keep going?

Copyright © McGraw-Hill Education. Permission required for reproduction or display.

PC	R1
x3007	14
x3007	13
x3007	12
x3007	11
x3007	10
x3007	9
x3007	8
x3007	7
x3007	6
x3007	5
x3007	4
x3007	3
x3007	2
x3007	1
x3007	0
x3007	-1
x3007	-2
x3007	-3
x3007	-4

Problem: A value of zero is loaded into R2. Therefore, a 1 is never shifted into the sign bit. Therefore, the loop is never exited -- INFINITE LOOP.

Fix: Add a check for zero when value is first loaded. Exit and set R1 = -1.

NOTE: This only fails when value is zero. If you didn't happen to test that case, then your customer will find the bug instead!

[Access the text alternative for slide images.](#)

Lessons

Use breakpoints to observe the behavior of the program, paying special attention to register values that affect control flow.

Single-step through code as needed, to understand why register are getting their value. (Wrong opcode, wrong operand, ...?)

Test with a variety of input data.

- Especially look for corner cases that may behave differently, such as negative numbers, or zero.

Don't assume that comments are correct!

- Comments express what the programmer intended, but only the instructions determine how the program actually behaves.



Because learning changes everything.[®]

www.mheducation.com

From Bits and Gates to C and Beyond

Assembly Language

Chapter 7

Human-Friendly Programming

Computers need binary instruction encodings...

0001110010000110

Humans prefer symbolic languages...

a = b + c

High-level languages allow us to write programs in clear, precise language that is more like English or math. Requires a program (compiler) to transalte from symbolic language to machine instructions.

Examples: C, Python, Fortran, Java, ...

We will introduce C in Chapters 11 to 19.

Assembly Language: Human-Friendly ISA Programming

Assembly Language is a low-level symbolic language, just a short step above machine instructions.

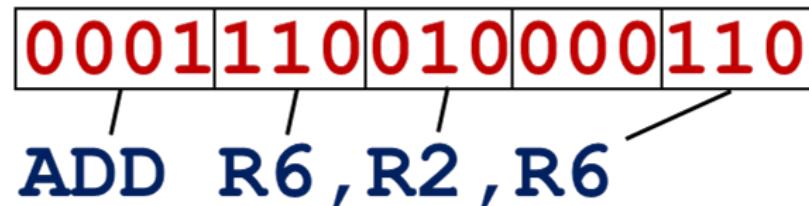
- Don't have to remember opcodes (ADD = 0001, NOT = 1001, ...).
- Give symbolic names to memory locations -- don't have to do binary arithmetic to calculate offsets.
- Like machine instructions, allows programmer explicit, instruction-level specification of program.

Disadvantage:

Not portable. Every ISA has its own assembly language.
Program written for one platform does not run on another.

Assembly Language

Very similar format to instructions -- replace bit fields with symbols.



For the most part, one line of assembly language = one instruction.

Some additional features for allocating memory, initializing memory locations, service calls.

Numerical values specified in hexadecimal (x30AB) or decimal (#10).

[Access the text alternative for slide images.](#)

Example Program

```
;  
; Program to multiply a number by the constant 6  
;  
    .ORIG x3050  
    LD    R1, SIX  
    LD    R2, NUMBER  
    AND   R3, R3, #0      ; Clear R3. It will  
                        ; contain the product.  
;  
; The inner loop  
;  
AGAIN  ADD   R3, R3, R2  
      ADD   R1, R1, #-1    ; R1 keeps track of  
      BRp  AGAIN           ; the iteration.  
;  
HALT  
;  
NUMBER .BLKW 1  
SIX    .FILL x0006  
;  
.END
```

Labels

Instructions

Comments

Assembler Directives

[Access the text alternative for slide images.](#)

Assembly Language Syntax

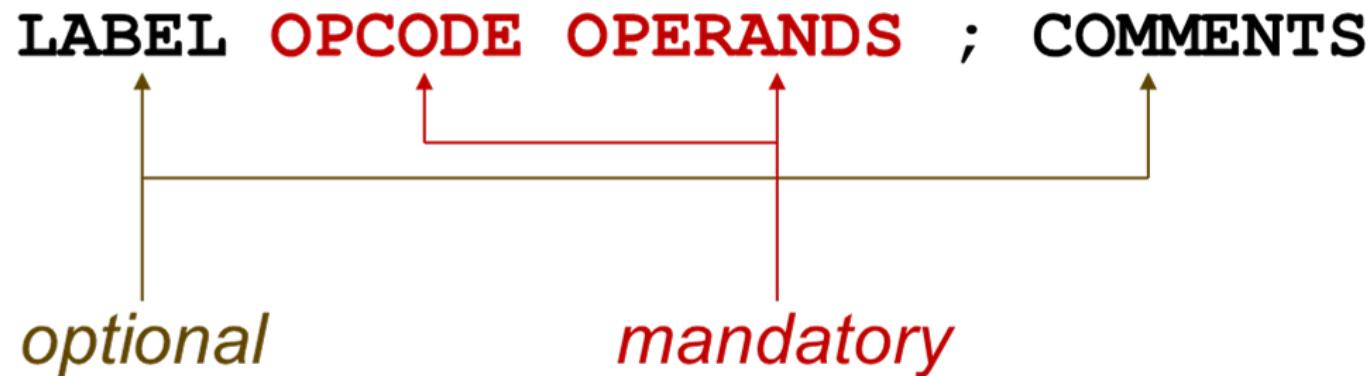
Each line of a program is one of the following:

- An instruction.
- An assembler directive (or pseudo-op).
- A comment.

Whitespace (between symbols) and case are ignored.

Comments (beginning with “;”) are also ignored.

An instruction has the following format:



[Access the text alternative for slide images.](#)

Mandatory: Opcode and Operands

Opcodes

Reserved symbols that correspond to LC-3 instructions.

Listed in Appendix A and Figure 5.3.

- For example: ADD, AND, LD, LDR, ...

reserved means that it cannot be used as a label

Operands

- Registers -- specified by Rn, where n is the register number.
- Numbers -- indicated by # (decimal) or x (hex).
- Label -- symbolic name of memory location.
- Separated by comma (whitespace ignored).
- Number, order, and type correspond to instruction format.

```
ADD R1,R1,R3      ; DR, SR1, SR2
ADD R1,R1,#3       ; DR, SR1, Imm5
LD  R6,NUMBER      ; DR, address (converted to PCoffset)
BRz LOOP           ; nzp becomes part of opcode, address
```

Optional: Label and Comment

Label

- Placed at the beginning of the line.
- Assigns a symbolic name to the address corresponding to that line.

```
LOOP      ADD      R1, R1, #-1      ; LOOP is address of ADD  
          BRp      LOOP
```

Comment

A semicolon, and anything after it on the same line, is a comment.

Ignored by assembler.

Used by humans to document/understand programs.

Tips for useful comments:

- Avoid restating the obvious, as “decrement R1.”
- Provide additional insight, as in “accumulate product in R6.”
- Use comments and empty lines to separate pieces of program.

Assembler Directive

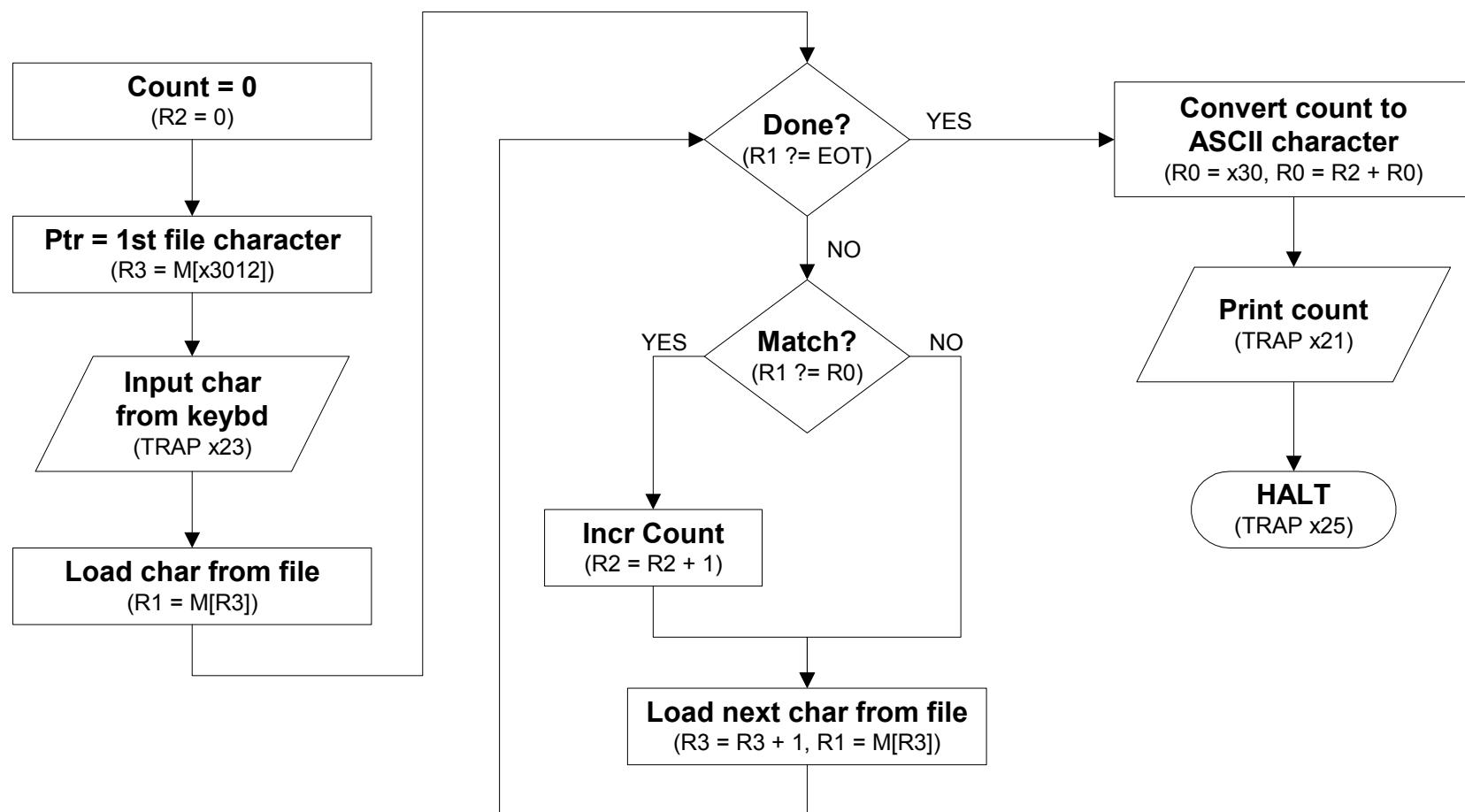
Pseudo-operation

- Does not refer to an actual instruction to be executed.
- Tells the assembler to do something.
- Looks like an instruction, except "opcode" starts with a dot.

<i>Opcode</i>	<i>Operand</i>	<i>Meaning</i>
.ORIG	address	starting address of program
.END		end of program
.BLKW	n	allocate n words of storage
.FILL	n	allocate one word, initialize with value n
.STRINGZ	n-character string	allocate n+1 locations, initialize w/ characters and null terminator

Sample Program: Counting Occurrences in a File

Once again, we show the program that counts the number of times (up to nine) a user-specified character appears in a file.



[Access the text alternative for slide images.](#)

Assembly Language Program 1

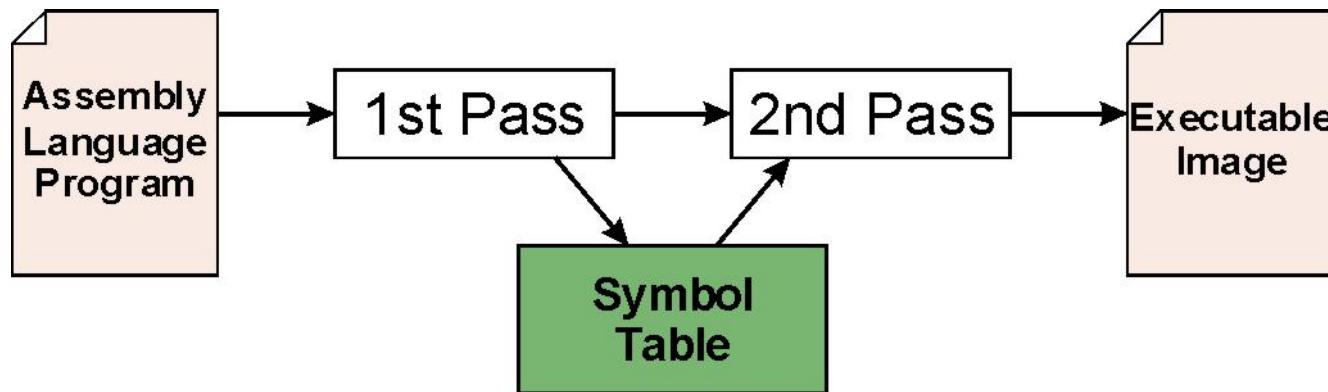
```
;  
; Program to count occurrences of a character in a file.  
; Character to be input from the keyboard.  
; Result to be displayed on the monitor.  
; Program only works if no more than 9 occurrences are found.  
;  
;  
; Initialization  
;  
    .ORIG    x3000  
    AND      R2, R2, #0          ; R2 is counter, initially 0  
    LD       R3, PTR            ; R3 is pointer to characters  
    TRAP    x23                ; R0 gets character input  
    LDR     R1, R3, #0          ; R1 gets first character  
;  
; Test character for end of file  
;  
TEST     ADD      R4, R1, #-4      ; Test for EOT (ASCII x04)  
        BRz     OUTPUT            ; If done, prepare the output  
;  
; Test character for match.  If a match, increment count.  
;  
    NOT      R1, R1  
    ADD      R1, R1, #1  
    ADD      R1, R1, R0          ; Compute R0-R1 to compare  
    BRnp    GETCHAR            ; If no match, do not increment count  
    ADD      R2, R2, #1
```

Assembly Language Program 2

```
;  
; Get next character from file.  
;  
GETCHAR    ADD      R3, R3, #1      ; Point to next character.  
            LDR      R1, R3, #0      ; R1 gets next char to test  
            BRnzp   TEST  
;  
; Output the count.  
;  
OUTPUT     LD       R0, ASCII      ; Load the ASCII template  
            ADD      R0, R0, R2      ; Covert binary count to ASCII  
            TRAP    x21          ; ASCII code in R0 is displayed.  
            TRAP    x25          ; Halt machine  
;  
; Storage for pointer and ASCII template  
;  
ASCII      .FILL    x0030  
PTR       .FILL    x4000  
.END
```

Assembly Process

The assembler is a program that translates an assembly language (.asm) file to a binary object (.obj) file that can be loaded into memory.



First Pass:

- Scan program file, check for syntax errors.
- Find all labels and calculate the corresponding addresses: the symbol table.

Second Pass:

- Convert instructions to machine language, using information from symbol table.

[Access the text alternative for slide images.](#)

First Pass: Construct the Symbol Table

1. Find the .ORIG statement,
which tells us the address of the first instruction.
 - Initialize location counter (LC), which keeps track of the current instruction.
2. For each non-empty line in the program:
 - If line contains a label, add label and LC to symbol table.
 - Increment LC.
 - NOTE: If statement is .BLKW or .STRINGZ, increment LC by the number of words allocated.
3. Stop when .END statement is reached.

NOTE: A line that contains only a comment is considered an empty line.

First Pass on Sample Program (Comments Removed)

```
--          .ORIG    x3000
x3000      AND      R2, R2, #0
x3001      LD       R3, PTR
x3002      TRAP    x23
x3003      LDR     R1, R3, #0
x3004      TEST     ADD    R4, R1, #-4
x3005      BRz     OUTPUT
x3006      NOT     R1, R1
x3007      ADD     R1, R1, #1
x3008      ADD     R1, R1, R0
x3009      BRnp    GETCHAR
x300A      ADD     R2, R2, #1
x300B      GETCHAR ADD    R3, R3, #1
x300C      LDR     R1, R3, #0
x300D      BRnzp   TEST
x300E      OUTPUT   LD     R0, ASCII
x300F      ADD     R0, R0, R2
x3010      TRAP    x21
x3011      TRAP    x25
x3012      ASCII    .FILL  x0030
x3013      PTR     .FILL  x4000
--          .END
```

Label	Address
TEST	x3004
GETCHAR	x300B
OUTPUT	x300E
ASCII	x3012
PTR	x3013

Second Pass: Convert to Machine Instructions

1. Find the .ORIG statement,
which tells us the address of the first instruction.
 - Initialize location counter (LC), which keeps track of the current instruction.
2. For each non-empty line in the program:
 - If line contains an instruction, translate opcode and operands to binary machine instruction. For label, lookup address in symbol table and subtract (LC+1). Increment LC.
 - If line contains .FILL, convert value/label to binary. Increment LC.
 - If line contains .BLKW, create n copies of x0000 (or any arbitrary value). Increment LC by n.
 - If line contains .STRINGZ, convert each ASCII character to 16-bit binary value. Add null (x0000). Increment LC by n+1.
3. Stop when .END statement is reached.

Errors during Code Translation

While assembly language is being translated to machine instructions, several types of errors may be discovered.

- Immediate value too large -- can't fit in Imm5 field.
- Address out of range -- greater than LC+1+255 or less than LC+1-256.
- Symbol not defined, not found in symbol table.

If error is detected, assembly process is stopped and an error message is printed for the user.

Beyond a Single Object File

Larger programs may be written by multiple programmers, or may use modules written by a third party. Each module is assembled independently, each creating its own **object file** and **symbol table**.

To execute, a program must have all of its modules combined into a single **executable** image.

Linking is the process to combine all of the necessary object files into a single executable.

External Symbols

In the assembly code we're writing, we may want to symbolically refer to information defined in a different module.

For example, suppose we don't know the starting address of the file in our counting program. The starting address and the file data could be defined in a different module.

We want to do this:

```
PTR    .FILL   STARTOfFile
```

To tell the assembler that `STARTOfFile` will be defined in a different module, we could do something like this:

```
.EXTERNAL  STARTOfFile
```

This tells the assembler that it's not an error that `STARTOfFile` is not defined. It will be up to the linker to find the symbol in a different module and fill in the information when creating the executable.



Because learning changes everything.[®]

www.mheducation.com

From Bits and Gates to C and Beyond

Data Structures

Chapter 8

Data Structures

Basic data types (Chapter 2): integer, floating-point number, character

Each represents a single value, typically using 1 to 2 words of memory.

Often directly supported by machine instructions.

Data structure = a more complex data item, or a collection of items, composed of the basic types

- Examples: company organization chart, music playlist, schedule of courses, etc.
- Array = sequence of values, like the character file in earlier chapters
- Stack = Last-in, First-out (LIFO) sequence
- Queue = First-in, First-out (FIFO) sequence
- String = sequence of characters

Subroutine: Reusable Module

Before implementing data structures, we introduce the **subroutine** -- a mechanism for writing **reusable code modules**. These will be useful for implementing abstract operations on data structures.

There are different names for this concept in various languages, including **functions** (C) and **methods** (C++, Java).

Main idea:

We want to write code once, and use it many times.

We need a way to transfer control to the subroutine, possibly giving it some data to work on -- this is a **subroutine call**.

When the subroutine has finished its work, we need a way to get back to the program who called it -- this is the **return**.

We will show the call and return mechanisms provided by the LC-3.

Example: Convert Repeated Code to a Subroutine

```
Copyright © McGraw-Hill Education. Permission required for reproduction or display.  
01 START ST R1.SaveR1 ; Save registers needed  
02 ST R2.SaveR2 ; by this routine  
03 ST R3.SaveR3  
04 :  
05 LD R2,Newline  
06 L1 LDI R3,DSR  
BRzp L1 : Loop until monitor is ready  
08 STI R2,DDR : Move cursor to new clean line  
09 :  
0A LEA R1,Prompt : Starting address of prompt string  
0B Loop LDR R0,R1,#0 : Write the input prompt  
0C BRz Input : End of prompt string  
0D L2 LDI R3,DSR  
BRzp L2 : Loop until monitor is ready  
0F STI R0,DDR : Write next prompt character  
10 ADD R1,R1,#1 : Increment prompt pointer  
11 BRnzp Loop : Get next prompt character  
12 :  
13 Input LDI R3,KBSR  
BRzp Input : Poll until a character is typed  
15 LDI R0,KBDR : Load input character into R0  
16 L3 LDI R3,DSR  
BRzp L3 : Loop until monitor is ready  
18 STI R0,DDR : Echo input character  
19 :  
20 L4 LDI R3,DSR  
BRzp L4 : Loop until monitor is ready  
21 STI R2,DDR : Move cursor to new clean line  
22 LD R1,SaveR1 : Restore registers  
23 LD R2,SaveR2 : to original values  
24 LD R3,SaveR3  
25 JMP R7 : Do the program's next task  
26 :  
27 SaveR1 .BLKW 1 : Memory for registers saved  
28 SaveR2 .BLKW 1  
29 SaveR3 .BLKW 1  
30 DSR .FILL xFE04  
31 DDR .FILL xFE06  
32 KBSR .FILL xFE00  
33 KBDR .FILL xFE02  
34 Newline .FILL x000A : ASCII code for newline  
35 Prompt .STRINGZ "Input a character>"
```

Here is some code from Figure 8.1. Will be discussed in Chapter 9, so don't worry about what it does, but notice the repetitive highlighted code.

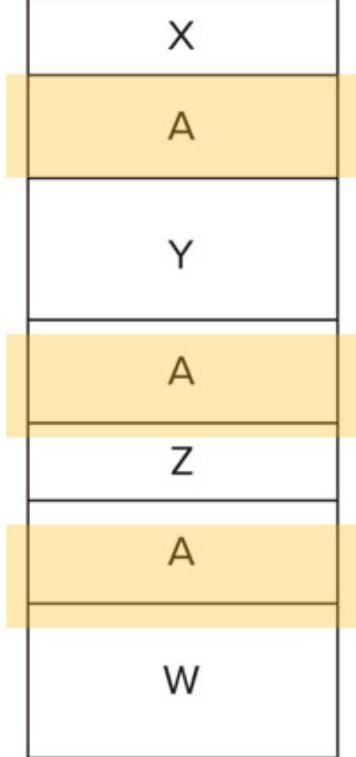
Each uses a different label and different registers, but can be generalized as:

label	LDI	R3,DSR
	BRzp	Label1
	STI	Reg,DDR

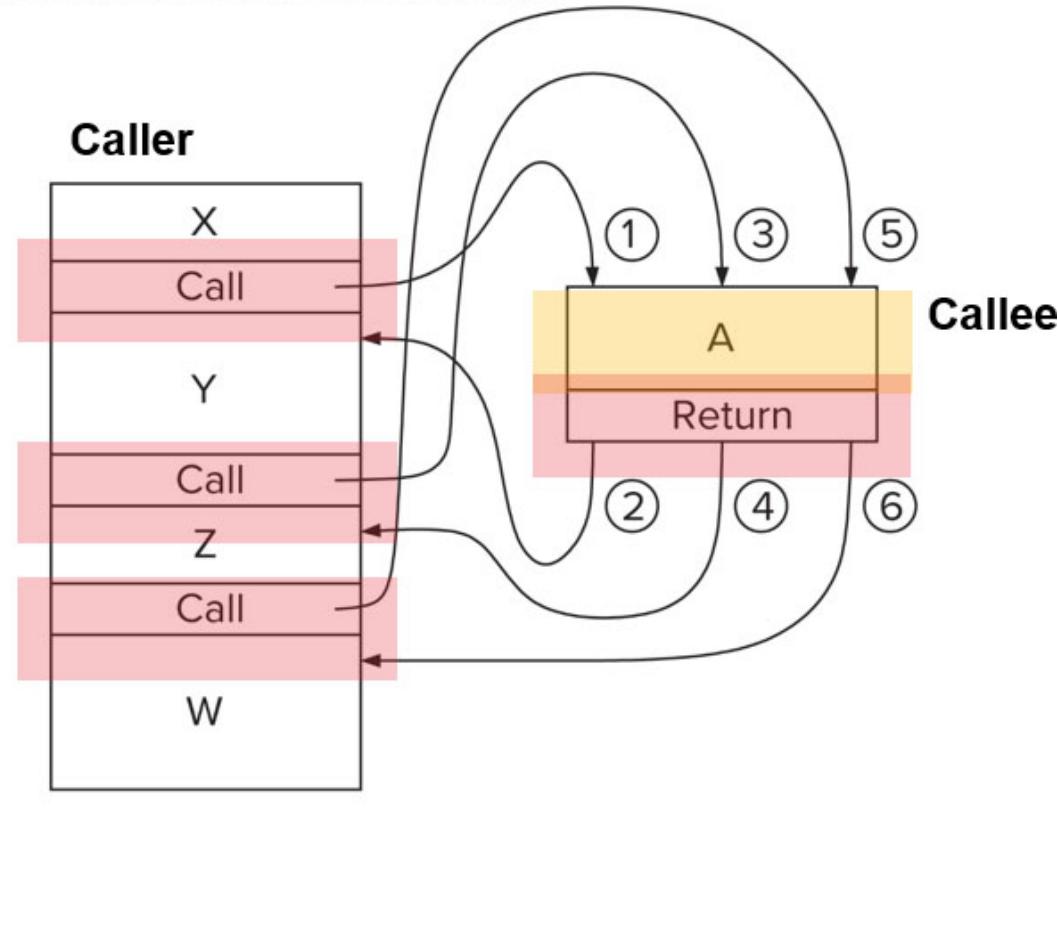
Wouldn't it be nice to write that code once and use it whenever we need it, rather than writing it four times?

Convert Repeated Code to a Subroutine

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



(a) Without subroutines



(b) With subroutines

Call + return allow the same code to be executed from multiple places in the program.

[Access the text alternative for slide images.](#)

Advantages of Subroutines

Reusability

- Only have to write (and debug!) the code once.
- If you find an error, you can fix it in one place, rather than looking for all the places in the code where you did the same thing.

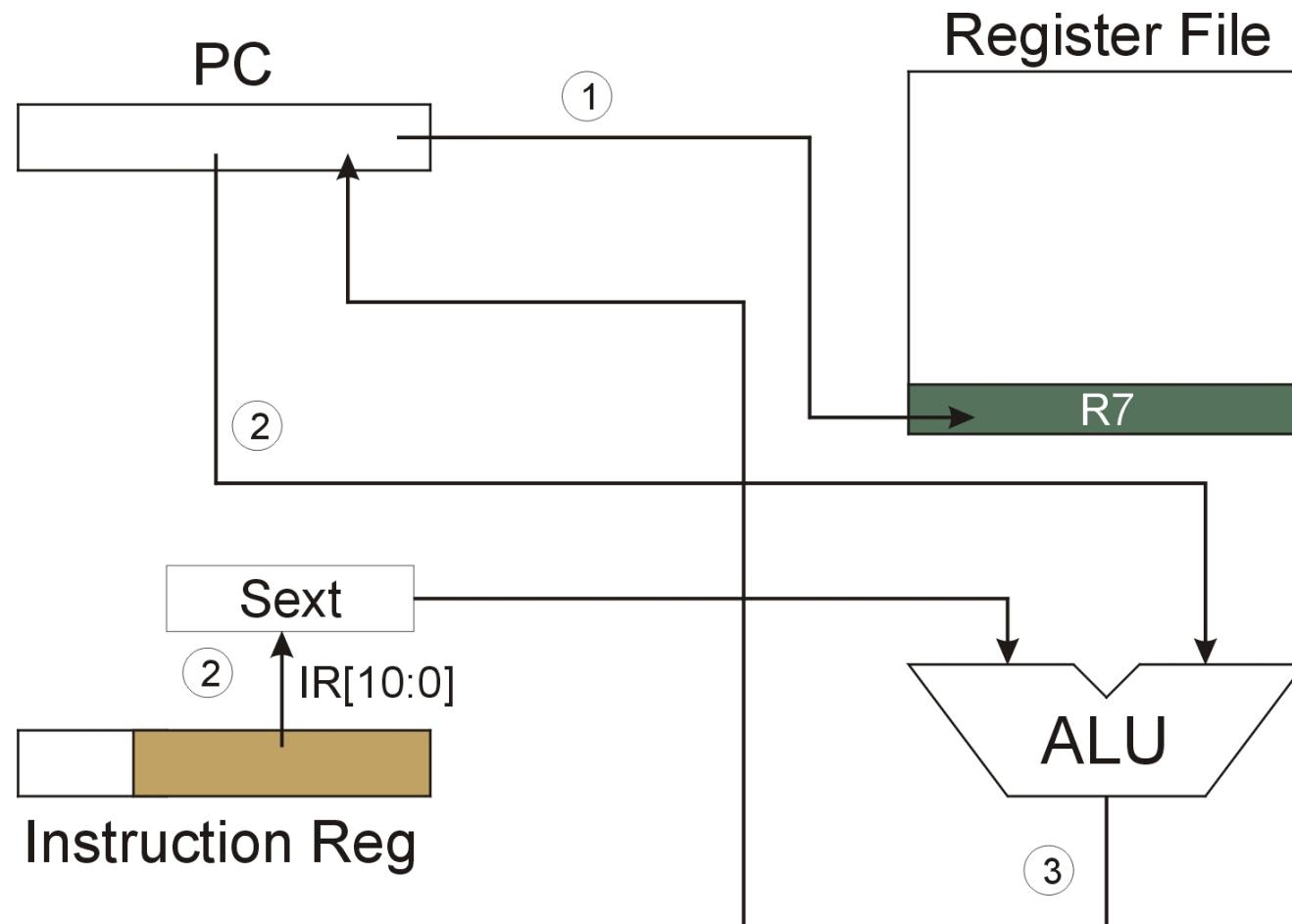
Abstraction

- Encapsulate a complex task (for example, "square root") into a subroutine with a symbolic name (SQRT).
- Long sequence of instructions is replaced with CALL SQRT.
- Program becomes easier to read and understand, and the reader does not get lost in the details of how to compute a square root.

Modularity

- Can divide program up into smaller pieces and assign to different programmers, or solve on different days.

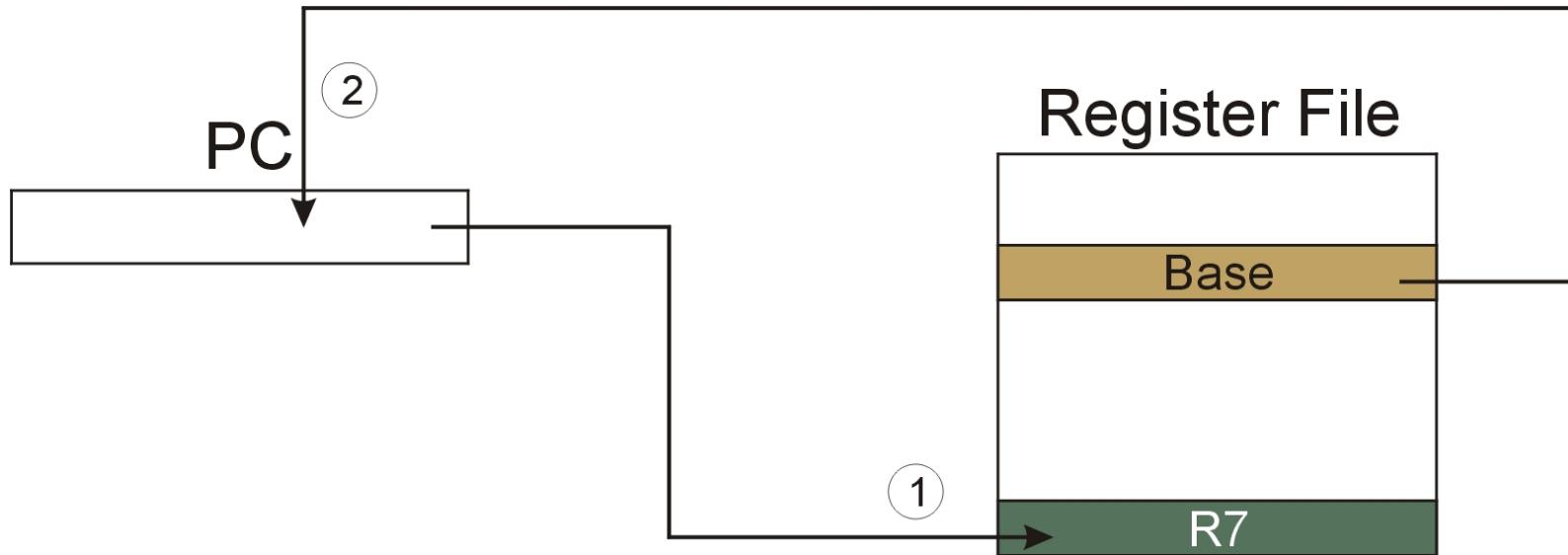
JSR (PC-Relative mode)



[Access the text alternative for slide images.](#)

JSRR (Base+offset (zero) mode)

JSRR	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	0	0	0	0		Base	0	0	0	0	0	0	0



Using a base register allows subroutine to be anywhere in memory.
Same opcode as JSR, but different assembly mnemonic (JSRR).
Bit 11 distinguishes between the two addressing modes.

[Access the text alternative for slide images.](#)

Returning from a Subroutine

We must be careful to not change R7 during the execution of the subroutine. When we are ready to return, we want to put R7's value (the saved PC) back into the PC. We already have an instruction for that!

JMP R7

Because this occurs a lot, and because using R7 as the base register has a special meaning for subroutines, the assembler provides an alternate syntax:

RET

NOTE that RET is listed in the instruction set in Chapter 5, but it's not a separate instruction. It's just a special case of JMP.

Calling a Subroutine

For the LC-3, we already know how to transfer control to another piece of code. We could use BR (if close) or JMP (if far away).

But how do we get back?

Once we've changed the PC, we've lost track of where in memory the calling code resides.

Solution: A new instruction **JSR** that **saves a copy of the PC** before changing it.

- If we save a copy of the PC, we can simply write that value back into the PC to get back to the calling code. Remember: PC has already been incremented (during Fetch), so we are returning to the instruction after the JSR.
- LC-3 uses R7 to store the saved PC.

Saving and Restoring Registers

When we write to a register, its previous value is gone forever.

If we need that value later, we must **save** it somewhere else, before overwriting its register. Once we've saved it, we can **restore** it (put it back into the register) when we need it again.

Where do we save it?

- Copy it to another register.
OK, but there are only a few registers in the LC-3, so that's not likely to be a general solution.
- **Store it to a memory location.**
Yes -- use .BLKW to reserve a memory location in the program where we can store a register's value to save it, and we can load it when we need to restore.

Saving and Restore Registers in Subroutines

A subroutine is separate and independent from the caller code.
The subroutine will need to use registers to perform its job, but it does not know which registers have information needed by the caller.

Therefore...

A subroutine should **save** any register that it changes, and **restore** that register to the caller's value before it returns.

This is known as a callee-save strategy, because it's the callee code that is responsible for saving/restoring registers.

One exception: The *caller* must save R7 (if needed) before the call, because JSR changes R7 before the callee code is executed.

Interface: Passing Data Into and Out of a Subroutine

We often need to give information to the subroutine.

Example: If we have a square root subroutine (SQRT), how does the subroutine know what value to take the square root of?

This is known as "passing" data into a subroutine.

Likewise, there is often information that we want back from the subroutine.

Example: If we call SQRT, how do we get the computed value?

This is known as returning information from the subroutine, or passing data out.

The scheme for passing data into and out of a subroutine is known as its **interface**. It must be documented and followed by every caller to this subroutine.

Passing Data in the LC-3

The JSR(R) instruction only provides a way to transfer control to the subroutine -- it does not provide any means of transferring data in and out.

In the LC-3, there are only two places where data can be stored:

- Registers (R0 to R6) -- can't use R7, because it will contain the saved PC.
- Memory.

Either will work, and there are pros and cons to each approach.

Generally, for a small amount of data, registers are easy to use. For large collections of data, we can pass a pointer (address) that gives the location of the data.

The **interface** tells how the transfer of data will work, and both the caller and the callee must follow the interface.

Example Subroutine: Summing an Array of Integers 1

```
; Subroutine: ArraySum  
; Computes the sum of a sequence of integers  
; Inputs:  
;   R0 = pointer to the starting address of the array  
;   R1 = number of integers in the array (size)  
; Output:  
;   R2 = computed sum
```

```
ArraySum    ST    R0,ASumR0      ; save registers  
            ST    R1,ASumR1  
            ST    R3,ASumR3  
  
            AND   R2,R2,#0      ; initialize sum to zero  
  
            ADD   R1,R1,#0      ; check size -- if <= 0, exit  
            BRnz ASDone
```

Example Subroutine: Summing an Array of Integers 2

```
ASLoop      LDR  R3,R0,#0      ; load value from array
            ADD  R2,R2,R3      ; add to computed sum
            ADD  R0,R0,#1      ; point to next value
            ADD  R1,R1,#-1     ; until counter is zero
            BRp ASLoop

; output value (sum) is in R2, ready to return

ASDone      LD   R3,ASumR3    ; restore registers
            LD   R1,ASumR1
            LD   R0,ASumR0
            RET

ASumR0     .BLKW 1          ; space for saved registers
ASumR1     .BLKW 1
ASumR3     .BLKW 1
```

Library Subroutines

It's common for a collection of related subroutines to be packaged together into a **library**. The programmer can then use the subroutines, according to the documented interface, without concern about the details of implementation.

When symbols (for example, subroutine names) are defined in modules, the **EXTERNAL** mechanism discussed in Chapter 7 can be used.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.

```
01      ...
02      ...
03      .EXTERNAL SQRT
04      ...
05      ...
06      LD      R0,SIDE1
07      BRZ    S1
08      JSR    SQUARE
09      S1    ADD    R1,R0,#0
0A      LD      R0,SIDE2
0B      BRZ    S2
0C      JSR    SQUARE
0D      S2    ADD    R0,R0,R1 : R0 contains argument x
0E      LD      R4,BASE ; BASE contains starting address of SQRT routine
0F      JSRR
0G      R4
0H      ST      R0,HYPOT
0I      BRnzp NEXT_TASK
0J      SQUARE ADD    R2,R0,#0
0K      ADD    R3,R0,#0
0L      AGAIN ADD    R2,R2,#-1
0M      BRz    DONE
0N      ADD    R0,R0,R3
0O      BRnzp AGAIN
0P      DONE   RET
0Q      BASE   .FILL  SQRT
0R      SIDE1 .BLKW  1
0S      SIDE2 .BLKW  1
0T      HYPOT .BLKW  1
0U      ...
0V      ...
```

Abstract Data Types

Now, we're ready to talk about data structures.

Each data structure is described as an abstract data type.

Instead of simply describing how data is stored in memory, we specify **operations** on the data structure, implemented as subroutines.

By using the subroutines, the caller program is separated from the detailed implementation and storage details. This provides a level of abstraction -- if the implementation changes, the caller does not need to be changed.

We will show an implementation of each data structure, but keep in mind that there are alternatives -- a topic for another class.

Stack

A stack is a **last-in, first-out (LIFO)** sequence of data items.

The **last** thing you put in the stack will be the **first** thing removed.

The operations defined for a stack are:

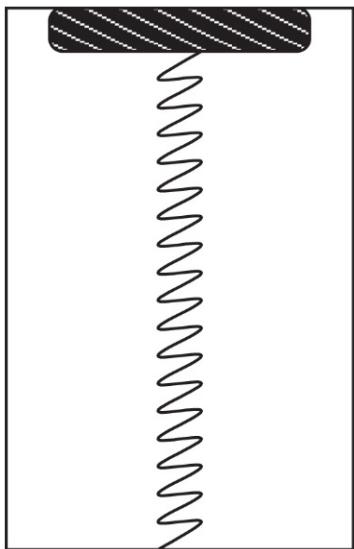
push an item onto the stack, and

pop an item from the stack.

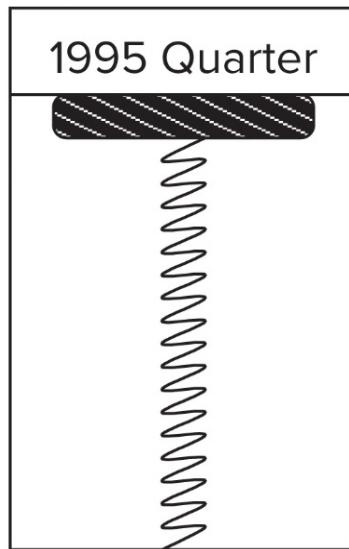
Stack: Coin Holder Analogy

Consider a coin holder that contains quarters. We identify each coin by the year it was minted.

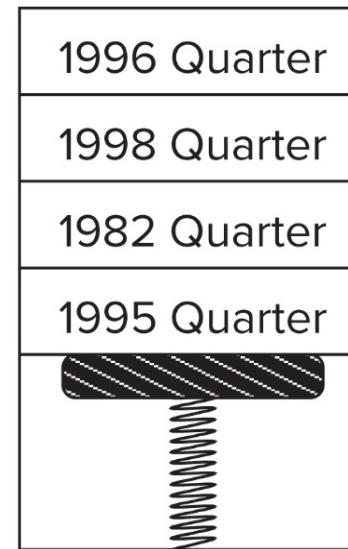
Copyright © McGraw-Hill Education. Permission required for reproduction or display.



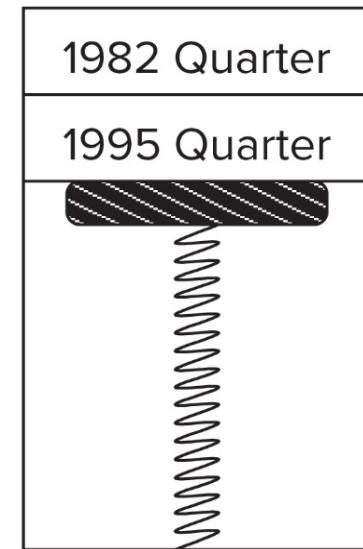
(a) Initial state
(Empty)



(b) After one push



(c) After three pushes



(d) After two pops

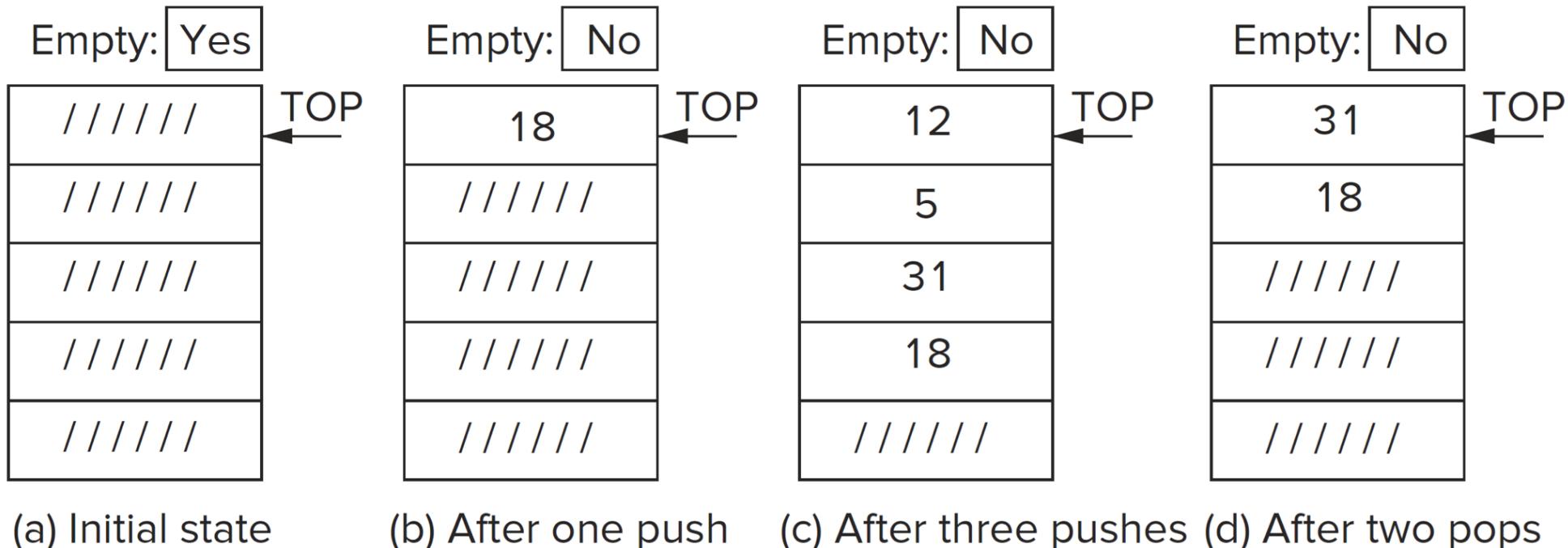
Coins are popped in the opposite order from when they are pushed:
Last-in, First-out.

[Access the text alternative for slide images.](#)

Digital Hardware Stack

If we want to implement a stack of integers using digital hardware, we can provide a number of registers and move data between the registers on each push/pop operation.

We use an additional register to keep track of whether the stack is empty.

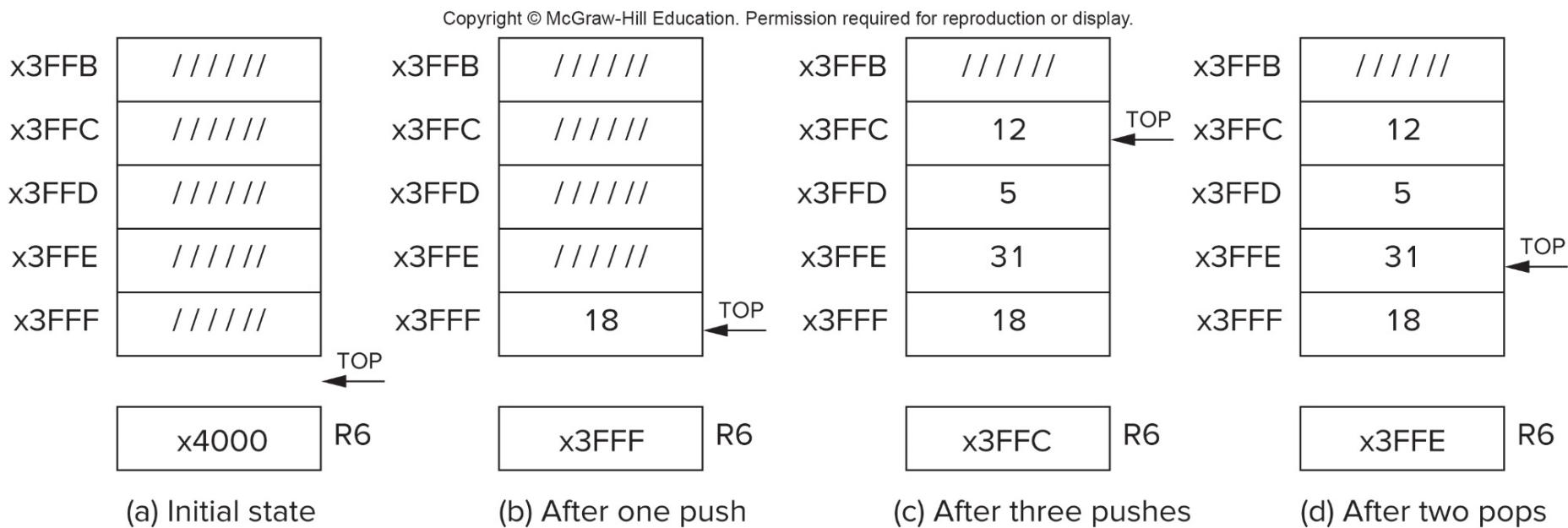


[Access the text alternative for slide images.](#)

Software Stack in Memory

If we use the same approach in software, where the stack data is stored in memory locations, it requires a lot of loads and stores to move all of the data for each push/pop.

Instead, we move a pointer to the top of stack, and leave the data in place. In the figure, we use R6 to hold the address of the top of stack. By convention, we grow the stack downward, to lower addresses.



[Access the text alternative for slide images.](#)

LC-3 Implementation

1

If R6 holds the top-of-stack (TOS) pointer, the following code is used to push the value in R0 onto the stack.

```
PUSH    ADD R6,R6,#-1 ; move ptr to make room  
        STR R0,R6,#0  
        RET
```

The following code will pop the value at the TOS to R0. The stack pointer (R6) is incremented to deallocate that memory location.

```
POP     LDR R0,R6,#0  
        ADD R6,R6,#1 ; move ptr to remove item  
        RET
```

By adding RET to the end, we've made subroutines that can abstract the stack operations for users. It may be silly to have such a small subroutine, but we're about to make them more sophisticated...

Stack Underflow

What happens if we pop more times than we have pushed?

In the previous code, R6 keeps getting larger and we load garbage data that was never pushed on the stack.

You could say that the user of the stack shouldn't do that, and you would be right, but we can also make the subroutine a little friendlier by detecting the underflow condition.

In the version of POP on the next slide, we use R0 to get the value, and we use R5 to indicate whether underflow occurs. If R5=0, there's no overflow and the popped data is valid. If R5=1, then underflow was detected, and the value in R0 is unchanged from before.

NOTE: We must predetermine the memory location of the stack, and some code must initialize the stack pointer (R6) to point at the next highest memory location.

POP subroutine with Underflow Detection

```
POP      ST    R1,POP_SR1      ; save registers  
        AND   R5,R5,#0       ; assume no overflow  
        LD    R1,EMPTY       ; check whether R6 is at bottom  
        ADD   R1,R6,R1  
        BRz  POP_Fail  
  
        LDR   R0,R6,#0       ; if no underflow, get data  
        ADD   R6,R6,#1       ; increment stack pointer  
        LD    R1,POP_SR1     ; restore registers  
        RET  
  
POP_Fail ADD   R5,R5,#1       ; set R5 to indicate underflow  
        LD    R1,POP_SR1     ; restore registers  
        RET  
  
EMPTY    .FILL xC000        ; negative of x4000  
POP_SR1  .BLKW 1
```

Stack Overflow

If we keep pushing items, we may overrun the storage that is allocated for the stack -- this is called "overflow."

Similar to the underflow case, the next slide shows a subroutine that checks for overflow and uses R5 to indicate when failure has occurred.

In this case, we assume that the stack is allocated five memory locations, from x3FFB to x3FFF. R6 should never be allowed to go lower than x3FFB.

PUSH subroutine with Overflow Detection

```
PUSH      ST    R1,POP_SR1      ; save registers  
          AND   R5,R5,#0        ; assume no overflow  
          LD    R1,FULL        ; check whether R6 is at top  
          ADD   R1,R6,R1  
          BRz  PUSH_Fail  
  
          ADD   R6,R6,#-1       ; if no overflow, push  
          STR   R0,R6,#0  
          LD    R1,PUSH_SR1    ; restore registers  
          RET  
  
PUSH_Fail ADD   R5,R5,#1        ; set R5 to indicate underflow  
          LD    R1,PUSH_SR1    ; restore registers  
          RET  
  
FULL     .FILL xC005        ; negative of x3FFB  
POP_SR1  .BLKW 1
```

Combined PUSH/POP Implementation

Figure 8.11 shows an implementation of both PUSH and POP where code and storage locations are shared for efficiency.

```
POP          AND    R5,R5,#0      ; R5 <- success
             ST     R1,Save1    ; Save registers that
             ST     R2,Save2    ; are needed by POP
             LD     R1,EMPTY    ; EMPTY contains -x4000
             ADD   R2,R6,R1    ; Compare stack pointer to x4000
             BRz  fail_exit   ; Branch if stack is empty
;
             LDR   R0,R6,#0      ; The actual "pop"
             ADD   R6,R6,#1      ; Adjust stack pointer
             BRnzp success_exit

;
PUSH         AND    R5,R5,#0      ; Save registers that
             ST     R1,Save1    ; are needed by PUSH
             ST     R2,Save2    ; FULL contains -x3FFB
             LD     R1,FULL     ; Compare stack pointer to x3FFB
             ADD   R2,R6,R1    ; Branch if stack is full
             BRz  fail_exit

;
             ADD   R6,R6,#-1    ; Adjust stack pointer
             STR   R0,R6,#0      ; The actual "push"
success_exit LD     R2,Save2    ; Restore original
             LD     R1,Save1    ; register values
             RET

;
fail_exit    LD     R2,Save2    ; Restore original
             LD     R1,Save1    ; register values
             ADD   R5,R5,#1      ; R5 <- failure
             RET

;
EMPTY        .FILL  xc000      ; EMPTY contains -x4000
FULL         .FILL  xc005      ; FULL contains -x3FFB
Save1        .FILL  x0000
Save2        .FILL  x0000
```

Recursion: Can a Subroutine Call Itself?

Recursion is a technique in which a subroutine (function) can call itself.

Advantage: Can be an elegant and efficient expression of a problem.

Disadvantage: When used improperly, can introduce significant run-time overhead in execution time and memory usage.

Why discuss now? A **stack** is a natural data structure to use for recursion.

Will show two bad examples (but useful for explaining the concepts) and one good example of using recursion to solve problems.

Bad Example #1: Factorial

Factorial is a mathematical function that computes the product of the first n positive integers. The symbol for "n factorial" is n!

$$n! = n \times (n - 1) \times (n - 2) \times \boxed{?} \times 2 \times 1$$

A simpler (recursive) way to define it:

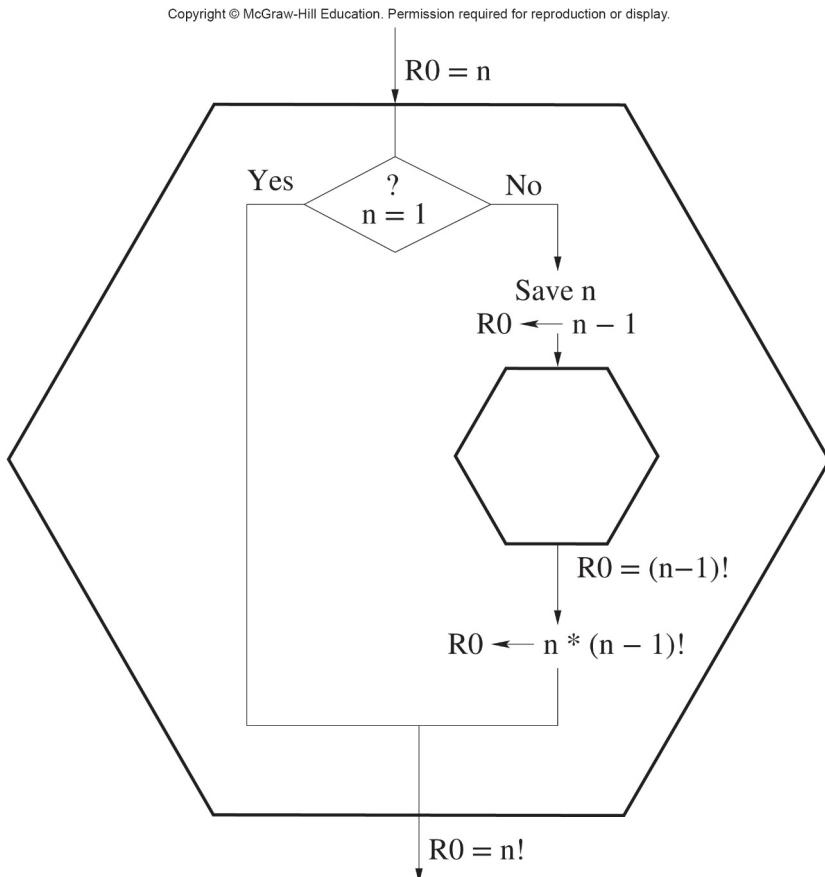
$$n! = n \times (n - 1)!$$

We also need to note that $1! = 1$. That's what allows us to multiply only the numbers > 0 .

Now factorial is defined in terms of itself.

A Factorial Subroutine

If we have a factorial subroutine FACT that takes a input n, then we could call FACT to compute the factorial of $(n - 1)$ and multiply by n. If $n = 1$, we just return 1 without calling FACT.



FACT	ST	R1 , Save1
	ADD	R1 , R0 , #-1
	BRz	DONE
	ADD	R1 , R0 , #0
	ADD	R0 , R1 , #-1
B	JSR	FACT
	MUL	R0 , R0 , R1
DONE	LD	R1 , Save1
	RET	
Save1	.BLKW	1

For the purposes of this example, we are pretending that the LC-3 has a multiply (MUL) opcode.

[Access the text alternative for slide images.](#)

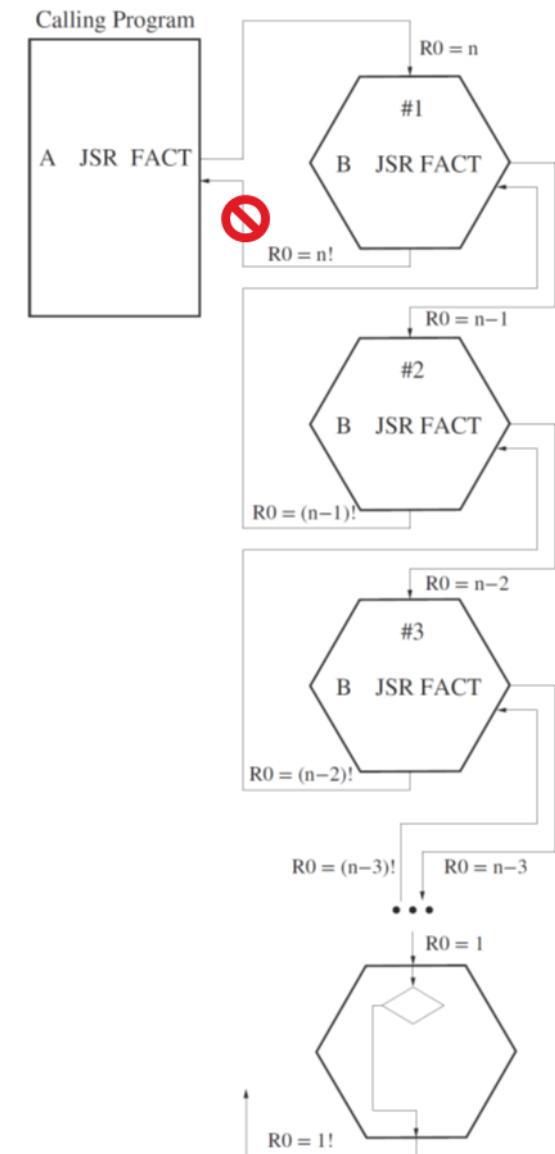
Problem: Overwritten Return Address (R7)

Unfortunately, the code does not work.

When original caller calls FACT, it sets the return address (R7) to A+1.

When FACT calls FACT, it sets the return address to B+1, overwriting the old value of R7. Therefore, when we are ready to return the original caller, we have lost the return address information and there's no way to get back.

Solution: Push R7 onto a **stack** before calling, and pop R7 from the stack before returning.



[Access the text alternative for slide images.](#)

Still a Problem: Overwriting Local Data

Recall that we save R0 (n) into R1, so that we can multiply by n when FACT($n - 1$) is returned.

Because the subroutine modifies R1, we are careful to save/restore it using this memory location.

However, the next time FACT is called, it will use the same memory location, overwriting the old data.

Solution: Push R0 onto a stack, and pop the stack before the multiply.

FACT	ST	R1 , Save1
	ADD	R1 , R0 , #-1
	BRz	DONE
B	ADD	R1 , R0 , #0
DONE	ADD	R0 , R1 , #-1
	JSR	FACT
	MUL	R0 , R0 , R1
Save1	LD	R1 , Save1
	RET	.BLKW 1

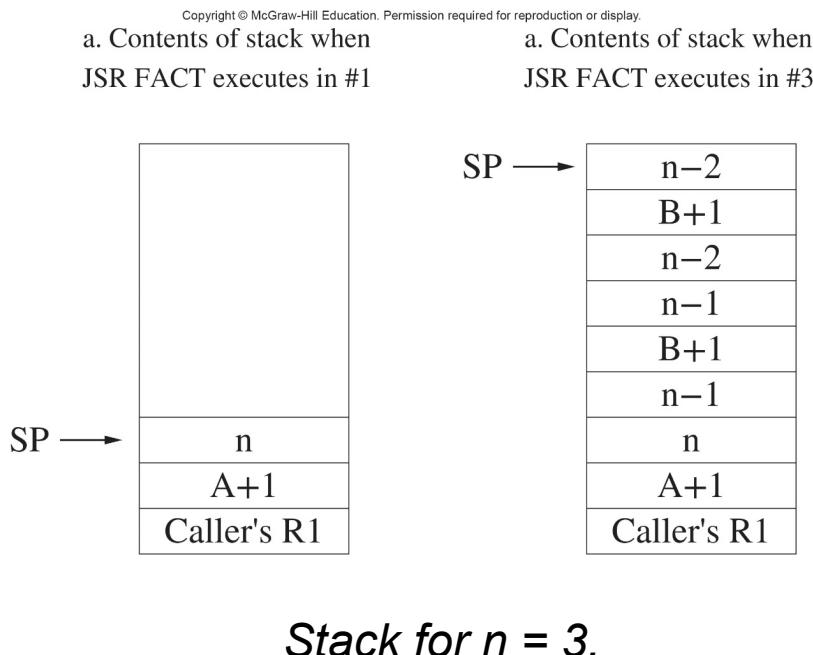
Correct Recursive Subroutine: Factorial

Figure 8.14 shows a correct implementation of the FACT subroutine, incorporating the stack solution of the previous problems.

```
Copyright © McGraw-Hill Education. Permission required for reproduction or display.  
FACT      ADD  R6,R6,#-1  
          STR  R1,R6,#0 ; Push Caller's R1 on the stack, so we can use R1.  
;  
          ADD  R1,R0,#-1 ; If n=1, we are done since 1! = 1  
          BRz NO_RECURSE  
;  
          ADD  R6,R6,#-1  
          STR  R7,R6,#0 ; Push return linkage onto stack  
          ADD  R6,R6,#-1  
          STR  R0,R6,#0 ; Push n on the stack  
;  
          ADD  R0,R0,#-1 ; Form n-1, argument of JSR  
B         JSR  FACT  
          LDR  R1,R6,#0 ; Pop n from the stack  
          ADD  R6,R6,#1  
          MUL  R0,R0,R1 ; form n*(n-1)!  
;  
          LDR  R7,R6,#0 ; Pop return linkage into R7  
          ADD  R6,R6,#1  
NO_RECURSE LDR  R1,R6,#0 ; Pop caller's R1 back into R1  
          ADD  R6,R6,#1  
          RET
```

Correct, but not Efficient

So why do we call this a bad example? Because the space and time required to push and pop data from the stack is large. The space required to the stack increases linearly with the input variable (n).



If we implement the algorithm with a loop (iteration), the number of instructions is smaller and there is no need for a stack.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.

FACT	ST	R1,SAVE_R1
	ADD	R1,R0,#0
	ADD	R0,R0, #-1
	BRz	DONE
AGAIN	MUL	R1,R1,R0
	ADD	R0,R0, #-1 ; R0 gets next integer for MUL
	BRnp	AGAIN
DONE	ADD	R0,R1,#0 ; Move $n!$ to R0
	LD	R1,SAVE_R1
	RET	
SAVE_R1	.BLKW	1

[Access the text alternative for slide images.](#)

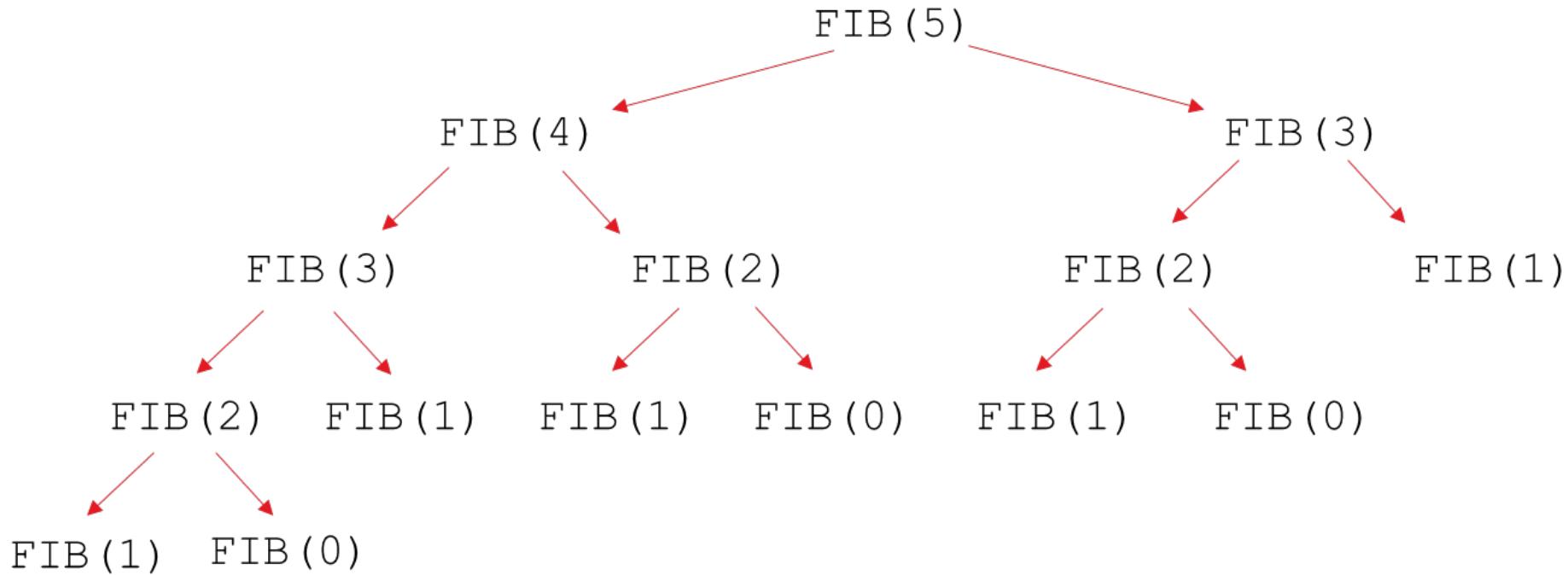
Bad Example #2: Fibonacci

Another popular example of recursion is calculating the n-th number in the Fibonacci series. Each number is equal to the sum of the previous two numbers in the series. Mathematically:

$$f(n) = \begin{cases} 0, & n=0 \\ 1, & n=1 \\ f(n-2) + f(n-1), & n \geq 2 \end{cases}$$

Based on the previous example, you can probably guess that $\text{FIB}(n)$ would call $\text{FIB}(n - 2)$ and $\text{FIB}(n - 1)$, and that we can use a stack to keep track of return addresses, local temporary values, etc.

Problem: Repeated Computation



$\text{FIB}(3)$ is called 2 times
 $\text{FIB}(2)$ is called 3 times
 $\text{FIB}(1)$ is called 5 times
 $\text{FIB}(0)$ is called 3 times

Iterative approach (next slide)
does not repeat and is
much faster!

[Access the text alternative for slide images.](#)

Iterative Fibonacci

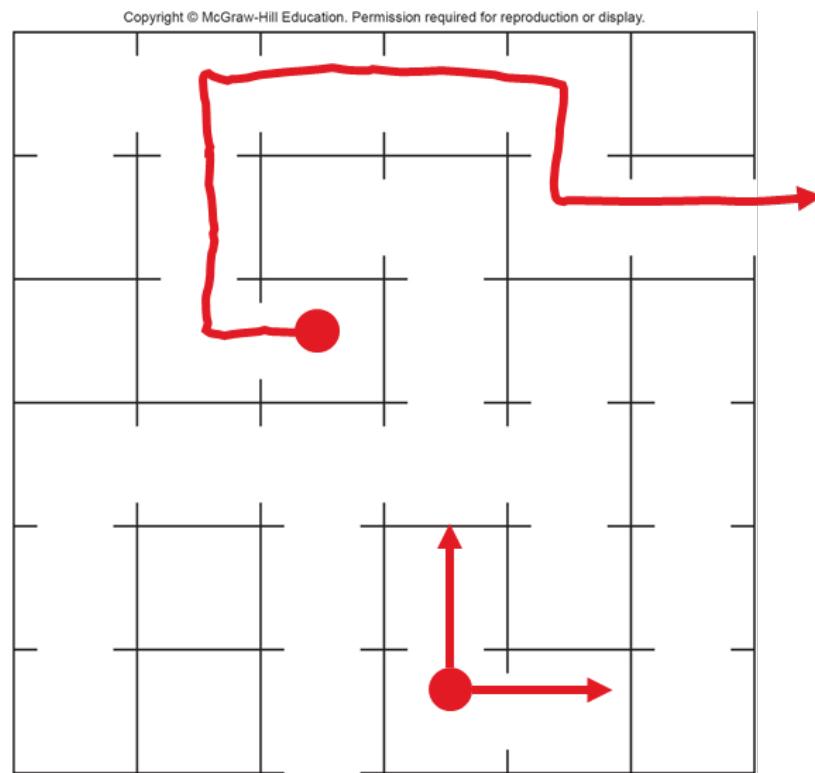
Copyright © McGraw-Hill Education. Permission required for reproduction or display.

```
FIB    ST     R1,SaveR1
      ST     R2,SaveR2
      ST     R3,SaveR3
      ST     R4,SaveR4
      ST     R5,SaveR5
;
NOT   R0,R0
ADD   R0,R0,#1 ; R0 contains -n
AND   R1,R1,#0 ; Suppose n=0
ADD   R5,R1,R0 ; R5 = 0 -n
BRz  DONE      ; if n=0, done almost
AND   R3,R2,#0 ; if n>0, set up R3 = FIB(0) = 0
ADD   R1,R3,#1 ; Suppose n=1
ADD   R5,R1,R0 ; R5 = 1-n
BRz  DONE      ; if n=1, done almost
ADD   R4,R1,#0 ; if n>1, set up R4 = FIB(1) = 1
;
AGAIN ADD   R1,R1,#1 ; We begin the iteration of FIB(i)
      ADD   R2,R3,#0 : R2= FIB(i-2)
      ADD   R3,R4,#0 : R3= FIB(i-1)
      ADD   R4,R2,R3 ; R4 = FIB(i)
      ADD   R5,R1,R0 ; is R1=n ?
      BRn  AGAIN
;
      ADD   R0,R4,#0 ; if n>1, R0=FIB(n)
      BRnzp RESTORE
DONE   ADD   R0,R1,#0 ; if n=0,1, FIB(n)=n
RESTORE LD    R1,SaveR1
      LD    R2,SaveR2
      LD    R3,SaveR3
      LD    R4,SaveR4
      LD    R5,SaveR5
RET
```

Good Example: Maze

Sometimes, the elegance of recursion provides a natural way to solve a complicated problem.

Example: Given a starting point in a maze, determine whether there is a path to exit the maze.



[Access the text alternative for slide images.](#)

Data Representation

In the 6×6 maze shown, there are 36 starting points.

For each point, use four bits to indicate whether there is a wall in each of the four directions. Use another bit to indicate whether this is an opening to the outside world.

out	N	E	S	W
-----	---	---	---	---

1 = door

0 = no door

Store data in "row-major" order:

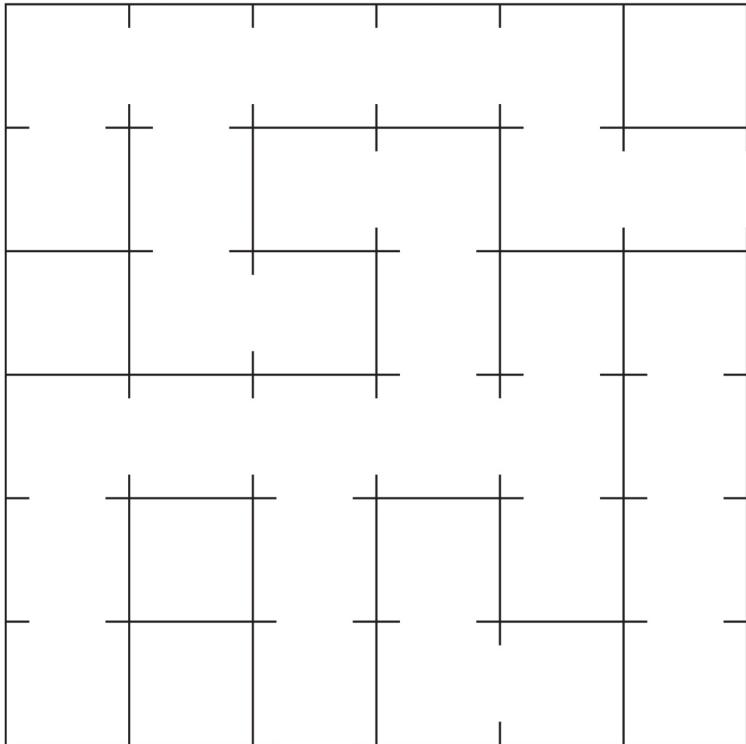
All cells in row 1 (top), left to right.

All cells in row 2, left to right.

Etc.

Maze Data

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



Copyright © McGraw-Hill Education. Permission required for reproduction or display.

```
00      .ORIG x5000
01 MAZE    .FILL x0006
02          .FILL x0007
03          .FILL x0005
04          .FILL x0005
05          .FILL x0003
06          .FILL x0000
07 ; second row: indices 6 to 11
08          .FILL x0008
09          .FILL x000A
0A          .FILL x0004
0B          .FILL x0003
0C          .FILL x000C
0D          .FILL x0015
0E ; third row: indices 12 to 17
0F          .FILL x0000
10          .FILL x000C
11          .FILL x0001
12          .FILL x000A
13          .FILL x0002
14          .FILL x0002
15 ; fourth row: indices 18 to 23
16          .FILL x0006
17          .FILL x0005
18          .FILL x0007
19          .FILL x000D
1A          .FILL x000B
1B          .FILL x000A
1C ; fifth row: indices 24 to 29
1D          .FILL x000A
1E          .FILL x0000
1F          .FILL x000A
20          .FILL x0002
21          .FILL x0008
22          .FILL x000A
23 ; sixth row: indices 30 to 35
24          .FILL x0008
25          .FILL x0000
26          .FILL x001A
27          .FILL x000C
28          .FILL x0001
29          .FILL x0008
2A          .END
```

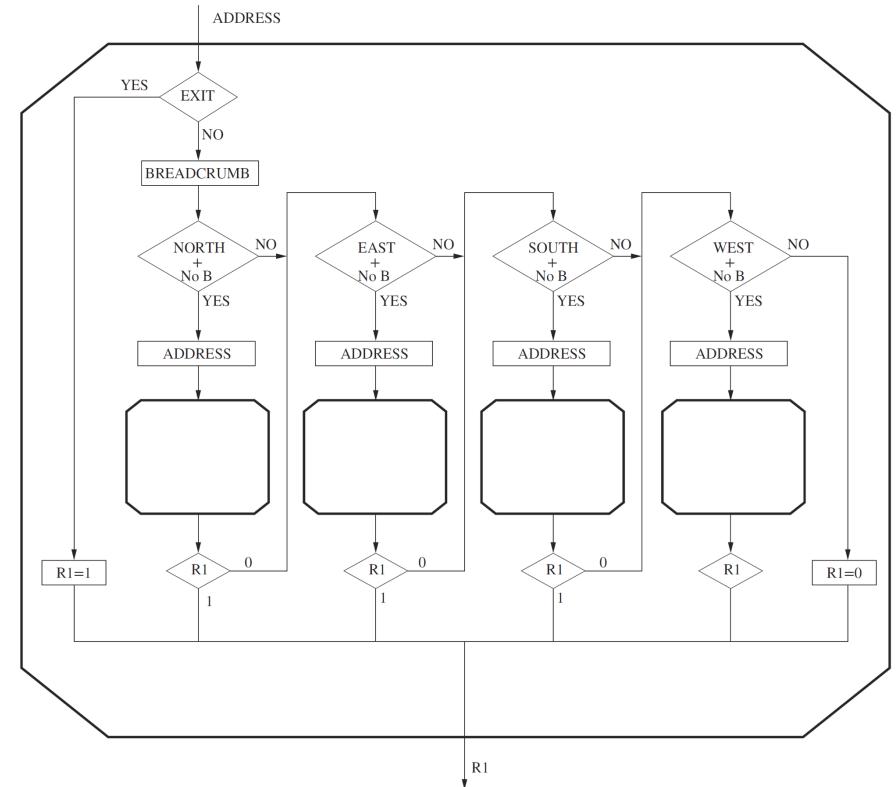
[Access the text alternative for slide images.](#)

Recursive Algorithm for Maze Traversal

A sketch of the recursive algorithm is shown in Figure 8.23.

If the current cell has an exit door, then we are done: success!

If not, we first "drop a breadcrumb" for this cell, so that we don't try to exit from here again. We can set bit 15 of this cell's data to show we've been here.



Next, if there's a door to the North, we ask whether there's an exit path from that cell -- this is the recursive call. If the answer is no, we try East, etc. If there is no path in any direction, we know that we are done.

See the implementation in Figure 8.24.

[Access the text alternative for slide images.](#)

Why is Maze a Good Example of Recursion?

The recursion greatly simplifies the description of the problem.

- Don't have to keep track of all the paths we've tried.
- Each cell "remembers" whether it has been visited, avoiding the possibility of infinite loops, etc.
- The problem is localized to a single cell. Instead of tracing out all the paths from here, we simply ask the neighbors.

This problem is simplified, because we just want a YES/NO answer. It's a little more complicated to actually determine what the exit path is, or what the shortest exit path would be, but recursion would still be useful in those cases.

An iterative version of this code would be much more complicated.

Stack and Recursion

The **stack** is a good data structure for storing state in a recursive algorithm.

- Return address
- Local state

The subroutine call/return mechanism is last-in, first-out:

Returns are done in the opposite order of calls.

The subroutine returning first is the last subroutine that was called.

This will be discussed again in Chapter 14, for C programs and other high-level languages.

Queue

A **queue** is a data structure that contains an sequence of items, accessed in **first-in, first-out (FIFO)** order.

The **first** thing you put in the queue will be the **first** thing removed.

The operations defined for a stack are:

insert an item at the back (rear) of the queue, and

remove an item from front of the queue.

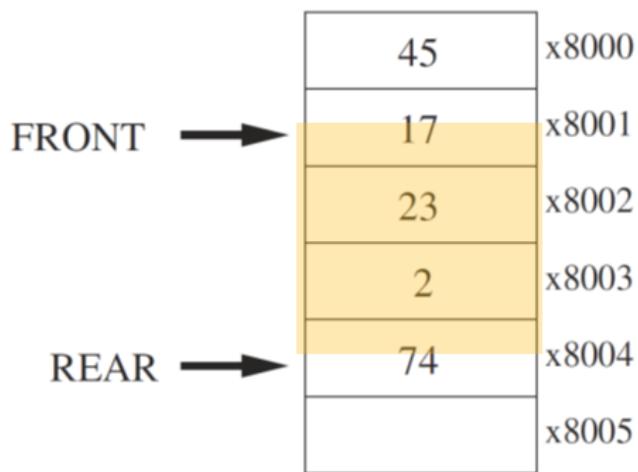
Examples:

- a (polite) line at the grocery store,
- a sequence of songs in a playlist,
- a line of cars at a gas pump

Queue Implementation

To implement a queue, we allocate a region of memory to store data items, and we use **two pointers** to indicate the front and rear. Like the stack, the pointers move only one address at a time.

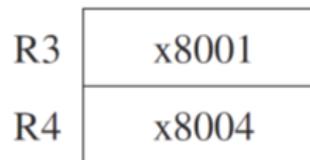
In the example below, memory x8000 to x8005 is allocated for a six queue.



23, 2, and 74 are the current items in the queue. (45 and 17 have been removed.)

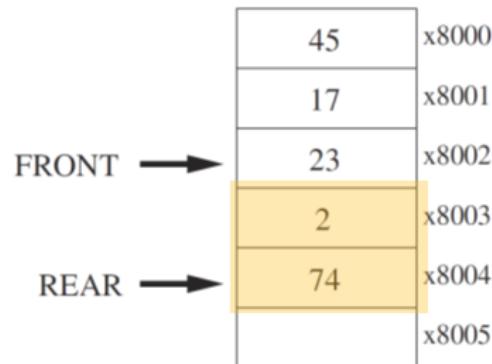
R3 is the front pointer, which is the address just before the first item.

R4 is the rear pointer, which refers to the last item inserted.



[Access the text alternative for slide images.](#)

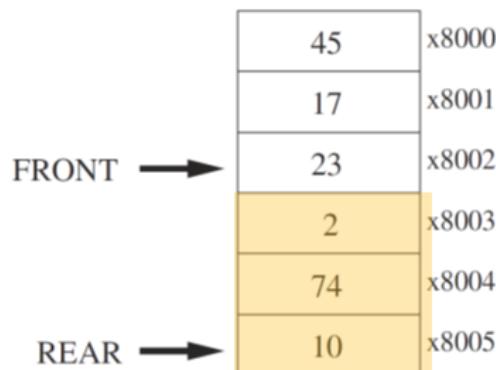
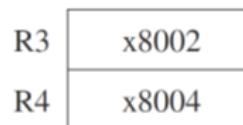
Queue: LC-3 Implementation



To remove, we move the front pointer ahead and load the data item.

ADD R3, R3, #1

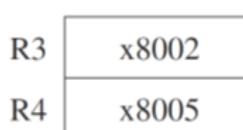
LDR R0, R3, #0



To insert, we move the rear pointer ahead and store the data item.

ADD R4, R4, #1

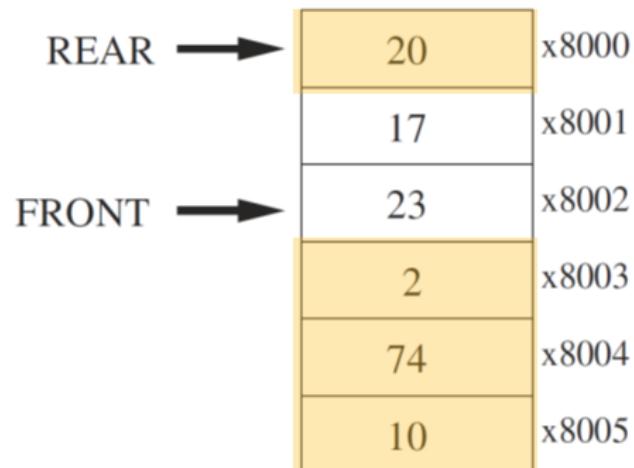
STR R0, R4, #0



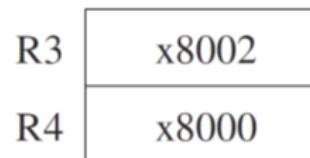
[Access the text alternative for slide images.](#)

Wrap-Around

It now appears that we cannot insert more data, because we've reached the end of our allocated storage. But there are only three elements in the queue, and three unused storage locations.



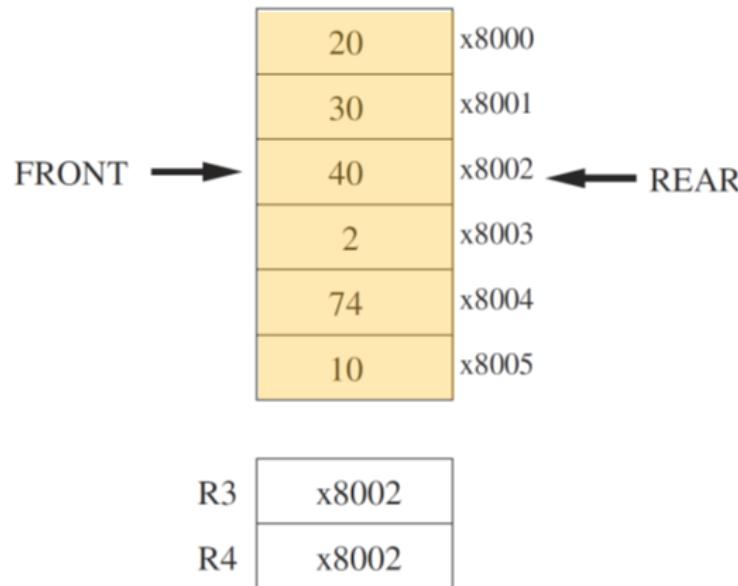
Since location x8000 is unused, we can wrap around the rear pointer and put a new element (20) in that location.



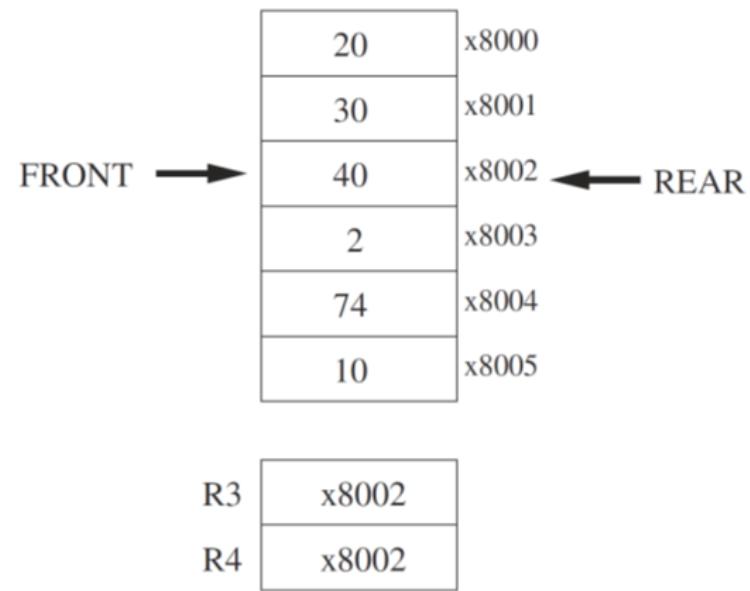
[Access the text alternative for slide images.](#)

Wrap-Around ₂

If we add two more items, all of our queue slots are full, and the front and rear pointers are equal.



Suppose we now start removing items.
Moving the front pointer forward and
wrapping around at x8005.
After six removals:

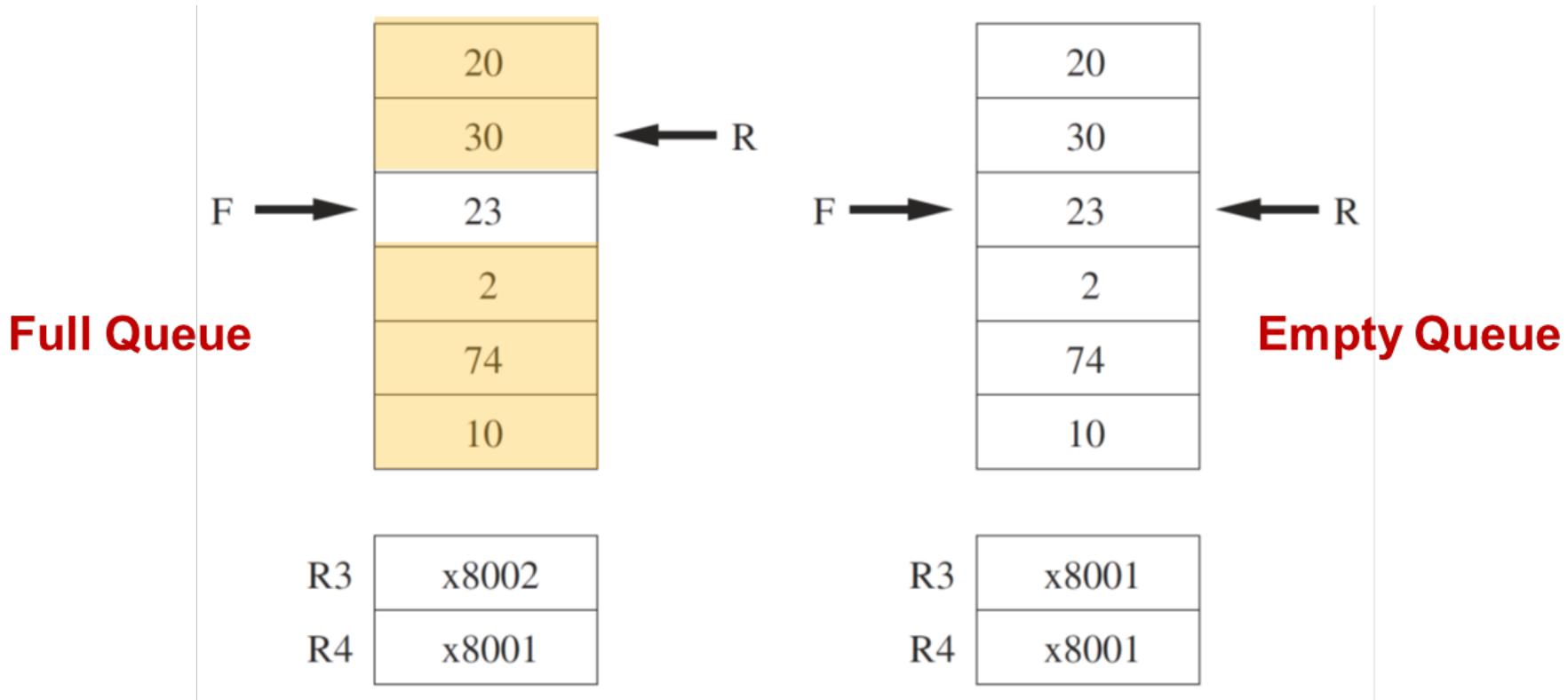


The full queue and the empty queue
look exactly the same -- not good!!

[Access the text alternative for slide images.](#)

Solution to the Wrap-Around Problem

Our solution is to only store $n - 1$ items in an queue with n memory locations.



Thus, front = rear only if the queue is empty.

[Access the text alternative for slide images.](#)

Overflow and Underflow

As with the stack, we want our subroutines to recognize overflow and underflow scenarios.

- If the queue is full, an insert operation will fail.
- If the queue is empty, a remove operation will fail.

On the next slide, we show an INSERT subroutine with overflow detection. The REMOVE subroutine with underflow detection is given in Figure 8.27.

Queue Insertion with Overflow Detection

```
INSERT      ST    R1,SaveR1      ; save registers
            AND   R5,R5,#0       ; initialize for no overflow
            LD    R1,NEG_LAST
            ADD   R1,R1,R4       ; R1 = rear - x8005
            BRnp SKIP1
            LD    R4,FIRST        ; wraparound
            BR    SKIP2          ; BR or BRnzp is unconditional branch
SKIP1       ADD   R4,R4,#1       ; if no wrap, increment rear ptr
SKIP2       NOT   R1,R4        ; compare front to rear
            ADD   R1,R1,#1
            ADD   R1,R1,R3
            BRz  FULL
            STR  R0,R4,#0       ; not full, store data
            BR   DONE
FULL        LD    R1,NEG_FIRST ; if full, undo rear ptr move
            ADD   R1,R1,R4       ; R1 = rear - x8000
            BRnp SKIP3
            LD    R4,LAST         ; undo wraparound
            BR   SKIP4
SKIP3       ADD   R4,R4,#-1      ; if no wrap, undo increment
SKIP4       ADD   R5,R5,#1       ; return error value
DONE        LD    R1,SaveR1
            RET
SaveR1      .BLKW 1           ; space to save reg
FIRST       .FILL  x8000        ; starting address of queue storage
NEG_FIRST   .FILL  x8000        ; negative of FIRST
LAST        .FILL  x8005        ; ending address of queue storage
NEG_LAST    .FILL  x7FFB        ; negative of LAST
```

String

A **string** is a sequence of characters.

- With the ASCII code, characters include letters, digits, punctuation, and other characters. (See Chapter 2.)
- A string can be of any length. The **null character** (ASCII x00) denotes the end of the string.

In most scenarios, we are given a **pointer** to the first character in the string. The code that processes the string proceeds until the null character is found.

LC-3 Implementation 2

Even though an ASCII character only requires 7 bits of information, we will usually store one character per (16-bit) word to simplify the code needed to process a string.

In the example below, the string **"Time flies."** is stored at location x4000.

This string has 11 characters, stored in 12 memory locations.

x4000	x0054	'T'
x4001	x0069	'i'
x4002	x006D	'm'
x4003	x0065	'e'
x4004	x0020	' '
x4005	x0066	'f'
x4006	x006C	'l'
x4007	x0069	'i'
x4008	x0065	'e'
x4009	x0073	's'
x400A	x002E	'..'
x400B	x0000	null

The .STRINGZ Directive

LC-3 Assembly Language has a directive that makes it easy to specify string data for a program.

```
.STRINGZ "Time flies."
```

Like .FILL, the directive will initialize data at the specified location. In this case, however, the number of memory locations is determined by the length of the string.

As shown on the previous slide, the directive above will allocate 12 memory locations and initialize each with a character. The null character is always in the last memory location.

The Z at the end is meant to remind you of that trailing zero.

.STRINGZ Example

```
; classic "Hello, World!" program in LC-3 assembly language

        .ORIG x3000
        LEA    R1, HELLO      ; get address of string
LOOP      LDR    R0,R1,#0       ; load character using pointer (R1)
        BRz   DONE          ; if null character, end of string
        TRAP  x21          ; print the character
        ADD    R1,R1,#1       ; move ptr to next char in string
        BR    LOOP
DONE      TRAP  x25          ; HALT
;
HELLO    .STRINGZ "Hello, World!\n"
; '\n' is a special character code that
; represents the linefeed character
; It is one character, not two.
.END
```

Character Operations

You will need several common operations in your bag of tricks to process ASCII characters. Here are a few:

Compare two characters in "alphabetical order"

Same as comparing numerically -- subtract one from the other

Check if a character is a decimal digit

Digit characters go from x30 ('0') to x39 ('9')

If you know a character is a letter, check if it's upper-case

Look at bit 5 -- 1 if lower-case ('a' = x61), 0 if upper-case ('A' = x41)
(Look at? AND with x20. If zero, bit is zero. If nonzero, bit is 1.)

Convert letter to uppercase

AND with xFFDF. Clears bit 5.

Convert from decimal digit to corresponding integer (for example, from '4' to 4)

Subtract x30

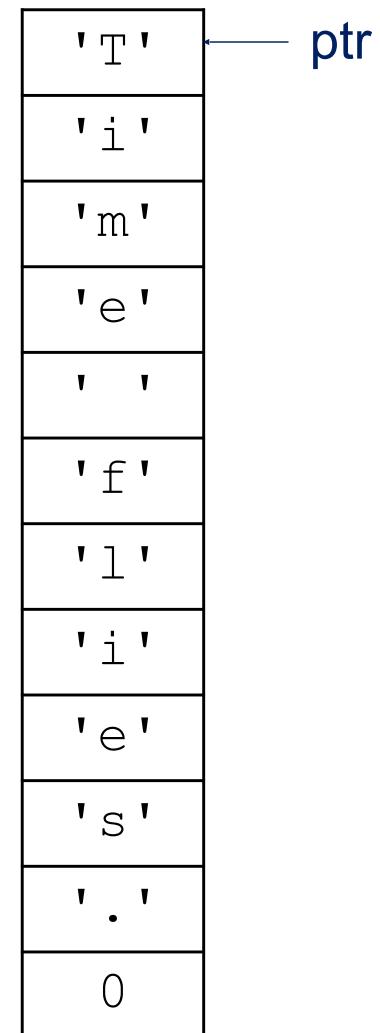
String Processing

A typical pattern when processing strings

1. Point to the start of the string
2. Load a character, using the pointer
3. Check for end of string -- exit loop
4. Do something to/with the character
5. Increment pointer
6. Loop back to 2

Examples:

- How many characters in the string?
- How many uppercase letters?
- How many digits?



String Subroutine

Problem:

Write a subroutine to **compare two strings**.

Inputs:

R0 = address of string 1

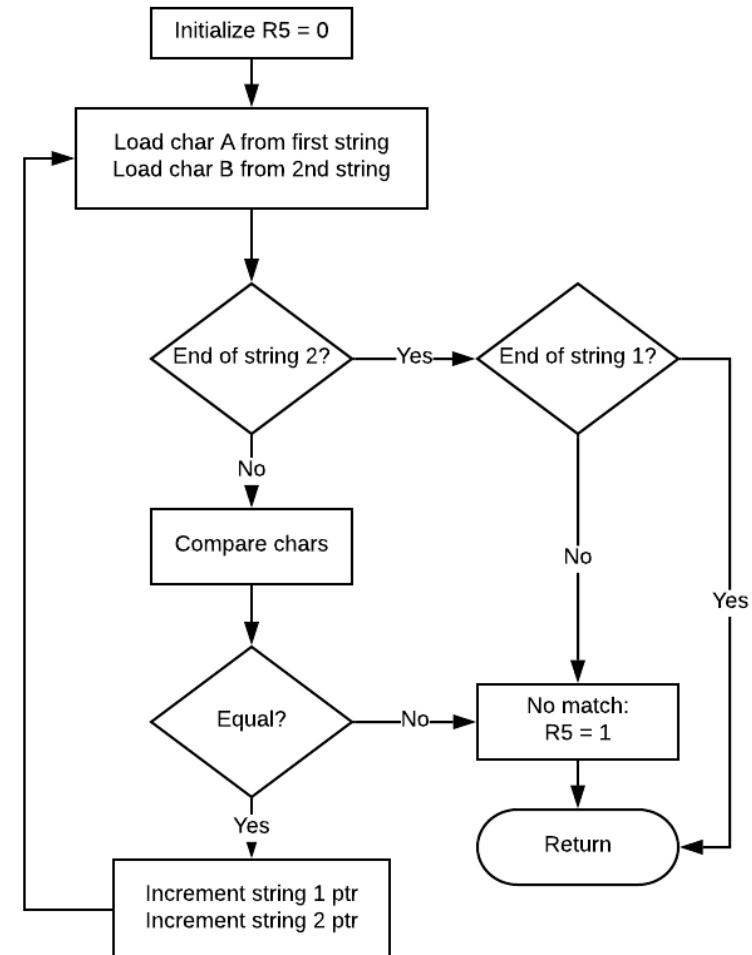
R1 = address of string 2

Output:

R5 = 0 if strings equal, 1 otherwise

Algorithm:

Move two pointers through strings until (a) characters are not equal or (b) reach the end of one of the strings. If both strings end at the same time, then all characters match.



[Access the text alternative for slide images.](#)

String Compare Subroutine

```
STRCMP      ST    R0,SaveR0      ; save registers
            ST    R1,SaveR1
            ST    R2,SaveR2
            ST    R3,SaveR3
            AND   R5,R5,#0      ; assume a match

NEXTCHAR    LDR   R2,R0,#0      ; char from string 1
            LDR   R3,R1,#0      ; char from string 2
            BRnp COMPARE       ; not end of string 2 -- compare
            ADD   R2,R2,#0
            BRz  DONE          ; end of both strings -- match!

COMPARE     NOT   R2,R2
            ADD   R2,R2,#1
            ADD   R2,R2,R3      ; if zero, chars are equal
            BRnp FAIL          ; if zero, chars are equal
            ADD   R0,R0,#1      ; increment pointers to check next chars
            ADD   R1,R1,#1
            BR   NEXTCHAR

FAIL        ADD   R5,R5,#1      ; return error value
DONE         LD    R0,SaveR0
            LD    R1,SaveR1
            LD    R2,SaveR2
            LD    R3,SaveR3
            RET

SaveR0      .BLKW 1      ; space to save reg
SaveR1      .BLKW 1      ; space to save reg
SaveR2      .BLKW 1      ; space to save reg
SaveR3      .BLKW 1      ; space to save reg
```



Because learning changes everything.[®]

www.mheducation.com

From Bits and Gates to C and Beyond

I/O

Chapter 9

Details of Input and Output

Operating System Basics

 Privilege, Priority, and Memory Space

I/O Basics

 Devices, Device Registers, Memory-Mapped I/O, Interrupts

TRAP Instruction

 Invoking a system call

Interrupt-Driven I/O

 Hardware-triggered service routines

Operating System

What is an Operating System (OS)?

Software that manages resources for the computing system.

Resources: memory, I/O devices, use of CPU, ...

Two primary goals:

- Optimize access to resources

- Make sure no software does harmful things to programs or data that it should not access

Key concepts: **privilege** and **priority**

Examples: Windows, MacOS, Linux

Privilege and Priority

Privilege = What is this software allowed to do / access?

- Special instructions (for example, HALT)

- Special memory address (for example, operating system code)

- Only privileged instructions should be able to access these things.

Supervisor mode = privileged, **User mode** = non-privileged

Priority = Which software is more urgent to execute?

- User programs operate at lowest priority level (0).

- Some events may need immediate attention, require system software:

 - Input from the keyboard: priority 4

 - Power interruption: priority 6

 - If both happen, power interruption code runs because it has a higher priority.

LC-3 Processor Status Register

Processor (hardware) needs to know the privilege level and the priority of the instruction that is running.

LC-3 has two privilege levels: supervisor, user.

LC-3 has eight priority levels: user = 0, highest priority = 7.

Information is kept in the Processor Status Register (PSR), which also contains the condition code bits.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	PSR
Pr						PL n						N	Z	P	cond codes	

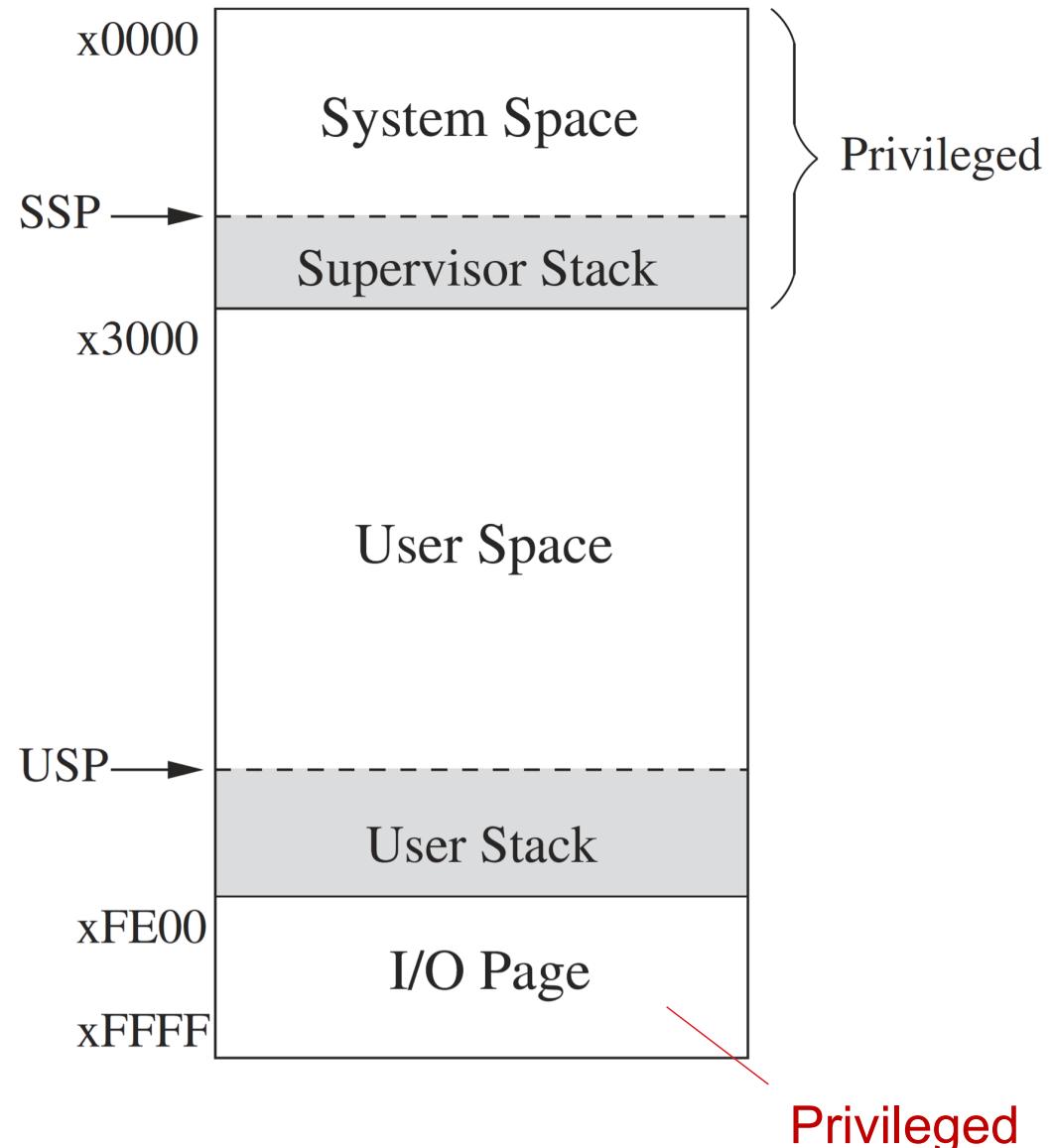
[Access the text alternative for slide images.](#)

LC-3 Privileged Memory

A portion of the memory address space is reserved for the operating system. This can only be accessed by instructions while running in supervisor mode.

While in supervisor mode, R6 refers to the top of the supervisor stack. In user mode, R6 refers to the top of the user stack. The values of the SSP and USP are saved in hardware and moved into/out of R6 when switching modes.

Addresses xFE00 to xFFFF are reserved for memory-mapped I/O, described later. This range of addresses is also privileged.



Privileged

[Access the text alternative for slide images.](#)

OS and I/O

The previous slides just give you an idea of the mechanisms provided by hardware to distinguish between OS (privileged) and user (non-privileged) software.

How does this relate to I/O?

We only want OS code to deal directly with I/O devices.

- Remove complexity and create abstraction for user code.
- Avoid dangerous behavior from programmers who may be incompetent or malicious.

We're now ready to discuss how to write software that interacts with input and output devices.

I/O: Connecting to the Outside World

Types of I/O devices are generally characterized by:

behavior: input, output, storage

input: keyboard, motion detector, network interface

output: monitor, printer, network interface

storage: disk, CD-ROM

data rate: how fast can data be transferred?

keyboard: 100 bytes/sec

disk: 30 MB/s

network: 1 Mb/s to 1 Gb/s

I/O Controller: Registers to interface with CPU

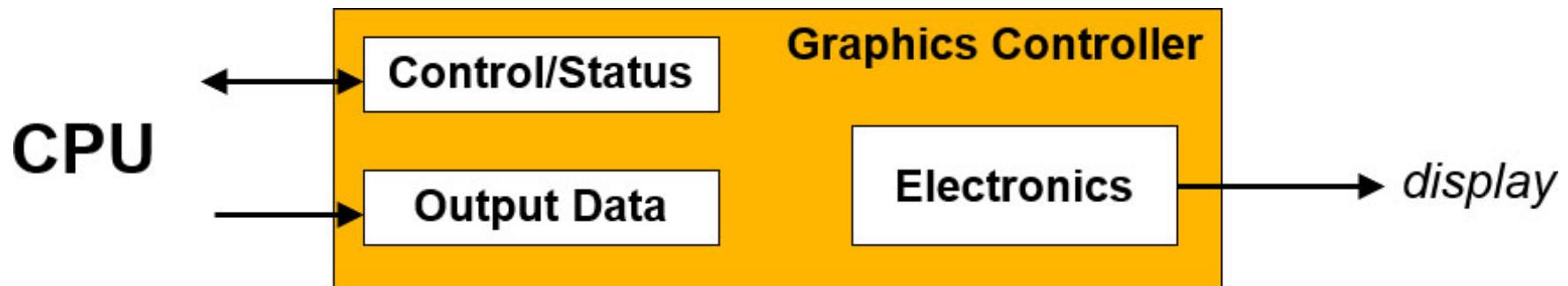
Control/Status Registers

CPU tells device what to do -- write to control register

CPU checks whether task is done -- read status register

Data Registers

CPU transfers data to/from device



Device electronics

performs actual operation

pixels to screen, bits to/from disk, characters from keyboard

[Access the text alternative for slide images.](#)

I/O Programming Interface

How are device registers identified?

memory-mapped versus **I/O instructions**

How is timing of transfer managed?

synchronous versus **asynchronous**

Who initiates / controls the transfer?

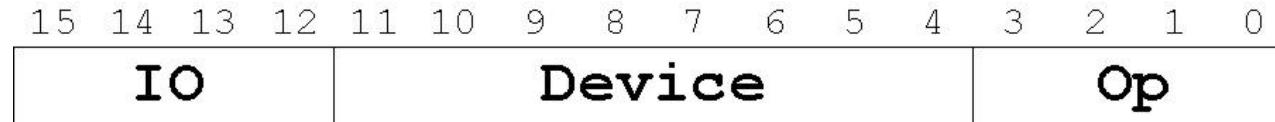
CPU (**polling**) versus device (**interrupts**)

Memory-Mapped versus I/O Instructions

Instructions

designate opcode(s) for I/O

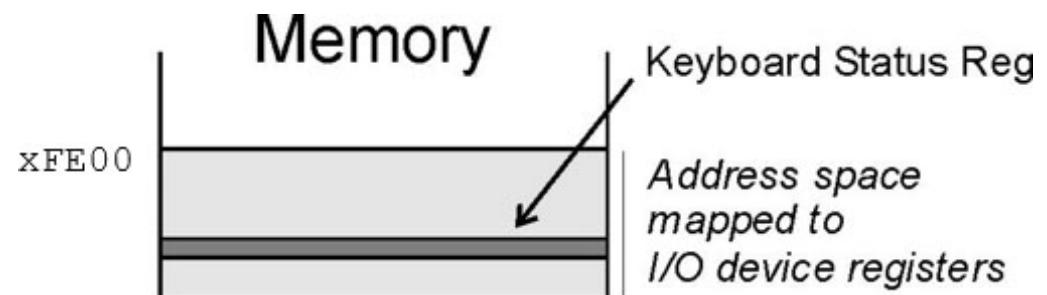
register and operation encoded in instruction



Memory-mapped

assign a memory address to each device register

use data movement instructions (LD/ST) for control and data transfer



Note: There is no memory here at all!
Hardware must recognize the address and direct the read/write to the proper I/O device register.

[Access the text alternative for slide images.](#)

Data Transfer Timing

Synchronous

Device supplies (or accepts) data at fixed, predictable times
CPU reads/writes every X cycles

Asynchronous

Data rate and availability is not predictable
CPU must *synchronize* with device for each transfer, to make sure data is not missed.

Data Transfer Control

Polling

CPU decides when it wants to perform I/O operation.

Reads device status register (over and over) until (a) device has new input data or (b) device is ready to accept new output data.

"Are we there yet? Are we there yet? Are we there yet? ..."

Interrupts

Device recognizes when an interesting change in status occurs -- for example, new input data is available, or device ready to accept new data.

Device sends a signal to the CPU.

CPU interrupts the current program to handle the I/O event, then resumes when the event handling is complete.

"Wake me up when we get there."

I/O in the LC-3

The LC-3 provides the following mechanisms for I/O.

Memory-mapped Device Registers

Memory addresses xFE00 to xFFFF are reserved for device registers.
Must be implemented by the computer system hardware.
Access to these addresses is privileged.

Asynchronous Transfers

Though it's possible to interface with synchronous devices, we will only discuss asynchronous in this class.

Polling and Interrupts

Polling is performed by reading / writing device registers.

Interrupt mechanism will be described later in the chapter.

Input from the Keyboard

The Keyboard device in the LC-3 uses two registers.

Keyboard Data Register (KBDR = xFE02)

When a key is typed, its ASCII code is placed in KBDR[7:0].
KBDR[15:8] is always zero.

Keyboard Status Register (KBSR = xFE00)

Bit 15 is the "ready" bit.

KBSR[15] = 1 means new data has been written to the KBDR.
KBSR[15] = 0 means no new data has been written.

When a key is typed:

Device (hardware) puts ASCII code
into KBDR, and sets KBSR[15] to 1.

When KBDR is read by CPU:

Device sets KBSR[15] to 0.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



Basic Input Routine (Polling)

```
; Read from keyboard, put data in R0.  
START LDI R1, A ; read KBSR  
        BRzp START ; loop until ready bit  
set    LDI R0, B ; read KBDR  
        BRnzp NEXT  
A     .FILL xFE00 ; KBSR address  
B     .FILL xFE02 ; KBDR address
```

First two lines are polling loop. Keep reading status register until the ready bit is set.

Why did we choose bit 15 as the ready bit? Value looks negative when read.

Note the use of LDI. Device register addresses are far away, so we must use a register or memory location to hold the address.

Output to the Monitor

The Monitor device in the LC-3 uses two registers.

Display Data Register (DDR = xFE06)

An ASCII character written to this register will be displayed on the monitor.
Bits written to DDR[15:8] are ignored.

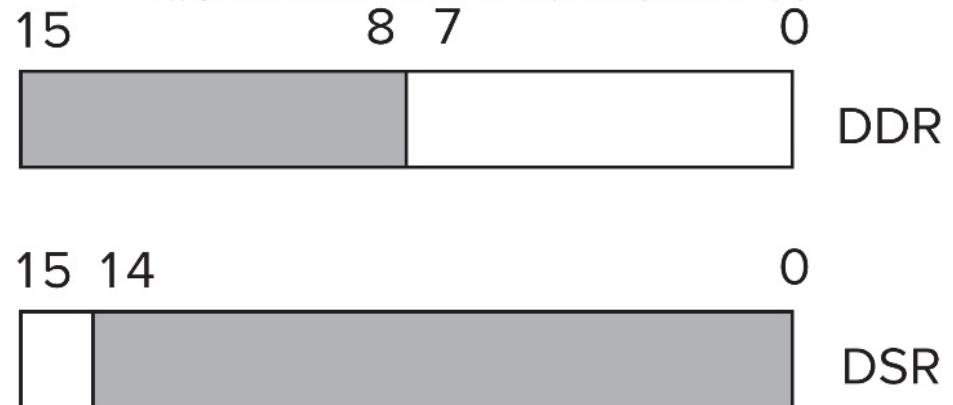
Display Status Register (DSR = xFE04)

Bit 15 is the "ready" bit.

DSR[15] = 1 means device is ready to accept the next character.

DSR[15] = 0 means the previous character has not been displayed.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



Basic Output Routine (Polling)

```
; Display character in R0 to monitor.  
START LDI R1, A      ; read DSR  
      BRzp START    ; loop until ready bit set  
      STI R0, B      ; write DDR  
      BRnzp NEXT  
A       .FILL xFE04    ; DSR address  
B       .FILL xFE06    ; DDR address
```

Keyboard Input with Echo

```
; Read from keyboard, put data in R0.  
; Echo character to monitor.  
START LDI R1, A      ; wait for keyboard input  
      BRzp START  
      LDI R0, B      ; read KBDR  
ECHO  LDI R1, C      ; wait for monitor ready  
      BRzp ECHO  
      STI R0, D      ; write DDR  
      BRnzp NEXT  
A     .FILL xFE00    ; KBSR address  
B     .FILL xFE02    ; KBDR address  
C     .FILL xFE04    ; DSR address  
D     .FILL xFE06    ; DDR address
```

Input with Prompt

How does user know it's time to enter a character from the keyboard?

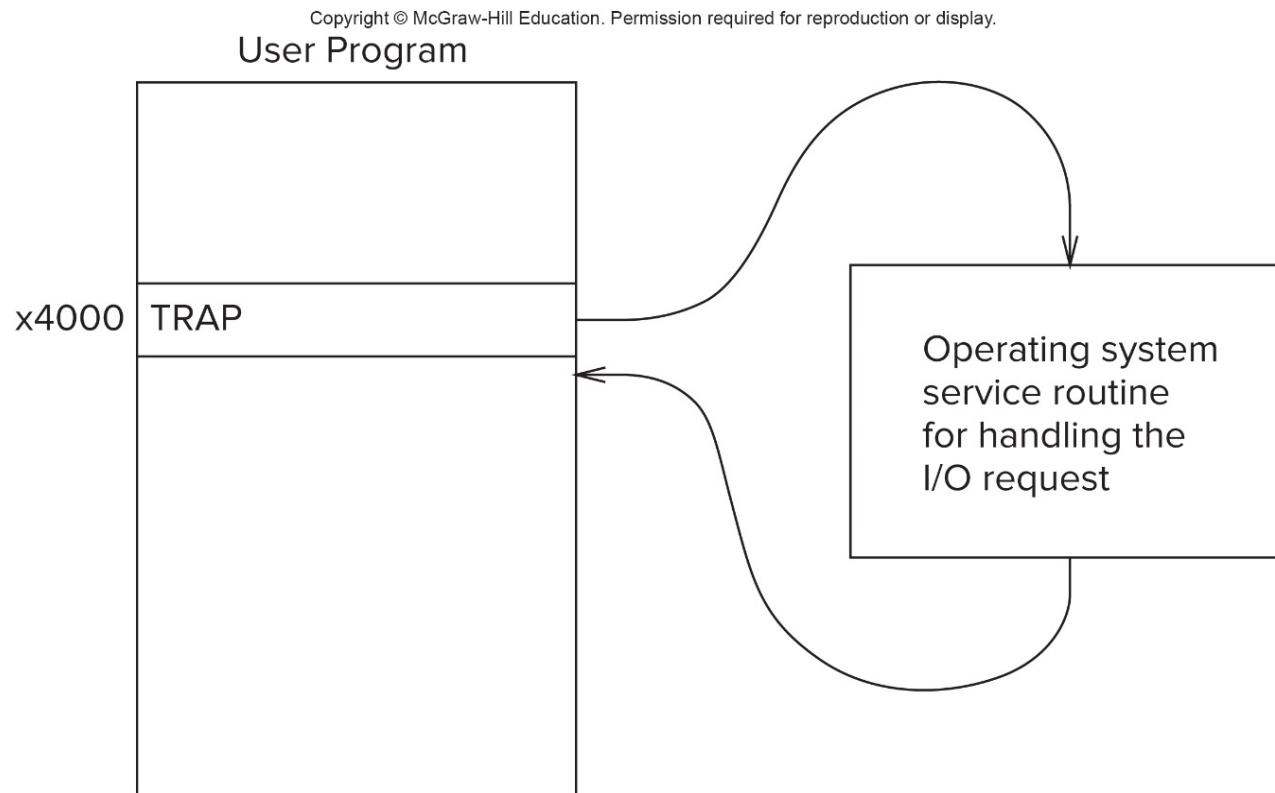
This routine prints a prompt string and then waits for keyboard input.

```
Copyright © McGraw-Hill Education. Permission required for reproduction or display.  
01 START ST R1,SaveR1 ; Save registers needed  
02 ST R2,SaveR2 ; by this routine  
03 ST R3,SaveR3  
04 ;  
05 LD R2,Newline  
06 L1 LDI R3,DSR  
07 BRzp L1 ; Loop until monitor is ready  
08 STI R2,DDR ; Move cursor to new clean line  
09 ;  
0A LEA R1,Prompt ; Starting address of prompt string  
0B Loop LDR R0,R1,#0 ; Write the input prompt  
0C BRz Input ; End of prompt string  
0D L2 LDI R3,DSR  
0E BRzp L2 ; Loop until monitor is ready  
0F STI R0,DDR ; Write next prompt character  
10 ADD R1,R1,#1 ; Increment prompt pointer  
11 BRnzp Loop ; Get next prompt character  
12 ;  
13 Input LDI R3,KBSR  
14 BRzp Input ; Poll until a character is typed  
15 LDI R0,KBDR ; Load input character into R0  
16 L3 LDI R3,DSR  
17 BRzp L3 ; Loop until monitor is ready  
18 STI R0,DDR ; Echo input character  
19 ;  
20 L4 LDI R3,DSR  
21 BRzp L4 ; Loop until monitor is ready  
22 STI R2,DDR ; Move cursor to new clean line  
23 LD R1,SaveR1 ; Restore registers  
24 LD R2,SaveR2 ; to original values  
25 LD R3,SaveR3  
26 BRnzp NEXT_TASK ; Do the program's next task  
27 ;  
28 SaveR1 .BLKW 1 ; Memory for registers saved  
29 SaveR2 .BLKW 1  
30 SaveR3 .BLKW 1  
31 DSR .FILL xFE04  
32 DDR .FILL xFE06  
33 KBSR .FILL xFE00  
34 KBDR .FILL xFE02  
35 Newline .FILL x000A ; ASCII code for newline  
36 Prompt .STRINGZ "Input a character"
```

OS Service Routines

Instead of requiring a user program to know the details of device registers, we **abstract** the interaction with I/O by providing a service routine.

The service routine is invoked using a system call: TRAP.



[Access the text alternative for slide images.](#)

LC-3 System Call (Trap) Mechanism

1. A set of service routines (part of the OS).
2. A table of starting addresses for the service routines. This is called the Trap Vector Table.
3. The TRAP instruction, which transfers control to the service routine specified in the Trap Vector Table.
4. A linkage mechanism for returning control back to the user program.

The LC-3 Trap Vector Table has one entry per trap vector (x00 to xFF). It is stored at memory locations x0000 to x00FF.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.

x0000	•
•	•
•	•
x0020	x03E0
x0021	x0420
x0022	x0460
x0023	x04A0
x0024	x04E0
x0025	x0520
•	•
•	•
•	•
x00FF	•

[Access the text alternative for slide images.](#)

TRAP versus JSR

TRAP has a lot in common with JSR. Both want to:

- a) Transfer control by putting an address into the PC.
- b) Provide a way to get back to the calling routine.

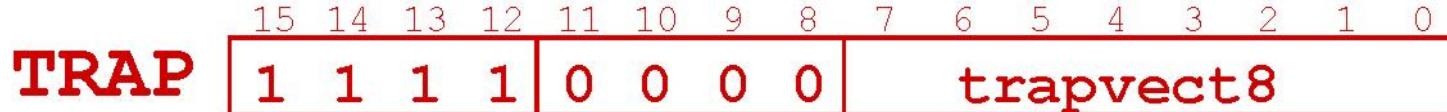
JSR uses:

- a) PC-Relative (JSR) or Base+offset (JSRR) addressing mode to specify the target address.
- b) R7 to save return address.

TRAP uses:

- a) Trap vector table to provide target address, indexed by trap vector.
- b) Supervisor stack to save return address.

TRAP Execution



1. Push PSR and PC onto Supervisor Stack.

If TRAP is executed in user mode, hardware switches R6 to point to the Supervisor Stack before pushing.

- Copy R6 to Saved_USP.
- Copy Saved_SSP to R6.

2. Set PSR[15] to 0, indicating supervisor mode.

This allows OS code (service routine) to execute privileged instructions and access privileged memory address.

3. Trap vector IR[7:0] is zero-extended to form the address of the specified entry in the trap vector table.

This address is loaded, which provides the starting address of the service routine. The starting address is placed into the PC.

RTI: Return from a Service Routine

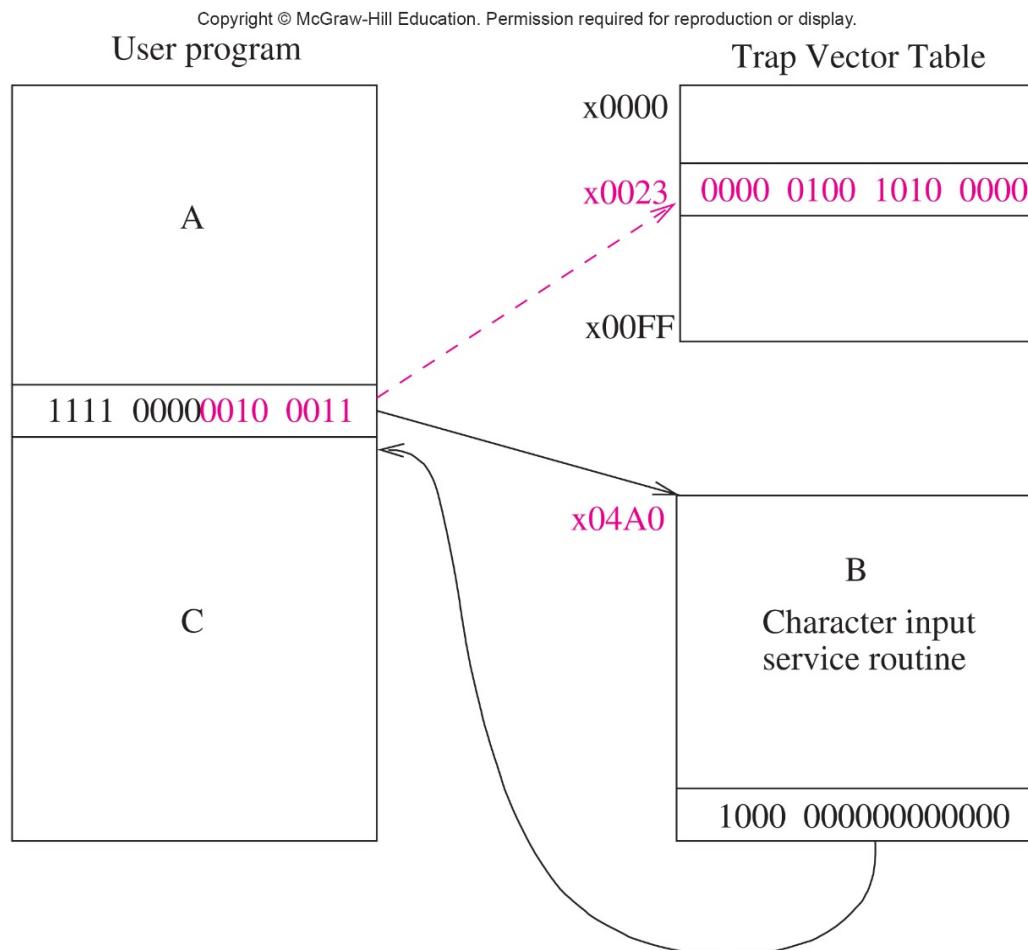
RTI	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RTI	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1. Pop **PC** from Supervisor Stack. This will resume the calling routine when this instruction completes.
2. Pop **PSR** from Supervisor Stack. This restores the priority and privilege level (and conditions codes) for the calling routine.
3. If PS[15] = 1, that means we have transitioned from supervisor mode to **user mode**, and we need to restore the User Stack Pointer.
 - Copy R6 to Saved_SS.
 - Copy Saved_USP to R6.

Note: If PS[15] = 0, that means the TRAP was called in supervisor mode, so the calling routine is still using the supervisor stack.

Trap Mechanism Summary

The picture illustrates what happens when the user program executes TRAP x23.



[Access the text alternative for slide images.](#)

Service Routine for Character Output

```
; Display character in R0 to monitor.  
.ORIG x0420 ; starting address of service routine  
ST R1,SaveR1  
  
Try    LDI R1,A      ; read DSR  
BRzp Try     ; loop until ready bit set  
STI R0,B      ; write DDR  
  
LD R1,SaveR1  
RTI  
  
A      .FILL xFE04    ; DSR address  
B      .FILL xFE06    ; DDR address  
SaveR1 .BLKW 1  
.END
```

Service Routine for Halting the Machine

In Chapter 3, we noted that we could halt the execution of the processor by ANDing a zero with the clock signal.

LC-3 uses the bit 15 of the Master Control Register (MCR) to control the clock signal. The MCR is memory-mapped to xFFFE.

Therefore, to stop the clock, we want to set MCR[15] to zero.

However, we don't want to change anything else in that register, so we first load the value, AND it with x7FFF, then write the result back.

In the following service routine, we also print a message to tell the user that the machine is about to halt.

Note that our service routine uses the TRAP instruction -- because the PC and PSR are pushed (and popped by RTI), we can safely nest calls to other service routines inside a service routine.

HALT Service Routine 1

```
.ORIG x0520
HALT      ST    R1, SaveR1      ; save registers
          ST    R0, SaveR0

          LD    R0, NewLine      ; print linefeed
TRAP x21
          LEA   R0, Message      ; print message
TRAP x22          ; (PUTS service routine)
          LD    R0, NewLine      ; another linefeed
TRAP x21

          LDI   R1, MCR          ; get current MCR
          LD    R0, MASK          ; bit mask to clear [15]
          AND  R1, R1, R0
          STI  R1, MCR          ; clear bit, stop clock

          ; return -- how can this code return if clock is stopped?
          LD    R1, SaveR1
          LD    R0, SaveR0
          RTI

          ; data on next page
```

HALT Service Routine ₂

```
NewLine      .FILL x0A      ; linefeed character
MASK        .FILL x7FFF    ; bit mask to clear bit 15
MCR         .FILL xFFFE    ; address of machine control reg (MCR)
Message     .STRINGZ "Halting the machine."

SaveR1      .BLKW 1
SaveR0      .BLKW 1

.END
```

Assembler Pseudo-Ops for Service Routines

For the programmer's convenience and to provide a layer of abstraction, the LC-3 assembler recognizes special "opcodes" to represent common usages of the TRAP instruction.

Pseudo-Opcode	Actual Instruction	Description
GETC	<code>TRAP x20</code>	read a single character (no echo) into R0
OUT	<code>TRAP x21</code>	print character (R0) to monitor
PUTS	<code>TRAP x22</code>	print string to monitor; R0 contains a pointer to the string
IN	<code>TRAP x23</code>	print a prompt, read a single character with echo into R0
HALT	<code>TRAP x25</code>	halt the machine

Interrupt-Driven I/O

I/O event may have nothing to do with the instructions that are being executed by the CPU. Allow the CPU instruction sequence to be **interrupted**, handle the I/O event, and then resume execution as if nothing had happened.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.

Program A is executing instruction n
Program A is executing instruction n+1
Program A is executing instruction n+2
1: Interrupt signal is detected
1: Program A is put into suspended animation
1: PC is loaded with the starting address of Program B
2: Program B starts satisfying I/O device's needs
2: Program B continues satisfying I/O device's needs
2: Program B continues satisfying I/O device's needs
2: Program B finishes satisfying I/O device's needs
3: Program A is brought back to life

Program A is executing instruction n+3
Program A is executing instruction n+4
.
.

Actual execution flow

Program A is executing instruction n
Program A is executing instruction n+1
Program A is executing instruction n+2
Program A is executing instruction n+3
Program A is executing instruction n+4
. . .



*Execution seen
by Program A*

Why Interrupts?

Processor can perform **useful computation** instead of "spinning" on ready bit.

Example:

Suppose program needs to (a) read a 100-character sequence from the keyboard and (b) process the input -- repeat 1000 times.

Assume user types 80 words/minute = 0.125 secs/character.

Assume processing takes 12.49999 seconds.

How long to execute entire program?

Without interrupts: $0.125 \times 100 = 12.5$ secs to read data. Most of that time is spent polling. Total time = 24.49999 seconds for each sequence = 7 hrs

With interrupts: Instead of polling, can process previous 100-char sequence and only spend a few instructions on each character as it's typed. By overlapping, time reduced to 12.5 seconds per sequence = 3.5 hrs.

Interrupt Mechanism: Two Parts

1. A mechanism that enables an I/O device to interrupt the processor
 - I/O device **must want** service.
 - I/O device **must have the right** to request service.
 - Device request **must be more urgent** than current processor execution.
2. A mechanism that handles the interrupt request
 - Similar to a subroutine call, but unscripted -- not anticipated or requested by the program code.

Part 1: Initiating the Interrupt Signal

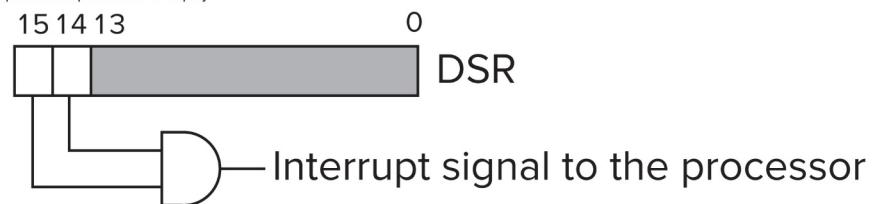
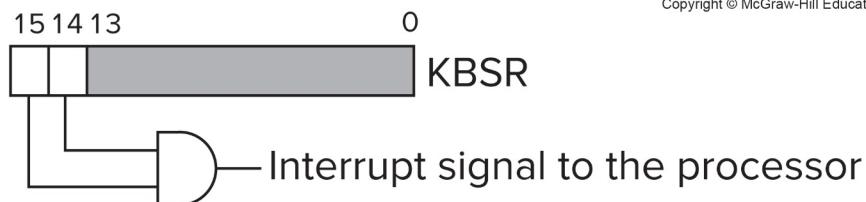
An interrupt signal may occur whenever some event occurs that causes the device to need service.

Examples: Key is pressed on the keyboard. Character has been displayed on the monitor. Network packet has arrived. Temperature sensor has exceeded its threshold.

Interrupts must be **enabled** from a particular device.

Typically controlled by a bit in the status/control register.

For LC-3: Both KBSR and DSR use bit 14 to enable interrupts. When this bit is set and the ready bit is set, an interrupt signal is generated.



[Access the text alternative for slide images.](#)

Interrupt Priority

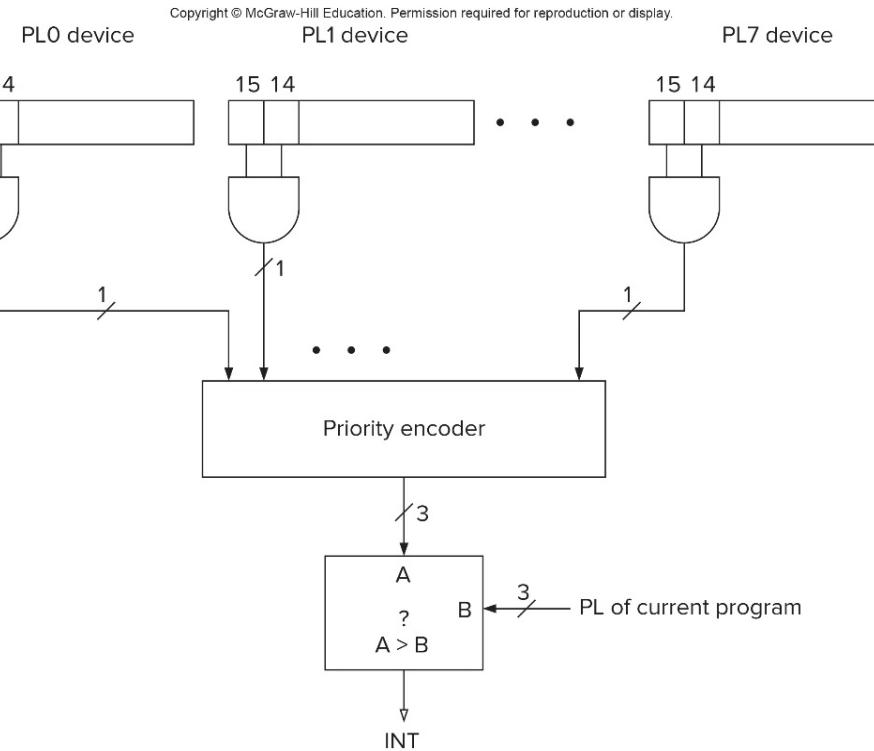
Is this interrupt more urgent than the code currently running?

LC-3 priority is held in PSR[10:8].

Each device is assigned an interrupt priority level: 0 to 7.

Interrupt signal (INT) is handled by CPU only if the priority is higher than the current program.

If multiple devices want to interrupt, only highest priority signal is sent to the CPU.



[Access the text alternative for slide images.](#)

Part 2: Handling the Interrupt

INT signal is inspected before the FETCH phase of each instruction.

This means that an interrupt will not disrupt the execution of an instruction -- it only changes execution flow between instructions. The instruction is the fundamental unit of execution: once an instruction is fetched, it is guaranteed to complete execution.

If the INT signal is asserted:

1. Save state of the currently executing program.
2. Transfer control to the code that handles this particular interrupt.
3. Restore state of the program and fetch the next instruction.

Saving Program State

Program state = **PC, PSR, registers, memory used by program**

Minimal state is **PC + PSR**. These will be **pushed** to the **Supervisor Stack**, as we do for the TRAP instruction. Interrupt handling code is responsible for saving and restoring registers as needed.

As with TRAP, changing from user mode to supervisor mode will swap R6 from the user stack pointer (USP) to the supervisor stack pointer (SSP).

NOTE: Condition codes are part of the PSR. This means that even if a program is interrupted right before a BR, it will make the same decision as if the interrupt didn't happen, because the condition codes will not change.

Transferring Control to Interrupt Handler

How does the hardware know what to put in the PC?

Where is the code that handles this interrupt?

Along with the INT signal, the interrupting device provides an 8-bit interrupt vector. This is used to look up a starting address for the interrupt handling code, similar to TRAP.

LC-3 Interrupt Vector Table is in memory at x0100 to x01FF.

Initializing PSR:

- Conditions codes are set to 010. (No particular reason, but must be set to something...).
- PSR[15] = 0, supervisor mode.
- PSR[10:8] = priority level of interrupt signal.

Returning from Interrupt: RTI

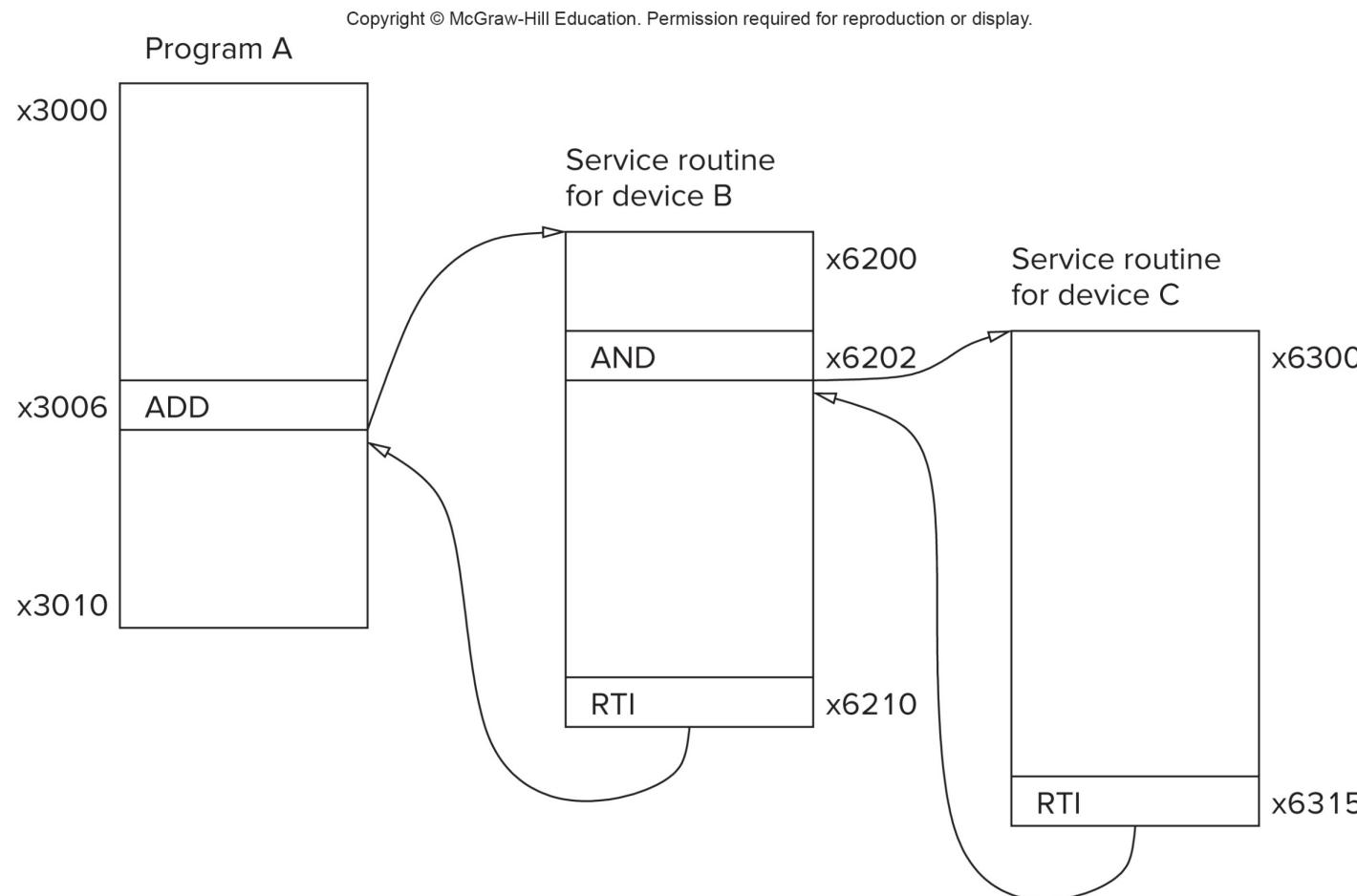
When we are ready to return from the interrupt handling routine, we are in the same situation as a TRAP service routine:

- PSR and PC of the "calling" routine are on the supervisor stack.
- Pop the stack, restore the USP (if going back to user mode).

This exact sequence is performed by the **RTI** instruction, described earlier.

Interrupt Example

Program A is interrupted by Device B. While interrupt service routine for B is running, it is interrupted by Device C, which has a higher priority.



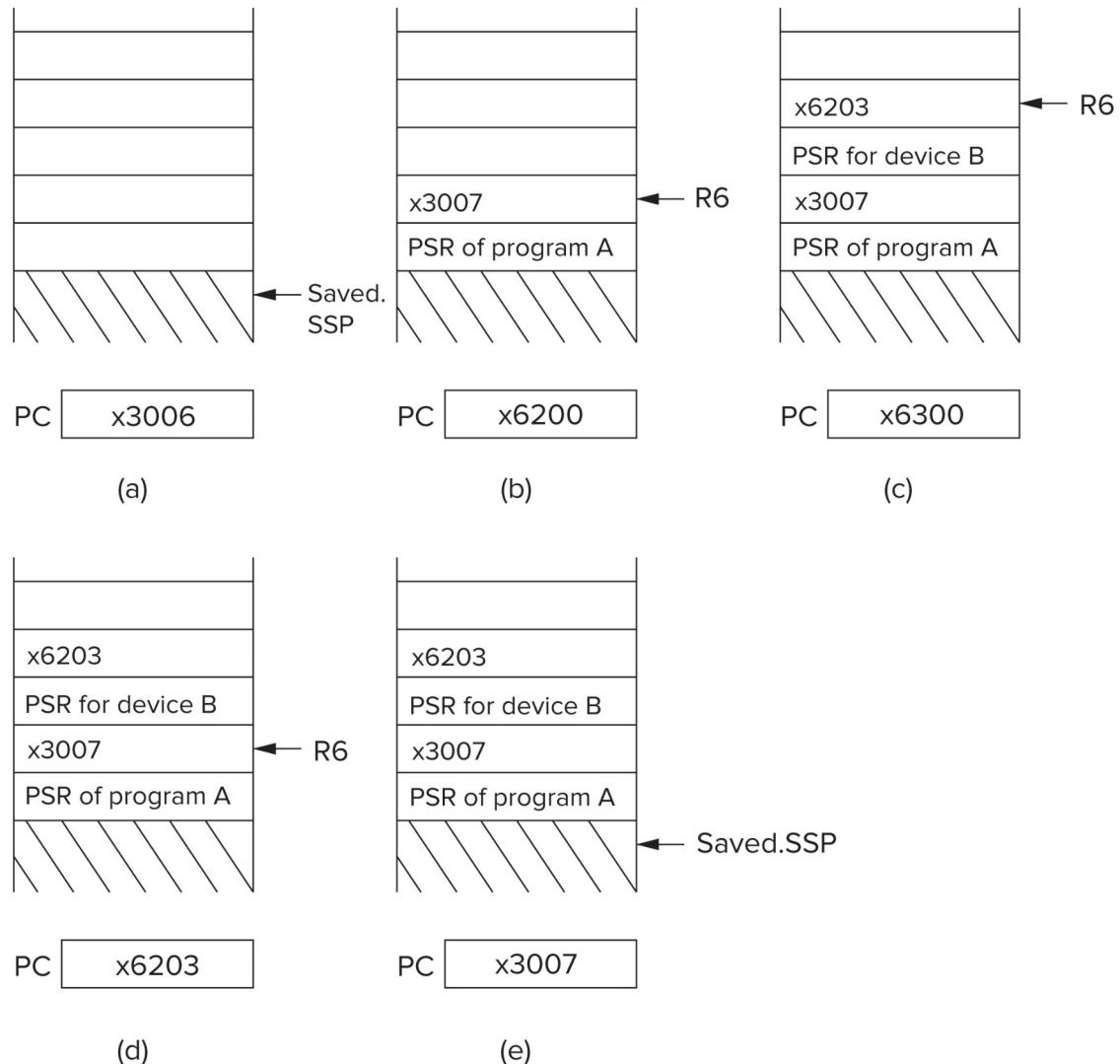
[Access the text alternative for slide images.](#)

Nested Interrupts: Enabled by the Stack

The nesting of interrupts described on the previous slide is enabled by the user of the supervisor stack to save PC and PSR state.

- (a) Program A is running.
- (b) Device B interrupts, saving A's PC and PSR. R6 is written to Saved_USP, and Saved_SSP is written to R6.
- (c) Device C interrupts, saving B's PC and PSR.
- (d) Device C handler returns, restoring B's PC and PSR and popping the stack.
- (e) Device B handler returns, restoring A's PC and PSR and popping the stack. R6 is written to Saved_SSP, and Saved_USP is written to R6.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



[Access the text alternative for slide images.](#)

Interrupts: Not Just for I/O

There are events that happen in a computer system that require attention, but are not (directly) related to I/O devices:

- Timer expires.
- Machine check -- component failure.
- Power failure.

These events can also generate INT signals, vectors, etc., and can be handled in the same way as the interrupts we have been discussing.

In addition, there may be abnormal events caused by the execution of instructions in the CPU, such as:

- Divide by zero.
- Illegal instruction.

These "internal interrupts" are often called **exceptions**.

Polling Revisited: What about Interrupts?

We usually think of a polling loop as "atomic" -- that is, if the ready bit is set and the branch is not taken, we "immediately" perform the related load/store operation to the device.

Example -- the output character loop:

```
POLL  LDI R1,DSR      ; check monitor  
      BRzp POLL        ; keep checking if not read  
      STI R0,DDR        ; ready -- store the character to device
```

But now we know that an interrupt can happen at any time, and significant time may elapse before returning control to the program.

What if an interrupt happens between the BRzp and the STI, and when the handler returns, the device is **no longer ready** to accept a character?

In other words, the status that was returned by the LDI is no longer valid because something else occurred between the LDI and the STI.

Disabling Interrupts¹

To avoid the problem (and many others like it), we may want to disable interrupts during certain sequences of instructions.

Mechanism:

- Use bit 14 of the PSR as a global interrupt enable bit. It is set to 1 by default, so that interrupts are enabled. But we can ignore all interrupts by setting it to 0.
- We also need to memory-map the PSR. We'll use xFFFC.

Disabling Interrupts 2

```
; To disable interrupts, write 0 to PSR[14].  
; Use a mask to preserve all other bits.  
    LDI R1,PSR      ; get current PSR bits  
    LD   R2,INTMASK  
    AND R2,R2,R1    ; R2 now has bit 14 clear  
    STI R2,PSR      ; disable interrupts  
    ...  
    ...  
    STI R1,PSR      ; enable when ready  
    BRnzp NEXT  
PSR    .FILL xFFFC    ; PSR address  
INTMASK .FILL xBFFF
```

Disabling Interrupts: DANGER

If we disable interrupts for long periods of time, we may miss very important events and prevent high-priority tasks from running.

Better: Code below shows a polling loop that reenables interrupts at the beginning of each loop -- this allows the important LDI - BR - STI sequence to be uninterrupted, but does not disable interrupts for the entire polling period. (See complete code in Figure 9.22.)

```
LDI    R1, PSR
LD     R2, INTMASK
AND   R2,R1,R2      ; R1=original PSR, R2=PSR with interrupts disabled

POLL
  STI   R1,PSR      ; enable interrupts (if they were enabled to begin)
  STI   R2,PSR      ; disable interrupts
  LDI   R3,DSR
  BRzp POLL        ; Poll the DSR
  STI   R0,DDR      ; Store the character into the DDR
  STI   R1,PSR      ; Restore original PSR
```

[Access the text alternative for slide images.](#)



Because learning changes everything.[®]

www.mheducation.com

From Bits and Gates to C and Beyond

A Calculator

Chapter 10

Calculator: A Comprehensive Example

Write a program that implements a simple arithmetic calculator.

- Operations: add, subtract, and multiply integers.
- User interface: input characters from keyboard, output to monitor.
- Stack-based calculator -- integers are pushed onto the stack and then operations are performed on the stack values.

Sample Run:

Enter a command: **123**

Push values onto the stack

Enter a command: **30**

Pop the top two values, add them, and push the result

Enter a command: **+**

Display (print) the value at the top of stack

Enter a command: **D**

User input in bold

Program output

Restriction: Integer inputs and results must be within the range -999 to +999.

Program Organization

Main program + 11 subroutines

- ASCII to binary conversion.
- Binary to ASCII conversion.
- Push value (from user) to the stack.
- Display (print) value at top of stack.
- Add.
- Multiply.
- Negate.
- Range check.
- Clear stack.
- Push (from Chapter 8).
- Pop (from Chapter 8).

Topics that need further explanation

Converting between ASCII
and 2's complement

Arithmetic using a stack

Data Type Conversion: ASCII Strings to Binary Integers

User input is based on ASCII strings -- characters typed on the keyboard.

In this program, users type decimal integers: "123."

A linefeed (or Enter) must be typed after each input.

LC-3 operations work on 2's complement binary integers, not strings.

So we have to convert the string "123" into the 2's complement representation of the integer 123.

Not all numbers are three digits -- user might type "6" or "30."

Key insight: Each digit corresponds to a **multiple of 1, 10, or 100**, depending on its position in the string.

ASCII to Binary: Data Storage

As characters are read, they will be stored in an array called ASCIIBUFF. Instead of using a null terminator for the string, we will track how many digits were entered.

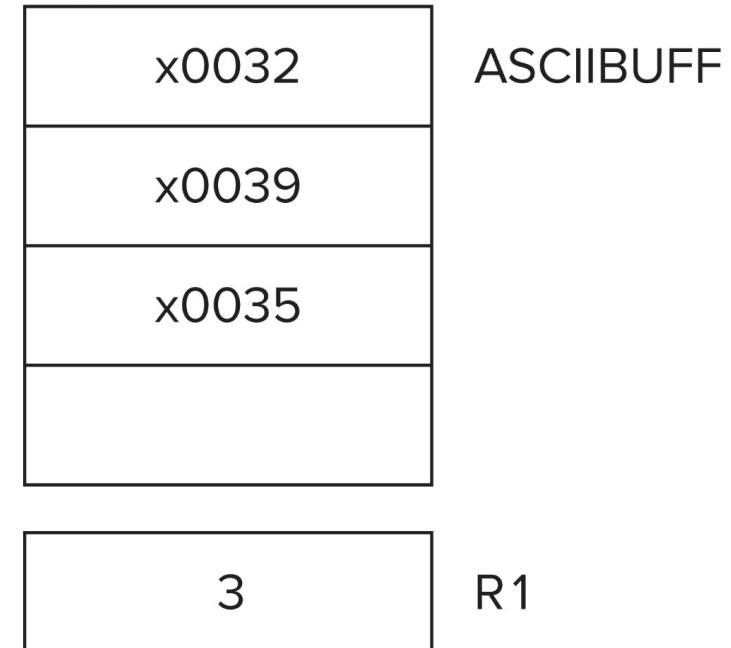
The figure shows "295" in the buffer.

The ASCII codes for '2' and '9' and '5' are in memory, and R1 tells us that three digits were entered.

Since no null terminator, why do we show four memory locations for the buffer? This same buffer will be used for converting binary to ASCII, and we will need space for a minus sign '-' for negative values.

Why not use a null terminator? We are doing character-by-character I/O and we have a limited number of characters -- the null terminator doesn't provide any benefit.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



[Access the text alternative for slide images.](#)

ASCII to Binary: Computation

Given the data structure on the previous slide, we can easily identify the ones digit, tens digit (if any), and hundreds digit (if any). Need to do the following:

- Convert ones digit to a binary value (subtract x30).
- Convert tens digit to binary value and multiply by 10.
- Convert hundreds digit to binary value and multiply by 100.
- Add the three values together.

No multiply instruction? Next slide...

ASCII to Binary: Multiply by Lookup Table

We do not need a general multiply routine, because there are only 10 interesting multiples of ten and 10 interesting multiples of 100.

Use a lookup table -- put the multiples of 10 (or 100) in an array, and choose the right value based on an index.

0	0
10	1
20	2
30	3
40	4
50	5
60	6
70	7
80	8
90	9

$n \times 10 = \text{array element } n$
Read memory location (array + n)

0	0
100	1
200	2
300	3
400	4
500	5
600	6
700	7
800	8
900	9

$n \times 100 = \text{array element } n$
Read memory location (array + n)

ASCII to Binary: Algorithm

Initialize R0 (result) to 0.

R1 is number of digits (1, 2, or 3).

Initialize pointer (R2) to ASCIIBUFF + R1 - 1.

If $R1 > 0$:

$R4 = M[R2]$ (ones digit)

Convert $R4$ to binary value (0-9).

Add to $R0$.

Decrement $R1$.

If $R1 > 0$:

$R4 = M[R2-1]$ (tens digit)

Convert $R4$ to binary value (0-9).

Multiply by 10 and add to $R0$.

Decrement $R1$.

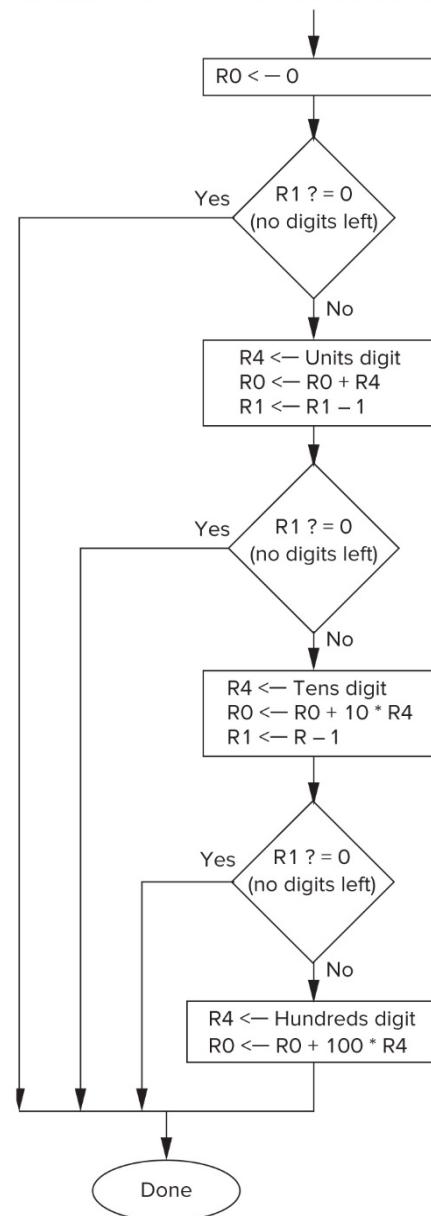
If $R1 > 0$:

$R4 = M[R2-2]$ (hundreds digit)

Convert $R4$ to binary value (0-9).

Multiply by 100 and add to $R0$.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



[Access the text alternative for slide images.](#)

ASCII to Binary Subroutine: Part 1

```
; Convert ASCII string to 2's complement binary.  
; ASCIIIBUFF is address of string buffer. R1 is number of digits.  
; Result returned in R0.
```

```
ASCIItoBinary    ST    R1, AtoB_SR1      ; save registers  
                  ST    R2, AtoB_SR2  
                  ST    R3, AtoB_SR3  
                  ST    R4, AtoB_SR4  
                  AND   R0, R0, #0      ; initialize result  
                  ADD   R1, R1, #0      ; if zero digits, result is 0  
                  BRz  AtoB_DONE  
  
                  LD    R2, AtoB_ASCIIIBUFF  
                  ADD   R2, R2, R1  
                  ADD   R2, R2, #-1      ; point to one's digit  
  
                  LDR   R4, R2, #0      ; load ones digit  
                  AND   R4, R4, x0F      ; lowest four bits = value  
                  ADD   R0, R0, R4      ; add to result  
                  ADD   R1, R1, #-1      ; decrement counter  
                  BRz  AtoB_DONE
```

ASCII to Binary Subroutine: Part 2

```
LDR R4, R2, #-1      ; load tens digit
AND R4, R4, x0F      ; convert to binary
LEA R3, Lookup10     ; index into lookup table
ADD R3, R3, R4
LDR R4, R3, #0
ADD R0, R0, R4      ; add to result
ADD R1, R1, #-1      ; decrement counter
BRz AtoB_DONE

LDR R4, R2, #-2      ; load hundreds digit
AND R4, R4, x0F
LEA R3, Lookup100    ; multiply by 100
ADD R3, R3, R4
LDR R4, R3, #0
ADD R0, R0, R4      ; add to result

AtoB_DONE           LD R1, AtoB_SR1    ; restore regs and return
                    LD R2, AtoB_SR2
                    LD R3, AtoB_SR3
                    LD R4, AtoB_SR4
                    RET
```

ASCII to Binary Subroutine: Part 3

```
AtoB_ASCIIBUFF    .FILL  ASCIIIBUFF      ; pointer to string buffer
AtoB_SR1          .BLKW  1
AtoB_SR2          .BLKW  1
AtoB_SR3          .BLKW  1
AtoB_SR4          .BLKW  1
Lookup10          .FILL  #0          ; lookup table for x10
                           .FILL  #10
                           .FILL  #20
                           .FILL  #30
                           .FILL  #40
                           .FILL  #50
                           ; 60-90 go here: removing values to fit on slide...
Lookup100         .FILL  #0
                           .FILL  #100
                           .FILL  #200
                           .FILL  #300
                           .FILL  #400
                           .FILL  #500
                           ; 600-900 go here...
```

Exercise: Make this code safer by checking if entered string is a 3-digit decimal number.

Binary to ASCII

Data Storage: Same buffer as before. Resulting buffer will always have four characters -- a sign (+ or -) and three digits, including leading zeroes.

Computation: Instead of multiplying by 10 or 100, we need to divide. The lookup table method is not so helpful this time. Our input value is in the range 0 to 999, so we would need 1000 entries in the lookup table. Instead, we will repeatedly subtract 10 (or 100) until the result goes negative. If we count how many times we subtracted, that is the quotient.

Example: Divide 392 by 100.

Quotient = 0

$392 - 100 = 292$: ≥ 0 , quotient = 1

$292 - 100 = 192$: ≥ 0 , quotient = 2

$192 - 100 = 92$: ≥ 0 , quotient = 3

$92 - 100 = -8$: < 0 , STOP! quotient = 3

Binary to ASCII: Algorithm

R0 is number to be converted.

Initialize pointer (R1) to ASCIIBUFF.

If R0 < 0:

 Store minus sign (x2D) to M[R1].

 Negate R0 (to make it positive).

Else:

 Store plus sign (xB2) to M[R1].

R2 = '0'

Each time we subtract 100 and get a positive value, add +1 to R2.

Store R2 (hundreds digit) to M[R1+1].

R2 = '0'

Each time we subtract 10 and get a positive value, add +1 to R2.

Store R2 (tens digit) to M[R1+2].

R2 = '0' + remainder (ones digit), store to M[R1+3].

Binary to ASCII Subroutine: Part 1

; Convert value in R0 to 4-character ASCII buffer: +xxx or -xxx.
; ASCIIIBUFF is address of string buffer.

```
BinaryToASCII      ST    R0, BtoA_SR0      ; save registers
                  ST    R1, BtoA_SR1
                  ST    R2, BtoA_SR2
                  ST    R3, BtoA_SR3
                  LD    R1, BtoA_ASCIIIBUFF ; pointer to buffer

                  ADD   R0, R0, #0       ; check for +/-  
BRn   NegSign
                  LD    R2, ASCIIIPplus ; store '+'
                  BRNzp Begin100
NegSign          LD    R2, ASCIINeg    ; store '-'
                  NOT   R0, R0
                  ADD   R0, R0, #1      ; absolute value

Begin100         STR   R2, R1, #0
```

Binary to ASCII Subroutine: Part 2

```
        LD    R2, ASCIIOffset ; R2 = '0' (hundreds digit)
        LD    R3, Neg100
Loop100      ADD   R0, R0, R3
              BRn  End100
              ADD   R2, R2, #1      ; next digit
              BRNzp Loop100
End100       STR   R2, R1, #1      ; store hundreds digit char
              LD    R3, Pos100
              ADD   R0, R0, R3      ; restore R0 to positive

        LD    R2, ASCIIOffset ; R2 = '0' (tens digit)
Loop10       ADD   R0, R0, #-10
              BRn  End10
              ADD   R2, R2, #1      ; next digit
              BRNzp Loop10
End10        STR   R2, R1, #2      ; store tens digit char
              ADD   R0, R0, #10     ; restore R0 to positive
```

Binary to ASCII Subroutine: Part 3

```
        LD    R2, ASCIIoffset ; R2 = '0' (ones digit)
        ADD   R2, R2, R0      ; convert to char
        STR   R2, R1, #3      ; store ones digit char

        LD    R0, BtoA_SR0    ; restore regs and return
        LD    R1, BtoA_SR1
        LD    R2, BtoA_SR2
        LD    R3, BtoA_SR3
        RET

ASCIIPlus          .FILL '+'
ASCIINeg           .FILL '-'
ASCIIOffset        .FILL '0'
Neg100            .FILL #-100
Pos100            .FILL #100
BtoA_SR0          .BLKW 1
BtoA_SR1          .BLKW 1
BtoA_SR2          .BLKW 1
BtoA_SR3          .BLKW 1
BtoA_ASCIIBUFF   .FILL ASCIIBUFF
```

Data Conversion Challenges

Challenge #1: ASCII to Binary (Exercise 10.4)

Devise an algorithm that can convert a input strings of arbitrary size.

We can't create an indefinite number of lookup tables, because we don't know the maximum power of 10 needed. Also assume that we don't know the maximum number of bits in a binary integers.

Challenge #2: Binary to ASCII (Exercise 10.6)

Devise an algorithm that does not create unneeded characters. In other words, do not create leading zeroes or a leading '+'. You can still assume that the range of integer values is -999 to +999.

Using a Stack for Arithmetic

While LC-3 and modern ISAs use general-purpose registers for temporary storage, some ISAs use no registers at all. So-called **stack machines** use a memory stack for all source and destination operands.

Example:

- ADD specifies no source or destination operands.
The instruction always **pops two values** from the stack, **adds** them together, and then **pushes the result** to the stack.

In this architecture, the number of temporary values is limited only by the size of the memory stack, and the programmer never needs to keep track of which register holds which value.

Arithmetic Expressions: Postfix Notation

We are used to writing arithmetic expressions using **infix** notation, with the operator in between the operands: **100 + 47**

For a stack machine, it's more convenient to write the expression using **postfix** notation, where the operator comes after the relevant operands, like this: **100 47 +**

This can be interpreted as: push 100, push 47, ADD

The result (147) is pushed onto the stack when the ADD is complete.

More complex expressions can be handled with more temporary values on the stack.

Infix

$(25 + 17) \times (3 + 2)$

Postfix

25 17 + 3 2 + x

There's a reasonably simple algorithm to convert an expression from infix to postfix notation, but it's beyond the scope of this discussion. (But it uses a stack!)

Arithmetic Subroutines

We will need three subroutines for the three operations performed by our calculator.

OpAdd Pop two integers from the stack, add, push the result.

OpMult Pop two integers from the stack, multiply, push the result.

OpNeg Pop one integer from the stack, negate, push the result.

These routines all use the User Stack and the Push/Pop subroutines defined in Chapter 8.

We also create a **RangeCheck** subroutine, to make sure that an integer value (in R0) is within the specified range of -999 to +999. If out of range, an error message is printed and a value of 1 is returned in R5. Otherwise, 0 is returned in R5.

OpAdd Subroutine

```
OpAdd           ; save regs: R0, R1, R5, R7 (omitted to save space)
JSR  POP          ; pop first arg into R0
ADD  R5, R5, #0   ; check for success
BRp OpAdd_EXIT

ADD  R1, R0, #0   ; move to R1
JSR  POP          ; pop second arg into R0
ADD  R5, R5, #0   ; check for success
BRp OpAdd_Restore1 ; if fail, put 1st back on stack

ADD  R0, R0, R1   ; perform the add
JSR  RangeCheck  ; check result
ADD  R5, R5, #0
BRp OpAdd_Restore2 ; if fail, put both args back
JSR  PUSH         ; push result

OpAdd_Restore2   ADD  R6, R6, #-1
OpAdd_Restore1   ADD  R6, R6, #-1
OpAdd_Exit        ; restore regs (omitted to save space)
RET
```

RangeCheck Subroutine

```
; R5 = 0 if -999 <= R0 <= +999; otherwise, R5 = 1
RangeCheck        LD    R5, Neg999
                  ADD   R5, R0, R5
                  BRp  BadRange ; R0 > 999
                  LD    R5, Pos999
                  ADD   R5, R0, R5
                  BRn  BadRange ; R0 < -999
                  AND   R5, R5, #0
                  RET
BadRange          ST    R0, RangeCheck_SR0
                  LEA   R0, RangeErrorMsg
                  PUTS
                  AND   R5, R5, #0
                  ADD   R5, R5, #1
                  LD    R0, RangeCheck_SR0
                  RET
Neg999           .FILL #-999
Pos999           .FILL #999
RangeErrorMsg    .FILL x000A ; linefeed
                  .STRINGZ "Error: Number is out of range."
RangeCheck_SR0   .BLKW 1
```

OpMult and OpNegate

Subroutines are similar to OpAdd.

OpMult -- see Figures 10.11 and 10.12

OpNeg -- see Figure 10.13

Complete Program: User Interface

Program will print a prompt for the user to enter a command.

If the command is X (exit), the machine halts.

Otherwise, the command is performed and the program prompts for another command. Each command will be echoed as the user types.

Cmd Character	Description
X	Exit the program. (Halts the machine.)
C	Clear the stack. All temporary values are removed.
+	ADD -- pop two values, add, push result.
*	MULTIPLY -- pop two values, multiply, push result.
-	NEGATE -- pop one value, negate, push result.
LF/CR	User types a positive decimal integer, up to 3 digits, followed by linefeed (or Enter). Value is pushed to the stack.

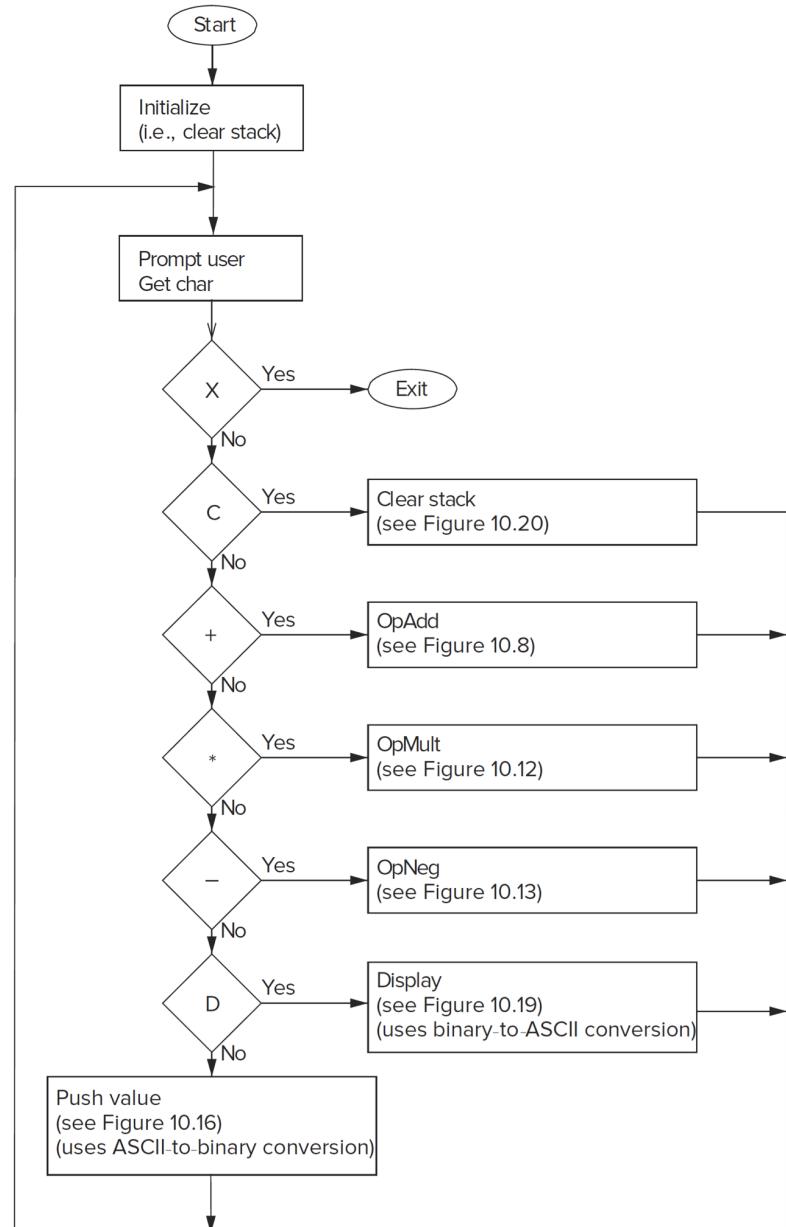
Program Flowchart

The flow of the main program is shown to the right. Many subroutines have been defined, but we have three more:

OpClear Clear the stack.

OpDisplay Convert top of stack to ASCII and print.

PushValue Convert user input to binary and push.



[Access the text alternative for slide images.](#)

OpClear Subroutine

```
; Clear the stack -- set R6 (stack pointer) to its max value  
; Main program defines StackBase as max. address allocated to stack  
  
OpClear          LD    R6, OpClear_StackBase  
                  ADD   R6, R6, #1      ; empty = one beyond highest addr  
                  RET  
  
OpClear_StackBase .FILL StackBase
```

OpDisplay Subroutine

```
; Convert top of stack to ASCII -- store chars in ASCIIBUFF
; (Do not pop the stack.) Print linefeed after string.

OpDisplay      ST    R0, OpDisplay_SR0      ; save regs
                ST    R5, OpDisplay_SR5
                ST    R7, OpDisplay_SR7
                JSR   POP               ; pop stack into R0
                ADD   R5, R5, #0        ; check for error
                BRp  OpDisplay_DONE
                JSR   BinaryToASCII     ; convert R0 to ASCII
                LD    R0, NewlineChar
                OUT
                LD    R0, OpDisplay_ASCIIBUFF
                PUTS
                ADD  R6, R6, #-1 ; push displayed number back on stack
                ; restore regs (omitted for space)
                RET
OpDisplay_DONE
NewlineChar     .FILL x000A
OpDisplay_ASCIIBUFF .FILL ASCIIBUFF
OpDisplay_SR0   .BLKW 1
OpDisplay_SR5   .BLKW 1
OpDisplay_SR7   .BLKW 1
```

PushValue Subroutine: Part 1

```
; Called when command is not X, C, +, *, or -.
; Expect user input to be 0-3 digits followed by linefeed.
; Convert to binary.

PushValue          ; save R0, R1, R2, R7 (omitted for space)
    LD   R1, PV_ASCIIIBUFF    ; ptr to string buffer
    LD   R2, MaxDigits        ; will count down to check # digits

ValueLoop         ADD  R3, R0, #-10      ; if linefeed
    BRz GoodValue
    ADD  R2, R2, #0          ; if 4th digit, bad value
    BRz TooLargeInput
    LD   R3, NegASCII0        ; check for digit
    ADD  R3, R0, R3
    BRn NotInteger
    LD   R3, NegASCII9
    ADD  R3, R0, R3
    BRp NotInteger
    ADD  R2, R2, #-1          ; count digit and store
    STR R0, R1, #0
    ADD  R1, R1, #1
    GETC                      ; read and echo next char
    OUT
    BRnzp ValueLoop
```

PushValue Subroutine: Part 2

```
GoodInput      LD    R2, PV_ASCIIIBUFF
                NOT   R2, R2
                ADD   R2, R2, #1
                ADD   R1, R1, R2          ; calculate # digits
                BRz  NoDigit
                JSR   ASCIIIToBinary     ; push binary value to stack
                JSR   PUSH
                BRnzp PushValue_DONE

NoDigit        LEA   R0, NoDigitMsg
                PUTS
                BRnzp PushValue_DONE

NotInteger     GETC           ; read chars until linefeed
                OUT
                ADD   R0, R0, #-10
                BRnp NotInteger
                LEA   R0, NotIntegerMsg
                PUTS
                BRnzp PushValue_DONE
```

PushValue Subroutine: Part 3

```
TooLargeInput    GETC                      ; get chars until linefeed
                OUT
                ADD  R0, R0, #-10
                BRnp TooLargeInput
                LEA   R0, TooManyDigits
                PUTS

PushValue_DONE  ; restore regs (omitted for space)
                RET

TooManyDigits  .FILL x000A      ; linefeed before message
                .STRINGZ "Too many digits"
                .FILL x000A
                .STRINGZ "Not an integer"
                .FILL x000A
                .STRINGZ "No digits entered"
                .FILL #3
                .FILL x-30
                .FILL x-39
                .FILL ASCIIBUFF

NotIntegerMsg
NoDigitMsg
MaxDigits
NegASCII0
NegASCII9
PV_ASCIIBUFF
```

Program Main Routine: Part 1

```
; stack of 10 values is allocated at end of main routine
    LEA    R6, StackBase
    ADD    R6, R6, #1          ; empty stack

NewCommand      LEA    R0, PromptMsg      ; print prompt
                PUTS
                GETC              ; get command character and echo
                OUT

TestX          LD     R1, NegX      ; check for X
                ADD    R1, R1, R0
                BRnp  TestC       ; if no match, go to next command
                HALT

TestC          LD     R1, NegC      ; check for C
                ADD    R1, R1, R0
                BRnp  TestAdd
                JSR    OpClear
                BRnzp NewCommand

; similar code (omitted) for +, *, -, D
```

Program Main Routine: Part 2

```
; if reaches this point, none of the other commands have matched  
; should be a number
```

```
EnterNumber      JSR    PushValue  
                  BRnzp NewCommand
```

```
PromptMsg       .FILL  x000A  
                  .STRINGZ "Enter a command: "
```

```
NegX            .FILL  xFFA8      ; negative of 'X'  
NegC            .FILL  xFFBD      ; negative of 'C'  
                  ; etc.
```

```
StackMax        .BLKW  9       ; space for stack
```

```
StackBase       .BLKW  1
```

```
ASCIIBUFF       .BLKW  4  
                  .FILL  x0000 ; null terminator for display string
```



Because learning changes everything.[®]

www.mheducation.com

From Bits and Gates to C and Beyond

Variables and Operators

Chapter 12

Bitwise Operators

C bitwise operators apply a logical operation across the individual bits of its operands.

Only applies to integers, or to smaller types that can be treated as integer (`char` or `bool`). The operation is applied to the 2's complement binary representation of the integer value.

Expression	Operation	Result
<code>0x1234 0x5678</code>	OR	<code>0x567c</code>
<code>0x1234 & 0x5678</code>	AND	<code>0x1230</code>
<code>0x1234 ^ 0x5678</code>	XOR	<code>0x444c</code>
<code>~0x1234</code>	NOT	<code>0xedcb</code>
<code>1234 & 5678</code>	AND	1026

Table 12.2 Bitwise Operators in C

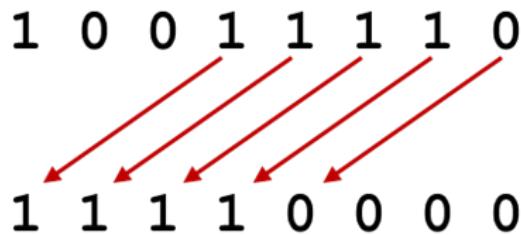
Operator symbol	Operation	Example usage
<code>~</code>	bitwise NOT	<code>~x</code>
<code>&</code>	bitwise AND	<code>x & y</code>
<code> </code>	bitwise OR	<code>x y</code>
<code>^</code>	bitwise XOR	<code>x ^ y</code>
<code><<</code>	left shift	<code>x << y</code>
<code>>></code>	right shift	<code>x >> y</code>

Hex and decimal are both integers. Don't let the notation confuse you. The same binary operation is performed, no matter whether you write the values in base-10 or base-16.

Shift Operators

The `<<` and `>>` operators are used to perform a bitwise shift of an integer value, to the left (`<<`) or right (`>>`). The left operand is the value to be shifted, and the right operand is the number of bit positions to shift.

For the purposes of illustration, let's assume that integer values are 8 bits.
Evaluate `0x9e << 3`



Each bit is shifted left by three positions. The leftmost bits are "shifted off" and we fill in from the right with zeroes. Result: `0xF0`

[Access the text alternative for slide images.](#)

Using Bitwise Operators

We've encountered the bitwise operators before, in Chapter 2, but let's review how they are typically used:

AND: clearing bits

- Create a mask with 0's for the bits you want to clear, 1's for the bits you want to keep.

OR: setting bits

- Create a mask with 1's for the bits you want to set, 0's for the bits you want to keep.

XOR: flipping bits

- Create a mask with 1's for the bits you want to flip, 0's for the bits you want to keep.

Machine-independent (portable) C code should not assume that you know the size of an `int` or any other data type.

Shift operators are useful to move bits where you need them to be. You might want to shift the mask, or you might want to shift the value being masked...

C Program

```
#include <stdio.h>    // need this to use printf and scanf
int main(void)
{
    int amount;    // number of bytes to transfer -- get from user
    int rate;      // avg network transfer rate -- get from user
    int time;      // transfer time in seconds
    int hours, minutes, seconds; // convert to hr:min:sec

    // Get input from user: file size (bytes) and network speed (bytes/sec)
    printf("How many bytes of data to be transferred? ");
    scanf("%d", &amount);
    printf("What is the transfer rate (in bytes/sec)? ");
    scanf("%d", &rate);

    // Calculate time in seconds
    time = amount / rate; // truncates any fractional part

    // Use arithmetic operators to calculate hours, mins, secs
    hours = time / 3600;           // throws away remainder
    minutes = (time % 3600) / 60; // remainder, converted to minutes
    seconds = ((time % 3600) % 60);

    // Output results
    printf("Time : %dh %dm %ds\n", hours, minutes, seconds);
}
```

Translating C Code to LC-3 Assembly Language

Before we learn about more features of the C language, let's consider how these basic C statements would be translated into LC-3 instructions.

What is compiler's task?

1. **Allocate memory** for variables in a systematic way.
Will build a **symbol table** to keep track of variable type, size, location.
2. **Generate instruction sequences** that carry out the computations specified by operators and statements.

For this class, we will compile "by hand" to get a sense of how these translations occur.

Symbol Table

For the LC-3 assembler, the symbol table tracked the memory address for each label in the program.

For the C compiler, we need to track the type and scope information. The type tells us how big each variable's memory object should be. The scope tells us where the memory object should be allocated.

Copyright © McGraw-Hill Education. Permission required for reproduction or display

Identifier	Type	Location (as an offset)	Scope	Other Info...	
amount	int	0	main	...	<i>This is a symbol table generated for the sample program we just wrote.</i>
hours	int	-3	main	...	<i>There is one entry for each declared variable.,</i>
minutes	int	-4	main	...	<i>The offset will be explained shortly.</i>
rate	int	-1	main	...	
seconds	int	-5	main	...	
time	int	-2	main	...	

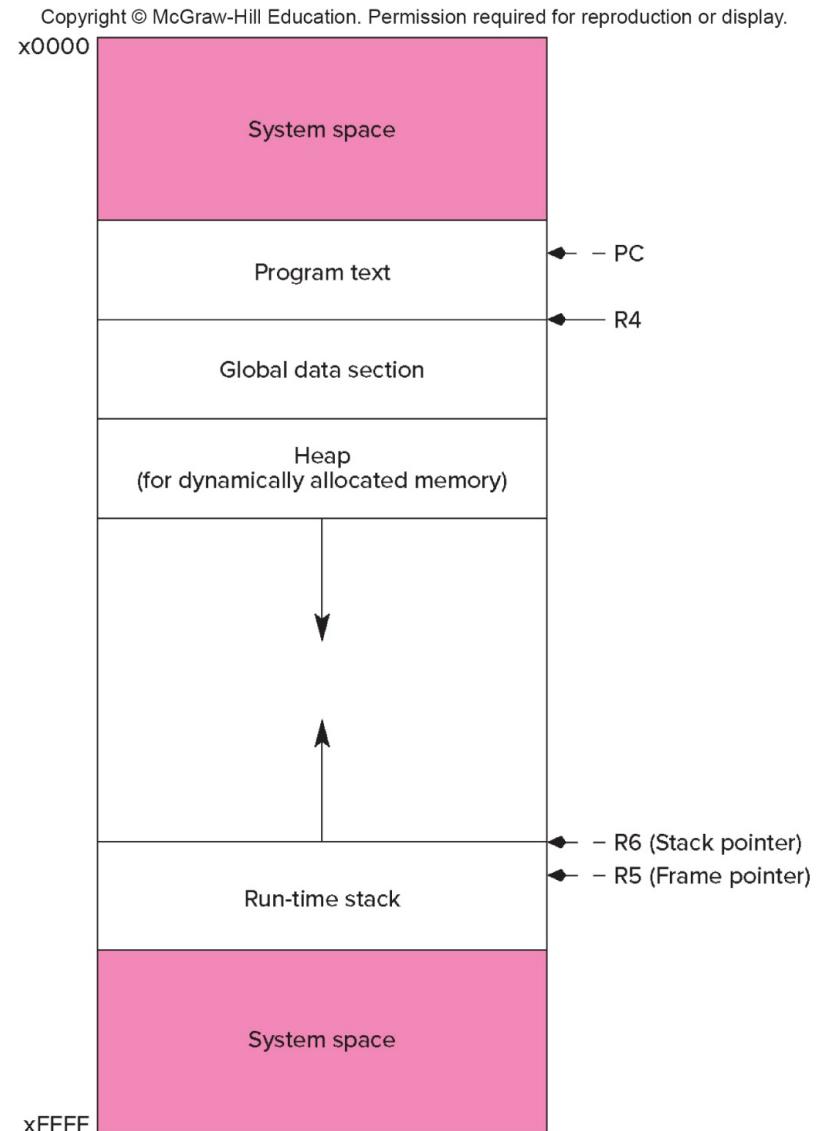
LC-3 Memory Map

The LC-3 operating system reserves some of the memory address space for trap vectors, service routine code, and memory-mapped I/O.

Program instructions will be placed in the "program text" section.

Global variables are allocated next. R4 is set to the first allocated address for globals.

Local variables are stored on the run-time stack. R5 points to the local variables of the currently-executing function.



[Access the text alternative for slide images.](#)

Stack Frame and Local Variables

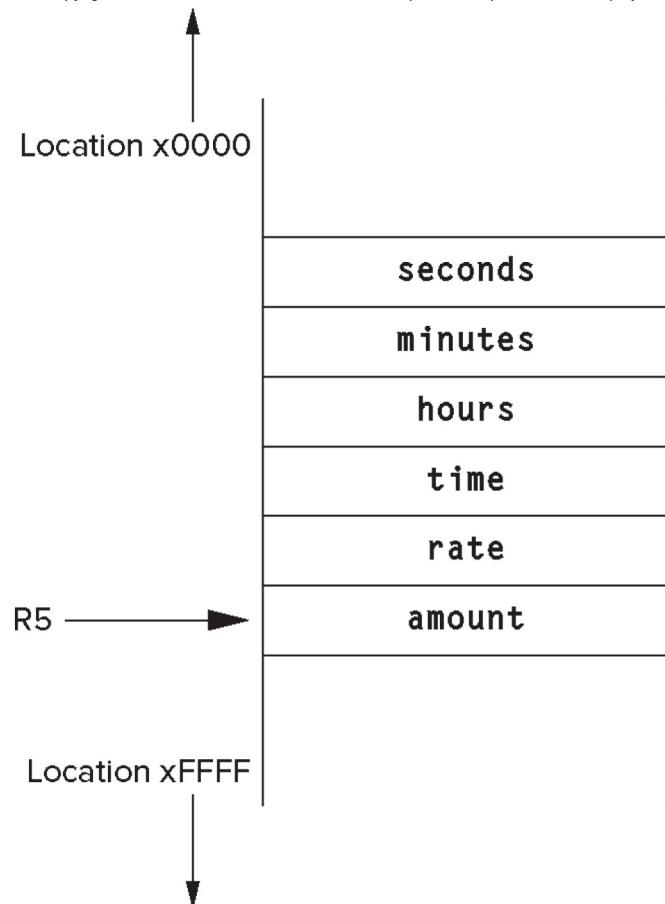
When the program is running, the run-time stack is used to keep track of state for active functions (subroutines). Each function that has been called gets a **stack frame** or **activation record** allocated on the stack.

Part of the stack frame is used to hold local variables, as shown below.

R5 points to the region where local variables are stored, and variables are allocated in the order in which they are declared.

The symbol table offset is the amount to be added to R5 to get the address of each variable. For example, the offset for `amount` is 0, and the offset for `time` is -2.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



[Access the text alternative for slide images.](#)

Generating LC-3 Instructions

Copyright © McGraw-Hill Education. Permission required for reproduction or display

The symbol table corresponds to the stack frame, as shown.

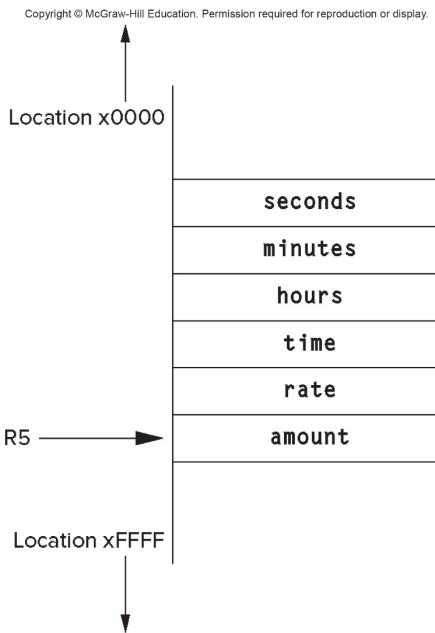
If we need to load the value of the amount variable into R0...

LDR R0, R5, #0

Suppose we've calculated the value to be stored into the hours variable, and the value is in R3...

STR R3, R5, #-3

Identifier	Type	Location (as an offset)	Scope	Other Info...
amount	int	0	main	...
hours	int	-3	main	...
minutes	Int	-4	main	...
rate	int	-1	main	...
seconds	int	-5	main	...
time	int	-2	main	...



A More Complete Example

Copyright © McGraw-Hill Education. Permission required for reproduction or display.

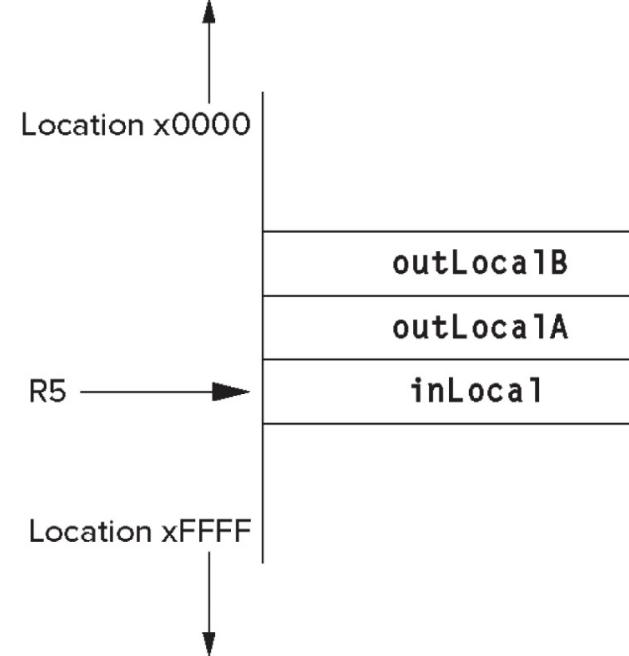
```
1 #include <stdio.h>
2
3 int inGlobal; // inGlobal is a global variable.
4                 // It is declared outside of all blocks
5
6 int main(void)
7 {
8     int inLocal = 5; // inLocal, outLocalA, outLocalB are all
9     int outLocalA; // local to main
10    int outLocalB;
11
12    // Initialize
13    inGlobal = 3;
14
15    // Perform calculations
16    outLocalA = inLocal & ~inGlobal;
17    outLocalB = (inLocal + inGlobal) - (inLocal - inGlobal);
18
19    // Print results
20    printf("outLocalA = %d, outLocalB = %d\n", outLocalA, outLocalB);
21 }
```

Symbol Table and Stack Frame

Copyright © McGraw-Hill Education. Permission required for reproduction or display.

Identifier	Type	Location (as an offset)	Scope	Other info...
inGlobal	int	0	global	...
inLocal	int	0	main	...
outLocalA	int	-1	main	...
outLocalB	int	-2	main	...

(a) Symbol table



(b) Activation record for main

The variable `inGlobal` is not in the stack frame because it is not local to the main function. Its offset is relative to R4.

[Access the text alternative for slide images.](#)

LC-3 Code for Each C Statement

```
; int inLocal = 5;  
; We don't normally generate code for declarations, but  
; but we need to initialize the variable.  
AND R0, R0, #0      ; create value 5 to put in variable  
ADD R0, R0, #5  
STR R0, R5, #0      ; inLocal is local, offset = 0  
  
; inGlobal = 3;  
AND R0, R0, #0      ; create value 3 to put in variable  
ADD R0, R0, #3  
STR R0, R4, #0      ; inGlobal is global, offset = 0  
  
; outLocalA = inLocal & ~inGlobal;  
LDR R0, R5, #0      ; get value of inLocal  
LDR R1, R4, #0      ; get value of inGlobal  
NOT R1, R1          ; bitwise NOT of R1 (~inGlobal)  
AND R2, R0, R1      ; perform AND  
STR R2, R5, #-1     ; store result to outLocalA (local, offset = -1)
```

LC-3 Code for Each C Statement ₂

```
; outLocalB = (inLocal + inGlobal) - (inLocal - inGlobal)
LDR R0, R5, #0      ; get inLocal
LDR R1, R4, #0      ; get inGlobal
ADD R0, R0, R1      ; calculate inLocal + inGlobal

LDR R2, R5, #0      ; get inLocal
LDR R3, R5, #0      ; get inGlobal
NOT R3, R3
ADD R3, R3, #1
ADD R2, R2, R3      ; calculate inLocal - inGlobal

NOT R2, R2          ; calculate -(inLocal - inGlobal)
ADD R2, R2, #1
ADD R0, R0, R2      ; calculate final value
STR R0, R5, #-2    ; store to outLocal B (local, offset = -2)
```

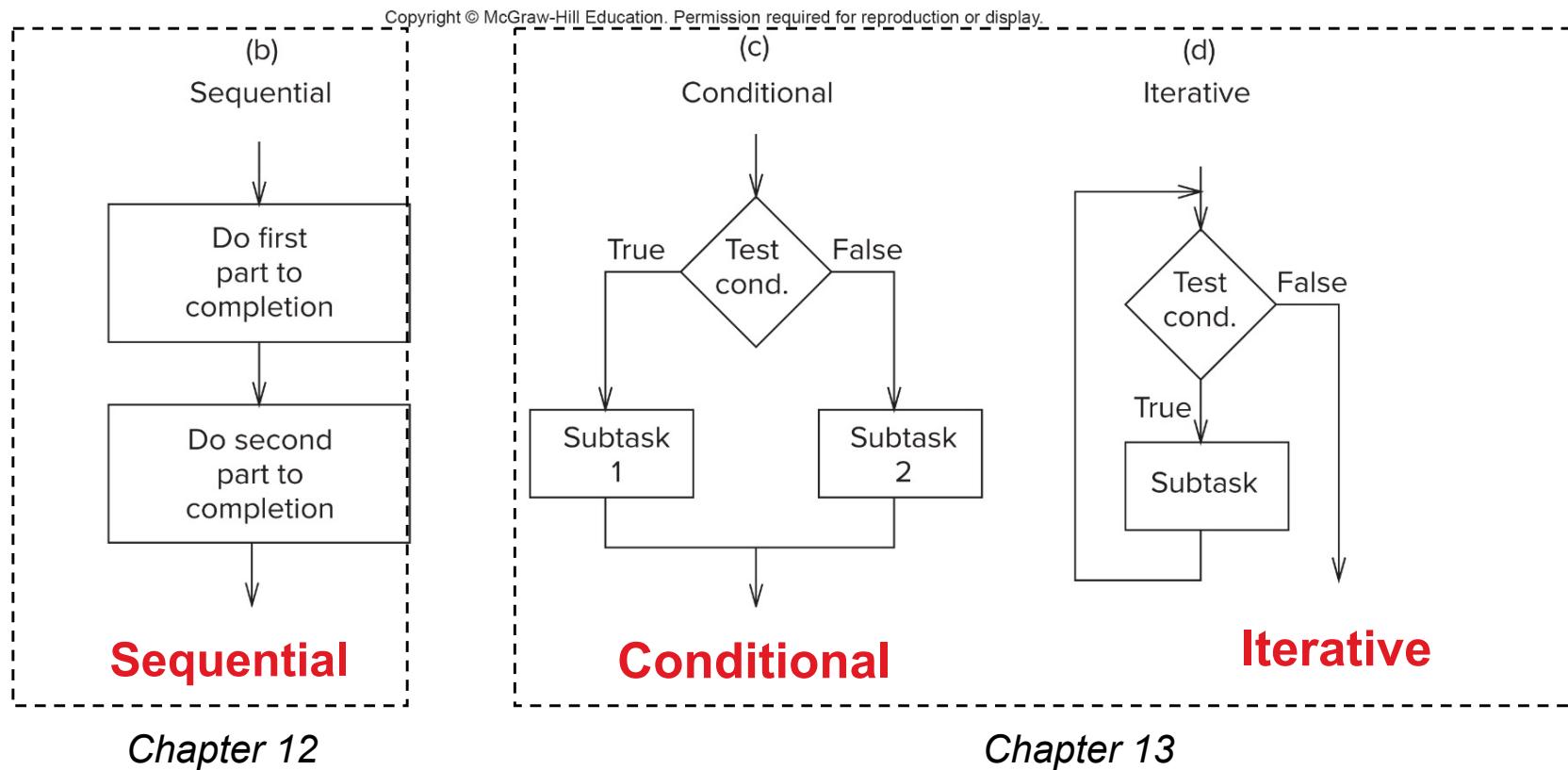
From Bits and Gates to C and Beyond

Control Structures

Chapter 13

Sequential, Conditional, Iterative

In Chapter 6, we discussed the three basic control structures. In this chapter, we will discuss the C language features for conditional and iterative code.



[Access the text alternative for slide images.](#)

if Examples

```
if (x <= 10)
    y = x * x + 5;
```

Style Guideline:

Put conditional statement on the next line and indent.
Clear that statement is conditional.

```
if (x <= 10) {
    y = x * x + 5;
    z = (2 * y) / 3;
}
```

Use of a compound statement. Both statements
are executed ONLY if condition is true.

Style: Open brace on same line as if (...). Indent
statements inside the block. Closing brace on its own
line, aligned with if.

```
if (x <= 10)
    y = x * x + 5;
    z = (2 * y) / 3;
```

Bad use of indentation. Looks like both statements
are conditional, but only the first is. z = ... should not
be indented.

if Examples

```
if (x = 2)  
    y = 5;
```

Common mistake!

Using assignment (=) instead of equal-to (==).

Condition changes value of x and is ALWAYS true!

```
if (x == 2)  
    y = 5;
```

Corrected version of previous statement.

```
if (x == 3)  
    if (y != 6) {  
        z = z + 1;  
        w = w + 2;  
    }
```

Nested if statements. Note the indentation.

LC-3 Code for if

Code uses the same pattern discussed in Chapter 6.

```
; if (x == 2) y = 5;  
; assume offset for x is 0, y is -1  
; (1) code to test condition  
LDR R0, R5, #0      ; load x  
ADD R0, R0, #-2  
BRnp NEXT          ; (2) BR if condition not true  
AND R0, R0, #0      ; (3) code for action  
ADD R0, R0, #5  
STR R0, R5, #-1    ; store to y  
NEXT  
...
```

LC-3 Code for if-else

```
if (x) {  
    y++;  
    z--;  
}  
else {  
    y--;  
    z++;  
}
```

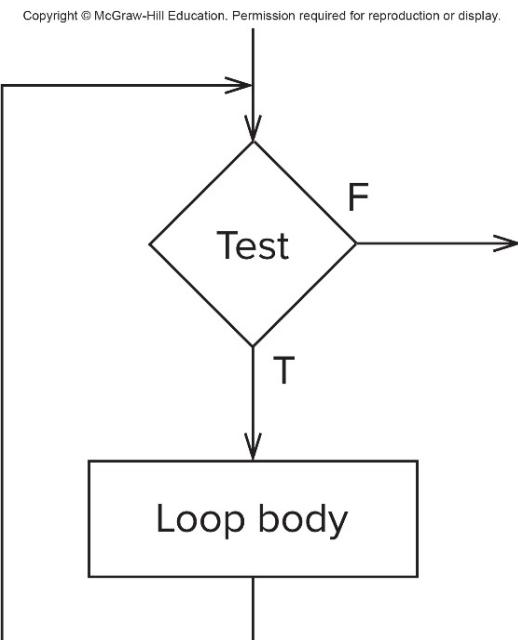
Offsets:

x = 0, y = -1, z = -2

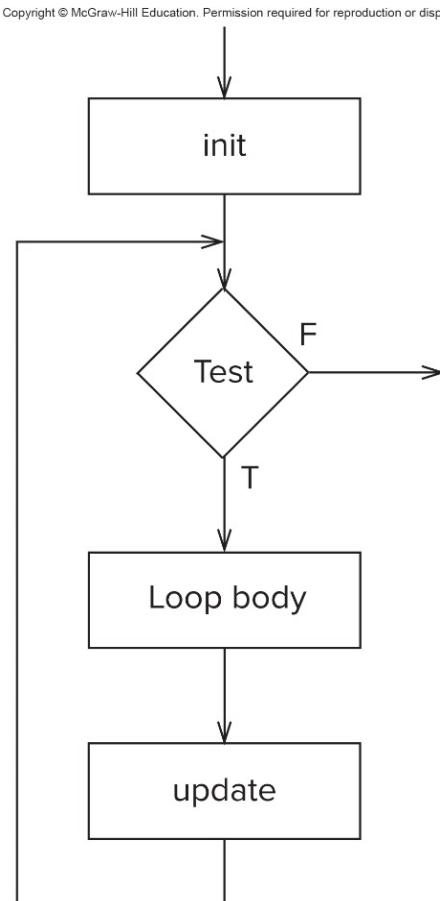
	LDR R0, R5, #0 ; condition (x)
	BRz ELSE ; jump if false
	LDR R0, R5, #-1 ; y++
	ADD R0, R0, #1
	STR R0, R5, #-1
	LDR R0, R5, #-2 ; z--
	ADD R0, R0, #-1
	STR R0, R5, #-2
	BRnzp NEXT ; skip else action
ELSE	LDR R0, R5, #-1 ; y--
	ADD R0, R0, #-1
	STR R0, R5, #-1
	LDR R0, R5, #-2 ; z++
	ADD R0, R0, #1
	STR R0, R5, #-2
NEXT	...

Iteration

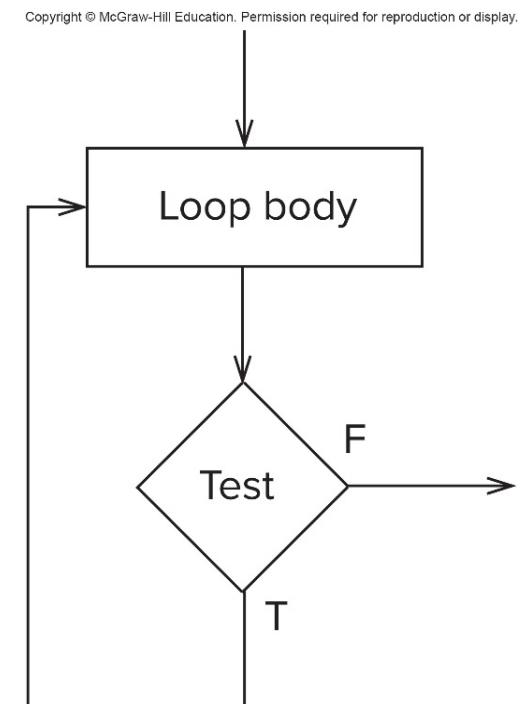
C provides three different iterative statements, representing three different ways to express a loop. Choose the one that best fits the problem.



while



for



do-while

[Access the text alternative for slide images.](#)

while Examples

```
while (x < 10) {  
    printf("%d ", x);  
    x = x + 1;  
}
```

```
char echo;  
while (echo != '\n') {  
    scanf("%c", &echo);  
    printf("%c", echo);  
}
```

```
while (x < 10)  
    printf("%d ", x);
```

Loop body can be simple or compound.
Indent the statements in the loop body, so that it's easier to tell what statements are inside the loop.

Common mistake #1: Forgetting to initialize.

Since we don't initialize echo in this code, we don't know whether it will be equal to linefeed ('\n') or not. We have no control over whether the loop will ever execute.

Solution: Make sure the loop test variable has a reasonable value before the first test.

Common mistake #2: Infinite loop.

Assuming x is less than 10, this loop will execute forever. Why?

Solution: Make sure to change loop variable in body.

LC-3 Code for while

```
int x = 0;  
while (x < 10) {  
    printf("%d ", x);  
    x = x + 1;  
}
```

```
AND R0, R0, #0      ; x = 0  
STR R0, R5, #0  
  
LOOP    LDR R0, R5, #0      ; test x < 10  
        ADD R0, R0, #-10  
        BRzp NEXT       ; exit if false  
  
        ; loop body  
        ...code to call printf...  
        LDR R0, R5, #0      ; x = x+1  
        ADD R0, R0, #1  
        STR R0, R5, #0  
        BRnzp LOOP       ; test again  
  
NEXT    ...
```

LC-3 Code for for

```
int x;  
int sum = 0;  
for (x = 0;  
     x < 10;  
     x++)  
    sum += x;
```

```
        AND R0, R0, #0      ; sum = 0  
        STR R0, R5, #-1  
        ; for statement: init expr  
        AND R0, R0, #0      ; x = 0  
        STR R0, R5, #0  
  
LOOP      LDR R0, R5, #0      ; test x < 10  
          ADD R0, R0, #-10  
          BRzp NEXT         ; exit if false  
  
          ; loop body  
          LDR R0, R5, #-1    ; sum  
          LDR R1, R5, #0      ; x  
          ADD R0, R0, R1  
          STR R0, R5, #-1    ; sum += x  
  
          ; update expression  
          LDR R0, R5, #0      ; x = x+1  
          ADD R0, R0, #1  
          STR R0, R5, #0  
          BRnzp LOOP         ; test again  
  
NEXT      ...
```

LC-3 Code for do-while

```
int x = 0;  
do {  
    printf("%d ", x);  
    x = x + 1;  
} while (x < 10);
```

```
AND R0, R0, #0      ; x = 0  
STR R0, R5, #0  
  
LOOP      ; loop body  
...code to call printf...  
LDR R0, R5, #0      ; x = x+1  
ADD R0, R0, #1  
STR R0, R5, #0  
  
LDR R0, R5, #0      ; test x < 10  
ADD R0, R0, #-10  
BRn LOOP      ; repeat if true  
  
NEXT      ...
```

break and **continue**

These two statements are used to alter the "normal" execution of a loop.

break:

Immediately **exits** the loop (or switch).

Unconditional branch to the next statement after the loop/switch.

continue:

Immediately goes to the **next iteration** of the loop, skipping the remainder of the loop body.

Unconditional branch to the "top" of the loop:

- `while` and `do-while` -- goes to the test expression.
- `for` -- goes to the update (re-init) expression.

Use these rarely for exceptional conditions. Use loop logic or conditionals in most cases: easier to understand.

Examples: break and continue

These two simplified examples illustrate the difference.

```
for (i=0; i<10; i++) {  
    if (i == 5)  
        break;  
    printf("%d ", i);  
}
```

Output:
0 1 2 3 4

```
for (i=0; i<10; i++) {  
    if (i == 5)  
        continue;  
    printf("%d ", i);  
}
```

Output:
0 1 2 3 4 6 7 8 9



Because learning changes everything.[®]

www.mheducation.com

From Bits and Gates to C and Beyond

Functions

Chapter 14

Implementing Functions

Each function needs space for its local variables and other state.
Where should it be allocated?

Option 1: Compiler allocates a memory space for each function.

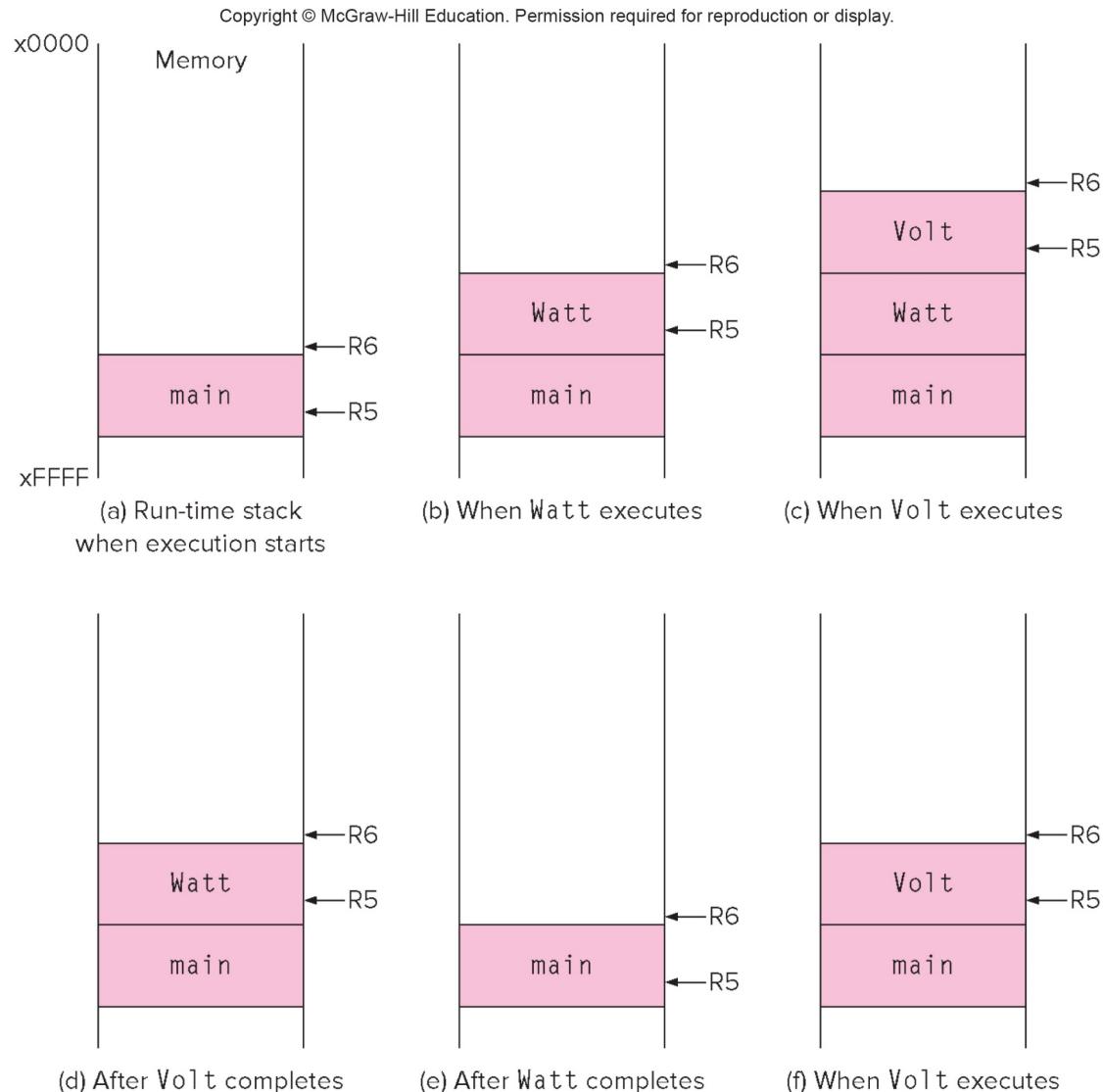
- Does not allow recursion -- local variables would be overwritten.
- Allocates memory for functions that may not be called.

Option 2: Program allocates space on a run-time stack.

- Allocates space as each function is called; space is deallocated when function returns. Only uses the space that is needed.
- Allows recursion -- each function invocation gets its own set of locals.
- Stack's LIFO behavior matches the call-return action of functions.

Push/Pop Stack Frames

```
Copyright © McGraw-Hill Education. Permission required for reproduction or display.  
1 int main(void)  
2 {  
3     int a;  
4     int b;  
5  
6     :  
7     b = Watt(a);      // main calls Watt first  
8     b = Volt(a, b);    // then calls Volt  
9 }  
10  
11 int Watt(int a)  
12 {  
13     int w;  
14  
15     :  
16     w = Volt(w, 10); // Watt calls Volt  
17  
18     return w;  
19 }  
20  
21 int Volt(int q; int r)  
22 {  
23     int k;  
24     int m;  
25  
26     :  
27     return k;  
28 }
```



[Access the text alternative for slide images.](#)

Implementation using LC-3 Instructions

Note: The approach of using a run-time stack is common to all processors, not just the LC-3. While the specific instructions will be different, the concepts are the same for different ISAs.

Three places where instructions are needed:

- Caller function -- pass arguments, transfer control to callee, use return value when control is given back.
- Callee function -- preamble to save state and allocate local variables.
- End of callee function -- return statements deallocate local variables and restores state before returning to caller.

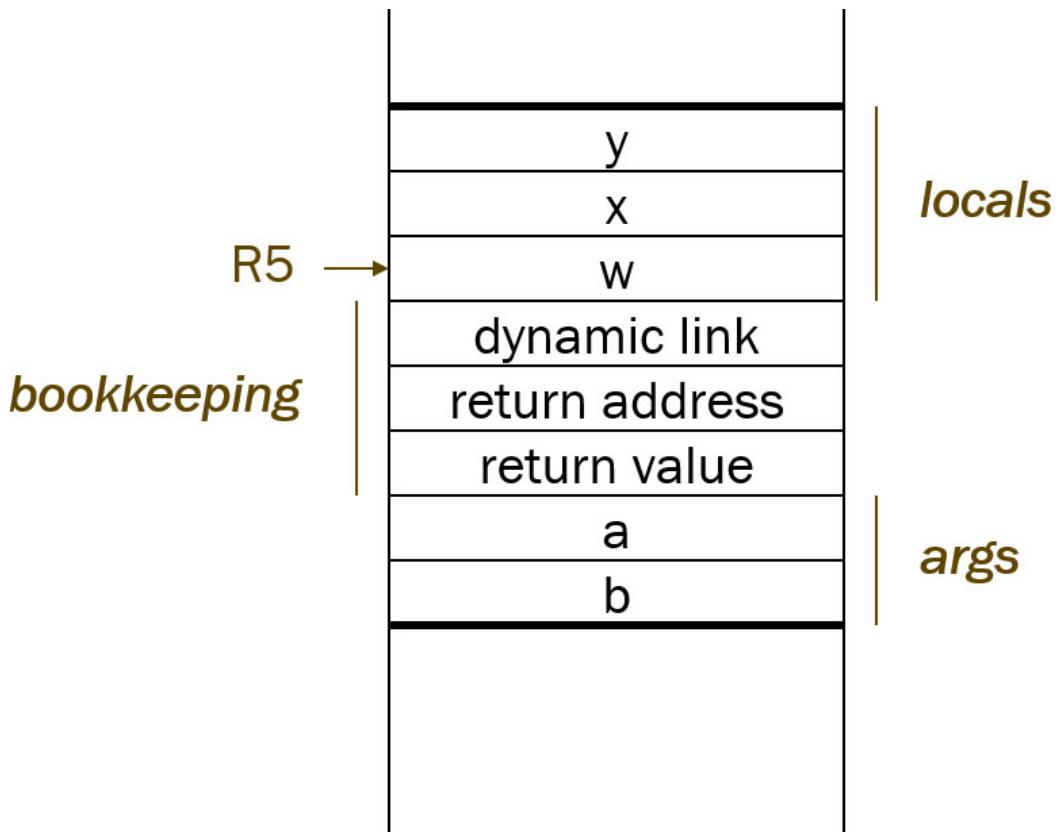
LC-3 Activation Record

The structure of a function's activation record is shown below.

As an example, the NoName function has two parameters (a and b) and three local variables (w, x, and y).

```
int NoName(int a, int b) {  
    int w, x, y;  
    ...  
}
```

Name	Type	Offset	Scope
a	int	4	NoName
b	int	5	NoName
w	int	0	NoName
x	int	-1	NoName
y	int	-2	NoName



[Access the text alternative for slide images.](#)

LC-3 Activation Record 2

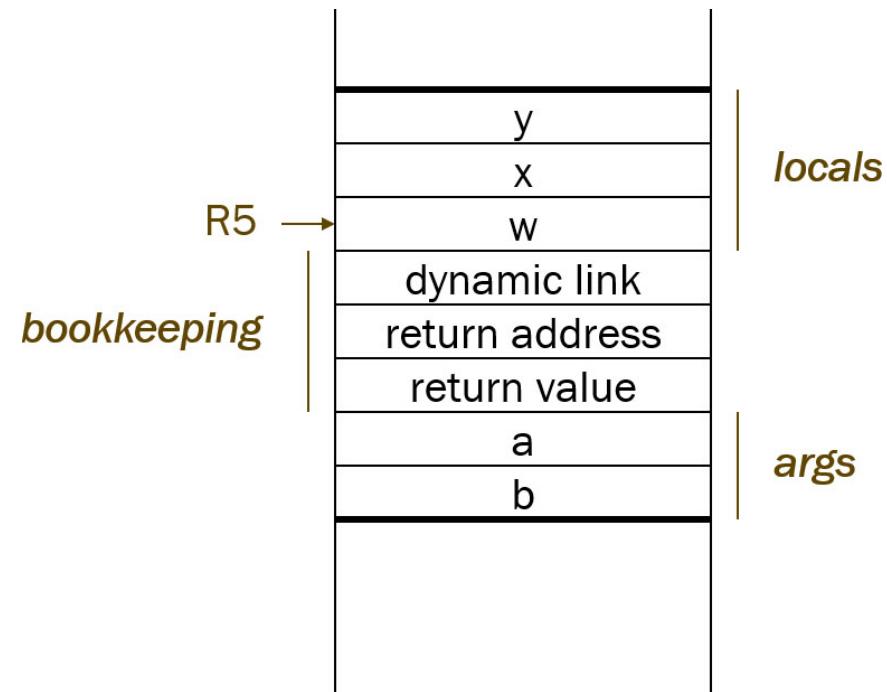
Local variables (if any) are allocated in stack order. R5 is the frame pointer, and points to the highest address of the local variable area.

The dynamic link is the saved frame pointer from the caller function. This allows us to restore the caller's state when we return.

The return address is the location of the next instruction in the caller. This is a saved version of R7, so that this function can make other function calls (JSR).

The return value will be placed above the argument values that were pushed by the caller.

The parameters (arguments) are pushed by the caller before the call. The names are known only by the callee, but the values are provided by the caller.



Caller Function

Consider this statement in the `Watt` function:

```
w = Volt(w, 10);
```

To execute the statement, we need to pass two values to the `Volt` function, call the `Volt` function, get the return value, and store it to `w`.

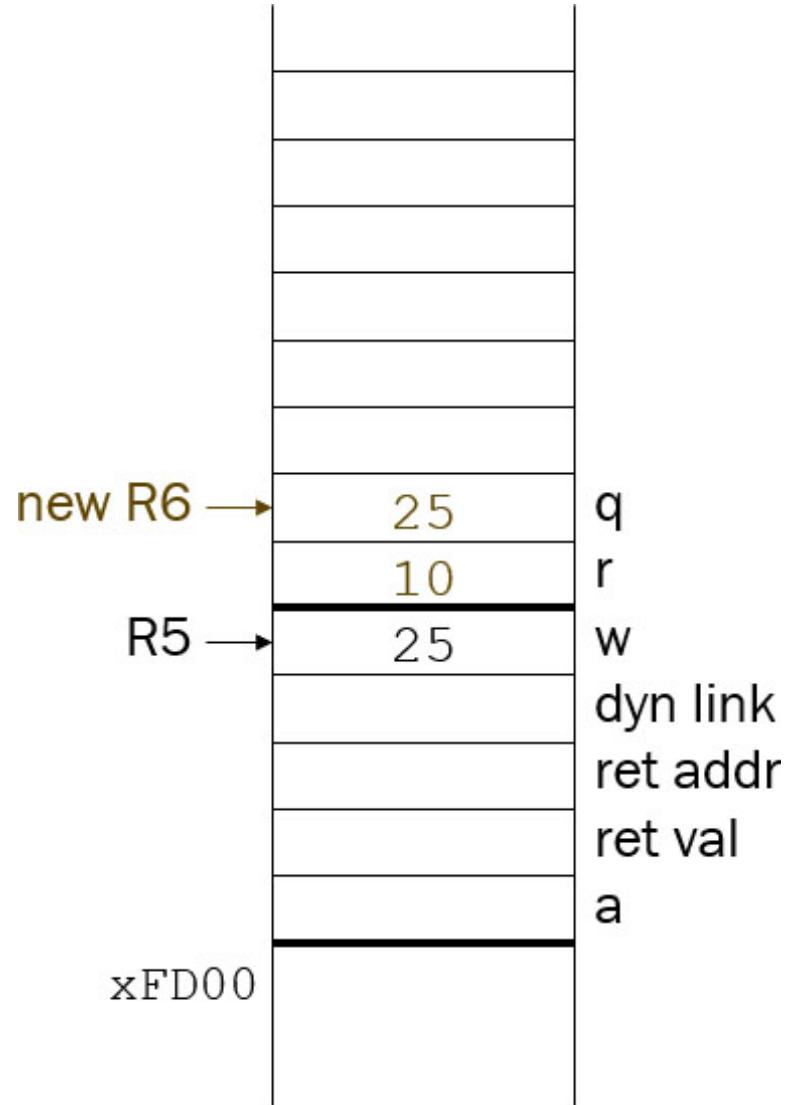
At the time of the call, the stack frame (activation record) for `Watt` is currently at the top of the run-time stack.

- R6 is the stack pointer -- it points to the top address on the stack.
- R5 is the frame pointer -- it points to the local variables of the current function (`Watt`).

Caller Function: Push the Arguments

To pass the arguments to a function, the caller pushes them onto the run-time stack. It pushes the arguments in right-to-left order.

```
; Volt(w, 10)
AND R0, R0, #0 ; push 10
ADD R0, R0, #10
ADD R6, R6, #-1
STR R0, R6, #0
LDR R0, R5, #0 ; push w
ADD R6, R6, #-1
STR R0, R6, #0
```



For this illustration, we show that w has a value of 25.

[Access the text alternative for slide images.](#)

Caller Function: Transfer Control to the Callee

Now that the values are on the stack, we use JSR to transfer control to the callee function (Volt). For our purposes we assume that the subroutine has a label that matches the function name.

JSR Volt

We use JSR instead of BR or JMP so that the return address will be saved in R7. The callee can use RET to give control back to the caller.

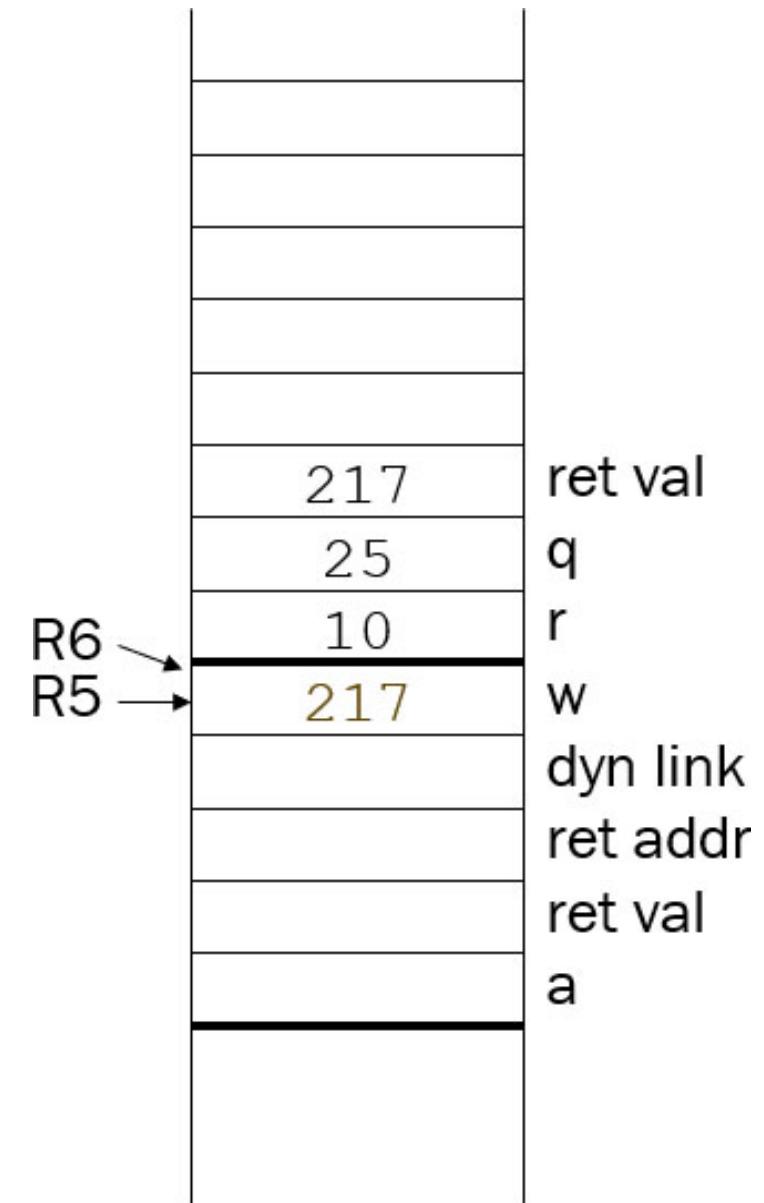
Caller Function: Using the Return Value

When the caller resumes, the return value is on the top of the stack, above the locations where the arguments were pushed.

The caller will then (a) pop the return value, and (b) pop the arguments.

Then the return value is used as directed by the caller (in this case, store to w).

```
LDR R0, R6, #0 ; pop return value  
ADD R6, R6, #1  
ADD R6, R6, #2 ; pop arguments  
; w = Volt(...);  
STR R0, R5, #0 ; store to w
```



[Access the text alternative for slide images.](#)

Caller: Complete Code

```
; w = Volt(w, 10);  
  
AND R0, R0, #0      ; push 10  
ADD R0, R0, #10  
ADD R6, R6, #-1  
STR R0, R6, #0  
LDR R0, R5, #0      ; push w  
ADD R6, R6, #-1  
STR R0, R6, #0  
  
JSR Volt           ; call function  
  
LDR R0, R6, #0      ; pop return value  
ADD R6, R6, #1  
ADD R6, R6, #2      ; pop arguments  
STR R0, R5, #0      ; store r.v. to w
```

At this point, the processor will start executing the Volt subroutine. But that code does not belong here. We will show the callee code next.

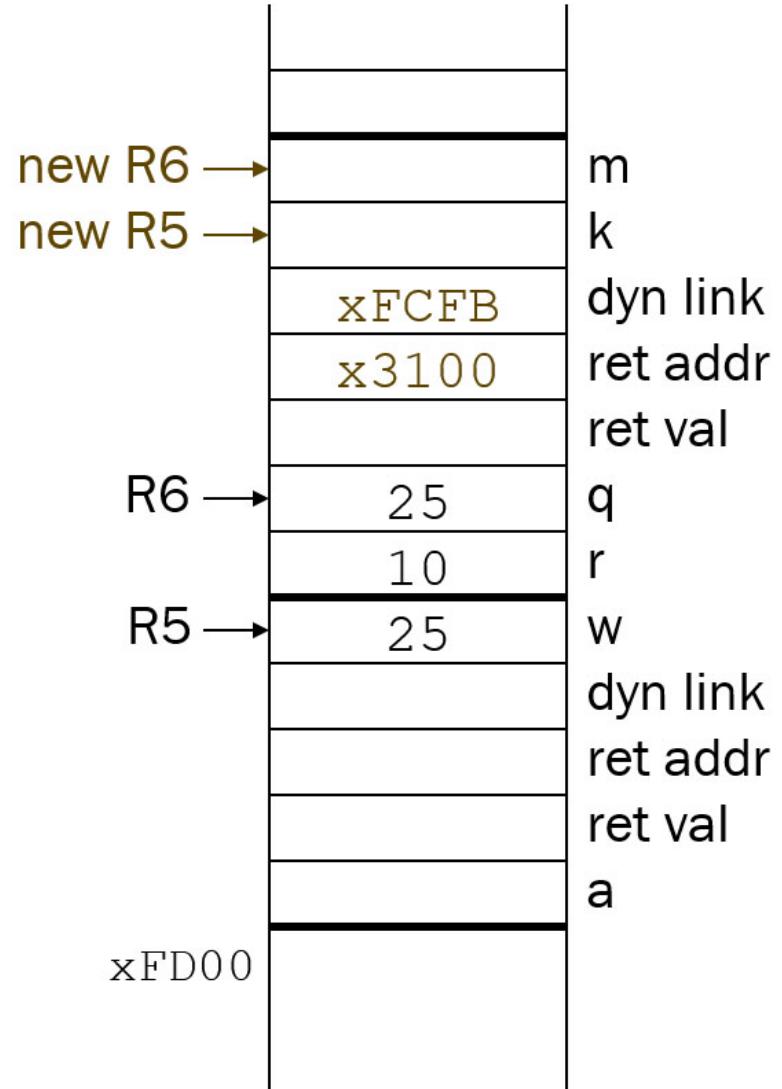
When callee returns, execution will resume here.

Callee Code: Preamble

When the callee function begins, the arguments are on the stack, and R5 points to the caller's local variables.

We need to allocate and fill in the rest of the callee function's activation record. And we need to set R5 to point to this function's local variables.

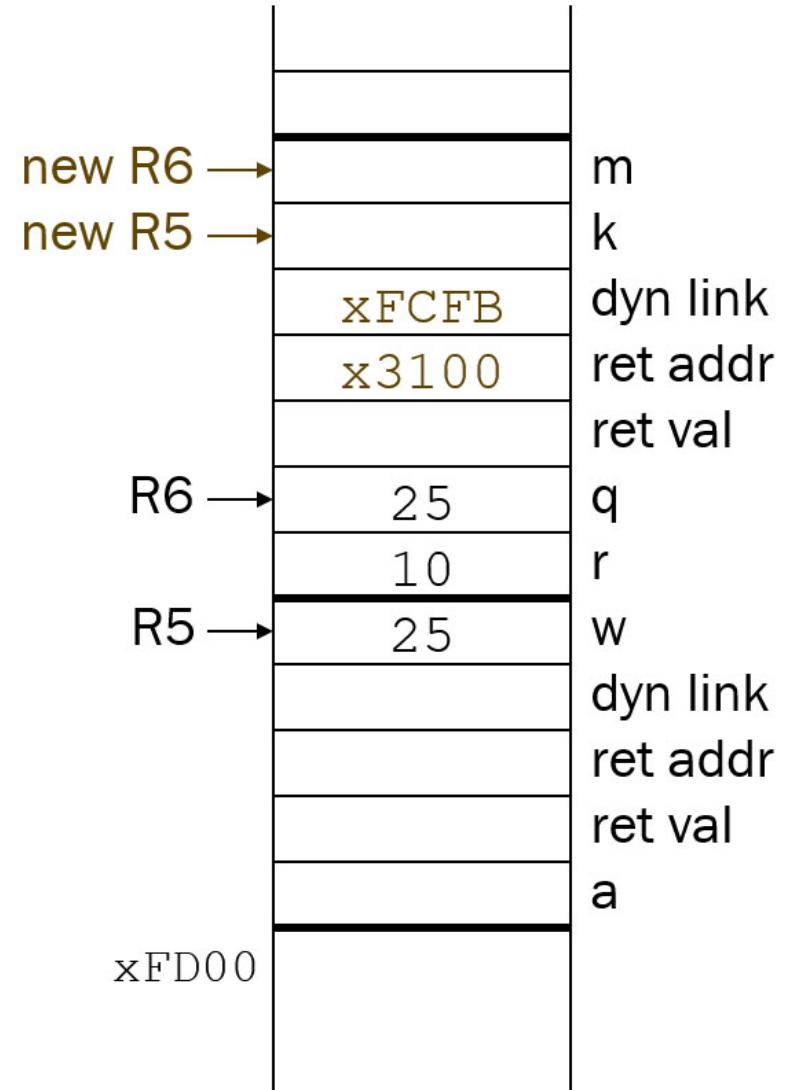
We call this the preamble code. It is executed at the beginning of every function. The results are shown by "new R5" and "new R6".



[Access the text alternative for slide images.](#)

Callee Code: Preamble 2

```
Volt ADD R6, R6, #-1 ; push r.v.  
; save return address  
ADD R6, R6, #-1 ; push R7  
STR R7, R6, #0  
; save caller's frame ptr  
ADD R6, R6, #-1 ; push R5  
STR R5, R6, #0  
; set R5 to callee locals  
ADD R5, R6, #-1  
; push space for locals  
ADD R6, R6, #-2
```



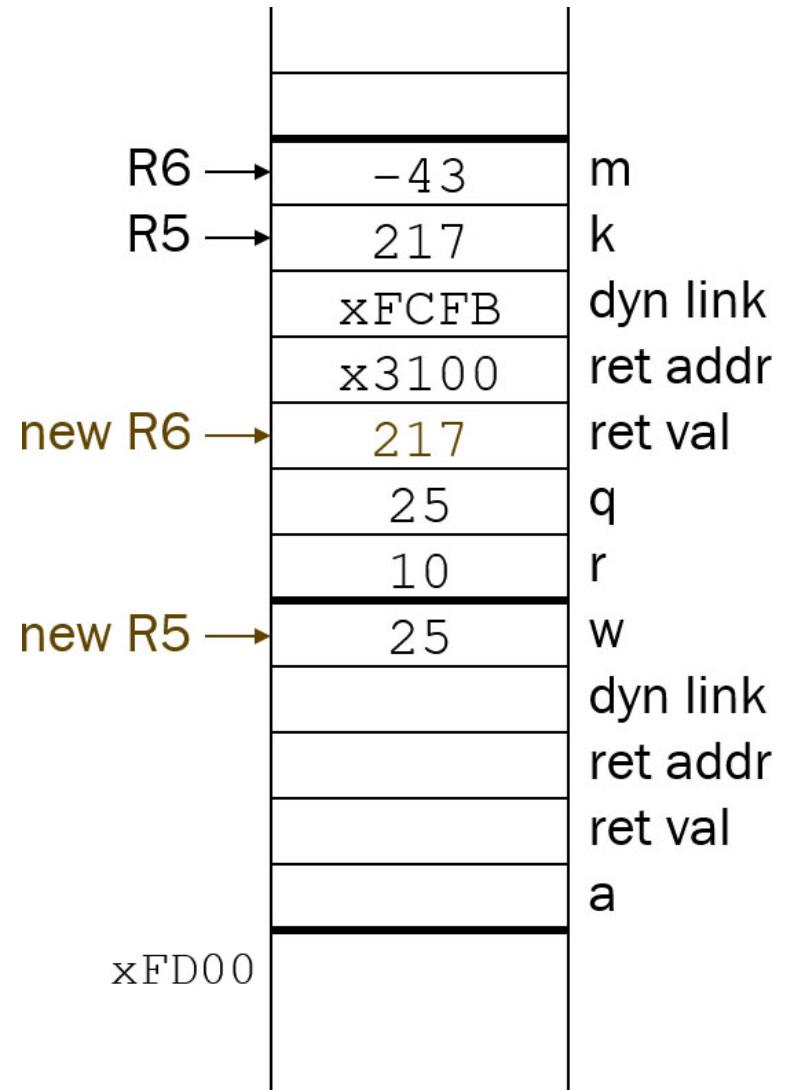
After generating code for the preamble, the compiler will start generating code for the statements in the function definition.

Callee Code: return

1

To execute the return statement, the function must:

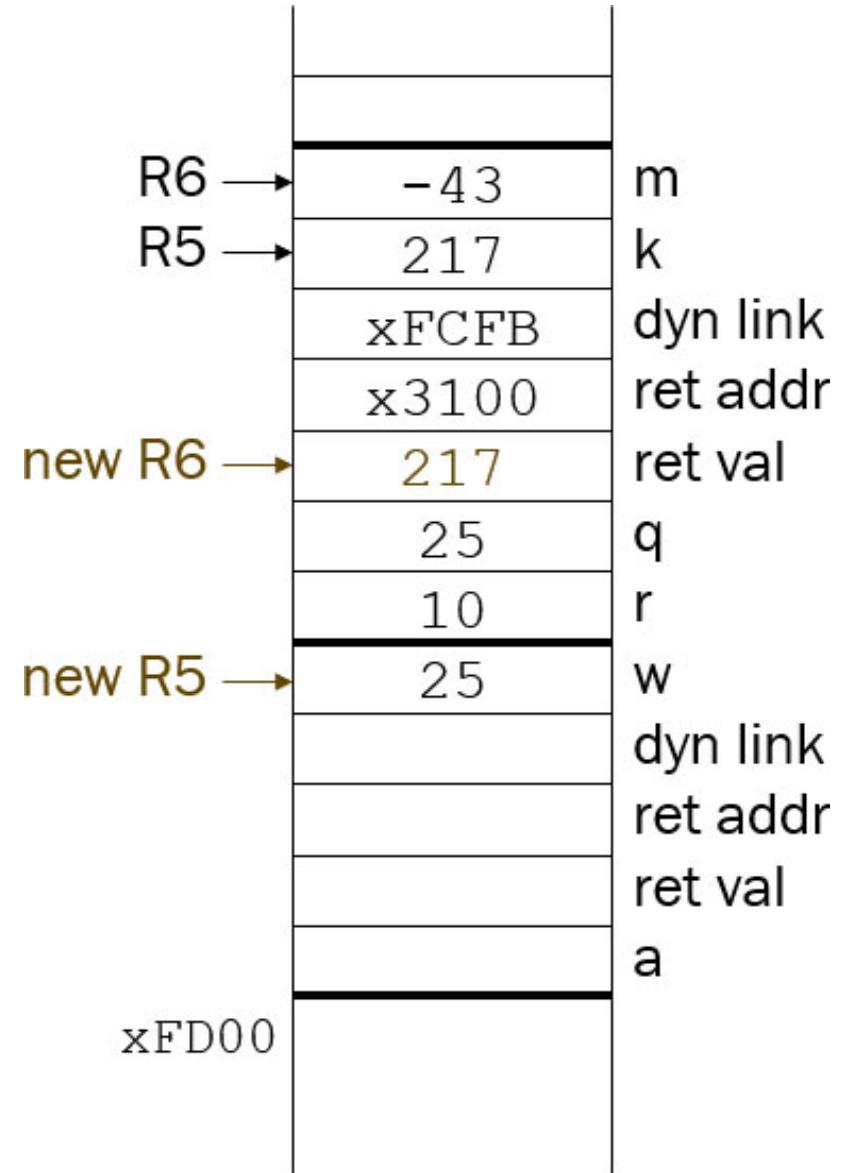
- (a) evaluate the return expression.
- (b) store the value to the return value.
- (c) pop (deallocate) local variables.
- (d) restore caller's frame pointer and return address.
- (e) return to the caller, with the return value left on the top of stack.



[Access the text alternative for slide images.](#)

Callee Code: return

```
; return k;  
  
LDR R0, R5, #0      ; k  
STR R0, R5, #3      ; -> ret value  
ADD R6, R6, #2      ; pop locals  
;  
; restore caller's frame ptr  
LDR R5, R6, #0  
ADD R6, R6, #1  
;  
; restore return address  
LDR R7, R6, #0  
ADD R6, R6, #1  
;  
; return to caller  
RET
```



Summary: Caller Code

1. Push argument values, from right to left.

```
; evaluate argument, assume it's in R0  
ADD R6, R6, #-1    ; push  
STR R0, R6, #0  
; repeat as necessary
```

2. Call function.

JSR function

3. Pop return value from stack, save in a register.

```
LDR R0, R6, #0    ; doesn't have to be R0  
ADD R6, R6, #1    ; assuming a one-word return value
```

4. Pop arguments from stack.

```
ADD R6, R6, #____ ; depends on what was pushed in part 1
```

Summary: Callee Code - Preamble

1. Push space for return value.

```
ADD R6, R6, #-1 ; assume one-word return value
```

Value depends on type of return value.
If no return value, instruction is not needed.

2. Push return address.

```
ADD R6, R6, #-1  
STR R7, R6, #0
```

3. Push dynamic link.

```
ADD R6, R6, #-1  
STR R5, R6, #0
```

4. Set new dynamic link.

```
ADD R5, R6, #-1
```

5. Push space for local variables.

```
ADD R6, R6, #-____ ; depends on local variables
```

Summary: Callee Code - Return

1. Evaluate return value and store into activation record.

```
; evaluate argument, assume it's in R0  
STR R0, R5, #3
```

2. Pop local variables.

```
ADD R6, R6, #____ ; depends on local variables
```

3. Pop/restore dynamic link.

```
LDR R5, R6, #0  
STR R6, R6, #1
```

4. Pop/restore return address.

```
LDR R7, R6, #0  
STR R6, R6, #1
```

5. Return to caller.

```
RET ; or JMP R7
```



Because learning changes everything.[®]

www.mheducation.com

From Bits and Gates to C and Beyond

Pointers and Arrays

Chapter 16

Pointers and Arrays

Pointer = address of a variable (or any memory location)

- Allows indirect access to variables – for example, allow a function to change a variable in the caller.
- Pointer-based data structures that can grow or shrink to meet needs.

Array = sequence of same-typed values, contiguous in memory

- Create a group of data values with a single name, rather than many different variables.
- Convenient representation of vectors, matrices, strings, etc.

We encountered both of these concepts when learning LC-3 programming. Now we'll see how they are expressed and used in C.

Pointer = Indirect Access

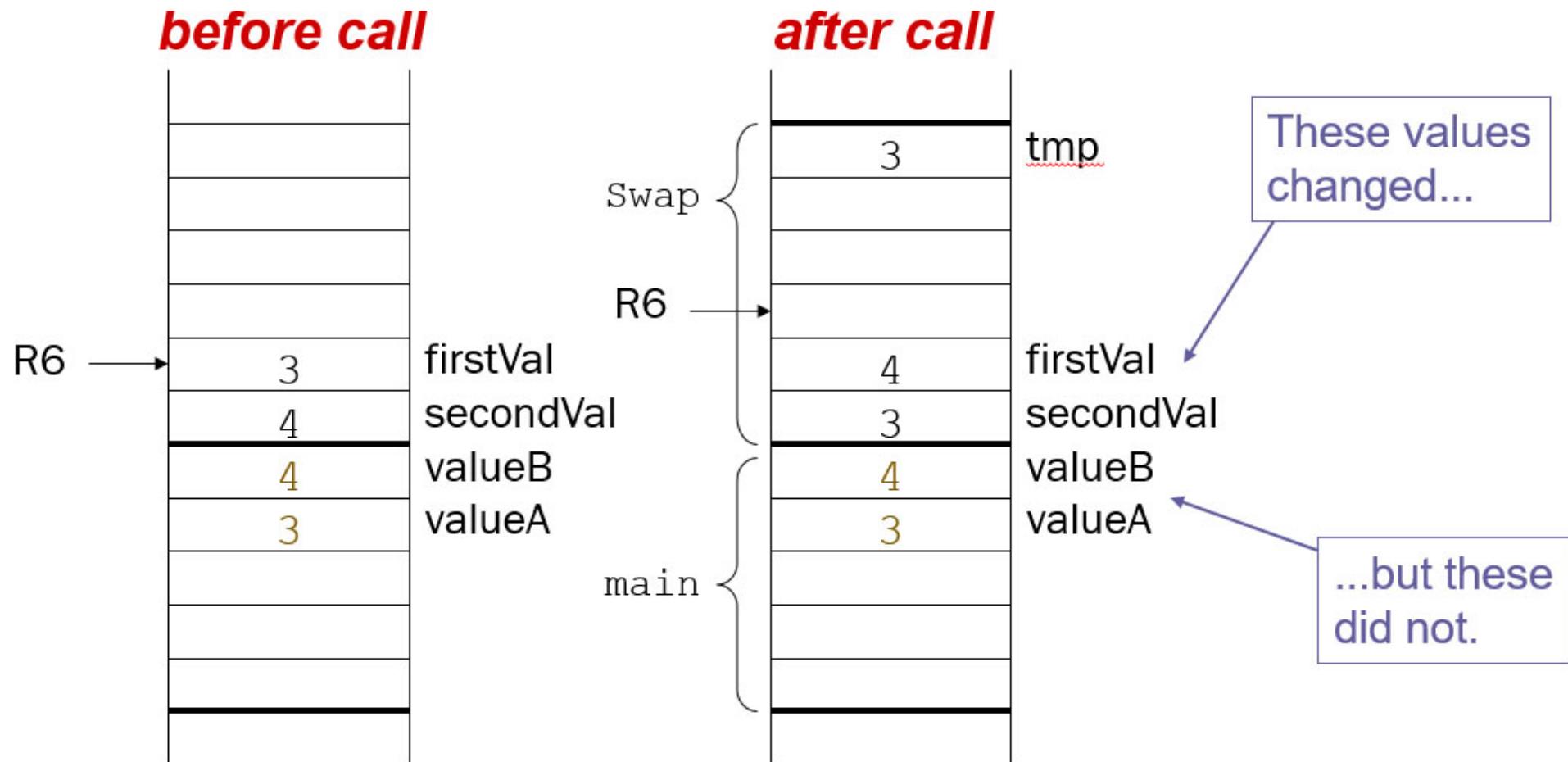
Suppose we want to write a function that swaps the values of two variables. Here is our first attempt:

```
void Swap(int firstVal, int secondVal)
{
    int tmp;
    tmp = firstVal;
    firstVal = secondVal;
    secondVal = tmp;
}
```

Does this work? Consider the following code -- what are the values of valueA and valueB when the function returns?

```
int valueA = 3;
int valueB = 4;
Swap(valueA, valueB);
```

Executing the Swap Function



To access something outside of its activation record, Swap needs an **address**.

[Access the text alternative for slide images.](#)

Pointers in C: Variables and Operators

C allows us to **declare a variable** that holds a **pointer** value.

We need to specify the type of data to which the pointer points.
Add * to the type name to declare a pointer to that type.

```
int * p; // variable p holds a pointer to an int
```

We say: the type of p is int*. We can point to any type: char*, double*, etc.

Operators

To create a pointer value, we use the (unary) & operator -- calculates the address of a variable. Suppose z is an int variable...

```
p = &z; // get the address of variable z, assign to p  
// now: p 'points to' z
```

To get the value in the location referenced by a pointer, use the (unary) * operator.

```
z = *p; // dereference the pointer p, assign value to z
```

Implementing the & Operator

```
int object;
int *ptr;

object = 4;
ptr = &object;

; offset of object = 0
; offset of ptr = -1

AND R0, R0, #0
ADD R0, R0, #4
STR R0, R5, #0 ; object = 4

ADD R0, R5, #0 ; address of object
STR R0, R5, #-1 ; ptr = &object
```

The compiler knows the address of every variable.

When we use LDR/STR, we add an offset to R5 (or R4) and then load/store.

To compute the address without load/store, just use ADD.

Implementing the * Operator

```
int object;
int *ptr;

object = 4;
ptr = &object;
*ptr = *ptr + 1;
```

```
; offset of object = 0
; offset of ptr = -1

AND R0, R0, #0
ADD R0, R0, #4
STR R0, R5, #0 ; object = 4
ADD R0, R5, #0 ; address of object
STR R0, R5, #-1 ; ptr = &object

LDR R0, R5, #-1 ; R0 has value of ptr
LDR R1, R0, #0 ; *ptr: use R0 as address to
                 ; load from where ptr points
ADD R1, R1, #1 ; *ptr + 1
STR R1, R0, #0 ; store to where ptr points
```

We load the value of a pointer variable, and then use that address for loads and stores that dereference the variable.

Passing a Reference (Pointer) to a Function

Now we can fix our Swap function. Instead of passing the values, we want to pass the addresses of the variables.

So the type of our parameters are `int*` instead of `int`.

```
void NewSwap(int *firstVal, int *secondVal)
{
    int tmp;    // still an int, because it will hold a value
    // use dereference to manipulate the non-local values
    tmp = *firstVal;
    *firstVal = *secondVal;
    *secondVal = tmp;
}
```

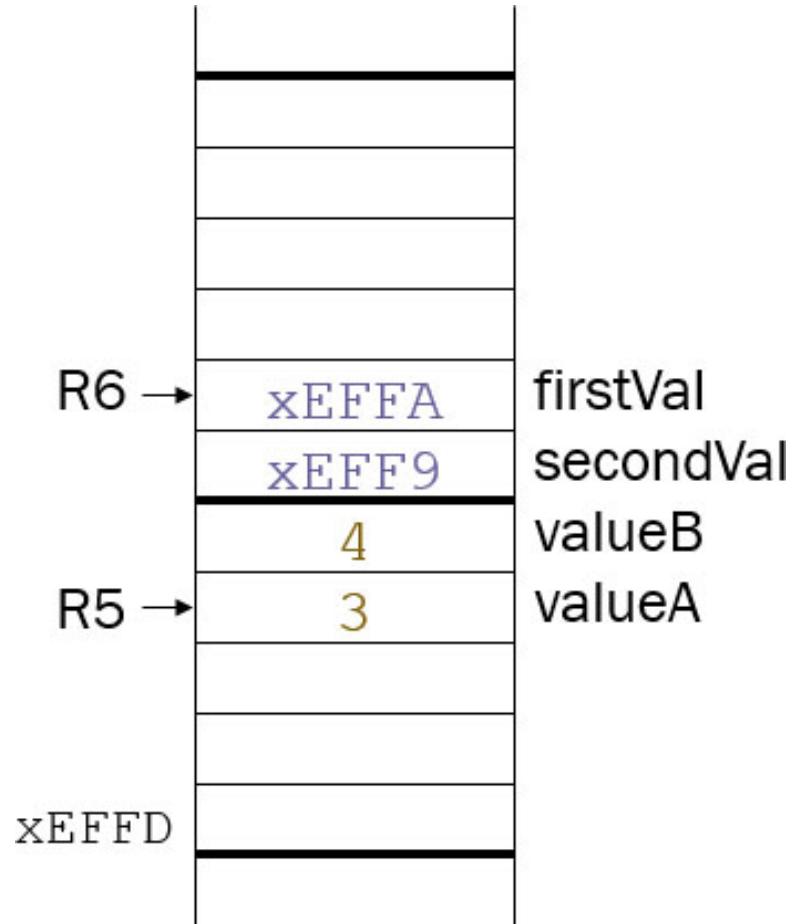
And when we call this function, we don't pass the values of `valueA` and `valueB`, we pass their addresses:

```
NewSwap(&valueA, &valueB);
```

Code that Calls NewSwap

```
NewSwap(&valueA, &valueB);
```

```
ADD R0, R5, #-1 ; &valueB  
ADD R6, R6, #-1 ; push  
STR R0, R6, #0  
ADD R0, R5, #0 ; &valueA  
ADD R6, R6, #-1 ; push  
STR R0, R6, #0  
  
JSR NewSwap  
  
ADD R6, R6, #3 ; pop r.v. (none)  
; and arguments
```



[Access the text alternative for slide images.](#)

Inside the NewSwap Function

```
; tmp = *firstVal;
```

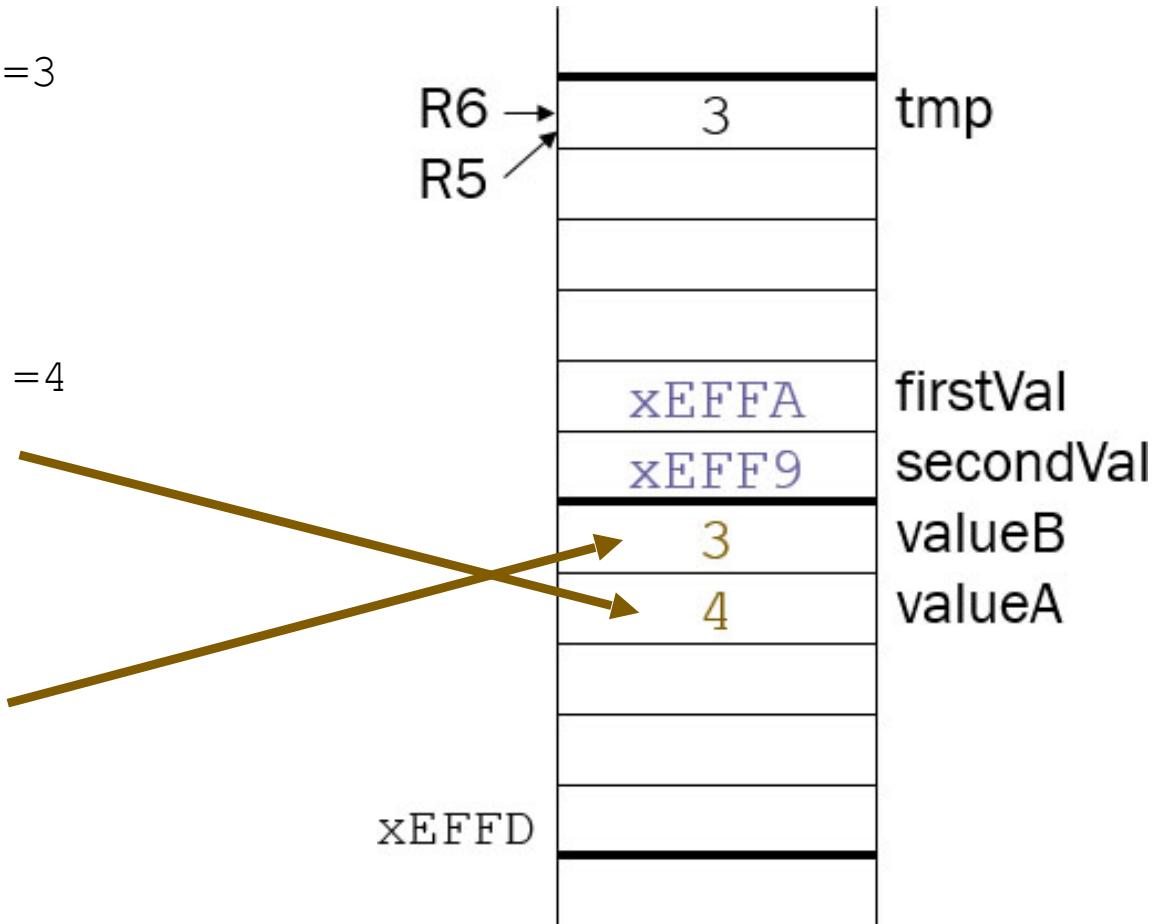
```
LDR R0, R5, #4 ; R0=xEFFA  
LDR R1, R0, #0 ; R1=M[xEFFA]=3  
STR R1, R5, #0 ; tmp=3
```

```
; *firstVal = *secondVal;
```

```
LDR R1, R5, #5 ; R1=xEFF9  
LDR R2, R1, #0 ; R2=M[xEFF9]=4  
STR R2, R0, #0 ; M[xEFFA]=4
```

```
; *secondVal = tmp;
```

```
LDR R2, R5, #0 ; R2=3  
STR R2, R1, #0 ; M[xEFF9]=3
```



[Access the text alternative for slide images.](#)

NULL Pointer

Sometimes we want to create a pointer that points to nothing.
Perhaps we don't know yet where it should point...

This is known as a **null pointer**.

By convention, we use the value 0 to represent the null pointer.
C provides the symbolic name **NULL** -- using NULL makes the code more
readable and makes it clear that we intend the pointer to be null.

```
int *ptr = NULL; // declare ptr, initialize it to NULL
```

Best practice: Initialize unused pointers to NULL. If you don't initialize, the
pointer will point to a random memory location, which can lead to strange and
interesting bugs.

If your code tries to dereference a null pointer, the program will crash -- that's a
much better outcome than reading/writing from an unknown memory location.

scanf and Pointers

We've actually been using pointers in our C programs all along:

```
scanf("%d", &inValue);
```

Now that you understand what `&` means, you can see that we are passing the address of the `inValue` variable to the `scanf` function. That's because we want `scanf` to modify that variable when it reads the input value -- the only way this can happen is if we tell `scanf` the location of the variable.

We can use this feature to create functions that calculate and return multiple values. A function can only have one return value, but we can use as many pointers as we want, giving locations to store additional values.

For an example, see the next slide...

Example: Using Pointers to Return Values 1

Write a function that divides two integers and returns both the quotient and the remainder. Since we can only have one return value, we'll use pointers to tell the function where to put the result.

Use the return value as an error code. Return -1 if the divisor is zero, and 0 otherwise.

```
int IntDivide(int x, int y, int *quoPtr, int *remPtr)
{
    if (y != 0) {
        *quoPtr = x / y;
        *remPtr = x % y;
        return 0;
    }
    else {
        return -1;
    }
}
```

Example: Using Pointers to Return Values 2

It's a good idea to return some sort of status / error code for a function like this. The caller needs to know whether anything useful was stored in the specified location.

```
int dividend, divisor;
int quotient, remainder;
int error;

// ... code to get dividend and divisor from user ...

error = IntDivide(dividend, divisor, &quotient, &remainder);
if (!error)
    printf("Quotient: %d, Remainder: %d\n", quotient, remainder);
else
    printf("IntDivide failed.\n");
```

Arrays

An array is a convenient way to create a collection of values, all of the same type, identified by a single name.

Consider creating a gradebook program for a class with 100 students. You need to store 100 different grades -- do you want 100 different variables?

Instead, we declare an array of doubles to hold a grade for each student.

```
double grades[100];
```

type

variable name

size -- number of elements

Array Indexing

Each element of the array is assigned an **index**, which indicates its place in the array.

A N-element array has indices from 0 to (N – 1).

To access an array element:

grades[0] ← *first element = element 0*

grades[99] ← *last element = element N-1*

grades[i] ← *index can be any integer expression*

grades[i+1]

It is convenient to use a loop to process elements of an array. For example, here we compute the average grade:

```
double sum = 0.0;  
for (int i = 0; i < 100; i++) sum += grade[i];  
sum = sum / 100.0;
```

Implementing Arrays

When we declare an array with a fixed size, the compiler allocates the appropriate amount of space on the stack (or in the global area).

Space = (# elements) x (size of element)

The elements are always contiguous in memory, and element 0 is always at the lowest allocated address.

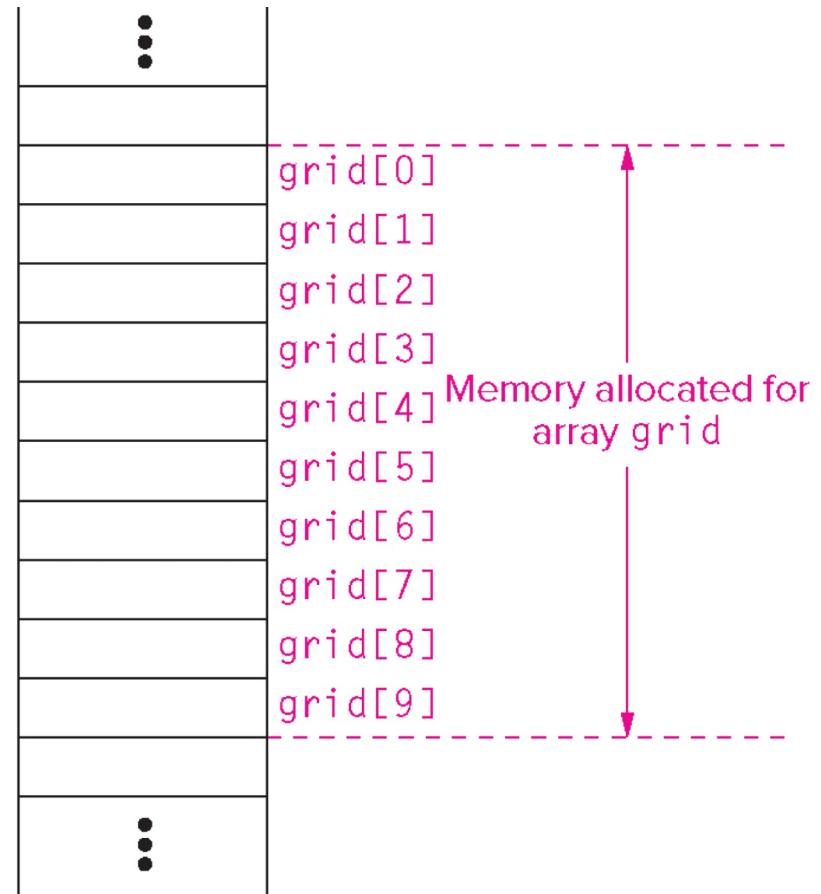
The figure to the right illustrates the memory allocation for this array:

```
int grid[10];
```

The symbol table stores the offset of the first element of the array.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.

Memory



[Access the text alternative for slide images.](#)

Accessing an Array Element

When an expression contains an array element, the compiler needs to load/store from/to the appropriate memory address. The compiler must do the following:

1. Get the starting address of the array.
2. Evaluate the index expression.
3. Multiply the index (step 2) by the size of array element -- the type of the array elements is part of the declaration, and is stored in the symbol table.
(For our LC-3 examples, we will use integer arrays, so the "multiply by one" step can be ignored.)
4. Add the scaled index (step 3) to the starting address (step 1).
5. Load or store using the address computed in step 5.

Implementation: Constant index

```
int grid[10];  
// ...  
grid[6] = grid[3] + 1;
```

```
; first element of grid has offset -9  
;  
; step 1 - get base address  
ADD R0, R5, #-9  
;  
; step 2, 3, 4, 5 -  
; add index to address and load  
LDR R1, R0, #3 ; R1 has grid[3]  
;  
ADD R1, R1, #1  
;  
; step 1 - R0 still has base address  
;  
; step 2, 3, 4, 5 -  
; add index to base address and store  
STR R1, R0, #6
```

When index is constant, compiler can calculate the address.
Can use offset of LDR/STR to combine add with load/store.

Implementation: Index expression

```
int grid[10];
int x;
// ...
grid[x+1] = grid[x] + 2;
```

```
; first element of grid has offset -9
; step 1 - get base address
ADD R0, R5, #-9

; step 2,3 - evaluate index (times 1)
LDR R1, R5, #-10 ; x is index
ADD R2, R0, R1 ; step 4 - add to address
LDR R3, R2, #0 ; step 5 - load

ADD R3, R3, #2 ; grid[x]+2

; step 2,3 - evaluate index (times 1)
LDR R1, R5, #-10
ADD R1, R1, #1 ; x + 1 is index
ADD R2, R0, R1 ; step 4 - add to address
STR R3, R2, #0 ; step 5 - store
```

When index is an expression, must generate code to **evaluate the expression** and **compute the address** at runtime.

Example: Counting Repeated Values 1

Write a program to

- a. read a sequence of integers from the keyboard,
- b. count the number of times each number is repeated,
- c. print each number and the number of times it was entered.

Use a symbolic value MAX_NUMS as the number of items.

Declare two arrays:

- One to hold the numbers that are entered.
- One to hold the number of times each input value occurs.

Example: Counting Repeated Values 2

```
#include <stdio.h>
#define MAX_NUMS 10

int main(void)
{
    int repIndex;          // loop index
    int numbers[MAX_NUM]; // array of input values
    int repeats[MAX_NUM]; // occurrences of each value

    // get numbers from user
    printf("Enter %d numbers.\n", MAX_NUMS);
    for (int index = 0; index < MAX_NUMS; index++) {
        printf("Input number %d: ", index);
        scanf("%d", &numbers[index]); // note: address of element
    }

    // next slide...
}
```

Example: Counting Repeated Values 3

```
// scan through entire array, counting number of
// repeats for each element in original array
for (int index = 0; index < MAX_NUMS; index++) {
    repeats[index] = 0;
    for (repIndex = 0; repIndex < MAX_NUMS; repIndex++) {
        if (numbers[repIndex] == numbers[index]) {
            repeats[index]++;
            // note: can increment/decrement an array element
        }
    }
}

// print results
for (int index = 0; index < MAX_NUMS; index++) {
    printf("Original number %d. Number of repeats %d.\n",
        numbers[index], repeats[index]);
}
```

Array as Parameters

In C, arrays are always passed by reference.

So far, when we call a function, we push a value on the stack for each parameter. When the expression is a variable name, this is essentially pushing a **copy** of the variable to the callee function.

Any change to the parameter variable is local to the callee function and does not affect the original variable (as we saw with the Swap function).

We do not want to create a copy of an array.

Could be thousands of elements! The callee function might just need to access a small part of the array, or just read the array, so it's a waste of effort to copy all of those elements.

Instead, we pass a reference (pointer!) to the array.

Function: Calculate the Average of an Array

```
// declaring the function  
int Average(int data[]); // [] tells compiler that the parameter  
                         // is an array  
// for this problem, assume MAX_NUMS is the size of the array  
  
// function implementation  
int Average(int inputValues[]) {  
    int sum = 0;  
  
    for (index = 0; index < MAX_NUMS; index++) {  
        sum += inputValues[index]; // just access array as usual  
    }  
    return sum / MAX_NUMS; // note: integer division  
}
```

Calling the Function

```
#define MAX_NUMS 10
int Average(int n[]);

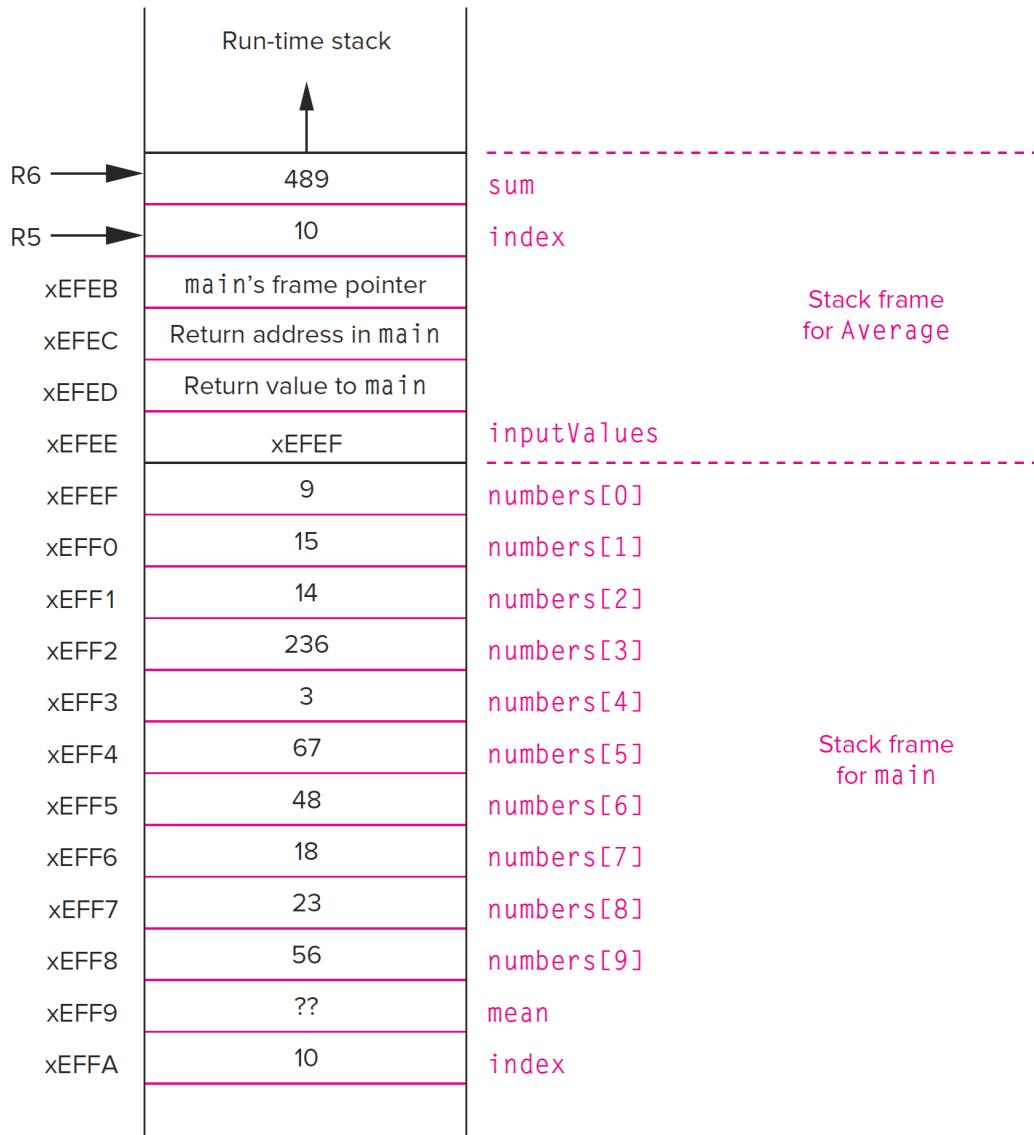
int main(void) {
    int mean;      // will hold average
    int numbers[MAX_NUMS];    // data values

    printf("Enter %d numbers.\n", MAX_NUMS);
    for (int index = 0; index < MAX_NUMS; index++) {
        printf("Input number %d: ", index);
        scanf("%d", &numbers[index]);
    }

    mean = Average(numbers);
    // name of array represents a pointer to the first element

    printf("The average is %d.\n", mean);
}
```

View of Run-Time Stack



[Access the text alternative for slide images.](#)

Notes on Passing Array as Parameter

Array is always passed by reference. This is part of the language specification. There's no way to override this behavior.

Callee function cannot determine the size of the array.
In the callee's activation record, there is only a pointer.

Three options:

- Size is implicit (for example, five cards in a poker hand) or represented by a symbolic value (example, MAX_NUMS).
- Caller passes the size as an additional parameter -- most common.
`int Average(int data[], int size);`
- Array includes a sentinel value that is used to designate the end of the array. This is common with strings (next...).

String = Array of Characters

There is no native string type in C. Instead, we use an array of characters to hold the string data.

```
char word[10];
```

The array must be large enough to hold the character data, including the null character that terminates the string.

Therefore, the `word` array is large enough to hold a 9-character string.

Take care to distinguish the array that holds the data and the string itself (which is the content of the array).

Initializing a String

First method -- store data to the string array a character at a time.

```
char word[10];      // we want string to be "giraffe"  
  
word[0] = 'g';  
word[1] = 'i';  
word[2] = 'r';  
word[3] = 'a';  
word[4] = 'f';  
word[5] = 'f';  
word[6] = 'e';  
word[7] = '\0';    // null terminator
```

Second method -- initialize using a literal string.

```
char word[10] = "giraffe";
```

This only works to **initialize** a string array. **Cannot assign a literal string to an array**, because there is no assignment operator in C that works for arrays.

Basic I/O with Strings

The %s format code is used for strings for both input and output.

```
printf ("%s", word);
```

The `printf` function matches a char array with `%s`, will print character in the array until null '`\0`' is reached.

```
scanf ("%s", word);
```

The `scanf` function will read characters until the next white space character, storing them into the array. It will put a terminating null character at the end.

Note: No ampersand! The variable `word` is already a pointer -- it represents the address of the first location in the array.

String Example: Calculate the Length of a String

```
int StringLength(char string[]) {  
    int index = 0;  
  
    while (string[index] != '\0') {  
        index = index + 1;  
    }  
    return index;  
}
```

The length of a string does not include the null terminator.

String Example: Reversing a String

```
int StringLength(char str[]); // previous slide
void CharSwap(char *firstChar, char *secondChar); // NewSwap for
char
void Reverse(char str[]); // reverse string in place

void Reverse(char string[]) {
    int length;
    length = StringLength(string);
    for (int index = 0; index < length/2; index++)
        CharSwap(&string[index], &string[length-(index+1)]);
}
```

Relationship between Arrays and Pointers

We've noted that an array variable is actually a pointer -- it represents the address of the first element of the array.

It's not actually a pointer variable, because there is no separate memory location that holds the address; it is calculated by the compiler whenever needed.

But can use its value as a pointer, including assigning it to a pointer variable.

```
char word[10];    // array of characters
char *cptr;       // variable that holds a pointer to a char
cptr = word;      // cptr now holds address of first element of word
```

The difference is that I can now change the value stored in `cptr`, because it stores the address in a memory location.

```
cptr++;    // now point to the next array element
```

I can't do the same thing with `word`.

Pointer Notation versus Array Notation

We can always rewrite an array expression as a pointer expression.

We can also rewrite a pointer expression as an array expression, but that only makes sense if the pointer actually points to (or into) an array.

Table 16.1 Expressions with Similar Meanings

	Using a Pointer	Using Name of Array	Using Array Notation
Address of array	cptr	word	&word[0]
0th element	*cptr	*word	word[0]
Address of element n	(cptr + n)	(word + n)	&word[n]
Element n	*(cptr + n)	*(word + n)	word[n]

Advice: Use whichever notation is most "natural" to the problem you're solving. Pointer notation can be more compact, but it might be harder for a reader of the code to understand.

[Access the text alternative for slide images.](#)

Pointer Arithmetic

In the previous slide, we show that `cptr+n` is the address of element n of the array: `&word[n]`.

But what if the array holds something other than integers -- something that has a size other than 1? Do we have to do something like `cptr + (2*n)` if each element is two memory locations?

No, we don't.

Pointer arithmetic is not the same as integer arithmetic. Since we know what type of data is being referenced, the compiler will do the appropriate multiplication by the size of the data type.

This means that `cptr+n` works for an array of integers, an array of doubles, or an array of anything.

It also makes the code portable, where an `int` may occupy a different number of memory locations for different architectures. It insulates the programmer from having to know the exact memory sizes for different platforms.

Example of Pointer Notation

One common use of pointer notation is for functions that manipulate strings. Here is the `StringLength` function rewritten using pointers.

Note that the incoming parameter is explicitly typed as a pointer variable. This is not a contradiction with earlier statements -- the parameter is a variable and it contains a pointer, so it can be treated as such.

```
int StringLength(char *string) {
    int length = 0;
    while (*string != '\0') {
        length++;
        string++; // move to the next character in string
    }
    return length;
}
```

We can also move the incrementing of the pointer inside the test condition and remove the extraneous comparison with zero...

```
while (*string++) length++;
```

Example: Insertion Sort

We show an example that sorts the items (integers) in an array using the **insertion sort** algorithm.

The goal is to sort the integers in ascending order. We segregate the array into a sorted part (at the beginning) and an unsorted part (at the end). When we start, the sorted part is only element 0 and elements 1 through N-1 are unsorted; we repeatedly pick the first integer from the unsorted part and insert it into its proper position in the sorted part -- this increases the sorted part by one each iteration.

Given an array A with elements 0 through N-1,

```
For index from 1 to N-1
    item = A[index]
    For pos from index-1 to 0
        If A[pos] > item
            Shift A[pos] to A[pos+1]
        Else
            A[pos] = item, exit inner loop
        End If
    End For
End For
```

Insertion Sort Function

```
void InsertionSort(int list[]) {  
    int unsorted;      // index marking beginning of unsorted part  
    int sorted;        // index for sorted part  
    int unsortedItem;  // temp value for moving value  
  
    for (unsorted = 1; unsorted < MAX_NUMS; unsorted++) {  
        unsortedItem = list[unsorted]; // item to be inserted  
        // working backwards, shift sorted values to the right  
        // until we make reach the position of the new item  
  
        for (sorted = unsorted - 1;  
             (sorted >= 0) && (list[sorted] > unsortedItem);  
             sorted--) {  
            list[sorted+1] = list[sorted]; // shift to right  
        }  
        // after loop, sorted+1 is position where item should go  
        list[sorted+1] = unsortedItem;  
    }  
}
```

Warning: No Bounds Checking on Array Access

C does not check to make sure that the index expression is within the range of elements allocated by the array.

On Slide 18, there is no step between 3 and 4 that says "make sure the index is non-negative and less than the array size."

If your code goes beyond the array (in either direction), the code generated by the compiler will simply read/write that memory location.

This is a common source of errors in array-based code, and you must take steps to guard against it.

- If the index expression is based on input data from the user, or a parameter passed from another function, you may want to insert an if-statement to check its value.
- If the index is generated locally, make sure it is not possible to create an index outside the proper range.

Variable-Length Arrays

Early versions of C required that the size of an array be known by the compiler when it is declared. This is the only way that the compiler knows how much space to allocate on the stack.

In recent versions (since 1999), you are allowed to declare arrays for which the length is a variable (or derived from a variable). For instance...

```
int Calculate(int len) {  
    int data[len]; // length not known until runtime  
    ...  
}
```

Requires a different way of allocating the array, which will be explained when we discuss *dynamic memory allocation* in later chapters.

Multi-Dimensional Arrays

We can extend the notion of a one-dimensional array to two or more dimensions.

Consider a image captured by a digital camera.

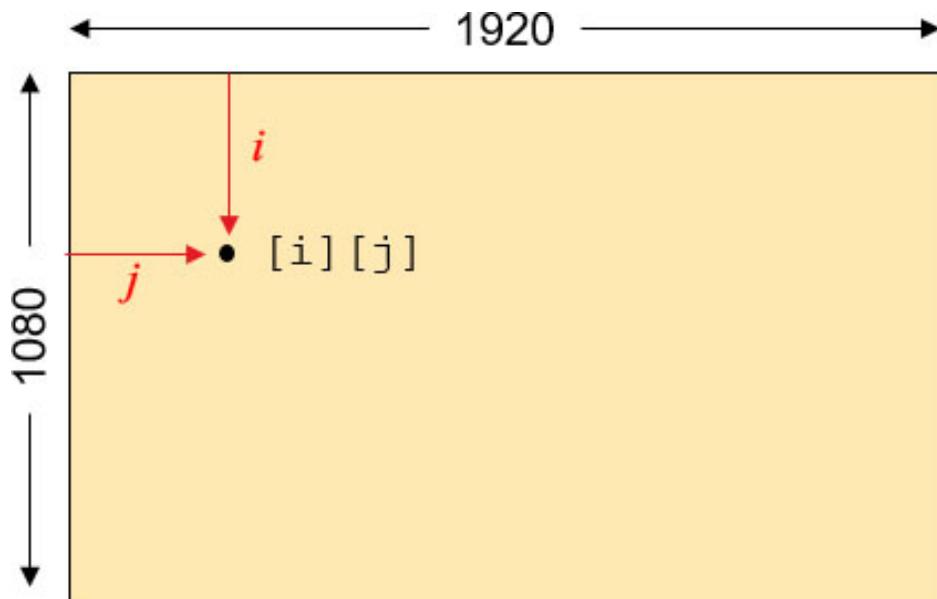
It is helpful to consider a two-dimensional grid of pixels, for example with 1080 rows and 1920 columns.

We can declare a two-dimensional array like this:

```
int image[1080][1920];
```

And we refer to a specific element using two indices instead of one:

```
image[38][283] = 0xFFFF;  
// row 38, col 283
```



[Access the text alternative for slide images.](#)

Memory Allocation of a Two-Dimensional Array

Even if we think of this a two dimensions, it still must be mapped into a one-dimensional memory.

C uses **row-major** order, in which elements of a column are contiguous in memory. We can think of this as an "array of column arrays".

To calculate the offset of element $[i][j]$:

$$\text{offset} = (i \times \text{column_size}) + j$$

This array requires over two million elements, which is too large to fit in the LC-3 memory! In the example to the right, the memory space is extended to 32 bits just to illustrate the concept.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.

x0001 EA00	image[0][0]
x0001 EA01	image[0][1]
x0001 EA02	image[0][2]
x0001 EA03	image[0][3]
	⋮
x0001 F17E	image[0][1918]
x0001 F17F	image[0][1919]
x0001 F180	image[1][0]
x0001 F181	image[1][1]
	⋮
x0001 F8FE	image[1][1918]
x0001 F8FF	image[1][1919]
x0001 F900	image[2][0]
x0001 F901	image[2][1]
	⋮

[Access the text alternative for slide images.](#)

More than Two Dimensions

We can extend the concept beyond two dimensions:

```
int dataVolume[40][50][60]; // 120,000 elements
```

The memory allocation scheme follows the same pattern.

- Elements which differ by one in the rightmost dimension are in consecutive memory locations.
- Move through indices in right-to-left order.

To calculate the offset of $[i][j][k]$:

$$\text{offset} = (i \times \dim_0 \times \dim_1) + (j \times \dim_0) + k$$



Because learning changes everything.[®]

www.mheducation.com

From Bits and Gates to C and Beyond

Recursion

Chapter 17

Fibonacci

Compute the n -th number in the Fibonacci series.

Math: Recurrence Equation

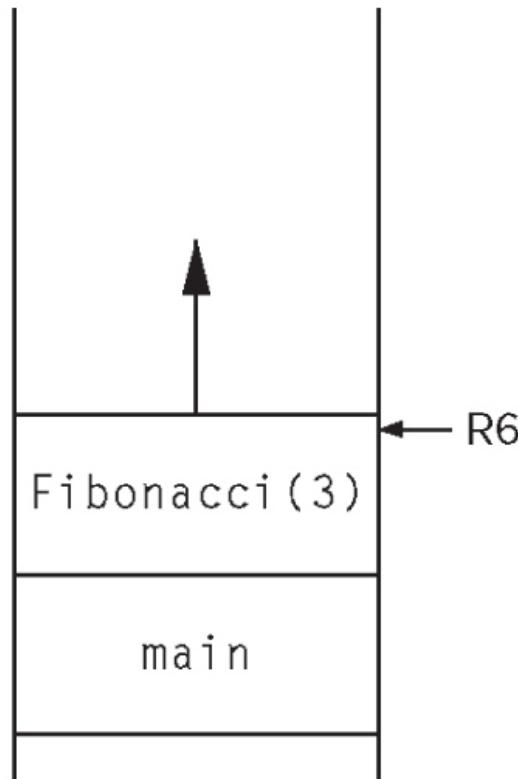
$$\begin{aligned}f(n) &= f(n - 1) + f(n - 2) \\f(1) &= 1 \\f(0) &= 1\end{aligned}$$

C: Recursive Function

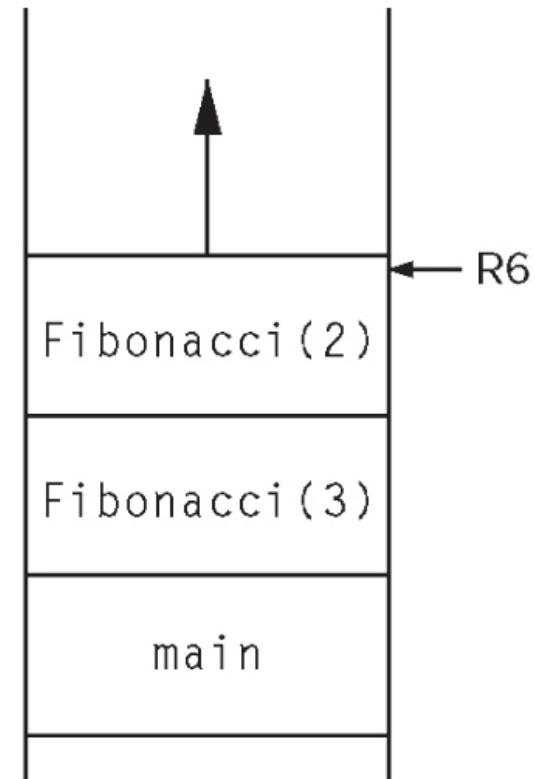
```
int Fibonacci(int n) {  
    int sum;  
    if (n == 1 || n == 0)  
        return 1;  
    else  
        sum = Fibonacci(n-1) +  
              Fibonacci(n-2);  
    return sum;  
}
```

Fibonacci: Calling Sequence 1

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



Step 1: Initial call

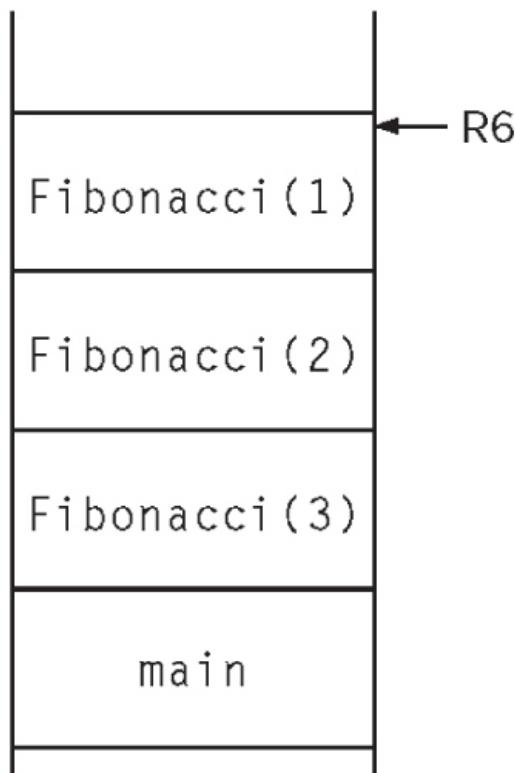


Step 2: Fibonacci(3) calls Fibonacci(2)

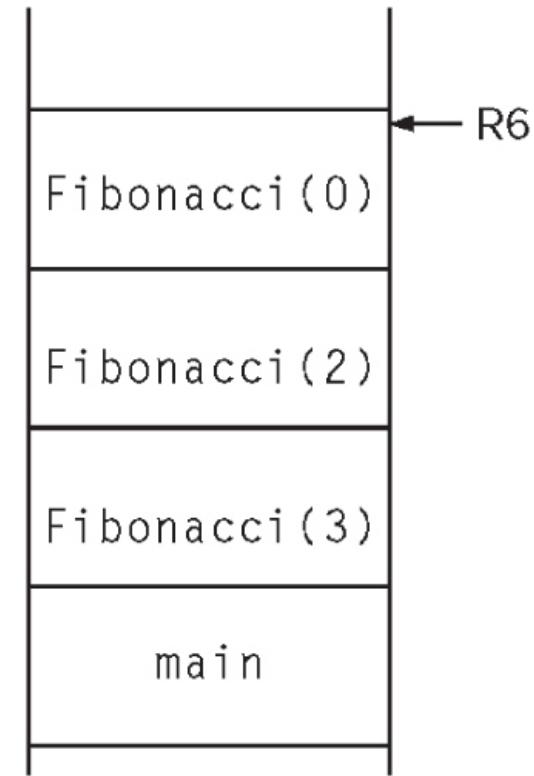
[Access the text alternative for slide images.](#)

Fibonacci: Calling Sequence 2

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



Step 3: Fibonacci(2) calls Fibonacci(1)

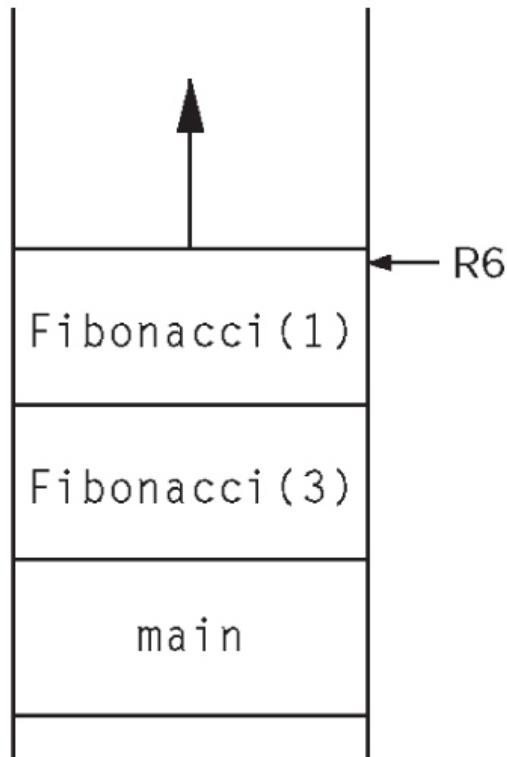


Step 4: Fibonacci(2) calls Fibonacci(0)

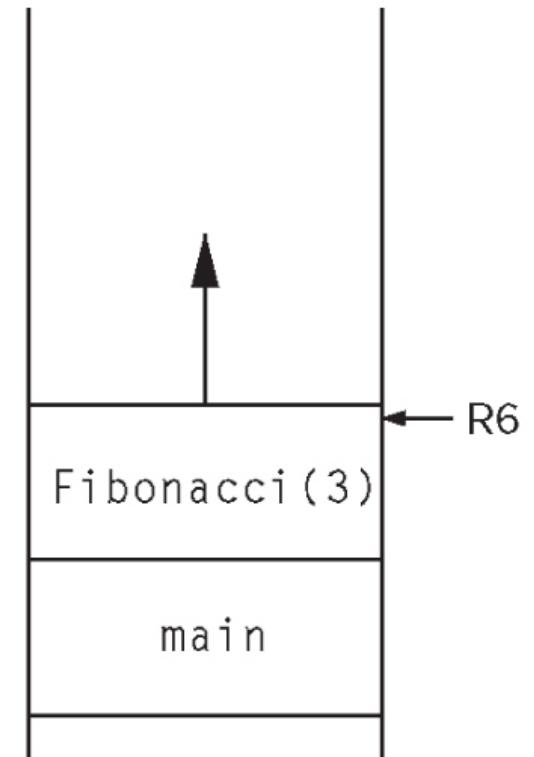
[Access the text alternative for slide images.](#)

Fibonacci: Calling Sequence 3

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



Step 5: Fibonacci (3) calls Fibonacci (1)



Step 6: Back to the starting point

[Access the text alternative for slide images.](#)

Translate Fibonacci to LC-3

Fibonacci

```
ADD R6, R6, #-1      ; space for return val
ADD R6, R6, #-1      ; push return addr
STR R7, R6, #0
ADD R6, R6, #-1      ; push dynamic link
STR R5, R6, #0
ADD R5, R6, #-1      ; set frame pointer
ADD R6, R6, #-2      ; push local (sum) and one temp variable

;; if (n == 0 || n == 1)
LDR R0, R5, #4      ; get n
BRn FIB_ELSE        ; skip to recursive case
ADD R0, R0, #-1
BRp FIB_ELSE        ; skip to recursive case

;; return 1
AND R0, R0, #0
ADD R0, R0, #1
STR R0, R5, #3      ; store to return value
BRnzp FIB_END       ; finish return
```

Translate Fibonacci to LC-3²

```
;;; else sum = Fibonacci(n-1) + Fibonacci(n-2)

FIB ELSE
    LDR R0, R5, #4      ; push n-1
    ADD R0, R0, #-1
    ADD R6, R6, #-1
    STR R0, R6, #0
    JSR Fibonacci        ; Fibonacci(n-1)
    LDR R0, R6, #0      ; get return value
    ADD R6, R6, #2      ; pop r.v. and args
    STR R0, R5, #-1    ; store to temp variable

    LDR R0, R5, #4      ; push n-2
    ADD R0, R0, #-2
    ADD R6, R6, #-1
    STR R0, R6, #0
    JSR Fibonacci        ; Fibonacci(n-2)
    LDR R0, R6, #0      ; get return val and pop stack
    ADD R6, R6, #2
    LDR R1, R5, #-1    ; get temp, compute sum
    ADD R0, R0, R1
    STR R0, R5, #0
```

Translate Fibonacci to LC-3

```
;; return sum  
LDR R0, R5, #0 ; store to return value  
STR R0, R5, #3  
  
FIB_END  
ADD R6, R6, #2 ; pop local/temp  
LDR R5, R6, #0 ; pop dynamic link  
ADD R6, R6, #1  
LDR R7, R6, #0 ; pop return address  
ADD R6, R6, #1  
RET
```

Fibonacci Observations

Generated code uses stack to store local versions of sum.

No additional code is needed to support recursion -- it comes as a natural consequence of the way functions are implemented.

- Why is it important that we push the return address (R7) in the preamble?

Compiler creates a temporary variable to hold the value of Fibonacci ($n-1$) while it computes Fibonacci ($n-2$). This is a common technique for saving temporary state while evaluating a complex expression -- in this case, the sum of two function calls.

- Why can't we just keep the value in R0 and use R1 for the second call?

As noted in Chapter 8, this is easy to write but very inefficient, because it recomputes the same value again and again.



Because learning changes everything.[®]

www.mheducation.com

From Bits and Gates to C and Beyond

Dynamic Data Structures in C

Chapter 19

Define a Structure

Instead, we want to capture all of the information about an airplane in a single memory object. (We are using the word "object" very informally here. It's not the same as an object in a language like C++ or Python.)

Essentially, we define a new type and tell the compiler what the components of that type are:

```
struct flightType {  
    char ID[7];          // max 7 characters  
    int altitude;        // in meters  
    int longitude;       // in tenths degrees  
    int latitude;        // in tenths of degrees  
    int heading;         // in tenths of degrees  
    double airSpeed;     // in km/hr  
};
```

The keyword `struct` is used to tell the compiler that we are defining a new type, and `struct flightType` is the name of that type. Each component (field) of the new type of value has a type and a name.

Fields

```
struct flightType {  
    char ID[7];          // max 7 characters  
    int altitude;        // in meters  
    int longitude;       // in tenths degrees  
    int latitude;         // in tenths of degrees  
    int heading;          // in tenths of degrees  
    double airSpeed;      // in km/hr  
};
```

Once we have a variable, we use the dot operator (.) to access the various components (fields) of the value.

```
struct flightType plane;    // allocates a variable  
  
plane.airspeed = 800.0;  
plane.altitude = 10000;  
x = plane.heading;
```

Memory Allocation

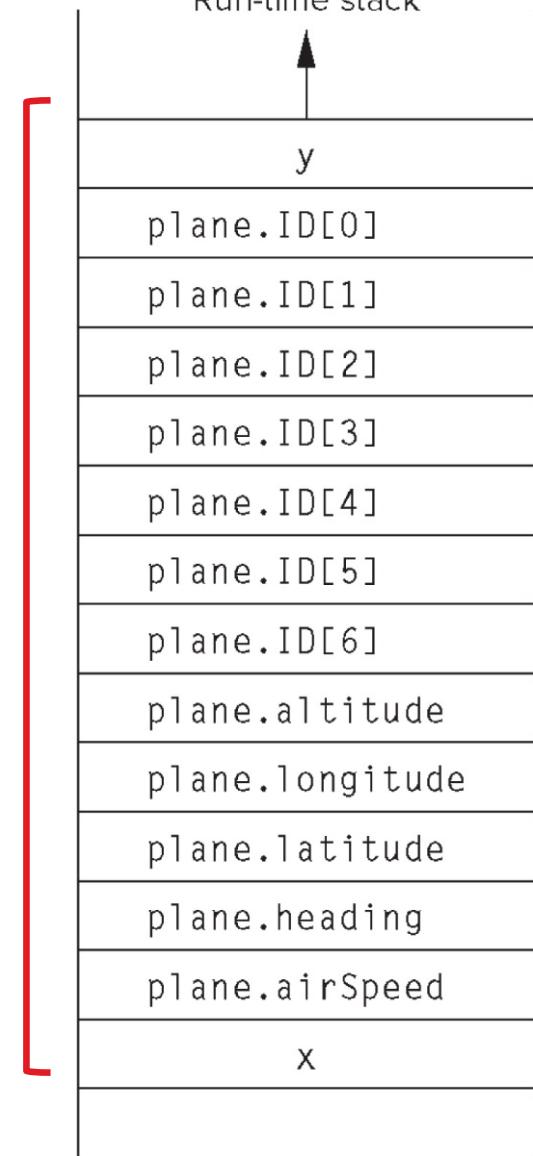
A struct is allocated in a contiguous block of memory. The fields are allocated in the order in which they appear in the definition.

```
struct flightType {  
    char ID[7];  
    int altitude;  
    int longitude;  
    int latitude;  
    int heading;  
    double airSpeed;  
};
```

```
// local variables  
int x;  
struct flightType plane;  
int y;
```

Copyright © McGraw-Hill Education. Permission required for reproduction or display.

Run-time stack



Note: In this illustration, a double is assumed to be one word.

Using `typedef` to Name a Type

To simplify code, or to make it more expressive, we can give a new user-defined name to any type.

```
typedef type_spec name ;
```

This allows us to create a name for our struct type, so that we don't have to type `struct` each time we declare a variable.

```
typedef struct flightType Flight;
```

```
Flight plane; // declare a variable of type Flight  
// allocates a struct flightType
```

By convention, a user-defined type starts with an upper-case letter.

Implementing Structures

The size of a struct variable is determined by the size of its fields.

- LC-3: Size of a Flight is 12 words: 7 char, 4 int, 1 double.

Each field has a constant offset from the beginning of the struct.

Compiler knows the offset from the definition; kept in the symbol table.

To access a field:

1. Get the base address of the struct object.
2. Add the offset to the base address.
3. Load/store to read/write the field.

Generally, steps 2 and 3 can be done in one instruction (LDR/STR).

Implementation Example

```
int x;  
Flight plane;  
int y;  
  
plane.altitude = 0;
```

```
; memory layout: slide 7  
AND R0, R0, #0      ; create zero  
ADD R1, R5, #-12    ; base addr of plane  
STR R0, R1, #7      ; store to altitude  
; offset = 7
```

Array of Structures

Now we want a program that handles 100 airplanes.

Allocate an array:

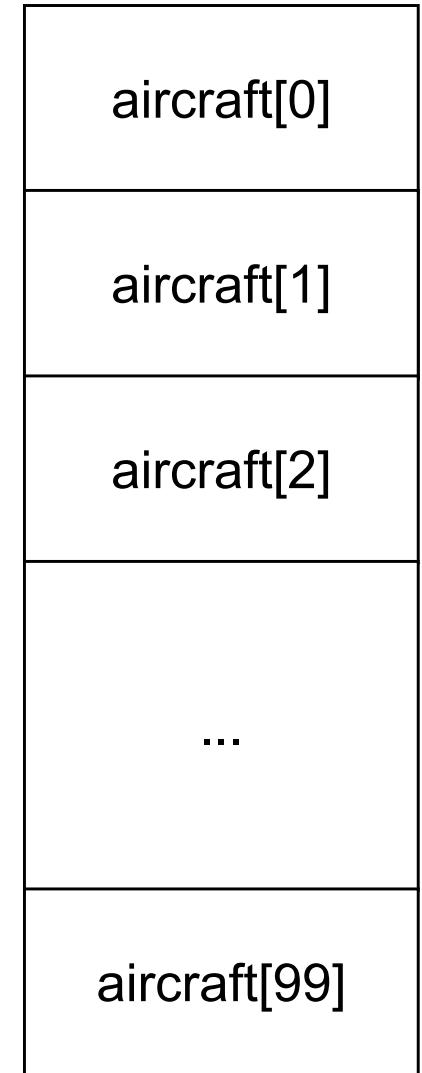
```
Flight aircraft[100];
```

Each element of the array is a struct flightType.

Each element of the array is 12 words (LC-3).

To get to the next element of the array, add 12.

```
// calculate average air speed
sum = 0.0;
for (i = 0; i < 100; i++) {
    sum += aircraft[i].airSpeed;
}
sum = sum / 100.0;
```



Pointer to a Structure

Since a structure is a type of memory object, we can create a pointer to it.

```
Flight *aircraftPtr;
```

```
aircraftPtr = &aircraft[34]; // pt to a particular plane
```

How would we access the longitude field of that aircraft?

```
(*aircraftPtr).longitude
```

Parentheses are required, because the field operator (.) has a higher precedence than the dereference operator (*).

Since we use pointers a lot, and this notation is rather awkward and ugly, C provides a special operator (->) that dereferences a struct pointer before accessing a field.

```
aircraftPtr->longitude
```

Key point: Use dot (.) with a struct value, and arrow (->) with a struct pointer.

Passing a Structure to a Function

Structures are **passed by value**.

If we pass a structure as an argument, a copy of the structure is created. For this reason, we usually choose to pass a pointer to a structure, rather than the structure itself.

```
double AirDistance(Flight *plane1, Flight *plane2);  
// in the function implementation,  
// use plane1->longitude, etc.
```

Dynamic Memory Allocation

Suppose we don't know how many planes to allocate?

The user of our program will tell us, but we won't know the size of the array until the program runs.

We can specify a maximum allocate for that -- wasteful.

Instead, we use **dynamic memory allocation** to allocate exactly the amount of memory we want, exactly when we want to use it.

Two main concepts:

- The **heap** is where dynamic memory objects are allocated.
- We use standard functions **malloc** and **free** to allocate and deallocate memory.

The Heap

Recall the LC-3 memory map.

We have two areas to allocate variables:

1. Global data section:

fixed size, determined when we compiler the program

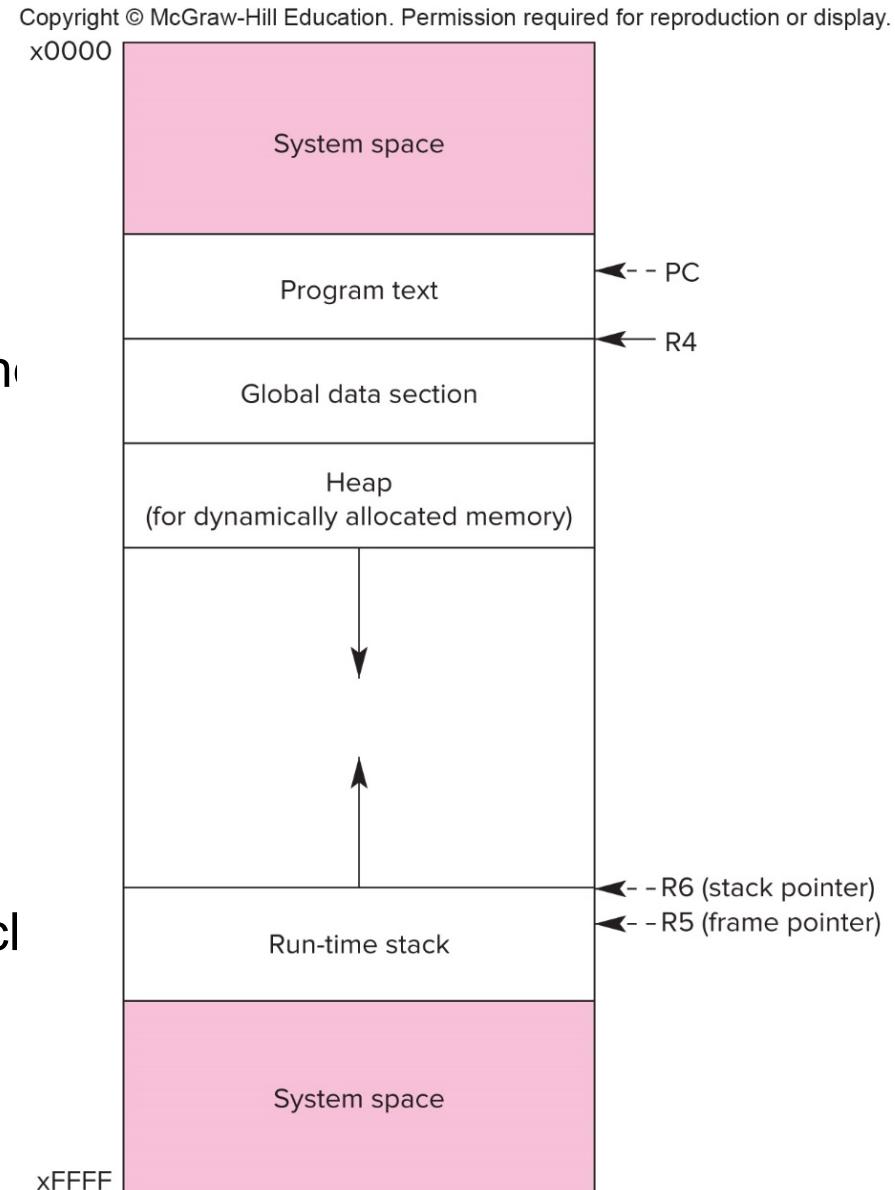
2. Runtime stack:

variable size, push/pop local variables and temporaries during function calls, grows downward in memory in a LIFO fashion

Now we add a third area:

The **heap** grows upward in memory.

It is not LIFO. The operating system keeps track of what memory is available and what is allocated. Program asks OS to allocate and deallocate memory as needed.



Memory Allocation: malloc

The `malloc` function allocates a block of memory and returns a pointer to the allocated block.

Argument = **number of bytes** to allocate

If the allocation cannot be performed, returns NULL.

```
int numAircraft;  
Flight *planes;  
  
printf("Total number of aircraft?\n");  
scanf("%d", &numAircraft);  
  
planes = malloc(24 * numAircraft);  
// allocate an array of Flights, each takes 24 bytes on LC-3
```

Bytes? Now our code is not portable! We would have to change it for each different platform. Instead, use the compiler operator `sizeof`.

```
planes = malloc(sizeof(Flight) * numAircraft);
```

Casting the Pointer from malloc

The `malloc` function has no idea what it is allocating, just its size.

So how does it know what kind pointer to return? It doesn't.
It is defined to return a generic pointer type: `void*`

It's legal to assign a `void*` value to any kind of pointer value, but we want to "cast" the pointer to the correct type, like this:

```
planes = (Flight*) malloc(sizeof(Flight) * numAircraft);
```

Casting specifies an explicit type conversion. This allows the compiler to double-check that we're doing the right thing.

Finally, we should always check to be sure that memory was actually allocated.

```
scanf("%d", &numAircraft);
planes = (Flight*) malloc(sizeof(Flight) * numAircraft);
if (planes == NULL) {
    printf("Memory allocation error!\n");
    // could return or do something else...
}
```

Memory Deallocation: `free`

If we keep allocating memory, we will eventually run out of space.

When we are finished with memory that was allocated, we should give it back to the operating system to use for other parts of the program.

The `free` function does this.

Argument = pointer that was previously returned by `malloc`

Return value: none

```
free(planets);
```

It's important to use exactly the same pointer value that was returned. The OS remembers the size that was allocated for that pointer, so it can correctly return the right amount of memory to the heap.

Best practice: Always free memory when it is no longer being used.

Memory Allocation Bugs

Using `malloc` and `free` correctly can be tricky, and is a notorious source of bugs in complex programs.

Example: The variable that holds the pointer gets changed, and the value is lost. This is known as a memory leak, because there is no way to ever deallocate the memory without having the pointer.

Example: The memory object is allocated inside a function and passed out for the rest of the program to use. Who "owns" this pointer? How do you know when it should be freed?

Example: We have a variable that points to an allocated object. We have called `free`, but the variable still contains the pointer. This is known as a "dangling pointer." We can still dereference the pointer -- at best, it now points to garbage data and the program crashes; at worst, we get garbage data that looks legitimate and we use it, creating a very hard-to-find error.

Linked List

Instead of pre-allocating a number of items (e.g., array), we allocate an item whenever we need it (using `malloc`).

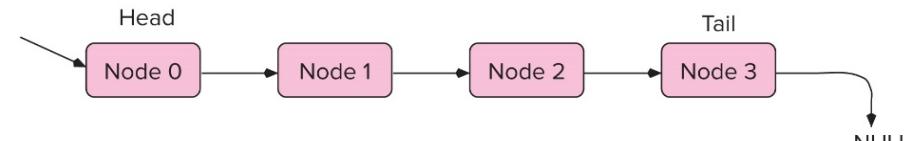
An item is typically called a **node** of the list.

The items are allocated in arbitrary places in memory (on the heap). Each node has a **pointer** that tells where to find the **next node** in the list.

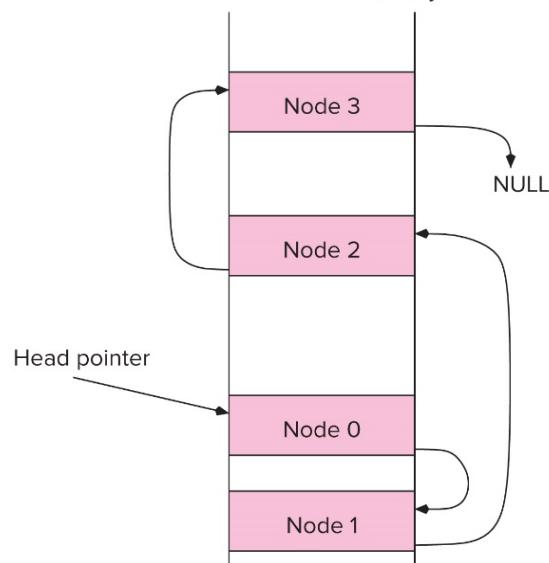
A **head** pointer points to the beginning of the list.

A null pointer means that we're at the end of the list: there is no next node.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.
A linked list in abstract form



A linked list in memory



C Implementation of a Linked List

A linked list node must have (a) data and (b) a **pointer to the next node**.

```
typedef struct flightType Flight;  
  
struct flightType {  
    char ID[6];  
    int altitude;  
    int longitude;  
    int latitude;  
    int heading;  
    double airSpeed;  
    Flight *next;  
};  
  
Flight *list = NULL; // list is represented by head pointer  
                    // initially, the pointer is NULL  
                    // because the list is empty
```

Create a List Node

```
Flight * CreateFlight(char *ID, int longitude, int latitude,
                      int altitude, double airSpeed) {
    Flight *newFlight;
    // allocate a new node
    newFlight = (Flight*)malloc(sizeof(Flight));
    // initialize node data
    strcpy(newFlight->ID, ID);
    newFlight->longitude = longitude;
    newFlight->latitude = latitude;
    newFlight->altitude = altitude;
    newFlight->airSpeed = airSpeed;
    // initialize pointer
    newFlight->next = NULL;
    return newFlight;
}
```

Traversing a Linked List

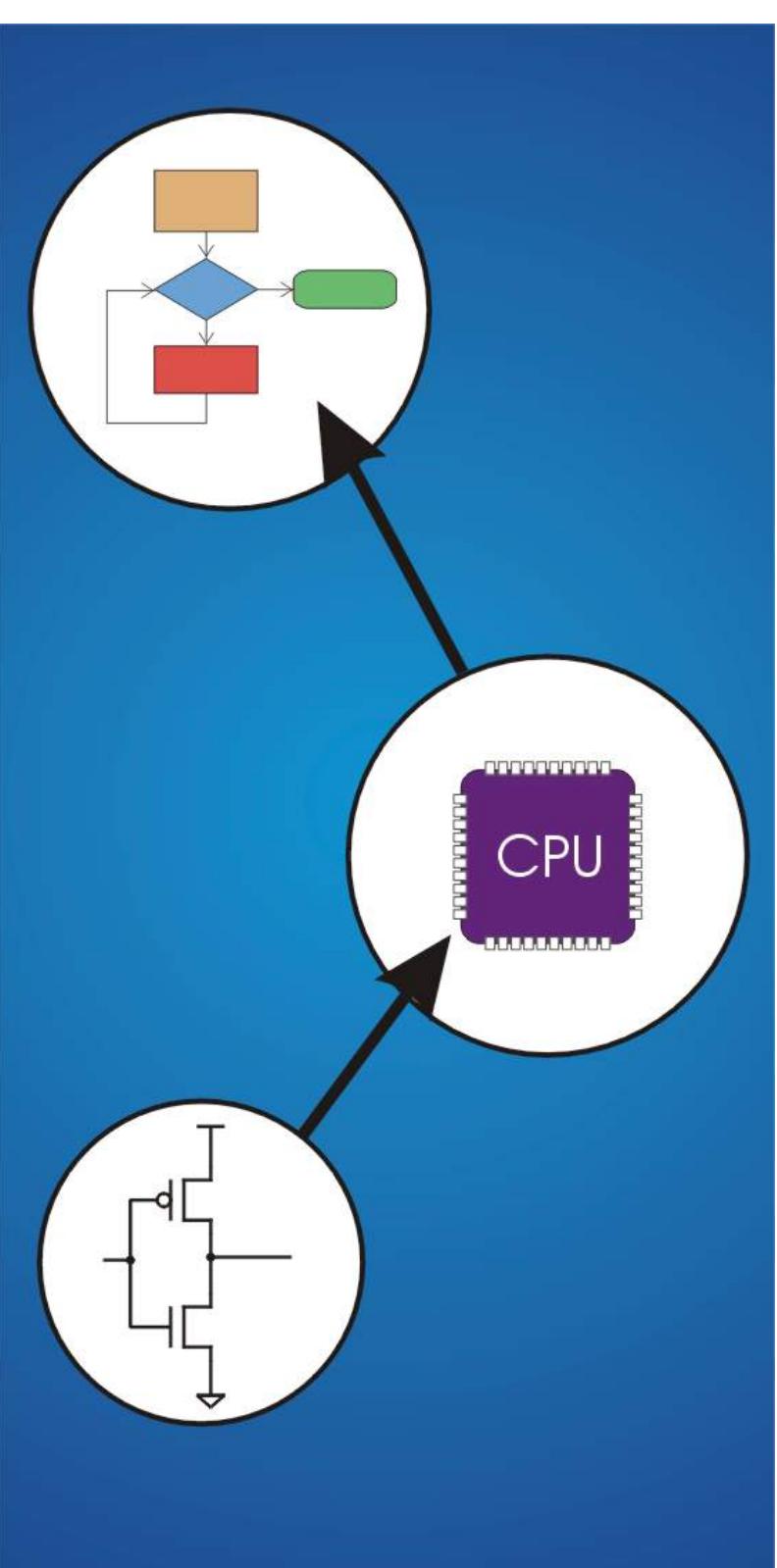
Suppose we have a linked list created already. (We'll see how in the next slides.) We often want to visit each node in the list, either to do something to each node or to look for a particular node.

```
void PrintAirspace(Flight *list) {  
    // list points to the head of a linked list of Flights  
    int count = 1;  
    printf("Aircraft in Airspace-----\n");  
    while (list != NULL) { // while not at the end  
        printf("Aircraft: %d\n", count);  
        printf("Longitude: %d\n", list->longitude);  
        printf("Latitude: %d\n", list->latitude);  
        // etc. etc.  
        count = count + 1;  
        list = list->next; // move to next node  
    }  
    printf("\n\n");  
}
```



Because learning changes everything.[®]

www.mheducation.com



Wrap-up Questions

Course Outline

Bits and Bytes

- How do we represent information using binary representation?

Processor and Instruction Set

- What are the components of a processor?
- What operations (instructions) will we implement?

Assembly Language Programming

- How do we use processor instructions to implement algorithms?
- How do we write modular, reusable code? (subroutines)

I/O, Traps, and Interrupts

- How does processor communicate with outside world?

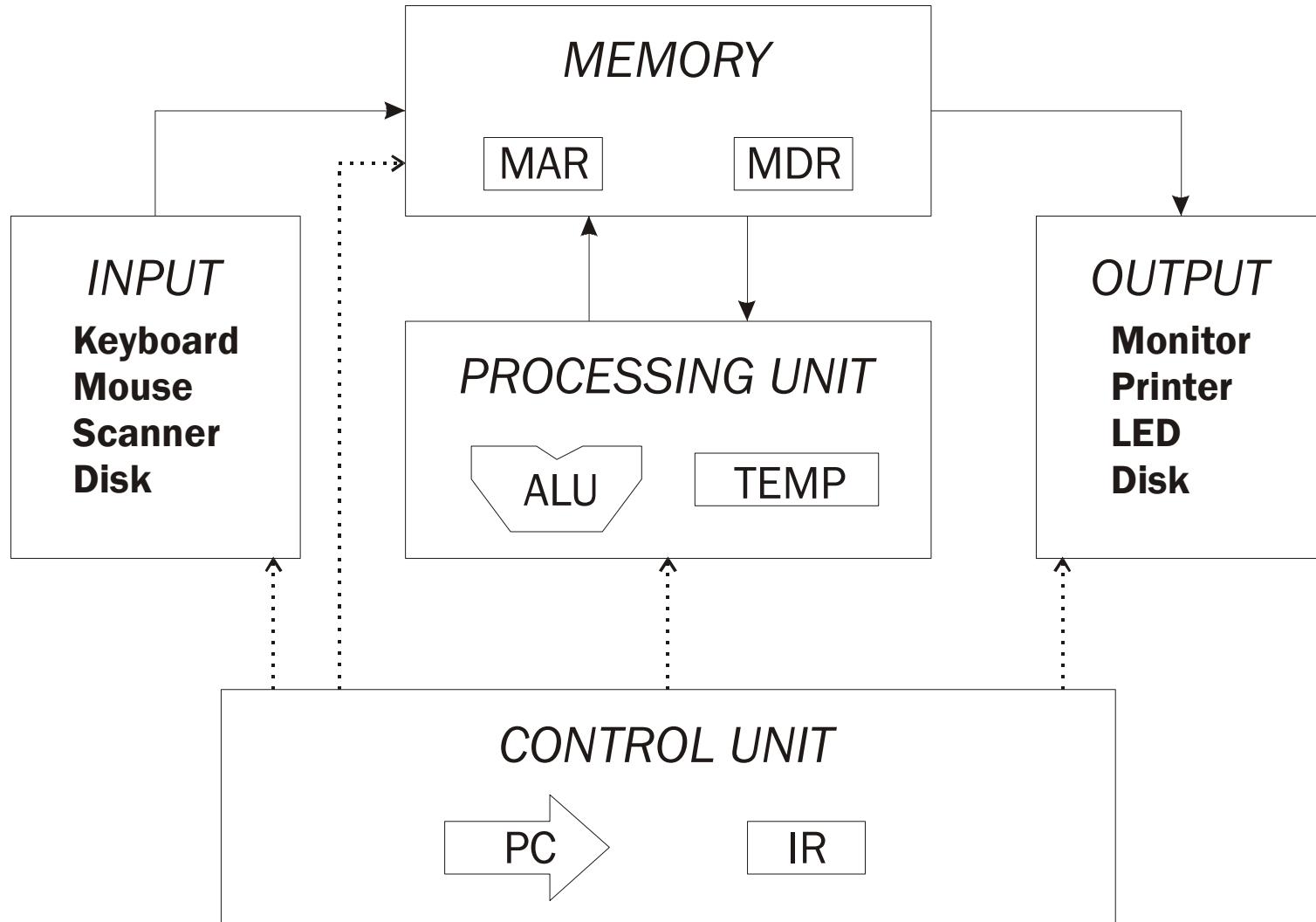
C Programming

- How do we implement high-level programming constructs at the machine level?

Chapter 2: binary representation & data types

- Why is 2's complement representation used to represent signed integers?
- What does this represent: 00100011
- Show two ways (in C language) to determine whether a number is even or odd

Von Neumann Model



Memory and Instructions

- **What is the address space (number of locations) and the addressability (# of bits stored at each location) of the LC-3?**
- **What are the 6 phases of the instruction processing cycle?**

LC-3

- **What are the addressing modes of the LC-3?**
- **LC-3 instructions fall into three classes. What are they?**
- **What does the first line of an LC-3 machine language or assembly language program contain? Why is it needed?**

Tricky assembly language question

What is the value of R1 when the HALT instruction is reached?

```
.ORIG x3000  
LD R0,A  
AND R1,R1,#0  
STR R1,R0,#3  
ADD R1,R1,#5  
TRAP x25  
A .fill x3000  
.END
```

Reverse Assembly

Fill in the blanks:

	.ORIG x3000	0011000000000000
	AND R0, R0, x0	010100000100000
	AND R1, R1, x0	0101001001100000
	ADD R1, R1, x9	0001001001101001
		000010000000100
	LD R2, FF	0010010000001000
	LEA R3, FF	1110011000000111
		0111001011000010
	LEA R7, DD	1110111000000011
EE	NOT R5, R5	1001101101111111
	BRnz DD	0000110000000001
	NOT R4, R3	1001100011111111
		0110110010000001
	TRAP x25	1111000000100101
		1101000000000000
	.FILL xFF00	1111111100000000
	.FILL xFAFA	1111101011111010
	.END	

I/O

- What does it mean to say that I/O is memory-mapped?
- What is the danger of not testing the DSR before writing data to the screen?
- What determines whether an interrupt signal is sent to the control unit?
- When does the control unit test for an interrupt signal?

TRAP routines and subroutines

- **What are TRAP routines and how do they differ from subroutines?**
- **How do they receive and send information to the caller?**
- **What needs to be done before one subroutine calls another?**

Stacks

- Draw a picture of a software stack as implemented on the LC-3
 - How does one keep track of the top of the stack?
 - How is a stack used to service interrupts?

C & LC-3 Assembly Language

- **Compile the following into LC-3 assembly language**

```
int i = 0, j = 2;  
i = j + 10;  
if(i > 10)  
    i = i - 10;  
putchar(i);
```

Runtime stack and heap

For each item highlighted in the code below, indicate where that item is allocated in memory. Is it on the runtime stack or is it on the heap?

```
int main() {
    int x;
    int *y;

x = 3;

y = (int *) malloc(sizeof(int));

*y = 4;

    return 0;
}
```

Many More Low Level Programming Opportunities!

- CS 3413 Operating Systems I
- CS 3853 Computer Architecture and Organization
- CS 3873 Net-centric Computing
- CS 4405 Operating Systems II
- CS 4735 Computer Graphics
- CS 4745 Introduction to Parallel Processing
- Several security courses

An excellent reference book: *Computer Systems, a Programmer's Perspective*, by Bryant and O'Hallaron , 3rd ed, Prentice Hall