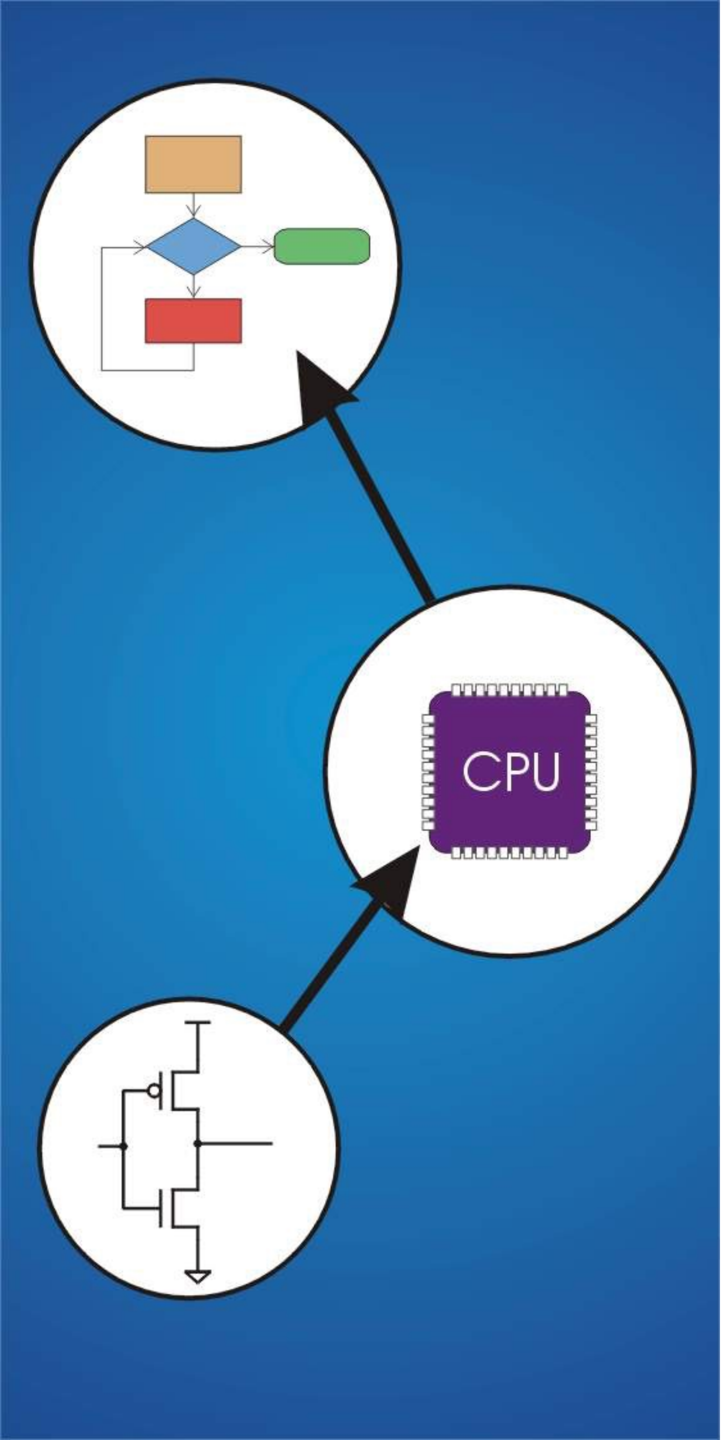

Chapter 2
Bits, Data Types, and Operations

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"**

Could base state on *value* of voltage,
but control and detection circuits more complex.

- **compare turning on a light switch to measuring or regulating voltage**

Computer is a binary digital system.

Digital system:
- finite number of symbols

Binary (base two) system:
- has two states: 0 and 1

Basic unit of information is the *binary digit*, or *bit*.

Values with more than two states require multiple bits.

- **A collection of two bits has four possible states:**
  00, 01, 10, 11
- **A collection of three bits has eight possible states:**

  000, 001, 010, 011, 100, 101, 110, 111
- ***A collection of n bits has 2^n possible states.***

What kinds of data do we need to represent?

- **Numbers** – signed, unsigned, integers, floating point, complex, rational, irrational, …
- **Logical** – true, false
- **Text** – characters, strings, …
- **Images** – pixels, colors, shapes, …
- **Sound**
- …

Data type:

- ***representation*** **and its associated** ***operations*** **within the computer**

We'll start with numbers…

Unsigned Integers

Non-positional notation

- could represent a number (“5”) with a string of ones (“11111”)
- problems?

Weighted positional notation

- like decimal numbers: “329”
- “3” is worth 300, because of its position, while “9” is only worth 9

329

10^2 10^1 10^0

$\boxed{3x100 + 2x10 + 9x1 = 329}$

most significant 101 *least significant*

2^2 2^1 2^0

$\boxed{1x4 + 0x2 + 1x1 = 5}$

Unsigned Integers (cont.)

An *n*-bit unsigned integer represents 2^n values: from 0 to 2^n-1.

2^2	2^1	2^0	
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

carry

$$
\begin{array}{r}
10010 \\
+\underline{\quad 1001} \\
11011
\end{array}
\qquad
\begin{array}{r}
10010 \\
+\underline{\quad 1011} \\
11101
\end{array}
\qquad
\begin{array}{r}
1111 \\
+\underline{\qquad 1} \\
10000
\end{array}
$$

$$
\begin{array}{r}
10111 \\
+\underline{\quad\ 111}
\end{array}
$$

Subtraction, multiplication, division,…

Signed Integers

With n bits, we have 2^n distinct values.

- assign about half to positive integers (1 through 2^{n-1}) and about half to negative (- 2^{n-1} through -1)
- that leaves two values: one for 0, and one extra

Positive integers

- just like unsigned – zero in *most significant* (MS) bit
  00101 = 5

Negative integers

- sign-magnitude – set MS bit to show negative, other bits are the same as unsigned
  10101 = -5
- one's complement – flip every bit to represent negative
  11010 = -5
- in either case, MS bit indicates sign: 0=positive, 1=negative

Two's Complement

Problems with sign-magnitude and 1's complement

- two representations of zero (+0 and –0)
- arithmetic circuits are complex
  - ➢How to add two sign-magnitude numbers?
    - e.g., try 2 + (-3)
  - ➢How to add to one's complement numbers?
    - e.g., try 4 + (-3)

Two's complement **representation developed to make circuits easy for arithmetic.**

- for each positive number (X), assign value to its negative (-X), such that X + (-X) = 0 with "normal" addition, ignoring carry out

	00101	(5)		**01001**	(9)
+	**11011**	(-5)	+	**10111**	(-9)
	00000	(0)		**00000**	(0)

Two's Complement Representation

If number is positive or zero,

- normal binary representation, zeroes in upper bit(s)

If number is negative,

- start with positive number
- flip every bit (i.e., take the one's complement)
- then add one

00101	(5)		01001	(9)
11010	(1's comp)		10110	(1's comp)
+ 1			+ 1	
11011	(-5)		10111	(-9)

Two's Complement Shortcut

To take the two's complement of a number:

- **copy bits from right to left until (and including) the first "1"**
- **flip remaining bits to the left**

```
  011010000
  100101111   (1's comp)
+         1
-----------
  100110000
```

```
       0110 | 10000
(flip)      |        (copy)
       1001 | 10000
```

Two's Complement Signed Integers

MS bit is sign bit – it has weight -2^{n-1}.

Range of an n-bit number: -2^{n-1} through $2^{n-1} - 1$.

- **The most negative number (-2^{n-1}) has no positive counterpart.**

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

-2^3	2^2	2^1	2^0	
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

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.**

$$X = 01101000_{two}$$
$$= 2^6+2^5+2^3 = 64+32+8$$
$$= 104_{ten}$$

Assuming 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

$$X = 00100111_{two}$$
$$= 2^5+2^2+2^1+2^0 = 32+4+2+1$$
$$= 39_{ten}$$

$$X = 11100110_{two}$$
$$-X = 00011010$$
$$= 2^4+2^3+2^1 = 16+8+2$$
$$= 26_{ten}$$
$$X = -26_{ten}$$

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

Assuming 8-bit 2's complement numbers.

Converting Decimal to Binary (2's C)

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, take two's complement.**

$X = 104_{ten}$

104/2 =	52 r0	*bit 0*
52/2 =	26 r0	*bit 1*
26/2 =	13 r0	*bit 2*
13/2 =	6 r1	*bit 3*
6/2 =	3 r0	*bit 4*
3/2 =	1 r1	*bit 5*
1/2 =	0 r1	*bit 6*

$X = 01101000_{two}$

Converting Decimal to Binary (2's C)

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, take two's complement.**

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

$X = 104_{ten}$

104 - 64 =	40	*bit 6*
40 - 32 =	8	*bit 5*
8 - 8 =	0	*bit 3*

$X = 01101000_{two}$

Operations: Arithmetic and Logical

Recall:
a data type includes *representation* and *operations*.

We now have a good representation for signed integers, so let's look at some arithmetic operations:

- **Addition**
- **Subtraction**
- **Sign Extension**

We'll also look at overflow conditions for addition.

Multiplication, division, etc., can be built from these basic operations.

Logical operations are also useful:

- **AND**
- **OR**
- **NOT**

Addition

As we've discussed, 2's comp. addition is just binary addition.

- **assume all integers have the same number of bits**
- **ignore carry out**
- **for now, assume that sum fits in n-bit 2's comp. representation**

01101000 (104)		**11110110** (-10)	
+ **11110000** (-16)		+ **11110111** (-9)	
01011000 (98)		**11101101** (-19)	

Assuming 8-bit 2's complement numbers.

Subtraction

Negate subtrahend (2nd no.) and add.

- **assume all integers have the same number of bits**
- **ignore carry out**
- **for now, assume that difference fits in n-bit 2's comp. representation**

	01101000	(104)		**11110110**	(-10)
-	**00010000**	(16)	-		(-9)
	01101000	(104)		**11110110**	(-10)
+	**11110000**	(-16)	+		(9)
	01011000	(88)			(-1)

Assuming 8-bit 2's complement numbers.

Sign Extension

To add two numbers, we must represent them with the same number of bits.

If we just pad with zeroes on the left:

4-bit		**8-bit**	
0100	(4)	**00000100**	(still 4)
1100	(-4)	**00001100**	(12, not -4)

Instead, replicate the MS bit -- the sign bit:

4-bit		**8-bit**	
0100	(4)	**00000100**	(still 4)
1100	(-4)	**11111100**	(still -4)

Overflow

If operands are too big, then sum cannot be represented as an *n*-bit 2's comp number.

01000	(8)	**11000**	(-8)
+ 01001	(9)	**+ 10111**	(-9)
10001	(-15)	**01111**	(+15)

We have overflow if:

- **signs of both operands are the same, and**
- **sign of sum is different.**

Another test -- easy for hardware:

- **carry into MS bit does not equal carry out**

Logical Operations

Operations on logical TRUE or FALSE

- two states -- takes one bit to represent: TRUE=1, FALSE=0

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

- operation applied to each bit independently

Examples of Logical Operations

AND

- **useful for clearing bits**
  - ➢**AND with zero = 0**
  - ➢**AND with one = no change**

```
      11000101
AND   00001111
      --------
      00000101
```

OR

- **useful for setting bits**
  - ➢**OR with zero = no change**
  - ➢**OR with one = 1**

```
      11000101
OR    00001111
      --------
      11001111
```

NOT

- **unary operation -- one argument**
- **flips every bit**

```
NOT   11000101
      --------
      00111010
```

Hexadecimal Notation

It is often convenient to write binary (base-2) numbers as hexadecimal (base-16) numbers instead.

- **fewer digits -- four bits per hex digit**
- **less error prone -- easy to corrupt long string of 1's and 0's**

Binary	Hex	Decimal
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	Hex	Decimal
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Converting from Binary to Hexadecimal

Every four bits is a hex digit.

- **start grouping from right-hand side**

011101010001111010011010111

3 A 8 F 4 D 7

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

Fractions: Fixed-Point

How can we represent fractions?

- **Use a "binary point" to separate positive from negative powers of two -- just like "decimal point."**
- **2's comp addition and subtraction still work.**
  - **if binary points are aligned**

$2^{-1} = 0.5$

$2^{-2} = 0.25$

$2^{-3} = 0.125$

$$
\begin{array}{rl}
 \mathbf{00101000.101} & (40.625) \\
 + \quad \underline{\mathbf{11111110.110}} & (-1.25) \\
 \mathbf{00100111.011} & (39.375)
\end{array}
$$

No new operations -- same as integer arithmetic.

Very Large and Very Small: Floating-Point

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

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

Use equivalent of "scientific notation": $F \times 2^E$

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

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

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

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

Floating Point Example

Single-precision IEEE floating point number:

$$\underbrace{1}_{\textit{sign}}\underbrace{01111110}_{\textit{exponent}}\underbrace{10000000000000000000000}_{\textit{fraction}}$$

- **Sign is 1 – number is negative.**
- **Exponent field is 01111110 = 126 (decimal).**
- **Fraction is 0.100000000000… = 0.5 (decimal).**

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

Floating-Point Operations

Will regular 2's complement arithmetic work for Floating Point numbers?

(*Hint*: In decimal, how do we compute $3.07 \times 10^{12} + 9.11 \times 10^{8}$?)

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

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/) 16bits representation
(한글 U+AC00~U+D7A3 가나다순으로 11,172자)

No new operations -- integer arithmetic and logic.

Other Data Types

Text strings

- sequence of characters, terminated with NULL (0)
- typically, no hardware support

Image

- array of pixels
  - monochrome: one bit (1/0 = black/white)
  - color: red, green, blue (RGB) components (e.g., 8 bits each)
  - other properties: transparency
- hardware support:
  - typically none, in general-purpose processors
  - MMX -- multiple 8-bit operations on 32-bit word

Sound

- sequence of fixed-point numbers

LC-3 Data Types

Some data types are supported directly by the instruction set architecture.

For LC-3, there is only one hardware-supported data type:

- **16-bit 2's complement signed integer**
- **Operations: ADD, AND, NOT**

Other data types are supported by <u>interpreting</u> 16-bit values as logical, text, fixed-point, etc., in the software that we write.