Chapter 2
Bits, Data Types, and Operations

Based on slides © McGraw-Hill
Additional material © 2004/2005 Lewis/Martin

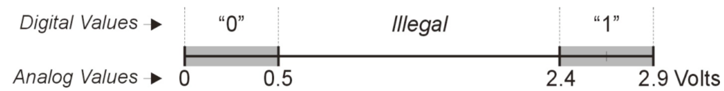
How do we represent data in a computer?

At the lowest level, a computer has electronic "plumbing"

- **Operates 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"**



Alternative: Base state on *value* of voltage

- **On/Off light switch versus dimmer switch**
- **Problem: Control/detection circuits more complex**

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: the *binary digit*, or *bit*

3+ state values 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, real, floating point, complex, rational, irrational, …**
- **Text – characters, strings, …**
- **Images – pixels, colors, shapes, …**
- **Sound**
- **Logical – true, false**
- **Instructions**
- **…**

Data type:

- ***Representation* and *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

base-10 (decimal)

329

10^2 10^1 10^0

3x100 + 2x10 + 9x1 = 329

most significant

101

least significant

base-2 (binary)

2^2 2^1 2^0

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	val
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

```
  10010      10010      01111
+ 01001    + 01011    + 00001
```

```
  10111
+   111
```

Subtraction, multiplication, division,…

Signed Integers

With n bits, we have 2^n distinct values

- Assign "half" to positive integers (1 through ~2^{n-1}) and "half" to negative (~-2^{n-1} through -1)
- That leaves two values: one for 0, and one extra

Positive integers

- Just like unsigned with zero in most significant bit
  00101 = 5

Negative integers

- Sign-magnitude: set high-order 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

Problem

Signed-magnitude and 1's complement

- Two representations of zero (+0 and –0)
- Arithmetic circuits are complex
  - ➢ How do we add two sign-magnitude numbers?
    - – *e.g.,* try 2 + (-3)

```
  00010  (2)
+ 10011  (-3)
  10001  (-1)     ?
```

  - ➢ How do we add to one's complement numbers?
    - – *e.g.,* try 4 + (-3)

```
  00100  (4)
+ 11100  (-3)
  00001  (1)      ?
```


Two's Complement

Idea

- Find representation to make arithmetic simple and consistent

Specifics

- 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)     +   _____  (-9)
  00000  (0)          00000  (0)
```


Two's Complement (cont.)

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)
```


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                    011|010000
  100101111  (1's comp)   (flip)    |   (copy)
+         1
  100110000                    100|110000
```

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 $2^{n-1} - 1$

- **Note: 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**

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

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.

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. **Change to positive decimal number**
2. **Divide by two – remainder is least significant bit**
3. **Keep dividing by two until answer is zero, recording remainders from right to left**
4. **Append a zero as the MS bit; if original number 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. **Change to positive 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***

Operations for signed integers

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

Logical operations are also useful

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

And. . .

- **Overflow conditions for addition**

Addition

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)
+	**<u>11110000</u>**	(-16)	+	**11110111**	(-9)
	01011000	(88)			(-19)

Assuming 8-bit 2's complement numbers.

Subtraction

Negate 2nd operand 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)
-	**<u>00010000</u>**	(16)	-	**11110111**	(-9)

	01101000	(104)		**11110110**	(-10)
+	**<u>11110000</u>**	(-16)	+		(9)
	01011000	(88)			(-1)

Assuming 8-bit 2's complement numbers.

Sign Extension

To add

- Must represent numbers with same number of bits

What if we just pad with zeroes on the left?

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

No, let's replicate the MSB (the sign bit)

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

Overflow

What if operands are too big?

- Sum cannot be represented as *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 MSB does not equal carry out

Logical Operations

Operations on logical TRUE or FALSE

- Two states: 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	Binary	Hex	Decimal
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	A	10
0011	3	3	1011	B	11
0100	4	4	1100	C	12
0101	5	5	1101	D	13
0110	6	6	1110	E	14
0111	7	7	1111	F	15

Converting from Binary to Hexadecimal

Every group of four bits is a hex digit

- **Start grouping from right-hand side**

011101010001111010011010111

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**

$2^{-1} = 0.5$
$2^{-2} = 0.25$
$2^{-3} = 0.125$

00101000.101 (40.625)
+ 11111110.110 (-1.25)
00100111.011 (39.375)

No new operations -- same as integer arithmetic

Very Large and Very Small: Floating-Point

Problem

- **Large values: 6.023 x 10^{23} -- requires 79 bits**
- **Small values: 6.626 x 10^{-34} -- requires >110 bits**

Use equivalent of "scientific notation": F x 2^E
Need to represent F (*fraction*), E (*exponent*), and sign.
IEEE 754 Floating-Point Standard (32-bits):

1b	*8b*	*23b*
S	Exponent	Fraction

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

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

Floating Point Example

Single-precision IEEE floating point number

10111111010000000000000000000000

sign *exponent* *fraction*

- **Sign is 1: number is negative**
- **Exponent field is 01111110 = 126 (decimal)**
- **Fraction is 0.100000000000… = 1/10 = 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
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.

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 supported data type
- 16-bit 2's complement signed integer
- Operations: ADD, AND, NOT (and sometimes MUL)

Other data types?
- Supported by interpreting 16-bit values as logical, text, fixed-point, *etc.*, in the software that we write

Next Time

Lecture
- Digital logic structures: transistors and gates

Reading
- Chapter 3-3.2

Quiz
- Online

Upcoming
- HW1 due this Friday!