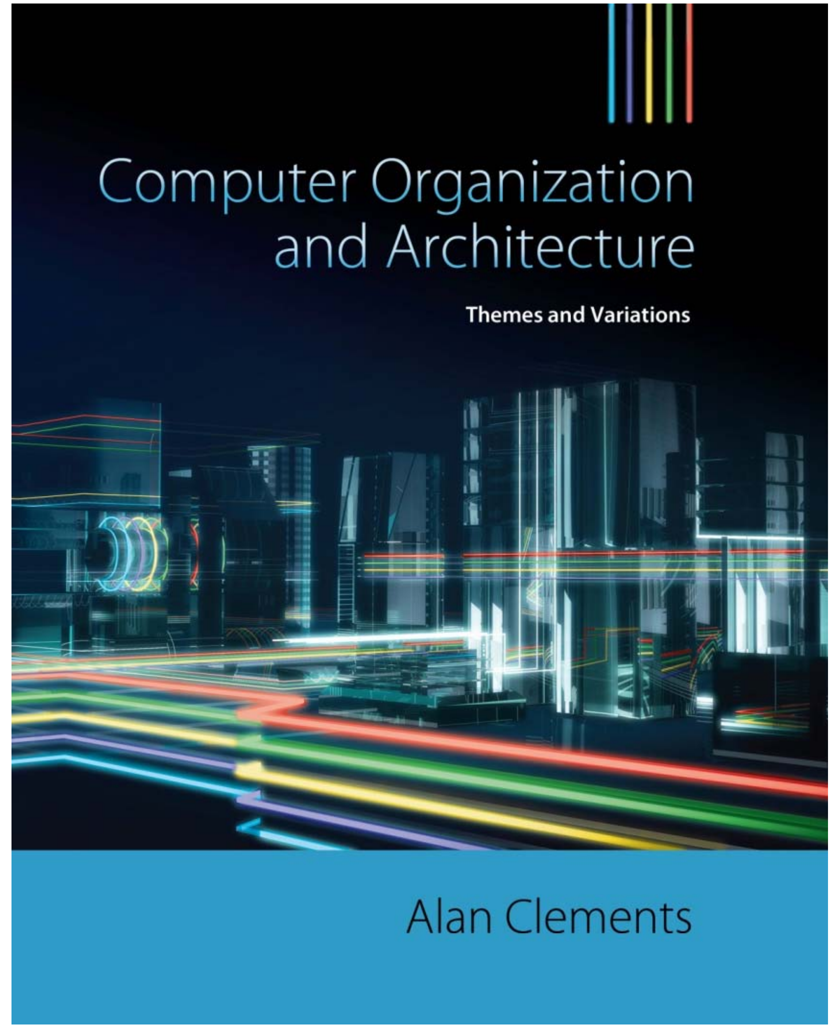
Part 1

Chapter 2

Computer Arithmetic and Digital Logic



These slides are being provided with permission from the copyright for in-class (CS2208B) use only. The slides must not be reproduced or provided to anyone outside of the class.

All download copies of the slides and/or lecture recordings are for personal use only. Students must destroy these copies within 30 days after receipt of final course evaluations.

Music: *"Corporate Success"* by *Scott Holmes*, used under *Attribution-NonCommercial License*

CENGAGE Learning

Bits and Bytes

- In digital computers, data is represented using *Bits* (*Binary digiT)s*.
- A bit has *two values* that we call 0 and 1, low and high, false and true , clear and set, and so on.
- Using bits, it is easy to represent real-world quantities.
  - Sound and images can easily be converted to bits.
  - Strings of bits can be converted back to sound or images.
- We call a *unit of 8 bits* a *byte*. This is a convention.

Bit Patterns

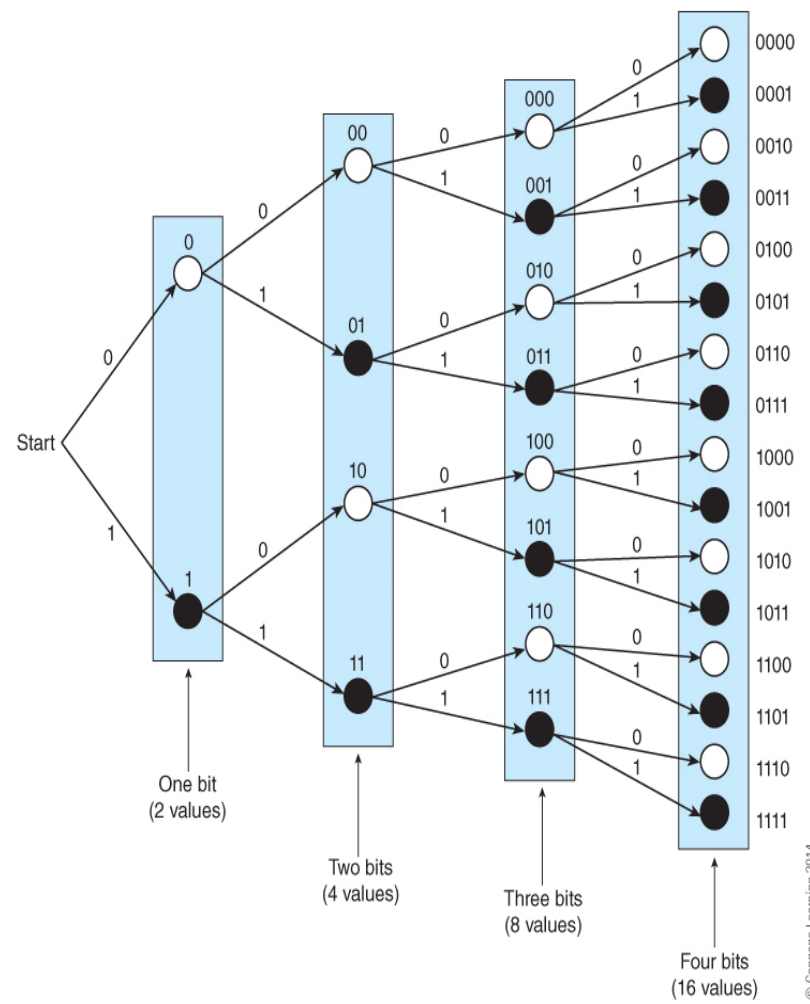
- One bit can have two values, 0 or 1.
- Two bits can have four values, 00, 01, 10, 11.
- Each time you introduce a bit, you double the number of possible combinations, as Figure 2.1 demonstrates.
- 3 bits can have (2^3) 8 values
- 4 bits can have (2^4) 16 values
- 5 bits can have (2^5) 32 values
- 6 bits can have (2^6) 64 values
- 7 bits can have (2^7) 128 values
- 8 bits can have (2^8) 256 values
- 9 bits can have (2^9) 512 values
- **10 bits can have (2^{10}) 1 K values**
- 11 bits can have (2^{11}) 2 K values
- 12 bits can have (2^{12}) 4 K values

- **20 bits can have (2^{20}) 1 Mega values**
- 23 bits can have (2^{23}) 8 Mega values

- **30 bits can have (2^{30}) 1 Giga values**
- 34 bits can have (2^{34}) 16 Giga values

n bites → 2^n values.

FIGURE 2.1 The binary tree



Bit Patterns

- One of the first quantities to be represented in digital form was the character (*letters*, *numbers*, and *symbols*).
  - This was necessary in order to transmit text across the networks that were developed as a result of the invention of the telegraph.
- This led to a standard code for characters called ASCII (*American Standard Code for Information Interchange*)
  - 7 bit code
  - representing up to $2^7 = 128$ characters of Latin alphabet.
- Today, the *16-bit unicode* has been devised to represent a much greater range of characters including *non-Latin alphabets*.

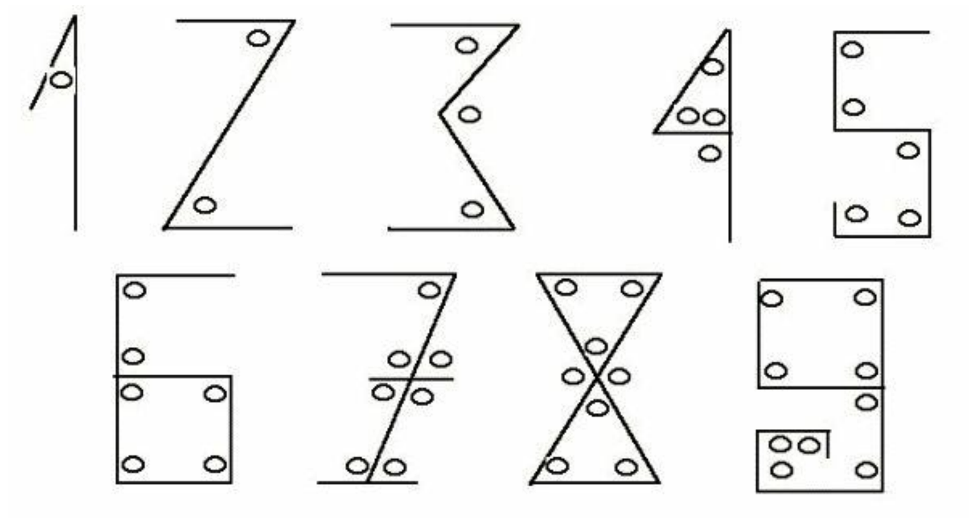
NON-Printable characters.

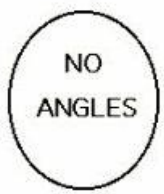
ASCII Code

code	value	code	value	code	value	code	value	code	value	code	value	code	value	code	value
0	NUL	1	SOH	2	STX	3	ETX	4	EOT	5	ENQ	6	ACK	7	BEL
8	BS	9	HT	10	NL	11	VT	12	NP	13	CR	14	SO	15	SI
16	DLE	17	DC1	18	DC2	19	DC3	20	DC4	21	NAK	22	SYN	23	ETB
24	CAN	25	EM	26	SUB	27	ESC	28	FS	29	GS	30	RS	31	US
32	SP	33	!	34	"	35	#	36	$	37	%	38	&	39	‘
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127	DEL

Numbers and Binary Arithmetic

- One of the great advances in history was the move away from Roman numerals (I, II, III, IV, V, VI, VII, VIII, IX, X, …, L, …, C, …, D, …, M, … where I=1, V=5, X=10, L=50, C=100, D=500, and M=1000) to the *Hindu-Arabic* notation (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) that we use today.
  - Invented by *Muhammad Musa **Al-Khwarizmi*** (*Born 780—Died 850*)
  - *Numerals* represent the number of angles is a digit





Numbers and Binary Arithmetic

- Arithmetic calculations are remarkably difficult using Roman numerals, but they are far *simpler using Hindu-Arabic* **positional notation** system.

- In positional notation, the *n*-digit integer *N* is written as a sequence of digits in the form

$$a_{n-1}\ a_{n-2}\ \ldots\ a_i\ \ldots\ a_1\ a_0$$

  - For example, when N is 278, then $a_2 = 2$, $a_1 = 7$, and $a_0 = 8$.

- The value of this number expressed in positional notation in the base *b* is defined as

$$N = a_{n-1}\times b^{n-1} \ldots + a_1\times b^1 + a_0\times b^0$$

i.e.,

$$N = 2\times10^2 + 7\times10^1 + 8\times10^0 = 200 + 70 + 8 = 278$$

Numbers and Binary Arithmetic

- Positional notation can be extended to express real values by using a *radix point* to separate the integer and fractional part, e.g.,
  - decimal point in base ten arithmetic or
  - binary point in base two arithmetic.
- A real value in decimal arithmetic is written in the form 1234.567.
- To generalize, if we have
  - n digits to the left of the radix point and
  - m digits to the right of the radix point,

  we can write the number as $a_{n-1}\ a_{n-2} \ldots a_i \ldots a_1\ a_0 \,.\, a_{-1}\ a_{-2} \ldots a_{-m}$
- The value of this number expressed in positional notation in the base b is defined as

$$N = a_{n-1}\times b^{n-1} \ldots + a_1\times b^1 + a_0\times b^0 + a_{-1}\times b^{-1} + a_{-2}\times b^{-2} \ldots + a_{-m}\times b^{-m}$$

$$= \sum_{i=-m}^{n-1} a_i\, b^i$$

Warning!

- Any integer number can be *accurately* converted from a base to the other without any error.
- Some fractions that can be represented in base cannot be represented in another base
  - for example 0.1_{10} cannot be *accurately* converted into binary form.

$0.1_{10} = 1 \times 10^{-1}.$

$=$

Binary Arithmetic

- These tables cover the fundamental arithmetic operations.

Produces sum & carry

Produces difference & borrow

Addition	Subtraction	Multiplication
$0 + 0 = 0$	$0 - 0 = 0$	$0 \times 0 = 0$
$0 + 1 = 1$	$0 - 1 = 1$ (borrow 1)	$0 \times 1 = 0$
$1 + 0 = 1$	$1 - 0 = 1$	$1 \times 0 = 0$
$1 + 1 = 0$ (carry 1)	$1 - 1 = 0$	$1 \times 1 = 1$

Binary Arithmetic

- These tables cover the fundamental arithmetic operations.

Produces sum & carry

Produces difference & borrow

Addition	Subtraction	Multiplication
0 + 0 = 0 (carry 0)	0 – 0 = 0 (borrow 0)	0 × 0 = 0
0 + 1 = 1 (carry 0)	0 – 1 = 1 (borrow 1)	0 × 1 = 0
1 + 0 = 1 (carry 0)	1 – 0 = 1 (borrow 0)	1 × 0 = 0
1 + 1 = 0 (carry 1)	1 – 1 = 0 (borrow 0)	1 × 1 = 1

Addition (three bits)	Subtraction (three bits)
0 + 0 + 0 = 0 (carry 0)	**0 – 0** – 0 = 0 (borrow 0)
0 + 0 + 1 = 1 (carry 0)	**0 – 0** – 1 = 1 (borrow 1)
0 + 1 + 0 = 1 (carry 0)	**0 – 1** – 0 = 1 (borrow 1)
0 + 1 + 1 = 0 (carry 1)	**0 – 1** – 1 = 0 (borrow 1)
1 + 0 + 0 = 1 (carry 0)	**1 – 0** – 0 = 1 (borrow 0)
1 + 0 + 1 = 0 (carry 1)	**1 – 0** – 1 = 0 (borrow 0)
1 + 1 + 0 = 0 (carry 1)	**1 – 1** – 0 = 0 (borrow 0)
1 + 1 + 1 = 1 (carry 1)	**1 – 1** – 1 = 1 (borrow 1)

- The digital logic necessary to implement bit-level arithmetic operations is trivial.

Binary Arithmetic

- When you add two binary numbers, you add same position bits together, one column at a time, starting with the least-significant bit.
- Any carry-out is added to the next column on the left.

Example 1	Example 2	Example 3	Example 4
1	11111		111 11
00101010	10011111	00110011	01110011
+01001101	+00000001	+11001100	+01110011
01110111	10100000	11111111	11100110

Addition (three bits)

0 + 0 + 0 = 0 (carry 0)
0 + 0 + 1 = 1 (carry 0)
0 + 1 + 0 = 1 (carry 0)
0 + 1 + 1 = 0 (carry 1)
1 + 0 + 0 = 1 (carry 0)
1 + 0 + 1 = 0 (carry 1)
1 + 1 + 0 = 0 (carry 1)
1 + 1 + 1 = 1 (carry 1)

Addition

0 + 0 = 0 (carry 0)
0 + 1 = 1 (carry 0)
1 + 0 = 1 (carry 0)
1 + 1 = 0 (carry 1)

Binary Arithmetic

- When subtracting binary numbers, you have to remember that *0 – 1 results in a **difference** 1 and a **borrow from the column on the left**.*

The borrow is not correct in the book.

Example 1	Example 2	Example 3	Example 4	Example 5
	1	1	1 1111	
01101001	10011111	10111011	10110000	01100011
-01001001	-01000001	-10000100	-01100011	-10110000
00100000	01011110	00110111	01001101	

We have reversed the subtraction (smaller from larger) as we do in conventional mathematic.

```
1   1111
 10110000
-01100011
 01001101

 01100011
-10110000
-01001101
```

Computers do not operate in this way!!

Subtraction

0 – 0 = 0 (borrow 0)
0 – 1 = 1 (borrow 1)
1 – 0 = 1 (borrow 0)
1 – 1 = 0 (borrow 0)

Subtraction (three bits)

0 – **0** – 0 = 0 (borrow 0)
0 – **0** – 1 = 1 (borrow 1)
0 – **1** – 0 = 1 (borrow 1)
0 – **1** – 1 = 0 (borrow 1)
1 – **0** – 0 = 1 (borrow 0)
1 – **0** – 1 = 0 (borrow 0)
1 – **1** – 0 = 0 (borrow 0)
1 – **1** – 1 = 1 (borrow 1)

The *multiplicand* and *multiplier* are mixed up in the textbook

Binary Arithmetic

- In multiplication, Multiplicand × Multiplier = Product
- The following demonstrates the multiplication of 01101001_2 (the multiplicand) by 01001001_2 (the multiplier).
- You start with the least-significant bit of the multiplier and test whether it is a **0** or a **1**. If it is a **0**, you write down *n* zeros; if it is a **1**, you write down the multiplicand (this value is called a partial product).
- You then test the next bit of the multiplier to the left and carry out the same operation—in this case you write either *n* zeros or the multiplicand one place to the left (i.e., the partial product is shifted left).
- The process is continued until you have examined each bit of the multiplier in turn.
- Finally you add together the *n* partial products to generate the product of the multiplicand times the multiplier.

Multiplicand	Multiplier	Step	Partial products														
01101001	0100100**1**	1								0	1	1	0	1	0	0	1
01101001	010010**0**1	2							0	0	0	0	0	0	0	0	
01101001	01001**0**01	3						0	0	0	0	0	0	0	0		
01101001	0100**1**001	4					0	1	1	0	1	0	0	1			
01101001	010**0**1001	5				0	0	0	0	0	0	0	0				
01101001	01**0**01001	6			0	0	0	0	0	0	0	0					
01101001	0**1**001001	7		0	1	1	0	1	0	0	1						
01101001	**0**1001001	8	0	0	0	0	0	0	0	0							
		Result	**0**	**0**	**1**	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**1**

Binary Arithmetic

- Note that,
  - A computer does not perform multiplication operations in this way, as this would require ***storing*** the n partial products, followed by the simultaneous addition of n words.

  - A better technique is to add up the partial products as they are formed.

Multiplicand	Multiplier	Step	Partial products														
01101001	0100100**1**	1								0	1	1	0	1	0	0	1
01101001	010010**0**1	2							0	0	0	0	0	0	0	0	
01101001	01001**0**01	3						0	0	0	0	0	0	0	0		
01101001	0100**1**001	4					0	1	1	0	1	0	0	1			
01101001	010**0**1001	5				0	0	0	0	0	0	0	0				
01101001	01**0**01001	6			0	0	0	0	0	0	0	0					
01101001	0**1**001001	7		0	1	1	0	1	0	0	1						
01101001	**0**1001001	8	0	0	0	0	0	0	0	0							
		Result	**0**	**0**	**1**	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**1**

Range, Precision, Accuracy and Errors

Range:

- The variation between the largest and smallest values that can be represented in a given memory location
  - An n bits binary number has a range from 0 to $2^n - 1$.

    For example, a 1 byte has a range from 0 to 255.

- Note that: the number of possible values that can be encoded in an n bits binary number is 2^n

Range, Precision, Accuracy and Errors

Precision:

- The precision of a number is
  a measure of *how well (precise) we can represent the number*;
  - π cannot be exactly represented by a binary or a decimal real number – no matter how many bits we take.
    - If we use 5 decimal digits to represent π (i.e., 3.1415),
      we say that its precision is 1 in 10^5, or
      you can say 5 significant figures
    - If we use 10 decimal digits to represent π (i.e., 3.141592653),
      we say that its precision is 1 in 10^{10}, or
      you can say 10 significant figures

Range, Precision, Accuracy and Errors

Accuracy:

- The difference between *a representation* and *its actual value*
  - If we measure the temperature of a liquid as 51.32° and its actual temperature is 51.34°, the error is 0.02°.
  - The error (*true value – actual value*) is a one way to measure the accuracy.
- It is tempting to mix accuracy and precision.
- Be careful, they are ***not the same***.
  - The temperature of the liquid may be measured as 51.320001° which has a precision of 8 significant figures, but if its actual temperature is 51.34° the error will be 0.019999°.
- What matters to us as computer designers, programmers, and users is
  - how errors *arise*,
  - how they are *controlled*, and
  - how their effects are *minimalized*.