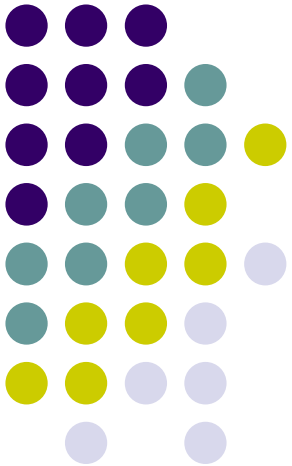
Computer Organization and Architecture

Carl Hamacher, Zvonko Vranesic, Safwat Zaky,

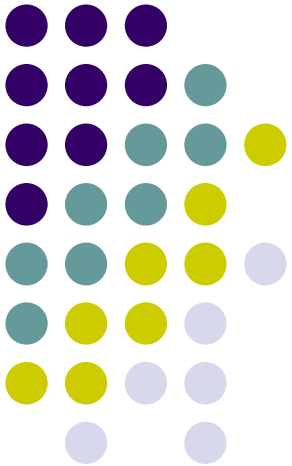
Computer Organization, 5th Edition,

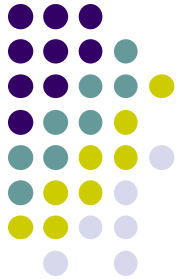
Tata McGraw Hill, 2002.



Machine Instructions and Programs - Part 1

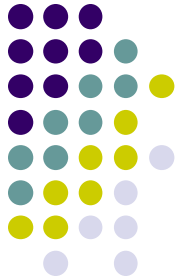
Module 1





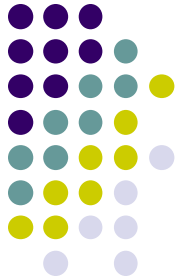
Numbers, Arithmetic Operations, and Characters

Introduction



- Computers are built using logic circuits that operate on information represented by two-valued electrical signals
  - Labelled as 0 and 1
- We define the amount of information represented by such a signal as a *bit* of information, where *bit* stands for *binary digit*.
- The most natural way to represent a number in a computer system is by a string of bits, called a binary number.

Number Representation



- Consider an n-bit vector

$$B = b_{n-1} \ldots \ldots b_1 b_0$$

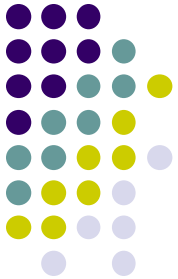
Where $b_i = 0\ or\ 1$ for $0 \leq i \leq n-1$

- This vector can represent unsigned integer values V in the range 0 to $2^n - 1$, where

$$V(B) = b_{n-1} \times 2^{n-1} + \cdots + b_1 \times 2^1 + b_0 \times 2^0$$

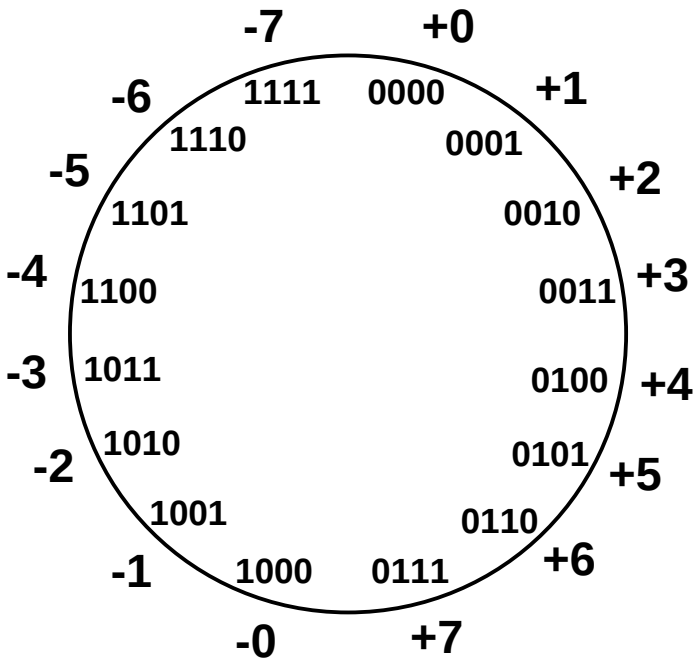
- We obviously need to represent both positive and negative numbers.

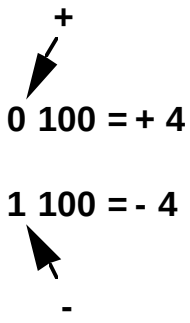
Signed Integer



- 3 major representations:
  - Sign-and-magnitude
  - 1’s complement
  - 2’s complement
- Assumptions:
  - 4-bit machine word
  - 16 different values can be represented
  - Roughly half are positive, half are negative

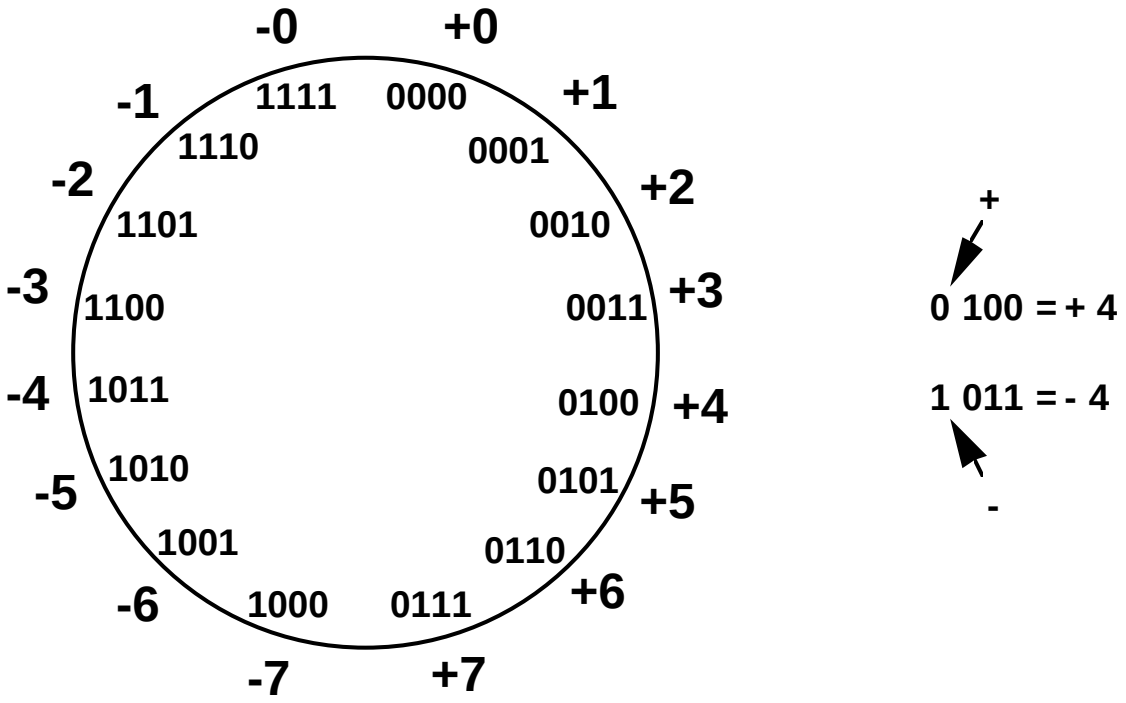
Sign-and-Magnitude Representation





High order bit is sign: 0 = positive (or zero), 1 = negative
Three low order bits is the magnitude: 0 (000) thru 7 (111)
Number range for n bits = +/-2^{n-1} -1
Two representations for 0

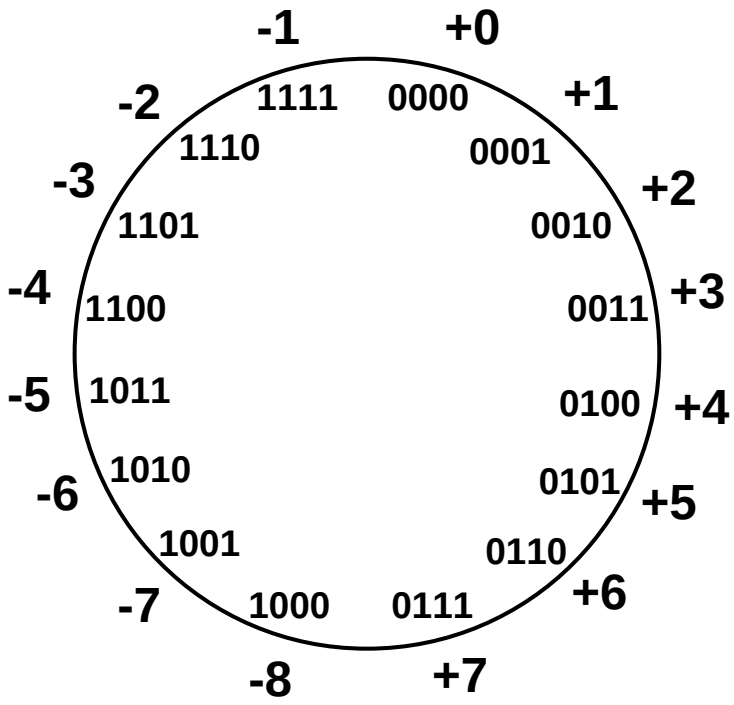
1’s Complement Representation

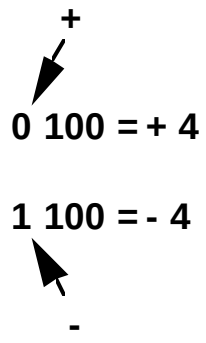


- Subtraction implemented by addition & 1's complement
- Still two representations of 0! This causes some problems
- Some complexities in addition

2’s Complement Representation

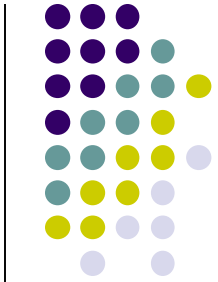
like 1's comp except shifted one position clockwise





- Only one representation for 0
- One more negative number than positive number

Binary, Signed-Integer Representations



Page 28

B	Values represented		
$b_3 b_2 b_1 b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+ 7	+ 7	+ 7
0 1 1 0	+ 6	+ 6	+ 6
0 1 0 1	+ 5	+ 5	+ 5
0 1 0 0	+ 4	+ 4	+ 4
0 0 1 1	+ 3	+ 3	+ 3
0 0 1 0	+ 2	+ 2	+ 2
0 0 0 1	+ 1	+ 1	+ 1
0 0 0 0	+ 0	+ 0	+ 0
1 0 0 0	- 0	- 7	- 8
1 0 0 1	- 1	- 6	- 7
1 0 1 0	- 2	- 5	- 6
1 0 1 1	- 3	- 4	- 5
1 1 0 0	- 4	- 3	- 4
1 1 0 1	- 5	- 2	- 3
1 1 1 0	- 6	- 1	- 2
1 1 1 1	- 7	- 0	- 1

Figure 2.1. Binary, signed-integer representations.

Addition of Positive Numbers

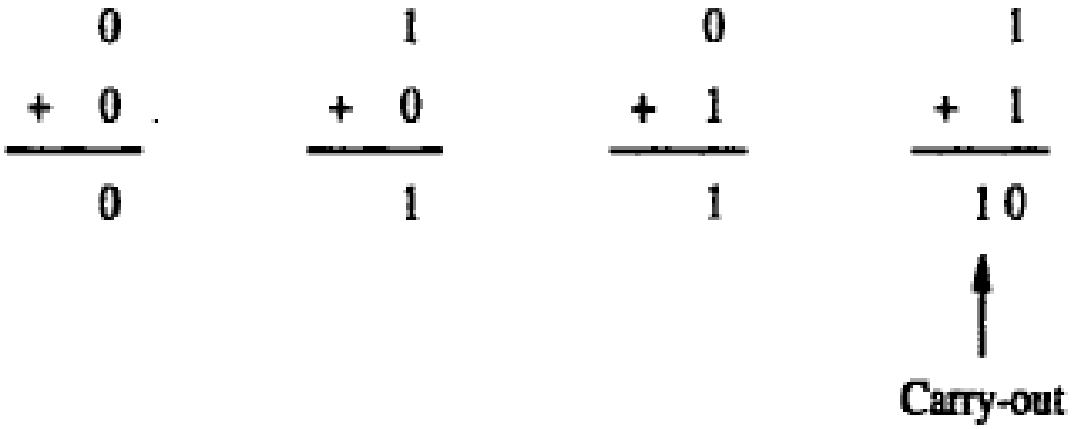
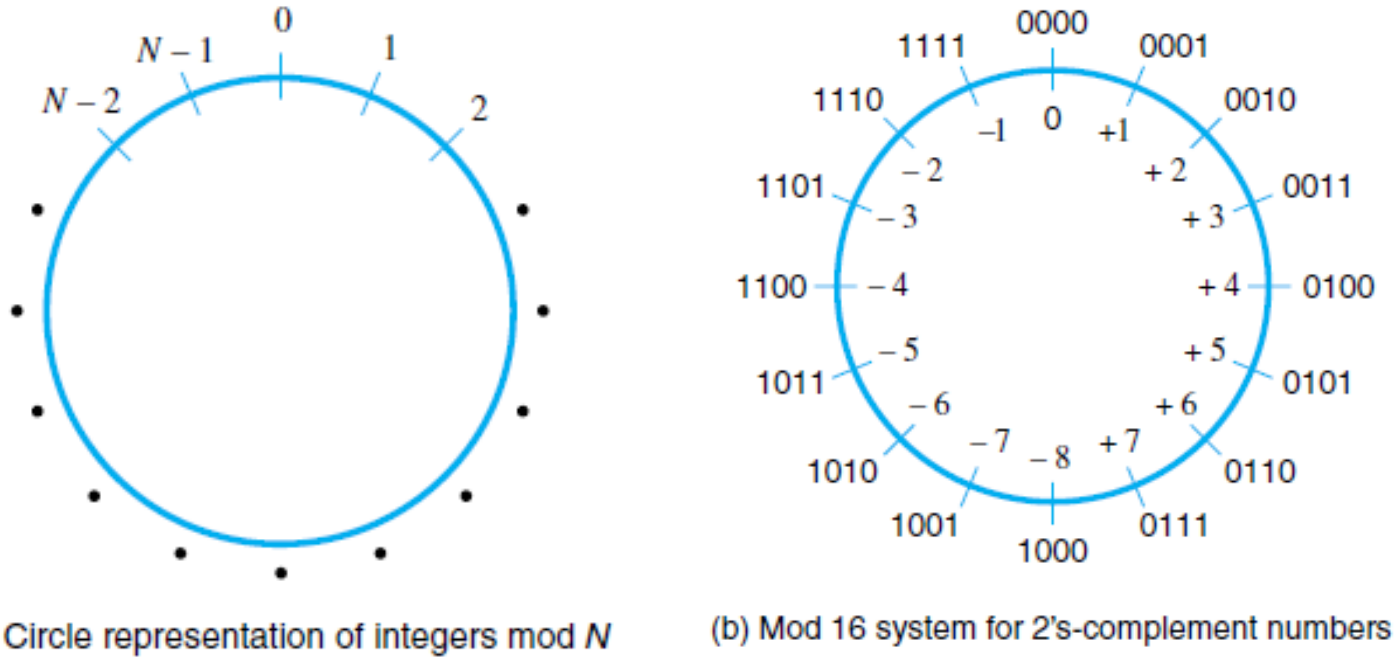


Figure 2.2 Addition of 1-bit numbers.

Addition and Subtraction of Signed Numbers

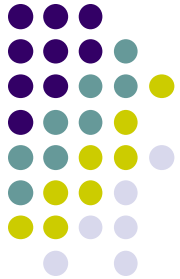


(a) Circle representation of integers mod N

(b) Mod 16 system for 2's-complement numbers

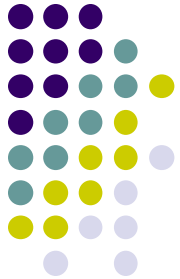
Figure 2.3 Modular number systems and the 2's-complement system.

Addition and Subtraction – 2's Complement



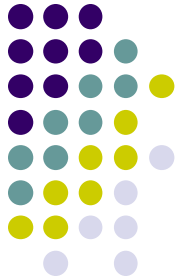
- To add two numbers, add their n-bit representations, ignoring the carry-out signal from the most significant bit (MSB) position. The sum will be the algebraically correct value in the 2's complement representation as long as the answer is in the range -2^{n-1} through $+2^{n-1}-1$.

Addition and Subtraction – 2's Complement..



- To subtract two numbers X and Y, that is, to perform $X - Y$, form the 2's complement of Y and then add it to X. Again, the result will be the algebraically correct value in the 2's complement representation system if the answer is in the range -2^{n-1} through $+ 2^{n-1} - 1$.

Examples



If carry-in to the high order bit = carry-out then ignore carry

4	0100	-4	1100
+ 3	0011	+ (-3)	1101
7	0111	-7	11001

if carry-in differs from carry-out then overflow

4	0100	-4	1100
- 3	1101	+ 3	0011
1	10001	-1	1111

Simpler addition scheme makes twos complement the most common choice for integer number systems within digital systems

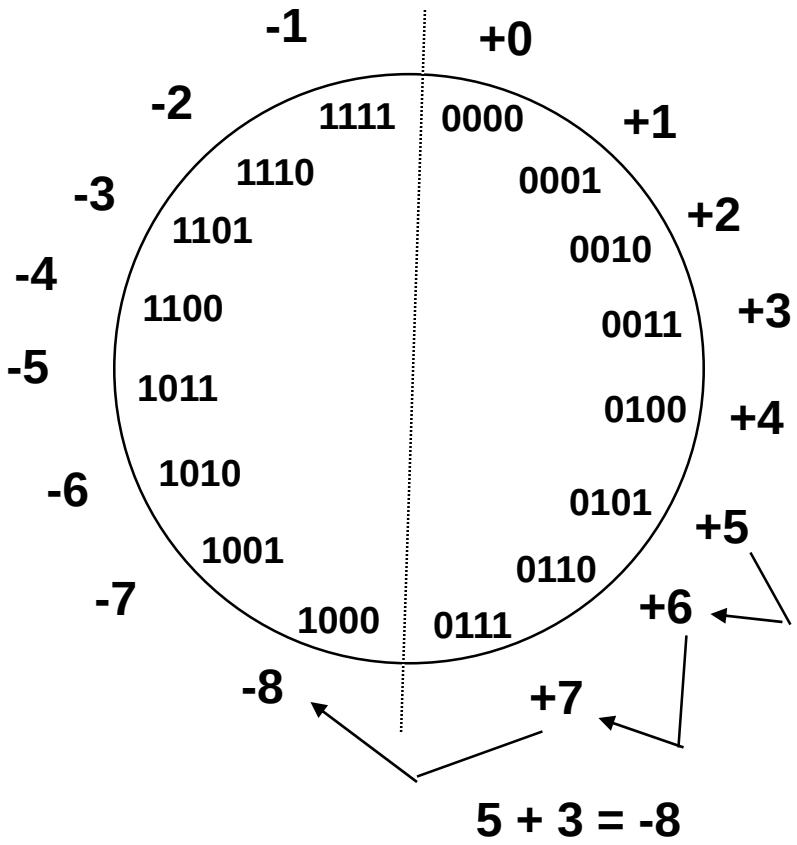
Examples

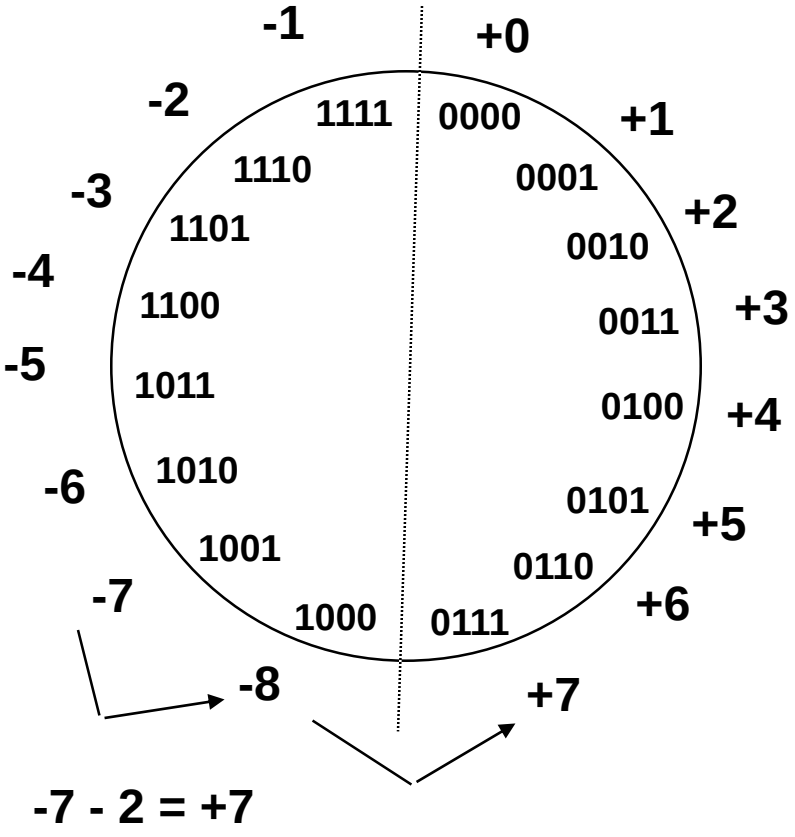
Page 31

(a)	0 0 1 0	(+2)		(b)	0 1 0 0	(+4)
	+ 0 0 1 1	(+3)			+ 1 0 1 0	(- 6)
	0 1 0 1	(+5)			1 1 1 0	(- 2)
(c)	1 0 1 1	(- 5)		(d)	0 1 1 1	(+7)
	+ 1 1 1 0	(- 2)			+ 1 1 0 1	(- 3)
	1 0 0 1	(- 7)			0 1 0 0	(+4)
(e)	1 1 0 1	(- 3)	⇒		1 1 0 1	
	- 1 0 0 1	(- 7)			+ 0 1 1 1	
					0 1 0 0	(+4)
(f)	0 0 1 0	(+2)	⇒		0 0 1 0	
	- 0 1 0 0	(+4)			+ 1 1 0 0	
					1 1 1 0	(- 2)
(g)	0 1 1 0	(+6)	⇒		0 1 1 0	
	- 0 0 1 1	(+3)			+ 1 1 0 1	
					0 0 1 1	(+3)
(h)	1 0 0 1	(- 7)	⇒		1 0 0 1	
	- 1 0 1 1	(- 5)			+ 0 1 0 1	
					1 1 1 0	(- 2)
(i)	1 0 0 1	(- 7)	⇒		1 0 0 1	
	- 0 0 0 1	(+1)			+ 1 1 1 1	
					1 0 0 0	(- 8)
(j)	0 0 1 0	(+2)	⇒		0 0 1 0	
	- 1 1 0 1	(- 3)			+ 0 0 1 1	
					0 1 0 1	(+ 5)

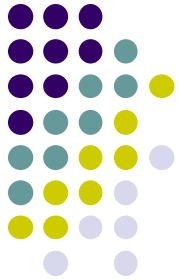
Figure 2.4. 2's-complement Add and Subtract operations.

Overflow - Add two positive numbers to get a negative number or two negative numbers to get a positive number





Overflow Conditions



```
      0 1 1 1
  5     0 1 0 1
  3     0 0 1 1
 -8     1 0 0 0
```

Overflow

```
      1 0 0 0
 -7     1 0 0 1
 -2     1 1 0 0
  7   1 0 1 1 1
```

Overflow

```
      0 0 0 0
  5     0 1 0 1
  2     0 0 1 0
  7     0 1 1 1
```

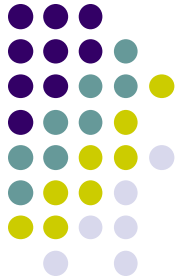
No overflow

```
      1 1 1 1
 -3     1 1 0 1
 -5     1 0 1 1
 -8   1 1 0 0 0
```

No overflow

Overflow when carry-in to the high-order bit does not equal carry out

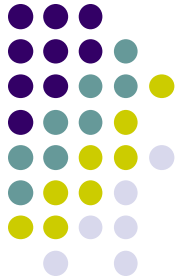
Characters



- In addition to numbers, computers must be able to handle nonnumeric text information consisting of characters.
- Characters can be letters of the alphabet, decimal digits, punctuation marks, and so on.
- They are represented by codes that are usually eight bits long.
  - American Standards Committee on Information Interchange (ASCII) code is widely used.

ASCII Table

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
(sp)	32	0040	0x20	@	64	0100	0x40	`	96	0140	0x60
!	33	0041	0x21	A	65	0101	0x41	a	97	0141	0x61
"	34	0042	0x22	B	66	0102	0x42	b	98	0142	0x62
#	35	0043	0x23	C	67	0103	0x43	c	99	0143	0x63
$	36	0044	0x24	D	68	0104	0x44	d	100	0144	0x64
%	37	0045	0x25	E	69	0105	0x45	e	101	0145	0x65
&	38	0046	0x26	F	70	0106	0x46	f	102	0146	0x66
'	39	0047	0x27	G	71	0107	0x47	g	103	0147	0x67
(	40	0050	0x28	H	72	0110	0x48	h	104	0150	0x68
)	41	0051	0x29	I	73	0111	0x49	i	105	0151	0x69
*	42	0052	0x2a	J	74	0112	0x4a	j	106	0152	0x6a
+	43	0053	0x2b	K	75	0113	0x4b	k	107	0153	0x6b
,	44	0054	0x2c	L	76	0114	0x4c	l	108	0154	0x6c
-	45	0055	0x2d	M	77	0115	0x4d	m	109	0155	0x6d
.	46	0056	0x2e	N	78	0116	0x4e	n	110	0156	0x6e
/	47	0057	0x2f	O	79	0117	0x4f	o	111	0157	0x6f
0	48	0060	0x30	P	80	0120	0x50	p	112	0160	0x70
1	49	0061	0x31	Q	81	0121	0x51	q	113	0161	0x71
2	50	0062	0x32	R	82	0122	0x52	r	114	0162	0x72
3	51	0063	0x33	S	83	0123	0x53	s	115	0163	0x73
4	52	0064	0x34	T	84	0124	0x54	t	116	0164	0x74
5	53	0065	0x35	U	85	0125	0x55	u	117	0165	0x75
6	54	0066	0x36	V	86	0126	0x56	v	118	0166	0x76
7	55	0067	0x37	W	87	0127	0x57	w	119	0167	0x77
8	56	0070	0x38	X	88	0130	0x58	x	120	0170	0x78
9	57	0071	0x39	Y	89	0131	0x59	y	121	0171	0x79
:	58	0072	0x3a	Z	90	0132	0x5a	z	122	0172	0x7a
;	59	0073	0x3b	[	91	0133	0x5b	{	123	0173	0x7b
<	60	0074	0x3c	\	92	0134	0x5c	|	124	0174	0x7c
=	61	0075	0x3d	]	93	0135	0x5d	}	125	0175	0x7d
>	62	0076	0x3e	^	94	0136	0x5e	~	126	0176	0x7e
?	63	0077	0x3f	_	95	0137	0x5f				



Memory Locations, and Addresses

Memory Locations and Addresses

- Memory consists of many millions of storage cells, each of which can store 1 bit.
- Data is usually accessed in *n*-bit groups called *words*.
  - *n* is called word length.

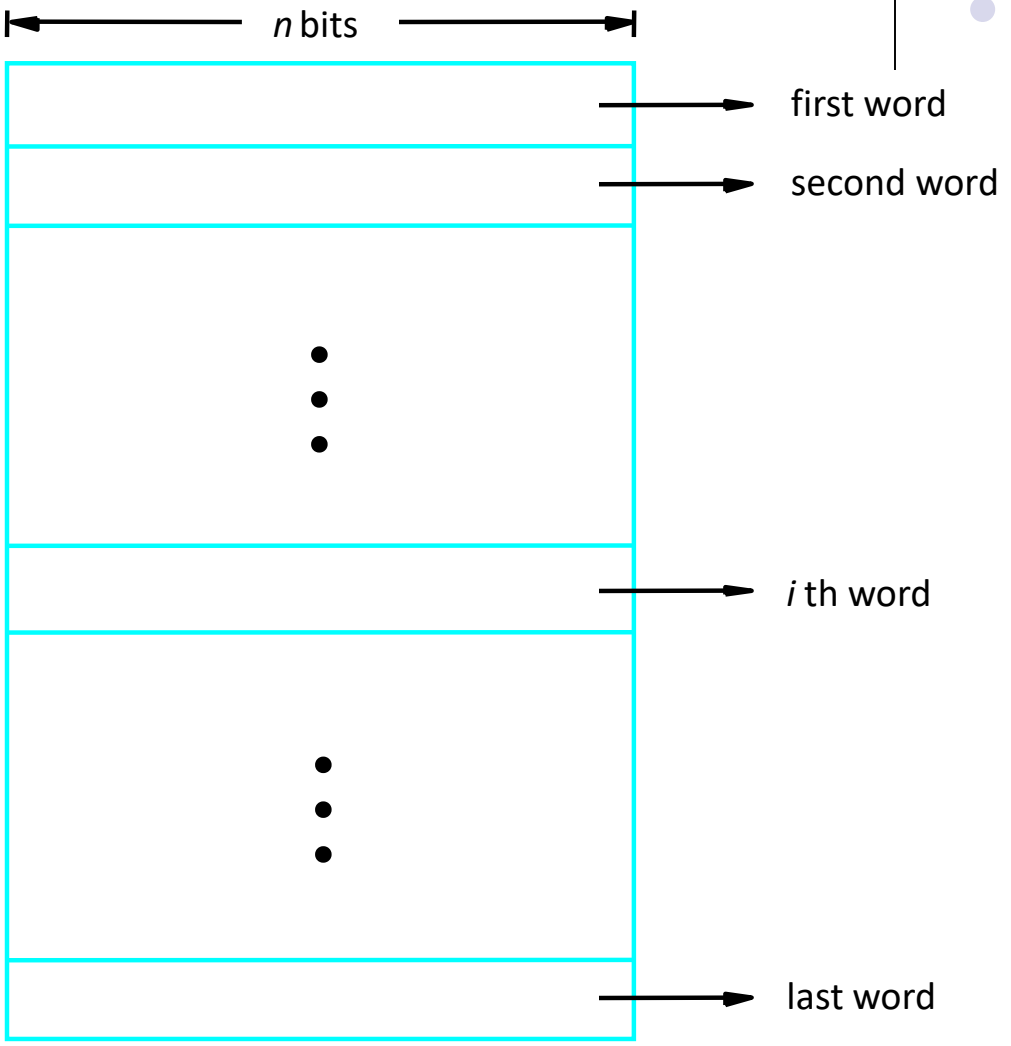
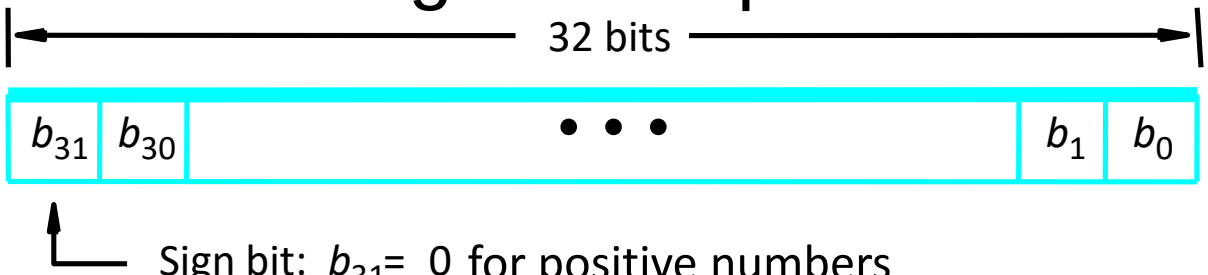


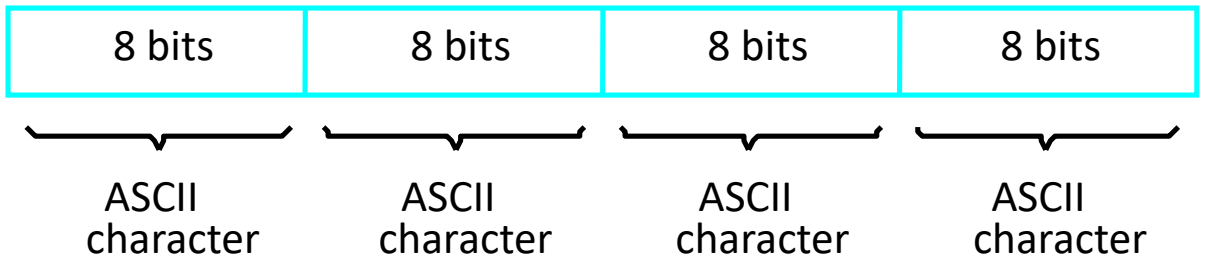
Figure 2.5. Memory words.

Memory Locations and Addresses..

- 32-bit word length example



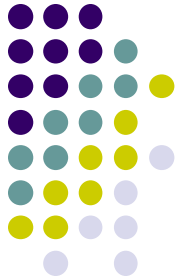
(a) A signed integer



(b) Four characters

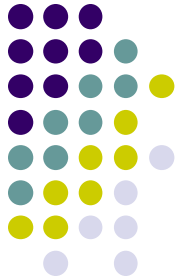
Figure 2.6 Examples of encoded information in a 32-bit word.

Memory Locations and Addresses..



- To retrieve information from memory, either for one word or one byte (8-bit), addresses for each location are needed.
- A *k*-bit address memory has 2^k memory locations, namely 0 – 2^k-1, called memory space.
- 24-bit memory: 2^{24} = 16,777,216 = 16M (1M=2^{20})
- 32-bit memory: 2^{32} = 4G (1G=2^{30})
- 1K(kilo)=2^{10}
- 1T(tera)=2^{40}

Byte Addressability



- A byte is always 8 bits, but the word length typically ranges from 16 to 64 bits.
- It is impractical to assign distinct addresses to individual bit locations in the memory.
- The most practical assignment is to have successive addresses refer to successive byte locations in the memory – byte-addressable memory.
- Byte locations have addresses 0, 1, 2, … If word length is 32 bits, they successive words are located at addresses 0, 4, 8,…

Big-Endian and Little-Endian Assignments

Big-Endian: lower byte addresses are used for the most significant bytes of the word

Little-Endian: opposite ordering. lower byte addresses are used for the less significant bytes of the word

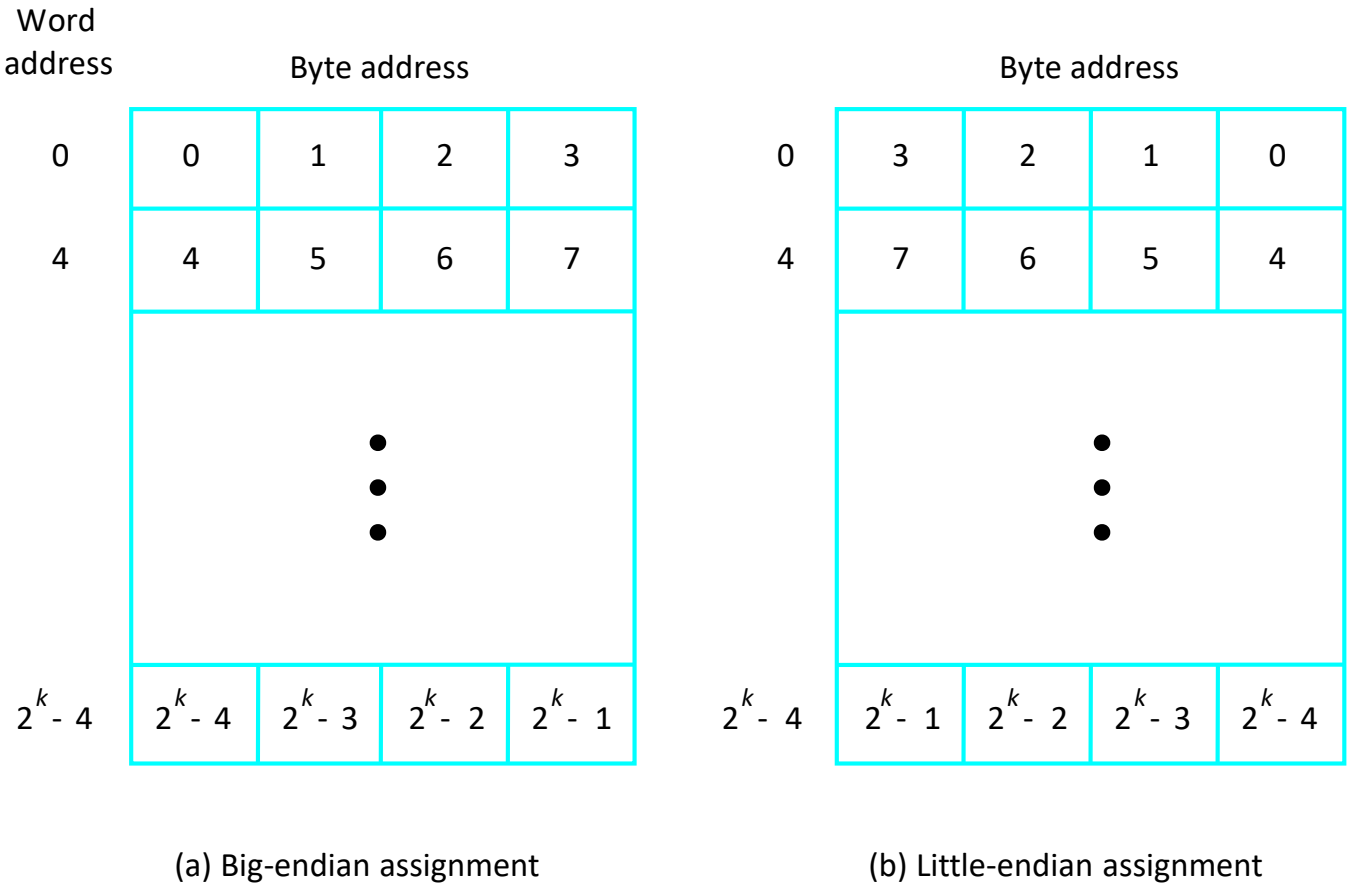
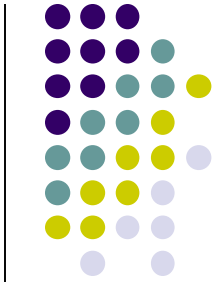


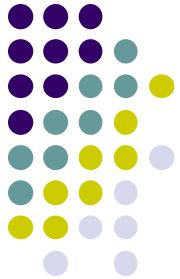
Figure 2.7. Byte and word addressing.

Word Alignment



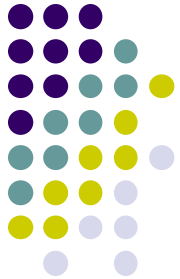
- Address ordering of bytes
- Word alignment
  - Words are said to be aligned in memory if they begin at a byte address. that is a multiple of the num of bytes in a word.
    - 16-bit word: word addresses: 0, 2, 4,….
    - 32-bit word: word addresses: 0, 4, 8,….
    - 64-bit word: word addresses: 0, 8,16,….

Accessing numbers, characters, and character strings



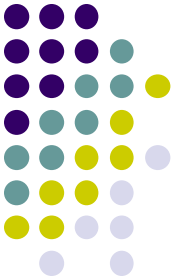
- A number usually occupies one word.
  - It can be accessed in the memory by specifying its word address.
- Similarly, individual characters can be accessed by their byte address.
- It is necessary to handle character strings of variable length.
  - The beginning of the string is indicated by giving the address of the byte containing its first character.
  - Successive byte locations contain successive characters of the string.

Accessing numbers, characters, and character strings..



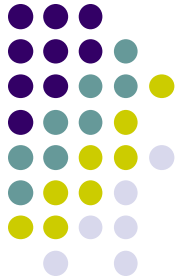
- There are two ways to indicate the length of the string.
  - A special control character with the meaning "end of string" can be used as the last character in the string.
  - Or a separate memory word location or processor register can contain a number indicating the length of the string in bytes.

Memory Operations



- Load (or Read or Fetch)
  - Copy the content. The memory content doesn't change.
  - Address – Load
  - Registers can be used
- Store (or Write)
  - Overwrite the content in memory
  - Address and Data – Store
  - Registers can be used

Floating-Point Numbers and Operations



- In the 2's complement system, the signed value F, represented by the n-bit binary fraction

$$B = b_0.\, b_{-1} b_{-2} \ldots b_{-(n-1)}$$

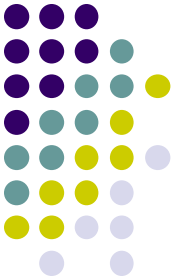
is given by

$$F(B) = -b_0 \times 2^0 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \cdots + b_{-(n-1)} \times 2^{-(n-1)}$$

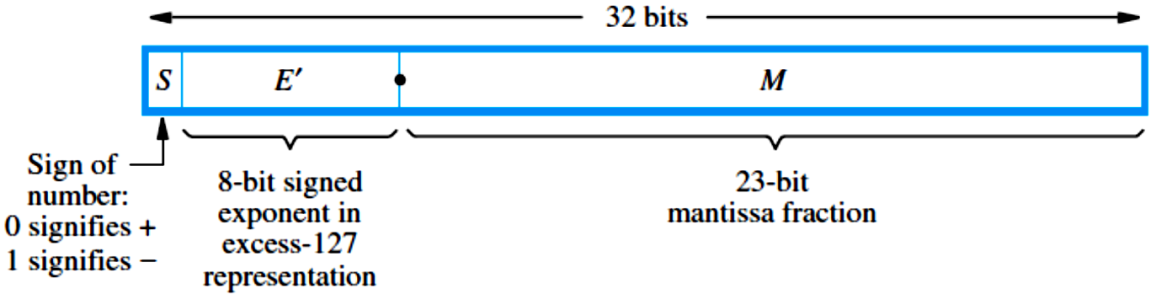
where the range of F is $-1 \leq F \leq 1 - 2^{-(n-1)}$

- For 32-bit format, the range is approximately 0 to $\pm\, 2.15 \times 10^9$ for integers and $\pm 4.55 \times 10^{-10}$ to $\pm\, 1$ for fractions.

IEEE Standard for Floating-Point Numbers

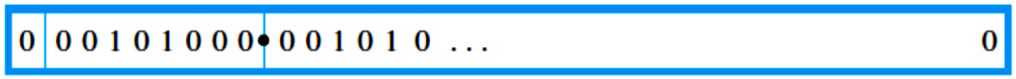


- A binary floating-point number can be represented by
  - A sign for the number
  - Some significant bits
  - A signed scale factor exponent for an implied base of 2
- The basic IEEE format is a 32-bit representation, shown in Figure 6.24*a*
  - Based on 2008 version of IEEE (Institute of Electrical and Electronics Engineers) Standard 754, labelled 754-2008



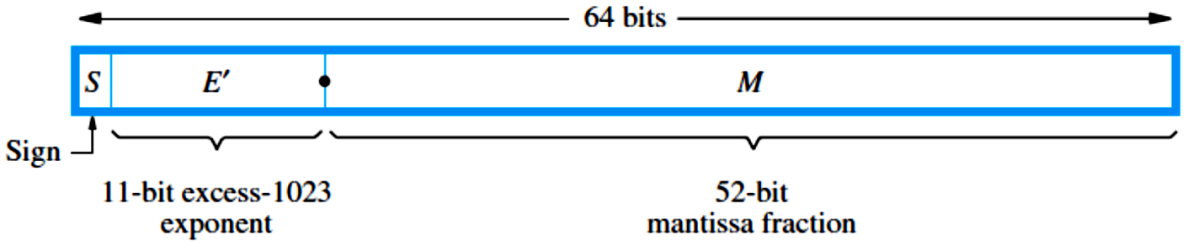
$$\text{Value represented} = \pm 1.M \times 2^{E'-127}$$

(a) Single precision



$$\text{Value represented} = 1.001010\ldots0 \times 2^{-87}$$

(b) Example of a single-precision number

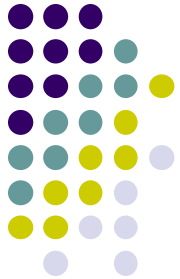


$$\text{Value represented} = \pm 1.M \times 2^{E'-1023}$$

(c) Double precision

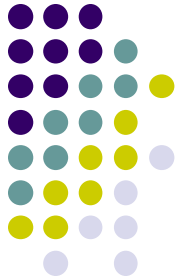
Figure 6.24 IEEE standard floating-point formats.

IEEE Standard for Floating-Point Numbers..



- The leftmost bit represents the sign, S, for the number.
- The next 8 bits, E', represent the signed exponent of the scale factor (with an implied base of 2)
- The remaining 23 bits, M, are the fractional part of the significant bits.

IEEE Standard for Floating-Point Numbers..



- The full 24-bit string, *B*, of significant bits, called the *mantissa*, always has a leading 1, with the binary point immediately to its right.
- Therefore, the mantissa

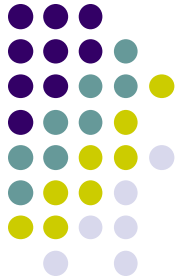
$$B = 1.M = 1.\, b_{-1} b_{-2} \ldots b_{-23}$$

has the value

$$\mathrm{V}(B) = 1 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \cdots + b_{-23} \times 2^{-23}$$

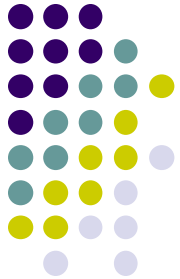
- By convention, when the binary point is placed to the right of the first significant bit, the number is said to be *normalized*.

IEEE Standard for Floating-Point Numbers..



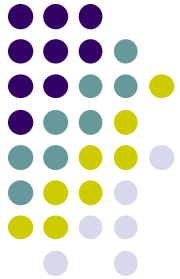
- Instead of the actual signed exponent, E, the value stored in the exponent field is an unsigned integer $E' = E + 127$.
  - This is called the *excess*-127 format.
  - E' is in the range $0 \leq E' \leq 255$.
- The use of the excess-127 representation for exponents simplifies comparison of the relative sizes of two floating-point numbers.

IEEE Standard for Floating-Point Numbers..



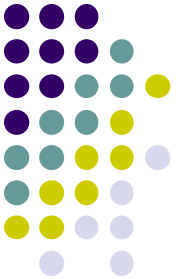
- 32-bit representation – single-precision
  - 8-bit excess-127 exponent E' with range $1 \leq E' \leq 254$ for normal values
  - 0 and 255 indicate special values
  - The actual exponent, E', is in the range $-126 \leq E' \leq 127$ providing scale factors of 2^{-126} to 2^{127} (approximately $10^{\pm 38}$).
  - The 54-bit mantissa provides a precision equivalent to about 7 decimal digits

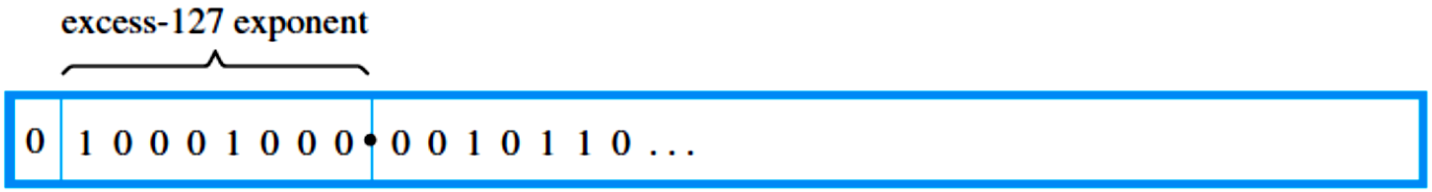
IEEE Standard for Floating-Point Numbers..



- 64-bit representation – double-precision
  - 11-bit excess-1023 exponent E' with range $1 \leq E' \leq 2046$ for normal values
  - 0 and 2047 indicate special values
  - The actual exponent, E', is in the range $-1022 \leq E \leq 1023$, providing scale factors of 2^{-1022} to 2^{1023}(approximately $10^{\pm 308}$).
  - The 53-bit mantissa provides a precision equivalent to about 16 decimal digits

IEEE Standard for Floating-Point Numbers..





(There is no implicit 1 to the left of the binary point.)

$$\text{Value represented} = +0.0010110\ldots \times 2^{9}$$

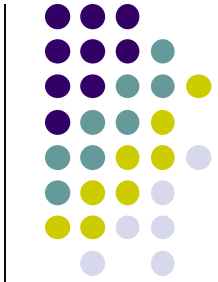
(a) Unnormalized value

$$\text{Value represented} = +1.0110\ldots \times 2^{6}$$

(b) Normalized version

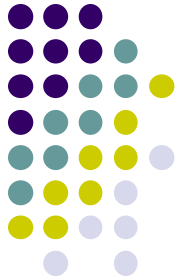
Figure 6.25 Floating-point normalization in IEEE single-precision format.

IEEE Standard for Floating-Point Numbers..

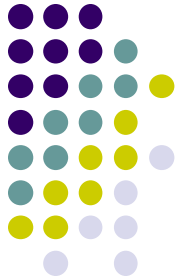


- Two basic aspects of operating with floating-point numbers
- First, if a number is not normalized, it can be put in normalized form by shifting the binary point and adjusting the exponent.
  - Underflow
- Second, as computations proceed, a number that does not fall in the representable range of normal numbers might be generated.
  - Overflow

Special Values

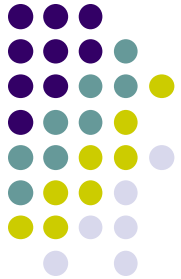


- The end values 0 and 255 of the excess-127 exponent E' are used to represent special values.
  - When $E' = 0$ and $M = 0$, the value 0 is represented.
  - When $E' = 255$ and $M = 0$, the value ∞ is represented.
  - When $E' = 0$ and $M \neq 0$, *denormal* numbers are represented.
  - When $E' = 255$ and $M \neq 0$, the value represented is called *Not a Number* (NaN).



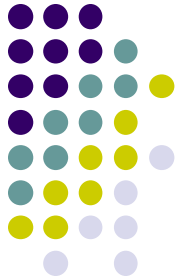
Instructions and Instruction Sequencing

“Must-Perform” Operations



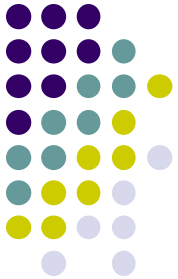
- A computer must have instructions capable of performing four types of operations:
  - Data transfers between the memory and the processor registers
  - Arithmetic and logic operations on data
  - Program sequencing and control
  - I/O transfers

Register Transfer Notation



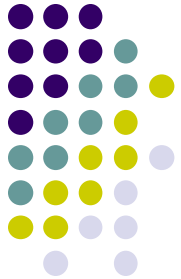
- Identify a location by a symbolic name standing for its hardware binary address
  - Names for addresses of memory location may be LOC, PLACE, A, VAR2
  - Processor register names may be R0, R5
  - I/O register names may be DATAIN, OUTSTATUS
- Contents of a location are denoted by placing square brackets around the name of the location
  - R1←[LOC]
  - R3 ←[R1]+[R2]

Assembly Language Notation



- Represent machine instructions and programs.
- Move LOC, R1 = R1←[LOC]
- Add R1, R2, R3 = R3 ←[R1]+[R2]

CPU Organization

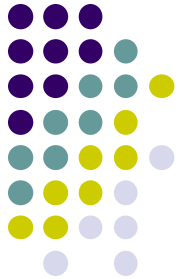


- Single Accumulator
  - Result usually goes to the Accumulator
  - Accumulator has to be saved to memory quite often
- General Register
  - Registers hold operands thus reduce memory traffic
  - Register bookkeeping
- Stack
  - Operands and result are always in the stack

Basic Instruction Types

- Three-Address Instructions
  - Add R1, R2, R3 — R3 ← R1 + R2
- Two-Address Instructions
  - Add R1, R2 — R2 ← R1 + R2
- One-Address Instructions
  - Add M — AC ← AC + [M]
- Zero-Address Instructions
  - Add — TOS ← TOS + (TOS – 1)
- RISC Instructions
  - Lots of registers. Memory is restricted to Load & Store

Basic Instruction Types..



Example: Evaluate (A+B) * (C+D)

- Three-Address

1. Add A, B, R1 ; R1 ← [A] + [B]
2. Add C, D, R2 ; R2 ← [C] + [D]
3. Multiply R1, R2, X ; X ← [R1] * [R2]

Basic Instruction Types..

Example: Evaluate (A+B) * (C+D)

- Two-Address

1.	Move	A, R1	; R1 ← [A]
2.	Add	B, R1	; R1 ← [R1] + [B]
3.	Move	C, R2	; R2 ← [C]
4.	Add	D, R2	; R2 ← [R2] + [D]
5.	Multiply	R1, R2	; R2 ← [R1] * [R2]
6.	Move	R2, X	; X ← [R2]

Basic Instruction Types..

Example: Evaluate (A+B) * (C+D)

- One-Address

1.	Load	A	; AC ← A
2.	Add	B	; AC ← AC + B
3.	Store	T	; T ← AC
4.	Load	C	; AC ← [C]
5.	Add	D	; AC ← AC + [D]
6.	Multiply	T	; AC ← AC * [T]
7.	Store	X	; X ← AC

Basic Instruction Types..

Example: Evaluate (A+B) * (C+D)

- Zero-Address

1.	Push	A	; TOS ← A
2.	Push	B	; TOS ← B
3.	Add		; TOS ← (A + B)
4.	Push	C	; TOS ← C
5.	Push	D	; TOS ← D
6.	Add		; TOS ← (C + D)
7.	Multiply		; TOS ← (C+D)*(A+B)
8.	Pop	X	; X ← TOS

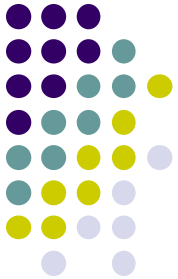
Basic Instruction Types..

Example: Evaluate (A+B) * (C+D)

- RISC

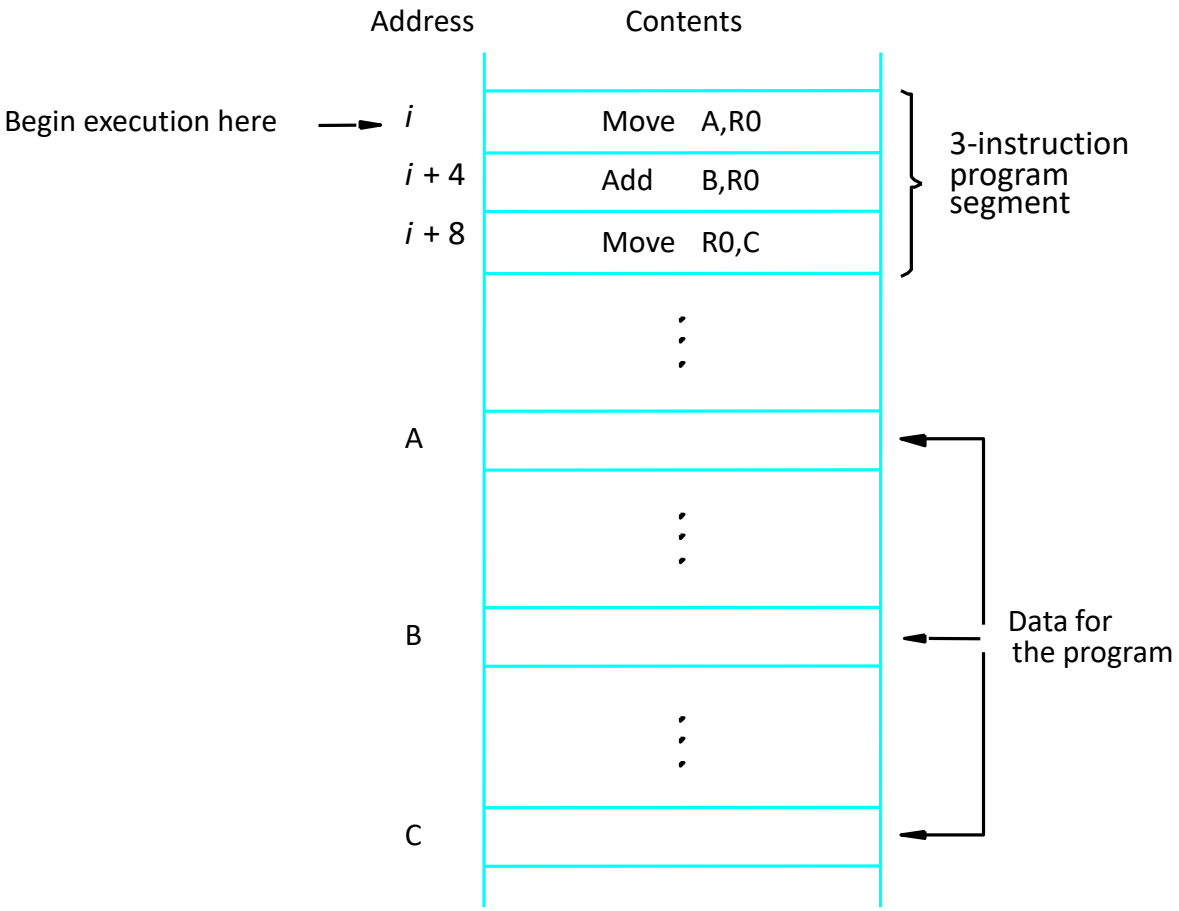
1.	Load	A, R1	; R1 ← [A]
2.	Load	B, R2	; R2 ← [B]
3.	Load	C, R3	; R3 ← [C]
4.	Load	D, R4	; R4 ← [D]
5.	Add	R1, R2	; R2 ← R1 + R2
6.	Add	R3, R4	; R4 ← R3 + R4
7.	Multiply	R2, R4	; R4 ← R2 * R4
8.	Store	R4, X	; X ← R4

Using Registers



- Registers are faster
- Shorter instructions
  - The number of registers is smaller, only few bits are needed to specify the register (e.g. 32 registers need 5 bits)
- Potential speedup
- Minimize the frequency with which data is moved back and forth between the memory and processor registers.

Instruction Execution and Straight-Line Sequencing



Assumptions:
- One memory operand per instruction
- 32-bit word length
- Memory is byte addressable
- Full memory address can be directly specified in a single-word instruction

Two-phase procedure
-Instruction fetch
-Instruction execute

Figure 2.8. A program for C ← [A] + [B].

Branching

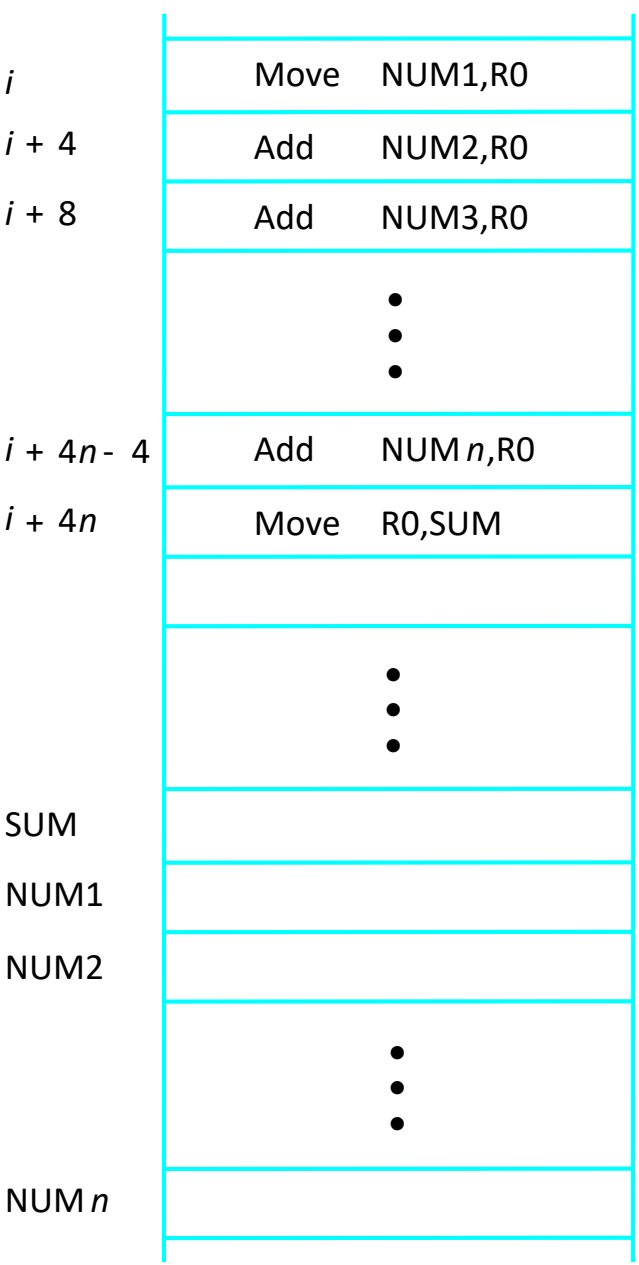


Figure 2.9. A straight-line program for adding *n* numbers.

Branching

Branch target

Conditional branch

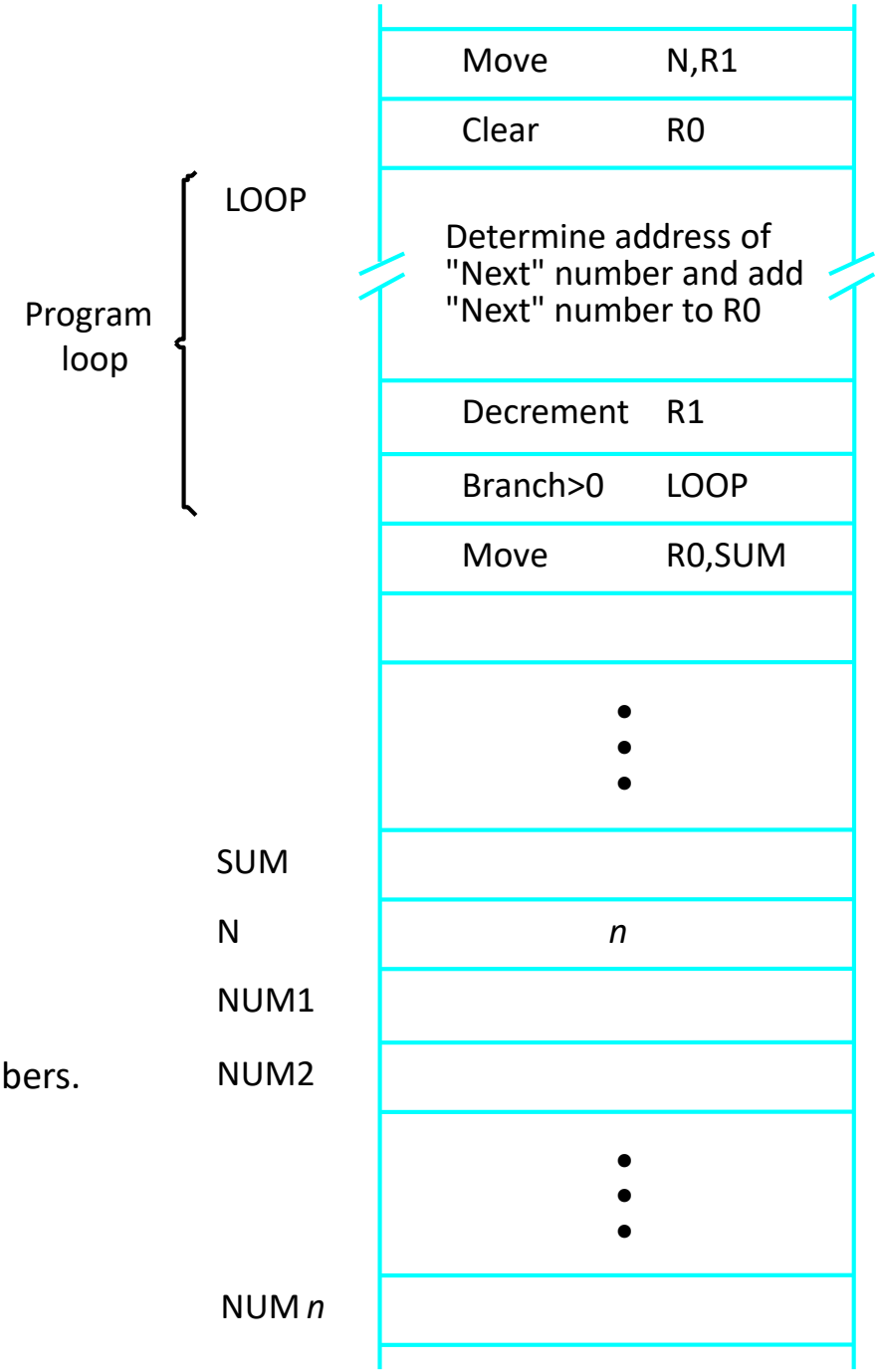
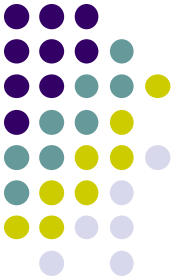


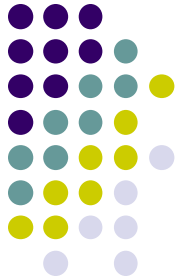
Figure 2.10. Using a loop to add *n* numbers.

Condition Codes



- The processor keeps track of information about the results of various operations for use by subsequent conditional branch instructions.
  - Accomplished by recording the required information in individual bits, often called condition code flags.
- These flags are usually grouped together in a special processor register called the condition code register or status register.

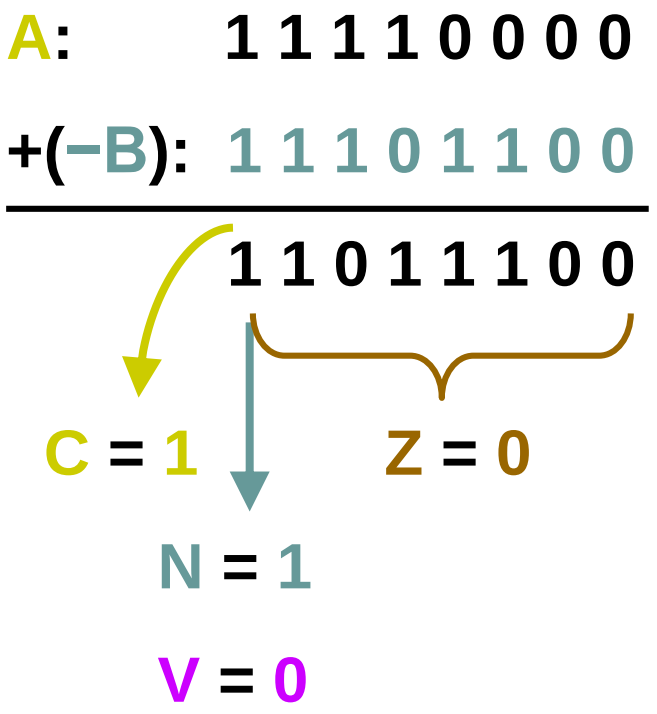
Condition Codes



- Four commonly used flags are
- N (negative)
  - Set to 1 if the result is negative; otherwise, cleared to 0
- Z (zero)
  - Set to 1 if the result is 0; otherwise, cleared to 0
- V (overflow)
  - Set to 1 if arithmetic overflow occurs; otherwise, cleared to 0
- C (carry)
  - Set to 1 if a carry-out results from the operation; otherwise, cleared to 0

Conditional Branch Instructions

- Example:
  - A: 1 1 1 1 0 0 0 0
  - B: 0 0 0 1 0 1 0 0



Status Bits