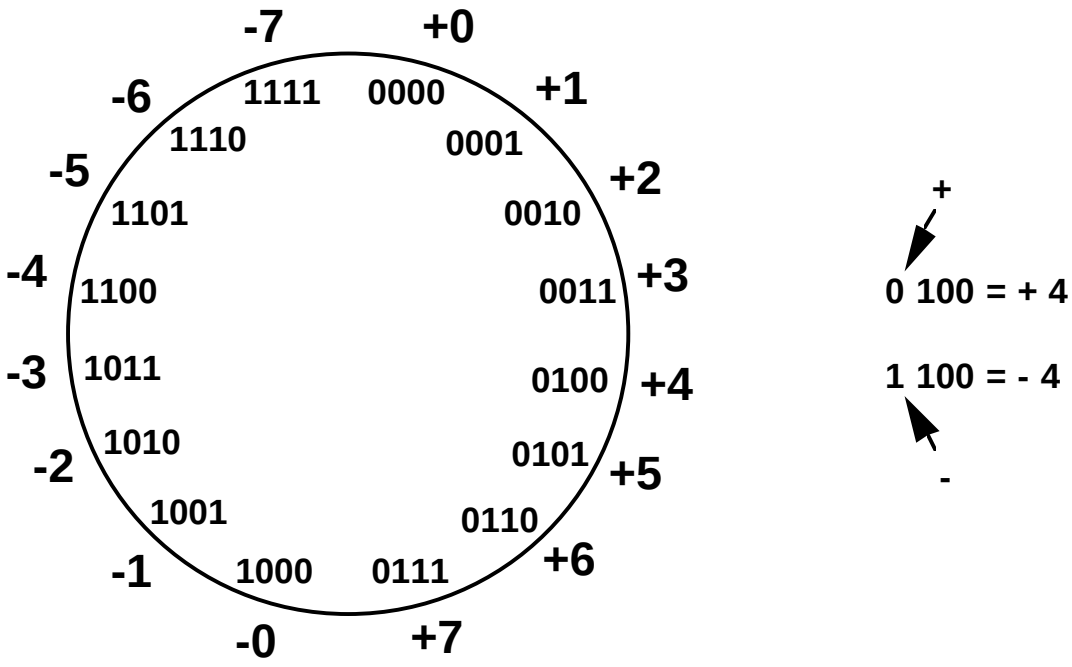
Machine Instructions and Programs

Number, Arithmetic Operations, and Characters

Signed Integer

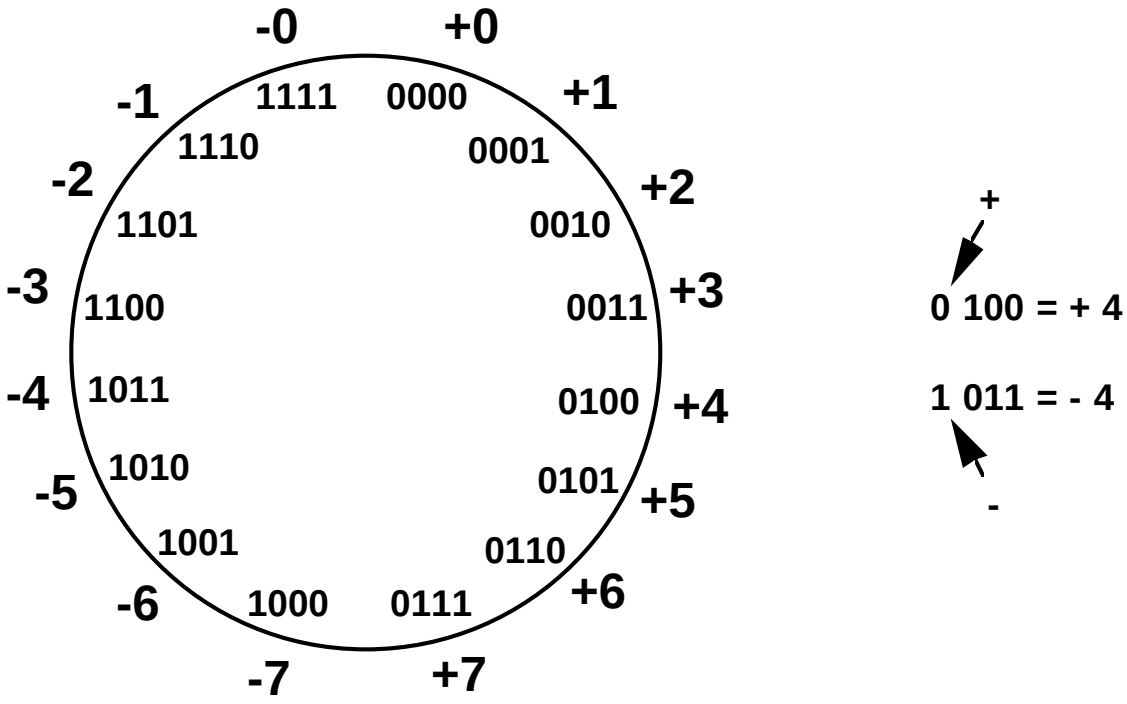
- 3 major representations:
  - Sign-magnitude
  - One's complement
  - Two's complement
- Assumptions:
  - 4-bit machine word
  - 16 different values can be represented
  - Roughly half are positive, half are negative

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

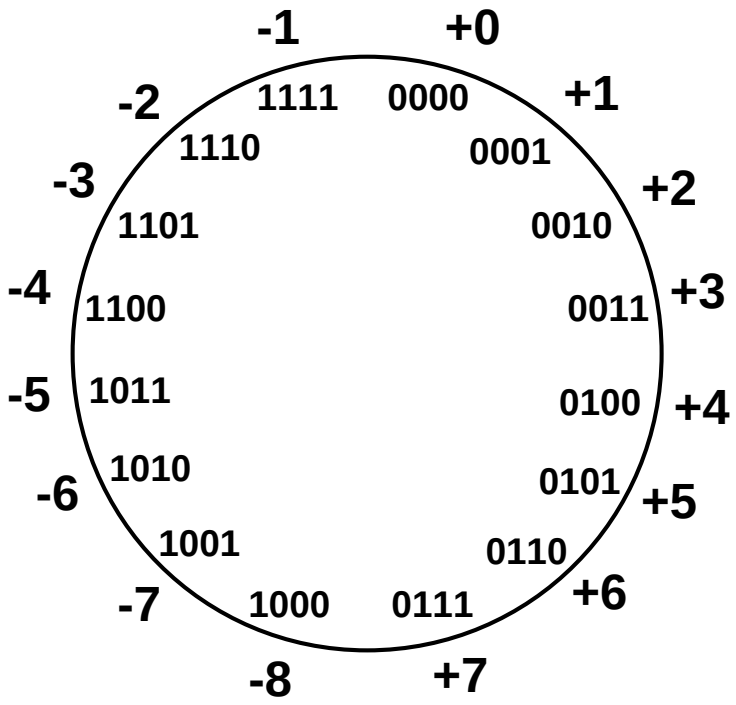
One’s Complement Representation

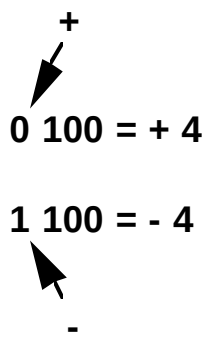


- Subtraction implemented by addition & 1's complement
- Still two representations of 0! This causes some problems
- Some complexities in addition

Two'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

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

Binary, signed-integer representations.

Addition and Subtraction – 2's Complement

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

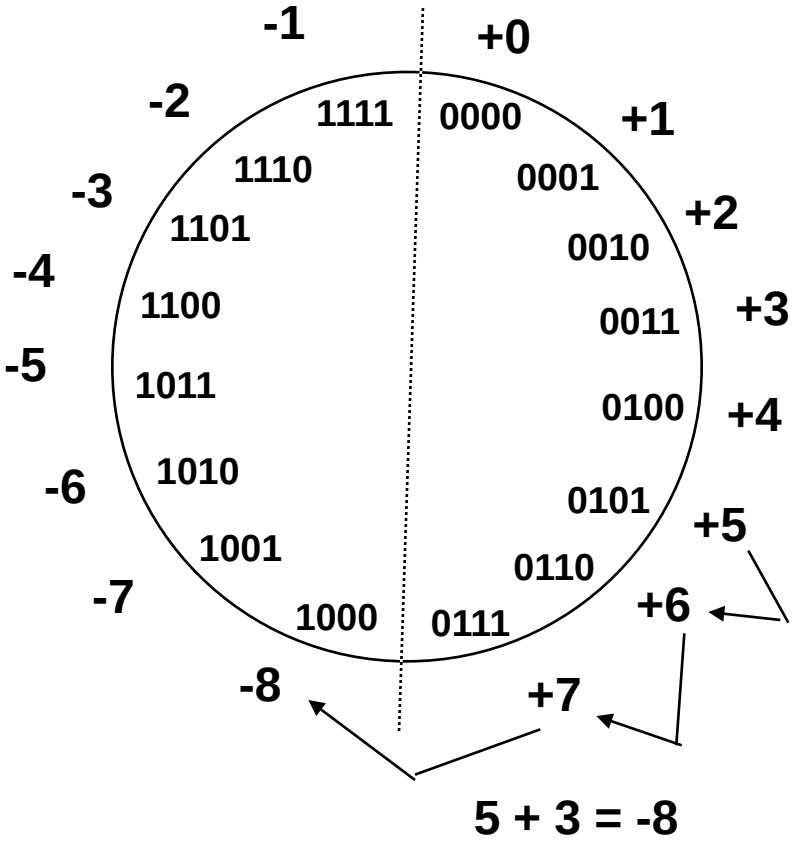
Simpler addition scheme makes twos complement the most common choice for integer number systems within digital systems

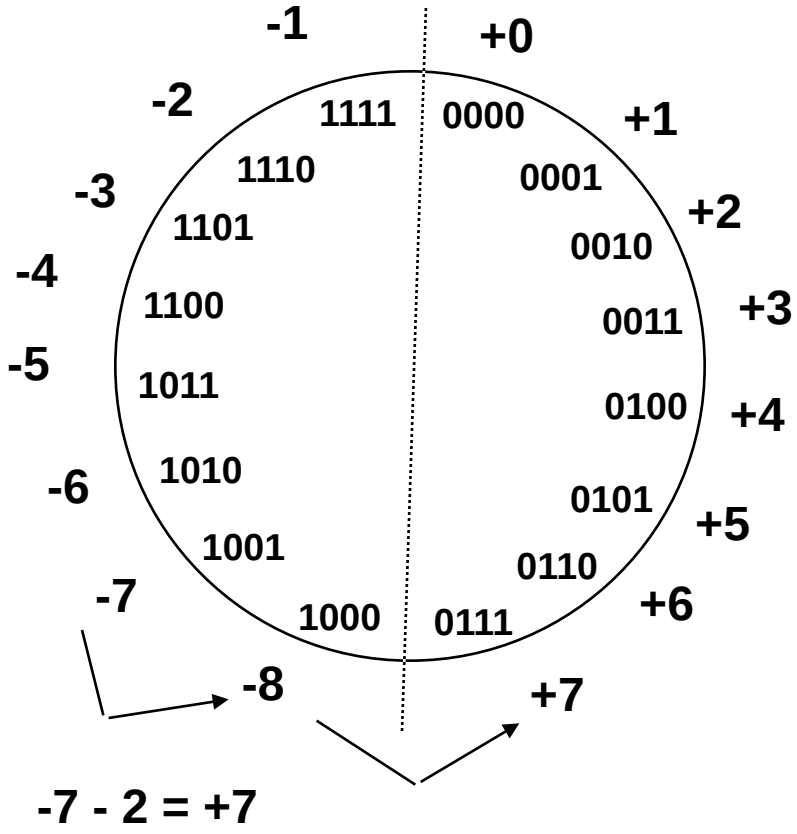
2’s-Complement Add and Subtract Operations

(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)

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

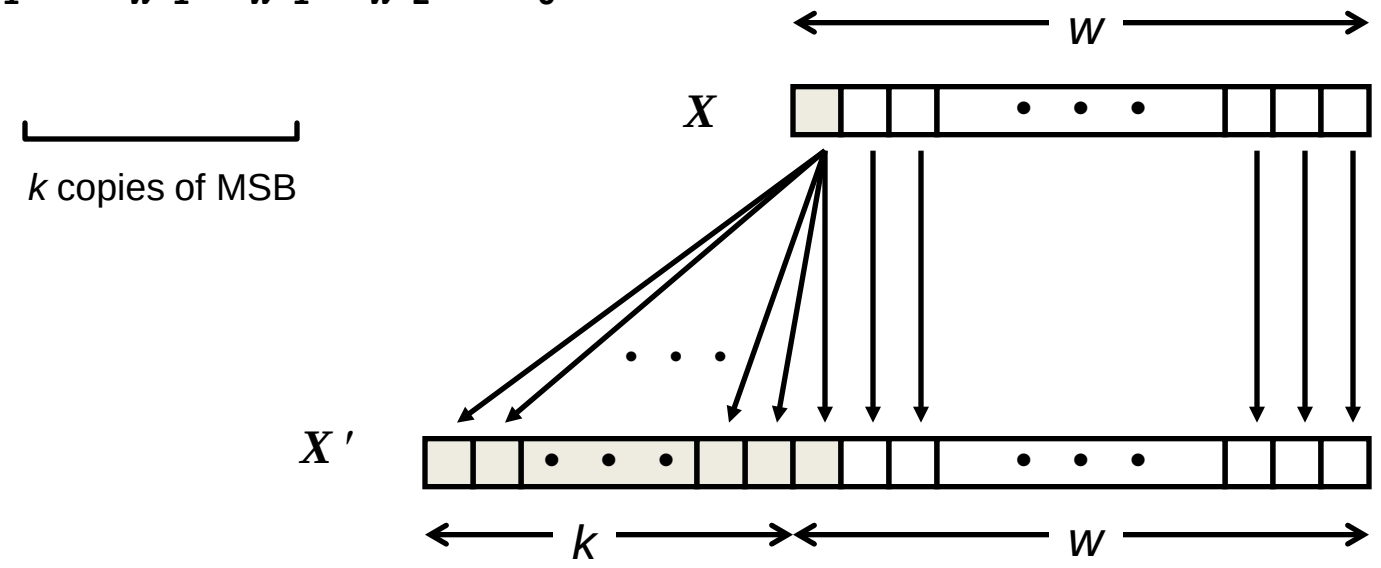
	1 1 1 1	
-3	1 1 0 1	
-5	1 0 1 1	
-8	1 1 0 0 0	

No overflow

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

Sign Extension

- Task:
  - Given *w*-bit signed integer *x*
  - Convert it to *w*+*k*-bit integer with same value
- Rule:
  - Make *k* copies of sign bit:
  - $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0$



Sign Extension Example

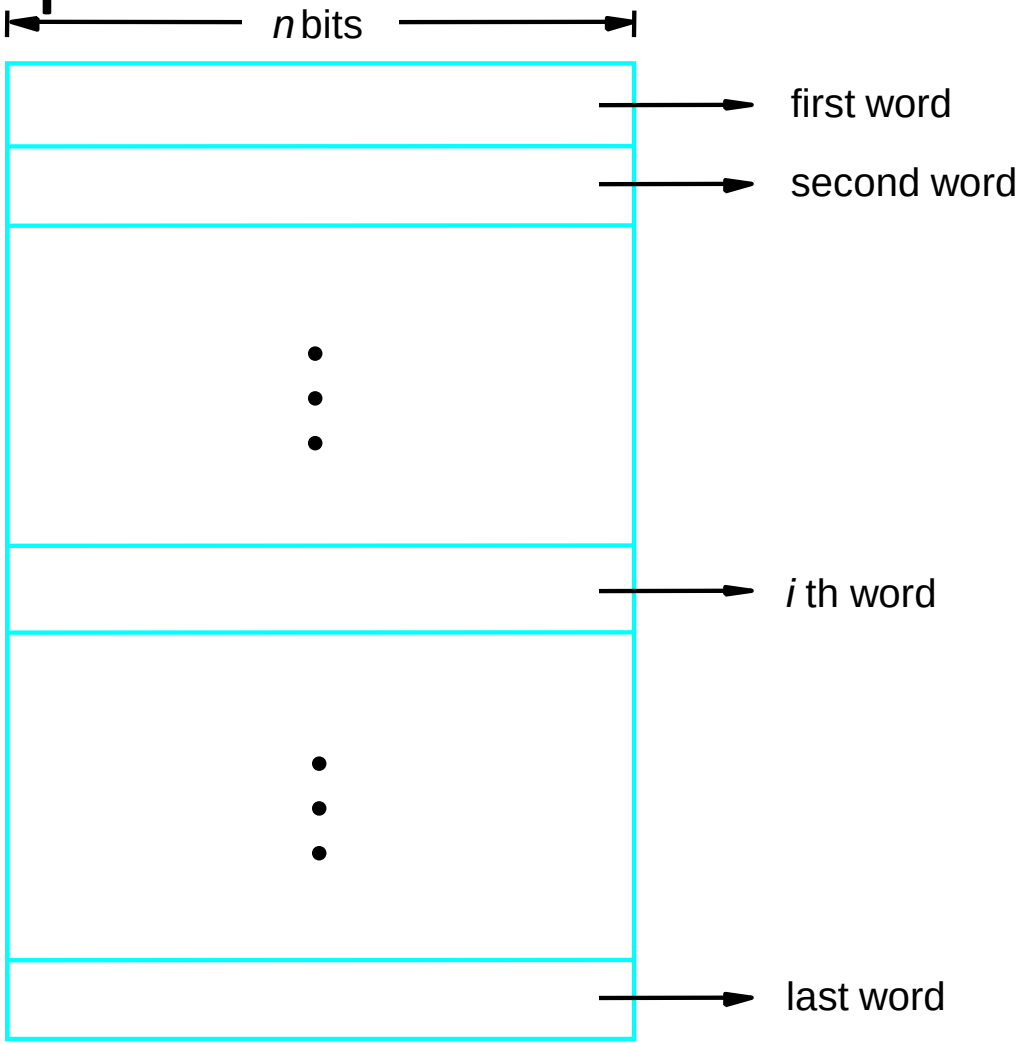
```
short int x =  15213;
int      ix = (int) x;
short int y = -15213;
int      iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 C4 92	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

Memory Locations, Addresses, and Operations

Memory Location, Addresses, and Operation

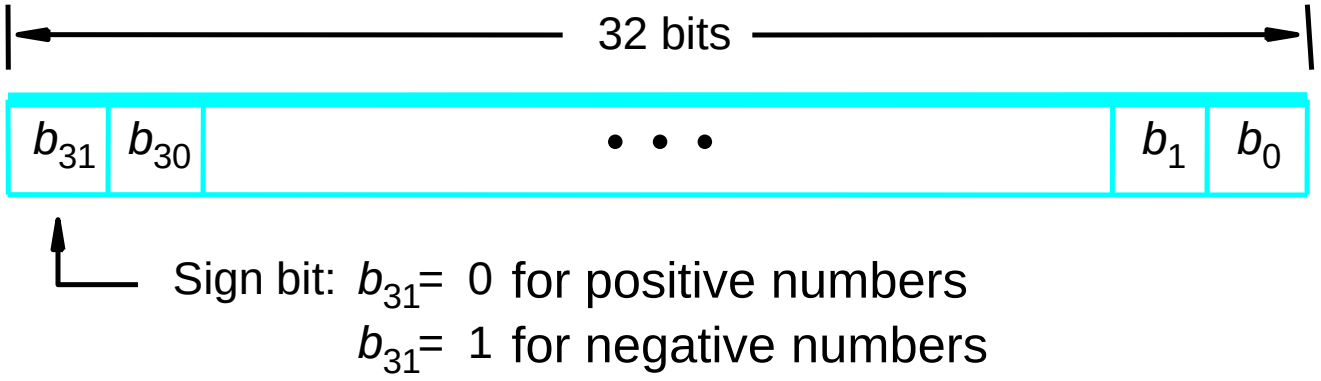
- Memory consists of many millions of storage cells, each of which can store 1 bit.
- Data is usually accessed in *n*-bit groups. *n* is called word length.



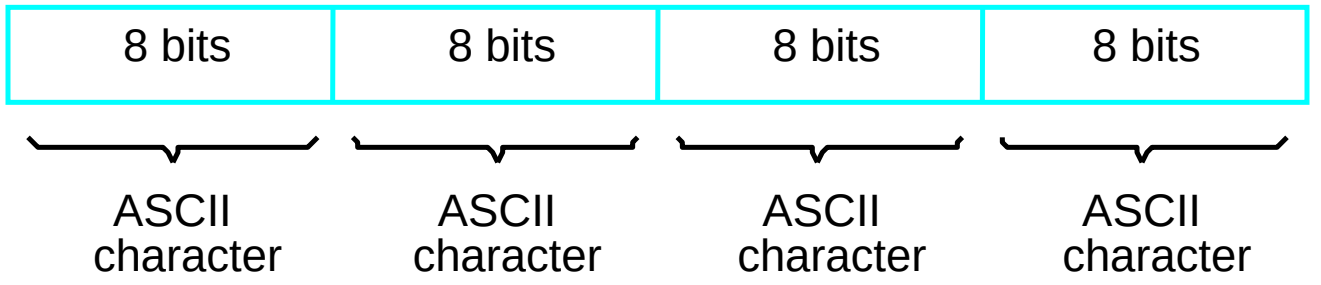
Memory words.

Memory Location, Addresses, and Operation

- 32-bit word length example



(a) A signed integer



(b) Four characters

Memory Location, Addresses, and Operation

- 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/ address 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}

Memory Location, Addresses, and Operation

- 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

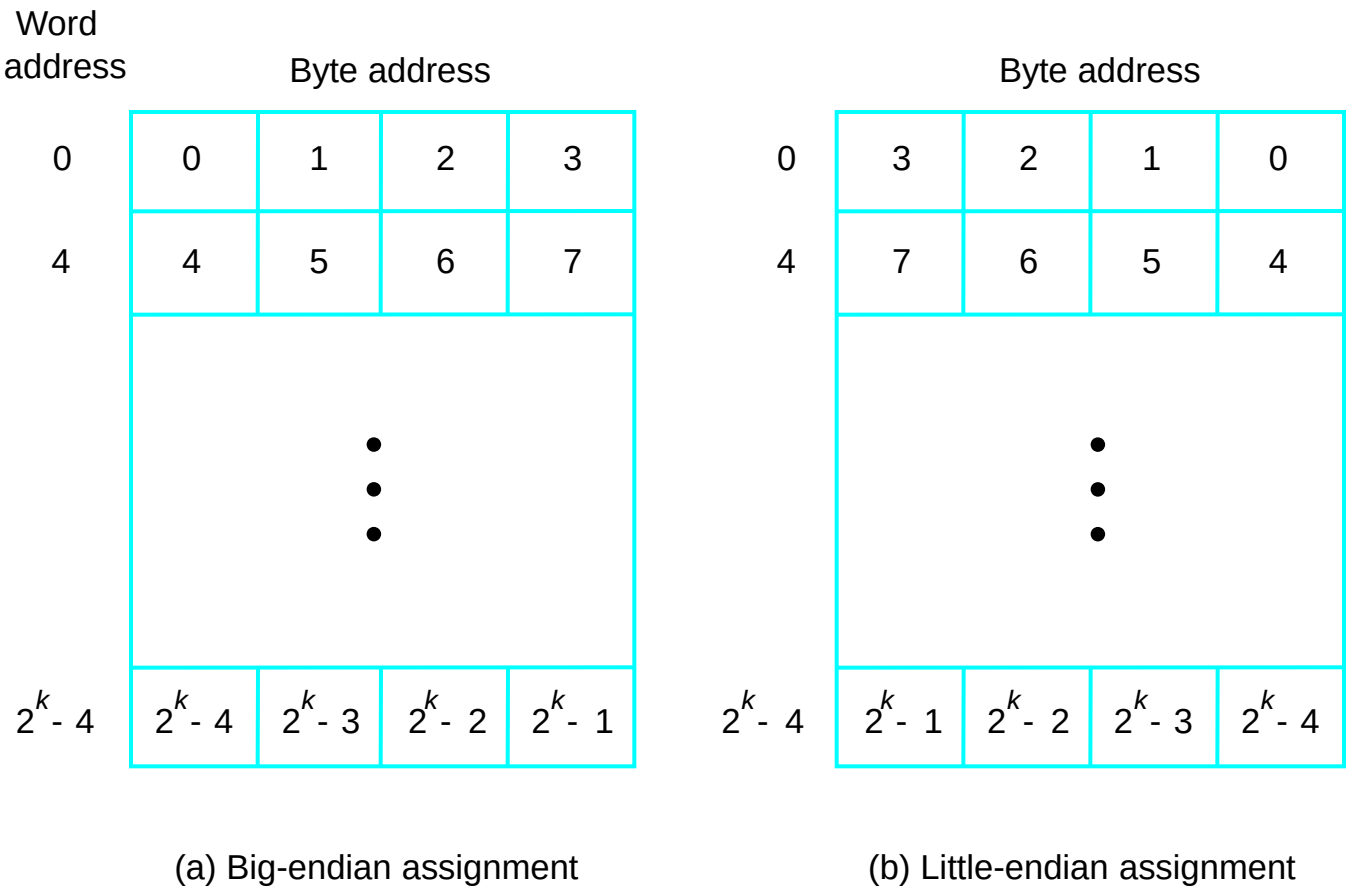


Figure 2.7. Byte and word addressing.

Memory Location, Addresses, and Operation

- Address ordering of bytes
- Word alignment
  - Words are said to be aligned in memory if they begin at a byte addr. 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,….
- Access numbers, characters, and character strings

Memory Operation

- 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

Instruction and Instruction Sequencing

“Must-Perform” 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 (LOC, R0,…)
- Contents of a location are denoted by placing square brackets around the name of the location (R1←[LOC], R3 ←[R1]+[R2])
- Register Transfer Notation (RTN)

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

Instruction Formats

- 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

Instruction Formats

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

- Three-Address

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

Instruction Formats

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

- Two-Address

1.	MOV	A, R1	; R1 ← [A]
2.	ADD	B, R1	; R1 ← [R1] + [B]
3.	MOV	C, R2	; R2 ← [C]
4.	ADD	D, R2	; R2 ← [R2] + [D]
5.	MUL	R2, R1	; R1 ← [R1] * [R2]
6.	MOV	R1, X	; X ← [R1]

Instruction Formats

Example: Evaluate X = (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.	MUL	T	; AC ← [AC] * [T]
7.	STORE	X	; X ← [AC]

Instruction Formats

Example: Evaluate X = (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.	MUL		; TOS ← (C+D)*(A+B)
8.	POP	X	; X ← [TOS]

Instruction Formats

Example: Evaluate X = (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, R1     ; R1 ← [R1] + [R2]
6. ADD    R3, R4, R3     ; R3 ← [R3] + [R4]
7. MUL    R1, R3, R1     ; R1 ← [R1] * [R3]
8. STORE  R1, X          ; X ← [R1]
```

Using Registers

- Registers are faster
- Shorter instructions
  - The number of registers is smaller (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.