DATA REPRESENTATION

Kinds Of Data

- Numbers
  - Integers
    - Unsigned
    - Signed
  - Reals
    - Fixed-Point
    - Floating-Point
  - Binary-Coded Decimal
- Text
  - ASCII Characters
  - Strings
- Other
  - Graphics
  - Images
  - Video
  - Audio

Numbers Are Different!

- Computers use binary (not decimal) numbers (0's and 1's).
  - Requires more digits to represent the same magnitude.
- Computers store and process numbers using a fixed number of digits (“fixed-precision”).
- Computers represent signed numbers using 2's complement instead of sign-plus-magnitude (not our familiar “sign-plus-magnitude”).

Positional Number Systems

- Numeric values are represented by a *sequence* of digit symbols.
- Symbols represent numeric *values*.
  - Symbols are not limited to ‘0’-’9’!
- Each symbol’s contribution to the total value of the number is *weighted* according to its position in the sequence.

Polynomial Evaluation

Whole Numbers (Radix = 10):

$1234_{10} = 1 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$

With Fractional Part (Radix = 10):

$36.72_{10} = 3 \times 10^1 + 6 \times 10^0 + 7 \times 10^{-1} + 2 \times 10^{-2}$

General Case (Radix = R):

$(S_1S_0.S_{-1}S_{-2})_R =$

$S_1 \times R^1 + S_0 \times R^0 + S_{-1} \times R^{-1} + S_{-2} \times R^{-2}$

Converting Radix R to Decimal

$$36.72_8 = 3 \times 8^1 + 6 \times 8^0 + 7 \times 8^{-1} + 2 \times 8^{-2}$$

$$= 24 + 6 + 0.875 + 0.03125$$

$$= 30.90625_{10}$$

Important: Polynomial evaluation doesn’t work if you try to convert in the *other* direction – I.e., from decimal to something else! Why?

Binary to Decimal Conversion

Converting to decimal, so we can use polynomial evaluation:

$$
\begin{aligned}
10110101_2 \\
&= 1\times2^7 + 0\times2^6 + 1\times2^5 + 1\times2^4 + 0\times2^3 \\
&\quad + 1\times2^2 + 0\times2^1 + 1\times2^0 \\
&= 128 + 32 + 16 + 4 + 1 \\
&= 185_{10}
\end{aligned}
$$

Hexadecimal Numbers (Radix = 16)

- The *number* of digit symbols is determined by the radix (e.g., 16)
- The *value* of the digit symbols range from 0 to 15 (0 to R-1).
- The *symbols* are 0-9 followed by A-F.
- Conversion between binary and hex is trivial!
- Use as a shorthand for binary (significantly fewer digits are required for same magnitude).

Memorize This!

Hex	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Hex	Binary
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Binary/Hex Conversions

Hex digits are in one-to-one correspondence with groups of four binary digits:

```
0011 1010 0101 0110 . 1110 0010 1111 1000
  3    A    5    6  .   E    2    F    8
```

- Conversion is a simple table lookup!
- Zero-fill on left and right ends to complete the groups!
- Works because $16 = 2^4$ (power relationship)

Exercises

- Convert the following to Decimal
  - 1234_8
  - 1234_7
  - 110101101110_2
  - 12.35_5
  - 1101100111.101_2
  - 10001110.111_2
  - 123.1_3
  - 10101011.001_2
  - 1234.1234_8

Decimal to Binary Conversion

- Converting to binary – can’t use polynomial evaluation!
- Whole part and fractional parts must be handled separately!
  - Whole part: Use *repeated division*.
  - Fractional part: Use *repeated multiplication*.
  - Combine results when finished.

Decimal to Binary Conversion
(Whole Part: Repeated Division)

- Divide by target radix (2 in this case)
- Remainders become digits in the new representation (0 <= digit < R)
- Digits produced in right to left order.
- Quotient is used as next dividend.
- Stop when the quotient becomes zero, but use the corresponding remainder.

Decimal to Binary Conversion
(Whole Part: Repeated Division)

$97 \div 2 \rightarrow$	quotient = 48,	remainder = 1 (LSB)
$48 \div 2 \rightarrow$	quotient = 24,	remainder = 0.
$24 \div 2 \rightarrow$	quotient = 12,	remainder = 0.
$12 \div 2 \rightarrow$	quotient = 6,	remainder = 0.
$6 \div 2 \rightarrow$	quotient = 3,	remainder = 0.
$3 \div 2 \rightarrow$	quotient = 1,	remainder = 1.
$1 \div 2 \rightarrow$	quotient = 0 (Stop)	remainder = 1 (MSB)

Result = $1\,1\,0\,0\,0\,0\,1_2$

Decimal to Binary Conversion
(Whole Part:)

- Easier to *know* binary positions…. start to think in binary!
- $97_{10} = ?_2$
- $2^6 = 64$, 33 remainder => 7th bit set
- $2^5 = 32$, 1 remainder = > 6th & 1st bit set
- => 1100001

Decimal to Binary Conversion
(Fractional Part: Repeated Multiplication)

- Multiply by target radix (2 in this case)
- Whole part of product becomes digit in the new representation (0 <= digit < R)
- Digits produced in left to right order.
- Fractional part of product is used as next multiplicand.
- Stop when the fractional part becomes zero (sometimes it won’t).

Decimal to Binary Conversion
(Fractional Part: Repeated Multiplication)

$.1 \times 2 \Rightarrow$ 0.2 (fractional part = .2, whole part = 0)

$.2 \times 2 \Rightarrow$ 0.4 (fractional part = .4, whole part = 0)

$.4 \times 2 \Rightarrow$ 0.8 (fractional part = .8, whole part = 0)

$.8 \times 2 \Rightarrow$ 1.6 (fractional part = .6, whole part = 1)

$.6 \times 2 \Rightarrow$ 1.2 (fractional part = .2, whole part = 1)

Result = $.00011001100110011_2$…..

(How much should we keep?)

Exercises

- Convert to Binary
  - 562_{10}
  - 123.675_{10}
  - 234.1_{10}
  - 745.5_{10}
  - 999.25_{10}
  - 1234.125_{10}
- Convert to Octal
  - 562.4_{10}
- **Convert to Decimal**
  - $\mathbf{12212.1_3}$
- Convert to Binary
  - 123.46_8
- **Convert to Hex.**
  - $\mathbf{2122.1_3}$

Moral

- Some fractional numbers have an exact representation in one number system, but not in another! E.g., 1/3rd has no exact representation in decimal, but does in base 3!
- What about 1/10th when represented in binary?
- What does this imply about *equality comparisons* of real numbers?
- Can these *representation errors* accumulate?

See any problems with this code?

```
for (i = 0; i < 10000; i += 0.1)
{
    printf("i = %d\n", i);
};
```

Counting

- Principle is the same regardless of radix.
  - Add 1 to the least significant digit.
  - If the result is less than R, write it down and copy all the remaining digits on the left.
  - Otherwise, write down zero and add 1 to the next digit position, etc.

Counting in Binary

Dec	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Note the pattern!

- LSB (bit 0) toggles on *every* count.
- Bit 1 toggles on every *second* count.
- Bit 2 toggles on every *fourth* count.
- Etc….

Question:

- Do you trust the used car salesman that tells you that the 1966 Cortina he wants to sell you has only the 13,000 miles that it's odometer shows?
- If not, what has happened?
- Why?

Representation Rollover

- Consequence of *fixed precision.*
- Computers use fixed precision!
- Digits are lost on the left-hand end.
- Remaining digits are still correct.
- Rollover while counting . . .

  Up: "999999" → "000000" (R^n-1 → 0)

  Down: "000000" → "999999" (0 → R^n-1)

Rollover in Unsigned Binary

- Consider an 8-bit byte used to represent an unsigned integer:
  - Range: 00000000 → 11111111 (0 → 255_{10})
  - Incrementing a value of 255 should yield 256, but this exceeds the range.
  - Decrementing a value of 0 should yield –1, but this exceeds the range.
  - Exceeding the range is known as *overflow*.

Surprise! Rollover is <u>not</u> synonymous with overflow!

- Rollover describes a pattern sequence behavior.
- Overflow describes an arithmetic behavior.
- Whether or not rollover causes overflow <u>depends on how the patterns are interpreted</u> as numeric values!
  – E.g., In signed two's complement representation, 11111111 ➔ 00000000 corresponds to counting from minus one to zero.

Exercises

- State where rollover and/or overflow occurs

  242 + 13 (unsigned chars)

  127 + 1 (unsigned chars)

  128 + 128 (unsigned chars)

  -3 + 5 (signed chars)

  100 + 28 (signed chars)

  -100 + 28 (signed chars)

  -100 – 28 (signed chars)

  -100 – 50 (signed chars)

Two Interpretations

unsigned signed

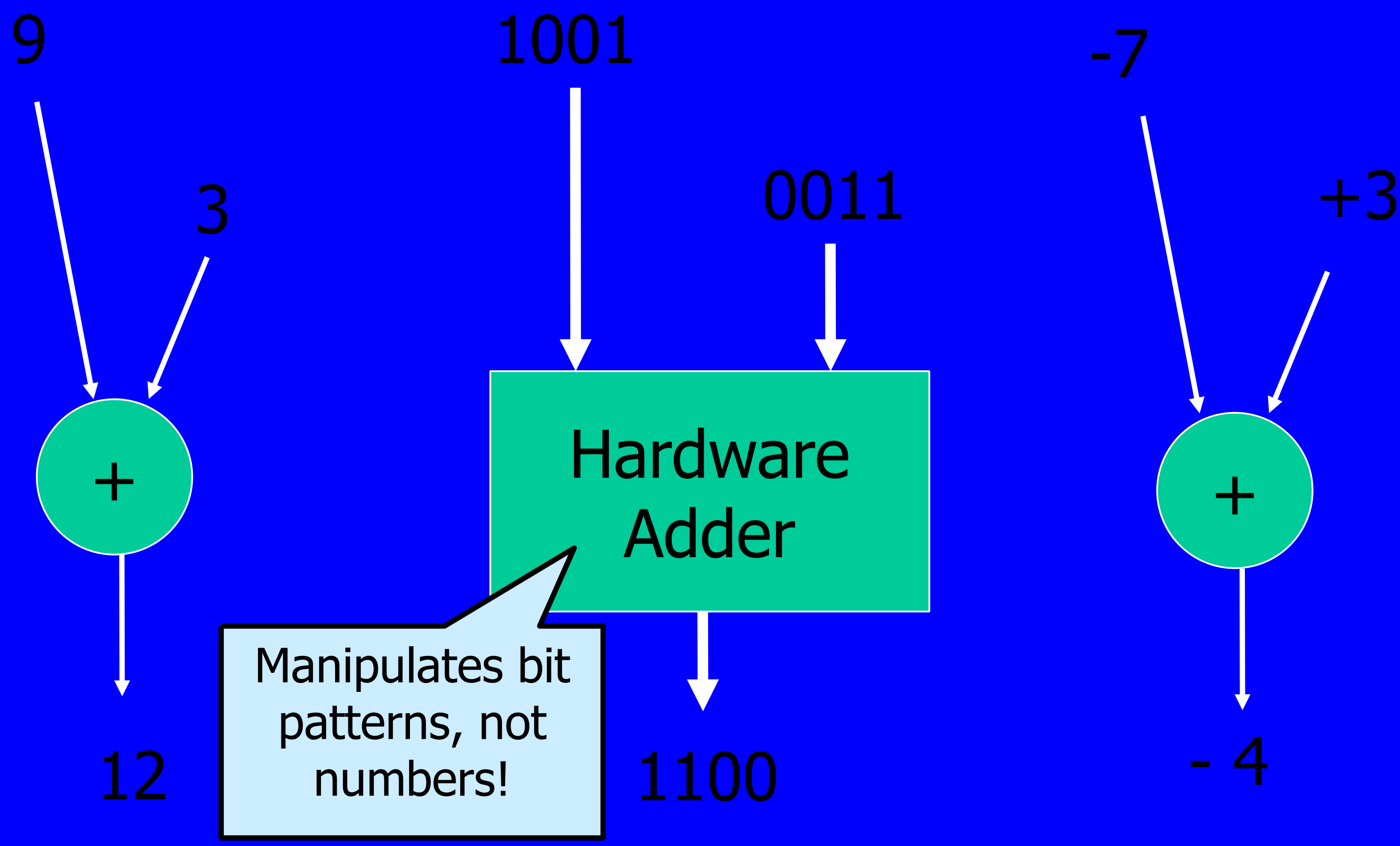
$167_{10} \longleftarrow 10100111_2 \longrightarrow -89_{10}$

- Signed vs. unsigned is a matter of interpretation; thus <u>a single bit pattern can represent two different values.</u>

- Allowing both interpretations is useful:

  Some data (e.g., count, age) can never be negative, and having a greater range is useful.

One Hardware Adder Handles Both!
(or subtractor)



Which is Greater: 1001 or 0011?

Answer: It depends!

So how does the computer decide:

“if (x > y)..”   /* Is this true or false? */

It’s a matter of interpretation, and depends on how x and y were declared: signed? Or unsigned?

Which is Greater: 1001 or 0011?

```
signed int x, y ;          MOV  EAX,[x]
                       CMP  EAX,[y]
if (x > y) ...          ➲   JLE  Skip_Then_Clause
```

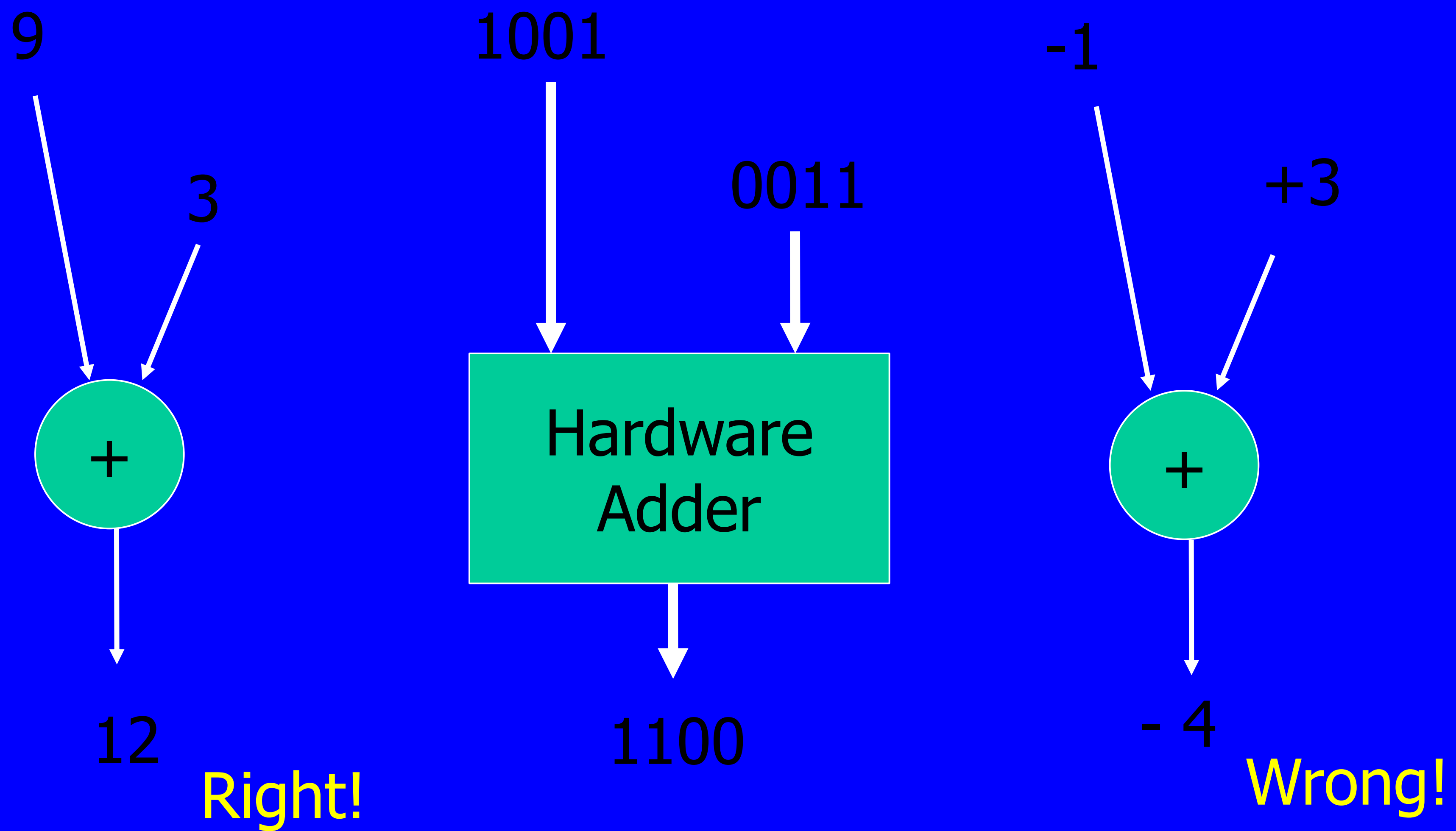
```
unsigned int x, y ;        MOV  EAX,[x]
                       CMP  EAX,[y]
if (x > y) ...          ➲   JBE  Skip_Then_Clause
```

Why Not Sign+Magnitude?

+3	0011
+2	0010
+1	0001
+0	0000
-0	1000
-1	1001
-2	1010
-3	1011

- Complicates addition :
  - To add, first check the signs. If they agree, then add the magnitudes and use the same sign; else subtract the *smaller* from the *larger* and use the sign of the larger.
  - How do you determine which is smaller/larger?
- Complicates comparators:
  - Two zeroes!

Why Not Sign+Magnitude?



Why 2's Complement?

+3	0011
+2	0010
+1	0001
0	0000
-1	1111
-2	1110
-3	1101
-4	1100

1. Just as easy to determine sign as in sign+magnitude.
2. Almost as easy to change the sign of a number.
3. Addition can proceed w/out worrying about which operand is larger.
4. A single zero!
5. One hardware adder works for both signed and unsigned operands.

Changing the Sign

Sign+Magnitude:

+4 = 0100

Change 1 bit

-4 = 1100

2's Complement:

+4 = 0100

Invert

$\overline{+4} = 1011$

Increment

$\underline{\quad +1}$

-4 = 1100

Easier Hand Method

<u>Step 2:</u> Copy the inverse of the remaining bits.

+4 = 0100

-4 = 1100

<u>Step 1:</u> Copy the bits from right to left, through and including the first 1.

Representation Width

Be Careful! You must be sure to pad the original value out to the full representation width before applying the algorithm!

Apply algorithm

Expand to 8-bits

Wrong: +25 = 11001 ➲ 00111 ➲ 00000111 = +7

Right: +25 = 11001 ➲ 00011001 ➲ 11100111 = -25

If positive: Add leading 0's
If negative: Add leading 1's

Apply algorithm

Converting 2's Complement to Decimal

- If positive simply apply polynomial evaluation (or just think in binary!) and mark as positive

01110110 ➲ $+118_{10}$

00010110 ➲ $+22_{10}$

Converting 2's Complement to Decimal

- If Negative:
  - First form 2's complement
  - Then convert to Decimal as for positive
  - Mark as Negative

11101010 ➲ 00010110 ➲ -22

Form 2's Complement

Convert to Decimal

Converting 2's Complement to Decimal

- Don't forget to pad!

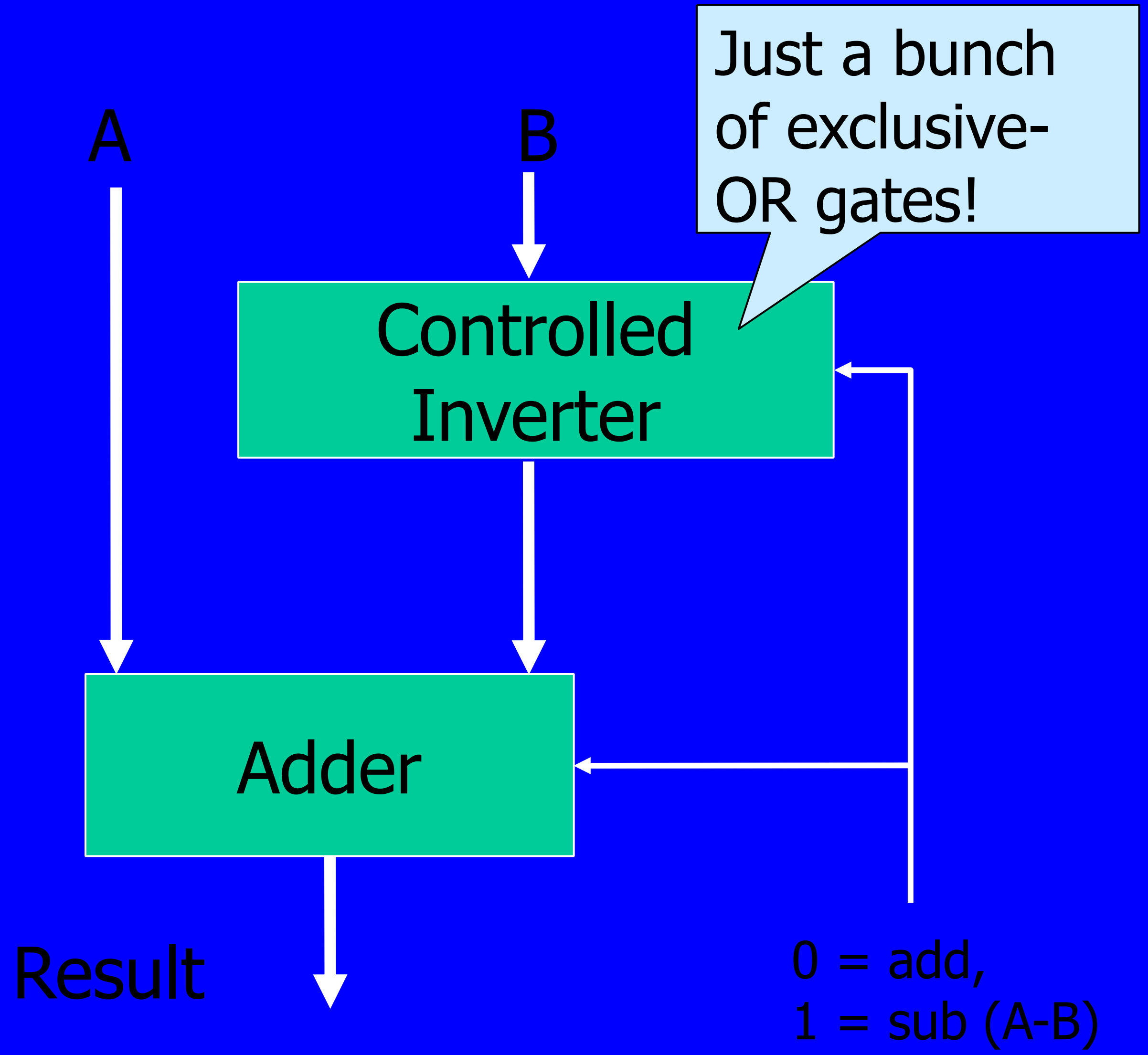
101010 ➲ 11101010 ➲ 00010110 ➲ -22

Pad

Form 2's Complement

Convert to Decimal

Subtraction Is Easy!



2's Complement Anomaly!

-128 = 1000 0000 (8 bits)

+128?

Step 1: Invert all bits ➲ 0111 1111

Step 2: Increment ➲ 1000 0000

Same result with either method! Why?

Exercises

- Give the 2's complement of the following

  101 (8 bit)

  123_{10} (8 bit)

  123_{10} (16 bit)

  -128_{10} (8 bit)

  127_{10} (8 bit)

  123_{8} (8 bit)
- Add the following and give the ans in decimal

  11101011 + 0011 (8 bit)

Range of Unsigned Integers

Each of 'n' bits can have one of two values.

Total # of patterns of n bits = $\underbrace{2\times 2\times 2\times\ldots 2}_{\text{'n' 2's}}$

'n' 2's

$= 2^n$

If n-bits are used to represent an unsigned integer value:

Range: 0 to 2^n-1 (2^n different values)

Range of Signed Integers

- Half of the 2^n patterns will be used for positive values, and half for negative.
- Half is 2^{n-1}.
- Positive Range: 0 to $2^{n-1}-1$ (2^{n-1} patterns)
- Negative Range: -2^{n-1} to -1 (2^{n-1} patterns)
- 8-Bits (n = 8): -2^7 (-128) to $+2^7-1$ (+127)

Unsigned Overflow

1100	(12)
+0111	(7)
10011	

Lost

(Result limited by word size)

0011	(3) **wrong**

Value of lost bit is 2^n (16).

16 + 3 = 19

(The right answer!)

Signed Overflow

- Overflow is impossible ☺ when adding (subtracting) numbers that have different (same) signs.

- Overflow occurs when the magnitude of the result extends into the sign bit position:

01111111 ➲ (0)10000000

This is not rollover!

Signed Overflow

-120_{10}	➔	10001000_2
$\underline{-17}_{10}$		$+\underline{11101111}_2$
sum: -137_{10}		101110111_2
		01110111_2 (keep 8 bits)
		$(+119_{10})$ **wrong**

Note: $119 - 2^8 = 119 - 256 = -137$

Floating-Point Reals

31	30 23	22 0
±	Exponent	Fractional Bits of Significand

Three components:

Base is implied

$$\pm \text{ significand} \times 2^{\text{exponent}}$$

Sign

A biased integer value

An <u>unsigned</u> fractional value

Fixed-Point Reals

Three components:

Implied binary point

0· · · 00.00 · · ·0

Whole part

Fractional part

Fixed vs. Floating

- Floating-Point:

  Pro: Large dynamic range determined by exponent; resolution determined by significand.

  Con: Implementation of arithmetic in hardware is complex (slow).

- Fixed-Point:

  Pro: Arithmetic is implemented using regular integer operations of processor (fast).

  Con: Limited range and resolution.

Representation of Characters

Representation

00100100

Interpretation

$

ASCII
Code

Character Constants in C

- To distinguish a character that is used as data from an identifier that consists of only one character long:

  x   is an identifier.
  ‘x’   is a character constant.

- The value of ‘x’ is the ASCII code of the character x.

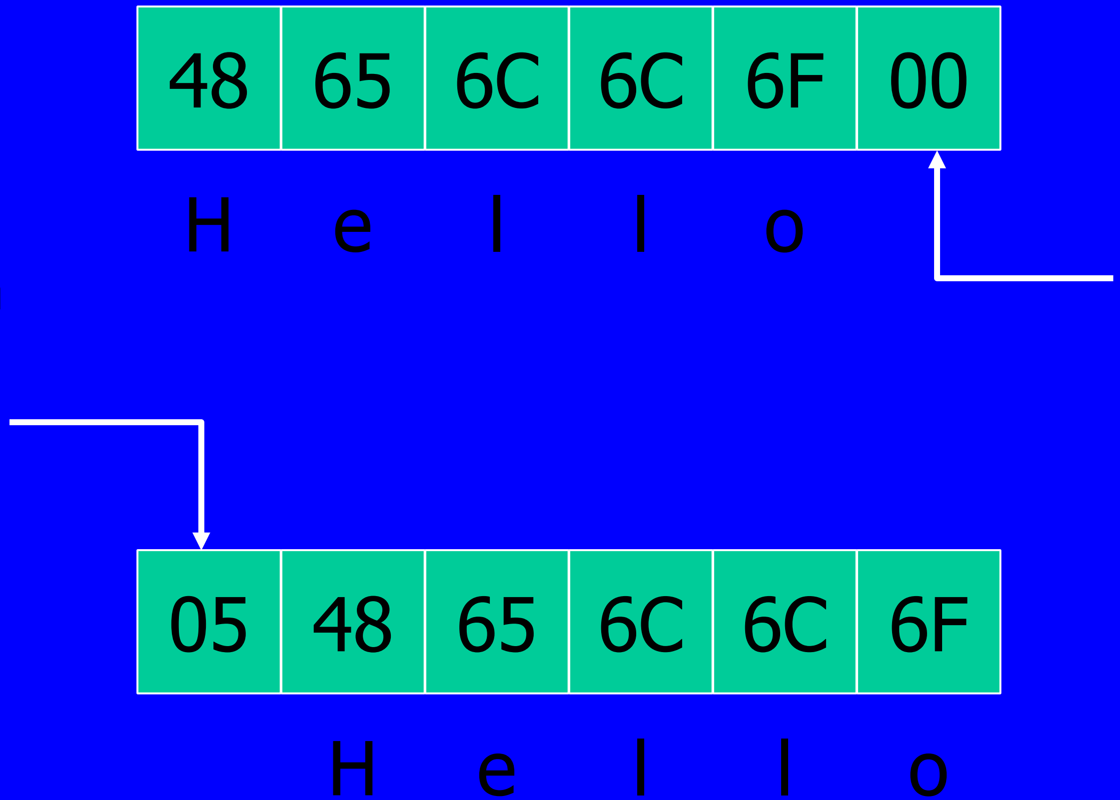
Character Escapes

- A way to represent characters that do not have a corresponding graphic symbol.

Escape Character		Character Constant
\b	Backspace	'\b'
\t	Horizontal Tab	'\t'
\n	Linefeed	'\n'
\r	Carriage return	'\r'

See Table 2-9 in the text for others.

Representation of Strings



C uses a terminating "NUL" byte of all zeros at the end of the string.

Pascal uses a prefix count at the beginning of the string.

String Constants in C

Character string

COEN 20 is “fun”!

C string constant

“COEN 20 is \“fun\”!”

4	4	4	4	2	3	3	2	6	7	2	2	6	7	6	2	2	0
3	F	5	E	0	2	0	0	9	3	0	2	6	5	E	2	1	0
C	O	E	N		2	0		i	s		“	f	u	n	”	!	\0

Binary Coded Decimal (BCD)

Packed (2 digits per byte):

0111	0011
7	3

Unpacked (1 digit per byte):

0000	0111	0000	0011
	7		3