Binary Representation

- The basis of all digital data is binary representation.
- Binary - means 'two'
  - 1, 0
  - True, False
  - Hot, Cold
  - On, Off
- We must be able to handle more than just values for real world problems
  - 1, 0, 56
  - True, False, Maybe
  - Hot, Cold, LukeWarm, Cool
  - On, Off, Leaky

Number Systems

- To talk about binary data, we must first talk about number systems
- The decimal number system (base 10) you should be familiar with!
  - A digit in base 10 ranges from 0 to 9.
  - A digit in base 2 ranges from 0 to 1 (binary number system). A digit in base 2 is also called a 'bit'.
  - A digit in base R can range from 0 to R-1
  - A digit in Base 16 can range from 0 to 16-1 (0,1,2,3,4,5,5,6,7,8,9,A,B,C,D,E,F). Use letters A-F to represent values 10 to 15. Base 16 is also called Hexadecimal or just 'Hex'.

Positional Notation

Value of number is determined by multiplying each digit by a weight and then summing. The weight of each digit is a POWER of the BASE and is determined by position.

$953.78 = 9 \times 10^2 + 5 \times 10^1 + 3 \times 10^0 + 7 \times 10^{-1} + 8 \times 10^{-2}$
$= 900 + 50 + 3 + .7 + .08 = 953.78$

$\% 1011.11 = 1x2^3 + 0x2^2 + 1x2^1 + 1x2^0 + 1x2^{-1} + 1x2^{-2}$
$= 8 + 0 + 2 + 1 + 0.5 + 0.25$
$= 11.75$

$\$ A2F = 10x16^2 + 2x16^1 + 15x16^0$
$= 10 \times 256 + 2 \times 16 + 15 \times 1$
$= 2560 + 32 + 15 = 2607$

Base 10, Base 2, Base 16

The textbook uses subscripts to represent different bases (ie. $A2F_{16}$, 953.78_{10}, 1011.11_2)

I will use special symbols to represent the different bases. The default base will be decimal, no special symbol for base 10.

The '$' will be used for base 16 ($A2F)
Will also use 'h' at end of number (A2Fh)

The '%' will be used for base 2 (%10101111)

If ALL numbers on a page are the same base (ie, all in base 16 or base 2 or whatever) then no symbols will be used and a statement will be present that will state the base (ie, all numbers on this page are in base 16).

Common Powers

$2^{-3} = 0.125$
$2^{-2} = 0.25$
$2^{-1} = 0.5$
$2^0 = 1$
$2^1 = 2$
$2^2 = 4$
$2^3 = 8$
$2^4 = 16$
$2^5 = 32$
$2^6 = 64$
$2^7 = 128$
$2^8 = 256$
$2^9 = 512$
$2^{10} = 1024$
$2^{11} = 2048$
$2^{12} = 4096$

$16^0 = 1 = 2^0$
$16^1 = 16 = 2^4$
$16^2 = 256 = 2^8$
$16^3 = 4096 = 2^{12}$

$2^{10} =$ 1024 = 1 K
$2^{20} =$ 1048576 = 1 M (1 Megabits) = 1024 K = $2^{10} \times 2^{10}$
$2^{30} =$ 1073741824 = 1 G (1 Gigabits)

Conversion of Any Base to Decimal

Converting from ANY base to decimal is done by multiplying each digit by its weight and summing.

Binary to Decimal

$\% 1011.11 = 1x2^3 + 0x2^2 + 1x2^1 + 1x2^0 + 1x2^{-1} + 1x2^{-2}$
$= 8 + 0 + 2 + 1 + 0.5 + 0.25$
$= 11.75$

Hex to Decimal

$A2Fh = 10x16^2 + 2x16^1 + 15x16^0$
$= 10 \times 256 + 2 \times 16 + 15 \times 1$
$= 2560 + 32 + 15 = 2607$

Conversion of Decimal Integer To ANY Base

Divide Number N by base R until quotient is 0. Remainder at EACH step is a digit in base R, from Least Significant digit to Most significant digit.

Convert 53 to binary

53/2 = 26, rem = 1 ← Least Significant Digit
26/2 = 13, rem = 0
13/2 = 6 , rem = 1
6 /2 = 3, rem = 0
3/2 = 1, rem = 1
1/2 = 0, rem = 1 ← Most Significant Digit

53 = % 110101
$= 1x2^5 + 1x2^4 + 0x2^3 + 1x2^2 + 0x2^1 + 1x2^0$
= 32 + 16 + 0 + 4 + 0 + 1 = 53

Least Significant Digit Most Significant Digit

53 = % 110101

Most Significant Digit (has weight of 2^5 or 32). For base 2, also called Most Significant Bit (MSB). Always LEFTMOST digit.

Least Significant Digit (has weight of 2^0 or 1). For base 2, also called Least Significant Bit (LSB). Always RIGHTMOST digit.

More Conversions

Convert 53 to Hex

53/16 = 3, rem = 5
3 /16 = 0 , rem = 3
53 = 35h
$= 3 \times 16^1 + 5 \times 16^0$
= 48 + 5 = 53

Hex (base 16) to Binary Conversion

Each Hex digit represents 4 bits. To convert a Hex number to Binary, simply convert each Hex digit to its four bit value.

Hex Digits to binary:
$ 0 = % 0000
$ 1 = % 0001
$2 = % 0010
$3 = % 0011
$4 = % 0100
$5 = % 0101
$6 = % 0110
$7 = % 0111
$8 = % 1000

Hex Digits to binary (cont):
$ 9 = % 1001
$ A = % 1010
$ B = % 1011
$ C = % 1100
$ D = % 1101
$ E = % 1110
$ F = % 1111

Hex to Binary, Binary to Hex

A2Fh = % 1010 0010 1111

345h = % 0011 0100 0101

Binary to Hex is just the opposite, create groups of 4 bits starting with least significant bits. If last group does not have 4 bits, then pad with zeros for unsigned numbers.

% 1010001 = % 0101 0001 = 51h

Padded with a zero

A Trick!

If faced with a large binary number that has to be converted to decimal, I first convert the binary number to HEX, then convert the HEX to decimal. Less work!

% 110111110011 = % 1101 1111 0011
= D F 3
$= 13 \times 16^2 + 15 \times 16^1 + 3x16^0$
= 13 x 256 + 15 x 16 + 3 x 1
= 3328 + 240 + 3
= 3571

Of course, you can also use the binary, hex conversion feature on your calculator. Too bad calculators won't be allowed on the first test, though…...

Binary Numbers Again

Recall than N binary digits (N bits) can represent unsigned integers from 0 to 2^N-1.

4 bits = 0 to 15
8 bits = 0 to 255
16 bits = 0 to 65535

Besides simply representation, we would like to also do arithmetic operations on numbers in binary form. Principle operations are addition and subtraction.

V 0.1 13

Binary Arithmetic, Subtraction

The rules for binary arithmetic are:

- 0 + 0 = 0, carry = 0
- 1 + 0 = 1, carry = 0
- 0 + 1 = 1, carry = 0
- 1 + 1 = 0, carry = 1

The rules for binary subtraction are:

- 0 - 0 = 0, borrow = 0
- 1 - 0 = 1, borrow = 0
- 0 - 1 = 1, borrow = 1
- 1 - 1 = 0, borrow = 0

Borrows, Carries from digits to left of current of digit.

Binary subtraction, addition works just the same as decimal addition, subtraction.

V 0.1 14

Binary, Decimal addition

Decimal

```
  34
+ 17
------
  51
```

from LSD to MSD:
7+4 = 1; with carry out of 1 to next column

1 (carry) + 3 + 1 = 5.
answer = 51.

Binary

```
  % 101011
+ % 000001
---------------
    101100
```

From LSB to MSB:
1+1 = 0, carry of 1
1 (carry)+1+0 = 0, carry of 1
1 (carry)+0 + 0 = 1, no carry
1 +0 = 1
0 + 0 = 0
1 + 0 = 1
answer = % 101100

V 0.1 15

Subtraction

Decimal

```
  900
- 001
-------
  899
```

0-1 = 9; with borrow of 1 from next column
0 -1 (borrow) - 0 = 9, with borrow of 1
9 - 1 (borrow) - 0 = 8.
Answer = 899.

Binary

```
  % 100
- % 001
-------
    011
```

0-1 = 1; with borrow of 1 from next column
0 -1 (borrow) - 0 = 1, with borrow of 1
1 - 1 (borrow) - 0 = 0.
Answer = % 011.

V 0.1 16

Hex Addition

```
  3Ah
+ 28h
--------
  62h
```

A+8 = 2; with carry out of 1 to next column

1 (carry) + 3 + 2 = 6.
answer = $ 62.

Decimal check.

3Ah = 3 x 16 + 10
= 58
28h = 2 x 16 + 8
= 40
58 + 40 = 98

62h = 6 x 16 + 2
= 96 + 2 = 98!!

V 0.1 17

Hex addition again

Why is Ah + 8h = 2 with a carry out of 1?

The carry out has a weight equal to the BASE (in this case 16). The digit that gets left is the excess (BASE - sum).

Ah + 8h = 10 + 8 = 18.

18 is GREATER than 16 (BASE), so need a carry out!

Excess is 18 - BASE = 18 - 16 = 2, so '2' is digit.

Exactly the same thing happens in Decimal.
5 + 7 = 2, carry of 1.
5 + 7 = 12, this is greater than 10!.
So excess is 12 - 10 = 2, carry of 1.

V 0.1 18

Hex Subtraction

34h
\- 27h
\--------
0Dh

4-7 = D; with borrow of 1 from next column

3 - 1 (borrow) - 2 = 0.
answer = $ 0D.

Decimal check.

34h = 3 x 16 + 4
= 52
27h = 2 x 16 + 7
= 39
52 - 39 = 13

0Dh = 13 !!

Hex subtraction again

Why is 4h – 7h = $D with a borrow of 1?

The borrow has a weight equal to the BASE (in this case 16).

BORROW +4h – 7h = 16 + 4 -7 = 20 -7 = 13 = Dh.

Dh is the result of the subtraction with the borrow.

Exactly the same thing happens in decimal.
3 - 8 = 5 with borrow of 1
borrow + 3 - 8 = 10 + 3 - 8 = 13 - 8 = 5.

Fixed Precision

With paper and pencil, I can write a number with as many digits as I want:

1,027,80,032,034,532,002,391,030,300,209,399,302,992,092,920

A microprocessor or computing system usually uses FIXED PRECISION for integers; they limit the numbers to a fixed number of bits:

$	AF4500239DEFA231	64 bit number, 16 hex digits
$	9DEFA231	32 bit number, 8 hex digits
$	A231	16 bit number, 4 hex digits
$	31	8 bit number, 2 hex digits

High end microprocessors use 64 or 32 bit precision; low end microprocessors use 16 or 8 bit precision.

Unsigned Overflow

In this class I will use 8 bit precision most of the time, 16 bit occassionally.

Overflow occurs when I add or subtract two numbers, and the correct result is a number that is outside of the range of allowable numbers for that precision. I can have both unsigned and signed overflow (more on signed numbers later)

8 bits -- unsigned integers 0 to 2^8 -1 or 0 to 255.

16 bits -- unsigned integers 0 to 2^{16}-1 or 0 to 65535

Unsigned Overflow Example

Assume 8 bit precision; ie. I can't store any more than 8 bits for each number.

Lets add 255 + 1 = 256. The number 256 is OUTSIDE the range of 0 to 255! What happens during the addition?

255 = $ FF
\+ 1 = $ 01
\-------------------
256 /= $00

/= means Not Equal

$F + 1 = 0, carry out
$F + 1 (carry) + 0 = 0, carry out
Carry out of MSB falls off end, No place to put it!!!
Final answer is WRONG because could not store carry out.

Unsigned Overflow

A carry out of the Most Significant Digit (MSD) or Most Significant Bit (MSB) is an OVERFLOW indicator for addition of UNSIGNED numbers.

The correct result has overflowed the number range for that precision, and thus the result is incorrect.

If we could STORE the carry out of the MSD, then the answer would be correct. But we are assuming it is discarded because of fixed precision, so the bits we have left are the incorrect answer.

Signed Integer Representation

We have been ignoring large sets of numbers so far; ie. the sets of signed integers, fractional numbers, and floating point numbers.

We will not talk about fractional number representation (10.3456) or floating point representation (i.e. 9.23×10^{13}).

We WILL talk about signed integer representation.

The *PROBLEM* with signed integers (- 45, + 27, -99) is the SIGN! How do we encode the sign?

The sign is an extra piece of information that has to be encoded in addition to the magnitude. Hmmmmm, what can we do??

Twos Complement Examples

-5	=	% 11111011	=	$ FB	
+5	=	% 00000101	=	$ 05	
+127	=	% 01111111	=	$ 7F	
-127	=	% 10000001	=	$ 81	
-128	=	% 10000000	=	$80	(note the extended range!)
+ 0	=	% 00000000	=	$ 00	
- 0	=	% 00000000	=	$ 00	(only 1 zero!!!)

For 8 bits, can represent the signed integers -128 to +127.

For N bits, can represent the signed integers

$$-2^{(N-1)} \quad \text{to} \quad +2^{(N-1)} - 1$$

Note that negative range extends one more than positive range.

Twos Complement Comments

Twos complement is the method of choice for representing signed integers.

There is only one zero, and K + (-K) = 0.

-5 + 5 = $ FB + $ 05 = $00 = 0 !!!

Normal binary addition is used for adding numbers that represent twos complement integers.

A common Question from Students

A question I get asked by students all the time is :

Given a hex number, how do I know if it is in 2's complement or 1's complement; is it already in 2's complement or do I have put it in 2's complement, etc, yadda,yadda, yadda….

If I write a HEX number, I will ask for a decimal representation if you INTERPRET the encoding as a particular method (i.e, either 2's complement, 1's complement, signed magnitude).

A Hex or binary number BY ITSELF can represent ANYTHING (unsigned number, signed number, character code, colored llamas, etc). You MUST HAVE additional information that tells you what the encoding of the bits mean.

Example Conversions

$FE as an 8 bit unsigned integer = 254
$FE as an 8 bit twos complement integer = -2

$7F as an 8 bit unsigned integer = 127
$7f as an 8 bit twos complement integer = +127

To do hex to signed decimal conversion, we need to determine sign (Step 1), determine Magnitude (step 2), combine sign and magnitude (Step 3)

Hex to Signed Decimal Conversion Rules

Given a Hex number, and you are told to convert to a signed integer (Hex number uses 2s complement encoding)

STEP 1: Determine the sign! If the Most Significant Bit is zero, the sign is positive. If the MSB is one, the sign is negative.

$F0 = % 11110000 (MSB is '1'), so sign of result is '-'
$64 = % 01100100 (MSB is '0'), so sign of result is '+'.

If the Most Significant Hex Digit is > 7, then MSB = '1' !!!
(eg, $8,9,A,B,C,D,E,F => MSB = '1' !!!)

Hex to Signed Decimal (cont)

STEP 2 (positive sign): If the sign is POSITIVE, then just convert the hex value to decimal.

$64 is a positive number, decimal value is
6 x 16 + 4 = 100.

Final answer is +100.

$64 as an 8 bit twos complement integer = +100

Hex to Signed Decimal (cont)

STEP 2 (negative sign): If the sign is Negative, then need to compute the magnitude of the number.

We will use the trick that - (-N) = + N
i.e. Take the negative of a negative number will give you the positive number. In this case the number will be the magnitude.

For 2s complement representation, complement and add one.
$F0 = % 11110000 => %00001111 + 1 = %00010000 = $10 = 16

Hex to Signed Decimal (cont)

STEP 3 : Just combine the sign and magnitude to get the result.

$F0 as 8 bit twos complement number is -16

$64 as an 8 bit twos complement integer = +100

Signed Decimal to Hex conversion 2's complement

Step 1: Ignore the sign, convert the magnitude of the number to binary.

34 = 2 x 16 + 2 = $ 22 = % 00100010
20 = 1 x 16 + 4 = $ 14 = % 00010100

Step 2 (positive decimal number): If the decimal number was positive, then you are finished!

+34 as an 8 bit 2s complement number is $ 22 = % 00100010

Signed Decimal to Hex conversion (cont)

Step 2 (negative decimal number): Need to do more if decimal number was negative. To get the final representation, we will use the trick that:

- (+N) = -N
i.e., if you take the negative of a positive number, get Negative number.

For 2s complement, complement and add one.
20 = % 00010100 => %11101011 + 1 = %11101100 = $EC

Signed Decimal to Hex conversion (cont)

Final results:

+34 as an 8 bit 2s complement number is $ 22 = % 00100010

-20 as an 8 bit 2s complement number is $ EC = % 11101100

Two's Complement Overflow

Consider two 8-bit 2's complement numbers. I can represent the signed integers -128 to +127 using this representation.

What if I do (+1) + (+127) = +128. The number +128 is OUT of the RANGE that I can represent with 8 bits. What happens when I do the binary addition?

+127 = $ 7F

\+ +1 = $ 01

\-------------------

128 /= $80 (this is actually -128 as a twos complement number!!! - the wrong answer!!!)

How do I know if overflowed occurred? Added two POSITIVE numbers, and got a NEGATIVE result.

Detecting Two's Complement Overflow

Two's complement overflow occurs is:

Add two POSITIVE numbers and get a NEGATIVE result
Add two NEGATIVE numbers and get a POSITIVE result

I CANNOT get two's complement overflow if I add a NEGATIVE and a POSITIVE number together.

The Carry out of the Most Significant Bit means nothing if the numbers are two's complement numbers.

Some Examples

All hex numbers represent signed decimal in two's complement format.

$ FF = -1

\+ $ 01 = + 1

\--------

$ 00 = 0

Note there is a carry out, but the answer is correct. Can't have 2's complement overflow when adding positive and negative number.

$ FF = -1

\+ $ 80 = -128

\--------

$ 7F = +127 (incorrect!!)

Added two negative numbers, got a positive number. Twos Complement overflow.

Adding Precision (unsigned)

What if we want to take an unsigned number and add more bits to it?

Just add zeros to the left.

128 = $80 (8 bits)
= $0080 (16 bits)
= $00000080 (32 bits)

Adding Precision (two's complement)

What if we want to take a twos complement number and add more bits to it?
Take whatever the SIGN BIT is, and extend it to the left.

-128 = $80 = % 10000000 (8 bits)
= $FF80 = % 1111111110000000 (16 bits)
= $FFFFFF80 (32 bits)

\+ 127 = $7F = % 01111111 (8 bits)
= $007F = % 0000000001111111 (16 bits)
= $0000007F (32 bits)

This is called SIGN EXTENSION. Extending the MSB to the left works for two's complement numbers and unsigned numbers.

Binary Codes (cont.)

N bits (or N binary Digits) can represent 2^N different values.

(for example, 4 bits can represent 2^4 or 16 different values)

N bits can take on unsigned decimal values from 0 to 2^N-1.

Codes usually given in tabular form.

000	black
001	red
010	pink
011	yellow
100	brown
101	blue
110	green
111	white

Codes for Characters

Also need to represent Characters as digital data. The ASCII code (American Standard Code for Information Interchange) is a 7-bit code for Character data. Typically 8 bits are actually used with the 8th bit being zero or used for error detection (parity checking). 8 bits = 1 Byte. (see Table 2.5, pg 47, Uffenbeck).

'A' = % 01000001 = $41
'&' = % 00100110 = $26

7 bits can only represent 2^7 different values (128). This enough to represent the Latin alphabet (A-Z, a-z, 0-9, punctuation marks, some symbols like $), but what about other symbols or other languages?

ASCII

American Standard Code for Information Interchange

Table 2.5 American Standard Code for Information Interchange (ASCII)*

<table>
<tr><th rowspan="2">Least Significant Bit</th><th colspan="8">Most Significant Bit</th></tr>
<tr><th>0
0000</th><th>1
0001</th><th>2
0010</th><th>3
0011</th><th>4
0100</th><th>5
0101</th><th>6
0110</th><th>7
0111</th></tr>
<tr><td>0 0000</td><td>NUL</td><td>DLE</td><td>SP</td><td>0</td><td>@</td><td>P</td><td>`</td><td>p</td></tr>
<tr><td>1 0001</td><td>SOH</td><td>DC1</td><td>!</td><td>1</td><td>A</td><td>Q</td><td>a</td><td>q</td></tr>
<tr><td>2 0010</td><td>STX</td><td>DC2</td><td>"</td><td>2</td><td>B</td><td>R</td><td>b</td><td>r</td></tr>
<tr><td>3 0011</td><td>ETX</td><td>DC3</td><td>#</td><td>3</td><td>C</td><td>S</td><td>c</td><td>s</td></tr>
<tr><td>4 0100</td><td>EOT</td><td>DC4</td><td>$</td><td>4</td><td>D</td><td>T</td><td>d</td><td>t</td></tr>
<tr><td>5 0101</td><td>ENQ</td><td>NAK</td><td>%</td><td>5</td><td>E</td><td>U</td><td>e</td><td>u</td></tr>
<tr><td>6 0110</td><td>ACK</td><td>SYN</td><td>&</td><td>6</td><td>F</td><td>V</td><td>f</td><td>v</td></tr>
<tr><td>7 0111</td><td>BEL</td><td>ETB</td><td>'</td><td>7</td><td>G</td><td>W</td><td>g</td><td>w</td></tr>
<tr><td>8 1000</td><td>BS</td><td>CAN</td><td>(</td><td>8</td><td>H</td><td>X</td><td>h</td><td>x</td></tr>
<tr><td>9 1001</td><td>HT</td><td>EM</td><td>)</td><td>9</td><td>I</td><td>Y</td><td>i</td><td>y</td></tr>
<tr><td>A 1010</td><td>LF</td><td>SUB</td><td>*</td><td>:</td><td>J</td><td>Z</td><td>j</td><td>z</td></tr>
<tr><td>B 1011</td><td>VT</td><td>ESC</td><td>+</td><td>;</td><td>K</td><td>[</td><td>k</td><td>{</td></tr>
<tr><td>C 1100</td><td>FF</td><td>FS</td><td>,</td><td><</td><td>L</td><td>\</td><td>l</td><td>|</td></tr>
<tr><td>D 1101</td><td>CR</td><td>GS</td><td>–</td><td>=</td><td>M</td><td>]</td><td>m</td><td>}</td></tr>
<tr><td>E 1110</td><td>SO</td><td>RS</td><td>.</td><td>></td><td>N</td><td>^</td><td>n</td><td>~</td></tr>
<tr><td>F 1111</td><td>SI</td><td>US</td><td>/</td><td>?</td><td>O</td><td>—</td><td>o</td><td>DEL</td></tr>
</table>

*Bit 7 of the code is assumed to be 0.

Source: J. Uffenbeck, *Microcomputers and Microprocessors: The 8080, 8085, and Z-80*, Prentice Hall, Englewood Cliffs, N.J., 1985.

UNICODE

UNICODE is a 16-bit code for representing alphanumeric data. With 16 bits, can represent 2^{16} or 65536 different symbols. 16 bits = 2 Bytes per character.

$0041-005A A-Z
$0061-4007A a-z

Some other alphabet/symbol ranges

$3400-3d2d Korean Hangul Symbols
$3040-318F Hiranga, Katakana, Bopomofo, Hangul
$4E00-9FFF Han (Chinese, Japenese, Korean)

UNICODE used by Web browsers, Java, most software these days.

Codes for Decimal Digits

There are even codes for representing decimal digits. These codes use 4-bits for EACH decimal digits; it is NOT the same as converting from decimal to binary.

BCD Code
0 = % 0000
1 = % 0001
2 = % 0010
3 = % 0011
4 = % 0100
5 = % 0101
6 = % 0110
7 = % 0111
8 = % 1000
9 = % 1001

In BCD code, each decimal digit simply represented by its binary equivalent.
96 = % 1001 0110 = $ 96 (BCD code)

Advantage: easy to convert
Disadvantage: takes more bits to store a number:

255 = % 1111 1111 = $ FF (binary code)
255 = % 0010 0101 0101 = $ 255 (BCD code)

takes only 8 bits in binary, takes 12 bits in BCD.

Gray Code for decimal Digits

Gray Code
0 = % 0000
1 = % 0001
2 = % 0011
3 = % 0010
4 = % 0110
5 = % 1110
6 = % 1010
7 = % 1011
8 = % 1001
9 = % 1000

A Gray code changes by only 1 bit for adjacent values. This is also called a 'thumbwheel' code because a thumbwheel for choosing a decimal digit can only change to an adjacent value (4 to 5 to 6, etc) with each click of the thumbwheel. This allows the binary output of the thumbwheel to only change one bit at a time; this can help reduce circuit complexity and also reduce signal noise.

What do you need to Know?

- Convert hex, binary integers to Decimal
- Convert decimal integers to hex, binary
- Convert hex to binary, binary to Hex
- N binary digits can represent 2^N values, unsigned integers 0 to 2^N-1.
- Addition, subtraction of binary, hex numbers
- Detecting unsigned overflow
- Converting a decimal number to Twos Complement
- Converting a hex number in 2s complement to decimal

What do you need to know? (cont)

- Number ranges for 2s complement
- Overflow in 2s complement
- Sign extension in 2s complement
- ASCII, UNICODE are binary codes for character data
- BCD code is alternate code for representing decimal digits
- Gray codes can also represent decimal digits; adjacent values in Gray codes change only by one bit.