Binary Arithmetic

CS 64: Computer Organization and Design Logic
Lecture #2

Ziad Matni
Dept. of Computer Science, UCSB

Adding this Class

- The class is full – I will not be adding more ppl ☹
  - Even if others drop

Lecture Outline

- Review of positional notation, binary logic
- Bitwise operations
- Bit shift operations
- Two’s complement
- Addition and subtraction in binary
- Multiplication in binary

Positional Notation in Decimal

Continuing with our example…

642 in base 10 *positional notation* is:

$$
\begin{aligned}
& 6 \times 10^2 = 6 \times 100 = 600 \\
+\ & 4 \times 10^1 = 4 \times 10 = 40 \\
+\ & 2 \times 10^0 = 2 \times 1 = 2 \quad = 642 \text{ in base 10}
\end{aligned}
$$

6	4	2
100	10	1

642 $_{\text{(base 10)}}$ **= 600 + 40 + 2**

Positional Notation

This is how you convert any base number into decimal!

What if "642" is expressed in the base of 13?

$$
\begin{aligned}
& 6 \times 13^2 &= \; & 6 \times 169 &= 1014 \\
+ \; & 4 \times 13^1 &= \; & 4 \times 13 &= 52 \\
+ \; & 2 \times 13^0 &= \; & 2 \times 1 &= 2
\end{aligned}
$$

6	4	2
13^2	13^1	13^0

$642_{\text{(base 13)}} = 1014 + 52 + 2$

$= 1068_{\text{(base 10)}}$

Positional Notation in Binary

11101 in base 2 *positional notation* is:

$$
\begin{aligned}
& \mathbf{1 \times 2^4 = 1 \times 16 = 16} \\
+ & \mathbf{1 \times 2^3 = 1 \times 8 \quad = 8} \\
+ & \mathbf{1 \times 2^2 = 1 \times 4 \quad = 4} \\
+ & \mathbf{0 \times 2^1 = 1 \times 2 \quad = 0} \\
+ & \mathbf{1 \times 2^0 = 1 \times 1 \quad = 1}
\end{aligned}
$$

So, **11101** in base 2 is 16 + 8 + 4+ 0 + 1 = **29** in base 10

Convenient Table…

HEXADECIMAL	BINARY
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

HEXADECIMAL (Decimal)	BINARY
A (10)	1010
B (11)	1011
C (12)	1100
D (13)	1101
E (14)	1110
F (15)	1111

Always Helpful to Know…

N	2^N
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024 = 1 kilobits

N	2^N
11	2048 = 2 kb
12	4 kb
13	8 kb
14	16 kb
15	32 kb
16	64 kb
17	128 kb
18	256 kb
19	512 kb
20	1024 kb = 1 megabits

N	2^N
21	2 Mb
22	4 Mb
23	8 Mb
24	16 Mb
25	32 Mb
26	64 Mb
27	128 Mb
28	256 Mb
29	512 Mb
30	1 Gb

Converting Binary to Octal and Hexadecimal

(or any base that's a power of 2)

NOTE THE FOLLOWING:

- Binary is 1 bit
- Octal is 3 bits
- Hexadecimal is 4 bits

- Use the "group the bits" technique
  - Always start from the *least significant digit*
  - Group every 3 bits together for bin → oct
  - Group every 4 bits together for bin → hex

Converting Binary to Octal and Hexadecimal

- Take the example: ***10100110***

...to octal:

1 0	1 0 0	1 1 0
2	***4***	***6***

246 in octal

...to hexadecimal:

1 0 1 0	0 1 1 0
10	***6***

A6 in hexadecimal

Converting Decimal to Other Bases

Algorithm for converting number in base 10 to other bases

While (the quotient is not zero)

1. Divide the decimal number by the new base
2. Make the remainder the next digit to the left in the answer
3. Replace the original decimal number with the quotient
4. Repeat until your quotient is zero

Example: What is 98 (base 10) in base 8?

98 / 8 = 12 R 2

12 / 8 = 1 R 4

1 / 8 = 0 R 1

1 4 2

In-Class Exercise: Converting Decimal into Binary & Hex

Convert 54 (base 10) into binary and hex:

- 54 / 2 = 27 R 0
- 27 / 2 = 13 R 1
- 13 / 2 = 6 R 1
- 6 / 2 = 3 R 0
- 3 / 2 = 1 R 1
- 1 / 2 = 0 R 1

Sanity check:
110110
= 2 + 4 + 16 + 32
= 54

54 (decimal) = 110110 (binary)
= 36 (hex)

Binary Logic Refresher
NOT, AND, OR

X	NOT X $\overline{X}$
0	1
1	0

X	Y	X AND Y X && Y X.Y
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	X OR Y X || Y X + Y
0	0	0
0	1	1
1	0	1
1	1	1

Binary Logic Refresher
Exclusive-OR (XOR)

The output is "1" only if the inputs are opposite

X	Y	X XOR Y $X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0

Bitwise NOT

- Similar to logical NOT (!), except it works on a bit-by-bit manner

- In C/C++, it's denoted by a tilde: **~**

```
~(1001) = 0110
```

Exercises

- Sometimes hexadecimal numbers are written in the **0xhh** notation, so for example:

    The hex 3B would be written as 0x3B

- What is ~(0x04)?
  - Ans: 0xFB
- What is ~(0xE7)?
  - Ans: 0x18

Bitwise AND

- Similar to logical AND (&&), except it works on a bit-by-bit manner

- In C/C++, it's denoted by a single ampersand: &

```
(1001 & 0101) = 1 0 0 1
                & 0 1 0 1

                = 0 0 0 1
```

Exercises

- What is (0xFF) & (0x56)?
  - Ans: 0x56
- What is (0x0F) & (0x56)?
  - Ans: 0x06
- What is (0x11) & (0x56)?
  - Ans: 0x10
- Note how & can be used as a “masking” function

Bitwise OR

- Similar to logical OR (||), except it works on a bit-by-bit manner
- In C/C++, it's denoted by a single pipe: |

```
(1001 | 0101) = 1 0 0 1
                | 0 1 0 1

                = 1 1 0 1
```

Exercises

- What is (0xFF) | (0x92)?
  - Ans: 0xFF
- What is (0xAA) | (0x55)?
  - Ans: 0xFF
- What is (0xA5) | (0x92)?
  - Ans: B7

Bitwise XOR

- Works on a bit-by-bit manner
- In C/C++, it's denoted by a single carat: ^

```
(1001 ^ 0101) =  1 0 0 1
               ^  0 1 0 1

                = 1 1 0 0
```

Exercises

- What is (0xA1) ^ (0x13)?
  - Ans: 0xB2
- What is (0xFF) ^ (0x13)?
  - Ans: 0xEC
- Note how (1^b) is always ~b
  and how (0^b) is always b

Bit Shift ***Left***

- Move all the bits N positions to the left
- What do you do the positions now empty?
  – You put in N 0s

- Example: Shift “1001” 2 positions to the left

1001 << 2 = **100100**

- Why is this useful as a form of multiplication?

Multiplication by Bit Left Shifting

- Veeeery useful in CPU (ALU) design
  - Why?

- Because you don't have to design a multiplier
- You just have to design a way for the bits to shift

Bit Shift ***Right***

- Move all the bits N positions to the ***right***, subbing-in either N 0s or N 1s on the left
- Takes on two different forms

- Example: Shift “1001” 2 positions to the right

  1001 >> 2 = either **0010** or **1110**

- The information carried in the last 2 bits is *lost*.
- If Shift Left does multiplication, what does Shift Right do?
  - It divides, but it truncates the result

Two Forms of Shift Right

- Subbing-in 0s makes sense
- What about subbing-in the leftmost bit with 1?

- It's called ***"arithmetic"*** shift right:

  1100 (arithmetic) >> 1 = 1110

- It's used for *twos-complement* purposes
  - *What?*

Negative Numbers in Binary

- So we know that, for example, $6_{(10)} = 110_{(2)}$
- But what about $-6_{(10)}$???

- What if we added one more bit on the far left to denote "negative"?
  – i.e. becomes the new MSB

- So: **110** (+6) becomes **1110** (–6)
- But this leaves a lot to be desired
  – Bad design choice…

Twos Complement Method

- This is how Twos Complement fixes this.
- Let's write out $\mathbf{-6_{(10)}}$ in 2s-Complement binary in **4 bits**:

First take the unsigned (abs) value (i.e. 6) and convert to binary:	**0110**
Then negate it (i.e. do a "NOT" function on it):	**1001**
Now add 1:	**1010**

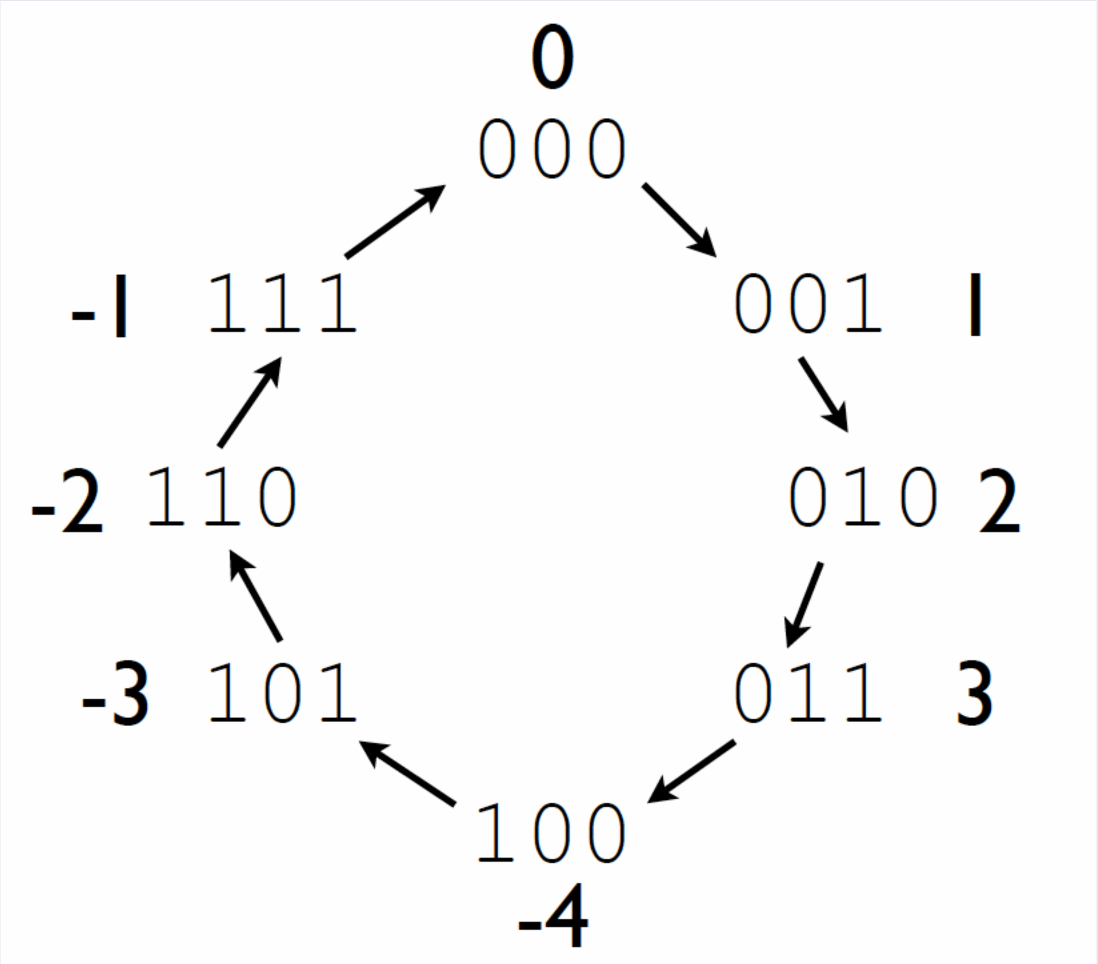
So, $-6_{(10)} = 1010_{(2)}$

Let's do it Backwards… By doing it THE SAME EXACT WAY!

- 2s-Complement to Decimal method **is the same!**

- Take **1010** from our previous example
- Negate it and it becomes **0101**
- Now add 1 to it & it becomes **0110**, which is $6_{(10)}$

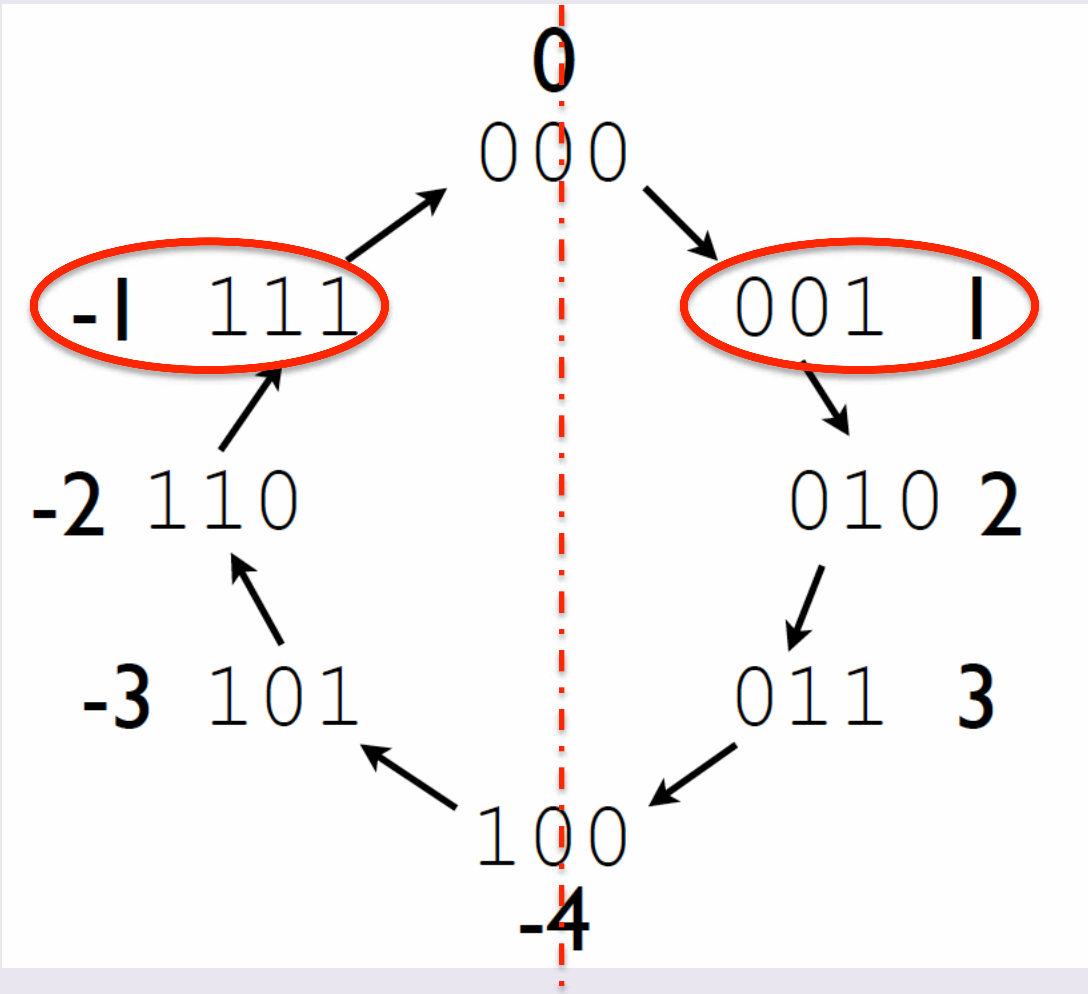
Another View of 2s Complement



NOTE:

In Two's Complement, if the number's MSB is "1", then that means it's a negative number and if it's "0" then the number is positive.

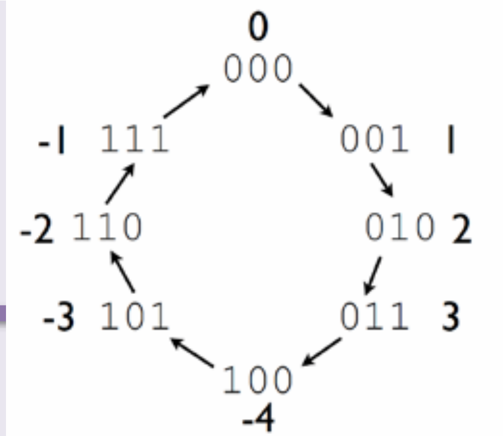
Another View of 2s Complement



NOTE:
Opposite numbers show up as symmetrically opposite each other in the circle.

NOTE AGAIN:
When we talk of 2s complement, we must also mention the number of bits involved

Ranges



- The *range* represented by number of bits differs between positive and negative binary numbers

- Given **N** bits, the range represented is:

  0 to **$+2^N - 1$** *for positive numbers*

and **-2^{N-1}** to **$+2^{N-1} - 1$**

for 2's Complement negative numbers

Addition

- We have an elementary notion of adding single digits, along with an idea of carrying digits
  - Example: when adding 3 to 9, we put forward 2 and carry the 1 (i.e. to mean 12)

- We can build on this notion to add numbers together that are more than one digit long

- Example:

$$
\begin{array}{rccc}
 & 1 & 1 & \\
 & 1 & 2 & 3 \\
+ & 3 & 8 & 9 \\
\hline
 & 5 & 1 & 2
\end{array}
$$

Addition in Binary

- Same mathematical principal applies

```
carry carry carry carry
1     1     1     1
        0 0 1 1          3
      + 1 1 0 1       + 13
      ---------       ----
      1 0 0 0 0         16
```

Exercises

Implementing an 8-bit adder:

- What is (0x52) + (0x4B) ?
  - Ans: 0x9D, output carry bit = 0
- What is (0xCA) + (0x67)?
  - Ans: 0x31, output carry bit = 1

YOUR TO-DOs

- Assignment #1
  - Look for it over the weekend on the class website
  - LAB #1 is on MONDAY!

- Next week, we will discuss more Arithmetic topics and start exploring Assembly Language
  - Do your readings!
    (again: found on the class website)

</LECTURE>