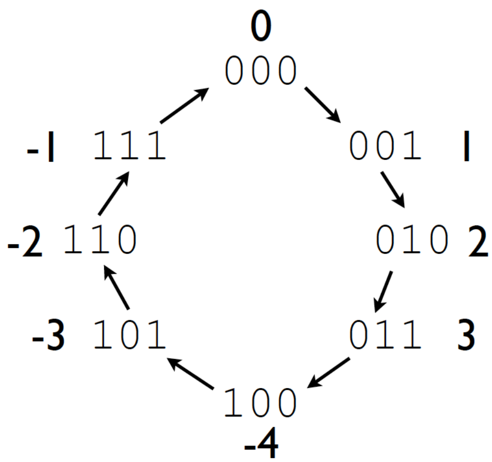


Binary Arithmetic

CS 64: Computer Organization and Design Logic

Lecture #2

Winter 2020

Ziad Matni, Ph.D.
Dept. of Computer Science, UCSB

```
        1
 10001001
 10010101
---------
100111110
```

Administrative Stuff

- **The class is still full… ☹**

- Did you check out the syllabus?
- Did you check out the class website?
- Did you check out Piazza (and get access to it)?
- Did you check out Gradescope (and get an account on it)?
- Do you understand how you will be submitting your assignments?

Lecture Outline

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

So Why Digital Design?

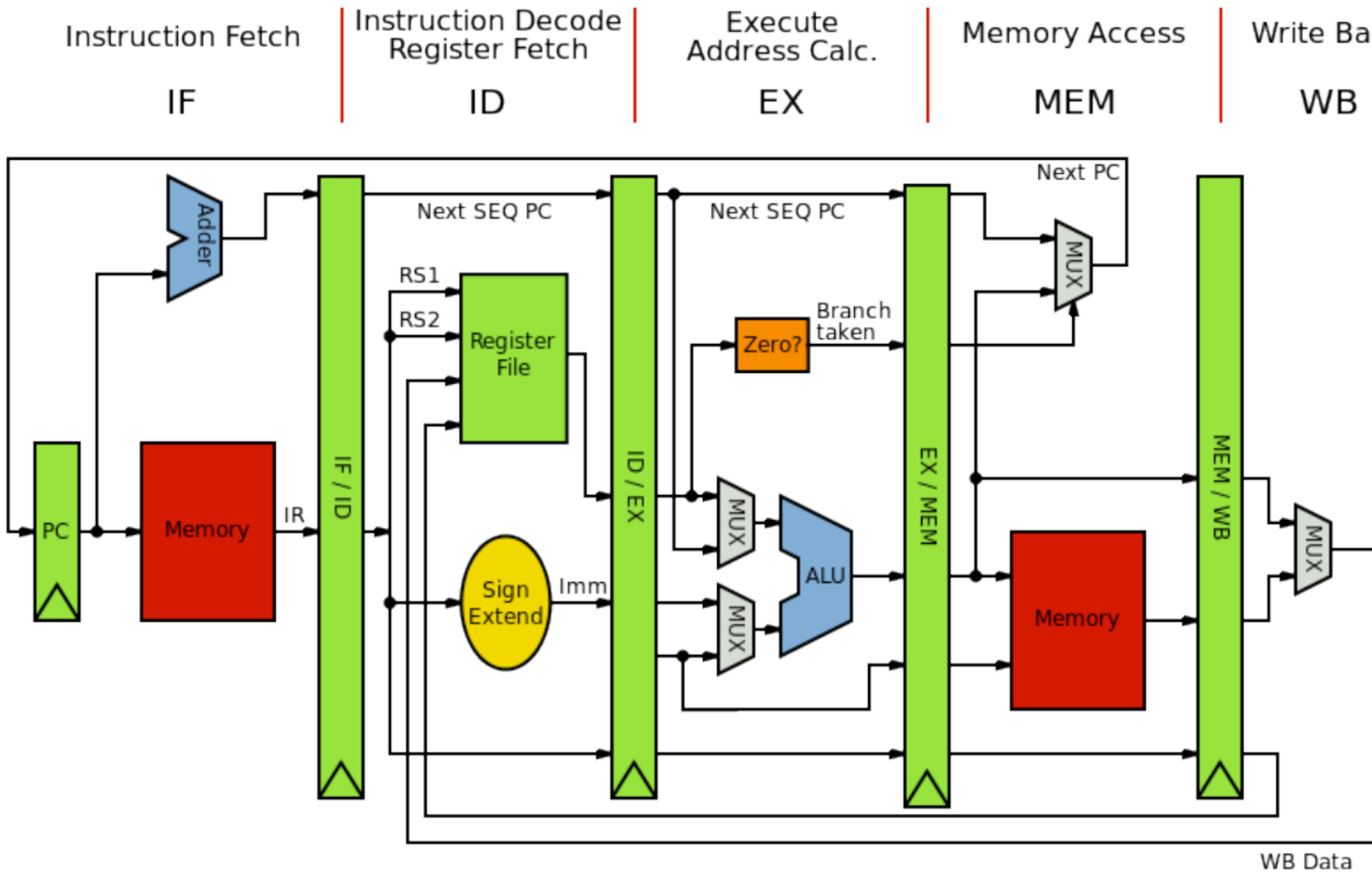
- Because that’s where the “magic” happens

- Logical decisions are made with 1s and 0s

- Physically (*engineering-ly?*), this comes from electrical currents switching one way or the other & also how semiconducting material work, etc…

- But we don’t have to worry about the physics part in this class…

Digital Design of a CPU (Showing Pipelining)



Digital Design in this Course

- We will not go into “deep” dives with digital design in this course
  - For that, check out CS 154 (Computer Architecture) and also courses in ECE

- We will, however, delve deep enough to understand the ***fundamental*** workings of digital circuits and how they are used for ***computing purposes***.

COMPUTERS ARE DIGITAL MACHINES

Counting Numbers in Different Bases

- We "normally" count in 10s
  - Base 10: **decimal** numbers
  - We use *10* numerical symbols in Base *10*: "**0**" thru "**9**"
- Computers count in 2s
  - Base 2: **binary** numbers
  - We use *2* numerical symbols in Base *2*: "**0**" and "**1**"
  - Represented with 1 bit (Note: $2^1 = 2$)

Counting Numbers in Different Bases

Other convenient bases in computer architecture:

- Base 8: **octal** numbers
  - Number symbols are 0 thru 7
  - Represented with **3 bits** ($2^3 = 8$)

- Base 16: **hexadecimal** numbers
  - Number symbols are 0 thru F:

    including A = 10, B = 11, C = 12, D = 13, E = 14, F = 15
  - Represented with **4 bits** ($2^4 = 16$)

- **Why are 4 bit representations convenient???**

What's in a Number?

642

What *is* that???

Well, what NUMERICAL BASE are you expressing it in?

Positional Notation

642 in base 10 (decimal) can be described in "*positional notation*" as:

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

6	4	2
10^2	10^1	1

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

Positional Notation

642 in base 16 (hexadecimal) can be described in "*positional notation*" as:

$$
\begin{aligned}
6 \times \underline{16}^2 &= 6 \times 256 = 1536 \\
+\, 4 \times \underline{16}^1 &= 4 \times 16 \quad = 64 \\
+\, 2 \times \underline{16}^0 &= 2 \times 1 \quad\ \ = 2 \qquad = 1602 \text{ in base 16}
\end{aligned}
$$

6	4	2
16^2	16^1	1

Positional Notation

This is how you convert any base number into decimal!

Each digit gets multiplied by B^N

Where:

B = the base

N = the position of the digit

*Example: given the number **642** in **base 8**:*

Number in decimal = $\mathbf{6} \times 8^2 + \mathbf{4} \times 8^1 + \mathbf{2} \times 8^0$

*= **418***

Positional Notation in Binary

11101 in base 2 *positional notation* is:

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

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

This is easy if you remember your powers of 2

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

Another 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

Converting Binary to Octal and Hexadecimal
(or any base that's a power of 2)

NOTE THE FOLLOWING:

- Binary is 1 bit per digit (0 or 1)
- Octal is 3 bits per digit (0, 1, 2, 3, 4, 5, 6 or 7)
- Hexadecimal is 4 bits per digit (0 thru F)

- 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 (group in 3s):

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

246 in octal

REMEMBER:
Start your grouping from the Least Significant Bit (LSB)!!!

...to hexadecimal (group in 4s):

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

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 **-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**

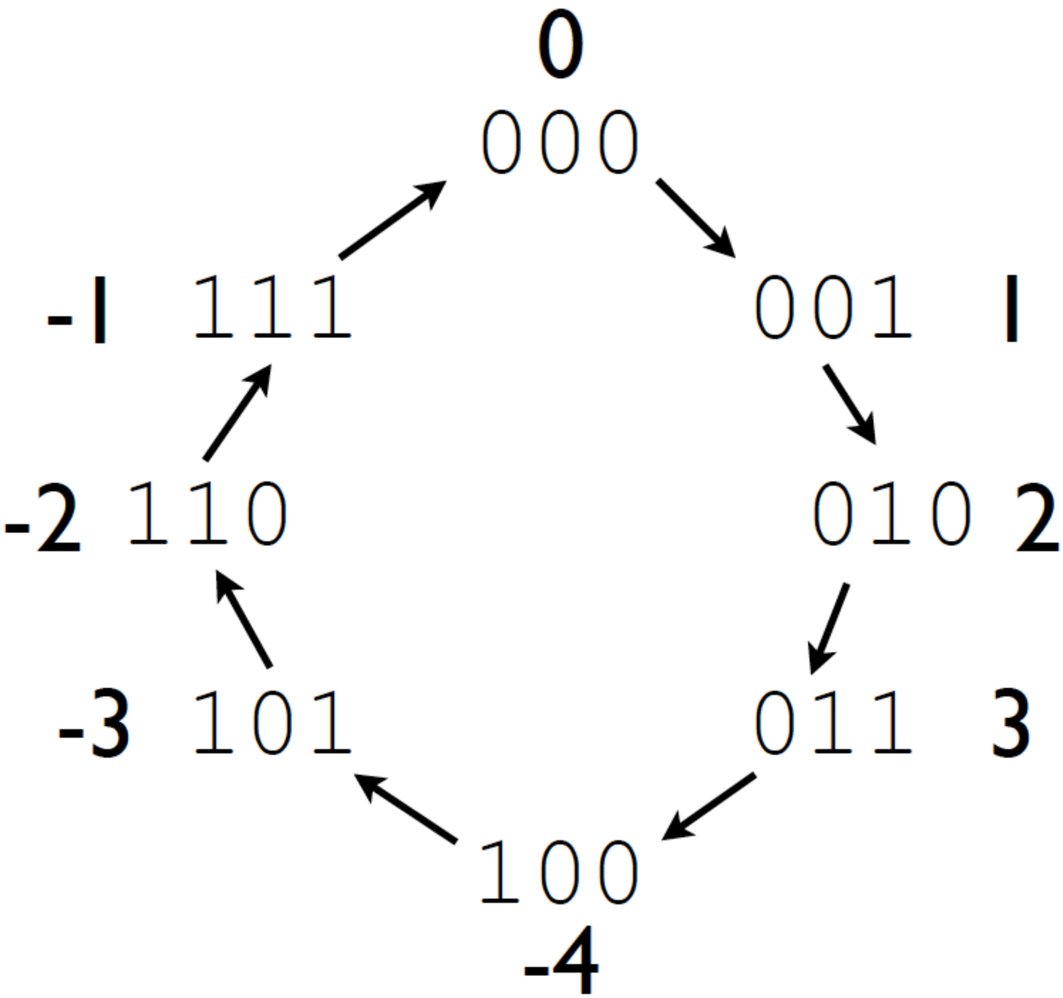
So, $-6_{(10)} = 1010_{(2)}$ according to this rule

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 $6_{(10)}$

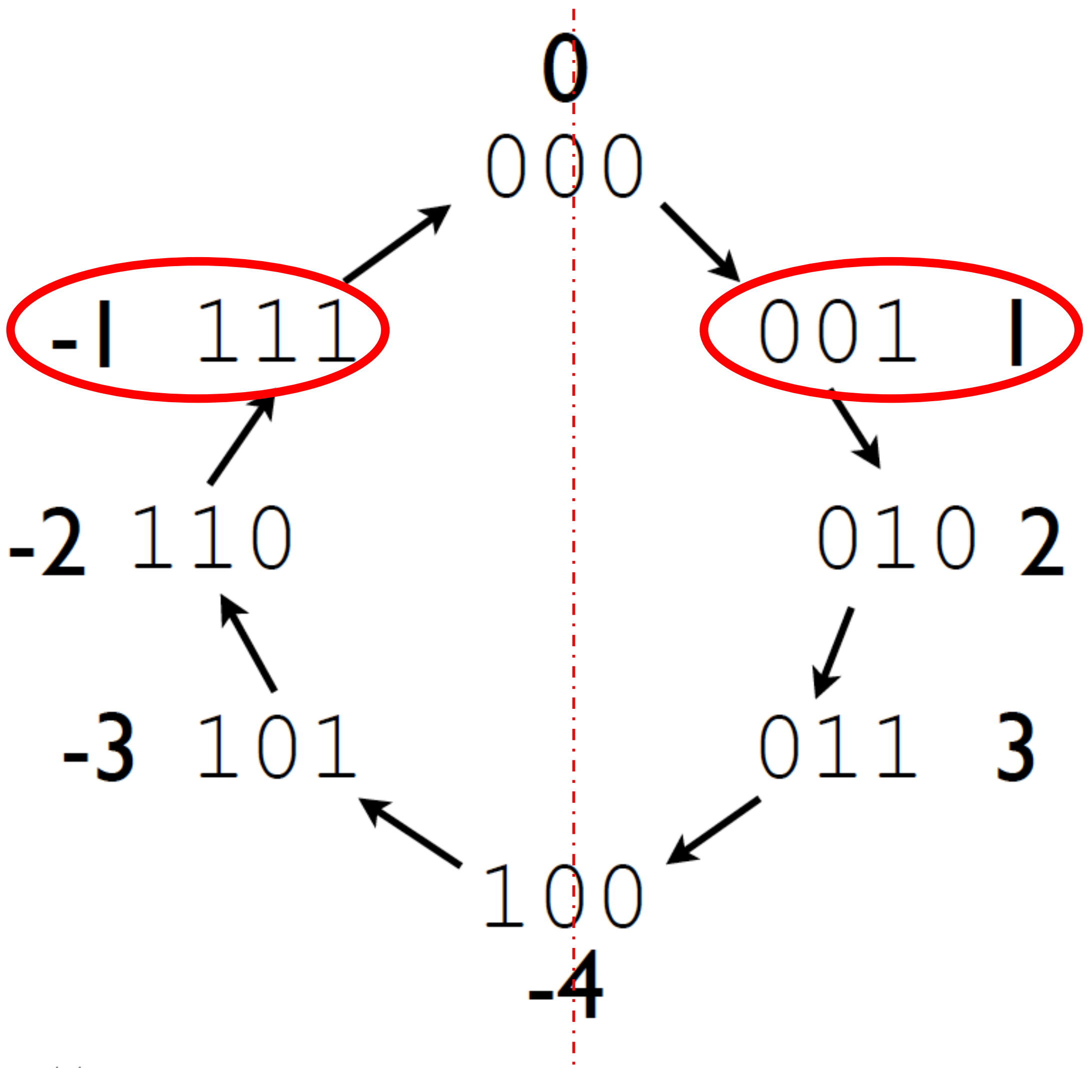
Another View of 2s Complement



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.

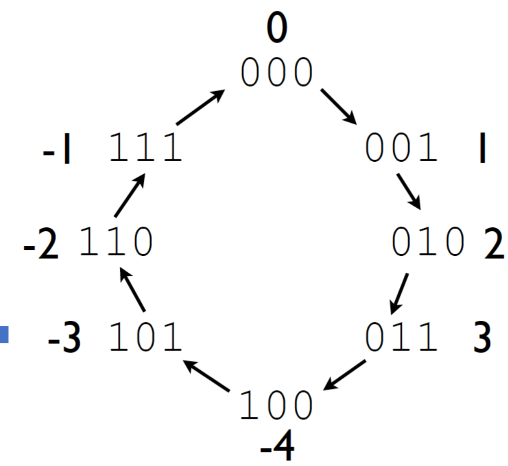
Another View of 2s Complement



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:

```
   1 1      carried digits
   1 2 3
+  3 8 9
   -----
   5 1 2
```

Addition in Binary

- Same mathematical principal applies

Q: *What's being assumed here???*

A: *That these are purely positive numbers*

```
carry carry carry carry
  1    1    1    1
         0  0  1  1
+        1  1  0  1
--------------------
     1  0  0  0  0
```

$$\begin{array}{r} 3 \\ +\ 13 \\ \hline 16 \end{array}$$

***Theoretically**, I can add any binary no. with N1 digits to any other binary no. with N2 digits.*

BUT THERE IS A PRACTICAL LIMITATION!
***Practically**, a CPU must have a defined no. of digits that it's working with.*

WHY???

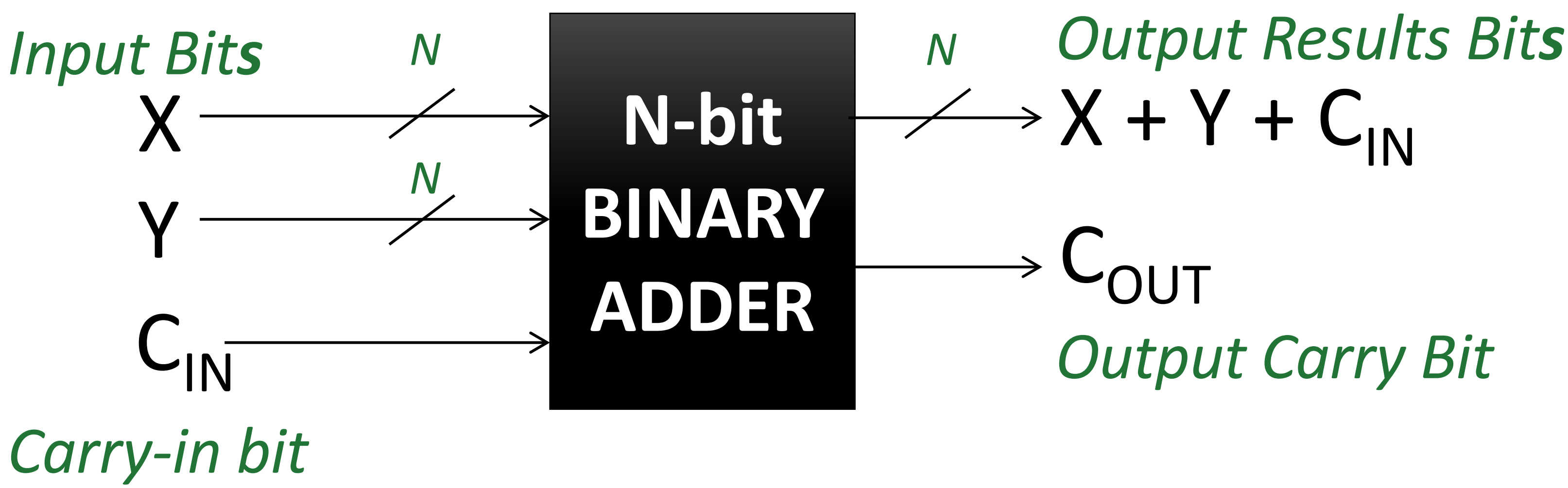
Exercises

*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

Black Box Perspective of ANY N-Bit Binary Adder



This is a useful perspective for either writing
an N-bit adder function in code,
or for designing the actual digital circuit that does this!

Output Carry Bit Significance

- For unsigned (i.e. positive) numbers,
  C_{OUT} = 1 means that the result **did not fit into the number of bits allotted**

- Could be used as an error condition for software
- For example, **you've designed a 16-bit adder** and during some calculation of positive numbers, your carry bit/flag goes to "1". Conclusion?
- Your result is
  outside the maximum range allowed by 16 bits.

Carry vs. Overflow

- The **carry** bit/flag works for – and is looked at – only for ***unsigned (positive)*** numbers

- A similar bit/flag works is looked at for if ***signed*** (two's complement) numbers are used in the addition: the **overflow** bit

Overflow: for Negative Number Addition

- What about if I'm adding two ***negative*** numbers?
  Like: 1001 + 1011?
  - Then, I get: 0100 with the extra bit set at 1
  - 1 0100 is the same as 16 + 8 = 24
  - *Sanity Check*:
    That's adding (-7) + (-5), so I expected **-12**, NOT 24!!!
    so what's wrong here?

- The answer is that -12 is beyond the capability of 4 bits in 2's complement!!!

How Do We Determine if Overflow Has Occurred?

- When adding 2 *signed* numbers: ***x + y = s***

	if	$\mathbf{x, y > 0}$	AND $\mathbf{s < 0}$
OR	if	$\mathbf{x, y < 0}$	AND $\mathbf{s > 0}$

Then, overflow has occurred

Example 1

Side-note:
What is the range of signed numbers w/ 8 bits?

-2^7 to $(2^7 - 1)$, or
-128 to 127

Add: -39 and 92 in *signed* 8-bit binary

```
              1  Cin_signed_bit
             1101 1001
-39
 92          0101 1100
---          ---------
 53
            10011 0101
  Cout
```

That's 53 in signed 8-bits! Looks ok!

There's a carry-out (we don't care)
But there is no overflow (V)
Note that V = 0, while Cout = 1 and Cin_signed_bit = 1

Example 2

$V = Cout \oplus Cin\_signed\_bit$

Add: 104 and 45 in *signed* 8-bit binary

Cin_signed_bit

```
         1
104    0110 1000
 45    0010 1101
---    ---------
149    1001 0101
```

Cout = 0

$\underbrace{1001\ 0101}$ *That's NOT 149 in signed 8-bits!*

There's no carry-out (again, we don't care)

But there <u>**is**</u> overflow!

Given that this binary result is not 149, but actually <u>**–107**</u> !

Note that V = 1, while Cout = 0 and Cin_signed_bit = 1

YOUR TO-DOs

- Do your reading for next week's classes
  - Ch. 2.6
- Start on Assignment #1 for lab
  - I'll put it up on our main website this afternoon
  - Meet up in the lab this Thursday
  - Do the lab assignment: setting up CSIL + exercises
  - You have to submit it as a **PDF** using ***Gradescope***
  - Due next week on **Tuesday, 1/14, by 11:59:59 PM**

</LECTURE>

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 number of 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 (which is relatively easier)

Bit Shift *Right*

- Move all the bits N positions to the ***right***, subbing-in either N number of 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?*

Exercise

- Given an argument that's a 32-bit integer number **i**, write a function in C++ that can isolate the bit in **position 5** of that integer and print it.

- Example: **i** = 1266
- In 32-bits of binary, that's:

  0000 0000 0000 0000 0000 0100 1111 0010
- So, the bit in position 5 is the highlighted one (it's 1)
- So your code should print out "**1**"

- Answer:

```
void print5(int i):
{
    i >> 5;
    i = i & 1;
    cout << i;
}
```