Binary Arithmetic: Bit Shifting, 2s Complement Intro to Assembly Language

CS 64: Computer Organization and Design Logic
Lecture #3
Winter 2019

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

Why do CPU programmers celebrate Christmas and Halloween on the same day?

Because Oct-31 = Dec-25

Administrative Stuff

- The class is still full… waitlist is closed… ☹
- Assignment 2 is this Thursday
- Linux Questions
- Reminder of Office Hours!
  - Prof. Matni Th. 1 – 2:30 PM *SSMS 4409*
  - TA Bay-Yuan Fr. 11 AM – 1 PM *Trailer 936*
  - TA Shiyu Fr. 3 – 5 PM *Trailer 936*

Any Questions From Last Lecture?

5-Minute Pop Quiz!!!

YOU MUST SHOW YOUR WORK!!!

1. Calculate and give your answer in hexadecimal:

   ~(0x3E | 0xFC)

2. Convert from binary to decimal AND to hexadecimal. Use any technique(s) you like:

   a) **1001001**

   b) **10010010**

Answers…

1. Calculate and give your answer in hexadecimal:

   ~(0x3E | 0xFC) **= ~(0xFE) = 0x01**

2. Convert from binary to decimal AND hexadecimal. Use any technique you like:

   a) 1001001 **= 0100 1001 = 0x49**
   = 1 + 8 + 64 = 73

   b) 10010010 **= 1001 0010 = 0x92**
   I see that it's (1001001) x 2 = 146

Lecture Outline

- Bit shift operations
- Two’s complement
- Addition and subtraction in binary

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?*

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**

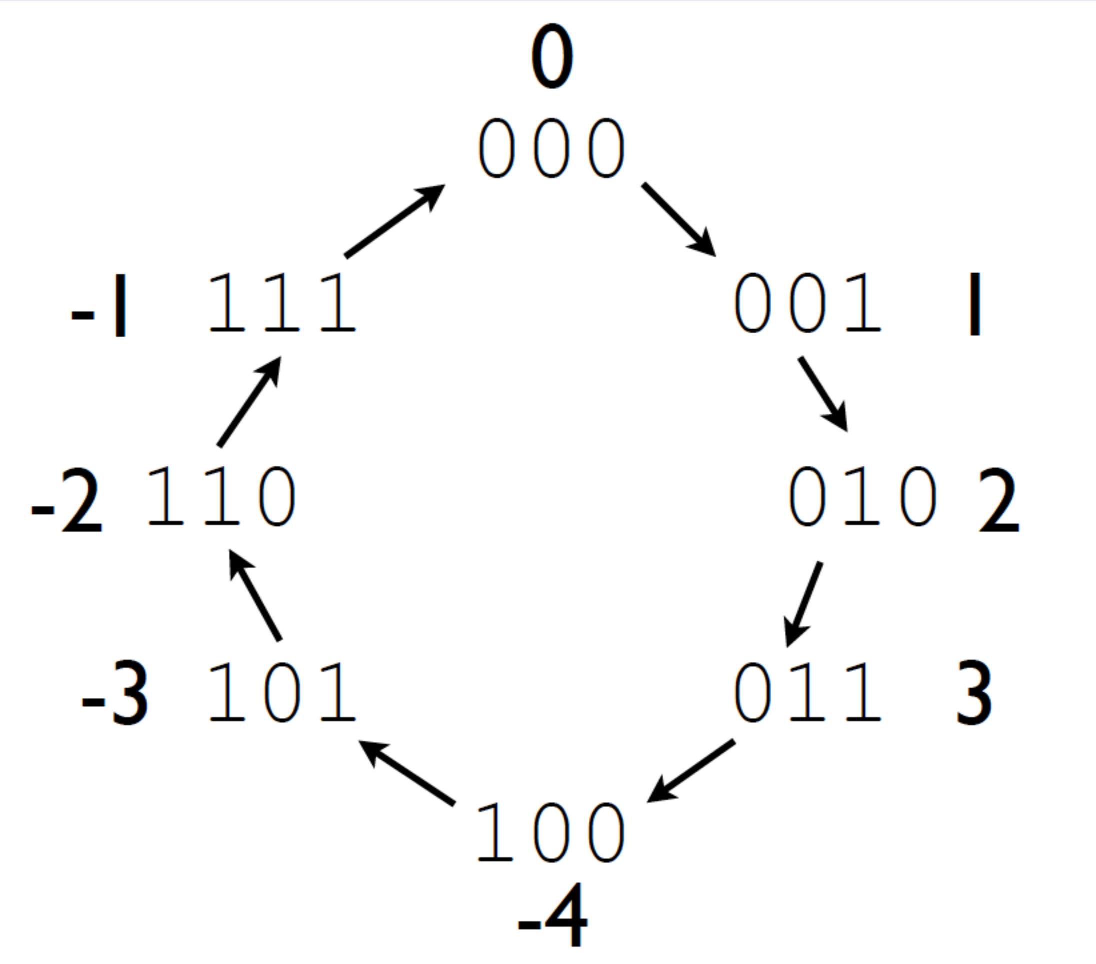
So, $-6_{(10)} = 1010_{(2)}$ according to this rule

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 $6_{(10)}$

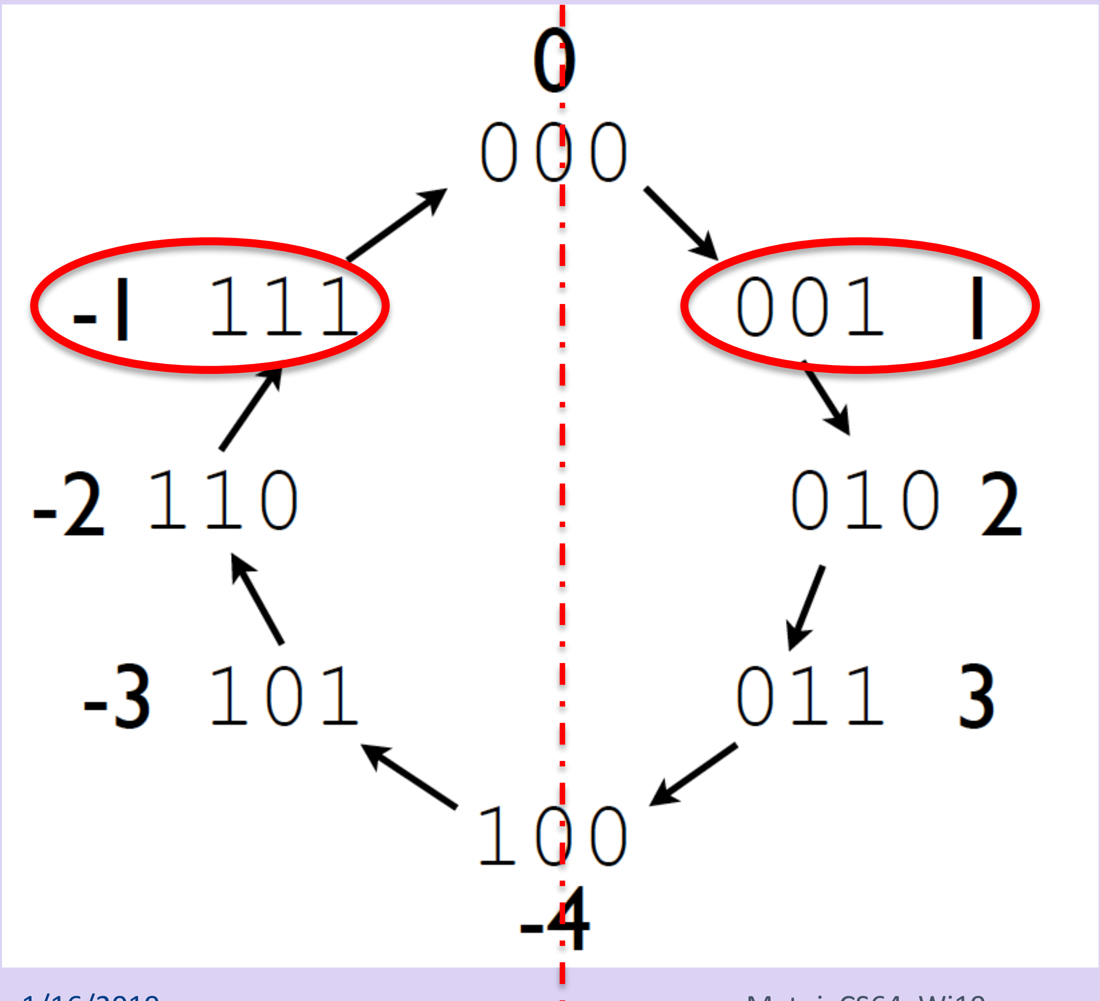
Another View of 2s Complement



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.

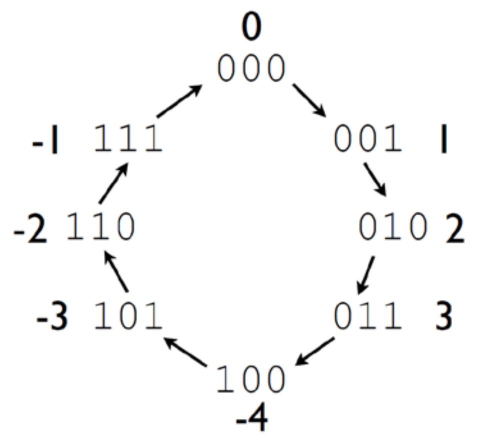
Another View of 2s Complement



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
```

```
     3
+   13
------
    16
```

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

***Practically**, a CPU must have a defined no. of digits that it's working with.*

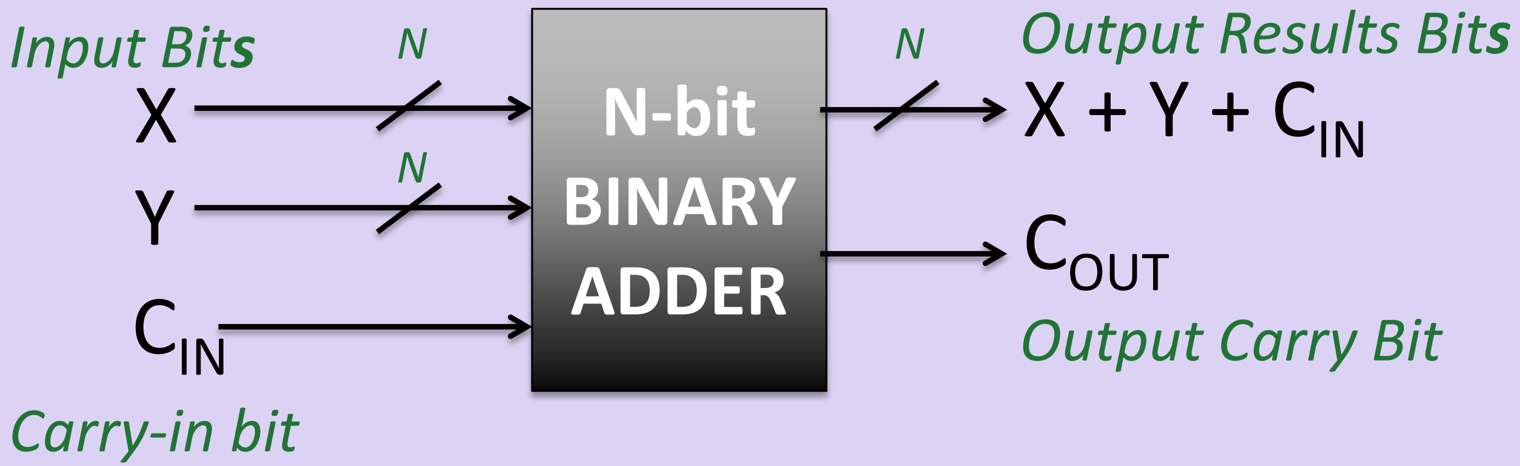
<u>WHY???</u>

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 – <u>only</u> 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
  - Sanity Check:
    That's adding (-7) + (-5), so I expected -12, so what's wrong here?
  - The answer is beyond the capability of 4 bits in 2's complement!!!

- The extra bit in this case is called **overflow** and it indicates that the addition of negative numbers has resulted in a number that's

  ***beyond the range of the given bits*.**

How Do We Determine if Overflow Has Occurred?

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

if **x, y > 0** AND **s < 0**

OR if **x, y < 0** AND **s > 0**

Then, overflow has occurred

Example 1

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!

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

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

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⊕Cin_signed_bit

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

```
                  1 ← Cin_signed_bit
 104          0110 1000
  45          0010 1101
 ---          ---------
 149          1001 0101
      Cout = 0
```

That's NOT 149 in signed 8-bits!

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

But there **is** overflow!

Given that this binary result is not 149, but actually **–107** !

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

YOUR TO-DOs

- Assignment #2 coming up!

- Next lesson: **Assembly Language!**
  - Do your readings!!

</LECTURE>