ECE 550D
Fundamentals of Computer Systems and Engineering

Fall 2023

Digital Arithmetic

Xin Li & Dawei Liu
Duke Kunshan University

Slides are derived from work by
Andrew Hilton, Tyler Bletsch and Rabih Younes (Duke)

Last Time in ECE 550….

- Who can remind us what we talked about last time?
  - Numbers
    - Binary
    - Hex
    - One hot
  - Binary Numbers and Math
    - Overflow

Designing a 1-bit adder

- What boolean function describes the low bit?
  - XOR
- What boolean function describes the high bit?
  - AND

```
0 + 0 = 00
0 + 1 = 01
1 + 0 = 01
1 + 1 = 10
```


Designing a 1-bit adder

- Remember how we did binary addition:
  - Add the **two bits**
  - Do we have a **carry-in** for this bit?
  - Do we have to **carry-out** to the next bit?

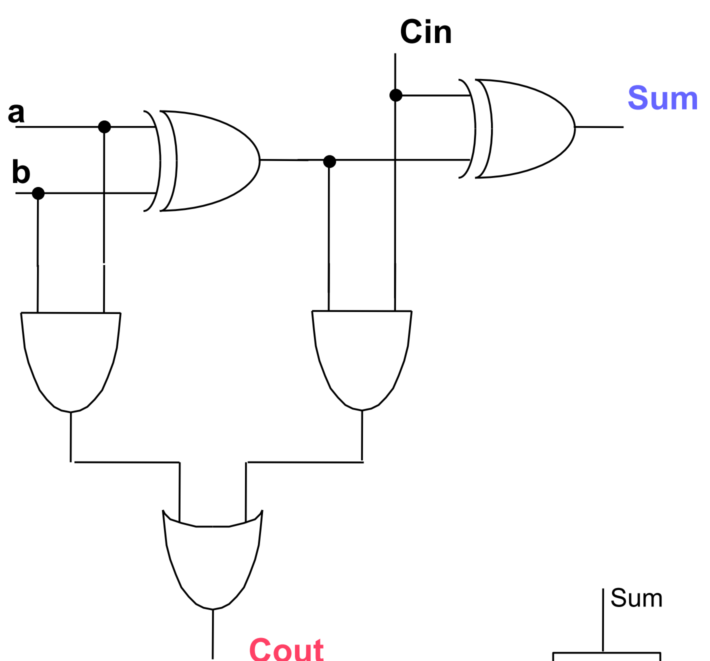
```
 01101100
  01101101
 +00101100
  10011001
```

Designing a 1-bit adder

- So we'll need to add three bits (including carry-in)
- Two-bit output is the **carry-out** and the **sum**

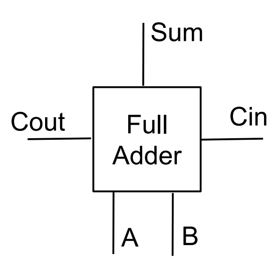
a		b		C_{in}		
0	+	0	+	0	=	00
0	+	0	+	1	=	01
0	+	1	+	0	=	01
0	+	1	+	1	=	10
1	+	0	+	0	=	01
1	+	0	+	1	=	10
1	+	1	+	0	=	10
1	+	1	+	1	=	11

A 1-bit Full Adder

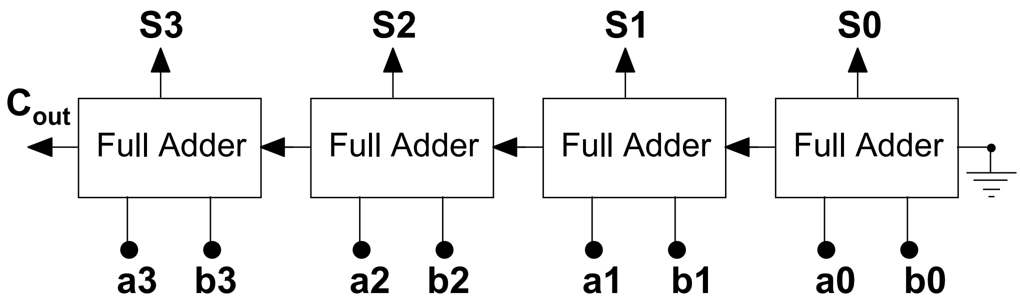



```
 01101100
 01101101
+00101100
 10011001
```

a	b	C_{in}	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

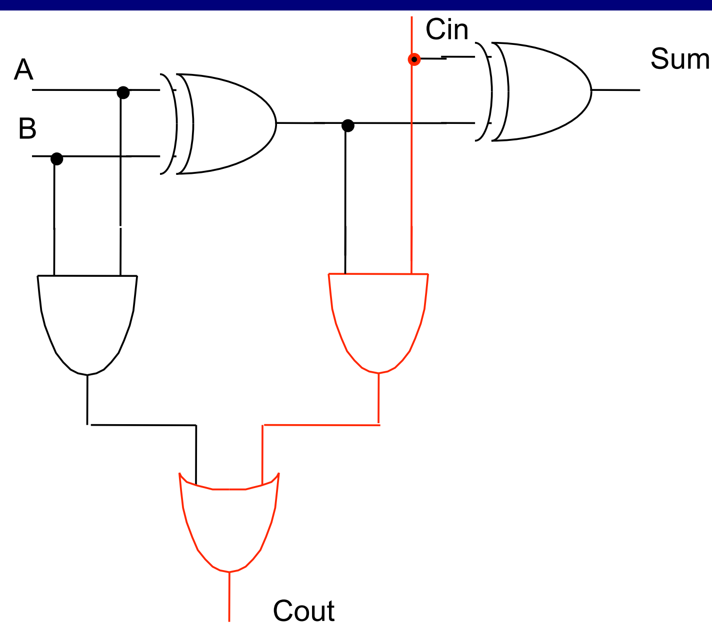


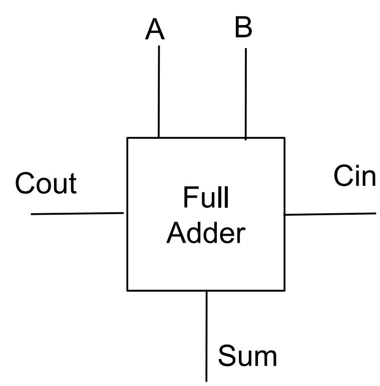
Ripple Carry



- Full Adder = Add 1 Bit
  - Can chain together to add many bits
  - Upside: Simple
  - Downside?
    - Slow. Let's see why.

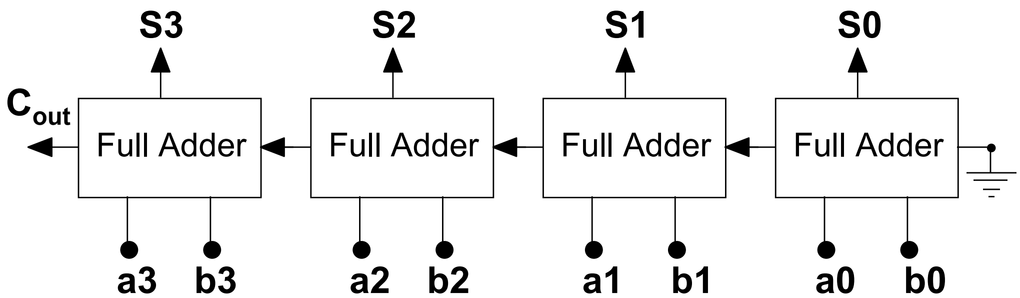
Full adder delay





- Cout depends on Cin
  - 2 "gate delays" through full adder for carry

Ripple Carry



- Carries form a chain
  - CO of bit N is CI of bit N+1
- For few bits (e.g., 4) no big deal
  - For realistic numbers of bits (e.g., 32, 64), slow

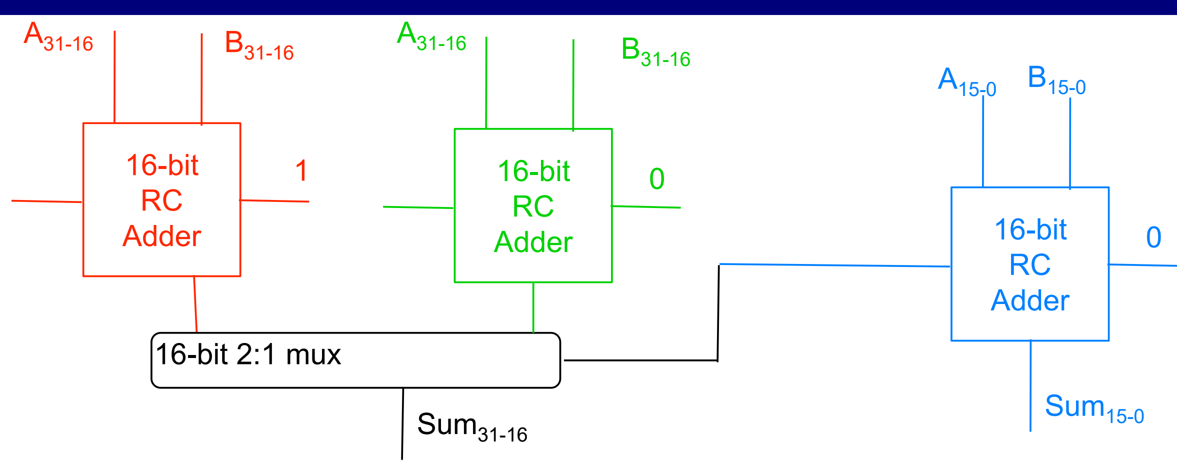
Adding

- Adding is important
  - Want to fit add in single clock cycle
    - (More on clocking soon)
    - Why? Add is ubiquitous
- Ripple Carry is slow
  - Maybe can do better?
  - But seems like Cin always depends on prev Cout
  - …and Cout always depends on Cin…

Hardware != Software

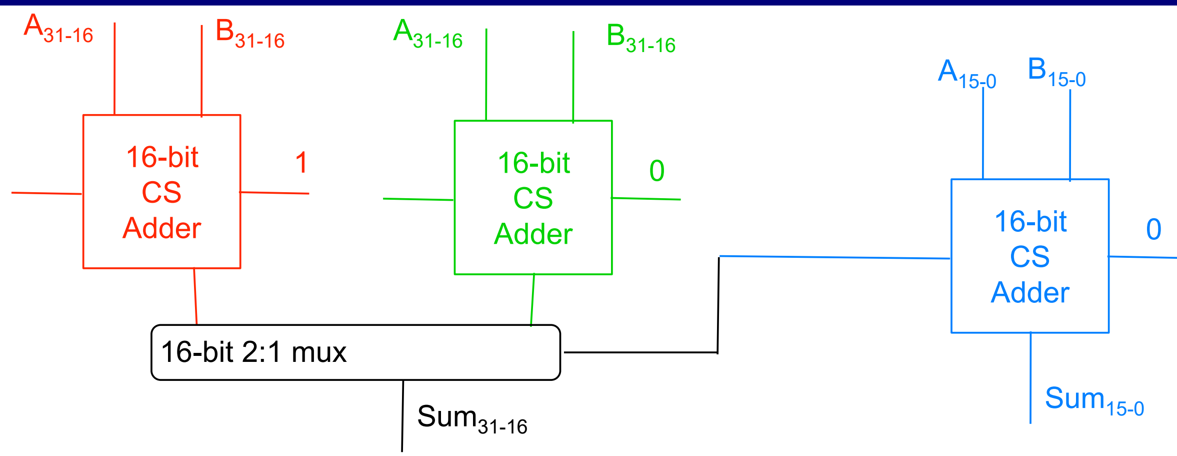
- If this were software, we'd be out of luck
  - But hardware is different
  - Parallelism: can do many things at once
  - Speculation: can guess

Carry Select



- Do three things at once (32 gates)
  - Add low 16 bits
  - Add high 16 bits assuming CI = 0
  - Add high 16 bits assuming CI =1
- Then pick correct assumption for high bits

Carry Select



- Could apply same idea again
  - Replace 16-bit RC adders with 16-bit CS adders
    - Reduce delay for 16 bit add from 32 to 18
    - Total 32 bit adder delay = 20
- So… just go nuts with this right?

Tradeoffs

- Tradeoffs in doing this
  - Power and Area (~= number of gates)
    - Roughly double every "level" of carry select we use
  - Less return on increase each time
    - Adding more mux delays
  - Wire delays increase with area
    - Not easy to count in slides
    - But will eat into real performance
- Fancier adders exist:
  - Carry-lookahead, conditional sum adder, carry-skip adder, carry-complete adder, etc…

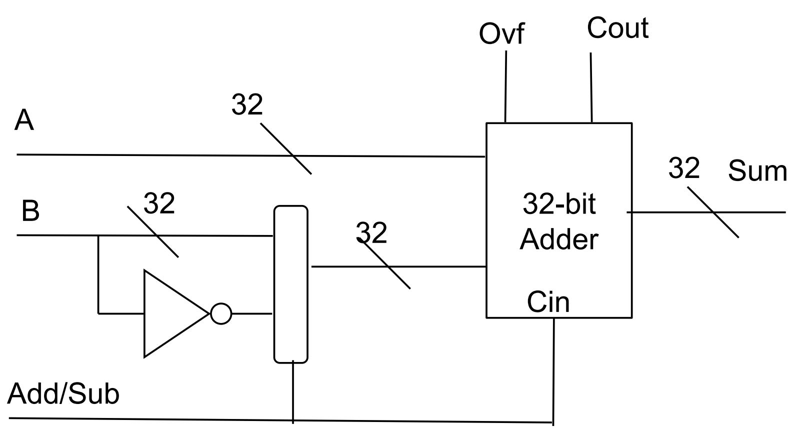
Recall: Subtraction

- 2's complement makes subtraction easy:
  - Remember: A - B = A + (-B)
  - And: -B = ~B + 1

    ↑ that means flip bits ("not")
  - So we just flip the bits and start with CI = 1
  - Fortunate for us: **makes circuits easy**

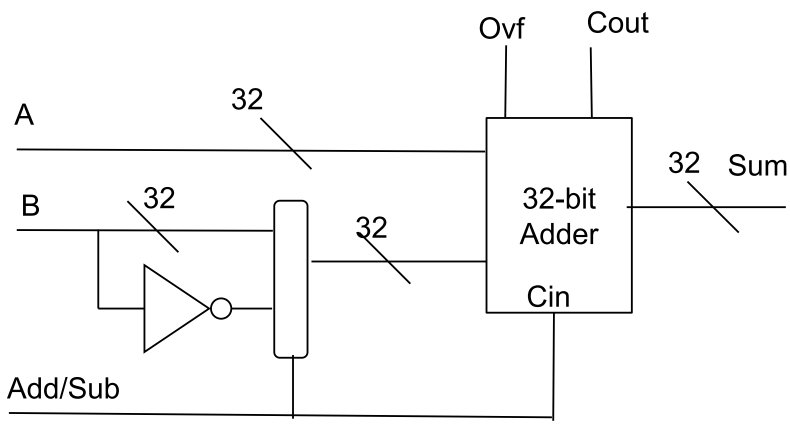
```
                             1
  0110101       ->     0110101
- 1010010            + 0101101
---------            ---------
```


32-bit Adder/subtractor



- Inputs: A, B, Add/Sub (0=Add,1 = Sub)
- Outputs: Sum, Cout, Ovf (Overflow)

32-bit Adder/subtractor



- By the way:
  - That thing has about 3,000 transistors
  - Aren't you glad we have abstraction?

Arithmetic Logic Unit (ALU)

- ALUs do a variety of math/logic
  - Add
  - Subtract
  - Bit-wise operations: And, Or, Xor, Not
  - Shift (left or right)

- Take two inputs (A,B) + operation (add,shift..)
  - Do a variety in parallel, then mux based on op

Bit-wise operations: SHIFT

- Left shift (<<)
  - Moves left, bringing in 0s at right, excess bits "fall off"
  - 10010001 << 2 = 01000100
  - x << k corresponds to x * 2^k
- Logical (or unsigned) right shift (>>)
  - Moves bits right, bringing in 0s at left, excess bits "fall off"
  - 10010001 >> 3 = 00010010
  - x >>k corresponds to x / 2^k for unsigned x
- Arithmetic (or signed) right shift (>>)
  - Moves bits right, brining in (sign bit) at left
  - 10010001 >> 3= 11110010
  - x >>k corresponds to x / 2^k for signed x

Shift: Implementation…?

- Suppose an 8-bit number

  $b_7b_6b_5b_4b_3b_2b_1b_0$

Shifted left by a 3 bit number

  $s_2s_1s_0$

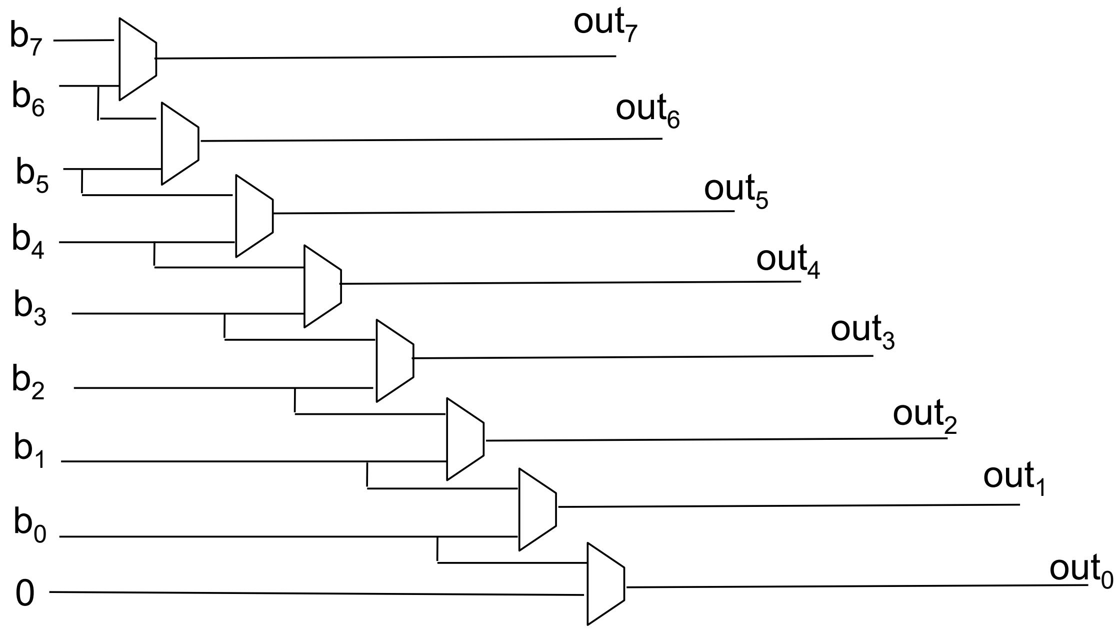
- Option 1: Truth Table?
  - 2048 rows? Not appealing

    …but you can do it. Truth table gives this expression for output bit 0:

(b0 & !b1 & !b2 & !b3 & !b4 & !b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & !b3 & !b4 & !b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & !b3 & !b4 & !b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & !b3 & !b4 & !b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & b3 & !b4 & !b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & b3 & !b4 & !b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & b3 & !b4 & !b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & b3 & !b4 & !b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & !b3 & b4 & !b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & !b3 & b4 & !b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & !b3 & b4 & !b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & !b3 & b4 & !b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & b3 & b4 & !b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & b3 & b4 & !b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & b3 & b4 & !b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & b3 & b4 & !b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & !b3 & !b4 & b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & !b3 & !b4 & b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & !b3 & !b4 & b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & !b3 & !b4 & b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & b3 & !b4 & b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & b3 & !b4 & b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & b3 & !b4 & b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & b3 & !b4 & b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & !b3 & b4 & b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & !b3 & b4 & b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & !b3 & b4 & b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & !b3 & b4 & b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & b3 & b4 & b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & b3 & b4 & b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & b3 & b4 & b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & b3 & b4 & b5 & !b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & !b3 & !b4 & !b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & !b3 & !b4 & !b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & !b3 & !b4 & !b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & !b3 & !b4 & !b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & b3 & !b4 & !b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & b3 & !b4 & !b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & b3 & !b4 & !b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & b3 & !b4 & !b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & !b3 & b4 & !b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & !b3 & b4 & !b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & !b3 & b4 & !b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & !b3 & b4 & !b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & b3 & b4 & !b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & b3 & b4 & !b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & b3 & b4 & !b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & b3 & b4 & !b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & !b3 & !b4 & b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & !b3 & !b4 & b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & !b3 & !b4 & b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & !b3 & !b4 & b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & b3 & !b4 & b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & b3 & !b4 & b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & b3 & !b4 & b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & b3 & !b4 & b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & !b3 & b4 & b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & !b3 & b4 & b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & !b3 & b4 & b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & !b3 & b4 & b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & b3 & b4 & b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & b3 & b4 & b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & b3 & b4 & b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & b3 & b4 & b5 & b6 & !b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & !b3 & !b4 & !b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & !b3 & !b4 & !b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & !b3 & !b4 & !b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & !b3 & !b4 & !b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & b3 & !b4 & !b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & b3 & !b4 & !b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & b3 & !b4 & !b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & b3 & !b4 & !b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & !b3 & b4 & !b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & !b3 & b4 & !b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & !b3 & b4 & !b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & !b3 & b4 & !b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & b3 & b4 & !b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & b3 & b4 & !b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & b3 & b4 & !b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & b3 & b4 & !b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & !b3 & !b4 & b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & !b3 & !b4 & b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & !b3 & !b4 & b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & !b3 & !b4 & b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & b3 & !b4 & b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & b3 & !b4 & b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & b3 & !b4 & b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & b3 & !b4 & b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & !b3 & b4 & b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & !b3 & b4 & b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & !b3 & b4 & b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & !b3 & b4 & b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & b3 & b4 & b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & b3 & b4 & b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & b3 & b4 & b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & b3 & b4 & b5 & !b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & !b3 & !b4 & !b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & !b3 & !b4 & !b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & !b3 & !b4 & !b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & !b3 & !b4 & !b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & b3 & !b4 & !b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & b3 & !b4 & !b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & b3 & !b4 & !b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & b3 & !b4 & !b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & !b3 & b4 & !b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & !b3 & b4 & !b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & !b3 & b4 & !b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & !b3 & b4 & !b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & b3 & b4 & !b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & b3 & b4 & !b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & b3 & b4 & !b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & b3 & b4 & !b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & !b3 & !b4 & b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & !b3 & !b4 & b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & !b3 & !b4 & b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & !b3 & !b4 & b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & b3 & !b4 & b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & b3 & !b4 & b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & b3 & !b4 & b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & b3 & !b4 & b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & !b3 & b4 & b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & !b3 & b4 & b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & !b3 & b4 & b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & !b3 & b4 & b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & !b2 & b3 & b4 & b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & !b2 & b3 & b4 & b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & !b1 & b2 & b3 & b4 & b5 & b6 & b7 & !s0 & !s1 & !s2) | (b0 & b1 & b2 & b3 & b4 & b5 & b6 & b7 & !s0 & !s1 & !s2)

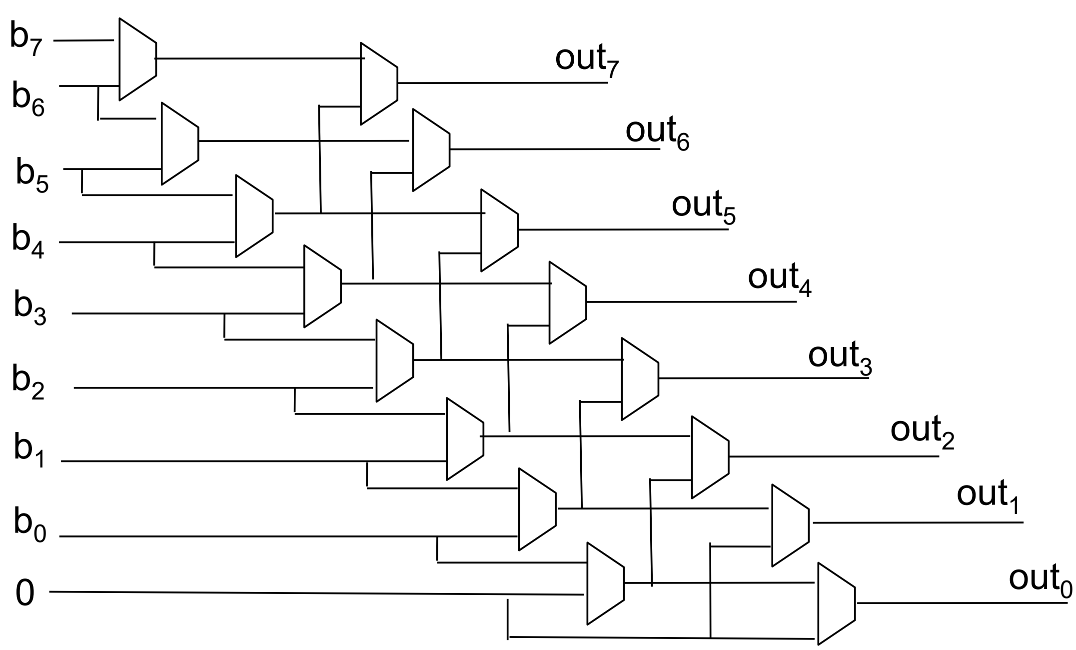
Let’s simplify

- Simpler problem: 8-bit number shifted by 1 bit number (shift amount selects each mux)



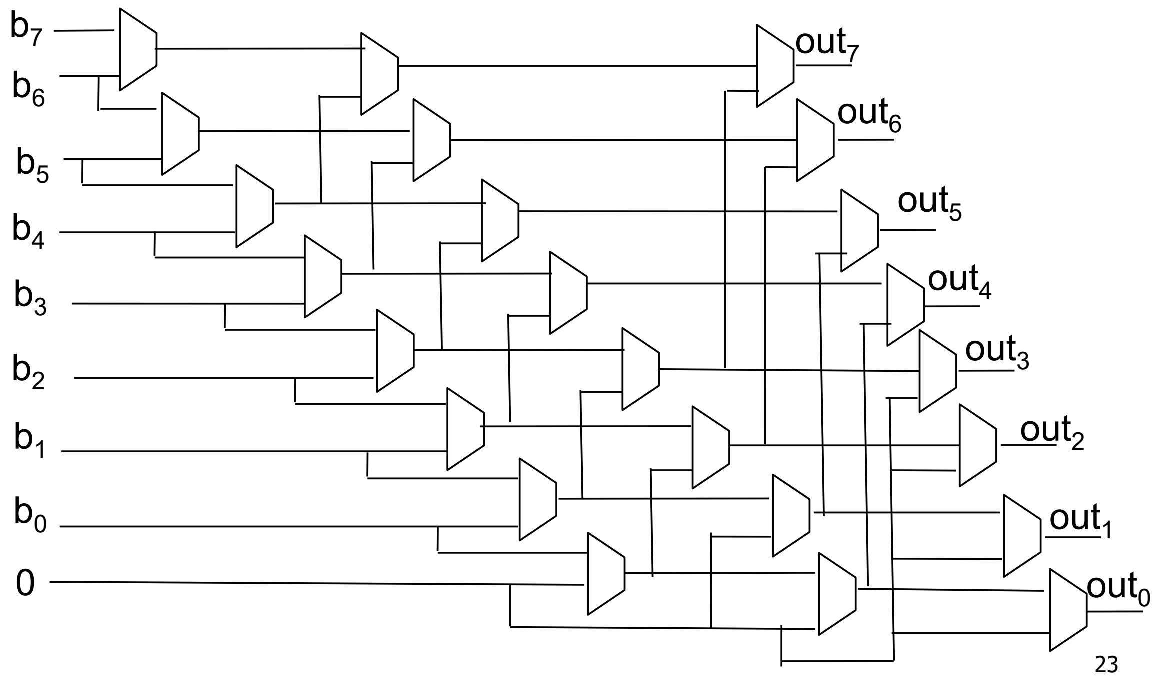
Let’s simplify

- Simpler problem: 8-bit number shifted by 2 bit number



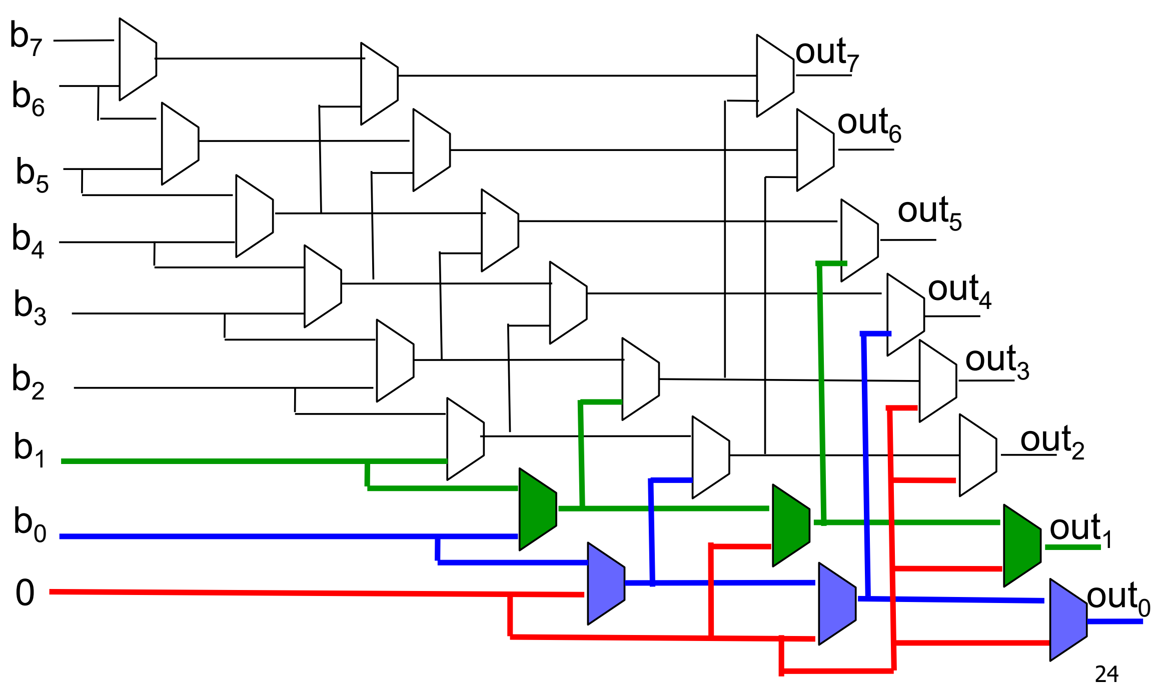
Now shifted by 3-bit number

- Full problem: 8-bit number shifted by 3 bit number



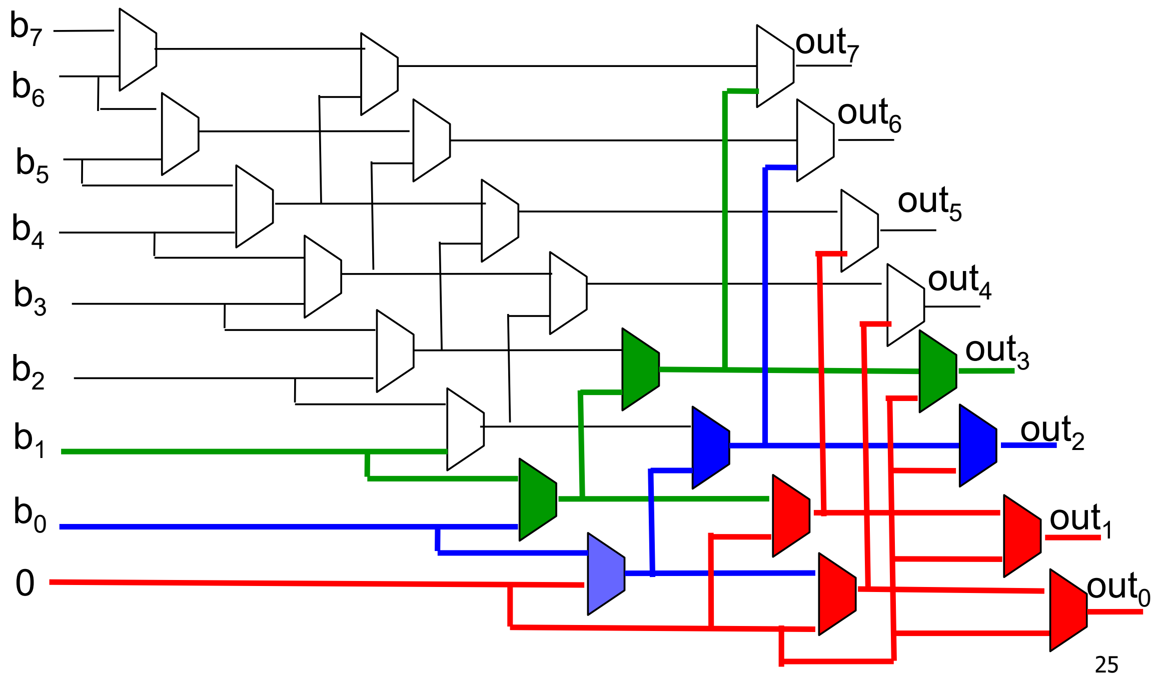
Now shifted by 3-bit number

- Shifter in action: shift by 000 (all muxes have S=0)



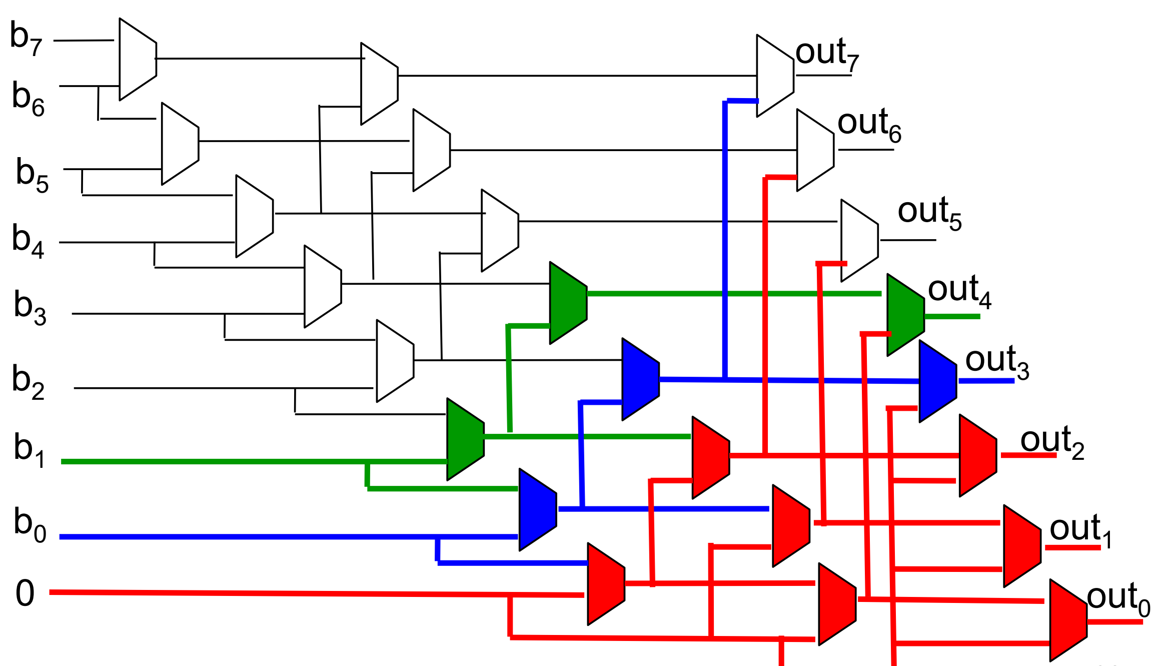
Now shifted by 3-bit number

- Shifter in action: shift by 010
  - From L to R: S = 0, 1, 0



Now shifted by 3-bit number

- Shifter in action: shift by 011
  - From L to R: S= 1, 1, 0 (reverse of shift amount)



What About Non-integer Numbers?

- There are infinitely many real numbers between two integers
- Many important numbers are real
  - Pi = 3.14159265358965…
  - ½ = 0.5
- How could we represent these sorts of numbers?
  - Floating Point

Floating Point

- Think about scientific notation for a second:
- For example:

  $6.02 * 10^{23}$
- Real number, but comprised of ints:
  - 6 generally only 1 digit here
  - 2 any number here
  - 10 always 10 (base we work in)
  - 23 can be positive or negative
- Can we do something like this in binary?

Floating Point

- How about:
- $+/-\ X.YYYYYY * 2^{+/-N}$

- Big numbers: large positive N
- Small numbers (<1): negative N

- This is “floating point” : most common way

IEEE single precision floating point

- Specific format called IEEE single precision:
- $+/-\ 1.YYYYY * 2^{(N-127)}$
- “float” in Java, C, C++,…

- S: 1 sign bit (+ = 0, 1 = -)
- E: 8 bit biased exponent (do N-127)
- M: 23-bit mantissa (YYYYY)

Binary fractions

- 1.YYYY   has a binary point
  - Like a decimal point but in binary
  - After a decimal point, you have
    - tenths
    - hundredths
    - thousandths
    - …
- So after a binary point you have…
  - Halves
  - Quarters
  - Eighths
  - …

Floating point example

- Binary fraction example:
  $101.101 = 4 + 1 + ½ + {}^{1}/_{8} = 5.625$
- For floating point, needs normalization:
  $1.01101 * 2^2$
- Sign is +, which = 0
- Exponent = 127 + 2 = 129 = 1000 0001
- Mantissa = 1.011 0100 0000 0000 0000 0000

31	30 … 23	22 … 0
0	1000 0001	011 0100 0000 0000 0000 0000

Floating Point Representation

Example:

What floating-point number is:

0xC1580000?

Answer

What floating-point number is

0xC1580000?

1100 0001 0101 1000 0000 0000 0000 0000

	31	30 … 23	22 … 0
X =	1	1000 0010	101 1000 0000 0000 0000 0000
	S	E	M

Sign = 1 which is negative

Exponent = (128+2)-127 = 3

Mantissa = 1.1011

-1.1011×2^{3} = -1101.1 = -13.5

Trick question

- How do you represent 0.0?
  - Why is this a trick question?
  - 0.0 = 000000000
  - But need 1.XXXXX representation?
- S = 0/1
- E = 0…0
- M = 0…0
- Results in +/- 0 in FP (but they are “equal”)

Other weird FP numbers

- Exponent = 1111 1111 also not standard
  - S = 0 and M = 0: $+\infty$
  - S = 1 and M = 0: $-\infty$
  - S = 0/1 and M $\neq$ 0: Not a Number (NaN)

    sqrt(-42) = NaN

Floating Point Representation

- Double Precision Floating point:

  64-bit representation:
  - 1-bit sign
  - 11-bit (biased) exponent
  - 52-bit fraction (with implicit 1).

- "double" in Java, C, C++, …

S	Exp	Mantissa
1	11-bit	52 - bit

Danger: floats cannot hold all ints!

- Many programmers think:
  - Floats can represent all ints
  - NOT true
- Doubles can represent all 32-bit ints

  (but not all 64-bit ints)

Wrap Up

- Implementation of Math
  - Addition/Subtraction
  - Shifting
- Floating Point Numbers
  - IEEE representation
  - Denormalized Numbers
- Next Time:
  - Storage
  - Clocking