ECE 550D
Fundamentals of Computer Systems and Engineering

Fall 2023

Number Representations

Xin Li & Dawei Liu
Duke Kunshan University

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

Last time….

- Who can remind us what we talked about last time?
  - Combinatorial Logic
  - Sum-of-products
  - Simplification
  - Muxes

Next: logic to work with numbers

- Computers do one thing: math
  - And they do it well/fast
  - Fundamental rule of computation: "Everything is a number"
    - Computers can only work with numbers
    - Represent things as numbers
  - Specifically: good at **binary** math
    - Base 2 number system: matches circuit voltages
      - 1 (Vcc)
      - 0 (Ground)
    - Use fixed sized numbers
      - How many **bits**
  - Quick primer on binary numbers/math
    - Then how to make circuits for it

Numbers for computers

- We usually use base 10:
  - $12345 = 1 * 10^4 + 2 * 10^3 + 3 * 10^2 + 4 * 10^1 * 5 * 10^0$
  - Recall from third grade: 1's place, 10's place, 100's place…
    - Yes, we are going to re-cover 3rd grade math, but in binary
  - What is the biggest digit that can go in any place?
- Base 2:
  - 1's place, 2's place, 4's place, 8's place, ….
  - What is the biggest digit that can go in any place?

Basic Binary

- Advice: memorize the following
  - $2^0 = 1$
  - $2^1 = 2$
  - $2^2 = 4$
  - $2^3 = 8$
  - $2^4 = 16$
  - $2^5 = 32$
  - $2^6 = 64$
  - $2^7 = 128$
  - $2^8 = 256$
  - $2^9 = 512$
  - $2^{10} = 1024$

Binary continued:

- Binary Number Example: 101101
  - Take a second and figure out what number this is

Binary continued:

- Binary Number Example: 101101
  - Take a second and figure out what number this is

  1 in 32’s place = 32

  0 in 16’s place

  1 in 8’s place = 8

  1 in 4’s place = 4

  0 in 2’s place

  1 in 1’s place = 1

  —

  45

Converting Numbers

- Converting Decimal to Binary

  Suppose I want to convert 457 to binary

  Think for a second about how to do this

Decimal to binary using remainders

?	Quotient	Remainder
457 ÷ 2 =	228	1
228 ÷ 2 =	114	0
114 ÷ 2 =	57	0
57 ÷ 2 =	28	1
28 ÷ 2 =	14	0
14 ÷ 2 =	7	0
7 ÷ 2 =	3	1
3 ÷ 2 =	1	1
1 ÷ 2 =	0	1

111001001

Decimal to binary using comparison

111001001

Num	Compare 2^n	≥ ?
457	256	1
201	128	1
73	64	1
9	32	0
9	16	0
9	8	1
1	4	0
1	2	0
1	1	1

Hexadecimal: Convenient shorthand for Binary

- Binary is not easy to write
  - 425,000 decimal = 1100111110000101000 binary
  - Generally about 3x as many binary digits as decimal
  - Converting (by hand) takes some work and thought
- Hexadecimal (aka “hex”)—base 16—is convenient:
  - Easy mapping to/from binary
  - Same or fewer digits than decimal
  - 425,000 decimal = 0x67C28
  - Generally write “0x” on front to make clear “this is hex”
  - Digits from 0 to 15, so use A—F for 10—15.

Hexadecimal

Binary ⇔ Hex conversion is straightforward.
Every 4 binary bits = 1 hex digit. If # of bits not a multiple of 4, add implicit 0s on left as needed

Hex digit	Binary	Decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Indicates a hex number

0xDEADBEEF

1101 1110 1010 1101 1011 1110 1110 1111

0x02468ACE

0000 0010 0100 0110 1000 1010 1100 1110

0x13579BDF

0001 0011 0101 0111 1001 1011 1101 1111

Binary to/from hexadecimal

- 0101101100100011_2 -->
- 0101 1011 0010 0011_2 -->
- 5 B 2 3_{16}

1 F 4 B_{16} -->

0001 1111 0100 1011_2 -->

0001111101001011_2

Hex digit	Binary	Decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Binary Math : Addition

- Suppose we want to add two numbers:

```
   00011101
+  00101011
-----------
```

- How do we do this?

Binary Math : Addition

- Suppose we want to add two numbers:

```
    00011101          695
+   00101011        + 232
```

- How do we do this?
  - Let's revisit decimal addition
  - Think about the process as we do it

Binary Math : Addition

- Suppose we want to add two numbers:

```
    00011101          695
+   00101011        + 232
                        7
```

- First add one's digit 5+2 = 7

Binary Math : Addition

- Suppose we want to add two numbers:

```
                         1
    00011101            695
+   00101011          + 232
------------          -----
                         27
```

- First add one's digit 5+2 = 7
- Next add ten's digit 9+3 = 12 (2 carry a 1)

Binary Math : Addition

- Suppose we want to add two numbers:

```
                         1
    00011101            695
+   00101011          + 232
------------          -----
                        927
```

- First add one's digit 5+2 = 7
- Next add ten's digit 9+3 = 12 (2 carry a 1)
- Last add hundred's digit 1+6+2 = 9

Binary Math : Addition

- Suppose we want to add two numbers:

```
     00011101
 +   00101011
 ------------
```

- Back to the binary:
- First add 1's digit 1+1 = …?

Binary Math : Addition

- Suppose we want to add two numbers:

```
            1
     00011101
 +   00101011
 ------------
            0
```

- Back to the binary:
- First add 1's digit 1+1 = 2 (0 carry a 1)

Binary Math : Addition

- Suppose we want to add two numbers:

```
           11
     00011101
 +   00101011
 ------------
           00
```

- Back to the binary:
- First add 1's digit 1+1 = 2 (0 carry a 1)
- Then 2's digit: 1+0+1 =2 (0 carry a 1)
- You all finish it out….

Binary Math : Addition

- Suppose we want to add two numbers:

```
      111111
     00011101    = 29
 +   00101011    = 43
 ------------
     01001000    = 72
```

- Can check our work in decimal

Negative Numbers

- May want negative numbers too!
- Many ways to represent negative numbers:
  - Sign/magnitude
  - Biased
  - 1’s complement
  - 2’s complement

2’s Complement Integers

- To negate, flip bits, add 1:
  - 1’s complement + 1
- Pros:
  - Easy to compute with
  - One representation of 0
- Cons:
  - More complex negation
  - Extra negative number (-8)

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

Binary Math : Addition

- Revisit binary math for a minute:

```
     01011101
 +   01101011
 ------------
```


Binary Math : Addition

- What about this one:

```
     1111111
     01011101    =   93
 +   01101011    =  107
 ------------
     11001000    =  -56
```

- But… that can't be right?
  - What do you expect for the answer?
  - What is it in 8-bit signed 2's complement?

Integer Overflow

- Answer should be 200
  - Not representable in 8-bit signed representation
  - No right answer
- Called Integer Overflow
  - Signed addition: CI != CO of last bit
  - Unsigned addition: CO != 0 of last bit
- Can detect in hardware
  - Signed: XOR CI and CO of last bit
  - Unsigned: CO of last bit
  - What processor does: depends

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 (next time)

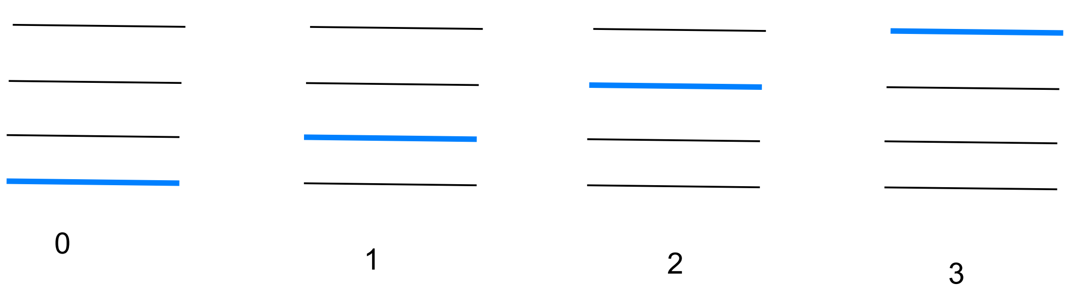
```
                                 1
  00110101      ->        00110101
- 01010010              + 10101101
----------              ----------
```

Signed and Unsigned Ints

- Most programming languages support two int types
  - Signed: negative and positive
  - Unsigned: positive only, but can hold larger positive numbers
- Addition and subtraction:
  - Same, except overflow detection
  - x86: one add instruction, sets two different flags for overflows
- Inequalities
  - Different operations for signed/unsigned
  - Can someone give an example? (Let's say 4-bit numbers)

One hot representation

- Binary representation convenient for math
- Another representation:
  - One hot: one wire per number
  - At any time, one wire = 1, others = 0



  - Very convenient in many cases

Converting to/from one hot

- Converting from 2^N bits one hot to N bits binary=**encoder**
  - E.g., "an 8-to-3 encoder"
- Converting from N bits binary to 2^N bits one hot=**decoder**
  - E.g., "a 4-to-16 decoder"

Lets build a 4-to-2 encoder

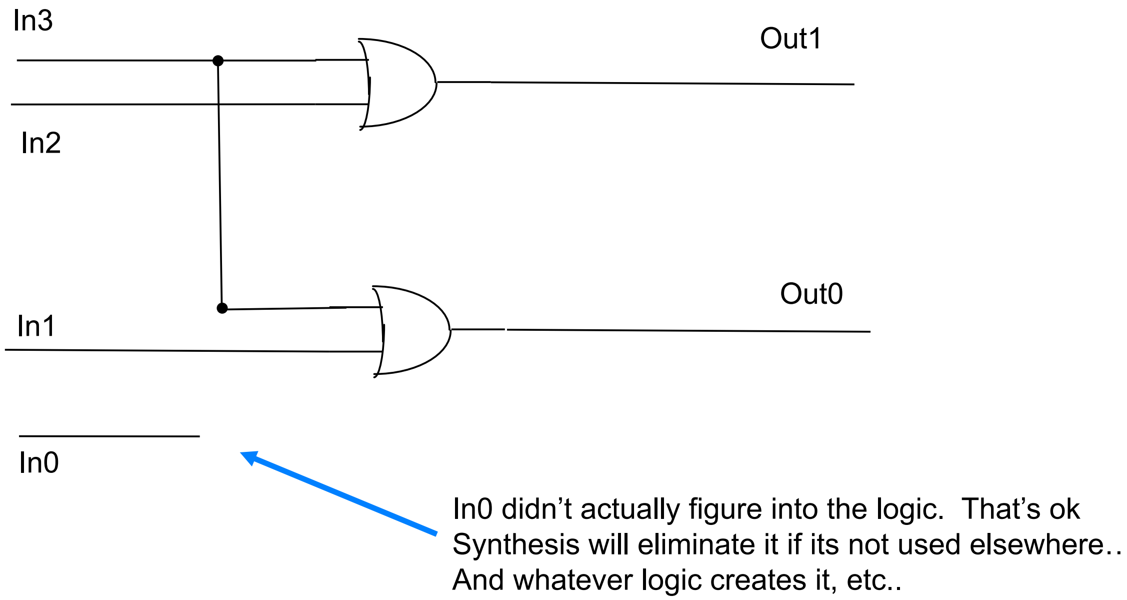
- Start with a truth table
  - Input constrained to 1-hot: don't care about invalid inputs
    - Can do anything we want

In0	In1	In2	In3	Out1	Out0
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

- Simplest formulas:
  - Out0 = In1 or In3 [alternatively: Out0 = In0 nor In2]
  - Out1 = In2 or In3 [alternatively: Out1 = In0 nor In1]

4-to-2 encoder

- Our 4-to-2 encoder
  - Note: the dots here show connections
    - Don't confuse with open circles which mean NOT



Lets build a 2-to-4 decoder

- Start with a truth table
  - Now input unconstrained

In1	In0	Out0	Out1	Out2	Out3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Now sum-of-powers more useful, do for each of the 4 outputs:

Out0 = (Not In1) and (Not In0)

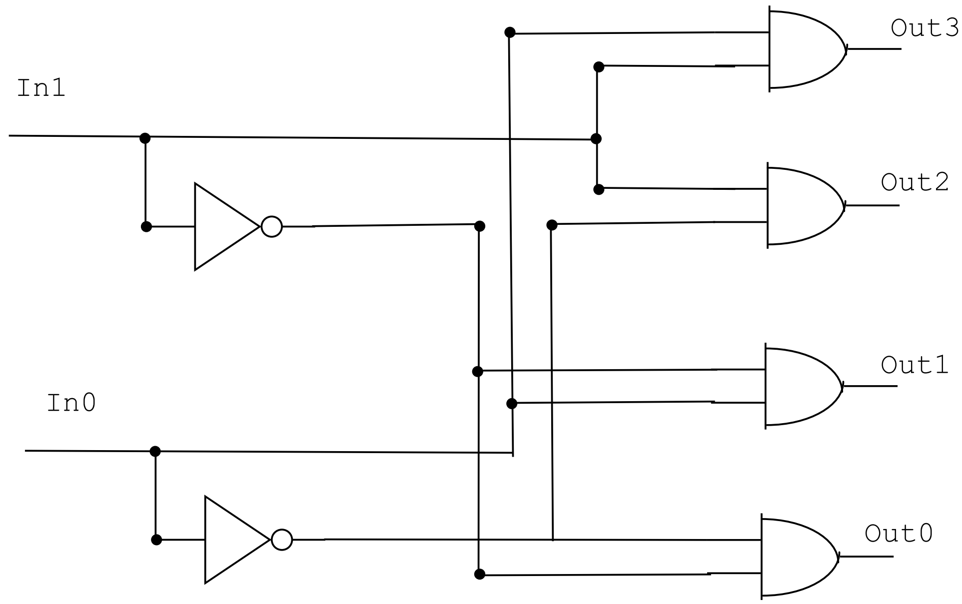
Out1 = (Not In1) and In0

Out2 = In1 and (Not In0)

Out3 = In1 and In0

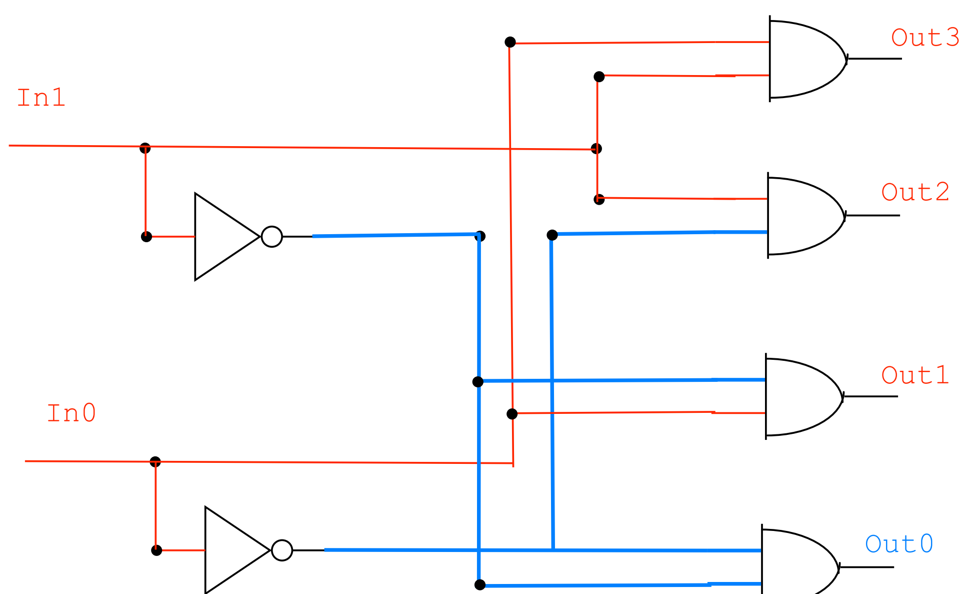
2-to-4 decoder

- 2-to-4 decoder



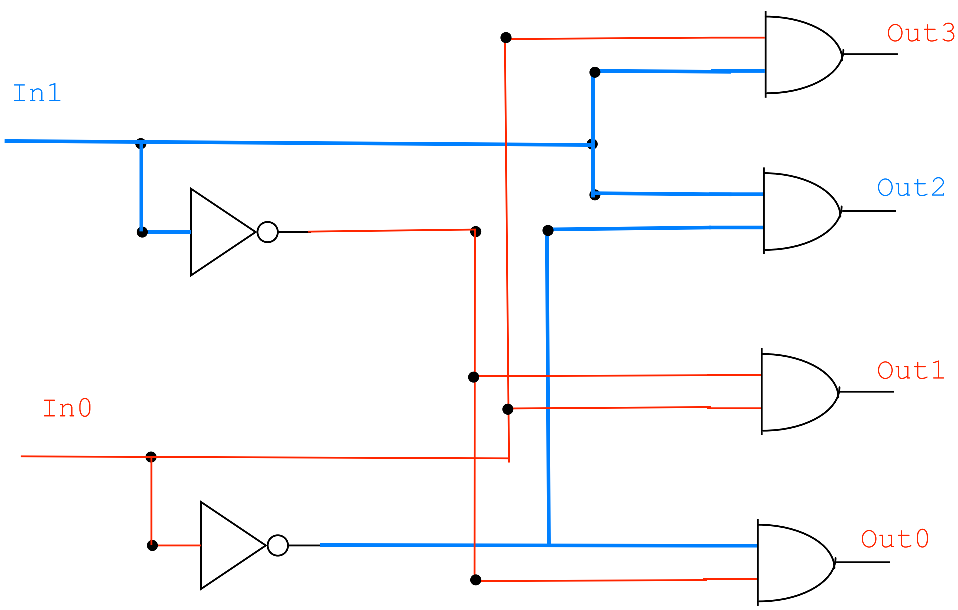
2-to-4 decoder

- 2-to-4 decoder



2-to-4 decoder

- 2-to-4 decoder



Delays

- Mentioned before: switching not instant
  - Not going to try to calculate delays by hand (tools can do)
  - But good to know where delay comes from, to tweak/improve
- Gates:
  - Switching the transistors in gates takes time
  - More gates (in series) = more delay
- Fan-out: how many gates the output drives
  - Related to capacitance
  - High fan-out = slow
  - Sometimes better to replicate logic to reduce its fan-out
- Wire delay:
  - Signals take time to travel down wires

Wrap Up

- Number Representations
  - Binary number
  - 2’s complement
  - One hot representation
  - Encoder and decoder