## Course Information

- **Instructors:**
  - Mostafizur Rahman
  - E-mail: rahmanmo@umkc.edu
  - Office: FH 570H

- **Office hours:**
  - Available Before & After Class on Mondays & Wednesdays
  - By appointments

- **Course Prerequisites:** E&C-ENGR 226 and E&C-ENGR 227

- **Textbook:** *Logic and Computer Design Fundamentals*, by M. Mano and C. Kime

## Course Information

| Evaluations | Point Value |
|---|---|
| Quizzes | Quiz: 10 points, |
| In class activities | Classwork: 20 points |
| Assignments | 20 points |
| 2 Mid Term Exam | 15+15 = 30 points |
| Final Project | 20 points |
| **Total points** | **100 points** |

- Class participation is a must. You need to have hardware/software resources to participate/lead discussions
- Class activities are due within 30minutes of class end time
- No extra credit activities, late submissions without prior notice

## Grading Policy

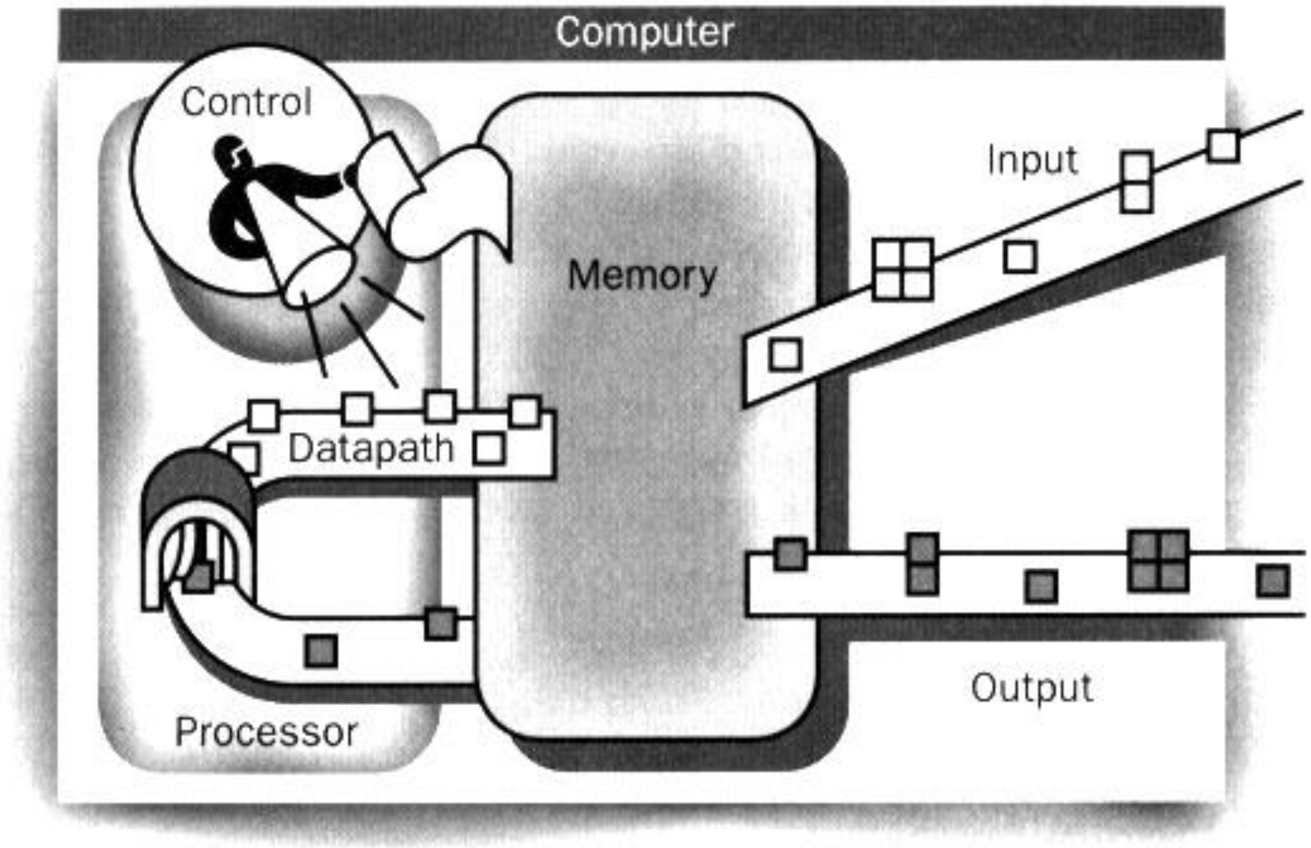
- Cluster grading scheme will be applied. Starting with the highest scores, students will be clustered within 3-5 points margin. A typical clustering is as follows:
- >90 = A
  87-90 = A-
  83-86 = B+
  78-82 = B
  73-77 = B-
  67-72 = C+
  62-66 = C
  56 - 61 = C-
  51 - 55 = D+
  46 - 50 = D
  41 - 45 = D-
  40 or lower F

## Course Objectives

- **Understand Computer Architecture and Organizations**
    - CPU
    - Memory/Cache
    - Instruction Set Architecture
    - Cache Hierarchy
    - Input & Output
    - CPU Pipelining
- **Learn Verilog and How to use it**

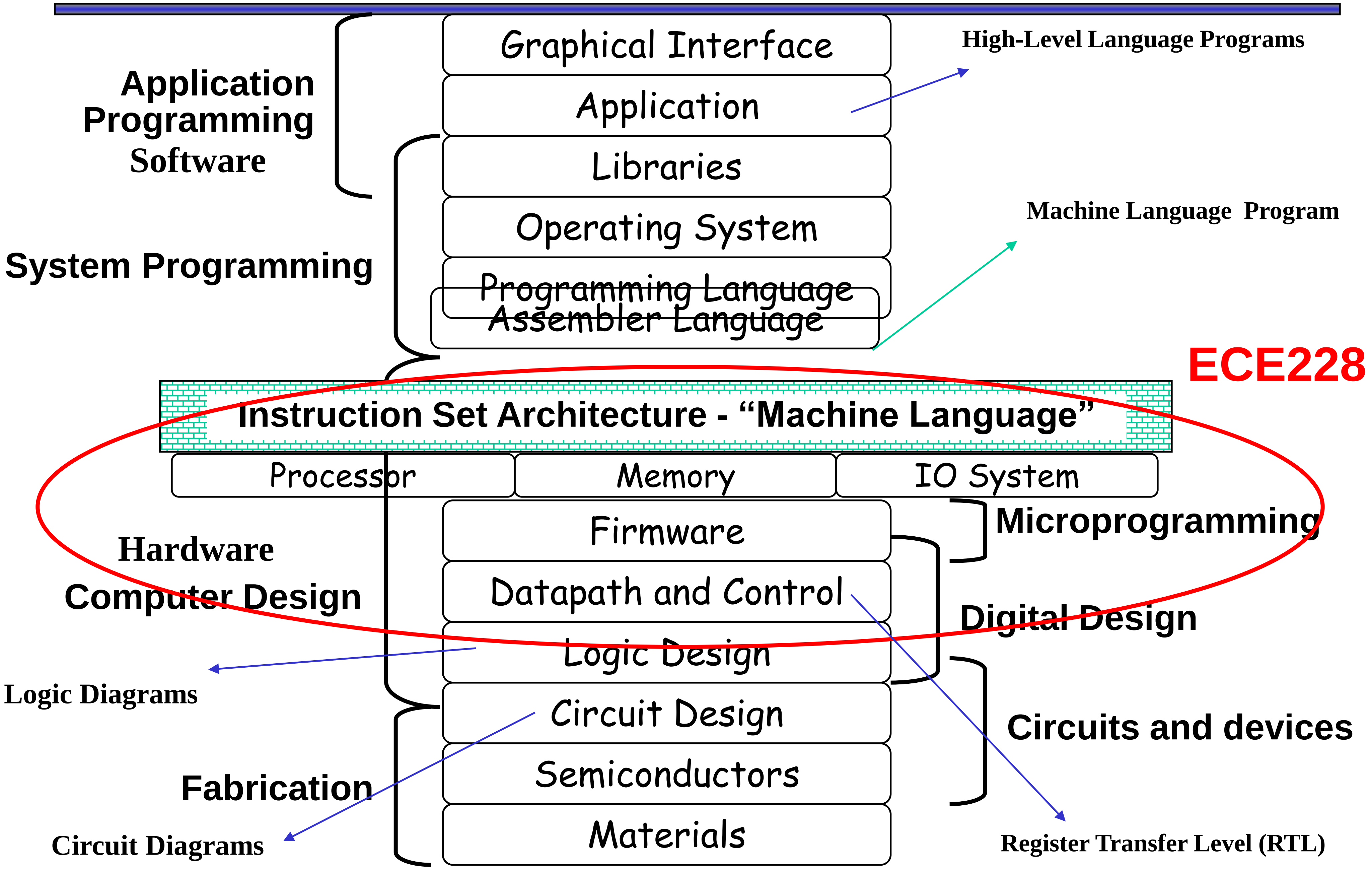
## Zooming into a Chip

## Computer System - Five Classic Components



- Partitioning of the computing engine into components:
  - Central Processing Unit (CPU): **Control Unit** (instruction decode, sequencing of operations), **Datapath** (registers, arithmetic and logic unit, buses)
  - **Memory**: Instruction and operand storage.
  - **Input**/**Output** (I/O)

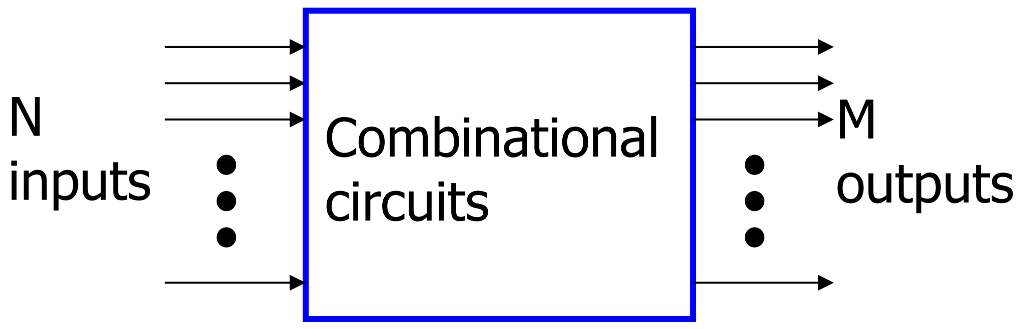
# Computing System - Levels of Abstraction



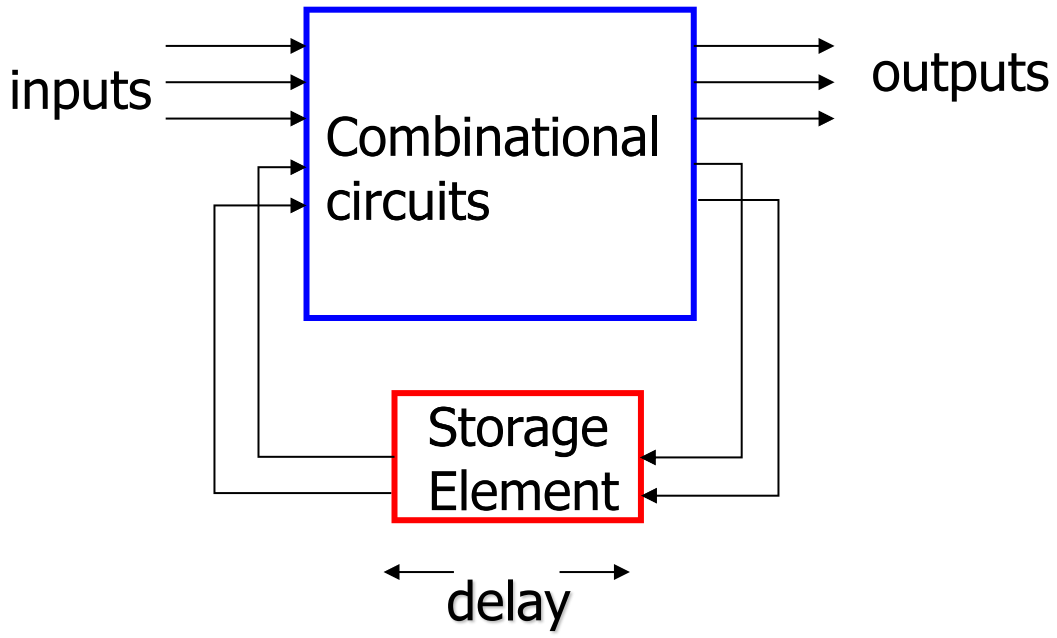
Coordination of many *levels of abstraction*

## Logic Design - Brief Review

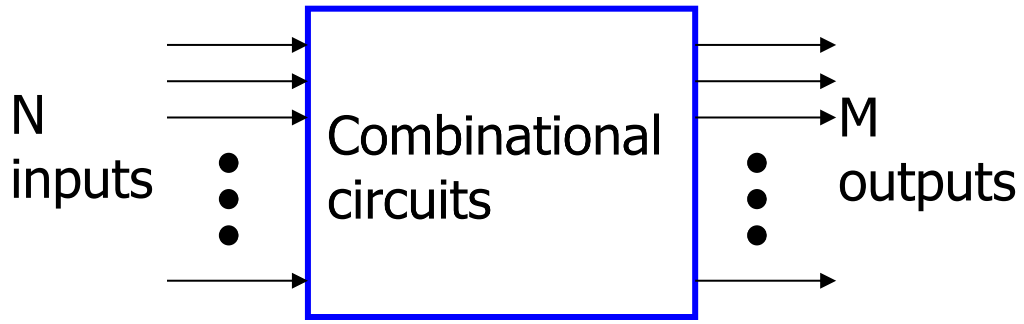
- Logic circuits
  - Combinational



  - Sequential

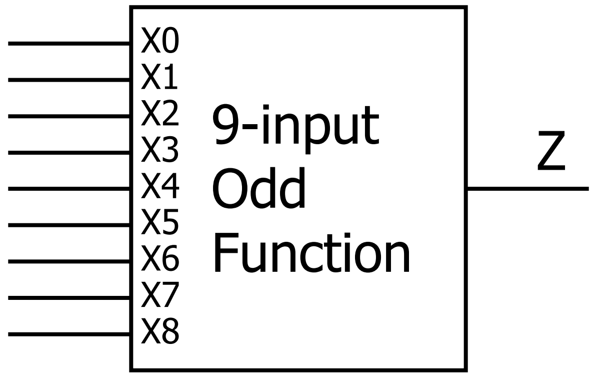


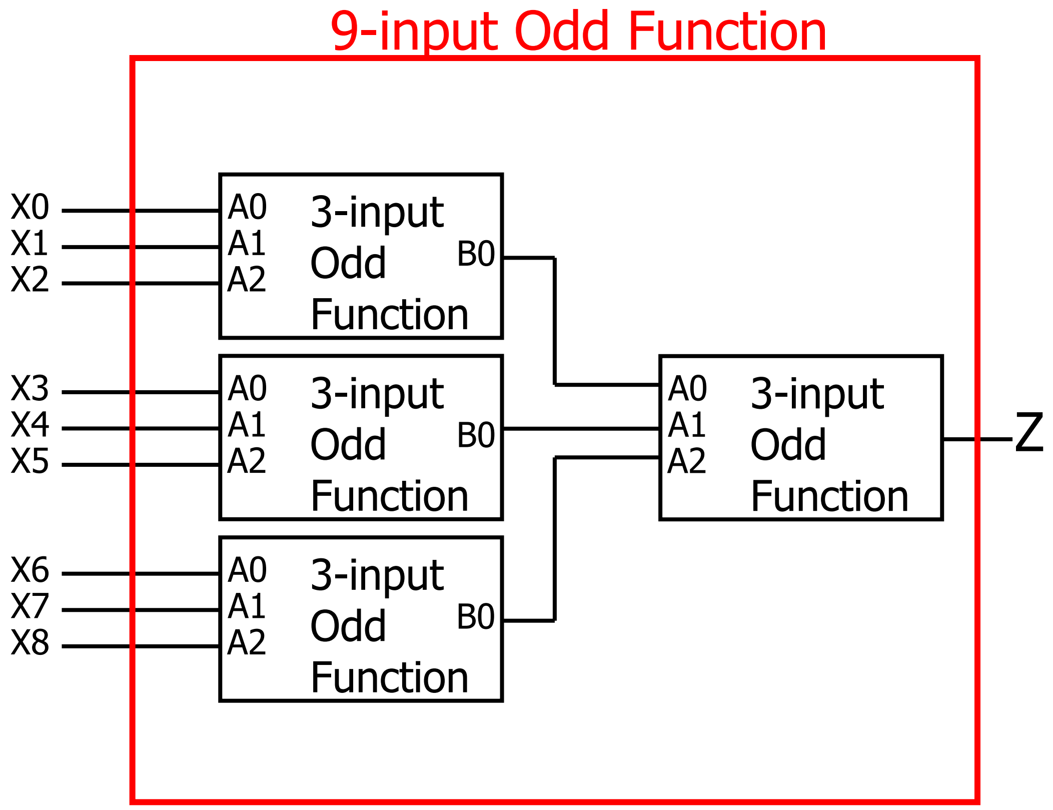
## Combinational Logic



- Outputs, "at any time", are determined by the input combination
- When input changed, output changed immediately
  - Note that real circuits are imperfect and have "propagation delay"
- A combinational circuit
  - Performs logic operations that can be specified by a set of Boolean expressions
  - Can be built hierarchically
- Basic combinatorial circuits
  - Multiplexers
  - Demultiplexers
  - Decoders
  - Comparators

## Design Hierarchy Example





Function Specification:
To detect odd number of "1" inputs, i.e. Z=1 when there is an odd number of "1" present in the inputs

How to design a 3-input Odd Function?

## Derive Truth Table for Desired Functionality

| A | B | C | F |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

$$\begin{array}{c|cccc} A \backslash BC & 00 & 01 & 11 & 10 \\ \hline 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{array}$$

$$F = \overline{A}\,\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\,\overline{C} + ABC$$

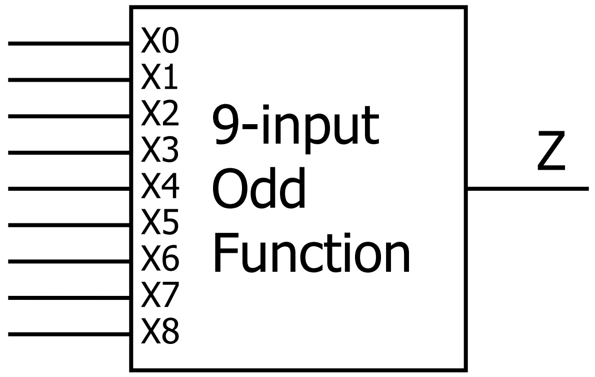
$$= \overline{A}(\overline{B}C + B\overline{C}) + A(\overline{B}\,\overline{C} + BC)$$

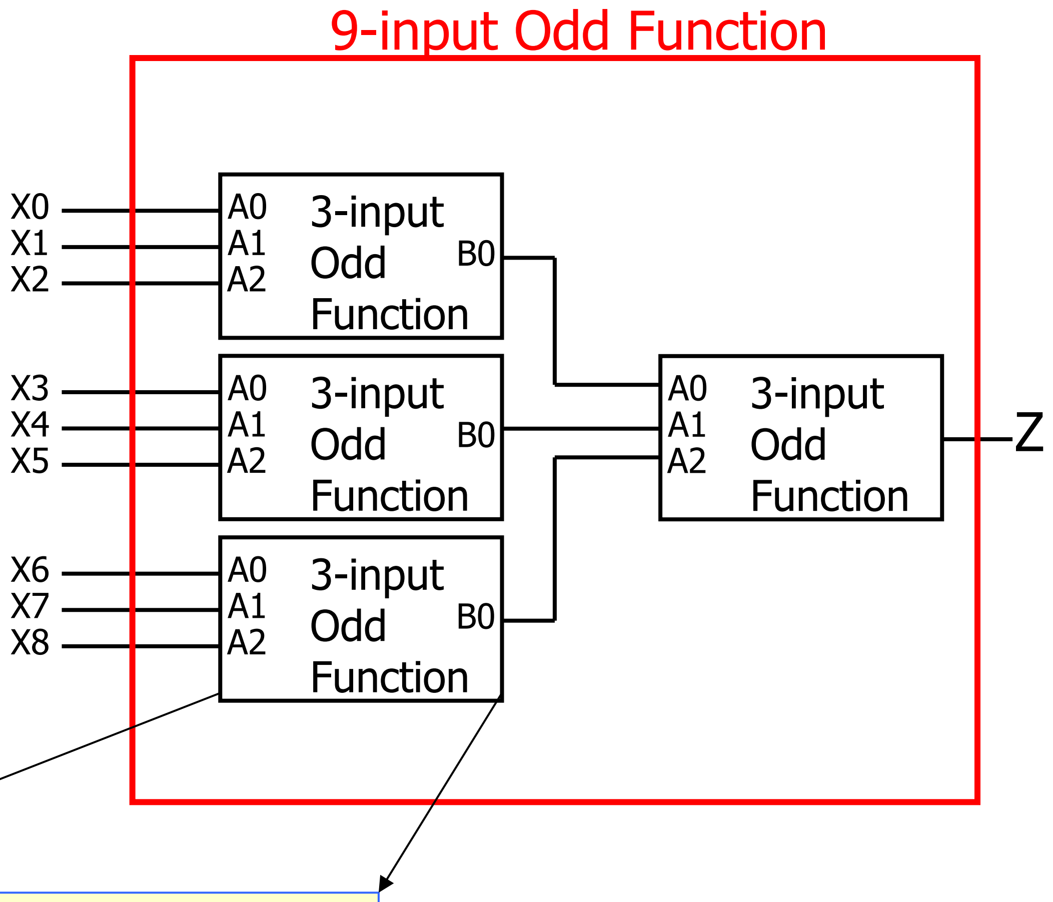
$$= \overline{A}(B \oplus C) + A(\overline{B \oplus C})$$

$$= A \oplus (B \oplus C)$$

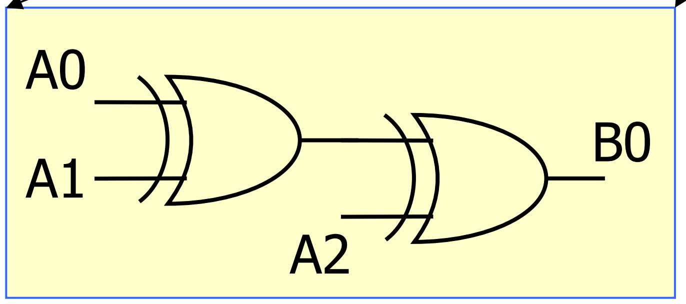
$$= A \oplus B \oplus C$$

# Design Hierarchy Example





3-input Odd function:
B0=A0 ⊕A1⊕A2



## What a decoder does

- A n-to-$2^n$ decoder takes an n-bit input and produces $2^n$ outputs. The n inputs represent a binary number that determines which of the $2^n$ outputs is *uniquely* true.
- A 2-to-4 decoder operates according to the following truth table.
  - The 2-bit input is called S1S0, and the four outputs are Q0-Q3.
  - If the input is the binary number i, then output Qi is uniquely true.

| S1 | S0 | Q0 | Q1 | Q2 | Q3 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |

- For instance, if the input S1 S0 = 10 (decimal 2), then output Q2 is true, and Q0, Q1, Q3 are all false.
- This circuit “decodes” a binary number into a “one-of-four” code.

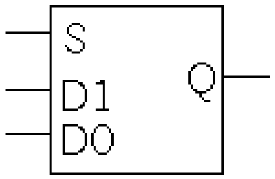
## Enable inputs

- Many devices have an additional enable input, which is used to "activate" or "deactivate" the device.
- For a decoder,
  - EN=1 activates the decoder, so it behaves as specified earlier. Exactly one of the outputs will be 1.
  - EN=0 "deactivates" the decoder. By convention, that means *all* of the decoder's outputs are 0.
- We can include this additional input in the decoder's truth table:

| EN | S1 | S0 | Q0 | Q1 | Q2 | Q3 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 |

## Multiplexers

- A $2^n$-to-1 multiplexer sends one of $2^n$ input lines to a single output line.
  - A multiplexer has two sets of *inputs*:
    - $2^n$ data input lines
    - n select lines, to pick one of the $2^n$ data inputs
  - The mux *output* is a single bit, which is one of the $2^n$ data inputs.
- The simplest example is a 2-to-1 mux:



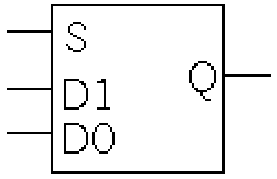
$$Q = S' D0 + S D1$$

- The select bit S controls which of the data bits D0-D1 is chosen:
  - If S=0, then D0 is the output (Q=D0).
  - If S=1, then D1 is the output (Q=D1).

## Truth table abbreviations

- Here is a full truth table for this 2-to-1 mux, based on the equation:

$$Q = S' D0 + S D1$$



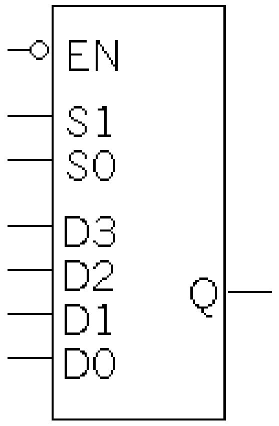
| S | D1 | D0 | Q |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

- Here is another kind of abbreviated truth table.
  - Input variables appear in the output column.
  - This table implies that when S=0, the output Q=D0, and when S=1 the output Q=D1.

| S | Q |
|---|---|
| 0 | D0 |
| 1 | D1 |

## A 4-to-1 multiplexer

- Here is a block diagram and abbreviated truth table for a 4-to-1 mux.



| EN' | S1 | S0 | Q |
|---|---|---|---|
| 0 | 0 | 0 | D0 |
| 0 | 0 | 1 | D1 |
| 0 | 1 | 0 | D2 |
| 0 | 1 | 1 | D3 |
| 1 | x | x | 1 |

$$Q = S1' \; S0' \; D0 + S1' \; S0 \; D1 + S1 \; S0' \; D2 + S1 \; S0 \; D3$$

There are two logic blocks: odd function and even function generators. And there is a clock input. When the clock is high, odd function generator displays its output and when the clock is low, even function generator displays its output. Can you picture it?

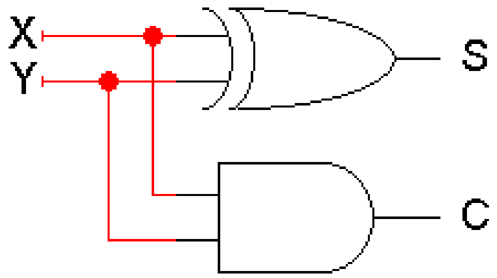
# Computer Arithmetic

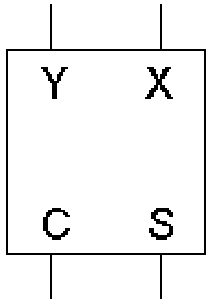
## Adding two bits

- We start with a half adder, which adds two bits and produces a two-bit result: a sum (the right bit) and a carry out (the left bit).
- Here are truth tables, equations, circuit and block symbol.

| X | Y | C | S |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

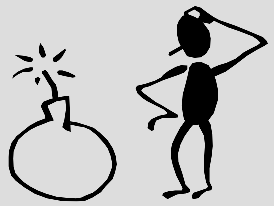
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 10





$C = XY$

$S = X'Y + XY'$
$= X \oplus Y$



Be careful! Now we're using + for both arithmetic addition and the logical OR operation.

## Full adder equations

- A full adder circuit takes three bits of input, and produces a two-bit output consisting of a sum and a carry out.
- Using Boolean algebra, we get the equations shown here.
  - XOR operations simplify the equations a bit.

| $X$ | $Y$ | $C_{in}$ | $C_{out}$ | $S$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

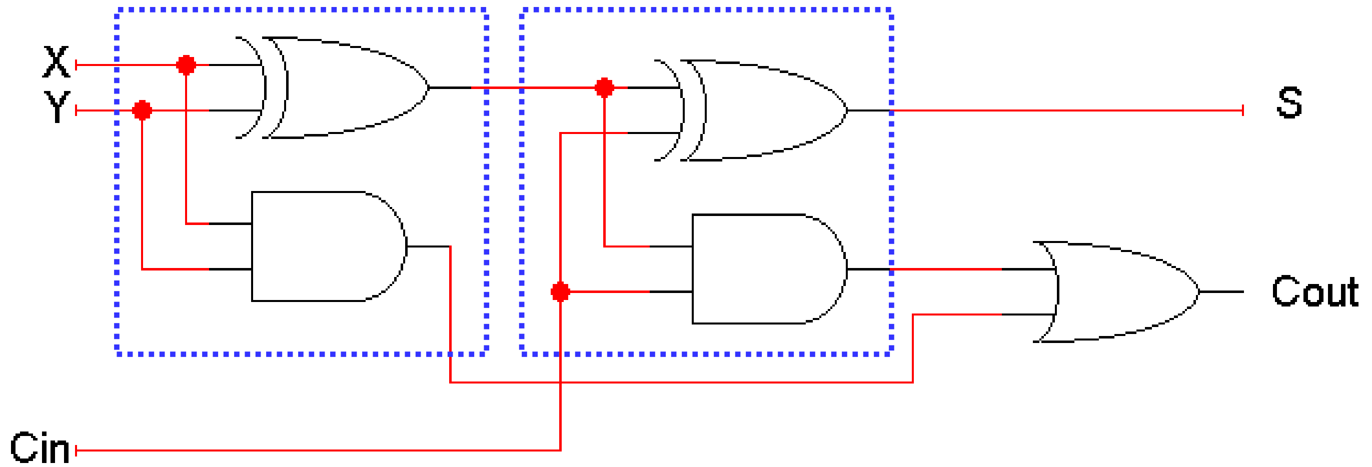
$$\begin{aligned} S &= \Sigma m(1,2,4,7) \\ &= X' Y' C_{in} + X' Y C_{in}' + X Y' C_{in}' + X Y C_{in} \\ &= X' (Y' C_{in} + Y C_{in}') + X (Y' C_{in}' + Y C_{in}) \\ &= X' (Y \oplus C_{in}) + X (Y \oplus C_{in})' \\ &= X \oplus Y \oplus C_{in} \end{aligned}$$

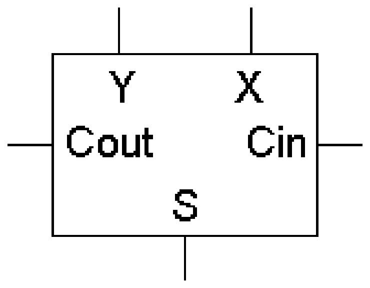
$$\begin{aligned} C_{out} &= \Sigma m(3,5,6,7) \\ &= X' Y C_{in} + X Y' C_{in} + X Y C_{in}' + X Y C_{in} \\ &= (X' Y + X Y') C_{in} + XY(C_{in}' + C_{in}) \\ &= (X \oplus Y) C_{in} + XY \end{aligned}$$

## Full adder circuit

- These things are called half adders and full adders because you can build a full adder by putting together two half adders!

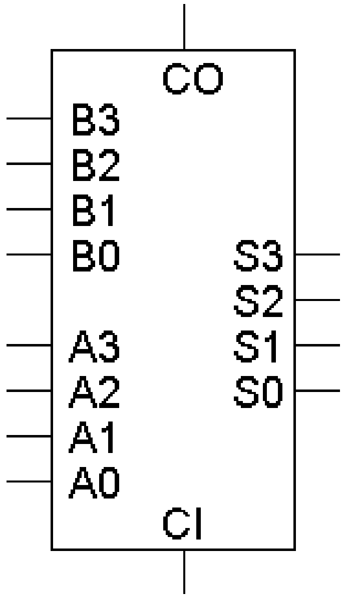
$$S = X \oplus Y \oplus C_{in}$$
$$C_{out} = (X \oplus Y)\, C_{in} + XY$$

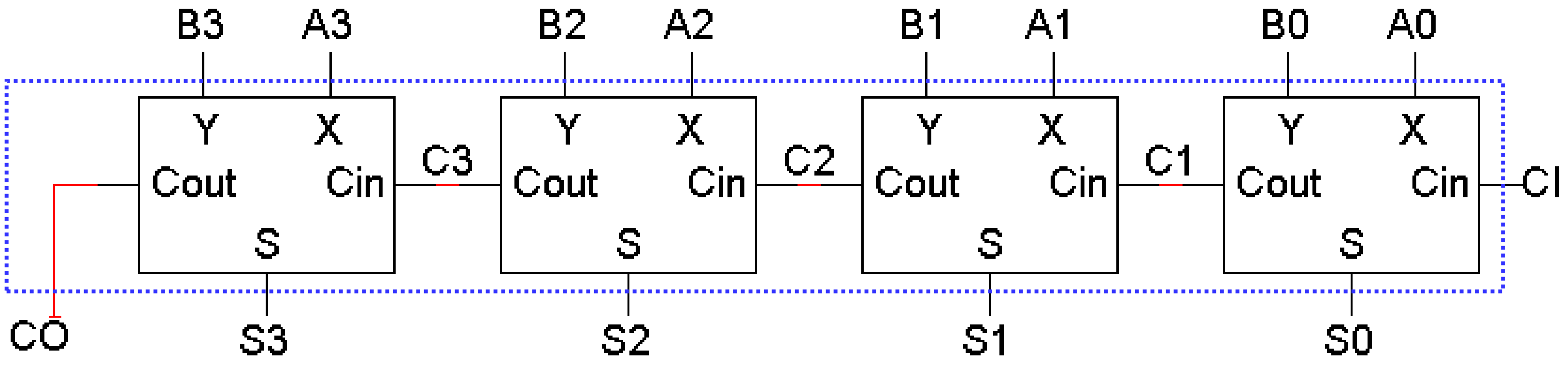




# A 4-bit adder

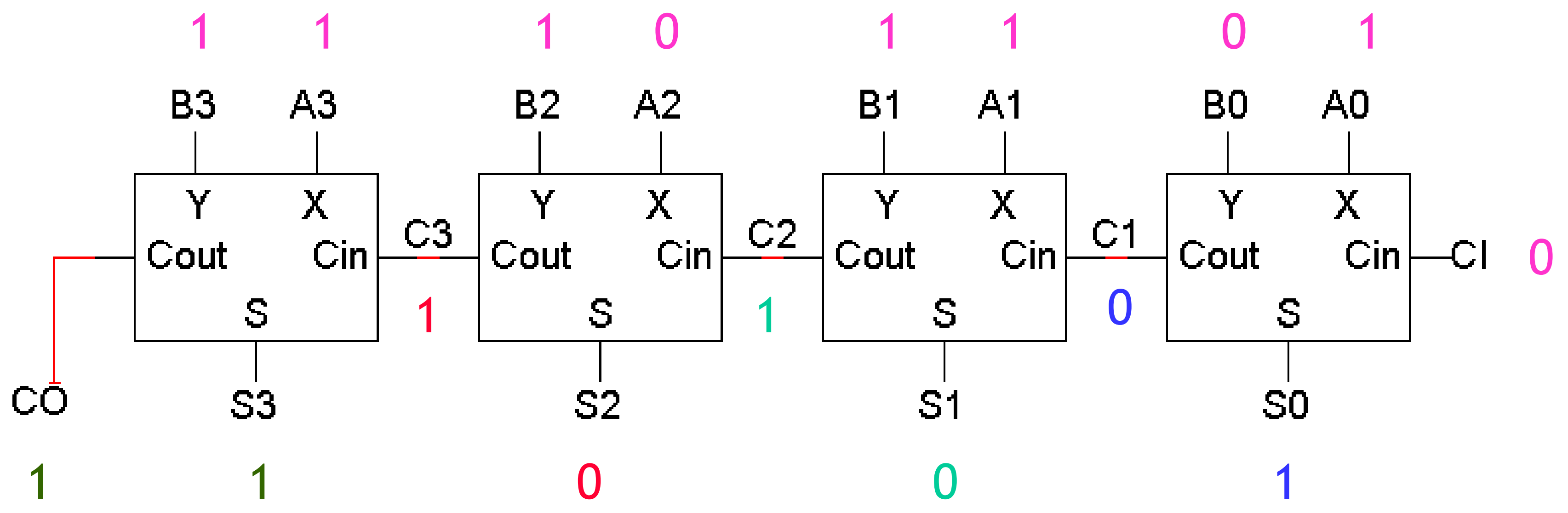
- Four full adders together make a 4-bit adder.
- There are nine total inputs:
  - Two 4-bit numbers, A3 A2 A1 A0 and B3 B2 B1 B0
  - An initial carry in, CI
- The five outputs are:
  - A 4-bit sum, S3 S2 S1 S0
  - A carry out, CO
- Imagine designing a nine-input adder without this hierarchical structure—you'd have a 512-row truth table with five outputs!





## An example of 4-bit addition

- Let’s try our initial example: A=1011 (eleven), B=1110 (fourteen).



1. Fill in all the inputs, including CI=0
2. The circuit produces C1 and S0 (1 + 0 + 0 = 01)
3. Use C1 to find C2 and S1 (1 + 1 + 0 = 10)
4. Use C2 to compute C3 and S2 (0 + 1 + 1 = 10)
5. Use C3 to compute CO and S3 (1 + 1 + 1 = 11)

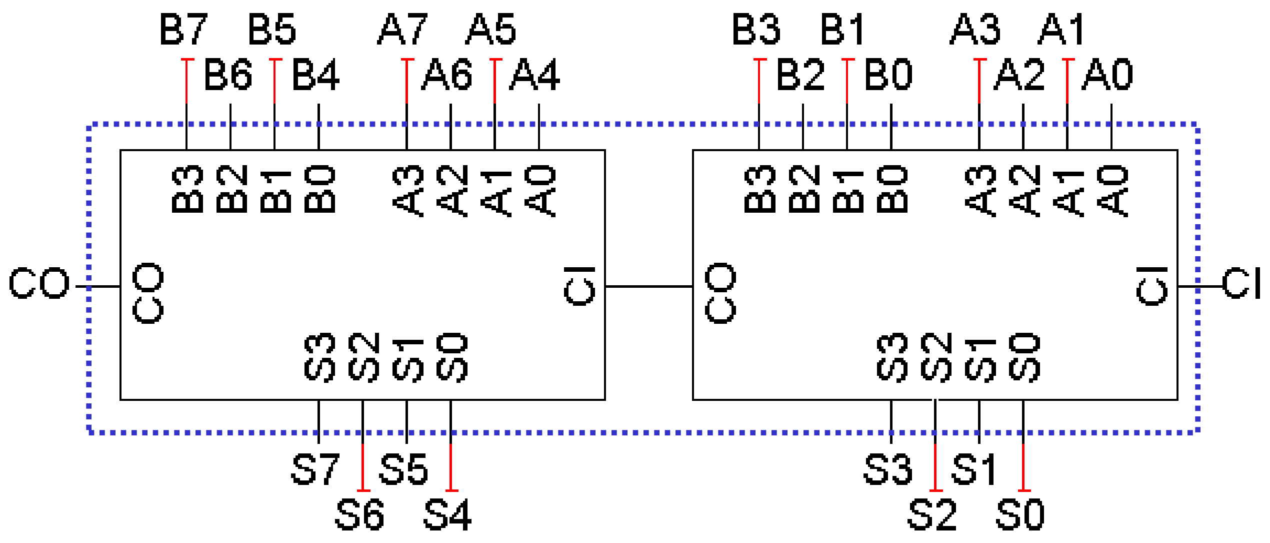
The final answer is 11001 (twenty-five).

## Question

When you add two 4-bit numbers the carry in is always 0, so why does the 4-bit adder have a CI input?

## Hierarchical adder design

- Here is an 8-bit adder, for example.



## Negative Numbers and Subtraction

- The adders we designed can add only non-negative numbers
  - If we can represent negative numbers, then subtraction is "just" the ability to add two numbers (one of which may be negative).
- We'll look at three different ways of representing signed numbers.
- How can we decide which representation is better?
  - The best one should result in the simplest and fastest operations.
- We're mostly concerned with two particular operations:
  - Negating a signed number, or converting x into -x.
  - Adding two signed numbers, or computing x + y.
  - So, we will compare the representation on how fast (and how easily) these operations can be done on them

## Signed magnitude representation

- Humans use a signed-magnitude system: we add + or - in front of a magnitude to indicate the sign.
- We could do this in binary as well, by adding an extra sign bit to the front of our numbers. By convention:
  - A 0 sign bit represents a positive number.
  - A 1 sign bit represents a negative number.
- Examples:

| | | |
|---|---|---|
| $1101_2$ | $= 13_{10}$ | (a 4-bit unsigned number) |
| 0 1101 | $= +13_{10}$ | (a positive number in 5-bit signed magnitude) |
| 1 1101 | $= -13_{10}$ | (a negative number in 5-bit signed magnitude) |

| | | |
|---|---|---|
| $0100_2$ | $= 4_{10}$ | (a 4-bit unsigned number) |
| 0 0100 | $= +4_{10}$ | (a positive number in 5-bit signed magnitude) |
| 1 0100 | $= -4_{10}$ | (a negative number in 5-bit signed magnitude) |

## One’s complement representation

- A different approach, one’s complement, negates numbers by complementing each bit of the number.
- We keep the sign bits: 0 for positive numbers, and 1 for negative. The sign bit is complemented along with the rest of the bits.
- Examples:

$1101_2 = 13_{10}$ (a 4-bit unsigned number)

0 1101 $= +13_{10}$ (a positive number in 5-bit one’s complement)

1 0010 $= -13_{10}$ (a negative number in 5-bit one’s complement)

$0100_2 = 4_{10}$ (a 4-bit unsigned number)

0 0100 $= +4_{10}$ (a positive number in 5-bit one’s complement)

1 1011 $= -4_{10}$ (a negative number in 5-bit one’s complement)

## Two's complement

- Our final idea is two's complement. To negate a number, complement each bit (just as for ones' complement) and then add 1.
- Examples:

|  |  |
|---|---|
| $1101_2 = 13_{10}$ | (a 4-bit unsigned number) |
| $0\ 1101 = +13_{10}$ | (a positive number in 5-bit two's complement) |
| $1\ 0010 = -13_{10}$ | (a negative number in 5-bit *ones'* complement) |
| $1\ 0011 = -13_{10}$ | (a negative number in 5-bit two's complement) |
| $0100_2 = 4_{10}$ | (a 4-bit unsigned number) |
| $0\ 0100 = +4_{10}$ | (a positive number in 5-bit two's complement) |
| $1\ 1011 = -4_{10}$ | (a negative number in 5-bit *ones'* complement) |
| $1\ 1100 = -4_{10}$ | (a negative number in 5-bit two's complement) |

## Ranges of the signed number systems

- How many negative and positive numbers can be represented in each of the different systems on the previous page?

| | Unsigned | Signed Magnitude | One's complement | Two's complement |
|---|---|---|---|---|
| Smallest | 0000 (0) | 1111 (-7) | 1000 (-7) | 1000 (-8) |
| Largest | 1111 (15) | 0111 (+7) | 0111 (+7) | 0111 (+7) |

- In general, with n-bit numbers including the sign, the ranges are:

| | Unsigned | Signed Magnitude | One's complement | Two's complement |
|---|---|---|---|---|
| Smallest | 0 | $-(2^{n-1}-1)$ | $-(2^{n-1}-1)$ | $-2^{n-1}$ |
| Largest | $2^n-1$ | $+(2^{n-1}-1)$ | $+(2^{n-1}-1)$ | $+(2^{n-1}-1)$ |

## Making a subtraction circuit

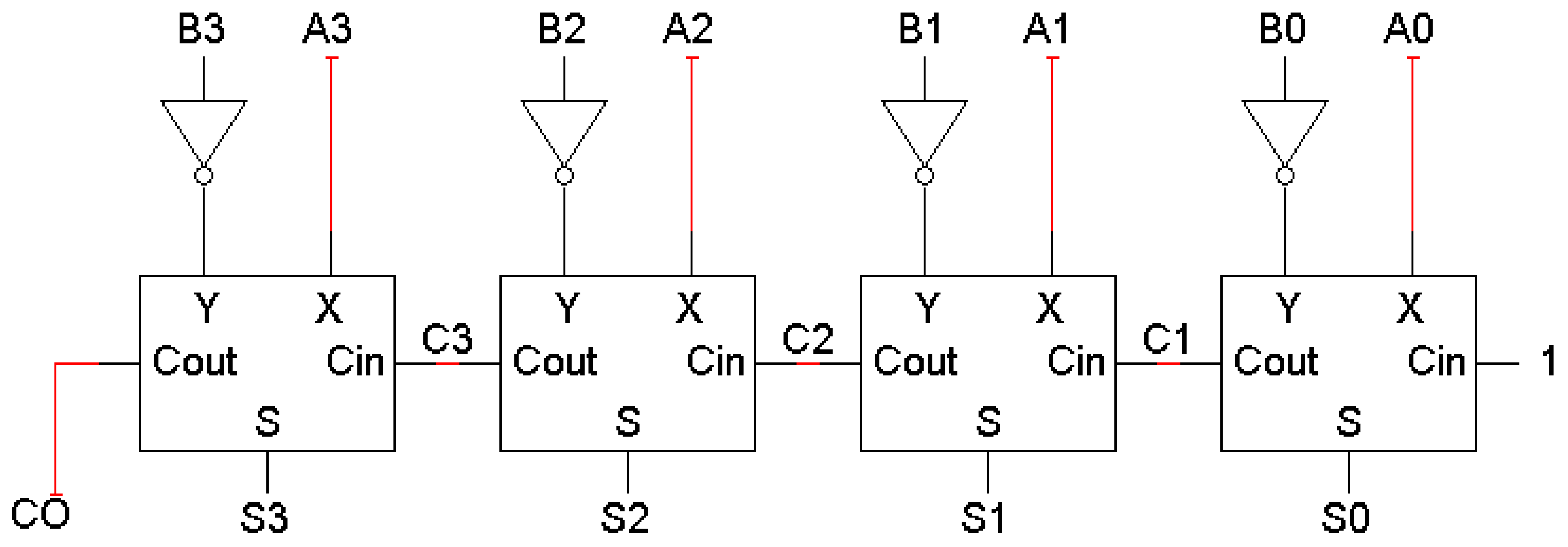
- We could build a subtraction circuit directly, similar to the way we made unsigned adders.
- However, by using two’s complement we can convert any subtraction problem into an addition problem. Algebraically,

$$A - B = A + (-B)$$

- So to subtract B from A, we can instead *add* the negation of B to A.
- This way we can re-use the unsigned adder hardware.

# A two’s complement subtraction circuit

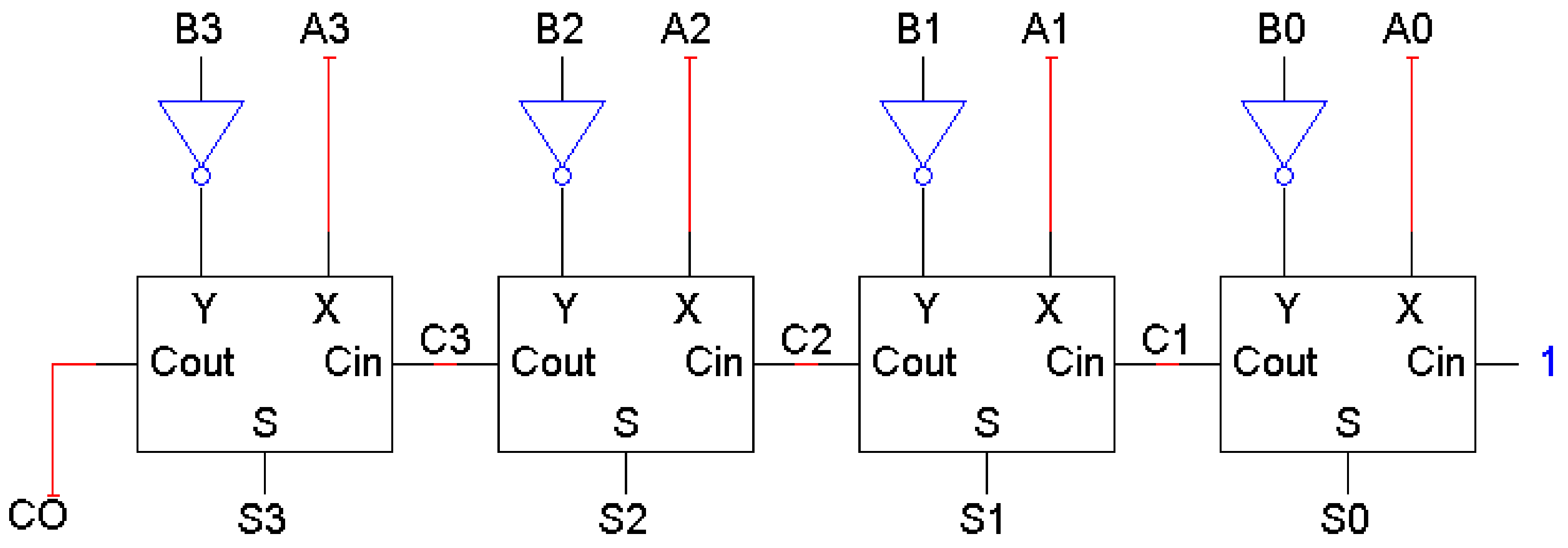
- To find A - B with an adder, we’ll need to:
  - Complement each bit of B.
  - Set the adder’s carry in to 1.
- The net result is A + B’ + 1, where B’ + 1 is the two’s complement negation of B.



- Remember that A3, B3 and S3 here are actually sign bits.

## Small differences

- The only differences between the adder and subtractor circuits are:
  - The subtractor has to negate B3 B2 B1 B0.
  - The subtractor sets the initial carry in to 1, instead of 0.



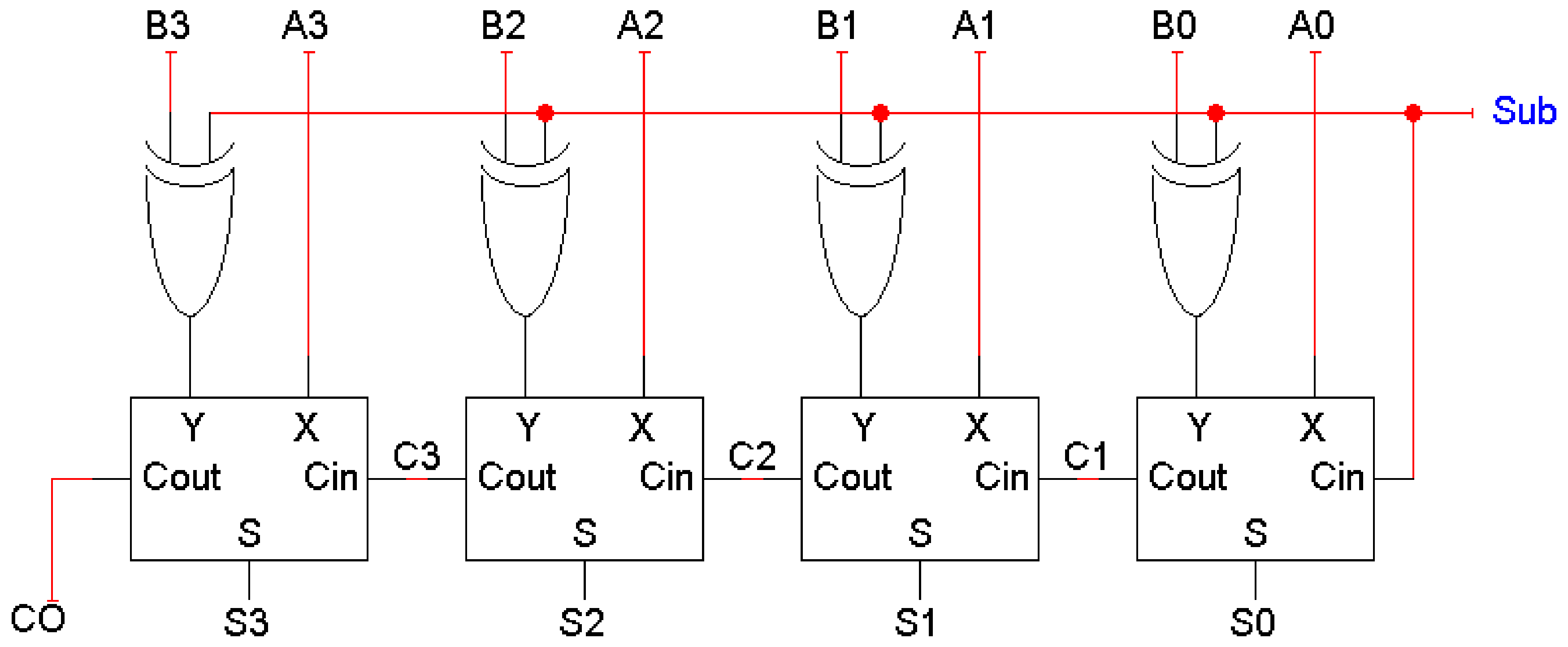
- It's not too hard to make one circuit that does *both* addition and subtraction.

## An adder-subtractor circuit

- XOR gates let us selectively complement the B input.

  $X \oplus 0 = X$ $X \oplus 1 = X'$

- When Sub = 0, the XOR gates output B3 B2 B1 B0 and the carry in is 0. The adder output will be A + B + 0, or just A + B.
- When Sub = 1, the XOR gates output B3' B2' B1' B0' and the carry in is 1. Thus, the adder output will be a two's complement subtraction, A - B.



## Signed overflow

- With two's complement and a 4-bit adder, for example, the largest representable decimal number is +7, and the smallest is -8.
- What if you try to compute 4 + 5, or (-4) + (-5)?

```
     0 1 0 0    (+4)             1 1 0 0    (-4)
 +   0 1 0 1    (+5)         +   1 0 1 1    (-5)
 ------------------          ------------------
   0 1 0 0 1    (-7)           1 0 1 1 1    (+7)
```

- We cannot just include the carry out to produce a five-digit result, as for unsigned addition. If we did, (-4) + (-5) would result in +23!
- Also, unlike the case with unsigned numbers, the carry out *cannot* be used to detect overflow.
  - In the example on the left, the carry out is 0 but there *is* overflow.
  - Conversely, there are situations where the carry out is 1 but there is *no* overflow.

## Detecting signed overflow

- The easiest way to detect signed overflow is to look at all the sign bits.

$$\begin{array}{rrl} & \boxed{0}\,1\,0\,0 & (+4) \\ + & \boxed{0}\,1\,0\,1 & (+5) \\ \hline & 0\,1\,0\,0\,1 & (-7) \end{array} \qquad\qquad \begin{array}{rrl} & \boxed{1}\,1\,0\,0 & (-4) \\ + & \boxed{1}\,0\,1\,1 & (-5) \\ \hline & 1\,0\,1\,1\,1 & (+7) \end{array}$$

- Overflow occurs only in the two situations above:
  - If you add two *positive* numbers and get a *negative* result.
  - If you add two *negative* numbers and get a *positive* result.
- Overflow cannot occur if you add a positive number to a negative number. Do you see why?

## Sign extension

- In everyday life, decimal numbers are assumed to have an infinite number of 0s in front of them. This helps in “lining up” numbers.
- To subtract 231 and 3, for instance, you can imagine:

$$\begin{array}{r} 231 \\ \underline{-\ \ 003} \\ 228 \end{array}$$

- You need to be careful in extending signed binary numbers, because the leftmost bit is the *sign* and not part of the magnitude.
- If you just add 0s in front, you might accidentally change a negative number into a positive one!
- For example, going from 4-bit to 8-bit numbers:
  - 0101 (+5) should become 0000 0101 (+5).
  - But 1100 (-4) should become 1111 1100 (-4).
- The proper way to extend a signed binary number is to replicate the sign bit, so the sign is preserved.