Lecture -5
Combination Circuits-1

Prepared By: Dr. Shahriyar Masud Rizvi

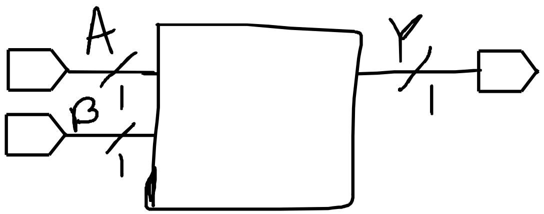
Adder

- Addition is one of the most common operations performed by computing devices such as computers, mobile phones, and in fact, any device that is powered by a microprocessor.
- They are essential in digital filters as well as many machine learning models/devices.

- Let us consider adding two binary bits.
- The rules of binary addition provide us the following results.
  - $0 + 0 = 0$
  - $0 + 1 = 1$
  - $1 + 0 = 1$
  - $1 + 1 = 0$, with a carryout of 1

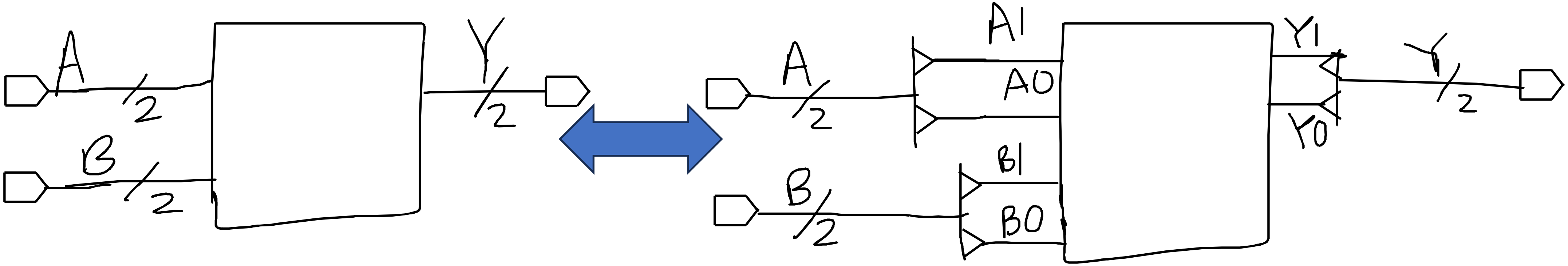
Adder

- Wires vs Busses
- Digital signals can be grouped together depending on their operation or source for ease of verification. A grouping of digital signals (bundle of wires) is called a bus.



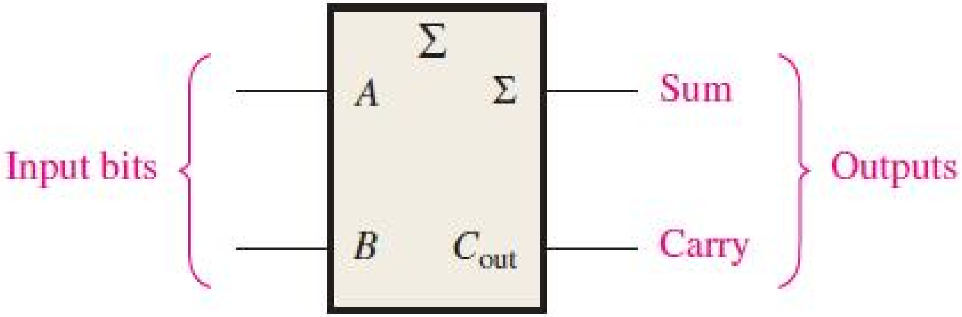
Here, A, B and Y are 1-bit signals.

The following device, A, B and Y are busses, which are 2-bit wide. In other words, they are 2-bit signals.
A bus is composed of bits A0 and A1, where they represent LSB and MSB respectively.
Similarly, B bus is composed of bits B0 and B1 and Y bus is composed of bits Y0 and Y1.

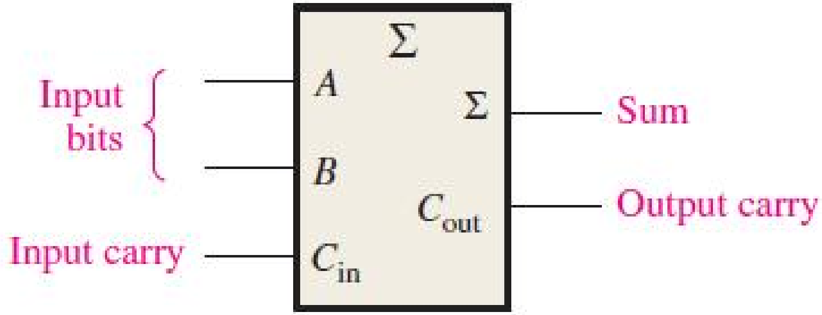


1-bit Adders

- There are two types of 1-bit adders namely, **Half-adder** and a **Full-adder**.
- Half adder can add two 1-bit signals, while Full adder can add three 1-bit signals.



Block diagram of a Half-Adder



Block diagram of a Full-Adder

Multi-bit Adders

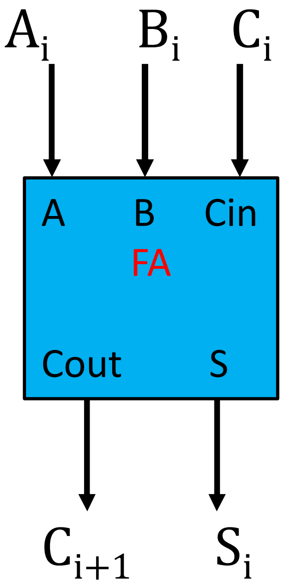
- Multi-bit adders are constructed by cascading (series placement) of full adders.
- In multi-bit adders, carryout of one stage becomes carry in of next stage.
- This happens for traditional (decimal-based) addition also as shown below.
- As shown below, when addition of two numbers (such as 9 and 5) cannot be represented by the available numbers (0 to 9), a 1 is added to the addition of next stage. Here, 1 is carry out of stage 0 and carry in of stage 1.

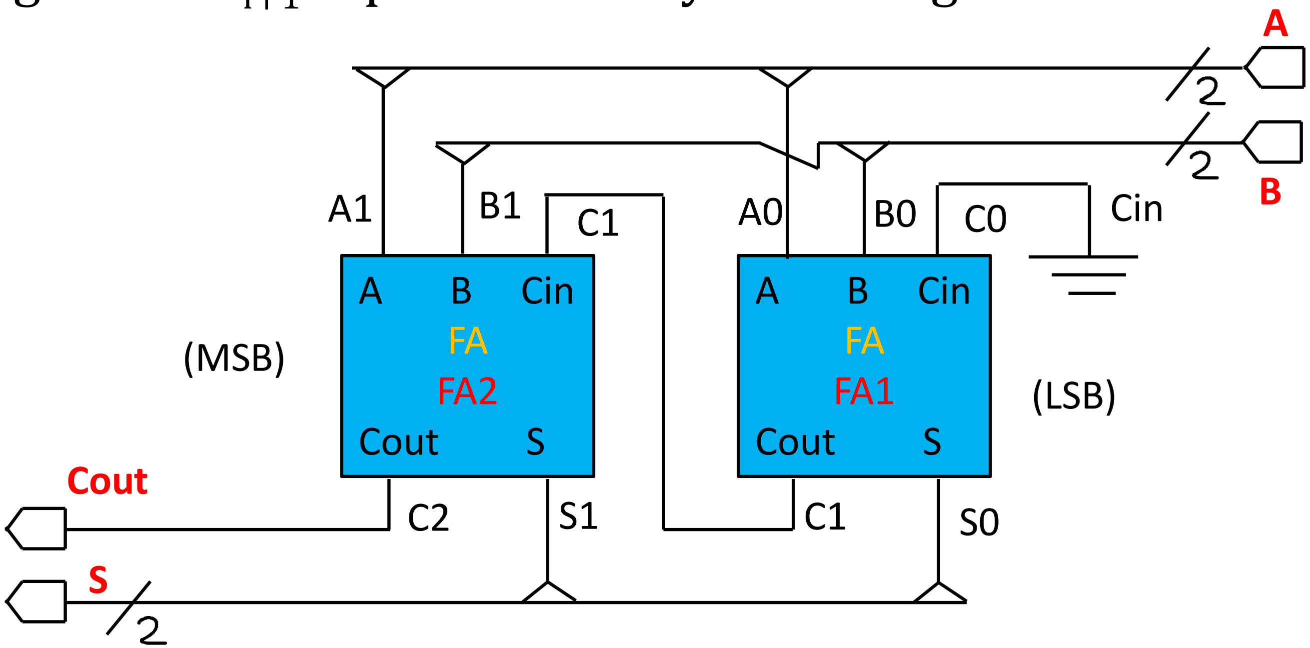
stage1 stage0

Carry	1	
Number1	2	9
Number2	3	5
Sum	6	4

Multi-bit Adders

- Multi-bit adders are constructed by cascading (series placement) of full adders.
- Carryout of full adders that add lesser significant bits becomes carry in of full adders that add more significant bits.
- So, input signals of a full adder in a multi-bit adder (adder chain) like the following can be denoted as A_i, B_i and C_i, while the output signals can be denoted as S_i and C_{i+1}. Here, C_i represents carryin of stage i and C_{i+1} represents carryout of stage i.

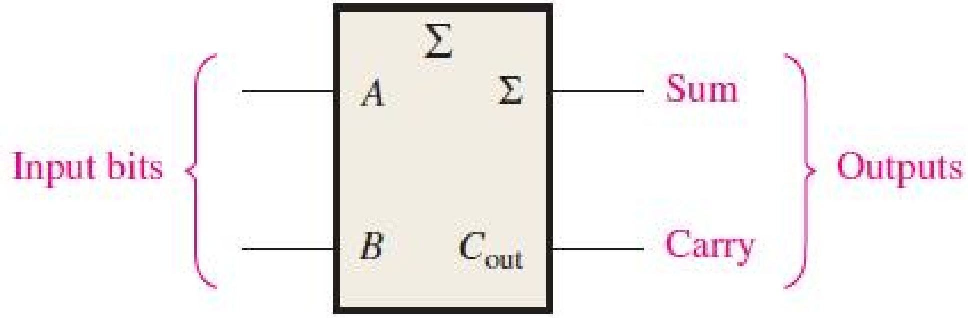


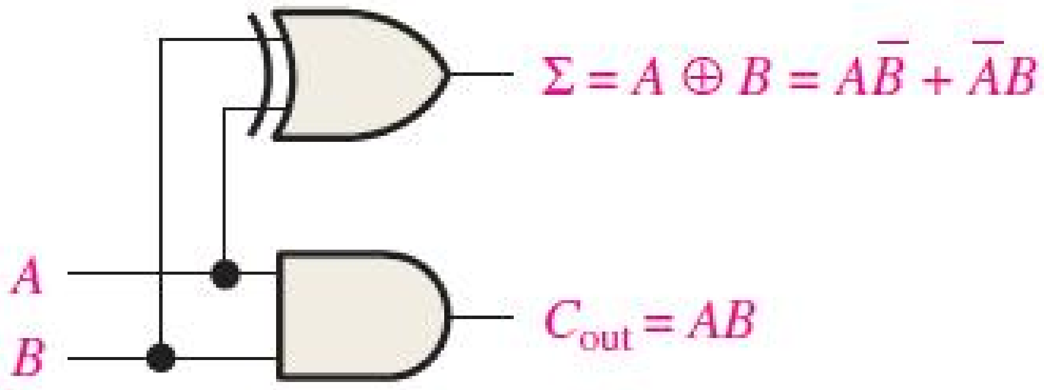


Half Adder

- Half adder can add two 1-bit signals. It produces a sum and a carryout.
- Let us denote the inputs as A and B and outputs as S and C_{OUT}

A	B	S	C_{OUT}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



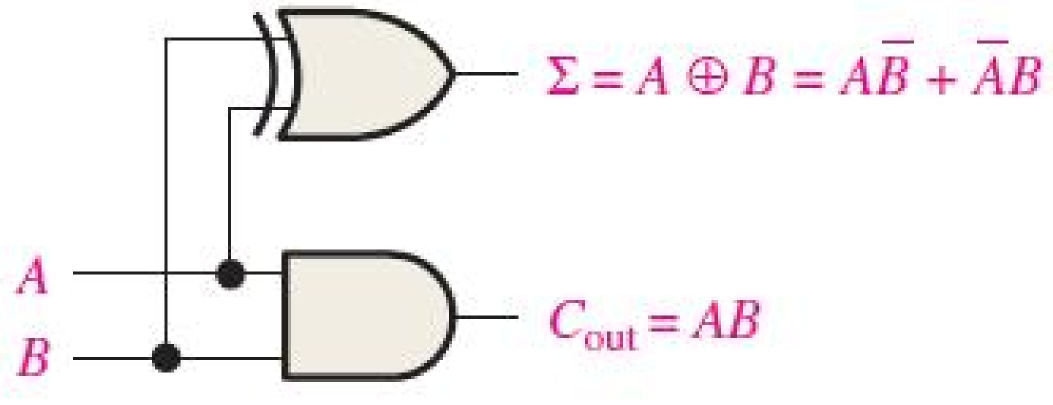


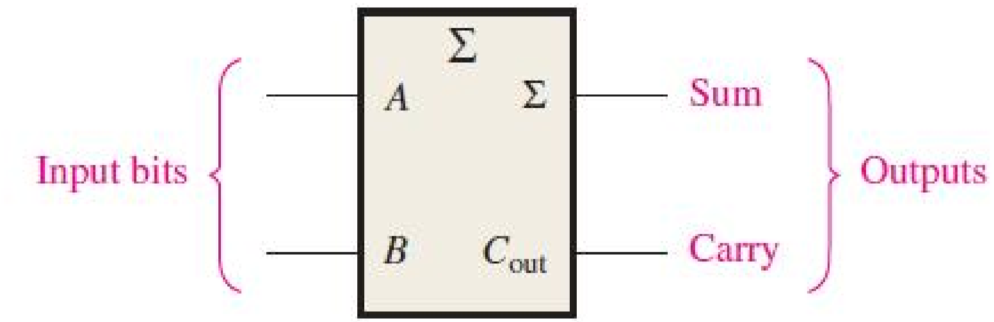
- $S = \bar{A}.\mathrm{B} + \mathrm{A}.\bar{B} = \mathrm{A} \oplus \mathrm{B}$
- $C_{OUT} = \mathrm{A} . \mathrm{B}$

Half Adder

- $S = \bar{A}.\mathrm{B} + \mathrm{A}.\bar{B} = \mathrm{A} \oplus \mathrm{B}$
- $\mathrm{C_{OUT}} = \mathrm{A} \,.\, \mathrm{B}$

```
//Device name and I/O ports
module HA (input wire A, B,
           output wire S, Cout);

//Define behavior/structure of the circuit

assign S= A ^ B;
assign Cout= A & B;

endmodule
```

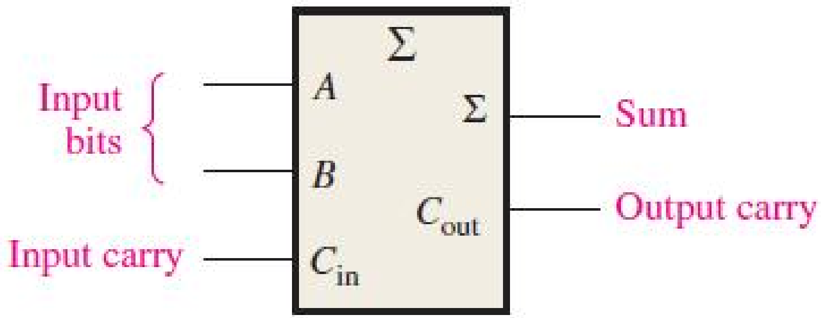




Full Adder

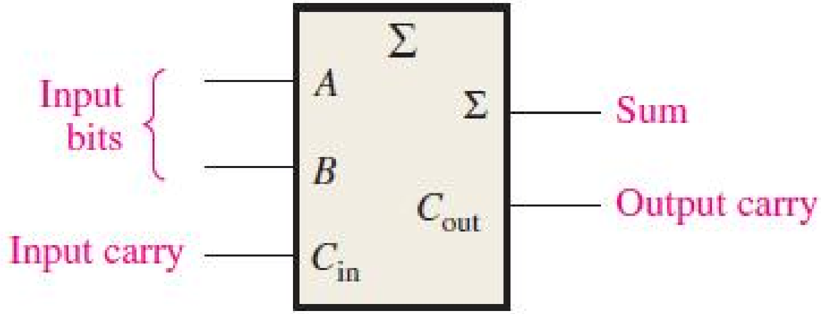
- Half adder can add two 1-bit signals. It produces a sum and a carryout.
- Let us denote the inputs as A and B and outputs as S and C_{OUT}
- $\mathbf{S = \overline{A}\,\overline{B}C_{IN} + \overline{A}B\overline{C_{IN}} + A\overline{B}\,\overline{C_{IN}} + ABC_{IN} = A \oplus B \oplus C}$
- $\mathbf{C_{OUT} = \overline{A}BC_{IN} + A\overline{B}C_{IN} + AB\overline{C_{IN}} + ABC_{IN} = AB + (A \oplus B)C_{IN}}$

C_{IN}	A	B	S	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

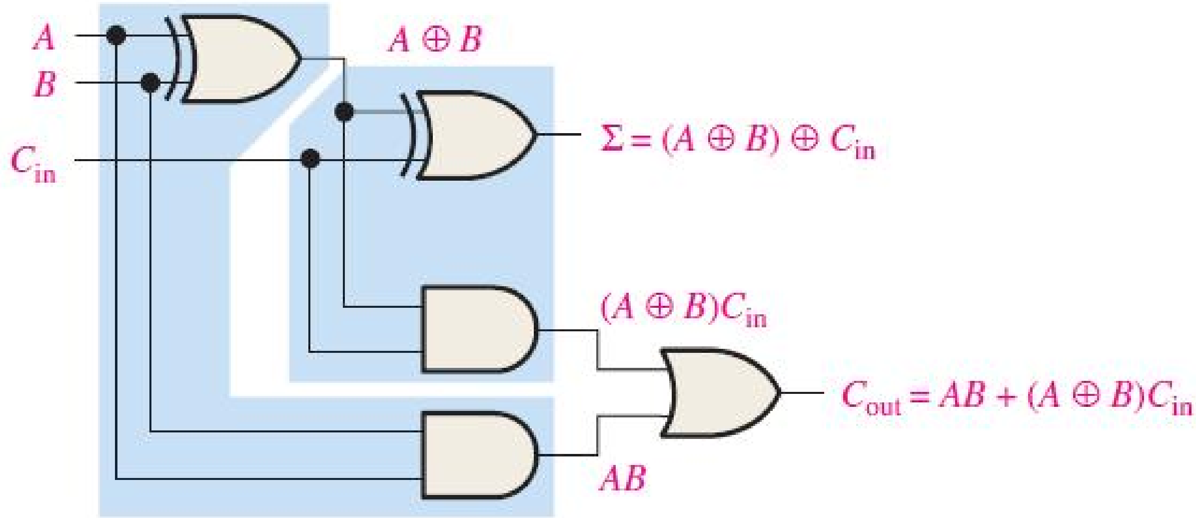


Full Adder

- $\mathbf{S = \overline{A}\,\overline{B}C_{IN} + \overline{A}B\overline{C_{IN}} + A\overline{B}\,\overline{C_{IN}} + ABC_{IN} = A \oplus B \oplus C}$
- $\mathbf{C_{OUT} = \overline{A}BC_{IN} + A\overline{B}C_{IN} + AB\overline{C_{IN}} + ABC_{IN} = AB + (A \oplus B)C_{IN}}$

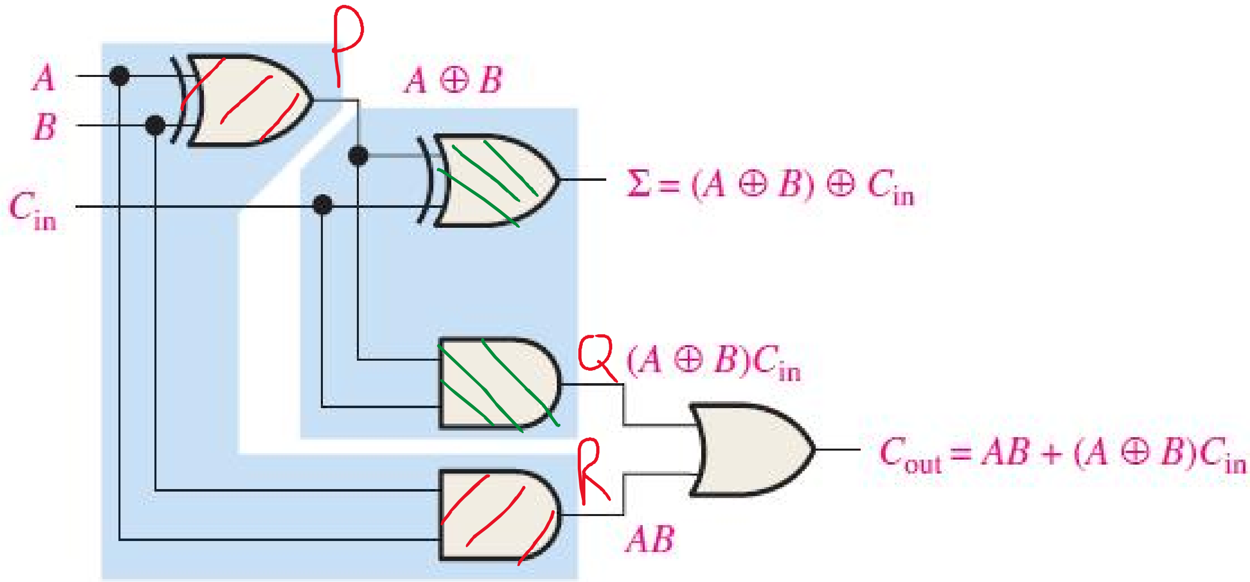



```
//Device name and I/O ports
module FA (input wire Cin, A, B,
           output wire S, Cout);

//Define behavior/structure of the circuit

assign S= A ^ B ^ Cin;
assign Cout= (A & B) | ((A ^ B ) & Cin);

endmodule
```



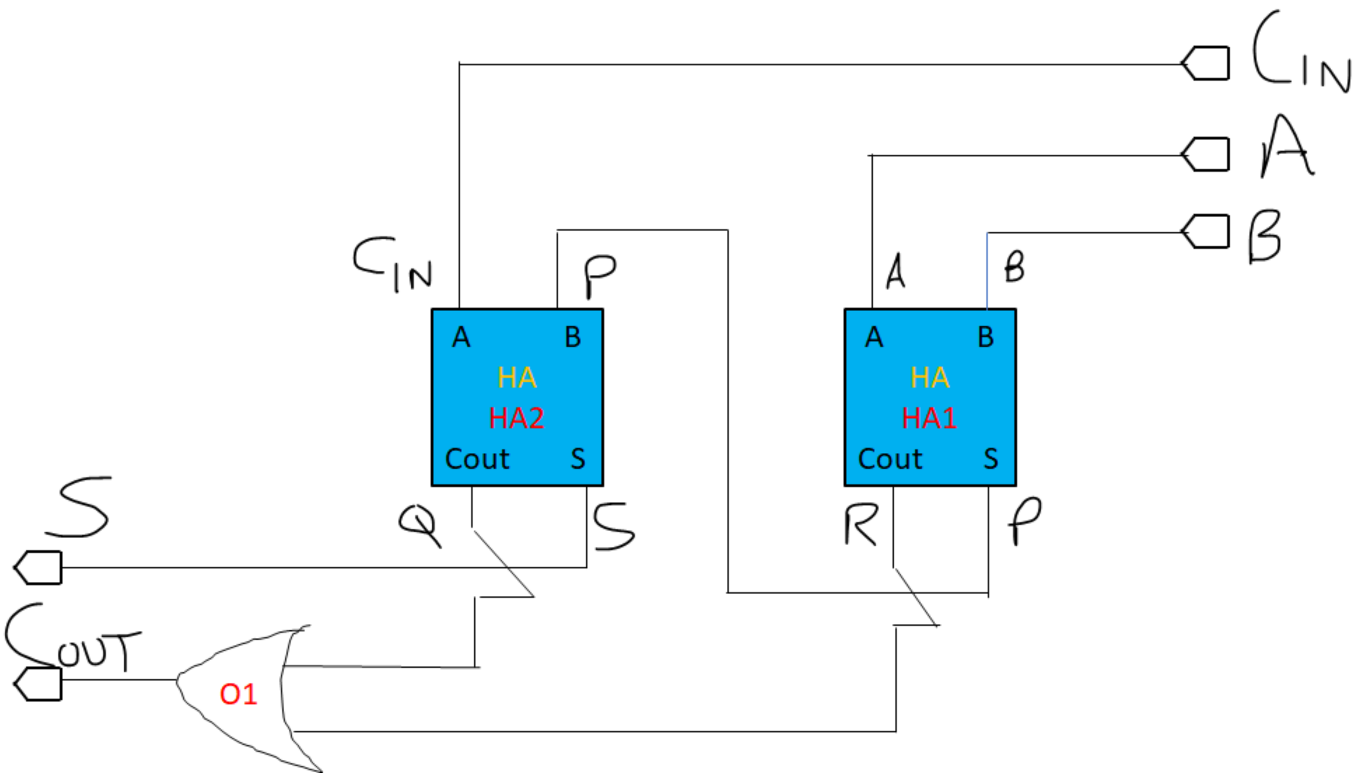
Full Adder from Half Adders

- A Full Adder can be built from two half adder and an OR gate. Let's examine the schematic.
- A full adder does a "half" addition of A and B and another "half" addition of (A⊕B) and C_{IN}. It also does a half-adder style carryout computation of A and B and another such carryout computation between (A⊕B) and C_{IN}.
- So, the XOR gate in the figure generating P and the AND gate generating R are forms a half adder. The XOR gate generating S and the AND gate generating Q forms another half adder. An OR gate is needed to compute COUT, which performs OR operation between Q and R.



Full Adder from Half Adders

- Note that half adder (HA) now is a sub-system (component) of the overall system that is full adder.
- The port names of a generic HA are A, B, as inputs and S and COUT as outputs. Ports of different instances (copies) of HA (such as HA1, HA2) will have different actual signals (such as A, B, CIN, P, R, P, Q, S) connected to them.

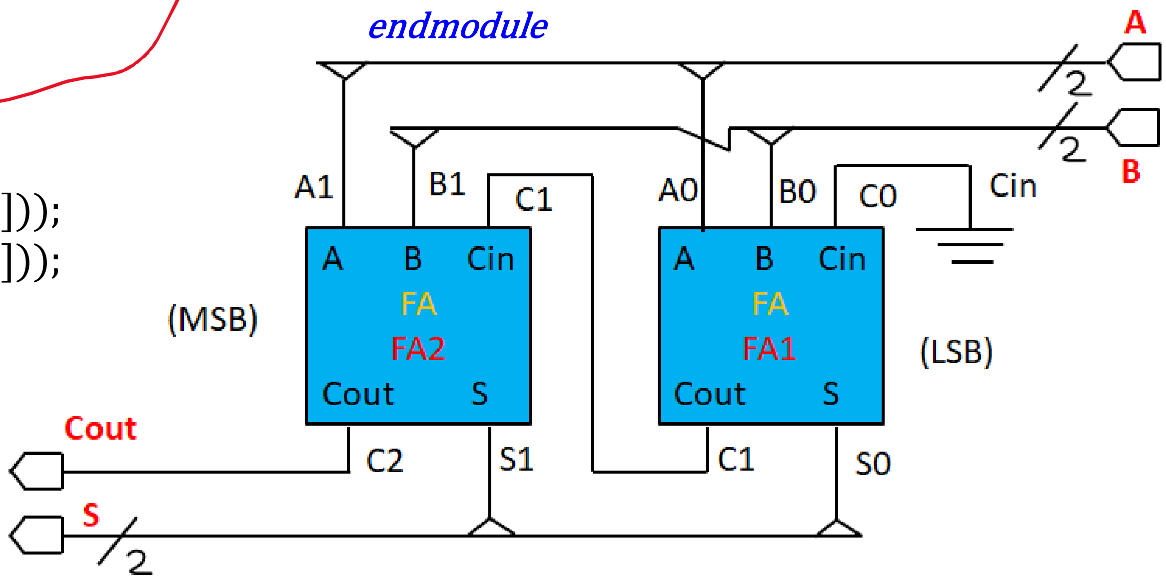


Multi-bit Adders from Full Adders

- A 2-bit adder can be constructed by cascading (series placement) of two full adders.

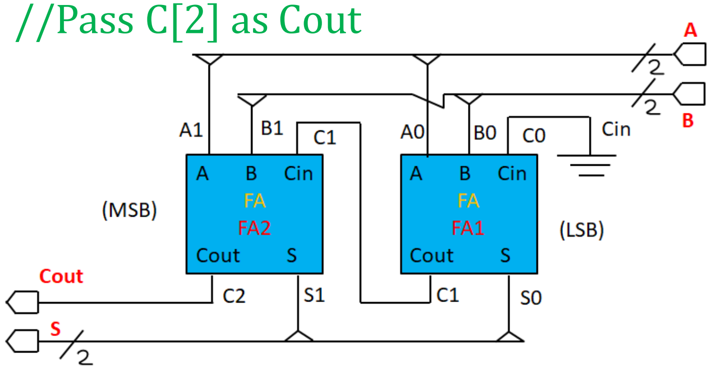
```
//Top-Level module
//Device name and I/O ports
module Adder_2Bit (input wire [1:0] A, B,
                   output wire [1:0] S,
                   output wire Cout);

//Define internal C incorporating all carry type signals
wire [2:0] C;
assign C[0] = 0;   //Use C[0] just for a “symmetric look”

//Define behavior/structure of the circuit
//Instantiating two half adders
FA FA1 (.Cin(C[0]), .A(A[0]), .B(B[0]), .S (S[0]), .Cout (C[1]));
FA FA2 (.Cin(C[1]), .A(A[1]), .B(B[1]), .S (S[1]), .Cout (C[2]));

assign Cout = C[2];      //Pass C[2] as Cout

endmodule
```

```
//Level-2
//Device name and I/O ports
module FA (input wire Cin, A, B,
           output wire S, Cout);

//Define behavior/structure of the circuit

assign S= A ^ B ^ Cin;
assign Cout= (A & B) | ((A ^ B ) & Cin);

endmodule
```



Multi-bit Adders from Full Adders

```
//Top-Level module
//Device name and I/O ports
module Adder_2Bit (input wire [1:0] A, B,
                   output wire [1:0] S,
                   output wire Cout);

//Define internal C incorporating all carry type signals
wire [2:0] C;
assign C[0] = 0;   //Use C[0] just for a "symmetric look"

//Define behavior/structure of the circuit
//Instantiating two half adders
FA FA1 (.Cin(C[0]), .A(A[0]), .B(B[0]), .S (S[0]), .Cout (C[1]));
FA FA2 (.Cin(C[1]), .A(A[1]), .B(B[1]), .S (S[1]), .Cout (C[2]));

assign Cout = C[2];      //Pass C[2] as Cout

endmodule
```



Syntax

In Verilog, a sub-system (component) can be defined inn a *module* and then instantiated in an upper-level *module*.

Here, the Adder_2bit is built from two instances of FA (full adder). So, FA is the component. FA1 and FA2 are instances of FA. You need to write *module* only for FA, since FA1 and FA2 are just copies (instances) of FA.

When components are instantiated, generic ports of a component are preceded by a "." and then the actual signal coming in or going out through that port for a specific instance is listed inside a bracket.

So, FA FA1 (.Cin(C[0]), .A (A[0], …………); means C[0] is connected to Cin port of FA1 (which is an instance of FA). Similarly, A[0] signals is connected to port A of FA1.

Component_name instance_name (.port1_name(signal1_name), .port2_name(signal2_name), .port3_name(signal3_name)……);

Multi-bit Adders from Full Adders

```
//Top-Level module
//Device name and I/O ports
module Adder_2Bit (input wire [1:0] A, B,
                   output wire [1:0] S,
                   output wire Cout);

//Define internal C incorporating all carry type signals
wire [2:0] C;
assign C[0] = 0;   //Use C[0] just for a “symmetric look”

//Define behavior/structure of the circuit
//Instantiating two half adders
FA FA1 (.Cin(C[0]), .A(A[0]), .B(B[0]), .S (S[0]), .Cout (C[1]));
FA FA2 (.Cin(C[1]), .A(A[1]), .B(B[1]), .S (S[1]), .Cout (C[2]));

assign Cout = C[2];     //Pass C[2] as Cout

endmodule
```

Syntax

In Verilog, a bus is defined in the following manner. The MSB and LSB are mentioned in a bracket, separated by a colon (:).

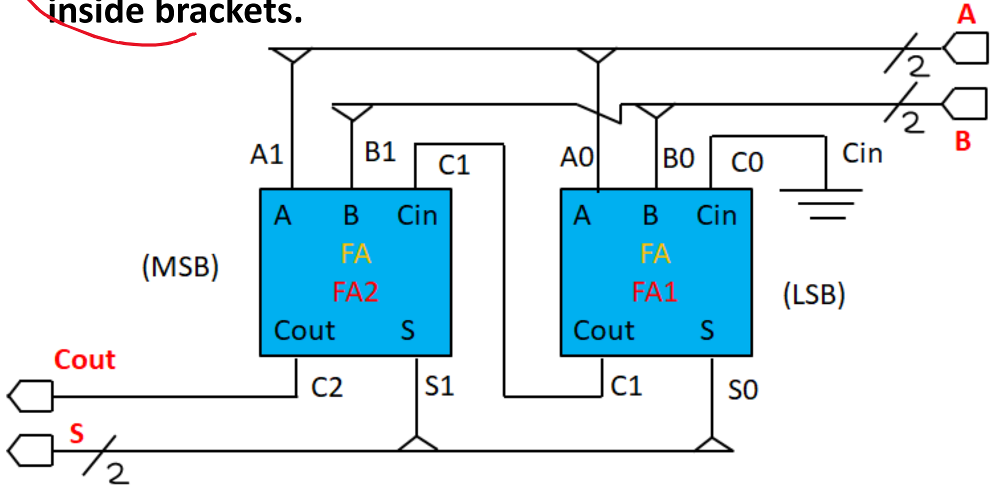
A 2-bit bus A will be declared as wire [1:0] A
Here MSB is bit-1, LSB is bit-0.

A 3-bit bus A will be declared as wire [2:0] A
Here MSB is bit-2, LSB is bit-0.

A 4-bit bus A will be declared as wire [3:0] A
Here MSB is bit-3, LSB is bit-0.

Note that single-bit signals such as Cout do not need to defined with MSB and LSB.

Single bits from busses can be referenced with specific bits inside brackets.



Overflow, Number Wheel

- In digital design, input and output sizes of a sub-system are kept the same in many cases. So, if this is followed the carryout becomes an extra signal. Even if carryout is not considered part of the result due to the above requirement, it can serve another purpose—that is it serves as an overflow indicator.
- In addition, HIGH carryout indicates overflow.
- Overflow means that the result could not be mapped within the available number of bits.

- For example, consider $011\ (3) + 110\ (6) = 001\ (1)$ in a 3-bit adder.

$$
\begin{array}{r}
011\ (3) \\
+\ \ 110\ (6) \\
\hline
1\ 001\ (1)
\end{array}
$$

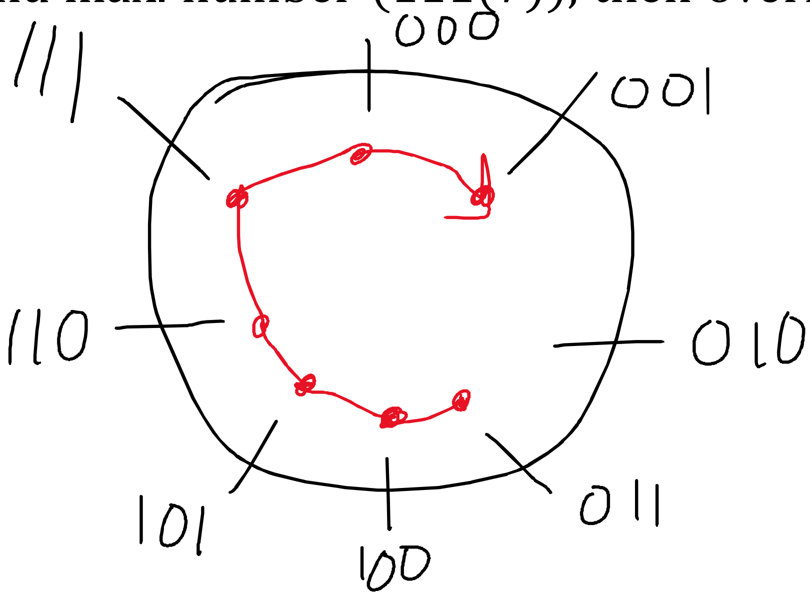
Overflow, Number Wheel

- For example, consider 011 (3) + 110 (6) in a 3-bit adder.

$$
\begin{array}{r}
011\ (3) \\
+\ 110\ (6) \\
\hline
1\ 001\ (1)
\end{array}
$$

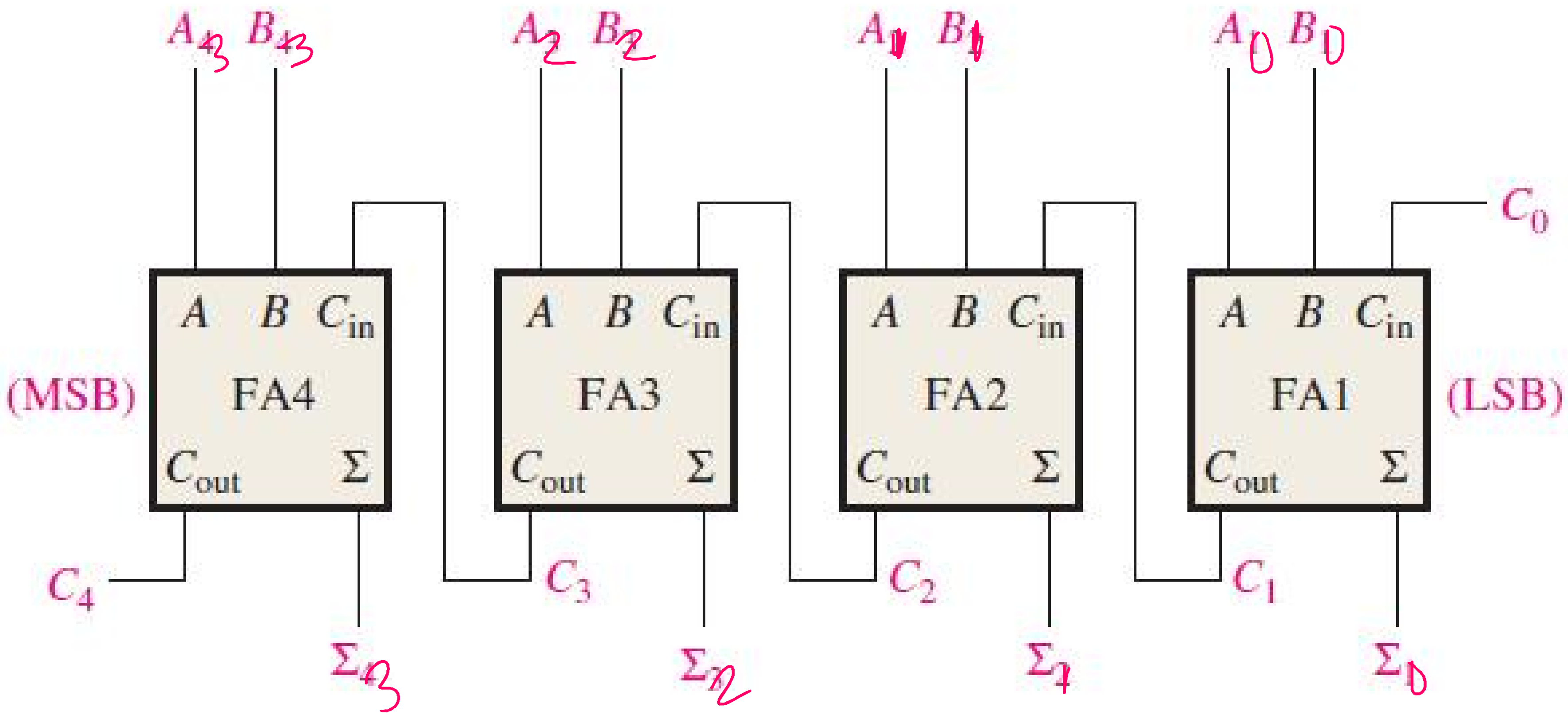
We can graphically represent the addition in a number wheel.
We go clock-wise for addition.
For this example, start at 3, and then go 6 steps clock-wise to get the addition result.

If the arrow goes beyond max. number (111(7)), then overflow occurs. From number wheel, we get 3 + 6 = 1 as well.



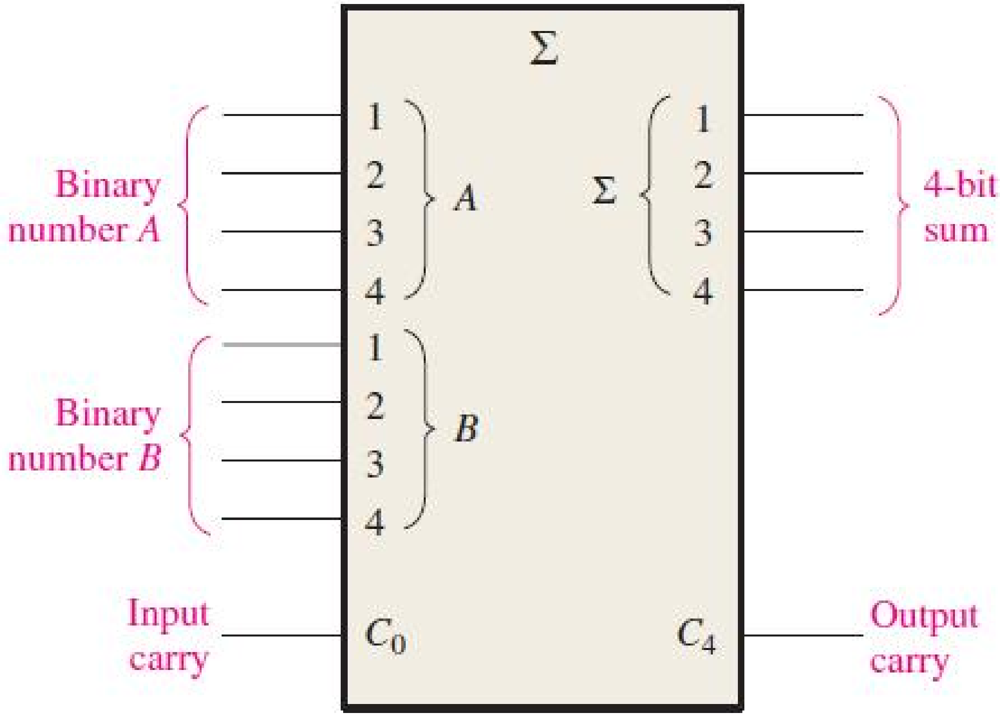
Parallel Adder

- A 4-bit Parallel adder to add two 4bit numbers can be designed using the same principle.



Adder Expansion

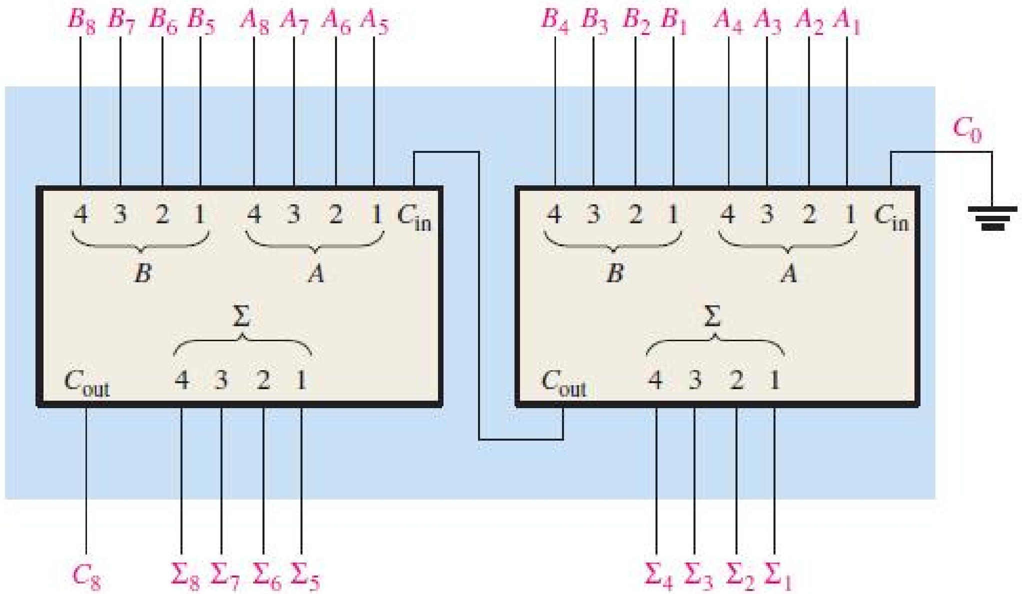
- If an n-bit adder IC is available, larger sized adders can be built by cascading multiple n-bit adders.
- For instance, an 8-bit adder can be built from two 4-bit adders.
- A 4-bit is called a nibble.
- This is the concept of adder expansion.



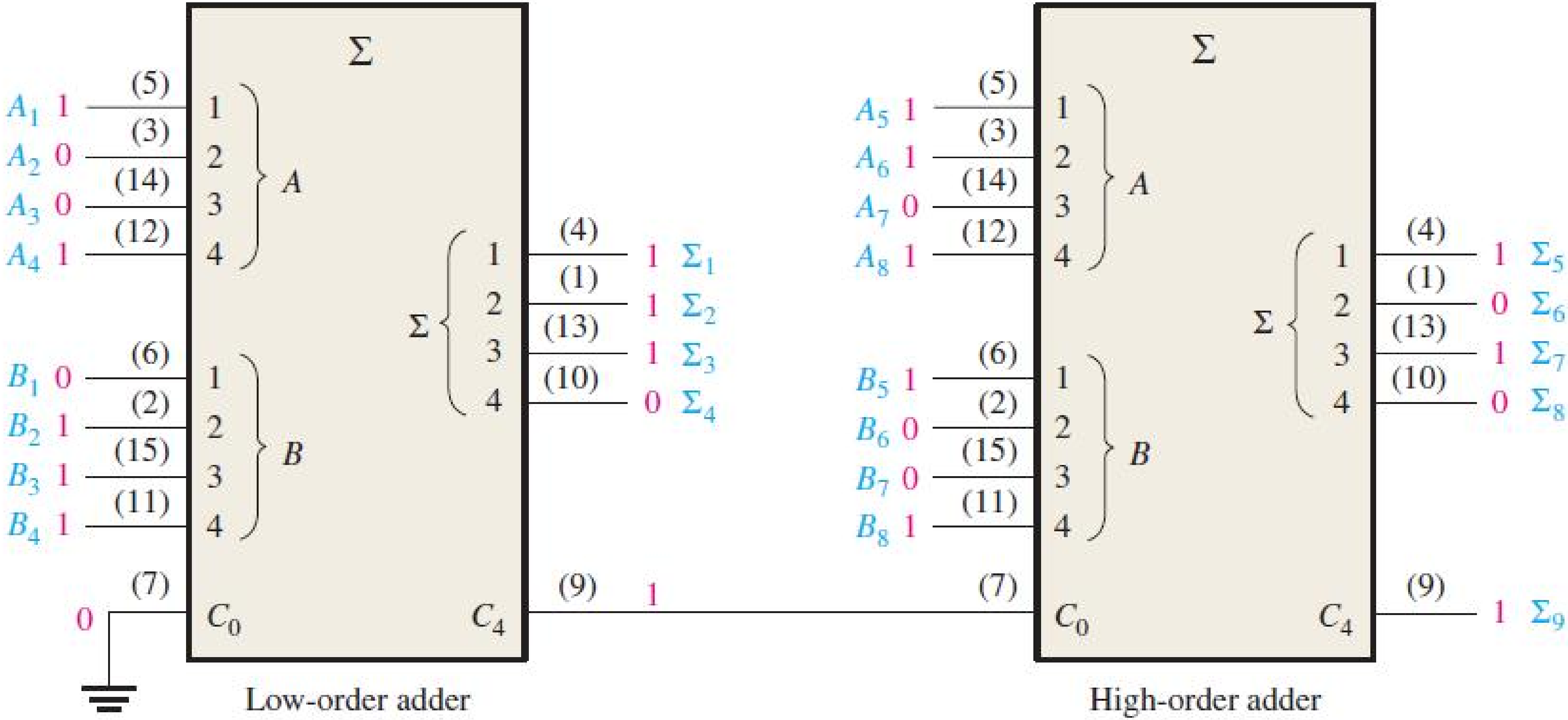
Adder Expansion

- We can cascade two blocks of 4-bit parallel adder to add two numbers of 8-bits.

		1	1	1					
	1	0	1	1		1	0	0	1
	1	0	0	1		1	1	1	0
1	0	1	0	1		0	1	1	1
C	S_8	S_7	S_6	S_5		S_4	S_3	S_2	S_1



Example of Adder Expansion



Subtractor

- The logic equation for subtraction can be developed from truthtable. One can build half-subtractor and full-subtractor from truthtable. Similar, to adders, multi-bit subtracters can be built from full-subtractor.
- However, subtracters are typically realized with 2’s compliment addition in industrial EDA tools. This is because, when 2’s compliment subtraction is used, adders can be used for both addition and subtraction purposes.
- Subtraction of two numbers is defined as addition of 2’s compliment version of one of the inputs with the other inputs kept unchanged. So, one input is kept unchanged, while the other input is provided to the adder in its 2’s compliment form.

$$
\begin{aligned}
Y_{sub} &= A - B \\
&= A + (-B) \\
&= A + (2\text{'s compliment of } B)
\end{aligned}
$$

Subtractor

- 2’s compliment is formed from addition of 1’s compliment with 1.
- 1’s compliment is the NOT operation performed on an input.

So, $Y_{sub} = A + (2\text{'s compliment of } B)$

$= A + (1\text{'s compliment of } B + 1)$

$= A + \bar{B} + 1$

- Example,
- Subtract 3 from 5 in a 3-bit arithmetic.

- $A = 101\ (5)$
- $B = 011\ (3) \rightarrow$ 1’sC of $B = 100 \rightarrow$ 2’s C of $B = 100 + 1 = 101$

- $A = 101\ (5)$
- 2’sC of $B = 101$ (2’sC of B)
- $Y_{sub} = 1\ 010\ (2)$, carryout is not part of the subtraction result.

Subtractor

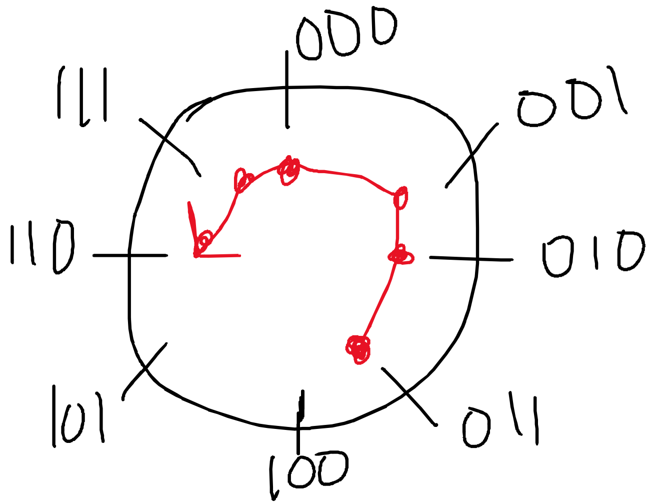
- Example,
- Subtract 5 from 3 in a 3-bit arithmetic.

- A = 011 (3)
- B = 101 (5)→ 1'sC of B = 010→ 2's C of B = 010 + 1 = 011

- A = 011 (3)
- 2'sC of B = 011 (2'sC of B)
- Y_{sub} = 0 110 (6), carryout is not part of the subtraction result.

Overflow (Subtraction), Number Wheel

- In subtraction, LOW carryout indicates overflow.
- Subtraction Overflow or underflow means that the result could not be mapped within the available number of bits. This happens when a large number is subtracted from a small number. In integer arithmetic, there is no negative number. In other words, there is no number smaller than zero.
- We can graphically represent the subtraction in a number wheel.
- One needs to go counter-clock-wise direction for subtraction.
- For this example, to get the result of 3 – 5, start at 3, and then go 5 steps counter-clock-wise to get the subtraction result.
- If the arrow goes beyond min. number (000(0)), then overflow occurs. From number wheel, we get 3 – 5 = 6 as well.



Adder-Subtractor

- Since both addition and subtraction uses addition, one can develop a common expression for addition and subtraction provided there is a control signal that selects whether addition or subtraction should be performed.
- Addition→ A + B
  Subtraction→ A – B = A + (-B) = A + $\bar{B}$ + 1
- If we can control the 2nd input to the adder (say, B_to_add), regarding whether it should be B or $\bar{B}$, we can use the same adder to produce both addition and subtraction.
- Secondly, subtraction requires an extra 1, while the addition does not need any.
- So, if the control signal, say AS, is chosen in such a way that a AS = 0 ensures addition and AS = 1 ensures subtraction, then the control signal itself can be added to the addition process. This will provide that extra 1 for subtraction.
- So, the common equation for adder-subtractor becomes **S = A + B_to_add + AS.**
- In addition, this equation becomes A + B + 0. For subtraction, it becomes A + $\bar{B}$ + 1.

	AS	B_to_add	S	S values
Addition	0	B	A + B_to_add + AS	A + B + 0
Subtraction	1	$\bar{B}$		A + $\bar{B}$ + 1

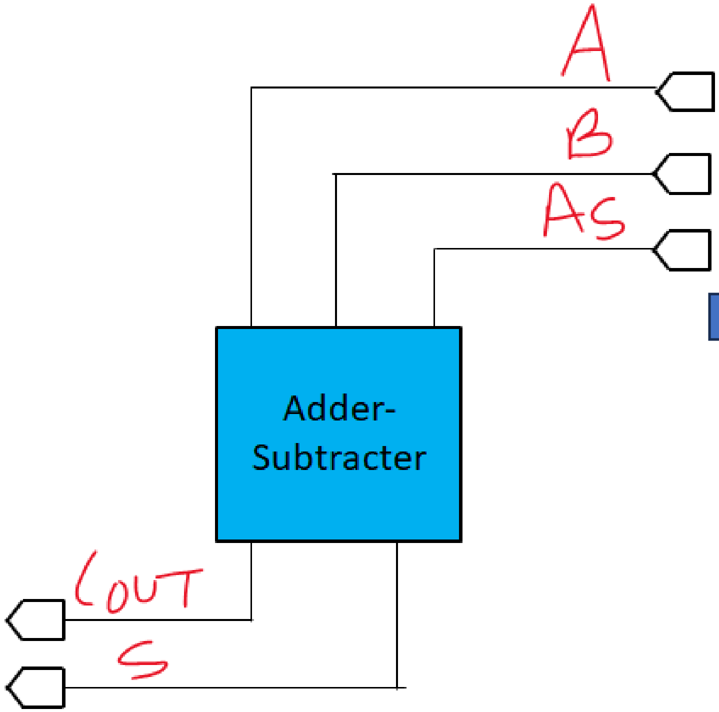
Adder-Subtractor

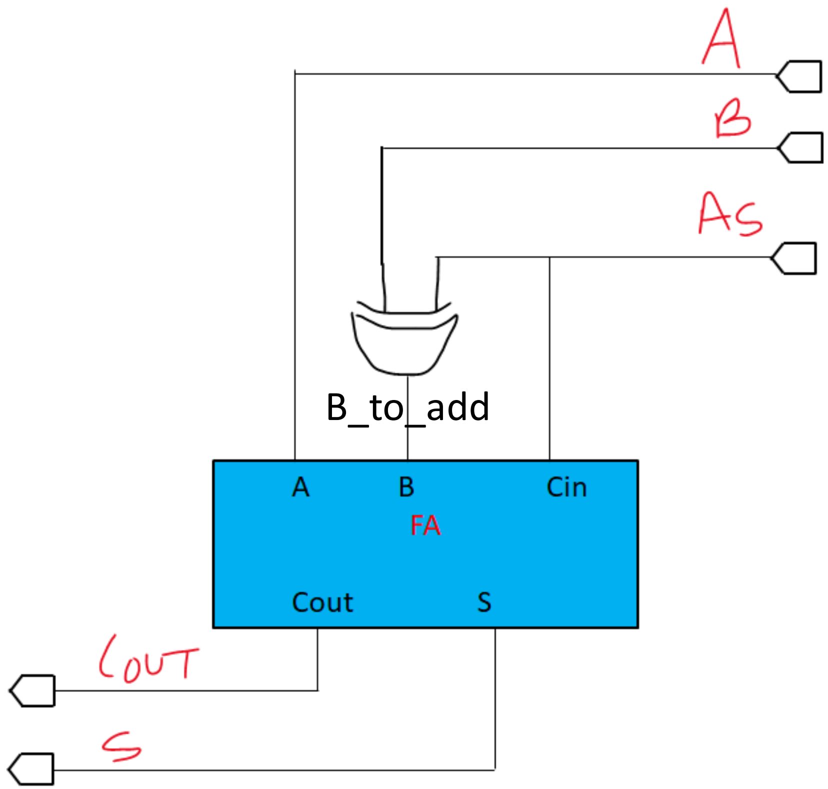
- The common equation for adder-subtractor $\mathrm{S = A + B\_to\_add + AS}$.
- For addition, this equation becomes $\mathrm{A + B + 0}$. For subtraction, it becomes $\mathrm{A} + \bar{B} + 1$.
- From the truthtable, $\mathrm{B\_to\_add} = \overline{\boldsymbol{AS}}\,.\,\mathrm{B} + \mathrm{AS}.\,\bar{\boldsymbol{B}} = \mathrm{AS} \oplus \mathrm{B}$
- This XOR gate ensures that the adder receives B for addition and $\bar{B}$ for subtraction.

	AS	B_to_add	S	S values
Addition	0	B	$\mathrm{A + B\_to\_add + AS}$	A + B + 0
Subtraction	1	$\bar{B}$		A + $\bar{B}$ + 1

Adder-Subtractor

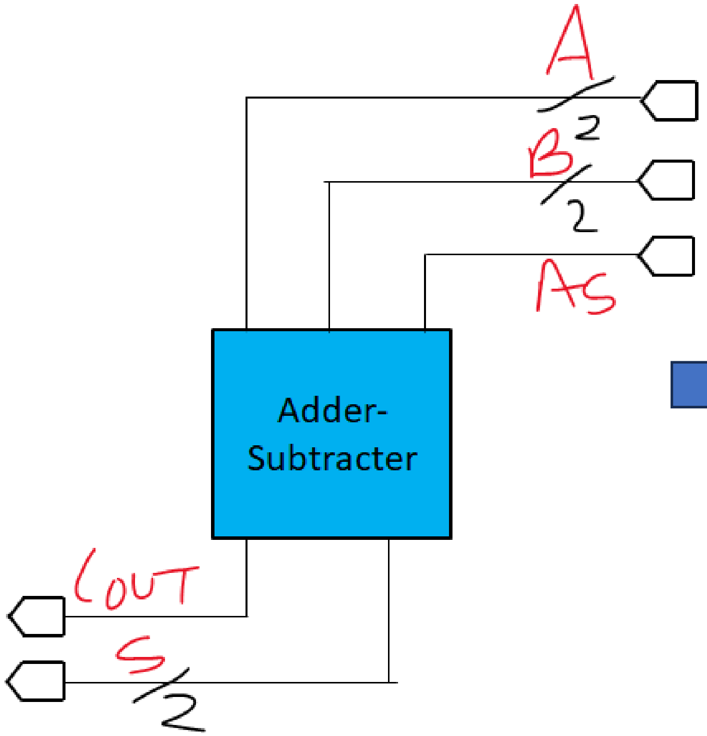
- $S = A + B\_to\_add + AS.$ $B\_to\_add = \overline{AS} \,.\, B + AS.\,\overline{B} = AS \oplus B$
- This XOR gate ensures that the adder receives B for addition and $\bar{B}$ for subtraction.
- A 1-bit adder-subtracter utilizes a full adder and an XOR gate. The control signal AS would be connected to the Cin port of the full adder.

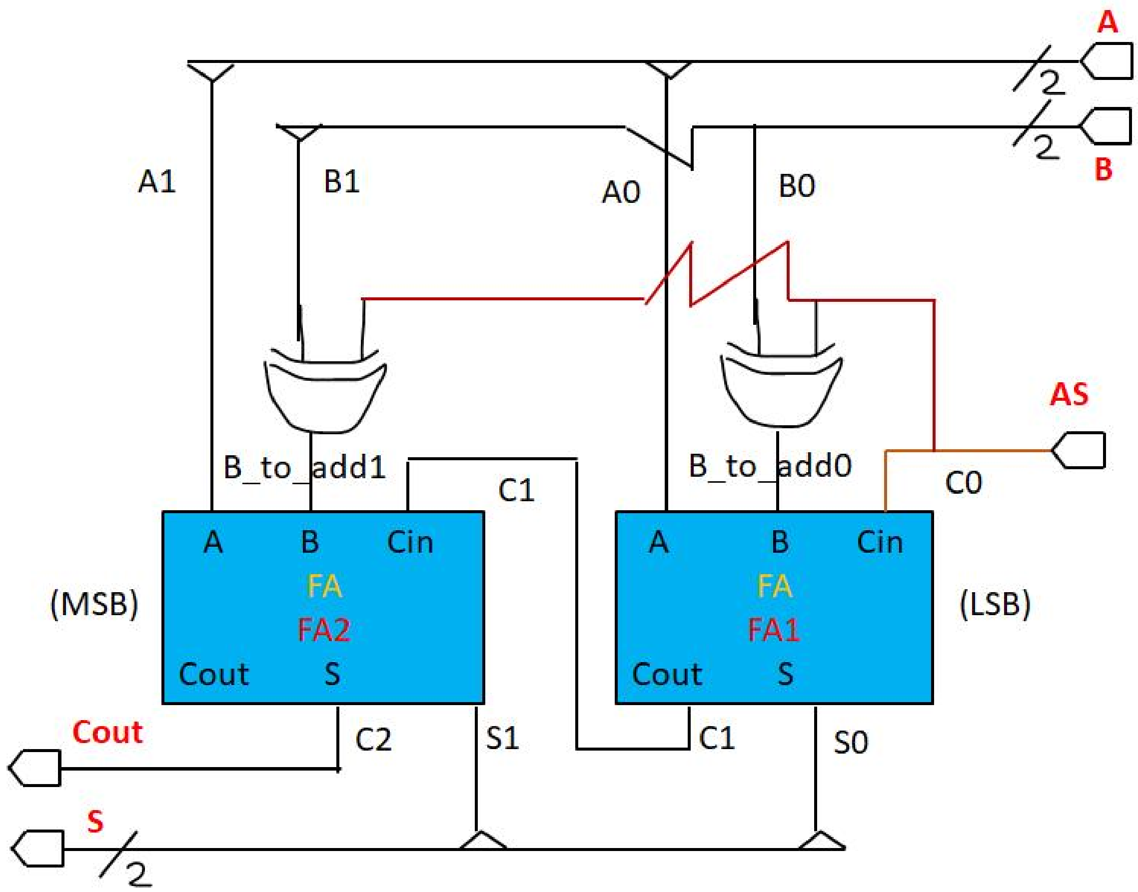




Adder-Subtractor

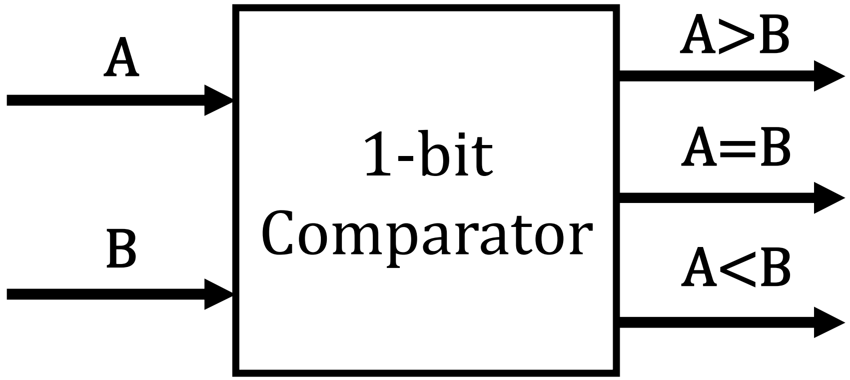
- A 2-bit adder-subtracter would utilize two full adders and two XOR gates, as shown below. The control signal AS would be connected to the Cin port of the full adder for the LSB bit.





Digital Magnitude Comparator

- A comparator is a combinational circuit which can be used to compare between two number.
- A magnitude comparator has three possible outputs; A is greater than B, A is equal to B and A is less than B.
- The truth-table for a 1-bit comparator can be constructed as:



A	B	A>B	A=B	A<B
0	0	0	1	0
0	1	0	0	1
1	0	1	0	0
1	1	0	1	0

The Boolean expression for the outputs:

$$\mathbf{F}_{A=B} = \bar{A}\bar{B} + AB$$

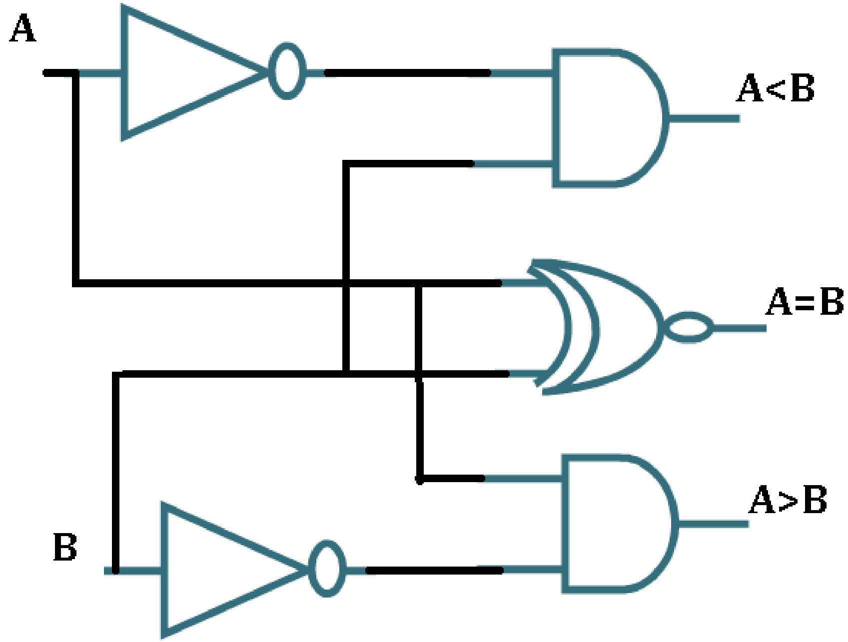
$$\mathbf{F}_{A<B} = \bar{A}B$$

$$\mathbf{F}_{A>B} = A\bar{B}$$

Digital Magnitude Comparator

The Boolean expression for the outputs:

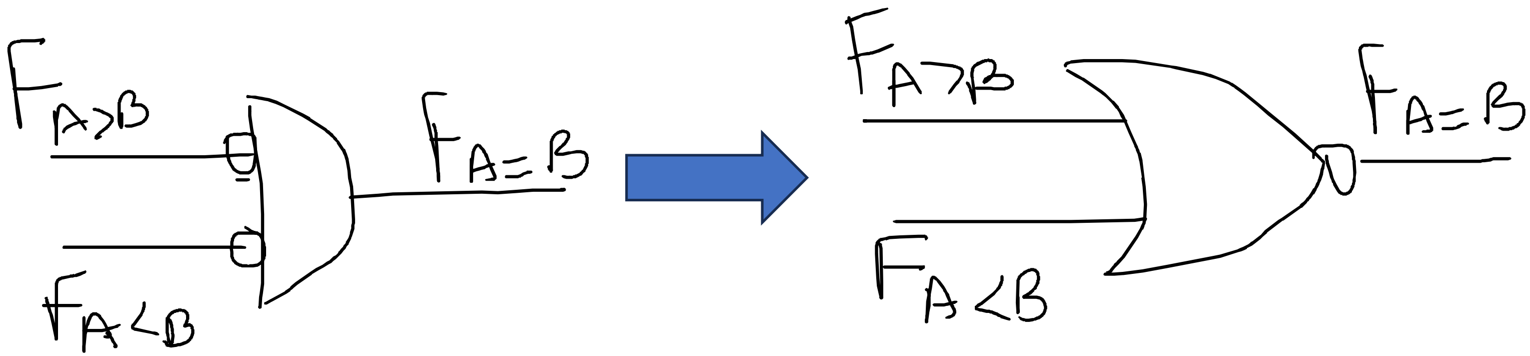
$$F_{A=B} = \bar{A}\bar{B} + AB = \overline{A \oplus B}$$
$$F_{A<B} = \bar{A}B$$
$$F_{A>B} = A\bar{B}$$



Digital Magnitude Comparator

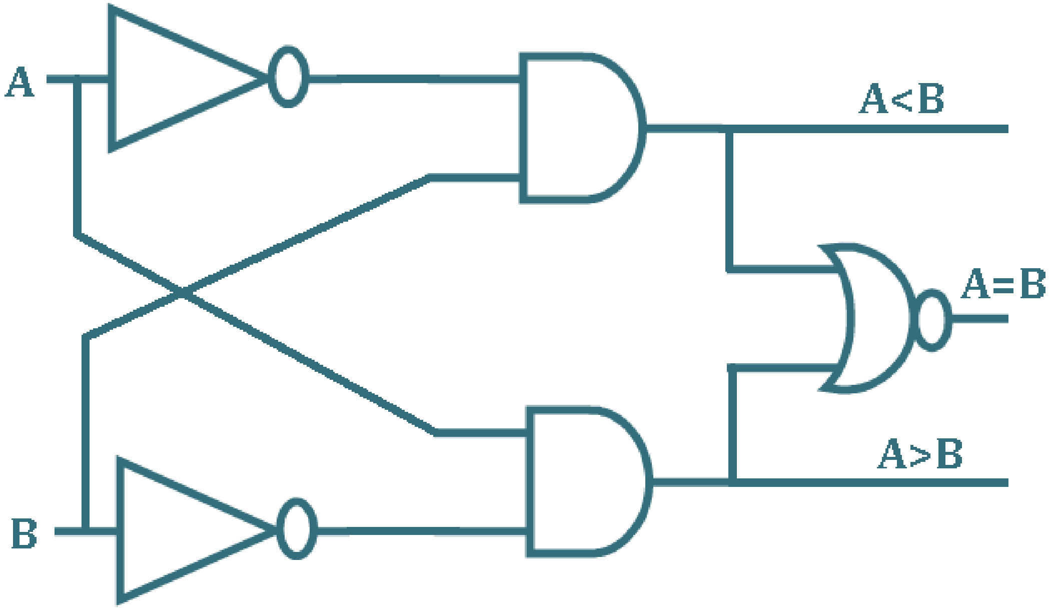
- Note that $F_{A>B}$, $F_{A<B}$ can be implemented by AND gates. The $F_{A=B}$ however requires an XNOR gate, which is more expensive than AND gates.
- One option to reduce number of gates or area would be to implement $F_{A>B}$, $F_{A<B}$ and realize that $F_{A=B}$ will only be true when the other two comparisons are false.

- $\mathbf{F_{A=B} = \overline{F_{A>B}} . \overline{F_{A<B}}}$



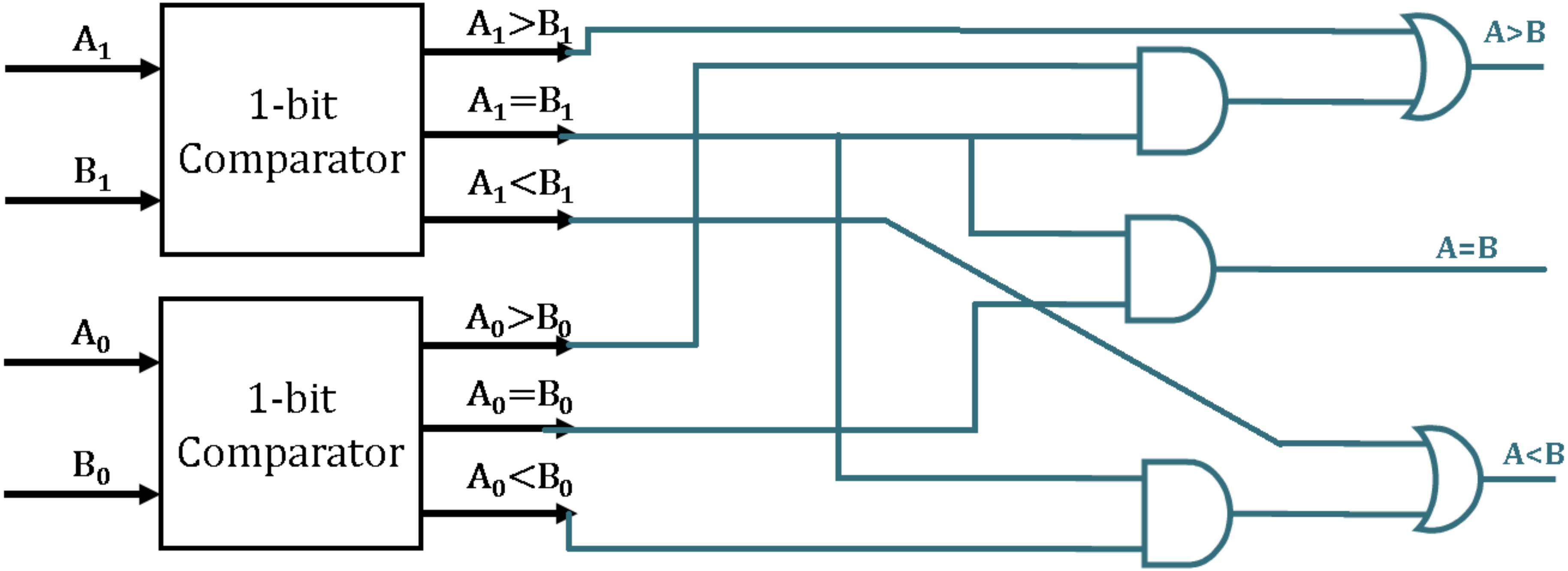
- Using DeMorgan's duality, one can represent the above logic (AND with inverted inputs) with a NOR gate. $\boldsymbol{\overline{X}.\overline{Y} = \overline{X+Y}}$

Digital Magnitude Comparator



Digital Magnitude Comparator

- Two 1-bit comparators can be logically connected to make a 2-bit number comparator.
- The logic behind the connections are:
  - Let $\mathbf{X_0} = (\mathbf{A_0} == \mathbf{B_0})$ and $\mathbf{X_1} = (\mathbf{A_1} == \mathbf{B_1})$
  - For $\mathbf{A} = \mathbf{B}$, the Boolean expression is X_1X_0
  - For $\mathbf{A} > \mathbf{B}$, the Boolean expression is $\mathbf{A_1}\overline{\mathbf{B_1}} + \mathbf{X_1}(\mathbf{A_0}\overline{\mathbf{B_0}})$
  - For $\mathbf{A} < \mathbf{B}$, the Boolean expression is $\overline{\mathbf{A_1}}\mathbf{B_1} + \mathbf{X_1}(\overline{\mathbf{A_0}}\mathbf{B_0})$



References

1. Thomas L. Floyd, “Digital Fundamentals” 11th edition, Prentice Hall – Pearson Education.

Thank You