CPEG 422/622
EMBEDDED SYSTEMS DESIGN

Chengmo Yang

chengmo@udel.edu

Evans 201C

LECTURE 3

2'S COMPLEMENT ADDER/SUBTRACTOR

WHERE ARE WE?

- Last lecture we had:
  - FPGA structure and programming
  - VHDL design structure

- This lecture we will discuss:
  - Adder design
  - 2's complement adder/subtractor

ADDER LOGIC

- Half adder
- Full adder
- Carry ripple adder
- Carry look-ahead adder

BINARY ADDITION LOGIC

Single bit:

$$\begin{array}{r} 0 \\ +\ 0 \\ \hline 0 \end{array} \qquad \begin{array}{r} 0 \\ +\ 1 \\ \hline 1 \end{array} \qquad \begin{array}{r} 1 \\ +\ 0 \\ \hline 1 \end{array} \qquad \begin{array}{r} 1 \\ +\ 1 \\ \hline 1\,0 \end{array}$$

Carry

Truth table:

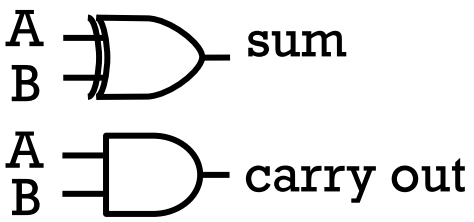
A	B	sum	carry out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

BINARY ADDITION LOGIC

Boolean expression:

sum = A XOR B
carry_out = A AND B

Logic representation:



Truth table:

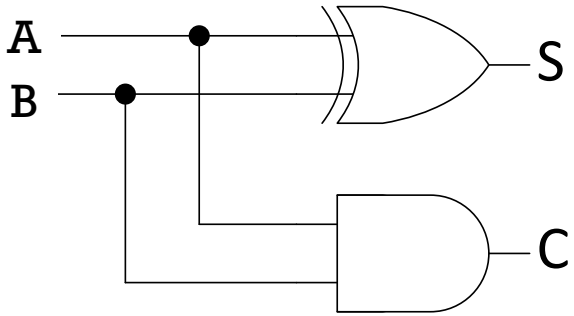
A	B	sum	carry out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

HALF ADDER

Truth table:

A	B	sum	carry out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Implementation:



Called a **half adder**

Problem: Cannot handle a carry in!

FULL ADDER

Q: How to handle carry in?
A: Add one extra input bit.

Truth table:

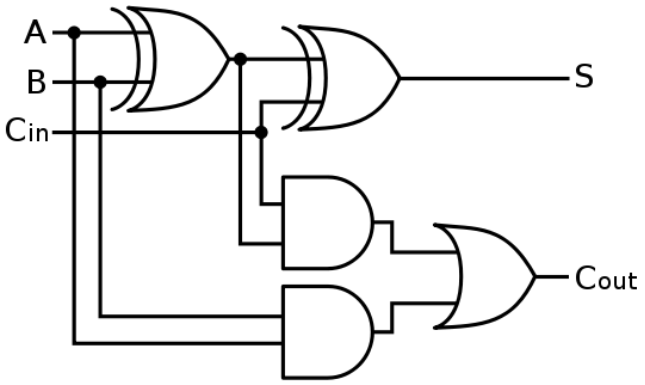
A	B	*Carry in* → Cin	*Carry out* → Cout	*Sum* → S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Boolean expression:

S =(A XOR B) XOR Cin

Cout=(A AND B) OR
((A XOR B) AND Cin)

Implementation:

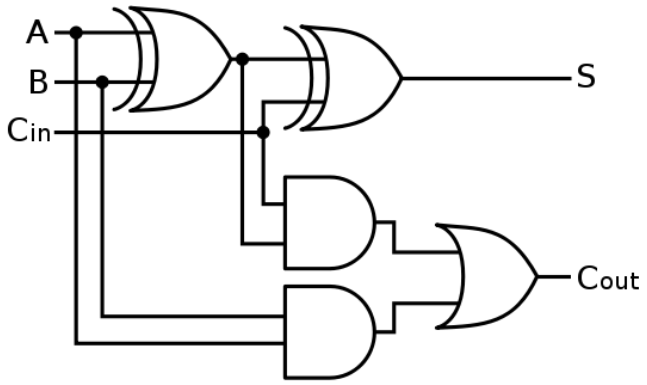


FULL ADDER

VHDL design:

```
library  ieee;
use  ieee.std_logic_1164.all;

entity full_adder is
Port (
   A : in STD_LOGIC;
   B : in STD_LOGIC;
   Cin : in STD_LOGIC;
   S : out STD_LOGIC;
   Cout : out STD_LOGIC
    );
end full_adder;

architecture full_adder of full_adder is
begin

 S <= A XOR B XOR Cin ;
 Cout <= (A AND B) OR (Cin AND (A XOR B)) ;

end full_adder;
```

Implementation:



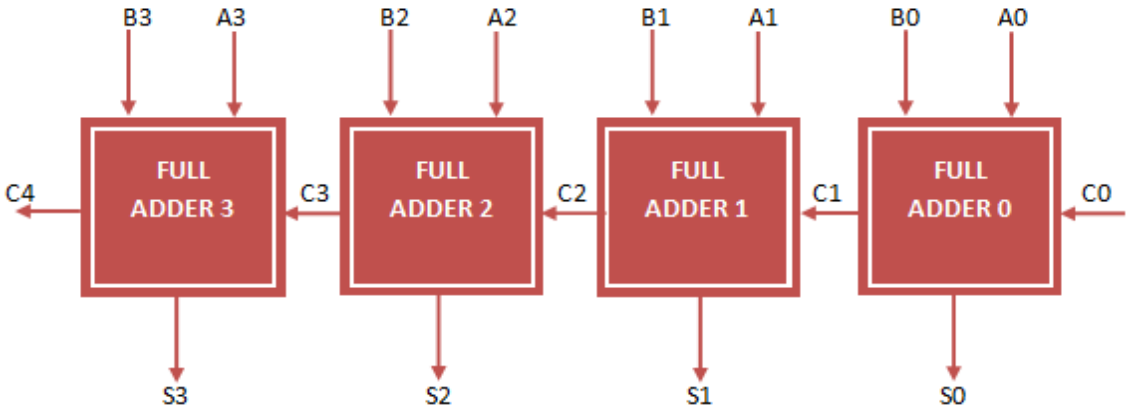
WHAT ABOUT MULTI-BIT ADDER?

CARRY RIPPLE ADDER

Q: How to add two 4-bit numbers?

Example: A = 1011; B = 0011;

	Stage 3	Stage 2	Stage 1	Stage 0	
Carry in	0	1	1	0	C_i
Augend	1	0	1	1	A_i
Addend	0	0	1	1	B_i
Sum	1	1	1	0	S_i
Carry out	0	0	1	1	C_{i+1}



4-bit carry ripple adder structure

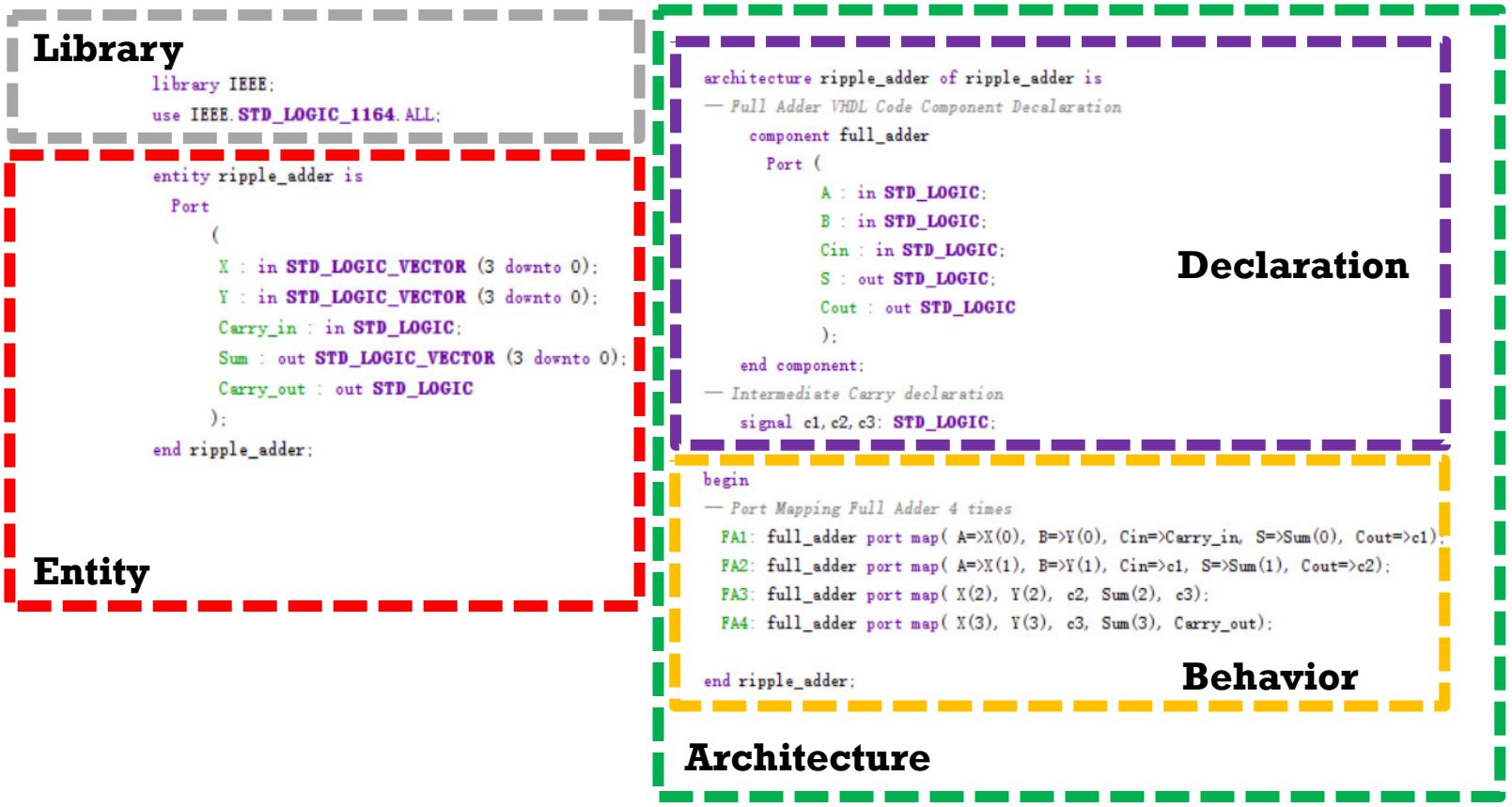
4-BIT CARRY RIPPLE ADDER

VHDL design:

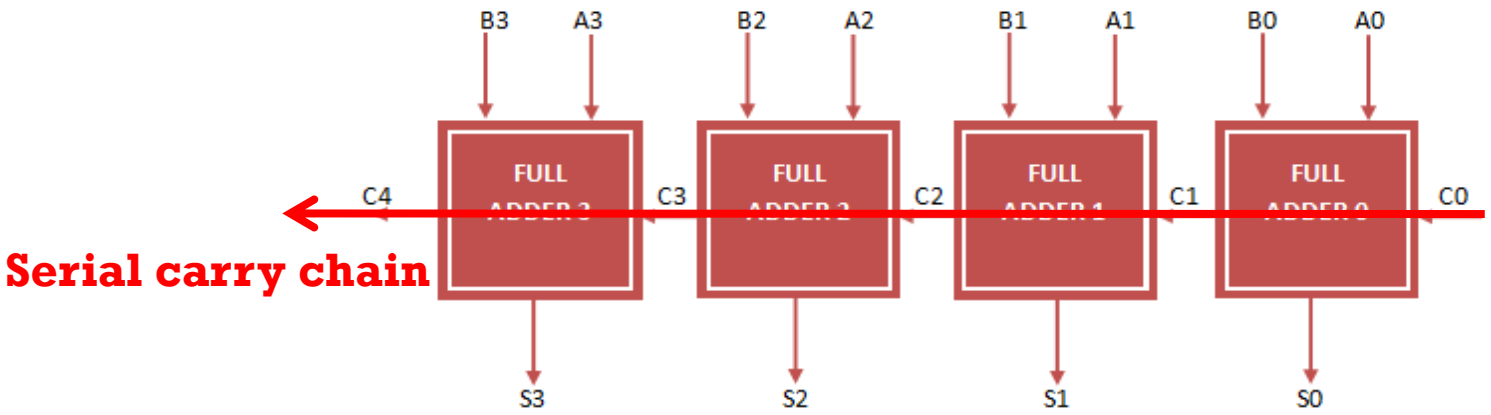
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ripple_adder is
  Port
      (
       X : in STD_LOGIC_VECTOR (3 downto 0);
       Y : in STD_LOGIC_VECTOR (3 downto 0);
       Carry_in : in STD_LOGIC;
       Sum : out STD_LOGIC_VECTOR (3 downto 0);
       Carry_out : out STD_LOGIC
      );
end ripple_adder;
```

```
architecture ripple_adder of ripple_adder is
-- Full Adder VHDL Code Component Decalaration
     component full_adder
       Port (
             A : in STD_LOGIC;
             B : in STD_LOGIC;
             Cin : in STD_LOGIC;
             S : out STD_LOGIC;
             Cout : out STD_LOGIC
             );
    end component;
-- Intermediate Carry declaration
    signal c1,c2,c3: STD_LOGIC;

begin
-- Port Mapping Full Adder 4 times
  FA1: full_adder port map( A=>X(0), B=>Y(0), Cin=>Carry_in, S=>Sum(0), Cout=>c1);
  FA2: full_adder port map( A=>X(1), B=>Y(1), Cin=>c1, S=>Sum(1), Cout=>c2);
  FA3: full_adder port map( X(2), Y(2), c2, Sum(2), c3);
  FA4: full_adder port map( X(3), Y(3), c3, Sum(3), Carry_out);

end ripple_adder;
```

VHDL RECAP


Library
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity ripple_adder is
Port
(
X : in STD_LOGIC_VECTOR (3 downto 0);
Y : in STD_LOGIC_VECTOR (3 downto 0);
Carry_in : in STD_LOGIC;
Sum : out STD_LOGIC_VECTOR (3 downto 0);
Carry_out : out STD_LOGIC
);
end ripple_adder;
Entity
architecture ripple_adder of ripple_adder is
-- Full Adder VHDL Code Component Decalaration
component full_adder
Port (
A : in STD_LOGIC;
B : in STD_LOGIC;
Cin : in STD_LOGIC;
S : out STD_LOGIC;
Cout : out STD_LOGIC
);
end component;
-- Intermediate Carry declaration
signal c1, c2, c3: STD_LOGIC;
Declaration
begin
-- Port Mapping Full Adder 4 times
FA1: full_adder port map(A=>X(0), B=>Y(0), Cin=>Carry_in, S=>Sum(0), Cout=>c1);
FA2: full_adder port map(A=>X(1), B=>Y(1), Cin=>c1, S=>Sum(1), Cout=>c2);
FA3: full_adder port map(X(2), Y(2), c2, Sum(2), c3);
FA4: full_adder port map(X(3), Y(3), c3, Sum(3), Carry_out);
end ripple_adder;
Behavior
Architecture

CARRY RIPPLE ADDER



4-bit carry ripple adder structure

Problem: Slow, it takes time to propagate a carry along the chain.

CARRY LOOK AHEAD ADDER

Let's take a closer look at full adder truth table

Truth table:

A	B	Cin	Cout	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

- If A = B = 0, then Cout = 0, no need to wait for Cin
- If A $\oplus$ B = 1, then Cout = Cin
- If A = B = 1, then Cout = 1, no need to wait for Cin

CARRY LOOK AHEAD ADDER

Carry look ahead: compute carries without waiting for previous stages.

Suppose i^{th} full adder:

Let's define

$G_i \equiv A_i B_i$ Carry generate

$P_i \equiv A_i \oplus B_i$ Carry propagate

These only depend on the inputs (Ai, Bi) of ith full adder

Full adder review

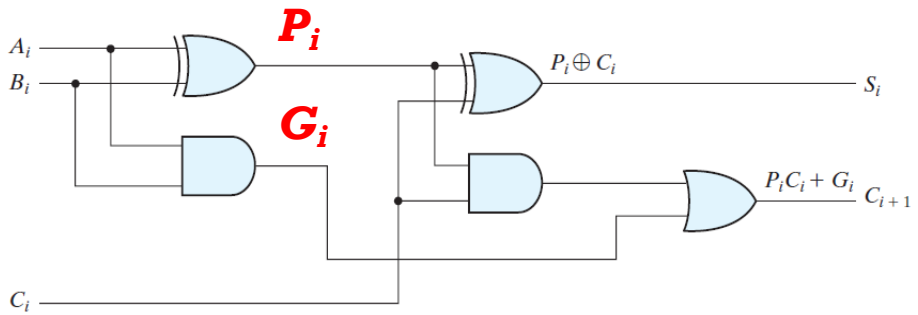
Therefore:

$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + P_i C_i$$

If $G_i = 1$, then generate C_{i+1}

If $P_i = 1$, then C_i propagates to C_{i+1}



CARRY LOOK AHEAD ADDER

For example, the carry inputs for a 4-bit adder:

C_0 = carry in

$C_1 = G_0 + P_0C_0$

$C_2 = G_1 + P_1C_1 = G_1 + P_1(G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0$

$C_3 = G_2 + P_2C_2 = G_2 + P_2(G_1 + P_1G_0 + P_1P_0C_0)$

$= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$

Only based on input variables

Pi = Ai ⊕ Bi, Gi = AiBi, C_0

NO NEED to wait for previous stages!

Full adder review

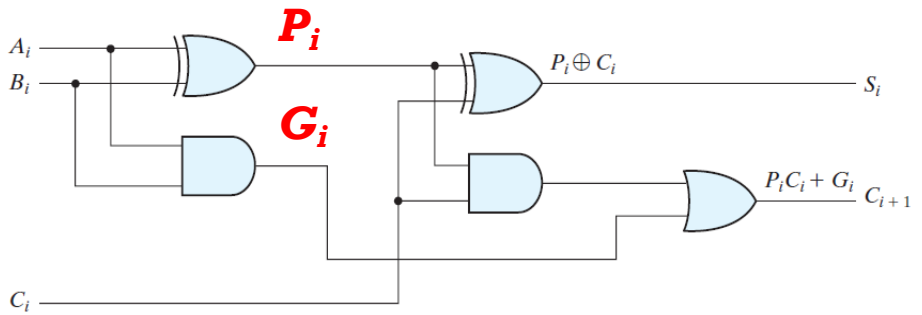
Therefore:

$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + P_i C_i$$

If $G_i = 1$, then generate C_{i+1}

If $P_i = 1$, then C_i propagates to C_{i+1}



CARRY LOOK AHEAD ADDER

For example, the carry inputs for a 4-bit adder:

C_0 = carry in

$C_1 = G_0 + P_0C_0$

$C_2 = G_1 + P_1C_1 = G_1 + P_1(G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0$

$C_3 = G_2 + P_2C_2 = G_2 + P_2(G_1 + P_1G_0 + P_1P_0C_0)$

$= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$

Only based on input variables

$Pi = Ai \oplus Bi$, $Gi = AiBi$, C_0

NO NEED to wait for previous stages!

Carry look ahead (CLA) Boolean expression

- This is called **carry look ahead (CLA) logic** → **faster** operation!
- We must sacrifice something → more complex hardware

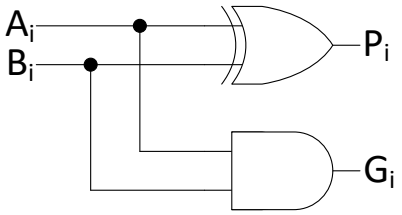
CARRY LOOK AHEAD ADDER

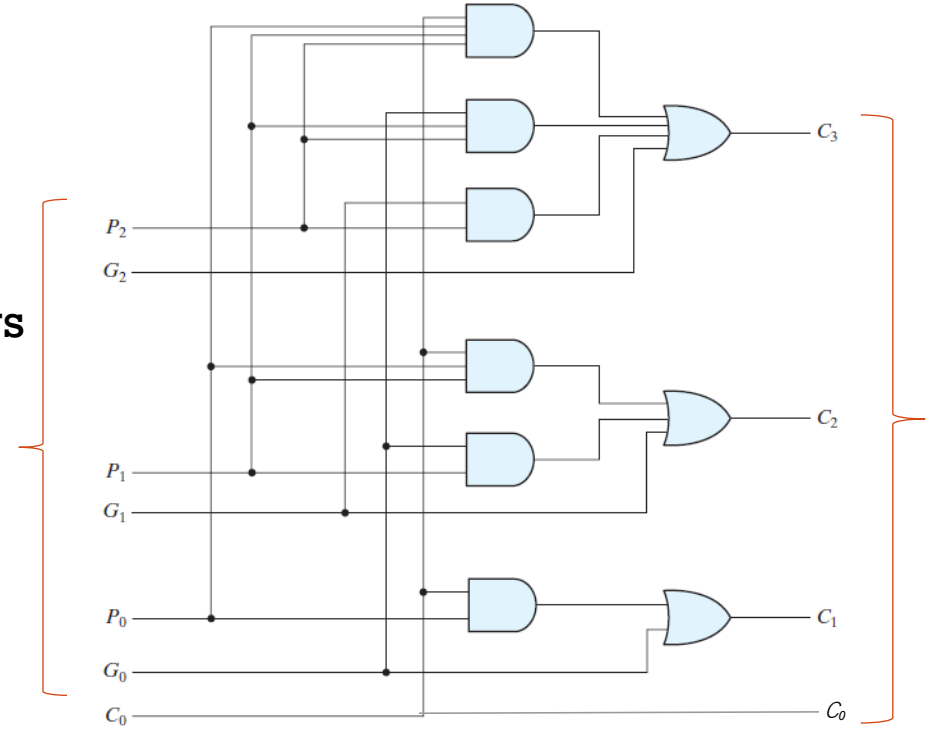
Carry look ahead implementation

$C_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$
$C_2 = G_1 + P_1G_0 + P_1P_0C_0$
$C_1 = G_0 + P_0C_0$
$C_0 = C_0$

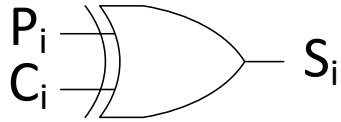
Q: What about C_4 (Cout)?

These come from the first half adders





These enter XOR gates to compute S_i



SUBTRACTION LOGIC

- 2's complement number representation
- 2's complement subtractor

TWO'S COMPLEMENT

- The **2's complement** of an N-bit number is defined as its complement with respect to $\mathbf{2^N}$.
- For example, number $(010)_2$, it's two's complement is $(110)_2$, because $(010)_2 + (110)_2 = (1000)_2 = \mathbf{2^3}=8$.

- Convert a binary number into its 2's complement in 2 steps:

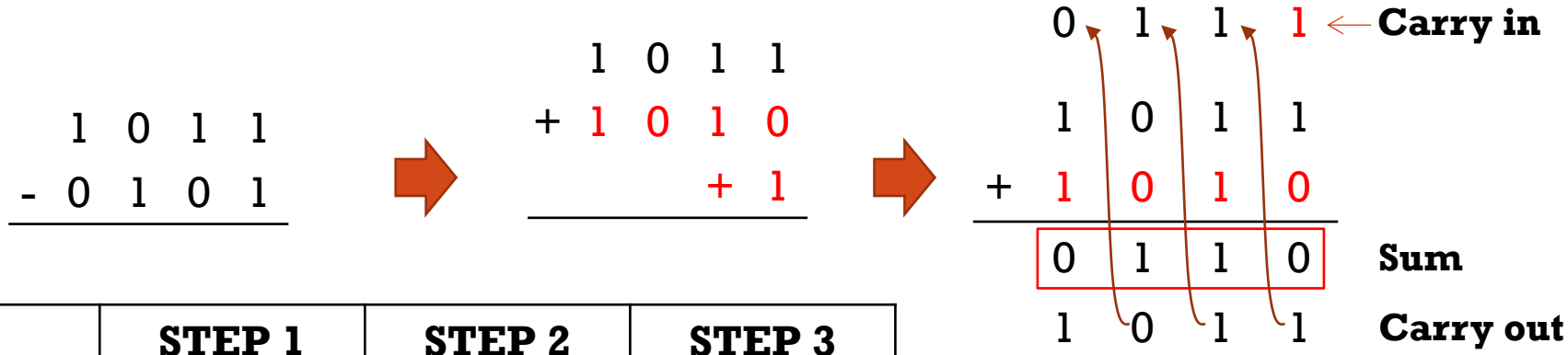
Step 1: inverting bits 010→101

Step 2: adding one 101+1→110

BINARY SUBTRACTION

$A-B=A+[B]_{comp}$ $[B]_{comp}$ is the two's complement of B

Example: $A=(1011)_2=11, B=(0101)_2=5$



	STEP 1	STEP 2	STEP 3
SUB	**Invert bits**	**Input carry = 1**	**Usual addition**
ADD	Do not invert bits	Input carry = 0	Usual addition

Answer $=(0110)_2=6$

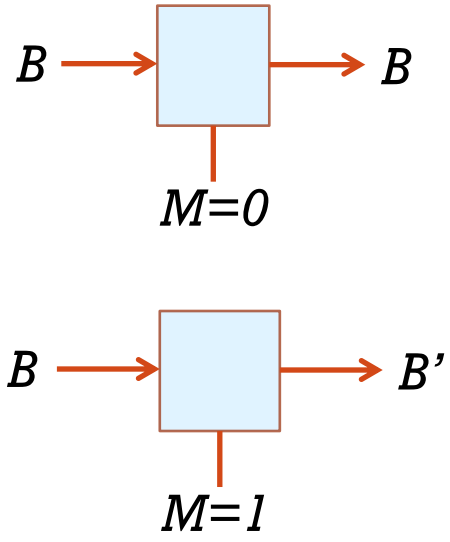
SUBTRACTOR

Subtractor can be implemented by adder as well, additionally:

- **Invert logic** for second operand B.
- Set carry in as 1.

Q: Is there a circuit that can invert or not invert, based on a "control signal"?

A: YES! **XOR gate**



	STEP 1	STEP 2	STEP 3
SUB	**Invert bits**	**Input carry = 1**	**Usual addition**
ADD	Do not invert bits	Input carry = 0	Usual addition

SUBTRACTOR

Subtractor can be implemented by adder as well, additionally:

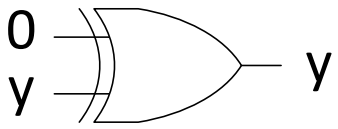
- **Invert logic** for second operand B.
- Set carry in as 1.

Q: Is there a circuit that can invert or not invert, based on a "control signal"?

A: YES! **XOR gate**

x	y	x⊕y
0	0	0
0	1	1
1	0	1
1	1	0

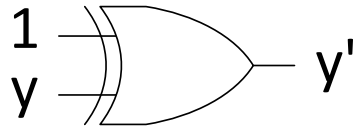
If x=0, then output=y



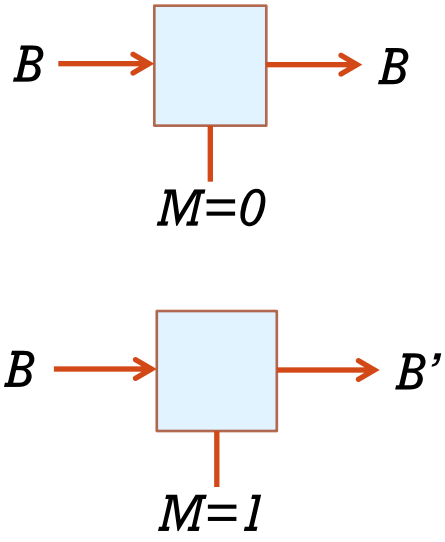
Buffer

x	y	x⊕y
0	0	0
0	1	1
1	0	1
1	1	0

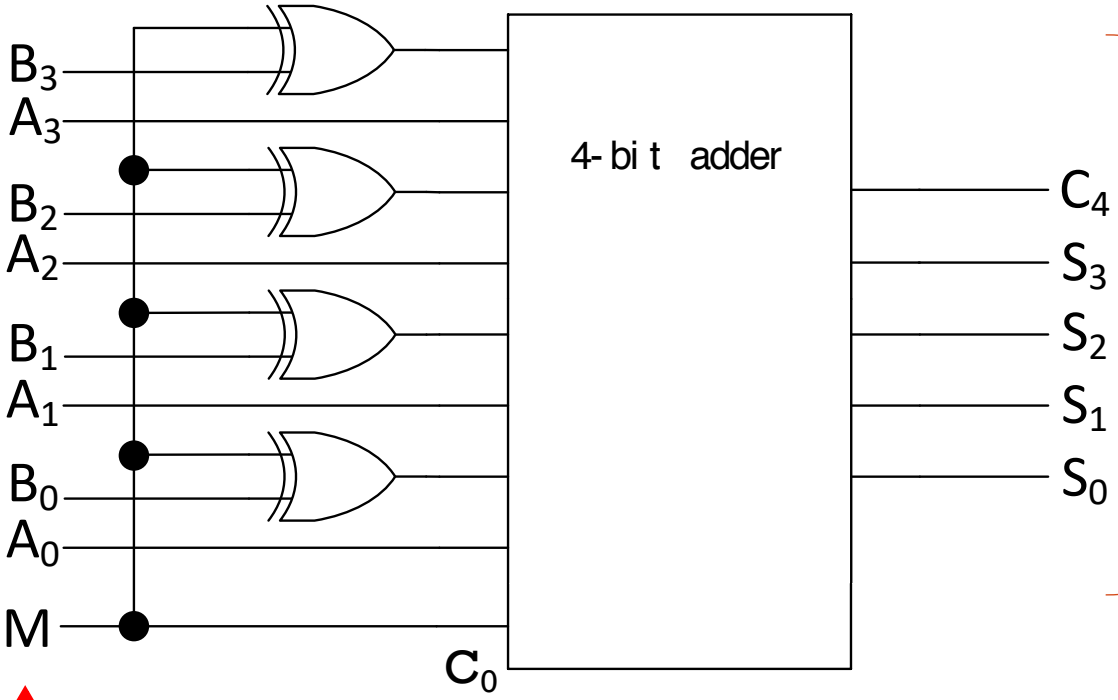
If x=1, then output=y'



Inverter



4-BIT ADDER/SUBTRACTOR



- If M=0, the circuit performs A+B
  - Do not invert $B_3B_2B_1B_0$
  - Input carry=0
- If M=1, the circuit performs A-B
  - Invert $B_3B_2B_1B_0$
  - Input carry=1

"Minus"

1) To invert or not to invert?
2) Input carry C_0= M

VHDL TIPS – VECTOR ASSIGNMENT

```
signal q: std_logic_vector (3 downto 0);
```

We can do:

1. q <= "1011";
2. q <= ('1','0','1','1'); -- single bit quotations
3. q <= (3=>'1', 2=>'0', 1=>'1', 0=>'1'); -- bit association
4. q <= (2=>'0', others => '1'); -- others assignment

We frequently use **others** for initialization or setting bits.

```
x <= "00000000";
```

is same as

```
x <= (others => '0');
```

VHDL TIPS – VECTOR OPERATION

Q: How to XOR "B", a std_logic_vector of N bits with "M", a std_logic signal?

A: Extend M to N bits first, then use bitwise XOR on vectors.

```
Given:
        B : in STD_LOGIC_VECTOR (31 downto 0);
        M: in STD_LOGIC;

Declare:
         signal m_ex, b_in: std_logic_vector (31 downto 0);

Assign:
        m_ex <= (others=>M);
        b_in <= B XOR m_ex; -- this performs bit-by-bit XOR
```

- As long as two vectors are of the same in length, bitwise operators work on them.

HOMEWORK

- Change the 4-bit carry ripple adder VHDL code to implement
  - A 4-bit adder/subtractor
  - A 4-bit carry look ahead (CLA) adder
- Submit your source code on Canvas by Monday.

PROJECT 1: 24-BIT ADDER/SUBTRACTOR

- Input: two 24-bit binary number A and B,

  single bit signal M:M=1(subtract), M=0(add)

- Output: sum (A+B) or difference (A-B)

  Overflow (0 or 1)

- Implement the adder/subtractor in two ways:

1. Carry-ripple
2. Carry-look ahead

- Compare the timing and complexity of these two implementations

NEXT WEEK

- Testbench
- More on overflow
- More on 2-level CLA