King Fahd University of Petroleum & Minerals
Computer Engineering Dept

COE 202 – Fundamentals of Computer Engineering

Term 081

Dr. Ashraf S. Hasan Mahmoud

Rm 22-148-3

Ext. 1724

Email: ashraf@kfupm.edu.sa

Background – Binary Addition – Adding Bits

- Adding Binary bits:
  - 0 + 0 ➔ 0 and the carry is 0
  - 0 + 1 ➔ 1 and the carry is 0
  - 1 + 0 ➔ 1 and the carry is 0
  - 1 + 1 ➔ 0 and the carry is 1
- Hence one can write the following truth table:

  $A_i + B_i$ ➔ S_i and the carry is C_{i+1}
- Note that S_i and C_{i+1} are two functions, each depends on A_i and B_i

A_i	B_i	S_i	C_{i+1}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Background – Binary Addition – Adding Bits (2)

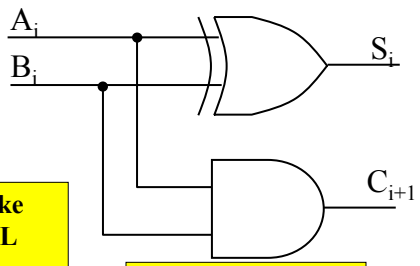
- The functions S_i and C_{i+1} are given by

$$S_i = \overline{A_i} B_i + A_i \overline{B_i} = A_i \oplus B_i$$

- and

$$C_{i+1} = A_i B_i$$

- Logic circuit is shown



Half Adder Circuit

This known as HALF Adder – It does not take into account incoming carry signal (see FULL Adder description – next)

Background – Binary Addition

- Adding n-bit binary numbers:
  - Example: Add the following two numbers 101001 and 1101

```
  0 0  1  0  0  1  0    ← Carry generated
    1  0  1  0  0  1    → Number A
+   0  0  1  1  0  1    → Number B
--------------------
  0 1  1  1  1  1  0    → Summation
```

- In general we have

```
  Cn Cn-1 Cn-2 … C2 C1 C0    ← Carry generated
     An-1 An-2 … A2 A1 A0    → Number A
+    Bn-1 Bn-2 … B2 B1 B0    → Number B
--------------------
  Cn Sn-1 Sn-2 … S2 S1 S0
```

Note first carry in signal (C_0) is always ZERO

- **The binary number (C_n S_{n-1} S_{n-2} … S_2 S_1 S_0) is the summation result**

Full Adder Circuit

- But in cases like the previous example, we need to add two bits in addition to the carry signal coming adding the previous two bits
- Hence one can write the following truth table:

  $A_i + B_i + C_i \rightarrow S_i$ and the carry is C_{i+1}

A_i	B_i	C_i	S_i	C_{i+1}
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 Circuit (2)

- **The logic functions for S_i and the carry is C_{i+1} are**

A_i \ $B_i C_i$	00	01	11	10
0		1		1
1	1		1	

A_i \ $B_i C_i$	00	01	11	10
0			1	
1		1	1	1

$$S_i = \overline{A_i}\,\overline{B_i}C_i + \overline{A_i}B_i\overline{C_i} + A_i\overline{B_i}\,\overline{C_i} + A_iB_iC_i$$

$$S_i = A_i \oplus B_i \oplus C_i$$

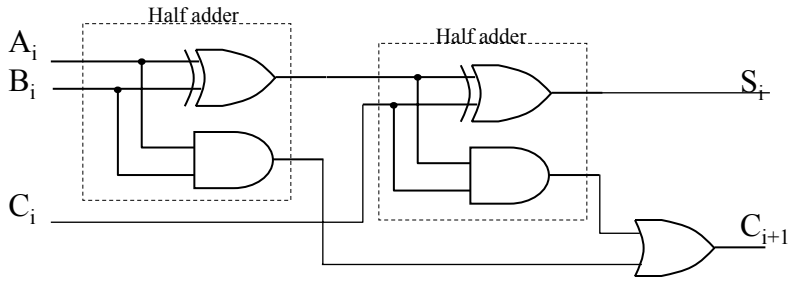
$$C_{i+1} = A_iB_i + A_iC_i + B_iC_i$$

$$C_{i+1} = A_iB_i + C_i(A_i + B_i)$$

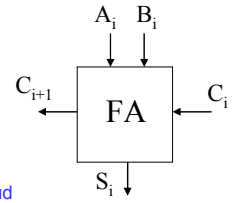
$$C_{i+1} = A_iB_i + C_i(A_i \oplus B_i)$$

Full Adder Circuit (4)

- **The logic circuits for S_i and the carry is C_{i+1} are**

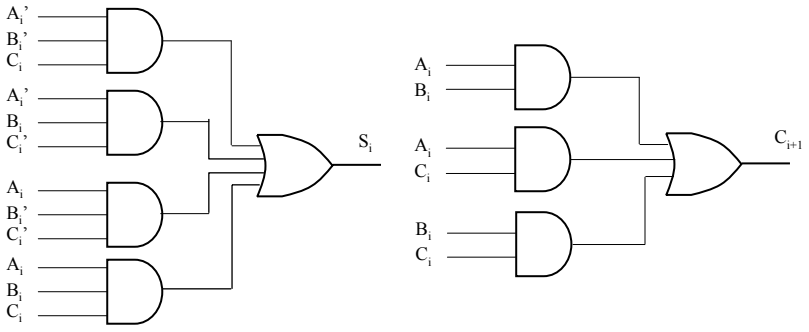


Another symbol for the full adder block



Full Adder Circuit (5)

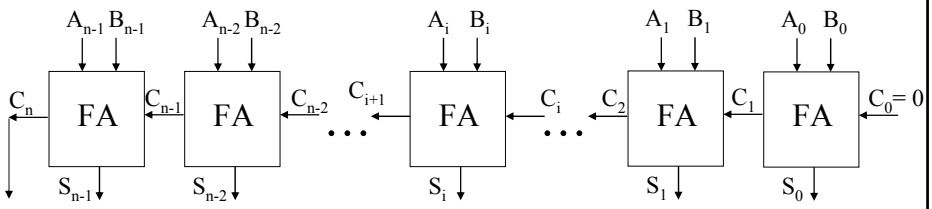
- **Using the standard form, the circuit is**



τ is the logic gate delay (including the inverter)
S_i output is available after 3τ delay
C_{i+1} output is available after 2τ delay

Ripple Carry Adder

- Using the FA block one can construct an n-bit binary adder as in



- The number $(C_n S_{n-1} S_{n-2} \ldots S_2 S_1 S_0)_2$ is equal to the summation of $(A_{n-1} A_{n-2} \ldots A_2 A_1 A_0)_2$ and $(B_{n-1} B_{n-2} \ldots B_2 B_1 B_0)_2$
- Note that C_0 is set to zero to get the right result
- If C_0 is set to 1, Then the result is equal to A + B + 1

Ripple Carry Adder Delay

- Time to get the summation:
  - Assume: If τ is the gate delay, then for a FA block, the S_i output is available after 3τ while the C_{i+1} output is available after 2τ – refer to FA structure
  - Apply the inputs at t = 0
  - The C_1 signal is generated at t = 2τ
  - The C_2 signal is generated at t = $2X2\tau$
  - The C_3 signal is generated at t = $3X2\tau$
  - . . .
  - The C_{n-1} signal is generated at t = $(n-1)X2\tau$
  - The S_n signal is generated at t = $(n-1)X2\tau + 3\tau$
  - The C_n signal is generated at t = $nX2\tau$
- Hence, total delay is $2n\tau$

Ripple Carry Adder Delay (2)

- The disadvantage:
  - The outputs (C and S) of one stage carry and summation can not be generated till the outputs of the previous stage are generated (Ripple effect)
- Delay is linearly proportional to n (size of binary number) – this is undesired
  - This means longer delays for longer word sizes

Carry Lookahead Adder

- n is the size of the binary number – or the word size for the ALU
- Ripple carry adder – results in delay that increases linearly with size of binary number, n
- To design fast CPUs you need fast logic circuits
- It is desirable to get the summation with a fixed delay that does not depend on n
- The carry lookahead adder provides just that

Carry Lookahead Adder Design

- The reason for the long delay is the time to propagate the carry signal till it reaches the final FA stage
- Let's examine the FA logic again (refer to FA section)
- The carry signal at the i^{th} stage is given by

$$C_{i+1} = A_i B_i + C_i (A_i \oplus B_i)$$

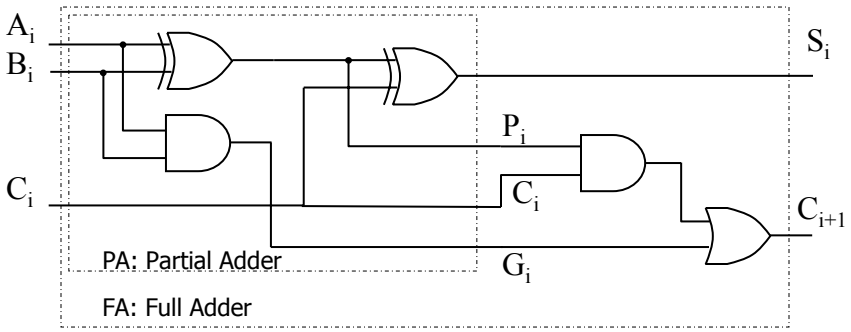
which could be written as $C_{i+1} = G_i + P_i C_i$

if we define $G_i = A_i B_i$ and $P_i = A_i \oplus B_i$

- G_i and P_i are referred to as the generate and propagate signals, respectively

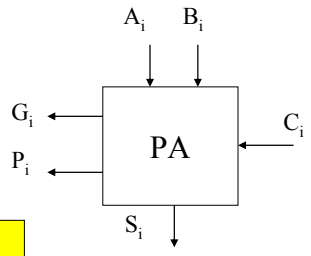
Carry Lookahead Adder Design (2)

- The new design for the FA block is as follows:



Carry Lookahead Adder Design (3)

- **A partial Adder block**



If we use the standard form,
τ is the logic gate delay (including the inverter)
S_i output is available after 3τ delay
G_i output is available after τ delay
P_i output is available after τ delay

Carry Lookahead Adder Delay

- **C_0 (the carry signal for first stage) is set to zero**
- **C_1 is equal to $G_0 + P_0C_0$**
  - **It takes 2τ to generate this signal**
- **C_2 is equal to $G_1 + P_1C_1 = G_1 + P_1(G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0$**
  - **It takes 2τ to generate this signal**
- **C_3 is equal to $G_2 + P_2C_2 = G_2 + P_2(G_1 + P_1G_0 + P_1P_0C_0) = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$**
  - **It takes 2τ to generate this signal**

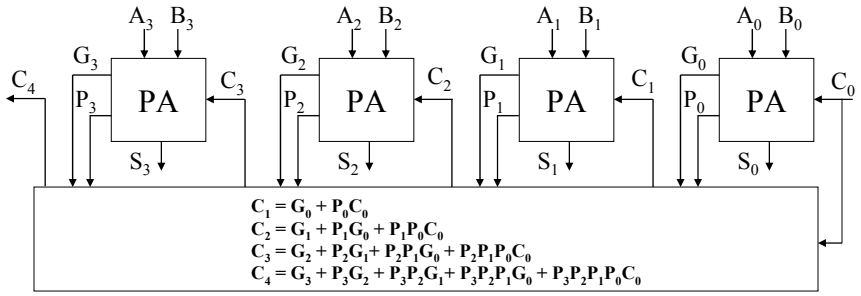
Carry Lookahead Adder Delay (2)

- C_4 is equal to $G_3 + P_3C_3 = G_3 + P_3(G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0) = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$
  - It takes 2τ to generate this signal
- In general, C_{i+1} is given by

$$C_{i+1} = G_i + P_iG_{i-1} + P_iP_{i-1}G_{i-2} + \ldots + P_iP_{i-1}\ldots P_1G_0 + P_iP_{i-1}\ldots P_1P_0C_0$$

Carry Lookahead Adder

- Block Diagram for 4-bit CLA



Carry Lookahead Adder Delay (3)

- Any carry signal depends only on C_0 and the generate (G) and propagate (P) functions only – It does not depend on the previous carry signal (except C_0 which is readily available)
- The generate (G) and propagate (P) signals can be generated simultaneously with one gate delay τ – for all stages
- Hence all carry signals at all stages can be available after 3τ delay

Carry Lookahead Adder Delay (4)

- Total Delay:
  - Assume all inputs (A, B, and C_0) were available at t = 0
  - All G and P functions will be available at t = τ
  - All carry signals (C_1 … $C_{n-1}C_n$) will be available at t = $\tau+2\tau = 3\tau$
  - The S_{n-1} signal will be available at t = $3\tau + 3\tau = 6\tau$
- Note delay to get summation is FIXED and does NOT depend on word size n – desirable feature

Carry Lookahead Adder - Refined

- **One Last issue to solve:**

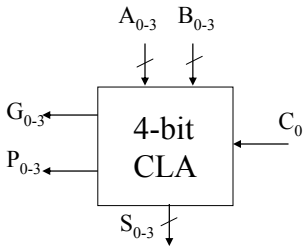
  C4 signal requires gates with 5 inputs

  C_5, C_6, etc will require gates with > 5 inputs – This is undesirable (higher delay)
- **Note the structure of function for $C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$**
  - **Let $G_{0-3} = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$ ➔group generate function**
  - **Let $P_{0-3} = P_3P_2P_1P_0$ ➔ group propagate function**
  - **Then C_4 can be written as**

  $C_4 = G_{0-3} + P_{0-3}C_0$
- **Hence the function for C_4 is very similar to that for C_1 – but it uses group generate/propagate functions as opposed to generate/propagate functions**

Carry Lookahead Adder - Refined (2)

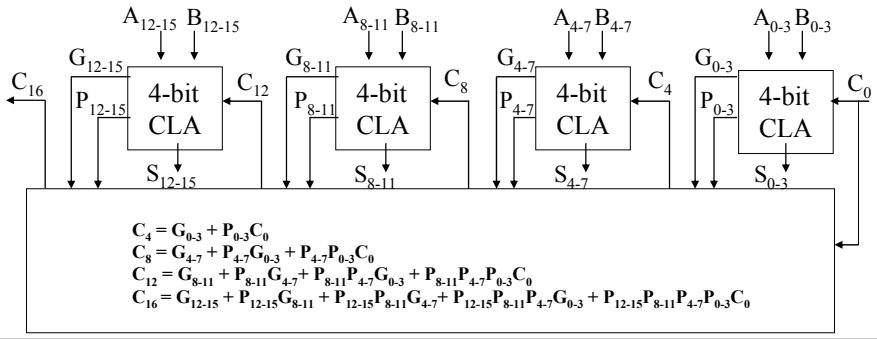
- **4-bit CLA block**



Accepts two 4-bit numbers A and B with initial carry signal C_0
Generates 4-bit summation in addition to group generate/functions
To do 4-bit additions – one needs to add logic to generate C_4 signal using G_{0-3}, P_{0-3}, and C_0

Carry Lookahead Adder - General

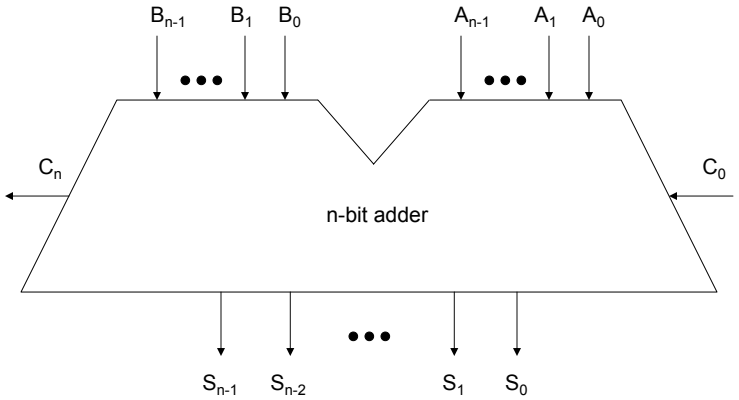
- **Block Diagram for 16-bit CLA**



- C_{16} (and all other carry signals) are available two gate delays after the time needed to generate the group generate/propagate signals.
- Group propagate signal requires one gate delay – while group generate requires two gate delays
- Hence, C_{16} is available 5 gate delays after A, B and C_0 are applied as inputs (assuming standard forms)

n-Bit Adder General

- **Diagram used in most text books**
  - **Could be ripple carry adder or carry lookahead adder**



Binary Numbers - Review

- Computers use fixed n-bit words to represent binary numbers
- ☞ It is the user (programmer) who makes the distinction whether the number is signed or unsigned
- Example:

```
main(){
 unsigned int X, Y;
 int   W, Z;
 ...
 }
```

- X and Y are defined as unsigned integers while W and Z are defined as signed integers

Addition of Unsigned Numbers - Review

- For n-bit words, the UNSIGNED binary numbers range from $(0_{n-1}0_{n-2}...0_10_0)_2$ to $(1_{n-1}1_{n-2}...1_11_0)_2$ i.e. they range from 0 to 2^{n-1}
- When adding A to B as in:

$$\begin{array}{llll} C_n & C_{n-1} & C_{n-2} \ \cdots \ C_2 & C_1 \ C_0 \quad \leftarrow \text{Carry generated} \\ & A_{n-1} & A_{n-2} \ \cdots \ A_2 & A_1 \ A_0 \quad \rightarrow \text{Number A} \\ + & B_{n-1} & B_{n-2} \ \cdots \ B_2 & B_1 \ B_0 \quad \rightarrow \text{Number B} \\ \hline C_n & S_{n-1} & S_{n-2} \ \cdots \ S_2 & S_1 \ S_0 \end{array}$$

- If C_n is equal to ZERO, then the result DOES fit into n-bit word $(S_{n-1}\ S_{n-2}\ \ldots\ S_2\ S_1\ S_0)$
- If C_n is equal to ONE, then the result DOES NOT fit into n-bit word

Subtraction of Unsigned Numbers - Review

- How to perform A – B (both defined as unsigned)?
- Procedure:
  1. Add the the 2's complement of B to A; this forms A + (2^n – B)
  2. If (A >= B), the sum produces end carry signal (C_n); discard this carry
  3. If A < B, the sum does not produce end carry signal (C_n); result is equal to 2^n – (B-A), the 2's complement of B-A – Perform correction:
     - Take 2's complement of sum
     - Place –ve sign in front of result
     - Final result is –(A-B)

Subtraction of Unsigned Numbers – Review (2)

- Example: X = 1010100 or $(84)_{10}$, Y = 1000011 or $(67)_{10}$ – Find X-Y and Y-X
- Solution:

A) X – Y: X = 1010100
2's complement of Y = 0111101
Sum = 10010001
Discard C_n (last bit) = 0010001 or $(17)_{10}$ ← X – Y

B) Y – X: X = 1000011
2's complement of X = 0101100
Sum = 1101111
C_n (last bit) is zero → need to perform correction
Y - X = -(2's complement of 1101111) = - 001001

2's Complement Review

- For n-bit words, the 2's complement SIGNED binary numbers range from $-(2^{n-1})$ to $+(2^{n-1}-1)$

  e.g. for 4-bit words, range = -8 to +7
- Note that MSB is always 1 for –ve numbers, and 0 for +ve numbers

2's Complement Review (2)

- Consider the following Example:

  How to represent –9 using 8-bit word?

A) Using signed magnitude:

$(+9)_{10} = (00001001)_2 \rightarrow (-9)_{10} = (10001001)_2$

The most significant bit is 1 (-ve number)

B) Using 1's complement:

$M = 2^n-1$, -9 in 1s complement = $M - 9 = (11111111)_2 - (00001001)_2 = (11110110)_2$

C) Using 2's complement:

$M = 2^n$, -9 in 2s complement = $M - 9 = (100000000)_2 - (00001001)_2 = (11110111)_2$

- Or simply:

1's complement: invert bits of number

2's complement: invert bits of number and add one to it

Subtraction of Signed Numbers

- Consider

```
  +6  0000 0110      -6   1111 1010
+ 13  0000 1101    +13   0000 0011
------ ------------  -----  ------------
 +19  0001 0011     +7   0000 0111

  +6  0000 0110      -6   1111 1010
-  13  1111 0011    - 13   1111 0011
------ ------------  -----  ------------
 - 7  1111 1001     -19   11101101
```

- Any carry out of sign bit position is DISCARDED
- -ve results are automatically in 2's complement form (no need for an explicit –ve sign)!

Subtraction of Signed Numbers (2)

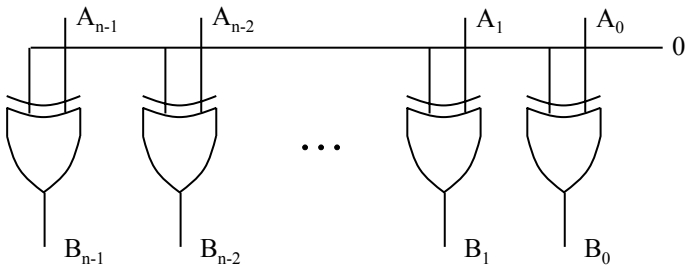
- Subtraction of two signed binary number when negative numbers are in 2's complement is simple: How to do A – B?

Take the 2's complement of the subtrahend B (including the sign bit) and add it to the minuend A (including the sign bit). A carry out of the sign bit position is discarded

Minuend ➔ A
Subtrahend ➔ - B
---------- -----
Result ➔ D

Subtractor - Background

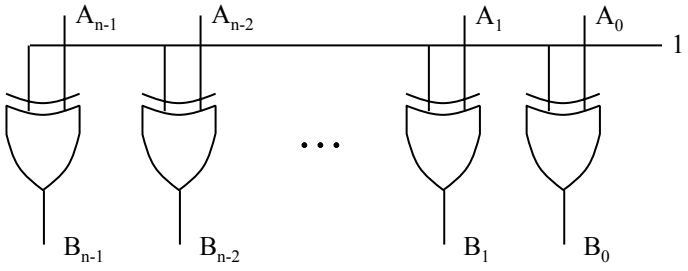
- **What is the number B equal to?**



B is equal to A

Subtractor – Background (2)

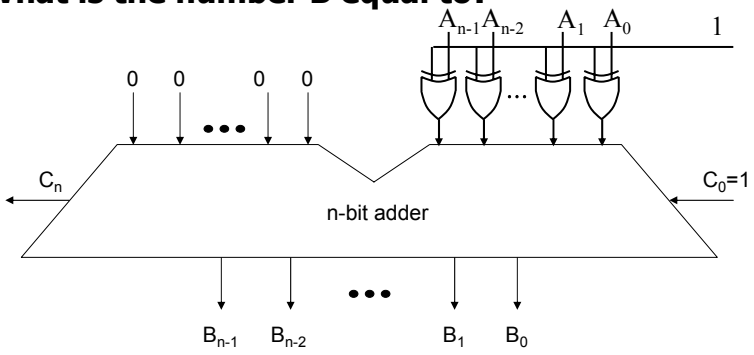
- **What is the number B equal to?**



B is equal to 1's complement of A ($B_i = A_i'$)

Subtractor – Background (3)

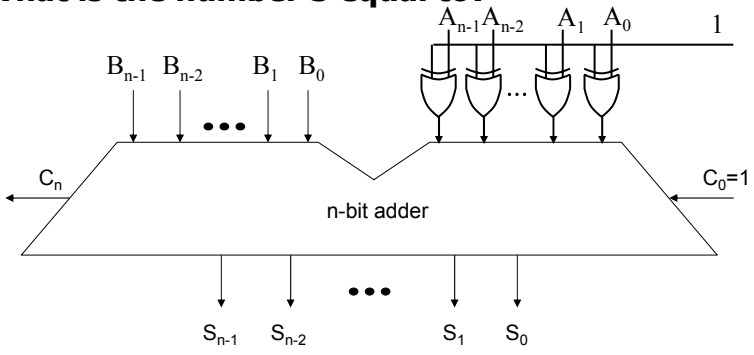
- **What is the number B equal to?**



B is equal to 2's complement of A
(B = -ve A)

Subtractor

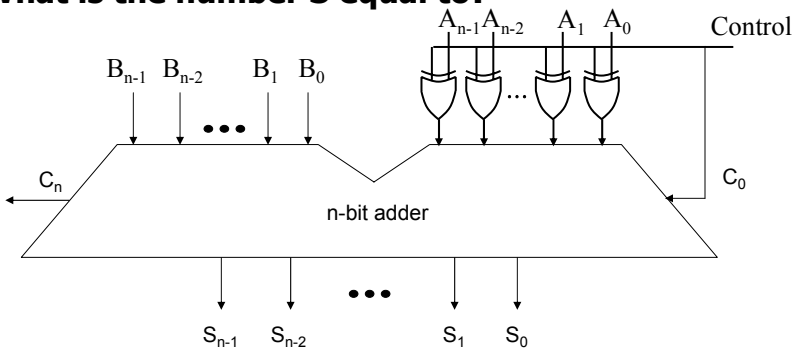
- **What is the number S equal to?**



S is equal to B +(-A)
Or S = B - A

Adder-Subtractor

- **What is the number S equal to?**



If (Control = 0) S = A+B
Else (Control = 1) S = B - A

Overflow Conditions

- **Computers use fixed word sizes to represent numbers**
- **Overflow flag: result addition or subtraction does NOT fit the fixed word size**
- **Examples: consider 8-bit words and using signed numbers**

carries:	0 1000 0000	carries	1 0110 0000
+70	0100 0110	-70	1011 1010
+80	0101 0000	-80	1011 0000
-----	----------	----	----------
+150	1001 0110	-150	0110 1010

- **Note both operation produced the wrong answer –because +150 or –150 are OUTSIDE the range of allowed number (only from –128 to +127)!**

Note that when C_{n-1} and C_n are different the results is outside the allowed range of numbers

Overflow Conditions (2)

- **When n-bit word is used to represent UNSINGED binary numbers:**
  - **Carry signal (C_n) resulting from adding the last two bits (A_{n-1} and B_{n-1}) detects an overflow**

```
If (Cn == 0) then {
   // no carry and no overflow, but correction step is
  required for   //subtraction
   correction_step: final result = -1 X 2's complement of
  result;
}
else {
   // overflow for addition, but no correction step is
  //required for subtraction
   process_overflow;
}
```

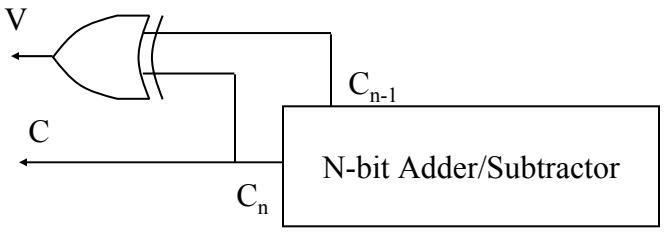

Overflow Conditions (3)

- **When n-bit word is used to represent SINGED binary numbers:**
  - **Carry signal into n-1 position (C_{n-1}) and the one resulting from adding the last two bits (A_{n-1} and B_{n-1}) determine an overflow ➔ Let overflow bit V = C_{n-1} XOR C_n**

```
If (V == 0) then {
   // no overflow, and addition/subtraction result is correct
   ;
}
else {
   // overflow has occurred for addition/subtraction, result
  // requires n+1 bits
   process_overflow;
}
```

Overflow Conditions - Summary

	Unsigned	Signed
Overflow Condition	C_n= 1 (no correction required)	V = C_n XOR C_{n-1}= 1

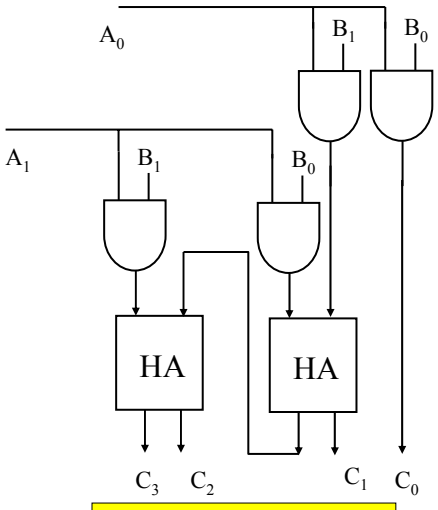


Overflow Detection logic for Addition and Subtraction

2-Bit Binary Multiplier

- **Consider the multiplication of B_1B_0 by A_1A_0**

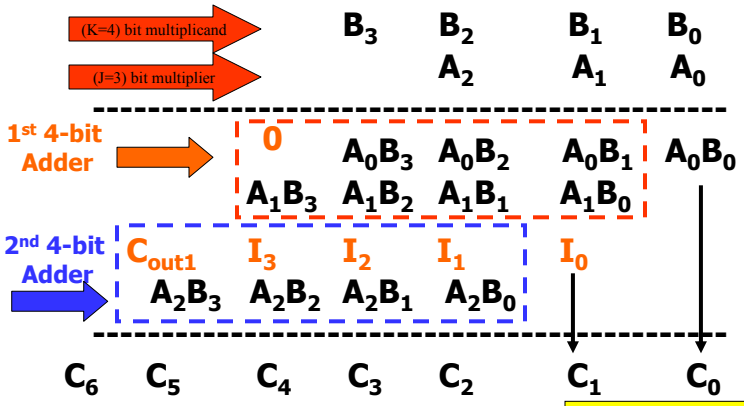
$$\begin{array}{cccc} & & B_1 & B_0 \\ & & A_1 & A_0 \\ \hline & & A_0B_1 & A_0B_0 \\ & A_1B_1 & A_1B_0 & \\ \hline C_3 & C_2 & C_1 & C_0 \end{array}$$



2-bit binary multiplier

A Bigger Binary Multiplier

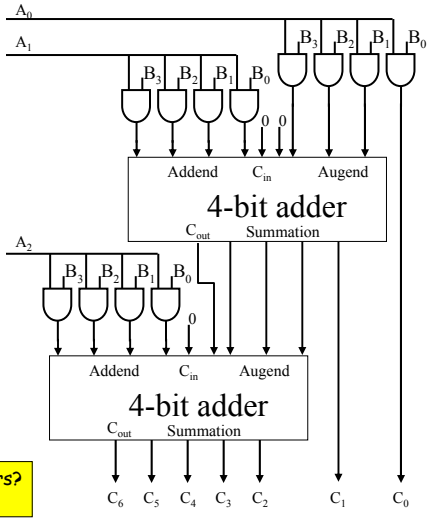
- Consider the multiplication of B = $B_3B_2B_1B_0$ by A = $A_2A_1A_0$



4X3-bit binary multiplier

4-Bit by 3-Bit Binary Multiplier

- For J multiplier bit and K multiplicand bit:
  - JXK AND gates
  - (J-1) K-bit adders to produce a product of J+K bits
- In the shown circuit:
  - J = 3 (multiplier = $A_2A_1A_0$)
  - K = 4 (multiplicand = $B_3B_2B_1B_0$)
  - Hence we need 3X4 = 12 AND gates and (3-1) Adders
  - Multiplication result in 3+4 bits

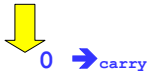


Exercise: Does this circuit work for signed numbers? Try to multiply 2 signed numbers

Decimal Arithmetic – Adding 2 BCD digits

- **Valid BCD digits:0, 1, 2, …, 9**
- **Example:**

Design a circuit that adds two BCD digits



1 1 0	BCD carry	1	1	0	→ carry in
4 4 8		0100	0100	1000	→ 1st digit
+ 4 8 9		0100	1000	1001	→ 2st digit
9 3 7	Binary sum	1001	1101	1 0001	→ add 6 if > 9
	Add 6		0110	0110	
	BCD sum	1	0011	1 0111	→ carry out
	BCD result	1001	0011	0111	→ BCD sum digit

When the BCD Sum is Greater Than 9?

1. **When the sum of two digits generates a carry (see previous example)**

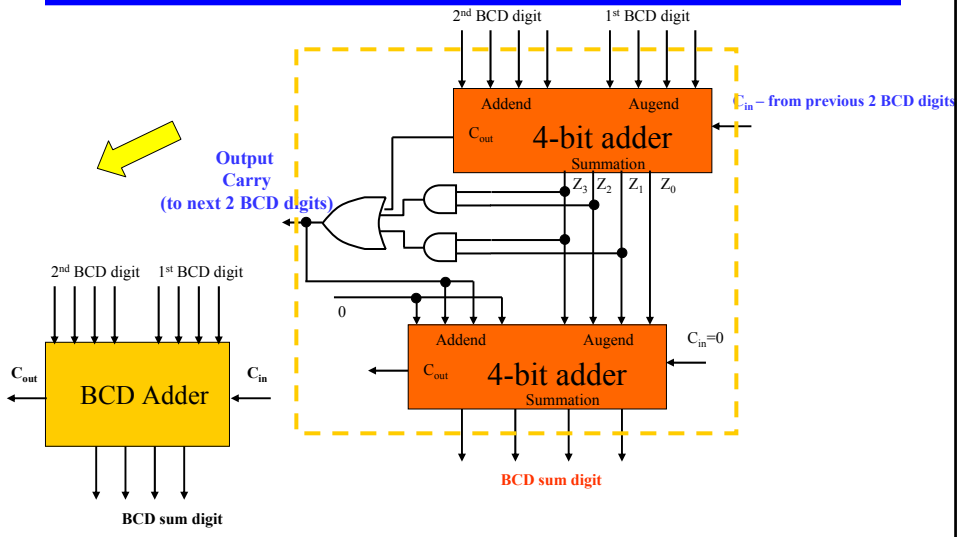
OR

2. **Sum of the two digits is 1010, 1011, 1100, 1101, 1110, 1111 (See problem 3-11 page 170)**
   - **If the sum is denoted by $Z_3Z_2Z_1Z_0$ then $F = Z_1Z_3 + Z_2Z_3$ is equal to 1 only if the number $Z_3Z_2Z_1Z_0$ is an invalid BCD digit**

- **Hence, to detect an invalid summation result where a correction (adding 6 is required) we need:**

$$F = \text{carry} + Z_1Z_3 + Z_2Z_3$$

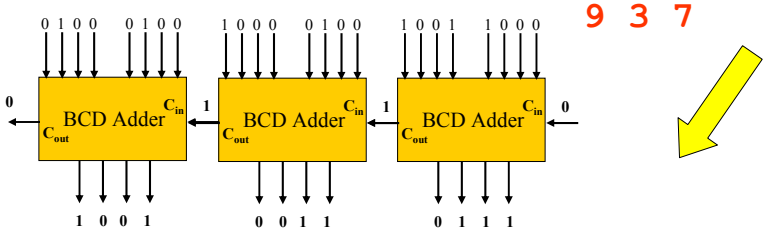
Circuit/Block Diagram of BCD Adder



Decimal Arithmetic – Adding 2 BCD Numbers?

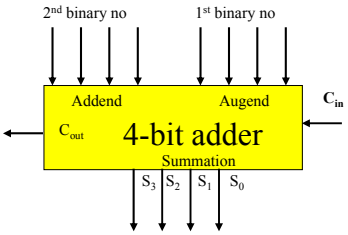
- **Consider the previous example:**

```
  4  4  8
+ 4  8  9
---------
  9  3  7
```



Example

- Design a circuit to sum three 4-bit binary numbers.
- Hint: Use two blocks of 4-bit adders plus any needed logic



Example: cont'd

- **Solution:**
- **Inputs:**
  - **First 4-bit number** **X = X3X2X1X0**
  - **Second 4-bit number** **Y = Y3Y2Y1Y0**
  - **Third 4-bit number** **Z = Z3Z2Z1Z0**
- **Output:**

```
      X3 X2 X1 X0
      Y3 Y2 Y1 Y0
      Z3 Z2 Z1 Z0
     -------------
F5 F4 F3 F2 F1 F0
```

→ **What is the size of out output? Why?**

- **Procedure: Add X to Y first – get the result and then add it to Z**
- **Step 1: Addition of X and Y**
  - **A 4-bit adder is required. This addition will result in a sum and a possible carry, as follows:**

```
    X3 X2 X1 X0
    Y3 Y2 Y1 Y0
   ----------------
C4 S3 S2 S1 S0
```

  - **Note that the input carry Cin = 0 in this 4-bit adder**

Example: cont'd

- **Solution:**
- **Step 2: Addition of S and Z**
  - **This resulting partial sum (i.e. S3S2S1S0) will be added to the third 4-bit number Z3Z2Z1Z0 by using another 4-bit adder as follows, resulting in a final sum and a possible carry:**

```
  S3 S2 S1 S0
  Z3 Z2 Z1 Z0
-----------------
D4 F3 F2 F1 F0
```

where F3F2F1F0 represents the final sum of the three inputs X, Y, and Z. Again, in this step, the input carry to this second adder will also be zero

- **Notice that in Step 1, a carry C4 was generated in bit position 4, while in Step 2, another carry D4 was generated also in bit position 4. These two carries must be added together to generate the final Sum bits of positions 4 and 5 (F4 and F5).**
- **Adding C4 and D4 requires a half adder. Thus, the output from this circuit will be six bits, namely F5 F4 F3F2F1F0**

Example: cont'd

- **Solution:**
- **Function:**

```
      X3 X2 X1 X0
      Y3 Y2 Y1 Y0 +
      Z3 Z2 Z1 Z0 +
     -------------
F5 F4 F3 F2 F1 F0
```