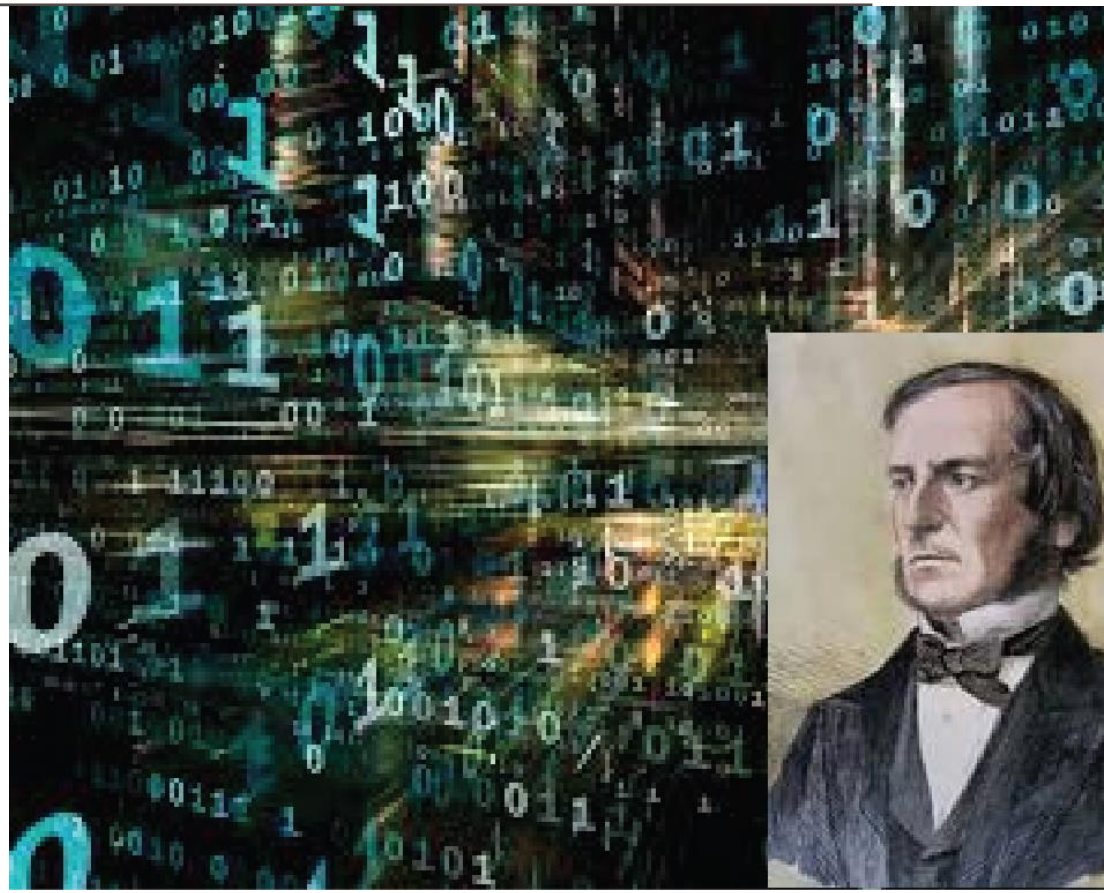
DIGITAL CIRCUITS

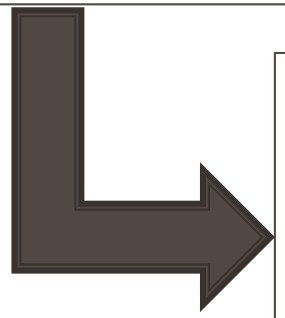
Week-9, Lecture-2
Combinational Circuits

Sneh Saurabh
5th October, 2018



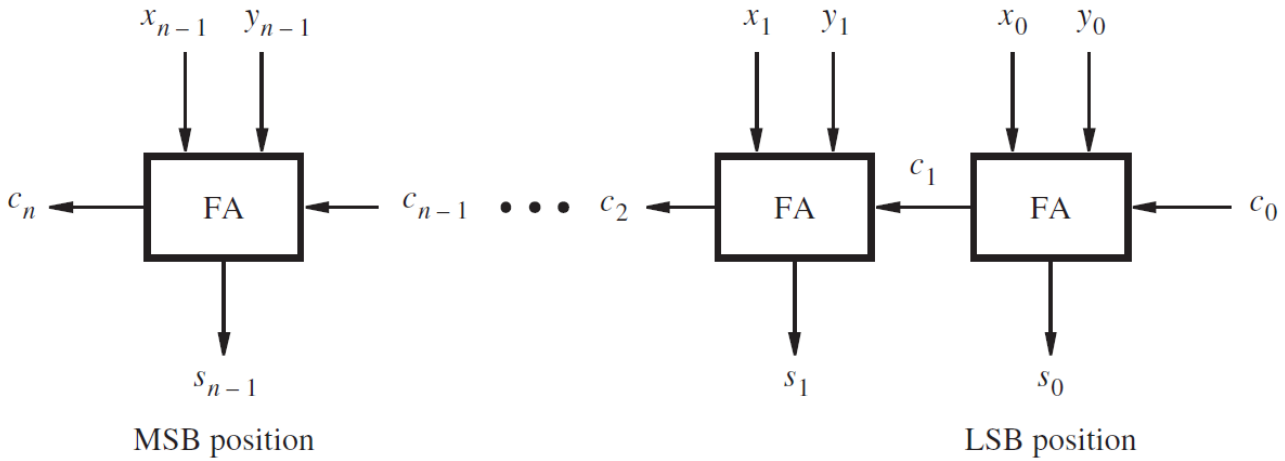
Digital Circuits: Announcements/Revision

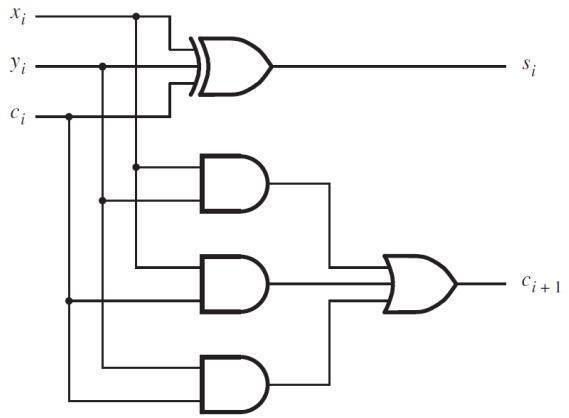
Adder/Subtractor



Improving Speed

Adder Subtractor Unit: Performance Issues





- For a full-adder the delay for the carry-out signal is equal to two gate delays
- The longest delay is along the carry computation path
  - Computation of c_n requires that c_{n-1} is computed, c_{n-1} waits for c_{n-2} and so on

- For n-stage ripple carry adder, there is $(2n + 1)$ gate delays
- When $n = 32$ or $n = 64$, the delay becomes unacceptably high
- **Fast adder circuit is required: computation of carry must be speeded**

Generate and Propagate Functions

- $c_{i+1} = x_i y_i + x_i c_i + y_i c_i$
- $c_{i+1} = x_i y_i + (x_i + y_i) c_i$
- $c_{i+1} = g_i + p_i c_i$

where $\boldsymbol{g_i = x_i y_i}$ and $\boldsymbol{p_i = (x_i + y_i)}$

Generate function:

- The function g_i is equal to 1 when both inputs x_i and y_i are equal to 1 (regardless of the value of the incoming carry to this stage c_i)
- Since in this case stage i is guaranteed to generate a carry-out, g is called the ***generate function***

Propagate function:

- The function p_i is equal to 1 when at least one of the inputs x_i and y_i is equal to 1
- In this case a carry-out is produced if $c_i = 1$
- The effect is that the carry-in of 1 is propagated through stage i
- Hence p_i is called the ***propagate function***

- Both *generate* and *propagate* functions $\boldsymbol{g_i}$ and $\boldsymbol{p_i}$ can be computed by a single level of gate AND/OR
- For computing $\boldsymbol{g_i}$ and $\boldsymbol{p_i}$ information from other stages are not required

Carry lookahead adder

- $c_{i+1} = g_i + p_i c_i$ where $\boldsymbol{g_i = x_i y_i}$ and $\boldsymbol{p_i = (x_i + y_i)}$

Expanding $c_i = g_{i-1} + p_{i-1} c_{i-1}$

$c_{i+1} = g_i + p_i(g_{i-1} + p_{i-1} c_{i-1})$

Expanding c_{i-1}, c_{i-2} till c_0

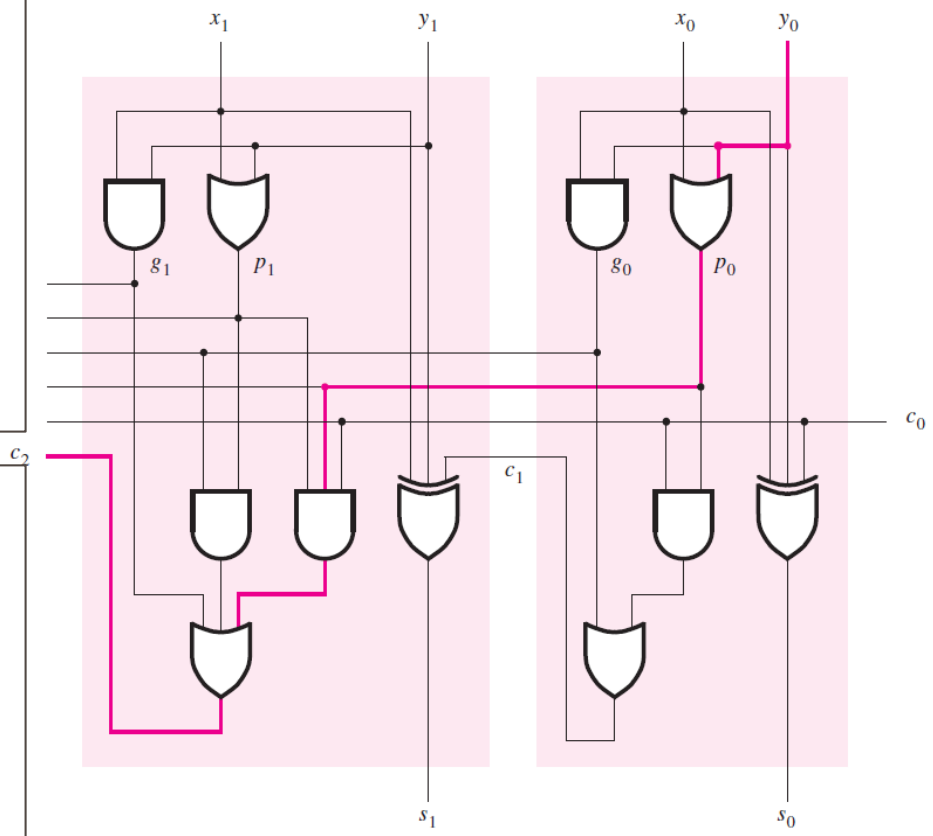
$$\boldsymbol{c_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \cdots + p_i p_{i-1} \ldots p_2 p_1 g_0 + p_i p_{i-1} \ldots p_1 p_0 c_0}$$

- Computation of c_{i+1} requires only two-levels (one AND and one OR)
- Computation of carry is very fast (computation of c_{i+1} does not wait for c_i)
- Adder based on this equation of carry is known as carry-lookahead adder (CLA)

Carry lookahead adder

- $c_0 = Input\ Carry$
- $c_1 = g_0 + p_0 c_0$
- $c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$
- $c_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$

Carry bit computation requires two level only



- One gate delay for all generate/propagate signal
- Two gate delays for AND/OR operation
- Therefore, carry bit computation takes 3 gate delays for all the stages (irrespective of number of bits)
- Key to improvement is the fast computation of carry bits

Propagate Functions: In terms of XOR

- $c_{i+1} = x_i y_i + x_i c_i + y_i c_i$

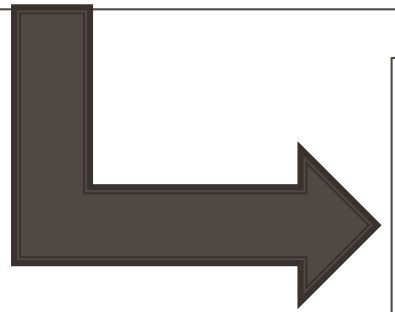
where $\boldsymbol{g_i = x_i y_i}$ and $\boldsymbol{p_i = (x_i + y_i)}$

- Propagate function can also be written as:

$$\boldsymbol{p_i = (x_i \oplus y_i)}$$

- Carry function written in the form of OR gate or XOR gate are **equivalent**
- $c_{i+1} = x_i y_i + (x_i + y_i) c_i \equiv x_i y_i + (x_i \oplus y_i) c_i$

Arithmetic Operations



Multiplication

Multiplication: by 2^i

- How to multiply a binary number by 2^i (i.e. $2, 4, 8, 16, \ldots$)?

- Shift the number left 'i' times and fill the number with 0 in LSB

- Assume the number is: $\mathrm{B} = b_{n-1}b_{n-2} \ldots b_1 b_0$

$2 \times \mathrm{B} = b_{n-1}b_{n-2} \ldots b_1 b_0 0$

$4 \times \mathrm{B} = b_{n-1}b_{n-2} \ldots b_1 b_0 00$

....

- This is correct for both unsigned and signed numbers
- Number of bits in the given number increases

- Assume: $\mathrm{B} = (1001)_2 = (9)_{10}$

$2 \times \mathrm{B} = (10010)_2 = (18)_{10}$

- Assume: $\mathrm{B} = (10111)_2 = -(9)_{10}$

$2 \times \mathrm{B} = (101110)_2 = -(18)_{10}$

Division: by 2^i

- How to divide a number by 2^i (i.e. $2, 4, 8, 16, \ldots$)?

- Shift the number right 'i' times and fill the left-most bit with the ***sign bit***

- Assume the number is: $\mathrm{B} = b_{n-1}b_{n-2} \ldots b_1 b_0$

  $\mathrm{B} \div 2 = b_{n-1}b_{n-1}b_{n-2} \ldots b_2 b_1$

  $\mathrm{B} \div 4 = b_{n-1}b_{n-1}b_{n-1} \ldots b_3 b_2$

  ....

- This is correct for both unsigned and signed numbers
- Number of bits in the given number remains same

- Assume: $\mathrm{B} = (011000)_2 = (24)_{10}$

  $\mathrm{B} \div 2 = (001100)_2 = (12)_{10}$

- Assume: $\mathrm{B} = (101000)_2 = (-24)_{10}$

  $\mathrm{B} \div 2 = (110100)_2 = (-12)_{10}$

Multiplier: based on shift and add

- Consider multiplication of two 4-bit numbers: $M = m_3 m_2 m_1 m_0$ and $Q = q_3 q_2 q_1 q_0$
- Need to find the product: $P = p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0$

- $Q = q_3 q_2 q_1 q_0 = (q_0 + 2 \times q_1 + 4 \times q_2 + 8 \times q_3)$

- $M \times Q = M \times (q_0 + 2 \times q_1 + 4 \times q_2 + 8 \times q_3)$

$= (M \times q_0) + (2 \times M \times q_1) + (4 \times M \times q_2) + (8 \times M \times q_3)$

- An eight-bit adder is used to compute the ***partial product***
- Total product is computed as a sum of partial products

- Partial product $P = p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0 = 0$ is initialized

- The bit q_0 is examined
  - If $q_0 = 1$, then M is added to the initial partial product, else 0 is added to P

- The bit q_1 is examined
  - If $q_1 = 1$, then $2 \times M$ is added to P, else 0 is added to P
  - $2 \times M$ is created just by shifting M one bit position to the left

- If $q_2 = 1$, then $4 \times M$ is added to P
- If $q_3 = 1$, then $8 \times M$ is added to P

- Since single adder is used it is slow

Multiplication: Method (Unsigned Numbers)

- Multiplication of binary number is performed in the same manner as decimal number

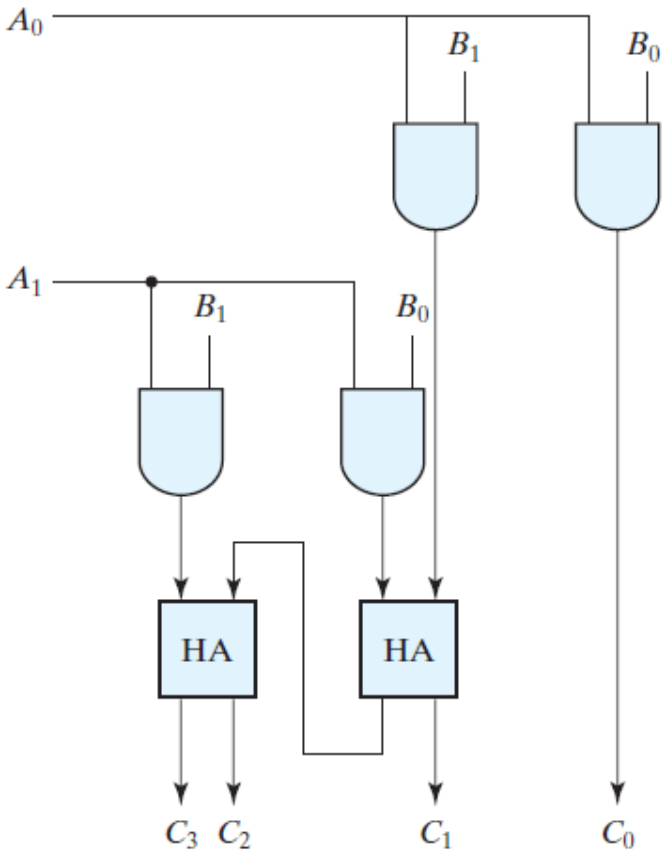
Multiplicand M	(14)	1 1 1 0
Multiplier Q	(11)	× 1 0 1 1
		1 1 1 0
		1 1 1 0
		0 0 0 0
		1 1 1 0
Product P	(154)	1 0 0 1 1 0 1 0

Multiplicand M	(11)	1 1 1 0
Multiplier Q	(14)	× 1 0 1 1
Partial product 0		1 1 1 0
		+ 1 1 1 0
Partial product 1		1 0 1 0 1
		+ 0 0 0 0
Partial product 2		0 1 0 1 0
		+ 1 1 1 0
Product P	(154)	1 0 0 1 1 0 1 0

- Faster multiplier can be built using multiple adders to compute partial products (in the form of an array)

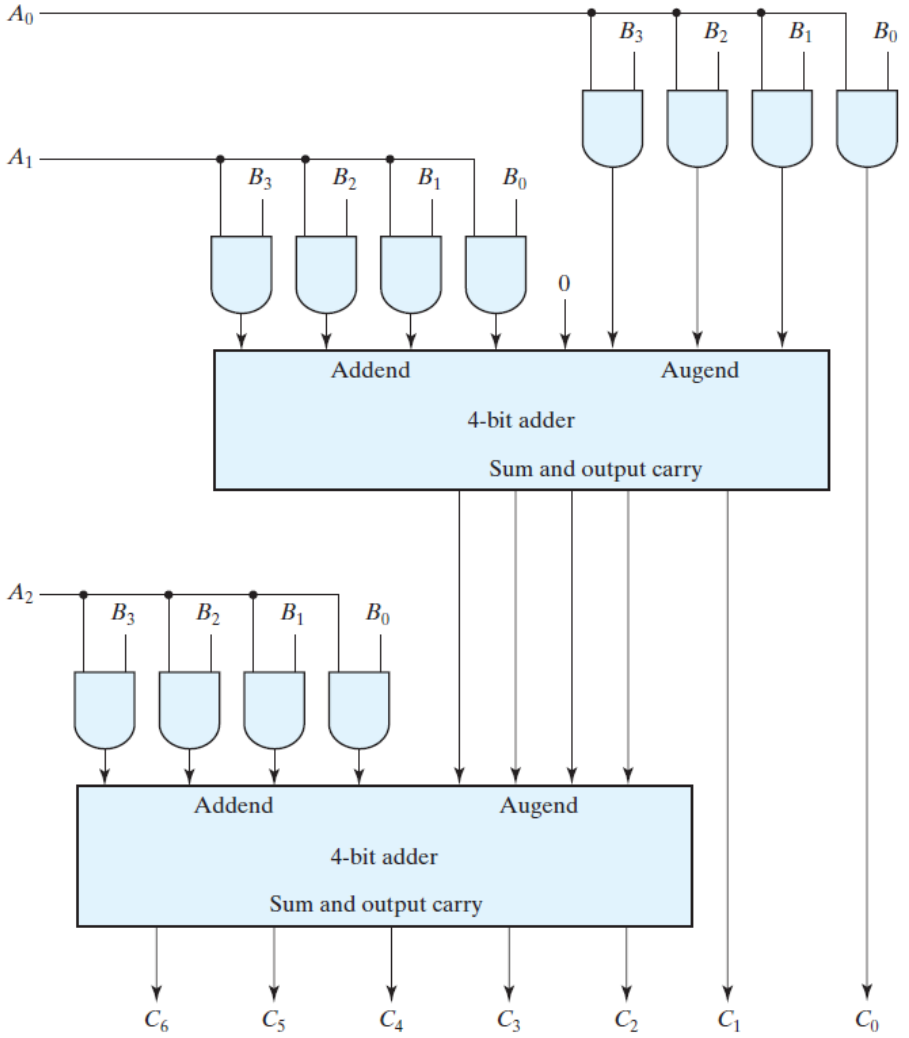
Multiplication: Array structure (2 × 2)

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



- Partial products are computed using HA's in this case
- If number of bits is more, then FA's are required

Multiplication: Array structure (4×3)



- A bit of the multiplier is AND-ed with each bit of the multiplicand in as many levels as there are bits in the multiplier
- The binary output in each level of AND gates is added with the ***partial product of the previous level*** to form a new partial product.
- For J multiplier bits and K multiplicand bits:
  - $(J \times K)$ AND gates needed
  - $(J-1)$ $K-bit$ adders to produce a product of (J + K) bits.

Digital Circuits: Practice Problems

Problems 4.10-4.17

from “Digital Design”– M. Morris Mano & Michael D. Ciletti, Ed-5, Pearson (Prentice-Hall).

Problems 5.4, 5.5, 5.7, 5.12, 5.13

from Fundamentals of Digital Logic with Verilog Design - S. Brown, Z. Vranesic