14:332:231 DIGITAL LOGIC DESIGN

Ivan Marsic, Rutgers University
Electrical & Computer Engineering
Fall 2013

Lecture #14: Adders, Subtracters, and ALUs

Binary Adder [Wakerly 4th Ed., Sec. 6.10, p. 474]

- Binary addition is used frequently
- Addition Development:
  - *Full-Adder* (FA), a 3-input bit-wise addition functional block, (Single bit)
  - *Ripple Carry Adder*, an iterative array to perform binary addition, and (Vector)
  - *Carry-Look-Ahead Adder* (CLA), a hierarchical structure to improve performance (check in Wikipedia: http://en.wikipedia.org/wiki/Carry_look-ahead_adder). (Improved adder)

Functional Block: Half Adder

- A 2-input, 1-bit width binary adder that performs the following computations:

X	0	0	1	1
+ Y	+ 0	+ 1	+ 0	+ 1
CO HS	0 0	0 1	0 1	1 0

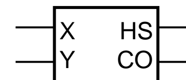
- A half adder adds two bits to produce a two-bit sum
- The low-order bit is named "half sum" (HS), and the high-order bit is named "carry out" (CO)
- The half adder can be specified as a *truth table* for HS and CO →

X	Y	CO	HS
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Half Adder

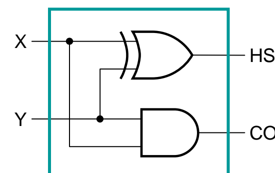
- Half-adder for 1-bit addends

X	Y	CO	HS
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



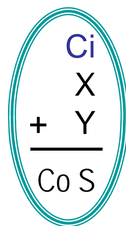
half sum: $HS = X \oplus Y$

carry out: $CO = X \cdot Y$



Full Adder [Recall *Binary Addition* from Lecture #2]

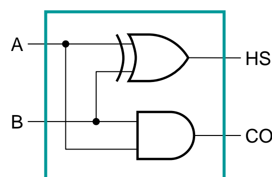
- Basic building block is "full adder"
  - 1-bit-wide adder, produces sum and carry outputs
- Truth table:



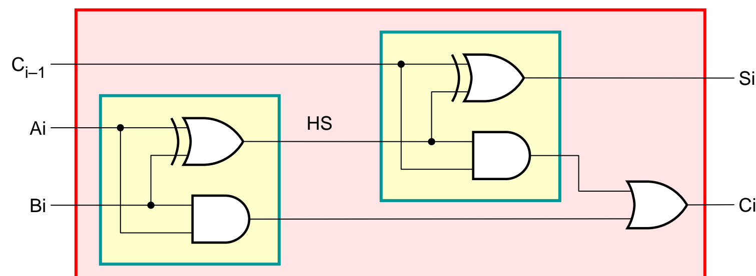
Row	Inputs			Outputs	
	X	Y	Cin	S	Cout
0	0	0	0	0	0
1	0	0	1	1	0
2	0	1	0	1	0
3	0	1	1	0	1
4	1	0	0	1	0
5	1	0	1	0	1
6	1	1	0	0	1
7	1	1	1	1	1

Full Adder from Half Adders

1-bit half adder



1-bit full adder



Full Adder

first term

$\mathbf{S} = HS \oplus CIN = X \oplus Y \oplus CIN = \underbrace{(X \cdot Y' + X' \cdot Y)}_{\text{first term}} \oplus CIN$

$= \underbrace{X \cdot Y' \cdot CIN' + X' \cdot Y \cdot CIN'}_{\text{first term direct}} + \underbrace{X' \cdot Y' \cdot CIN + X \cdot Y \cdot CIN}_{\text{first term complement}}$

first term direct — first term complement

$\mathbf{COUT} = X \cdot Y + \underbrace{X \cdot CIN + Y \cdot CIN}$

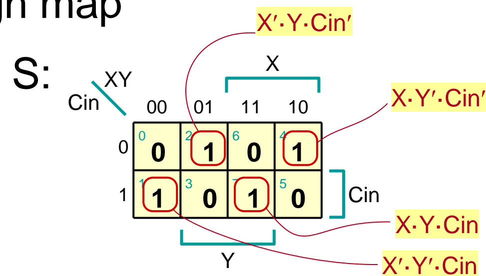
carry generated internally — carry generated by CIN

Logic Optimization: Full Adder

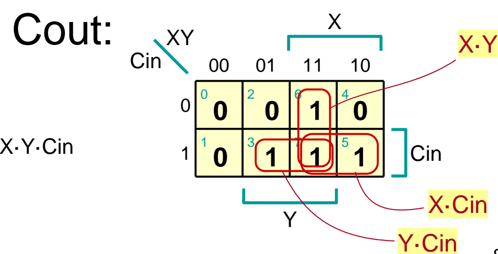
- Full adder Karnaugh map

Row	Inputs			Outputs	
	X	Y	Cin	S	Cout
0	0	0	0	0	0
1	0	0	1	1	0
2	0	1	0	1	0
3	0	1	1	0	1
4	1	0	0	1	0
5	1	0	1	0	1
6	1	1	0	0	1
7	1	1	1	1	1

S:



Cout:



$\mathbf{S} = X \cdot Y' \cdot Cin' + X' \cdot Y \cdot Cin' + X' \cdot Y' \cdot Cin + X \cdot Y \cdot Cin$

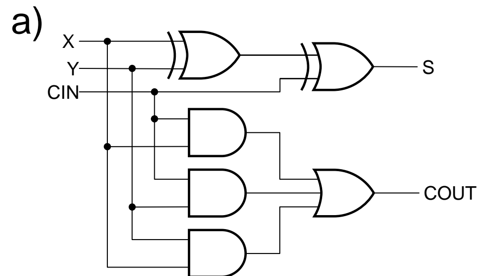
$\mathbf{Cout} = X \cdot Y + X \cdot Cin + Y \cdot Cin$

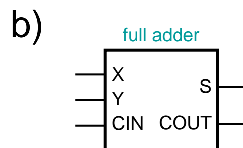
Full Adder Circuit

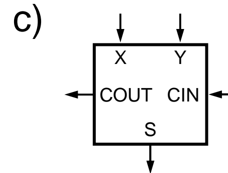
a) Gate-level circuit diagram

b) Logic symbol

c) Alternate logic symbol suitable for cascading





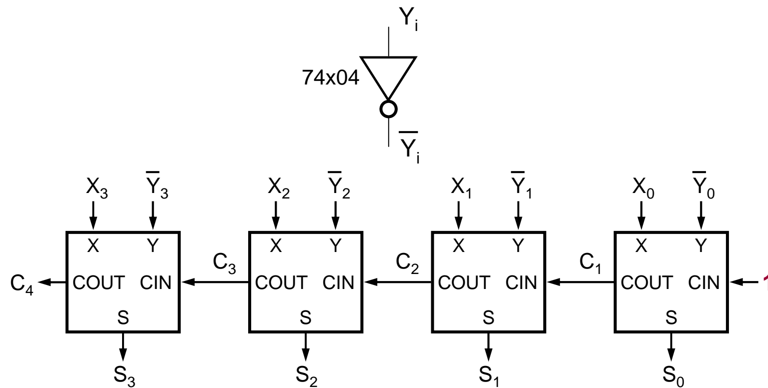


Subtraction

- Subtraction is the same as addition of the twos complement
- Recall Lecture #2:
  The two's complement is the bit-by-bit complement plus 1
- Therefore, $X - Y = X + \overline{Y} + 1$
  - Complement Y inputs to adder, set first C_{in} to 1

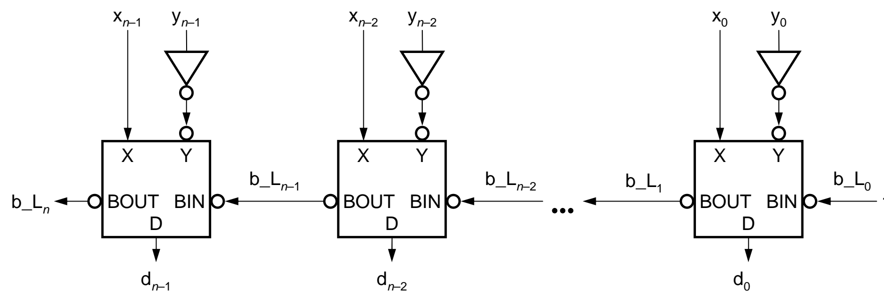
Subtractor Design Using Adders

- Ripple subtractor



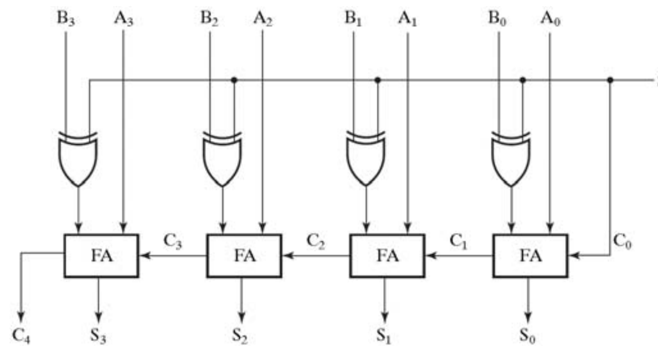
Subtractor Design Using Adders

- Ripple subtractor



2's Complement Adder/Subtractor

- Subtraction can be done by addition of the 2's Complement.
  1. Complement each bit (1's Complement.)
  2. Add 1 to the result.
- The circuit shown computes both **A + B** and **A – B**:
  - For S = 1, subtract, the 2's complement of B is formed by using XORs to form the 1's comp and adding the 1 applied to C_0.
  - For S = 0, add, B is passed through unchanged



How to Detect Overflow

- Rule was: Sign of the two operands identical and *different* from the sign of the result [recall Lecture #3]
- Sign = most significant bit (MSB)

$$OVR = X_{n-1} \cdot Y_{n-1} \cdot S'_{n-1} + X'_{n-1} \cdot Y'_{n-1} \cdot S_{n-1}$$

or:

$$OVR = C_{n-1} \oplus C_n$$ carry-in different from carry-out

011··· 1	$2^{n-1} - 1$
000··· 1	1
100··· 0	

OVR = 0·0·0 + 1·1·1 = 1 or

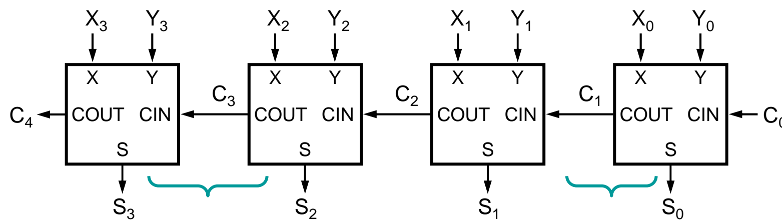
OVR = 1 ⊕ 0 = 1

Ripple Adder

- To add multiple operands, we "bundle" logical signals together into vectors and use *functional blocks* that operate on the vectors
- Example:
  4-bit **ripple carry adder**:
  Adds input vectors A(3:0) and B(3:0) to get a sum vector S(3:0)
- Note: carry-out of block *i* becomes carry-in of block *i* + 1

Description	Bit index 3 2 1 0	Name
Carry in	0 1 1 0	Ci
Augend	1 0 1 1	Ai
Addend	0 0 1 1	Bi
Sum	1 1 1 0	Si
Carry out	0 0 1 1	C_{i+1}

Ripple Adder



- It is relatively slow: For *n* bits, the worst case is:
- All of the adder's bits (and c_0) are present simultaneously

111··· 1	–1
000··· 1	+1

- tADD = $t_{XYC_{OUT}} + (n-2) \cdot t_{C_{IN}C_{OUT}} + t_{C_{IN}S}$

  LSB (*out* C_1) MSB (*in* C_{n-1})

- Carry look-ahead adders are the solution

Carry Lookahead Adder

- Uses a different circuit to calculate the carry out (calculates it *ahead* of the addition), to speed up the overall addition
- Requires more complex circuits
- Trade-off: speed vs. area (complexity, cost)

Carry Look-Ahead Addition

- **Carry generate**: input bits combination (x_i, y_i) that produces a carry-out of "1" $(c_{i+1} = 1)$ independent of lower-order bits $(x_0 \ldots x_{i-1}, y_0 \ldots y_{i-1})$ and c_0.
- **Carry propagate**: input bits combination (x_i, y_i) that produces a carry-out of "1" $(c_{i+1} = 1)$ when $c_i = 1$.

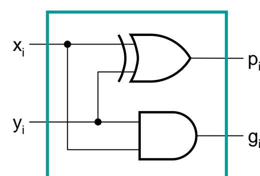
Inputs			Outputs		
X	Y	Cin	S	Cout	
0	0	0	0	0	No carry
0	0	1	1	0	No carry
0	1	0	1	0	No carry
0	1	1	0	1	Carry propagated
1	0	0	1	0	No carry
1	0	1	0	1	Carry propagated
1	1	0	0	1	Carry *generated*
1	1	1	1	1	Carry *generated* & propagated

Cin column: c_i; Cout column: c_{i+1}

$g_i = x_i \cdot y_i$ ← carry-generate

$p_i = x_i + y_i$ ← carry-propagate

$\Rightarrow c_{i+1} = g_i + p_i \cdot c_i$



Note that we could use half adder: $p_i^* = x_i \oplus y_i$

4-bit Carry Lookahead Adder

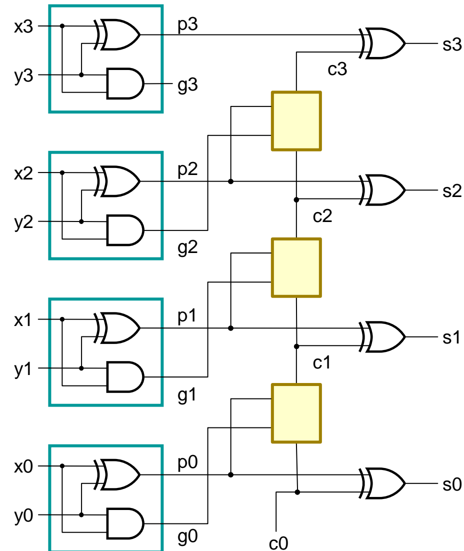
- Conceptual diagram

p_i^* see previous slide

$s_i = HS_i \oplus c_i = p_i \oplus c_i$

$c_{i+1} = g_i + p_i \cdot c_i$

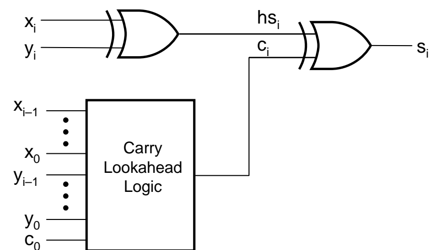
Note that $g_i = 1 \Rightarrow p_i = 1$
(but not vice versa)
$g_i \Rightarrow p_i \cdot g_i$



Carry Lookahead Logic

- Structure of one stage of a carry-lookahead adder:

$g_i = x_i \cdot y_i$ ← carry-generate signal
$p_i = x_i + y_i$ ← carry-propagate signal



Carry equations for first 4 adder stages

$c_1 = p_0 \cdot (g_0 + c_0)$

$c_2 = p_1 \cdot (g_1 + c_1)$
$\quad = p_1 \cdot (g_1 + p_0 \cdot (g_0 + c_0))$
$\quad = p_1 \cdot (g_1 + p_0) \cdot (g_1 + g_0 + c_0)$ distributivity theorem

$c_3 = p_2 \cdot (g_2 + c_2)$
$\quad = p_2 \cdot (g_2 + p_1 \cdot (g_1 + p_0) \cdot (g_1 + g_0 + c_0))$
$\quad = p_2 \cdot (g_2 + p_1) \cdot (g_2 + g_1 + p_0) \cdot (g_2 + g_1 + g_0 + c_0)$

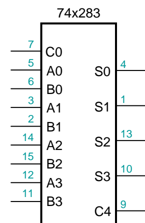
$c_4 = p_3 \cdot (g_3 + c_3)$
$\quad = p_3 \cdot (g_3 + p_2 \cdot (g_2 + p_1) \cdot (g_2 + g_1 + p_0) \cdot (g_2 + g_1 + g_0 + c_0))$
$\quad = p_3 \cdot (g_3 + p_2) \cdot (g_3 + g_2 + p_1) \cdot (g_3 + g_2 + g_1 + p_0) \cdot (g_3 + g_2 + g_1 + g_0 + c_0)$

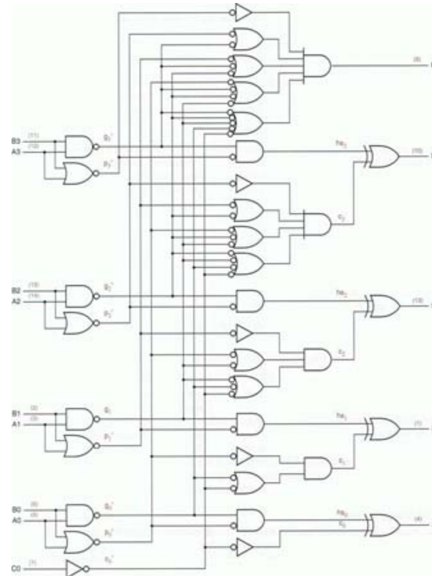
74x283 4-bit Adder

- Uses carry lookahead (CLA) internally
- Differences from general CLA design:
  - Active-low versions of carry-generate (g_i') and carry-propagate (p_i') (b/c inverting gates are faster)
  - Algebraic manipulation of the half-sum:

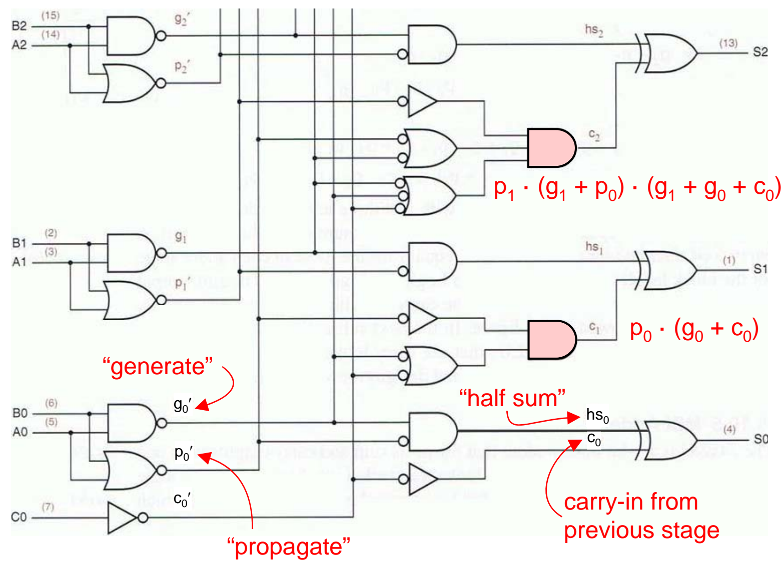
    $hs_i = x_i \oplus y_i = x_i \cdot y_i' + x_i' \cdot y_i$
    $\quad = (x_i + y_i) \cdot (x_i' + y_i')$
    $\quad = (x_i + y_i) \cdot (x_i \cdot y_i)'$
    $\quad = p_i \cdot g_i'$
  - Creates the carry signals using INVERT-OR-AND (has ≈ delay as a single inverting gate):

    $c_{i+1} = p_i \cdot g_i + p_i \cdot c_i$
    $\quad = p_i \cdot (g_i + c_i)$
- See Wakerly 4th ed., page 481, for carry equations





74x283 4-bit Adder (detail)



16-bit Group-ripple Adder

- Ripple carry between groups
- Total propagation delay ≈8 inverting gates