Binary Adder

- Binary Addition
  - single bit addition

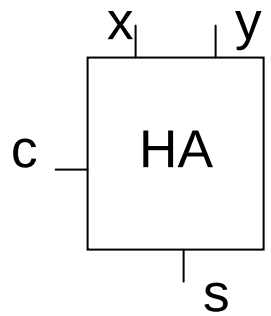
x	*y*	*x + y (binary sum)*
0	0	= 0
0	1	= 1
1	0	= 1
1	1	= 10 (binary, i.e. 2 in base-10)

  - sum of 2 binary numbers can be larger than either number
  - need a "carry-out" to store the overflow
- Half-Adder
  - 2 inputs (x and y) and 2 outputs (sum and carry)

x	*y*	*s*	*c*
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$s = x \oplus y$ XOR

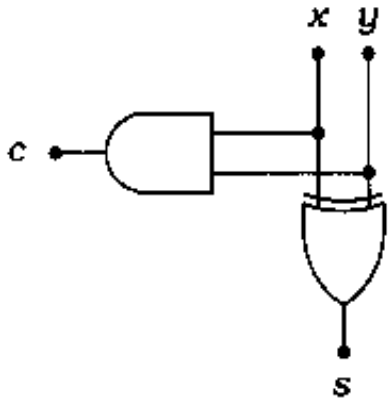
$c = x \bullet y$ AND



half-adder symbol

Half-Adder Circuits

- Simple Logic
  - using XOR gate

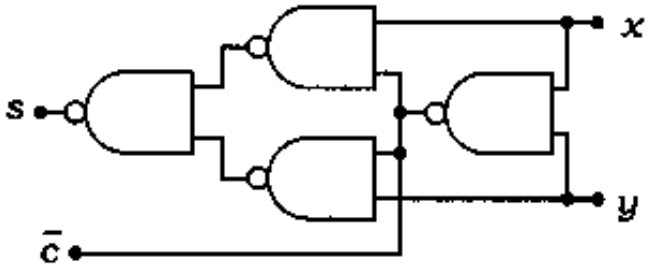


x	y	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

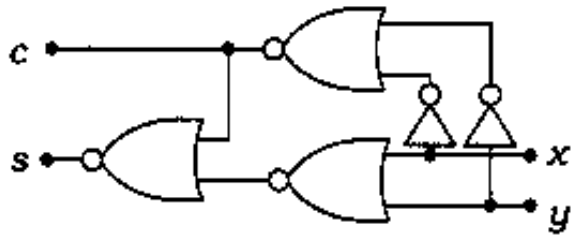
$s = x \oplus y$

$c = x \bullet y$

- Most Basic Logic
  - NAND and NOR only circuits



(a) NAND2 logic



(b) NOR-based network

Take-home Questions:

Which of these 3 half-adders will be fastest? slowest? why??

Which has fewest transistors? Which transition has the critical delay?

Full-Adder

- When adding more than one bit, must consider the carry of the previous bit
  - full-adder has a "carry-in" input
- Full-Adder Equation

for every i-th bit

	c_i	carry-in
	a_i	+ a
+	b_i	+ b
c_{i+1}	s_i	= carry-out, sum

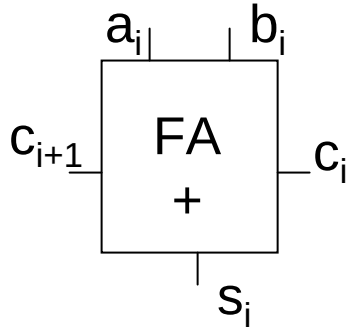
- Full-Adder Truth Table

a_i	b_i	c_i	s	c_{i+1}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

$$s_i = a_i \oplus b_i \oplus c_i$$
$$c_{i+1} = a_i \bullet b_i + c_i \bullet (a_i \oplus b_i)$$

if not trying to 'reuse' the $a_i \oplus b_i$ term from sum, can write

$$c_{i+1} = a_i \bullet b_i + c_i \bullet (a_i + b_i)$$

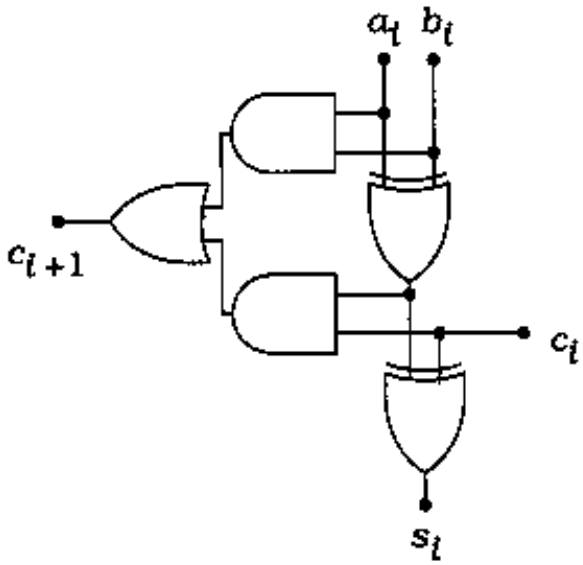


full-adder symbol

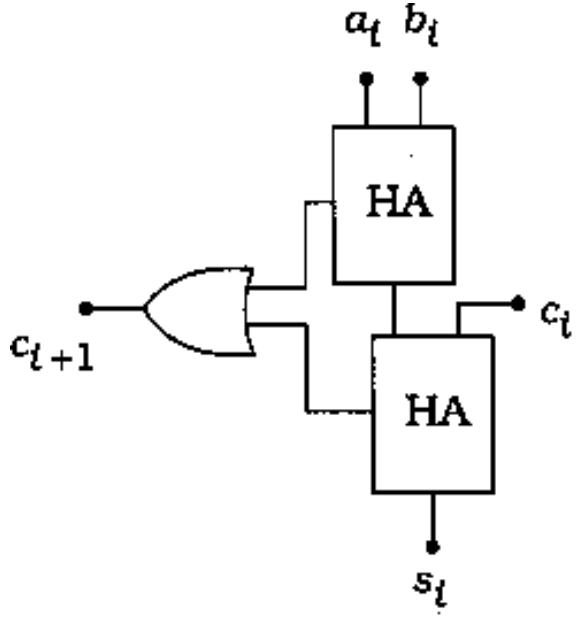
Full-Adder Circuits

Full-Adder Equations: $s_i = a_i \oplus b_i \oplus c_i$ and $c_{i+1} = a_i \bullet b_i + c_i \bullet (a_i \oplus b_i)$

- XOR-based FA



- HA-based FA



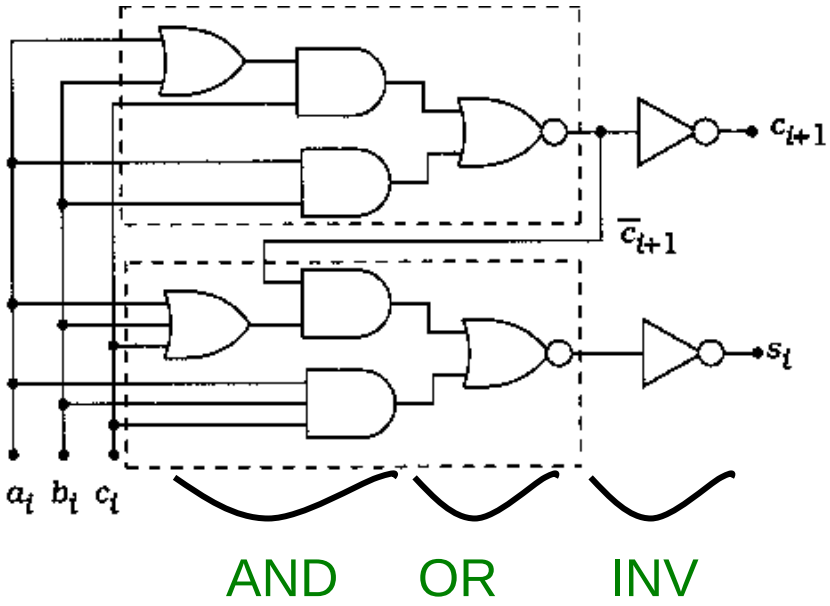
- Other FA Circuits
  - a few others options are covered in the textbook

Full Adder Circuits

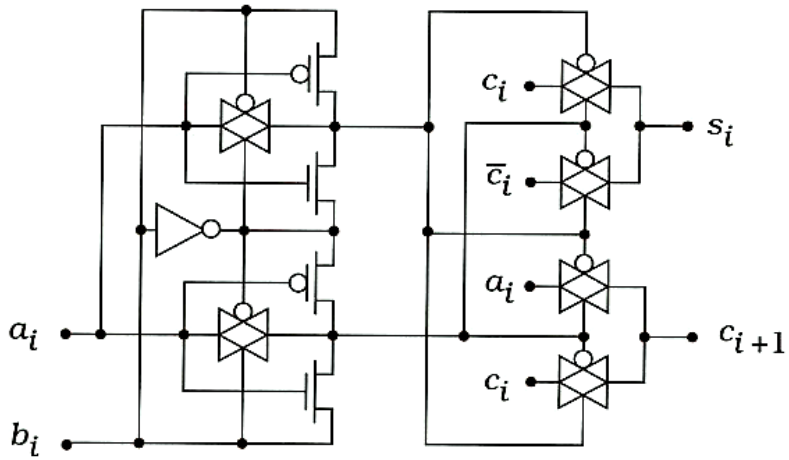
- AOI Structure FA
  - implements following SOP equations

$$c_{i+1} = a_i \bullet b_i + c_i \bullet (a_i + b_i)$$
$$\overline{s}_i = (a_i + b_i + c_i) \bullet c_{i+1} + (a_i \bullet b_i \bullet c_i)$$

  - sum delayed from carry



- Transmission Gate FA
  - sum and carry have about the same delay

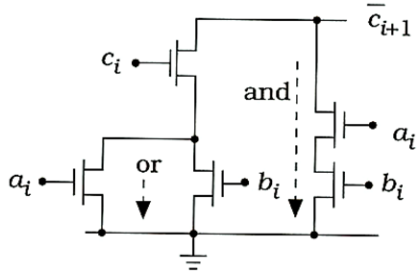


Full Adder in CMOS

- Consider nMOS logic for c_out

  $c_{i+1} = a_i \cdot b_i + c_i \cdot (a_i + b_i)$

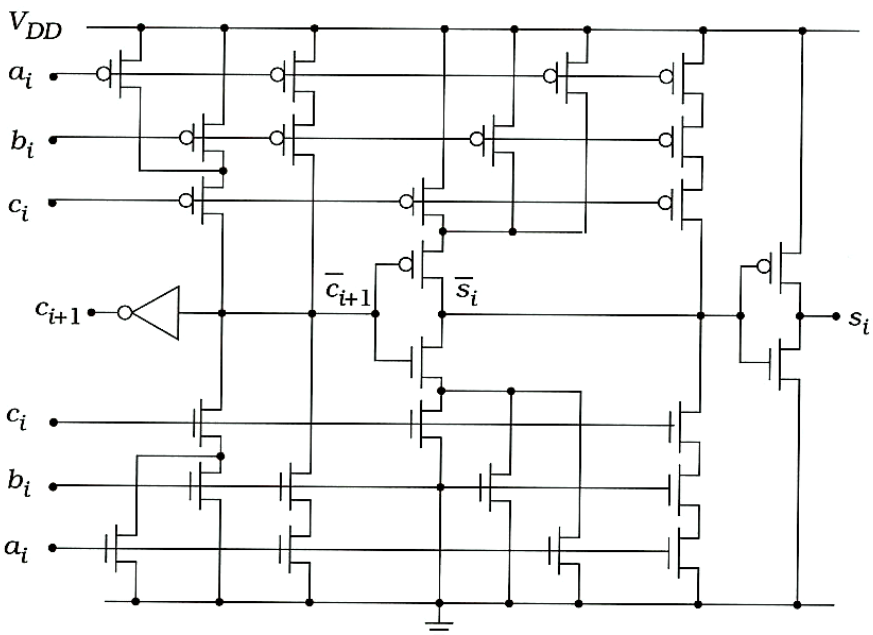
  - two "paths" to ground

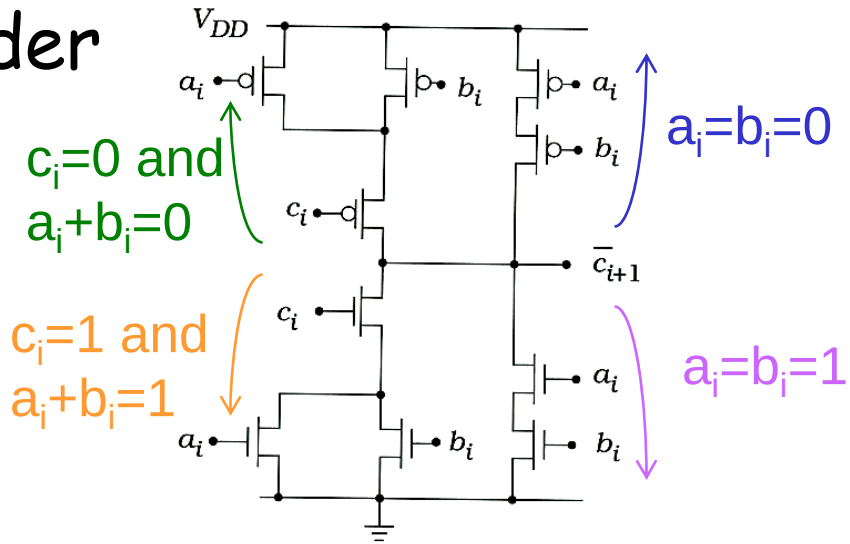


(a) Standard nFET logi

- Mirror CMOS Full Adder

  - carry out circuit





(b) Mirror circuit

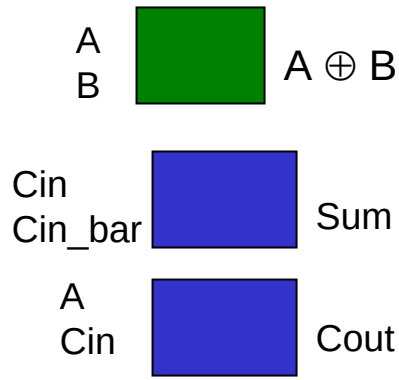
- complete circuit

FA Using 2:1 MUX

- If we re-arrange the FA truth table
  - can simplify the output (sum, carry) expressions

a_i	b_i	c_i	$a \oplus b$	s	c_{i+1}
0	0	0	0	0	0
1	1	0	0	0	1
0	0	1	0	1	0
1	1	1	0	1	1
0	1	0	1	1	0
1	0	0	1	1	0
0	1	1	1	0	1
1	0	1	1	0	1

If (A ⊕ B = 0), SUM=Cin; Cout=A;
Else, SUM=Cin_bar; Cout=Cin;



Partial Schematic
can you figure out the details?

- Implementation
  - use an XOR to make the decision (a⊕b=0?)
  - use a 2:1 MUX to select which equation/value of sum and carry to pass to the output

Binary Word Adders

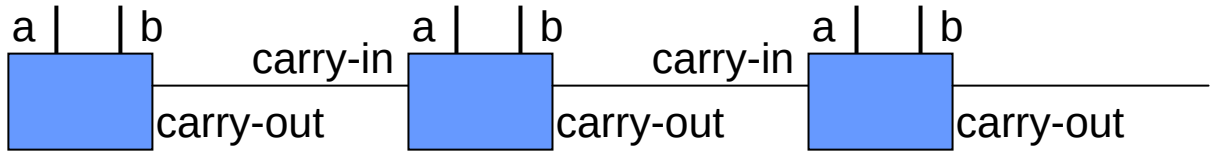
- Adding 2 binary (multi-bit) words
  - adding 2 n-bit word produces an n-bit sum and a carry
  - example: 4b addition

$$\begin{array}{r} a_3\ a_2\ a_1\ a_0 \\ +\ b_3\ b_2\ b_1\ b_0 \\ \hline c_4\ s_3\ s_2\ s_1\ s_0 \end{array}$$

4b input a
+ 4b input b
= carry-out, 4b sum

- Carry Bits
  - binary adding of n-bits will produce an n+1 carry
  - can be used as carry-in for next stage or as an overflow flag

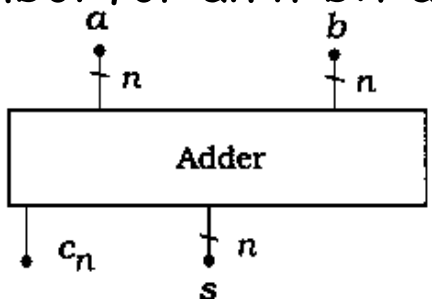
- Cascading Multi-bit Adders
  - carry-out from a binary word adder can be passed to next cell to add larger words
  - example: 3 cascaded 4b binary adders for 12b addition

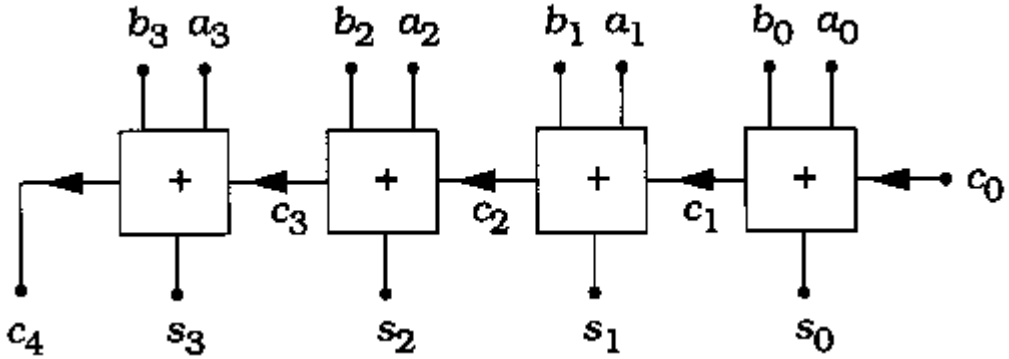


Ripple Carry Adder

- To use single bit full-adders to add multi-bit words
  - must apply carry-out from each bit addition to next bit addition
  - essentially like adding 3 multi-bit words
    - each c_i is generated from the i-1 addition
  - c_0 will be 0 for addition
    - kept in equation for generality
  - symbol for an n-bit adder

$$\begin{array}{rl} c_3\ c_2\ c_1\ c_0 & \text{carry-in bits} \\ a_3\ a_2\ a_1\ a_0 & \text{4b input a} \\ +\ b_3\ b_2\ b_1\ b_0 & \text{+ 4b input b} \\ \hline c_4\ s_3\ s_2\ s_1\ s_0 & \text{= carry-out, 4b sum} \end{array}$$



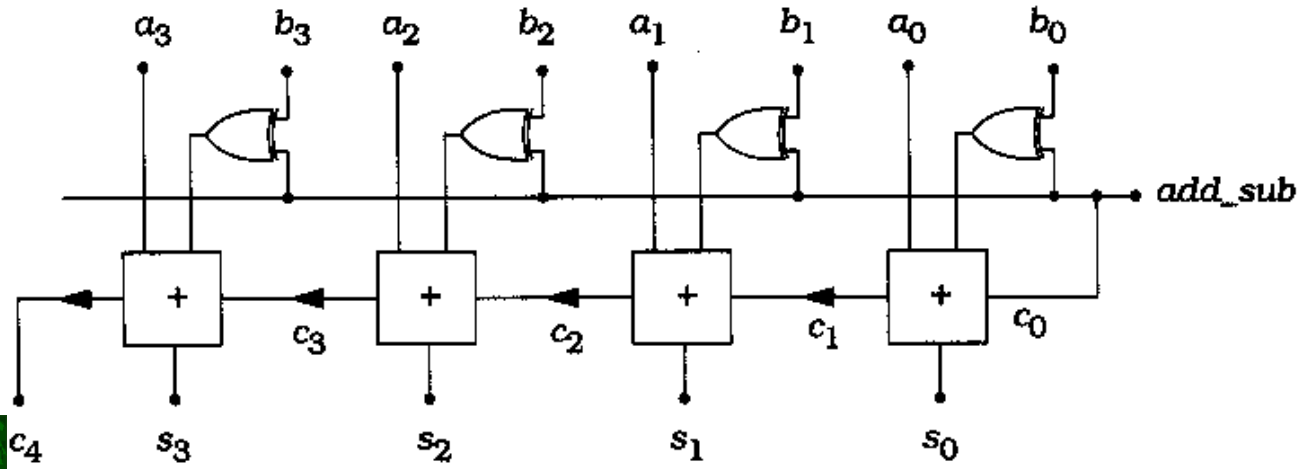


4b ripple-carry adder using 4 FAs

- Ripple-Carry Adder
  - passes carry-out of each bit to carry-in of next bit
  - for n-bit addition, requires n Full-Adders

Adder/Subtractor using R-C Adders

- Subtraction using 2's complements
  - 2's complement of X: $X_{2s} = \overline{X}+1$
    - invert and add 1
  - Subtraction via addition: Y - X = Y + X_{2s}
- R-C Adder/Subtactor Cell
  - control line, add_sub: 0 = add, 1 = subtract
  - XOR used to pass (add_sub=1) or invert (add_sub=0)
  - set first carry-in, c_0, to 1 will add 1 for 2's complement

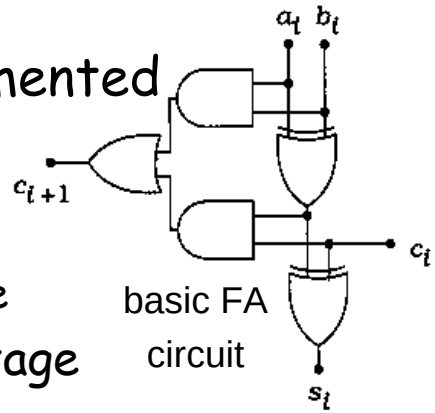


a = add_sub

a	b	a ⊕ b
0	0	0 b
0	1	1
1	0	1 $\overline{b}$
1	1	0

Ripple-Carry Adders in CMOS

- Simple to implement and connect for multi-bit addition
  - but, they are very slow
- Worse-case delays in R-C Adders
  - each bit in the cascade requires carry-out from the previous bit
    - major speed limitation of R-C Adders
  - delay depends somewhat on the type of FA implemented
  - general assumptions
    - worst delay in an FA is the sum
      - but carry is more important due to cascade structure
    - total delay is sum of delays to pass carry to final stage
    - total delay for n-input R-C adder



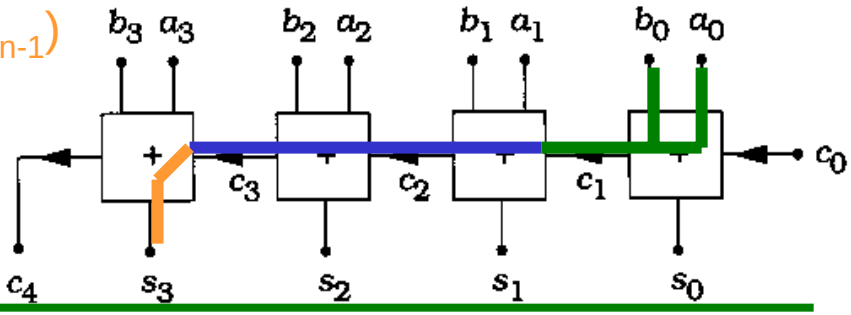
basic FA circuit

$t_n = t_d(a_0,b_0 \Rightarrow c_1) + (n-2)\ t_d(c_{in} \Rightarrow c_{out}) + t_d(c_{in} \Rightarrow s_{n-1})$

first stage delay: inputs to carry-out

middle stage (n-2) delay: carry-in to carry-out

last stage delay: carry-in to sum



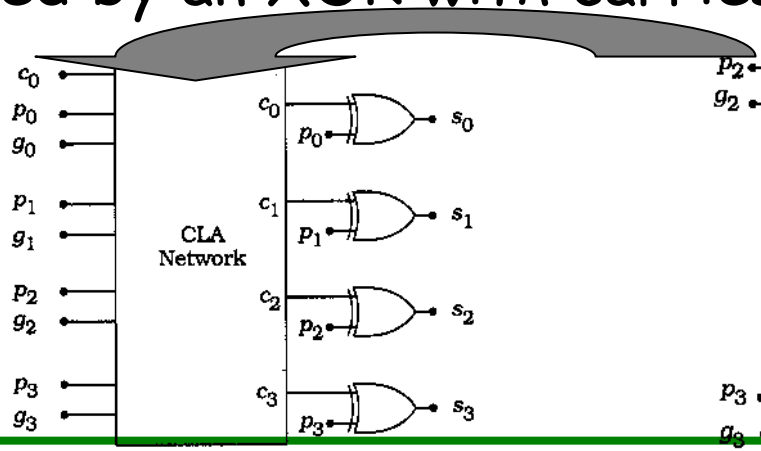
Carry Look-Ahead Adder

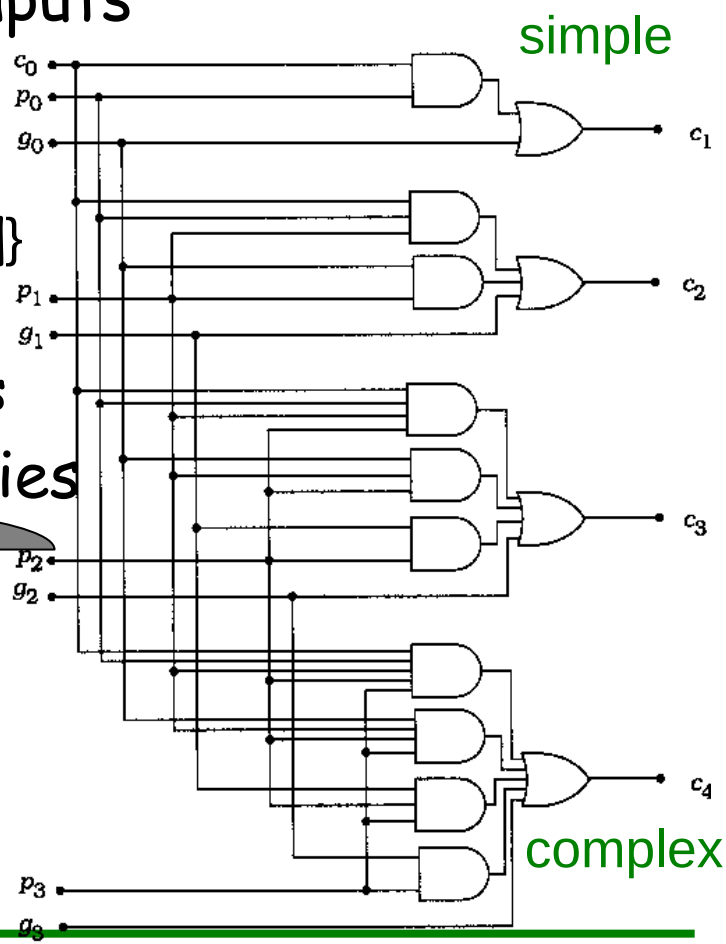
- CLA designed to overcome delay issue in R-C Adders
  - eliminates the ripple (cascading) effect of the carry bits
- Algorithm based calculating all *carry* terms at once
- Introduces generate and propagate signals
  - rewrite $c_{i+1} = a_i \bullet b_i + c_i \bullet (a_i \oplus b_i)$ → $\mathbf{c_{i+1} = g_i + c_i \bullet p_i}$
    - generate term, $\mathbf{g_i = a_i \bullet b_i}$
    - propagate term, $\mathbf{p_i = a_i \oplus b_i}$
  - approach: evaluate all g_i and p_i terms and use them to calculate all carry terms without waiting for a carry-out ripple
- All *sum* terms evaluated at once
  - the sum of each bit is: $\mathbf{s_i = p_i \oplus c_i}$
- Pros and Cons
  - no cascade delays; outputs expressed in terms of inputs only
  - requires complex circuits for higher bit-order adders (next slide)

Logic Circuits for a 4b CLA Adder

- Carry-out expressions for 4b CLA
  - $c_1 = g_0 + c_0 \cdot p_0$, $c_2 = g_1 + c_1 \cdot p_1$, $c_3 = g_2 + c_2 \cdot p_2$, $c_4 = g_3 + c_3 \cdot p_3$
- Expressed only in terms of known inputs
  - $c_2 = g_1 + p_1 \cdot (g_0 + c_0 \cdot p_0)$
  - $c_3 = g_2 + p_2 \cdot [g_1 + p_1 \cdot (g_0 + c_0 \cdot p_0)]$
  - $c_4 = g_3 + p_3 \cdot \{g_2 + p_2 \cdot [g_1 + p_1 \cdot (g_0 + c_0 \cdot p_0)]\}$
  - nested Sum-of-Products expressions
  - gets more complex for higher bit adders
- Sums obtained by an XOR with carries

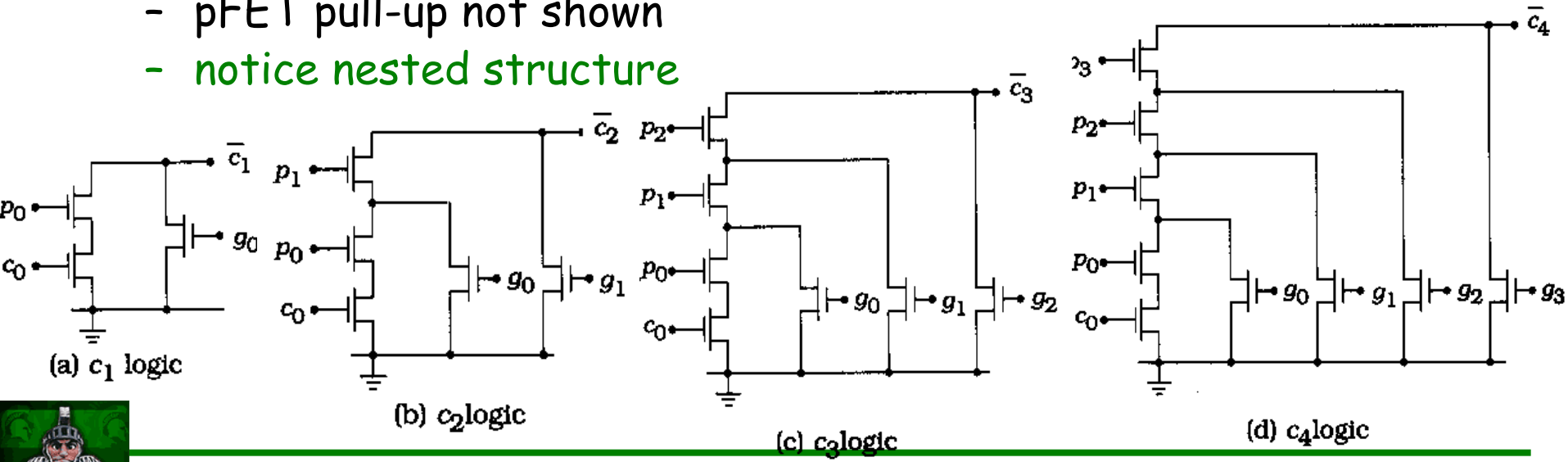
$$g_i = a_i \cdot b_i$$
$$p_i = a_i \oplus b_i$$





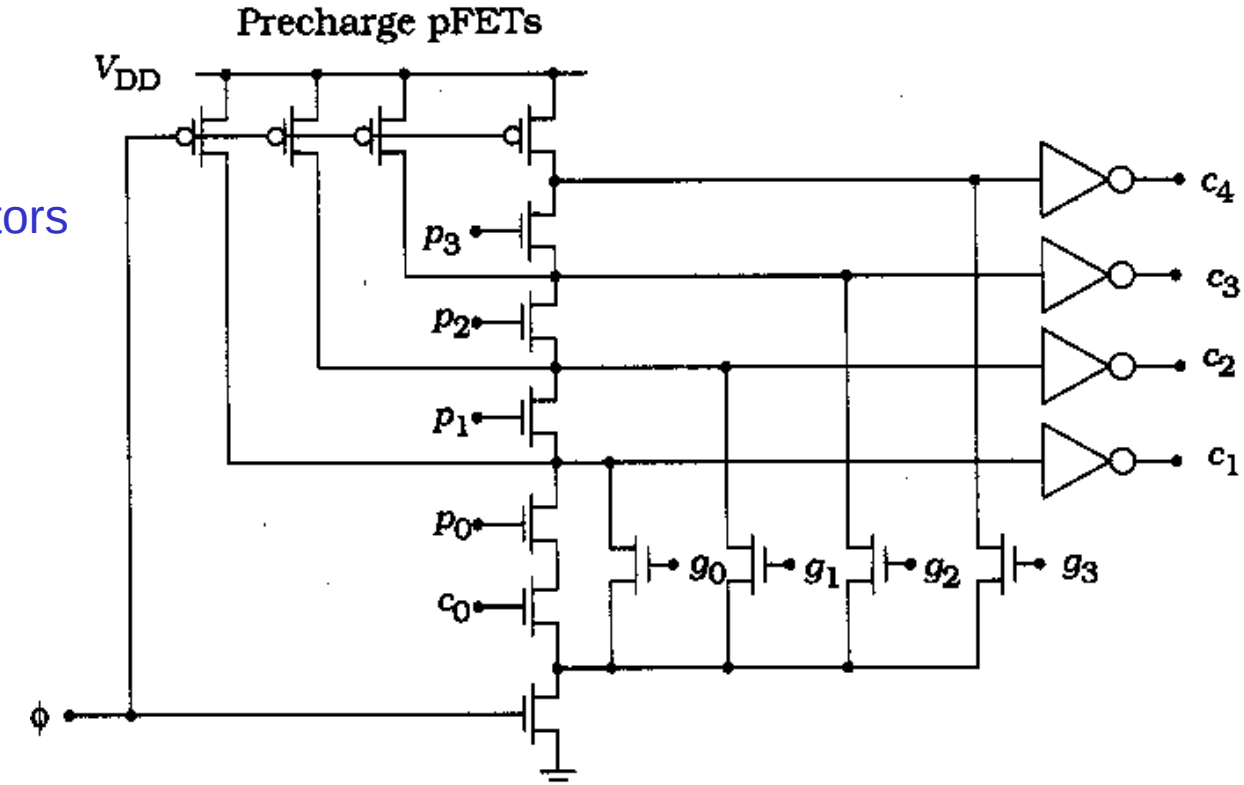
CLA Carry Generation in Reduced CMOS

- Reduce logic by constructing a CMOS push-pull network for each carry term
  - expanded carry terms
    - $c_1 = g_0 + c_0 \cdot p_0$
    - $c_2 = g_1 + g_0 \cdot p_1 + c_0 \cdot p_0 \cdot p_1$
    - $c_3 = g_2 + g_1 \cdot p_2 + g_0 \cdot p_1 \cdot p_2 + c_0 \cdot p_0 \cdot p_1 \cdot p_2$
    - $c_4 = g_3 + g_2 \cdot p_3 + g_1 \cdot p_2 \cdot p_3 + g_0 \cdot p_1 \cdot p_2 \cdot p_3 + c_0 \cdot p_0 \cdot p_1 \cdot p_2 \cdot p_3$
- nFETs network for each carry term
  - pFET pull-up not shown
  - notice nested structure



CLA in Advanced Logic Structures

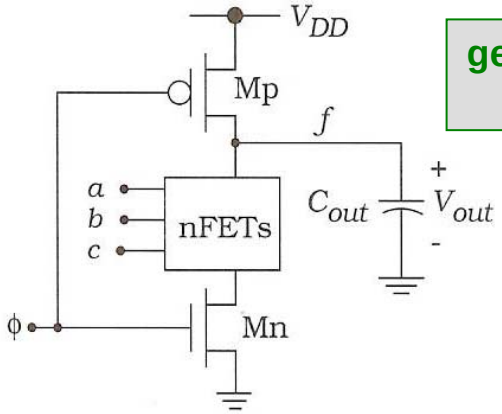
- CLA algorithm better implemented in dynamic logic
- Dynamic Logic (*jump to next slide*)
- Dynamic Logic CLA Implementation
  - multiple output domino logic (MODL)



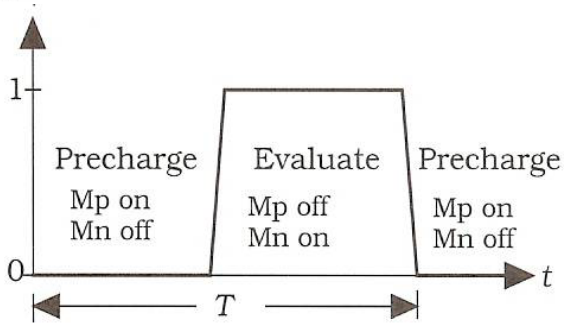
- significantly fewer transistors
- faster
- less chip area
- output only valid during evaluate period

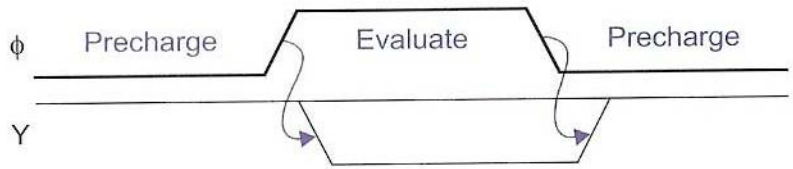
Dynamic Logic –Quick Look

- Advantages: fewer transistors & less power consumption
- General dynamic logic gate
  - nFET logic evaluation network
  - clocked "precharge" pull up pFET
  - clocked disabling nFET
- Precharge stage
  - clock-gated pull-up precharges output high
  - logic array disabled
- Evaluation stage
  - precharge pull-up disabled
  - logic array enabled & if true, discharges output
- Dynamic operation: output not always valid



generic dynamic logic gate





Manchester Carry Generation Concept

- Alternative structure for *carry* evaluation
  - define *carry* in terms of control signals such that
    - only one control is active at a given time
  - implement in **switch-logic**
- Consider single bit FA truth table
  - ***p*** OR ***g*** is high in 6 of 8 logic states
    - ***p*** and ***g*** are not high at the same time
  - introduce carry-kill, ***k***
    - on/high when neither p or g is high
    - carry_out always 0 when *k*=1
  - only one control signal (p, g, k) is active for each state

a_i	b_i	c_i	c_{i+1}	p_i	g_i	k_i
0	0	0	0	0	0	1
0	1	0	0	1	0	0
1	0	0	0	1	0	0
1	1	0	1	0	1	0
0	0	1	0	0	0	1
0	1	1	1	1	0	0
1	0	1	1	1	0	0
1	1	1	1	0	1	0

generate	$g_i = a_i \bullet b_i$
propagate	$p_i = a_i \oplus b_i$
carry-kill	$k_i = \overline{a_i + b_i}$

Manchester Carry Generation Concept

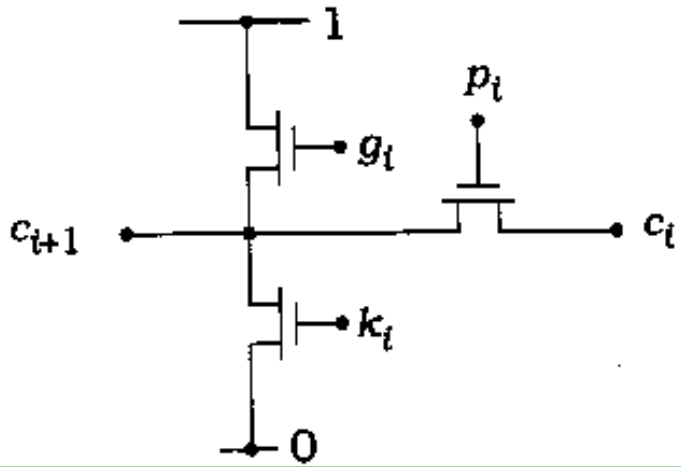
- Switch-logic implementation of truth table
  - 3 independent control signals ***g***, ***p***, ***k***
  - express carry_out in terms of ***g***, ***p***, ***k***

if $g = 1 \rightarrow c_{i+1} = 1$

if $p = 1 \rightarrow c_{i+1} = c_i$

if $k = 1 \rightarrow c_{i+1} = 0$

  - implement in **switch-logic**
    - only one switch ON at any time



generate	$g_i = a_i \bullet b_i$
propagate	$p_i = a_i \oplus b_i$
carry-kill	$k_i = \overline{a_i + b_i}$

a_i	b_i	c_i	c_{i+1}	p_i	g_i	k_i
0	0	0	0	0	0	1
0	1	0	0	1	0	0
1	0	0	0	1	0	0
1	1	0	1	0	1	0
0	0	1	0	0	0	1
0	1	1	1	1	0	0
1	0	1	1	1	0	0
1	1	1	1	0	1	0

Static CMOS Manchester Implementation

- Manchester carry generation circuit
- Static CMOS
  - modify for inverting logic
    - input → c_i_bar & output → c_{i+1}_bar
- New truth table
- Possible implementation
  - $\overline{c_{i+1}}$ = 1 if g_i=0
  - $\overline{c_{i+1}}$ = 0 if g_i=1 AND p_i=0
  - $\overline{c_{i+1}}$ = $\overline{c_i}$ if p_i=1
    - but g_i=0 here. problem?
  - carry-kill is not needed

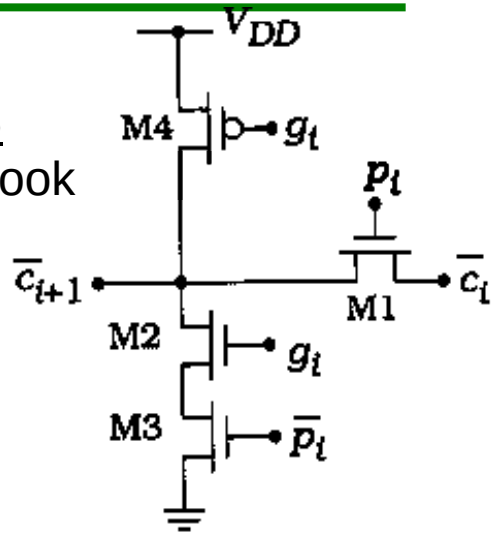
a_i	b_i	$\overline{c_i}$	$\overline{c_{i+1}}$	p_i	g_i
0	0	1	1	0	0
0	1	1	1	1	0
1	0	1	1	1	0
1	1	1	0	0	1
0	0	0	1	0	0
0	1	0	0	1	0
1	0	0	0	1	0
1	1	0	0	0	1

Static CMOS Manchester Implementation

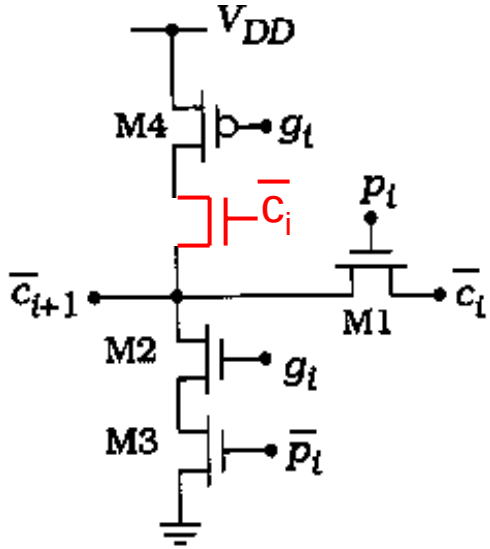
- Textbook Circuit Implementation
  - $\overline{c_{i+1}}$ = 1 if g_i=0
  - $\overline{c_{i+1}}$ = 0 if g_i=1 AND p_i=0
  - $\overline{c_{i+1}}$ = $\overline{c_i}$ if p_i=1
  - error
    - when g_i=0, p_i=1, $\overline{c_i}$=0, $\overline{c_{i+1}}$→0
    - pulled low through M1
    - but M4 pulls it high

a_i	b_i	$\overline{c_i}$	$\overline{c_{i+1}}$	p_i	g_i
0	0	1	1	0	0
0	1	1	1	1	0
1	0	1	1	1	0
1	1	1	0	0	1
0	0	0	1	0	0
0	1	0	0	1	0
1	0	0	0	1	0
1	1	0	0	0	1

static CMOS from textbook



- Possible Correction?
  - insert switch in pull-up path to disable when $\overline{c}_i$=0
  - solves error when g_i=0, p_i=1, c_i=0 → c_{i+1}=0
  - but introduces error when g_i=0, p_i=1, $\overline{c}_i$=0 → $\overline{c_{i+1}}$=1
    - M4 can not pull high since new nMOS cuts off path



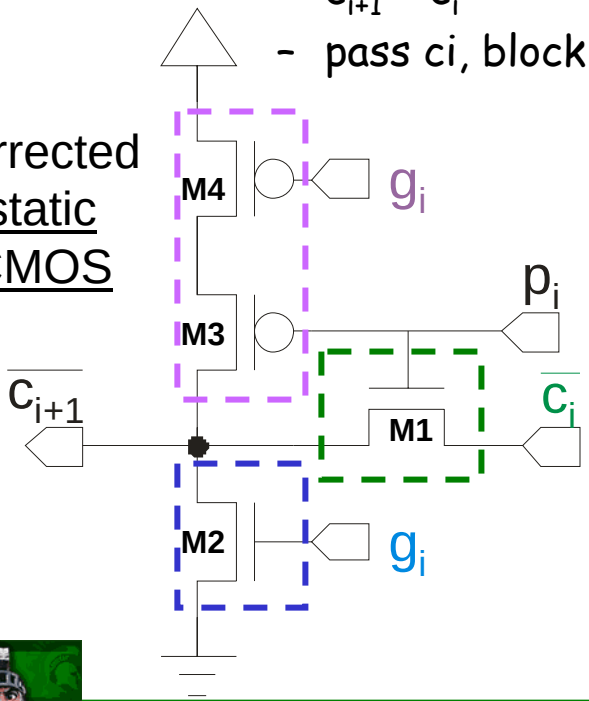
Manchester Implementation

- Corrected Manchester Carry Generation Circuit
  - truth table organized by p_i
    - if $p_i = 0$
      - $\overline{c_{i+1}} = g_i$ (NOT g_i)
      - block c_i, pass VDD or GND
    - if $p_i = 1$
      - $\overline{c_{i+1}} = \overline{c_i}$
      - pass ci, block VDD & GND

ai	bi	$\overline{ci}$	$\overline{ci+1}$	pi	gi	VDD	GND	Ci
0	0	1	1	**0**	0	act	x	x
0	0	0	1	**0**	0	act	x	x
1	1	1	0	**0**	1	x	act	x
1	1	0	0	**0**	1	x	act	x
0	1	1	1	1	0	x	x	act
1	0	1	1	1	0	x	x	act
0	1	0	0	1	0	x	x	act
1	0	0	0	1	0	x	x	act

act = active
x = disabled

corrected static CMOS



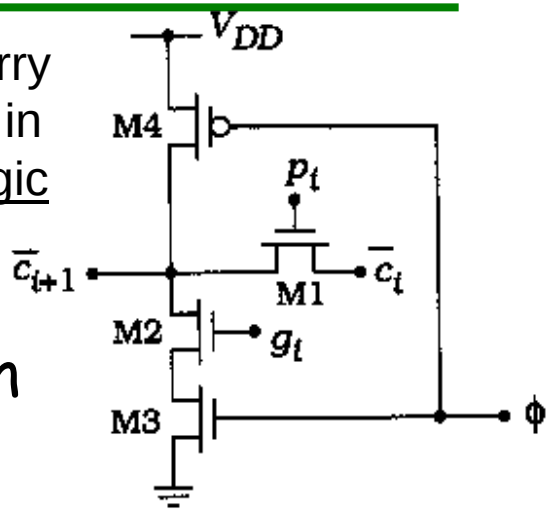
a_i	b_i	$\overline{c_i}$	$\overline{c_{i+1}}$	p_i	g_i
0	0	1	1	0	0
0	1	1	1	1	0
1	0	1	1	1	0
1	1	1	0	0	1
0	0	0	1	0	0
0	1	0	0	1	0
1	0	0	0	1	0
1	1	0	0	0	1

alternative design:
- do not add pMOS M3
- make W of M1 significantly larger than W of M4
→ C_i will override VDD

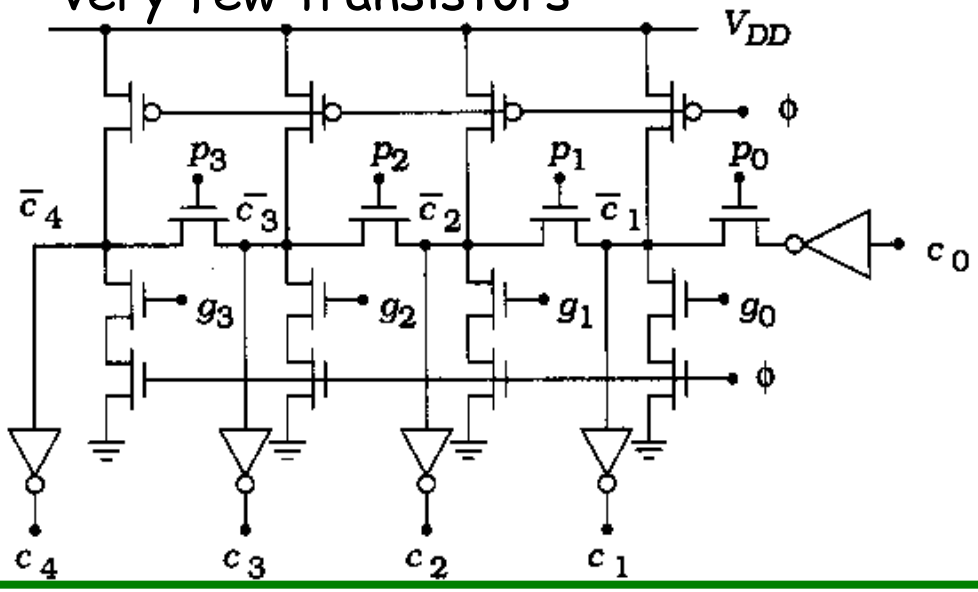
Manchester Implementation

- Dynamic Logic Circuit
  - evaluate when $\phi = 1$
  - c_{i+1} stays high unless
    - $g_i = 1$ ($c_{i+1} \rightarrow 0$) or $p_i = 1$ ($c_{i+1} \rightarrow c_i$)

single bit carry generation in dynamic logic



- 4b Dynamic Manchester Carry Generation
  - minor ripple delay
  - threshold drop on propagate
  - very few transistors

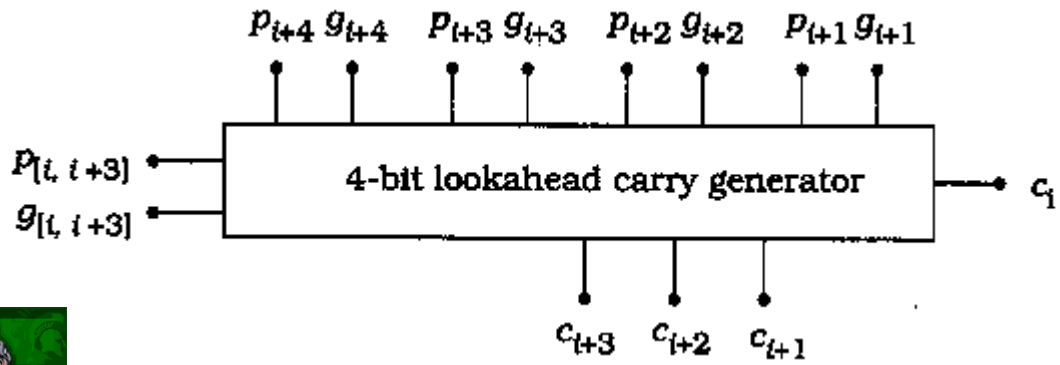


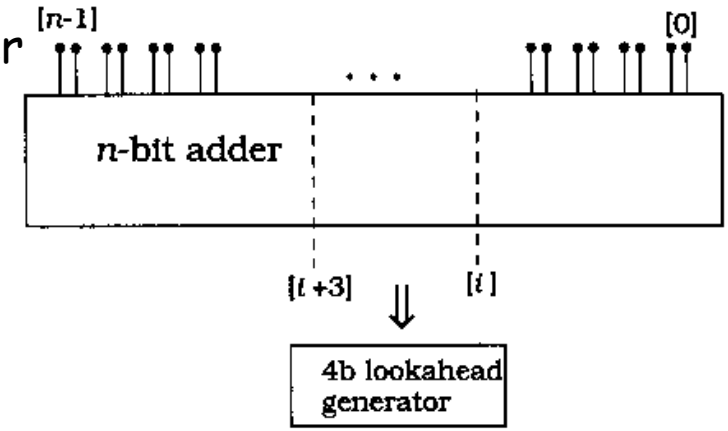
internal output, c_{i+1}
dynamically pulled high
propagate
pulled low (generate)

a_i	b_i	$\overline{c_i}$	$\overline{c_{i+1}}$	p_i	g_i
0	0	1	1	0	0
0	1	1	1	1	0
1	0	1	1	1	0
1	1	1	0	0	1
0	0	0	1	0	0
0	1	0	0	1	0
1	0	0	0	1	0
1	1	0	0	0	1

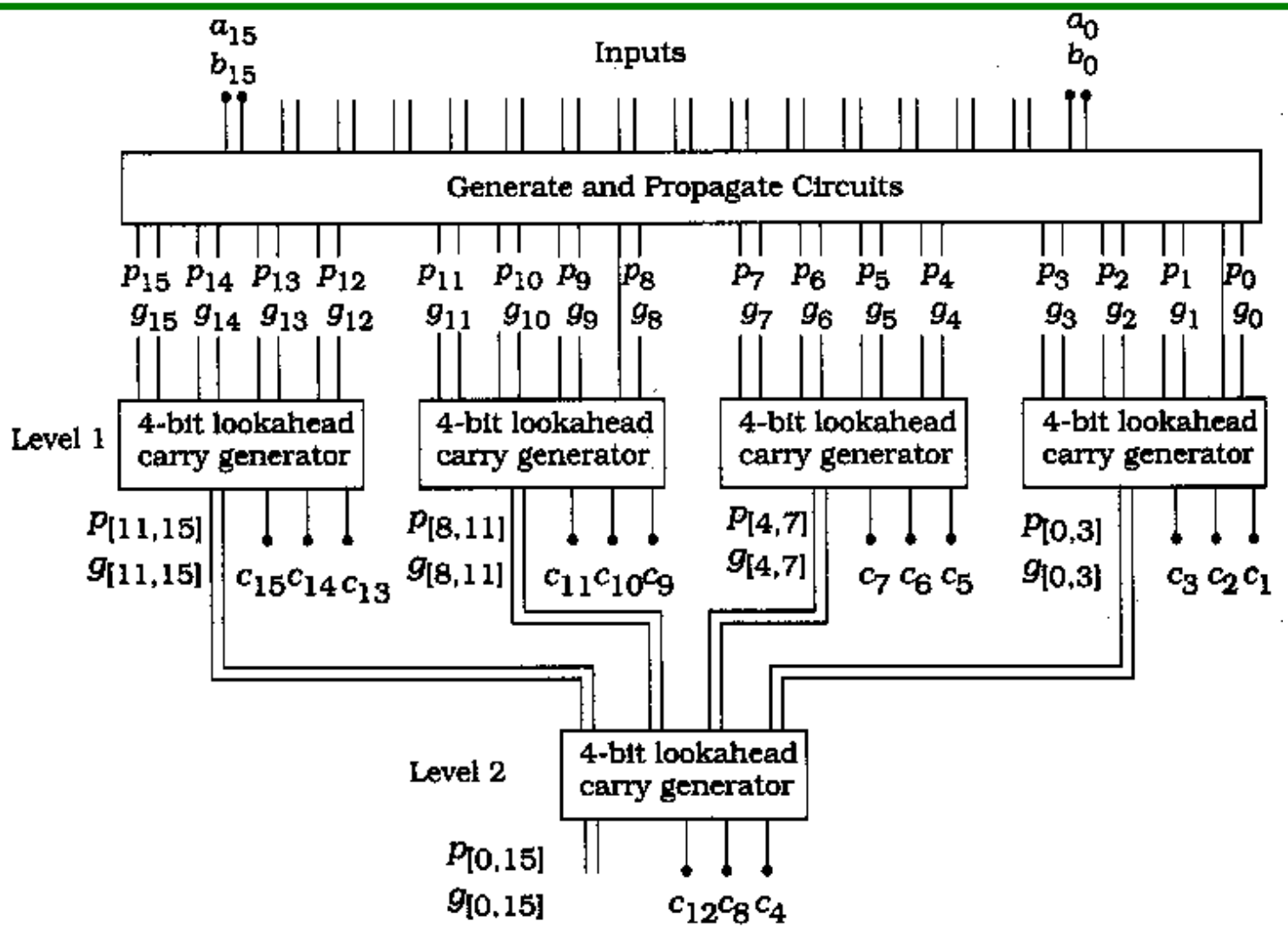
CLA for Wide Words

- number of terms in the carry equation increases with the width of the binary word to be added
  - gets overwhelming (and slow) with large binary words
- one method is to break wide adders into smaller blocks
  - e.g., use 4b blocks (4b is common, but could be any number)
  - must create **block** generate and propagate signals to carry information to the next block
    - $g_{[i,i+3]} = g_{i+3} + g_{i+2} \bullet p_{i+3} + g_{i+1} \bullet p_{i+2} \bullet p_{i+3} + g_i \bullet p_{i+1} \bullet p_{i+2} \bullet p_{i+3}$
    - $p_{[i,i+3]} = p_i \bullet p_{i+1} \bullet p_{i+2} \bullet p_{i+3}$
    - for block i thru i+3 of an n-sized adder





16b Adder Using 4b CLA Blocks



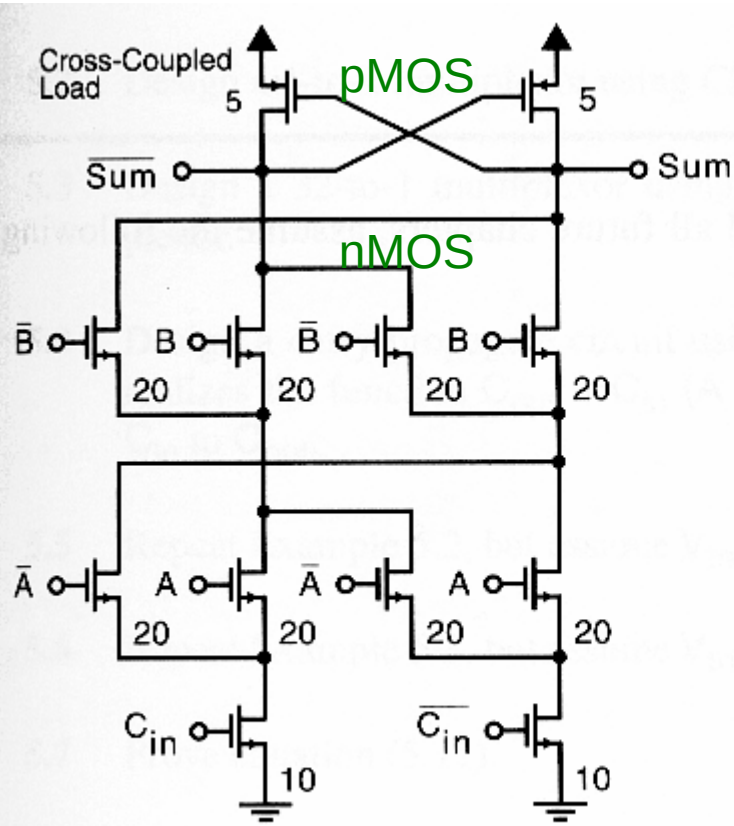
- Create SUMs from outputs of this circuit

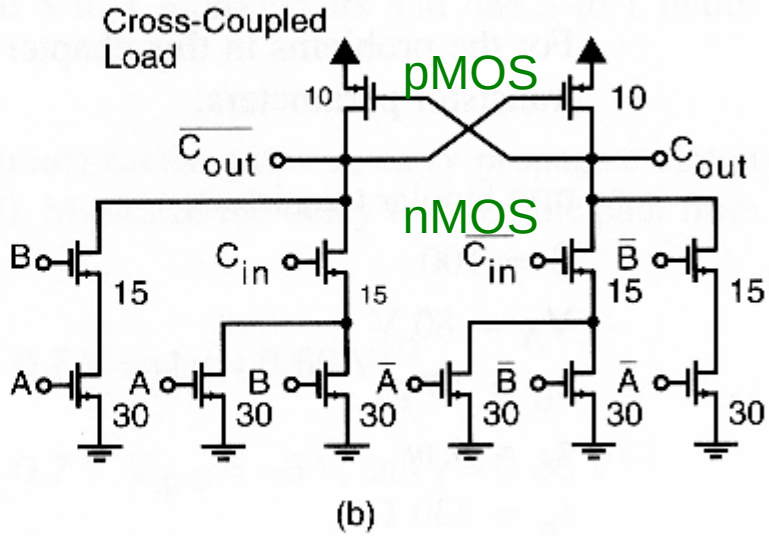
Other Adder Implementations

- Alternative implementations for high-speed adders
- Carry-Skip Adder
  - quickly generate a carry under certain conditions and skip the carry-generation block
    - recall $c_{i+1} = g_i + c_i \bullet p_i$, $g_i = a_i \cdot b_i$, $p_i = a_i \oplus b_i$
    - note generation of p_i is more complex (XOR) than g_i (AND)
      - so, generate p_i and check $c_i p_i$ case, skip g_i generation if $c_i p_i = 1$
- Carry-Select Adder
  - uses multiple adder blocks to increase speed
  - take a lot of chip area
- Carry-Save Adder
  - parallel FA, 3 inputs and 2 outputs
  - does not add carry-out to next bit (thus no ripple)
    - carry is saved for use by other blocks
  - useful for adding more than 2 numbers

Fully Differential Full Adder

- (a) sum-generate circuit
- (b) carry generate circuit

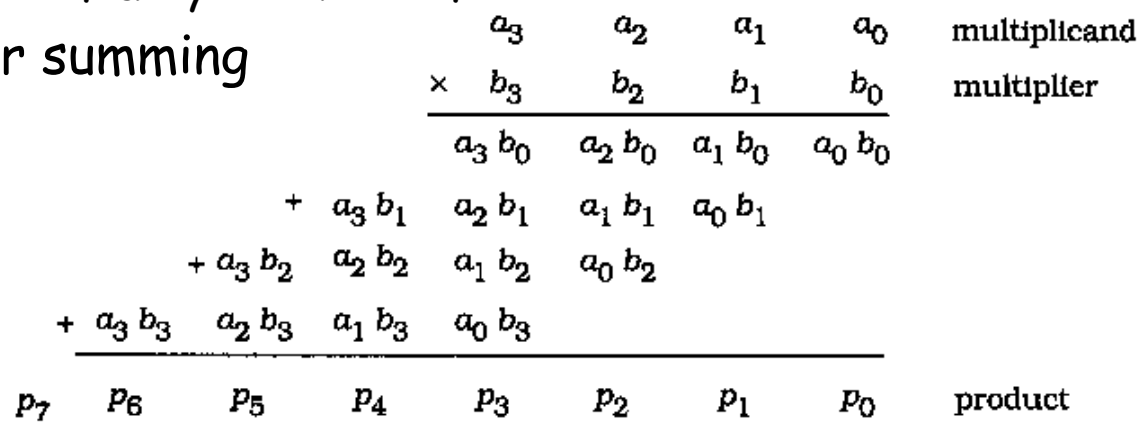




Multiplier Basics

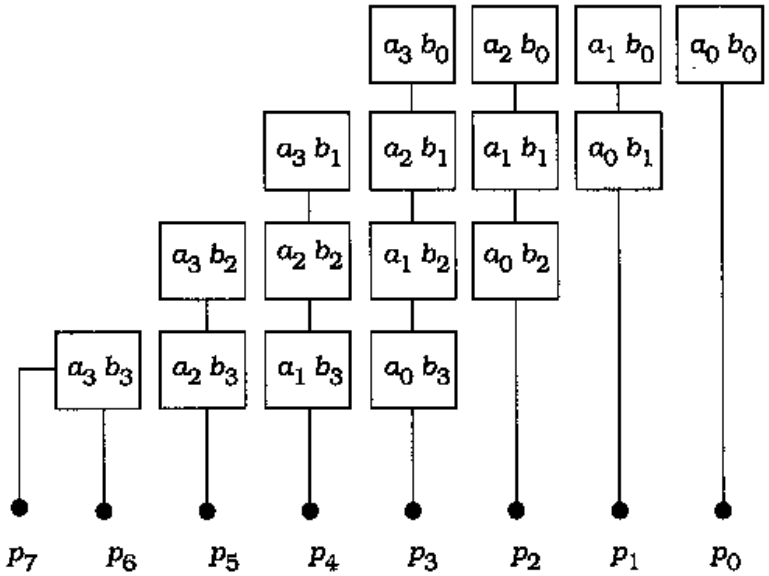
- Single-Bit Binary Multiplication
  - 0 x 0 = 0, 0 x 1 = 0, 1 x 0 = 0, 1 x 1 = 1
  - same result as logic AND operation (1b output, no carry!)
- Multiple-Bit Multiplication
  - n-bit word TIMES an n-bit word give 2n-bit product
  - 4b example
    - 16 single-bit multiplies
      - multiply each bit of ***a*** by each bit of ***b***
    - shift products for summing

note: can multiply by 2 by shifting the word to the left by one, multiply by 4 by left-shift twice, 8 three times, etc.

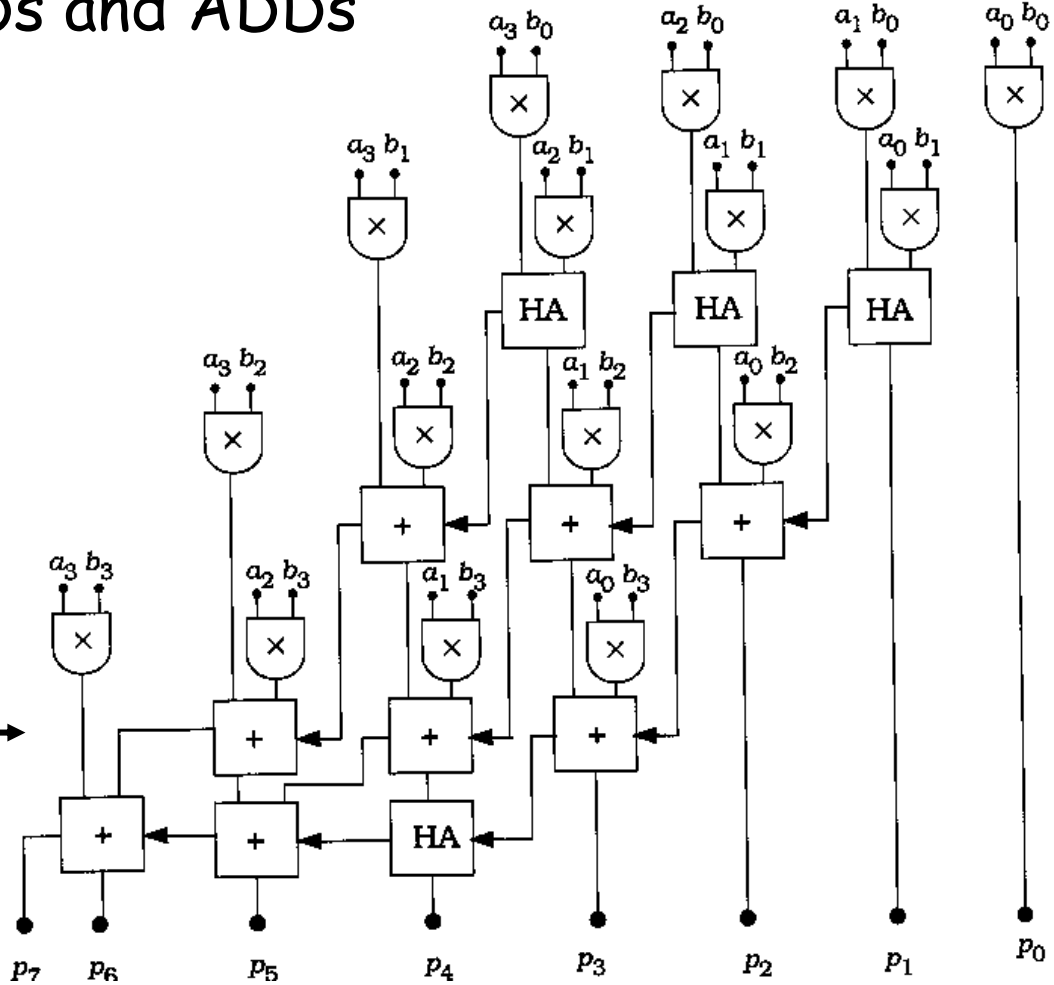


Implementing Multiplier Circuits

- Multiplication Sequence
  - organization of ANDs and ADDs



- 4x4 Array Multiplier Circuit
  - 8b output



Signed Multiplication: Booth Encoding

- Signed Numbers
  - 2's complement

$+m = m$

$-m = 2 - m$

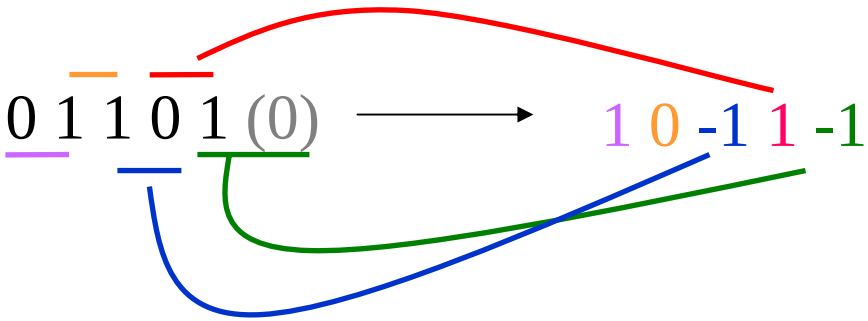
Ex: 3-bit signed numbers

3 = 0 1 1

2+3 = 5 = 2- (-3)

=> 1 0 1 = -3

- Booth Encoding
  - evaluate number 2-bits at a time
  - generate 'action' based on 2-bit sequence



-1: a sequence of " 1 0"

0: no change in sequence

1: a sequence of " 0 1"

$0\ 1\ 1\ 0\ 1 = (13)_{10} = [\ 1*2^4 + 0*2^3 - 1*2^2 + 1*2^1 - 1*2^0]$

Benefit: Number of shift-add reduces if long seq. of "1" or "0"

4b x 4b Booth Multiplication

Multiply m x r:

$A = m_3\ m_2\ m_1\ m_0\ \ x\ \ r_3\ r_2\ r_1\ r_0$

Ex : 0 1 0 1 x 1 0 1 1 (5* -5 = - 25)

n	r_n	r_{n-1}	*Action*	*A = 0 0 0 0*
0	1	0	Sub M (A-M)	1 0 1 1
			Shift Rt	1 1 0 1 1
1	1	1	Shift Rt	1 1 1 0 1 1
2	0	1	Add M (A+M)	0 0 1 1 1 1
			Shift Rt	0 0 0 1 1 1 1
3	1	0	Sub M (A-M)	1 1 0 0 1 1 1 =-25

Rules:

Start with product A = 0

Examine r_n, r_{n-1}

0 0 : shift R right

0 1 : add M (A+M)
shift A right

1 0 : sub M (A-M)
shift A right

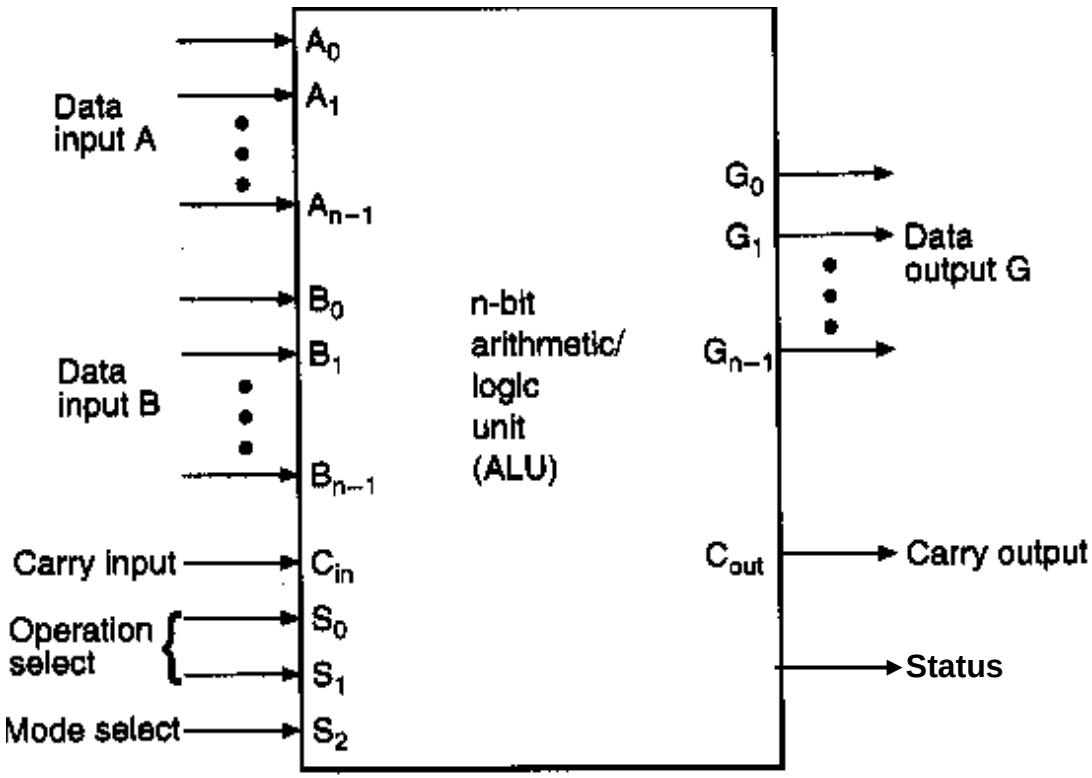
1 1 : shift A right

```
  1 1 1 0 1 1
  0 1 0 1
-------------
  0 0 1 1 1 1

  0 0 0 1 1 1 1
  1 0 1 1
-------------
  1 1 0 0 1 1 1
```

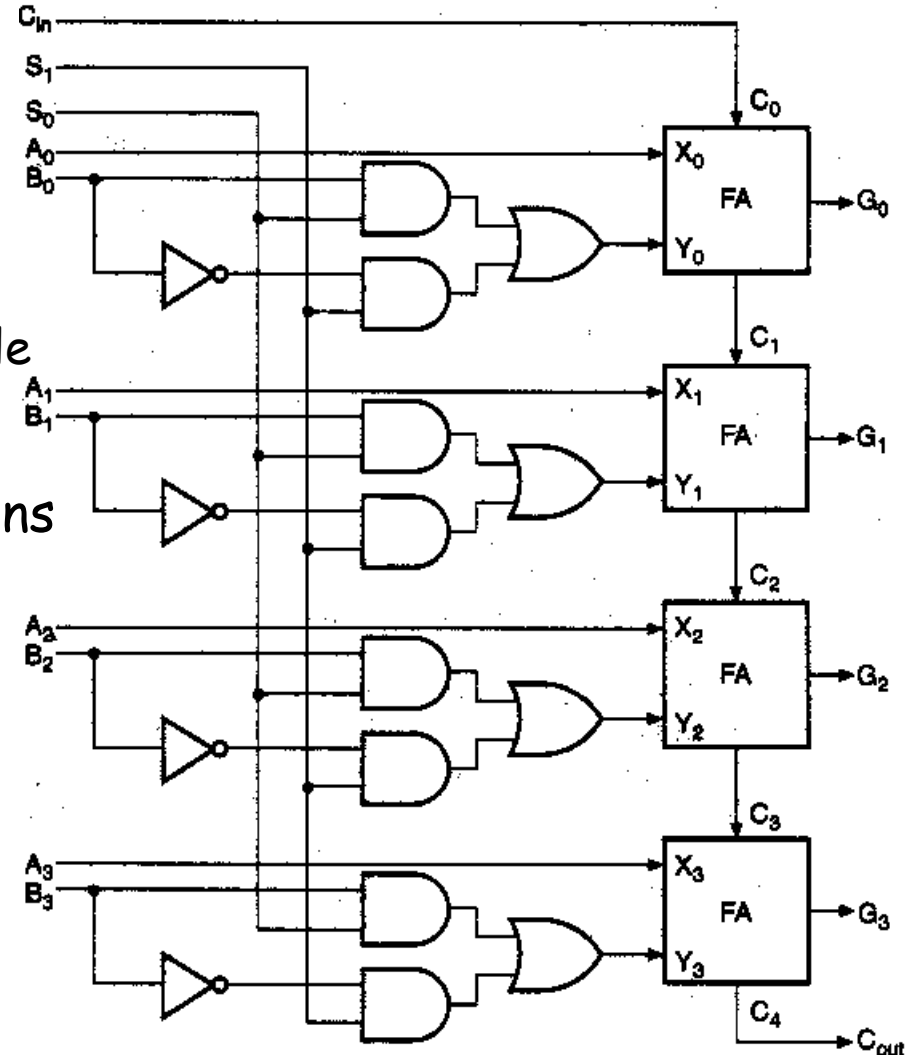
Arithmetic/Logic Unit Structure

- ALU performs basic arithmetic and logic functions in a single block
  - core unit in a microprocessor
- Basic n-bit ALU
  - Inputs
    - 2 n-bit inputs
    - carry in
    - function selects
  - Outputs
    - 1 n-bit result
    - carry out
    - status outputs
      - minus, zero, etc.



ALU Arithmetic Components

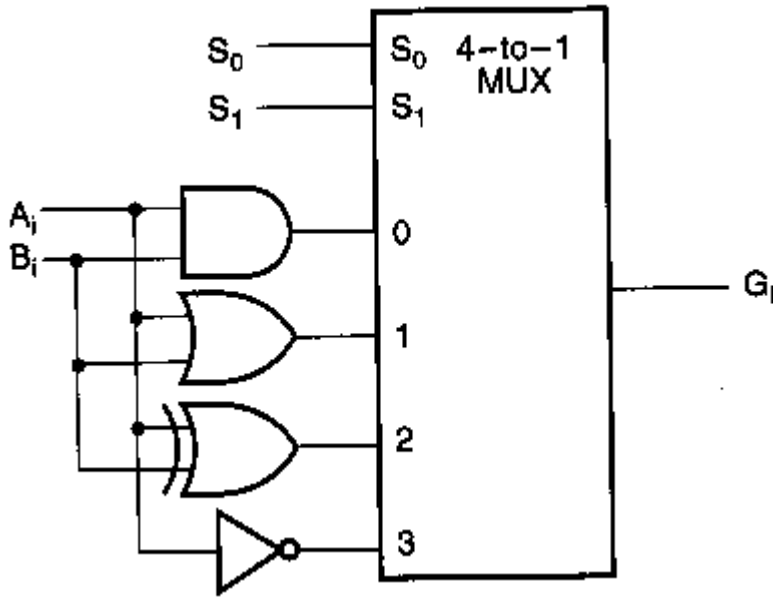
- ALU Components
  - Arithmetic Block
  - Logic Block
  - Date Movement
    - sometimes done in register file
- Arithmetic Block
  - implements arithmetic functions
  - add
  - subtract
  - increment/decrement
  - sometimes
    - multiply
    - divide



Example 4b Arithmetic Block

ALU Logic Components

- Logic Block
  - implements logic functions
  - NOT
  - AND
  - OR
  - XOR
- Date Movement
  - somewhere in the ALU
    - or in the register file
  - shift
  - rotate

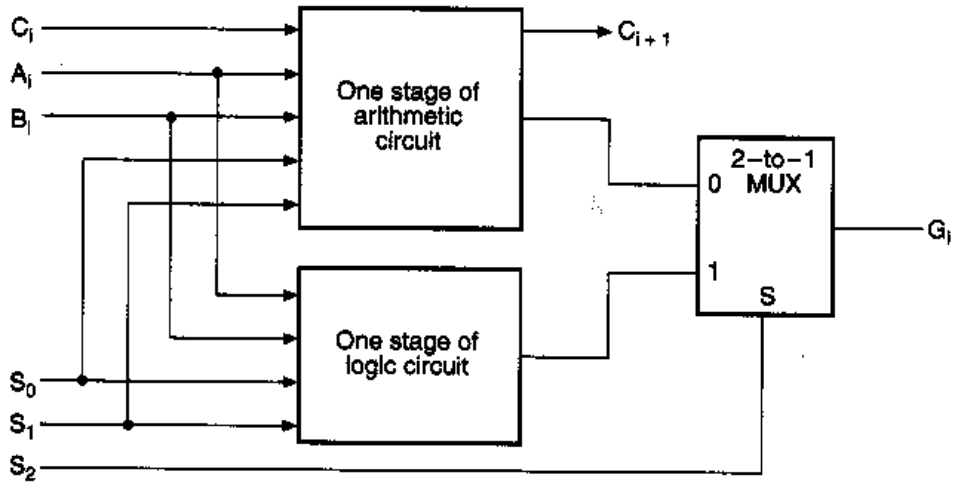


S_1	S_0	Output	Operation
0	0	$G = A \wedge B$	AND
0	1	$G = A \vee B$	OR
1	0	$G = A \oplus B$	XOR
1	1	$G = \overline{A}$	NOT

Example 1-bit Logic Block

Example ALU Organization & Function

- Example ALU Bit Slice
  - implementation of one bit



- Example Function Table

Operation Select					
S_2	S_1	S_0	C_{in}	Operation	Function
0	0	0	0	$G = A$	Transfer A
0	0	0	1	$G = A + 1$	Increment A
0	0	1	0	$G = A + B$	Addition
0	0	1	1	$G = A + B + 1$	Add with carry input of 1
0	1	0	0	$G = A + \overline{B}$	A plus 1's complement of B
0	1	0	1	$G = A + \overline{B} + 1$	Subtraction
0	1	1	0	$G = A - 1$	Decrement A
0	1	1	1	$G = A$	Transfer A
1	0	0	X	$G = A \wedge B$	AND
1	0	1	X	$G = A \vee B$	OR
1	1	0	X	$G = A \oplus B$	XOR
1	1	1	X	$G = \overline{A}$	NOT (1's complement)

function set for a simple ALU

function determined by select inputs