Chapter # 5: Arithmetic Circuits

Contemporary Logic Design

Randy H. Katz
University of California, Berkeley

June 1993

Motivation

Arithmetic circuits are excellent examples of comb. logic design

- ***Time vs. Space Trade-offs***

  Doing things fast requires more logic and thus more space

  Example: carry lookahead logic

- ***Arithmetic Logic Units***

  Critical component of processor datapath

  Inner-most "loop" of most computer instructions

Chapter Overview

- ***Binary Number Representation***

  Sign & Magnitude, Ones Complement, Twos Complement

- ***Binary Addition***

  Full Adder Revisited

- ***ALU Design***

- ***BCD Circuits***

- ***Combinational Multiplier Circuit***

- ***Design Case Study: 8 Bit Multiplier***

Number Systems

Representation of Negative Numbers

Representation of positive numbers same in most systems

Major differences are in how negative numbers are represented

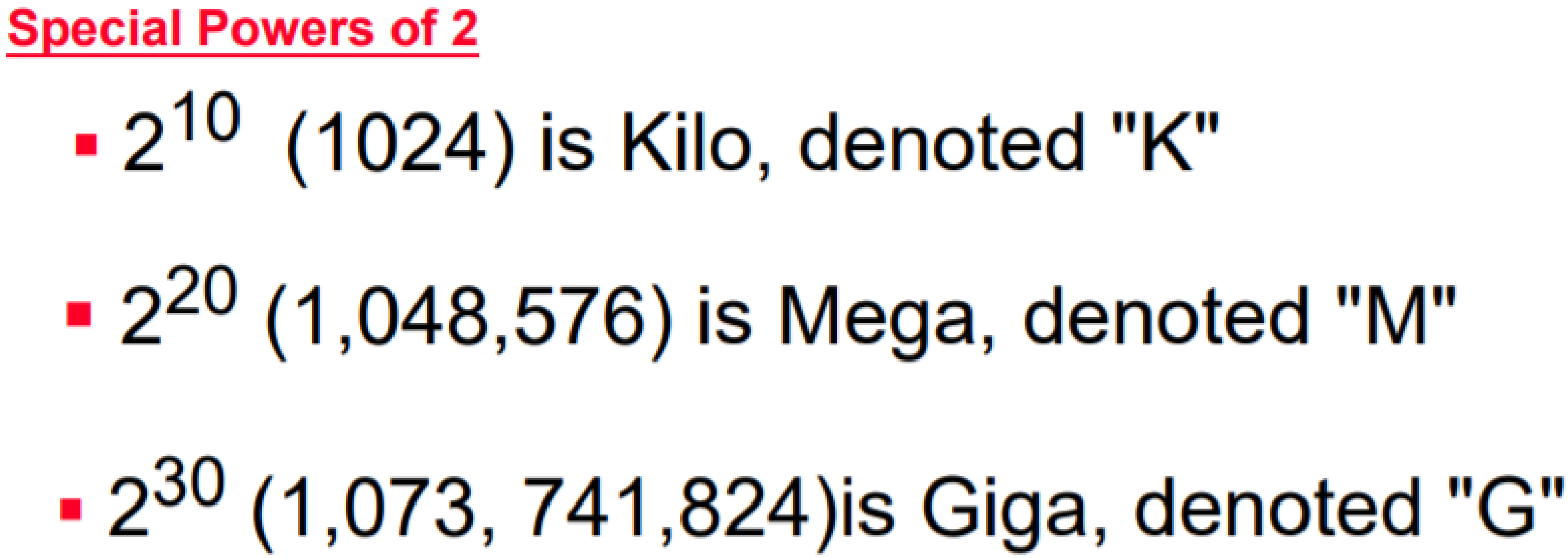
Three major schemes:

- sign and magnitude
- ones complement
- twos complement

Assumptions:

- we'll assume a 4 bit machine word
- 16 different values can be represented
- roughly half are positive, half are negative

Number Systems



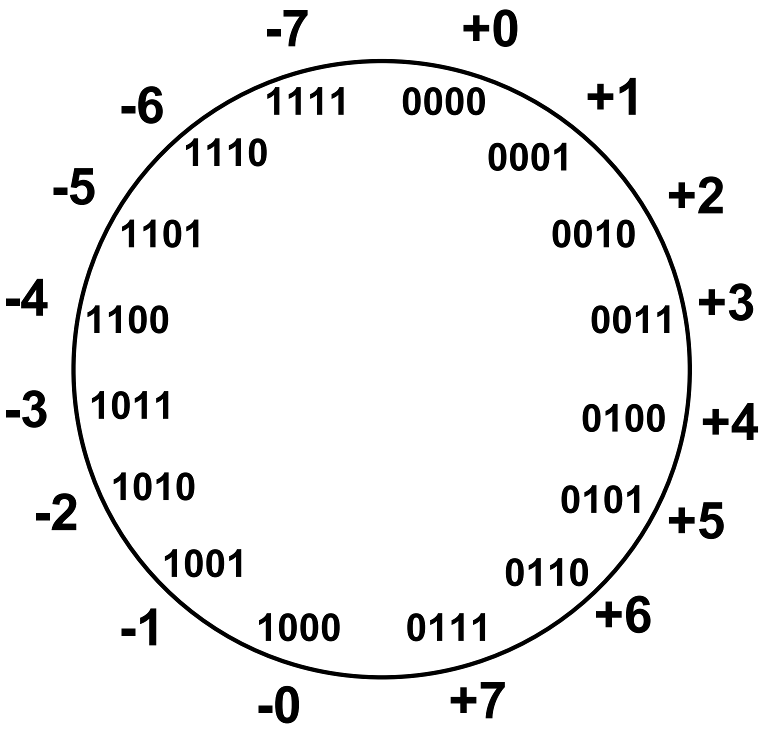
Special Powers of 2

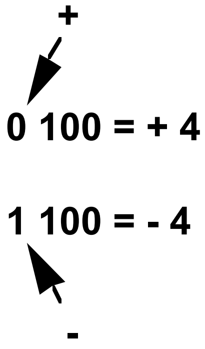
- 2^{10} (1024) is Kilo, denoted "K"
- 2^{20} (1,048,576) is Mega, denoted "M"
- 2^{30} (1,073, 741,824)is Giga, denoted "G"

- http://myweb.utaipei.edu.tw/~cyang/class/Digital_System/ch_01.pdf

Number Systems

Sign and Magnitude Representation




+

0 100 = + 4

1 100 = - 4

-

High order bit is sign: 0 = positive (or zero), 1 = negative

Three low order bits is the magnitude: 0 (000) thru 7 (111)

Number range for n bits = $+/-2^{n-1} -1$

Representations for 0

Number Systems

Sign and Magnitude

Cumbersome addition/subtraction

Must compare magnitudes to determine sign of result

Ones Complement

N is positive number, then $\overline{N}$ is its negative 1's complement

$$\overline{N} = (2^n - 1) - N$$

Example: 1's complement of 7

$$\begin{array}{rr} 2^4 = & 10000 \\ -1 = & \underline{00001} \\ & 1111 \\ -7 = & \underline{0111} \\ & 1000 \end{array} \quad = -7 \text{ in 1's comp.}$$

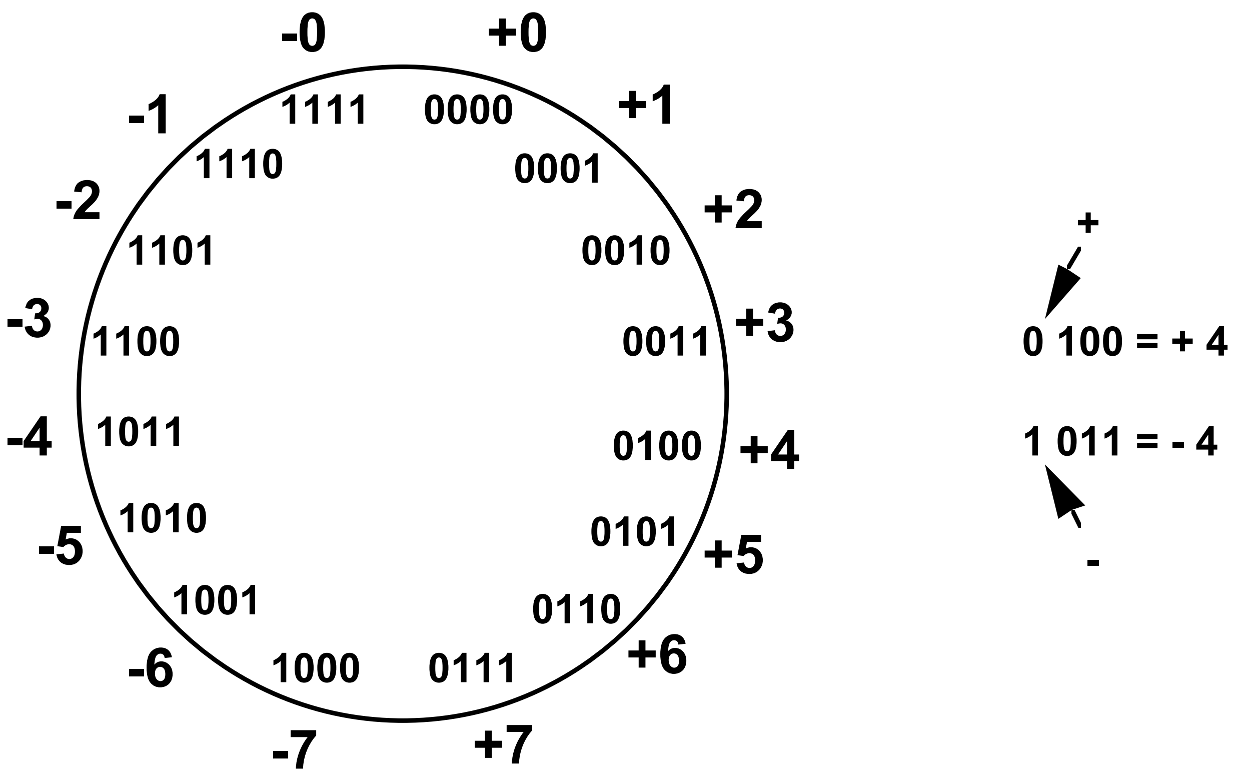
Shortcut method:

simply compute bit wise complement

0111 -> 1000

Number Systems

Ones Complement



Subtraction implemented by addition & 1's complement

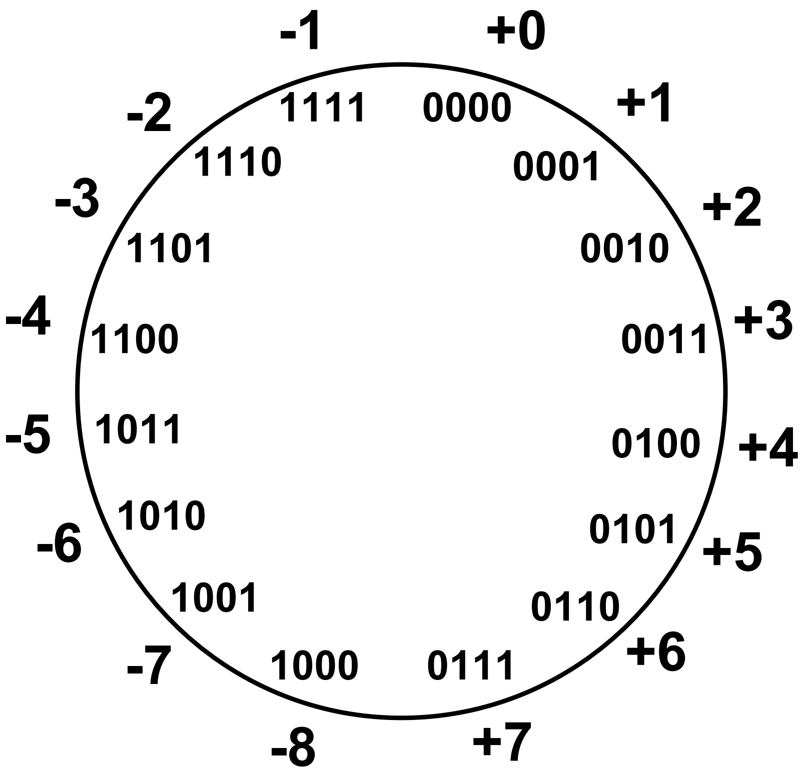
Still two representations of 0! This causes some problems

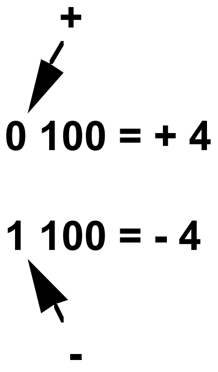
Some complexities in addition

Number Representations

Twos Complement

like 1's comp except shifted one position clockwise





Only one representation for 0

One more negative number than positive number

Number Systems

Twos Complement Numbers

$N^* = 2^n - N$

Example: Twos complement of 7

$$\begin{array}{rrl} 2^4 & = & 10000 \\ \text{sub } 7 & = & \underline{0111} \\ & & 1001 = \text{repr. of } -7 \end{array}$$

Example: Twos complement of -7

$$\begin{array}{rrl} 2^4 & = & 10000 \\ \text{sub } -7 & = & \underline{1001} \\ & & 0111 = \text{repr. of } 7 \end{array}$$

Shortcut method:

Twos complement = bitwise complement + 1

0111 -> 1000 + 1 -> 1001 (representation of -7)

1001 -> 0110 + 1 -> 0111 (representation of 7)

Number Representations

Addition and Subtraction of Numbers

Sign and Magnitude

result sign bit is the same as the operands' sign

$$\begin{array}{rr} 4 & 0100 \\ +3 & 0011 \\ \hline 7 & 0111 \end{array} \qquad \begin{array}{rr} -4 & 1100 \\ +(-3) & 1011 \\ \hline -7 & 1111 \end{array}$$

when signs differ, operation is subtract, sign of result depends on sign of number with the larger magnitude

$$\begin{array}{rr} 4 & 0100 \\ -3 & 1011 \\ \hline 1 & 0001 \end{array} \qquad \begin{array}{rr} -4 & 1100 \\ +3 & 0011 \\ \hline -1 & 1001 \end{array}$$

Number Systems

Addition and Subtraction of Numbers

Ones Complement Calculations

$$\begin{array}{rr} 4 & 0100 \\ +3 & 0011 \\ \hline 7 & 0111 \end{array}$$

$$\begin{array}{rr} -4 & 1011 \\ +(-3) & 1100 \\ \hline -7 & 10111 \\ \text{End around carry} & \hookrightarrow 1 \\ \hline & 1000 \end{array}$$

$$\begin{array}{rr} 4 & 0100 \\ -3 & 1100 \\ \hline 1 & 10000 \\ \text{End around carry} & \hookrightarrow 1 \\ \hline & 0001 \end{array}$$

$$\begin{array}{rr} -4 & 1011 \\ +3 & 0011 \\ \hline -1 & 1110 \end{array}$$

Addition and Subtraction of Binary Numbers

Ones Complement Calculations

Why does end-around carry work?

Its equivalent to subtracting 2^n and adding 1

$$M - N = M + \overline{N} = M + (2^n - 1 - N) = (M - N) + 2^n - 1 \qquad (M > N)$$

$$-M + (-N) = \overline{M} + \overline{N} = (2^n - M - 1) + (2^n - N - 1) \qquad M + N < 2^{n-1}$$

$$= 2^n + [2^n - 1 - (M + N)] - 1$$

after end around carry:

$$= 2^n - 1 - (M + N)$$

this is the correct form for representing $-(M + N)$ in 1's comp!

Number Systems

Addition and Subtraction of Binary Numbers

Twos Complement Calculations

4	0100	-4	1100
+ 3	0011	+ (-3)	1101
7	0111	-7	11001

If carry-in to sign = carry-out then ignore carry

if carry-in differs from carry-out then overflow

4	0100	-4	1100
- 3	1101	+ 3	0011
1	10001	-1	1111

Simpler addition scheme makes twos complement the most common choice for integer number systems within digital systems

Number Systems

Addition and Subtraction of Binary Numbers

Twos Complement Calculations

Why can the carry-out be ignored?

$-M + N$ when $N > M$:

$$M^* + N = (2^n - M) + N = 2^n + (N - M)$$

Ignoring carry-out is just like subtracting 2^n

$-M + -N$ where $N + M <$ or $= 2^{n-1}$

$$-M + (-N) = M^* + N^* = (2^n - M) + (2^n - N)$$

$$= 2^n - (M + N) + 2^n$$

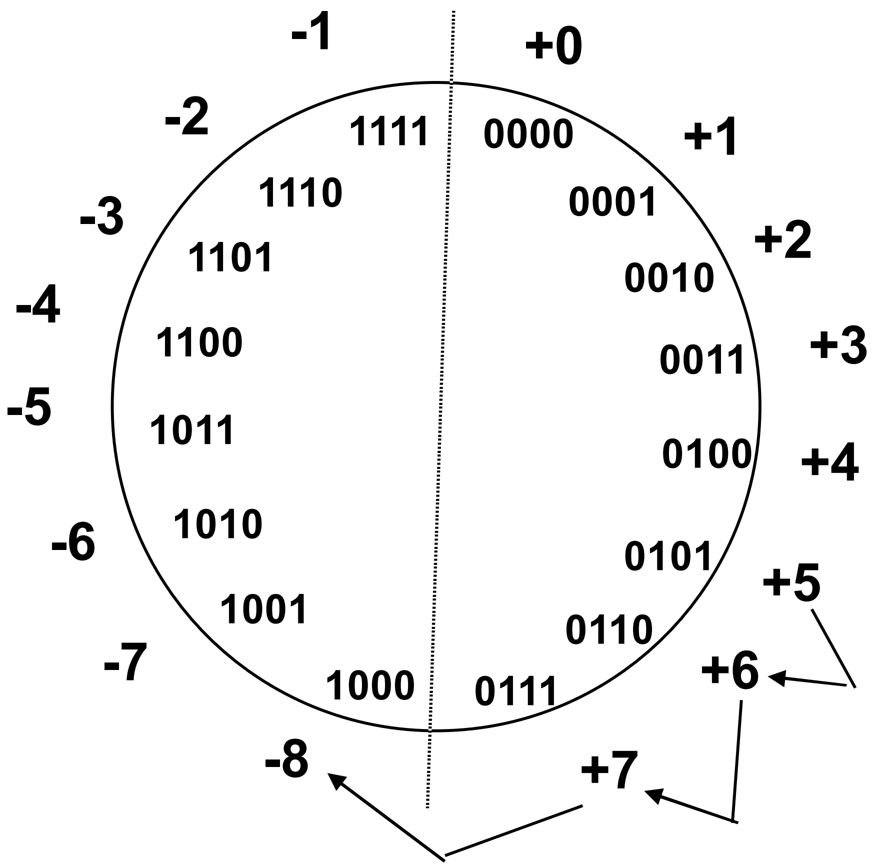
After ignoring the carry, this is just the right twos compl. representation for $-(M + N)$!

Number Systems

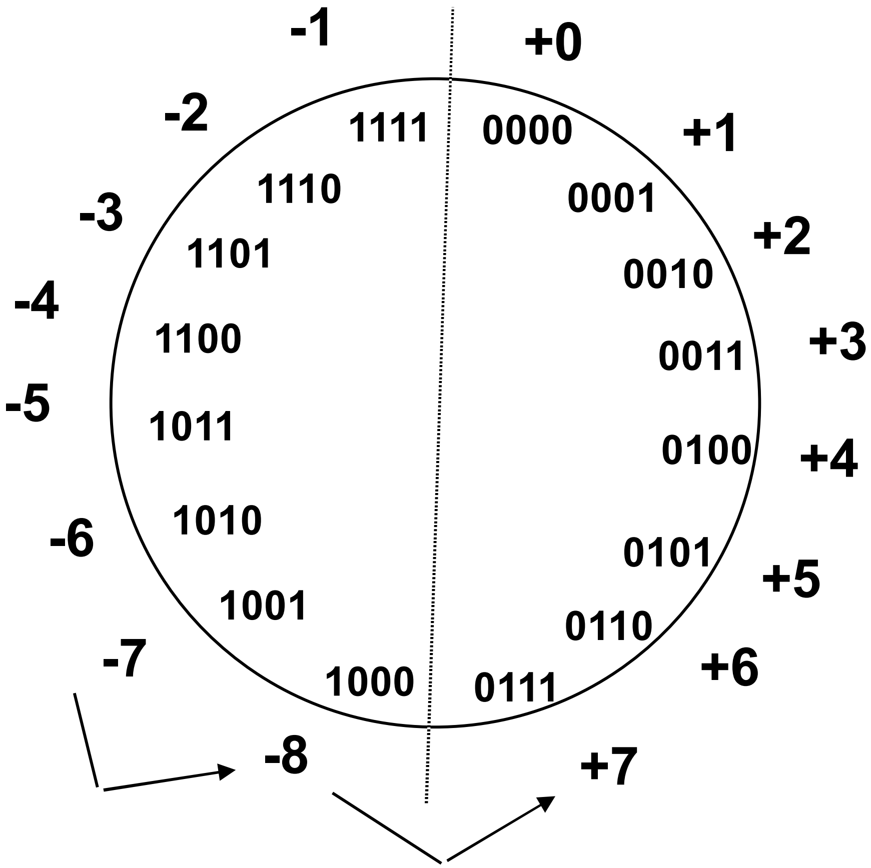
Overflow Conditions

Add two positive numbers to get a negative number

or two negative numbers to get a positive number



5 + 3 = -9



-7 - 2 = +7

Number Systems

Overflow Conditions

```
            0 1 1 1
 5           0 1 0 1

 3           0 0 1 1
----        ---------
-8           1 0 0 0
```

Overflow

```
             1 0 0 0
-7             1 0 0 1

-2             1 1 0 0
----        ----------
 7           1|0 1 1 1
```

Overflow

```
             0 0 0 0
 5             0 1 0 1

 2             0 0 1 0
----        -----------
 7             0 1 1 1
```

No overflow

```
             1 1 1 1
-3             1 1 0 1

-5             1 0 1 1
----        -----------
-8            1|1 0 0 0
```

No overflow

Overflow when carry in to sign does not equal carry out

Number Systems

RADIX OR BASE:-

The radix or base of a number system is defined as the number of different digits which can occur in each position in the number system.

RADIX POINT :-

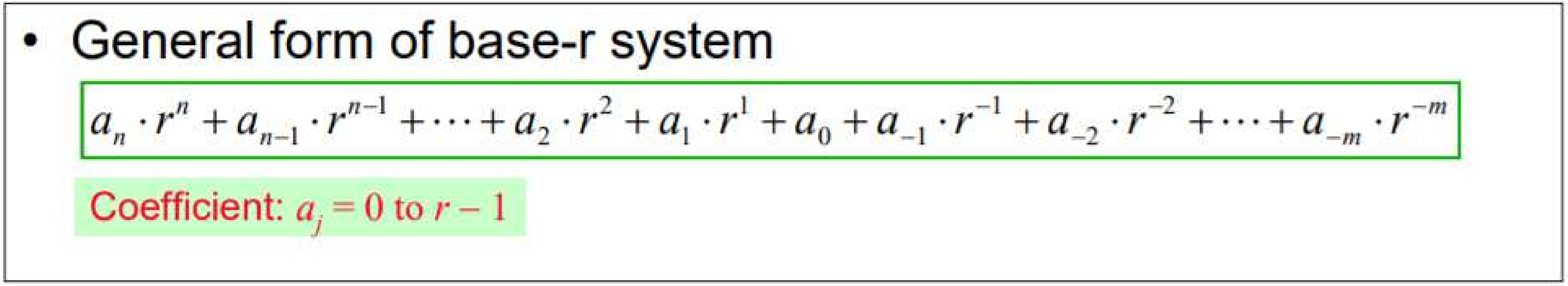
The generalized form of a decimal point is known as radix point. In any positional number system the radix point divides the integer and fractional part.

N_r = [Integer part . Fractional part]

↑

Radix point

- http://astorissa.in/Docs/Study_Materials/Electrical/4th_Semester/DIGITAL_ELECTRONICS.pdf

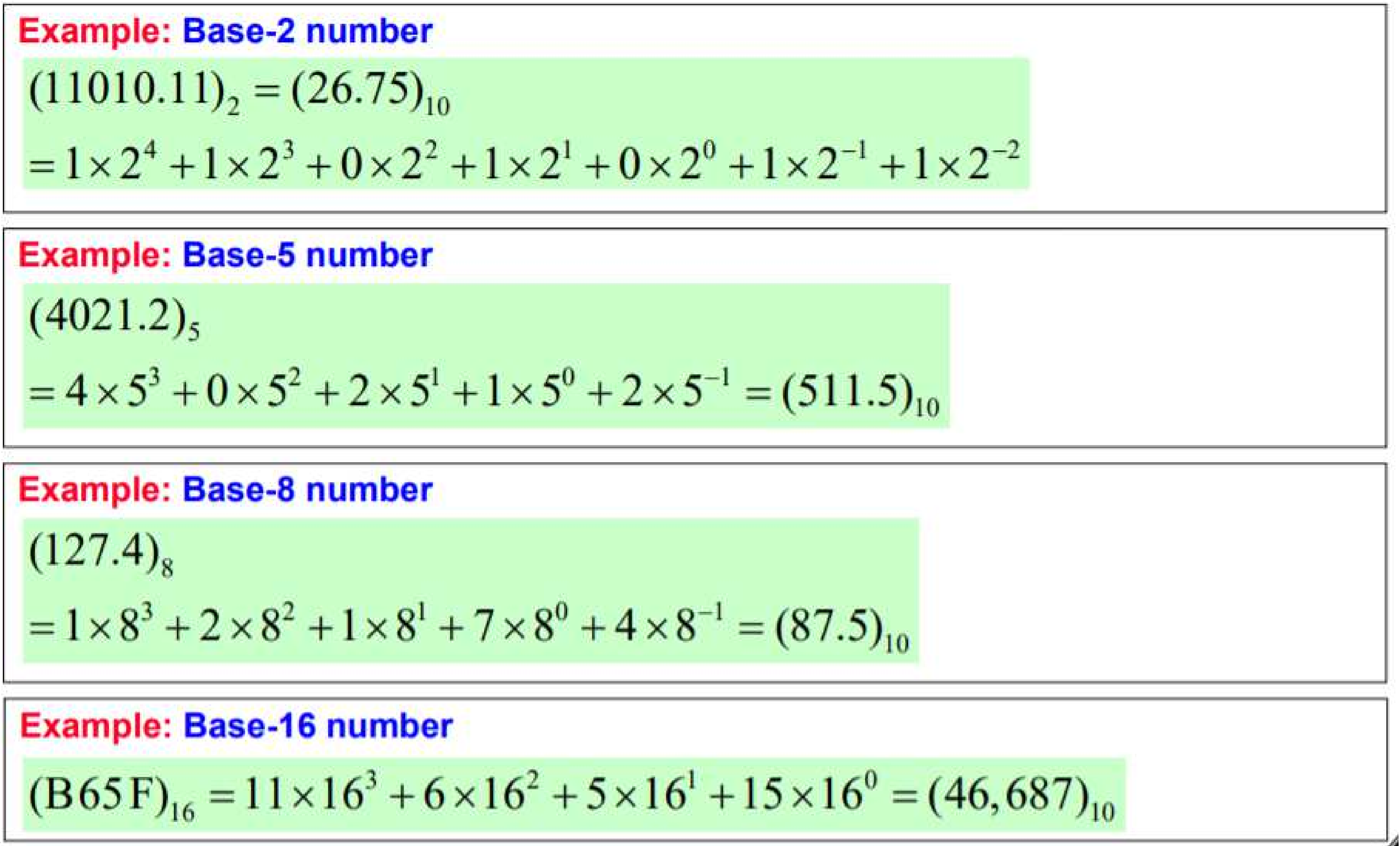


- General form of base-r system

$$a_n \cdot r^n + a_{n-1} \cdot r^{n-1} + \cdots + a_2 \cdot r^2 + a_1 \cdot r^1 + a_0 + a_{-1} \cdot r^{-1} + a_{-2} \cdot r^{-2} + \cdots + a_{-m} \cdot r^{-m}$$

Coefficient: $a_j = 0 \text{ to } r-1$

Number Systems



Example: Base-2 number

$$(11010.11)_2 = (26.75)_{10}$$
$$= 1\times 2^4 + 1\times 2^3 + 0\times 2^2 + 1\times 2^1 + 0\times 2^0 + 1\times 2^{-1} + 1\times 2^{-2}$$

Example: Base-5 number

$$(4021.2)_5$$
$$= 4\times 5^3 + 0\times 5^2 + 2\times 5^1 + 1\times 5^0 + 2\times 5^{-1} = (511.5)_{10}$$

Example: Base-8 number

$$(127.4)_8$$
$$= 1\times 8^3 + 2\times 8^2 + 1\times 8^1 + 7\times 8^0 + 4\times 8^{-1} = (87.5)_{10}$$

Example: Base-16 number

$$(B65F)_{16} = 11\times 16^3 + 6\times 16^2 + 5\times 16^1 + 15\times 16^0 = (46,687)_{10}$$

- http://myweb.utaipei.edu.tw/~cyang/class/Digital_System/ch_01.pdf

Networks for Binary Addition

Half Adder

With twos complement numbers, addition is sufficient

Ai	Bi	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

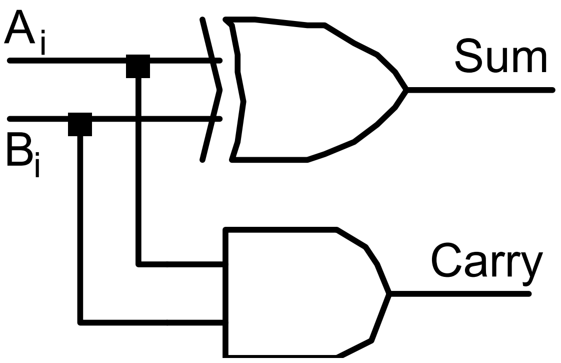
Bi \ Ai	0	1
0	0	1
1	1	0

Bi \ Ai	0	1
0	0	0
1	0	1

$$\text{Sum} = \overline{\text{Ai}}\ \text{Bi} + \text{Ai}\ \overline{\text{Bi}}$$

$$= \text{Ai} \oplus \text{Bi}$$

$$\text{Carry} = \text{Ai Bi}$$

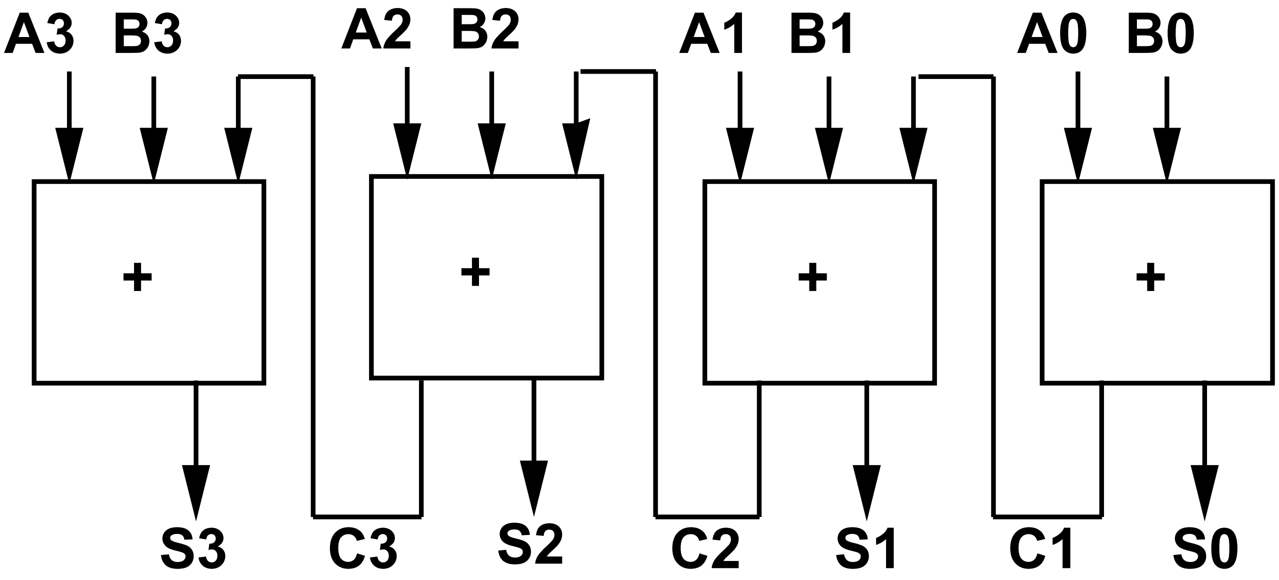


Half-adder Schematic

Networks for Binary Addition

Full Adder

Cascaded Multi-bit Adder



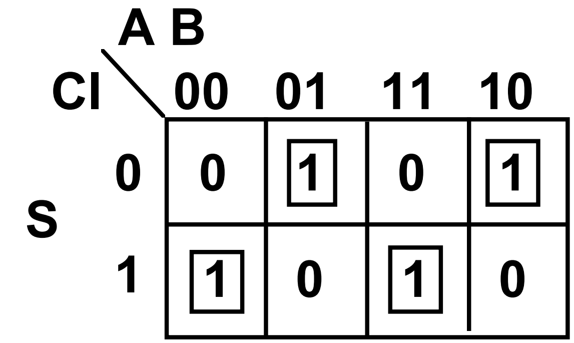
usually interested in adding more than two bits

this motivates the need for the full adder

Networks for Binary Addition

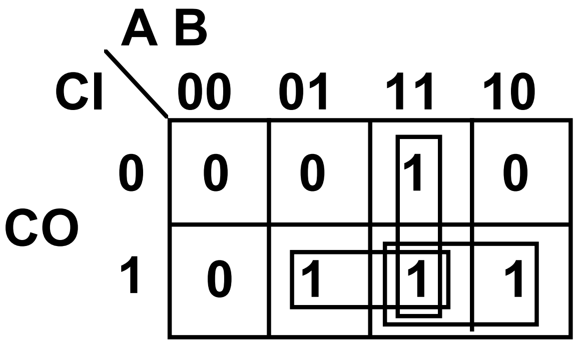
Full Adder

A	B	CI	S	CO
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



S

CI \ A B	00	01	11	10
0	0	1	0	1
1	1	0	1	0



CO

CI \ A B	00	01	11	10
0	0	0	1	0
1	0	1	1	1

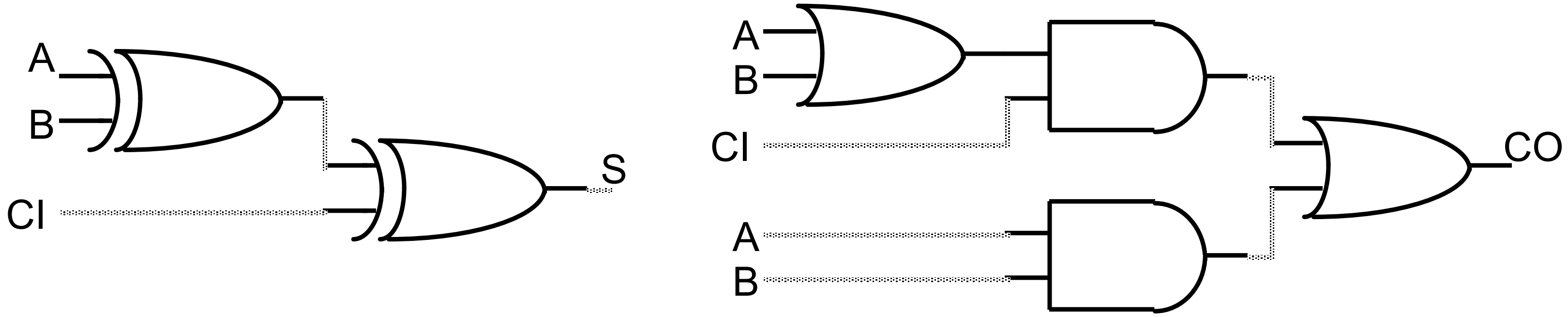
$$S = CI \text{ xor } A \text{ xor } B$$

$$CO = B\,CI + A\,CI + A\,B = CI\,(A + B) + A\,B$$

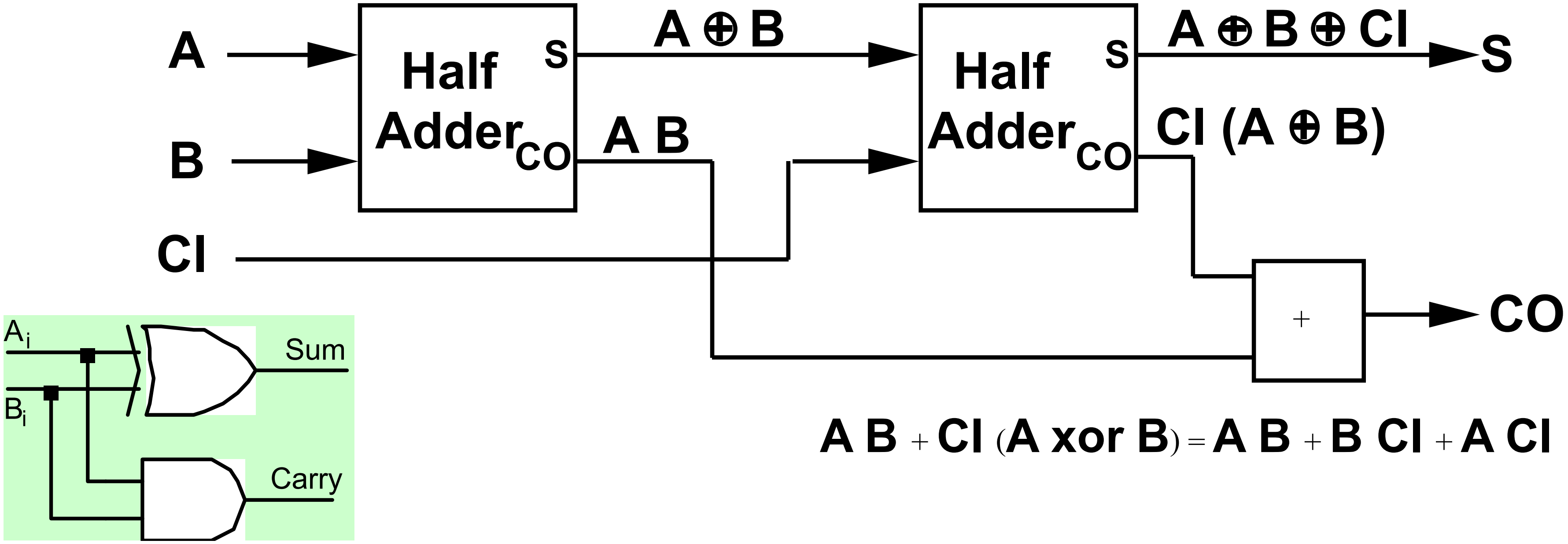
Networks for Binary Addition

Full Adder/Half Adder

Standard Approach: 6 Gates



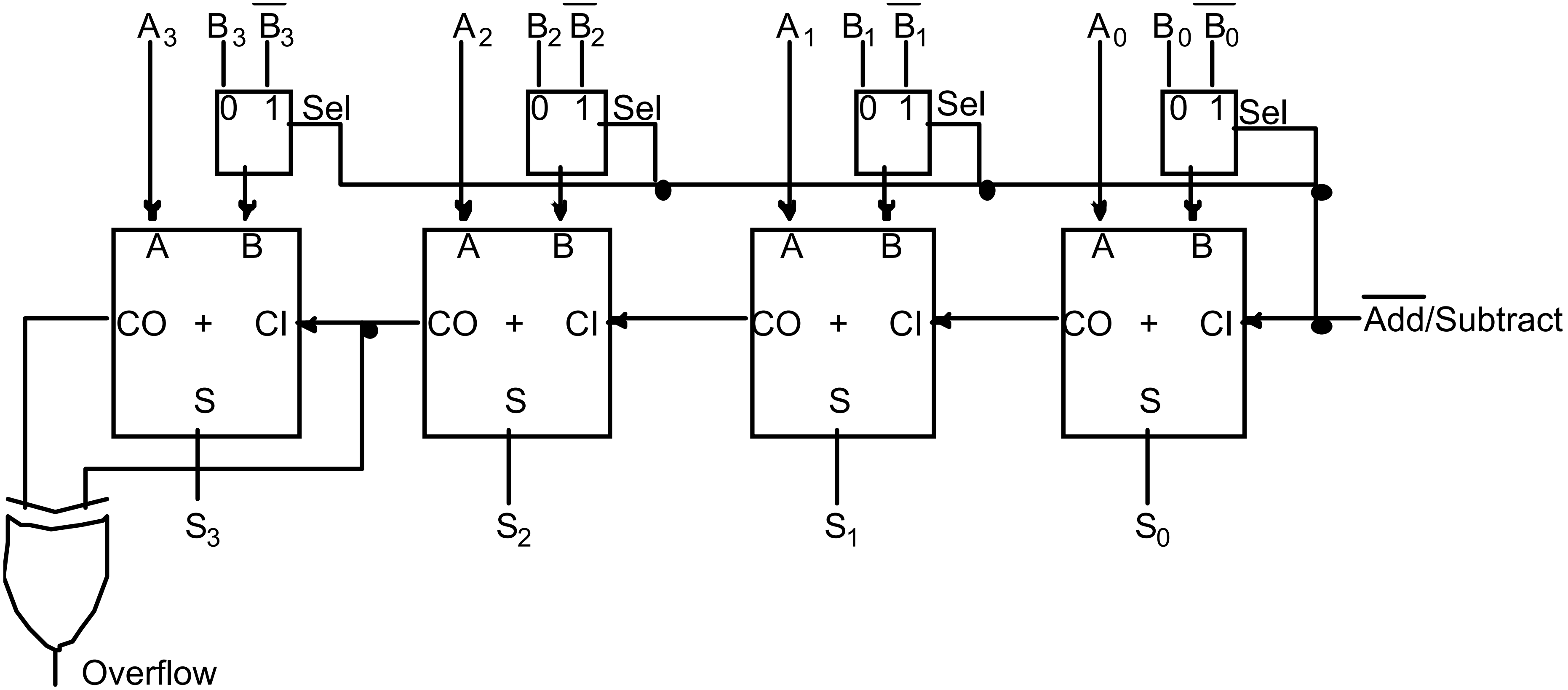
Alternative Implementation: 5 Gates



$$A\,B + CI\,(A \text{ xor } B) = A\,B + B\,CI + A\,CI$$

Networks for Binary Addition

Adder/Subtractor



$$A - B = A + (-B) = A + \overline{B} + 1$$

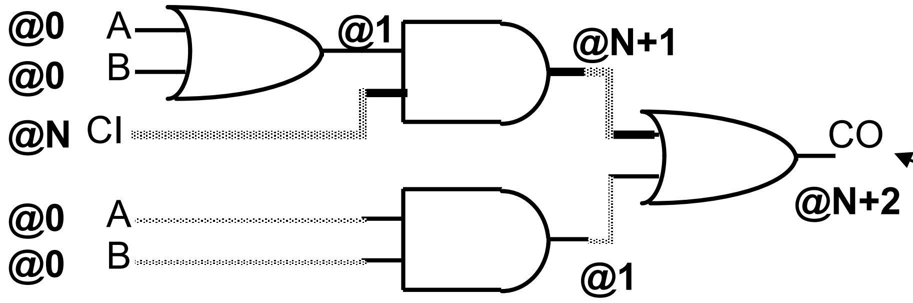
Subtraction in 2’s complement number systems

Networks for Binary Addition

Carry Lookahead Circuits

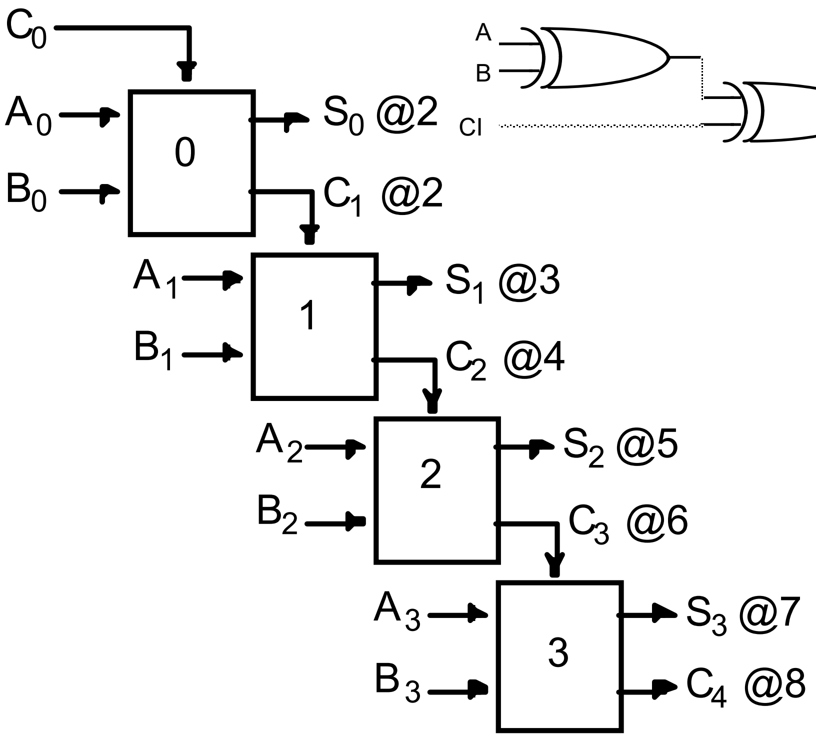
Critical delay: the propagation of carry from low to high order stages

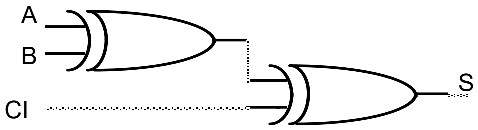
late arriving signal

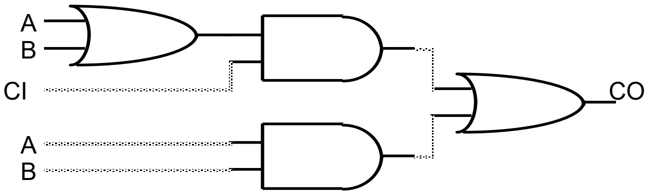


two gate delays to compute CO

4 stage adder







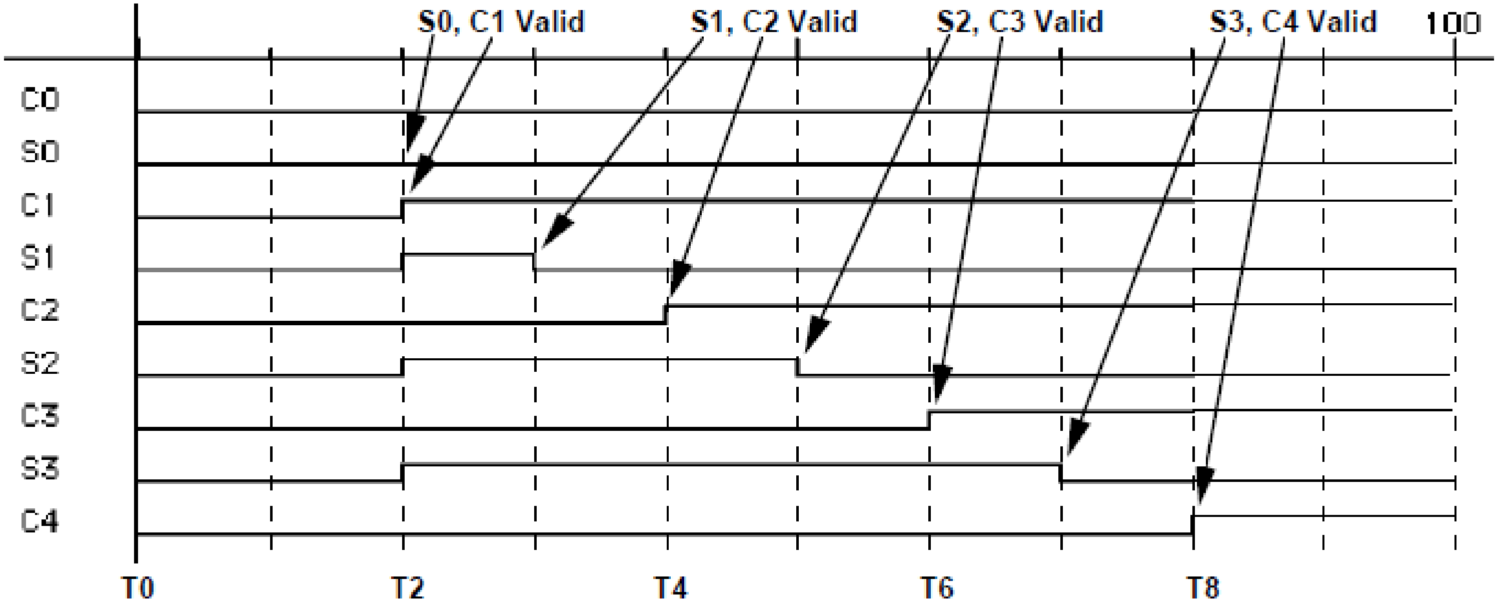
final sum and carry

Networks for Binary Addition

Carry Lookahead Circuits

Critical delay: the propagation of carry from low to high order stages

1111 + 0001
worst case
addition



T0: Inputs to the adder are valid

T2: Stage 0 carry out (C1)

T4: Stage 1 carry out (C2)

T6: Stage 2 carry out (C3)

T8: Stage 3 carry out (C4)

2 delays to compute sum

but last carry not ready
until 6 delays later

Networks for Binary Addition

Carry Lookahead Logic

Carry Generate Gi = Ai Bi — *must generate carry when A = B = 1*

Carry Propagate Pi = Ai xor Bi — *carry in will equal carry out here*

Sum and Carry can be reexpressed in terms of generate/propagate:

Si = Ai xor Bi xor Ci = Pi xor Ci

Ci+1 = Ai Bi + Ai Ci + Bi Ci

= Ai Bi + Ci (Ai + Bi)

= Ai Bi + Ci (Ai xor Bi)

= Gi + Ci Pi

Networks for Binary Addition

Carry Lookahead Logic

Reexpress the carry logic as follows:

$$C1 = G0 + P0\ C0$$

$$C2 = G1 + P1\ C1 = G1 + P1\ G0 + P1\ P0\ C0$$

$$C3 = G2 + P2\ C2 = G2 + P2\ G1 + P2\ P1\ G0 + P2\ P1\ P0\ C0$$

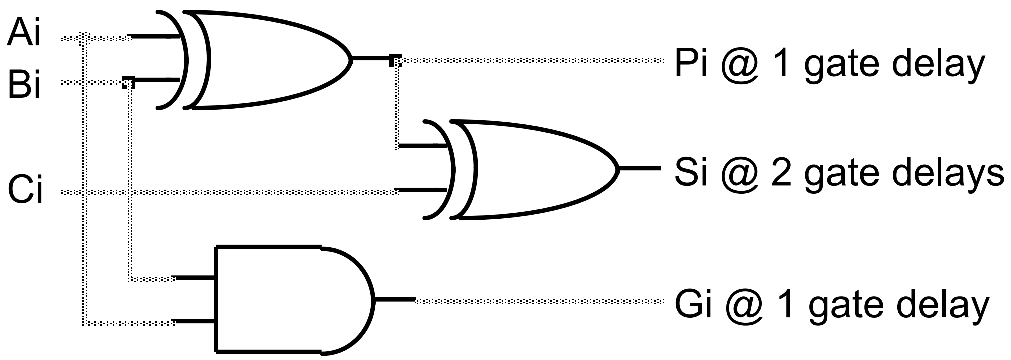
$$C4 = G3 + P3\ C3 = G3 + P3\ G2 + P3\ P2\ G1 + P3\ P2\ P1\ G0 + P3\ P2\ P1\ P0\ C0$$

Each of the carry equations can be implemented in a two-level logic network

Variables are the adder inputs and carry in to stage 0!

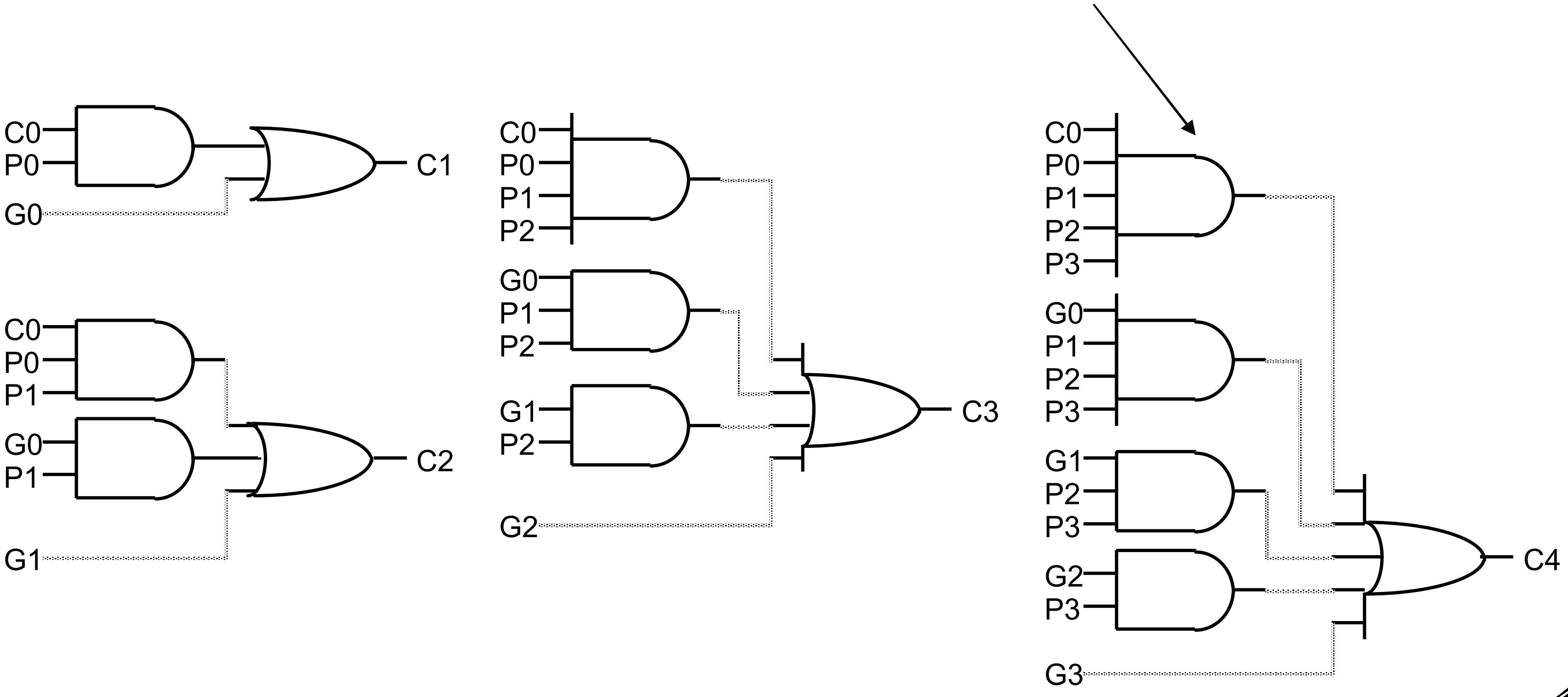
Networks for Binary Addition

Carry Lookahead Implementation



Adder with Propagate and Generate Outputs

Increasingly complex logic



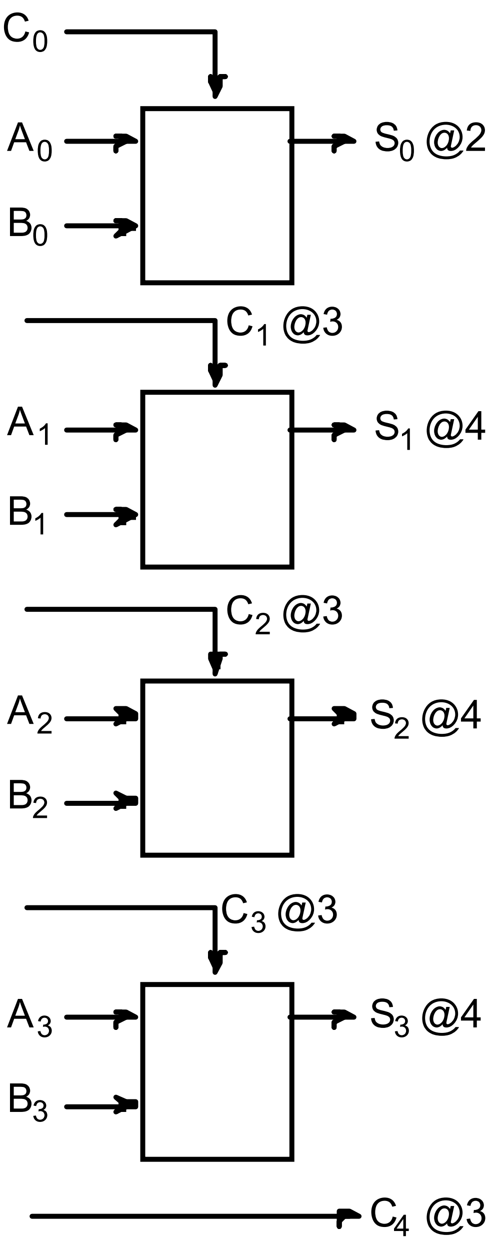
Networks for Binary Addition

Carry Lookahead Logic

Cascaded Carry Lookahead

Carry lookahead logic generates individual carries

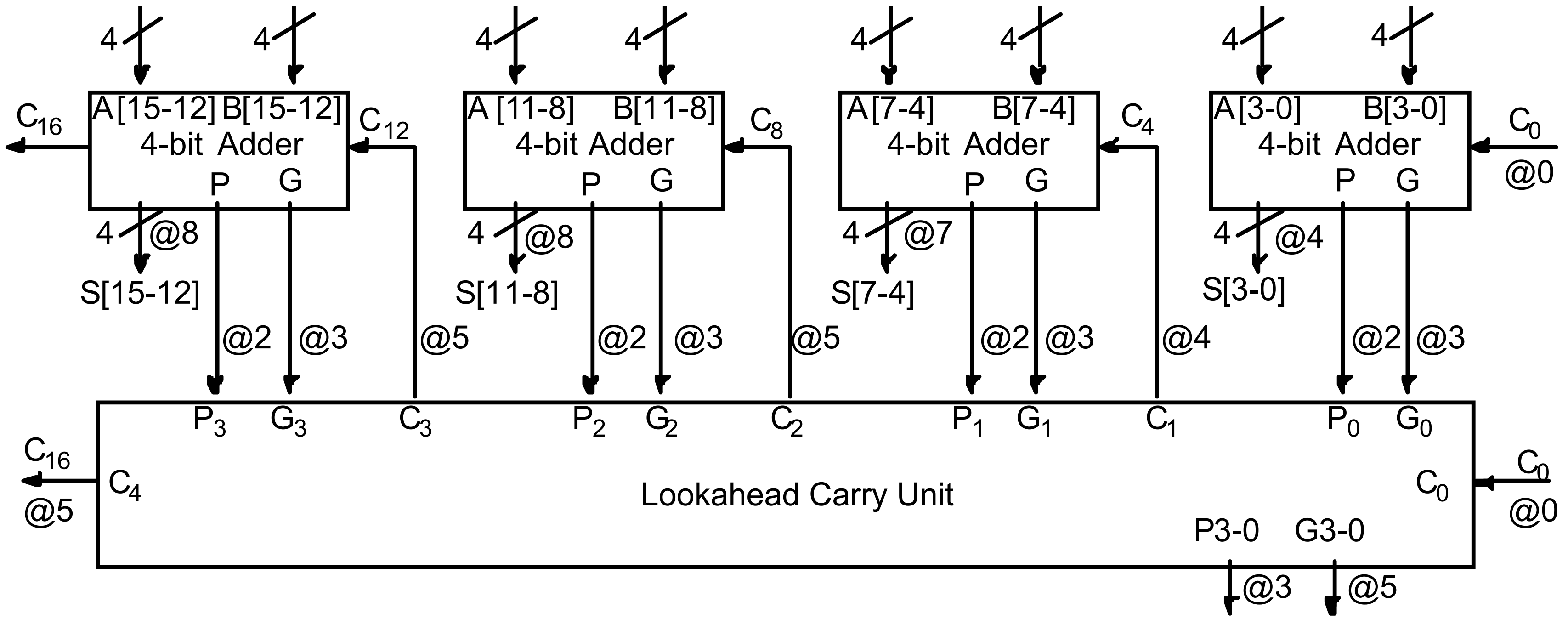
sums computed much faster



Networks for Binary Addition

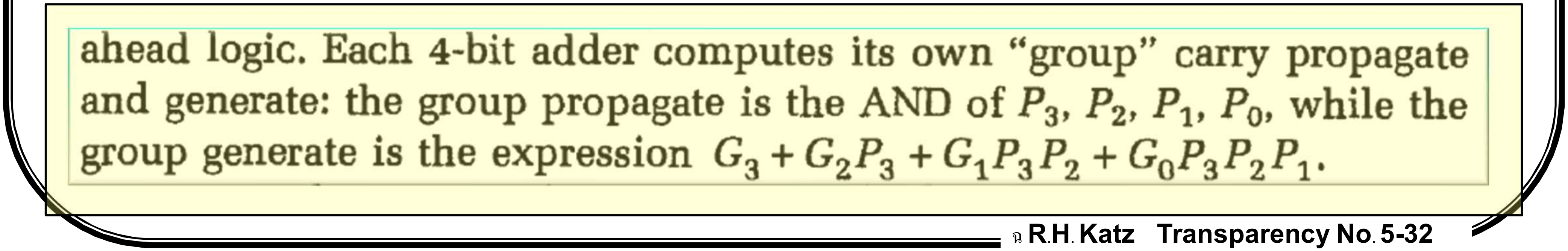
Carry Lookahead Logic

Cascaded Carry Lookahead

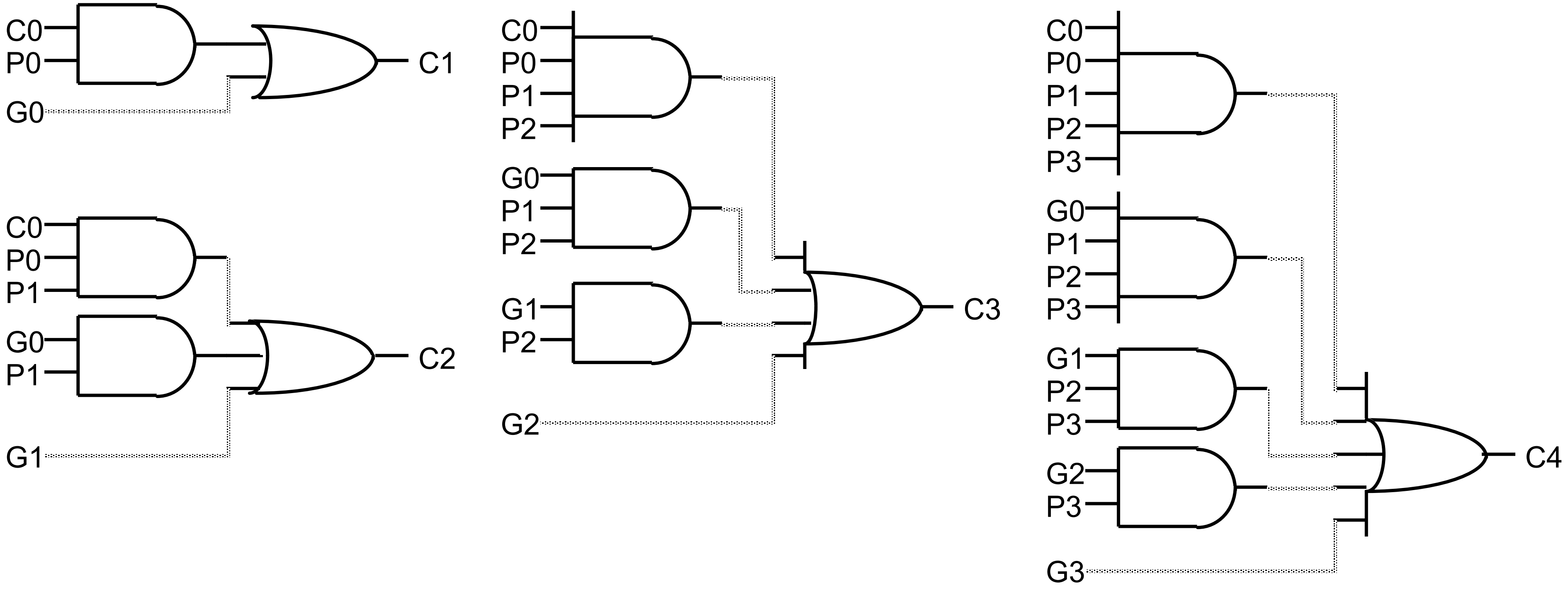


4 bit adders with internal carry lookahead

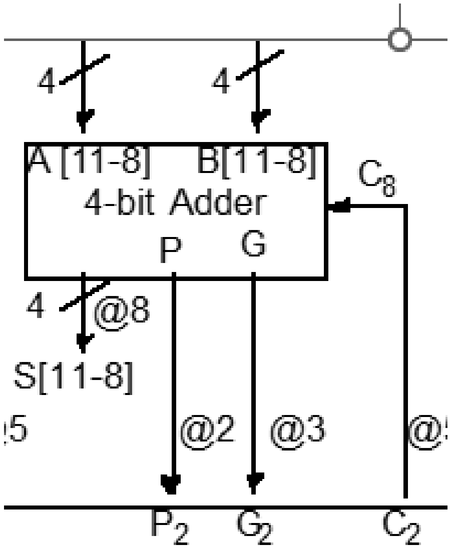
second level carry lookahead unit, extends lookahead to 16 bits

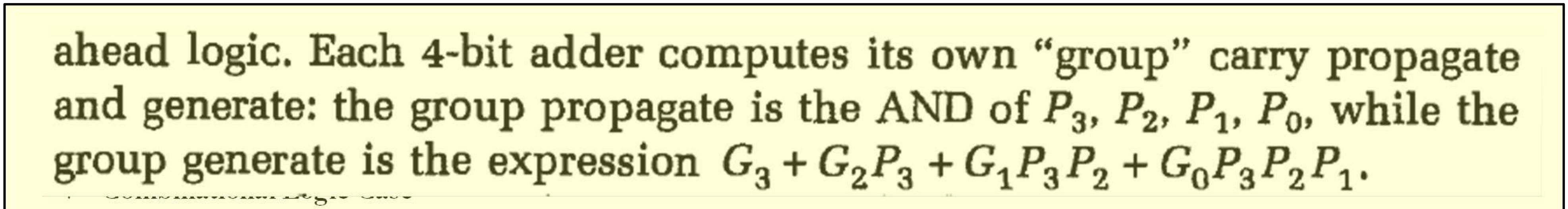

ahead logic. Each 4-bit adder computes its own "group" carry propagate and generate: the group propagate is the AND of P_3, P_2, P_1, P_0, while the group generate is the expression $G_3 + G_2P_3 + G_1P_3P_2 + G_0P_3P_2P_1$.

Calculating C1-4, C5-8, C9-12 etc. in each block



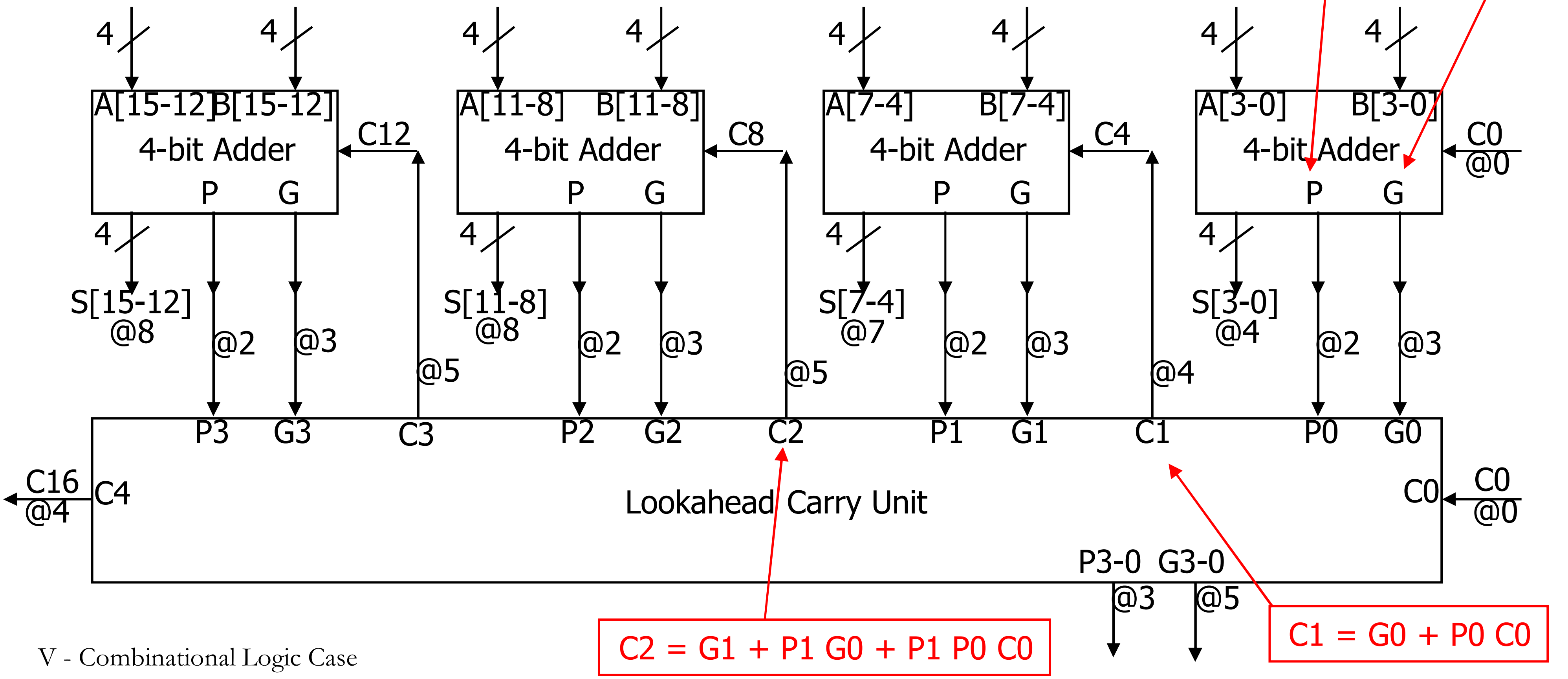
How each block computes 'group' G and P, case by case.




ahead logic. Each 4-bit adder computes its own "group" carry propagate and generate: the group propagate is the AND of P_3, P_2, P_1, P_0, while the group generate is the expression $G_3 + G_2P_3 + G_1P_3P_2 + G_0P_3P_2P_1$.

Carry-lookahead adder with cascaded carry-lookahead logic

- Carry-lookahead adder
  - 4 four-bit adders with internal carry lookahead
  - second level carry lookahead unit extends lookahead to 16 bits

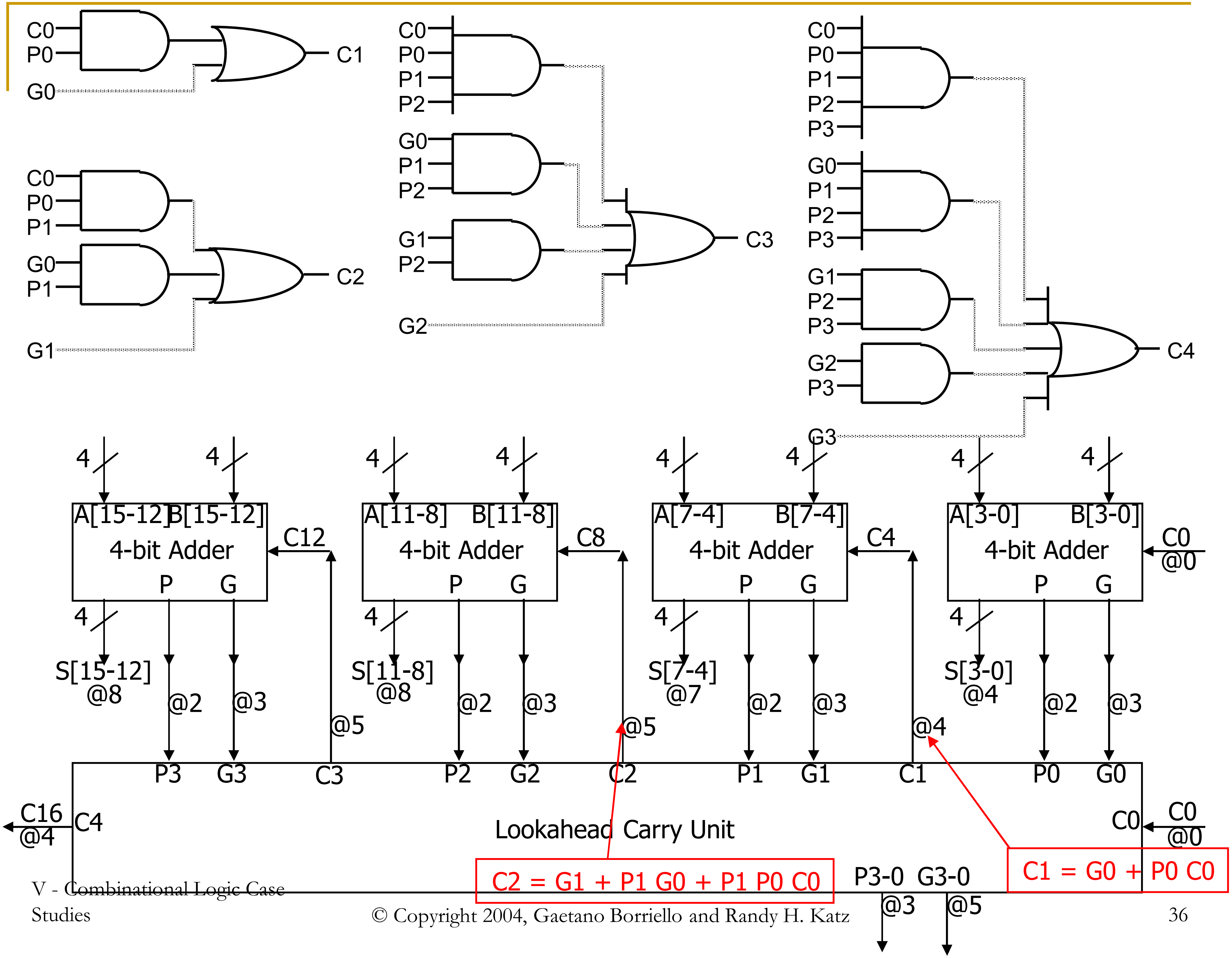


G = G3 + P3 G2 + P3 P2 G1 + P3 P2 P1 G0

P = P3 P2 P1 P0

C2 = G1 + P1 G0 + P1 P0 C0

C1 = G0 + P0 C0

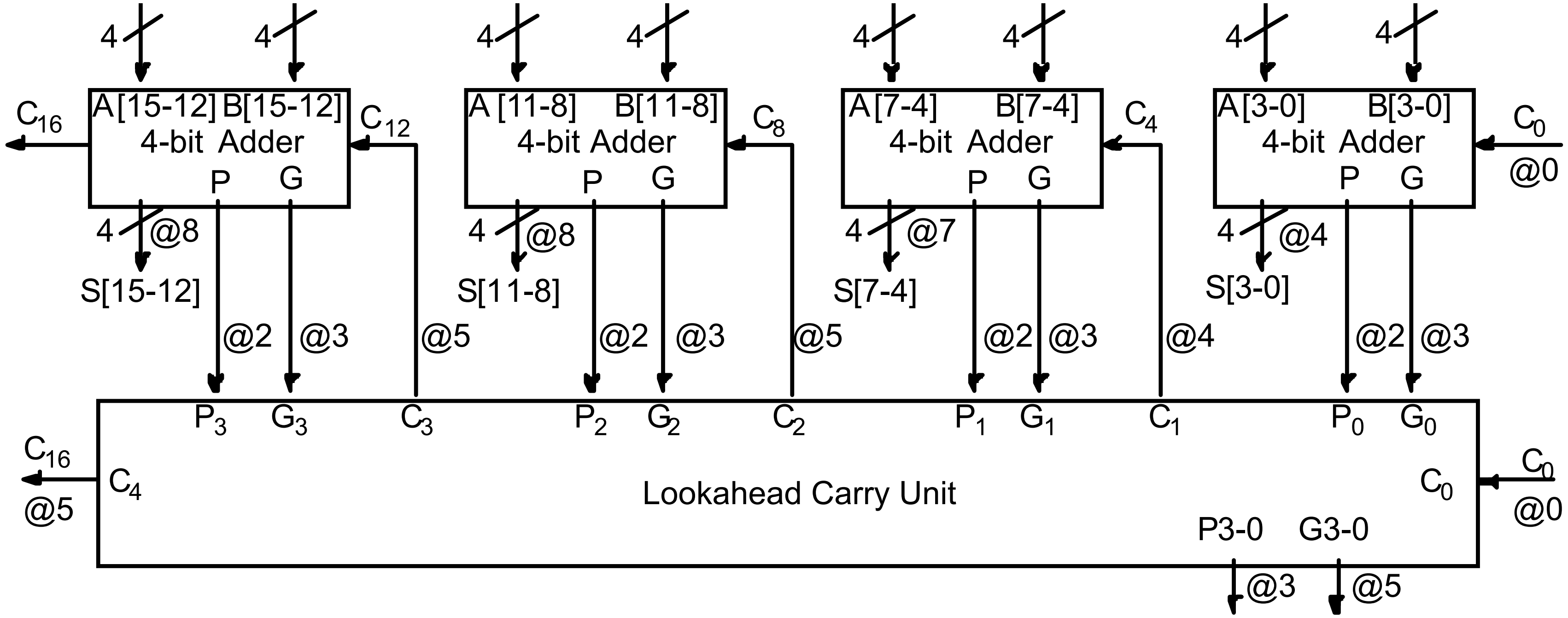


C2 = G1 + P1 G0 + P1 P0 C0

C1 = G0 + P0 C0

Counting delays

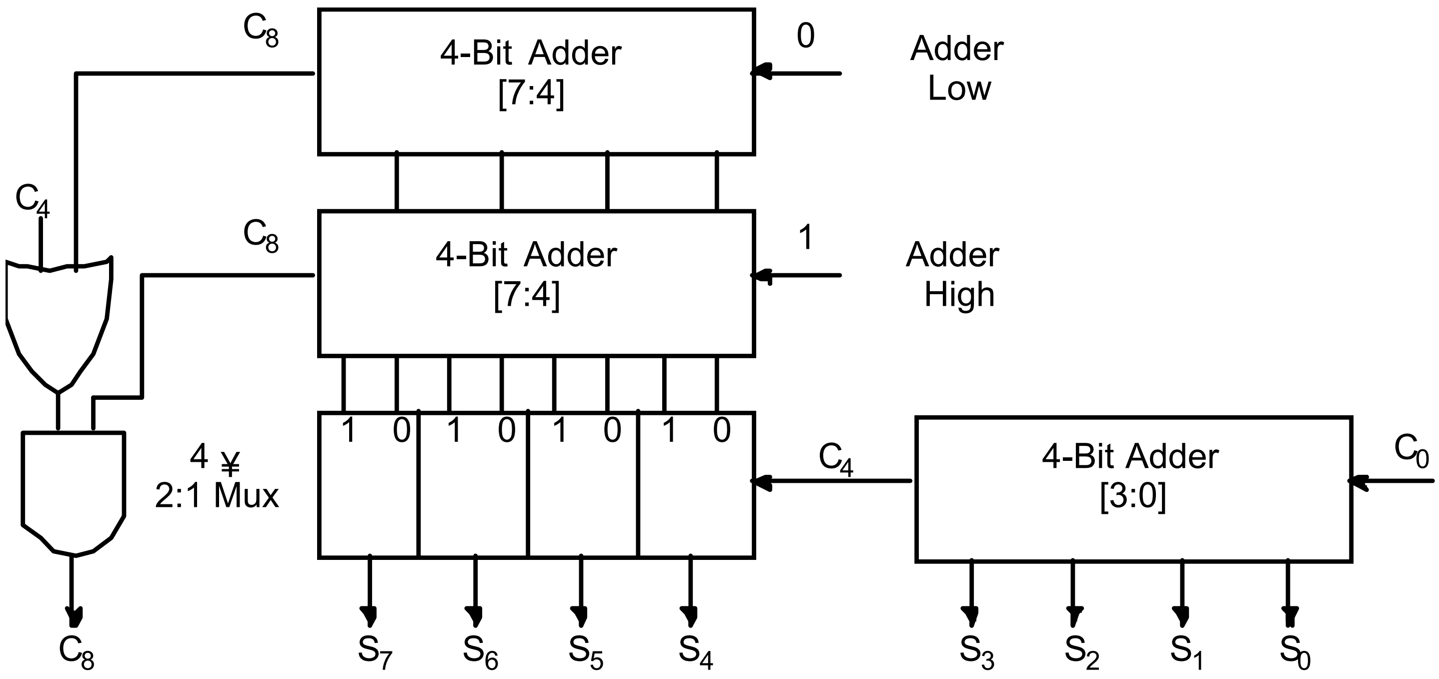
Carry Lookahead Logic Cascaded Carry Lookahead

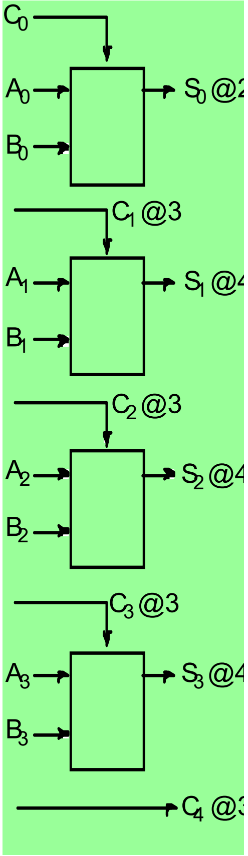


Networks for Binary Addition

Carry Select Adder

Redundant hardware to make carry calculation go faster





compute the high order sums in parallel

one addition assumes carry in = 0

the other assumes carry in = 1

Arithmetic Logic Unit Design

Sample ALU

M = 0, Logical Bitwise Operations

S1	S0	Function	Comment
0	0	Fi = Ai	Input Ai transferred to output
0	1	Fi = not Ai	Complement of Ai transferred to output
1	0	Fi = Ai xor Bi	Compute XOR of Ai, Bi
1	1	Fi = Ai xnor Bi	Compute XNOR of Ai, Bi

M = 1, C0 = 0, Arithmetic Operations

0	0	F = A	Input A passed to output
0	1	F = not A	Complement of A passed to output
1	0	F = A plus B	Sum of A and B
1	1	F = (not A) plus B	Sum of B and complement of A

M = 1, C0 = 1, Arithmetic Operations

0	0	F = A plus 1	Increment A
0	1	F = (not A) plus 1	Twos complement of A
1	0	F = A plus B plus 1	Increment sum of A and B
1	1	F = (not A) plus B plus 1	B minus A

Logical and Arithmetic Operations

Not all operations appear useful, but "fall out" of internal logic

Arithmetic Logic Unit Design

Sample ALU

Traditional Design Approach

Truth Table & Espresso

23 product terms!

```
.i 6
.o 2
.ilb m s1 s0 ci ai bi
.ob fi co
.p 23
111101 10
110111 10
1-0100 10
1-1110 10
10010- 10
10111- 10
-10001 10
010-01 10
-11011 10
011-11 10
--1000 10
0-1-00 10
--0010 10
0-0-10 10
-0100- 10
001-0- 10
-0001- 10
000-1- 10
-1-1-1 01
--1-01 01
--0-11 01
--110- 01
--011- 01
.e
```

Equivalent to
25 gates

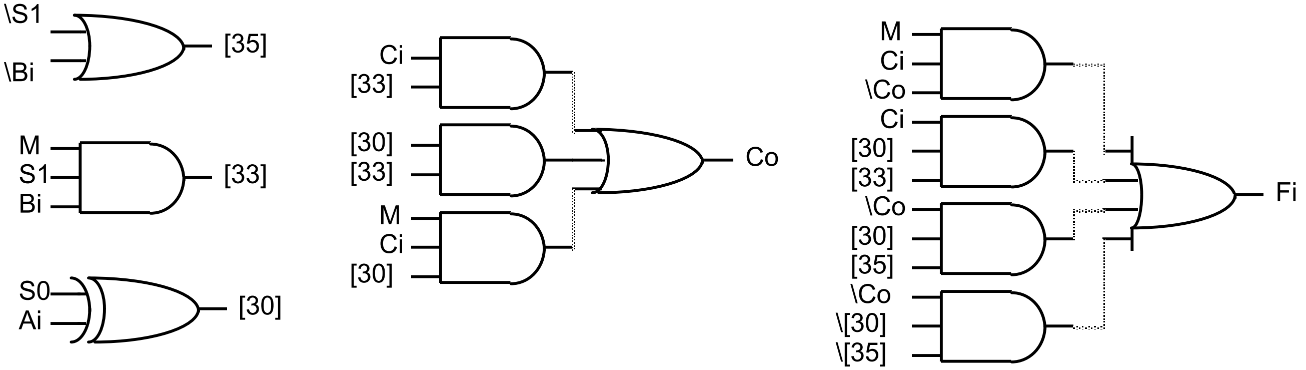
M	S1	S0	Ci	Ai	Bi	Fi	Ci+1
0	0	0	X	0	X	0	X
			X	1	X	1	X
	0	1	X	0	X	1	X
			X	1	X	0	X
	1	0	X	0	0	0	X
			X	0	1	1	X
			X	1	0	1	X
			X	1	1	0	X
	1	1	X	0	0	1	X
			X	0	1	0	X
			X	1	0	0	X
			X	1	1	1	X
1	0	0	0	0	X	0	X
			0	1	X	1	X
	0	1	0	0	X	1	X
			0	1	X	0	X
	1	0	0	0	0	0	0
			0	0	1	1	0
			0	1	0	1	0
			0	1	1	0	1
	1	1	0	0	0	1	0
			0	0	1	0	1
			0	1	0	0	0
			0	1	1	1	0
1	0	0	1	0	X	1	0
			1	1	X	0	1
	0	1	1	0	X	0	1
			1	1	X	1	0
	1	0	1	0	0	1	0
			1	0	1	0	1
			1	1	0	0	1
			1	1	1	1	1
	1	1	1	0	0	0	1
			1	0	1	1	1
			1	1	0	1	0
			1	1	1	0	1

Arithmetic Logic Unit Design

Sample ALU

Multilevel Implementation

```
.model alu.espresso
.inputs m s1 s0 ci ai bi
.outputs fi co
.names m ci co [30] [33] [35] fi
110--- 1
-1-11- 1
--01-1 1
--00-0 1
.names m ci [30] [33] co
-1-1 1
--11 1
111- 1
.names s0 ai [30]
01 1
10 1
.names m s1 bi [33]
111 1
.names s1 bi [35]
0- 1
-0 1
.end
```

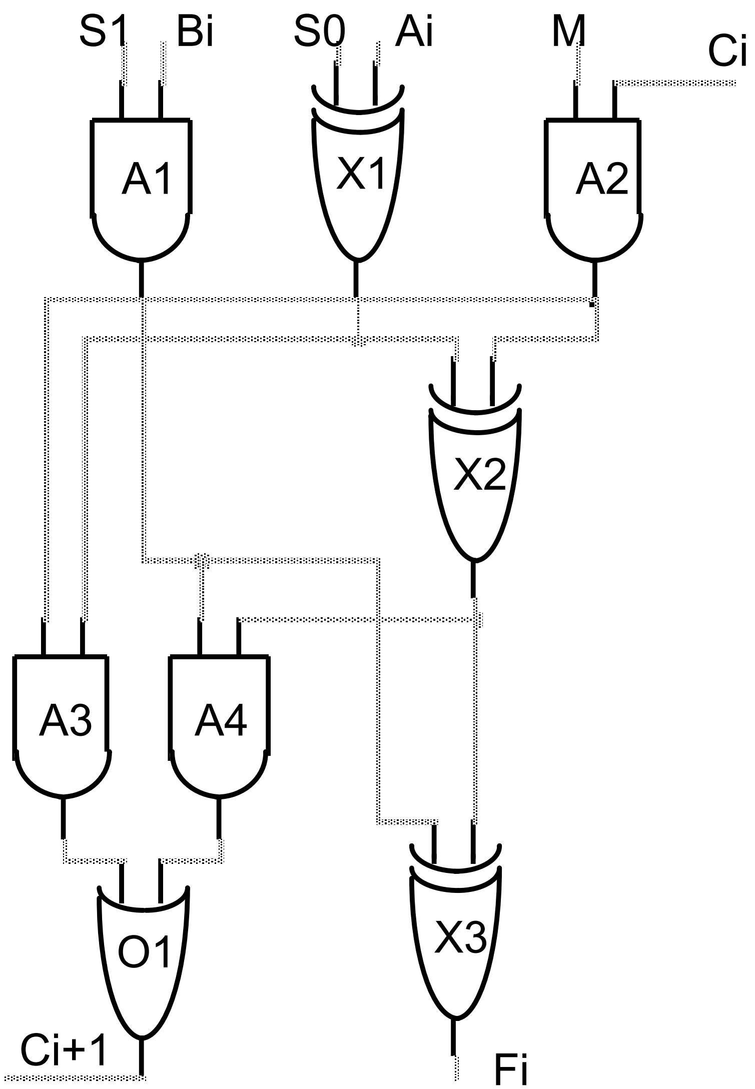


12 Gates

Arithmetic Logic Unit Design

Sample ALU

Clever Multi-level Logic Implementation



8 Gates (but 3 are XOR)

S1 = 0 blocks Bi
Happens when operations involve Ai only

Same is true for Ci when M = 0

Addition happens when M = 1

Bi, Ci to Xor gates X2, X3

S0 = 0, X1 passes A

S0 = 1, X1 passes $\overline{A}$

Arithmetic Mode:

Or gate inputs are Ai Ci and Bi (Ai xor Ci)

Logic Mode:

Cascaded XORs form output from Ai and Bi

Arithmetic Logic Unit Design

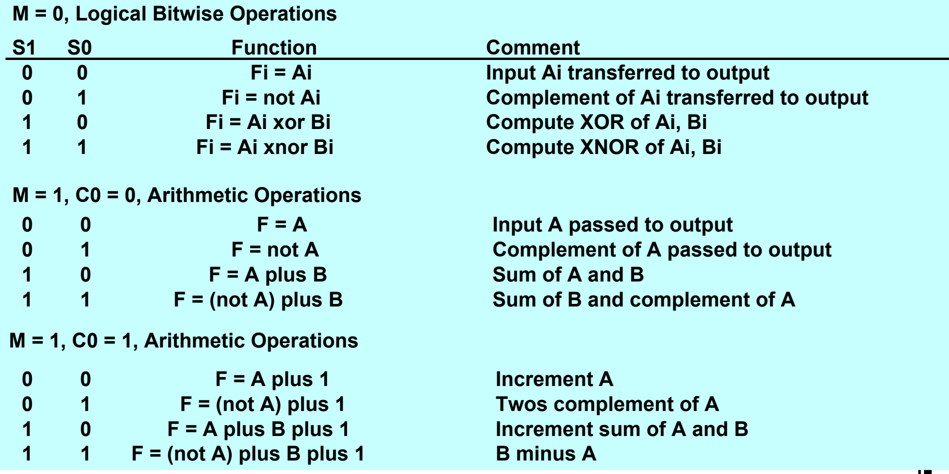
first-level gates
use S0 to complement Ai
S0 = 0 causes gate X1 to pass Ai
S0 = 1 causes gate X1 to pass Ai'
use S1 to block Bi
S1 = 0 causes gate A1 to make Bi go forward as 0
(don't want Bi for operations with just A)
S1 = 1 causes gate A1 to pass Bi
use M to block Ci
M = 0 causes gate A2 to make Ci go forward as 0
(don't want Ci for logical operations)
M = 1 causes gate A2 to pass Ci

other gates
for M=0 (logical operations, Ci is ignored)
Fi = S1 Bi xor (S0 xor Ai)
= S1'S0' (Ai) + S1'S0 (Ai') +
S1 S0' (Ai Bi' + Ai' Bi) + S1 S0 (Ai' Bi' + Ai Bi)
for M=1 (arithmetic operations)
Fi = S1 Bi xor ((S0 xor Ai) xor Ci) =
Ci+1 = Ci (S0 xor Ai) + S1 Bi ((S0 xor Ai) xor Ci) =

just a full adder with inputs S0 xor Ai, S1 Bi, and Ci

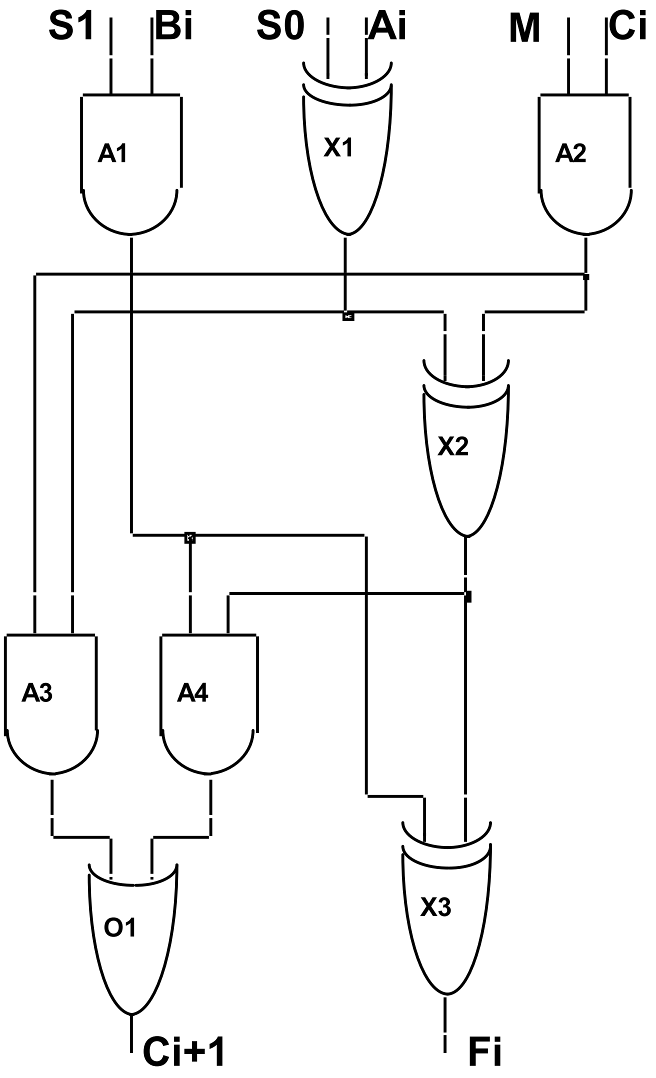
M	S1	S0	Ci	Ai	Bi	Fi	Ci+1
0	0	0	X	0	X	0	X
			X	1	X	1	X
	0	1	X	0	X	1	X
			X	1	X	0	X
	1	0	X	0	0	0	X
			X	0	1	1	X
			X	1	0	1	X
			X	1	1	0	X
	1	1	X	0	0	1	X
			X	0	1	0	X
			X	1	0	0	X
			X	1	1	1	X
1	0	0	0	0	X	0	X
			0	1	X	1	X
	0	1	0	0	X	1	X
			0	1	X	0	X
	1	0	0	0	0	0	0
			0	0	1	1	0
			0	1	0	1	0
			0	1	1	0	1
	1	1	0	0	0	1	0
			0	0	1	0	1
			0	1	0	0	0
			0	1	1	1	0
1	0	0	1	0	X	1	0
			1	1	X	0	1
	0	1	1	0	X	0	1
			1	1	X	1	0
	1	0	1	0	0	1	0
			1	0	1	0	1
			1	1	0	0	1
			1	1	1	1	1
	1	1	1	0	0	0	1
			1	0	1	1	1
			1	1	0	1	0
			1	1	1	0	1

Sample ALU – clever multi-level implementat



M = 0, Logical Bitwise Operations

S1	S0	Function	Comment
0	0	Fi = Ai	Input Ai transferred to output
0	1	Fi = not Ai	Complement of Ai transferred to output
1	0	Fi = Ai xor Bi	Compute XOR of Ai, Bi
1	1	Fi = Ai xnor Bi	Compute XNOR of Ai, Bi
M = 1, C0 = 0, Arithmetic Operations			
0	0	F = A	Input A passed to output
0	1	F = not A	Complement of A passed to output
1	0	F = A plus B	Sum of A and B
1	1	F = (not A) plus B	Sum of B and complement of A
M = 1, C0 = 1, Arithmetic Operations			
0	0	F = A plus 1	Increment A
0	1	F = (not A) plus 1	Twos complement of A
1	0	F = A plus B plus 1	Increment sum of A and B
1	1	F = (not A) plus B plus 1	B minus A



first-level gates
  use S0 to complement Ai
    S0 = 0   causes gate X1 to pass Ai
    S0 = 1   causes gate X1 to pass Ai'
  use S1 to block Bi
    S1 = 0   causes gate A1 to make Bi go forward as 0
              (don't want Bi for operations with just A)
    S1 = 1   causes gate A1 to pass Bi
  use M to block Ci
    M = 0   causes gate A2 to make Ci go forward as 0
             (don't want Ci for logical operations)
    M = 1   causes gate A2 to pass Ci

other gates
  for M=0 (logical operations, Ci is ignored)
    Fi = S1 Bi xor (S0 xor Ai)
      = S1'S0' (Ai) + S1'S0 (Ai') +
        S1 S0' (Ai Bi' + Ai' Bi) + S1 S0 (Ai' Bi' + Ai Bi)
  for M=1 (arithmetic operations)
    Fi = S1 Bi xor ((S0 xor Ai) xor Ci) =
    Ci+1 = Ci (S0 xor Ai) + S1 Bi ((S0 xor Ai) xor Ci) =

    just a full adder with inputs S0 xor Ai, S1 Bi, and Ci

Arithmetic Logic Unit Design

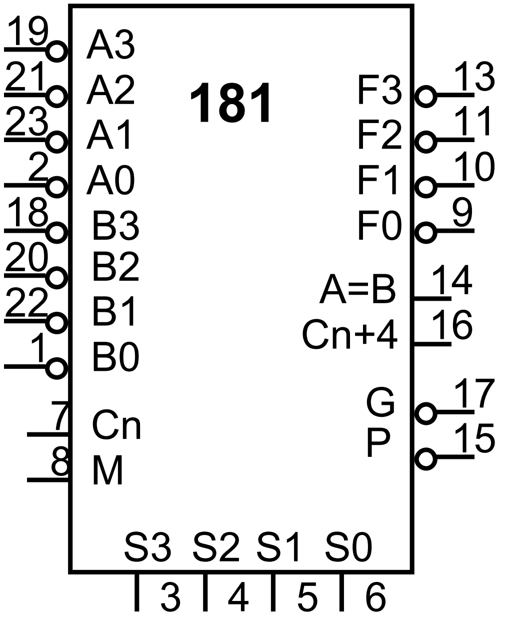
74181 TTL ALU

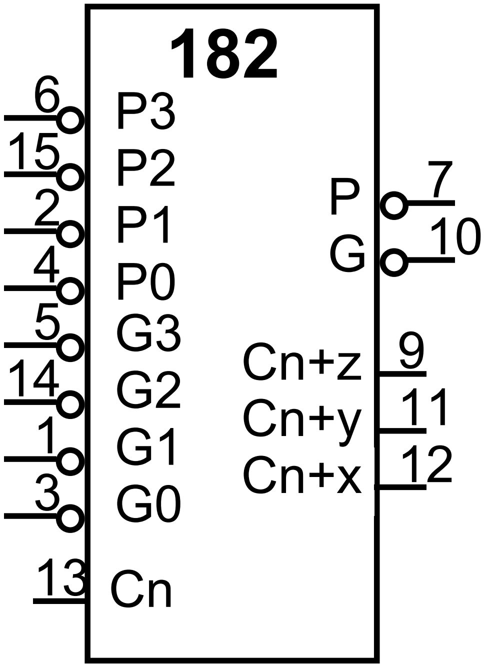
Selection				M = 1	M = 0, Arithmetic Functions	
S3	S2	S1	S0	Logic Function	Cn = 0	Cn = 1
0	0	0	0	F = not A	F = A minus 1	F = A
0	0	0	1	F = A nand B	F = A B minus 1	F = A B
0	0	1	0	F = (not A) + B	F = A (not B) minus 1	F = A (not B)
0	0	1	1	F = 1	F = minus 1	F = zero
0	1	0	0	F = A nor B	F = A plus (A + not B)	F = A plus (A + not B) plus 1
0	1	0	1	F = not B	F = A B plus (A + not B)	F = A B plus (A + not B) plus 1
0	1	1	0	F = A xnor B	F = A minus B minus 1	F = (A + not B) plus 1
0	1	1	1	F = A + not B	F = A + not B	F = A minus B
1	0	0	0	F = (not A) B	F = A plus (A + B)	F = (A + not B) plus 1
1	0	0	1	F = A xor B	F = A plus B	F = A plus (A + B) plus 1
1	0	1	0	F = B	F = A (not B) plus (A + B)	F = A (not B) plus (A + B) plus 1
1	0	1	1	F = A + B	F = (A + B)	F = (A + B) plus 1
1	1	0	0	F = 0	F = A	F = A plus A plus 1
1	1	0	1	F = A (not B)	F = A B plus A	F = AB plus A plus 1
1	1	1	0	F = A B	F= A (not B) plus A	F = A (not B) plus A plus 1
1	1	1	1	F = A	F = A	F = A plus 1

Arithmetic Logic Unit Design

74181 TTL ALU

Note that the sense of the carry in and out are OPPOSITE from the input bits

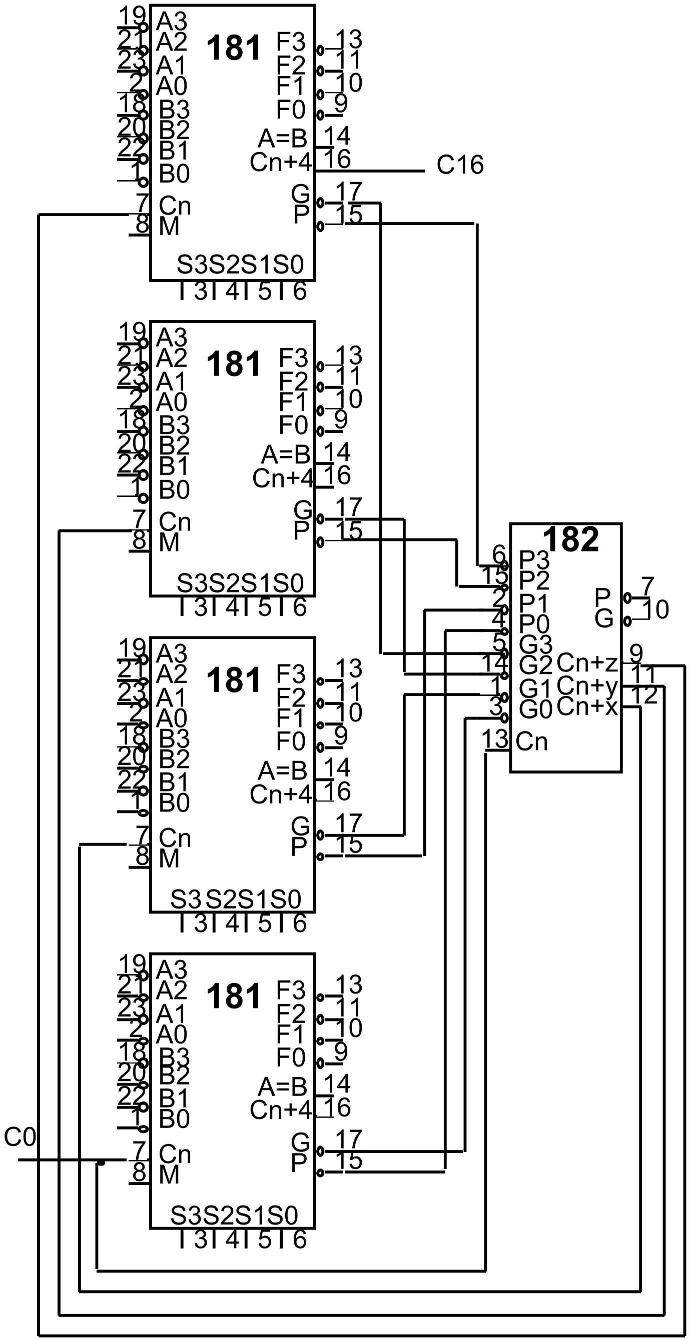




Fortunately, carry lookahead generator maintains the correct sense of the signals

Arithmetic Logic Unit Design

16-bit ALU with Carry Lookahead



BCD Addition

BCD Number Representation

Decimal digits 0 thru 9 represented as 0000 thru 1001 in binary

Addition:

5 = 0101

3 = $\underline{0011}$

1000 = 8

5 = 0101

8 = $\underline{1000}$

1101 = 13!

Problem
when digit
sum exceeds 9

Solution: add 6 (0110) if sum exceeds 9!

5 = 0101

8 = $\underline{1000}$

1101

6 = $\underline{0110}$

1 0011 = 1 3 in BCD

9 = 1001

7 = $\underline{0111}$

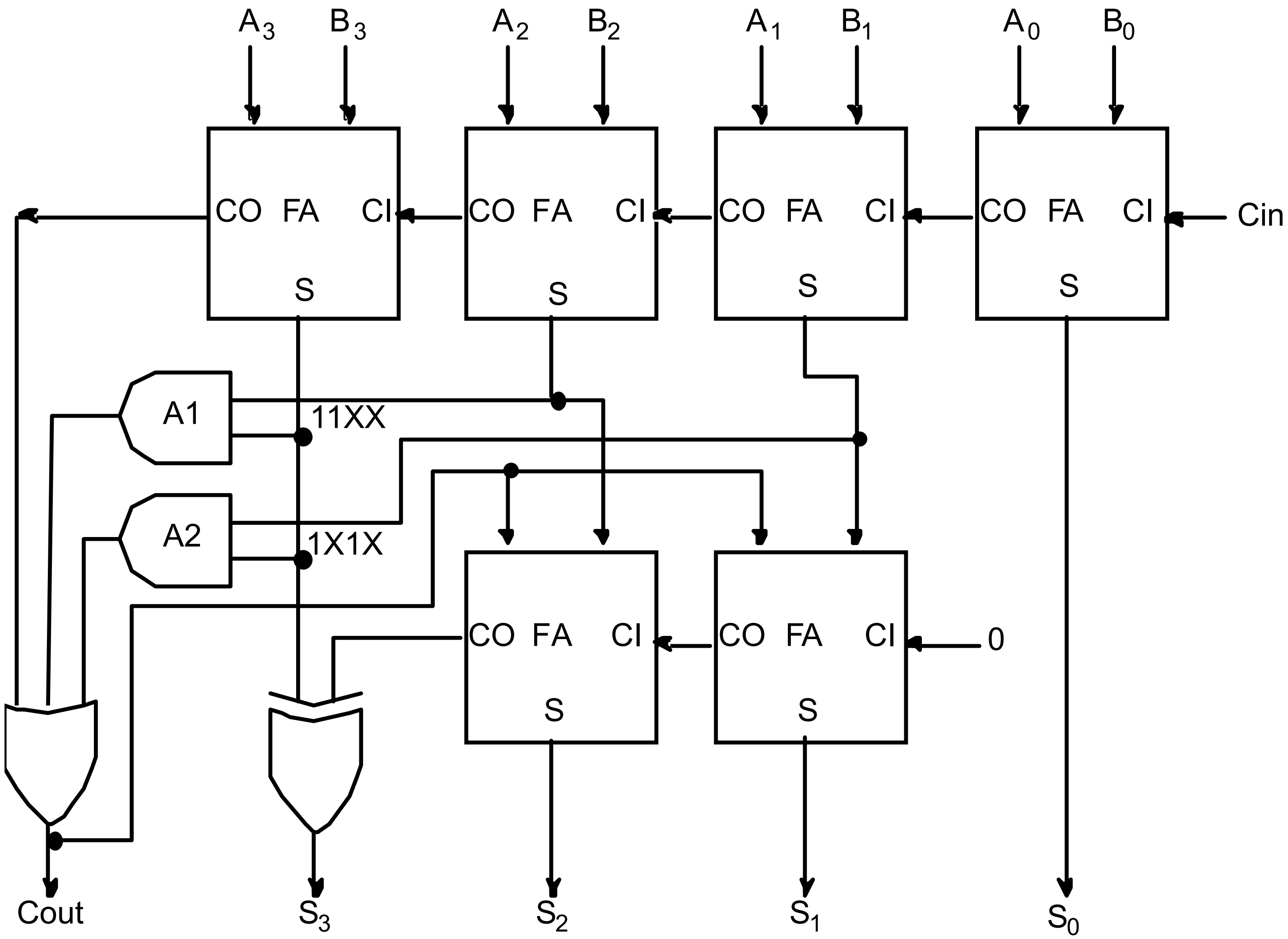
1 0000 = 16 in binary

6 = $\underline{0110}$

1 0110 = 1 6 in BCD

BCD Addition

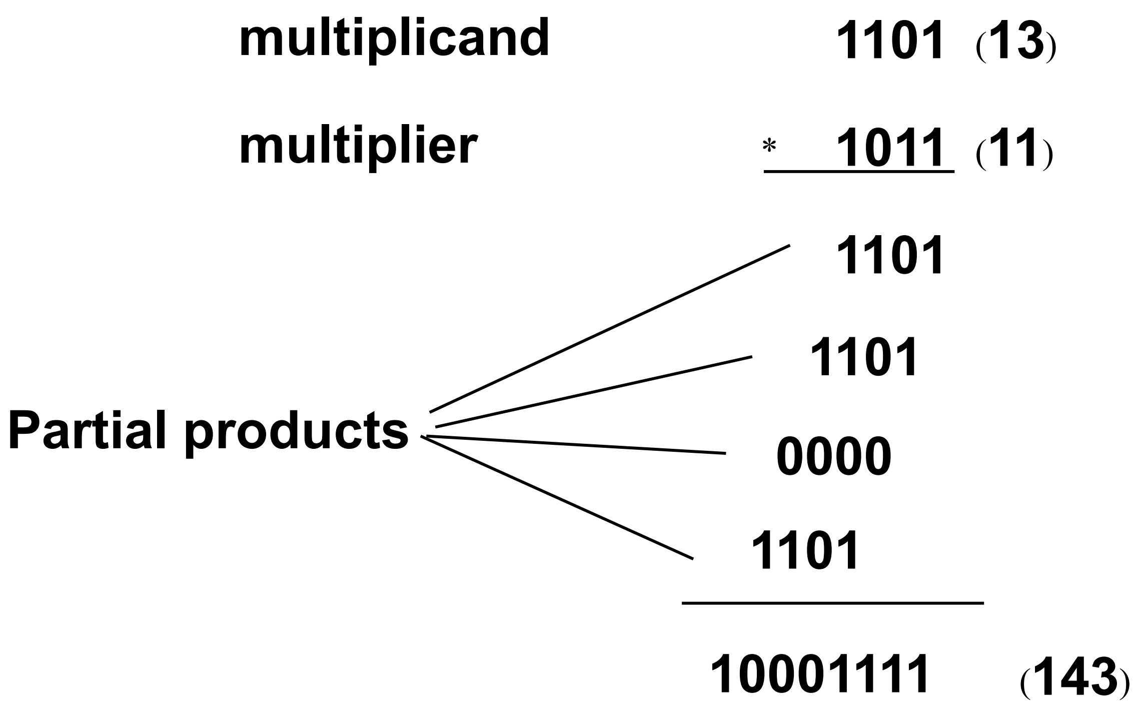
Adder Design



Add 0110 to sum whenever it exceeds 1001 (**11XX or 1X1X**)

Combinational Multiplier

Basic Concept



```
multiplicand           1101  (13)

multiplier           *  1011  (11)
                       ----
                       1101
                      1101
Partial products     0000
                    1101
                   --------
                   10001111   (143)
```

product of 2 4-bit numbers
is an 8-bit number

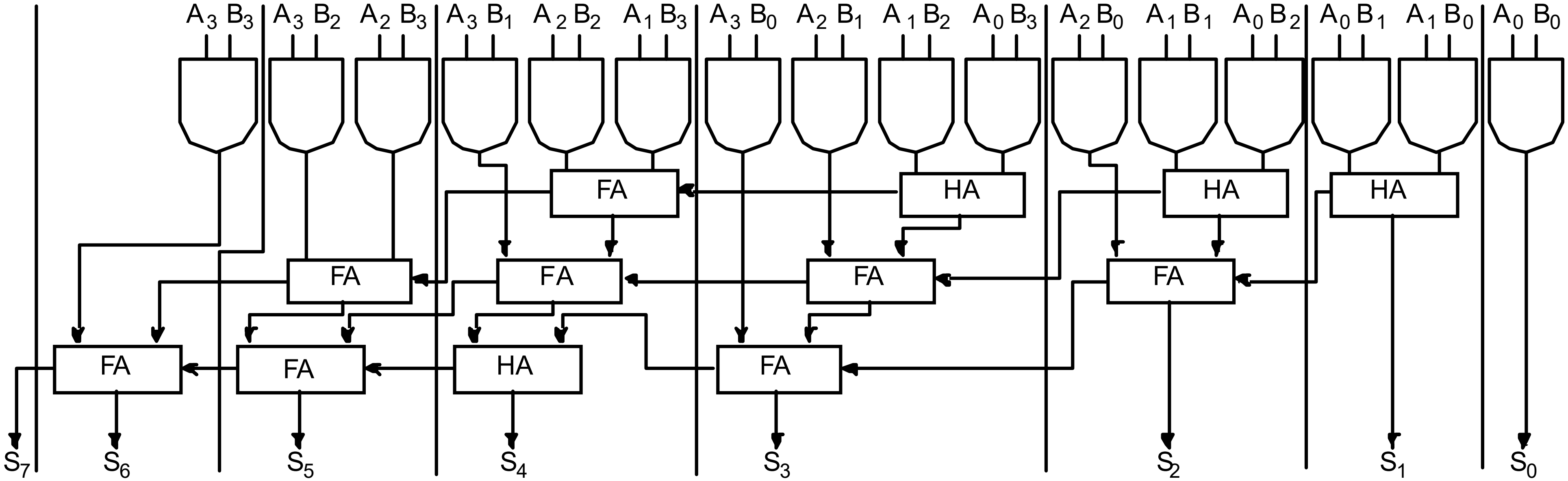
Combinational Multiplier

Partial Product Accumulation

				A3	A2	A1	A0
				B3	B2	B1	B0
				A3 B0	A2 B0	A1 B0	A0 B0
			A3 B1	A2 B1	A1 B1	A0 B1	
		A3 B2	A2 B2	A1 B2	A0 B2		
	A3 B3	A2 B3	A1 B3	A0 B3			
S7	S6	S5	S4	S3	S2	S1	S0

Combinational Multiplier

Partial Product Accumulation



Note use of parallel carry-outs to form higher order sums

12 Adders, if full adders, this is 6 gates each = 72 gates

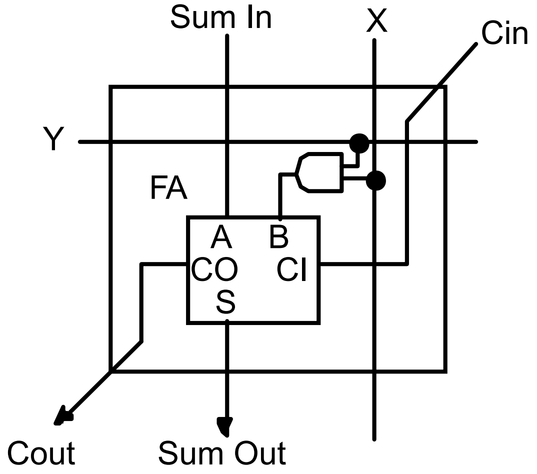
16 gates form the partial products

total = 88 gates!

Combinational Multiplier

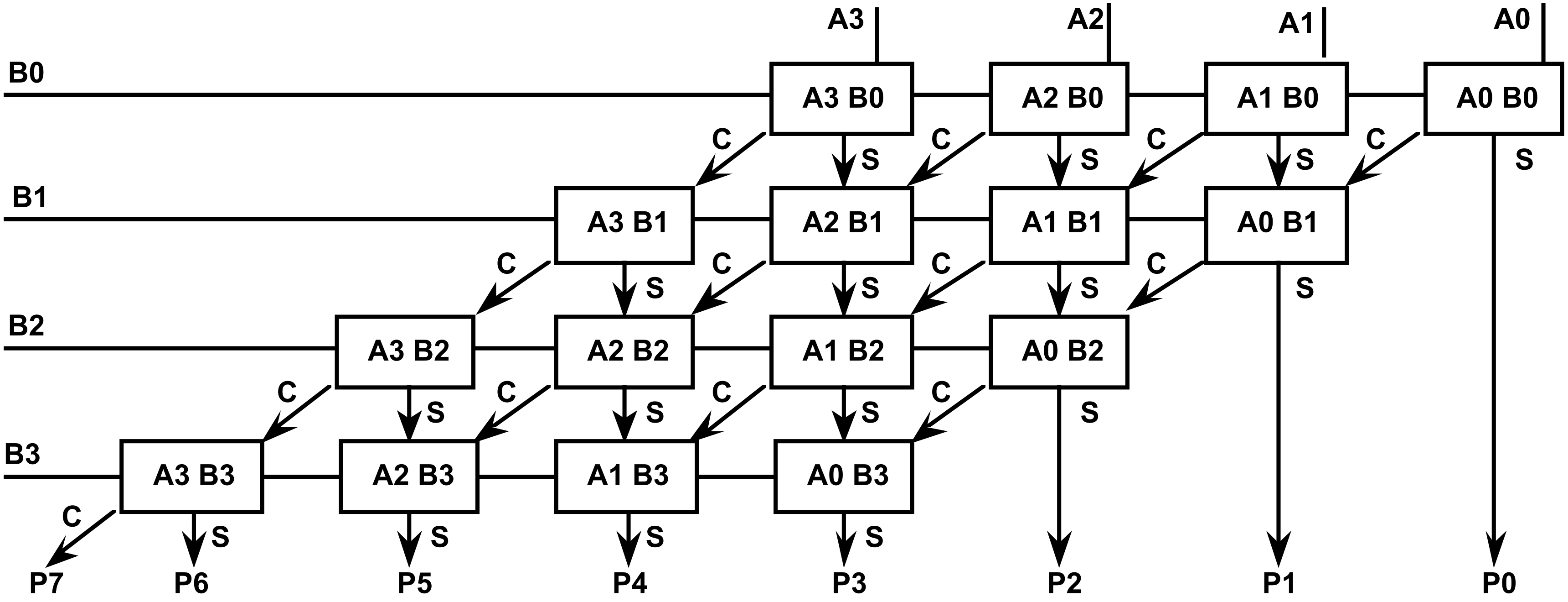
Another Representation of the Circuit

Building block: full adder + and

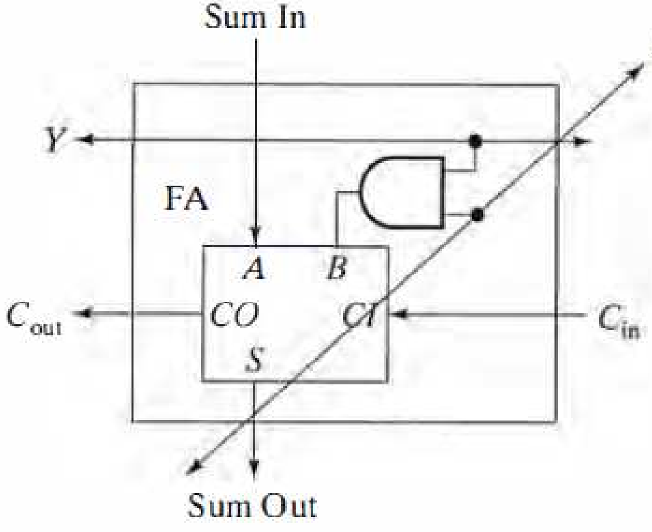


Partial Product Accumulation

				A3	A2	A1	A0
				B3	B2	B1	B0
				A2 B0	A2 B0	A1 B0	A0 B0
			A3 B1	A2 B1	A1 B1	A0 B1	
		A3 B2	A2 B2	A1 B2	A0 B2		
	A3 B3	A2 B3	A1 B3	A0 B3			
S7	S6	S5	S4	S3	S2	S1	S0



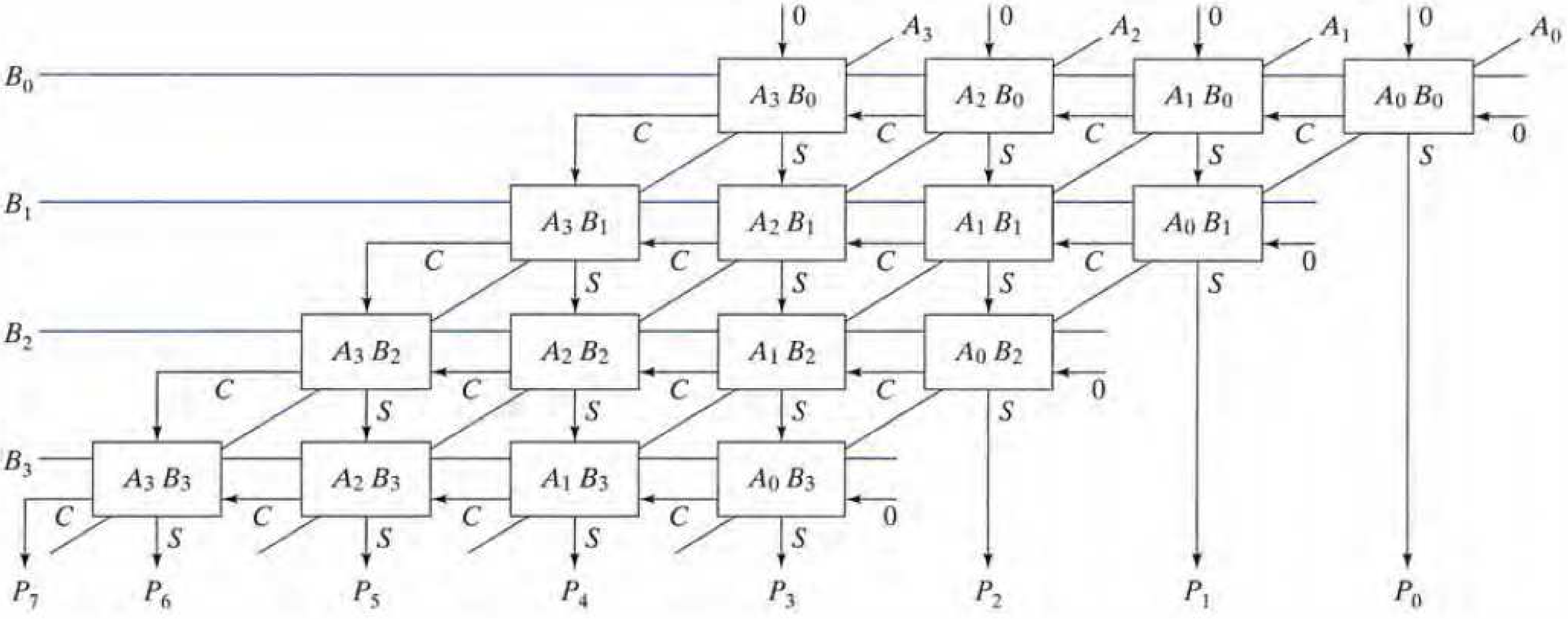
4 x 4 array of building blocks



(a) Basic building block

Partial Product Accumulation

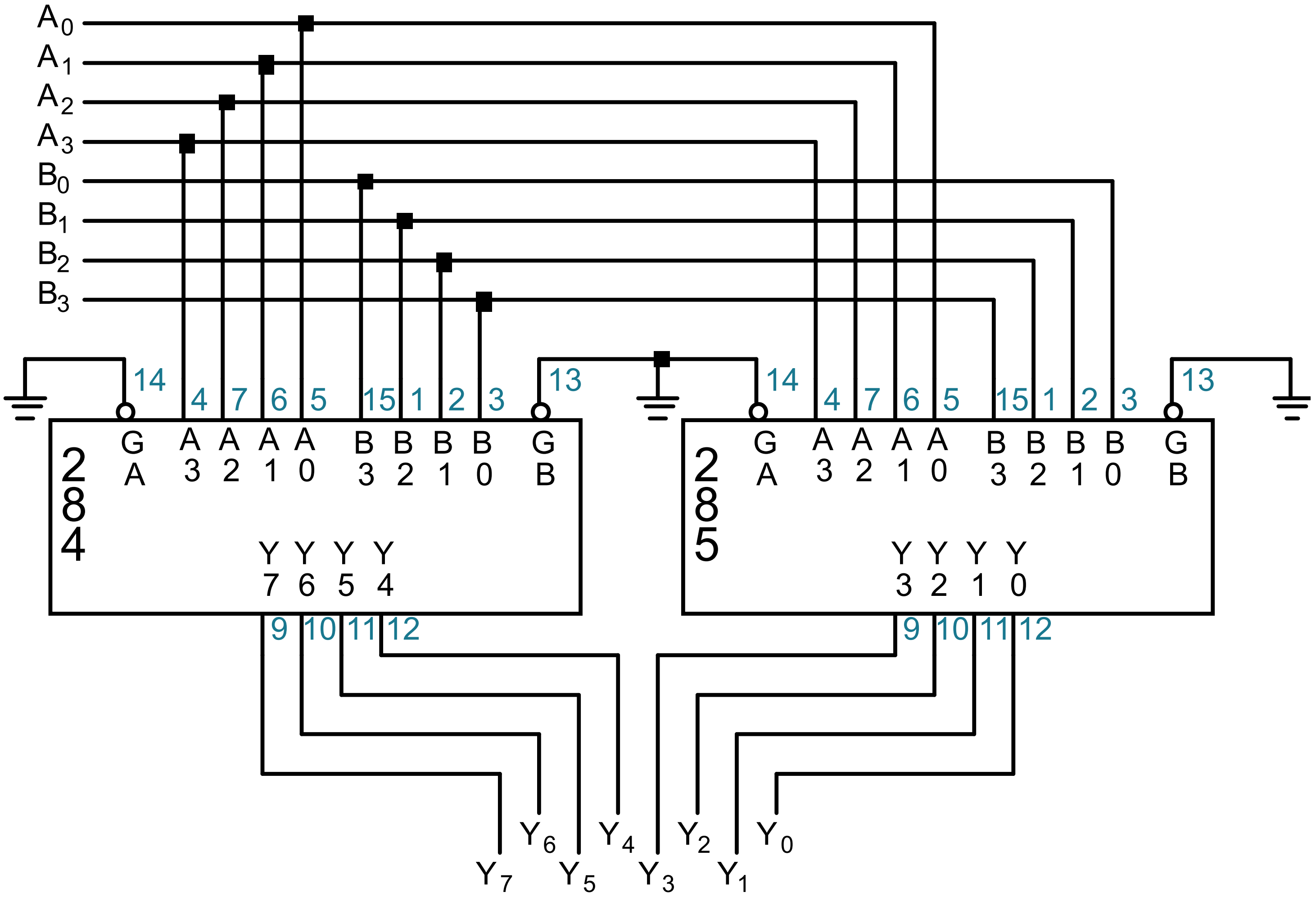
				A3	A2	A1	A0
				B3	B2	B1	B0
				A3 B0	A2 B0	A1 B0	A0 B0
			A3 B1	A2 B1	A1 B1	A0 B1	
		A3 B2	A2 B2	A1 B2	A0 B2		
	A3 B3	A2 B3	A1 B3	A0 B3			
S7	S6	S5	S4	S3	S2	S1	S0



(b) 4 × 4 combinational multiplier

Case Study: 8 x 8 Multiplier

TTL Multipliers

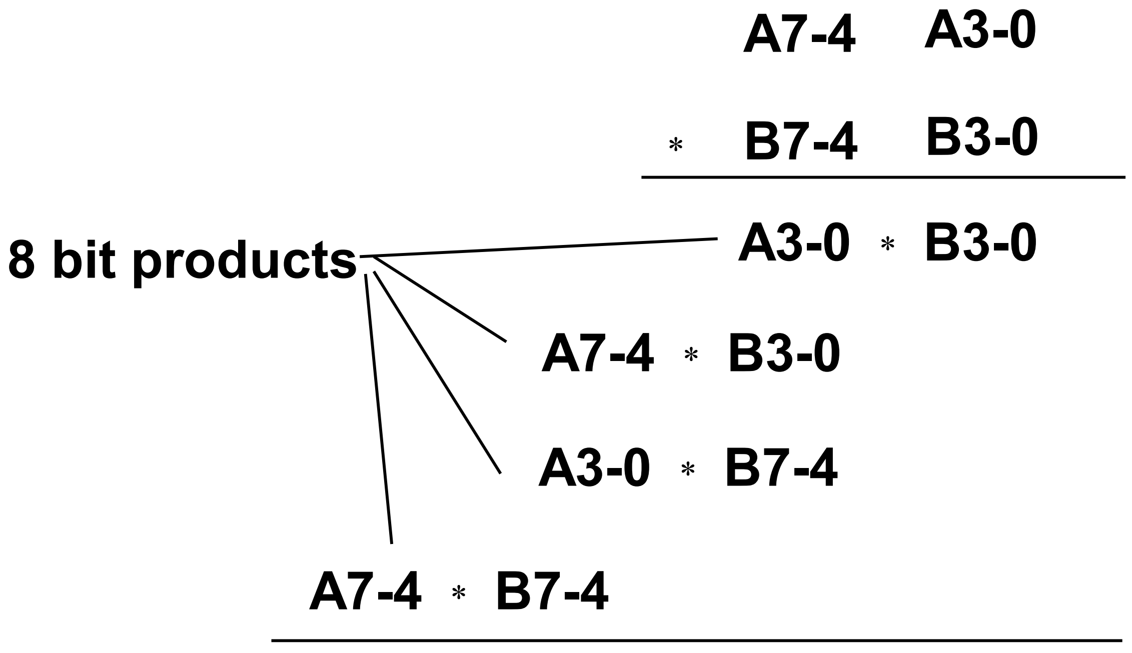


Two chip implementation of 4 x 4 multipler

Case Study: 8 x 8 Multiplier

Problem Decomposition

How to implement 8 x 8 multiply in terms of 4 x 4 multiplies?

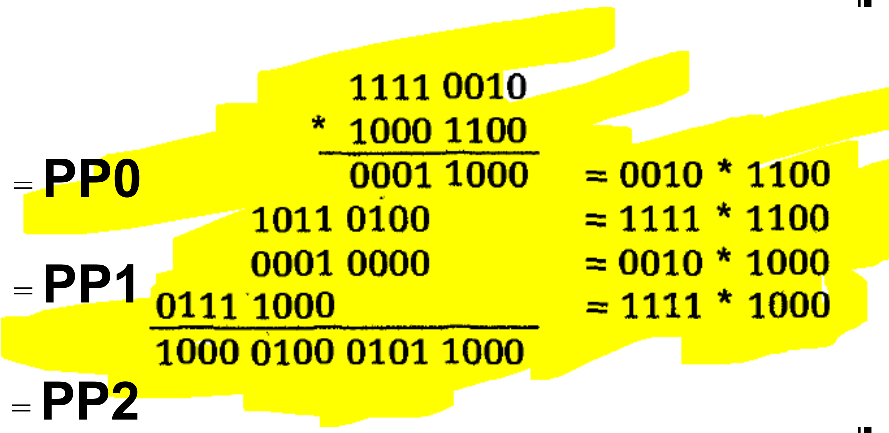


= PP0

= PP1

= PP2

= PP3



```
          1111 0010
        * 1000 1100
          0001 1000      = 0010 * 1100
     1011 0100           = 1111 * 1100
     0001 0000           = 0010 * 1000
0111 1000                = 1111 * 1000
1000 0100 0101 1000
```

P15-12 P11-8 P7-4 P3-0

P3-0 = PP0$_{3-0}$

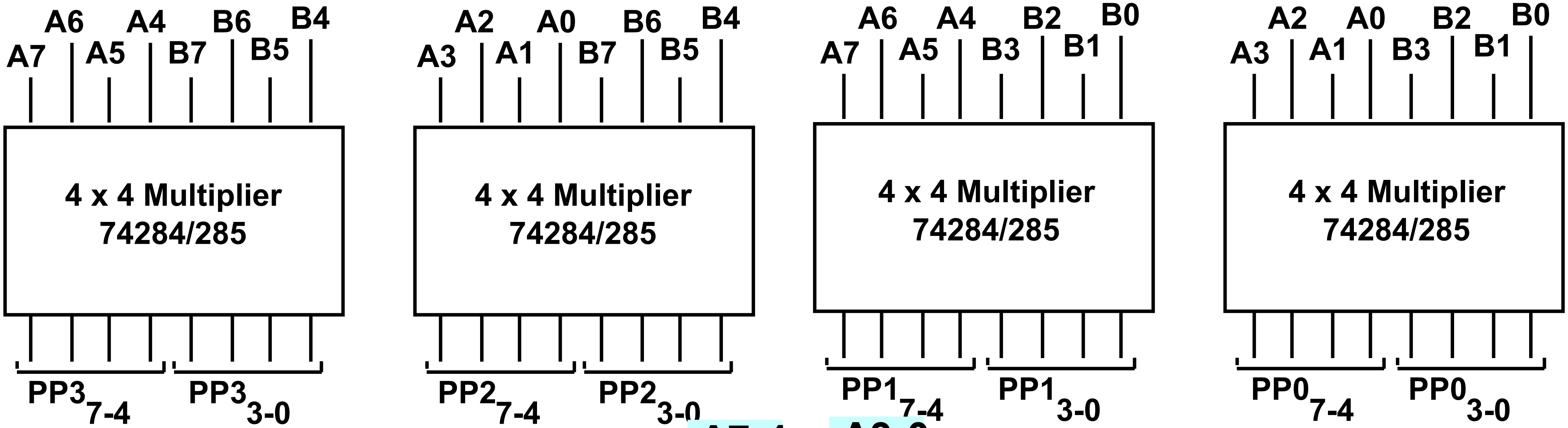
P7-4 = PP0$_{3-0}$ + PP1$_{3-0}$ + PP2$_{3-0}$ + Carry-in

P11-8 = PP1$_{7-4}$ + PP2$_{7-4}$ + PP3$_{3-0}$ + Carry-in

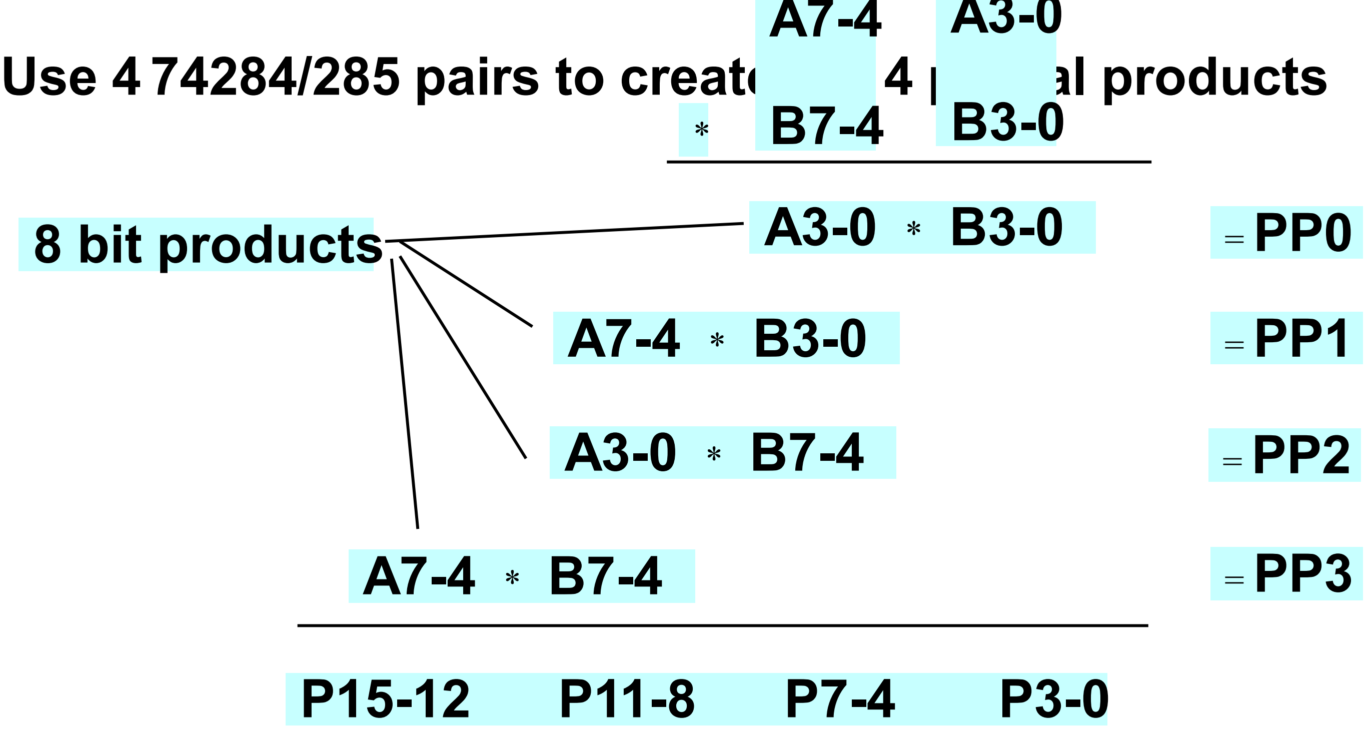
P15-12 = PP3$_{7-4}$ + Carry-in

Case Study: 8 x 8 Multiplier

Calculation of Partial Products



Use 4 74284/285 pairs to create 4 partial products



A7-4 A3-0
* B7-4 B3-0

8 bit products

A3-0 * B3-0 = PP0

A7-4 * B3-0 = PP1

A3-0 * B7-4 = PP2

A7-4 * B7-4 = PP3

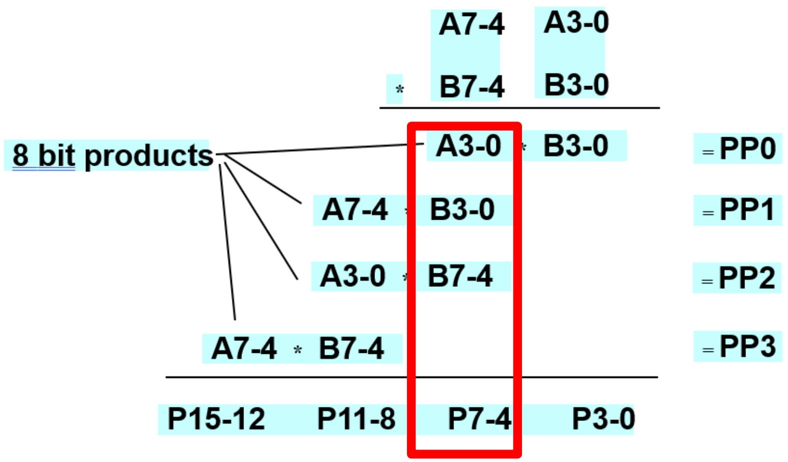
P15-12 P11-8 P7-4 P3-0

Case Study: 8 x 8 Multiplier

Three-At-A-Time Adder

Clever use of the Carry Inputs

Sum A[3-0], B[3-0], C[3-0]:



A7-4 A3-0

* B7-4 B3-0

8 bit products

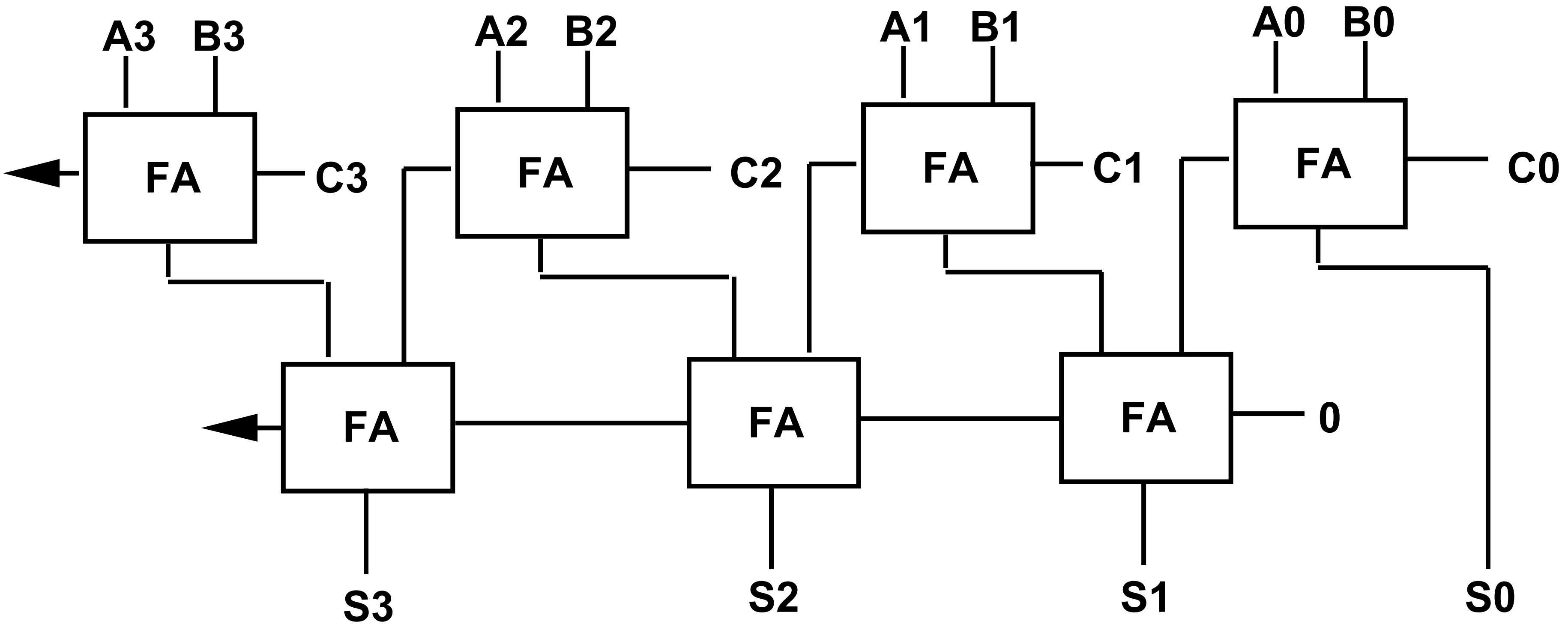
A3-0 * B3-0 = PP0

A7-4 * B3-0 = PP1

A3-0 * B7-4 = PP2

A7-4 * B7-4 = PP3

P15-12 P11-8 P7-4 P3-0

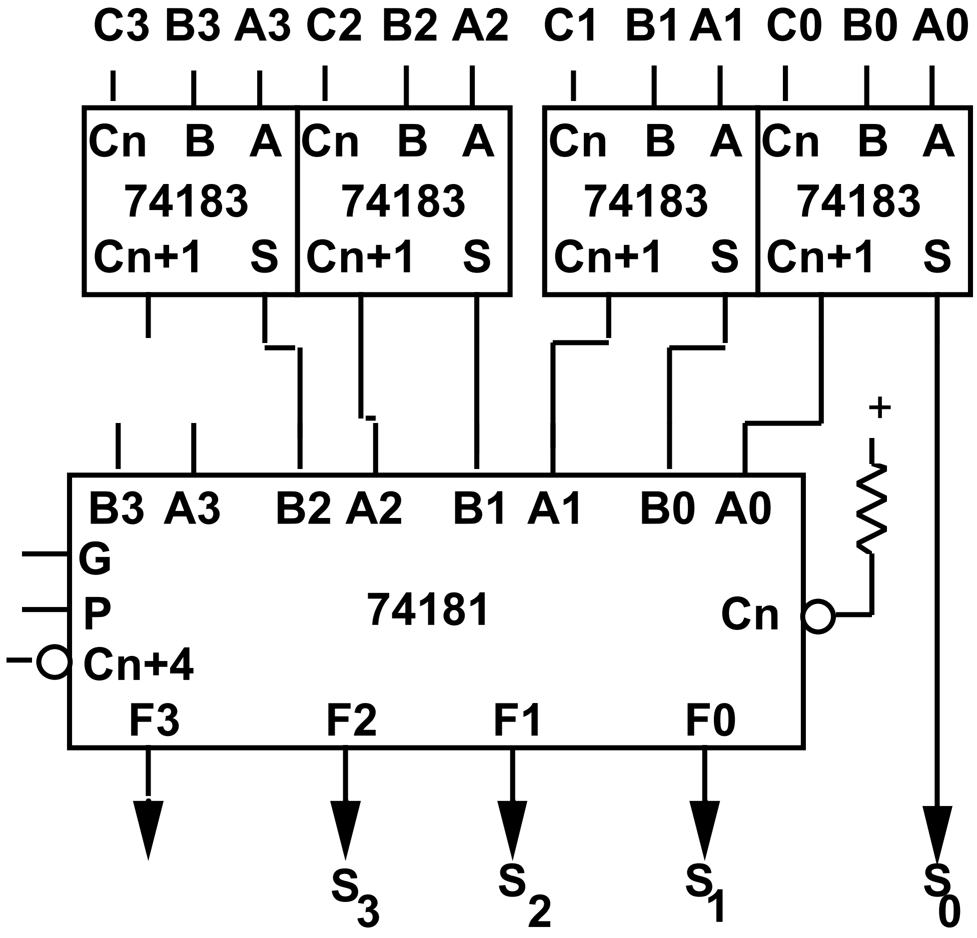


Two Level Full Adder Circuit

Note: Carry lookahead schemes also possible!

Case Study: 8 x 8 Multiplier

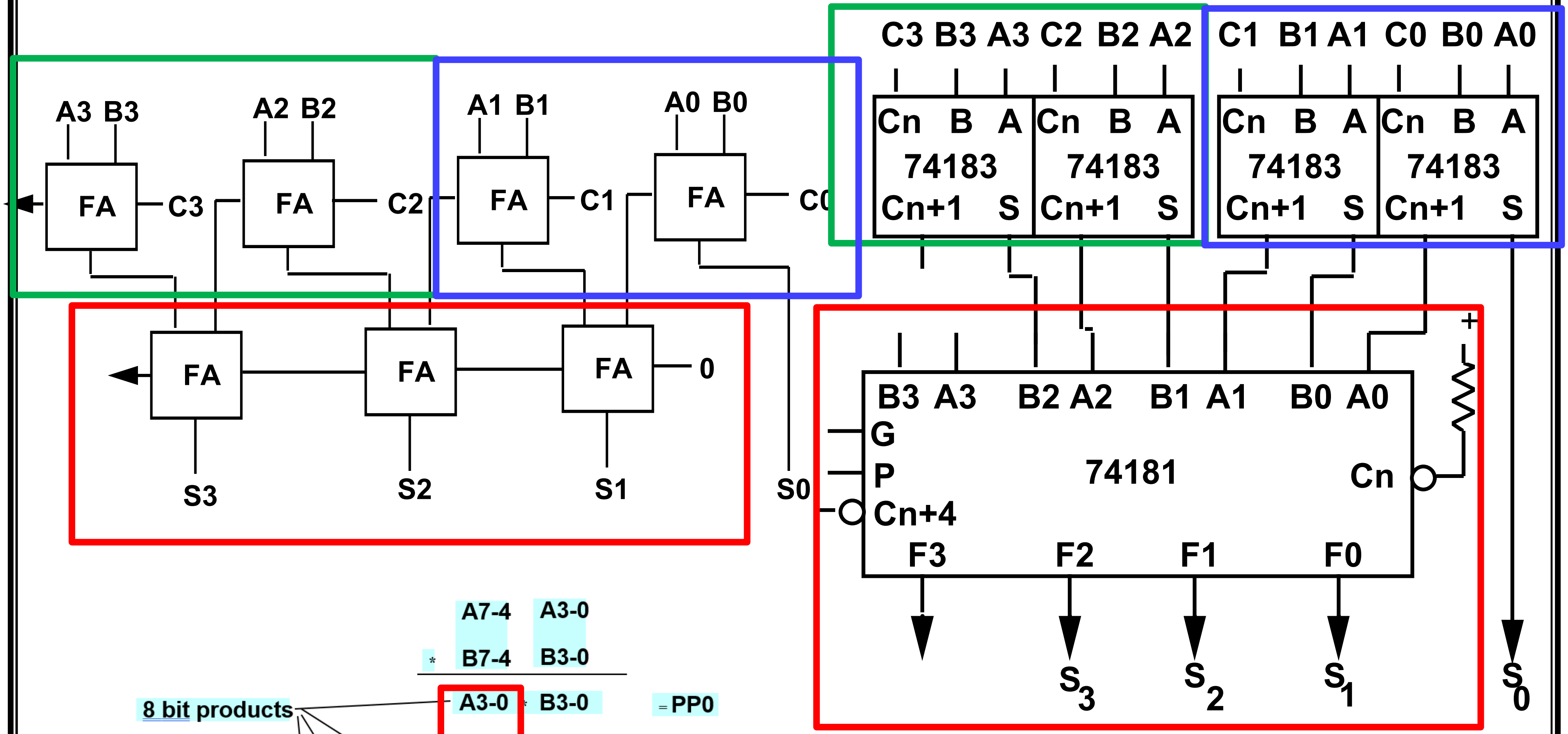
Three-At-A-Time Adder with TTL Components

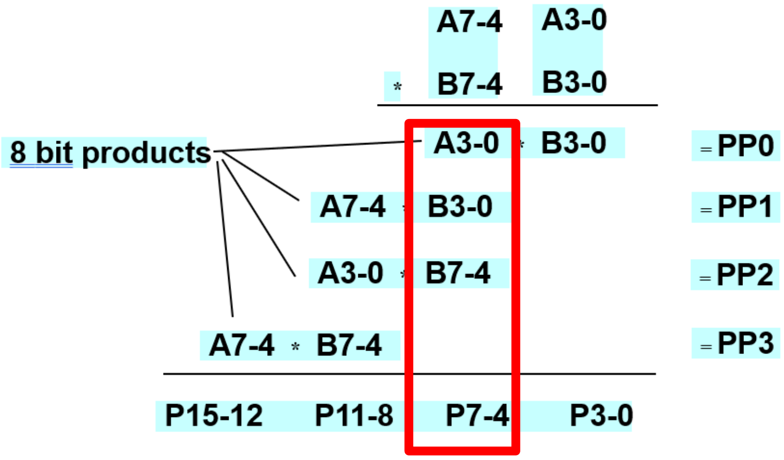


Full Adders
(2 per package)

Standard ALU configured as 4-bit cascaded adder
(with internal carry lookahead)

Note the off-set in the outputs





A7-4 A3-0

* B7-4 B3-0

8 bit products

A3-0 * B3-0 = PP0

A7-4 * B3-0 = PP1

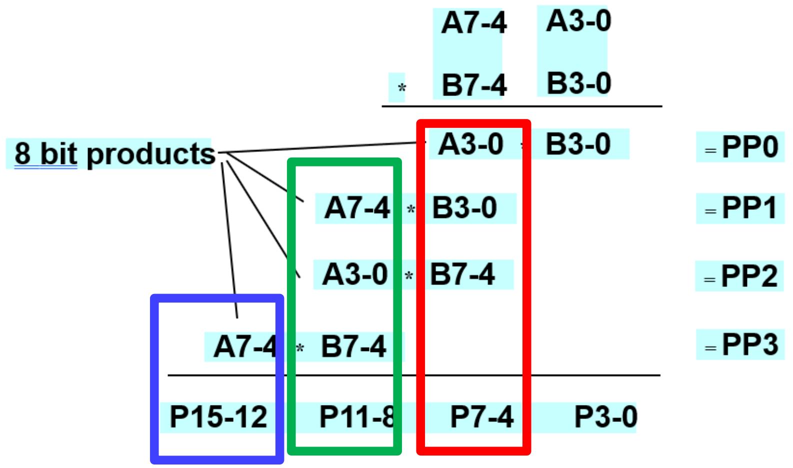
A3-0 * B7-4 = PP2

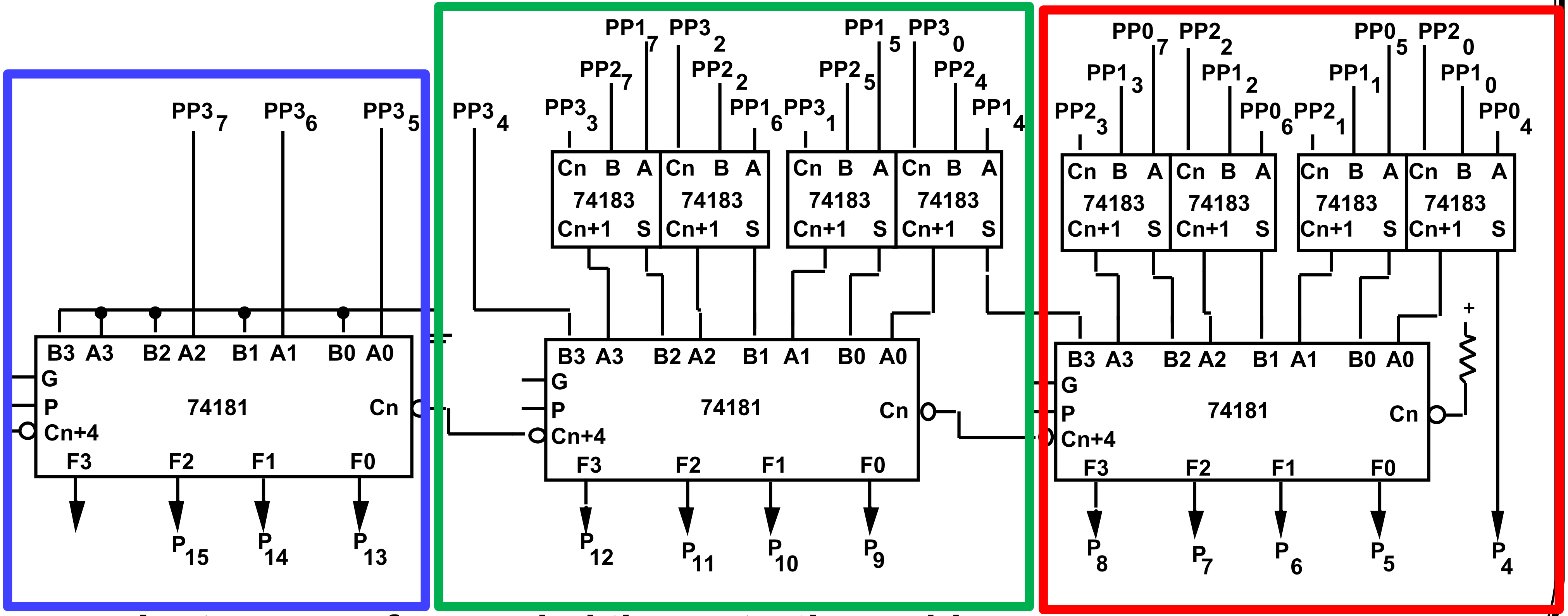
A7-4 * B7-4 = PP3

P15-12 P11-8 P7-4 P3-0

Case Study: 8 x 8 Multiplier

Accumulation of Partial Products

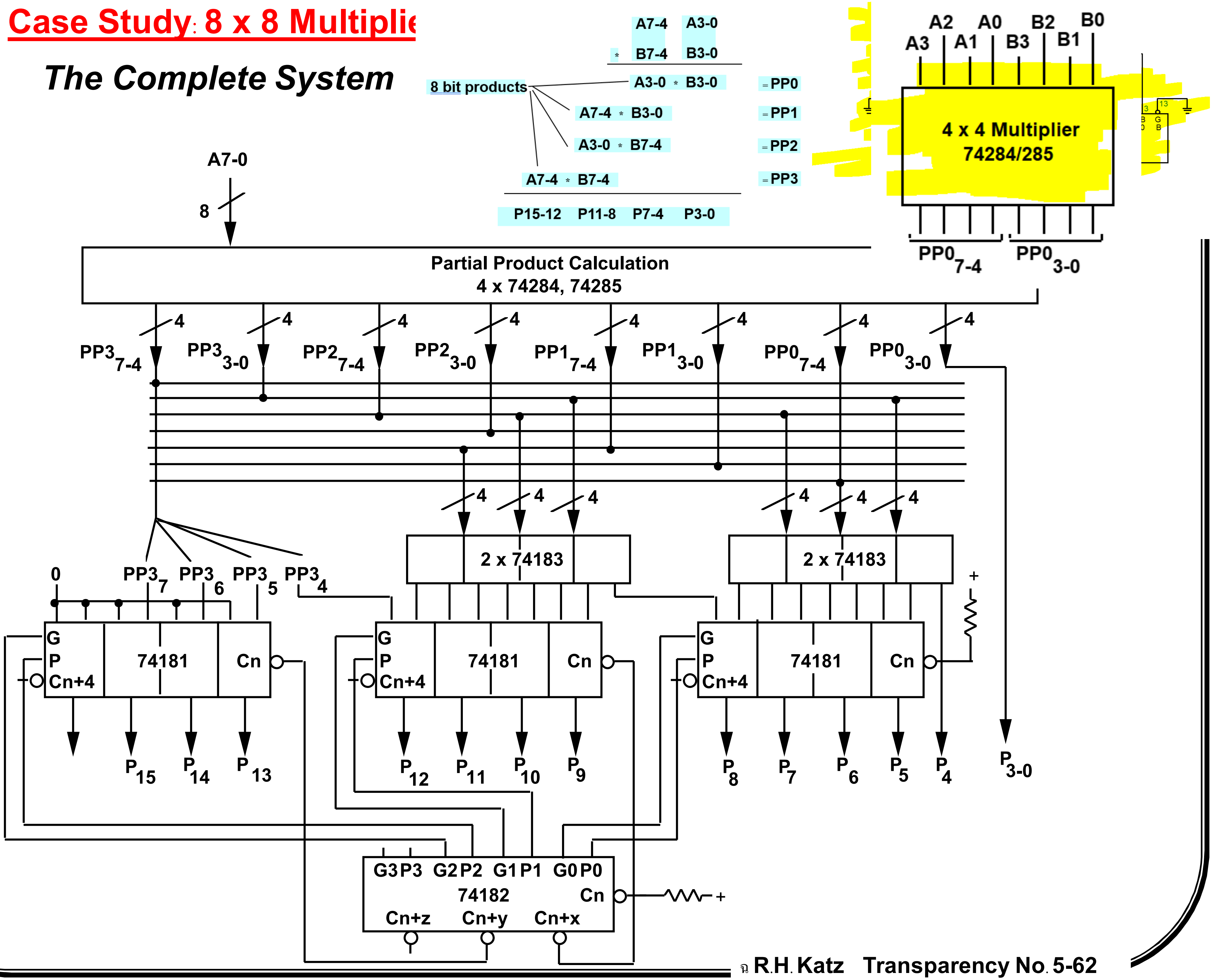




Just a case of cascaded three-at-a-time adders!

Case Study: 8 x 8 Multiplie

The Complete System



Case Study: 8 x 8 Multiplier

Package Count and Performance

4 74284/74285 pairs = 8 packages
4 74183, 3 74181, 1 74182 = 8 packages
16 packages total

Partial product calculation (74284/285) = 40 ns typ, 60 ns max

Intermediate sums (74183) = 9 ns/20ns = 15 ns average, 33 ns max

Second stage sums w/carry lookahead

74LS181: carry G and P = 20 ns typ, 30 ns max

74182: second level carries = 13 ns typ, 22 ns max

74LS181: formations of sums = 15 ns typ, 26 ns max

103 ns typ, 171 ns max

Chapter Review

We have covered:

- *Binary Number Representation*
  positive numbers the same
  difference is in how negative numbers are represented
  twos complement easiest to handle:
  one representation for zero, slightly
  complicated complementation, simple addition

- *Binary Networks for Additions*
  basic HA, FA
  carry lookahead logic

- *ALU Design*
  specification and implementation

- *BCD Adders*
  Simple extension of binary adders

- *Multipliers*
  4 x 4 multiplier: partial product accumulation
  extension to 8 x 8 case