Arithmetic Circuits

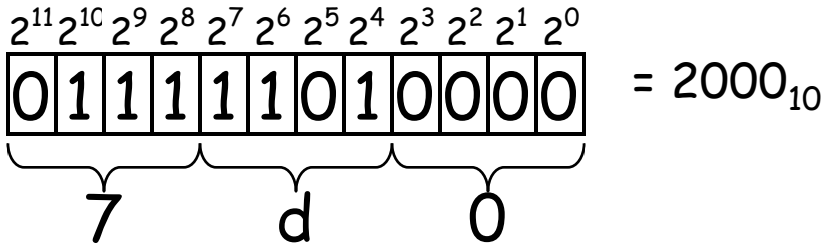
- Number representations
- Addition, subtraction
- Performance issues
  -- ripple carry
  -- carry bypass
  -- carry skip
  -- carry lookahead

Lab #3 due Thursday, report next Tuesday, no LPSets this week

Encoding numbers

It is straightforward to encode positive integers as a sequence of bits. Each bit is assigned a weight. Ordered from right to left, these weights are increasing powers of 2. The value of an n-bit number encoded in this fashion is given by the following formula:

$$v = \sum_{i=0}^{n-1} 2^i b_i$$



$= 2000_{10}$

Oftentimes we will find it convenient to cluster groups of bits together for a more compact notation. Two popular groupings are clusters of 3 bits and 4 bits.

03720

Octal - base 8

000 - 0
001 - 1
010 - 2
011 - 3
100 - 4
101 - 5
110 - 6
111 - 7

0x7d0

Hexadecimal - base 16

0000 - 0	1000 - 8
0001 - 1	1001 - 9
0010 - 2	1010 - a
0011 - 3	1011 - b
0100 - 4	1100 - c
0101 - 5	1101 - d
0110 - 6	1110 - e
0111 - 7	1111 - f

Binary Representation of Numbers

How to represent negative numbers?

- Three common schemes:
  - sign-magnitude, ones complement, twos complement
- Sign-magnitude: MSB = 0 for positive, 1 for negative
  - Range: $-(2^{N-1} - 1)$ to $+(2^{N-1} - 1)$
  - Two representations for zero: 0000… & 1000…
  - Simple multiplication but complicated addition/subtraction
- Ones complement: if N is positive then its negative is $\overline{N}$
  - Example: 0111 = 7, 1000 = -7
  - Range: $-(2^{N-1} - 1)$ to $+(2^{N-1} - 1)$
  - Two representations for zero: 0000… & 1111…
  - Subtraction is addition followed by ones complement

Representing negative integers

To keep our arithmetic circuits simple, we'd like to find a representation for negative numbers so that we can use a single operation (binary addition) when we wish to find the sum of two integers, independent of whether they are positive are negative.

We certainly want A + (-A) = 0. Consider the following 8-bit binary addition where we only keep 8 bits of the result:

```
  11111111
+ 00000001
  --------
  00000000
```

which implies that the 8-bit representation of -1 is 11111111. More generally

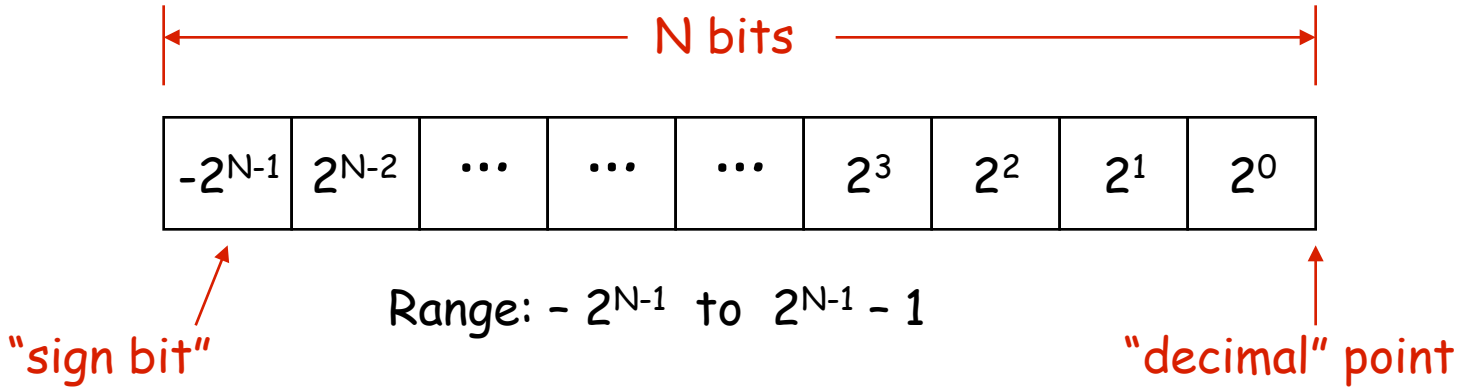
Negation: Complement and add 1

$$
\begin{aligned}
-A &= 0 - A \\
&= (-1 + 1) - A \\
&= (-1 - A) + 1 \\
&= \sim A + 1
\end{aligned}
$$

~ means bit-wise complement

$$
\begin{array}{rcccccccc}
 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
- & A_7 & A_6 & A_5 & A_4 & A_3 & A_2 & A_1 & A_0 \\
\hline
 & \overline{A_7} & \overline{A_6} & \overline{A_5} & \overline{A_4} & \overline{A_3} & \overline{A_2} & \overline{A_1} & \overline{A_0}
\end{array}
$$

Signed integers: 2's complement



8-bit 2's complement example:

$11010110 = -2^7 + 2^6 + 2^4 + 2^2 + 2^1 = -128 + 64 + 16 + 4 + 2 = -42$

If we use a two's complement representation for signed integers, the same binary addition mod 2^n procedure will work for adding positive and negative numbers (don't need separate subtraction rules). The same procedure will also handle unsigned numbers!

By moving the implicit location of "decimal" point, we can represent fractions too:

$1101.0110 = -2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3} = -8 + 4 + 1 + 0.25 + 0.125 = -2.625$

Sign extension

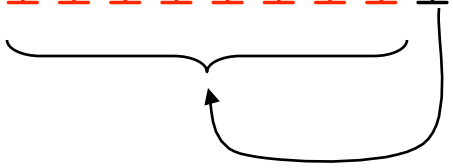
Consider the 8-bit 2's complement representation of:

```
42 = 00101010        -5 = ~00000101 + 1
                          =  11111010 + 1
                          =  11111011
```

What is their 16-bit 2's complement representation?

```
42 = 0000000000101010

-5 = 1111111111111011
```



Extend the MSB (aka the "sign bit") into the higher-order bit positions

Adder: a circuit that does addition

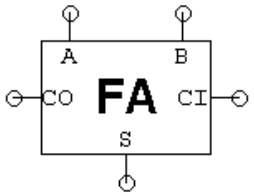
Here's an example of binary addition as one might do it by "hand":

Carries from previous column

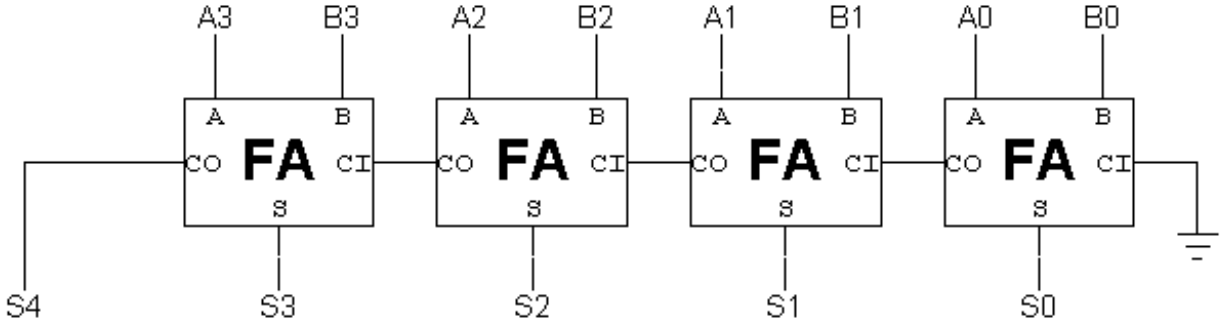
```
  1 1 0 1
    1101
+   0101
-------
   10010
```

Adding two N-bit numbers produces an (N+1)-bit result

If we build a circuit that implements one column:

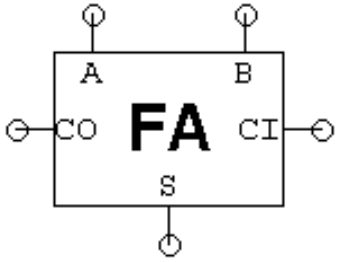


we can quickly build a circuit two add two 4-bit numbers…



"Ripple-carry adder"

"Full Adder" building block



The "half adder" circuit has only the A and B inputs

A	B	C	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

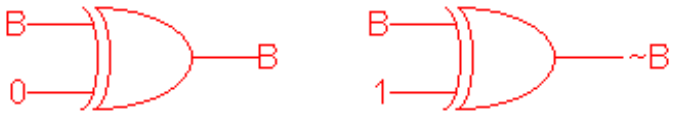
$$S = A \oplus B \oplus C$$

$$\begin{aligned} CO &= \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC \\ &= (\overline{A} + A)BC + (\overline{B} + B)AC + AB(\overline{C} + C) \\ &= BC + AC + AB \end{aligned}$$

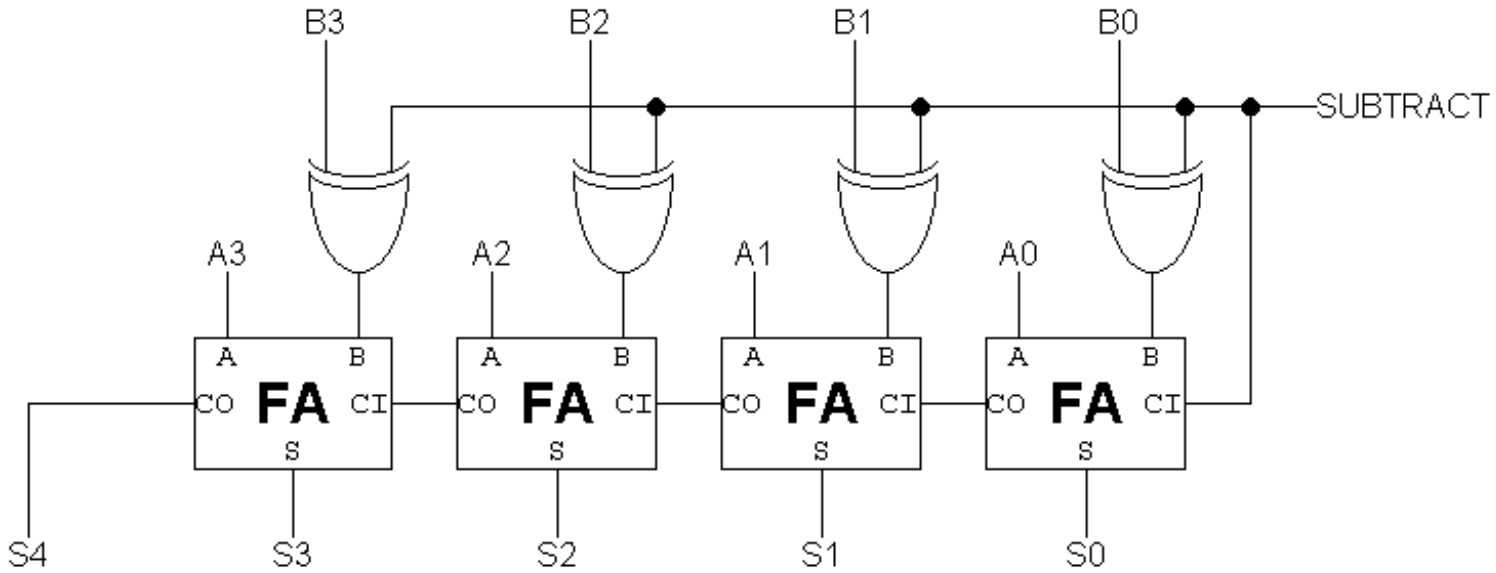
Subtraction: A-B = A + (-B)

Using 2's complement representation: -B = ~B + 1

~ = bit-wise complement



So let's build an arithmetic unit that does both addition and subtraction. Operation selected by *control input*:



Condition Codes

Besides the sum, one often wants four other bits of information from an arithmetic unit:

Z (zero): result is = 0 *big NOR gate*

N (negative): result is < 0 S_{N-1}

C (carry): indicates an add in the most significant position produced a carry, e.g., 1111 + 0001 *from last FA*

V (overflow): indicates that the answer has too many bits to be represented correctly by the result width, e.g., 0111 + 0111

$$V = A_{N-1}B_{N-1}\overline{S_{N-1}} + \overline{A_{N-1}}\,\overline{B_{N-1}}S_{N-1}$$

$$V = COUT_{N-1} \oplus CIN_{N-1}$$

To compare A and B, perform A-B and use condition codes:

Signed comparison:

LT	N⊕V
LE	Z+(N⊕V)
EQ	Z
NE	~Z
GE	~(N⊕V)
GT	~(Z+(N⊕V))

Unsigned comparison:

LTU	~C
LEU	~C+Z
GEU	C
GTU	~(~C+Z)

Condition Codes in Verilog

Z (zero): result is = 0

N (negative): result is < 0

C (carry): indicates an add in the most significant position produced a carry, e.g., 1111 + 0001

V (overflow): indicates that the answer has too many bits to be represented correctly by the result width, e.g., 0111 + 0111

```
wire [31:0] a,b,s;
wire z,n,v,c;
assign {c,s} = a + b;
assign z = ~|s;
assign n = s[31];
assign v = a[31]^b[31]^s[31]^c;
```

Might be better to use sum-of-products formula for V from previous slide if using LUT implementation (only 3 variables instead of 4).

Modular Arithmetic

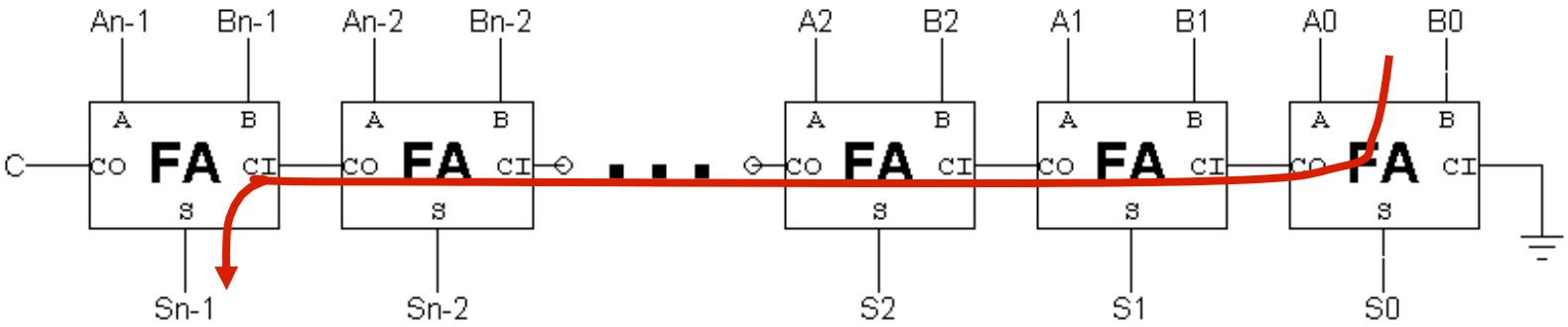
The Verilog arithmetic operators (+,-,*) all produce full-precision results, e.g., adding two 8-bit numbers produces a 9-bit result.

In many designs one chooses a "word size" (many computers use 32 or 64 bits) and all arithmetic results are truncated to that number of bits, i.e., arithmetic is performed modulo $2^{\text{word size}}$.

Using a fixed word size can lead to *overflow*, e.g., when the operation produces a result that's too large to fit in the word size. One can

- Avoid overflow: choose a sufficiently large word size
- Detect overflow: have the hardware remember if an operation produced an overflow – trap or check status at end
- Embrace overflow: sometimes this is exactly what you want, e.g., when doing index arithmetic for circular buffers of size 2^N.
- "Correct" overflow: replace result with most positive or most negative number as appropriate, aka *saturating arithmetic*. Good for digital signal processing.

Speed: t_{PD} of Ripple-carry Adder



Worse-case path: carry propagation from LSB to MSB, e.g., when adding 11…111 to 00…001.

$$t_{PD} = (N-1)*(t_{PD,OR} + t_{PD,AND}) + t_{PD,XOR} \approx \Theta(N)$$

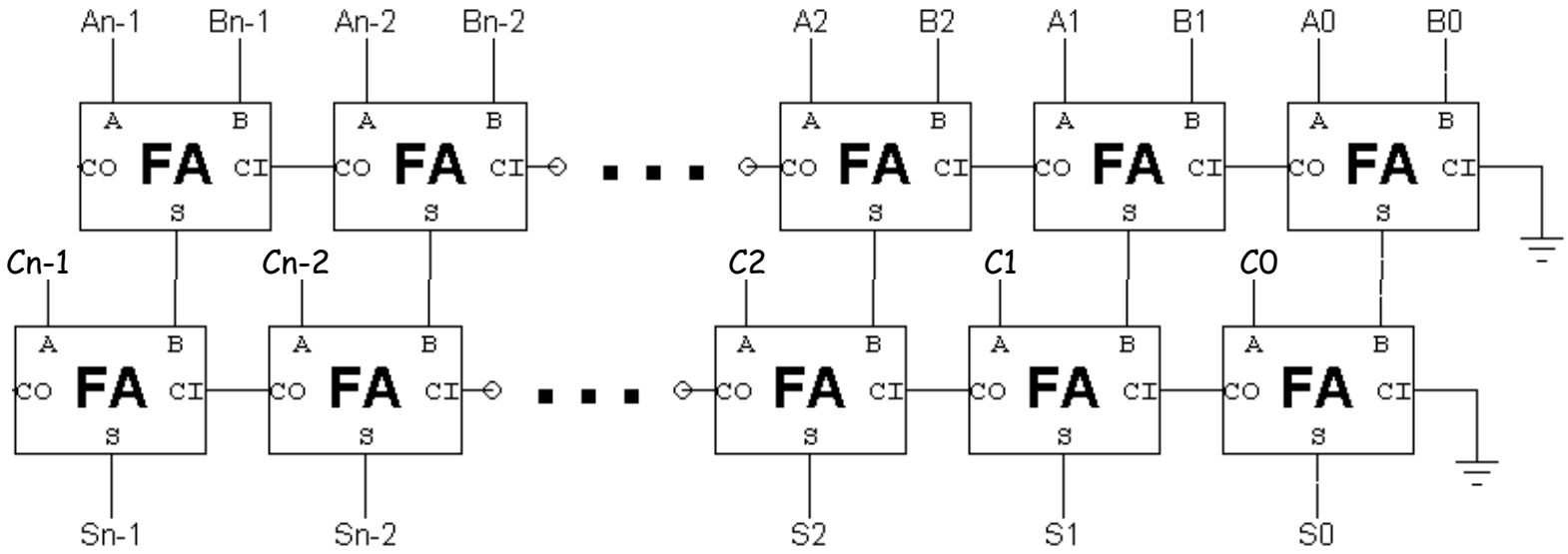
$(t_{PD,OR} + t_{PD,AND})$: CI to CO

$t_{PD,XOR}$: CI_{N-1} to S_{N-1}

$$t_{adder} = (N-1)t_{carry} + t_{sum}$$

$\Theta(N)$ is read "order N" : means that the latency of our adder grows at worst in proportion to the number of bits in the operands.

How about the t_{PD} of this circuit?



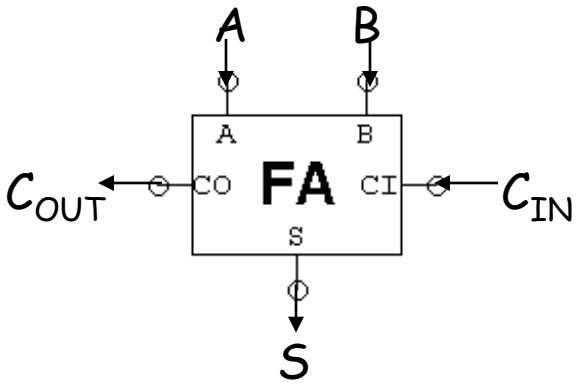
Is the t_{PD} of this circuit = 2 * $t_{PD,N\text{-}BIT\ RIPPLE}$?

Nope! t_{PD} of this circuit = $t_{PD,N\text{-}BIT\ RIPPLE}$ + $t_{PD,FA}$!!!

— Timing analysis is tricky!

Faster carry logic

Let's see if we can improve the speed by rewriting the equations for C_{OUT}:



$$C_{OUT} = AB + AC_{IN} + BC_{IN}$$
$$= AB + (A + B)C_{IN}$$
$$= G + P\,C_{IN}$$

generate propagate

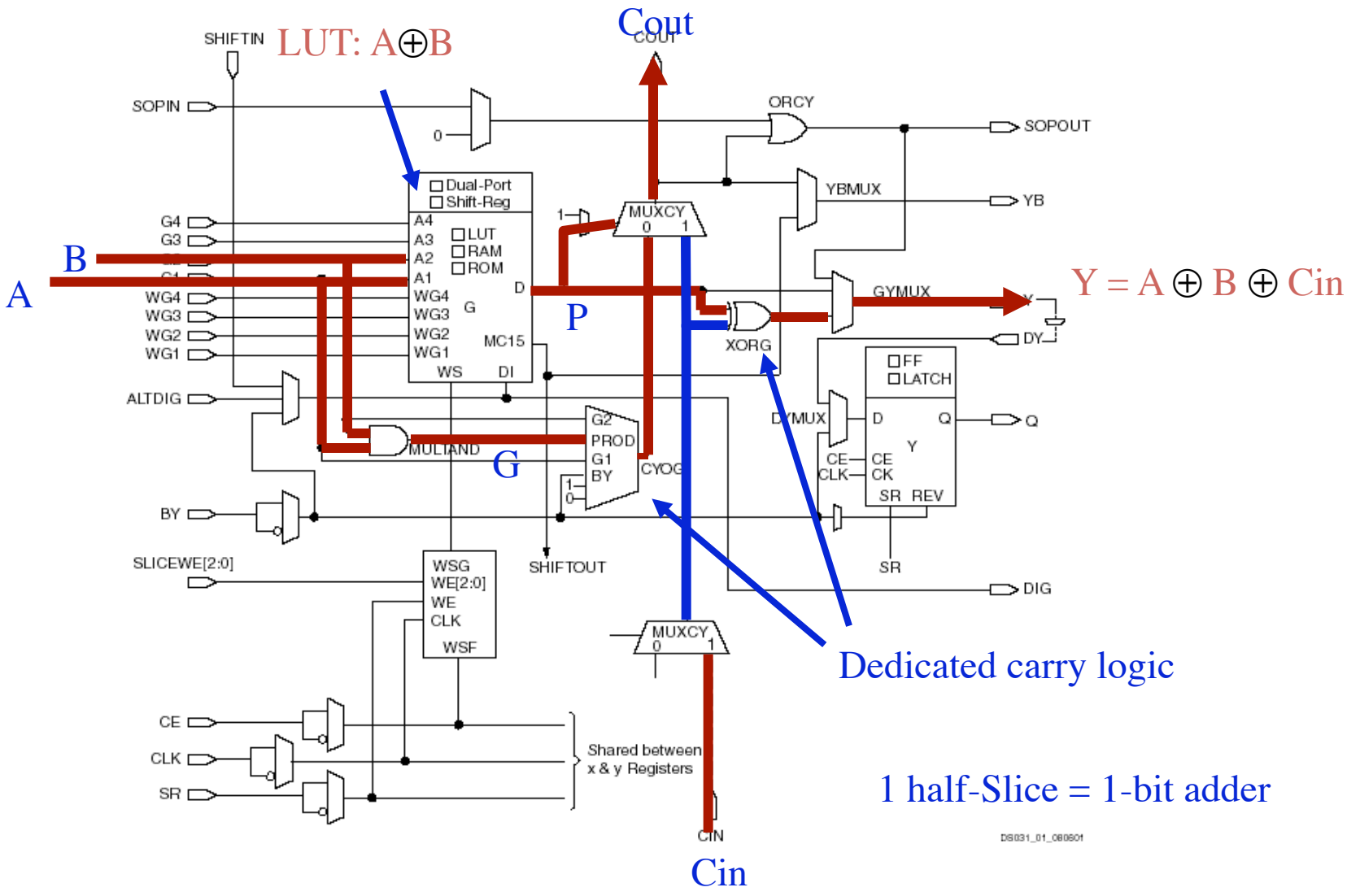
where $G = AB$
$P = A + B$

```
module fa(input a,b,cin, output s,cout);
  wire g = a & b;
  wire p = a ^ b;
  assign s = p ^ cin;
  assign cout = g | (p & cin);
endmodule
```

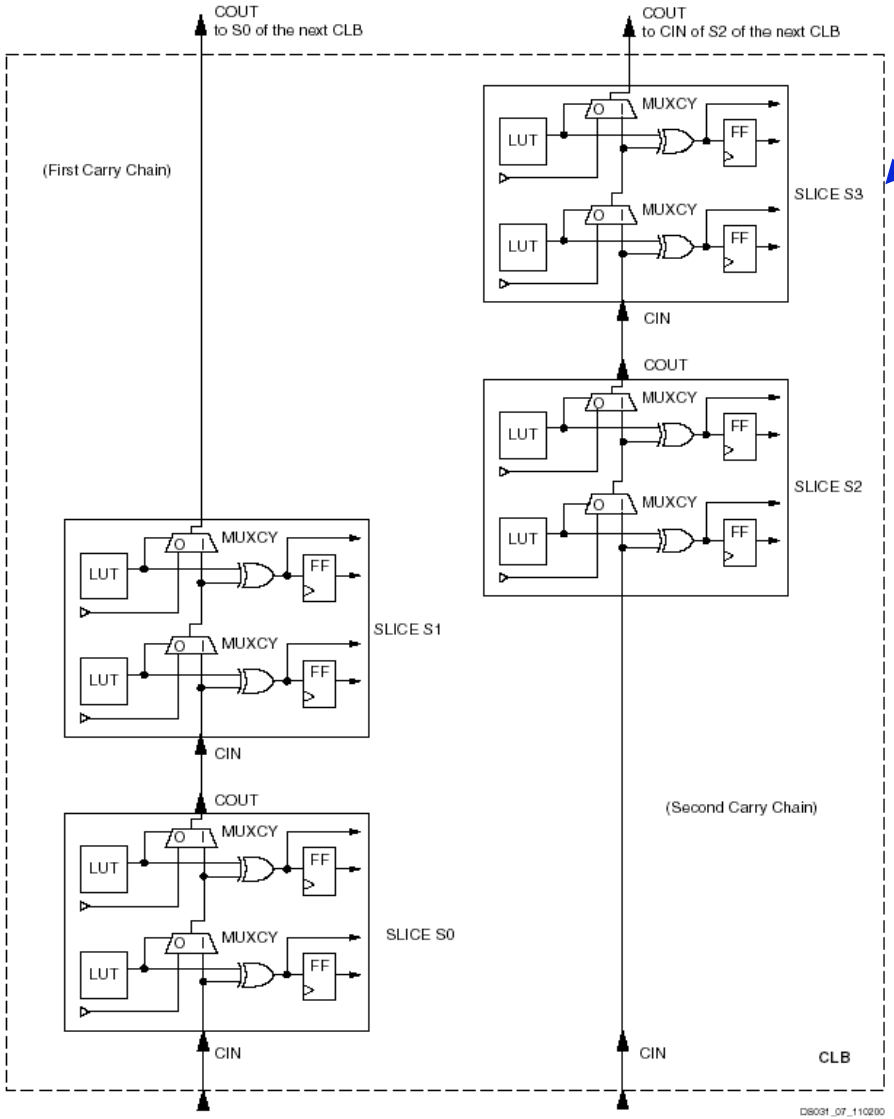
Actually, P is usually defined as P = A^B which won't change C_{OUT} but will allow us to express S as a simple function :

$S = P \hat{} C_{IN}$

Virtex II Adder Implementation

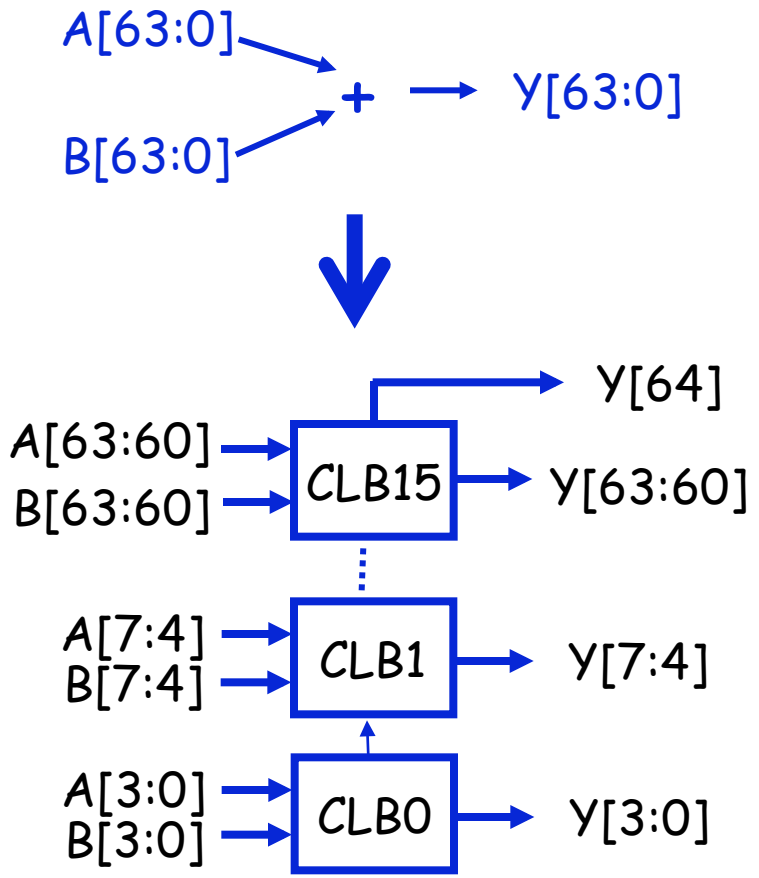


Virtex II Carry Chain



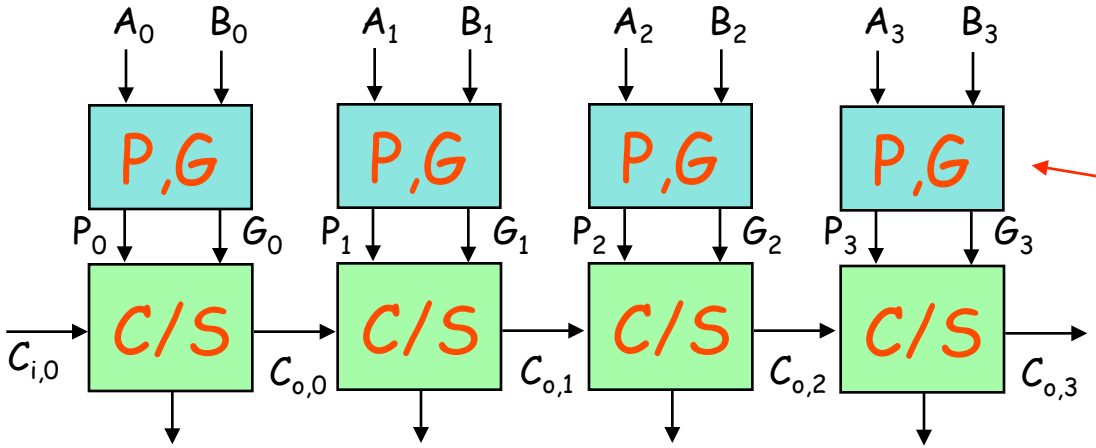
1 CLB = 4 Slices = 2, 4-bit adders

64-bit Adder: 16 CLBs

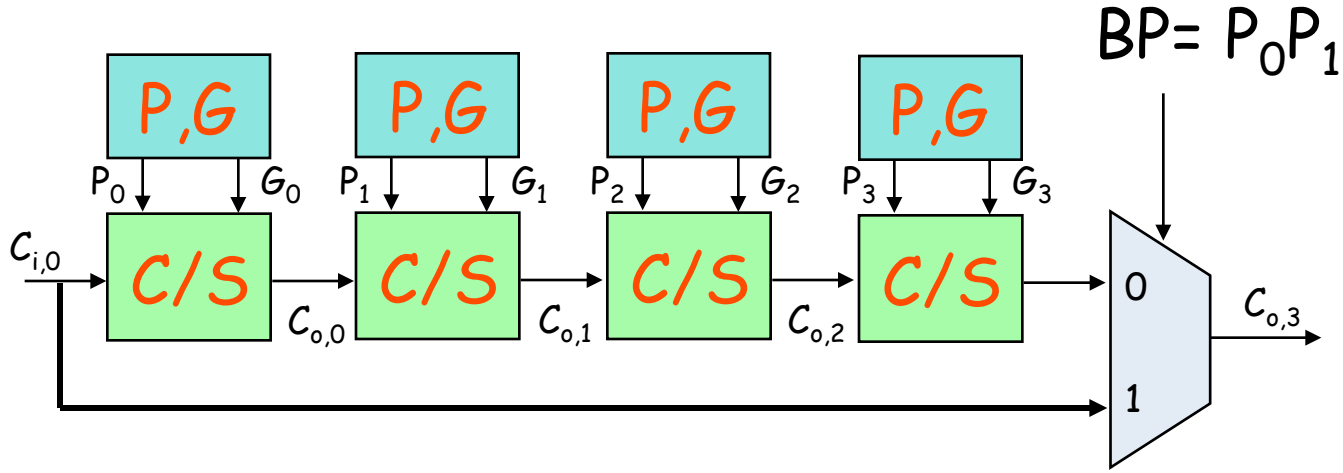


CLBs must be in same column

Carry Bypass Adder

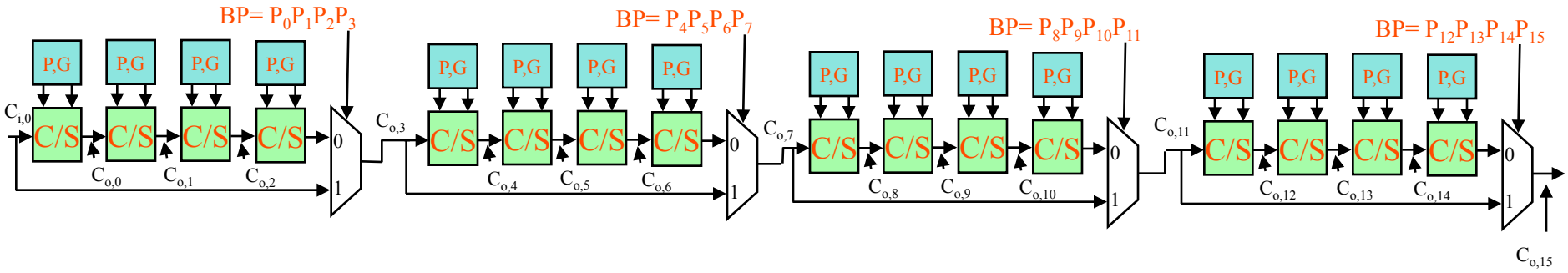


Can compute P, G in parallel for all bits



Key Idea: if (P_0 P_1 P_2 P_3) then $C_{o,3} = C_{i,0}$

16-bit Carry Bypass Adder



What is the worst case propagation delay for the 16-bit adder?

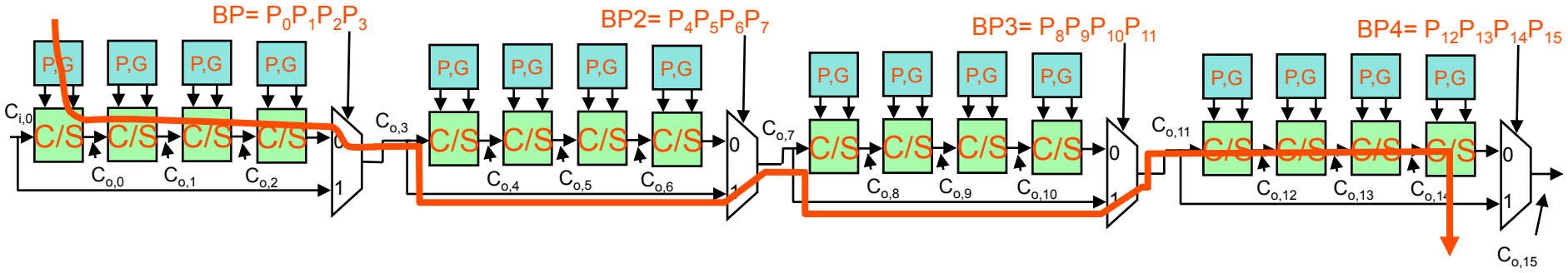
Assume the following for delay each gate:

P, G from A, B: 1 delay unit

P, G, C_i to C_o or Sum for a C/S: 1 delay unit

2:1 mux delay: 1 delay unit

Critical Path Analysis



For the second stage, is the critical path:

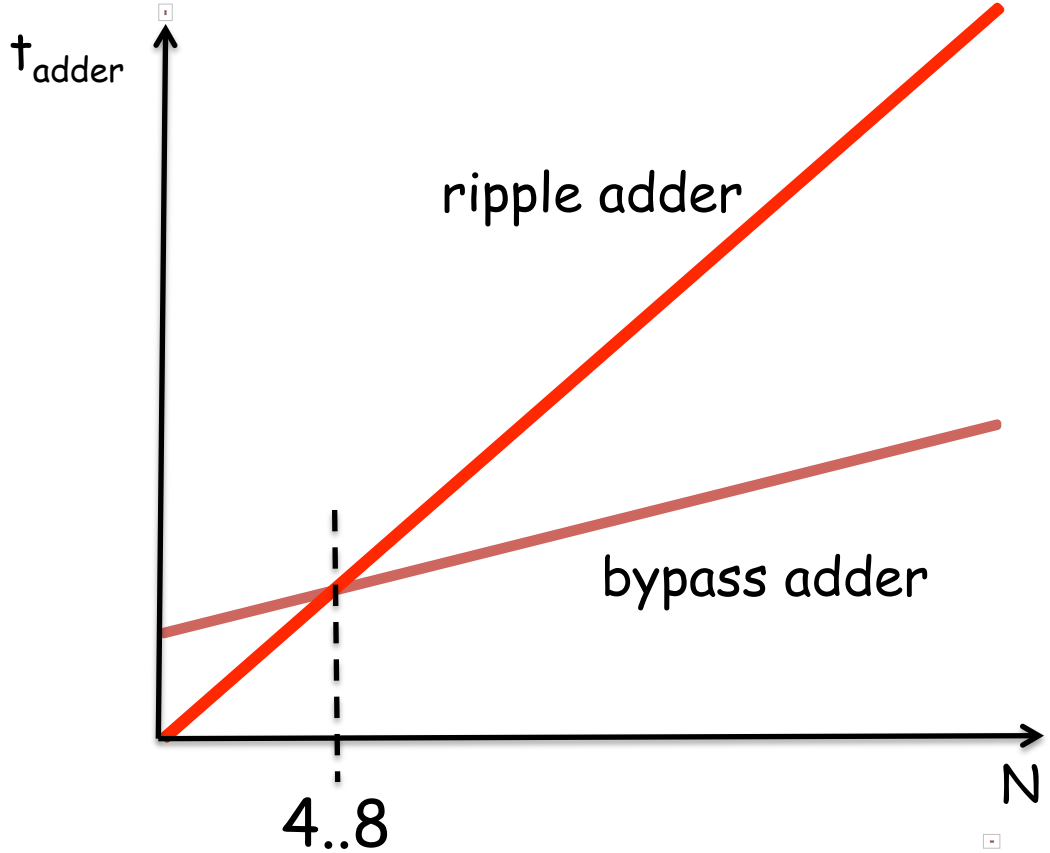
BP2 = 0 or BP2 = 1 ?

Message: Timing analysis is very tricky –
Must carefully consider data dependencies for *false paths*

Carry Bypass vs Ripple Carry

Ripple Carry: $t_{adder} = (N-1)\, t_{carry} + t_{sum}$

Carry Bypass: $t_{adder} = 2(M-1)\, t_{carry} + t_{sum} + (N/M-1)\, t_{bypass}$



M = bypass word size

N = number of bits being added

Carry Lookahead Adder (CLA)

- Recall that $C_{OUT} = G + P\,C_{IN}$ where G = A&B and P = A^B

- For adding two N-bit numbers:

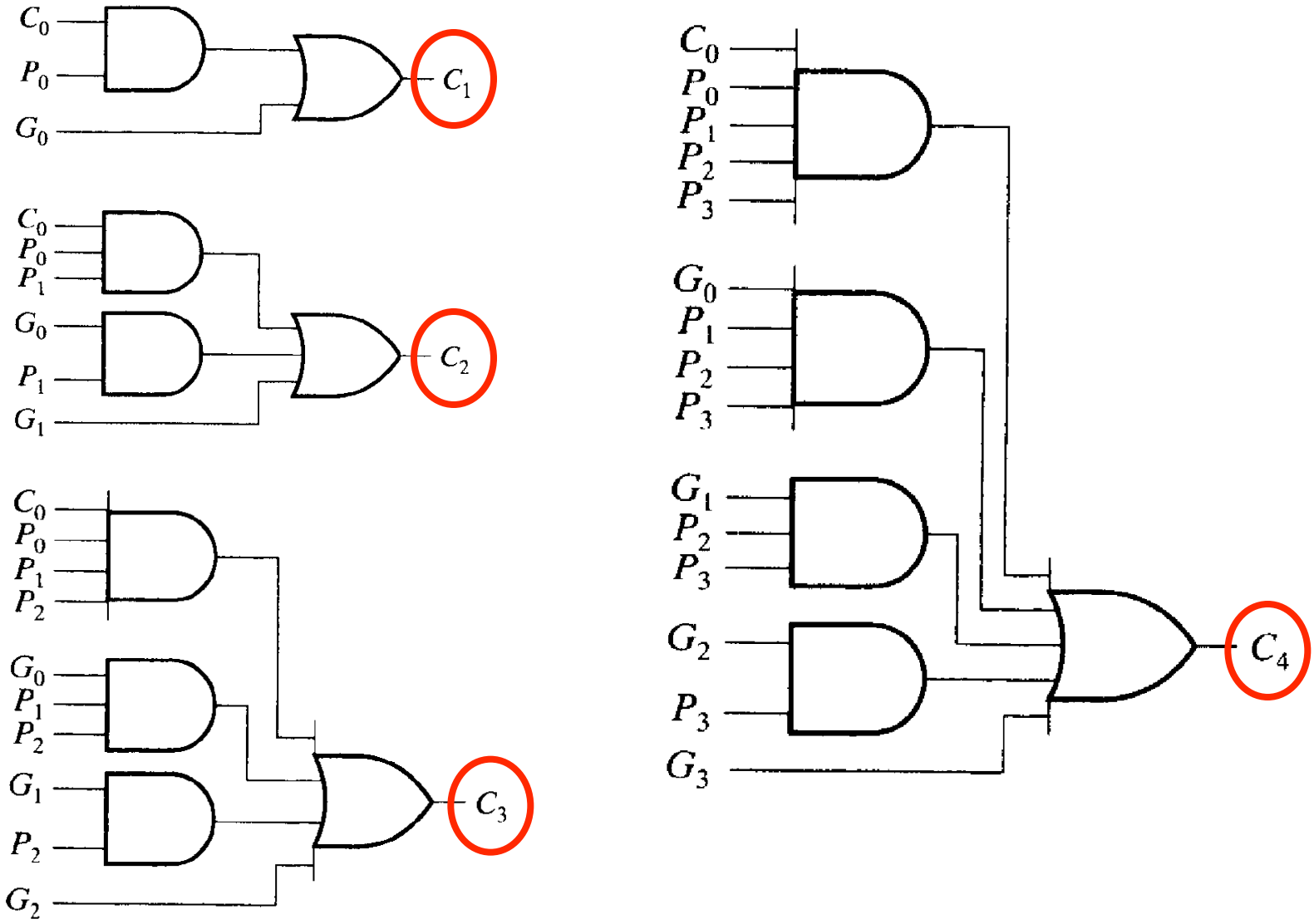
$$C_N = G_{N-1} + P_{N-1}C_{N-1}$$
$$= G_{N-1} + P_{N-1}G_{N-2} + P_{N-1}P_{N-2}C_{N-2}$$
$$= G_{N-1} + P_{N-1}G_{N-2} + P_{N-1}P_{N-2}G_{N-3} + \ldots + P_{N-1}\ldots P_0 C_{IN}$$

C_N in only 3 gate delays* :
1 for P/G generation, 1 for ANDs, 1 for final OR

*assuming gates with N inputs

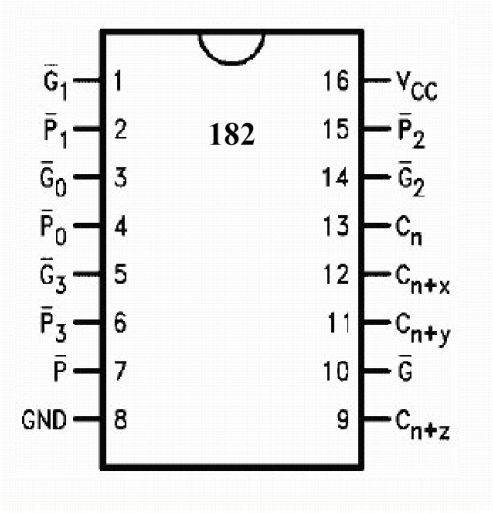
- Idea: pre-compute all carry bits as f(Gs,Ps,C_{IN})

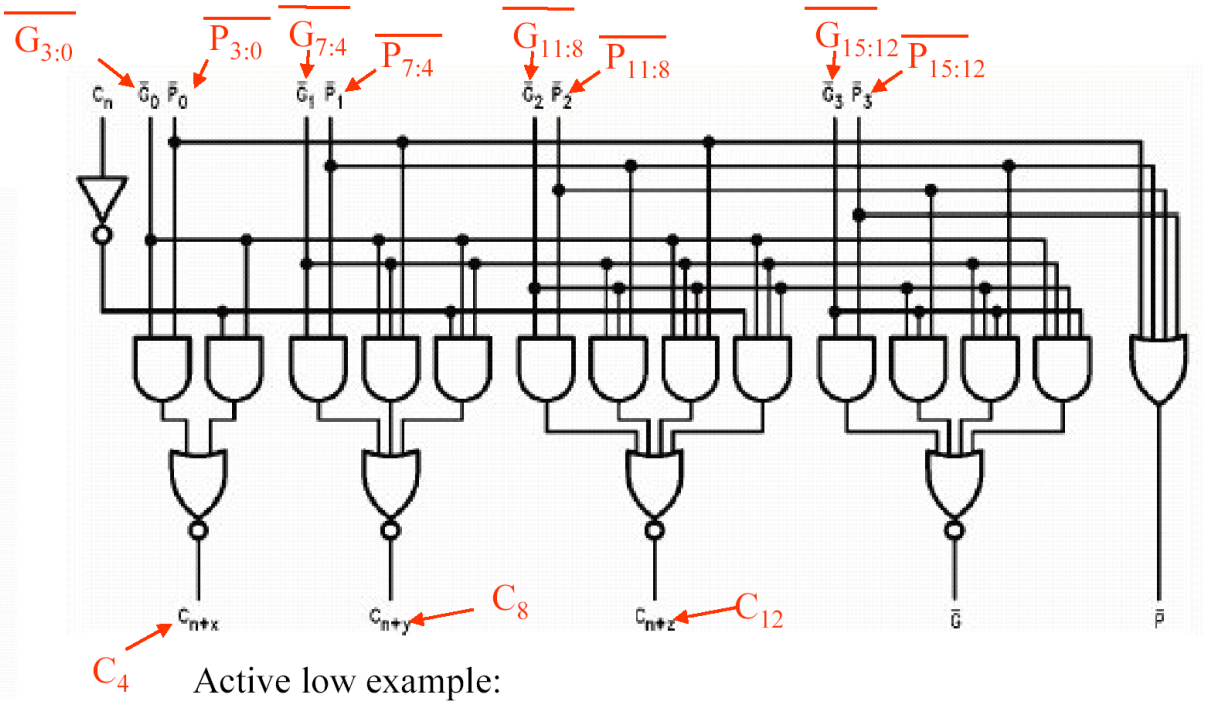
Carry Lookahead Circuits



The 74182 Carry Lookahead Unit

74182 carry lookahead unit





- high speed carry lookahead generator
- used with 74181 to extend carry lookahead beyond 4 bits
- correctly handles the carry polarity of the 181

Active low example:

$$C_{n+x} = \overline{\overline{G0}\cdot\overline{P0} + \overline{G0}\cdot\overline{C_n}}$$

$$= \overline{\overline{G0}\cdot\overline{P0}\cdot\overline{G0}\cdot\overline{C_n}}$$

$$= (G0+P0)\cdot(G0+C_n) = G0 + P0C_n$$

➢ $C_4 = G_{3:0} + P_{3:0}C_n$

$$C_{n+y} = C_8 = G_{7:4} + P_{7:4}G_{3:0} + P_{7:4}P_{3:0}C_{i,0} = G_{7:0} + P_{7:0}C_n$$

$$C_{n+z} = C_{12} = G_{11:8} + P_{11:8}G_{7:4} + P_{11:8}P_{7:4}G_{3:0} + P_{11:8}P_{7:4}P_{3:0}C_n$$

$$= G_{11:0} + P_{11:0}C_n$$

Block Generate and Propagate

G and P can be computed for groups of bits (instead of just for individual bits). This allows us to choose the maximum fan-in we want for our logic gates and then build a hierarchical carry chain using these equations:

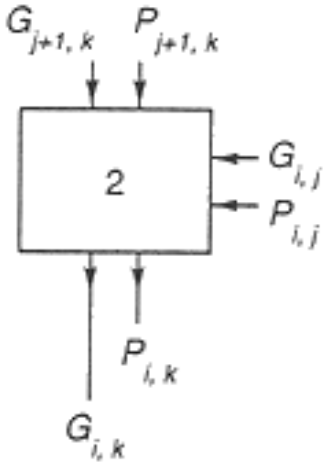
$$C_{J+1} = G_{IJ} + P_{IJ}C_I$$

$$G_{IK} = G_{J+1,K} + P_{J+1,K}\,G_{IJ}$$

$$P_{IK} = P_{IJ}\,P_{J+1,K}$$

where I < J and J+1 < K

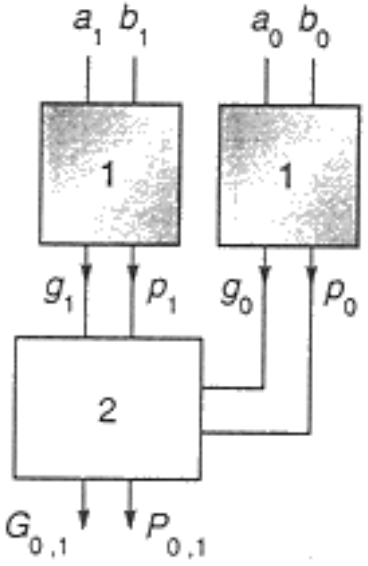
"generate a carry from bits I thru K if it is generated in the high-order (J+1,K) part of the block or if it is generated in the low-order (I,J) part of the block and then propagated thru the high part"



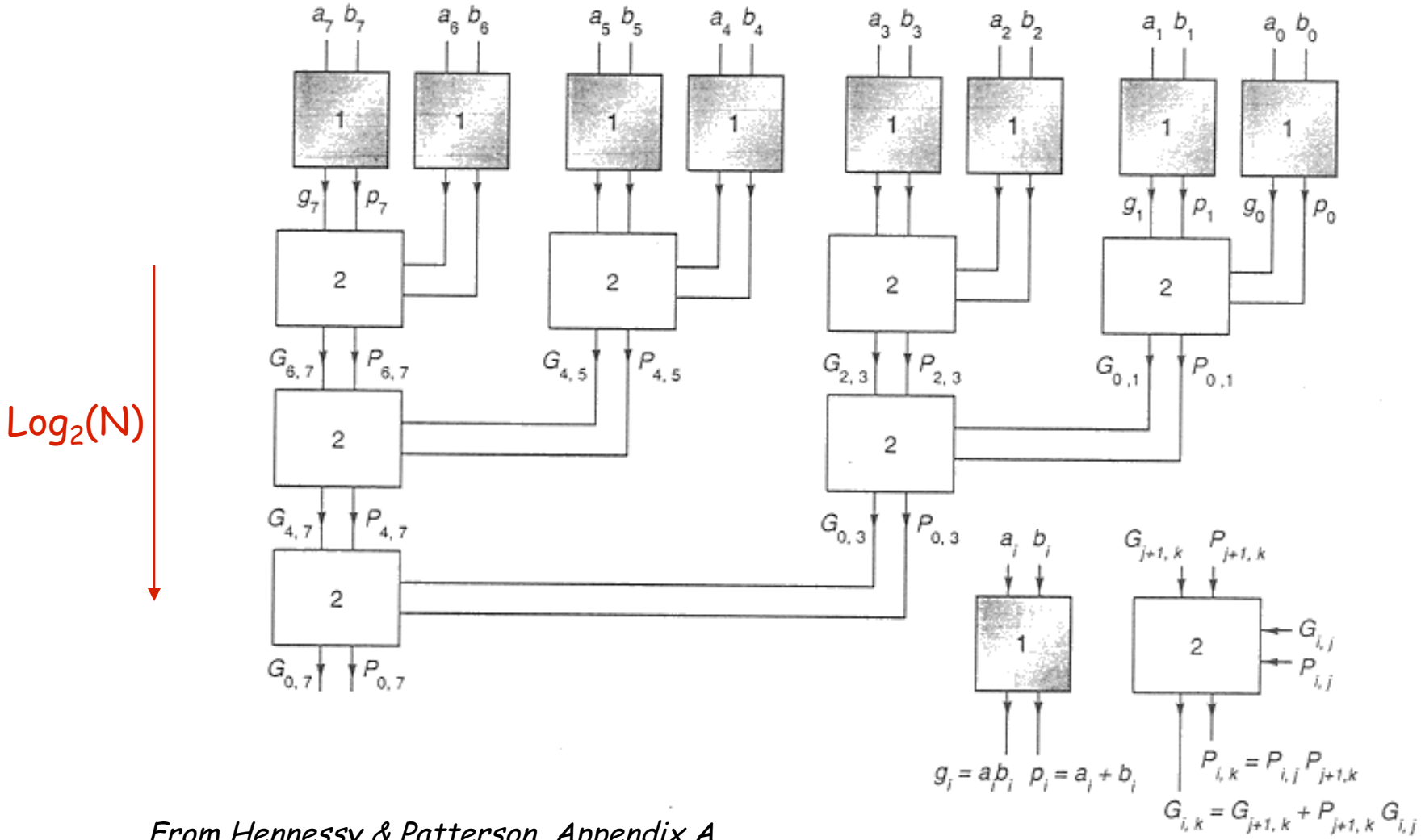
Hierarchical building block

P/G generation

1st level of lookahead

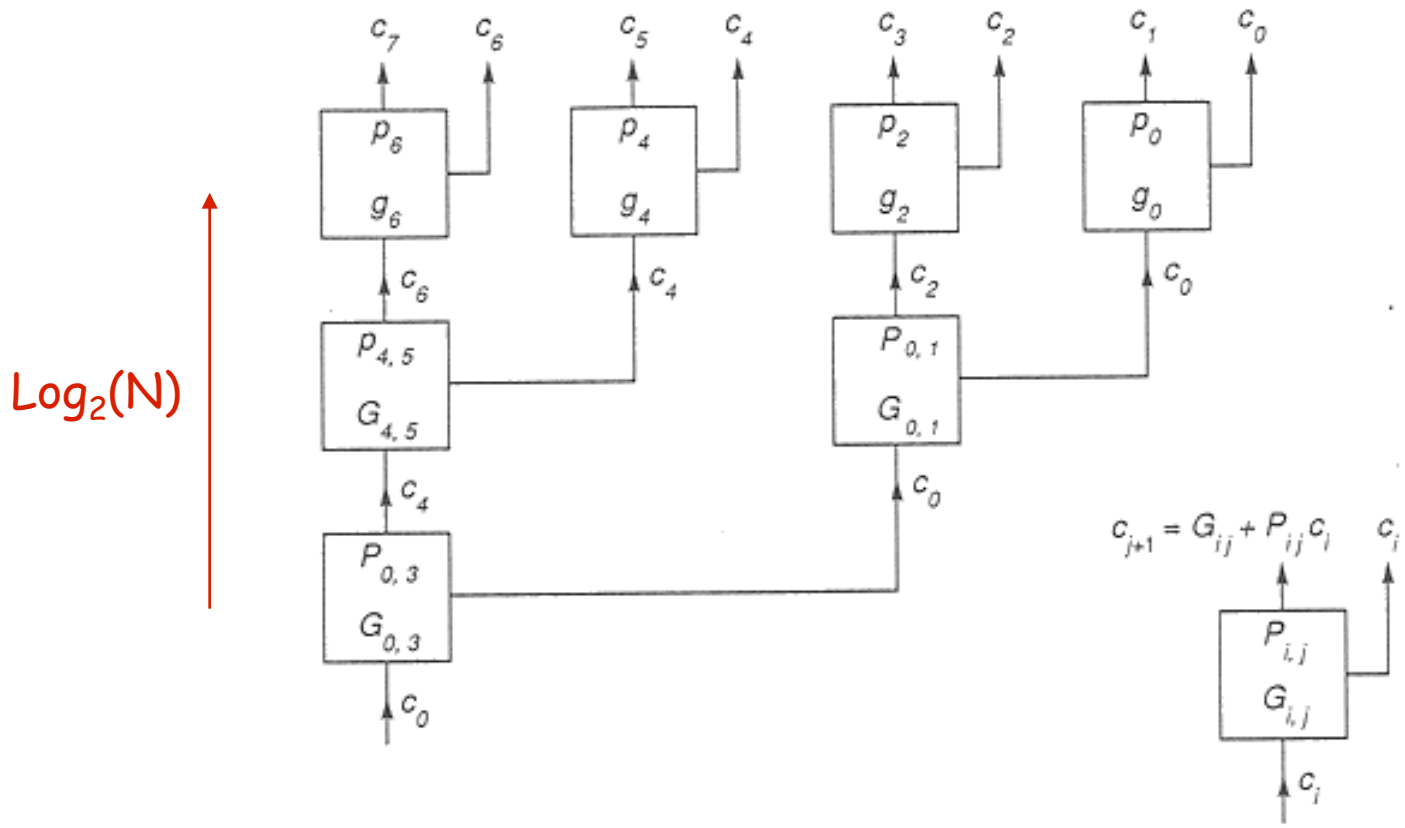


8-bit CLA (P/G generation)

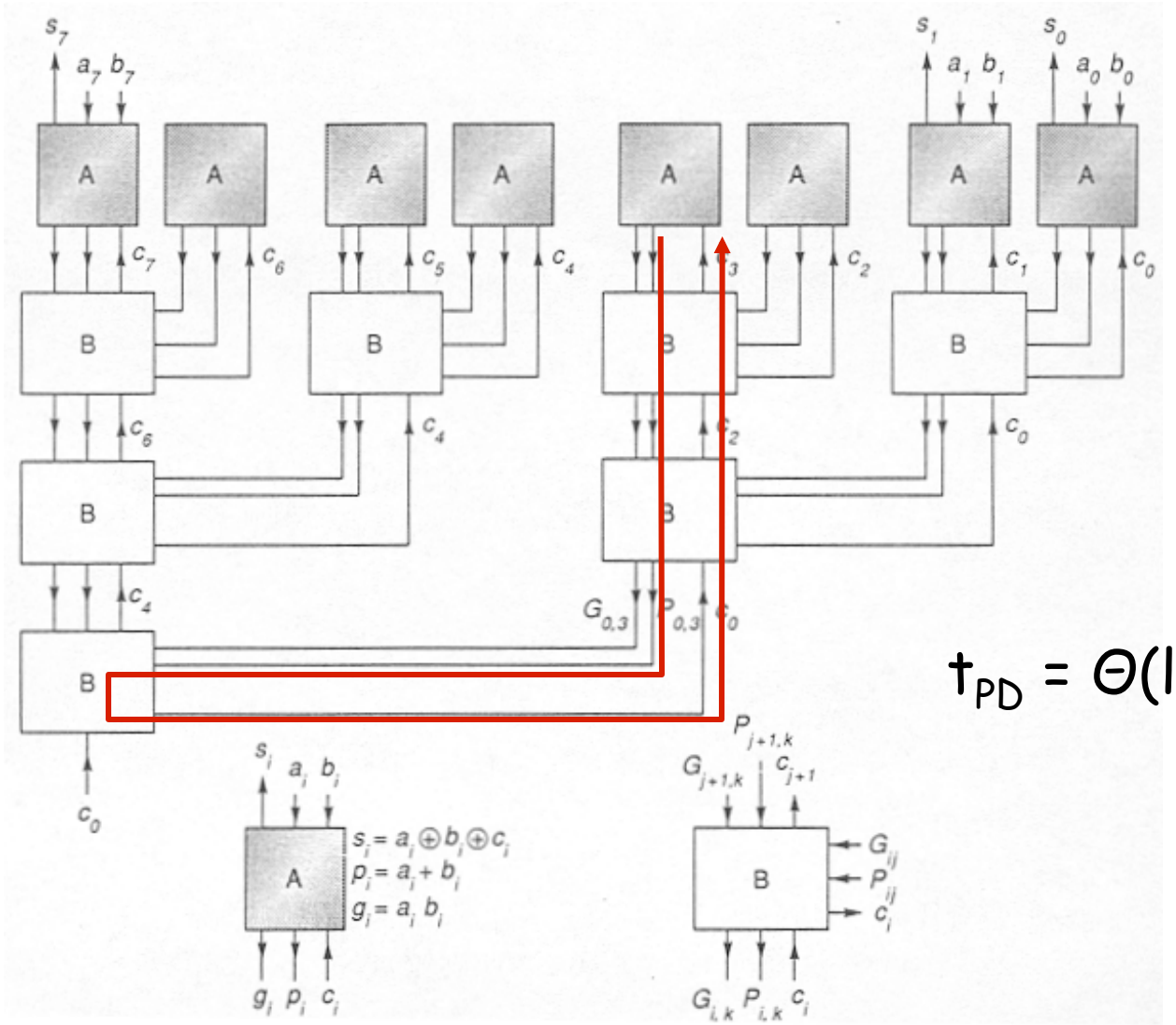


From Hennessy & Patterson, Appendix A

8-bit CLA (carry generation)



8-bit CLA (complete)



$t_{PD} = \Theta(\log(N))$