

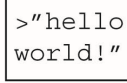


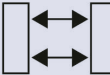
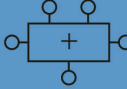
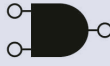
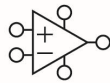


Chapter 5

Digital Design and Computer Architecture, 2nd Edition

David Money Harris and Sarah L. Harris

Chapter 5 :: Topics

- Introduction
- Arithmetic Circuits
- Number Systems
- Sequential Building Blocks
- Memory Arrays
- Logic Arrays

Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	

Introduction

- **Digital building blocks:**
 - Gates, multiplexers, decoders, registers, arithmetic circuits, counters, memory arrays, logic arrays
- **Building blocks demonstrate hierarchy, modularity, and regularity:**
 - Hierarchy of simpler components
 - Well-defined interfaces and functions
 - Regular structure easily extends to different sizes
- **Will use these building blocks in Chapter 7 to build microprocessor**

Design Tradeoffs

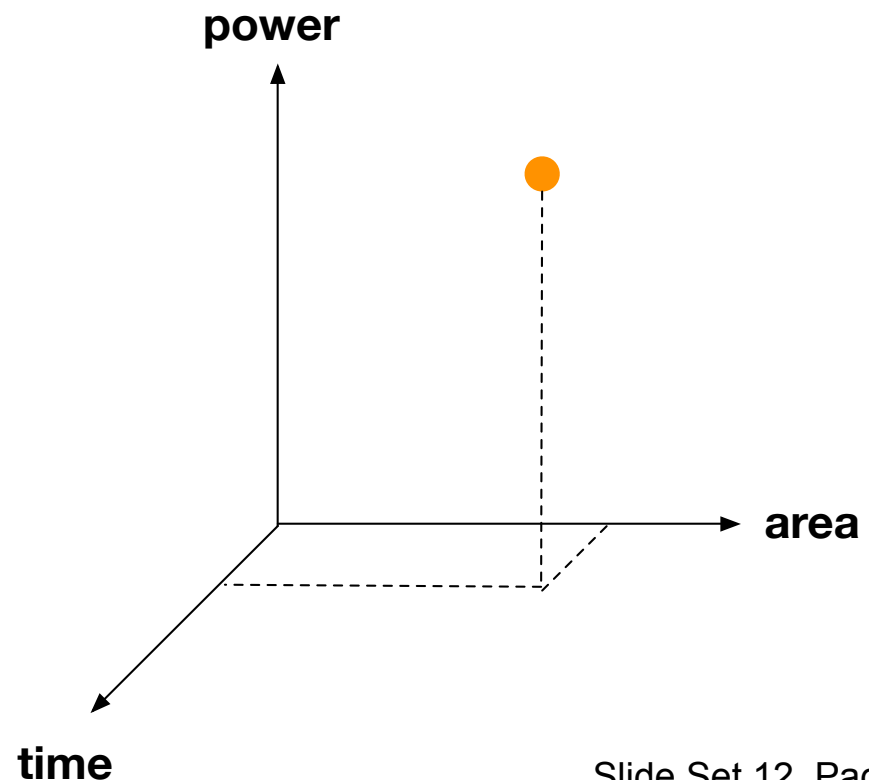
There are many ways to build adders (or any function).

Which is the right implementation?

Depends on your system's requirements

Optimization Metrics:

- Speed
- Power
- Area



Single Cycle vs. Multi Cycle Arithmetic

Arithmetic units can be written as:

1. Combinational Blocks

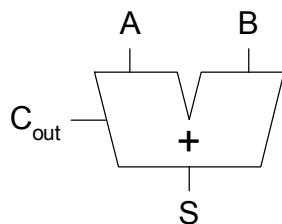
- Outputs depend only on inputs
- Results available in one clock-cycle

2. Multi-Cycle

- Result is computed over multiple cycles
- Can be much smaller (require fewer logic resources)
- Can be pipelined for higher clock frequency

1-Bit Adders

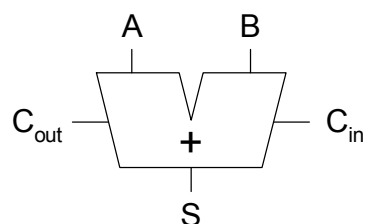
Half Adder



A	B	C _{out}	S
0	0		
0	1		
1	0		
1	1		

$$\begin{array}{l} S = \\ C_{out} = \end{array}$$

Full Adder

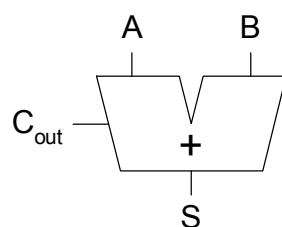


C _{in}	A	B	C _{out}	S
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

$$\begin{array}{l} S = \\ C_{out} = \end{array}$$

1-Bit Adders

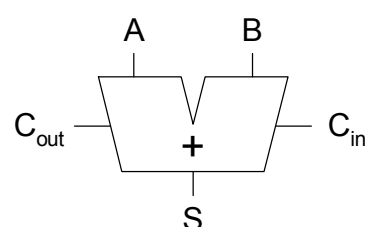
Half Adder



A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$\begin{matrix} S \\ C_{out} \end{matrix} =$$

Full Adder

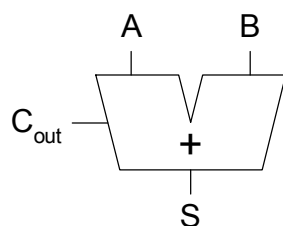


C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$\begin{matrix} S \\ C_{out} \end{matrix} =$$

1-Bit Adders

Half Adder

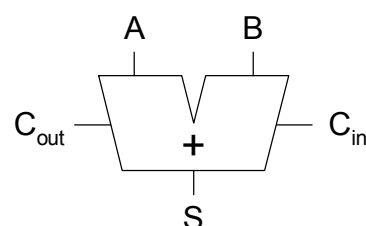


A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = A \oplus B$$

$$C_{out} = AB$$

Full Adder



C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

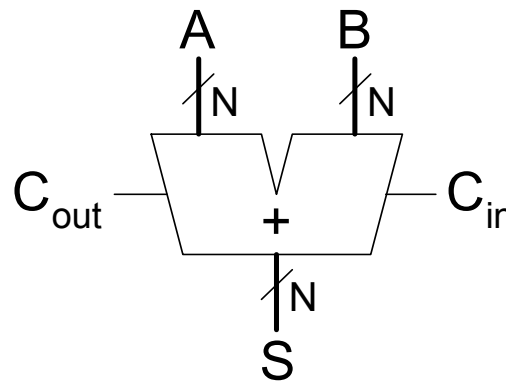
$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + AC_{in} + BC_{in}$$

Multibit Adders (CPAs)

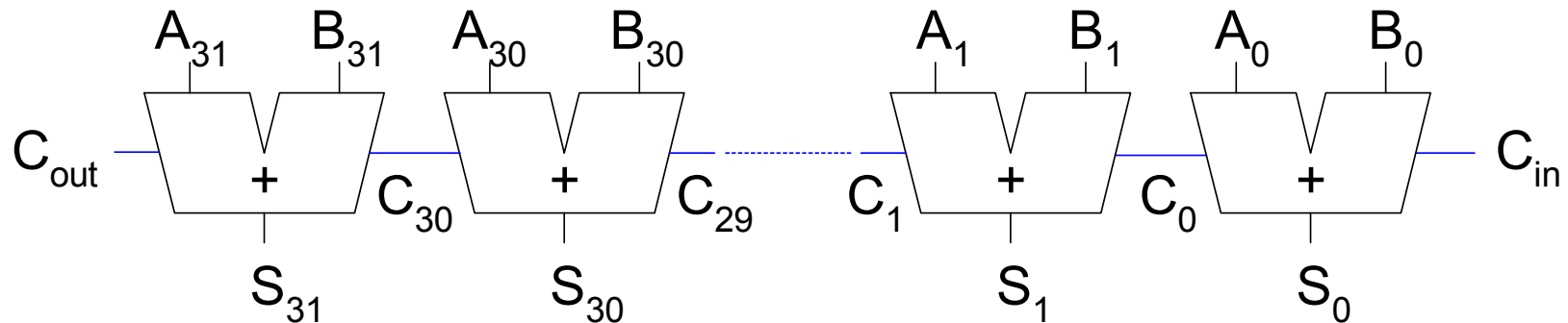
- Types of carry propagate adders (CPAs):
 - Ripple-carry (slow)
 - Carry-lookahead (fast)
 - Prefix (faster)
- Carry-lookahead and prefix adders faster for large adders but require more hardware

Symbol



Ripple-Carry Adder

- Chain 1-bit adders together
- Carry ripples through entire chain
- Disadvantage: **slow**



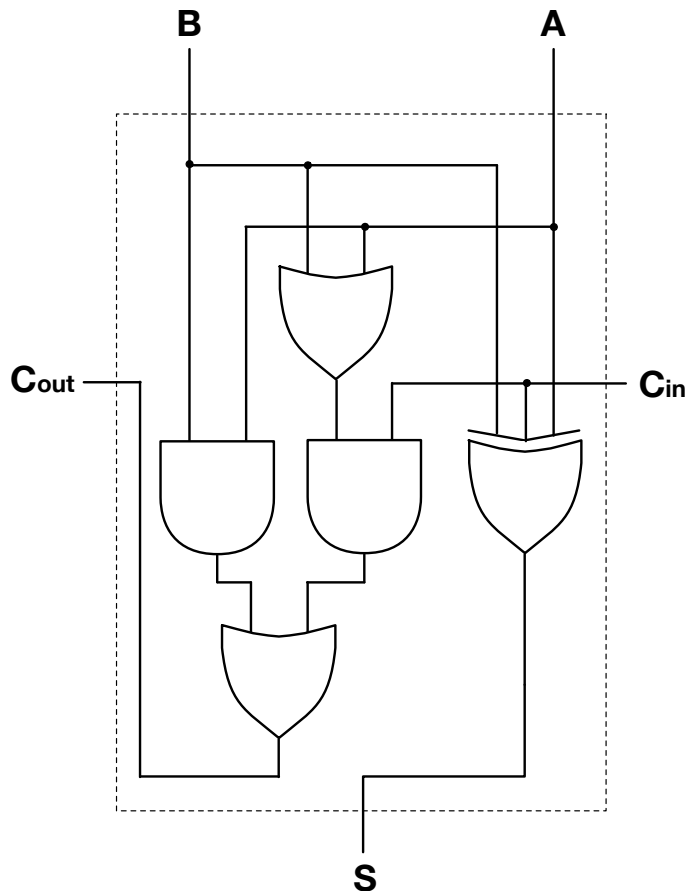
Ripple-Carry Adder Delay

$$t_{\text{ripple}} = Nt_{FA}$$

where t_{FA} is the delay of a full adder

Delay of a Full Adder

What is the critical path delay of a Full Adder?



Assume all gates have the same gate propagation delay t_{PD}

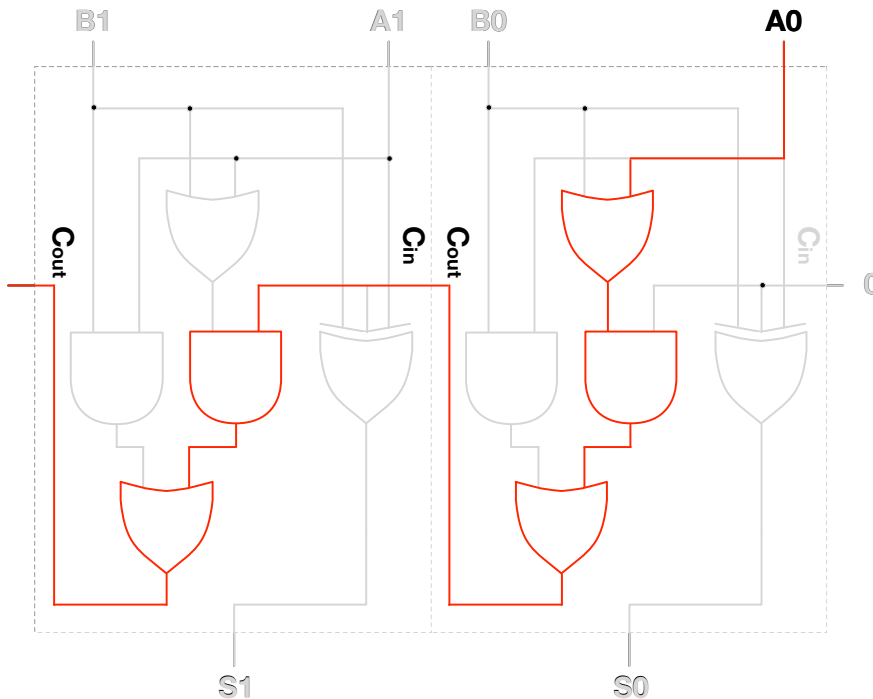
If inputs **A**, **B**, **C_{in}** arrive at time 0,

- **S** is ready after t_{PD}
A, B, C_{in} → XOR Gate → S
- **C_{out}** is ready after $3 t_{PD}$
A, B → OR → AND → OR → C_{out}

Critical Path Delay is $3 t_{PD}$

Delay of Carry Propagate Adder

Consider 2-bit Carry Propagate Adder



Inputs **A0, A1, B0, B1** arrive at time 0,

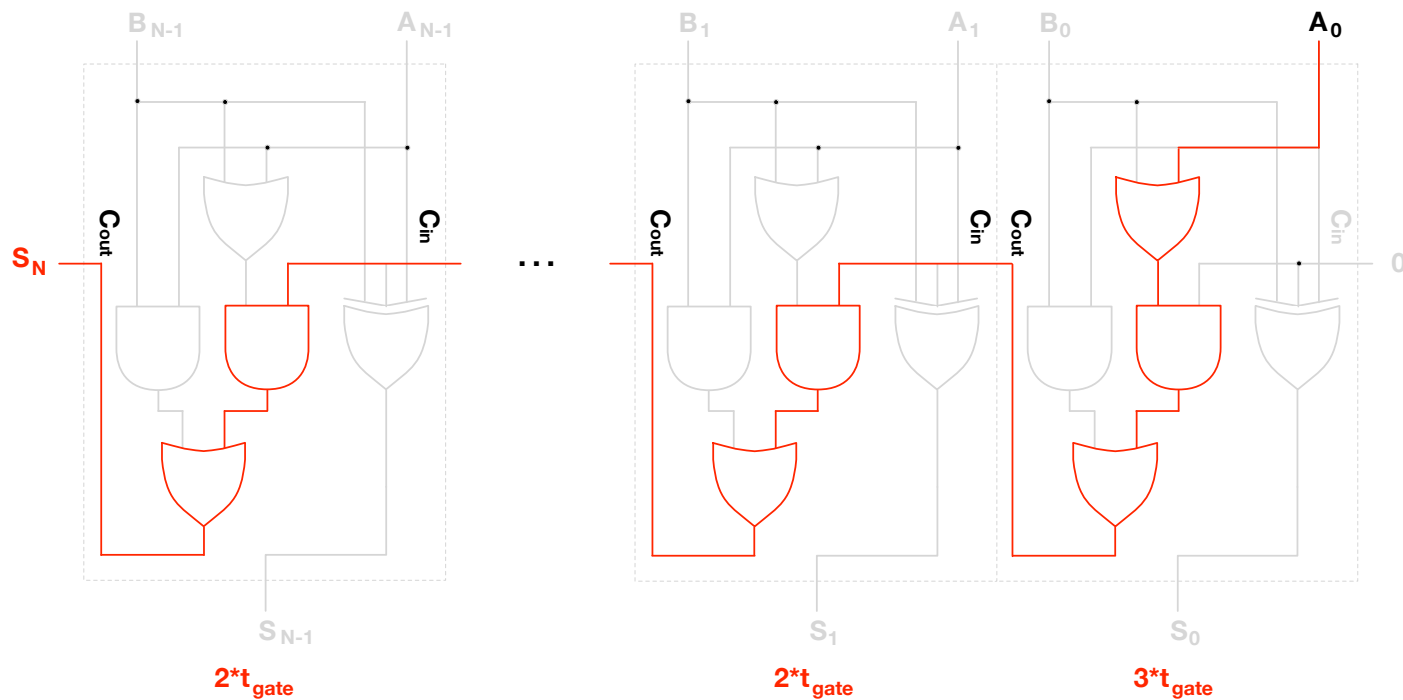
- **C_{out}** of first bit is ready after **3 t_{PD}**
- Delay of next **C_{out}** is ready after another **2 t_{PD}**

Critical Path Delay is 5 t_{PD}

Delay of Carry Propagate Adder

Delay for an N-bit Carry Propagate Adder is

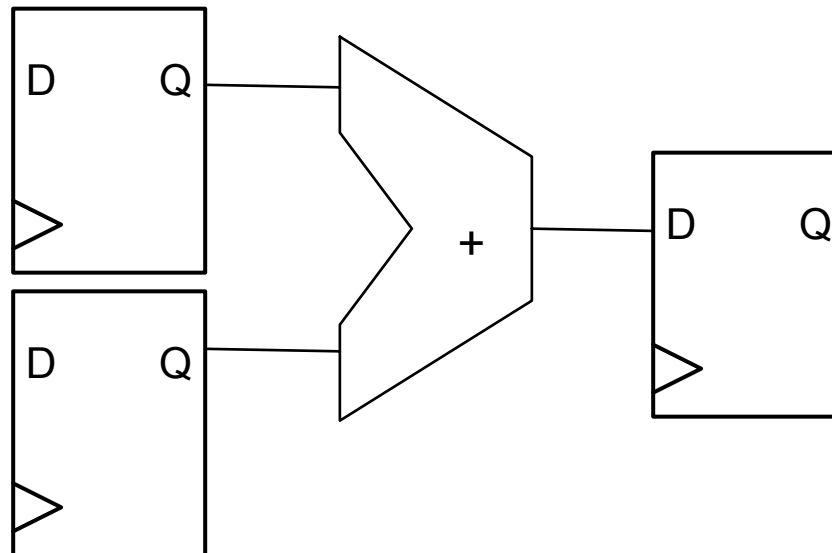
$$2(N-1)t_{PD} + 3t_{PD}$$



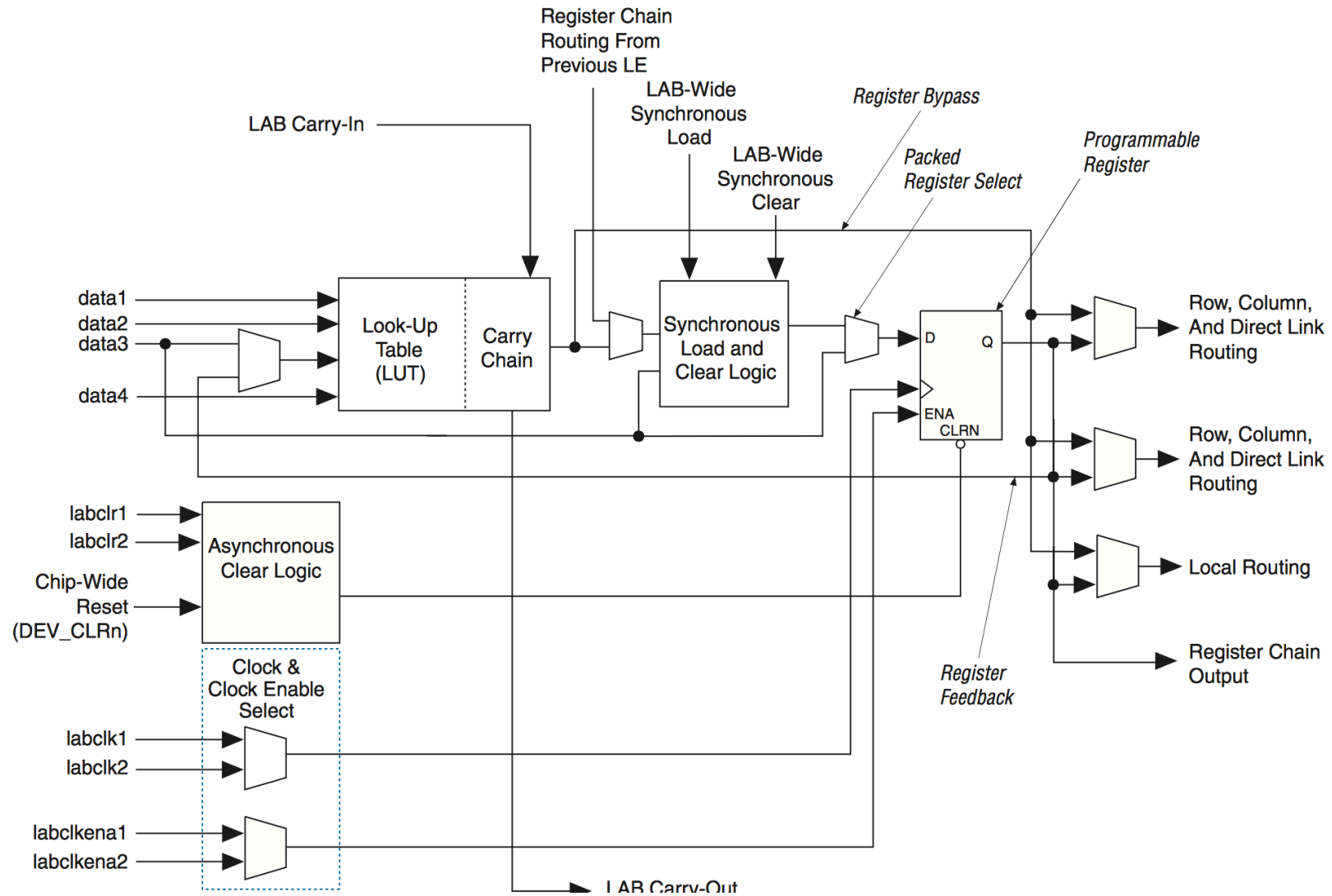
Delay is proportional to $N \rightarrow$ **SLOW** for large N

Ripple Adders in your Cyclone II device

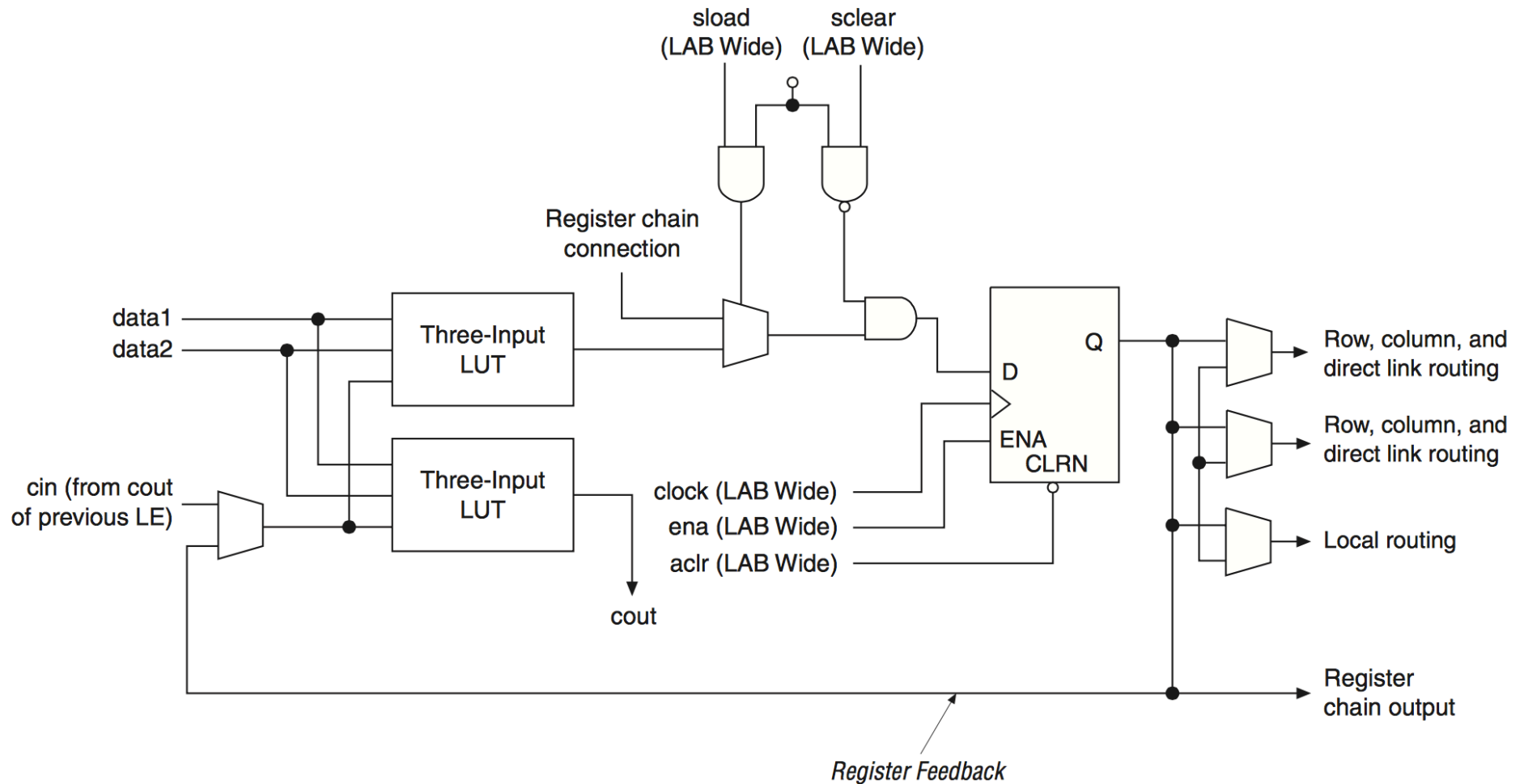
Size	# Logic Elements	Speed
16 bits	16	368 MHz
32 bits	32	249 MHz
64 bits	64	151 MHz
128 bits	128	85.5 MHz
256 bits	256	45.1 MHz
512 bits	512	23.3 MHz



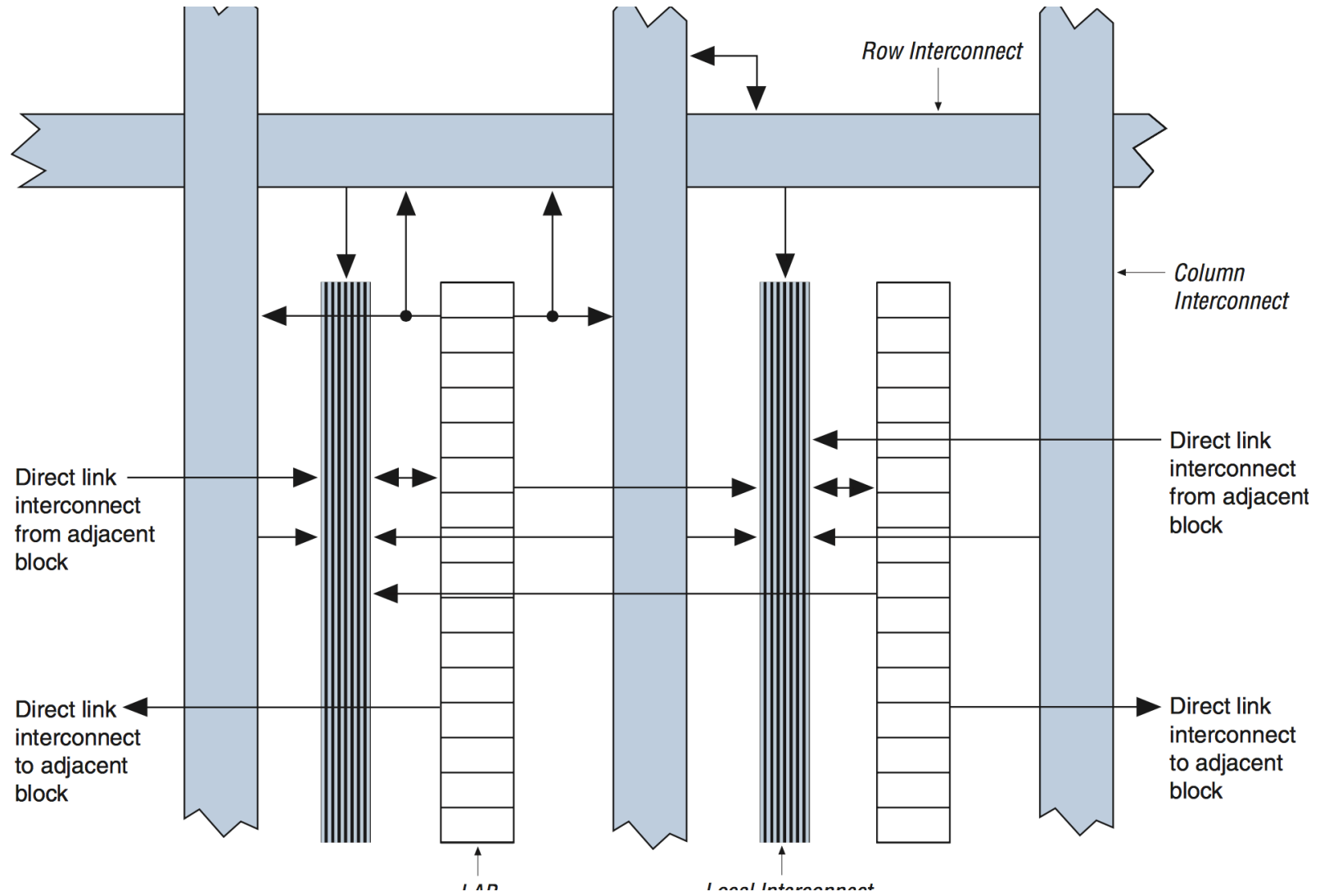
Cyclone II Logic Block



Cyclone II Logic block in Arithmetic Mode



Cyclone II LABs



Important point from these past few slides: FPGA Architectures have been optimized to implement ripple carry adders, so they are pretty good at it. But, there are ways to make even faster adders.

Can we make faster adders?

Carry Select Adder

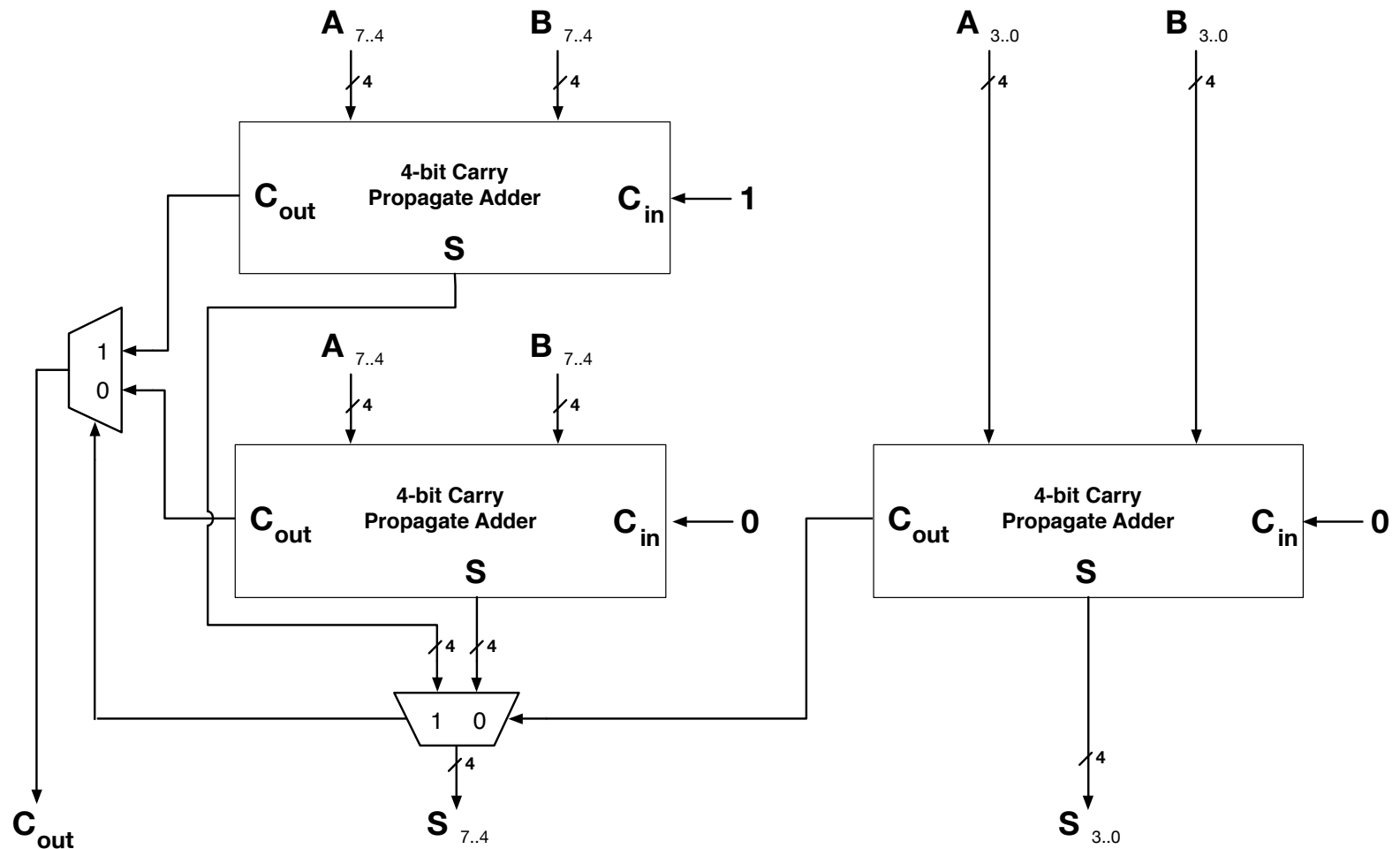
Approach

Carry Propagate Adders are slow because the high-order bits need to wait for the carry-in from lower-order bits.

**Calculate high-order bits for BOTH cases of carry-in.
Then select the correct case when carry-in is ready**

Trade-off area (use more gates) for faster performance

8-Bit Example



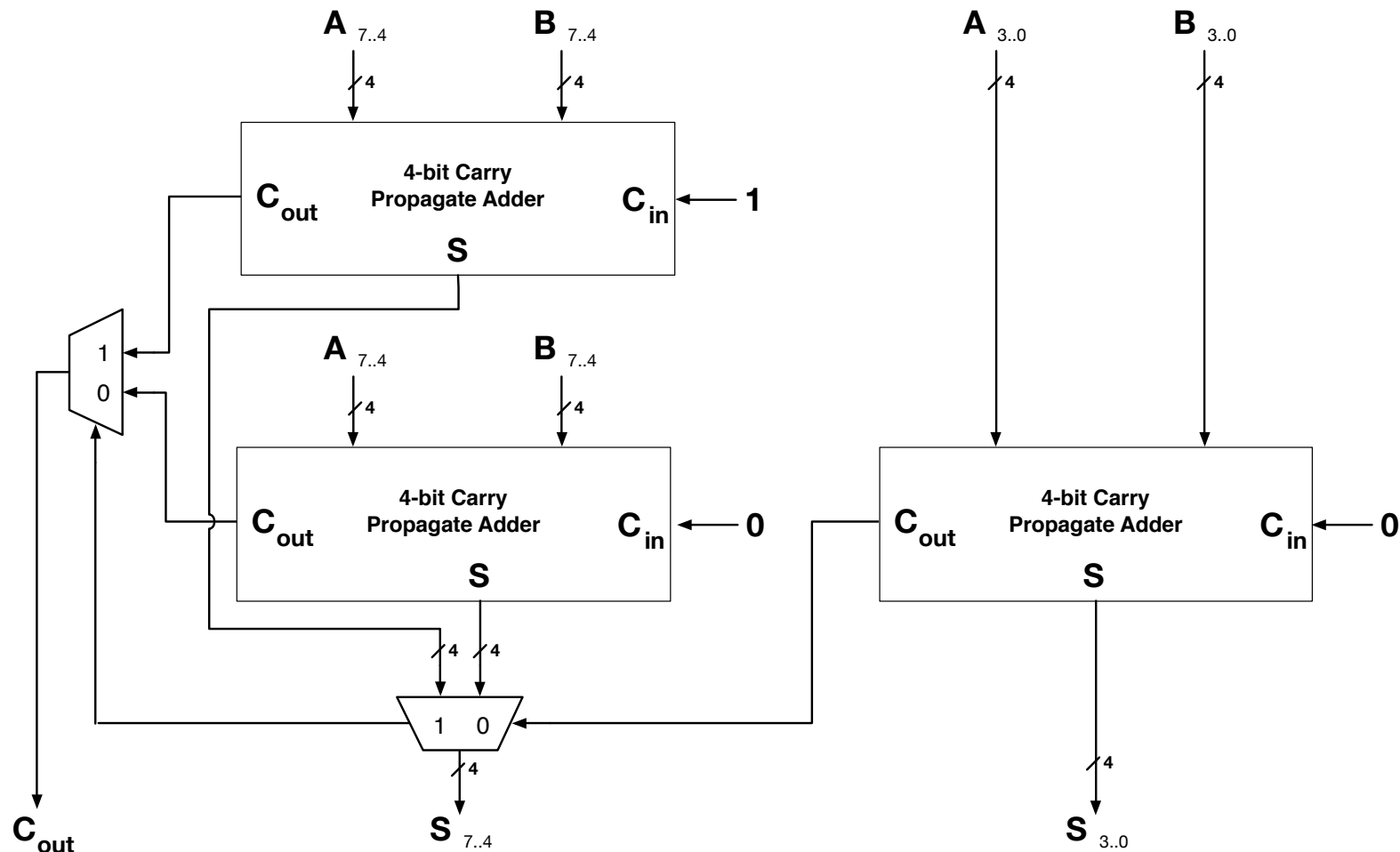
Bits 7..4 computed in parallel with bits 3..0
Exercise: What is the critical path delay?

In your Cyclone II device...

Size	Carry Prop. Adder		Carry Select Adder	
	# LEs	Freq.	# LEs	Freq.
32 bits	32	249 MHz	49	268 MHz
64 bits	64	151 MHz	97	192 MHz
128 bits	128	85.5 MHz	193	123 MHz

To implement this in VHDL:

Quartus II will not automatically construct this sort of adder.
Need to explicitly define each block using a separate assignment or module (exercise: try it)



Carry Look-Ahead Adder (CLA)

Carry-Lookahead Adder

- Compute carry out (C_{out}) for k -bit blocks using *generate* and *propagate* signals
- **Some definitions:**
 - Column i produces a carry out by either *generating* a carry out or *propagating* a carry in to the carry out
 - Generate (G_i) and propagate (P_i) signals for each column:
 - Column i will generate a carry out if A_i AND B_i are both 1.

$$G_i = A_i B_i$$

- Column i will propagate a carry in to the carry out if A_i OR B_i is 1.

$$P_i = A_i + B_i$$

- The carry out of column i (C_i) is:

$$C_i = A_i B_i + (A_i + B_i) C_{i-1} = G_i + P_i C_{i-1}$$



Carry-Lookahead Addition

- **Step 1:** Compute G_i and P_i for all columns
- **Step 2:** Compute G and P for k -bit blocks
- **Step 3:** C_{in} propagates through each k -bit propagate/generate block

Carry-Lookahead Adder

- **Example:** 4-bit blocks ($G_{3:0}$ and $P_{3:0}$) :

$$G_{3:0} = G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0))$$

$$P_{3:0} = P_3 P_2 P_1 P_0$$

- **Generally,**

$$G_{i:j} = G_i + P_i (G_{i-1} + P_{i-1} (G_{i-2} + P_{i-2} G_j))$$

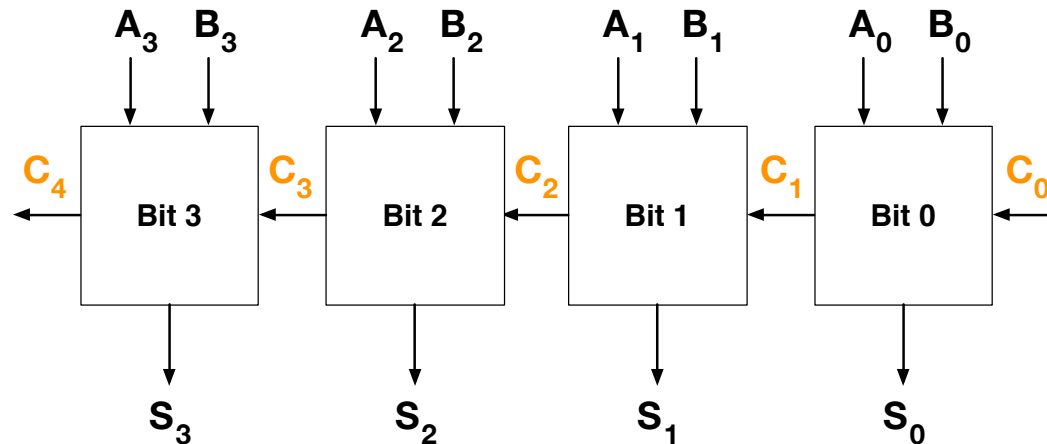
$$P_{i:j} = P_i P_{i-1} P_{i-2} P_j$$

$$C_i = G_{i:j} + P_{i:j} C_{i-1}$$

4-Bit Example

Let C_i denote the carry-in of stage i

this means that it is also the carry-out of the previous stage



$$C_1 = G_0 + P_0 \cdot C_0$$

$$C_2 = G_1 + P_1 \cdot C_1$$

$$C_3 = G_2 + P_2 \cdot C_2$$

$$C_4 = G_3 + P_3 \cdot C_3$$

Note: We are using
logical operators here:
+ means OR
. means AND

Carry Look-Ahead Logic

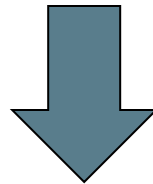
$$C_1 = G_0 + P_0 \cdot C_0$$

$$C_2 = G_1 + P_1 \cdot C_1$$

$$C_3 = G_2 + P_2 \cdot C_2$$

$$C_4 = G_3 + P_3 \cdot C_3$$

Perform Forward Substitution



$$C_1 = G_0 + P_0 \cdot C_0$$

$$C_2 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0$$

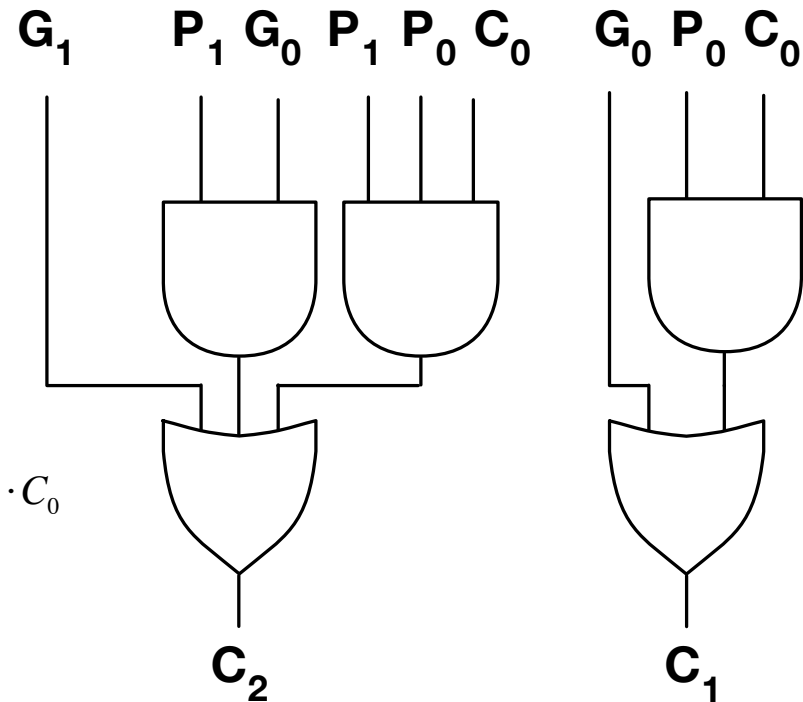
$$C_3 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0$$

$$C_4 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_0$$

Carry Look-Ahead Delay

$$\begin{aligned}C_1 &= G_0 + P_0 \cdot C_0 \\C_2 &= G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0 \\C_3 &= G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0 \\C_4 &= G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_0\end{aligned}$$

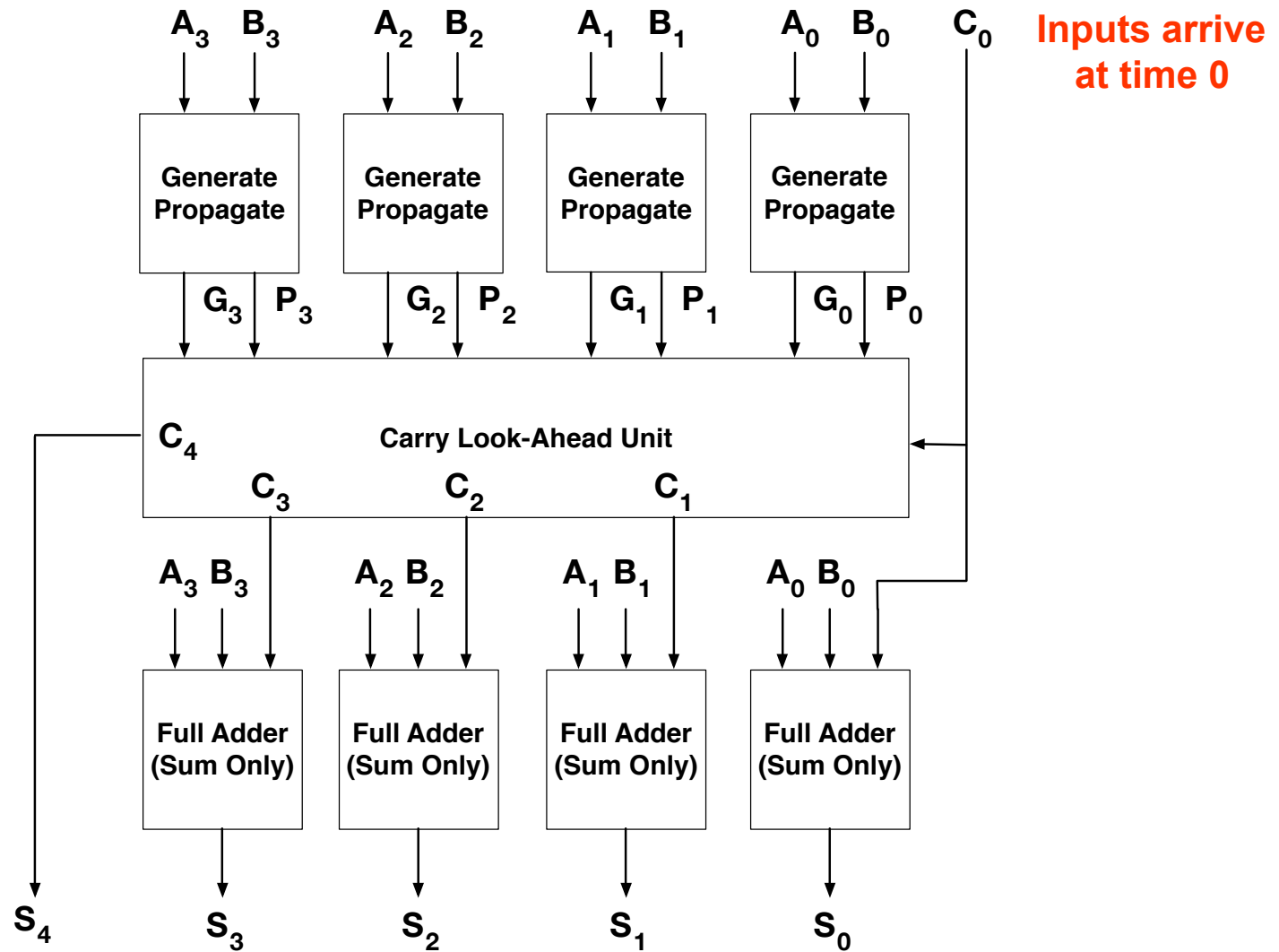
AND OR networks



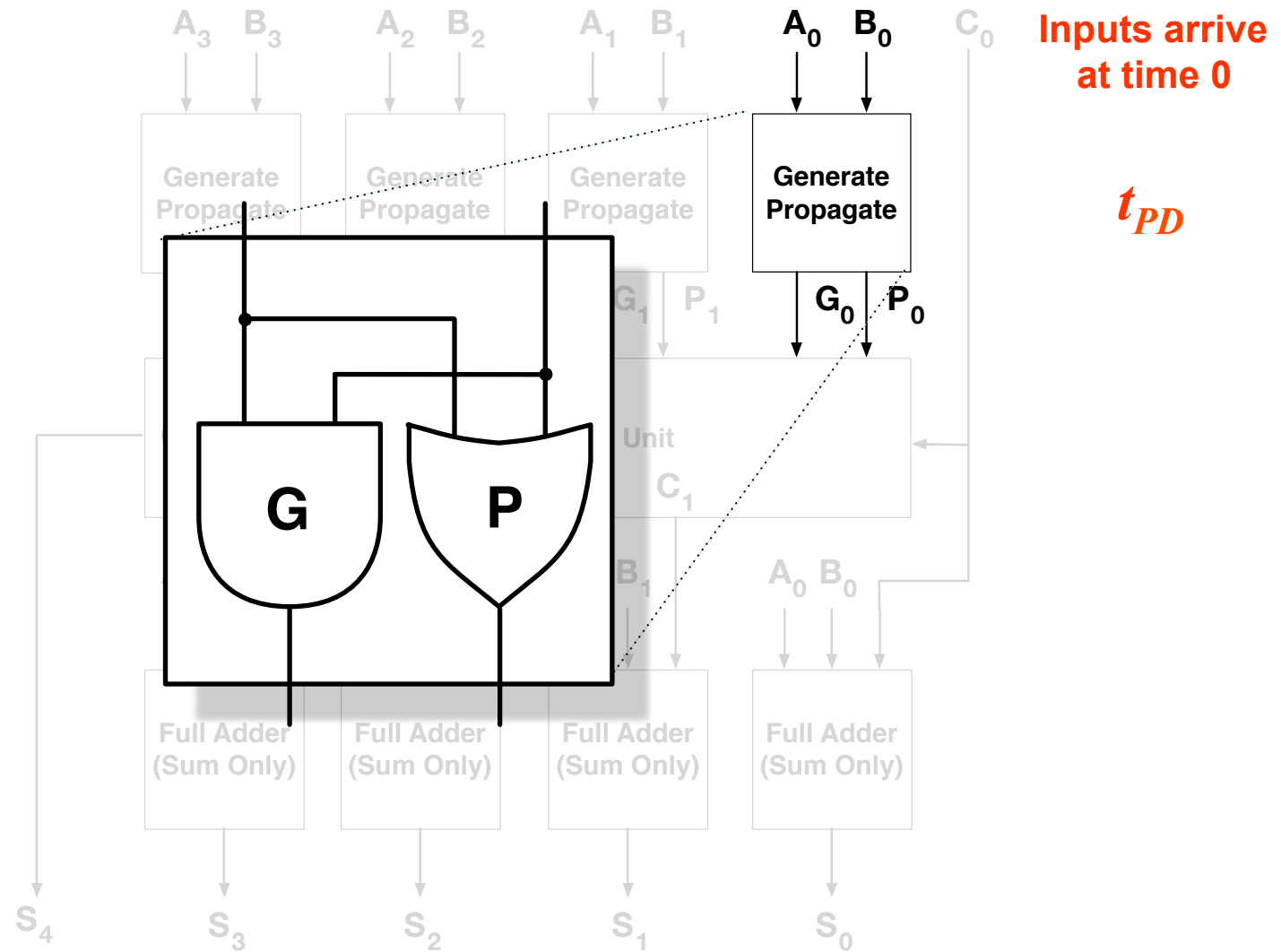
If C_0 and all G_i and P_i terms are available at the same time,
ALL C_i terms will be ready after **2 gate delays**

No Ripple Effect!

Overall Critical Path Delay

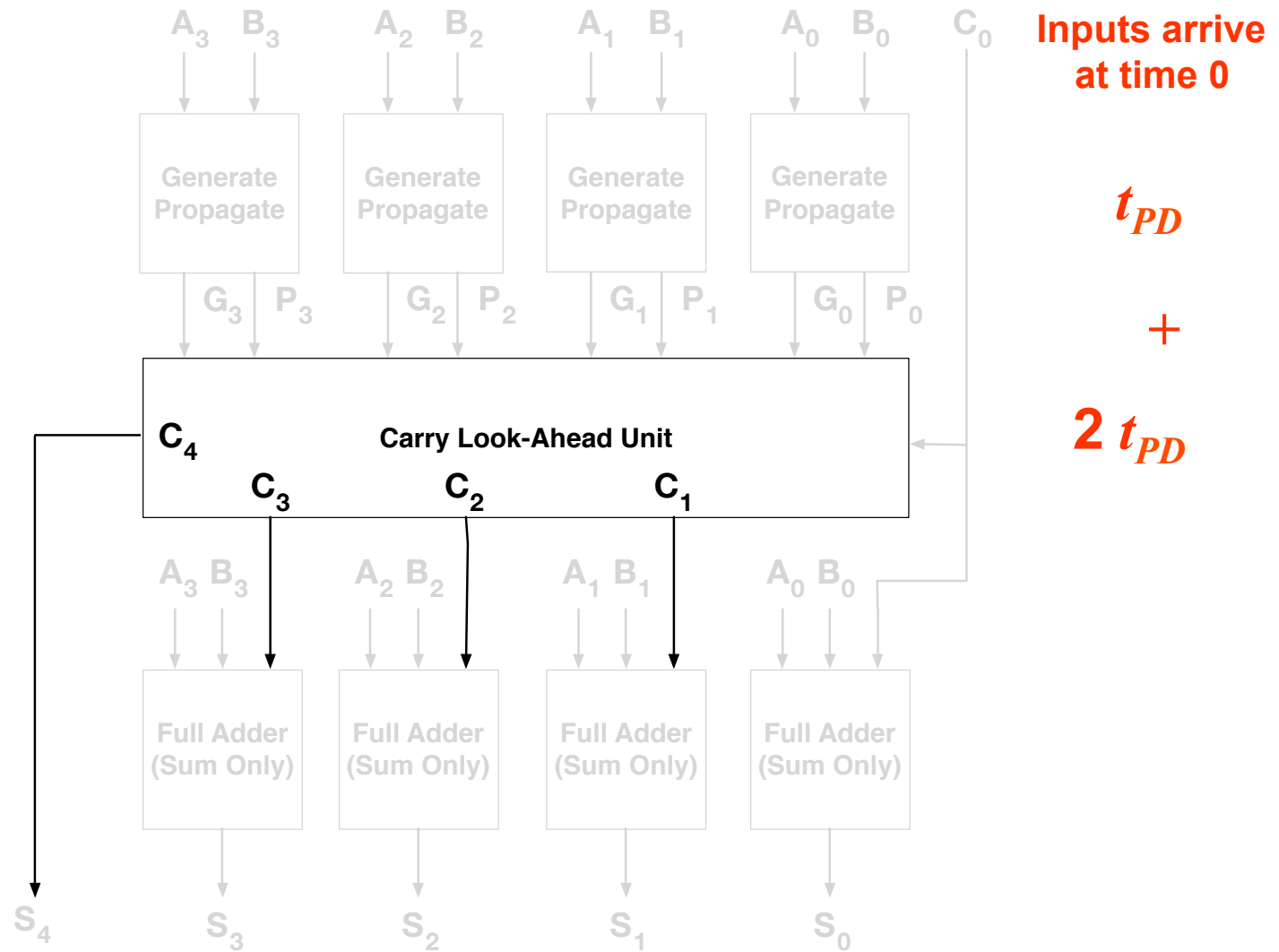


Overall Critical Path Delay



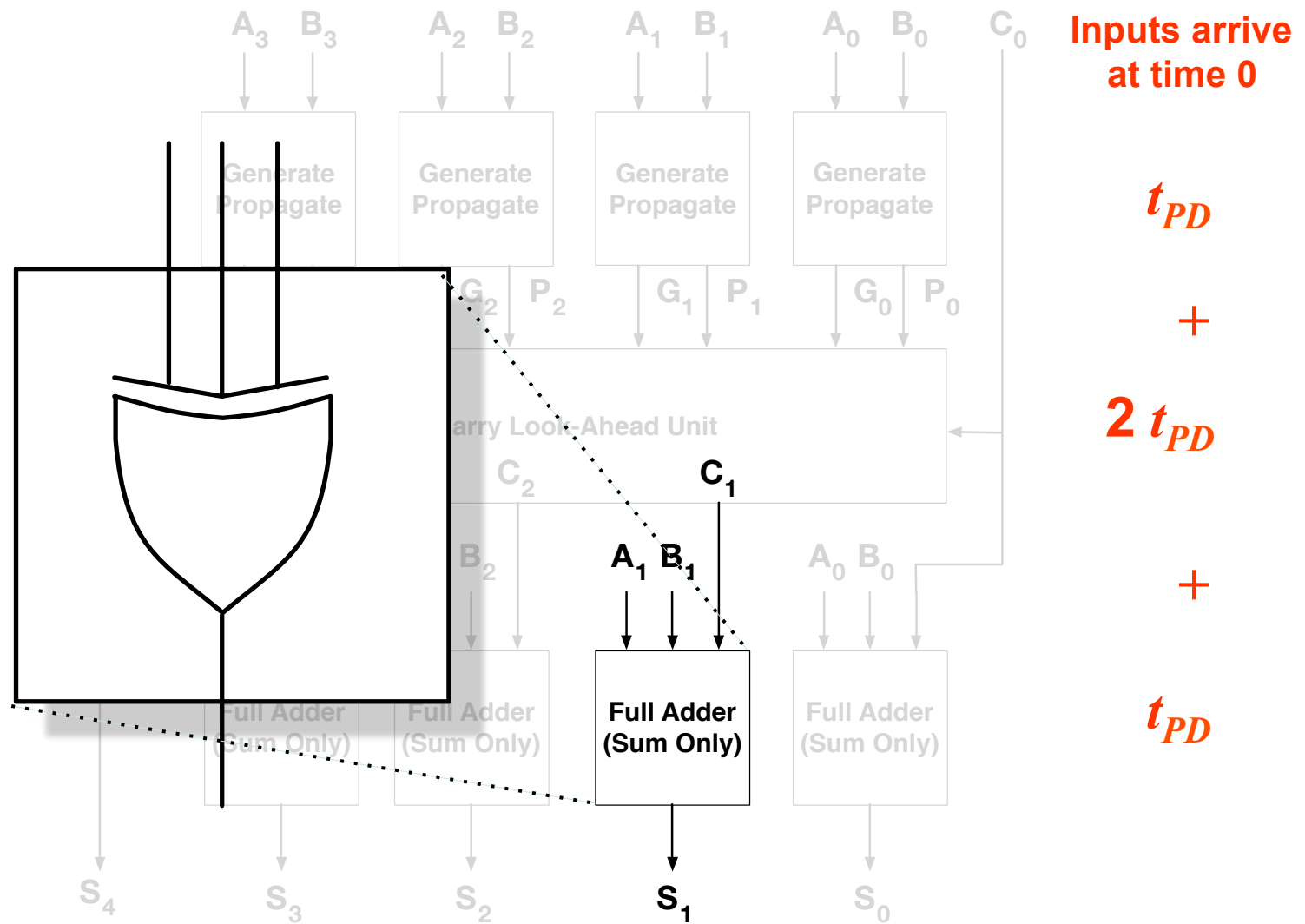
All G, P terms available after single gate delay

Overall Critical Path Delay

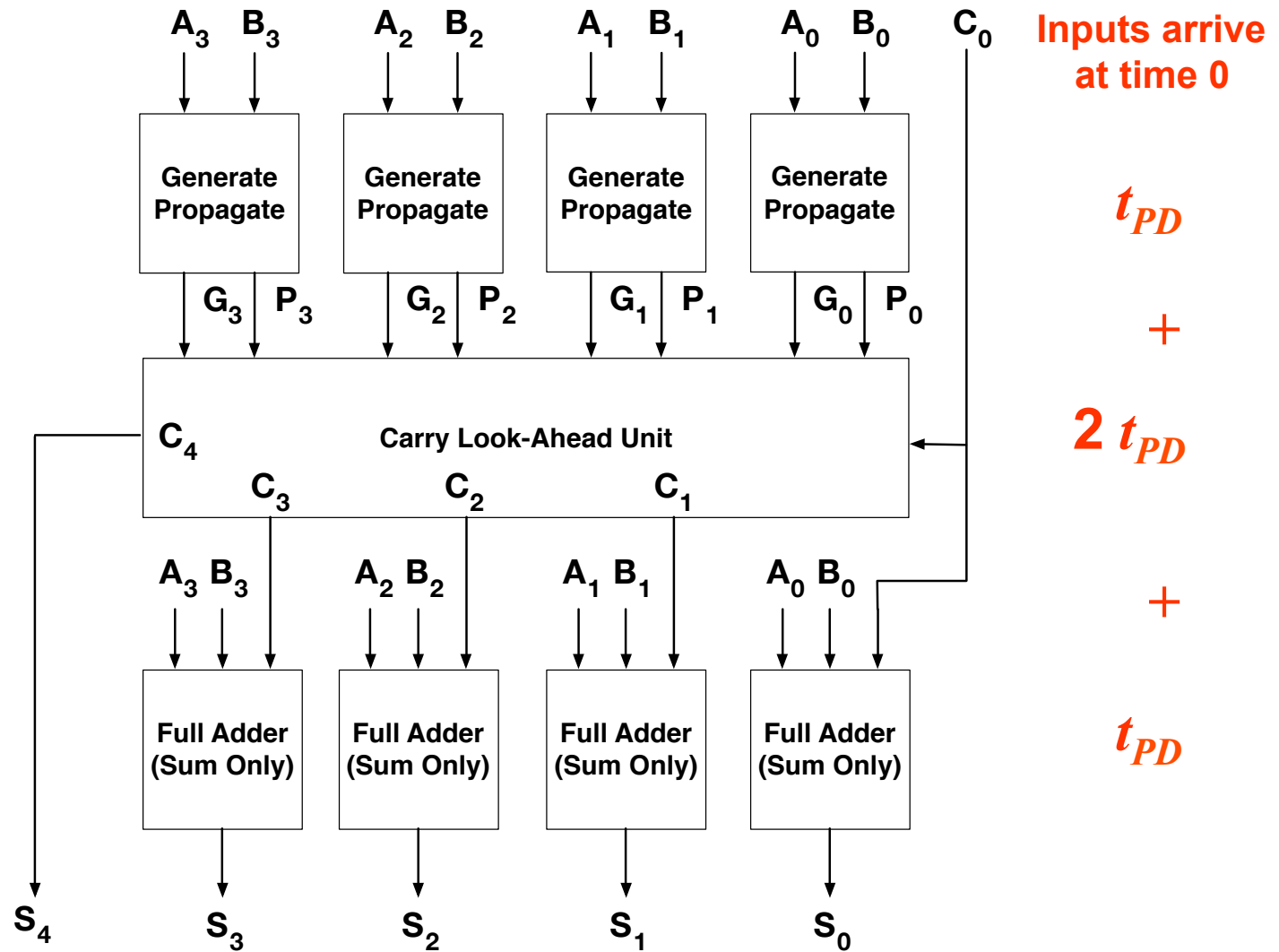


As discussed earlier

Overall Critical Path Delay



Overall Critical Path Delay



ALL outputs ready after $4 t_{PD}$

Scalability

In theory, we could build Carry Look-Ahead Adders of any size N

However, equations get more complex very quickly, and we need wider and wider gates (slow) in the carry logic.

$$C_1 = G_0 + P_0 \cdot C_0$$

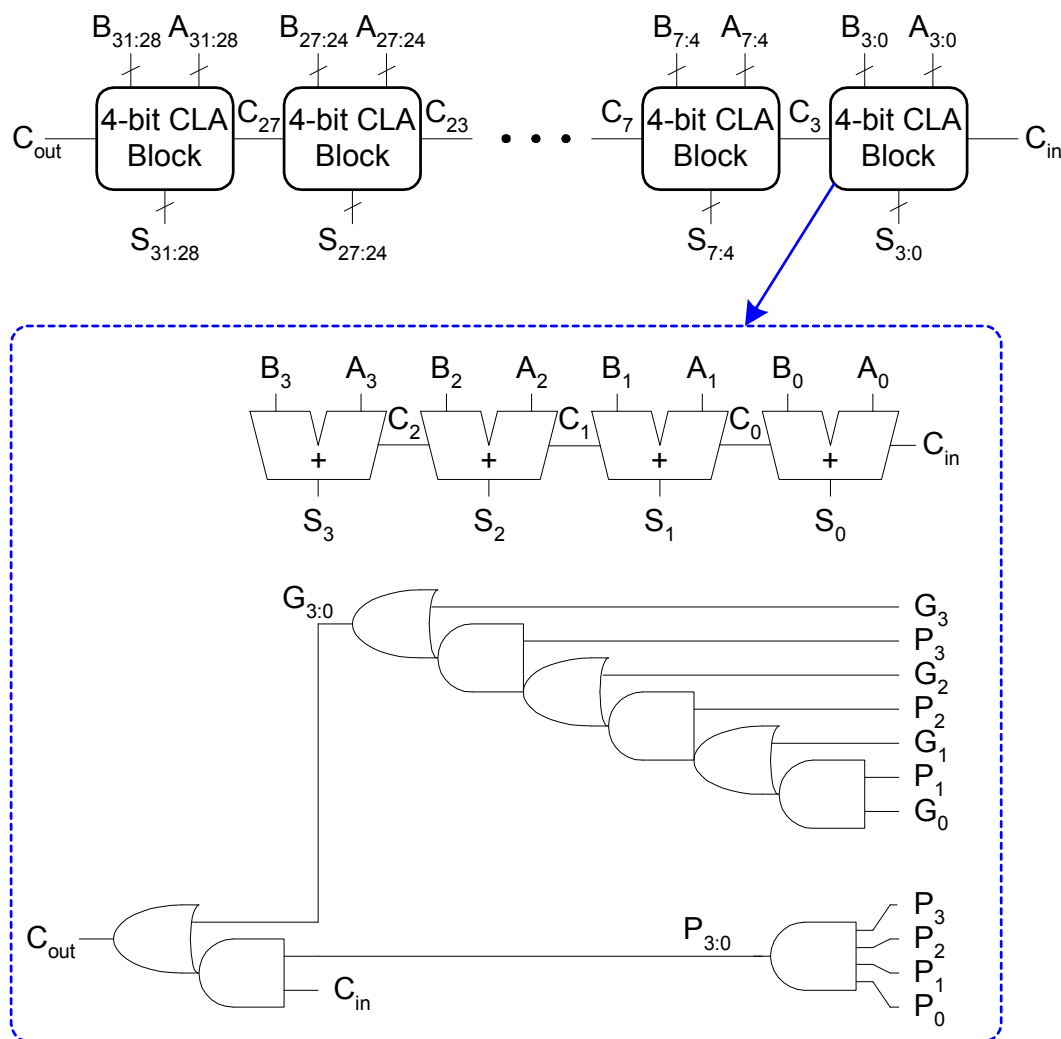
$$C_2 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0$$

$$C_3 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0$$

$$C_4 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_0$$

Typically do not extend beyond 4-bits

32-bit CLA with 4-bit Blocks



Carry-Lookahead Adder Delay

For N -bit CLA with k -bit blocks:

$$t_{CLA} = t_{pg} + t_{pg_block} + (N/k - 1)t_{AND_OR} + kt_{FA}$$

- t_{pg} : delay to generate all P_i, G_i
- t_{pg_block} : delay to generate all $P_{i:j}, G_{i:j}$
- t_{AND_OR} : delay from C_{in} to C_{out} of final AND/OR gate in k -bit CLA block

An N -bit carry-lookahead adder is generally much faster than a ripple-carry adder for $N > 16$

CLA Critical Path Delay

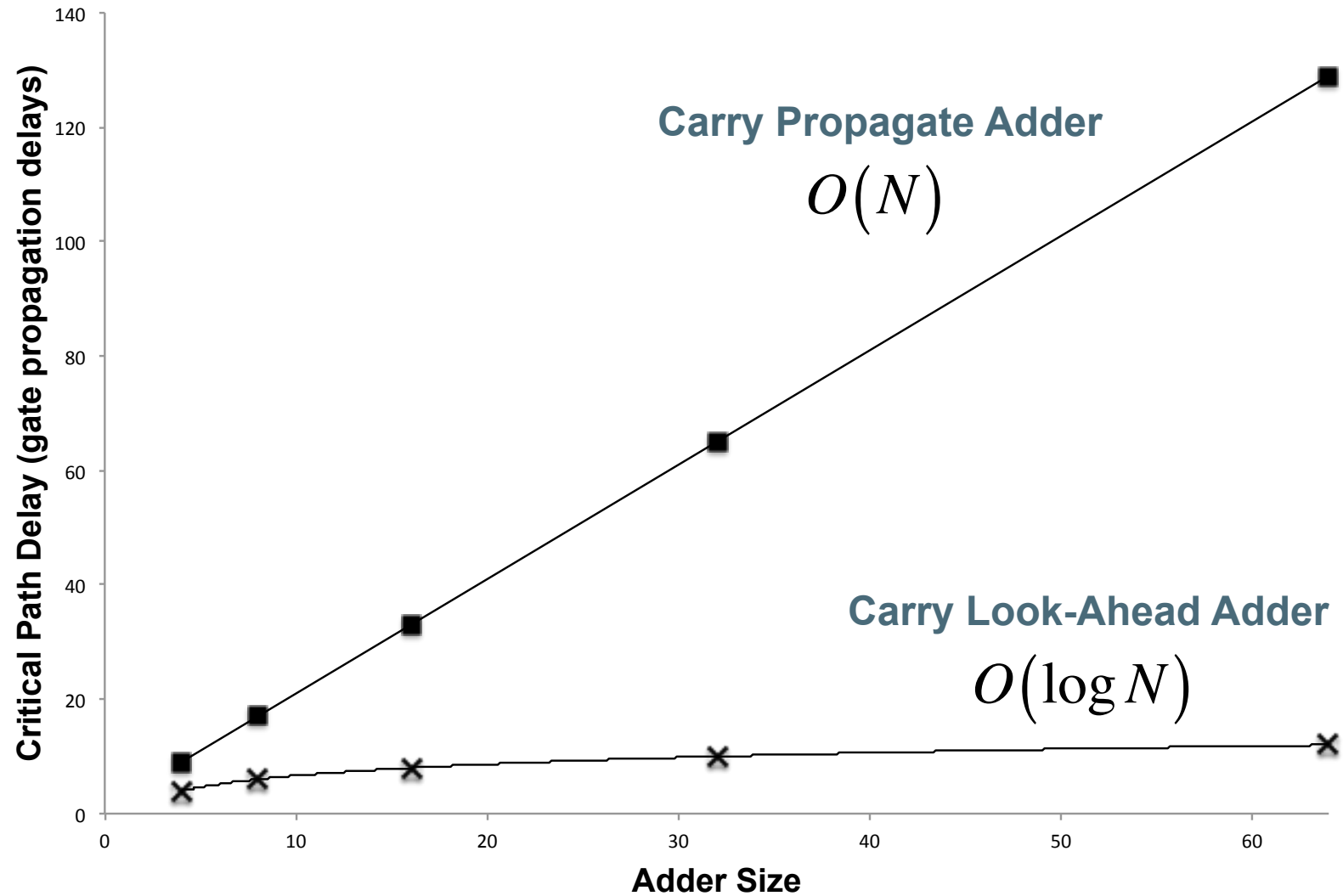
Delay for an N-bit Carry Look-Ahead Adder is

$$4 \log_4 (N) t_{PD}$$

(this is presented without proof, but you might be able to work it out given the logic)

Delay increases logarithmically w.r.t. N
Proportional to number of hierarchical “levels”

CPA vs CLA Critical Path Delay



On your Cyclone II:

For a 16 bit adder:

Ripple Carry-Propagate Adder: 368 MHz

Carry Look-ahead Adder (one level): 136 MHz

What happened? Look-ahead is supposed to be faster!?!?!?

On your Cyclone II..

Remember that the FPGA logic block has been optimized to implement carry-propagate adder.

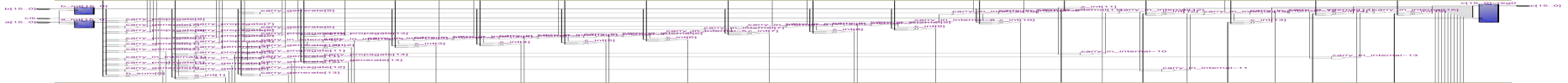
- Fast carry chains

In a carry-lookahead adder:

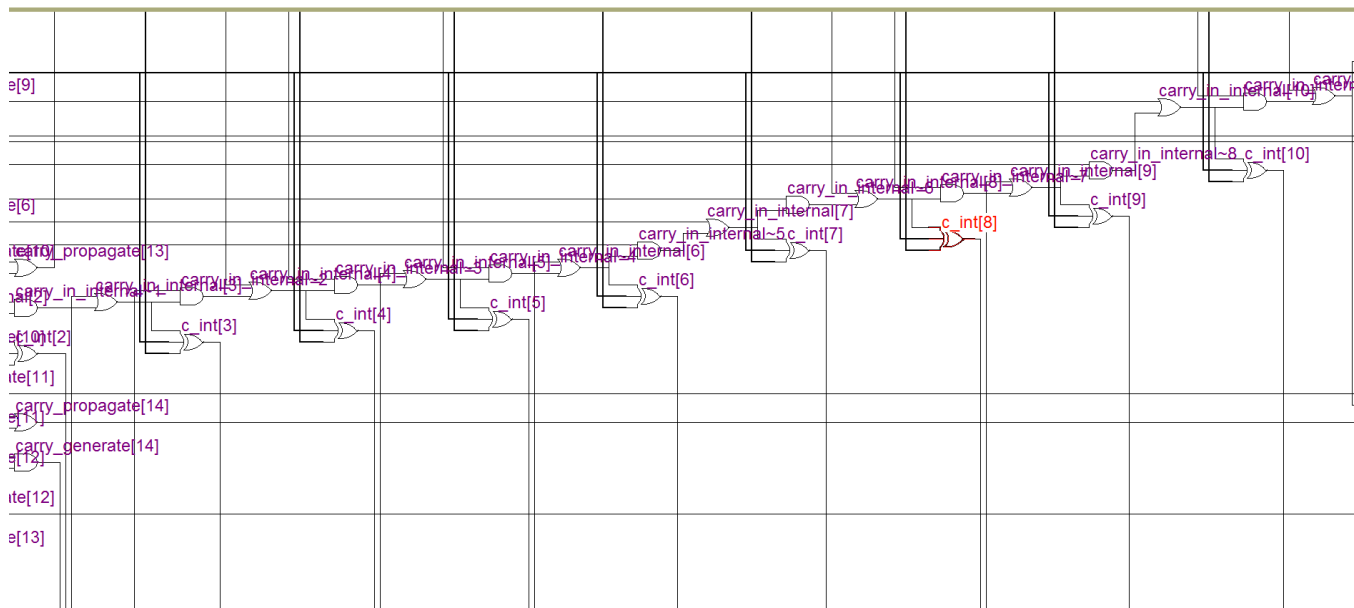
- Wide functions, need to break up into LUTs
- Each connection is made using general-purpose interconnect (as opposed to dedicated fast carry chains)

Even though there are fewer gate delays, the actual implementation on an FPGA is slower !

Quartus II synthesis:



↓
Zoom in...



Does that mean Carry-Lookahead is useless?

Does that mean Carry-Lookahead is useless?

No... much better if you make a custom chip (ASIC)

Moral: Always understand your underlying architecture (in our case, the FPGA).

Prefix Adder

- Computes carry in (C_{i-1}) for each column, then computes sum:

$$S_i = (A_i \oplus B_i) \oplus C_i$$

- Computes G and P for 1-, 2-, 4-, 8-bit blocks, etc. until all G_i (carry in) known
- $\log_2 N$ stages

Prefix Adder

- Carry in either *generated* in a column or *propagated* from a previous column.
- Column -1 holds C_{in} , so

$$G_{-1} = C_{in}, P_{-1} = 0$$

- Carry in to column i = carry out of column $i-1$:

$$C_{i-1} = G_{i-1:-1}$$

$G_{i-1:-1}$: generate signal spanning columns $i-1$ to -1

- Sum equation:

$$S_i = (A_i \oplus B_i) \oplus G_{i-1:-1}$$

- **Goal:** Quickly compute $G_{0:-1}$, $G_{1:-1}$, $G_{2:-1}$, $G_{3:-1}$, $G_{4:-1}$, $G_{5:-1}$, ... (called *prefixes*)

Prefix Adder

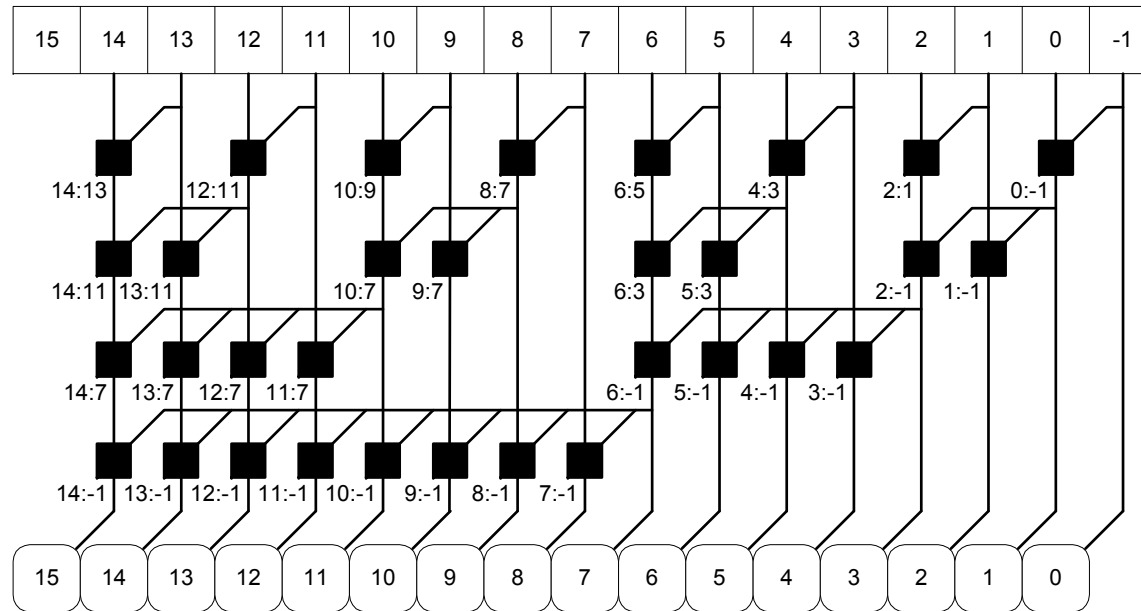
- Generate and propagate signals for a block spanning bits $i:j$:

$$G_{i:j} = G_{i:k} + P_{i:k} G_{k-1:j}$$

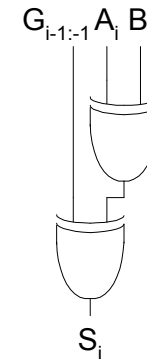
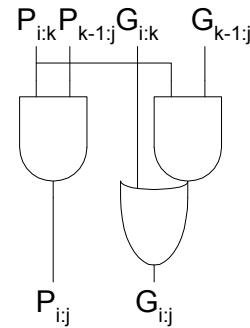
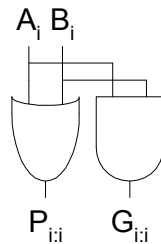
$$P_{i:j} = P_{i:k} P_{k-1:j}$$

- In words:
 - **Generate:** block $i:j$ will generate a carry if:
 - upper part ($i:k$) generates a carry or
 - upper part propagates a carry generated in lower part ($k-1:j$)
 - **Propagate:** block $i:j$ will propagate a carry if *both* the upper and lower parts propagate the carry

Prefix Adder Schematic



Legend



Prefix Adder Delay

$$t_{PA} = t_{pg} + \log_2 N(t_{pg_prefix}) + t_{XOR}$$

- t_{pg} : delay to produce $P_i G_i$ (AND or OR gate)
- t_{pg_prefix} : delay of black prefix cell (AND-OR gate)

Adder Delay Comparisons

Compare delay of: 32-bit ripple-carry, carry-lookahead, and prefix adders

- CLA has 4-bit blocks
- 2-input gate delay = 100 ps; full adder delay = 300 ps

Adder Delay Comparisons

Compare delay of: 32-bit ripple-carry, carry-lookahead, and prefix adders

- CLA has 4-bit blocks
- 2-input gate delay = 100 ps; full adder delay = 300 ps

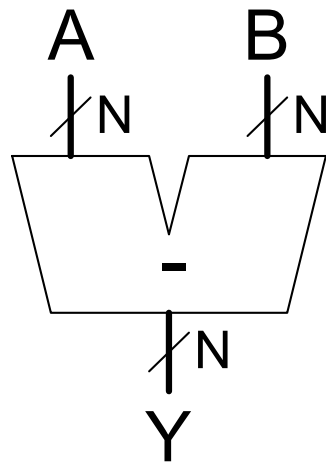
$$\begin{aligned} t_{\text{ripple}} &= Nt_{FA} = 32(300 \text{ ps}) \\ &= \mathbf{9.6 \text{ ns}} \end{aligned}$$

$$\begin{aligned} t_{CLA} &= t_{pg} + t_{pg_block} + (N/k - 1)t_{AND_OR} + kt_{FA} \\ &= [100 + 600 + (7)200 + 4(300)] \text{ ps} \\ &= \mathbf{3.3 \text{ ns}} \end{aligned}$$

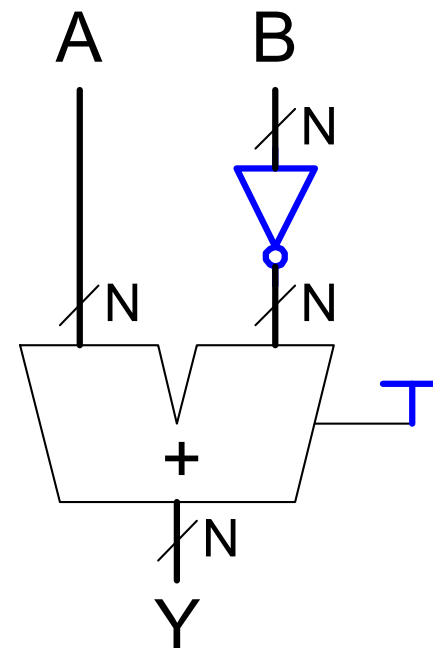
$$\begin{aligned} t_{PA} &= t_{pg} + \log_2 N(t_{pg_prefix}) + t_{XOR} \\ &= [100 + \log_2 32(200) + 100] \text{ ps} \\ &= \mathbf{1.2 \text{ ns}} \end{aligned}$$

Subtractor

Symbol

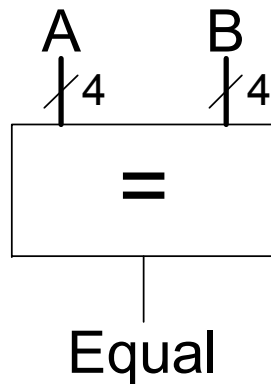


Implementation

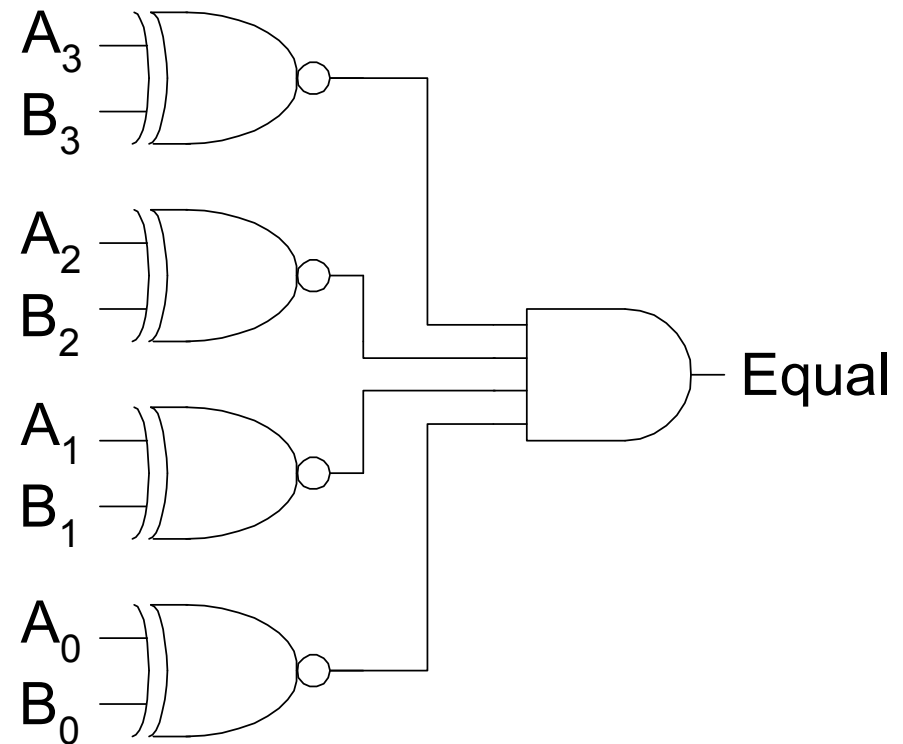


Comparator: Equality

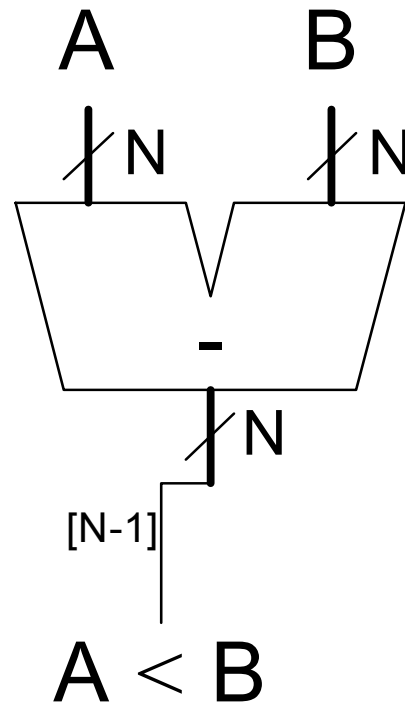
Symbol



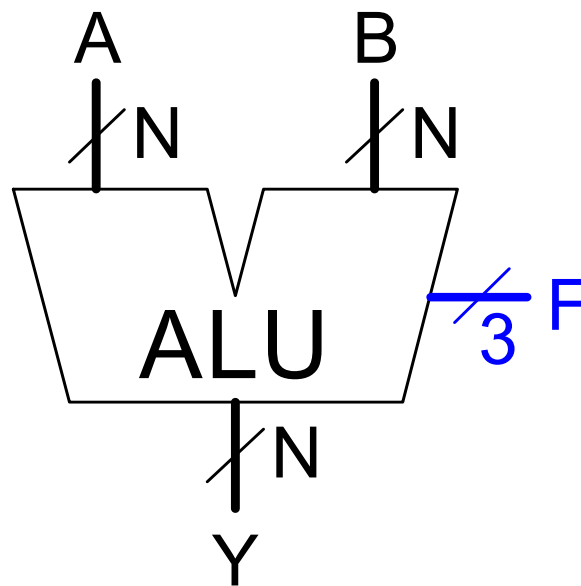
Implementation



Comparator: Less Than

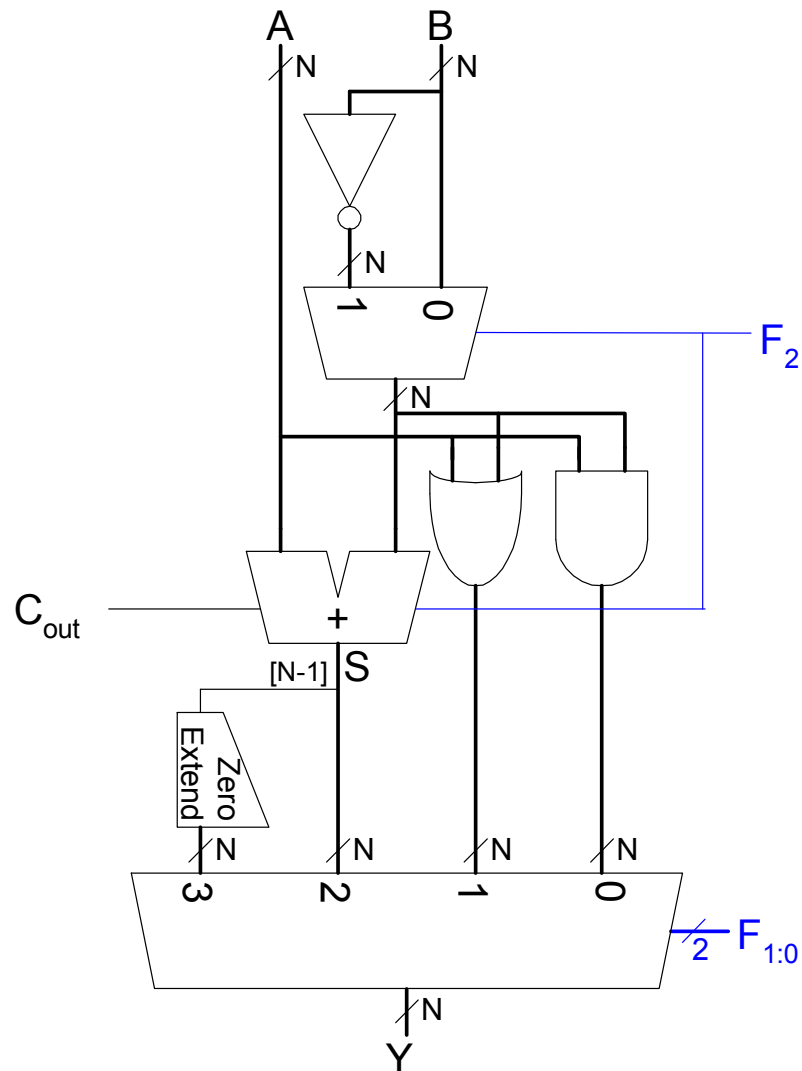


Arithmetic Logic Unit (ALU)



$F_{2:0}$	Function
000	$A \& B$
001	$A B$
010	$A + B$
011	not used
100	$A \& \sim B$
101	$A \sim B$
110	$A - B$
111	SLT

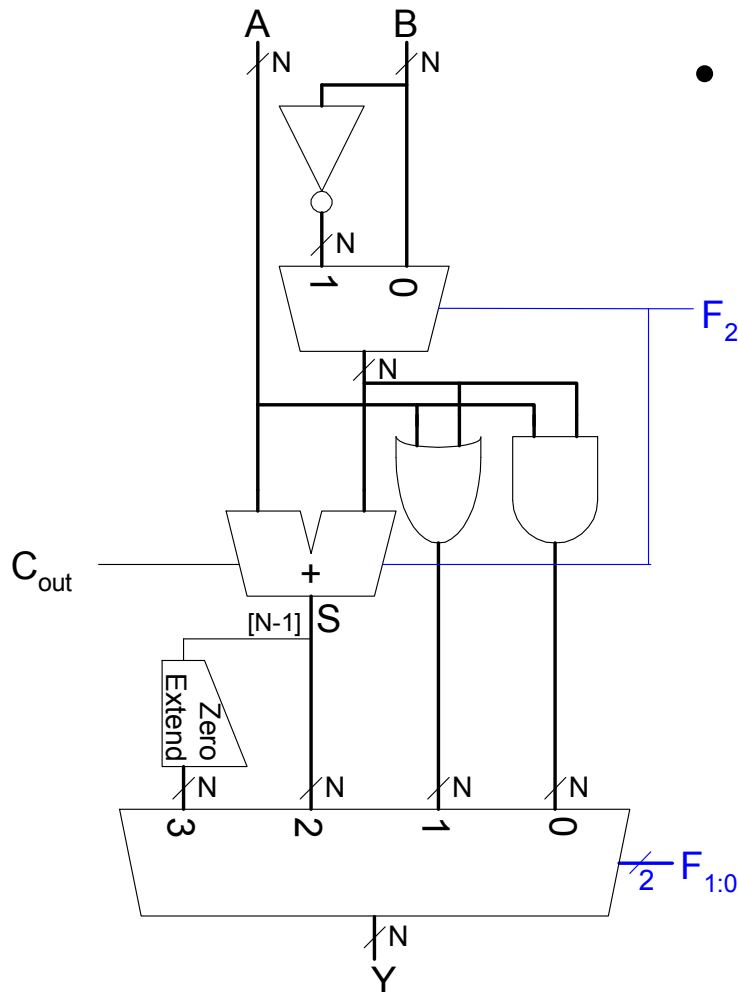
ALU Design



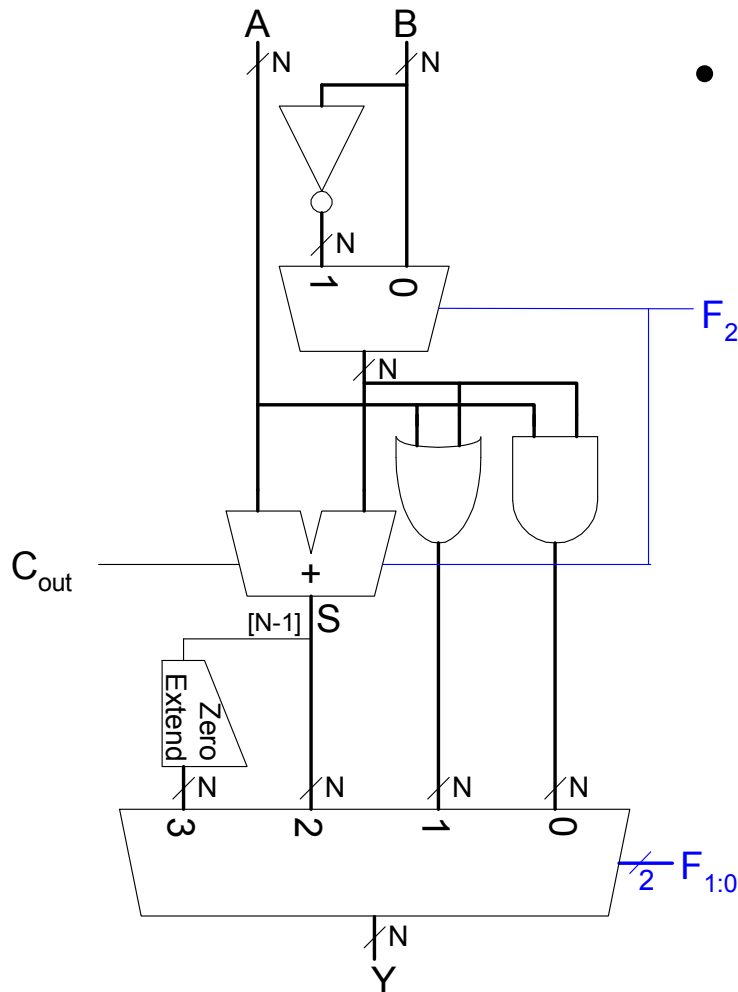
$F_{2:0}$	Function
000	A & B
001	A B
010	A + B
011	not used
100	A & \sim B
101	A \sim B
110	A - B
111	SLT

Set Less Than (SLT) Example

- Configure 32-bit ALU for SLT operation: $A = 25$ and $B = 32$



Set Less Than (SLT) Example



- Configure 32-bit ALU for SLT operation: $A = 25$ and $B = 32$
 - $A < B$, so Y should be 32-bit representation of 1 (0x00000001)
 - $F_{2:0} = 111$
 - $F_2 = 1$ (adder acts as subtracter), so $25 - 32 = -7$
 - -7 has 1 in the most significant bit ($S_{31} = 1$)
 - $F_{1:0} = 11$ multiplexer selects $Y = S_{31}$ (zero extended) = 0x00000001.

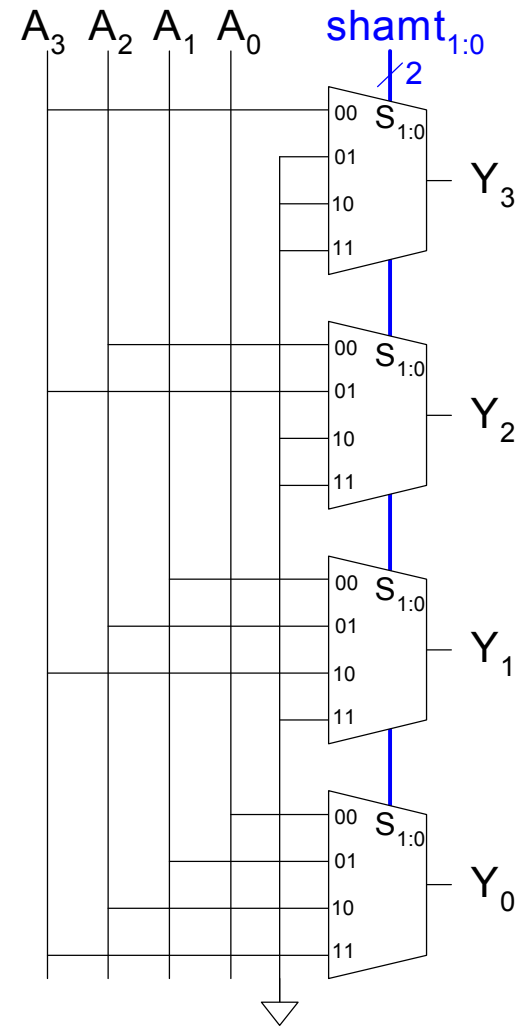
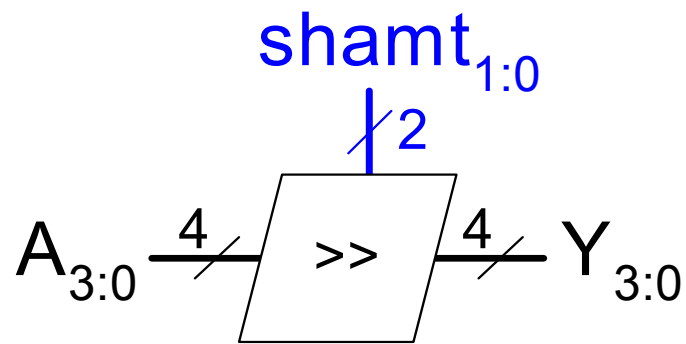
Shifters

- **Logical shifter:** shifts value to left or right and fills empty spaces with 0's
 - Ex: `11001 >> 2 =`
 - Ex: `11001 << 2 =`
- **Arithmetic shifter:** same as logical shifter, but on right shift, fills empty spaces with the old most significant bit (msb).
 - Ex: `11001 >>> 2 =`
 - Ex: `11001 <<< 2 =`
- **Rotator:** rotates bits in a circle, such that bits shifted off one end are shifted into the other end
 - Ex: `11001 ROR 2 =`
 - Ex: `11001 ROL 2 =`

Shifters

- **Logical shifter:**
 - Ex: 11001 >> 2 = 00110
 - Ex: 11001 << 2 = 00100
- **Arithmetic shifter:**
 - Ex: 11001 >>> 2 = 11110
 - Ex: 11001 <<< 2 = 00100
- **Rotator:**
 - Ex: 11001 ROR 2 = 01110
 - Ex: 11001 ROL 2 = 00111

Shifter Design



Shifters as Multipliers, Dividers

- $A \ll N = A \times 2^N$
 - **Example:** $00001 \ll 2 = 00100$ ($1 \times 2^2 = 4$)
 - **Example:** $11101 \ll 2 = 10100$ ($-3 \times 2^2 = -12$)
- $A \gg N = A \div 2^N$
 - **Example:** $01000 \gg 2 = 00010$ ($8 \div 2^2 = 2$)
 - **Example:** $10000 \gg 2 = 11100$ ($-16 \div 2^2 = -4$)

Multipliers

- **Partial products** formed by multiplying a single digit of the multiplier with multiplicand
- **Shifted** partial products **summed** to form result

Decimal

$$\begin{array}{r}
 230 \\
 \times 42 \\
 \hline
 460 \\
 + 920 \\
 \hline
 9660
 \end{array}$$

multiplicand
 multiplier
 partial
 products

result

$$230 \times 42 = 9660$$

Binary

$$\begin{array}{r}
 0101 \\
 \times 0111 \\
 \hline
 0101 \\
 0101 \\
 0101 \\
 + 0000 \\
 \hline
 0100011
 \end{array}$$

$$5 \times 7 = 35$$

Multiplication

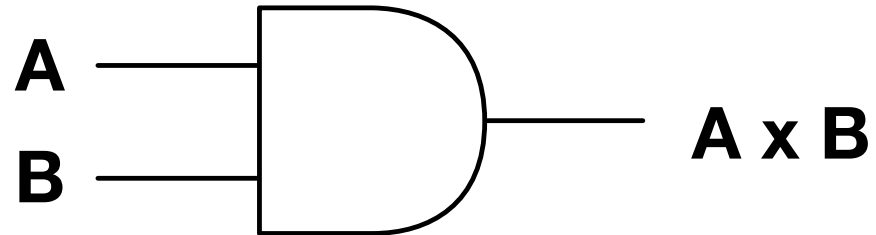
Multiplying **1-bit numbers** is AND operation

$$0 \times 0 = 0$$

$$0 \times 1 = 0$$

$$1 \times 0 = 0$$

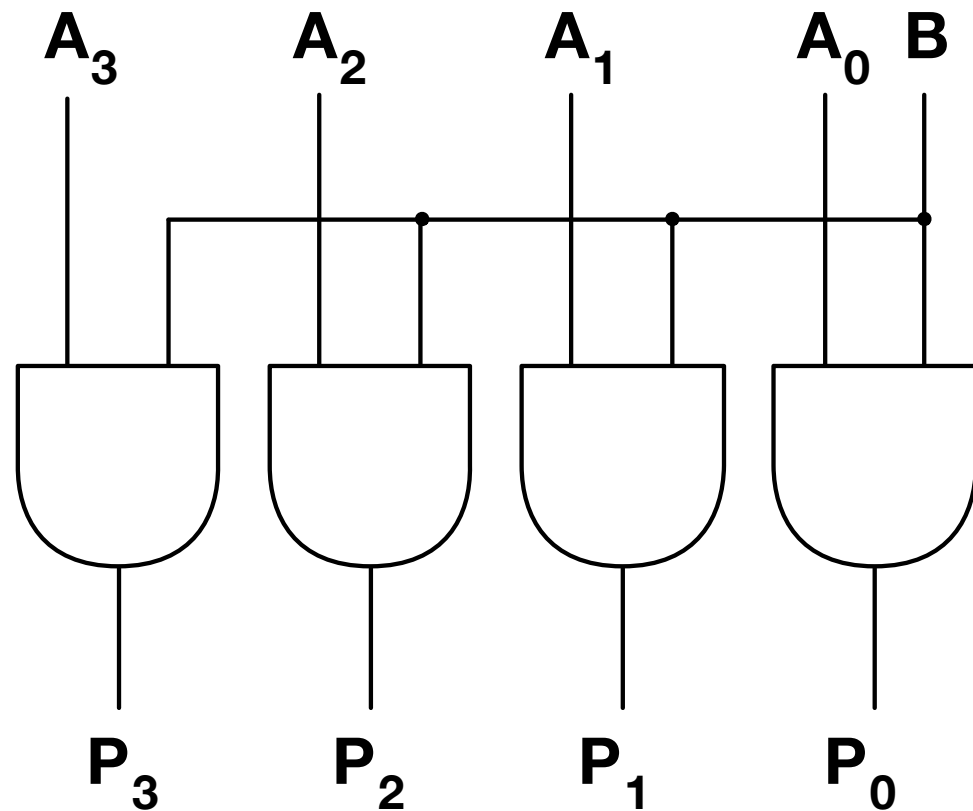
$$1 \times 1 = 1$$



Multiplication

Multiplying **1-bit x N-bit** is AND operation

$A \times B = P$
 $1011 \times 0 = 0000$
 $1011 \times 1 = 1011$



N-Bit x N-Bit Multiplication

Consider 4-bit x 4-bit Multiplication

	A_3	A_2	A_1	A_0
\times	B_3	B_2	B_1	B_0

N-Bit x N-Bit Multiplication

Consider 4-bit x 4-bit Multiplication

$$\begin{array}{rcccc} & A_3 & A_2 & A_1 & A_0 \\ X & B_3 & B_2 & B_1 & B_0 \\ \hline & A_3 \cdot B_0 & A_2 \cdot B_0 & A_1 \cdot B_0 & A_0 \cdot B_0 \end{array}$$

N-Bit x N-Bit Multiplication

Consider 4-bit x 4-bit Multiplication

	A_3	A_2	A_1	A_0
X	B_3	B_2	B_1	B_0
<hr/>				
	$A_3 \cdot B_0$	$A_2 \cdot B_0$	$A_1 \cdot B_0$	$A_0 \cdot B_0$
$A_3 \cdot B_1$	$A_2 \cdot B_1$	$A_1 \cdot B_1$	$A_0 \cdot B_1$	

N-Bit x N-Bit Multiplication

Consider 4-bit x 4-bit Multiplication

		A_3	A_2	A_1	A_0
	X	B_3	B_2	B_1	B_0
		$A_3 \cdot B_0$	$A_2 \cdot B_0$	$A_1 \cdot B_0$	$A_0 \cdot B_0$
	$A_3 \cdot B_1$	$A_2 \cdot B_1$	$A_1 \cdot B_1$	$A_0 \cdot B_1$	
$A_3 \cdot B_2$	$A_2 \cdot B_2$	$A_1 \cdot B_2$	$A_0 \cdot B_2$		

N-Bit x N-Bit Multiplication

Consider 4-bit x 4-bit Multiplication

			A_3	A_2	A_1	A_0
			B_3	B_2	B_1	B_0
			<hr/>			
			$A_3 \cdot B_0$	$A_2 \cdot B_0$	$A_1 \cdot B_0$	$A_0 \cdot B_0$
		$A_3 \cdot B_1$	$A_2 \cdot B_1$	$A_1 \cdot B_1$	$A_0 \cdot B_1$	
	$A_3 \cdot B_2$	$A_2 \cdot B_2$	$A_1 \cdot B_2$	$A_0 \cdot B_2$		
$A_3 \cdot B_3$	$A_2 \cdot B_3$	$A_1 \cdot B_3$	$A_0 \cdot B_3$			

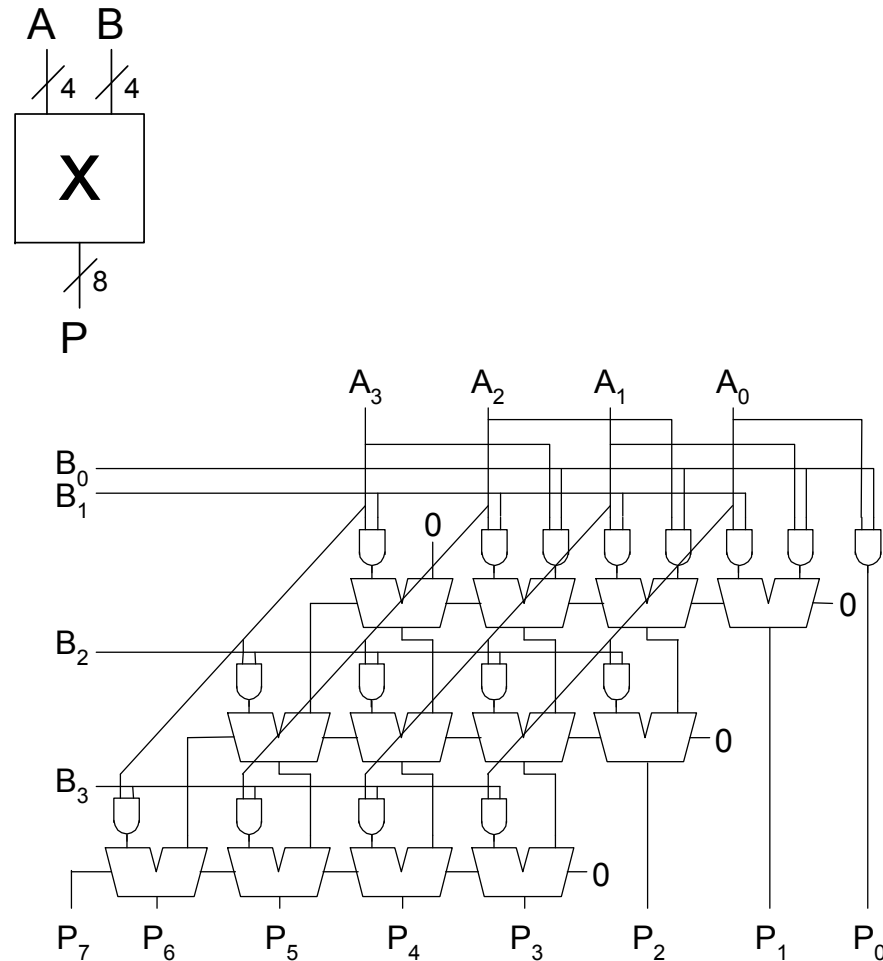
N-Bit x N-Bit Multiplication

Consider 4-bit x 4-bit Multiplication

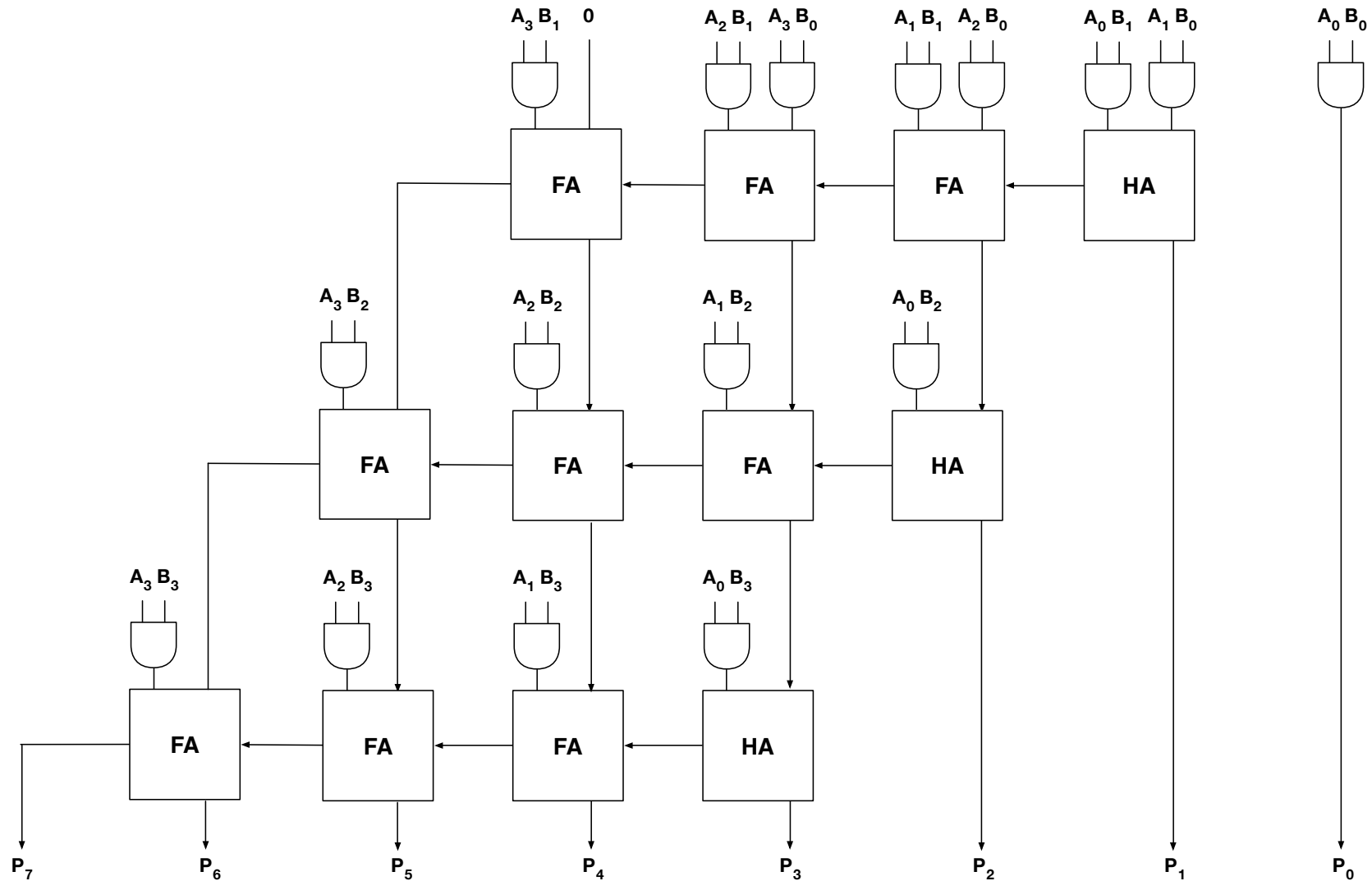
				A_3	A_2	A_1	A_0
			\times	B_3	B_2	B_1	B_0
				$A_3 \cdot B_0$	$A_2 \cdot B_0$	$A_1 \cdot B_0$	$A_0 \cdot B_0$
			$A_3 \cdot B_1$	$A_2 \cdot B_1$	$A_1 \cdot B_1$	$A_0 \cdot B_1$	
		$A_3 \cdot B_2$	$A_2 \cdot B_2$	$A_1 \cdot B_2$	$A_0 \cdot B_2$		
+	$A_3 \cdot B_3$	$A_2 \cdot B_3$	$A_1 \cdot B_3$	$A_0 \cdot B_3$			
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

4 x 4 Multiplier

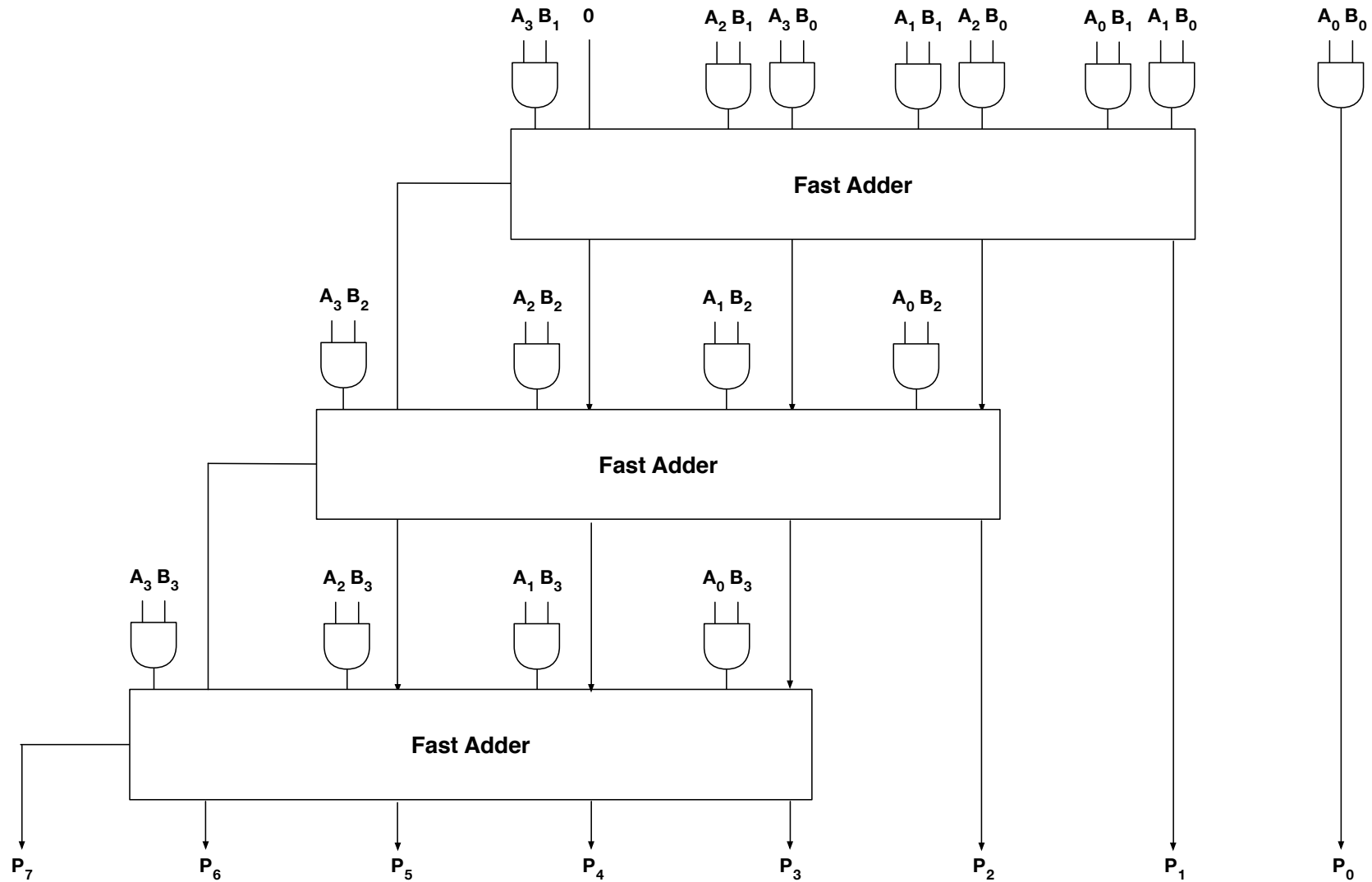
$$\begin{array}{r}
 \begin{array}{cccc}
 & A_3 & A_2 & A_1 & A_0 \\
 \times & B_3 & B_2 & B_1 & B_0 \\
 \hline
 & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 \\
 A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & \\
 A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & \\
 + & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 \\
 \hline
 P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
 \end{array}
 \end{array}$$



Using Carry Propagate Adder



Using Fast Adders



Large Multipliers

Construct large multipliers out of smaller multipliers

Construct 2N-bit x 2N-bit multiplier using N-bit x N-bit multipliers

Let A and B be 2N-bit numbers such that:

$$A = \underbrace{A_{2N-1}A_{2N-2} \dots A_N}_{A_H} \underbrace{A_{N-1}A_{N-2} \dots A_0}_{A_L}$$

$$B = \underbrace{B_{2N-1}B_{2N-2} \dots B_N}_{B_H} \underbrace{B_{N-1}B_{N-2} \dots B_0}_{B_L}$$

2N-Bit x 2N-Bit Multiplier

A_H is the upper N bits of A and A_L is the lower N bits of A

B_H is the upper N bits of B and B_L is the lower N bits of B

Therefore,

$$A = A_H \times 2^N + A_L \qquad B = B_H \times 2^N + B_L$$

And so,

$$\begin{aligned} A \times B &= (A_H \times 2^N + A_L) \times (B_H \times 2^N + B_L) \\ &= A_H B_H 2^{2N} + (A_H B_L + A_L B_H) 2^N + A_L B_L \end{aligned}$$

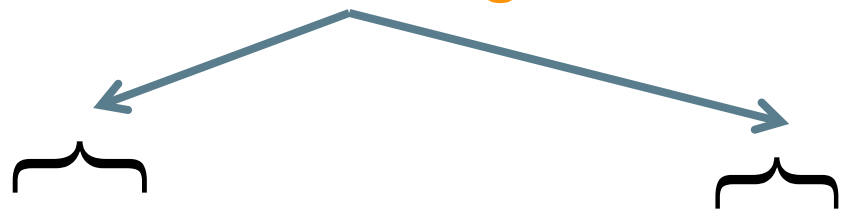
2N-Bit x 2N-Bit Multiplier

$$A \times B = \underbrace{A_H B_H}_{\text{N-Bit x N-Bit Multiplier}} 2^{2N} + \left(\underbrace{A_H A_L}_{\text{N-Bit x N-Bit Multiplier}} + \underbrace{A_L B_H}_{\text{N-Bit x N-Bit Multiplier}} \right) 2^N + \underbrace{A_L B_L}_{\text{N-Bit x N-Bit Multiplier}}$$

N-Bit x N-Bit Multipliers

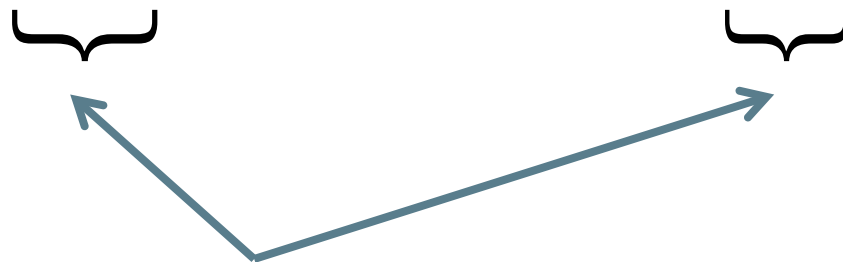
2N-Bit x 2N-Bit Multiplier

Bit Shifting


$$A \times B = A_H B_H 2^{2N} + (A_H A_L + A_L B_H) 2^N + A_L B_L$$

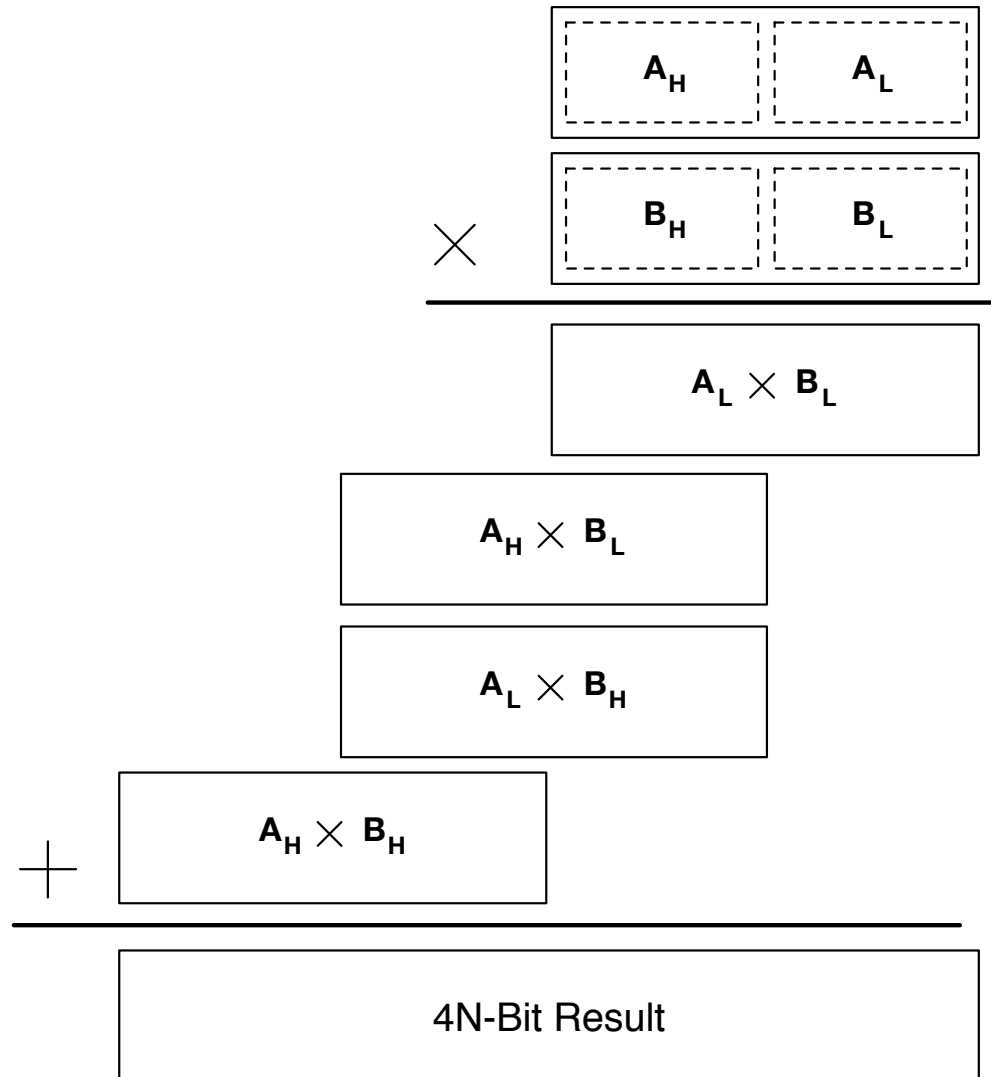
2N-Bit x 2N-Bit Multiplier

$$A \times B = A_H B_H 2^{2N} + (A_H A_L + A_L B_H) 2^N + A_L B_L$$



2N-Bit Adders

2N-Bit x 2N-Bit Multiplier



Aside: Sign Extension

Converting a 4-bit representation to an 8-bit representation:

If it is unsigned, just append with 0's:

$$0\ 1\ 0\ 1 = 5$$

$$0\ 0\ 0\ 0\ 0\ 1\ 0\ 1 = 5$$

If the number is a negative number (represented by two's complement) extend with 1's rather than 0's:

$$1\ 1\ 0\ 1 = -3$$

$$1\ 1\ 1\ 1\ 1\ 1\ 0\ 1 = -3$$

In general, take the MSB of the shorter representation, and sign extend with that value.

Signed Multiplier

Previous architecture assumes multiplier and multiplicand are unsigned

If they are signed, we need to do things a bit differently:


Suppose we want to multiply $(-1) * (2)$ [assumed two's compl. Repr]

$$\begin{array}{r} 1\ 1\ 1\ 1 \\ \times 0\ 0\ 1\ 0 \\ \hline \end{array}$$

Wrong answer!


Signed Multiplier

Problem: Need to sign-extend


$$\begin{array}{r} 1111 \\ \times 0010 \\ \hline 0000 \\ 1111 \\ 0000 \\ 0000 \\ \hline 00011110 = 30 \end{array}$$

Signed Multiplier

Sign Extended all
numbers


$$\begin{array}{r} 1111 \\ x0010 \\ \hline 00000000 \\ 1111111 \\ 00000 \\ 0000 \\ \hline 11111110 = -2 \end{array}$$

Signed Multiplier

This is not enough if the second number is negative:

$$\begin{array}{r} 0110 = 6 \\ \times 1111 = -1 \\ \hline 0000110 \\ 000110 \\ 00110 \\ 0110 \\ \hline 1011010 = -38 \end{array}$$

Signed Multiplier

Trick: subtract the last number rather than add it.

Easy way to do this is to replace the last number by it's negative value and then add as normal.

remember: $(-A)$ can be written by flipping all bits of A and adding 1

0 1 1 0 = 6		0 1 1 0 = 6
x 1 1 1 1 = -1		x 1 1 1 1 = -1
<hr/>		<hr/>
0 0 0 0 1 1 0		0 0 0 0 1 1 0
0 0 0 1 1 0		0 0 0 1 1 0
0 0 1 1 0		0 0 1 1 0
0 1 1 0	flip bits and add 1 →	1 0 1 0
<hr/>		<hr/>
1 0 1 1 0 1 0 = -38		1 1 1 1 0 1 0 = -6

Fmax For A Combinational Multiplier

In VHDL,

$$P := A * B;$$

(make sure **P** has twice the number of bits of **A** and **B**)

Synthesis tool will create a combinational multiplier

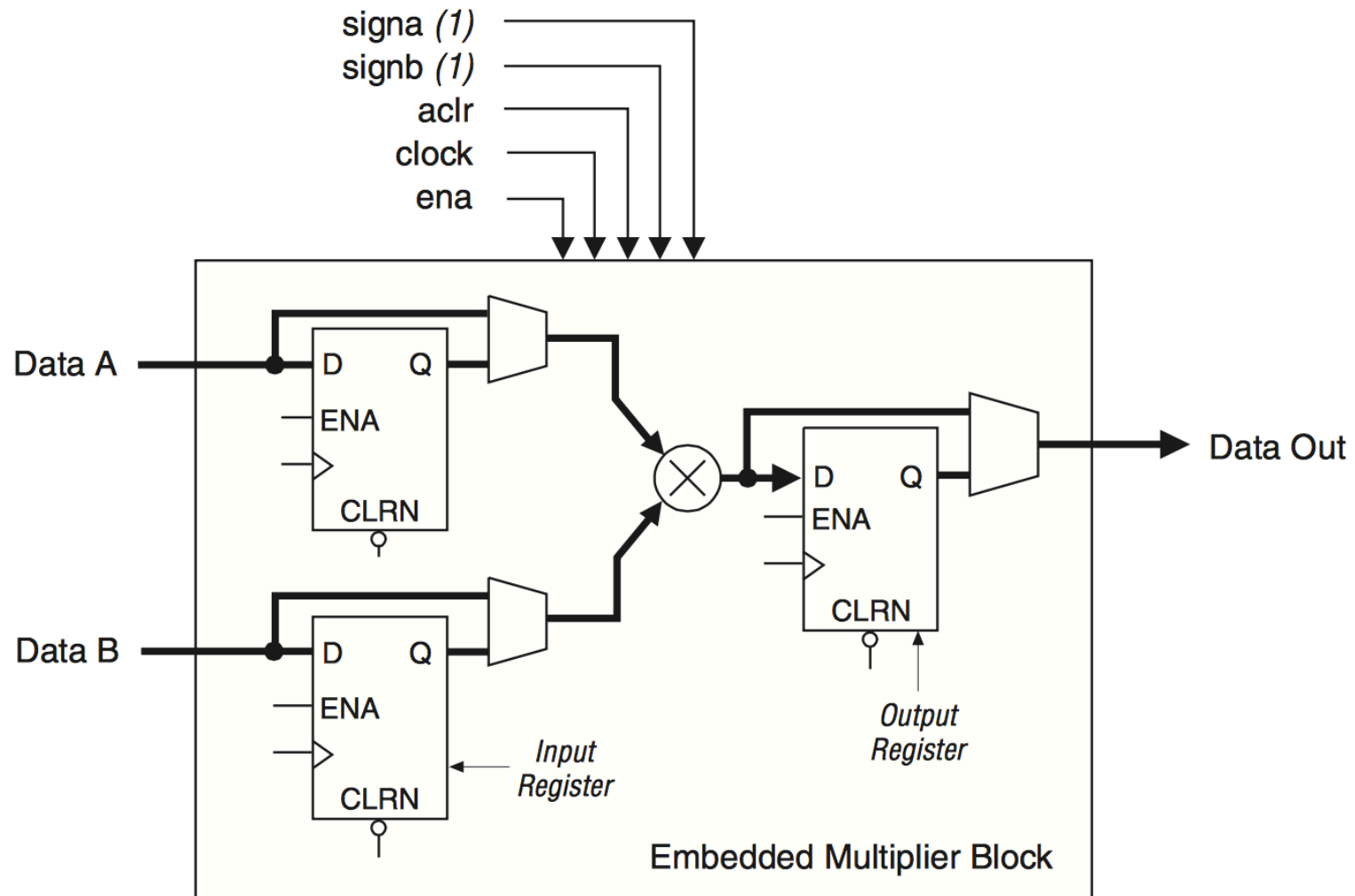
Multipliers can be implemented in either:

- Dedicated multiplier blocks embedded into the FPGA
- Converted to logic and mapped to lookup-tables

Quartus II will prefer to use the dedicated multiplier blocks, until they run out, and then use lookup-tables.

- You can change setting to forbid the use of multiplier blocks

Implementing Array Multipliers in our Device



Implementing Array Multipliers in our Device

Device	Embedded Multiplier Columns	Embedded Multipliers	9 × 9 Multipliers	18 × 18 Multipliers
EP2C5	1	13	26	13
EP2C8	1	18	36	18
EP2C15	1	26	52	26
EP2C20	1	26	52	26
EP2C35	1	35	70	35
EP2C50	2	86	172	86
EP2C70	3	150	300	150

In our Cyclone II device...

8x8 bit multiplier:

- Using embedded multipliers: 1 multiplier block, 260 MHz
- Using lookup-tables: 103 logic elements, 153 MHz

16x16 bit multiplier:

- Using embedded multipliers: 1 multiplier block, 260 MHz
- Using lookup-tables: 346 logic elements, 105 MHz

32x32 bit multiplier:

- Using embedded multipliers: 4 multiplier blocks, 106 MHz
- Using lookup-tables: 1434 logic elements, 70 MHz

Serial (Multi-cycle) Multiplier

Multi Cycle Multiplier

How do you multiply by hand?

$$\begin{array}{r} \mathbf{1101} \quad \mathbf{A} \\ \times \mathbf{1011} \quad \mathbf{B} \\ \hline \mathbf{1101} \\ \mathbf{11010} \\ \mathbf{000000} \\ + \mathbf{1101000} \\ \hline \mathbf{10001111} \quad \mathbf{P} \end{array}$$

One Possible Algorithm

$$\begin{array}{r} \mathbf{1101} \quad \mathbf{A} \\ \times \mathbf{1011} \quad \mathbf{B} \\ \hline \mathbf{1101} \\ \mathbf{11010} \\ \mathbf{000000} \\ + \mathbf{1101000} \\ \hline \mathbf{10001111} \quad \mathbf{P} \end{array}$$

```
P = 0
while B != 0:
    if B(0) == 1:
        P = P + A
    A = A << 1
    B = B >> 1
```

Note: This is NOT VHDL

Example

$$13 \times 11 = 143$$

$$1101 \times 1011 = 10001111$$

				1	1	0	1
--	--	--	--	---	---	---	---

A

1	0	1	1
---	---	---	---

B

							0
--	--	--	--	--	--	--	---

P

```
P = 0
while B != 0:
    if B(0) == 1:
        P = P + A
    A = A << 1
    B = B >> 1
```

Example

$$13 \times 11 = 143$$

$$1101 \times 1011 = 10001111$$

				1	1	0	1
--	--	--	--	---	---	---	---

A

1	0	1	1
---	---	---	---

B

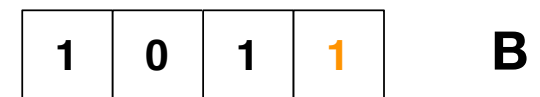
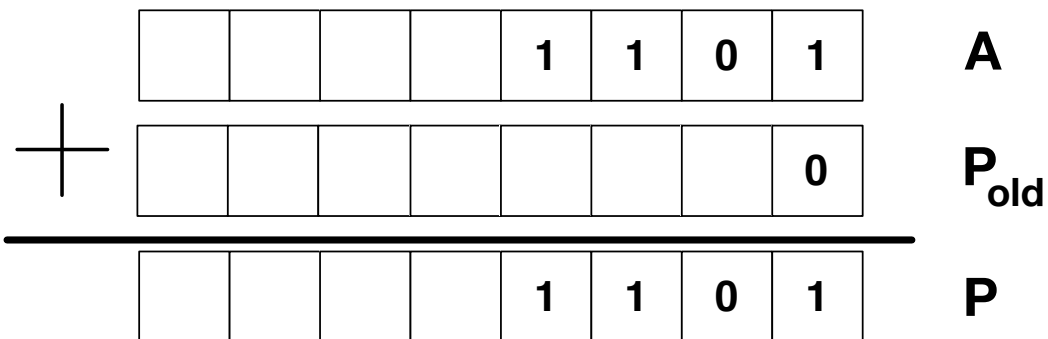
							0
--	--	--	--	--	--	--	---

P

```
P = 0
while B != 0:
    if B(0) == 1:
        P = P + A
    A = A << 1
    B = B >> 1
```

Example

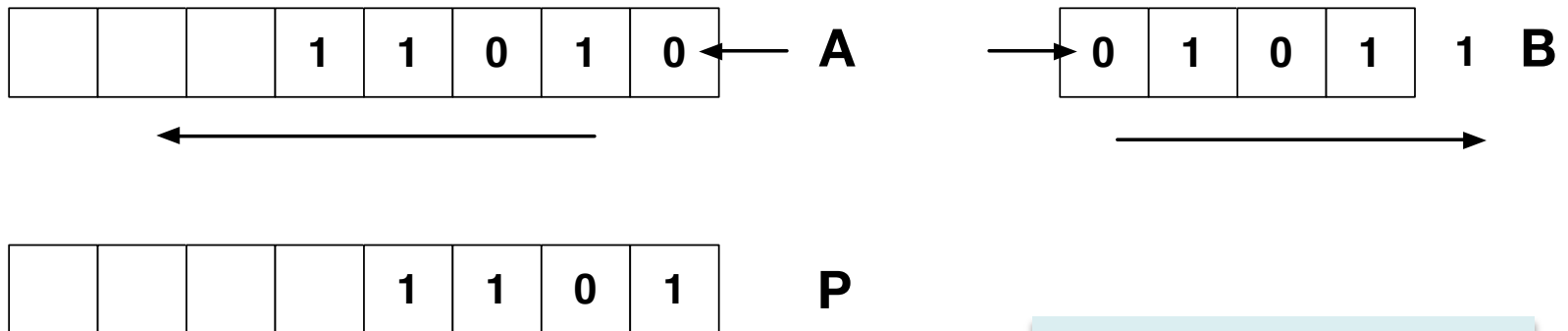
$$13 \times 11 = 143$$
$$1101 \times 1011 = 10001111$$



```
P = 0
while B != 0:
    if B(0) == 1:
        P = P + A
    A = A << 1
    B = B >> 1
```

Example

$$13 \times 11 = 143$$
$$1101 \times 1011 = 10001111$$



```
P = 0
while B != 0:
    if B(0) == 1:
        P = P + A
    A = A << 1
    B = B >> 1
```

Example

$$13 \times 11 = 143$$

$$1101 \times 1011 = 10001111$$

			1	1	0	1	0
--	--	--	---	---	---	---	---

A

0	1	0	1
---	---	---	---

B

				1	1	0	1
--	--	--	--	---	---	---	---

P

```
P = 0
while B != 0:
    if B(0) == 1:
        P = P + A
    A = A << 1
    B = B >> 1
```

Example

$$13 \times 11 = 143$$
$$1101 \times 1011 = 10001111$$

			1	1	0	1	0
				1	1	0	1
<hr/>							
		1	0	0	1	1	1

A

P_{old}

P

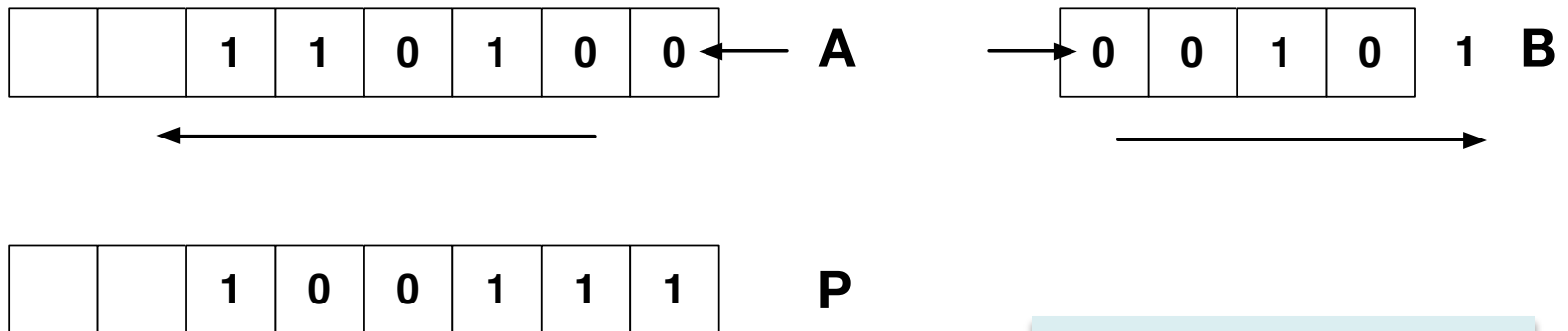
0	1	0	1
---	---	---	---

B

```
P = 0
while B != 0:
    if B(0) == 1:
        P = P + A
    A = A << 1
    B = B >> 1
```

Example

$$13 \times 11 = 143$$
$$1101 \times 1011 = 10001111$$



```
P = 0
while B != 0:
    if B(0) == 1:
        P = P + A
    A = A << 1
    B = B >> 1
```

Example

$$13 \times 11 = 143$$

$$1101 \times 1011 = 10001111$$

		1	1	0	1	0	0
--	--	---	---	---	---	---	---

A

0	0	1	0
---	---	---	---

B

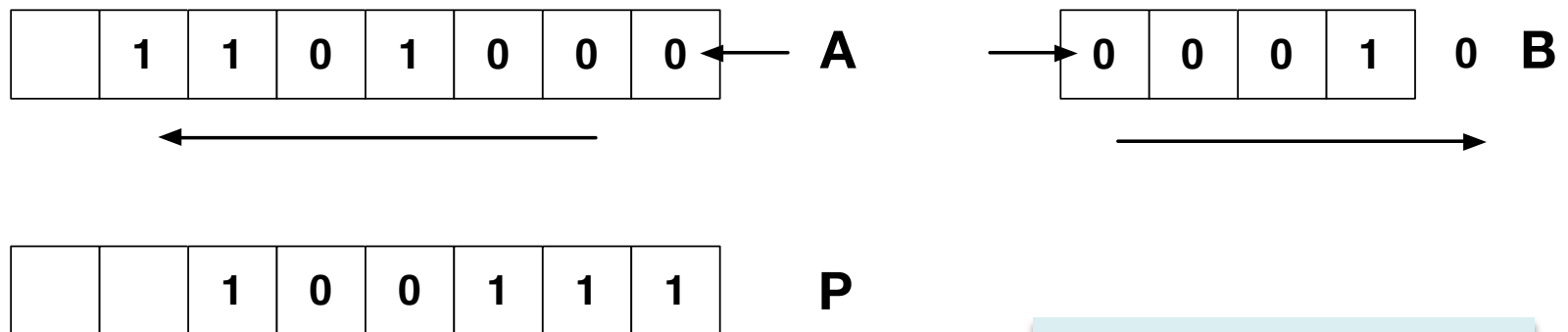
		1	0	0	1	1	1
--	--	---	---	---	---	---	---

P

```
P = 0
while B != 0:
    if B(0) == 1:
        P = P + A
    A = A << 1
    B = B >> 1
```


Example

$$13 \times 11 = 143$$
$$1101 \times 1011 = 10001111$$



```
P = 0
while B != 0:
    if B(0) == 1:
        P = P + A
    A = A << 1
    B = B >> 1
```

Example

$$13 \times 11 = 143$$

$$1101 \times 1011 = 10001111$$

	1	1	0	1	0	0	0
--	---	---	---	---	---	---	---

A

0	0	0	1
---	---	---	---

B

		1	0	0	1	1	1
--	--	---	---	---	---	---	---

P

```
P = 0
while B != 0:
    if B(0) == 1:
        P = P + A
    A = A << 1
    B = B >> 1
```

Example

$$13 \times 11 = 143$$

$$1101 \times 1011 = 10001111$$

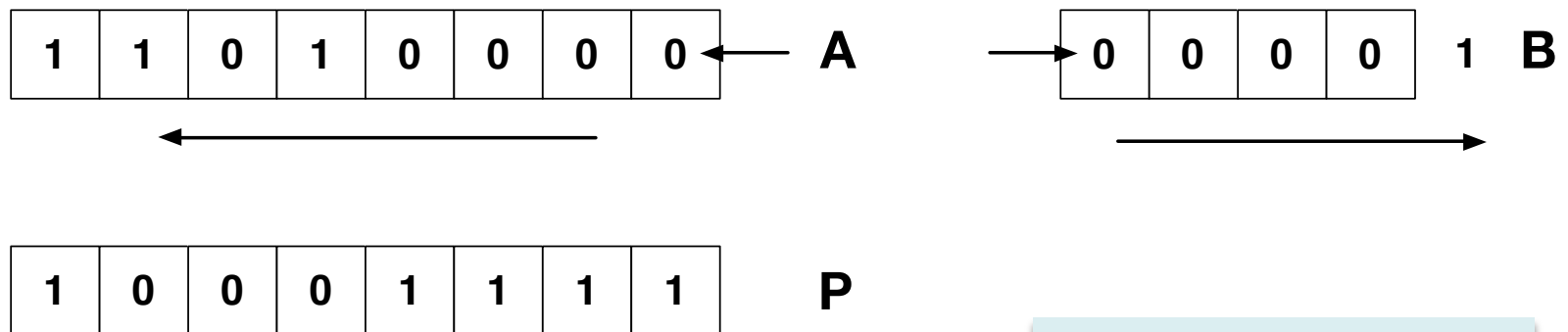
		1	1	0	1	0	0	0	A
+			1	0	0	1	1	1	P_{old}
<hr/>									
	1	0	0	0	1	1	1	1	P

0	0	0	1	B
---	---	---	---	----------

```
P = 0
while B != 0:
    if B(0) == 1:
        P = P + A
    A = A << 1
    B = B >> 1
```

Example

$$13 \times 11 = 143$$
$$1101 \times 1011 = 10001111$$



```
P = 0
while B != 0:
    if B(0) == 1:
        P = P + A
    A = A << 1
    B = B >> 1
```

Example

$$13 \times 11 = 143$$

$$1101 \times 1011 = 10001111$$

1	1	0	1	0	0	0	0
---	---	---	---	---	---	---	---

A

0	0	0	0
---	---	---	---

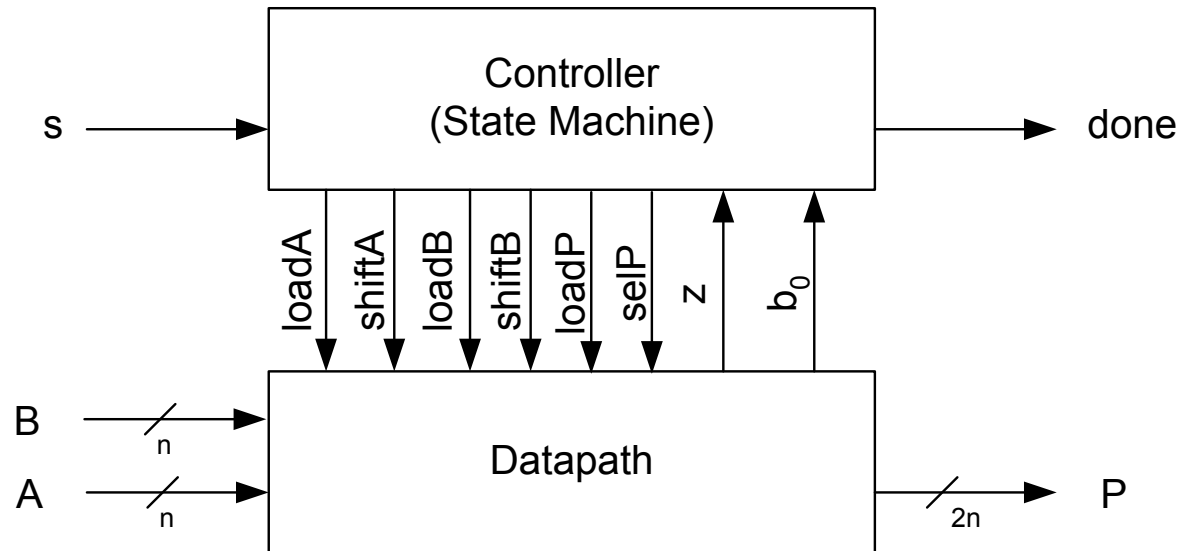
B

1	0	0	0	1	1	1	1
---	---	---	---	---	---	---	---

P

```
P = 0
while B != 0:
    if B(0) == 1:
        P = P + A
    A = A << 1
    B = B >> 1
```

Top – Level Schematic

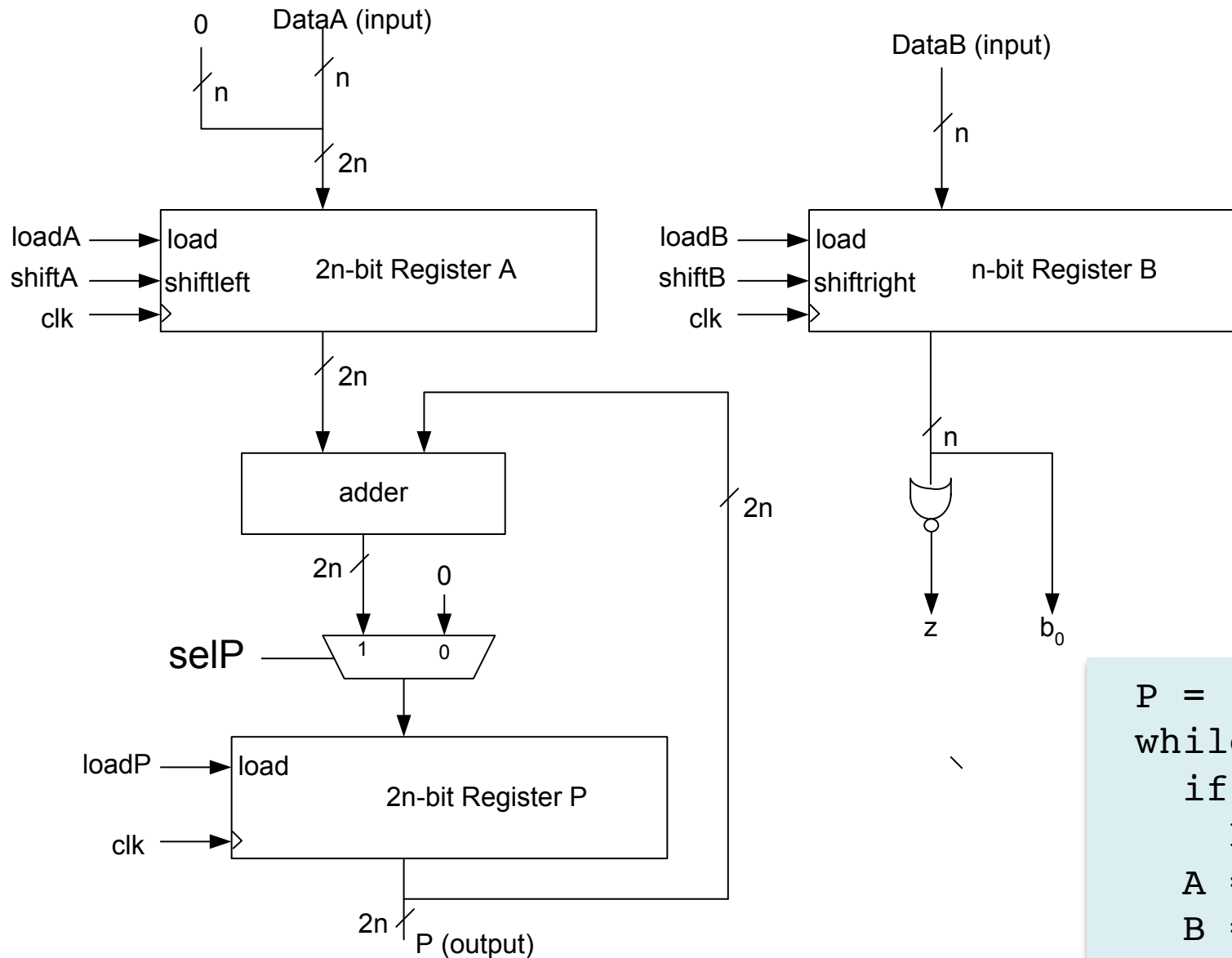


When **s** goes high, **n**-bit values are available on **A** and **B**.

The machine then multiplies, and when it is finished, asserts **done** and puts the result on **P**.

It then waits for **s** to go low.

Datapath



```
P = 0
while B != 0
    if B(0) == 1
        P = P + A
    A = A << 1
    B = B >> 1
```

DIVISION

Fmax For A Combinational Divider

In VHDL you can infer a combinational divider:

$$Q := A / B;$$

Measured on our Cyclone FPGA:

8 bits / 8 bits: Fmax = 79 MHz

16 bits / 16 bits: Fmax = 25 MHz

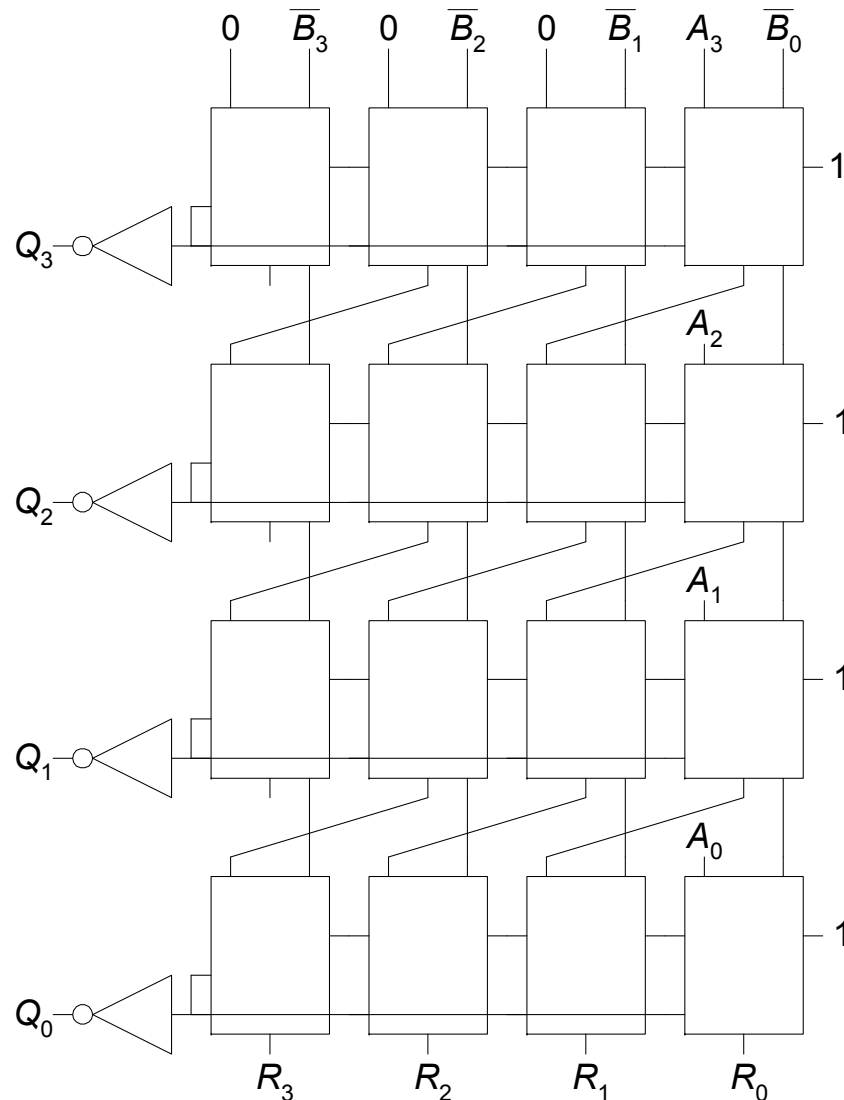
32 bits / 32 bits: Fmax = 9 MHz

64 bits / 64 bits: Fmax = 3 MHz (uses 13% of the logic resources!)

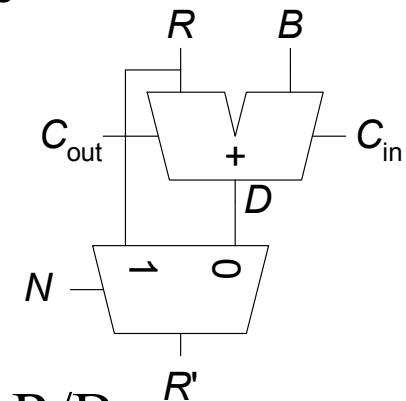
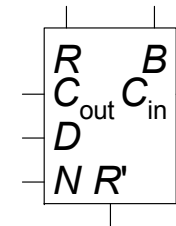
Too slow and too large.

Almost always use multi-cycle divider in practice

4 x 4 Divider



Legend



$$A/B = Q + R/B$$

Algorithm:

$$R' = 0$$

for $i = N-1$ to 0

$$R = \{R' \ll 1, A_i\}$$

$$D = R - B$$

if $D < 0$, $Q_i = 0$, $R' = R$

else $Q_i = 1$, $R' = D$

$$R' = R$$



Number Systems

- Numbers we can represent using binary representations
 - **Positive numbers**
 - Unsigned binary
 - **Negative numbers**
 - Two's complement
 - Sign/magnitude numbers
- What about **fractions**?

Numbers with Fractions

- Two common notations:
 - **Fixed-point:** binary point fixed
 - **Floating-point:** binary point floats to the right of the most significant 1

Dealing With Fractional Values

Two ways to represent non-integer numbers in binary

Fixed Point Representation

- Smaller hardware, simpler to implement. You'll do this in lab 4

Floating Point Representation

- Larger range of values, more accurate at extremes

Fractional Numbers

In decimal (base 10)

7241.0381

=

$$7 \times 10^3 + 2 \times 10^2 + 4 \times 10^1 + 1 \times 10^0 + 0 \times 10^{-1} + 3 \times 10^{-2} + 8 \times 10^{-3} + 1 \times 10^{-4}$$

In binary (base 2)

1001.0110

=

$$1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4}$$

Fixed-Point Numbers

- 6.75 using 4 integer bits and 4 fraction bits:

01101100

0110.1100

$$2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$$

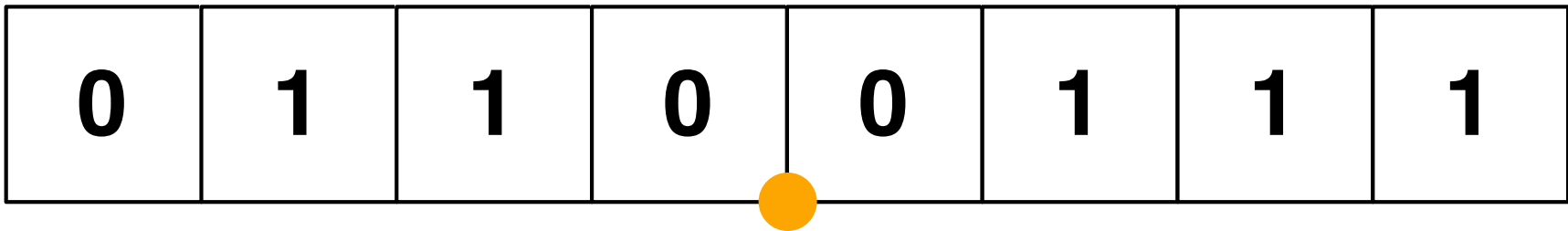
- Binary point is implied
- The number of integer and fraction bits must be agreed upon beforehand



Fixed Point Representation

Uses N-bits to represent a number

Location of radix point is FIXED



Location of radix point determines range and precision

Fixed-Point Number Example

- Represent 7.5_{10} using 4 integer bits and 4 fraction bits.

Fixed-Point Number Example

- Represent 7.5_{10} using 4 integer bits and 4 fraction bits.

01111000

Signed Fixed-Point Numbers

- **Representations:**
 - Sign/magnitude
 - Two's complement
- **Example:** Represent -7.5_{10} using 4 integer and 4 fraction bits
 - **Sign/magnitude:**
 - **Two's complement:**

Signed Fixed-Point Numbers

- **Representations:**
 - Sign/magnitude
 - Two's complement
- **Example:** Represent -7.5_{10} using 4 integer and 4 fraction bits

- **Sign/magnitude:**

11111000

- **Two's complement:**

1. +7.5: 01111000

2. Invert bits: 10000111

3. Add 1 to lsb: + 1
 —————
 10001000

Arithmetic with Fixed Point Numbers

Addition and Subtraction

- Treat bits as integers and perform operation. Make sure the decimal point is assumed to be in the same position for both operands

0	0	0	0	1	1	.	0	1	3.25
0	0	0	1	1	0	.	1	0	6.5
<hr/>									
0	0	1	0	0	1	.	1	1	9.75

```
signal position_x : unsigned(15 downto 0);
```

```
signal velocity_y : unsigned(15 downto 0);
```

```
position_x <= position_x + velocity_x;
```

Arithmetic with Fixed Point Numbers

Multiplication and Division

- Treat bits as integers and perform operation, but need to pay attention to radix point in result and shift accordingly

Example

0	0	0	0	1	1	.	0	1		3.25
0	0	0	1	1	0	.	1	0		6.5
<hr/>										
1	0	1	0	1	.	0	0	1	0	21.125

If your system is using 8-bit Fixed Point Numbers with 2 fractional bits, need to shift the result to 010101.00

Leads to loss of precision (result is 21)

If only we could have used 3 of the 8 bits for the fraction! → 10101.001

Another Way to Look at Numbers

$$\textit{significand} \times \textit{base}^{\textit{exponent}}$$

Decimal (base 10) Example:

$$7241.0381 = 72410381 \times 10^{-4}$$

In Fixed Point representation, radix point is fixed because exponent is fixed

In Floating Point Representation, radix point **FLOATS
because exponent can change**

Floating-Point Numbers

- Binary point floats to the right of the most significant 1
- Similar to decimal scientific notation

- For example, write 273_{10} in scientific notation:

$$273 = 2.73 \times 10^2$$

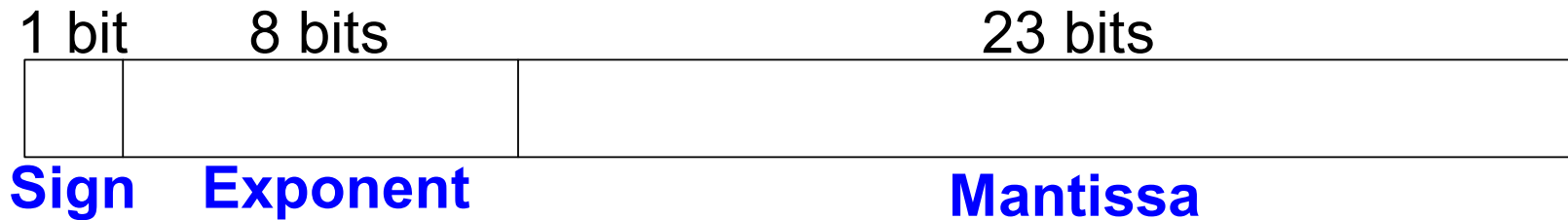
- In general, a number is written in scientific notation as:

$$\pm M \times B^E$$

- M = mantissa
- B = base
- E = exponent
- In the example, $M = 2.73$, $B = 10$, and $E = 2$



Floating-Point Numbers



- **Example:** represent the value 228_{10} using a 32-bit floating point representation

We show three versions –final version is called the **IEEE 754 floating-point standard**

Floating-Point Representation 1

1. Convert decimal to binary (**don't reverse steps 1 & 2!**):

$$228_{10} = 11100100_2$$

2. Write the number in “binary scientific notation”:

$$11100100_2 = 1.11001_2 \times 2^7$$

3. Fill in each field of the 32-bit floating point number:

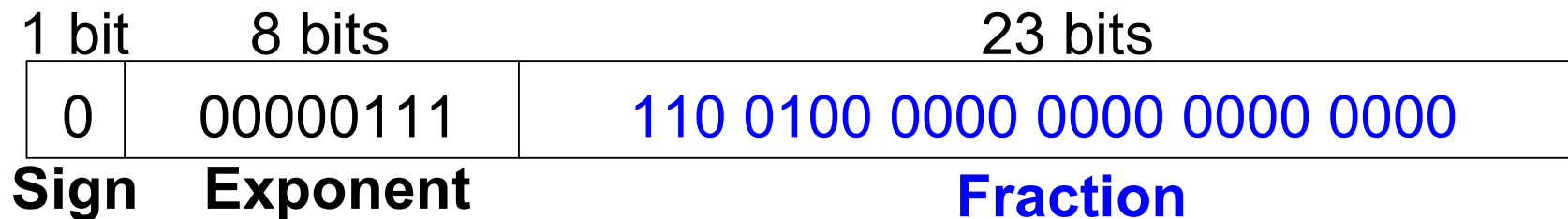
- The sign bit is positive (0)
- The 8 exponent bits represent the value 7
- The remaining 23 bits are the mantissa

1 bit	8 bits	23 bits
0	00000111	11 1001 0000 0000 0000 0000
Sign	Exponent	Mantissa



Floating-Point Representation 2

- First bit of the mantissa is always 1:
 - $228_{10} = 11100100_2 = \mathbf{1.11001} \times 2^7$
- So, no need to store it: *implicit leading 1*
- Store just fraction bits in 23-bit field



Floating-Point Representation 3

- *Biased exponent*: bias = 127 (01111111_2)

- Biased exponent = bias + exponent
- Exponent of 7 is stored as:

$$127 + 7 = 134 = 0x10000110_2$$

- The **IEEE 754 32-bit floating-point representation** of 228_{10}

1 bit	8 bits	23 bits
0	10000110	110 0100 0000 0000 0000 0000
Sign	Biased Exponent	Fraction

in hexadecimal: **0x43640000**

Floating-Point Example

Write -58.25_{10} in floating point (IEEE 754)

Floating-Point Example

Write -58.25_{10} in floating point (IEEE 754)

1. Convert decimal to binary:

$$58.25_{10} = 111010.01_2$$

2. Write in binary scientific notation:

$$1.1101001 \times 2^5$$

3. Fill in fields:

Sign bit: 1 (negative)

8 exponent bits: $(127 + 5) = 132 = 10000100_2$

23 fraction bits: 110 1001 0000 0000 0000 0000

1 bit	8 bits	23 bits
1	100 0010 0	110 1001 0000 0000 0000 0000
Sign	Exponent	Fraction

in hexadecimal: 0xC2690000



Floating-Point: Special Cases

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000000000000
∞	0	11111111	00000000000000000000000000000000
$-\infty$	1	11111111	00000000000000000000000000000000
NaN	X	11111111	non-zero

Floating-Point Precision

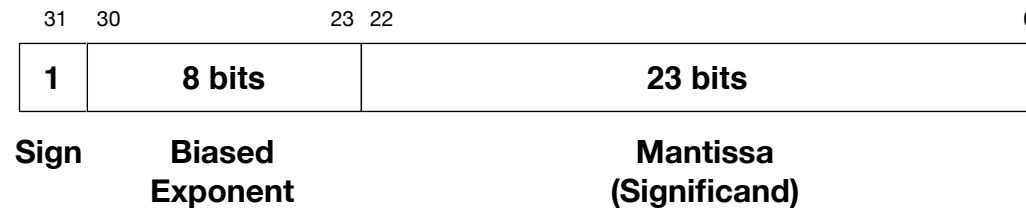
- **Single-Precision:**
 - 32-bit
 - 1 sign bit, 8 exponent bits, 23 fraction bits
 - bias = 127
- **Double-Precision:**
 - 64-bit
 - 1 sign bit, 11 exponent bits, 52 fraction bits
 - bias = 1023

IEEE 754 Binary Floating Point Standard

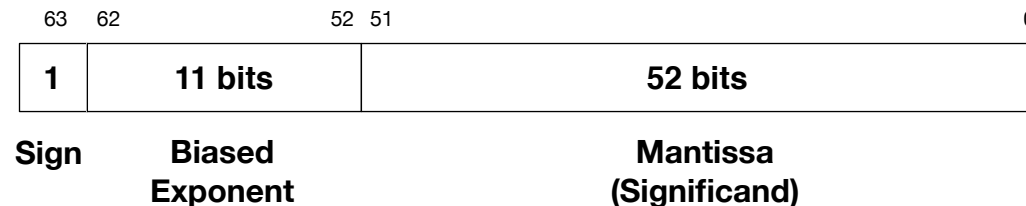
Binary floating point encoding scheme established in 1985 and used in almost every electronic/computing system

Most commonly used formats from the standard:

Single-Precision (32-bits)

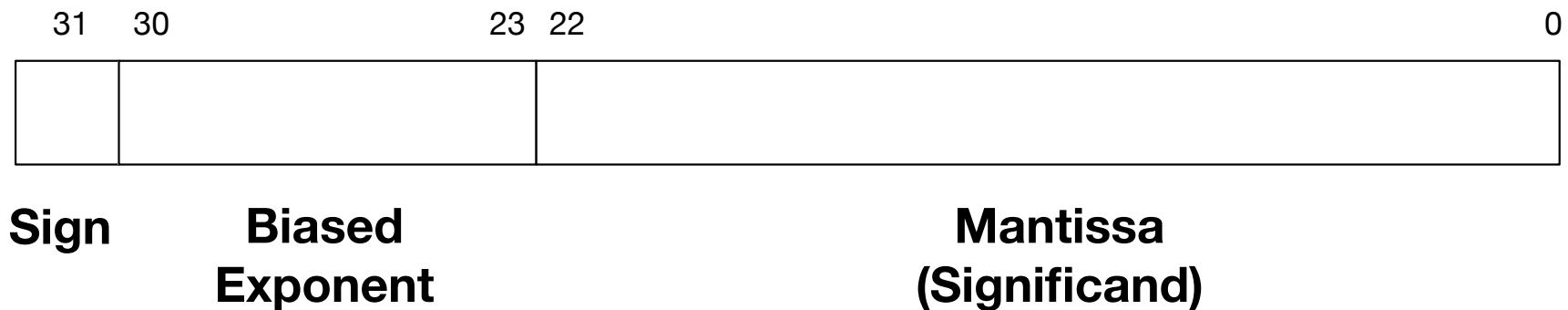


Double-Precision (64-bits)



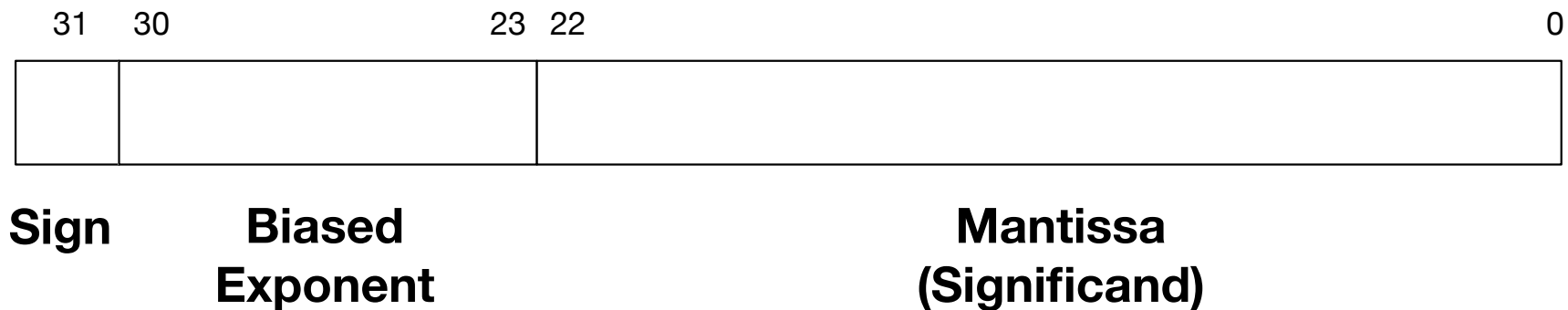
Converting a Number to IEEE754

$$50.5625_{10} = 110010.1001_2$$



Converting a Number to IEEE754

$$110010.1001 \rightarrow 1.100101001 \times 2^5$$

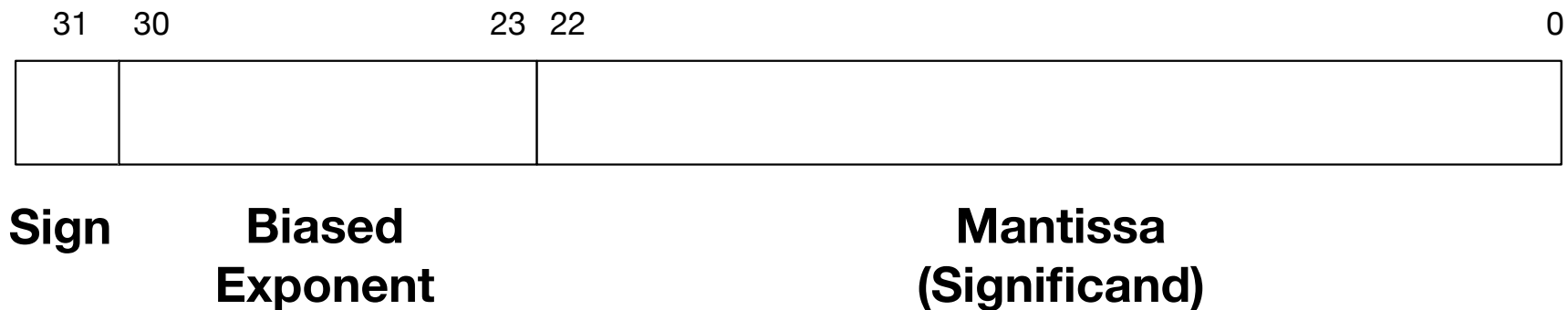


Convert number to scientific notation

This is called **NORMALIZATION**

Converting a Number to IEEE754

$$110010.1001 \rightarrow 1.100101001 \times 2^5$$



First bit of a normalized non-zero binary
value is **ALWAYS** 1
Don't need to store it

$$110010.1001 \rightarrow 1.100101001 \times 2^5$$


$$110010.1001 \rightarrow 1.100101001 \times 2^5$$


Converting a Number to IEEE754

$$110010.1001 \rightarrow 1.100101001 \times 2^5$$

$$5_{10} \rightarrow 101_2$$

Exponent needs to be signed

But Two's Complement makes comparisons more complex

Add a BIAS (offset) to put exponent into unsigned range

Exponent Bias

$$\text{bias} = 2^{k-1} - 1$$

where k is the number of bits used to store exponent

Single-Precision:

$$k = 8 \rightarrow \text{bias} = 2^{8-1} - 1 = 127$$

exponent in range of -126..+127

exponent after bias in range of 1..254 (0, 255 have special meaning)

Double-Precision:

$$k = 11 \rightarrow \text{bias} = 2^{11-1} - 1 = 1023$$

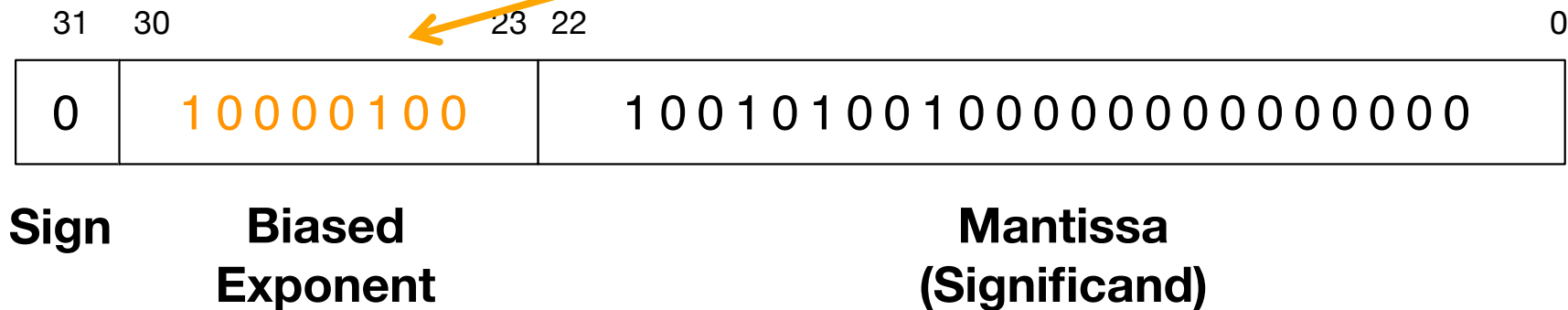
exponent in range of -1022..+1023

exponent after bias in range of 1..2046

Converting a Number to IEEE754

$$110010.1001 = 1.100101001 \times 2^5$$

$$\text{Biased exponent: } 5_{10} + 127_{10} = 10000100_2$$



Converting From IEEE754

$$(-1)^{sign} \times 1.(mantissa) \times 2^{exponent-bias}$$

Diagram illustrating the IEEE 754 single-precision floating-point format (32 bits):

- Sign:** 1 bit (bit 31)
- Biased Exponent:** 8 bits (bits 30-23)
- Mantissa (Significand):** 23 bits (bits 22-0)

The bit pattern shown is: 0 10000100 10010100100000000000000

$$(-1)^0 \times 1.100101001_2 \times 2^{10000100_2 - 127_{10}} = 50.562510_{10}$$

Floating-Point: Rounding

- **Overflow:** number too large to be represented
- **Underflow:** number too small to be represented
- **Rounding modes:**
 - Down
 - Up
 - Toward zero
 - To nearest
- **Example:** round 1.100101 (1.578125) to only 3 fraction bits
 - Down: 1.100
 - Up: 1.101
 - Toward zero: 1.100
 - To nearest: 1.101 (1.625 is closer to 1.578125 than 1.5 is)



Floating Point Arithmetic

In general...

Addition/Subtraction

1. Denormalize operands so that both have the same exponent
2. Perform operation on mantissa with integer arithmetic
3. Normalize result

Multiplication/Division

1. XOR the sign bits of operands
2. Add/subtract unbiased exponents using integer arithmetic
3. Perform operation on mantissa using integer arithmetic
(need to pay attention to location of radix points)

Don't worry about details now. You can always look these up.

Floating-Point Addition

1. Extract exponent and fraction bits
2. Prepend leading 1 to form mantissa
3. Compare exponents
4. Shift smaller mantissa if necessary
5. Add mantissas
6. Normalize mantissa and adjust exponent if necessary
7. Round result
8. Assemble exponent and fraction back into floating-point format

Floating-Point Addition Example

Add the following floating-point numbers:

0x3FC00000

0x40500000

Floating-Point Addition Example

1. Extract exponent and fraction bits

1 bit	8 bits	23 bits
0	01111111	100 0000 0000 0000 0000 0000
Sign	Exponent	Fraction
1 bit	8 bits	23 bits
0	10000000	101 0000 0000 0000 0000 0000
Sign	Exponent	Fraction

For first number (N1): $S = 0, E = 127, F = .1$

For second number (N2): $S = 0, E = 128, F = .101$

2. Prepend leading 1 to form mantissa

N1: 1.1

N2: 1.101

Floating-Point Addition Example

3. Compare exponents

$127 - 128 = -1$, so shift N1 right by 1 bit

4. Shift smaller mantissa if necessary

shift N1's mantissa: $1.1 \gg 1 = 0.11$ ($\times 2^1$)

5. Add mantissas

$$\begin{array}{r}
 0.11 \times 2^1 \\
 + 1.101 \times 2^1 \\
 \hline
 10.011 \times 2^1
 \end{array}$$

Floating Point Addition Example

6. **Normalize mantissa and adjust exponent if necessary**

$$10.011 \times 2^1 = 1.0011 \times 2^2$$

7. **Round result**

No need (fits in 23 bits)

8. **Assemble exponent and fraction back into floating-point format**

$$S = 0, E = 2 + 127 = 129 = 10000001_2, F = 001100..$$

1 bit	8 bits	23 bits
0	10000001	001 1000 0000 0000 0000 0000
Sign	Exponent	Fraction

in hexadecimal: **0x40980000**



Special IEEE754 Numbers

Zero

- Biased Exponent: all 0's
- Mantissa: all 0's
- Sign: 0 for +0; 1 for -0

Infinity

- Biased Exponent: all 1's
- Mantissa: all 1's
- Sign: 0 for +infinity; 1 for -infinity
- E.g. divide by +0 or divide by -0

Not a Number (NaN)

- Biased Exponent: all 1's
- Mantissa: non-zero
- Sign: don't care
- E.g. $\sqrt{-1}$

Subnormal Numbers

- Biased Exponent: all 0's
- Mantissa: Non-zero
- More on this in next

Need to support all of these for strict IEEE compliance

For many applications, strict compliance is not required

Underflow

Smallest positive single-precision **normalized** number:

0	00000001	00000000000000000000000000000000
---	----------	----------------------------------

$$1.00000000000000000000000000000000_2 \times 2^{-126} \approx 1.2_{10} \times 10^{-38}$$

UNDERFLOW occurs when an arithmetic operation leads to a result smaller than this value

Underflow

NORMAL NUMBERS

Numbers that can be represented by the normalized format

UNDERFLOW GAP

The range of 0 to the smallest normal number

SUBNORMAL NUMBERS

Numbers in the underflow gap

**IEEE754 has a mechanism to represent
a subset of the Subnormal Numbers**

Subnormal Numbers in IEEE754

To encode a subnormal number in IEEE754,

1. Biased Exponent has all 0's
→ represents smallest possible exponent
Single-Precision: -126
Double-Precision: -1022
2. Mantissa is interpreted as being preceded by 0. instead of 1.

Example:

0	00000000	001011000000000000000000
---	----------	--------------------------

$$0.001011_2 \times 2^{-126}$$

**Trade-off precision (number of significant digits)
for range**

IEEE754 Subnormal Range

Smallest positive single-precision subnormal number:

[illegible]

[illegible]

More about Floating Point Arithmetic

There is a lot more to be discussed on Floating-Point (rounding, accuracy, etc.) beyond scope of this course.

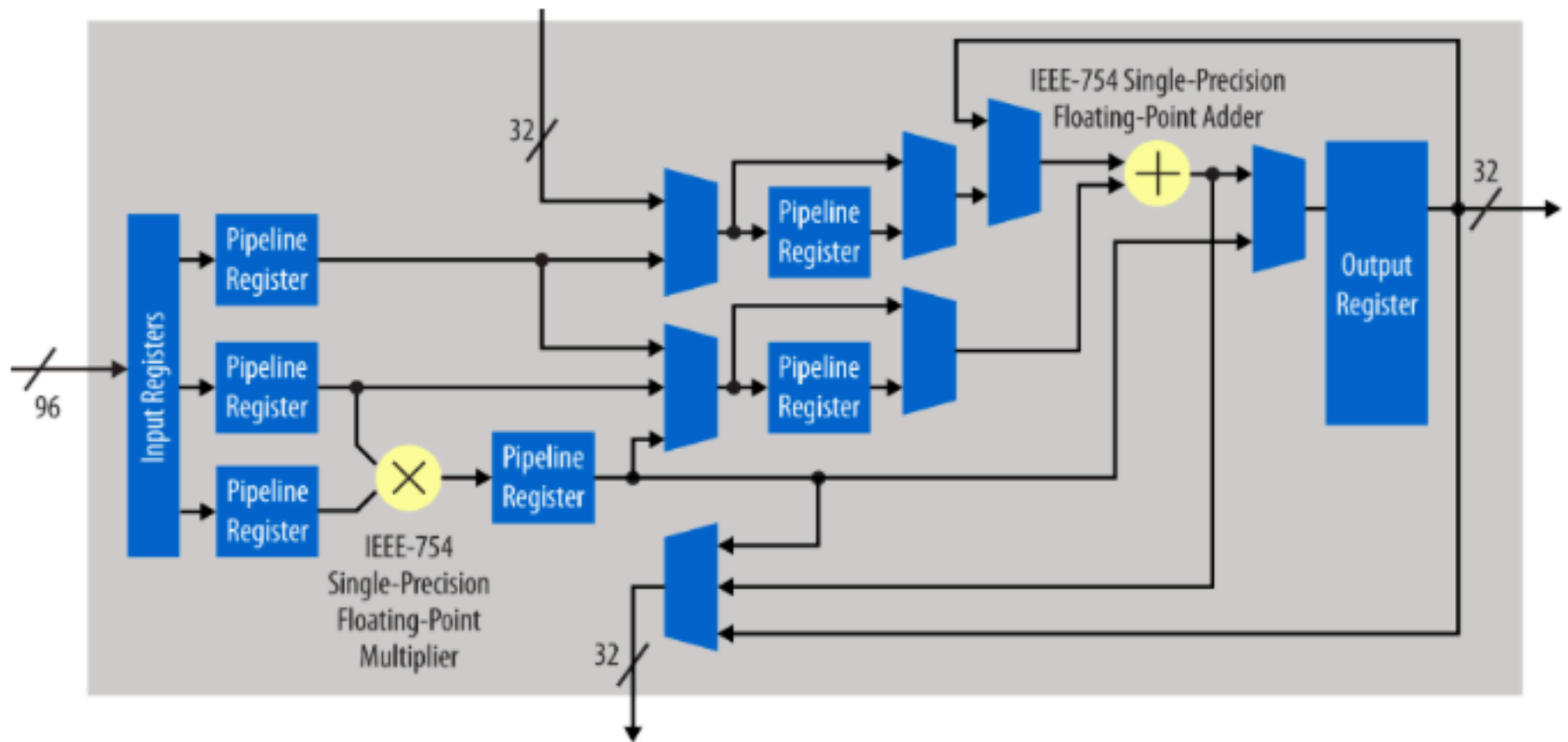
These details can be really important depending on your application

Notable Case:

1991 Dhahran, Saudi Arabia – 28 US Soldiers killed because missile interception system internal clock had drifted by 0.33s due to floating-point rounding error accumulated over several days.

0.33s translated into incorrect calculation on incoming missile location by about 600m. After correct initial detection, system looked at wrong part of the sky and found no missile. Thus it did not proceed with interception attempt.

Modern FPGAs



Fixed Point vs. Floating Point

Fixed Point

- Simpler circuitry (need fewer logic/routing resources) for arithmetic operations

Floating Point

- Higher dynamic range of representable values

Synthesizing Floating Point Datapaths

```
signal x : real;  
...  
x <= 0.13;
```

Is this Synthesizable?

- ✘ 10414 VHDL Unsupported Feature error at div.vhd(18): cannot synthesize non-constant real objects or values
 - ✘ 12153 Can't elaborate top-level user hierarchy
 - ✘ Quartus II 32-bit Analysis & Synthesis was unsuccessful. 2 errors, 2 warnings
-

No but can be very useful for simulation.

There are libraries available for working with floating-point (IEEE754) format. But if you need to synthesize them, you still either need to write the logic manually, or use a third-party components.