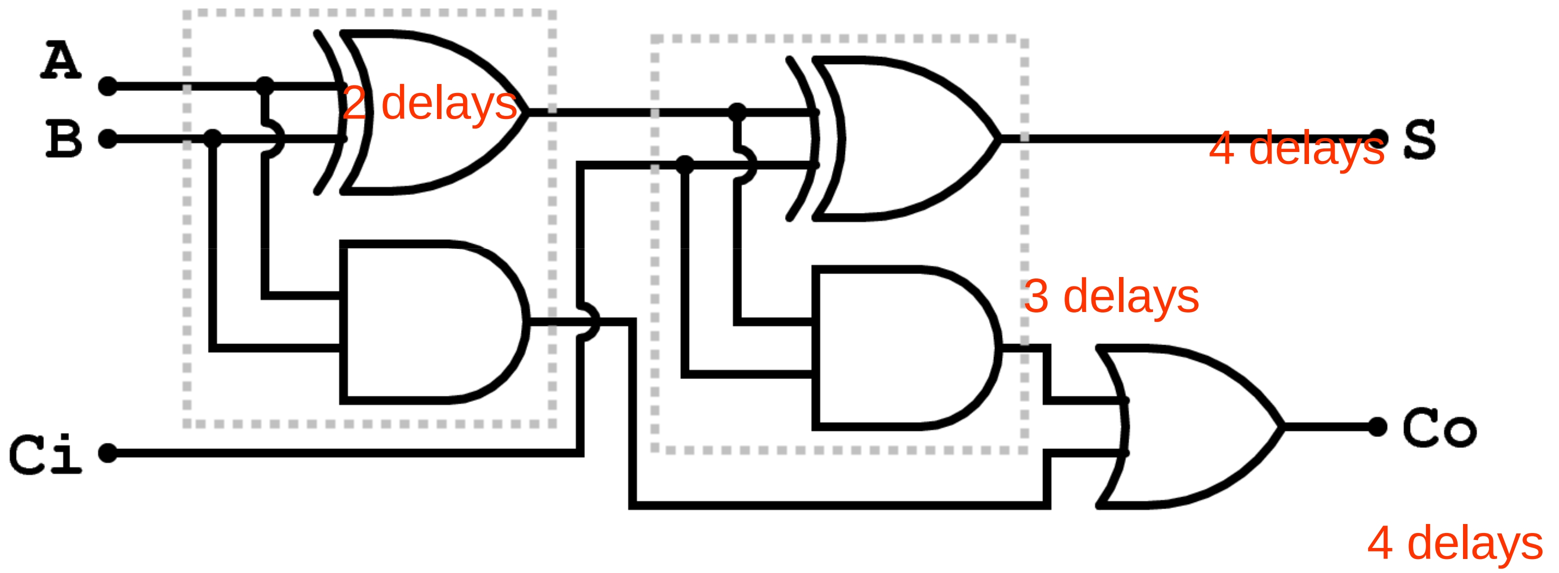
SEE 3243
Digital System

Lecturers :
Muhammad Mun'im Ahmad Zabidi
Muhammad Nadzir Marsono
Kamal Khalil

Week 6: Arithmetic Circuits II — CLA, Comparators, ALU, Multiplier

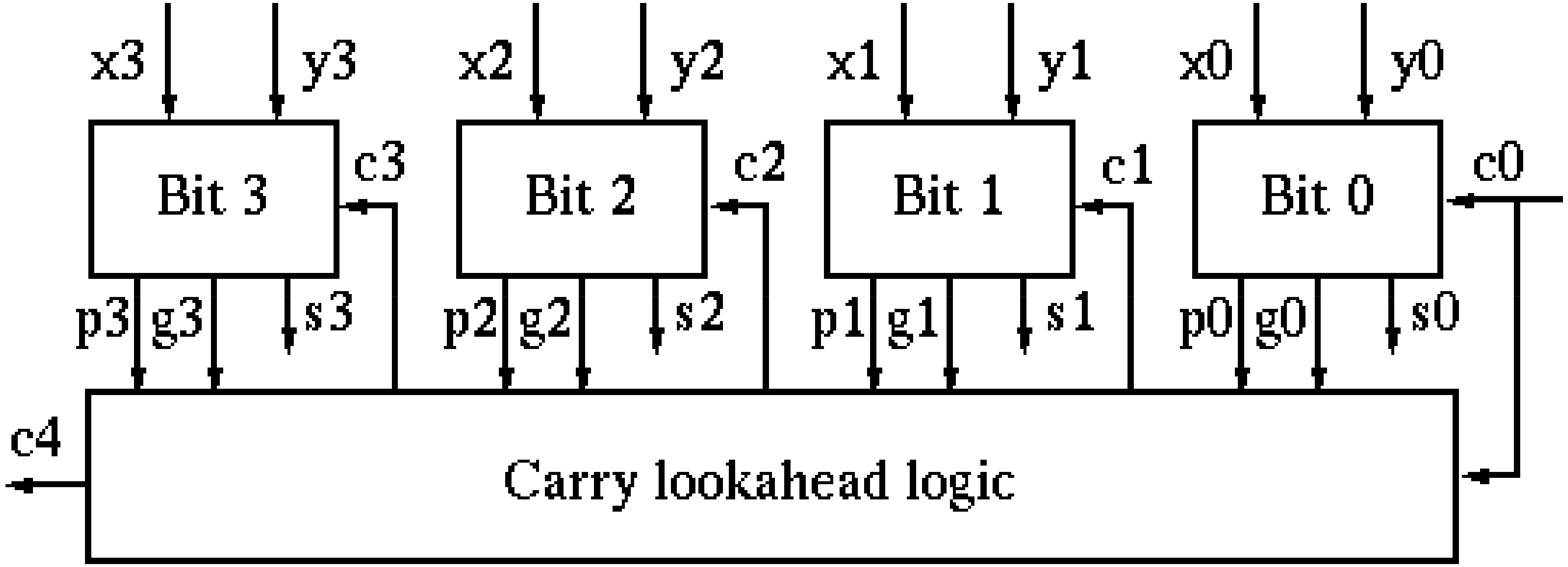
Full Adder Delay Analysis



From A, B to Co for one stage is 4 delays

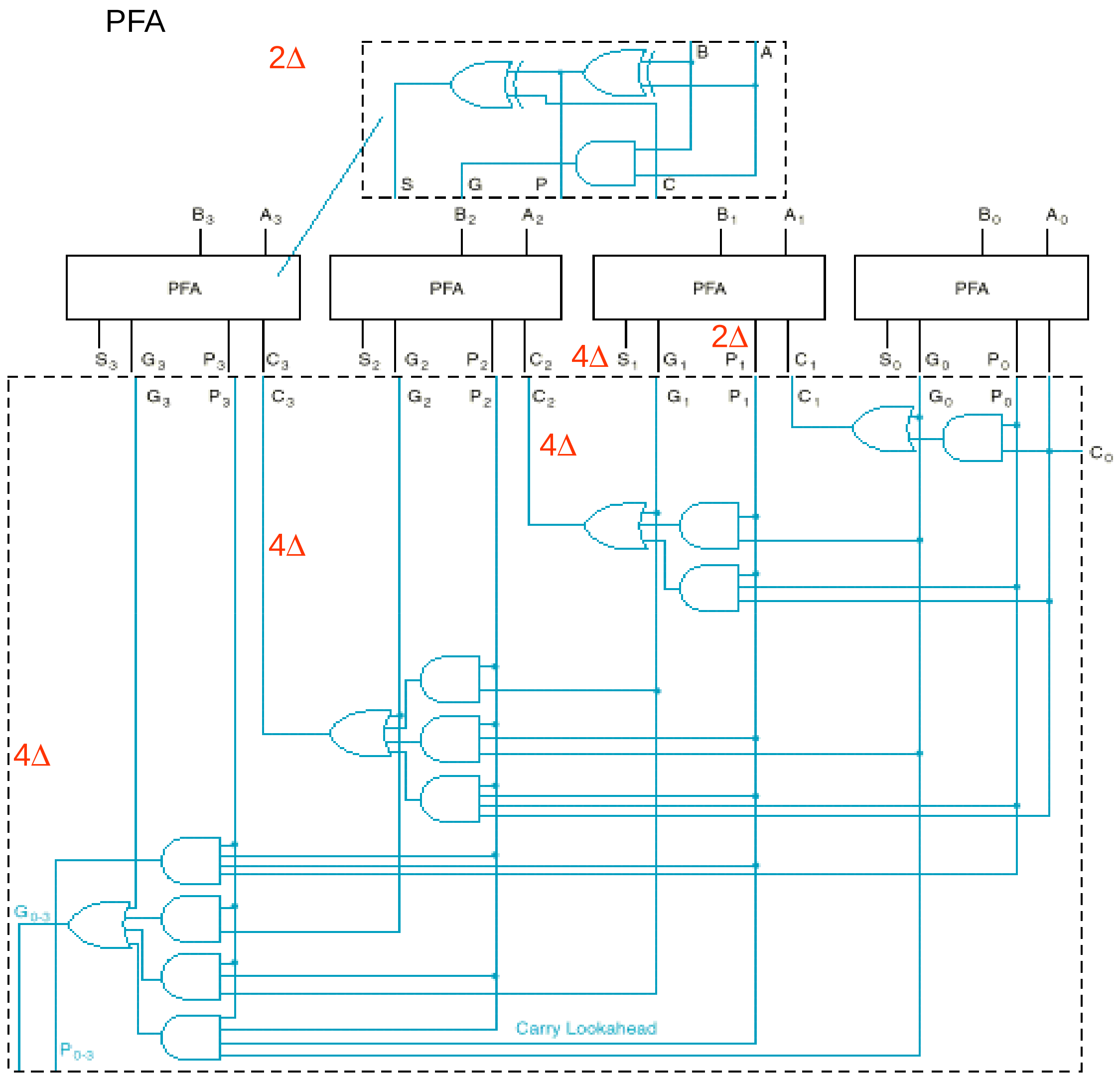
From Ci to Co is 2 delays for each subsequent stage or 2n + 2 for n stages

Ripple Carry Adder Analysis



- Total delay for final sum & carry is 2n+2 gate delays (n = # of stages)
- Assumes XOR is 2 delays
- Delay from C_i to C_{i+1} is 2 gate delays (except stage 0, where delay is 4 units)

Carry Lookahead Adder Analysis



6Δ

6Δ 5Δ

If we reduce the time to compute C_3, we can reduce delay to get S_3 (final sum) to 6 gate delays

Carry Lookahead Logic Derivation

Carry Generate $G_i = A_i B_i$ *must generate carry when A = B = 1*

Carry Propagate $P_i = A_i \text{ xor } B_i$

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

$$S_i = A_i \text{ xor } B_i \text{ xor } C_i = P_i \text{ xor } C_i$$

$$C_{i+1} = A_i B_i + Ai\ C_i + Bi\ C_i$$

$$= A_i B_i + (A_i + B_i) C_i$$

$$= A_i B_i + (A_i \text{ xor } B_i) C_i$$

$$= G_i + P_i C_i$$

Carry Lookahead Logic

- Reexpress the carry logic as follows:

$C_1 = G_0 + P_0\, C_0$

$C_2 = G_1 + P_1\, C_1$

$\quad = G_1 + P_1\, G_0 + P_1\, P_0\, C_0$

$C_3 = G_2 + P_2\, C_2$

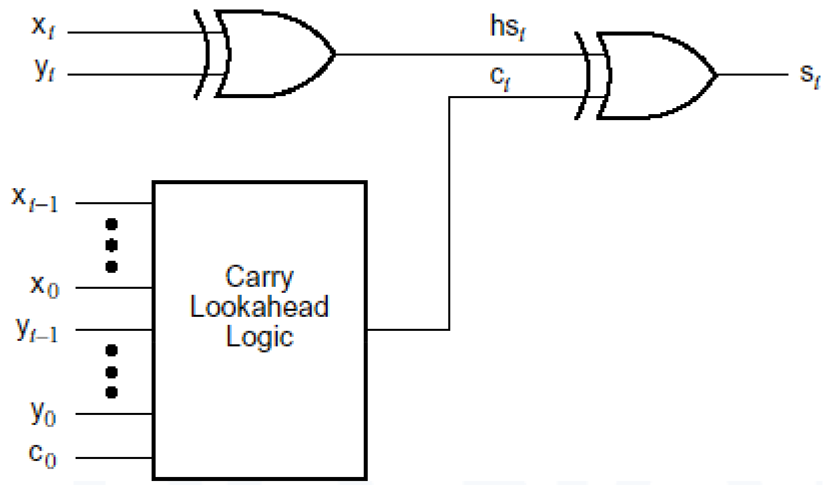
$\quad = G_2 + P_2\, G_1 + P_2\, P_1\, G_0 + P_2\, P_1\, P_0\, C_0$

$C4 = G_3 + P_3\, C_3$

$\quad = G_3 + P_3\, G_2 + P_3\, P_2\, G_1 + P_3\, P_2\, P_1\, G_0 + P_3\, P_2\, P_1\, P_0\, C_0$

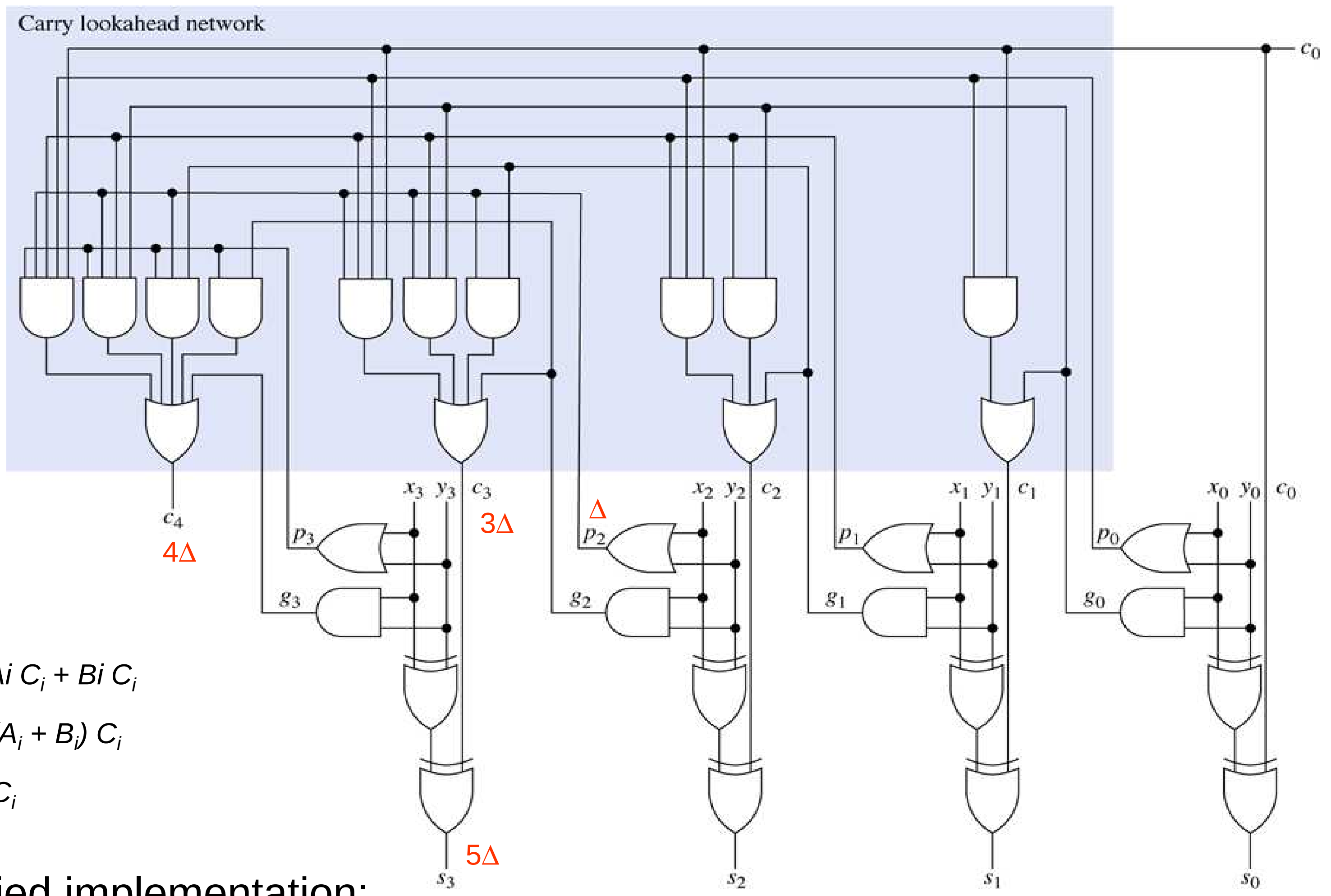
- Variables are the adder inputs and C_0 (carry in to stage 0)!

Structure of One Stage in CLA



- To compute S_i, only $x_{i-1} \ldots x_0$, $y_{i-1} \ldots y_0$ and c_0 are needed.
- No need to wait for c_{i-1}

Alternative CLA Design



$C_{i+1} = A_i\, B_i + Ai\, C_i + Bi\, C_i$

$= A_i\, B_i + (A_i + B_i)\, C_i$

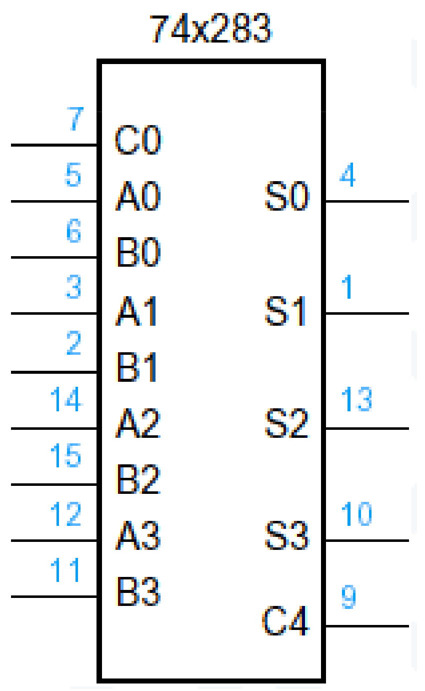
$= G_i + P_i\, C_i$

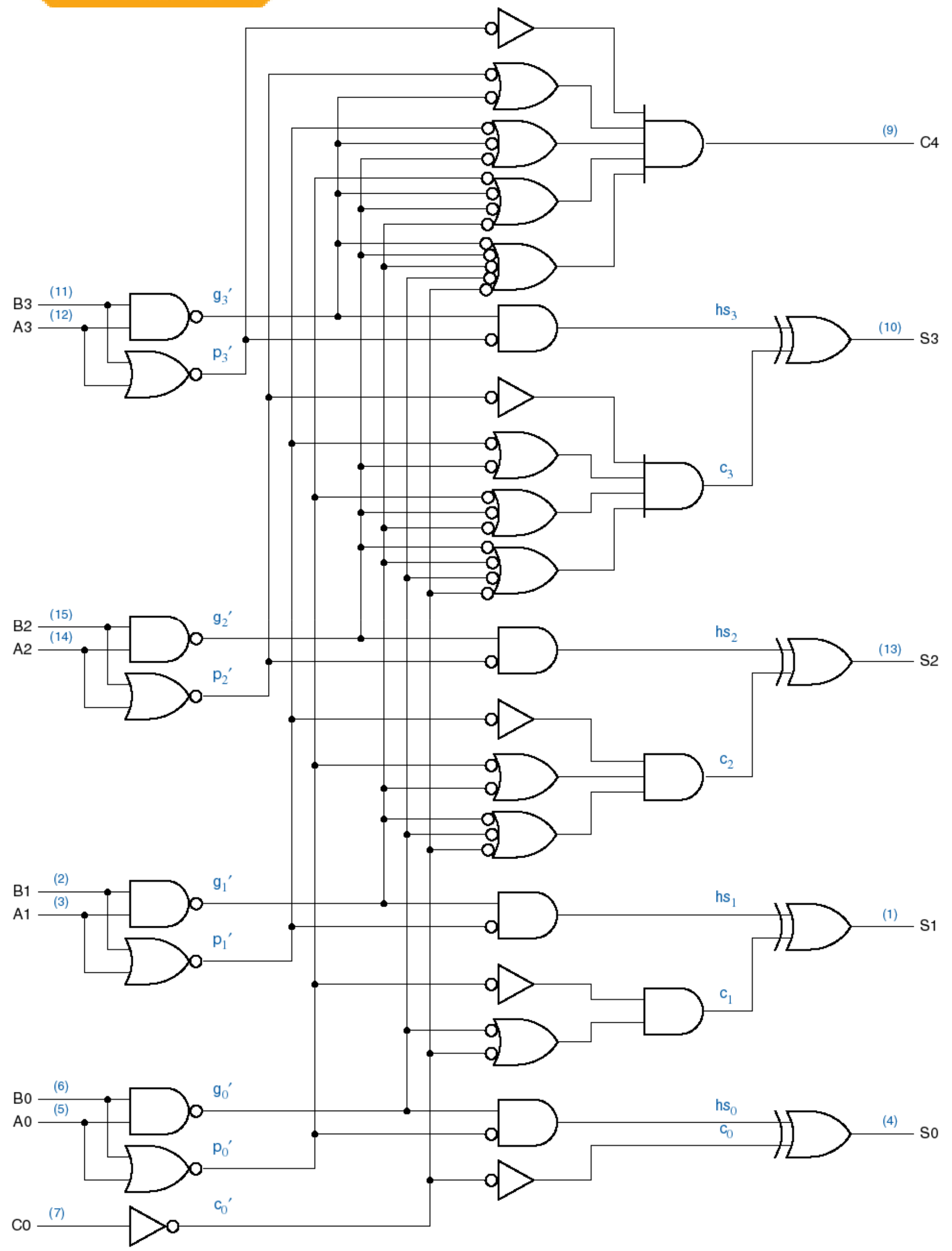
A modified implementation:

P_i computed using OR gates (slightly faster)

74x283 4-bit adder

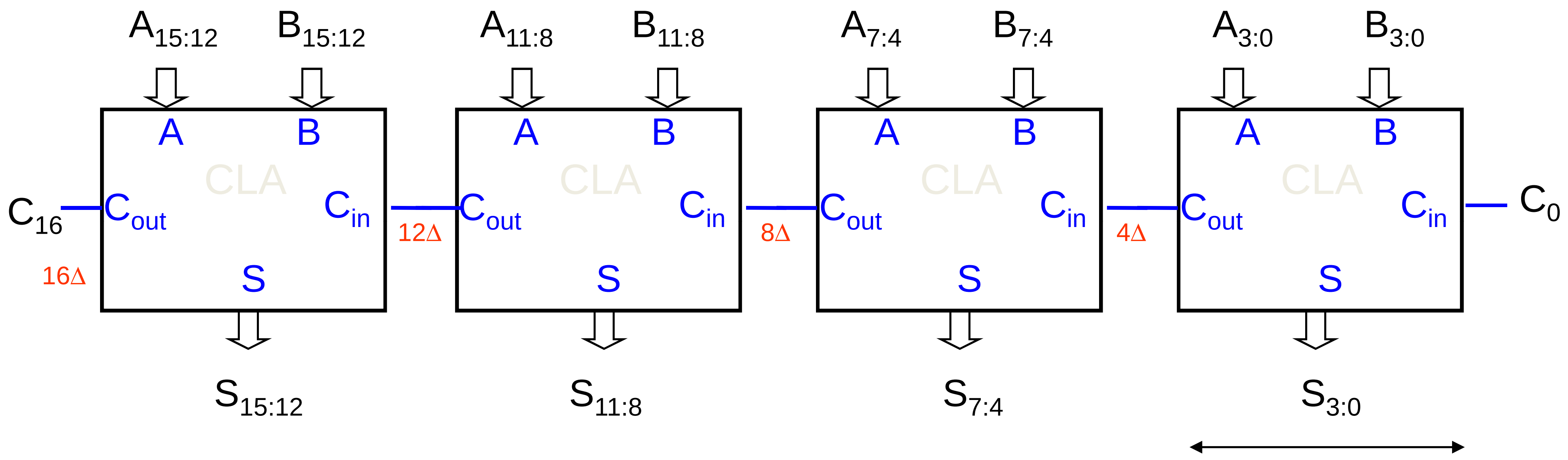
- Uses carry lookahead internally





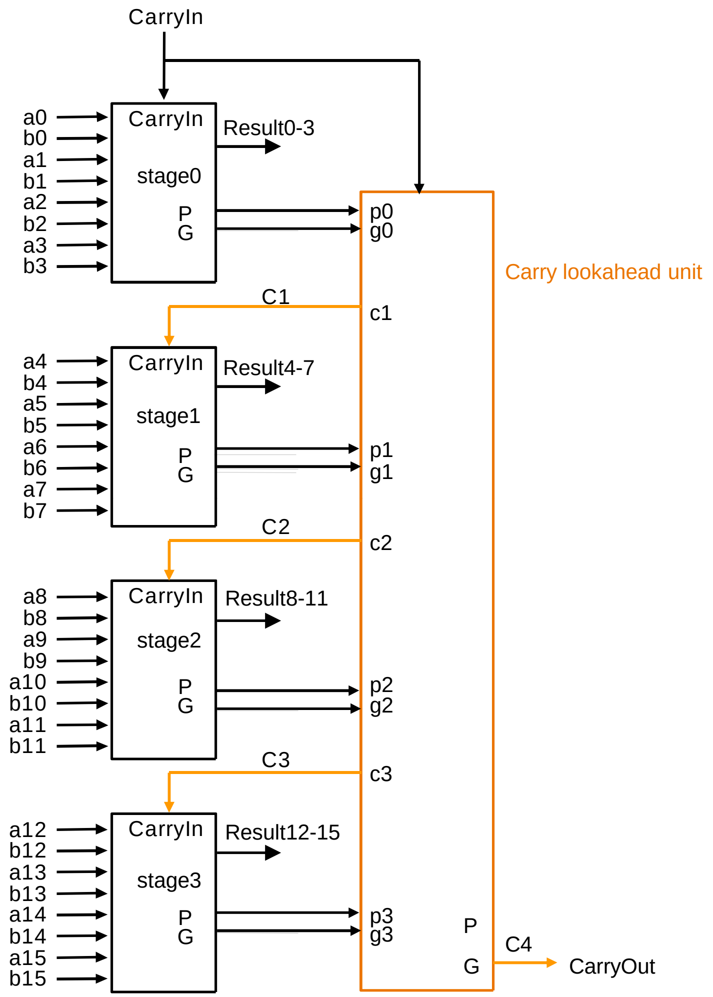
Cascading CLA

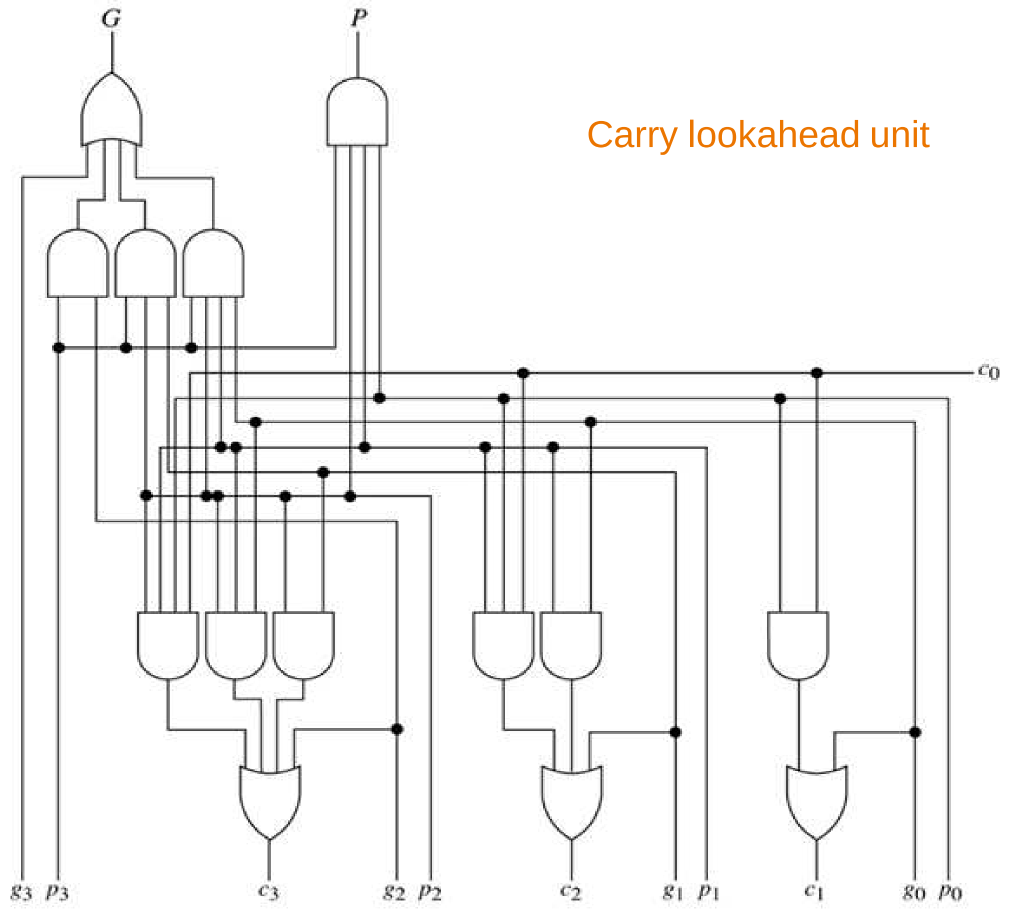
- Similar to ripple adder, but different latency



Delay of each stage
is 4 gate levels instead
of 10 for ripple adders

Hierarchical Carry Lookahead



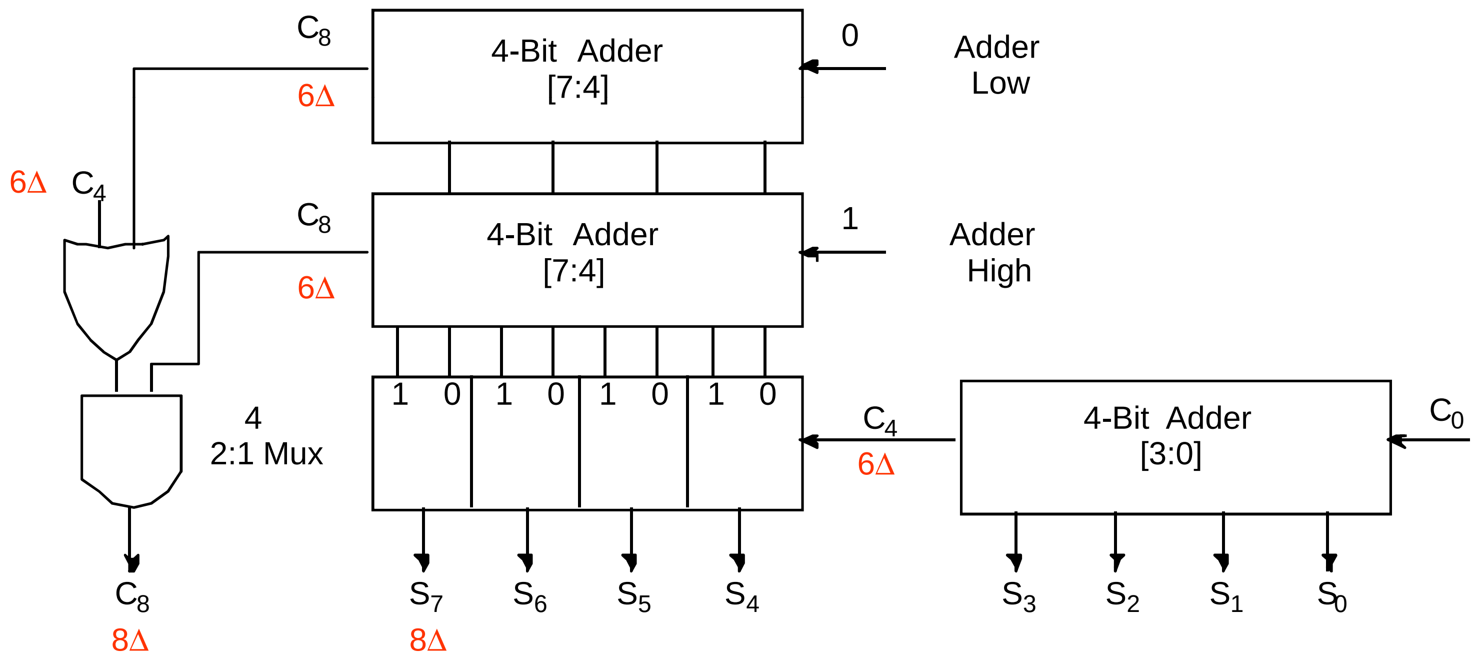


(a)

- Second level carry lookahead unit – extends lookahead to 16 bits
- If extended to 64 bits – reduces gate delay from 130 to 14, or improved by a factor of 9

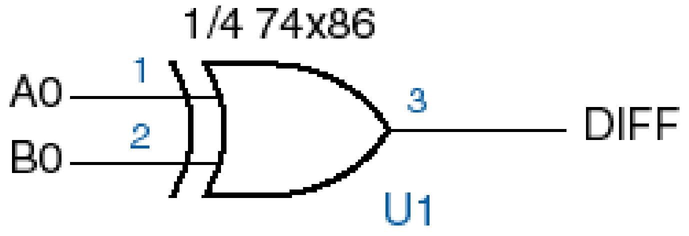
Carry Select Adder

- 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

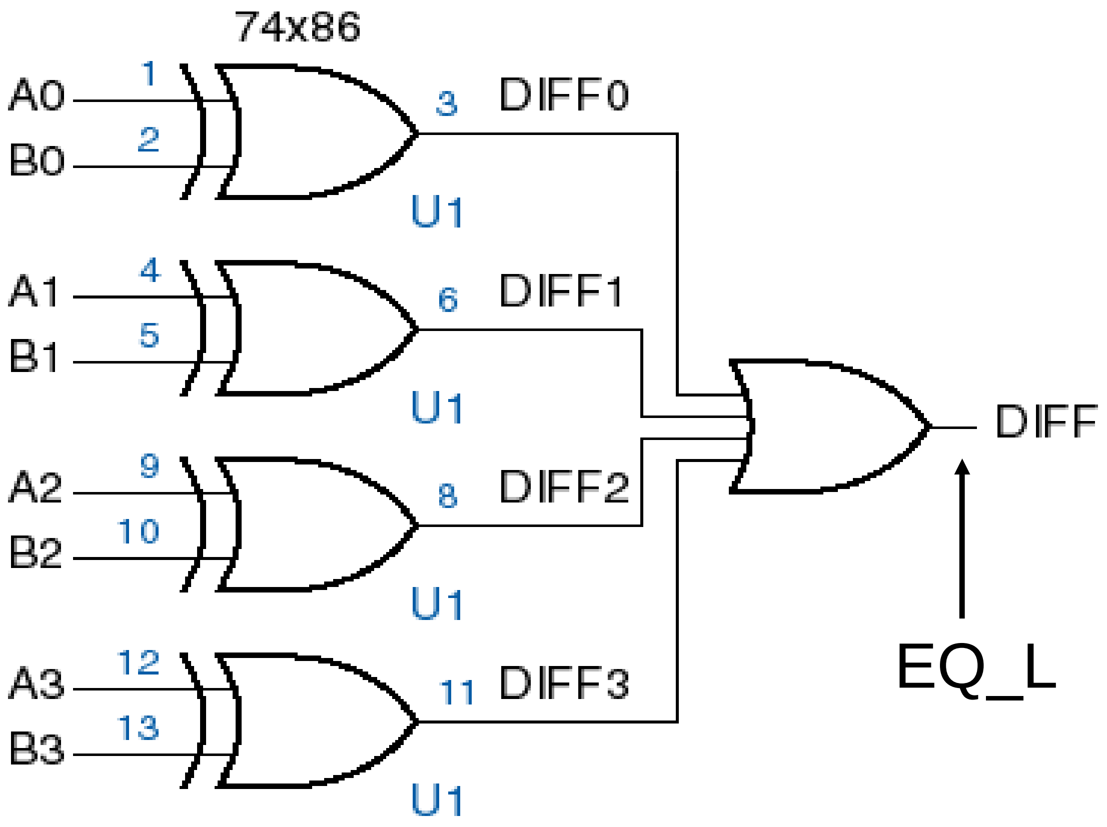


Equality Comparators

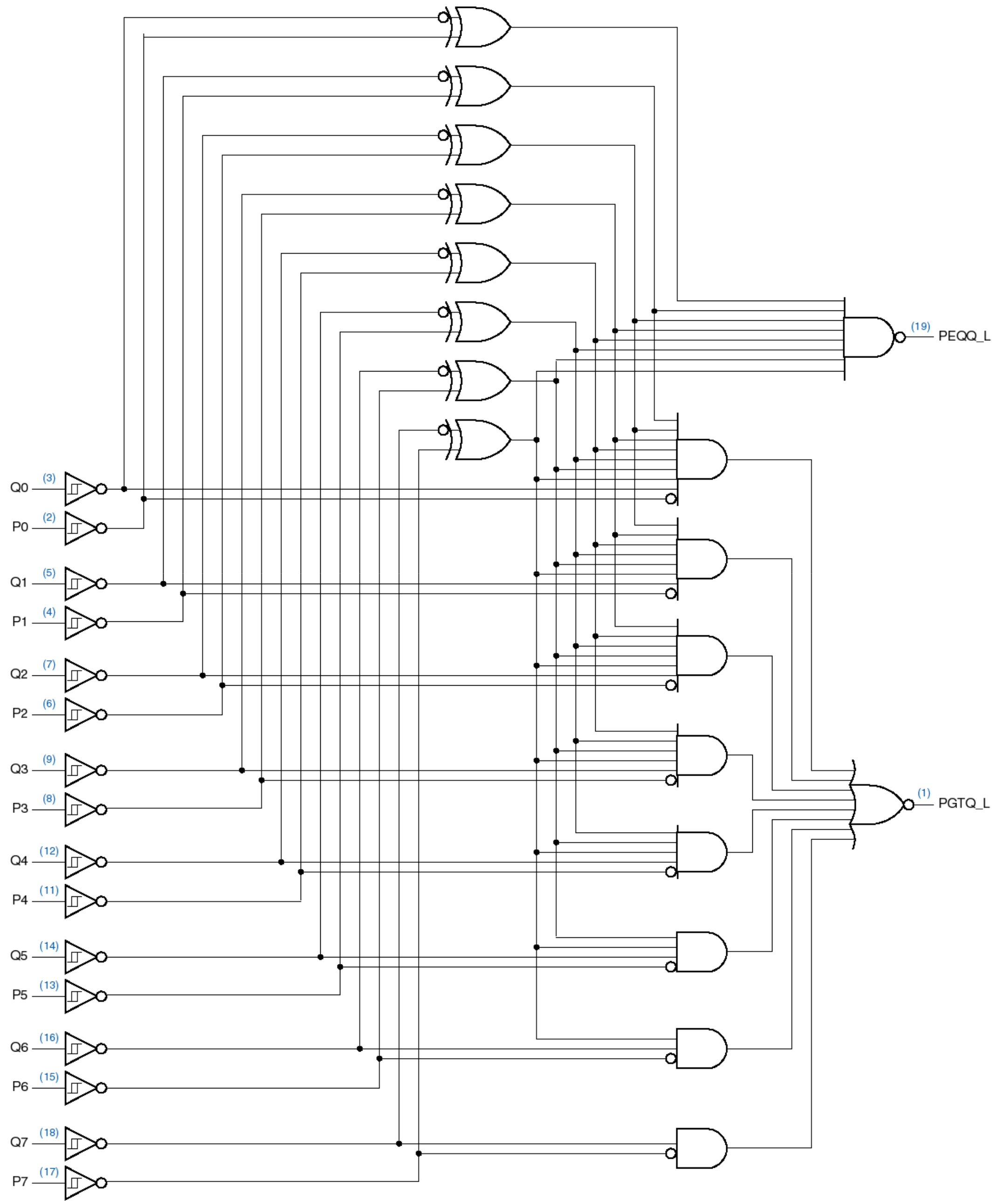
1-bit comparator

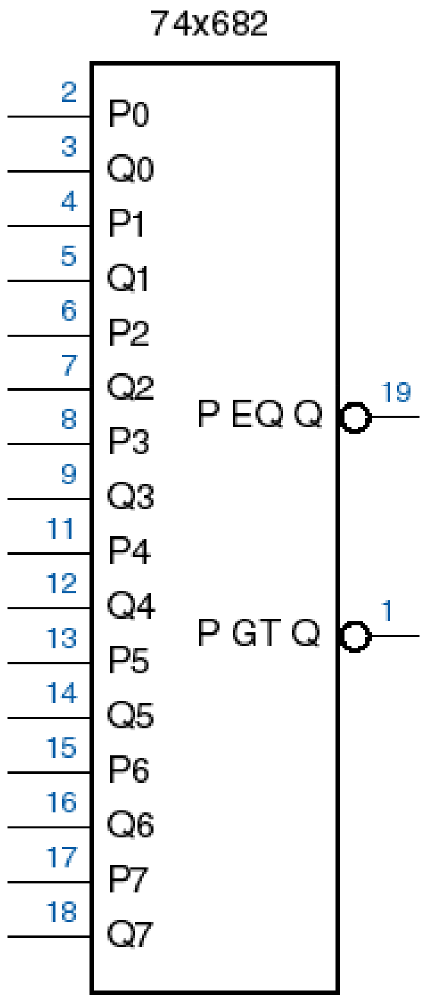


4-bit comparator

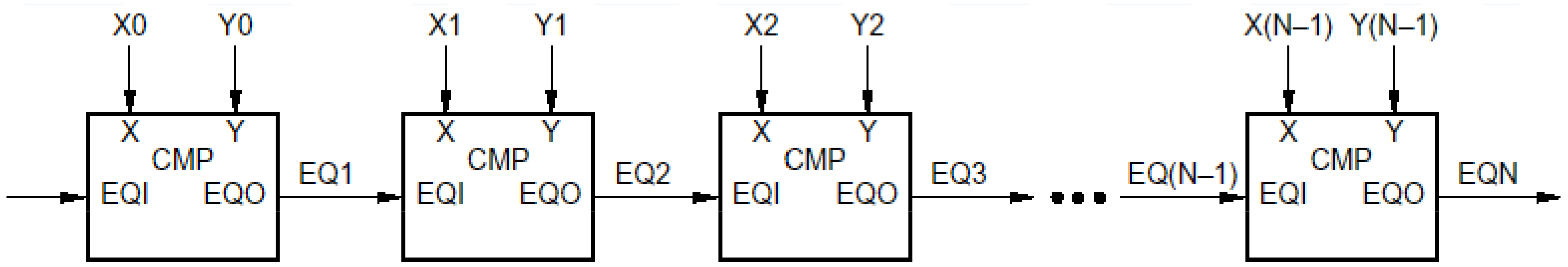


8-bit Magnitude Comparator



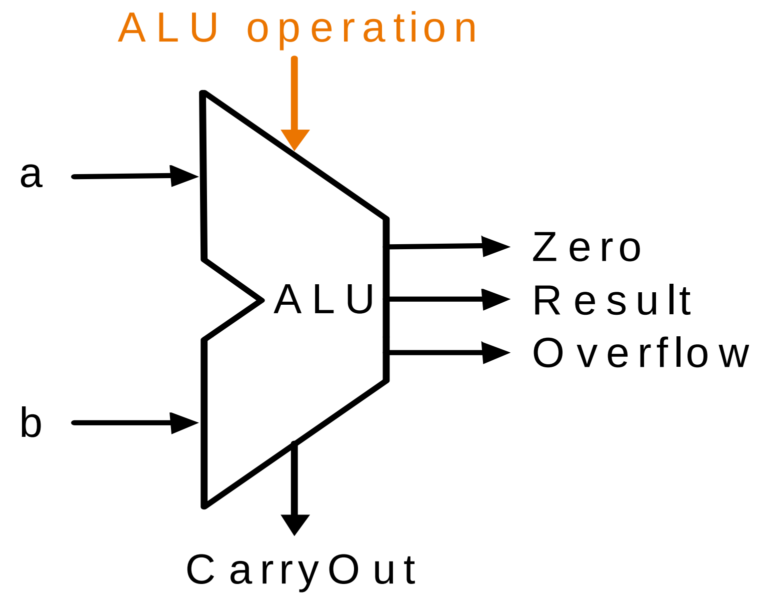


Iterative Comparator



Arithmetic Logic Unit

- Basic building block of every CPU.
- Combinational circuit.
- Does integer addition, subtraction.
- Also does all 16 bitwise logical operations.
- Does not do multiply, divide. They would be implemented either by a separate unit, or subroutines (slow but cheap).
- Floating operations are also one or more separate units. (More: faster, costlier.)
- Why combine arithmetic & logic? They share a lot of circuitry.

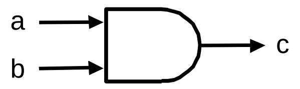


Common symbol for ALU

Sample ALU 1: Mux Approach

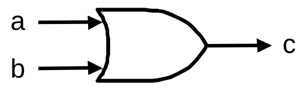
Start with Simple Logical Operations

1. AND gate (c = a . b)



a	b	c = a . b
0	0	0
0	1	0
1	0	0
1	1	1

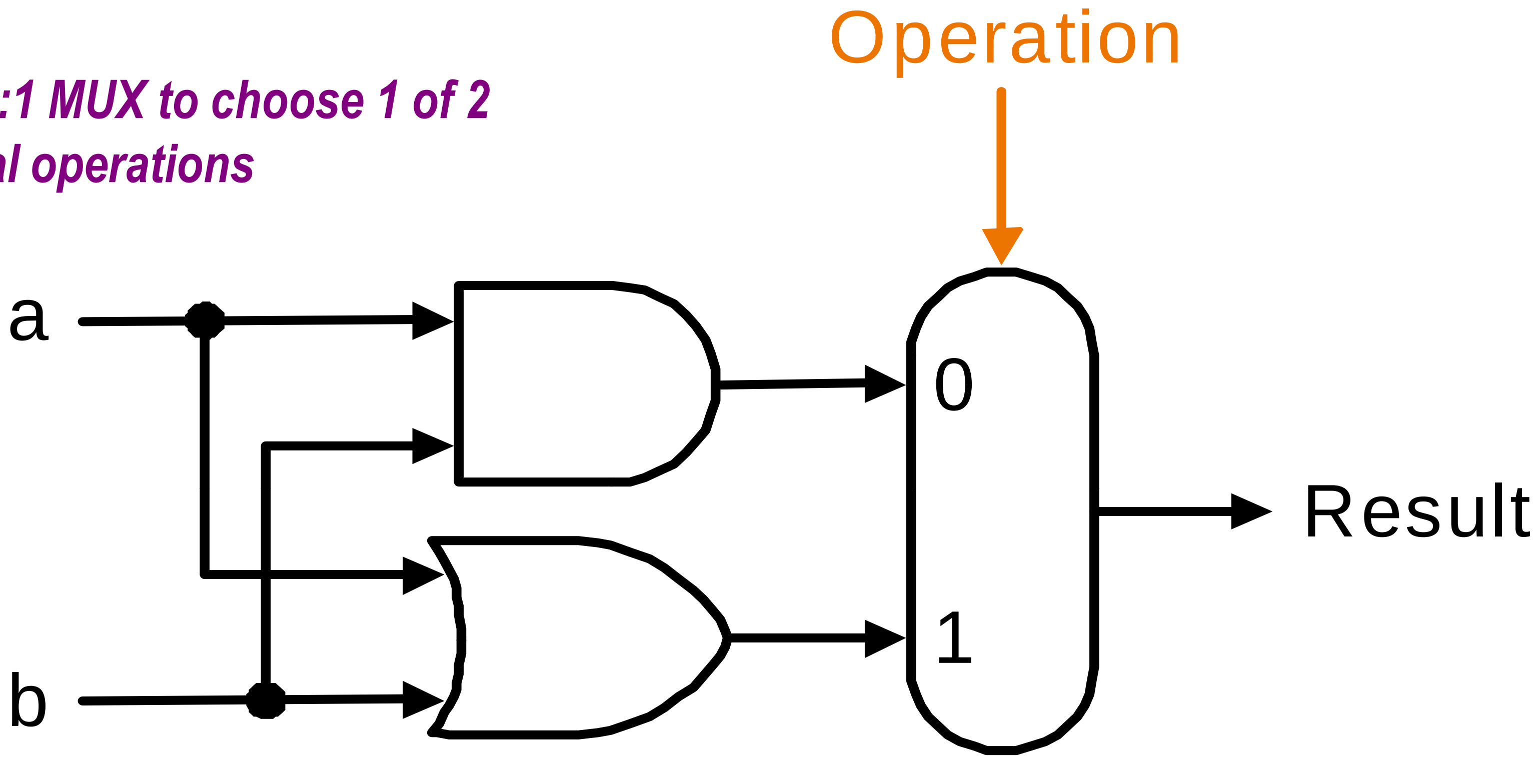
2. OR gate (c = a + b)



a	b	c = a + b
0	0	0
0	1	1
1	0	1
1	1	1

Sample ALU 1

Use 2:1 MUX to choose 1 of 2 logical operations



If Operation is 0, then Result = a AND b

If Operation is 1, then Result = a OR b

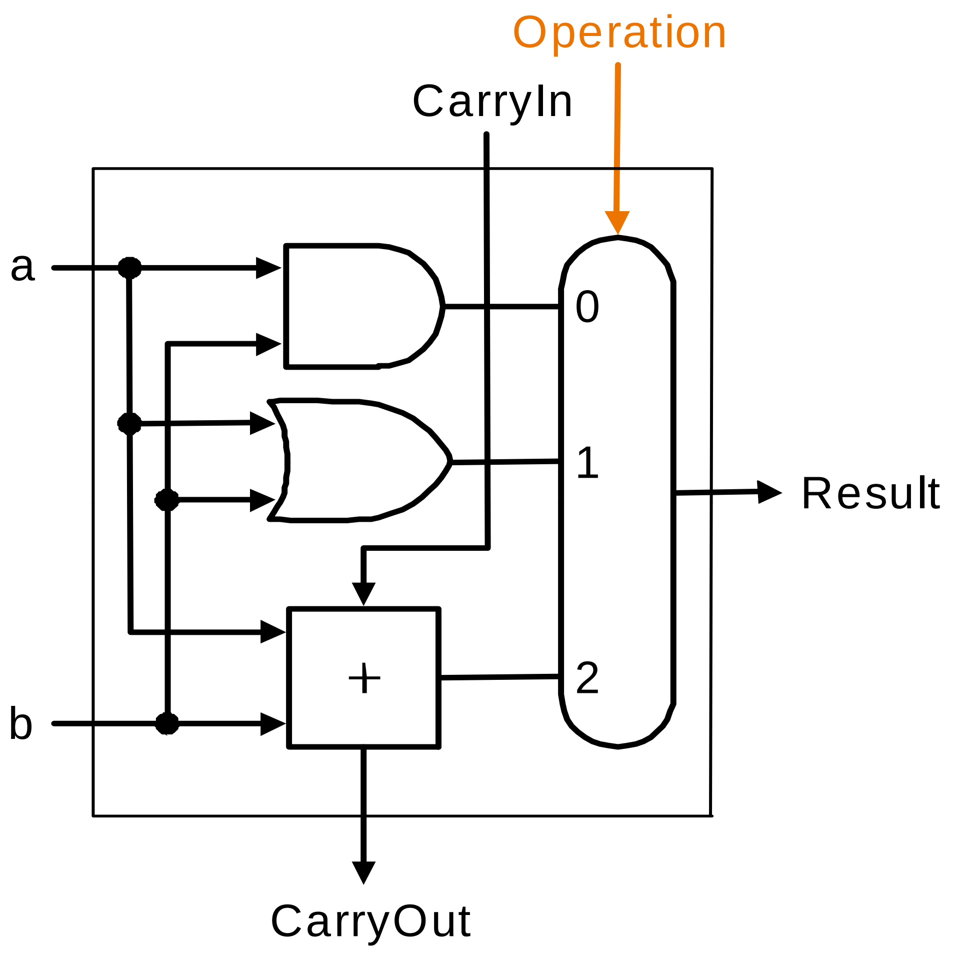
Sample ALU 1

Now add Full Adder for arithmetic

If Op is 0, then Result = a AND b

If Op is 1, then Result = a OR b

If Op is 2, then Result = sum of (a + b + CarryIn)



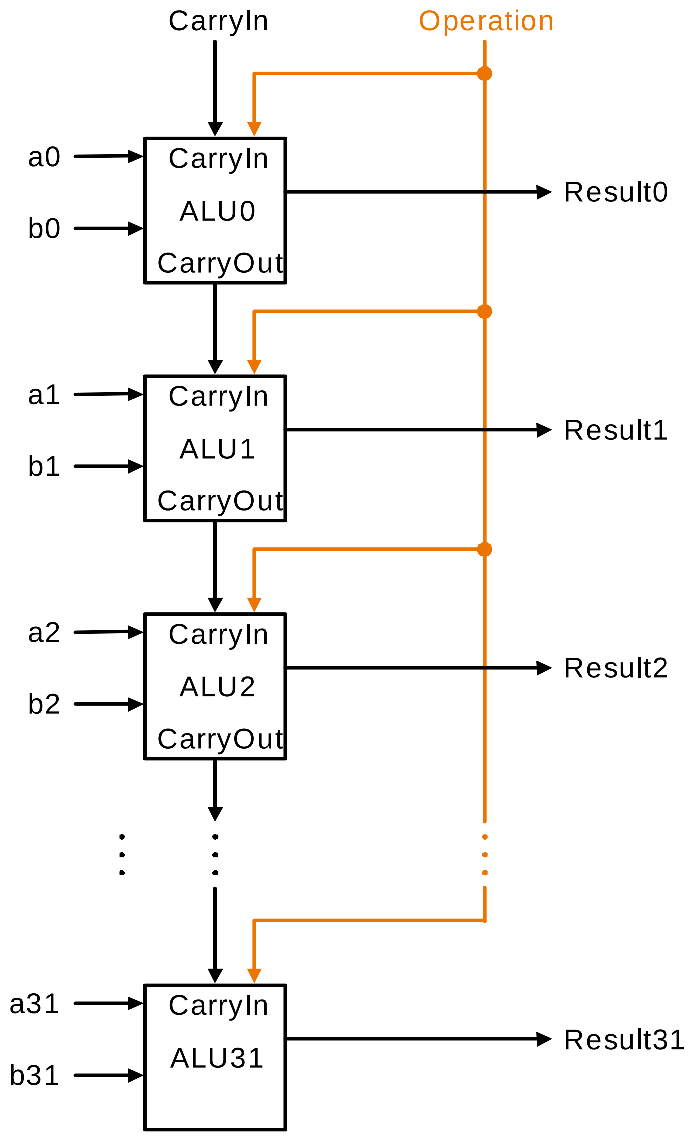
Sample ALU 1

Repeat the 1-bit ALU 32 times

If Op is 0, then Result$_i$ = a$_i$ AND b$_i$

If Op is 1, then Result$_i$ = a$_i$ OR b$_i$

If Op is 2, then Result$_i$ = sum of (a$_i$ + b$_i$)



ALU 1 with Subtraction Ability

If Op is 0, then Result = a AND b

If Op is 1, then Result = a OR b

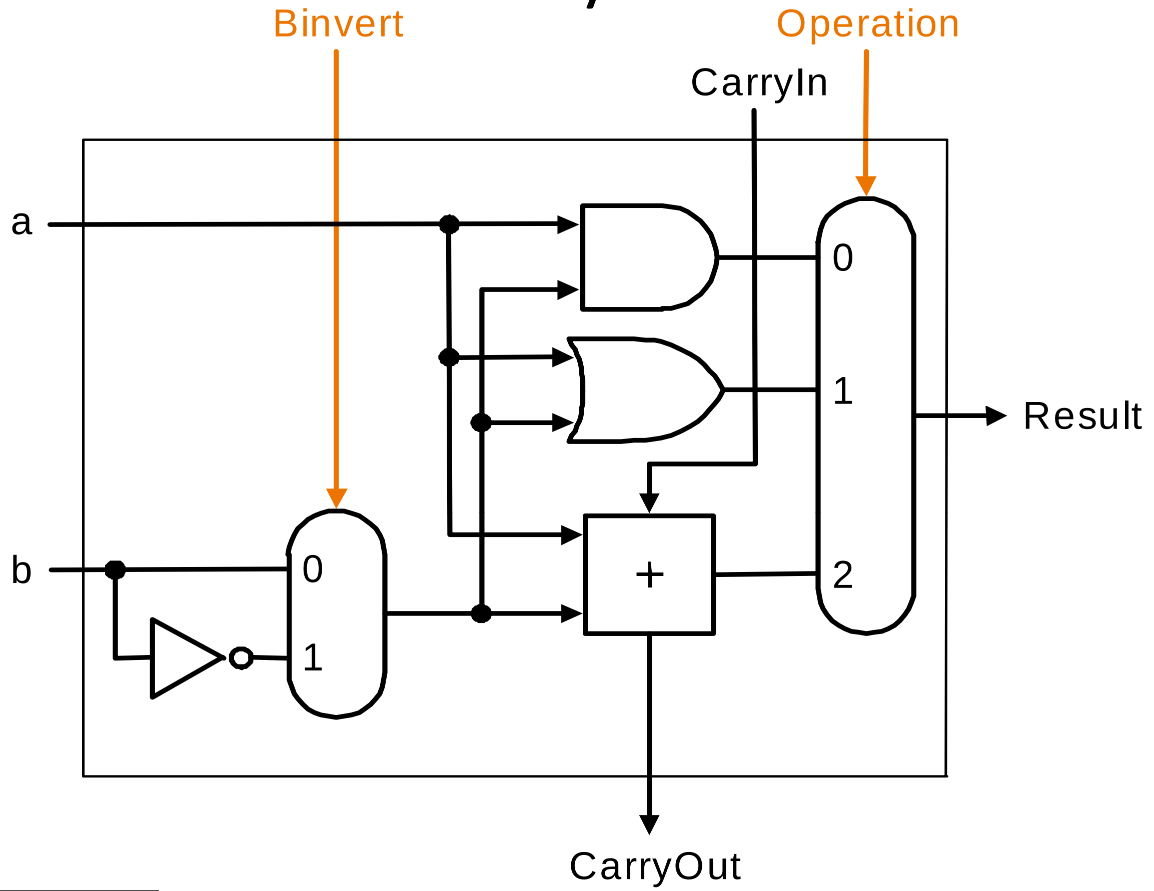
If Op is 2,

and if Binvert is 0,

then Result = sum (a + b)

if Binvert is 1,

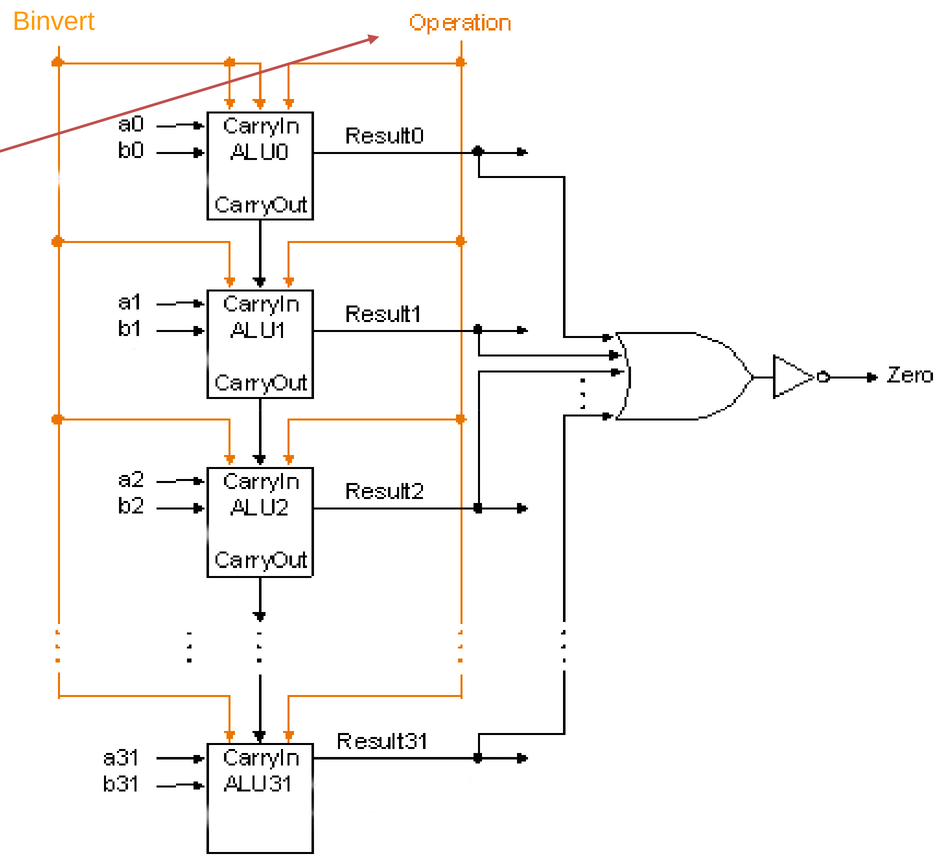
then Result = sum (a + (-b))



Note that (- b) is 1’s comp

Add a 1 into Carryin$_0$ to get 2’s comp

ALU 1 with Zero Detection



Control Lines	Function
000	and
001	or
010	add
110	sub

Sample ALU 2: Truth Table Approach

We want to design an ALU which can do the following operations:

m_1	m_0	**Operation**
0	0	A plus B
0	1	A minus B
1	0	A plus 1
1	1	A nor B

Assume inputs A and B are 4-bit 2’s complement numbers, and F is output.

One way of obtaining the circuit is by creating the truth table:

m1	m0	a3	a2	a1	a0	b3	b2	b1	b0	f3	f2	f1	f0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	1	0	0	0	1	A plus B
0	0	0	0	0	0	0	0	1	0	0	0	1	0	
.														
.														
.														
1	1	1	1	1	1	1	1	1	0	0	0	0	0	
1	1	1	1	1	1	1	1	1	1	0	0	0	0	A nor B

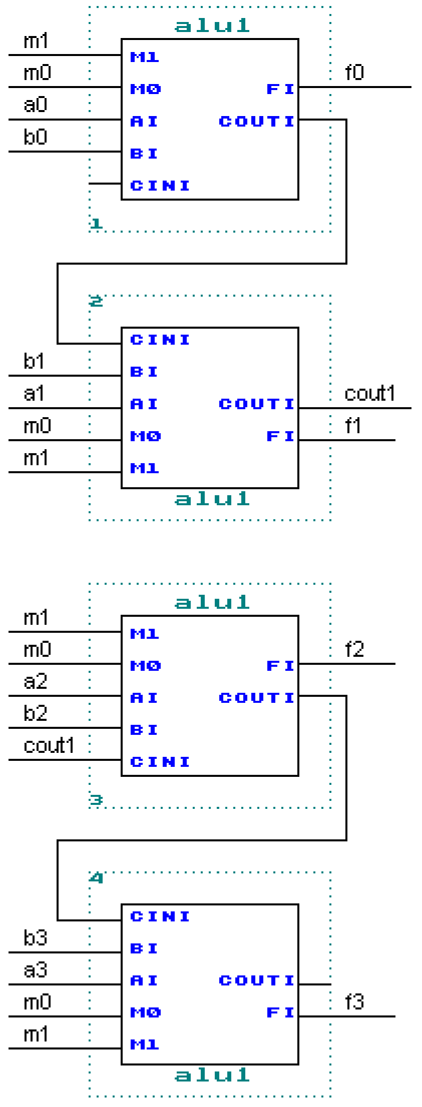
A huge truth table. Imagine truth table for 8-bit inputs!

Sample ALU 2

- Design a universal logic block (called a bit slice) that accepts only 1-bit of the inputs (per logic block).

- We then copy and connect this bit slice as many times as there are input bits.

m_1	m_0	Operation
0	0	A plus B
0	1	A minus B
1	0	A plus 1
1	1	A nor B



Sample ALU 2

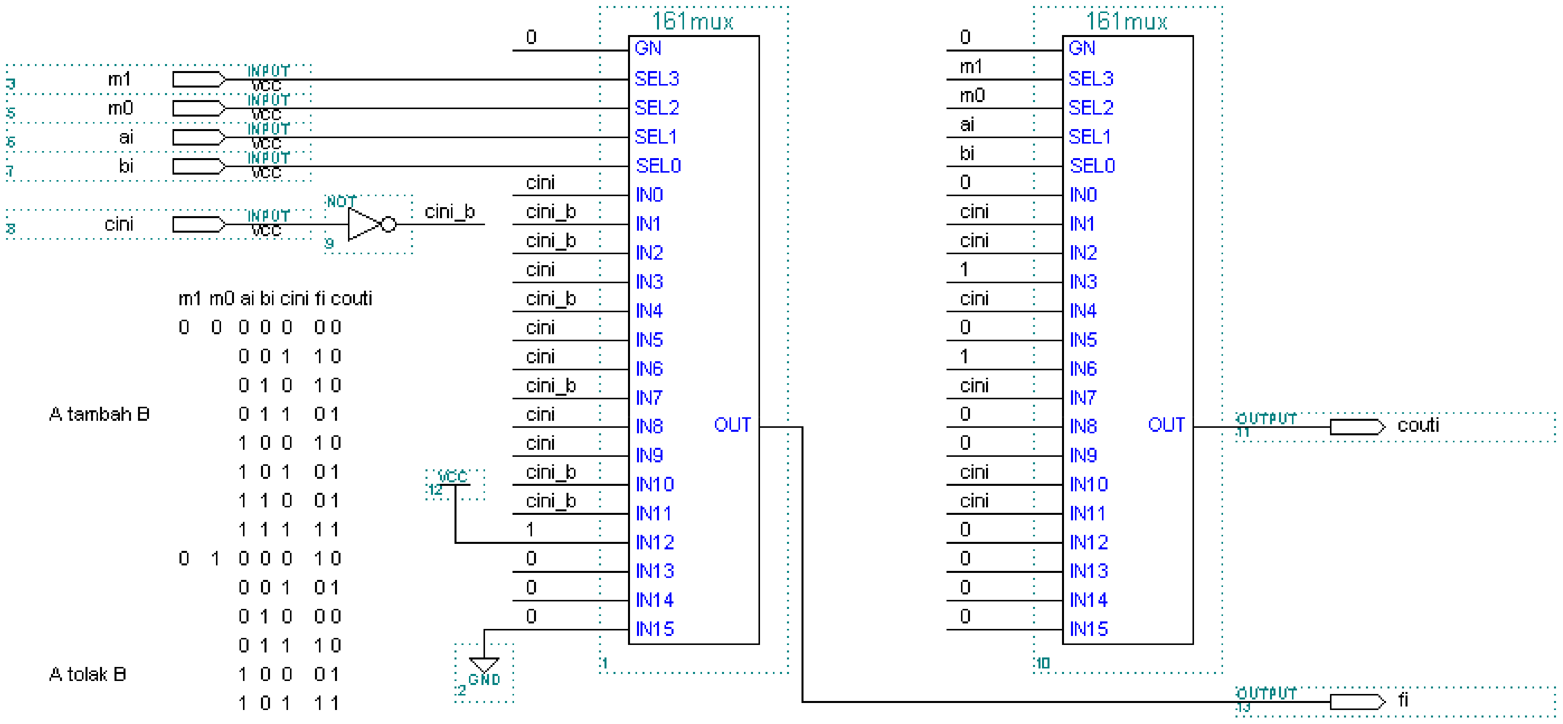
- Each bit slice has 5 inputs and 2 outputs. Truth table is on the right.
- Remember, the bit slice circuit is universal, i.e. exactly same circuit for all input bits.
- For A plus 1 operation for example, we don't need B input. But remember, it must be universal. Other operations require B input.
- Another example: NOR operation doesn't require cini input, but the truth table for NOR operation must have cini input.

	m1	m0	ai	bi	cini	fi	couti
	0	0	0	0	0	0	0
			0	0	1	1	0
			0	1	0	1	0
A tambah B			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
	0	1	0	0	0	1	0
			0	0	1	0	1
			0	1	0	0	0
			0	1	1	1	0
A tolak B			1	0	0	0	1
			1	0	1	1	1
			1	1	0	1	0
			1	1	1	0	1
	1	0	0	0	0	0	0
			0	0	1	1	0
			0	1	0	0	0
A tambah 1			0	1	1	1	0
			1	0	0	1	0
			1	0	1	0	1
			1	1	0	1	0
			1	1	1	0	1
	1	1	0	0	X	1	X
A NOR B			0	1	X	0	X
			1	X	X	0	X

m_1	m_0	Operation
0	0	A plus B
0	1	A minus B
1	0	A plus 1
1	1	A nor B

Sample ALU 2

Bit Slice Circuit for Sample ALU 2

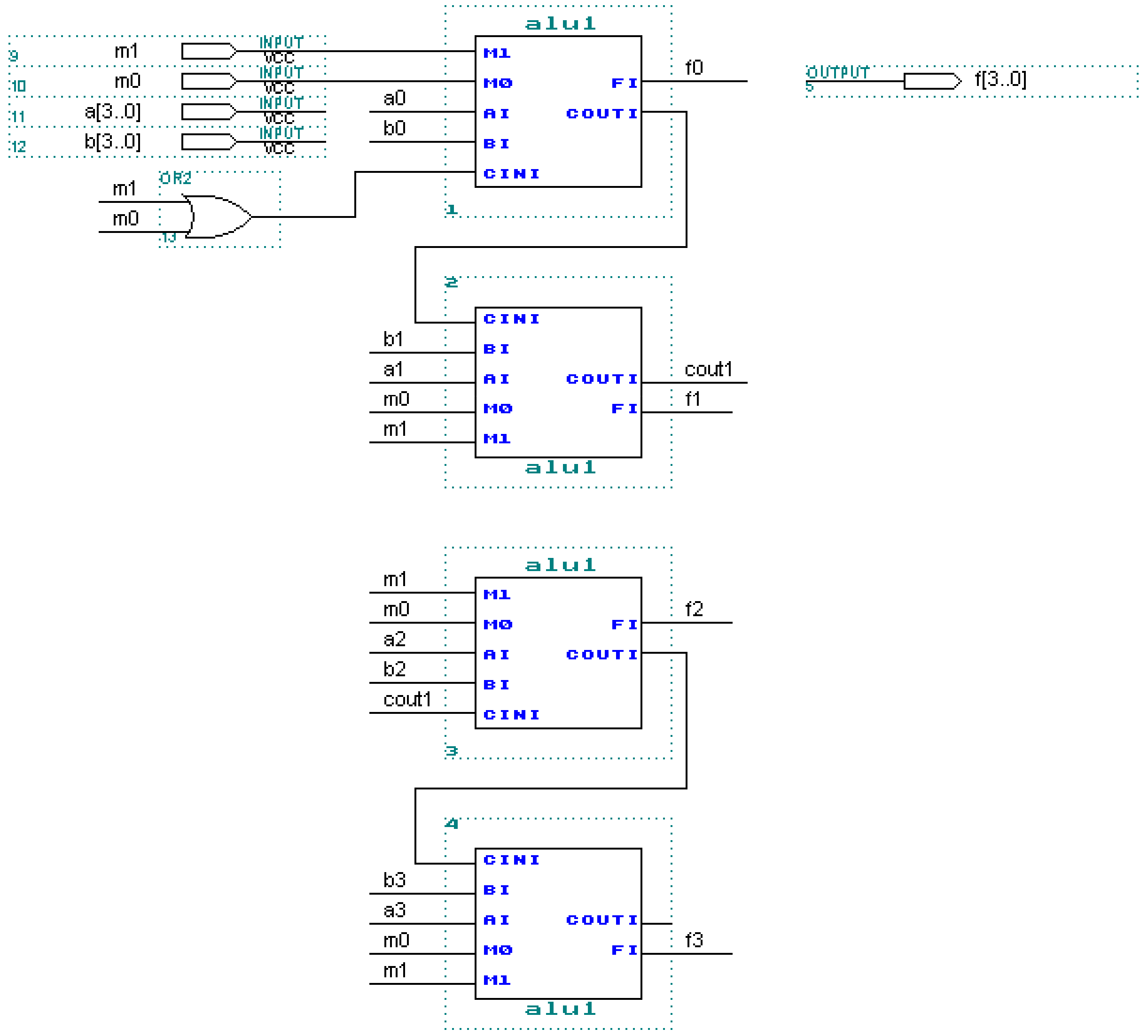


	m1	m0	ai	bi	cini	fi	couti
	0	0	0	0	0	0	0
			0	0	1	1	0
			0	1	0	1	0
A tambah B			0	1	1	0	1
			1	0	0	1	0
			1	0	1	0	1
			1	1	0	0	1
			1	1	1	1	1
	0	1	0	0	0	1	0
			0	0	1	0	1
			0	1	0	0	0
			0	1	1	1	0
A tolak B			1	0	0	0	1
			1	0	1	1	1
			1	1	0	1	0
			1	1	1	0	1
	1	0	0	0	0	0	0
			0	0	1	1	0
			0	1	0	0	0
A tambah 1			0	1	1	1	0
			1	0	0	1	0
			1	0	1	0	1
			1	1	0	1	0
			1	1	1	0	1
	1	1	0	0	X	1	X
A NOR B			0	1	X	0	X
			1	X	X	0	X

m1	m0	cin0
0	0	0
0	1	1
1	0	1
1	1	X

Sample ALU 2

Circuit for Sample ALU 2 for 4-Bit Inputs

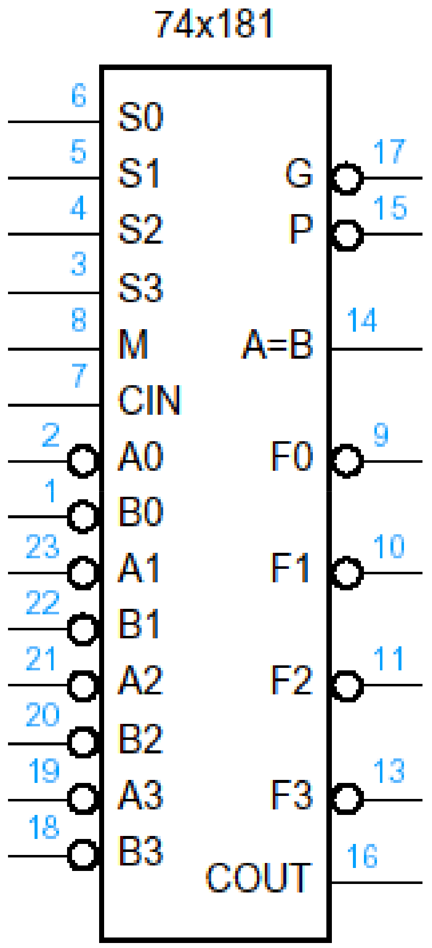


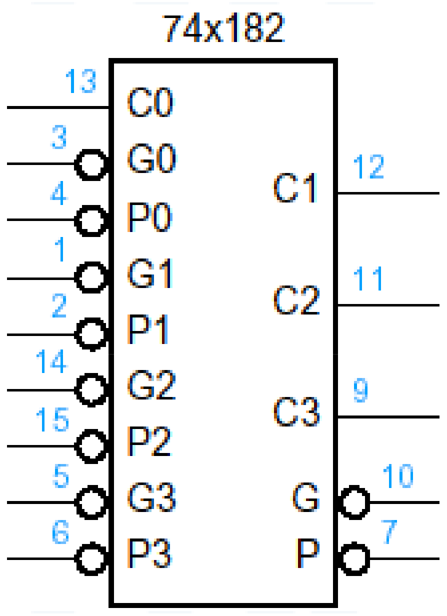
74x181 TTL ALU

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 plus (A + B) plus 1
1	0	0	1	F = A xor B	F = A plus B	F = A plus 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

- Due to arithmetic equivalence, active HIGH or active LOW input and outputs are available!
- Not all operations useful, but fall out when doing the useful ones

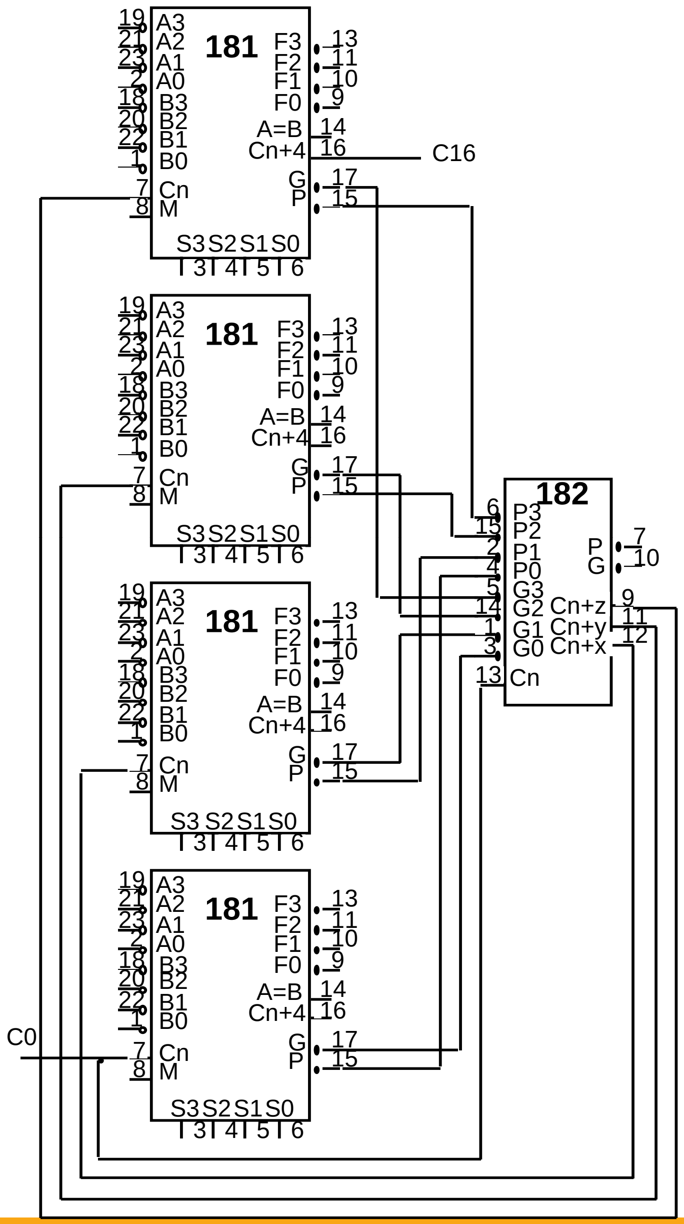
74x181 TTL ALU





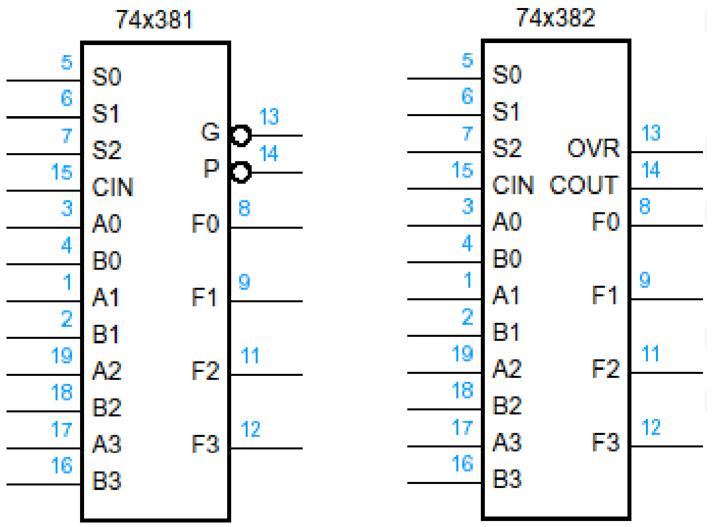
16-bit ALU with Carry Lookahead Unit

CLA unit speeds up calculations of multi-chip ALU



74x381 and 74x382 ALUs

Inputs			
S2	S1	S0	Function
0	0	0	F = 0000
0	0	1	F = B minus A minus 1 plus CIN
0	1	0	F = A minus B minus 1 plus CIN
0	1	1	F = A plus B plus CIN
1	0	0	F = A ⊕ B
1	0	1	F = A + B
1	1	0	F = A · B
1	1	1	F = 1111



- Compared to 74x181, these ALUs encode their select inputs more compactly, and provide only eight different but useful functions
- The difference?
  - 74x381 provides **group carry lookahead** outputs
  - 74x382 provides **ripple carry-out** and **overflow** outputs

Combinational Multiplier

- Product of 2 4-bit numbers is an 8-bit number
- Product of m-bit x n-bit numbers is an (m+n)-bit number

				A_3	A_2	A_1	A_0
				B_3	B_2	B_1	B_0
				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			
S_7	S_6	S_5	S_4	S_3	S_2	S_1	S_0

Partial products

				1	1	0	1	(13) multiplicand
		X		1	0	1	1	(11) multiplier
				1	1	0		
			1	1	0	1		
		0	0	0	0			
	1	1	0	1				
1	0	0	0	1	1	1	1	(143) product

Combinational Multiplier

		B_1	B_0
		A_1	A_0
		A_0B_1	A_0B_0
	A_1B_1	A_1B_0	
C_3	C_2	C_1	C_0

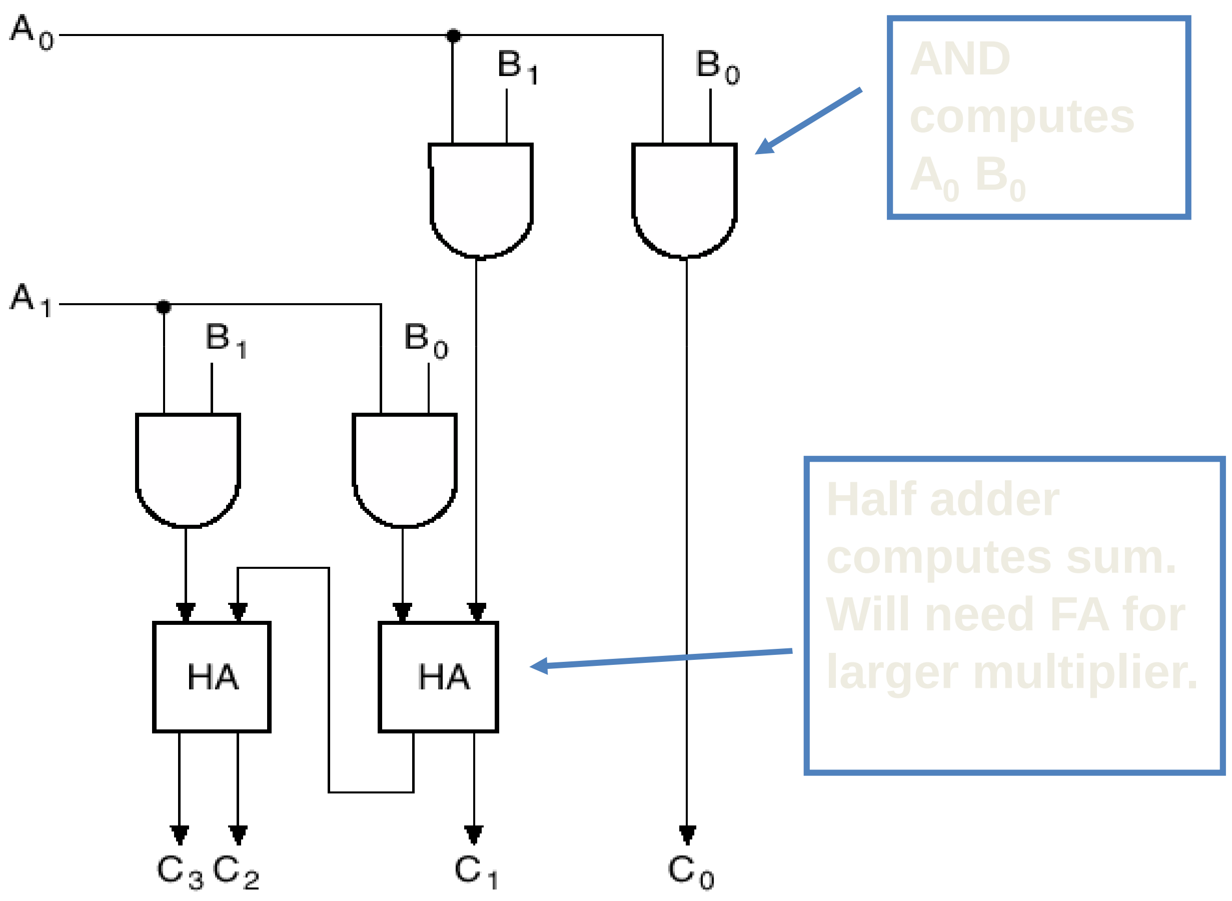
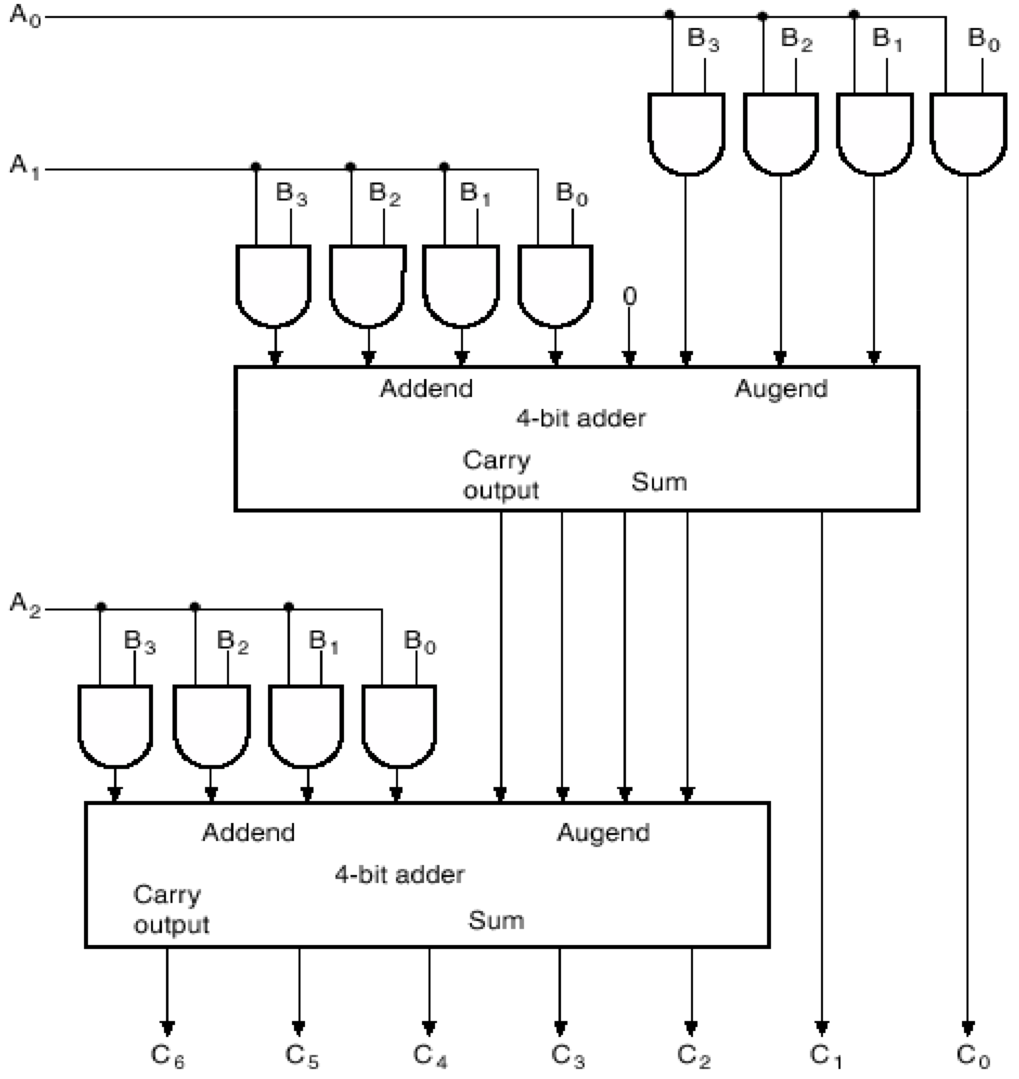
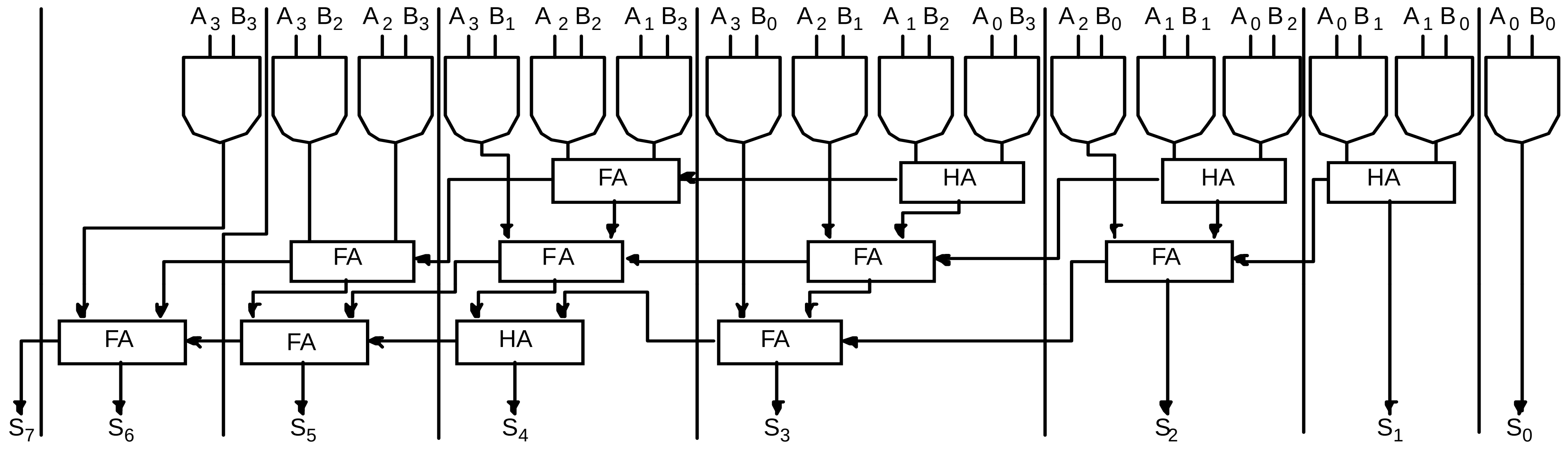


Fig. 3-33 A 2-Bit by 2-Bit Binary Multiplier

Basic Idea of A Larger Multiplier



4x4 Combinational Multiplier



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

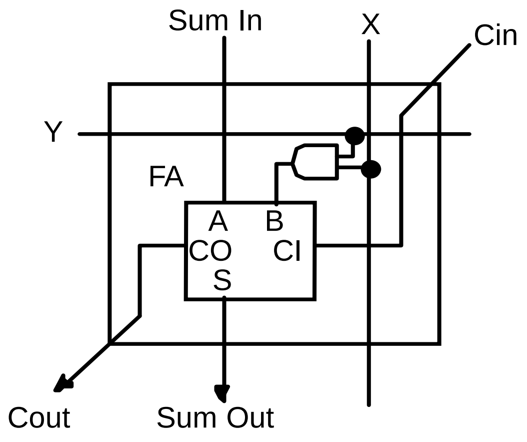
12 Adders, if full adders, this is 6 gates each = 72 gates

16 gates form the partial products

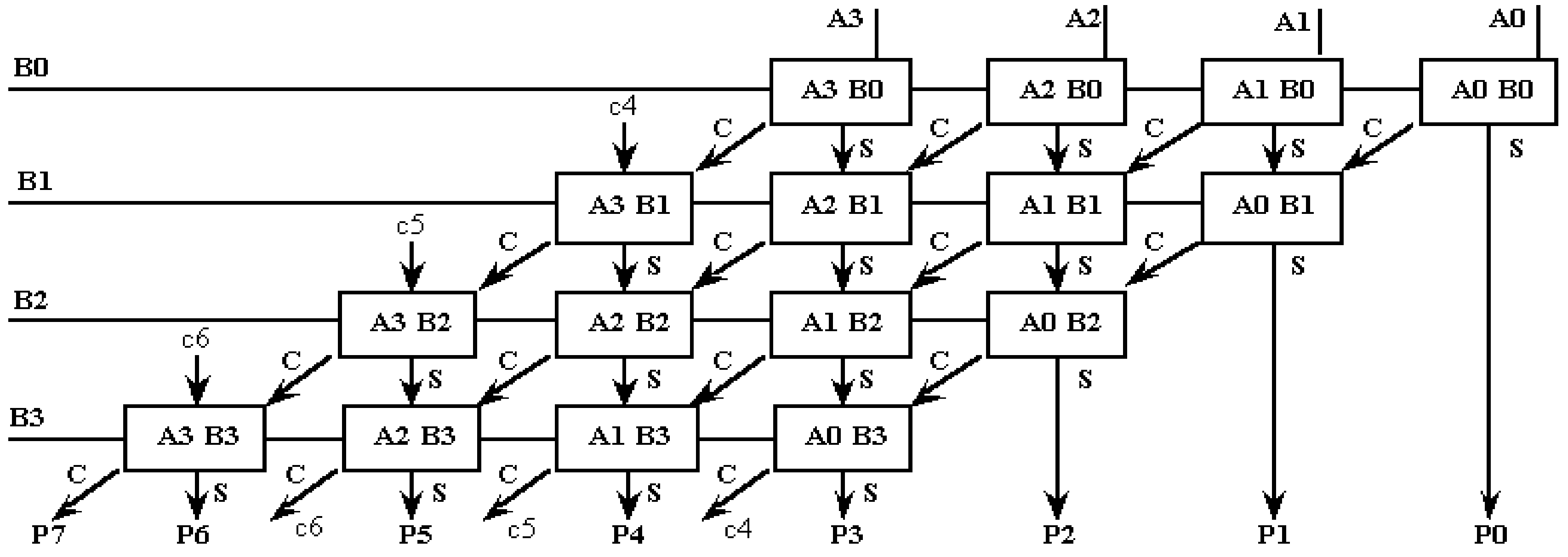
total = 88 gates!

Combinational Multiplier

Another Representation of the Circuit



Building block: full adder + and



4 x 4 array of building blocks