
Binary Number Systems and Arithmetic Circuits

Number systems

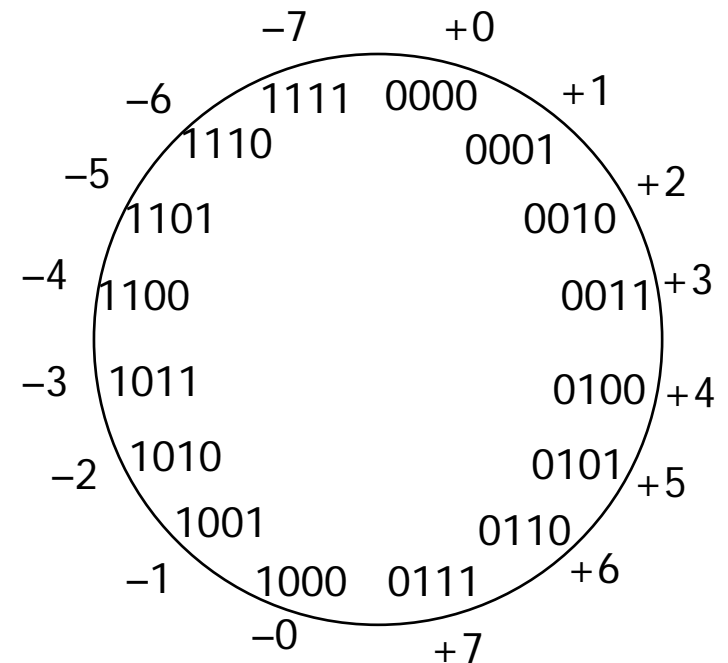
- Representation of positive numbers is the same in most systems
- Major differences are in how negative numbers are represented
- Representation of negative numbers come in three major schemes
 - sign and magnitude
 - 1s complement
 - 2s complement
 - excess code
- Assumptions
 - we'll assume a 4 bit machine word
 - 16 different values can be represented
 - roughly half are positive, half are negative

Sign and magnitude

- One bit dedicated to sign (positive or negative)
 - sign: 0 = positive (or zero), 1 = negative
- Rest represent the absolute value or magnitude
 - three low order bits: 0 (000) thru 7 (111)
- Range for n bits
 - $\pm(2^{n-1}-1)$ (two representations for 0)
- Cumbersome addition/subtraction
 - must compare magnitudes to determine sign of result

$$0\ 100 = +4$$

$$1\ 100 = -4$$



1s complement

- If N is a positive number, then the negative of N (its 1s complement or N') is $N' = (2^n - 1) - N$
 - example: 1s complement of 7

$$2^4 = 10000$$

$$-1 = \underline{-00001}$$

$$2^4 - 1 = 1111$$

$$-7 = \underline{-0111}$$

$$1000 = -7 \text{ in 1s complement form}$$

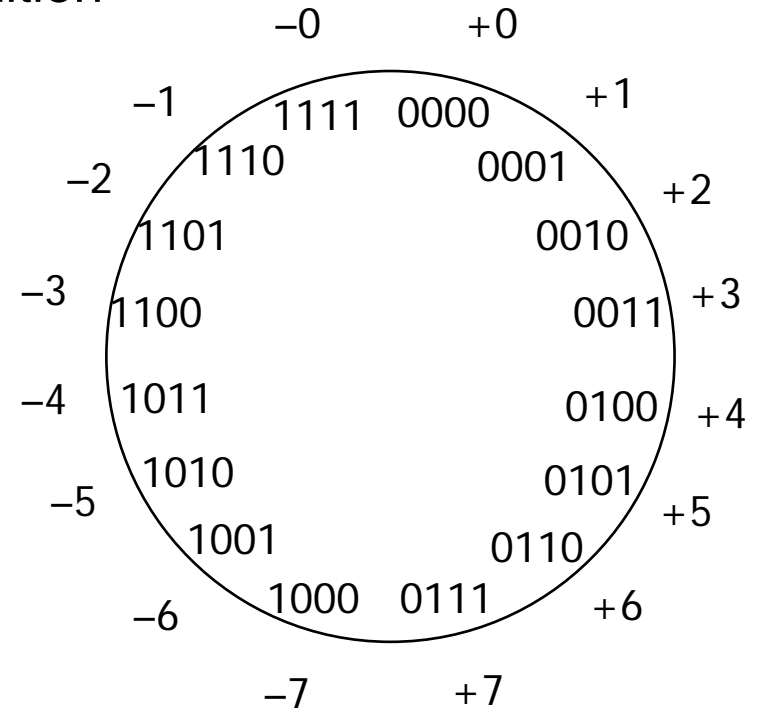
- shortcut: simply compute bit-wise complement (0111 \rightarrow 1000)

1s complement (cont'd)

- Subtraction implemented by 1s complement and then addition
- Two representations of 0
 - causes some complexities in addition
- High-order bit can act as sign bit

$$0\ 100 = +4$$

$$1\ 011 = -4$$

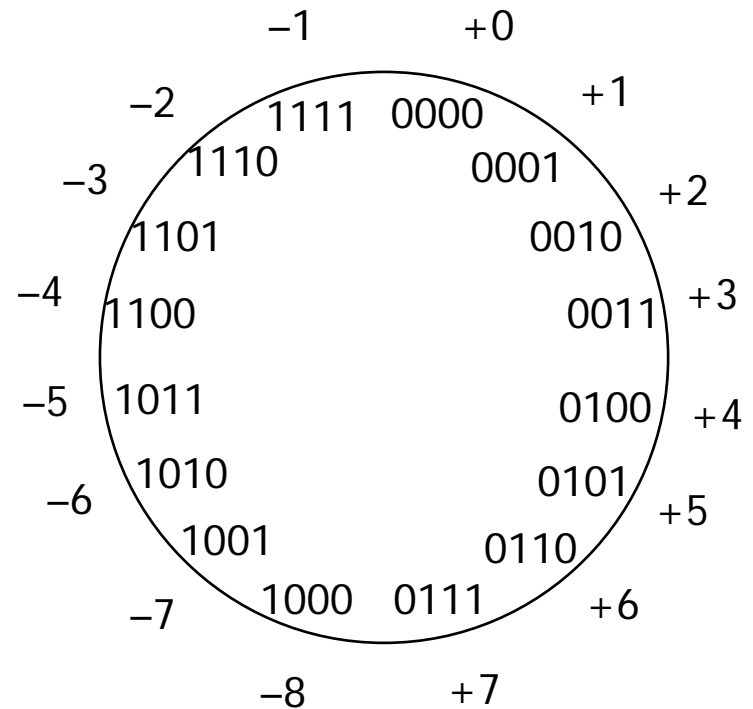


2s complement

- 1s complement with negative numbers shifted one position clockwise
 - only one representation for 0
 - one more negative number than positive numbers
 - high-order bit can act as sign bit

$$0\ 100 = +4$$

$$1\ 100 = -4$$



2s complement (cont'd)

- If N is a positive number, then the negative of N (its 2s complement or N^*) is $N^* = 2^n - N$

- example: 2s complement of 7

$$\begin{array}{r} 2^4 = 10000 \\ \text{subtract } 7 = \underline{0111} \\ \hline 1001 = \text{repr. of } -7 \end{array}$$

- example: 2s complement of -7

$$\begin{array}{r} 2^4 = 10000 \\ \text{subtract } -7 = \underline{1001} \\ \hline 0111 = \text{repr. of } 7 \end{array}$$

- shortcut: 2s complement = bit-wise complement + 1
 - $0111 \rightarrow 1000 + 1 \rightarrow 1001$ (representation of -7)
 - $1001 \rightarrow 0110 + 1 \rightarrow 0111$ (representation of 7)

Addition and subtraction

- Sign and Magnitude

when operands have the same sign, the result has the same sign as the operands

$$\begin{array}{r} 4 \quad 0100 \\ + 3 \quad 0011 \\ \hline 7 \quad 0111 \end{array}$$

$$\begin{array}{r} -4 \quad 1100 \\ + (-3) \quad 1011 \\ \hline -7 \quad 1111 \end{array}$$

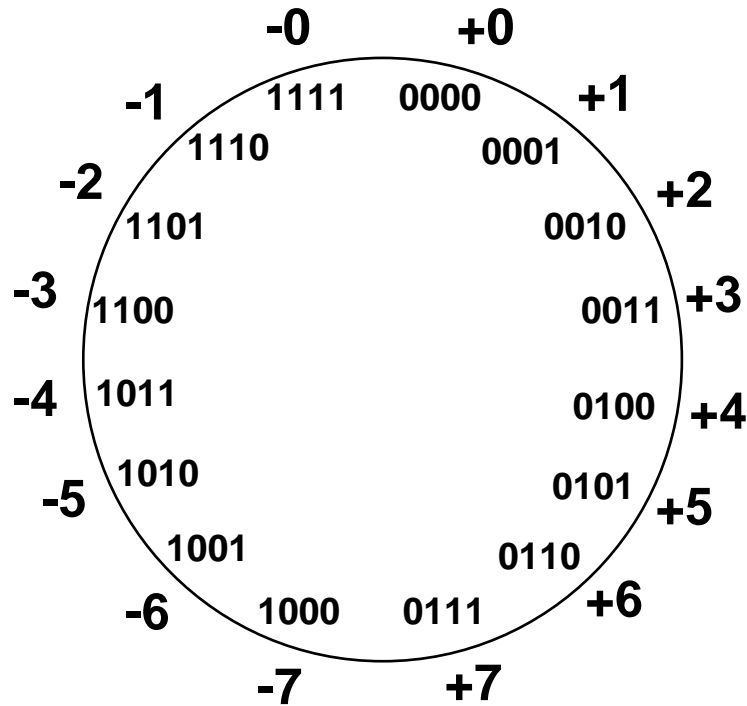
when signs differ, operation is subtract. sign of result depends on the sign of the number with larger magnitude

$$\begin{array}{r} 4 \quad 0100 \\ - 3 \quad 1011 \\ \hline 1 \quad 0001 \end{array}$$

$$\begin{array}{r} -4 \quad 1100 \\ + 3 \quad 0011 \\ \hline -1 \quad 1001 \end{array}$$

Addition and subtraction (cont'd)

■ Ones' Complement



4	0100	-4	1011
<u>+ 3</u>	<u>0011</u>	<u>+ (-3)</u>	<u>1100</u>
7	0111	-7	10111
		End around carry	└─→ 1
			<u>1000</u>

4	0100	-4	1011
<u>- 3</u>	<u>1100</u>	<u>+ 3</u>	<u>0011</u>
1	10000	-1	1110
	End around carry		└─→ 1
			<u>0001</u>

Addition and subtraction (cont'd)

- Why does end-around carry work?
 - It's equivalent to subtracting 2^n and adding 1

$$M - N = M + N' = M + (2^n - 1 - N) = (M - N) + 2^n - 1 \quad (M > N)$$

after end around carry:

$$= M - N$$

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

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

after end around carry:

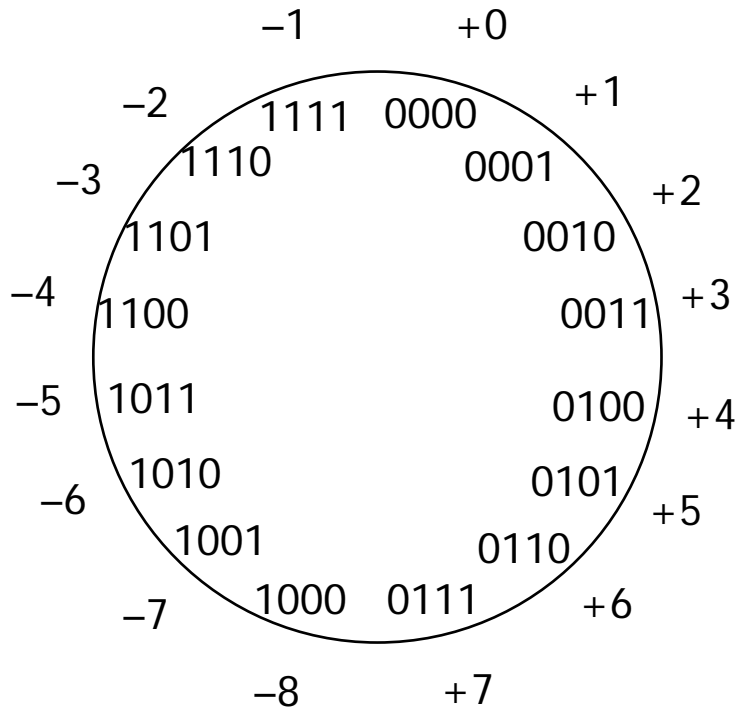
$$= 2^n - 1 - (M + N)$$

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

Addition and subtraction (cont'd)

- 2s complement

- simple scheme makes 2s complement the virtually unanimous choice for integer number systems in computers



4	0100	- 4	1100
<u>+ 3</u>	<u>0011</u>	<u>+ (- 3)</u>	<u>1101</u>
7	0111	- 7	11001
4	0100	- 4	1100
<u>- 3</u>	<u>1101</u>	<u>+ 3</u>	<u>0011</u>
1	10001	- 1	1111

Why can the carry-out be ignored?

- Can't ignore it completely
 - needed to check for overflow (see next two slides)
- When there is no overflow, carry-out may be true but can be ignored

– $M + N$ when $N > M$:

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

ignoring carry-out is just like subtracting 2^n

– $M + -N$ where $N + M \leq 2^{n-1}$

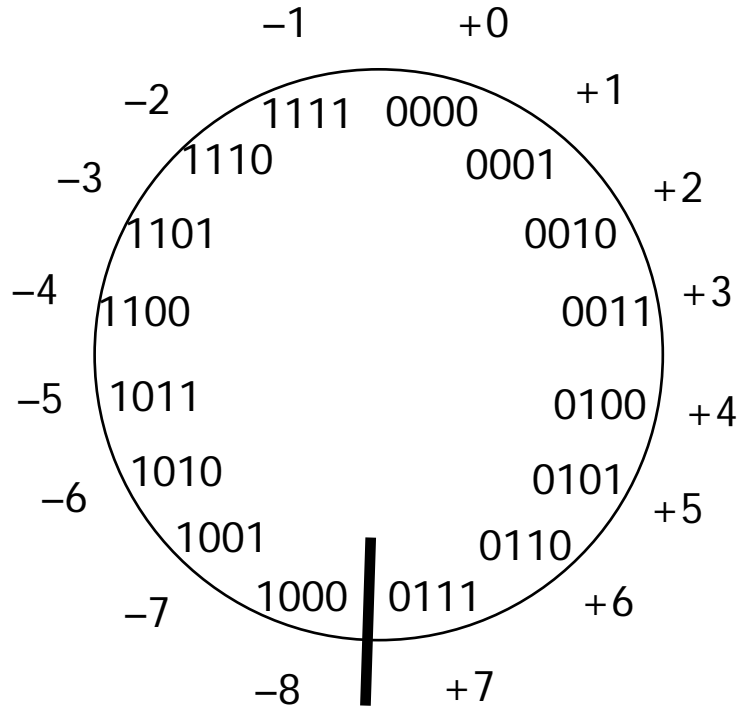
$$(-M) + (-N) = M^* + N^* = (2^n - M) + (2^n - N) = 2^n - (M + N) + 2^n$$

ignoring the carry, it is just the 2s complement representation for $-(M + N)$

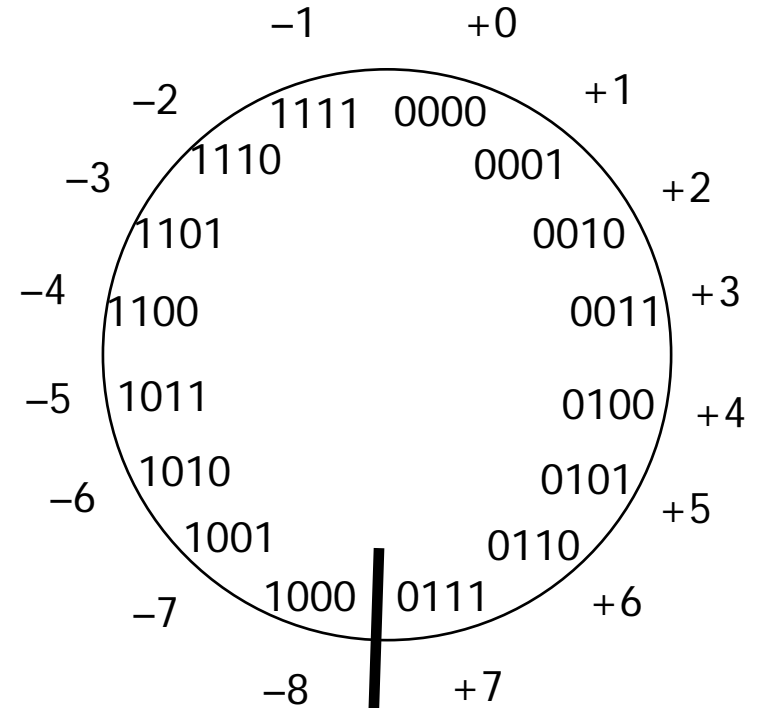
Overflow in 2s complement addition/subtraction

■ Overflow conditions

- ❑ add two positive numbers to get a negative number
- ❑ add two negative numbers to get a positive number



$$5 + 3 = -8$$



$$-7 - 2 = +7$$

Overflow conditions

- Overflow when carry into sign bit position is not equal to carry-out

$$\text{ovf} = c_{n-1} \oplus c_n$$

	0 1 1 1
5	0 1 0 1
<u>3</u>	<u>0 0 1 1</u>
-8	1 0 0 0

overflow

	1 0 0 0
-7	1 0 0 1
<u>-2</u>	<u>1 1 1 0</u>
7	1 0 1 1 1

overflow

	0 0 0 0
5	0 1 0 1
<u>2</u>	<u>0 0 1 0</u>
7	0 1 1 1

no overflow

	1 1 1 1
-3	1 1 0 1
<u>-5</u>	<u>1 0 1 1</u>
-8	1 1 0 0 0

no overflow

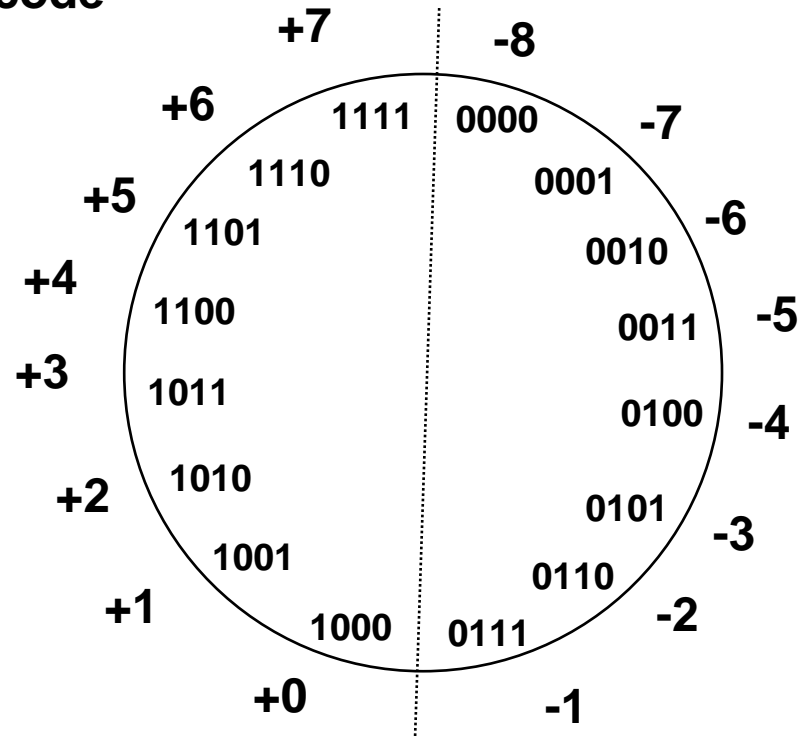
Excess code

Used for floating point representation



$$F = (-1)^s 1.M 2^E$$

Excess 8 code



Arithmetic circuits

- Excellent examples of combinational logic design
- Time vs. space trade-offs
 - doing things fast may require more logic and thus more space
 - example: carry lookahead logic
- Arithmetic and logic units
 - general-purpose building blocks
 - critical components of processor datapaths
 - used within most computer instructions

Circuits for binary addition

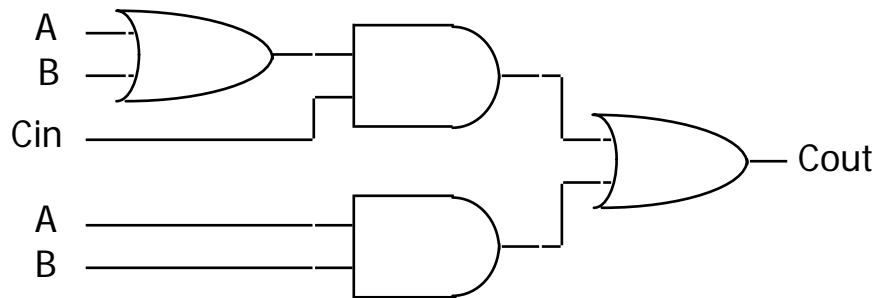
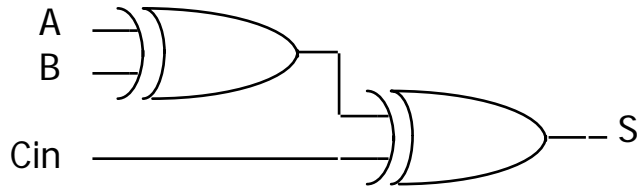
- Half adder (add 2 1-bit numbers)
 - $\text{Sum} = A_i' B_i + A_i B_i' = A_i \text{ xor } B_i$
 - $\text{Cout} = A_i B_i$
- Full adder (carry-in to cascade for multi-bit adders)
 - $\text{Sum} = C_i \text{ xor } A \text{ xor } B$
 - $\text{Cout} = B C_i + A C_i + A B = C_i (A + B) + A B = C_i (A \text{ xor } B) + A B$

Ai	Bi	Sum	Cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Ai	Bi	Cin	Sum	Cout
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

Full adder implementations

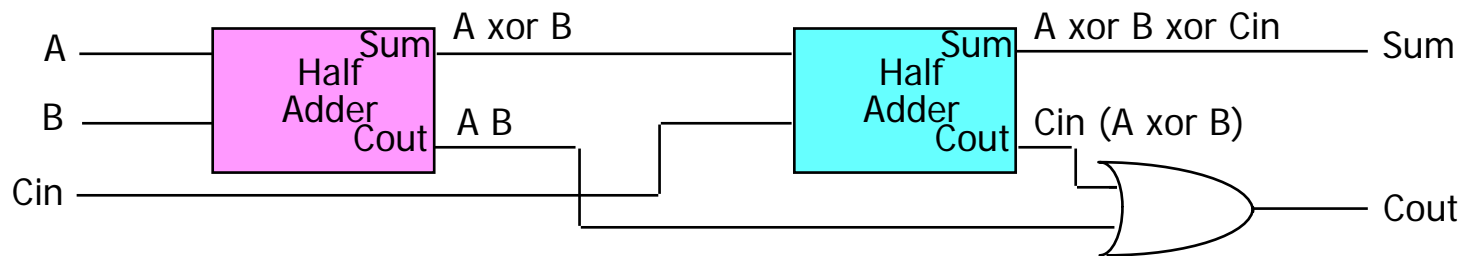
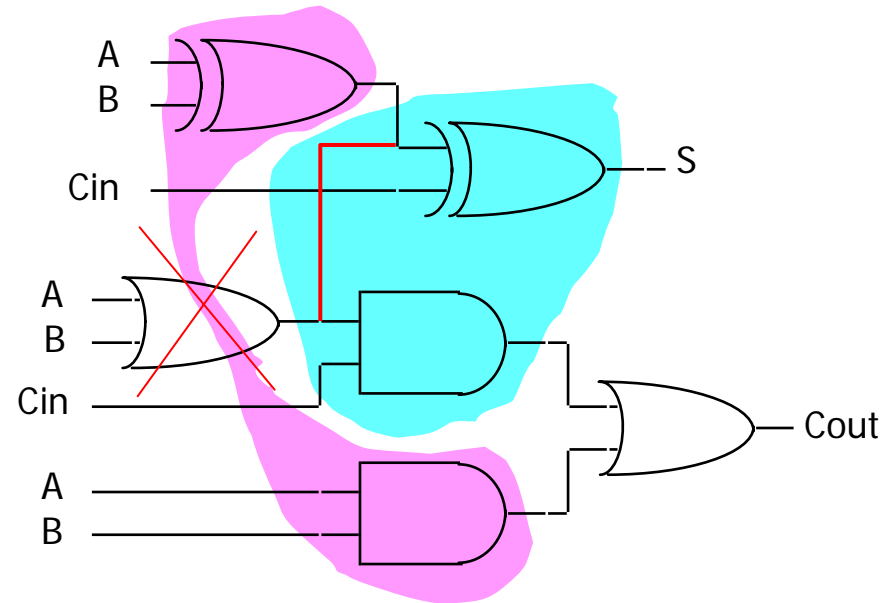
- Standard approach
 - ❑ 6 gates
 - ❑ 2 XORs, 2 ANDs, 2 ORs



Full adder implementations

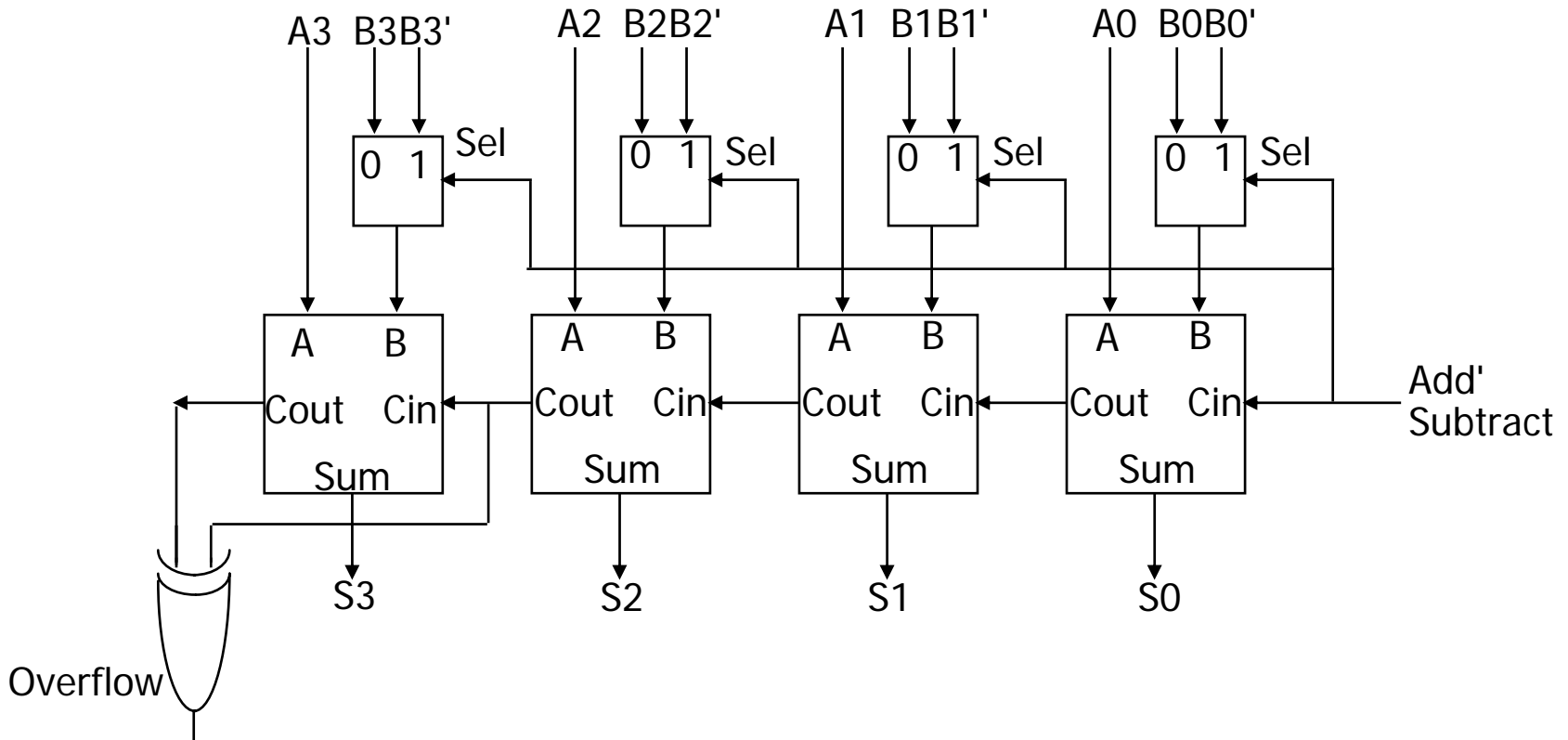
■ Alternative implementation

- $C_{out} = A B + B C_{in} + A C_{in} = A B + C_{in} (A \text{ xor } B)$
- 5 gates
- 2 XORs, 2 ANDs, 1 OR
- two half adders and one OR gate



Adder/subtractor

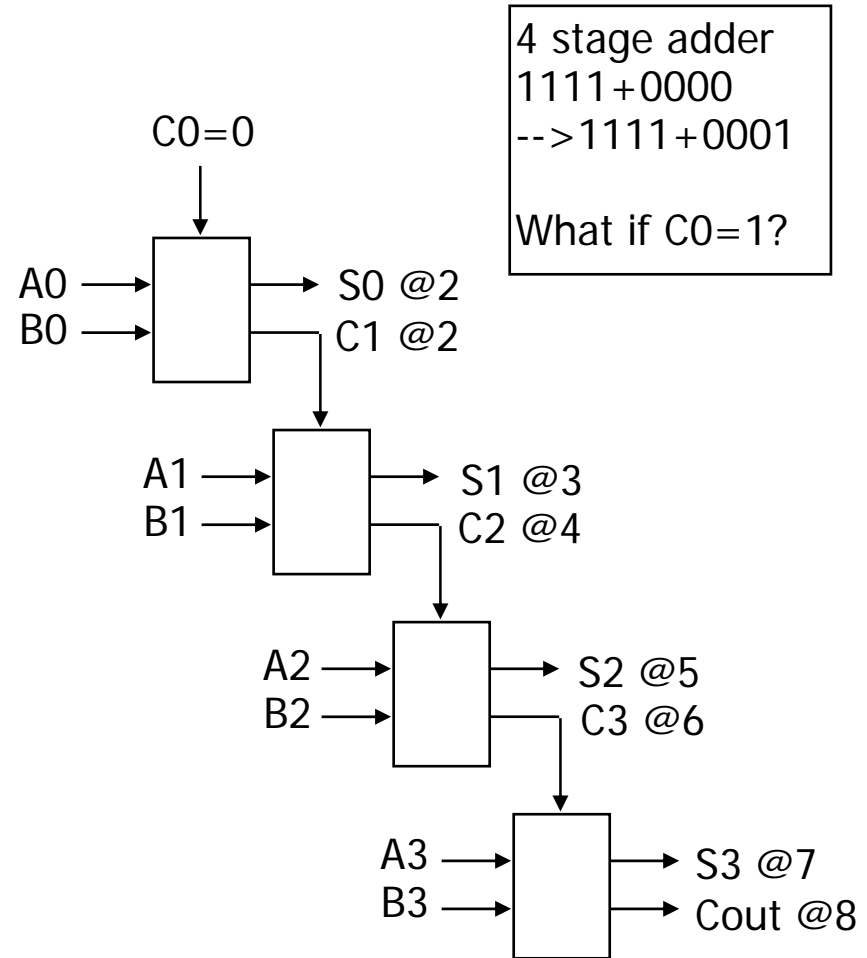
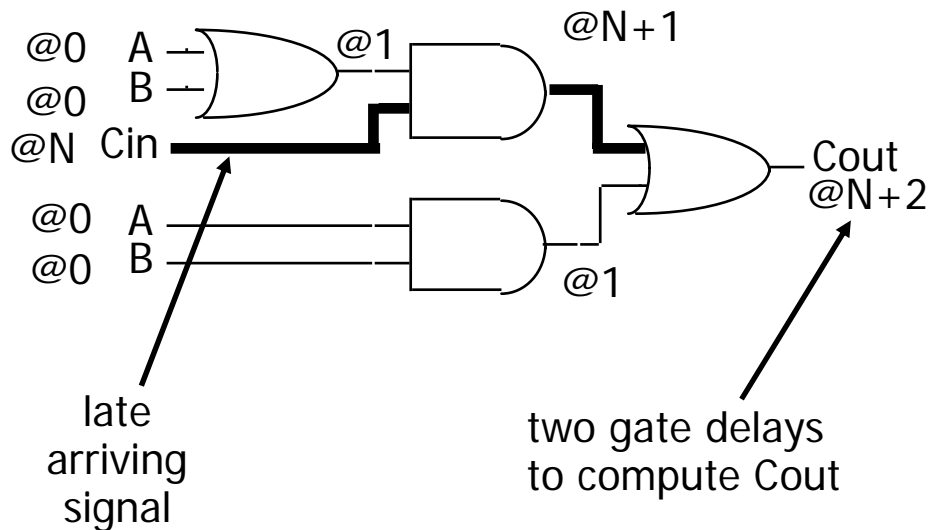
- Use an adder to do subtraction thanks to 2s complement representation
 - $A - B = A + (-B) = A + B' + 1$
 - control signal selects B or 2s complement of B



Ripple-carry adders

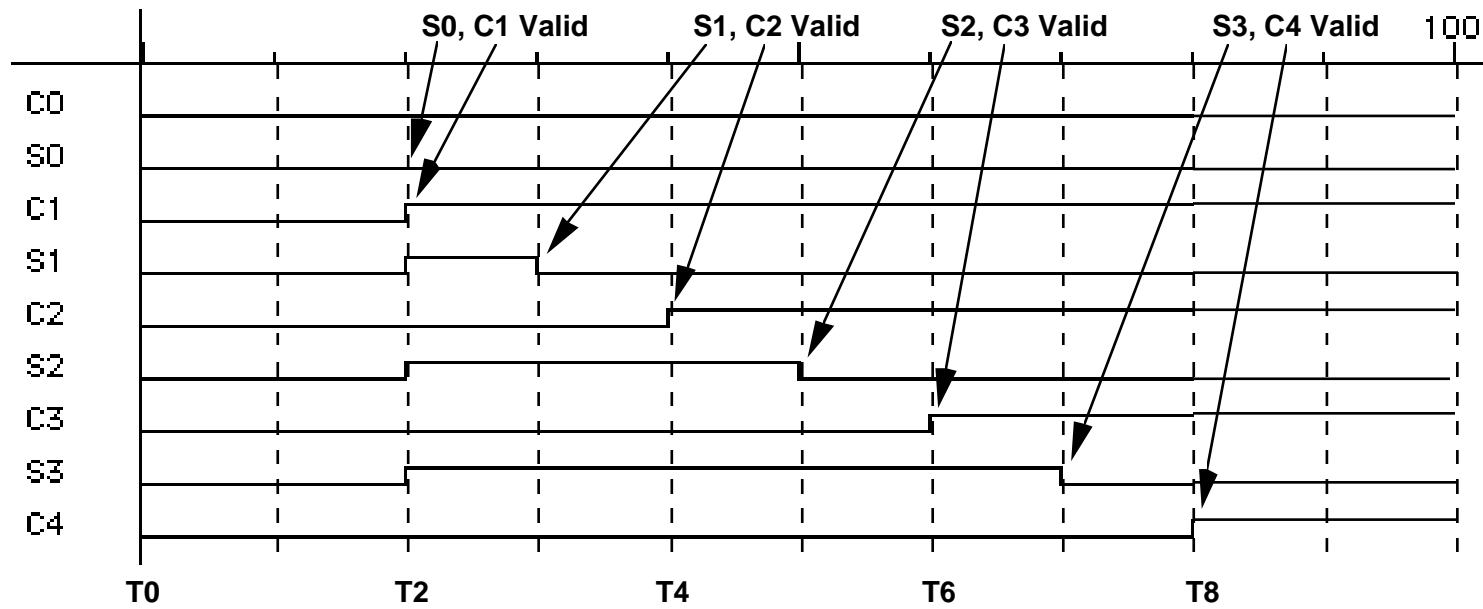
- Critical delay

- the propagation of carry from low to high order stages



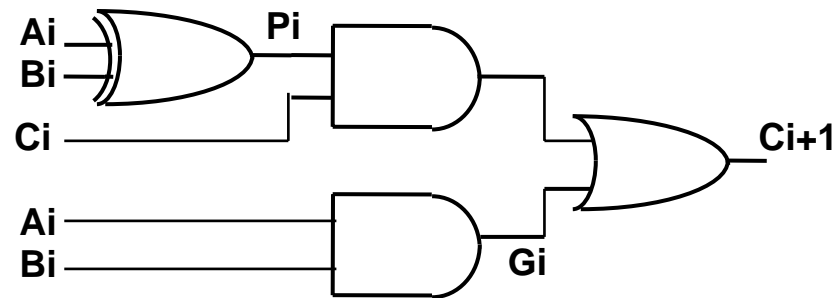
Ripple-carry adders (cont'd)

- Critical delay
 - the propagation of carry from low to high order stages
 - 1111 + 0001 is the worst case addition
 - carry must propagate through all bits



Carry-lookahead logic

- Carry generate: $G_i = A_i B_i$
 - must generate carry when $A = B = 1$
- Carry propagate: $P_i = A_i \text{ xor } B_i$
 - carry-in will equal carry-out here
- Sum and Cout can be re-expressed in terms of generate/propagate:
 - $S_i = A_i \text{ xor } B_i \text{ xor } C_i$
 $= P_i \text{ xor } C_i$
 - $C_{i+1} = A_i B_i + A_i C_i + B_i C_i$
 $= A_i B_i + C_i (A_i + B_i)$
 $= A_i B_i + C_i (A_i \text{ xor } B_i)$
 $= G_i + C_i P_i$

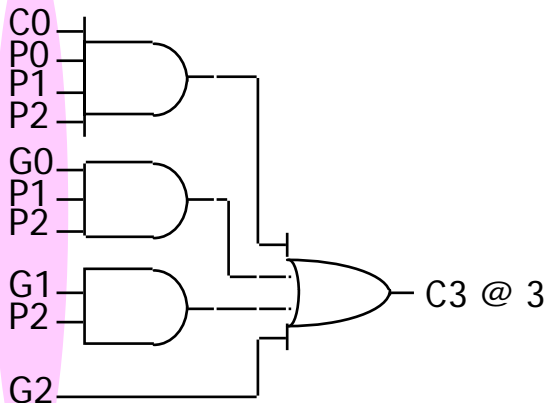
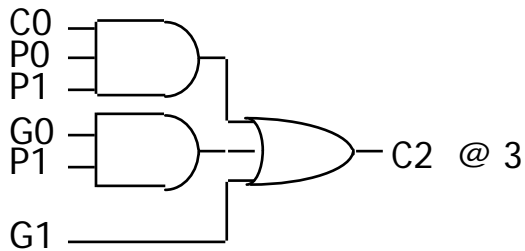
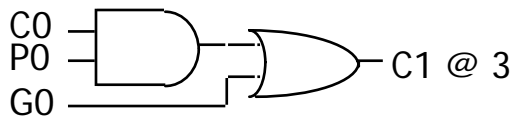
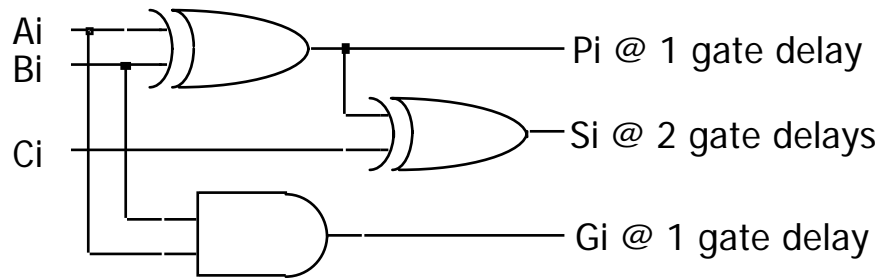


Carry-lookahead logic (cont'd)

- Re-express the carry logic as follows:
 - $C_1 = G_0 + P_0 C_0$
 - $C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_0$
 - $C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$
 - $C_4 = G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$
- Each of the carry equations can be implemented with two-level logic
 - all inputs are now directly derived from data inputs and not from intermediate carries
 - this allows computation of all sum outputs to proceed in parallel

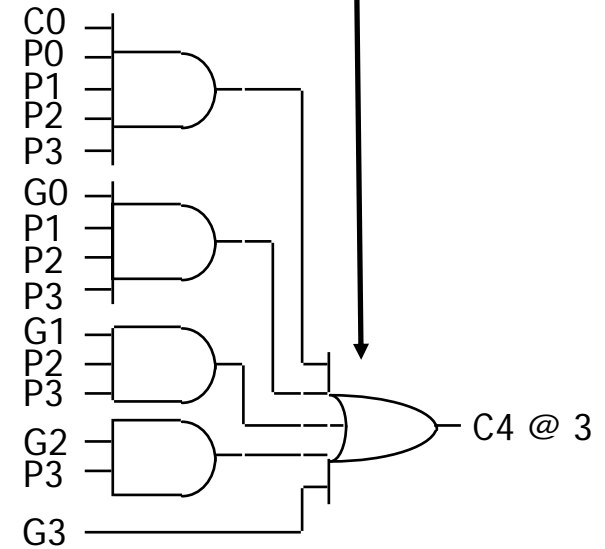
Carry-lookahead implementation

- Adder with propagate and generate outputs



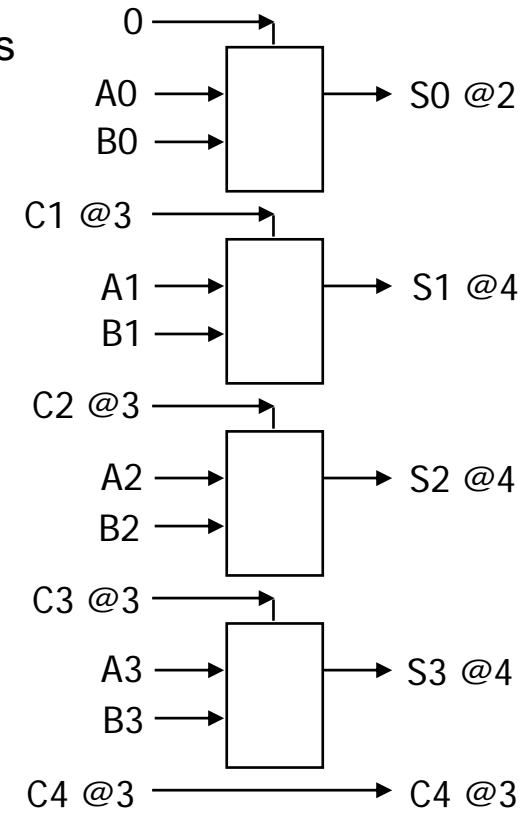
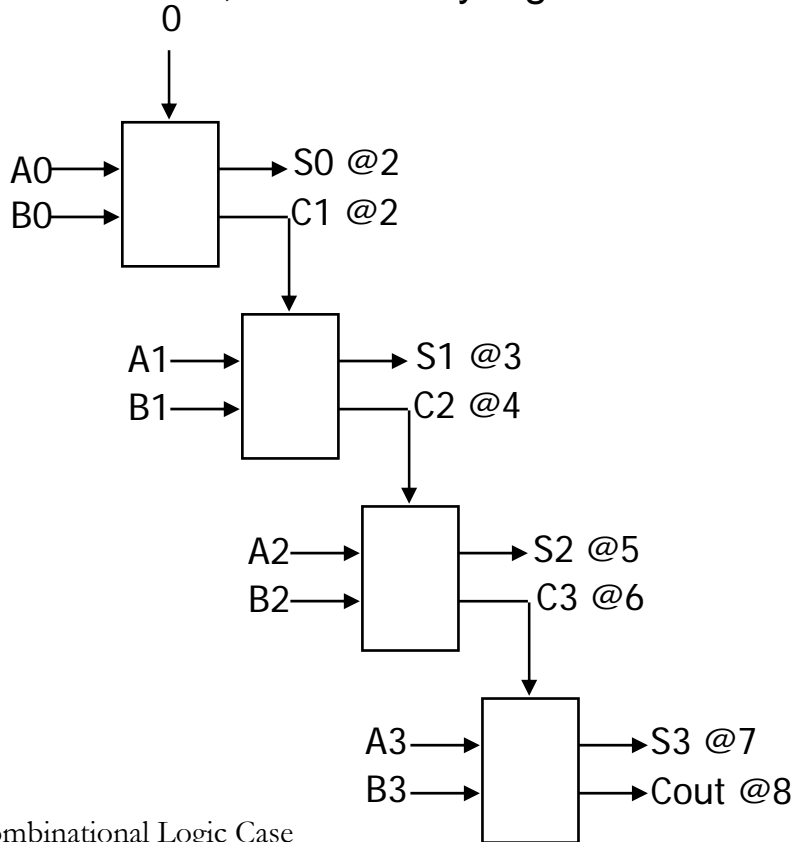
1 gate delay

increasingly complex logic for carries



Carry-lookahead implementation (cont'd)

- Carry-lookahead logic generates individual carries
 - sums computed much more quickly in parallel
 - however, cost of carry logic increases with more stages



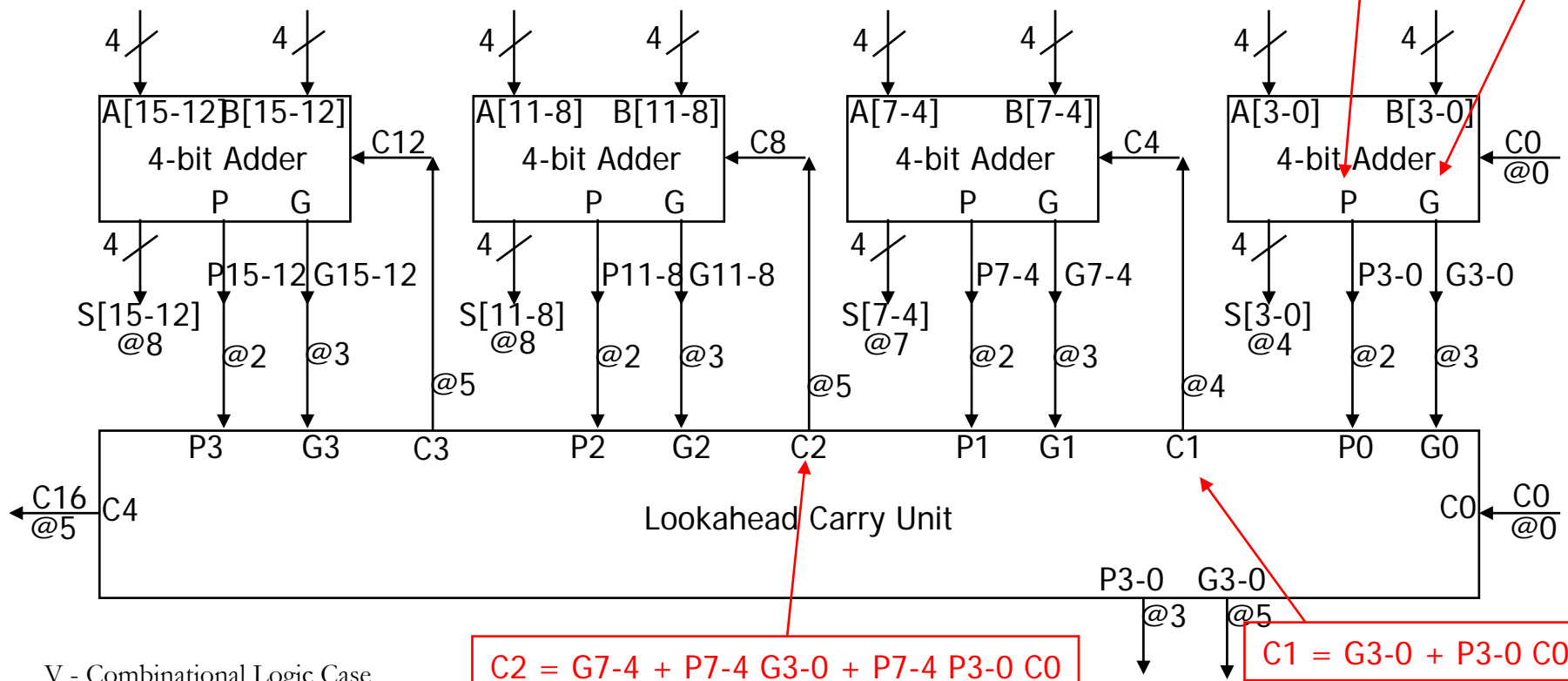
Carry-lookahead adder with cascaded carry-lookahead logic

- Carry-lookahead adder

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

$$G = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0$$

$$P = P_3 P_2 P_1 P_0$$

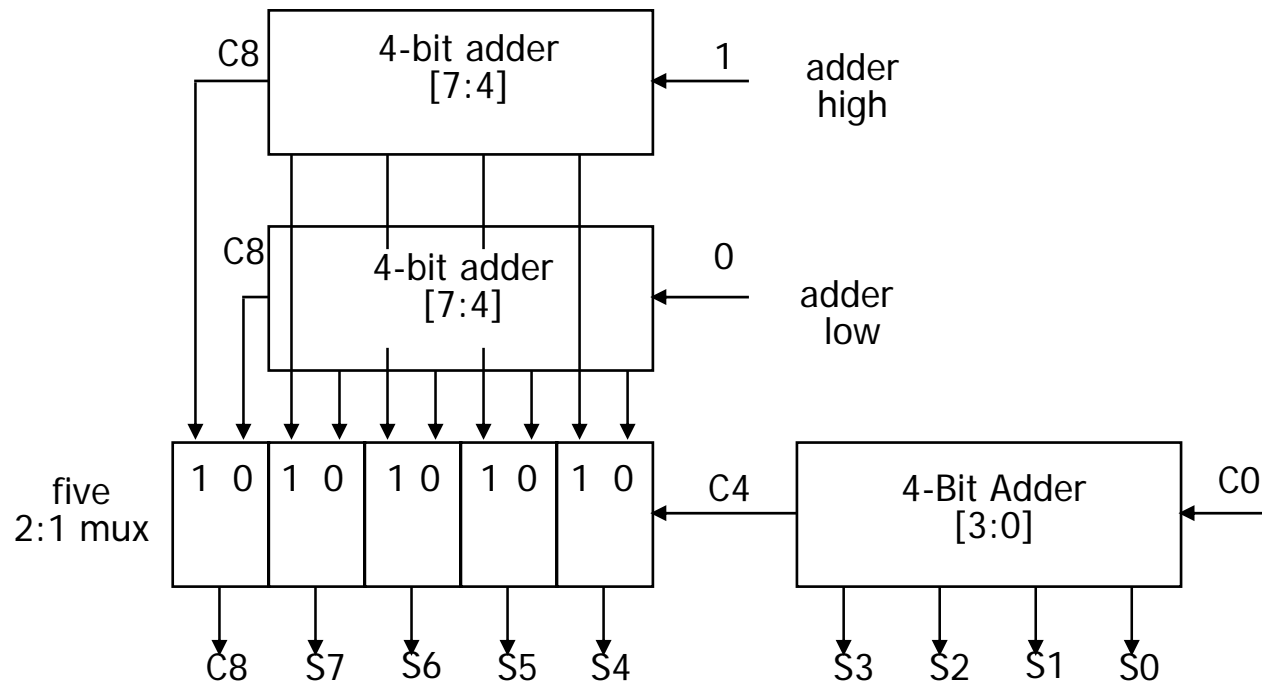


$$C_2 = G_{7-4} + P_{7-4} G_{3-0} + P_{7-4} P_{3-0} C_0$$

$$C_1 = G_{3-0} + P_{3-0} C_0$$

Carry-select adder

- Redundant hardware to make carry calculation go faster
 - ❑ compute two high-order sums in parallel while waiting for carry-in
 - ❑ one assuming carry-in is 0 and another assuming carry-in is 1
 - ❑ select correct result once carry-in is finally computed



Arithmetic logic unit design specification

M = 0, logical bitwise operations

S1	S0	Function	Comment
0	0	$F_i = A_i$	input A_i transferred to output
0	1	$F_i = \text{not } A_i$	complement of A_i transferred to output
1	0	$F_i = A_i \text{ xor } B_i$	compute XOR of A_i, B_i
1	1	$F_i = A_i \text{ xnor } B_i$	compute XNOR of A_i, B_i

M = 1, CO = 0, arithmetic operations

0	0	$F = A$	input A passed to output
0	1	$F = \text{not } A$	complement of A passed to output
1	0	$F = A \text{ plus } B$	sum of A and B
1	1	$F = (\text{not } A) \text{ plus } B$	sum of B and complement of A

M = 1, CO = 1, arithmetic operations

0	0	$F = A \text{ plus } 1$	increment A
0	1	$F = (\text{not } A) \text{ plus } 1$	twos complement of A
1	0	$F = A \text{ plus } B \text{ plus } 1$	increment sum of A and B
1	1	$F = (\text{not } A) \text{ plus } B \text{ plus } 1$	B minus A

logical and arithmetic operations
not all operations appear useful, but "fall out" of internal logic

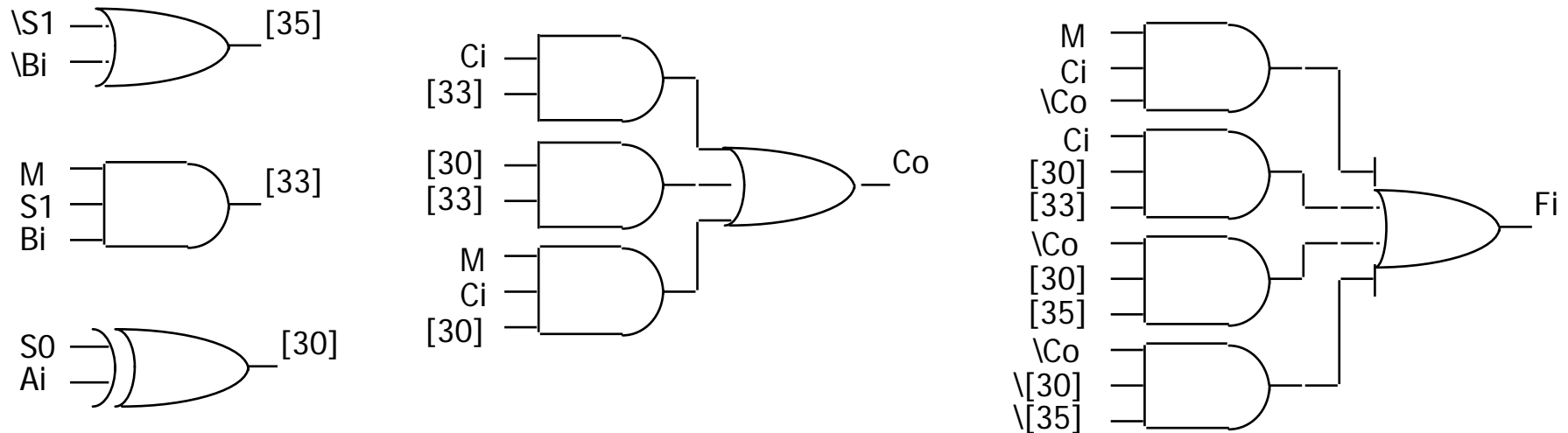
Arithmetic logic unit design (cont'd)

- Sample ALU – truth table

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

Arithmetic logic unit design (cont'd)

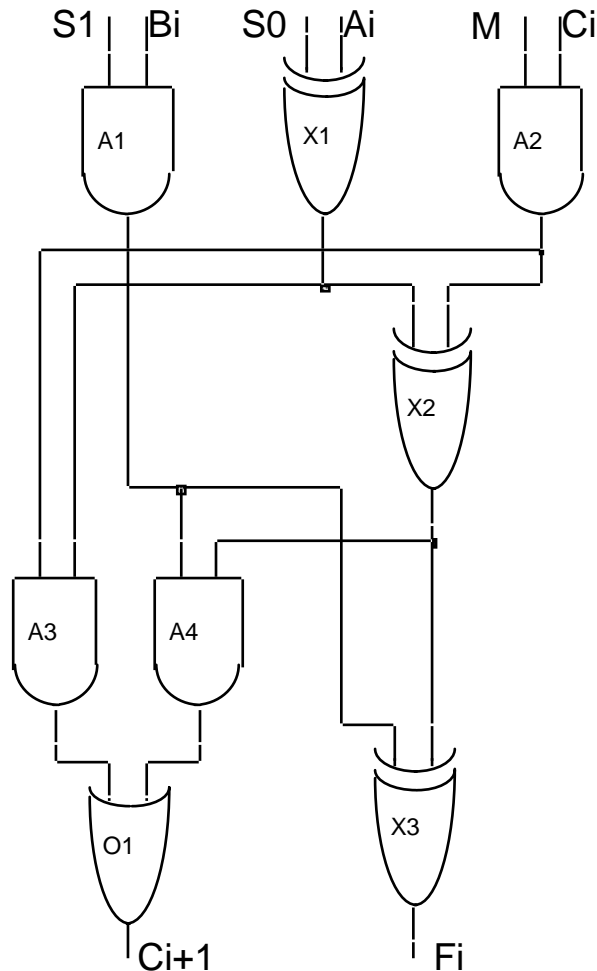
- Sample ALU – multi-level discrete gate logic implementation



12 gates

Arithmetic logic unit design (cont'd)

■ Sample ALU – clever multi-level implementation



first-level gates

use S0 to complement Ai

S0 = 0 causes gate X1 to pass Ai

S0 = 1 causes gate X1 to pass Ai'

use S1 to block Bi

S1 = 0 causes gate A1 to make Bi go forward as 0
(don't want Bi for operations with Ai)

S1 = 1 causes gate A1 to pass Bi

use M to block Ci

M = 0 causes gate A2 to make Ci go forward as 0
(don't want Ci for logical operations)

M = 1 causes gate A2 to pass Ci

other gates

for M=0 (logical operations, Ci is ignored)

$$F_i = S_1 B_i \text{ xor } (S_0 \text{ xor } A_i)$$

$$= S_1' S_0' (A_i) + S_1' S_0 (A_i') + S_1 S_0' (A_i B_i' + A_i' B_i) + S_1 S_0 (A_i' B_i' + A_i B_i)$$

for M=1 (arithmetic operations)

$$F_i = S_1 B_i \text{ xor } ((S_0 \text{ xor } A_i) \text{ xor } C_i) =$$

$$C_{i+1} = C_i (S_0 \text{ xor } A_i) + S_1 B_i ((S_0 \text{ xor } A_i) \text{ xor } C_i) =$$

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

Combinational multiplier

- Basic concept

multiplicand 1101 (13)
multiplier * 1011 (11)

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

Partial products

1101
1101
0000
1101

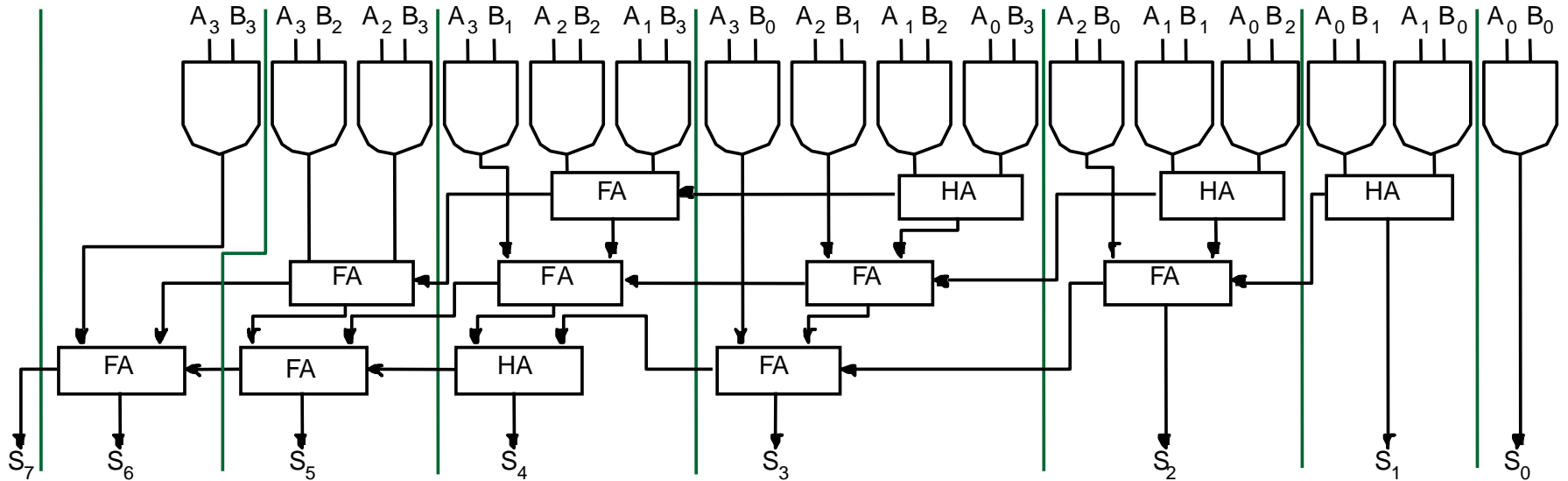
10001111 (143)

Combinational multiplier (cont'd)

- Partial product accumulation

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

Combinational multiplier (cont'd)



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

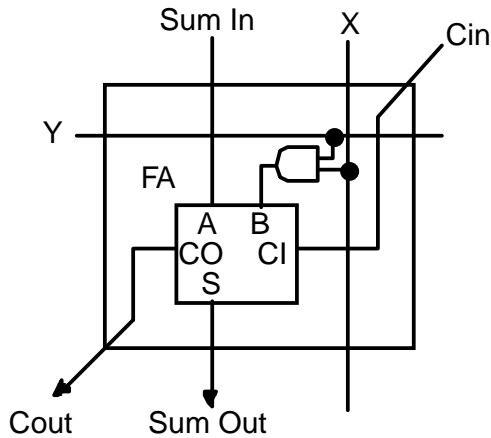
12 Adders, if full adders, this is 6 gates each = 72 gates

16 gates form the partial products

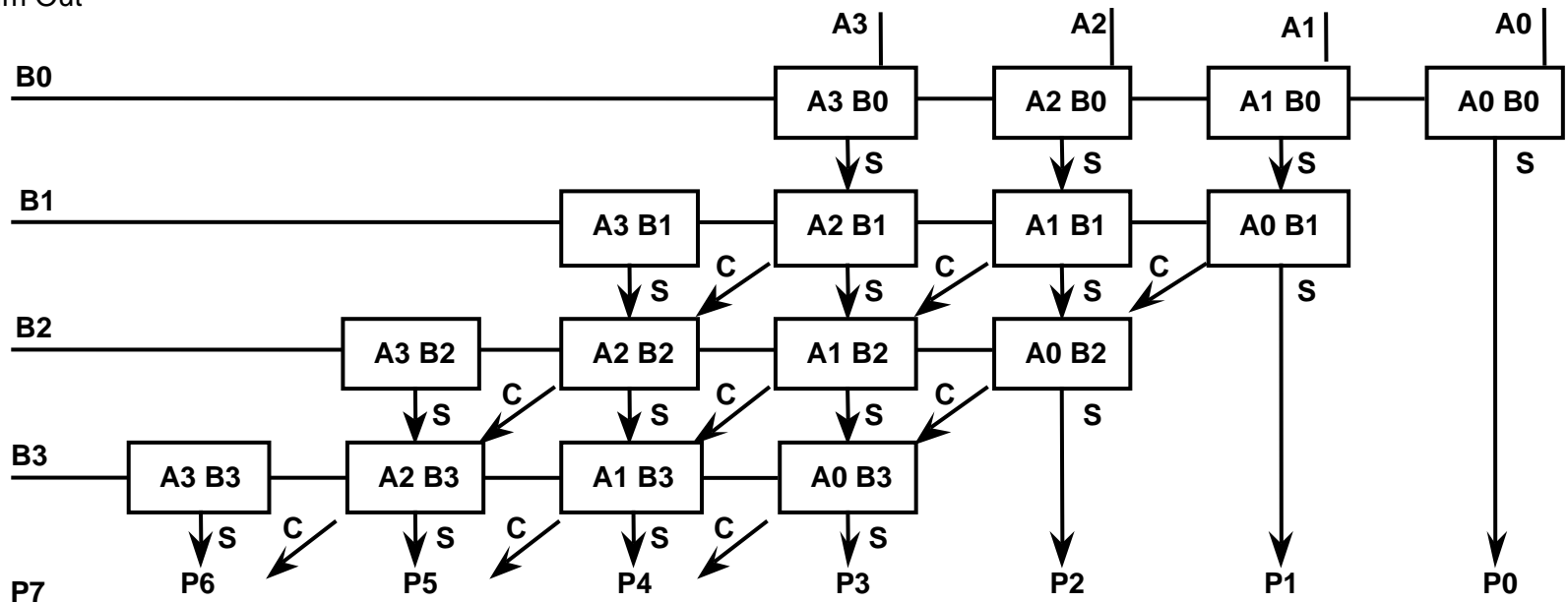
total = 88 gates!

Combinational multiplier (cont'd)

- Another representation of the circuit



Building block: full adder + and



4 x 4 array of building blocks