Arithmetic Circuits

COE 202

Digital Logic Design

Dr. Muhamed Mudawar

King Fahd University of Petroleum and Minerals

Presentation Outline

- Ripple-Carry Adder
- Magnitude Comparator
- Design by Contraction
- Signed Numbers
- Addition/Subtraction of Signed 2's Complement

Binary Addition

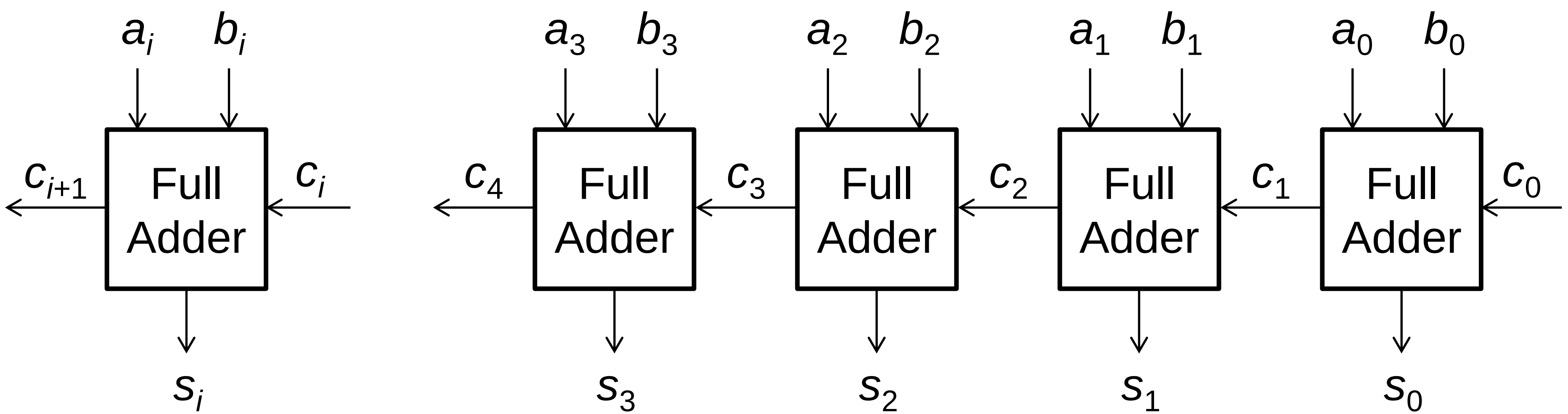
- Start with the least significant bit (rightmost bit)
- Add each pair of bits
- Include the carry in the addition

carry		**1**	**1**	**1**	**1**				
	0	0	1	1	0	1	1	0	(54)
+	0	0	0	1	1	1	0	1	(29)
	0	1	0	1	0	0	1	1	(83)
bit position:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	

Iterative Design: Ripple Carry Adder

- Uses **identical copies** of a full adder to build a large adder
- Simple to implement: can be extended to add any number of bits
- The **cell** (iterative block) is a **full adder**

  Adds 3 bits: a_i, b_i, c_i, Computes: Sum s_i and Carry-out c_{i+1}
- Carry-out of cell i becomes carry-in to cell $(i+1)$



Full-Adder Equations

$$s_i = a_i'b_i'c_i + a_i'b_ic_i' + a_ib_i'c_i' + a_ib_ic_i$$

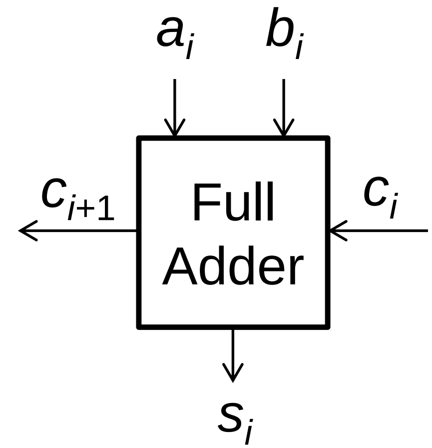
$$s_i = \text{odd function} = (a_i \oplus b_i) \oplus c_i$$

$$c_{i+1} = a_i'b_ic_i + a_ib_i'c_i + a_ib_ic_i' + a_ib_ic_i$$

$$c_{i+1} = (a_i'b_i + a_ib_i')\, c_i + a_ib_i\, (c_i' + c_i)$$

$$c_{i+1} = (a_i \oplus b_i)\, c_i + a_ib_i$$

K-map: $c_{i+1} = a_ib_i + a_ic_i + b_ic_i$



a_i b_i c_i	c_{i+1}	s_i
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

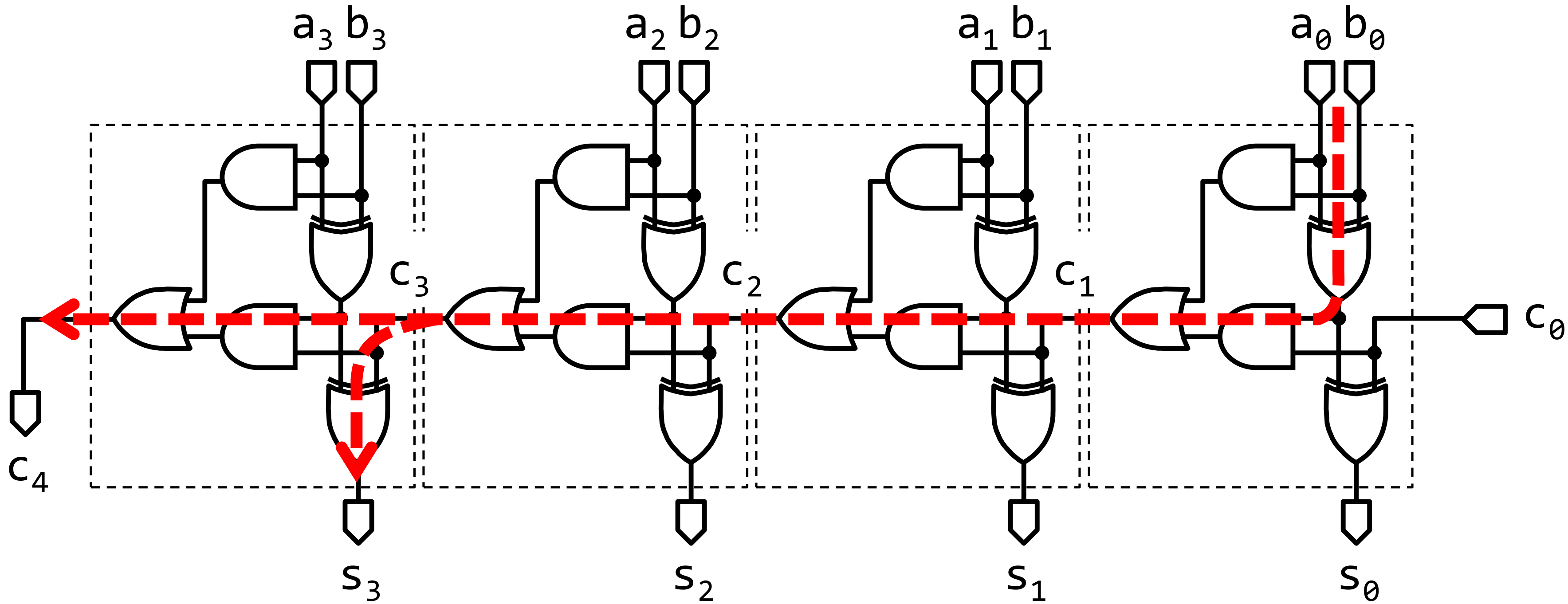
K-Map of s_i

a_i \ b_ic_i	00	01	11	10
0	0	1	0	1
1	1	0	1	0

K-Map of c_{i+1}

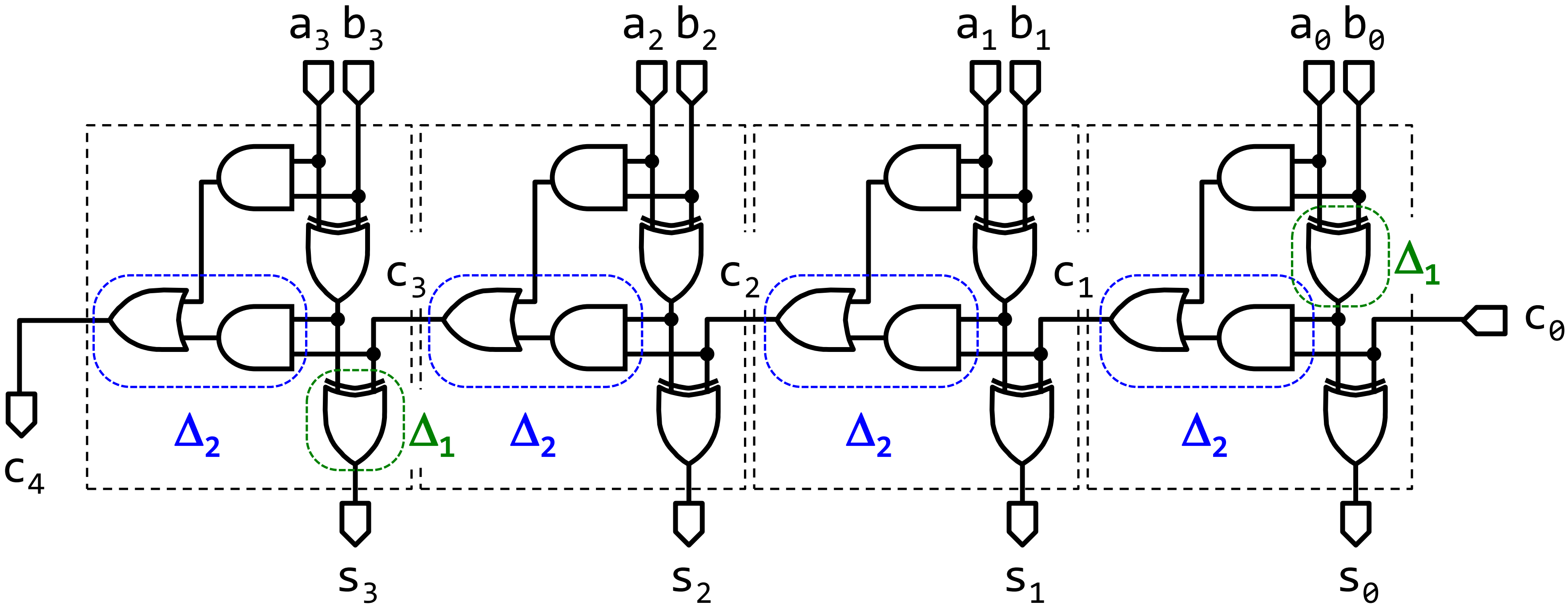
a_i \ b_ic_i	00	01	11	10
0	0	0	1	0
1	0	1	1	1

Carry Propagation



- Major drawback of ripple-carry adder is the **carry propagation**
- The carries are connected in a chain through the full adders
- This is why it is called a **ripple-carry adder**
- The **carry ripples** (propagates) through all the full adders

Longest Delay Analysis



Suppose the XOR delay is Δ_1 and AND-OR delay is Δ_2

For an *N*-bit ripple-carry adder, if all inputs are present at once:

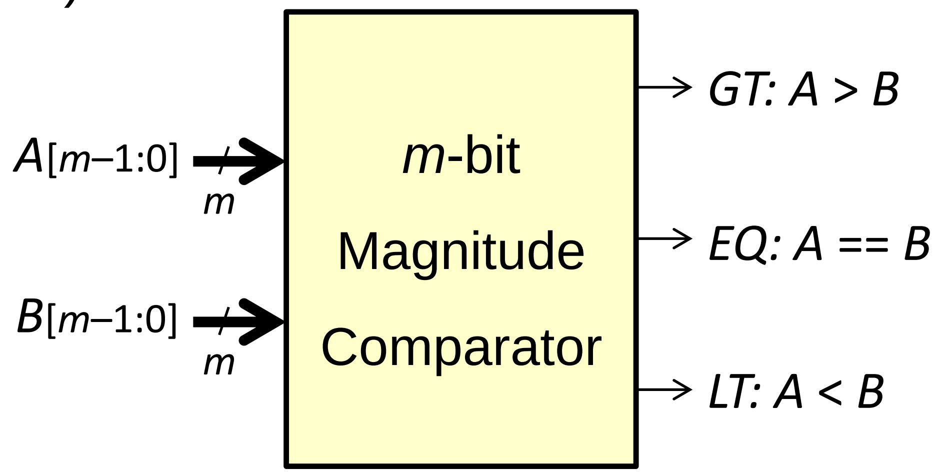
1. Most-significant sum-bit delay = $2\Delta_1 + (N - 1)\ \Delta_2$
2. Final Carry-out delay = $\Delta_1 + N\ \Delta_2$

Next . . .

- Ripple-Carry Adder
- Magnitude Comparator
- Design by Contraction
- Signed Numbers
- Addition/Subtraction of Signed 2's Complement

Magnitude Comparator

- A combinational circuit that compares two unsigned integers
- Two Inputs:
  - Unsigned integer *A* (*m*-bit number)
  - Unsigned integer *B* (*m*-bit number)
- Three outputs:
  - A > B (GT output)
  - A == B (EQ output)
  - A < B (LT output)



- Exactly one of the three outputs must be equal to 1
- While the remaining two outputs must be equal to 0

Example: 4-bit Magnitude Comparator

- Inputs:
  - $A = A_3A_2A_1A_0$
  - $B = B_3B_2B_1B_0$
  - 8 bits in total ➔ 256 possible combinations
  - Not simple to design using conventional K-map techniques
- The magnitude comparator can be designed at a higher level
- Let us implement first the EQ output (A is equal to B)
  - $EQ = 1 \leftrightarrow A_3 == B_3$, $A_2 == B_2$, $A_1 == B_1$, and $A_0 == B_0$
  - Define: $E_i = (A_i == B_i) = A_iB_i + A_i'B_i'$
  - Therefore, $EQ = (A == B) = E_3E_2E_1E_0$

The Greater Than Output

Given the 4-bit input numbers: A and B

1. If $A_3 > B_3$ then $GT = 1$, irrespective of the lower bits of A and B

   Define: $G_3 = A_3 B_3'$ ($A_3 == 1$ and $B_3 == 0$)

2. If $A_3 == B_3$ ($E_3 == 1$), we compare A_2 with B_2

   Define: $G_2 = A_2 B_2'$ ($A_2 == 1$ and $B_2 == 0$)

3. If $A_3 == B_3$ and $A_2 == B_2$, we compare A_1 with B_1

   Define: $G_1 = A_1 B_1'$ ($A_1 == 1$ and $B_1 == 0$)

4. If $A_3 == B_3$ and $A_2 == B_2$ and $A_1 == B_1$, we compare A_0 with B_0

   Define: $G_0 = A_0 B_0'$ ($A_0 == 1$ and $B_0 == 0$)

Therefore, $GT = G_3 + E_3 G_2 + E_3 E_2 G_1 + E_3 E_2 E_1 G_0$

The Less Than Output

We can derive the expression for the LT output, similar to GT

Given the 4-bit input numbers: A and B

1. If $A_3 < B_3$ then $LT = 1$, irrespective of the lower bits of A and B

   Define: $L_3 = A_3'B_3$ $\quad$ ($A_3 == 0$ and $B_3 == 1$)

2. If $A_3 = B_3$ ($E_3 == 1$), we compare A_2 with B_2

   Define: $L_2 = A_2'B_2$ $\quad$ ($A_2 == 0$ and $B_2 == 1$)

3. Define: $L_1 = A_1'B_1$ $\quad$ ($A_1 == 0$ and $B_1 == 1$)

4. Define: $L_0 = A_0'B_0$ $\quad$ ($A_0 == 0$ and $B_0 == 1$)

Therefore, $LT = L_3 + E_3L_2 + E_3E_2L_1 + E_3E_2E_1L_0$

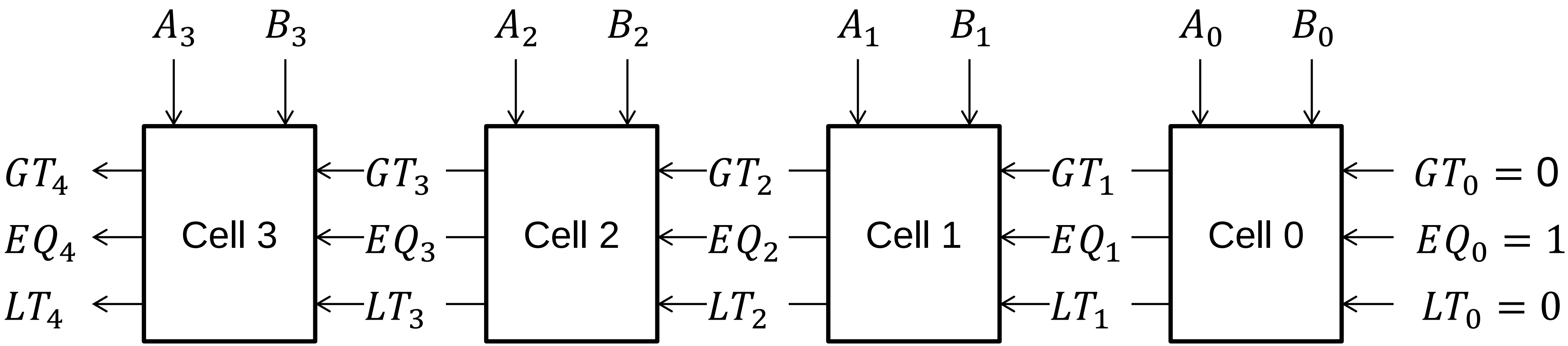
Knowing GT and EQ, we can also derive $LT = (GT + EQ)'$

Iterative Magnitude Comparator Design

- The Magnitude comparator can also be designed iteratively

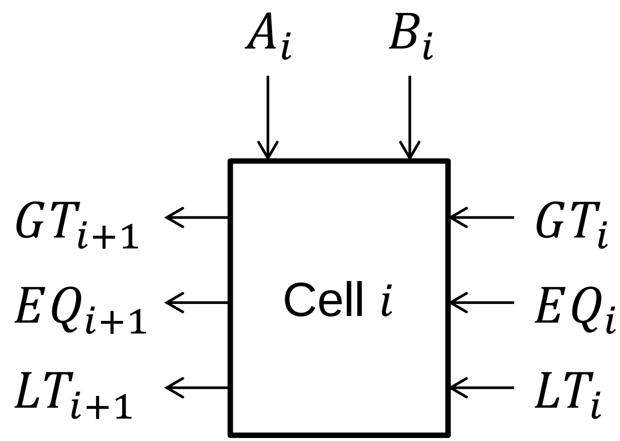
  4-bit magnitude comparator is implemented using 4 identical cells

  Design can be extended to any number of cells
- Comparison starts at least-significant bit
- Final comparator output: $GT = GT_4$, $EQ = EQ_4$, $LT = LT_4$



Cell Implementation

- Each Cell i receives as inputs:
  - Bit i of inputs A and B: A_i and B_i
  - GT_i, EQ_i, and LT_i from cell (i – 1)
- Each Cell i produces three outputs:
  - GT_{i+1}, EQ_{i+1}, and LT_{i+1}
  - Outputs of cell i are inputs to cell $(i + 1)$



- **Output Expressions** of Cell i

$EQ_{i+1} = E_i \, EQ_i$ $\qquad$ $E_i = A_i'B_i' + A_iB_i$ (A_i equals B_i)

$GT_{i+1} = A_i \; B_i' + E_i \, GT_i$ $\qquad$ A_iB_i' ($A_i > B_i$)

$LT_{i+1} = A_i'B_i + E_i \, LT_i$ $\qquad$ $A_i'B_i$ ($A_i < B_i$)

Third output can be produced for first two: $LT = (EQ + GT)'$

Next . . .

- Ripple-Carry Adder
- Magnitude Comparator
- Design by Contraction
- Signed Numbers
- Addition/Subtraction of Signed 2's Complement

Design by Contraction

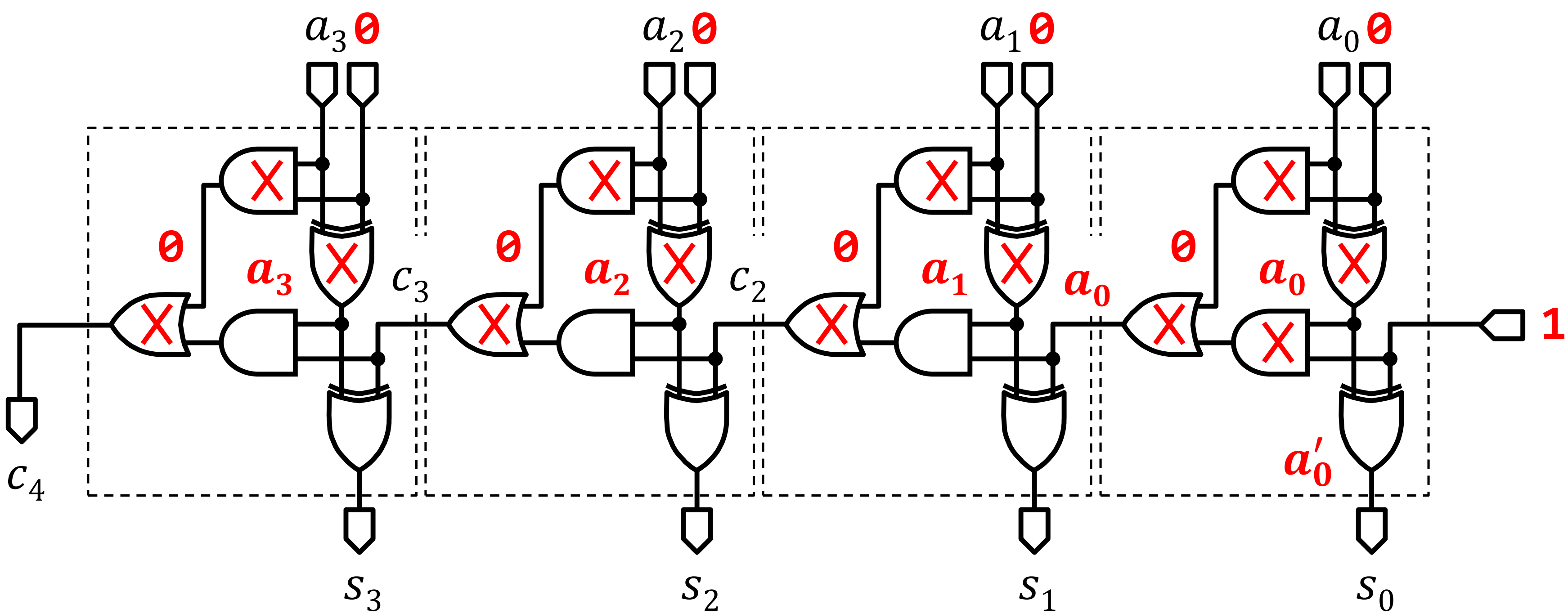
- Contraction is a technique for simplifying the logic
- Applying 0s and 1s to some inputs
- Equations are simplified after applying fixed 0 and 1 inputs
- Converting a function block to a more simplified function
- Examples of Design by Contraction
  - Incrementing a number by a fixed constant
  - Comparing a number to a fixed constant

Designing an Incrementer

- An incrementer is a special case of an adder

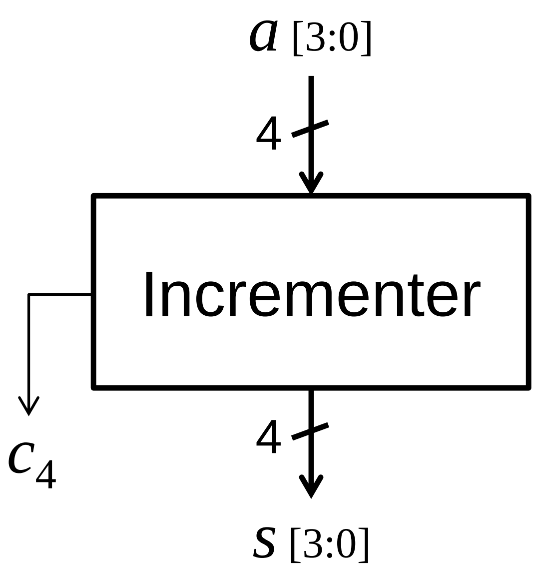
  $Sum = A + 1$ (B = **0**, C_0 = **1**)

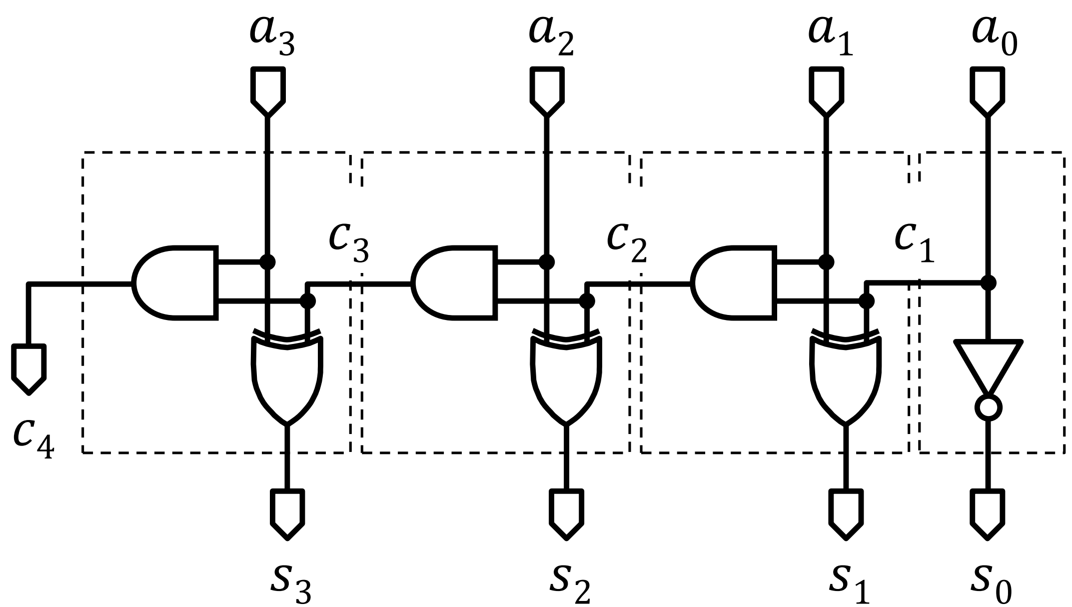
- An n-bit Adder can be simplified into an n-bit Incrementer



Simplifying the Incrementer Circuit

- Many gates were eliminated
- No longer needed when an input is a constant
- Last cell can be replicated to implemented an *n*-bit incrementer





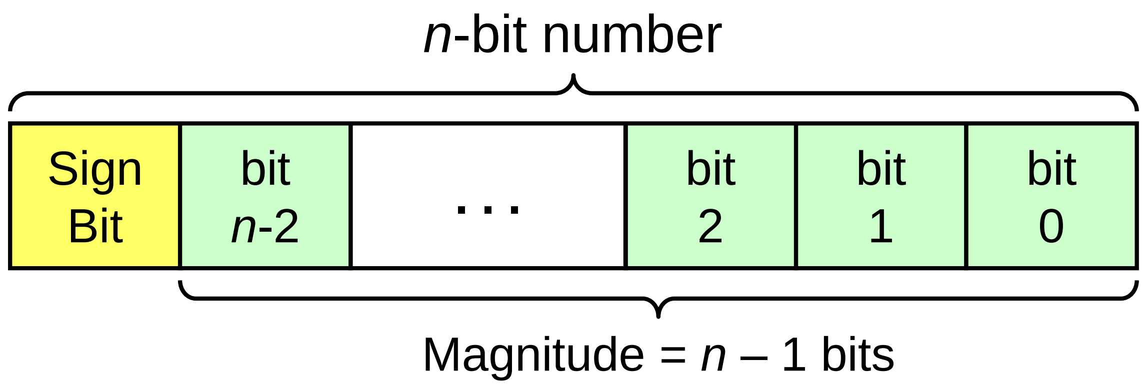
Next . . .

- ❖ Ripple-Carry Adder
- ❖ Magnitude Comparator
- ❖ Design by Contraction
- ❖ Signed Numbers
- ❖ Addition/Subtraction of Signed 2's Complement

Signed Numbers

- Several ways to represent a signed number
  - Sign-Magnitude
  - 1's complement
  - 2's complement
- Divide the range of values into two parts
  - First part corresponds to the positive numbers (≥ 0)
  - Second part correspond to the negative numbers (< 0)
- The 2's complement representation is widely used
  - Has many advantages over other representations

Sign-Magnitude Representation



- Independent representation of the sign and magnitude
- Leftmost bit is the sign bit: 0 is positive and 1 is negative
- Using *n* bits, largest represented magnitude = $2^{n-1} - 1$

Sign-magnitude 8-bit representation of +45

0	0	1	0	1	1	0	1

Sign-magnitude 8-bit representation of -45

1	0	1	0	1	1	0	1

Properties of Sign-Magnitude

- Symmetric range of represented values:

  For n-bit register, range is from $-(2^{n-1} – 1)$ to $+(2^{n-1} – 1)$

  For example, if n = 8 bits then range is -127 to +127

- Two representations for zero: +0 and -0 **NOT Good!**
- Two circuits are needed for addition & subtraction **NOT Good!**
  - In addition to an adder, a second circuit is needed for subtraction
  - Sign and magnitude parts should be processed independently
  - Sign bit should be examined to determine addition or subtraction
  - Addition of numbers of different signs is converted into subtraction
  - Increases the cost of the add/subtract circuit

Sign-Magnitude Addition / Subtraction

Eight cases for Sign-Magnitude Addition / Subtraction

Operation	ADD Magnitudes	Subtract Magnitudes	
		A >= B	A < B
(+A) + (+B)	+(A+B)		
(+A) + (-B)		+(A–B)	-(B–A)
(-A) + (+B)		-(A–B)	+(B–A)
(-A) + (-B)	-(A+B)		
(+A) – (+B)		+(A–B)	-(B–A)
(+A) – (-B)	+(A+B)		
(-A) – (+B)	-(A+B)		
(-A) – (-B)		-(A–B)	+(B–A)

1's Complement Representation

- Given a binary number A

  The 1's complement of A is obtained by inverting each bit in A

- Example: 1's complement of $(01101001)_2 = (10010110)_2$

- If A consists of n bits then:

  A + (1's complement of A) $= (2^n - 1) = (1\ldots111)_2$ (all bits are 1's)

- Range of values is $-(2^{n-1} - 1)$ to $+(2^{n-1} - 1)$

  For example, if $n = 8$ bits, range is -127 to +127

- Two representations for zero: +0 and -0 **NOT Good!**

  1's complement of $(0\ldots000)_2 = (1\ldots111)_2 = 2^n - 1$

  $-0 = (1\ldots111)_2$ **NOT Good!**

2's Complement Representation

- Standard way to represent signed integers in computers
- A simple definition for 2's complement:

  Given a binary number A

  The 2's complement of A = (1's complement of A) + 1
- Example: 2's complement of $(01101001)_2$ =

  $(10010110)_2 + 1 = (10010111)_2$
- If A consists of n bits then

  A + (2's complement of A) = 2^n

  2's complement of $A = 2^n - A$

Computing the 2's Complement

starting value	00100100_2 = +36
step1: Invert the bits (1's complement)	11011011_2
step 2: Add 1 to the value from step 1	+ 1_2
sum = 2's complement representation	11011100_2 = -36

2's complement of 11011100_2 (-36) = 00100011_2 + 1 = 00100100_2 = +36

The 2's complement of the 2's complement of *A* is equal to *A*

Another way to obtain the 2's complement:

Start at the least significant 1

Leave all the 0s to its right unchanged

Complement all the bits to its left

Binary Value

= 00100$\boxed{1}$00 least significant 1

2's Complement

= 11011$\boxed{1}$00

Properties of the 2's Complement

- Range of represented values: -2^{n-1} to $+(2^{n-1} - 1)$

  For example, if n = 8 bits then range is -128 to +127

- There is only **one zero** = $(0…000)_2$ (all bits are zeros)
- The 2's complement of A is the **negative of *A***
- The sum of A + (2's complement of A) **must be zero**

  The final carry is ignored

- Consider the 8-bit number $A = 00101100_2 = +44$

  2's complement of $A = 11010100_2 = -44$

  $00101100_2 + 11010100_2 =$ **1** 00000000_2 (8-bit sum is 0)

  Ignore final carry = 2^8

Values of Different Representations

8-bit Binary Representation	Unsigned Value	Sign Magnitude Value	1's Complement Value	2's Complement Value
00000000	0	+0	+0	0
00000001	1	+1	+1	+1
00000010	2	+2	+2	+2
. . .	. . .	. . .	. . .	. . .
01111101	125	+125	+125	+125
01111110	126	+126	+126	+126
01111111	127	+127	+127	+127
10000000	128	-0	-127	-128
10000001	129	-1	-126	-127
10000010	130	-2	-125	-126
. . .	. . .	. . .	. . .	. . .
11111101	253	-125	-2	-3
11111110	254	-126	-1	-2
11111111	255	-127	-0	-1

2's Complement Signed Value

- Positive numbers (sign-bit = 0)
  - Signed value = Unsigned value
- Negative numbers (sign-bit = 1)
  - Signed value = Unsigned value – 2^n
  - n = number of bits
- Negative weight for sign bit
  - The 2's complement representation assigns a negative weight to the sign bit (most-significant bit)

1	0	1	1	0	1	0	0
-128	64	32	16	8	4	2	1

-128 + 32 + 16 + 4 = -76

8-bit Binary	Unsigned Value	Signed Value
00000000	0	0
00000001	1	+1
00000010	2	+2
. . .	. . .	. . .
01111101	125	+125
01111110	126	+126
01111111	127	+127
10000000	128	-128
10000001	129	-127
10000010	130	-126
. . .	. . .	. . .
11111101	253	-3
11111110	254	-2
11111111	255	-1

Next . . .

- ❖ Ripple-Carry Adder
- ❖ Magnitude Comparator
- ❖ Design by Contraction
- ❖ Signed Numbers
- ❖ Addition/Subtraction of Signed 2's Complement

Converting Subtraction into Addition

- When computing **A – B**, convert **B** to its 2's complement

  A – B = A + (2's complement of B)

- **Same adder** is used for **both addition and subtraction**

  This is the biggest advantage of 2's complement

borrow:	-1 -1 -1	**carry:**	**1** 1 1 1	
	0 1 0 0 1 1 0 1		**0 1 0 0 1 1 0 1**	
–	**0 0 1 1 1 0 1 0**	**+**	**1 1 0 0 0 1 1 0**	(2's complement)
	0 0 0 1 0 0 1 1		**0 0 0 1 0 0 1 1**	(same result)

- Final carry is **ignored**, because

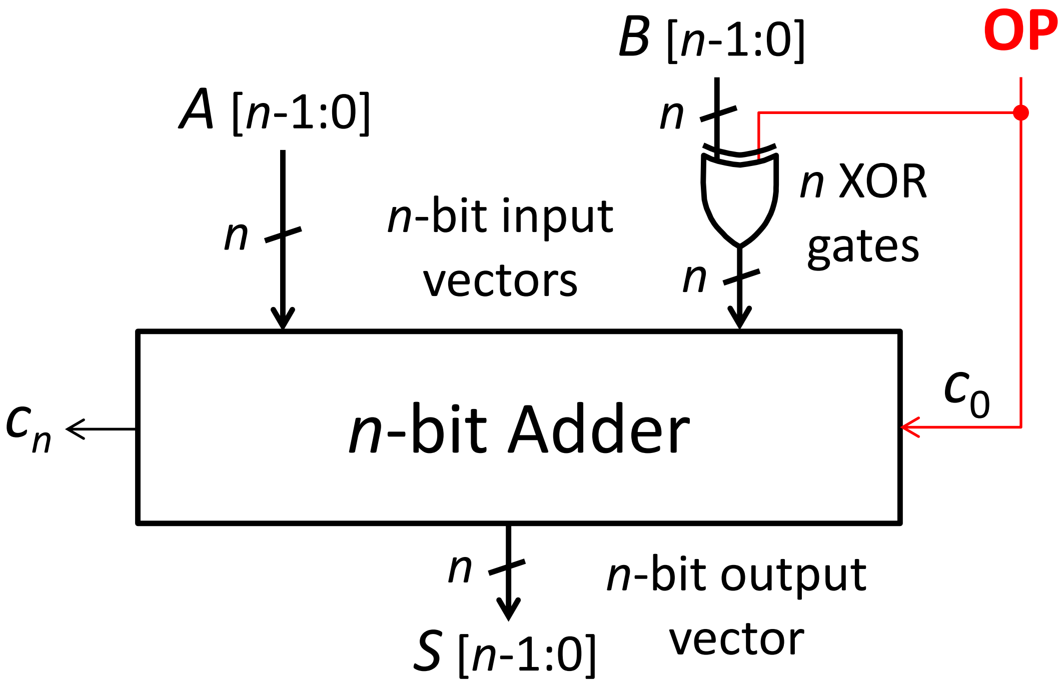
  A + (2's complement of B) = A + (2^n – B) = (A – B) + 2^n

  Final carry = 2^n, for *n*-bit numbers

Adder/Subtractor for 2's Complement

- Same adder is used to compute: (A + B) or (A – B)
- Subtraction (A – B) is computed as: A + (2's complement of B)

  2's complement of B = (1's complement of B) + 1
- Two operations: **OP = 0 (ADD)**, **OP = 1 (SUBTRACT)**



OP = 0 (ADD)

B XOR 0 = B

S = A + B + 0 = A + B

OP = 1 (SUBTRACT)

B XOR 1 = 1's complement of B

S = A + (1's complement of B) + 1

S = A + (2's complement of B)

S = A – B

Carry versus Overflow

- Carry is important when …
  - Adding **unsigned integers**
  - Indicates that the **unsigned sum** is out of range
  - Sum > maximum unsigned *n*-bit value
- Overflow is important when …
  - Adding or subtracting **signed integers**
  - Indicates that the **signed sum** is out of range
- Overflow occurs when …
  - Adding two positive numbers and the sum is negative
  - Adding two negative numbers and the sum is positive
- Simplest way to detect Overflow: $V = C_{n-1} \oplus C_n$
  - C_{n-1} and C_n are the carry-in and carry-out of the most-significant bit

Carry and Overflow Examples

❖ We can have carry without overflow and vice-versa

❖ Four cases are possible (Examples on 8-bit numbers)

					1					
		0	0	0	0	1	1	1	1	15
+		0	0	0	0	1	0	0	0	8
		0	0	0	1	0	1	1	1	23

Carry = 0 Overflow = 0

	1	1	1	1	1					
		0	0	0	0	1	1	1	1	15
+		1	1	1	1	1	0	0	0	248 (-8)
		0	0	0	0	0	1	1	1	7

Carry = 1 Overflow = 0

		1								
		0	1	0	0	1	1	1	1	79
+		0	1	0	0	0	0	0	0	64
		1	0	0	0	1	1	1	1	143 (-113)

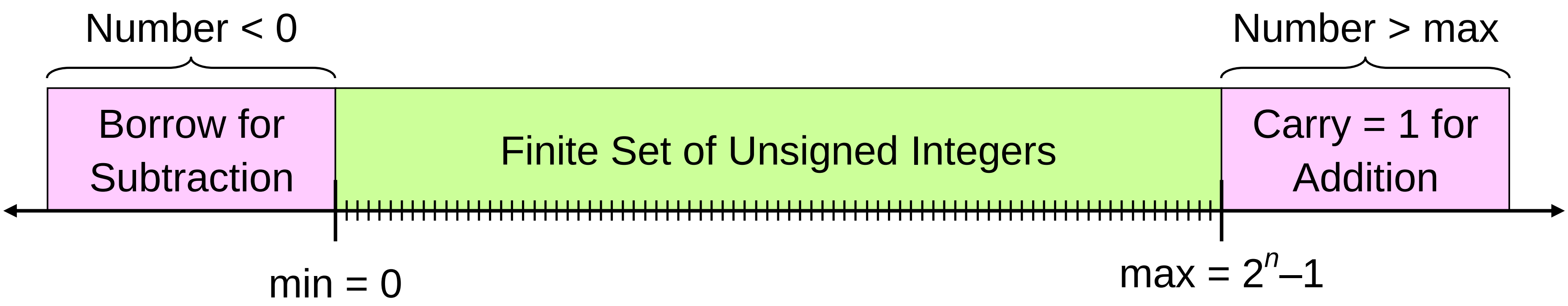
Carry = 0 Overflow = 1

	1			1	1					
		1	1	0	1	1	0	1	0	218 (-38)
+		1	0	0	1	1	1	0	1	157 (-99)
		0	1	1	1	0	1	1	1	119

Carry = 1 Overflow = 1

Range, Carry, Borrow, and Overflow

- Unsigned Integers: *n*-bit representation



- Signed Integers: 2's complement representation