WEEK 10-2017

Arithmetic Circuits

Iterative Combinational Circuits

- **Arithmetic circuits often designed to operate on binary input vectors and produce binary output vectors.**

1011010
10000001

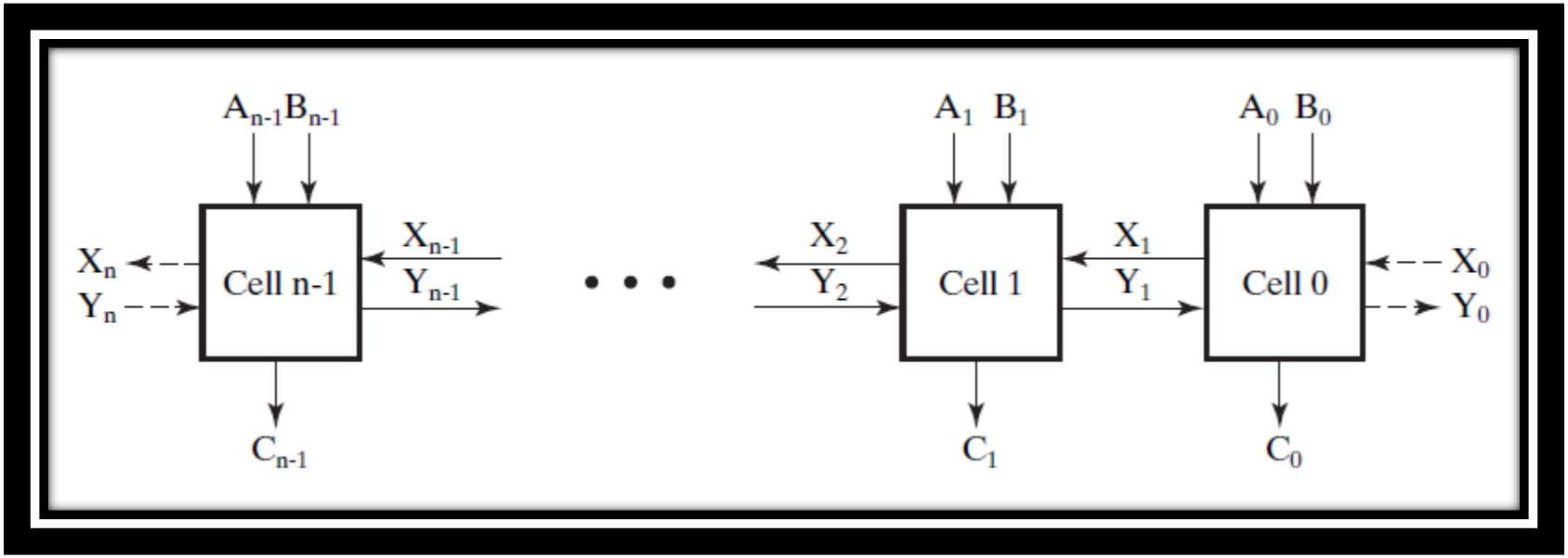
- **The function implemented requires that the same subfunction be applied to each bit position**

- **Can design functional block for the subfunction and duplicate it to obtain functional block for an overall function**

- **Each subfunction block is referred to as a *cell***

- **An array of interconnected cells forms an *iterative array***

Iterative Combinational Circuits



- **Example: *n* = 32 means 64 inputs and 32 outputs**
- **2^{64} truth table rows – impractical!**
- **Iterative array takes advantage of the regularity to make design feasible**

Binary Adders

Half Full

- **Binary addition is used frequently**
- **Similar to decimal addition but with two possible digits: 0 and 1**
- **Example – 8-bit addition:**

$B_1 + B_2 + C$

0 1 3
2 3 Two bits

0 1 0 0 1 1 0 1
0 1 0 1 0 1 1 0
1 0 1 0 0 0 1 1

0 1 0 1 1 1 0 0 0 ←
0 1 0 0 1 1 0 1
+ 0 1 0 1 0 1 1 0
1 0 1 0 0 0 1 1 ←

1 1 0
7 7
+ 8 6
1 6 3

Half Adder

- **A *Half Adder* generates the sum of two binary digits:**

X	0	1	0	1
+ Y	+ 0	+ 0	+ 1	+ 1
C S	0 0	0 1	0 1	1 0

- **The sum is expressed as a sum-bit *S*, and a carry-bit *C***

- **The half adder can be specified as a truth table and the output equations can be derived**

X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

S:

Y \ X	0	1
0	0	1
1	1	0

$S = \overline{X}Y + X\overline{Y} = X \oplus Y$

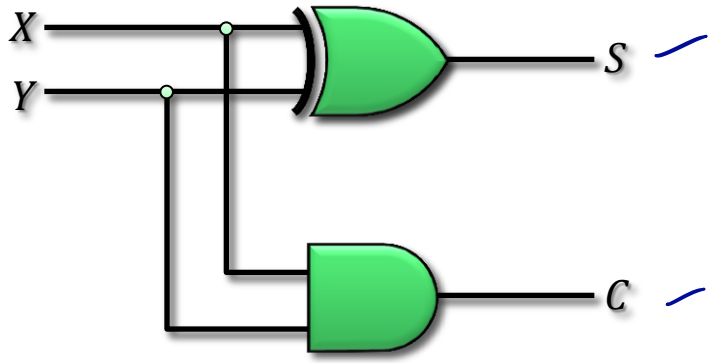
C:

Y \ X	0	1
0	0	0
1	0	1

$C = XY$

Half Adder

- **The logic diagram for the half adder:**



$S = X \oplus Y$

$C = XY$

X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

S:

Y \ X	0	1
0	0	1
1	1	0

$S = \overline{X}Y + X\overline{Y} = X \oplus Y$

C:

Y \ X	0	1
0	0	0
1	0	1

$C = XY$

Full Adder

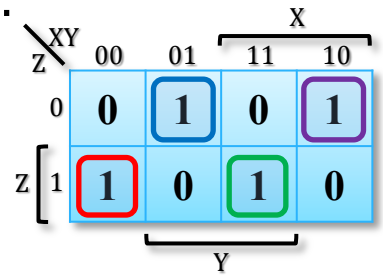
- **A Full Adder can be specified as a truth table and the output equations can be derived**

	Z		0		0		0		0		1		1		1		1
	X		0		0		1		1		0		0		1		1
+	Y	+	0	+	1	+	0	+	1	+	0	+	1	+	0	+	1
C	S	0	0	0	1	0	1	1	0	0	1	1	0	1	0	1	1

- **The full adder truth table and K-maps:**

X	Y	Z	C	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

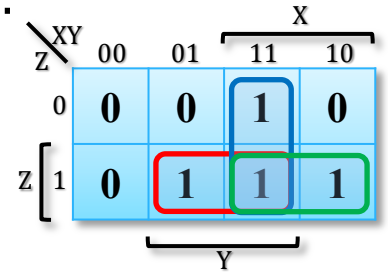
S:



$$S = \overline{X}\overline{Y}Z + \overline{X}Y\overline{Z} + X\overline{Y}\overline{Z} + XYZ$$

$$= X \oplus Y \oplus Z = (X \oplus Y) \oplus Z$$

C:



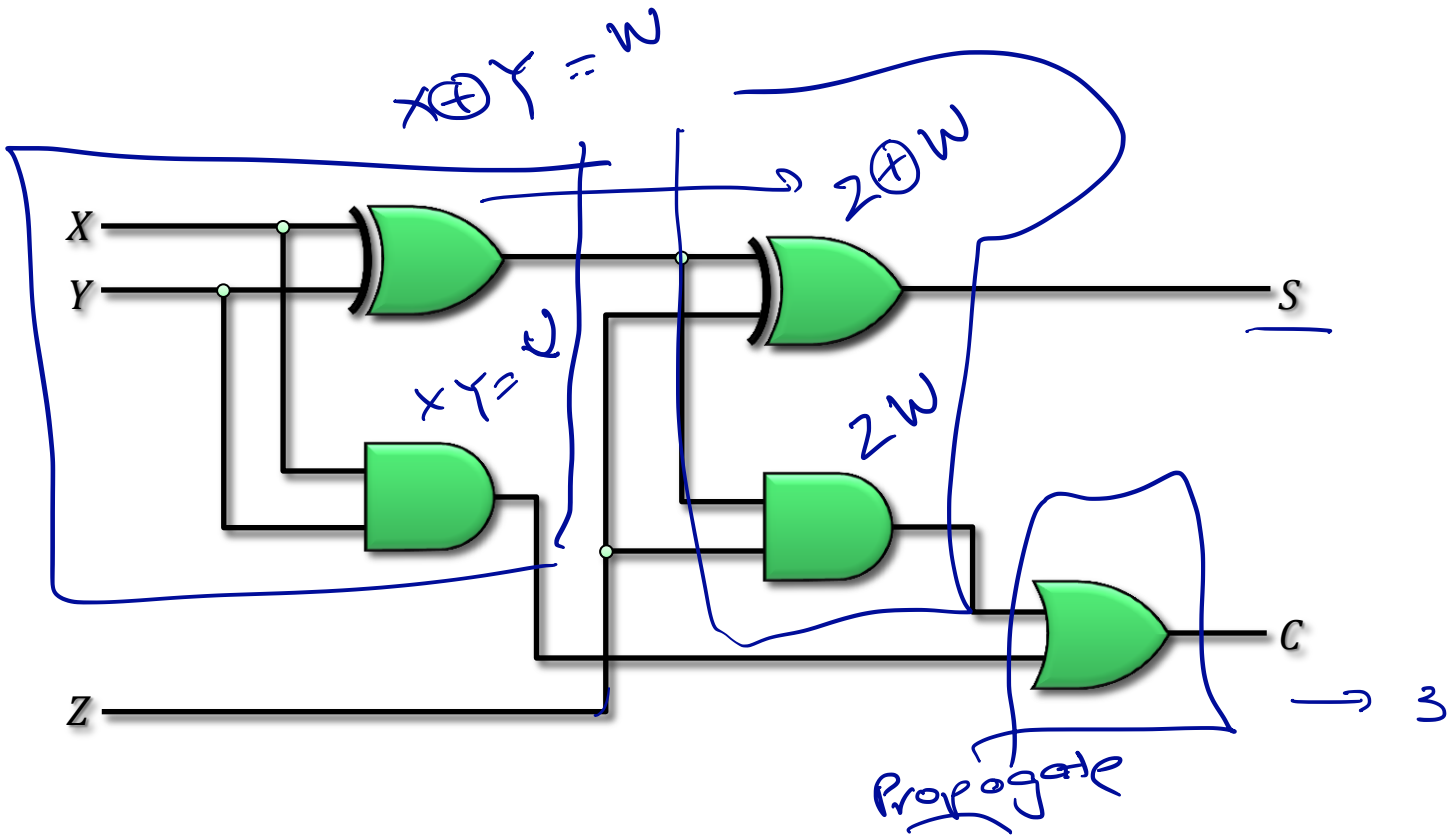
$$C = XY + XZ + YZ$$

$$= XY + Z(X + Y)$$

$$= XY + Z(XY + X \oplus Y)$$

$$= XY + Z(X \oplus Y)$$

Full Adder logic diagram

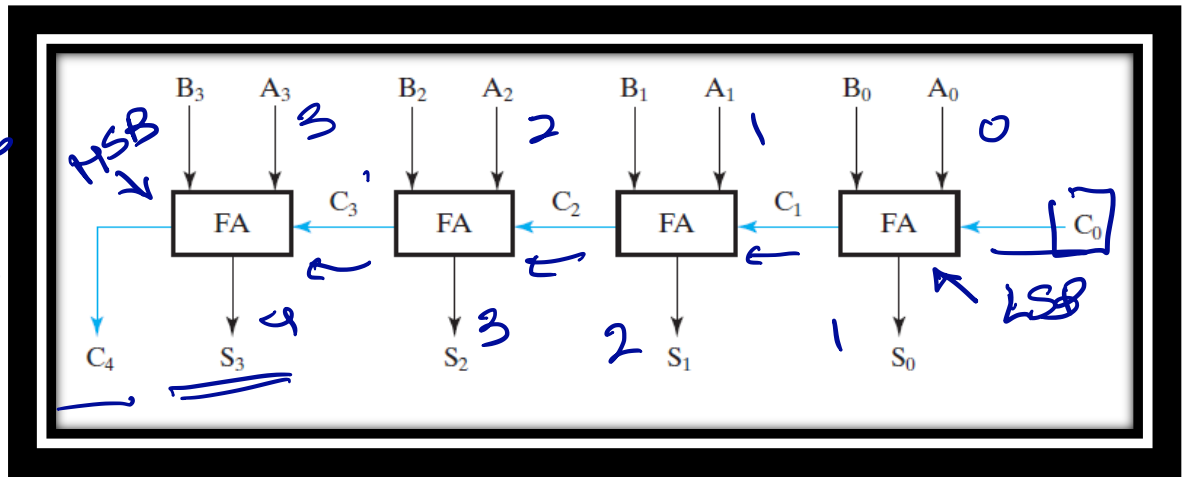


$$S = (X \oplus Y) \oplus Z$$

$$C = XY + Z(X \oplus Y)$$

Binary Ripple-Carry Adder

- An *n*-bit parallel adder formed by cascading *n* full adders
- Apply all *n*-bit inputs simultaneously
- Connect the carry output from one full adder to the carry input of the next
- The carry propagate through, from the LSB to the MSB



- Example:

C_i	0	0	1	1	0
A_i		1	0	1	1
B_i	+	0	0	1	1
S_i		1	1	1	0
C_{i+1}		0	0	1	1

Binary Ripple-Carry Adder

- Ripple-carry adders have long circuit delays
- The carry has to propagate through many gates until the final result is obtained
- Each of the *n* full-adders introduce two gate-delays in the carry path
- Example: for a 16-bit adder, the delay is 32 gate delays (+ overhead)

Binary Carry Lookahead Adder

- **Define: Carry *Generate***

$$G_i = A_i B_i$$

- **Must generate carry when $A = B = 1$**
- **Define: Carry *Propagate***

$$P_i = A_i \oplus B_i$$

- **The carry-out will equal carry-in**
- **Express the sum (S) and carry (C) in terms of generate/propagate:**

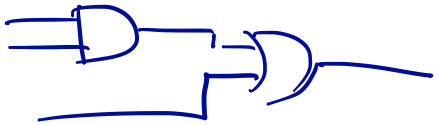
$$S_i = (A_i \oplus B_i) \oplus C_i$$
$$= P_i \oplus C_i$$
$$C_{i+1} = A_i B_i + C_i (A_i \oplus B_i)$$
$$= G_i + C_i P_i$$

C_i	0	1	0	0	0
A_i		0	1	0	0
B_i	+	0	1	0	0
S_i		1	0	0	0
C_{i+1}		0	1	0	0

C_i	0	1	1	0	0
A_i		0	0	1	0
B_i	+	0	1	1	0
S_i		1	0	0	0
C_{i+1}		0	1	1	0

Binary Carry Lookahead Adder

- **Re-express the carry equations:**



$$C_1 = G_0 + P_0C_0$$

$$C_2 = G_1 + P_1C_1 = G_1 + P_1G_0 + P_1P_0C_0$$

$$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$$

$$C_4 = G_3 + P_3C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$$

Binary Carry Lookahead Adder

- All carry bits can be calculated straight from the inputs
- In theory, 3 gate-delays regardless of number of bits, at the cost of more complex logic
- In practice, gates have limited number of inputs
- For larger adders, cascade 4-bit CLAs and connect to a group lookahead carry unit

Unsigned Binary Subtraction

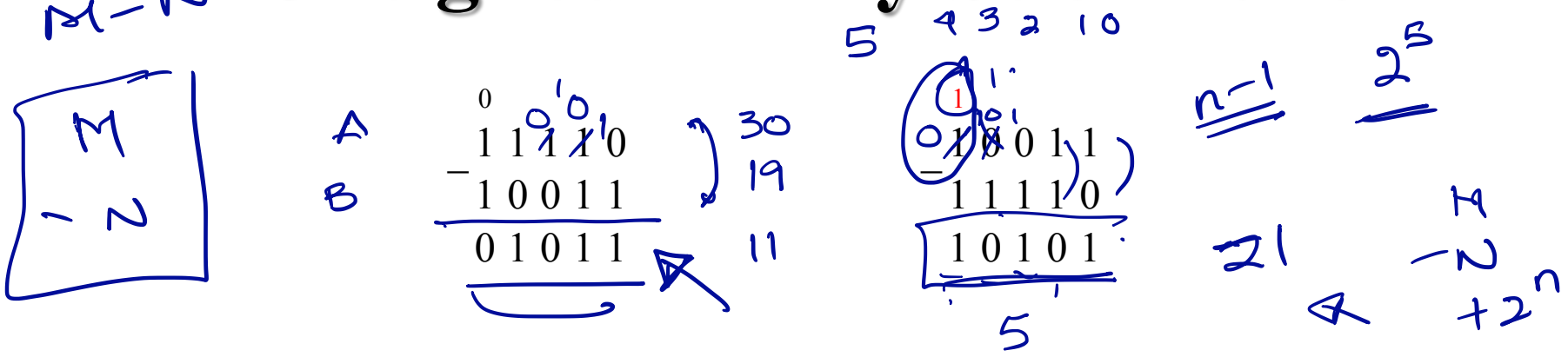
- **Subtraction involves comparing the subtrahend with the minuend and subtracting the smaller from the larger. If the minuend is larger than the subtrahend , then the result is a positive number; Otherwise, it will be negative.**

- **Subtraction with comparison is inefficient as comparison operation results in costly circuitry.**

- **As an alternative, we can simply subtract the subtrahend from the minuend and correct the result if negative.**

- **Consider the next example:**

Unsigned Binary Subtraction



- **If no borrow occurs into the most significant position, then we know that the subtrahend is not larger than the minuend and the result is positive and correct.**

- **If a borrow does occur into the most significant position, then we know the subtrahend is larger than the minuend. The result must then be negative, and so needs correction.**

- **How?**

Unsigned Binary Subtraction

- **The correct result when a borrow occurs**

$$M - N + 2^n$$

- **Instead the result we desire is N-M in magnitude. This correct result can be obtained by**

$$2^n - (M - N + 2^n) = \mathrm{N} - \mathrm{M}$$

- **In the previous example, the correct magnitude is 100000 – 10101 = 01011.**

Unsigned Binary Subtraction

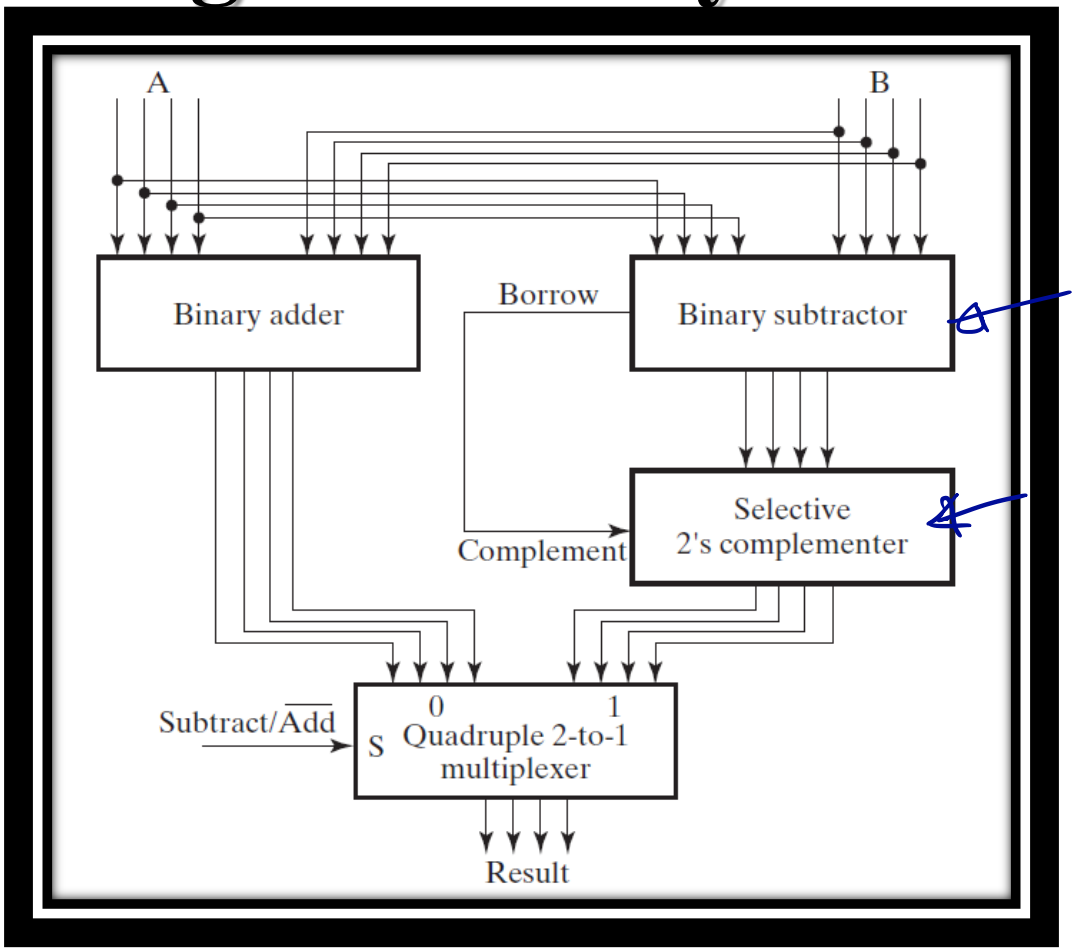
- In general, the subtraction of two n-digit numbers, $M - N$, in base 2 can be done as follows:

1. Subtract the subtrahend N from the minuend M.
2. If no end borrow occurs, then $M \geq N$, and the result is nonnegative and correct.
3. If an end borrow occurs, then $N > M$, and the difference, $M - N + 2^n$, is subtracted from 2^n, and a minus sign is appended to the result.

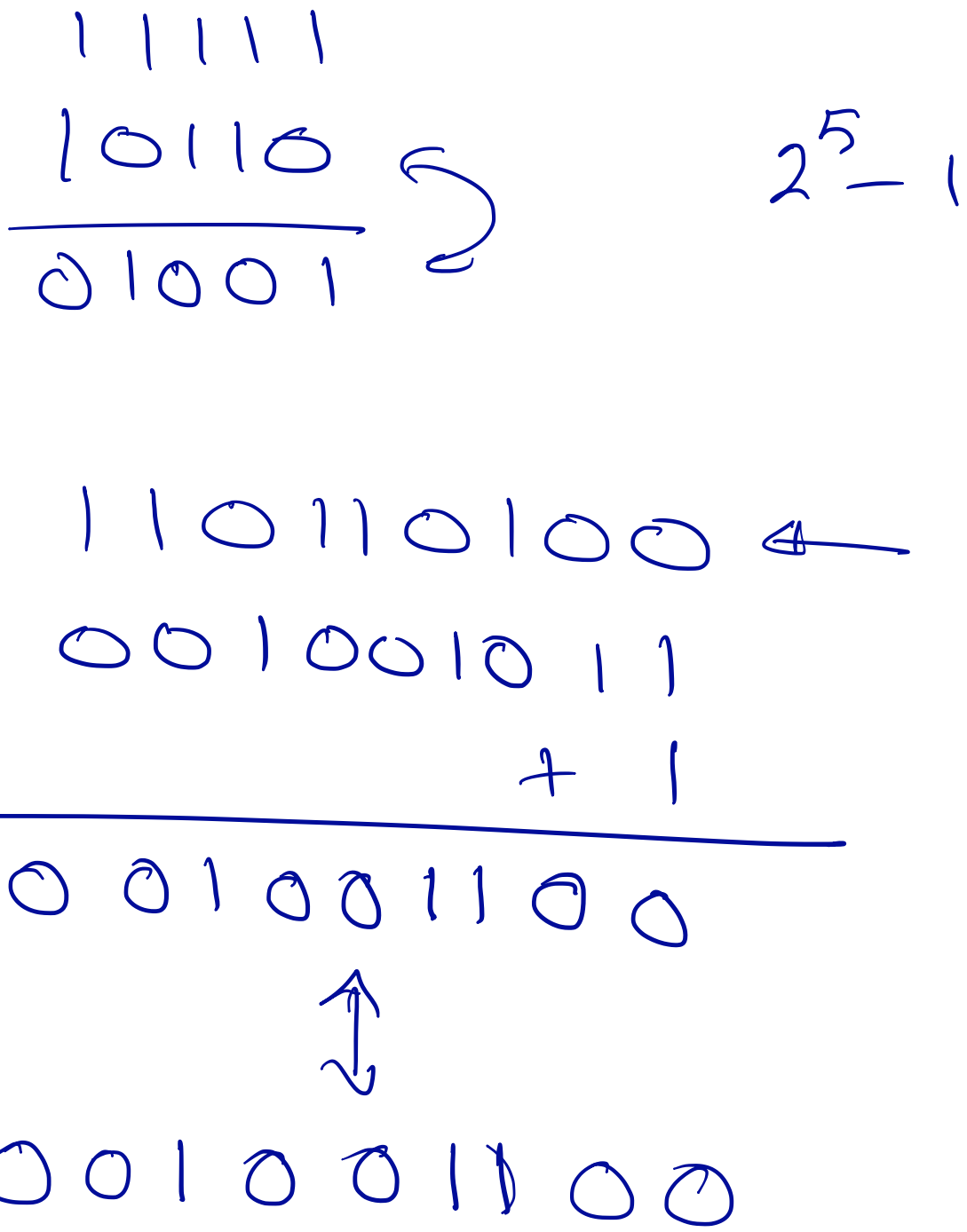
- The subtraction $2^n - N$ is called taking the 2s complement of N

- To do both unsigned addition and unsigned subtraction requires a complex circuit (next slide)

Unsigned Binary Subtraction



- **Would like to find shared simpler logic for both addition and subtraction**
- **Introduce *complements* as an approach**


1 1 1 1 1
1 0 1 1 0
0 1 0 0 1
$2^5 - 1$
1 1 0 1 1 0 1 0 0
1's 0 0 1 0 0 1 0 1 1
+ 1
2's 0 0 1 0 0 1 1 0 0
0 0 1 0 0 1 1 0 0

Complements

- For any radix *r*, the *Diminished Radix Complement* – called **(*r* – 1)'s complement** for radix *r* – is defined as: $(r^n - 1) - N$, for some number *N*
- The *Radix Complement* – called ***r*'s complement** for radix *r* – is defined as $r^n - N$, for some number *N*
- Example:

$$r = 2, N = 0111\ 0011, n = 8$$

$$r^n - 1 = 100000000 - 1 = 11111111$$

- So the **1s complement** of *N*:

$$\begin{array}{r} 11111111 \\ -\ 01110011 \\ \hline 10001100 \end{array}$$

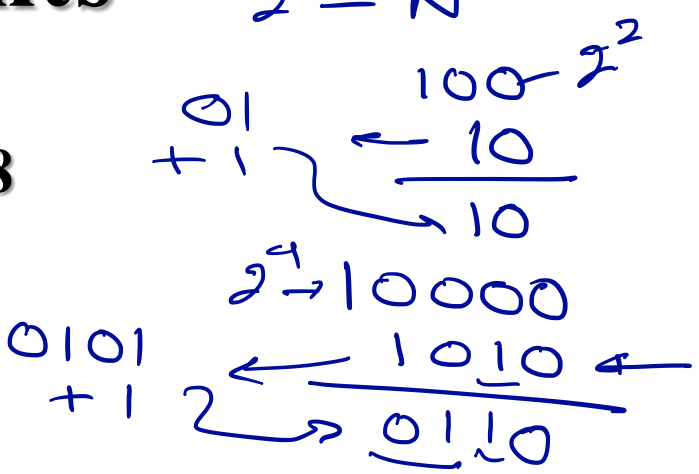
- Easily obtained by inverting all the bits in *N*

Complements

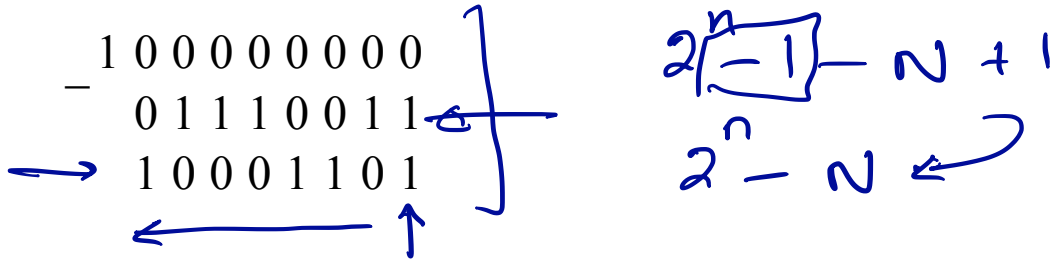
- **Example:**

$$r = 2, N = 0111\ 0011, n = 8$$

$$r^n = 100000000$$



- **So the 2s complement of *N*:**



- **Note the result is the 1s complement plus 1, a fact that can be used in designing hardware**

- **By using complements, it is possible to simplify hardware by sharing adder and subtractor logic**

→ 01100100 | 10010110

10011011 | 01101010

1's comp | 2's comp

Complements

- **The algorithm for subtraction of two unsigned binary numbers using addition**

1. **Add the 2s complement of the subtrahend N to the minuend M. This performs $M + (2^n - N) = M - N + 2^n$.**
2. **If $M \geq N$, the sum produces an end carry, 2^n. Discard the end carry, leaving result $M - N$.**
3. **If $M < N$, the sum does not produce an end carry, since it is equal to $2^n - (N - M)$, the 2s complement of $N - M$. Perform a correction, taking the 2s complement of the sum and placing a minus sign in front to obtain the result $-(N - M)$.**

Complements

- **Example:**

  Given *X* = 1010100 and *Y* = 1000011, perform the subtraction *X* – *Y* and *Y* – *X*:

$$
\begin{array}{rr}
 & 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0 \\
X = & 1\ 0\ 1\ 0\ 1\ 0\ 0 \\
+ & \\
\text{2s complement of } Y = & 0\ 1\ 1\ 1\ 1\ 0\ 1 \\
\text{Sum} = & 0\ 0\ 1\ 0\ 0\ 0\ 1
\end{array}
$$

$$
\begin{array}{rr}
 & 0\ 0\ 0\ 0\ 0\ 0\ 0 \\
Y = & 1\ 0\ 0\ 0\ 0\ 1\ 1 \\
+ & \\
\text{2s complement of } X = & 0\ 1\ 0\ 1\ 1\ 0\ 0 \\
\text{Sum} = & 1\ 1\ 0\ 1\ 1\ 1\ 1
\end{array}
$$

$Y - X$ = –(2s complement of 1101111)

= – 0010001

Signed Binary Numbers

10110111

- So far we dealt with unsigned binary numbers
- Positive numbers and zero were represented in the "usual" way
- Negative numbers were identified by adding a minus sign on paper, but no digital implementation
- Need a way to represent negative numbers in hardware
- Define the most significant bit to be the *sign* bit
- Use 0 for positive numbers and 1 for negative numbers
- Possible ways to represent integers:
  Signed-Magnitude or *Signed-Complement*

Signed-Magnitude

- The MSB stands for the sign (0 for +, 1 for –)
- The rest of the bits are interpreted as a positive magnitude
- 8-bit example:

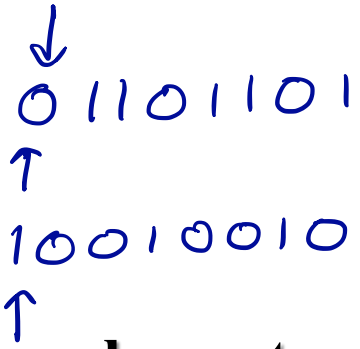
$$00011001_2 = +25_{10}$$

$$10100101_2 = -37_{10}$$

- Arithmetic using signed-magnitude is complex
- It requires checking of sign bits and choosing between addition or subtraction
- The carry-out or borrow determines whether a correction step should be applied to the result

Signed- complement

- **Negative numbers are the complements of the respective positive ones**
- **Two possibilities:**



1. **Signed 1s complement – uses 1s complement arithmetic**
2. **Signed 2s complement – uses 2s complement arithmetic**

- **Both will make the MSB correspond to the sign**
- **With 2s complement arithmetic we can use the same hardware as for unsigned numbers alone**

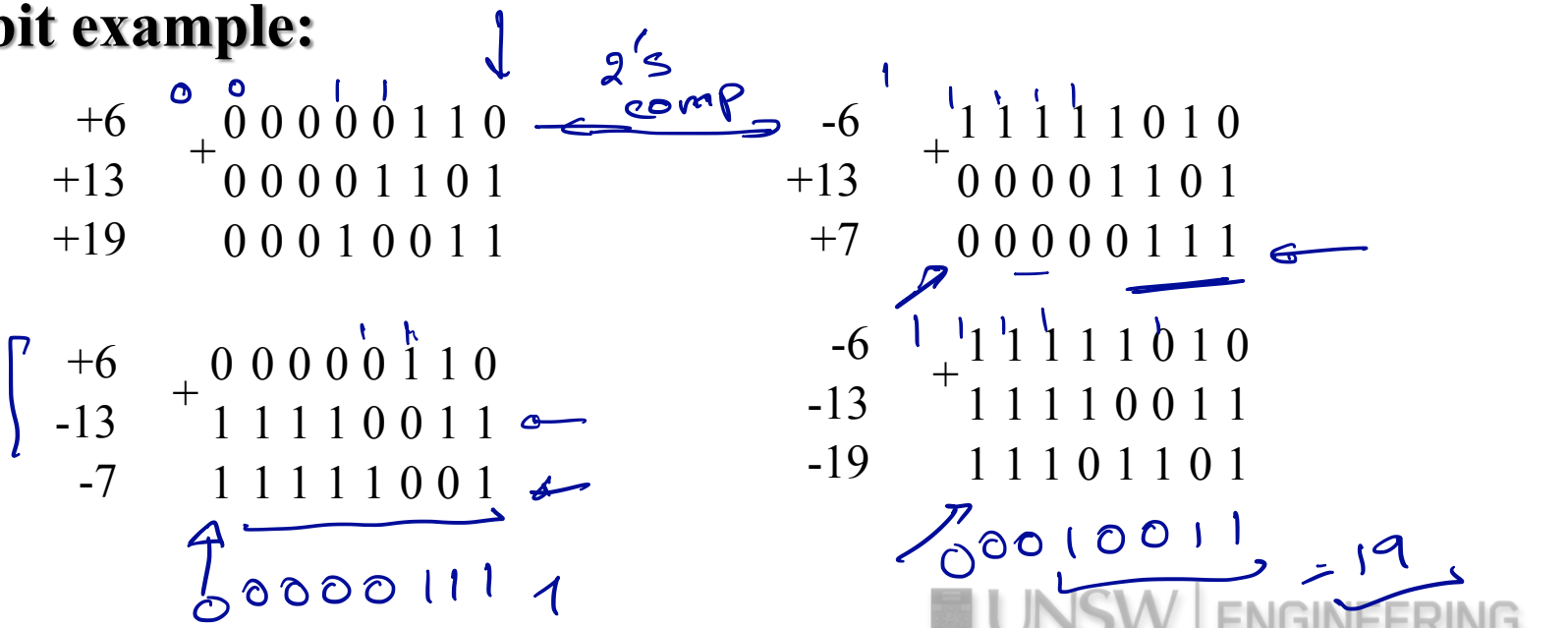
Signed- complement

- **Example – 3-bit numbers:**

Number	Sign-mag.	1s Comp.	2s Comp.
+3	011	011	011
+2	010	010	010
+1	001	001	001
+0	000	000	000
-0	100	111	–
-1	101	110	111
-2	110	101	110
-3	111	100	101
-4	–	–	100

2s Complement Arithmetic

- **2s complement arithmetic does not require special hardware – can use simple adders**
- **The addition of two signed binary numbers with negative numbers represented in signed 2s complement form is obtained from the addition of the two numbers**
- **A carry-out of the sign bit position is discarded**
- **8 bit example:**



2s Complement Arithmetic

- To convert negative binary numbers to decimal, use one of two methods:

1. Take the 2s complement of the negative number to obtain the corresponding positive one, and add a minus sign at the front
2. Convert as usual but subtract the MSB's value instead of adding

- Example: for the 8-bit number 1110 1101:

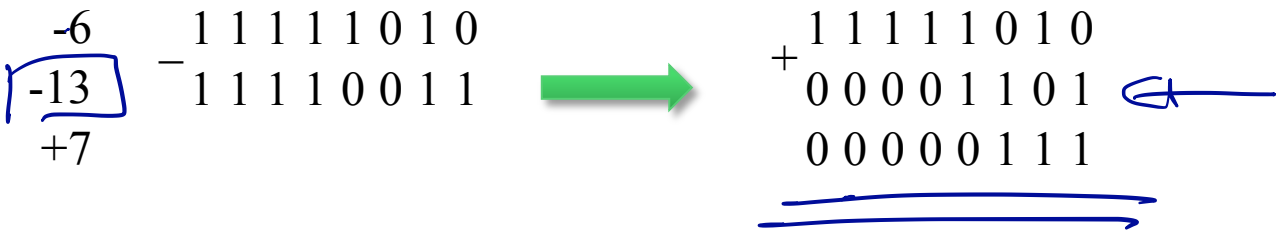
1. 2s complement: $0001\ 0011_2 = 19_{10}$

   So $1110\ 1101_2 = -19_{10}$

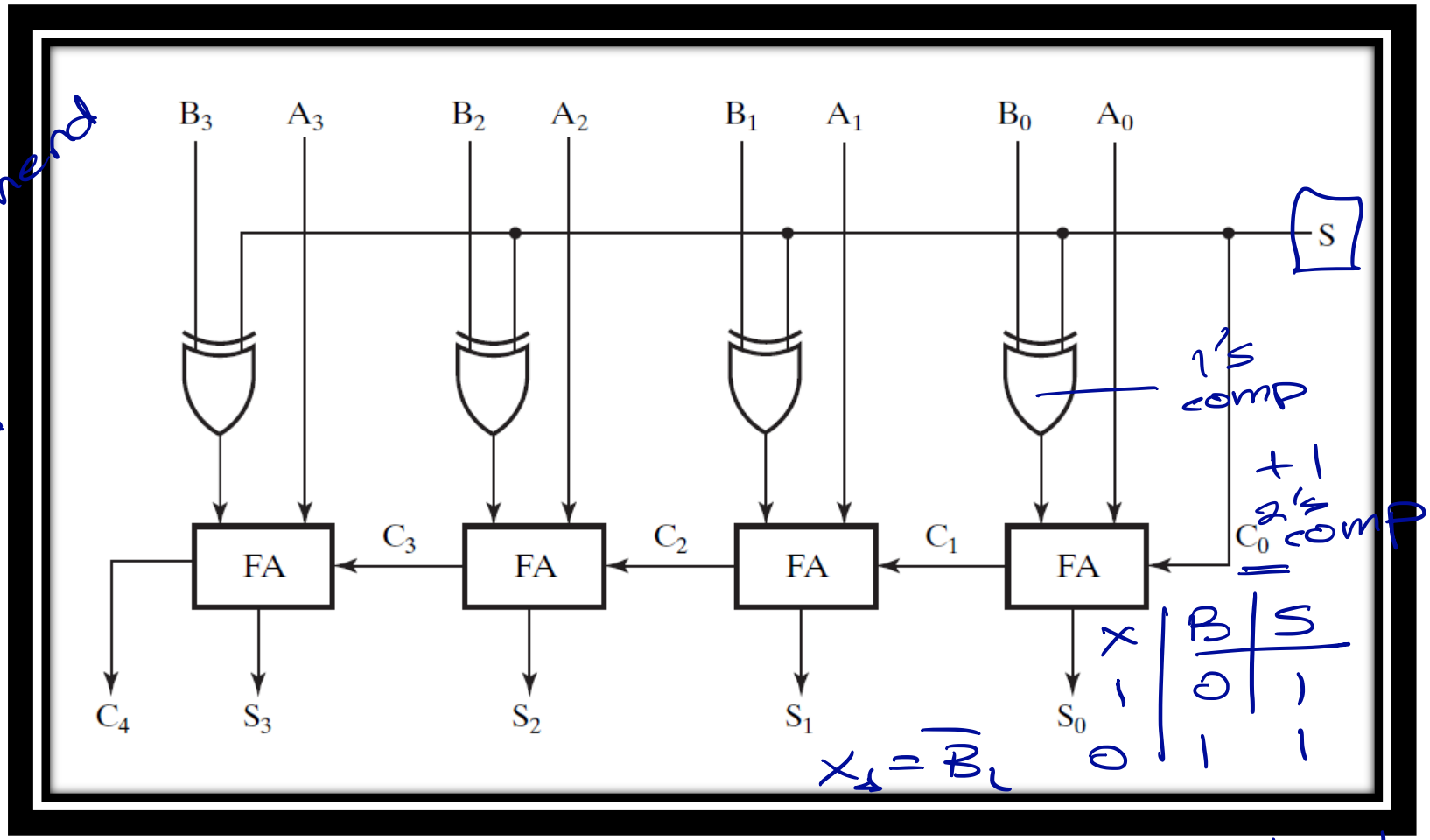
2. $1110\ 1101_2 = 1 + 4 + 8 + 32 + 64 - 128 = -19_{10}$

2s Complement Subtraction

- **Subtraction with 2s complement is simple**
- **Take the 2s complement of the subtrahend and add it to the minuend**
- **A carry-out of the sign bit position is discarded**
- **Works because: $A - B = A + (-B)$**
- **8-bit example:**



2s Complement Adder/Subtractor



B - subtrahend

A - minuend

1's comp

+1 2's comp

X	B	S
1	0	1
0	1	1

$X_i = \overline{B_i}$

S = 0 Add

S = 1 subtraction

$S \oplus B_i$ — $0 \oplus B_i$

S = 0

$X = B_i$

B	S
0	0
1	1

2s Complement Adder/Subtractor

- Signal *S* selects between addition (*S* = 0) or subtraction (*S* = 1)
- If *S* = 0, vector *B* propagate through the XOR gates and the carry-in is 0 – computes *A* + *B*
- If *S* = 1, vector *B* is complemented and the carry-in is 1 to obtain the 2s complement of *B* – computes *A* – *B*

Sign Extension

- To represent an *n*-bit number with larger number of bits (*n* + *m*):

111 +7
00000111 +7

- For unsigned numbers add *m* 0's at the front

- For signed numbers, extend the MSB at the front of the number

- 4 to 8 bits signed example:

  +7 = 0111 = 0000 0111

  -7 = 1001 = 1111 1001

Overflow

- ***Overflow*** **occurs if (*n* + 1) bits are required to contain the result from an *n*-bit addition or subtraction**
- **Example: 8-bits can represent values between -128 to +127. Calculate 70 + 80:**

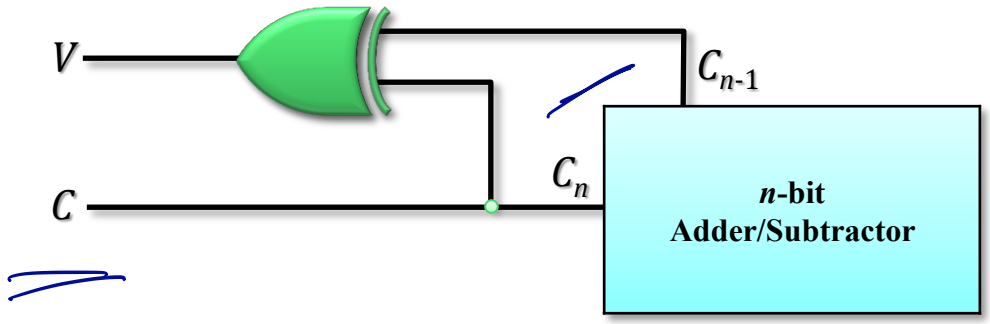
```
  0 1 0 0 0 1 1 0
+ 0 1 0 1 0 0 0 0
  1 0 0 1 0 1 1 0
```

- **Negative number! (MSB = 1)**
- **For *unsigned* addition:**
  - ***C*** **= 0 → No Overflow**
  - ***C*** **= 1 → Overflow**
- **For *signed* addition (and subtraction):**
  - ***V*** **= 0 → No Overflow**
  - ***V*** **= 1 → Overflow**

Overflow

- **Overflow occurs when the carry-in to the MSB is not equal to the carry-out from the MSB**

- **A simple circuit to detect overflow:**



N, Z, C, V – Status Flags

- **Status signals that provide information from the arithmetic unit in microprocessors**

- ***N*** **(*Negative*): 1 if the sum is negative; 0 if positive**
- ***Z*** **(*Zero*): 1 if the sum is zero; 0 if non-zero**
- ***C*** **(*Carry*): 1 if there is a carry-out; 0 if no carry**
- ***V*** **(*oVerflow*): 1 if signed overflow detected; 0 for no-overflow**

- **Can be used to compare numbers**

N, Z, C, V – Status Flags

- Status signals that provide information from the arithmetic unit in microprocessors

- *N* (*Negative*): 1 if the sum is negative; 0 if positive
- *Z* (*Zero*): 1 if the sum is zero; 0 if non-zero
- *C* (*Carry*): 1 if there is a carry-out; 0 if no carry
- *V* (*oVerflow*): 1 if signed overflow detected;
  0 for no-overflow

- Can be used to compare numbers

N, Z, C, V – Status Flags

- **These can be implemented by:**

$N = S_{n-1}$ **(value at MSB of sum)**

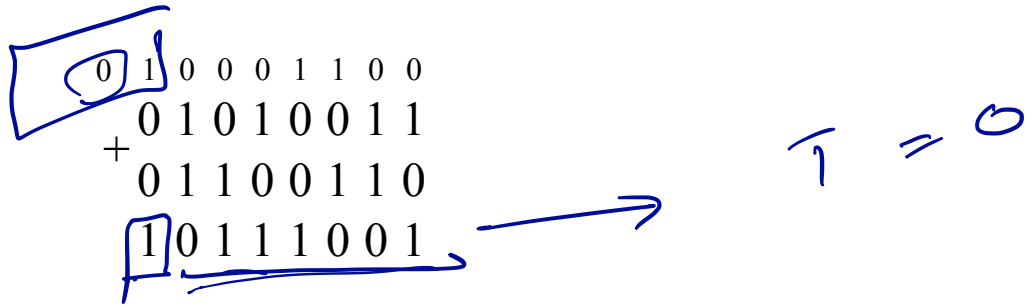
$Z = \overline{S_{n-1} + S_{n-2} + \ldots + S_0}$ **(detect if all sum bits are 0)**

$C = C_n$ **(carry out from MSB)**

$V = C_n \oplus C_{n-1}$ **(MSB carry-in ≠ carry-out)**

N, Z, C, V – Status Flags

- **Determine the value of the N, Z, C, V flags:**



$N = 1$

$Z = 0$

$C = 0$

$V = 1$