Lecture 11

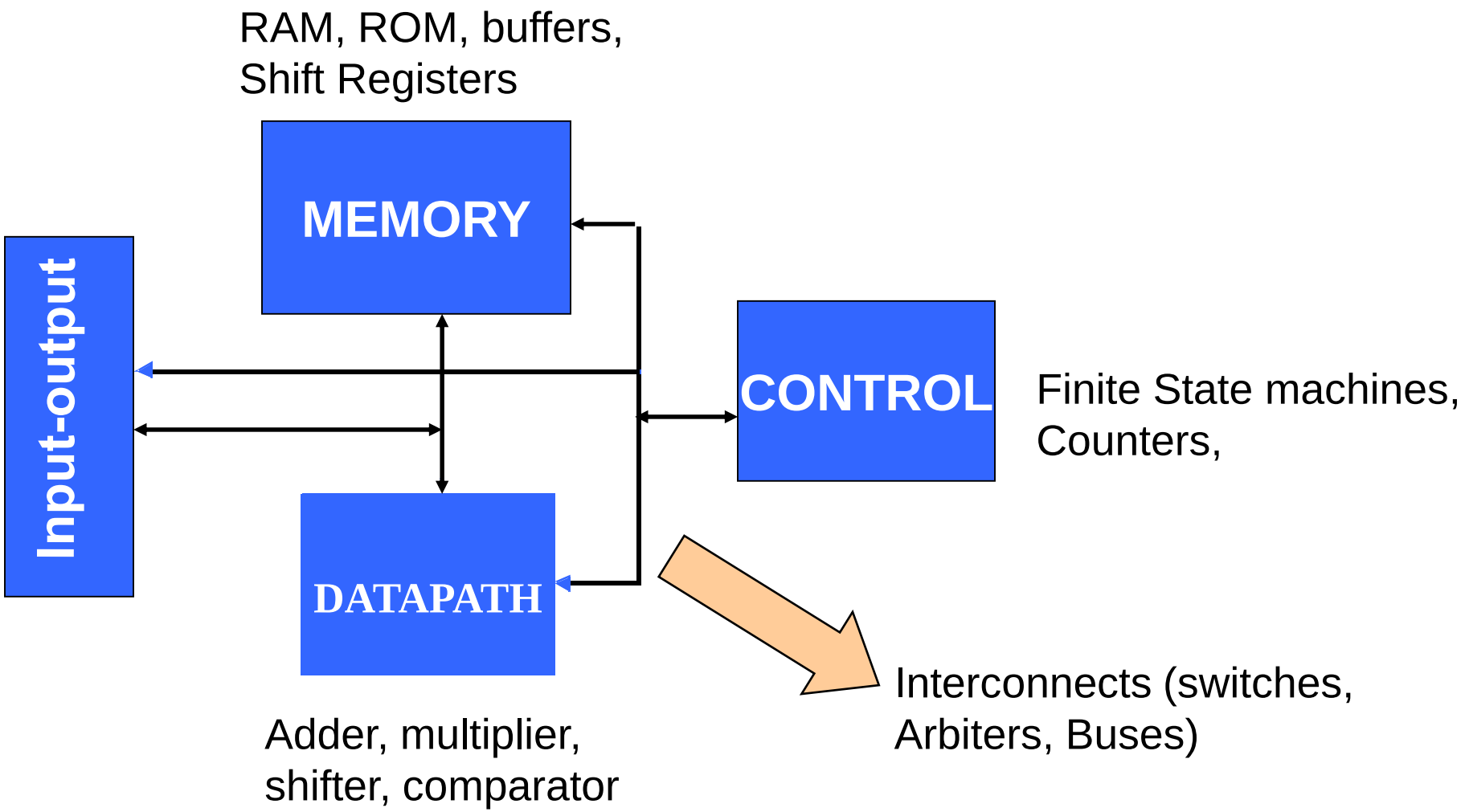
Arithmetic blocks

Learning outcomes

- Add binary numbers
- Subtract binary numbers
- Convert binary number in sign-magnitude representation
- Convert binary number to its one’s complement
- Convert binary number to its two’s complement
- Determine the decimal value of signed binary numbers
- Multiply binary numbers
- Implement arithmetic operations using logic circuits

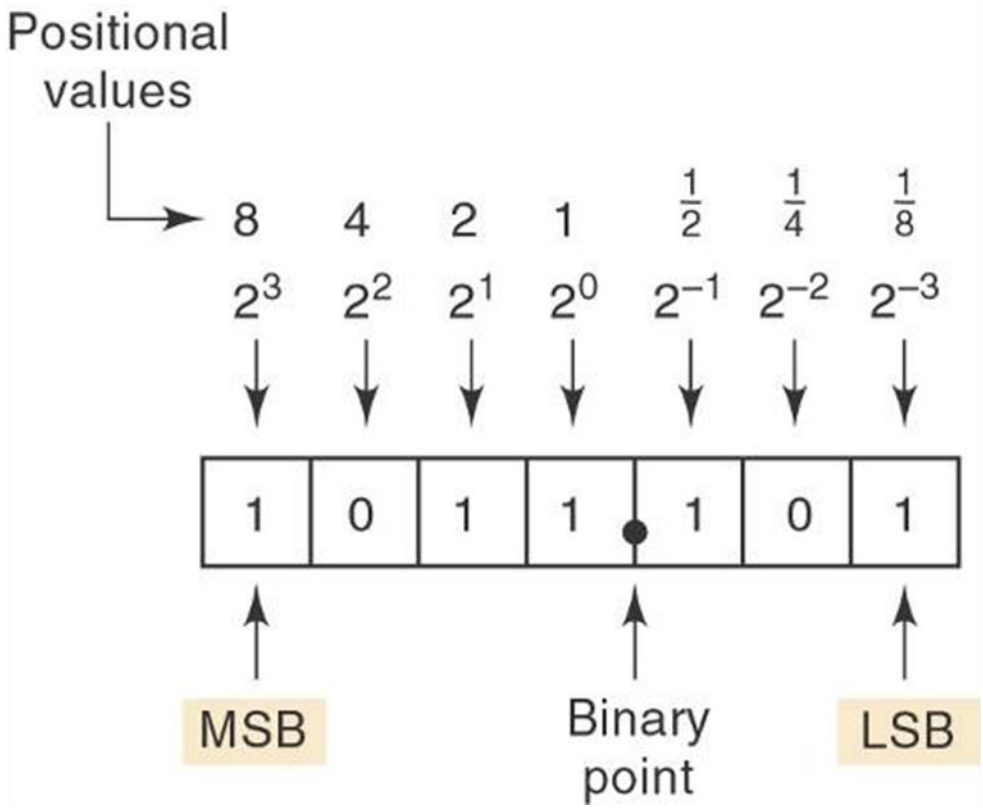
A Generic Digital Processor

- Datapath is the core of the processor and is the place where all computations are performed.



Binary Number System

- The Binary (base 2) System
  - 2 symbols: 0,1
    - Lends itself to electronic circuit design since only two different voltage levels are required.



Positional value may be stated as a digit multiplied by a power of 2.

Binary addition

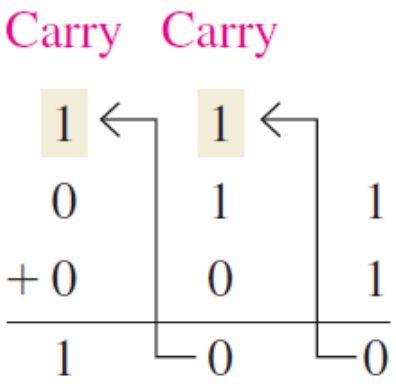
- Binary arithmetic is a very important operation in computers. The four basic rules for adding two bits X and Y are as follows:

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

- Where S is the sum and C is the carry
- Notice that the addition of two 1s yields a binary two $(10)_2$, with a sum of 0 and a carry of 1 over to the next column to the left.

Binary addition

- When there is a carry of 1 to the next column, you have a situation in which three bits are being added (a bit in each of the two numbers and a carry bit C_{in}). This situation is illustrated as follows:



- Adding 2 bits X and Y with a carry in C_{in} gives a sum S and a carry out C:

C_{in}	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

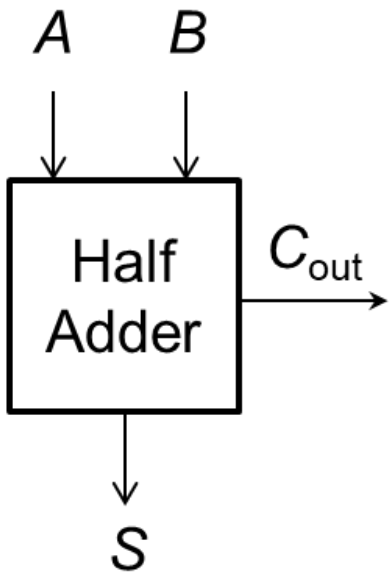
Binary addition

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

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	

Half adder

- Adds 2 input bits **A** and **B**
- Outputs 2 bits:
  - Carry bit $\mathbf{C_{out}}$
  - Sum bit **S**



$$S = A'B + AB'$$

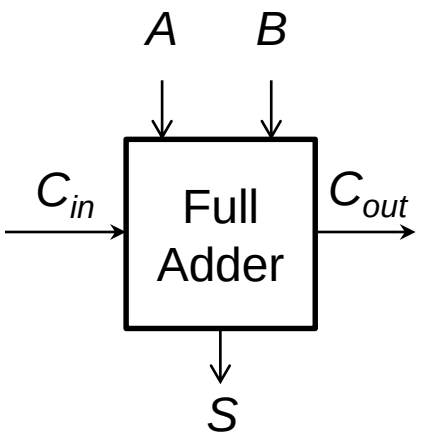
$$S = A \oplus B$$

$$C_{out} = AB$$

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

Full adder

- Full adder adds 3 bits: **A**, **B**, and $\mathbf{C_{in}}$
- Two output bits:
  - Carry bit $\mathbf{C_{out}}$
  - Sum bit **S**



- Sum bit is 1 if the number of 1's in the input is odd (odd function).
- Carry bit is 1 if the number of 1's in the input is 2 or 3.

A B C_{in}	S	C_{out}
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

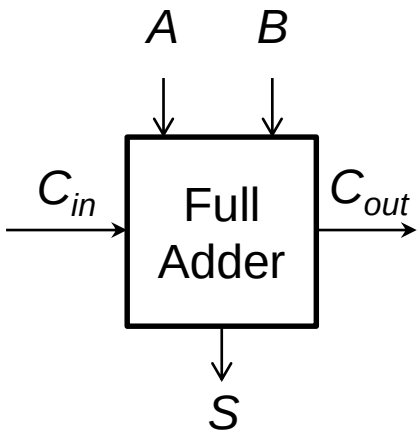
Full adder equations

K-Map of S

A \ BC_{in}	00	01	11	10
0	0	1	0	1
1	1	0	1	0

K-Map of C_{in}

A \ BC_{in}	00	01	11	10
0	0	0	1	0
1	0	1	1	1



$$S = AB'C_{in}' + A'BC_{in}' + A'B'C_{in} + ABC_{in}$$

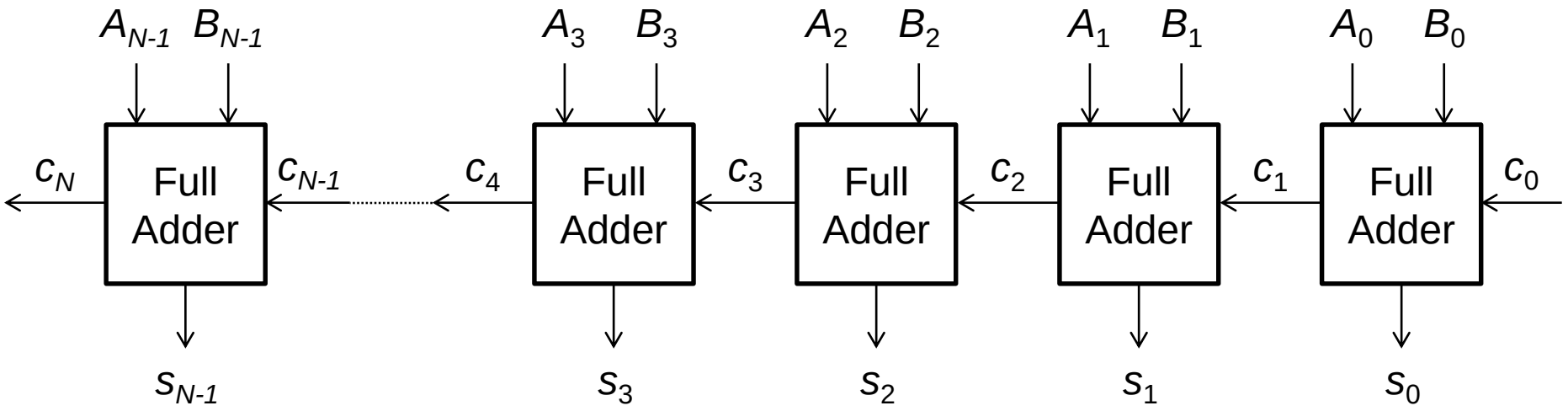
$$S = A \oplus B \oplus C_{in}$$

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

A B C_{in}	S	C_{out}
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

The Ripple Carry Adder

- Uses **identical copies** of a full adder (FA) to build a larger adder.
- Carry-out of cell *i* becomes carry-in to cell (*i*+1)
- Simple to implement
- Can be extended to add any number of bits
- This configuration is called a ripple-carry adder, since the carry bit "ripples" from one stage to the other.

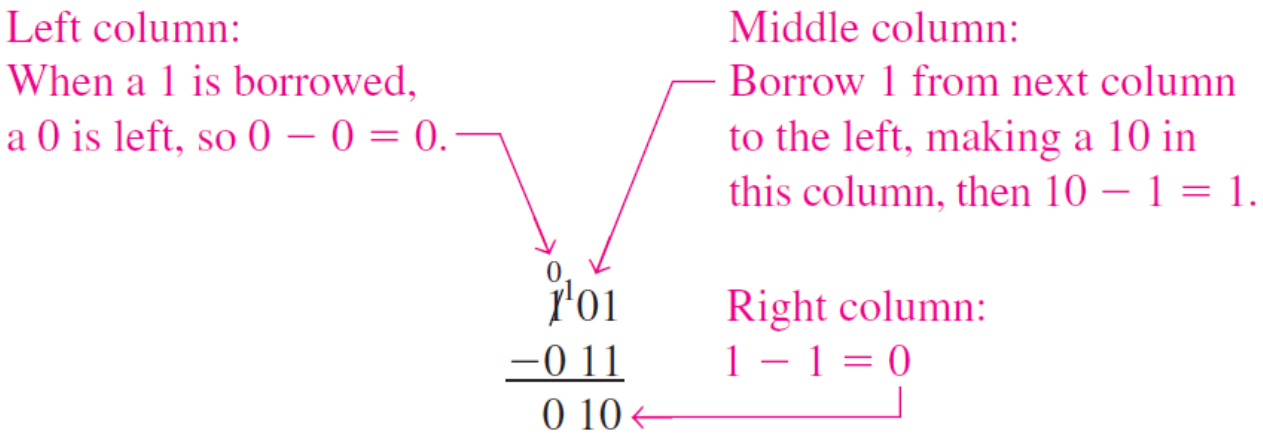


Binary Subtraction

- When subtracting 2 bits (X – Y), we get the difference (**D=X-Y**) and the **borrow-out B**, **with B=1 only for X<Y**.

X	0	0	1	1
– Y	– 0	– 1	– 0	– 1
B D	0 0	1 1	0 1	0 0

- When subtracting a 1 from a 0, you have to borrow a “1” from the next column to the left. In this case, a 10 (“2” in decimal) is created in the column being subtracted and a -1 is fed to the next column.



Binary Subtraction

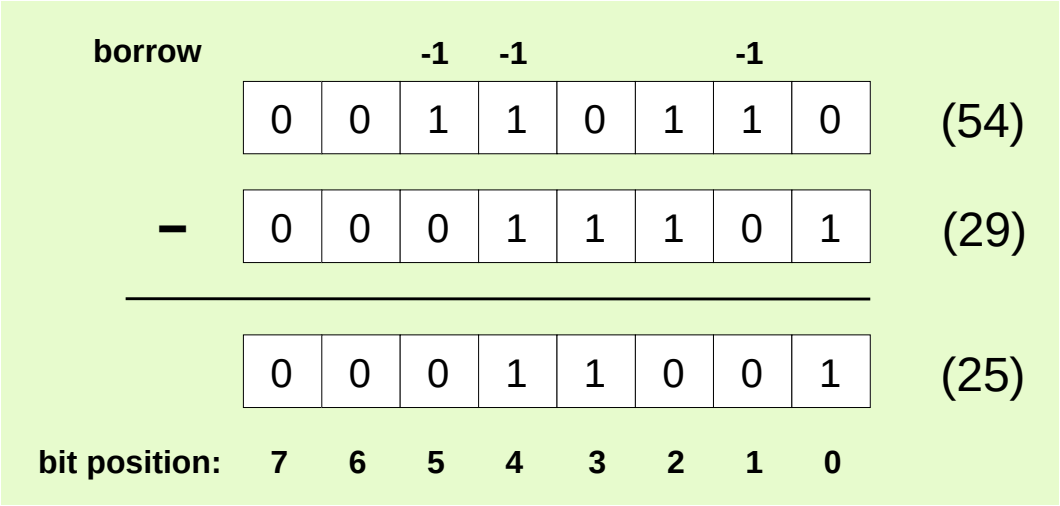
- When subtracting two bits (**X – Y**) with a **borrow-in B_{in}**, we get the **difference** (**$D=X-Y-B_{in}$**) and the **borrow-out** (**B_{out}**)

borrow-in	**-1**	**-1**	**-1**	**-1**	**-1**
	X	**0**	**0**	**1**	**1**
	– Y	**– 0**	**– 1**	**– 0**	**– 1**
	B_{out} D	**1 1**	**1 0**	**0 0**	**1 1**

- If $X<(Y+B_{in})$ then $B_{out}=1$ else $B_{out}=0$

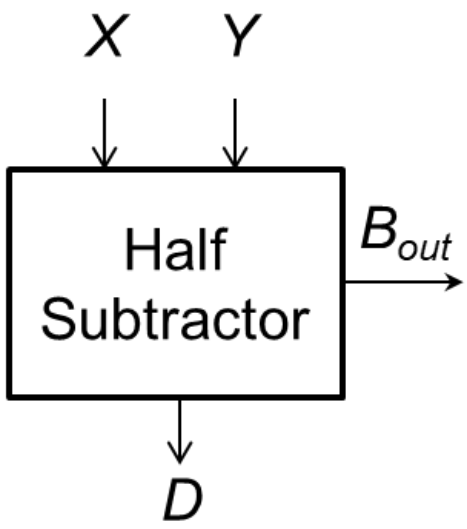
Binary Subtraction

- Start with the least significant bit (rightmost bit)
- Subtract each pair of bits
- Include the borrow in the subtraction, if present



Half Subtractor

- Subtracts 2 input bits ***X*** and ***Y***
- Outputs 2 bits:
  - Difference bit $\boldsymbol{D = X - Y}$
  - Borrow bit $\mathbf{B_{out}}$**=1 if** $\boldsymbol{X < Y}$



$$D = X'Y + XY'$$

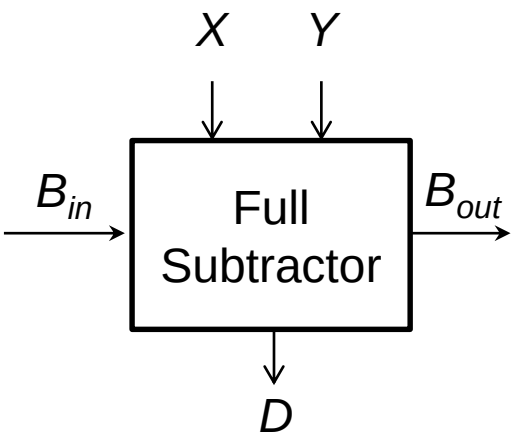
$$D = X \oplus Y$$

$$B_{out} = X'Y$$

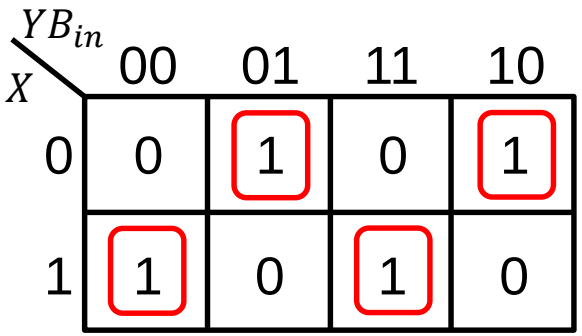
X	Y	D	B_{out}
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Full Subtractor equations

- Has three 1-bit inputs: **X**, **Y**, and $\mathbf{B_{in}}$
- Two output bits:
  - Difference bit $\boldsymbol{D = X - Y - B_{in}}$
  - Borrow bit $\mathbf{B_{out}}$**=1 if** $\boldsymbol{X < (Y + Z)}$
- Difference bit is 1 if the number of 1's in the input is odd (odd function as for a sum).



X Y B_{in}	D	B_{out}
0 0 0	0	0
0 0 1	1	1
0 1 0	1	1
0 1 1	0	1
1 0 0	1	0
1 0 1	0	0
1 1 0	0	0
1 1 1	1	1

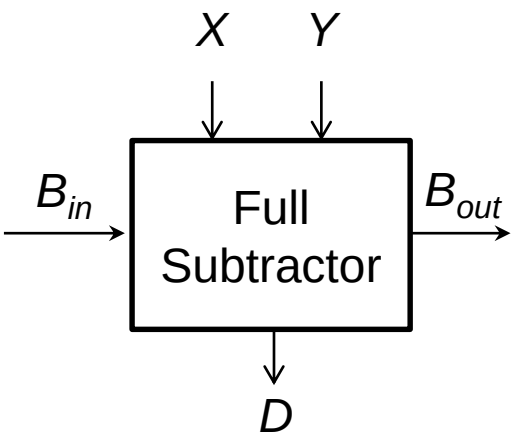


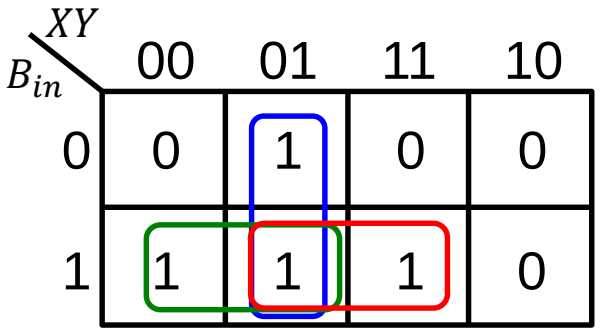
$$D = XY'B_{in}' + X'YB_{in}' + X'Y'B_{in} + XYB_{in}$$

$$D = X \oplus Y \oplus B_{in}$$

Full Subtractor equations

- Has three 1-bit inputs: **X**, **Y**, and $\mathbf{B_{in}}$
- Two output bits:
  - Difference bit $\boldsymbol{D = X - Y - B_{in}}$
  - Borrow bit $\mathbf{B_{out}}$**=1 if** $\boldsymbol{X < (Y + B_{in})}$



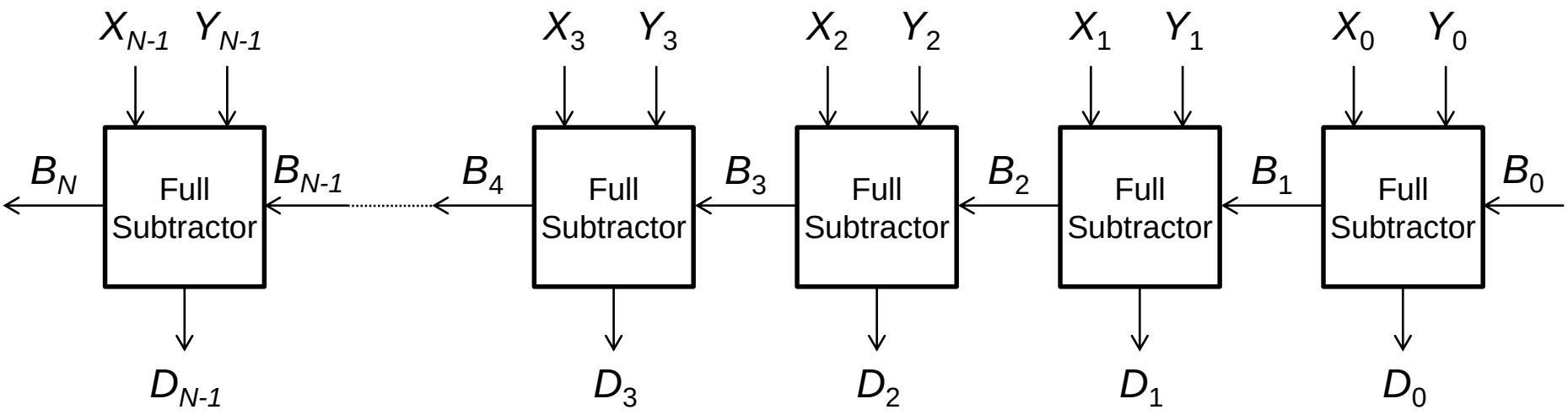


$$B_{out} = X'B_{in} + X'Y + YB_{in}$$

X Y B_{in}	D	B_{out}
0 0 0	0	0
0 0 1	1	1
0 1 0	1	1
0 1 1	0	1
1 0 0	1	0
1 0 1	0	0
1 1 0	0	0
1 1 1	1	1

The Ripple-Borrow Subtractor

- Uses **identical copies** of a full subtractor (FS) to build a larger subtractor.
- Borrow-out of cell *i* becomes borrow-in to cell (*i* +1).
- Simple to implement.
- Can be extended to subtract any number of bits.
- Called a ripple-borrow subtractor, since the borrow-out bit “ripples” from one stage to the other.



Combined Addition/Subtraction

- The adder we designed can add only non-negative numbers
  - If we can represent negative numbers, then subtraction is “just” the ability to add two numbers (one of which may be negative).
- To subtract Y from X, one can instead add the negation of B to A:

$$X - Y = X + (-Y)$$

- This would enable the use of a single ripple-carry adder to perform both addition (X+Y) and subtraction (X-Y).
- We’ll look at three different ways of representing signed numbers. The best one should result in the simplest and fastest operations.

Signed magnitude representation

- Humans use a signed-magnitude system: we add + or - in front of a magnitude to indicate the sign.
- We could do this in binary as well, by adding an extra sign bit to the front of our numbers. By convention:
  - A **0** sign bit represents a positive number.
  - A **1** sign bit represents a negative number.
- Examples:

	$1101_2 = 13_{10}$	(a 4-bit unsigned number)
0	$1101 = +13_{10}$	(a positive number in 5-bit signed magnitude)
1	$1101 = -13_{10}$	(a negative number in 5-bit signed magnitude)

	$0100_2 = 4_{10}$	(a 4-bit unsigned number)
0	$0100 = +4_{10}$	(a positive number in 5-bit signed magnitude)
1	$0100 = -4_{10}$	(a negative number in 5-bit signed magnitude)

Signed magnitude representation

- Negating a signed-magnitude number is trivial: just change the sign bit from 0 to 1, or vice versa.
- Adding numbers is difficult, though. To add, first check the signs. If they agree, then add the magnitudes and use the same sign; else subtract the smaller from the larger and use the sign of the larger.
- How do you determine which is smaller/larger?
- Comparators required: more complex circuitry needed to perform addition/subtraction.
- Two representations of zero!
- Therefore, sign and magnitude representation abandoned

Decimal	S.M.
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000
-0	1000
-1	1001
-2	1010
-3	1011
-4	1100
-5	1101
-6	1110
-7	1111
-8	—

	Unsigned	Signed Magnitude
Smallest	0	$-(2^{n-1}-1)$
Largest	2^n-1	$+(2^{n-1}-1)$

Converting signed-magnitude form to decimal

- Decimal values of positive and negative numbers in the sign-magnitude form are determined by summing the weights in all the magnitude bit positions where there are 1s and ignoring those positions where there are zeros. The sign is determined by examination of the sign bit.
- **Example:** Determine the decimal value of this signed binary number expressed in sign-magnitude: 10010101.

The seven magnitude bits and their powers-of-two weights are as follows:

2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	1	0	1	0	1

Summing the weights where there are 1s,

$$16 + 4 + 1 = 21$$

The sign bit is 1; therefore, the decimal number is $\mathbf{-21}$.

One's complement representation

- A different approach, one's complement, negates numbers by complementing each bit of the number.
- Positive integers have the most significant bit (leftmost) equal to 0
  - The magnitude of positive numbers is the same as the sign and magnitude representation
- Negative integers have the most significant bit equal to 1
  - Negative numbers in n-bit one's complement has the same binary representation as the unsigned binary number given by **$(2^n-1)-A$**

Decimal	S.M.	1's comp.
7	0111	0111
6	0110	0110
5	0101	0101
4	0100	0100
3	0011	0011
2	0010	0010
1	0001	0001
0	0000	0000
-0	1000	1111
-1	1001	1110
-2	1010	1101
-3	1011	1100
-4	1100	1011
-5	1101	1010
-6	1110	1001
-7	1111	1000
-8	–	–

	Unsigned	Signed Magnitude	One's complement
Smallest	0	$-(2^{n-1}-1)$	$-(2^{n-1}-1)$
Largest	2^n-1	$+(2^{n-1}-1)$	$+(2^{n-1}-1)$

Converting one's complement form to decimal

- Decimal values of **positive numbers** in the 1's complement form are determined by summing the weights in all bit positions where there are 1s and ignoring those positions where there are zeros.
- **Example:** Determine the decimal values of the signed binary numbers expressed in 1's complement: **00010111**

The bits and their powers-of-two weights for the positive number are as follows:

-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	0	1	0	1	1	1

Summing the weights where there are 1s,

$$16 + 4 + 2 + 1 = \mathbf{+23}$$

Converting one's complement form to decimal

- Decimal values of **negative numbers** are determined by assigning a negative value to the weight of the sign bit, summing all the weights where there are 1s, and adding 1 to the result.
- **Example:** Determine the decimal values of the signed binary numbers expressed in 1's complement: **11101000**.

The bits and their powers-of-two weights for the negative number are as follows. Notice that the negative sign bit has a weight of -2^7 or -128.

-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	1	0	1	0	0	0

Summing the weights where there are 1s,

$$-128 + 64 + 32 + 8 = -24$$

Adding 1 to the result, the final decimal number is

$$-24 + 1 = \mathbf{-23}$$

One's complement addition

- To add one's complement numbers:
  - First do unsigned addition on the numbers, including the sign bits.
  - Then take the carry out and add it to the sum.
- Two examples:

```
   0111     (+7)
+  1011   + (-4)
  ------  -------
 1 0010

   0010
+     1
  ------
   0011     (+3)
```

```
   0011     (+3)
+  0010   + (+2)
  ------  -------
 0 0101

   0101
+     0
  ------
   0101     (+5)
```

- This is simpler and more uniform than signed magnitude addition.

One's complement representation

- Still two zeros
- What is **-00000** ? Answer: **11111**
- Although used for a while on some computer products, one's complement was eventually abandoned because another solution was better.

Decimal	S.M.	1's comp.
7	0111	0111
6	0110	0110
5	0101	0101
4	0100	0100
3	0011	0011
2	0010	0010
1	0001	0001
0	0000	0000
-0	1000	1111
-1	1001	1110
-2	1010	1101
-3	1011	1100
-4	1100	1011
-5	1101	1010
-6	1110	1001
-7	1111	1000
-8	—	—

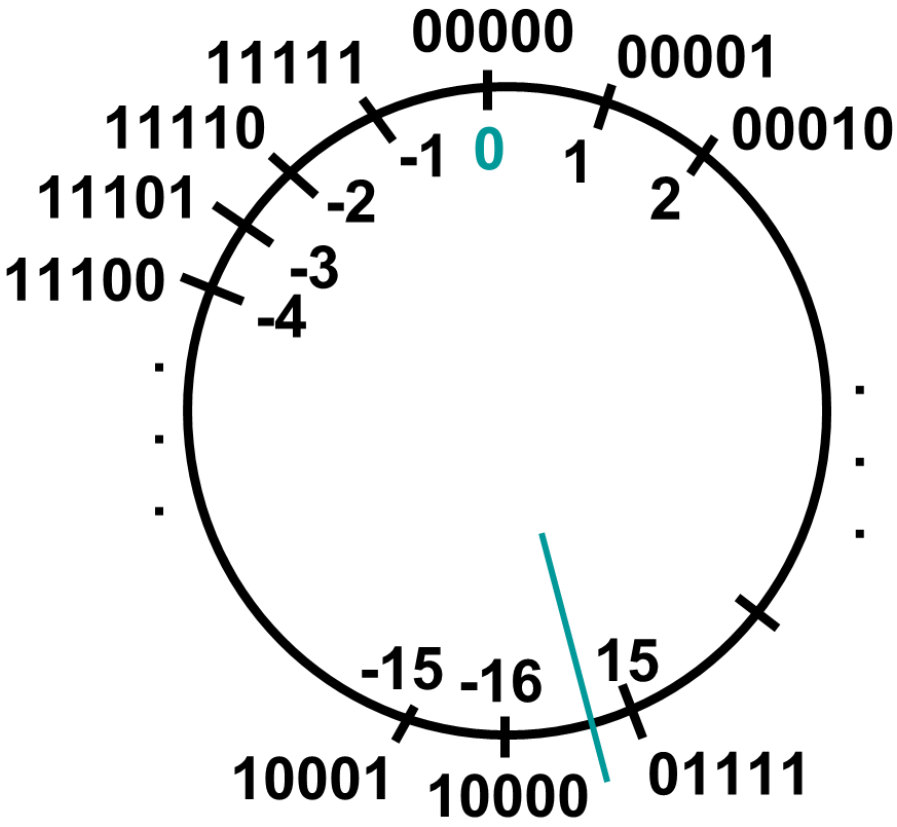
	Unsigned	Signed Magnitude	One's complement
Smallest	0	$-(2^{n-1}-1)$	$-(2^{n-1}-1)$
Largest	2^n-1	$+(2^{n-1}-1)$	$+(2^{n-1}-1)$

Two's complement representation

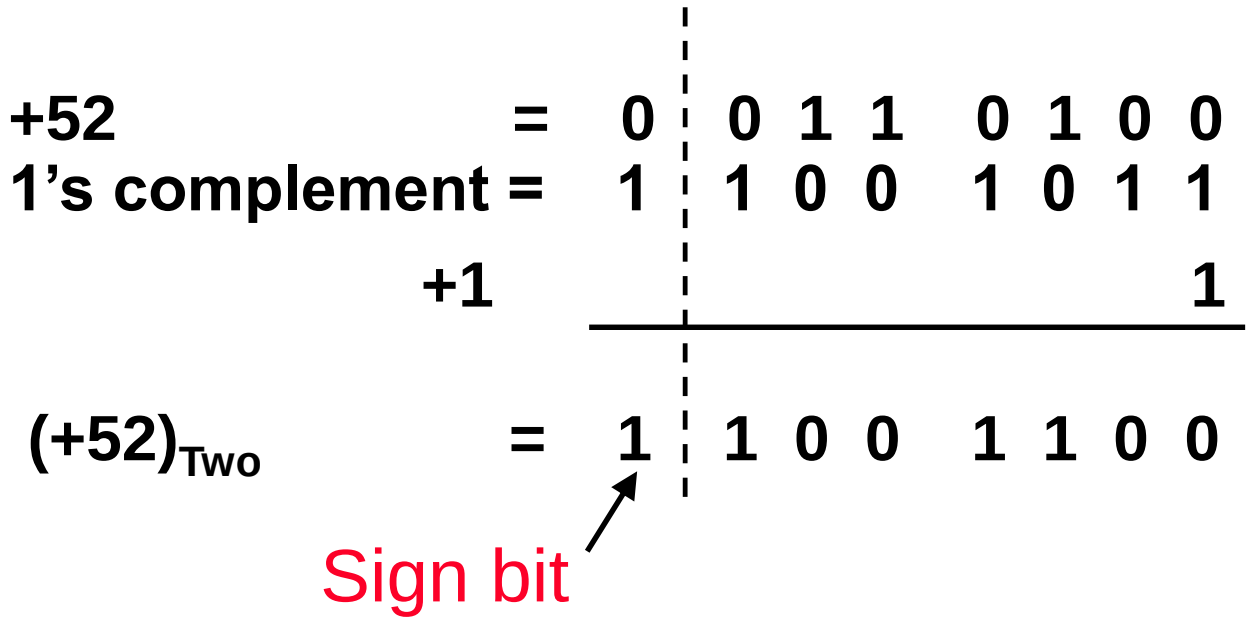
- This is the standard negative number representation. Better because as we shall see later, it allows us to perform the operation of subtraction by actually performing an addition.
- The 2's complement of a binary number is formed by taking the 1's complement of the number and adding 1 to the least significant bit position.
- **2's complement=1's complement +1**.
- This means that negative numbers in n-bit 2's complement has the same binary representation as the unsigned binary number given by $\mathbf{2^n - A}$

2's complement "Line": N=5

- Zero has a single representation.
- The 2's complement of 0 is itself.
- A number represented in 2's complement form has a 0 in the sign bit if it is positive, a 1 if it is negative.
- Special case where sign bit is 1 and all 0s: 100…0 is equivalent to -2^{N-1}
- -2^{N-1} does not have a positive decimal equivalent (it maps into itself).



Two's complement representation



$(-52)_{Two}$ = 0 | 0 1 1 0 1 0 0

= (+52)

- Taking the 2's complement of a binary number negates it, i.e. changes its sign

Converting two's complement form to decimal

- Decimal values of positive and negative numbers in the 2's complement form are determined by summing the weights in all bit positions where there are 1s and ignoring those positions where there are zeros. The weight of the sign bit in a negative number is given a negative value.
- **Example 1:** Determine the decimal values of the signed binary numbers expressed in 2's complement: **01010110**.

The bits and their powers-of-two weights for the positive number are as follows:

-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	0	1	0	1	1	0

Summing the weights where there are 1s,

$$64 + 16 + 4 + 2 = \mathbf{+86}$$

Converting two's complement form to decimal

- Decimal values of positive and negative numbers in the 2's complement form are determined by summing the weights in all bit positions where there are 1s and ignoring those positions where there are zeros. The weight of the sign bit in a negative number is given a negative value.
- **Example 2:** Determine the decimal values of the signed binary numbers expressed in 2's complement: **10101010**.

The bits and their powers-of-two weights for the negative number are as follows. Notice that the negative sign bit has a weight of $-2^7 = -128$.

-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	1	0	1	0	1	0

Summing the weights where there are 1s,

$$-128 + 32 + 8 + 2 = \mathbf{-86}$$

More about two's complement

- Two other equivalent ways to negate two's complement numbers:
  - You can subtract an n-bit two's complement number from 2^n.

$$\begin{array}{r} 100000 \\ -\ 01101 \quad (+13_{10}) \\ \hline 10011 \quad (-13_{10}) \end{array} \qquad \begin{array}{r} 100000 \\ -\ 00100 \quad (+4_{10}) \\ \hline 11100 \quad (-4_{10}) \end{array}$$

  - You can complement all of the bits to the left of the rightmost 1.

01101 = $+13_{10}$ (a positive number in two's complement)

10011 = -13_{10} (a negative number in two's complement)

00100 = $+4_{10}$ (a positive number in two's complement)

11100 = -4_{10} (a negative number in two's complement)

Two's complement addition

- Negating a two's complement number takes a bit of work, but addition is much easier than with the other two systems.
- To find A + B, you just have to:
  - Do unsigned addition on A and B, including their sign bits.
  - Ignore any carry out.
- For example, to find 0111 + 1100, or (+7) + (-4):
  - First add 0111 + 1100 as unsigned numbers:

$$\begin{array}{r} 0111 \\ +\ 1100 \\ \hline 10011 \end{array}$$

  - Discard the carry out (**1**).
  - The answer is 0011 (+3).

Another two's complement example

- To further convince you that this works, let's try adding two negative numbers—1101 + 1110, or (-3) + (-2) in decimal.
- Adding the numbers gives 11011:

$$\begin{array}{r} 1\,1\,0\,1 \\ +\ 1\,1\,1\,0 \\ \hline 1\,1\,0\,1\,1 \end{array}$$

- Dropping the carry out (1) leaves us with the answer, 1011 (-5).

Comparing the signed number systems

- Here are all the 4-bit numbers in the different systems.
- *Positive numbers are the same in all three representations.*
- Signed magnitude and one's complement have *two* ways of representing 0. This makes things more complicated.
- Two's complement has asymmetric ranges; there is one more negative number than positive number. Here, you can represent -8 but not +8.
- However, two's complement is preferred because it has only one 0, and its addition algorithm is the simplest.

Decimal	S.M.	1's comp.	2's comp.
7	0111	0111	0111
6	0110	0110	0110
5	0101	0101	0101
4	0100	0100	0100
3	0011	0011	0011
2	0010	0010	0010
1	0001	0001	0001
0	0000	0000	0000
-0	1000	1111	—
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8	—	—	1000

Ranges of the signed number systems

- How many negative and positive numbers can be represented in each of the different systems on the previous page?

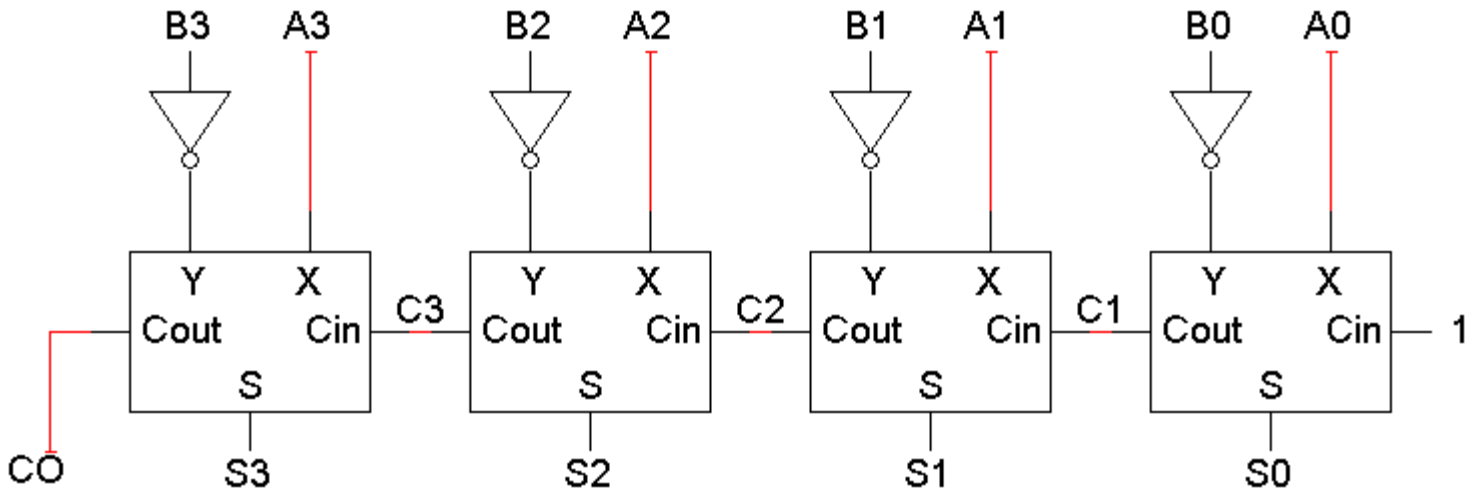
	Unsigned	Signed Magnitude	One's complement	Two's complement
Smallest	0000 (0)	1111 (-7)	1000 (-7)	1000 (-8)
Largest	1111 (15)	0111 (+7)	0111 (+7)	0111 (+7)

	Unsigned	Signed Magnitude	One's complement	Two's complement
Smallest	0	$-(2^{n-1}-1)$	$-(2^{n-1}-1)$	-2^{n-1}
Largest	2^n-1	$+(2^{n-1}-1)$	$+(2^{n-1}-1)$	$+(2^{n-1}-1)$

- In general, with n-bit numbers including the sign, the ranges are:

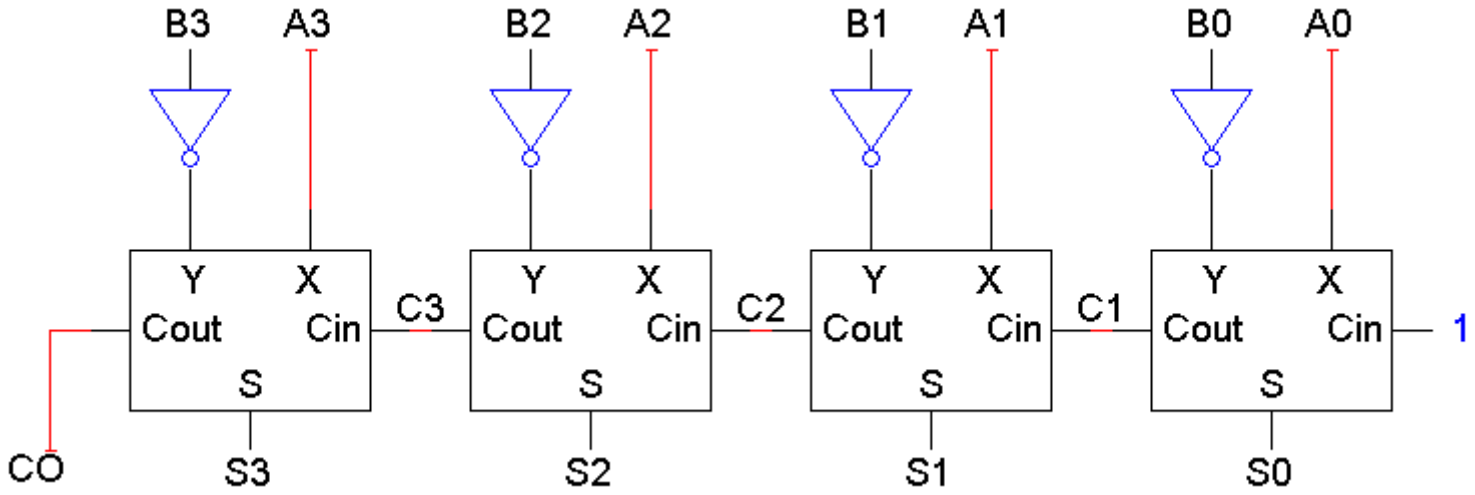
A two's complement subtraction circuit

- To find A - B with an adder, we'll need to:
  - Complement each bit of B.
  - Set the adder's carry in to 1.
- The net result is A + B' + 1, where B' + 1 is the two's complement negation of B.



Small differences

- The only differences between the adder and subtractor circuits are:
  - The subtractor has to negate B3 B2 B1 B0.
  - The subtractor sets the initial carry in to 1, instead of 0.

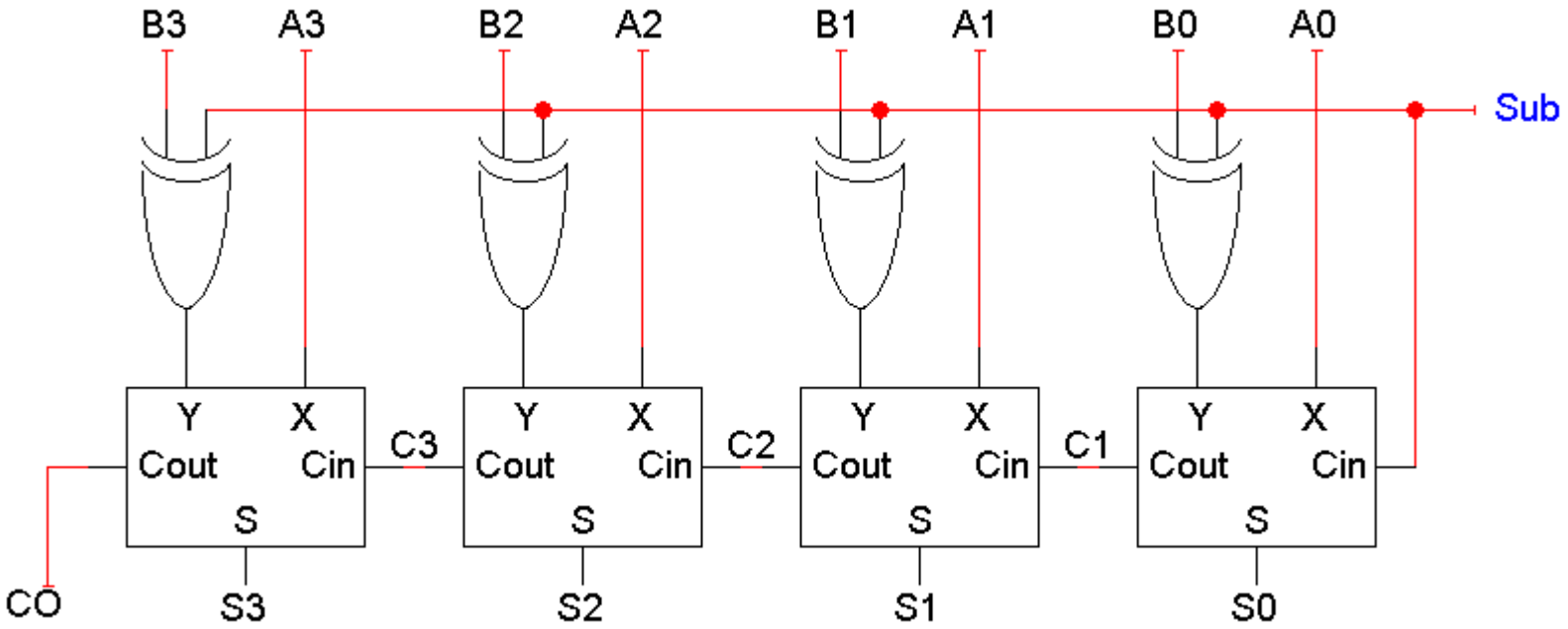


An adder-subtractor circuit

- XOR gates let us selectively complement the B input.

$$X \oplus 0 = X \qquad\qquad X \oplus 1 = X'$$

- When Sub = 0, the XOR gates output B3 B2 B1 B0 and the carry in is 0. The adder output will be A + B + 0, or just A + B.
- When Sub = 1, the XOR gates output B3' B2' B1' B0' and the carry in is 1. Thus, the adder output will be a two's complement subtraction, A - B.



Signed overflow

- With two's complement and a 4-bit adder, for example, the largest representable decimal number is +7, and the smallest is -8.
- What if you try to compute 4 + 5, or (-4) + (-5)?

```
  0100   (+4)          1100   (-4)
+ 0101   (+5)        + 1011   (-5)
-------              -------
 01001   (-7)         10111   (+7)
```

- We cannot just include the carry out to produce a five-digit result, as for unsigned addition. If we did, (-4) + (-5) would result in +23!
- Also, unlike the case with unsigned numbers, the carry out *cannot* be used to detect overflow.
  - In the example on the left, the carry out is 0 but there is overflow.
  - Conversely, there are situations where the carry out is 1 but there is no overflow.

Detecting signed overflow

- The easiest way to detect signed overflow is to look at all the sign bits.

```
  0100   (+4)
+ 0101   (+5)
-------
 01001   (-7)
```

```
  1100   (-4)
+ 1011   (-5)
-------
 10111   (+7)
```

- Overflow occurs only in the two situations above:
  - If you add two positive numbers and get a negative result.
  - If you add two negative numbers and get a positive result.
- Overflow cannot occur if you add a positive number to a negative number.

Sign extension

- In everyday life, decimal numbers are assumed to have an infinite number of 0s in front of them. This helps in “lining up” numbers.
- To subtract 231 and 3, for instance, you can imagine:

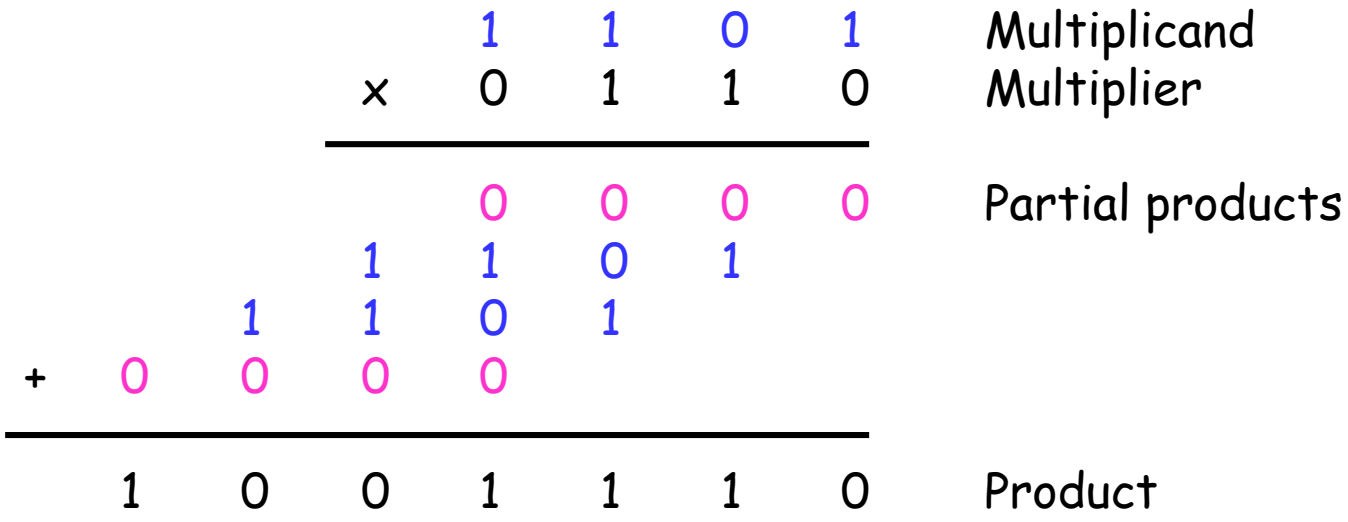
$$\begin{array}{r} 231 \\ -\ 003 \\ \hline 228 \end{array}$$

- You need to be careful in extending signed binary numbers, because the leftmost bit is the *sign* and not part of the magnitude.
- If you just add 0s in front, you might accidentally change a negative number into a positive one!
- For example, going from 4-bit to 8-bit numbers:
  - 0101 (+5) should become 0000 0101 (+5).
  - But 1100 (-4) should become 1111 1100 (-4).
- The proper way to extend a signed binary number is to replicate the sign bit, so the sign is preserved.

Subtraction summary

- A good representation for negative numbers makes subtraction hardware much easier to design.
- Two's complement is used most often (although signed magnitude shows up sometimes, such as in floating-point systems, which we'll discuss on Wednesday).
- Using two's complement, we can build a subtractor with minor changes to the adder from last week.
- We can also make a single circuit which can both add and subtract.
- Sign extension is needed to properly "lengthen" negative numbers.

Binary multiplication example



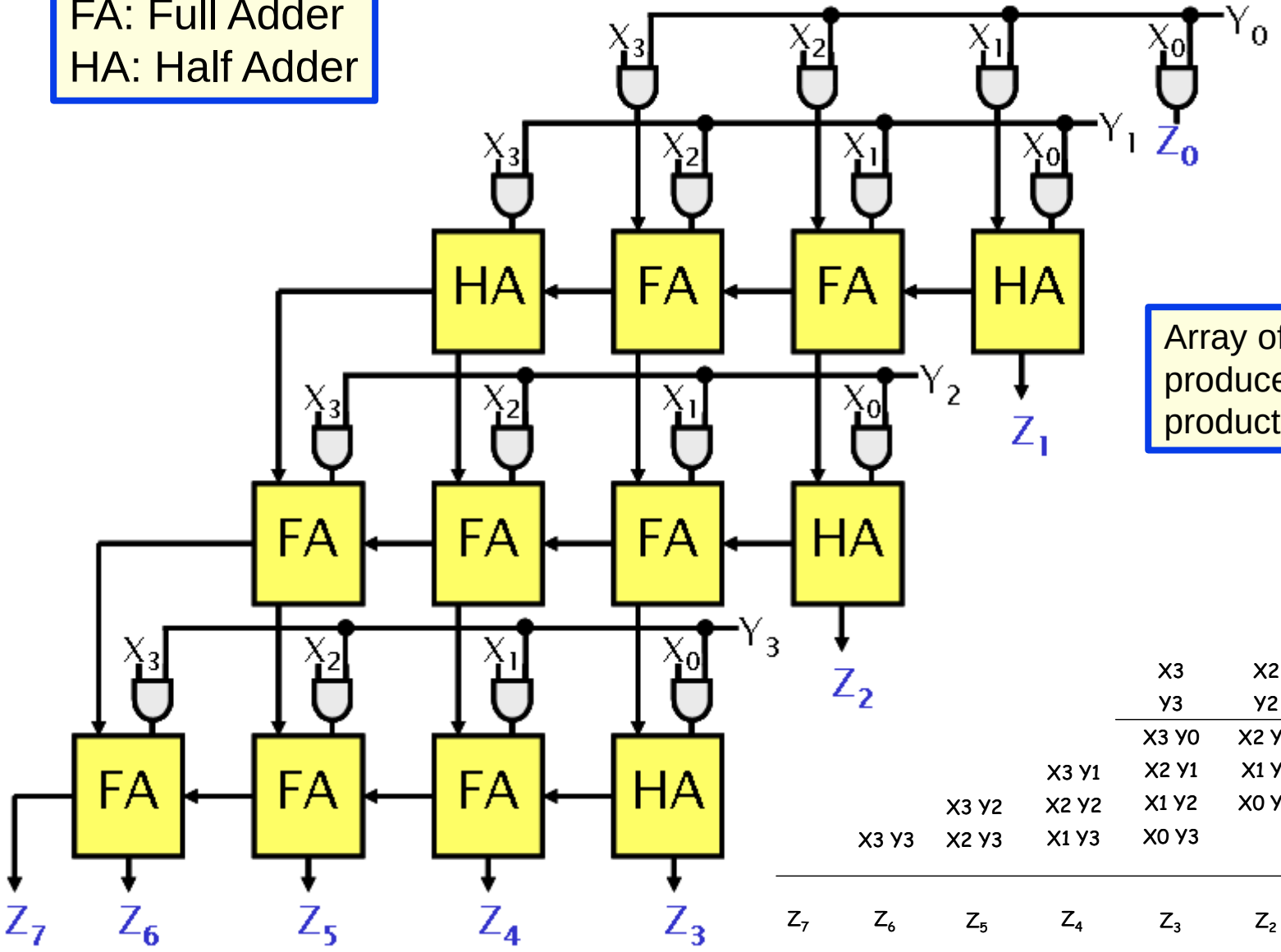
- Binary Multiplication table is simple:

  0×0=0 ; 0×1=0 ; 1×0=0 ; 1×1=1

- The multiplication process may be viewed to consist of two steps:
  - Evaluation of partial products
  - Accumulation of the shifted partial products.
- Since we always multiply by either 0 or 1, the partial products are always either 0000 or the multiplicand (1101 in this example).
- There are four partial products which are added to form the result.

A 4×4 array multiplier

FA: Full Adder
HA: Half Adder

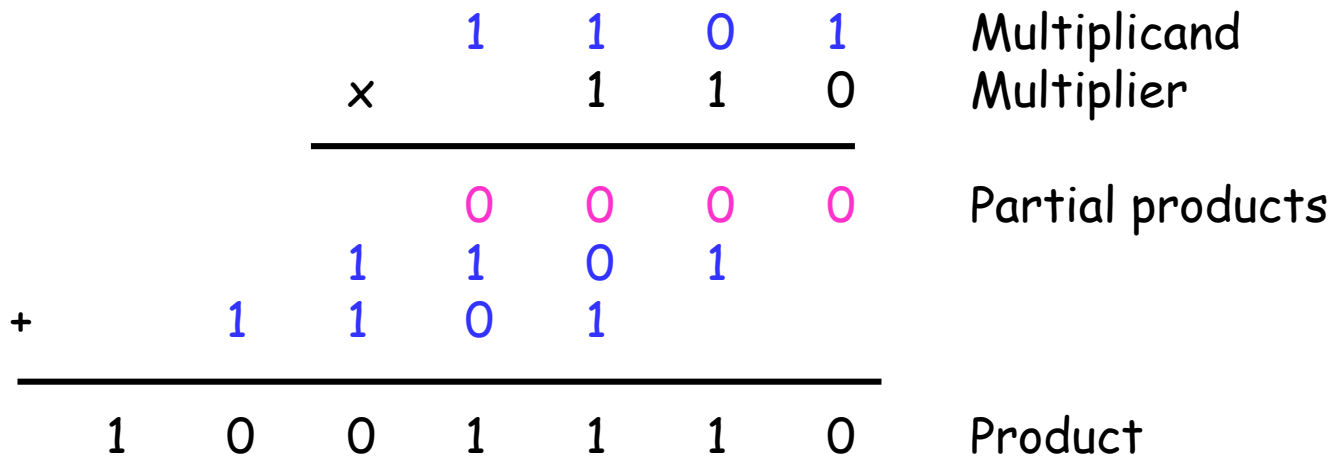


Array of AND gates produce the partial products X_iY_i

				X3	X2	X1	X0
				y3	y2	y1	y0
				X3 Y0	X2 Y0	X1 Y0	X0 Y0
			X3 Y1	X2 Y1	X1 Y1	X0 Y1	
		X3 Y2	X2 Y2	X1 Y2	X0 Y2		
	X3 Y3	X2 Y3	X1 Y3	X0 Y3			
Z_7	Z_6	Z_5	Z_4	Z_3	Z_2	Z_1	Z_0

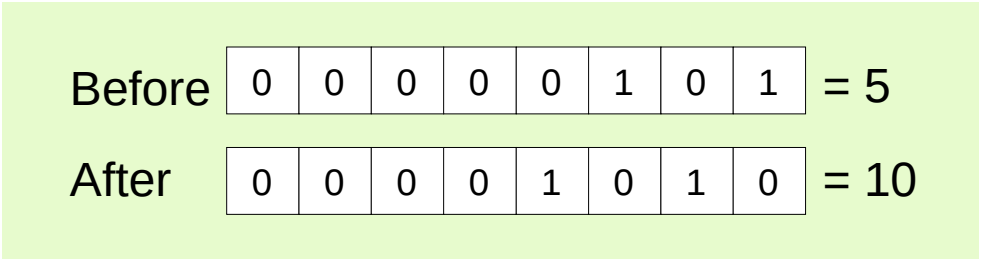
More on multipliers

- Multipliers are very complex circuits.
- There are n partial products, one for each bit of the multiplier.
- There are also a large number of full adders.
- n-bit multiplicand × n-bit multiplier = 2(n+m)-bit product
- The circuit for 32-bit/64-bit multiplication would be huge!



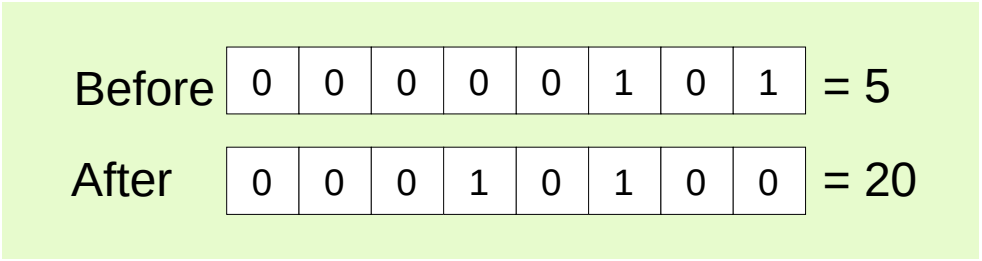
Shifting the Bits to the Left

- What happens if the bits are shifted to the left by 1 bit position?



Multiplication by 2

- What happens if the bits are shifted to the left by 2 bit positions?



Multiplication by 4

- Shifting the Bits to the Left by n bit positions is multiplication by 2^n
- As long as we have sufficient space to store the bits

Shifting the Bits to the Right

- What happens if the bits are shifted to the right by 1 bit position?

Before	0	0	1	0	0	1	1	**0**	= 38
After	0	0	0	1	0	0	1	1	= 19

Division by 2

- What happens if the bits are shifted to the right by 2 bit positions?

Before	0	0	1	0	0	1	**1**	**0**	= 38
After	0	0	0	0	1	0	0	1	= 9

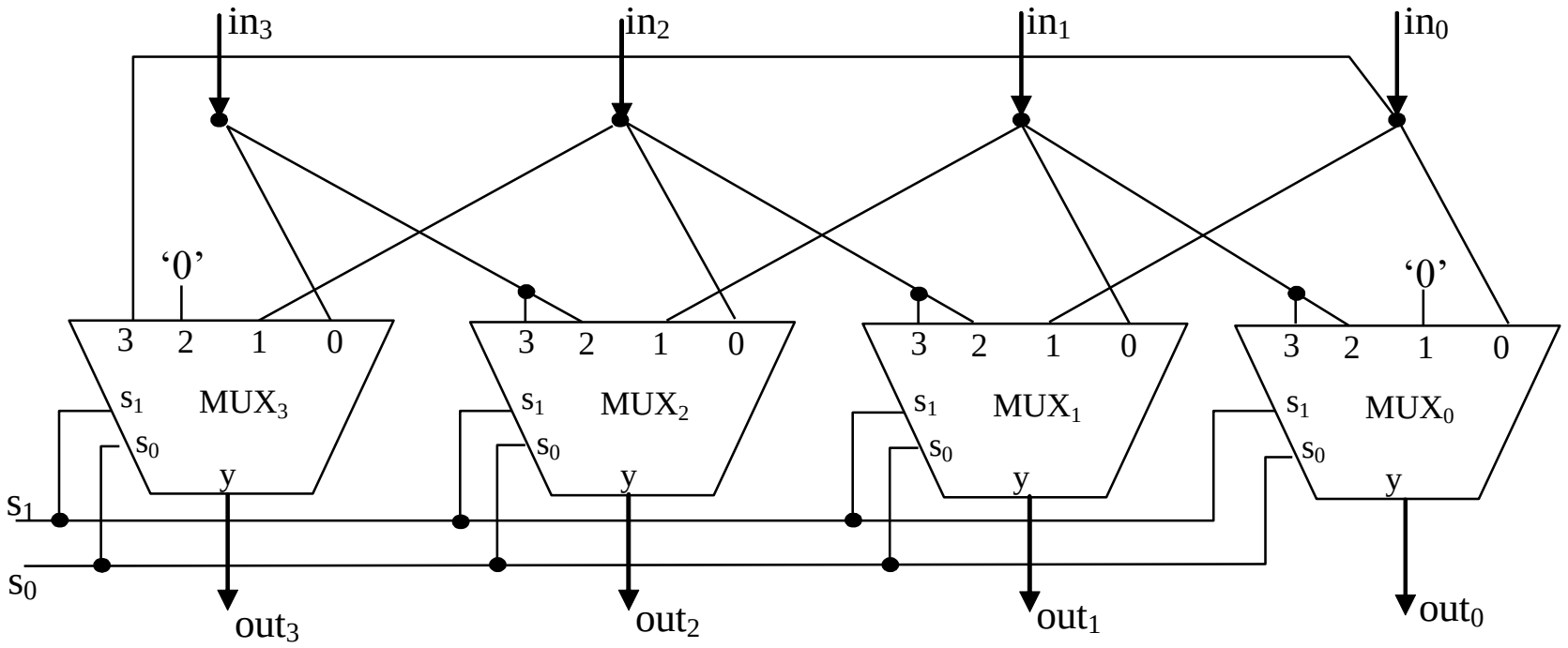
Division by 4

- Shifting the Bits to the Right by n bit positions is division by 2^n

Shift Operations

- Arithmetic Shift:
  - Shifts number left or right. Rt shift sign extends
    - 1011 LSR1 = 0101 1011 LSL1 = 0110
- Logical Shift:
  - Shifts number left or right and fills with 0's
    - 1011 LSR1 = 0101 1011 LSL1 = 0110
- Rotate:
  - Shifts number left or right and fills with lost bits
    - 1011 ROR1 = 1101 1011 ROL1 = 0111

4-bit combinational Shifter

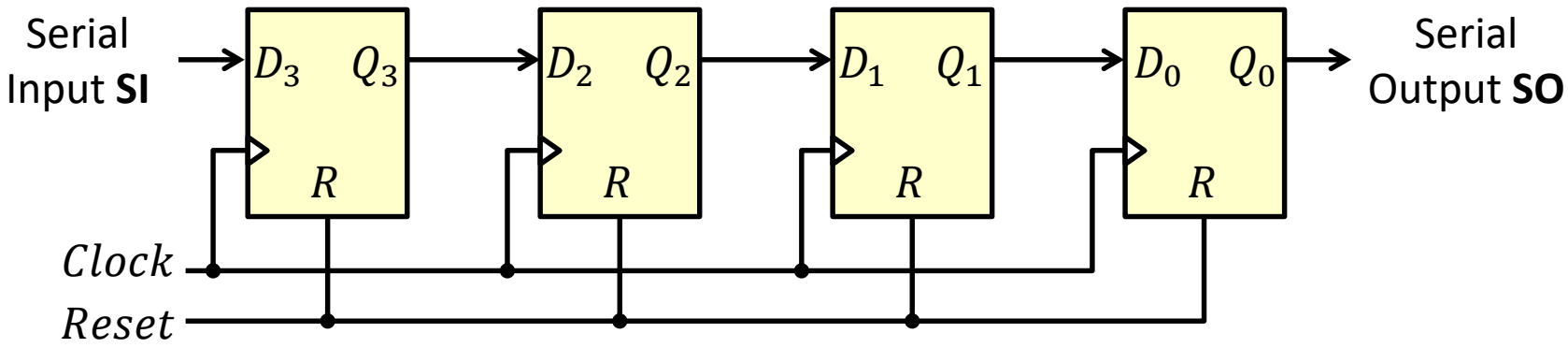


s_1	s_2	Operations
0	0	Pass through
0	1	Shift left and fill with 0
1	0	Shift right and fill with 0
1	1	Rotate right

- If fixed shift, simply wire the inputs to the correct output positions
- only 4 operations shown

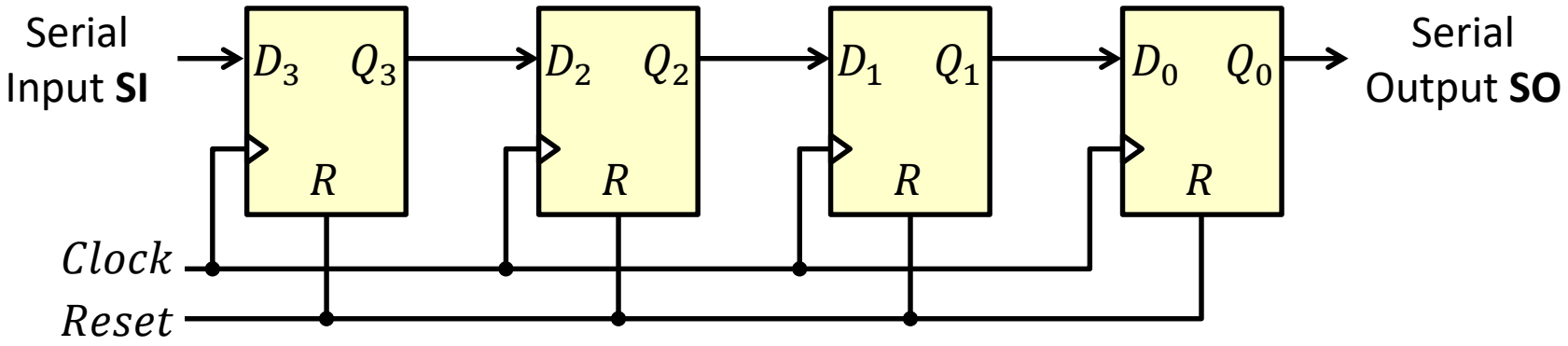
Shift Registers

- A shift register is a cascade of flip flops sharing the same clock
- Allows the data to be shifted from each flip-flop to its neighbor
- The output of a flip-flop is connected to the input of its neighbor
- Shifting can be done in either direction
- All bits are shifted simultaneously at the active edge of the clock



Right Shift Register

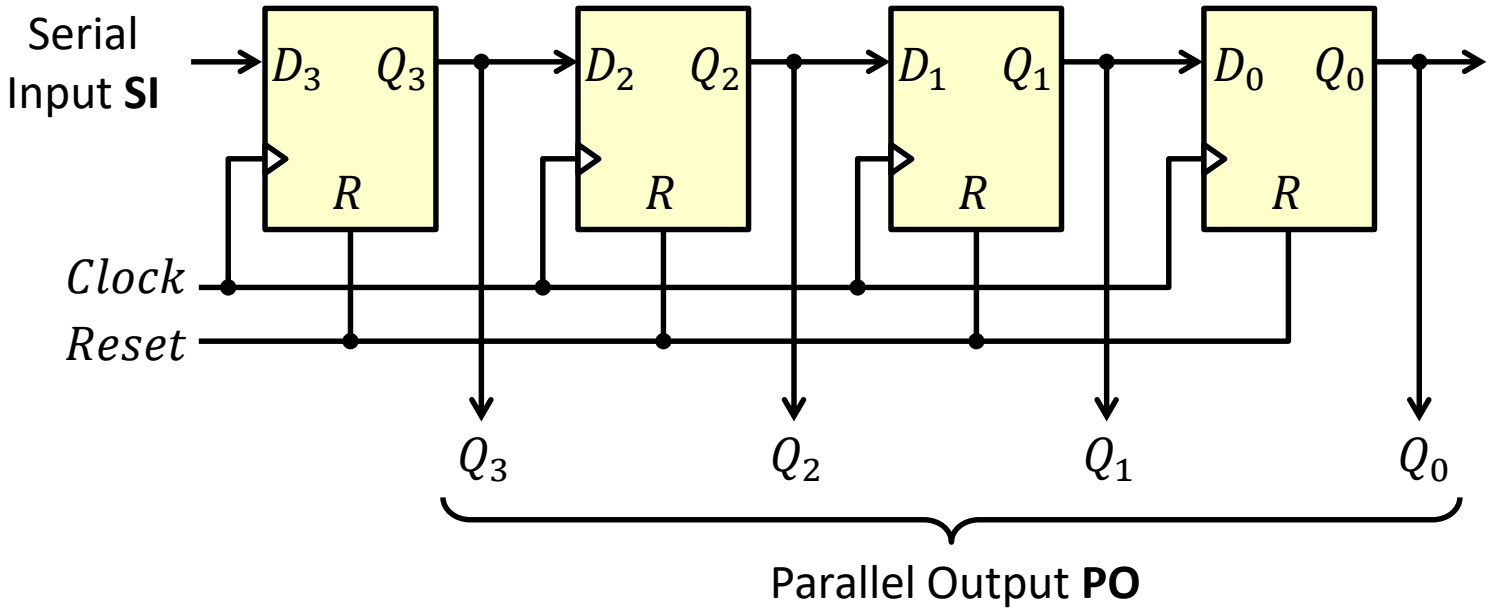
Timing of a Shift Register



Cycle	SI	Q3	Q2	Q1	Q0 = SO
T0	1	1	0	1	0
T1	0	1	1	0	1
T2	1	0	1	1	0
T3	1	1	0	1	1
T4	0	1	1	0	1
T5	1	0	1	1	0
T6	0	1	0	1	1

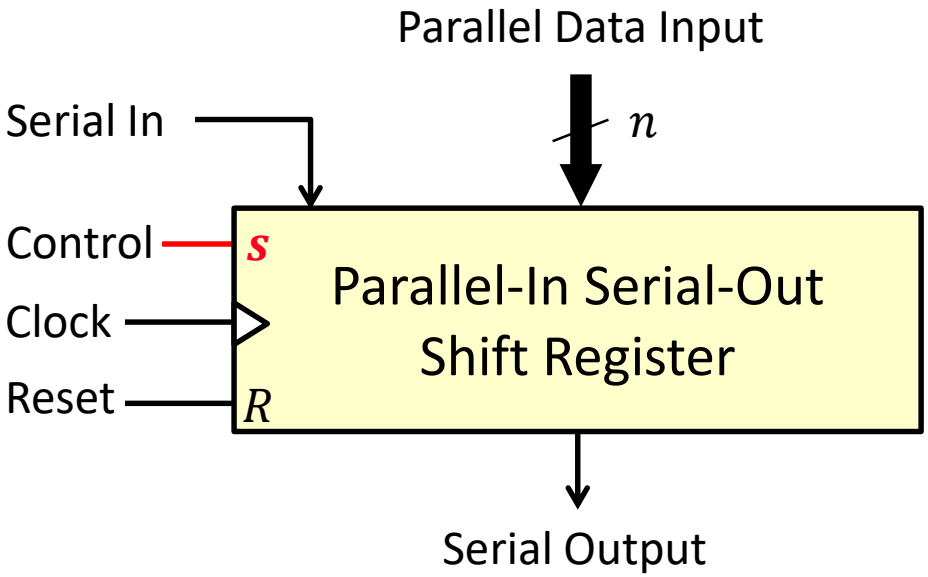
Shift Register with Parallel Output

- The output of a shift register can be serial or parallel
- A Serial-In Parallel-Out (SIPO) shift register is shown below
- All flip-flop outputs can be read in parallel



Parallel-In Serial-Out Shift Register

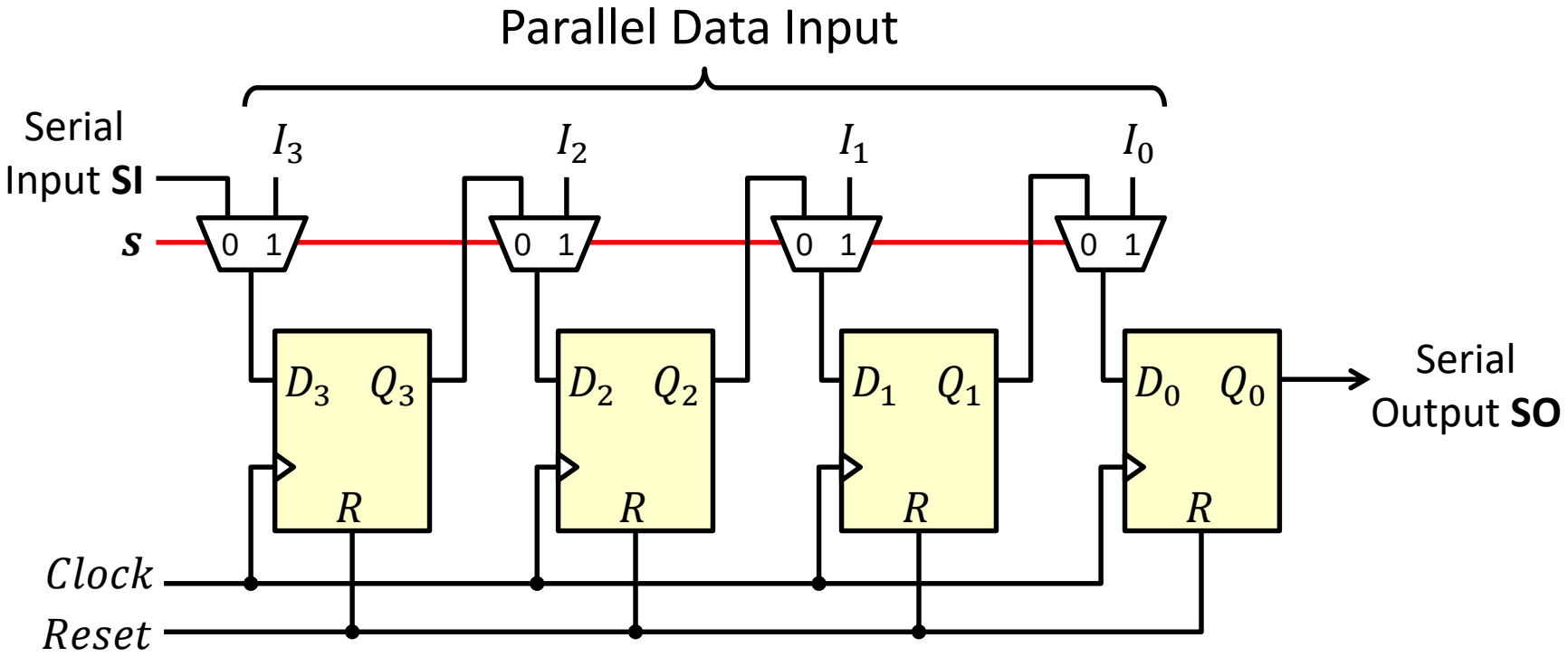
- A Parallel-In Serial-Out (PISO) Shift Register has:
  - n parallel data input lines
  - Serial Input
  - Serial Output
  - Control input **s**
  - Clock input
  - Reset input
- Two control functions:
  - **s = 0** ➔ Shift Data
  - **s = 1** ➔ Parallel Load n input bits



Parallel In Serial Out Shift Register

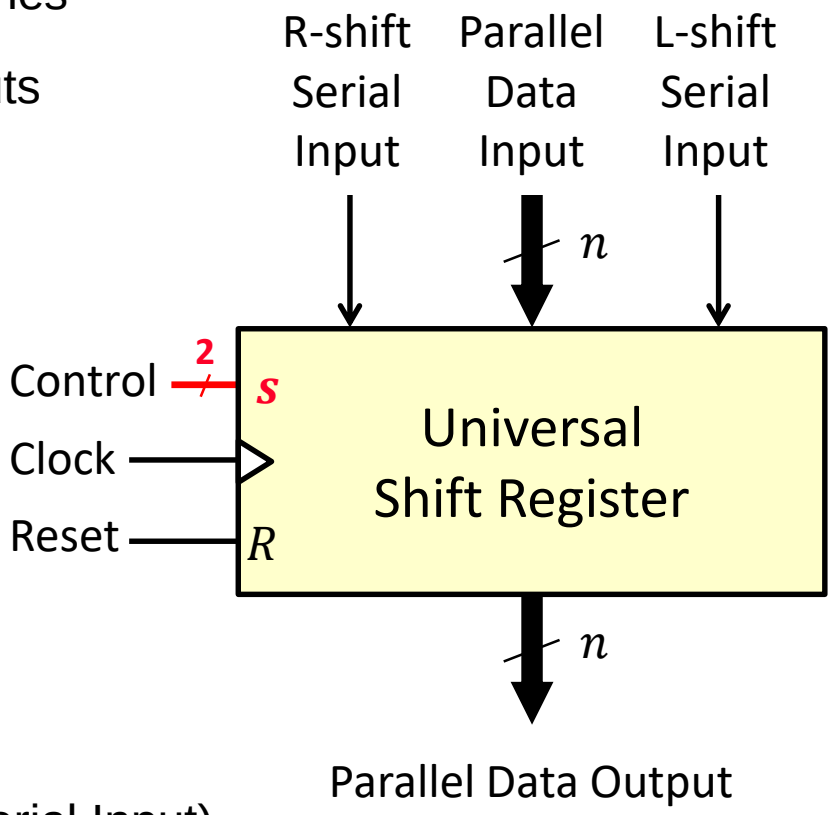
- Two control functions:

s = 0 ➔ Shift **s = 1** ➔ Load data



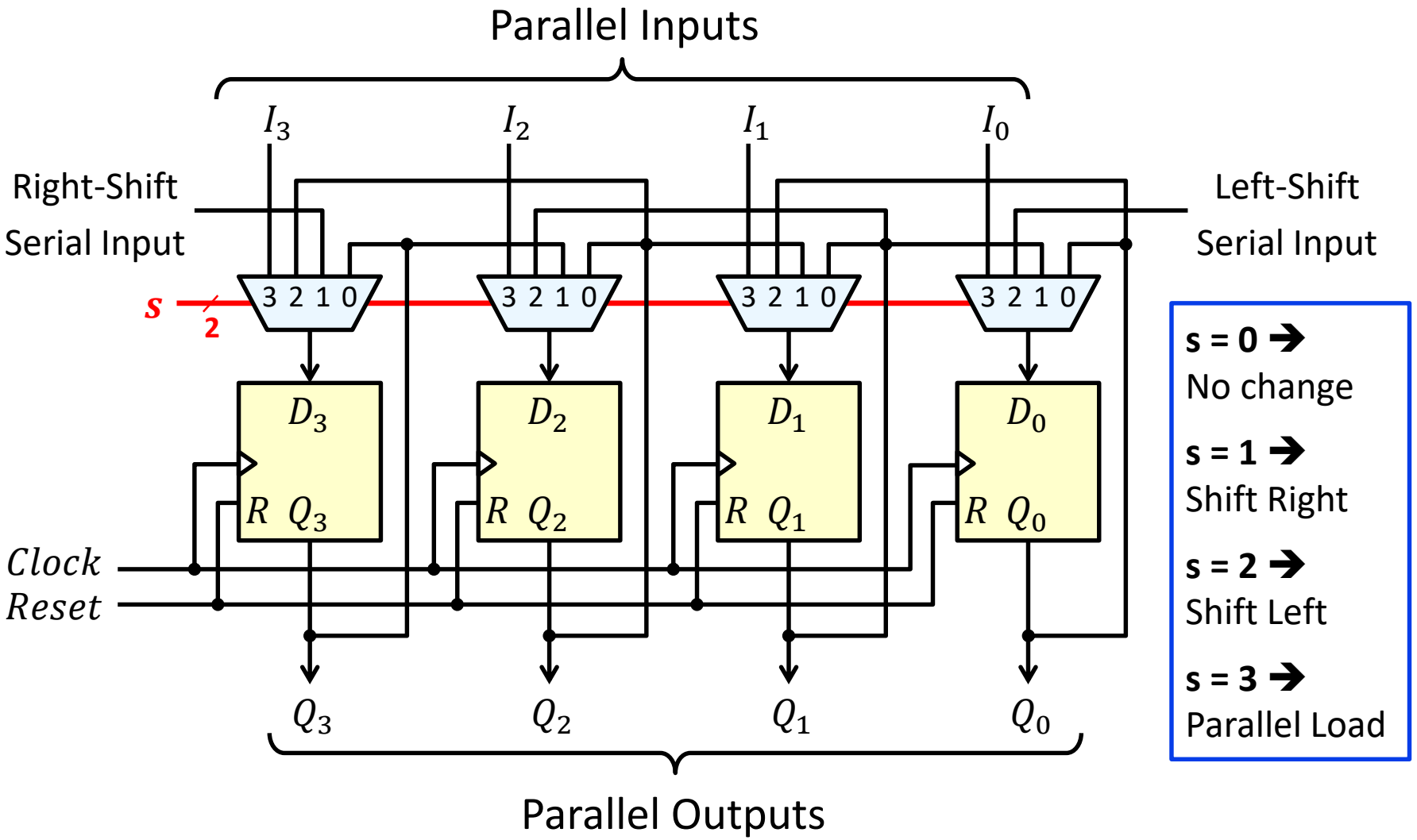
Universal Shift Register

- A Universal Shift Register has the following specification:
  - n parallel data input and n output lines
  - Right-shift and Left-shift Serial Inputs
  - Two control input lines **s**
  - Clock input
  - Reset input



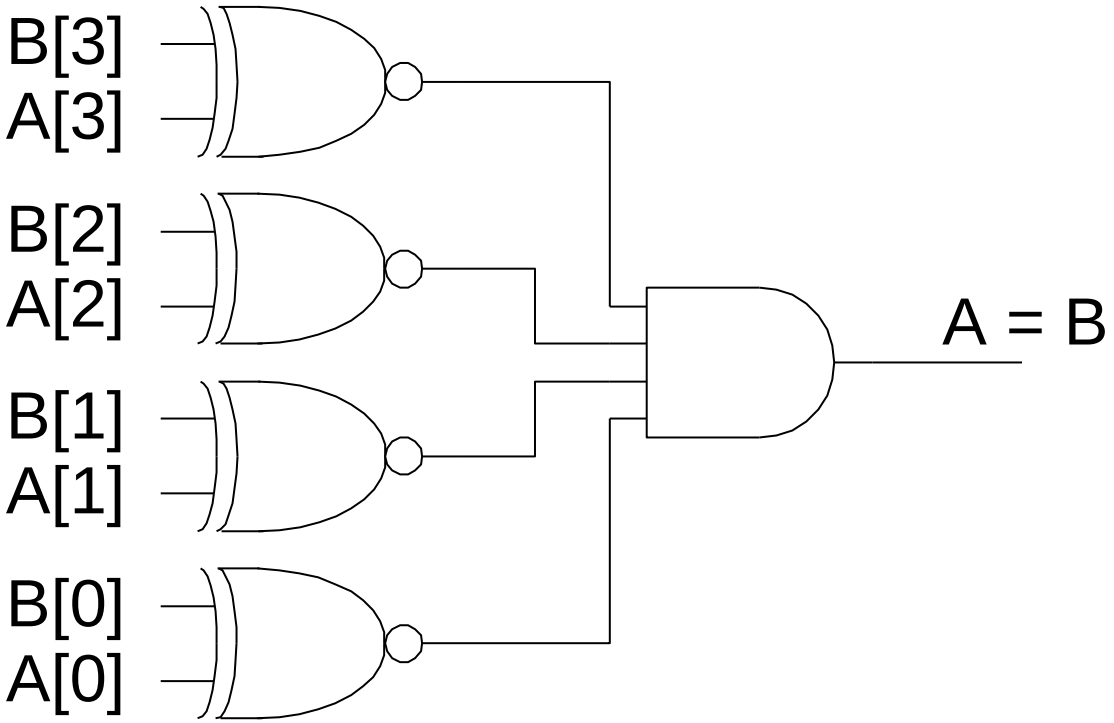
- Four control functions:
  - **s = 00** ➔ No change in value
  - **s = 01** ➔ Shift Right (Right-Shift Serial Input)
  - **s = 10** ➔ Shift Left (Left-Shift Serial Input)
  - **s = 11** ➔ Parallel Load n input bits

Universal Shift Register



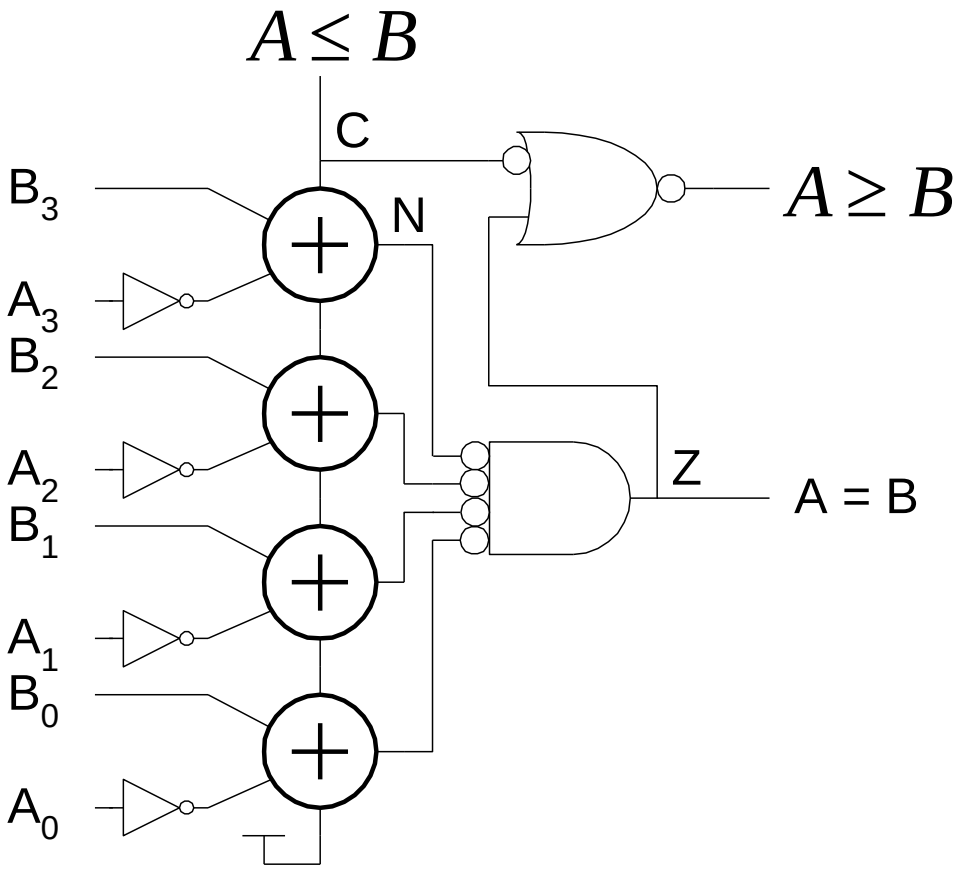
Comparators

- Check if each bit is equal (XNOR: equality gate)
- 1's detect on bitwise equality



Magnitude Comparator

- Compute B-A and look at sign
- B-A = B + A' + 1
- For unsigned numbers, carry out is sign bit

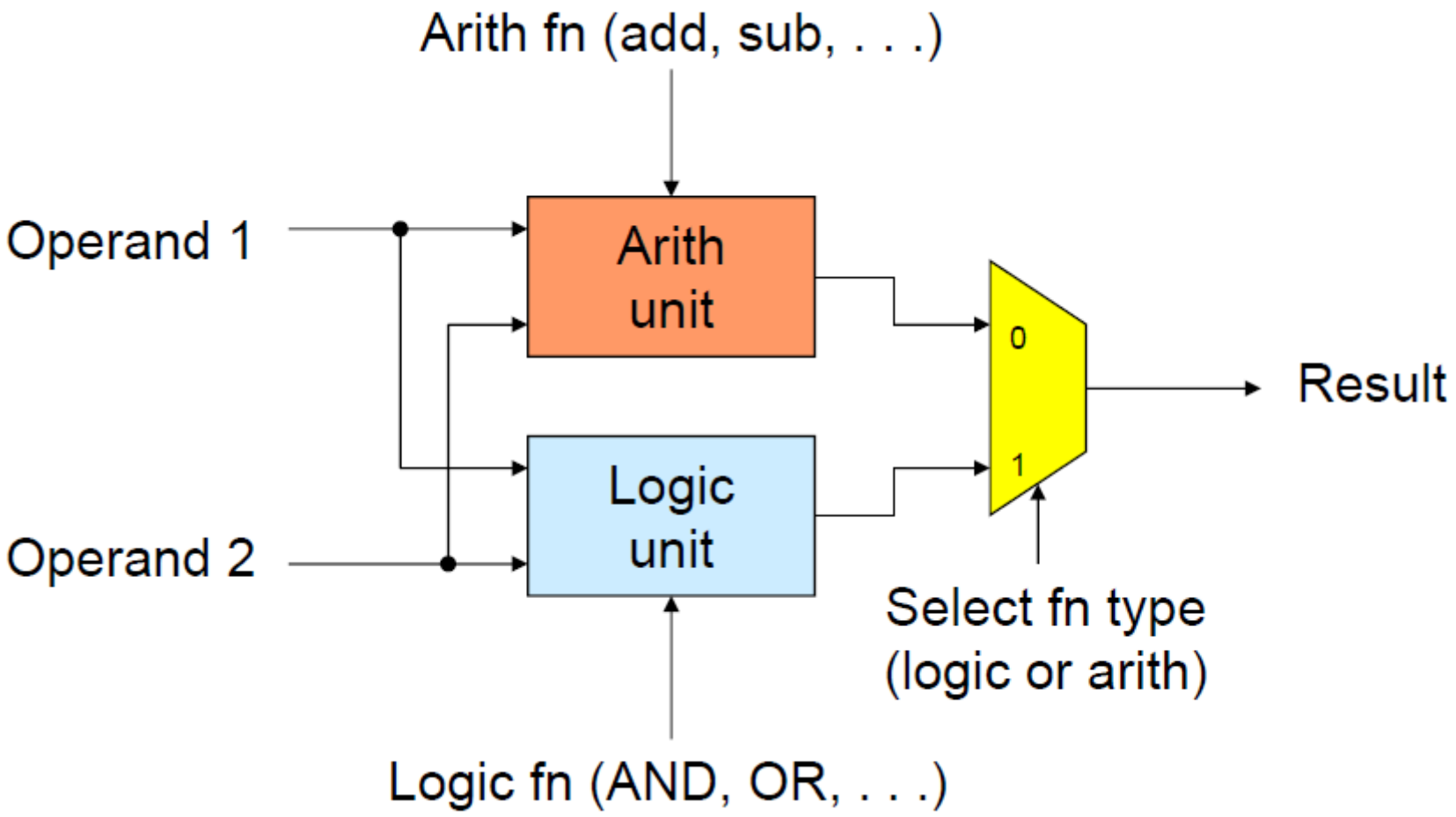


Signed vs. Unsigned

- For signed numbers, comparison is harder
  - C: carry out
  - Z: zero (all bits of A-B are 0)
  - N: negative (MSB of result)
  - V: overflow (inputs had different signs, output sign ≠ B)

Magnitude Comparison		
Relation	Unsigned Comparison	Signed comparison
$A = B$	Z	Z
$A \neq B$	$\overline{Z}$	$\overline{Z}$
$A < B$	$\overline{C + Z}$	$\overline{(N \oplus V) + Z}$
$A > B$	$\overline{C}$	$(N \oplus V)$
$A \leq B$	C	$(\overline{N \oplus V})$
$A \geq B$	$\overline{C} + Z$	$(N \oplus V) + Z$

Multi-function Arithmetic Logic Unit (ALU)



Summary

- Adders and multipliers are built hierarchically.
  - We start with half adders or full adders and work our way up.
  - Building these functions from scratch with truth tables and K-maps would be pretty difficult.
- The arithmetic circuits impose a limit on the number of bits that can be added. Exceeding this limit results in overflow.
- Multiplication and division by powers of 2 can be handled with simple shifting.
- It is possible to build a combinational circuit to carry out division, but in practice it is often done using shifts and subtracts.

Acknowledgments

- Credit is acknowledged where credit is due. Please refer to the full list of references.