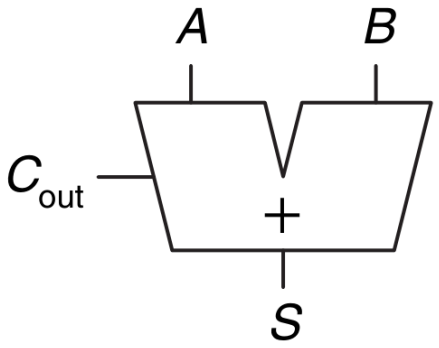
Digital Logic Design

Lecture 3

Combinational logic blocks

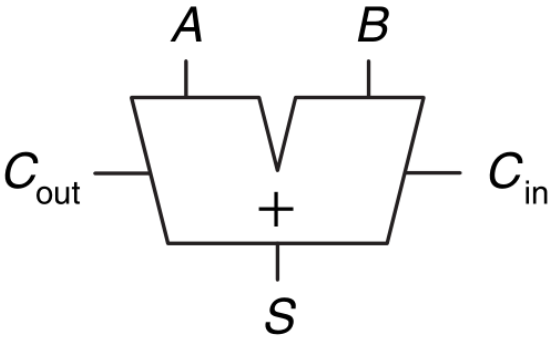
1-bit half adder



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

$$S = A \oplus B$$
$$C_{out} = AB$$

1-bit full adder

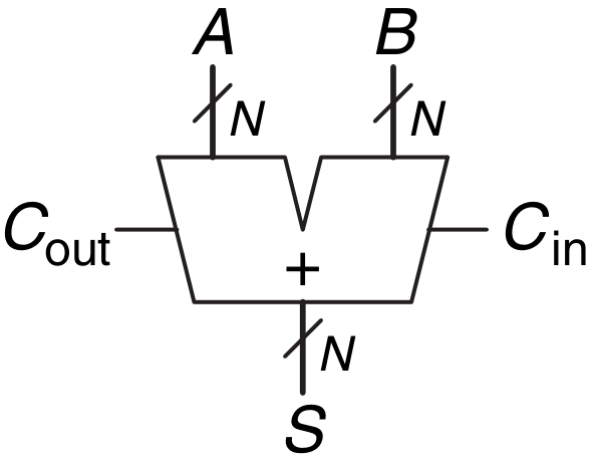


C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

Carry Propagate Adder (CPA)

An *N*-bit adder sums two *N*-bit inputs, *A* and *B*, and a carry in C_{in} to produce an *N*-bit result *S* and a carry out C_{out}

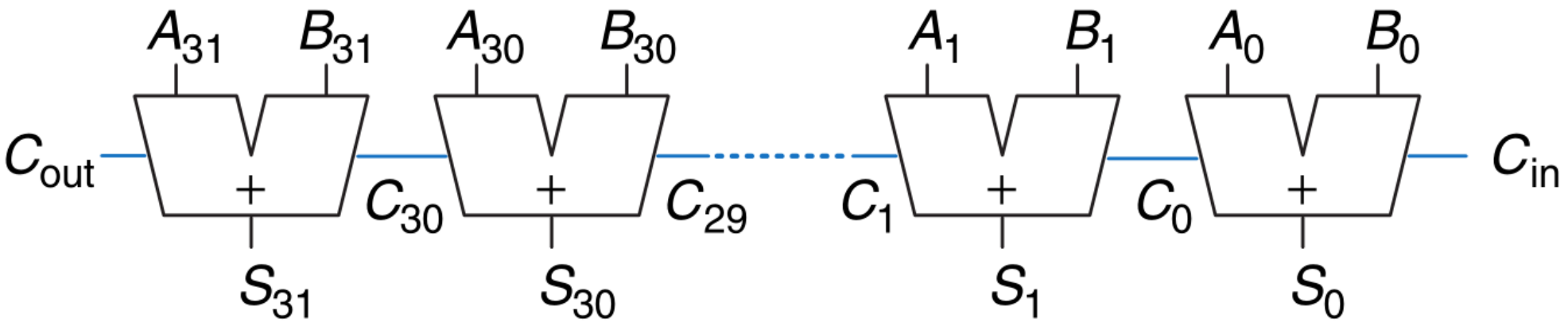


It is called a **carry propagate adder** (CPA) because the carry out of one bit propagates into the next bit.

CPA implementations:

- ripple-carry adders
- carry-lookahead adders
- many other types of adders

Ripple-carry adder (RCA)



Slow when *N* is large

S_{31} depends on C_{30}, which depends on C_{29}, which depends on C_{28}, and so forth all the way back to C_{in}

The delay grows directly with the number of bits

$$t_{RCA} = N\, t_{FA}$$

Carry-lookahead adder (CLA)

CLAs use **generate** (*G*) and **propagate** (*P*) signals that describe how a column or block determines the carry out.

The *i*-th column of an adder is said to **generate** a carry if it produces a carry out independent of the carry in.

$$G_i = A_i B_i$$

The column is said to **propagate** a carry if it produces a carry out whenever there is a carry in.

$$P_i = A_i \oplus B_i$$

The *i*-th column of an adder will generate a carry out C_i if it either generates a carry or propagates a carry in

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

The CLA has a 3-stage structure:

1) Calculate $\forall i \quad G_i = A_i B_i, \quad P_i = A_i \oplus B_i$

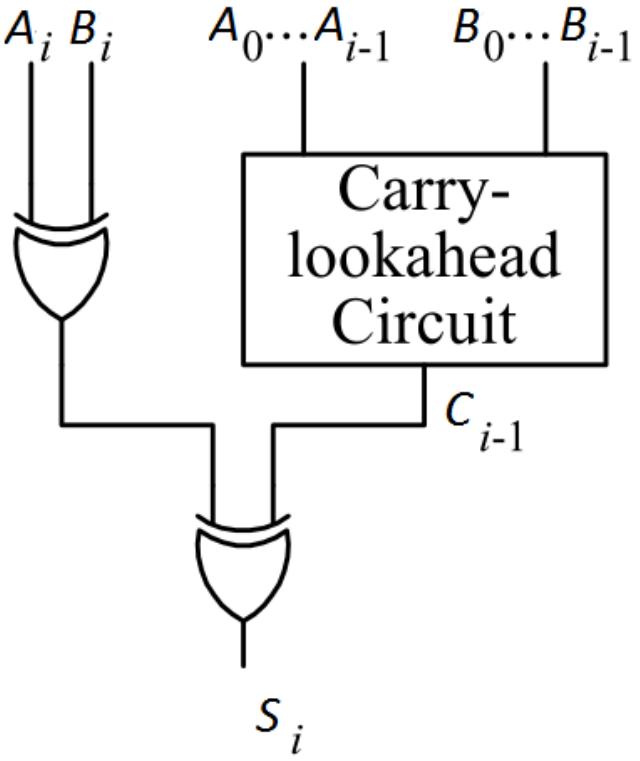
can be done in parallel

2) Calculate carry recursively

$$C_i = G_i + P_i C_{i-1} = G_i + P_i \left(G_{i-1} + P_{i-1} C_{i-2} \right) = \ldots$$

3) Calculate the sum

$$S_i = P_i \oplus C_{i-1}$$



Faster than RCA but still linear in N

Other adders

1960: J. Sklansky – conditional adder

1973: Kogge-Stone adder

1980: Ladner-Fisher adder

1981: H. Ling adder

1982: Brent-Kung adder

1987: Han Carlson adder

1999: S. Knowles

2001: Beaumont-Smith

References for adders

Beaumont-Smith, Cheng-Chew Lim, “Parallel Prefix Adder Design”, IEEE, 2001

Han, Carlson, “Fast Area-Efficient VLSI Adders, IEEE, 1987

Dimitrakopoulos, Nikolos, “High-Speed Parallel-Prefix VLSI Ling Adders”, IEEE 2005

Kogge, Stone, “A Parallel Algorithm for the Efficient solution of a General Class of Recurrence equations”, IEEE, 1973

Simon Knowles, “A Family of adders”, IEEE, 2001

Ladner, Fischer, “Parallel Prefix Computation”, ACM, 1980

Brent, Kung, “A regular Layout for Parallel Adders”, IEEE, 1982

H. Ling, “High-Speed Binary Adder”, IBM J. Res. And Dev., 1980

J. Sklansky, “Conditional-Sum Addition Logic”, IRE transactions on computers, 1960

D. Harris, “A Taxonomy of Parallel Prefix Networks”, IEEE, 2003

Verilog provides the + operation to specify a CPA.

```
module adder #(parameter N = 8)
              (input [N-1:0] a, b,
               input cin,
               output [N-1:0] s,
               output cout);
    assign {cout, s} = a + b + cin;
endmodule
```

Modern synthesis tools select among many possible implementations, choosing the cheapest (smallest) design that meets the speed requirements.

By changing `parameter N = 8` to `parameter N = 16` we can chage the 8 bit adder to a 16 bit adder.

Adder-subtractor combination

Instead of having to build a separate adder and subtractor units, we can modify the adder slightly to perform both operations.

Instead of $A - B$, we perform $A + (-B)$

In addition to the two input operands A and B, a select signal, s, is needed to select which operation to perform.

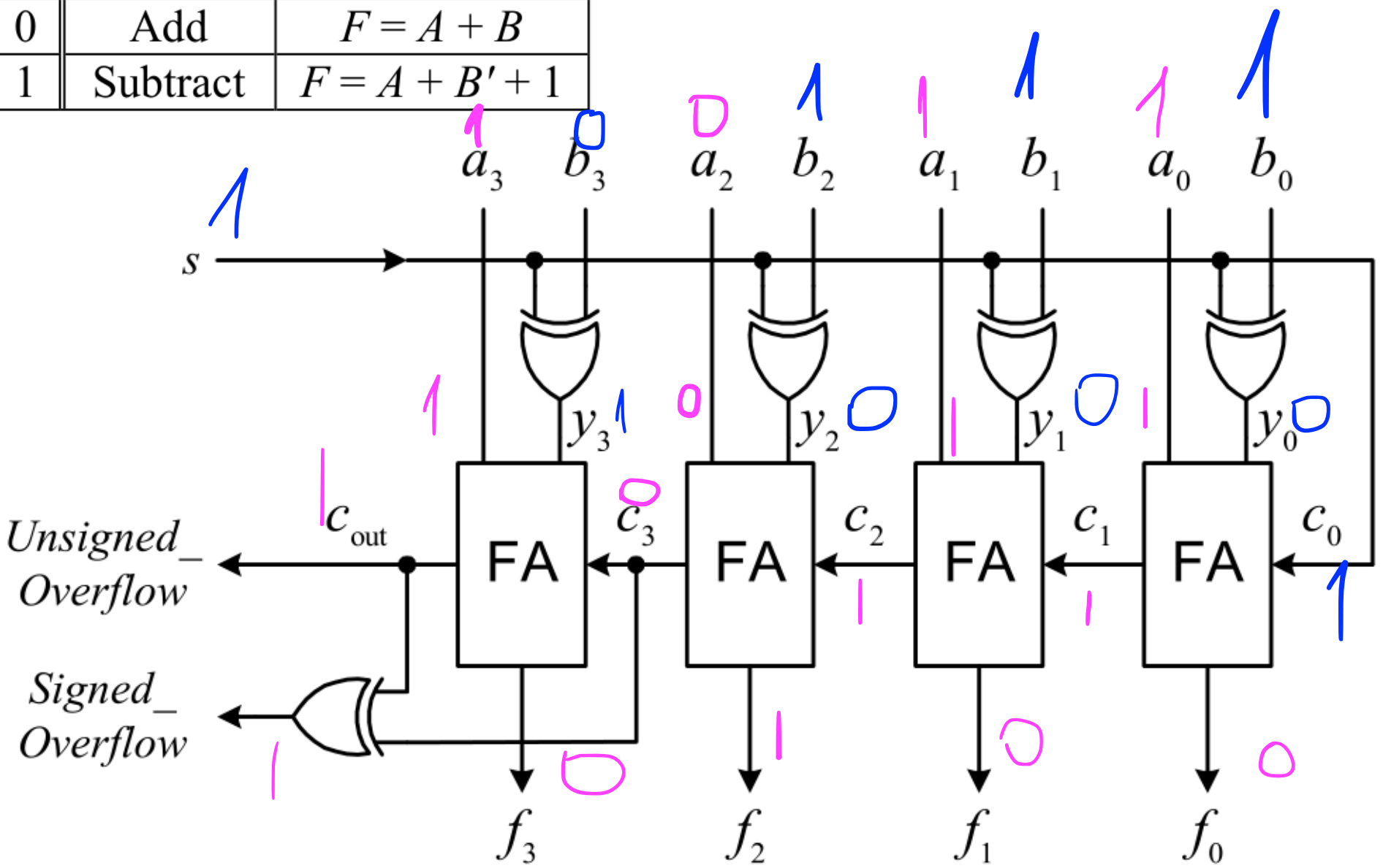
Addition: $s = 0$; Subtraction: $s = 1$

When $s = 0$, B does not need to be modified, and the initial carry-in signal $c0 = 0$.

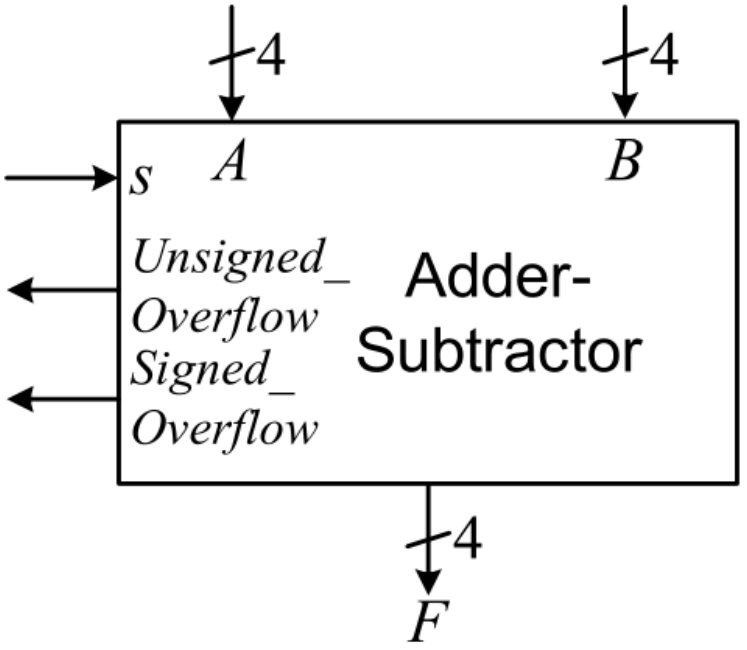
When $s = 1$, we need to invert the bits in B and add a 1. The addition of a 1 is accomplished by setting $c0$ to a 1.

Adder-subtractor combination

s	Function	Operation
0	Add	$F = A + B$
1	Subtract	$F = A + B' + 1$



Adder-subtractor combination



The adder-subtractor circuit has two different overflow signals, Unsigned_Overflow and Signed_Overflow because the circuit can deal with both signed and unsigned numbers.

Verilog subtractor

```
module subtractor #(parameter N = 8)
                   (input [N-1:0] a, b,
                    output [N-1:0] y);
    assign y = a - b;
endmodule
```

Arithmetic Logic Unit (ALU)

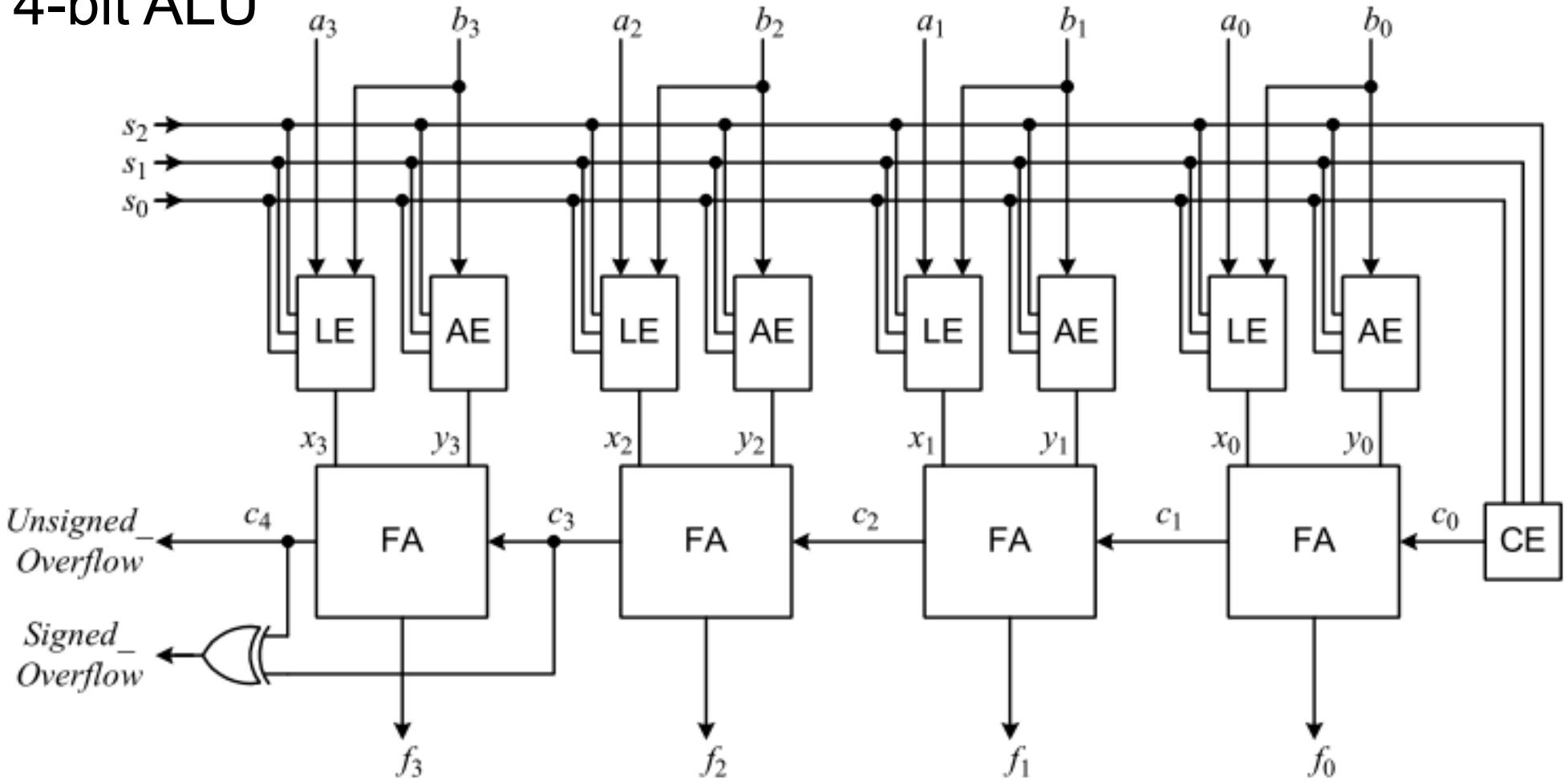
The arithmetic logic unit (ALU) is one of the main components inside a microprocessor.

It is responsible for performing arithmetic and logic operations, such as addition, subtraction, logical AND, and logical OR.

The ALU, however, is not used to perform multiplications or divisions.

Arithmetic Logic Unit (ALU)

4-bit ALU



The logic extender (LE) is for manipulating all logical operations; whereas, the arithmetic extender (AE) is for manipulating all arithmetic operations.

Arithmetic Logic Unit (ALU)

The LE performs the actual logical operations on the two primary operands, a_i and b_i, before passing the result to the first operand, x_i, of the FA.

The AE only modifies the second operand, b_i, and passes it to the second operand, y_i, of the FA where the actual arithmetic operation is performed.

The combinational circuit labeled CE (for carry extender) is for modifying the primary carry-in signal, c_0, so that arithmetic operations are performed correctly.

Logical operations do not use the carry signal, so c_0 is set to 0 for all logical operations.

Arithmetic Logic Unit (ALU)

When arithmetic operations are being performed, the LE must pass the first operand unchanged from the primary input a_i to the output x_i for the FA.

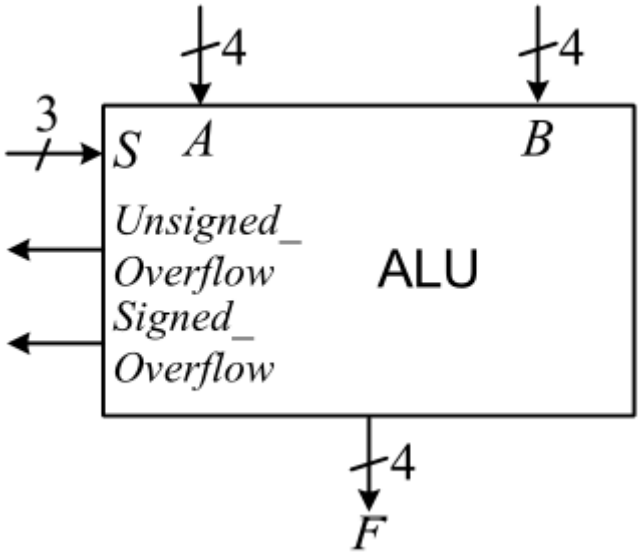
The output of the LE is passed to the first operand, x_i, of the FA.

Since this value is already the result of the logical operation, we do not want the FA to modify it but to simply pass it on to the primary output, f_i.

This is accomplished by setting both the second operand, y_i, of the FA, and c_0 to 0 since adding a 0 will not change the resulting value.

Arithmetic Logic Unit (ALU)

Logic symbol



Three select lines, s2, s1, and s0, are used to select the operations of the ALU.

With these three select lines, the ALU circuit can implement up to eight different operations.

Arithmetic Logic Unit (ALU)

Function table

s_2	s_1	s_0	Operation Name	Operation	x_i (LE)	y_i (AE)	c_0 (CE)
0	0	0	Pass	Pass A to output	a_i	0	0
0	0	1	AND	A AND B	a_i AND b_i	0	0
0	1	0	OR	A OR B	a_i OR b_i	0	0
0	1	1	NOT	A'	a_i'	0	0
1	0	0	Addition	$A + B$	a_i	b_i	0
1	0	1	Subtraction	$A - B$	a_i	b_i'	1
1	1	0	Increment	$A + 1$	a_i	0	1
1	1	1	Decrement	$A - 1$	a_i	1	0

LE truth table

s_2	s_1	s_0	x_i
0	0	0	a_i
0	0	1	$a_i\ b_i$
0	1	0	$a_i + b_i$
0	1	1	a_i'
1	×	×	a_i

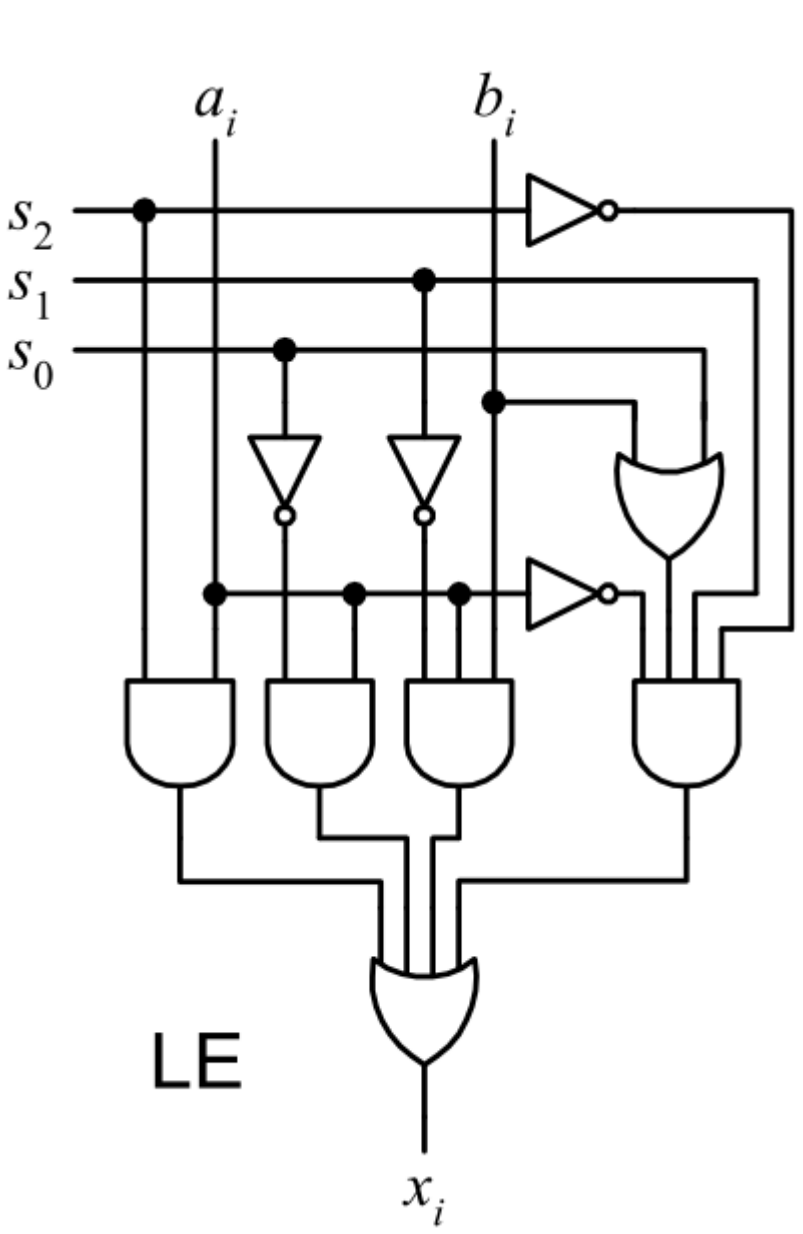
s_2	s_1	s_0	b_i	y_i
0	×	×	×	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

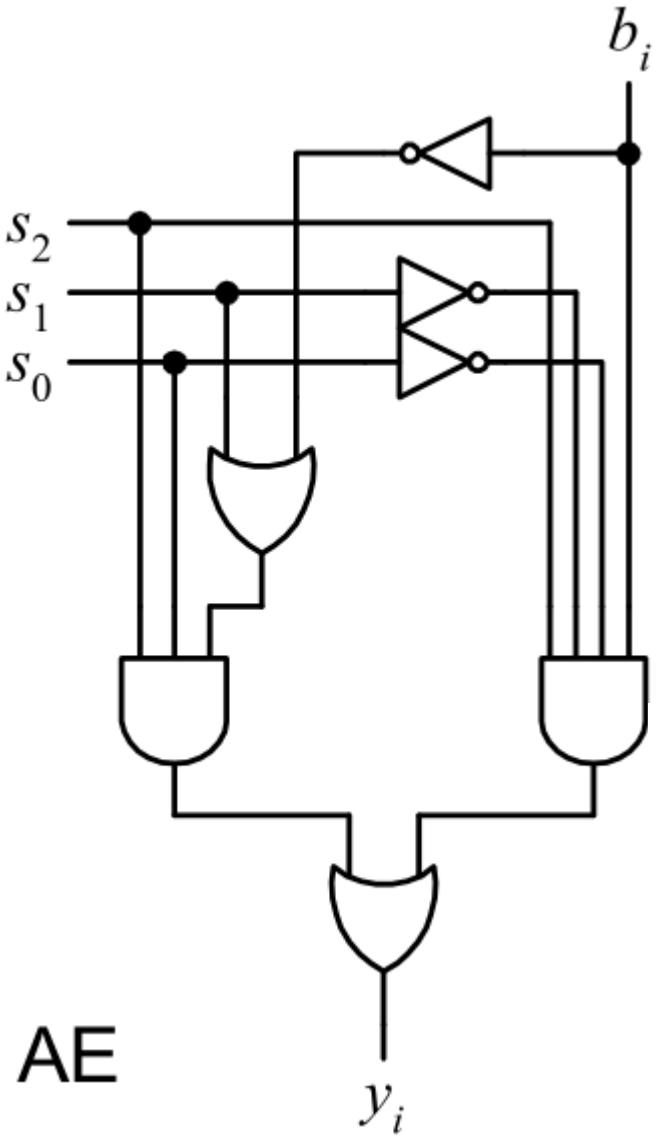
← AE truth table

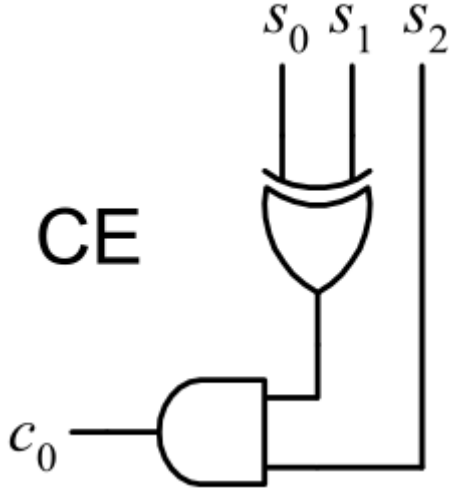
CE truth table

s_2	s_1	s_0	c_0
0	×	×	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Arithmetic Logic Unit (ALU)





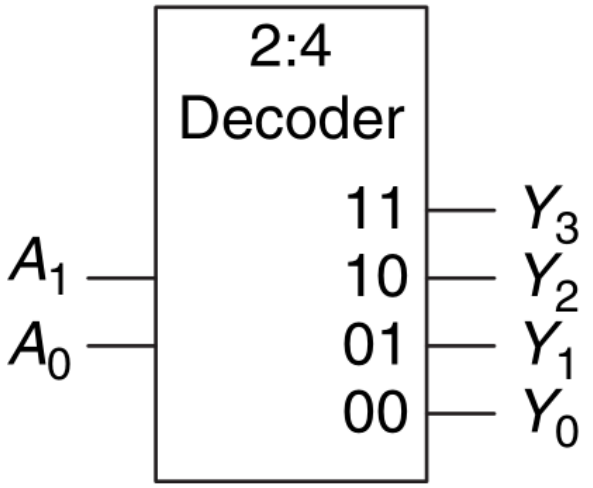


Decoder

also known as a **demultiplexer**

N inputs and 2^N outputs

It asserts exactly one of its outputs depending on the input combination.



A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

The outputs are called **one-hot**, because exactly one is "hot" (HIGH) at a given time.

An *m*-to-*n* decoder has *m* input lines, $A_{m-1}, \ldots, A_0$, and *n* output lines, $Y_{n-1}, \ldots, Y_0$, where $n = 2^m$

A decoder may have an enable line, E.

When the decoder is disabled with E set to 0, all the output lines are de-asserted.

Truth table for a 3-to-8 decoder

E	A_2	A_1	A_0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	×	×	×	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

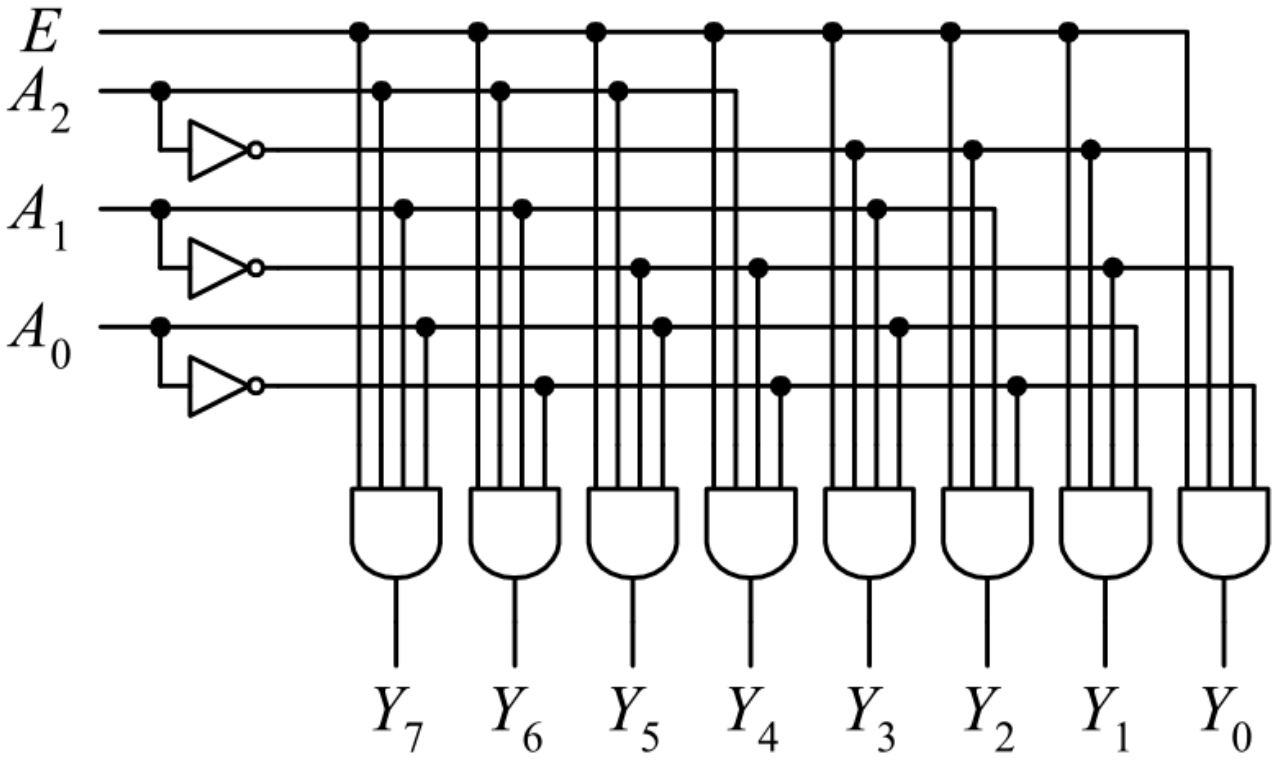
A decoder is used in a system having multiple components, and we want only one component to be selected or enabled at any one time.

For example, in a large memory system with multiple memory chips, only one memory chip is enabled at a time.

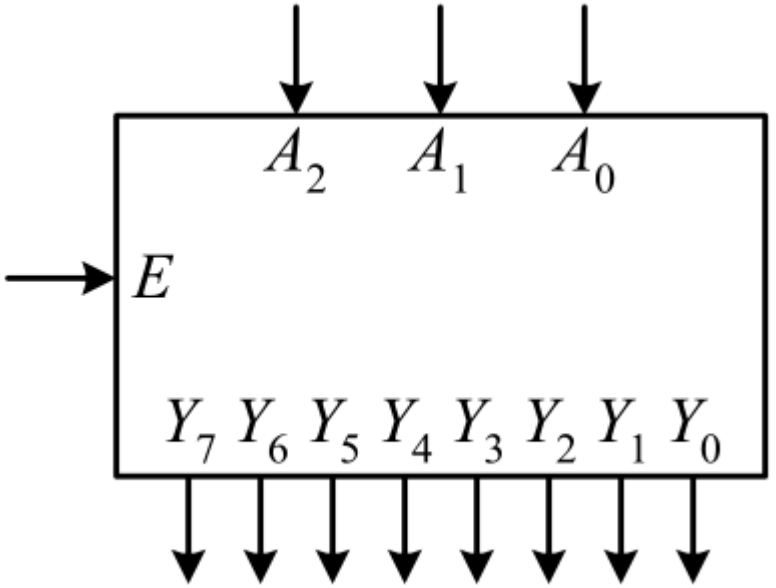
One output line from the decoder is connected to the enable input on each memory chip.

Thus, an address presented to the decoder will enable that corresponding memory chip.

3-to-8 decoder

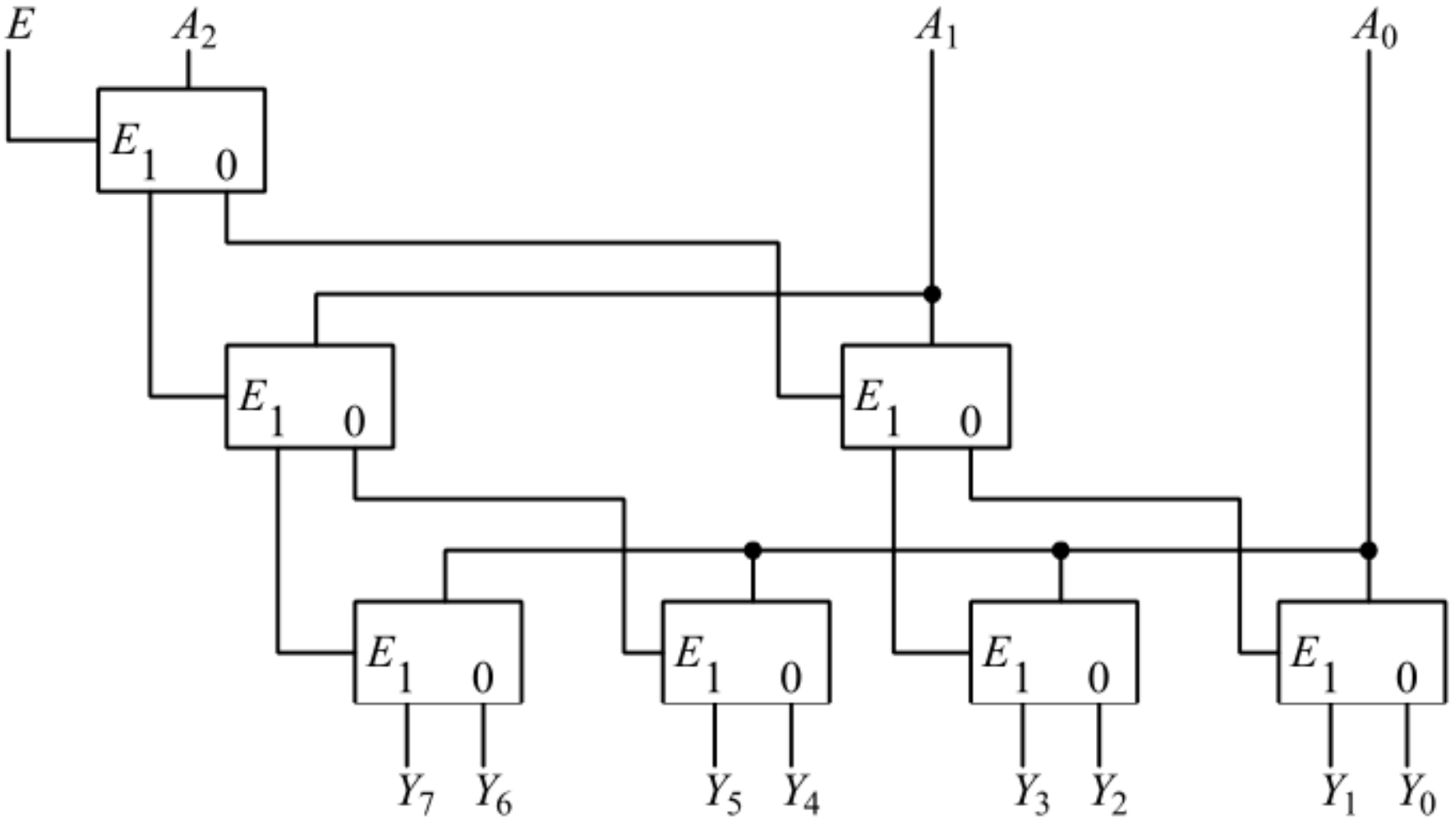


Logic symbol



Decoder

A larger size decoder can be implemented using several smaller decoders



A 3-to-8 decoder implemented with seven 1-to-2 decoders

Encoder

An encoder is almost like the inverse of a decoder where it encodes a 2^n-bit input data into an n-bit code.

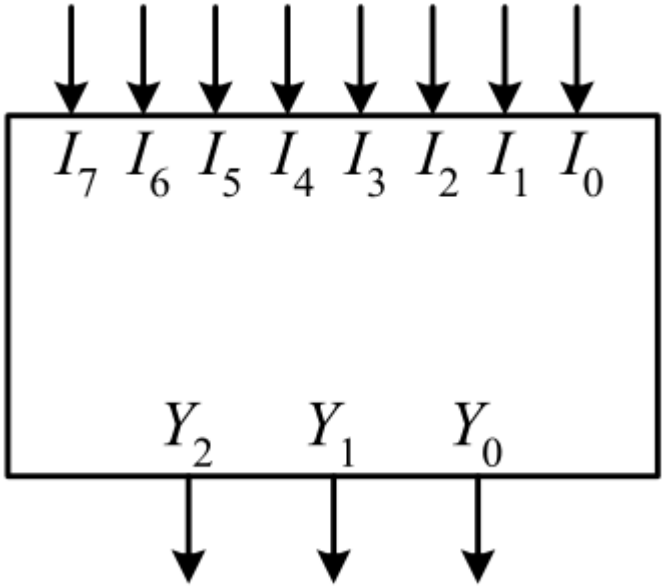
2^n inputs and n outputs

I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Exactly one of the input lines should have a 1 while the remaining input lines should have 0's.

Encoder

The output is the binary value of the index of the input line that has the 1.



Entries having multiple 1's in the truth table inputs are ignored.

Encoders are used to reduce the number of bits needed to represent some given data either in data storage or in data transmission.

Encoders are also used in a system with 2^n input devices, each of which may need to request for service.

One input line is connected to one input device.

Priority encoder

I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	Y_2	Y_1	Y_0	Z
0	0	0	0	0	0	0	0	×	×	×	0
0	0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	0	1	×	0	0	1	1
0	0	0	0	0	1	×	×	0	1	0	1
0	0	0	0	1	×	×	×	0	1	1	1
0	0	0	1	×	×	×	×	1	0	0	1
0	0	1	×	×	×	×	×	1	0	1	1
0	1	×	×	×	×	×	×	1	1	0	1
1	×	×	×	×	×	×	×	1	1	1	1

The table assumes that input I_7 has the highest priority, and I_0 has the lowest priority.

Z is set to a 1 when one or more inputs are asserted; otherwise, Z is set to 0. When Z is 0, all of the Y outputs are meaningless.

Comparator

– compares two binary values

Two types of comparators:

An **equality comparator** produces a single output indicating whether A is equal to B

A **magnitude comparator** produces one or more outputs indicating the relative values of A and B

The equality comparator checks to determine whether the corresponding bits in each column of *A* and *B* are equal using XNOR gates.

Magnitude comparison is usually done by computing *A*−*B* and looking at the sign (most significant bit) of the result.

Verilog comparators

```
module comparators # (parameter N = 8)
                     (input [N-1:0] a, b,
                      output eq, neq,
                      output lt, lte,
                      output gt, gte);

    assign eq = (a == b);
    assign neq = (a != b);
    assign lt = (a < b);
    assign lte = (a <= b);
    assign gt = (a > b);
    assign gte = (a >= b);
endmodule
```

Shifter

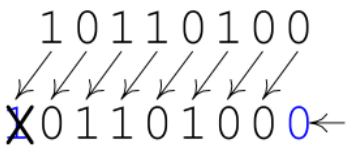
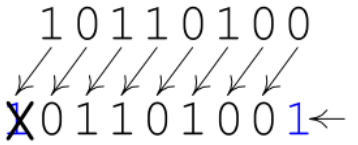
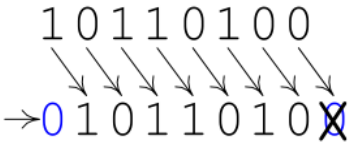
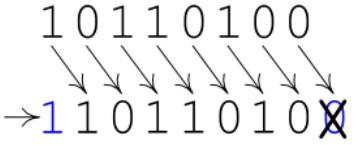
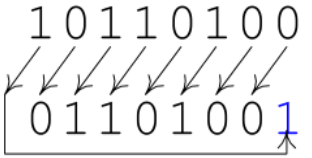
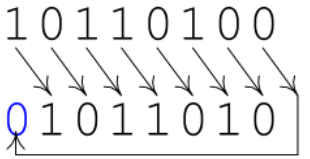
– used for shifting bits in a binary word one position either to the left or to the right.

The operations for the shifter are referred to either as **shifting** or **rotating**, depending on how the end bits are shifted in or out.

For a **shift** operation, the two end bits do not wrap around

For a **rotate** operation, the two end bits wrap around.

Shifter

Operation	Comment	Example
Shift left with 0	Shift bits to the left one position. The leftmost bit is discarded and the rightmost bit is filled with a 0.	
Shift left with 1	Same as above, except that the rightmost bit is filled with a 1.	
Shift right with 0	Shift bits to the right one position. The rightmost bit is discarded and the leftmost bit is filled with a 0.	
Shift right with 1	Same as above, except that the leftmost bit is filled with a 1.	
Rotate left	Shift bits to the left one position. The leftmost bit is moved to the rightmost bit position.	
Rotate right	Shift bits to the right one position. The rightmost bit is moved to the leftmost bit position.	

Shifter

For each bit position, a multiplexer is used to move a bit from either the left or right to the current bit position.

The size of the multiplexer will determine the number of operations that can be implemented.

Example: use a 4-to-1 multiplexer to implement the four operations.

For a 4-bit operand, we will need to use four 4-to-1 multiplexers.

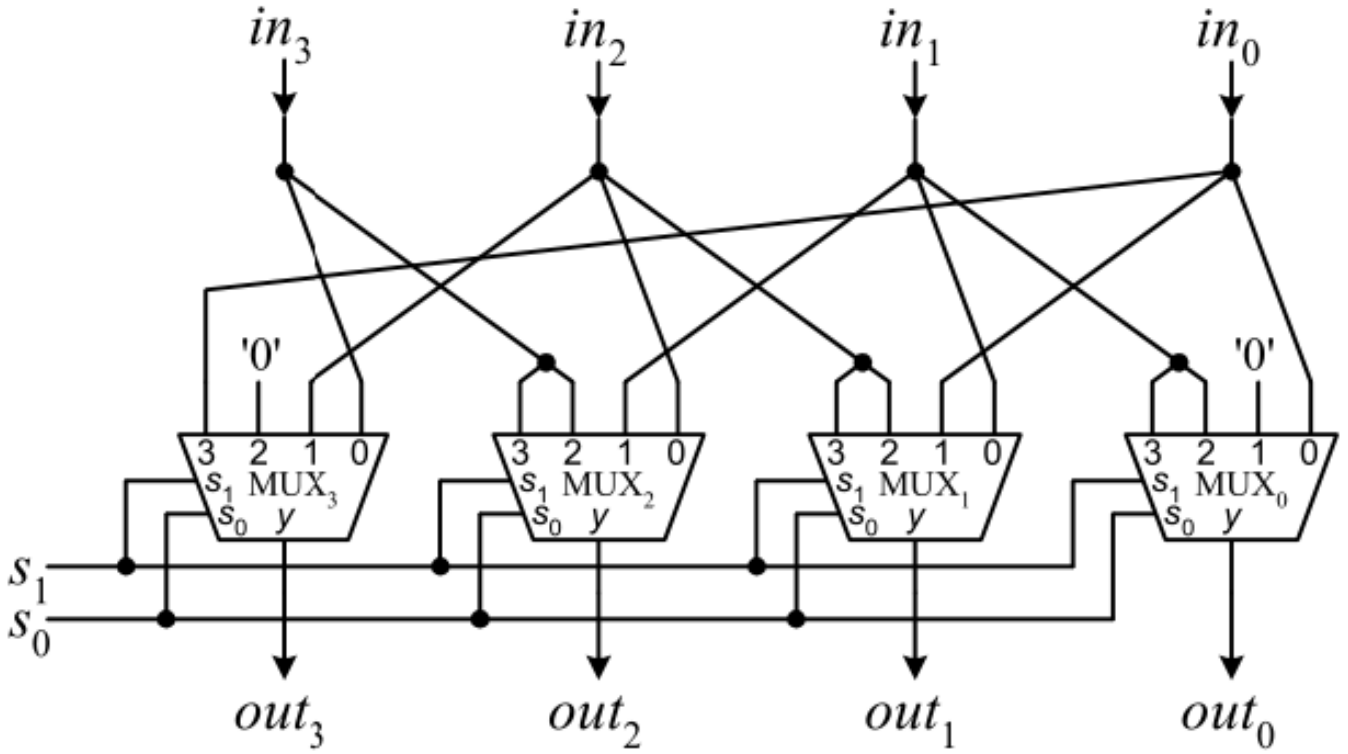
How the inputs to the multiplexers are connected will depend on the given operations.

Shifter

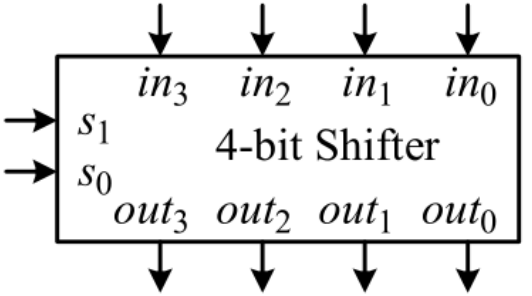
A 4-bit shifter

Operation table

s_1	s_0	Operation
0	0	Pass through
0	1	Shift left and fill with 0
1	0	Shift right and fill with 0
1	1	Rotate right



Logic symbol



Barrel shifter

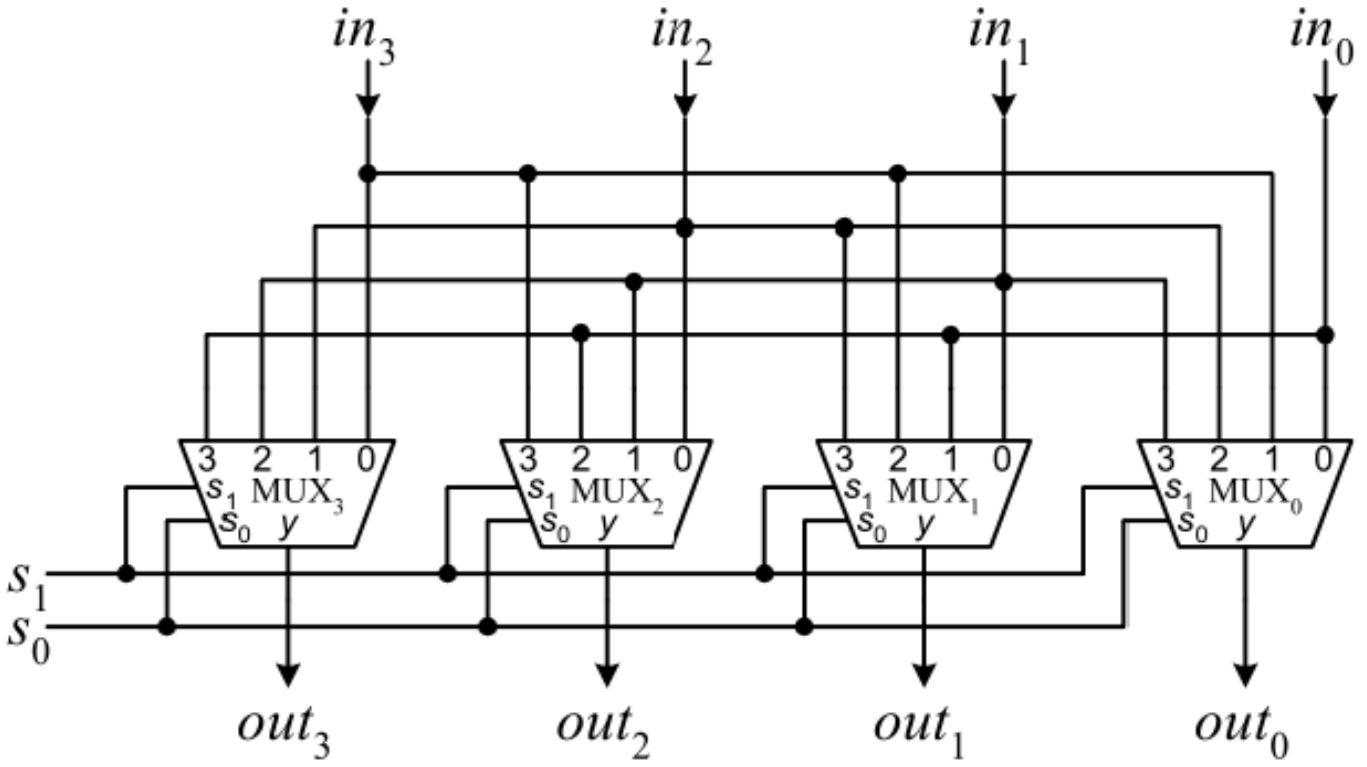
– a shifter that can shift or rotate the data by any number of bits in a single operation.

The select lines for a barrel shifter are used to determine how many bits to move.

An *n*-bit barrel shifter can shift the data bits by as much as $n - 1$ bit distance away in one operation.

Barrel shifter

Select $s_1\ s_0$	Operation	Output $out_3\ out_2\ out_1\ out_0$
00	No rotation	$in_3\ in_2\ in_1\ in_0$
01	Rotate left by 1 bit position	$in_2\ in_1\ in_0\ in_3$
10	Rotate left by 2 bit positions	$in_1\ in_0\ in_3\ in_2$
11	Rotate left by 3 bit positions	$in_0\ in_3\ in_2\ in_1$



Verilog multiplier

Many different multiplier designs with different speed/cost trade-offs exist.

Synthesis tools may pick the most appropriate design given the timing constraints.

```
module multiplier # (parameter N = 8)
                    (input [N-1:0] a, b,
                     output [2*N-1:0] y);
    assign y = a * b;
endmodule
```