Binary formats

Fixed-width binary numbers

Let's assume that we have a predetermined number of bits that we use to express a number: the *bit width*.

How many bits we have will vary with the context, but each number must use exactly that many bits.

$$\mathtt{0010\ 1010}_2$$

$$\mathtt{0000\ 0000\ 0010\ 1010}_2$$

$$\mathtt{0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010\ 1010}_2$$

Fixed-width binary numbers

Let's assume that we have a predetermined number of bits that we use to express a number: the *bit width*.

How many bits we have will vary with the context, but each number must use exactly that many bits.

$\mathtt{0010\ 1010}_2$

42 as an 8-bit number.

$\mathtt{0000\ 0000\ 0010\ 1010}_2$

42 as a 16-bit number.

42 as a 32-bit number.

$\mathtt{0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010\ 1010}_2$

Fixed-width binary numbers

Let's assume that we have a predetermined number of bits that we use to express a number: the *bit width*.

How many bits we have will vary with the context, but each number must use exactly that many bits.

$$0010\ 1010_2$$

most significant bit (MSB)

least significant bit (LSB)

$$0000\ 0000\ 0010\ 1010_2$$

most significant bit (MSB)

least significant bit (LSB)

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010\ 1010_2$$

most significant bit (MSB)

least significant bit (LSB)

Bitwise addition

Bitwise addition

Let's review how to add numbers in decimal.

Decimal addition

```
   144
+   28
```

Bitwise addition

Let's review how to add numbers in decimal.

Decimal addition

```
   144
+   28
```

4 + 8 = 12

Bitwise addition

Let's review how to add numbers in decimal.

Decimal addition

```
    1
  144
+  28
  ---
    2
```

2 is the sum, carry the 1

Bitwise addition

Let's review how to add numbers in decimal.

Decimal addition

```
    1
  144
+  28
  ---
   72
```

4 + 2 + the carried 1 = 7

Bitwise addition

Let's review how to add numbers in decimal.

Decimal addition

```
    1
  144
+  28
  172
```

1 + 0 = 1

Bitwise addition

Binary addition is exactly the same, but in base 2

Binary addition

```
   00010101
+     11100
```

Bitwise addition

Binary addition is exactly the same, but in base 2

Binary addition

```
   00010101
+     11100
-----------
          1
```

1+0=1

Bitwise addition

Binary addition is exactly the same, but in base 2

Binary addition

```
   00010101
+     11100
-----------
         01
```

0+0=0

Bitwise addition

Binary addition is exactly the same, but in base 2

Binary addition

```
        1
   00010101
+     11100
   --------
       001
```

1+1=10
carry the
one

Bitwise addition

Binary addition is exactly the same, but in base 2

Binary addition

```
      11
  00010101
+    11100
  --------
      0001
```

1+0+1=10
carry the
one

Bitwise addition

Binary addition is exactly the same, but in base 2

Binary addition

```
    111
  00010101
+    11100
  --------
    10001
```

1+1+1=11
carry the
one

Bitwise addition

Binary addition is exactly the same, but in base 2

Binary addition

```
    111
  00010101
+    11100
  --------
  110001
```

1+0+0=1

Bitwise addition

Binary addition is exactly the same, but in base 2

Binary addition

```
    111
  00010101
+    11100
  --------
   0110001
```

0+0+0=0

Bitwise addition

Binary addition is exactly the same, but in base 2

Binary addition

```
    111
  00010101
+    11100
  --------
  00110001
```

0+0+0=0

Negative numbers

The binary numbers we've used so far are *unsigned*: the have no sign, positive or negative, and are therefore assumed to always be positive.

$$\mathbf{42_{10} = 0010\ 1010_2}$$

Negative numbers

How shall we express negative numbers
using only a fixed number of bits?

$$42_{10} = 0010\ 1010_2$$

$$-42_{10} = ????\ ????_2$$

Negative numbers

How shall we express negative numbers using only a fixed number of bits?

Here's a proposal: let's reserve the most-significant to express the sign: 0 for positive, 1 for negative.

$$42_{10} = 0010\ 1010_2$$

$$-42_{10} = 1010\ 1010_2$$

Negative numbers

How shall we express negative numbers using only a fixed number of bits?

Here's a proposal: let's reserve the most-significant to express the sign: 0 for positive, 1 for negative. The remaining bits represent the number's magnitude.

This is *signed magnitude* representation.

$$42_{10} = 0010\ 1010_2$$

MSB=0
positive number

$$-42_{10} = 1010\ 1010_2$$

MSB=1
negative number

Negative numbers

How shall we express negative numbers using only a fixed number of bits?

Here's a proposal: let's reserve the most-significant to express the sign: 0 for positive, 1 for negative. The remaining bits represent the number's magnitude.

This is *signed magnitude* representation.

Problem #1

$$0_{10} = -0_{10} = 00000000_2 = 10000000_2$$

Zero has two representations.

Negative numbers

How shall we express negative numbers using only a fixed number of bits?

Here's a proposal: let's reserve the most-significant to express the sign: 0 for positive, 1 for negative. The remaining bits represent the number's magnitude.

This is *signed magnitude* representation.

Problem #1

$$0_{10} = -0_{10} = 00000000_2 = 10000000_2$$

Zero has two representations.

Problem #2

$$3_{10} = 0000\ 0011_2$$
$$-3_{10} = 1000\ 0011_2$$

Negative numbers

How shall we express negative numbers using only a fixed number of bits?

Here's a proposal: let's reserve the most-significant to express the sign: 0 for positive, 1 for negative. The remaining bits represent the number's magnitude.

This is *signed magnitude* representation.

Problem #1

$$0_{10} = -0_{10} = 00000000_2 = 10000000_2$$

Zero has two representations.

Problem #2

$$\begin{array}{rl} & 3_{10} = 0000\ 0011_2 \\ + & -3_{10} = 1000\ 0011_2 \\ \hline & -6_{10} = 1000\ 0110_2 \end{array}$$

Addition doesn't work

Negative numbers

Signed magnitude has serious disadvantages. Let's choose another way to represent negative numbers.

The new proposal says this: if you have a binary number, and you want to find the binary representation of its negation, (a) invert all the bits and then (b) add one.

This is *2's complement* representation.

Negative numbers

Signed magnitude has serious disadvantages. Let's choose another way to represent negative numbers.

The new proposal says this: if you have a binary number, and you want to find the binary representation of its negation, (a) invert all the bits and then (b) add one.

This is *2's complement* representation.

Example. We want to find the 8-bit 2's complement representation of -42.

We start with the 8-bit representation of 42.

0010 1010

Negative numbers

Signed magnitude has serious disadvantages. Let's choose another way to represent negative numbers.

The new proposal says this: if you have a binary number, and you want to find the binary representation of its negation, (a) invert all the bits and then (b) add one.

This is *2's complement* representation.

Example. We want to find the 8-bit 2's complement representation of -42.

We start with the 8-bit representation of 42.

0010 1010

Step 1

We invert all the bits.

Negative numbers

Signed magnitude has serious disadvantages. Let's choose another way to represent negative numbers.

The new proposal says this: if you have a binary number, and you want to find the binary representation of its negation, (a) invert all the bits and then (b) add one.

This is *2's complement* representation.

Example. We want to find the 8-bit 2's complement representation of -42.

We start with the 8-bit representation of 42.

0010 1010

Step 1

We invert all the bits.

1101 0101

Negative numbers

Signed magnitude has serious disadvantages. Let's choose another way to represent negative numbers.

The new proposal says this: if you have a binary number, and you want to find the binary representation of its negation, (a) invert all the bits and then (b) add one.

This is *2's complement* representation.

Example. We want to find the 8-bit 2's complement representation of -42.

We start with the 8-bit representation of 42.

0010 1010

Step 1

We invert all the bits.

1101 0101

Step 2

We add one.

Negative numbers

Signed magnitude has serious disadvantages. Let's choose another way to represent negative numbers.

The new proposal says this: if you have a binary number, and you want to find the binary representation of its negation, (a) invert all the bits and then (b) add one.

This is *2's complement* representation.

Example. We want to find the 8-bit 2's complement representation of -42.

We start with the 8-bit representation of 42.

0010 1010

Step 1

We invert all the bits.

1101 0101

Step 2

We add one.

1101 0110

Negative numbers

Signed magnitude has serious disadvantages. Let's choose another way to represent negative numbers.

The new proposal says this: if you have a binary number, and you want to find the binary representation of its negation, (a) invert all the bits and then (b) add one.

This is *2's complement* representation.

Another example. Let's say I give you the 8-bit binary number 0000 0011. What is the value in decimal?

Negative numbers

Signed magnitude has serious disadvantages. Let's choose another way to represent negative numbers.

The new proposal says this: if you have a binary number, and you want to find the binary representation of its negation, (a) invert all the bits and then (b) add one.

This is *2's complement* representation.

Another example. Let's say I give you the 8-bit binary number 0000 0011. What is the value in decimal?

2's complement numbers have the property that the most-significant bit will be zero for non-negative numbers. Therefore this number is positive 3.

Negative numbers

Signed magnitude has serious disadvantages. Let's choose another way to represent negative numbers.

The new proposal says this: if you have a binary number, and you want to find the binary representation of its negation, (a) invert all the bits and then (b) add one.

This is *2's complement* representation.

Another example. Let's say I give you the 8-bit binary number 11111101. What is the value in decimal?

Negative numbers

Signed magnitude has serious disadvantages. Let's choose another way to represent negative numbers.

The new proposal says this: if you have a binary number, and you want to find the binary representation of its negation, (a) invert all the bits and then (b) add one.

This is *2's complement* representation.

Another example. Let's say I give you the 8-bit binary number 11111101. What is the value in decimal?

Now we know it's a negative number, because the MSB is 1. To find its magnitude, we apply the procedure again:

1111 1101

Step 1

We invert all the bits.

Negative numbers

Signed magnitude has serious disadvantages. Let's choose another way to represent negative numbers.

The new proposal says this: if you have a binary number, and you want to find the binary representation of its negation, (a) invert all the bits and then (b) add one.

This is *2's complement* representation.

Another example. Let's say I give you the 8-bit binary number 11111101. What is the value in decimal?

Now we know it's a negative number, because the MSB is 1. To find its magnitude, we apply the procedure again:

1111 1101

Step 1

We invert all the bits.

0000 0010

Negative numbers

Signed magnitude has serious disadvantages. Let's choose another way to represent negative numbers.

The new proposal says this: if you have a binary number, and you want to find the binary representation of its negation, (a) invert all the bits and then (b) add one.

This is *2's complement* representation.

Another example. Let's say I give you the 8-bit binary number 11111101. What is the value in decimal?

Now we know it's a negative number, because the MSB is 1. To find its magnitude, we apply the procedure again:

1111 1101

Step 1

We invert all the bits.

0000 0010

Step 2

We add one.

0000 0011

Negative numbers

Signed magnitude has serious disadvantages. Let's choose another way to represent negative numbers.

The new proposal says this: if you have a binary number, and you want to find the binary representation of its negation, (a) invert all the bits and then (b) add one.

This is *2's complement* representation.

Another example. Let's say I give you the 8-bit binary number 11111101. What is the value in decimal?

Now we know it's a negative number, because the MSB is 1. To find its magnitude, we apply the procedure again:

1111 1101

Step 1

We invert all the bits.

0000 0010

Step 2

We add one.

0000 0011

This number is 3 in decimal.
We can conclude:
-3 = **11111101**

Negative numbers

Signed magnitude has serious disadvantages. Let's choose another way to represent negative numbers.

The new proposal says this: if you have a binary number, and you want to find the binary representation of its negation, (a) invert all the bits and then (b) add one.

This is *2's complement* representation.

Does 2's complement solve the problems with signed magnitude?

??Problem #1

$$0_{10} = -0_{10} = ??????????_2$$

Does zero have two representations?

??Problem #2

$$\begin{array}{rl} & 3_{10} = 0000\ 0011_2 \\ + & -3_{10} = 1111\ 1101_2 \\ \hline & ???????? \end{array}$$

Does addition work?

What is 14 in 8-bit 2's complement?

What is -14 in 8-bit 2's complement?

What is 14 in 16-bit 2's complement?

What is -14 in 16-bit 2's complement?

What is 14 in 8-bit 2's complement?

00001110

What is -14 in 8-bit 2's complement?

11110010

What is 14 in 16-bit 2's complement?

0000000000001110

What is -14 in 16-bit 2's complement?

1111111111110010

What is the largest number that can be expressed as 8-bit 2's complement?

What is the smallest number that can be expressed as 8-bit 2's complement?

What is the largest number that can be expressed as 8-bit unsigned?

What is the smallest number that can be expressed as 8-bit unsigned?

What is the largest number that can be expressed as 8-bit 2's complement?

0111 1111 = 127

What is the smallest number that can be expressed as 8-bit 2's complement?

1000 0000 = -128

What is the largest number that can be expressed as 8-bit unsigned?

1111 1111 = 255

What is the smallest number that can be expressed as 8-bit unsigned?

0000 0000 = 0

Binary arithmetic

Binary arithmetic examples

$$
\begin{array}{rr}
 & A_2A_1A_0 \\
+ & B_2B_1B_0 \\
\hline
 & S_2S_1S_0
\end{array}
$$

The sum of two 3-bit numbers will also be a 3-bit number. Carry out is discarded.

Binary arithmetic examples

```
  001
+ 011
-------
  ???
```

Binary arithmetic examples

```
  001
+ 011
-------
  100
```

In unsigned decimal: 1 + 3 = 4

Binary arithmetic examples

```
  011
+ 011
-------
  110
```

In unsigned decimal: 3 + 3 = 6

Binary arithmetic examples

```
  111
+ 010
-------
```

??????????????????

Binary arithmetic examples

```
  111
+ 010
-----
 1001
```

????????????????

Binary arithmetic examples

```
  111
+ 010
-----
 1001
```

Last bit is carry out, not part of the sum. It is ignored.

The arithmetic sum is greater than the maximum value that can be output by our adder. This is called *overflow*. We truncate the overflow bit, so the result of this sum is just 001.

Binary arithmetic examples

```
  111
+ 010
-------
  001
```

In unsigned decimal, this is 7 + 2 = 1.

This is the correct answer, despite its apparent weirdness.

As a consequence of overflow, the arithmetic exhibits *wrap-around*: once we reach the maximum expressible value, the sum wraps to the minimum expressible value: adding one to 111 yields zero.

Binary arithmetic examples

```
  111
+ 010
-------
  001
```

There is an alternative way to interpret this sum.

We can interpret these numbers as 2's complement. In that case, a one in the most significant bit indicates a negative value.

Binary arithmetic examples

```
  111
+ 010
-------
  001
```

There is an alternative way to interpret this sum.

We can interpret these numbers as 2's complement. In that case, a one in the most significant bit indicates a negative value.

In 2's complement decimal, this is -1 + 2 = 1.

Note that regardless of whether we interpret the numbers as unsigned or 2's complement, the sum is the same, the adder works the same, and the addition algorithm works the same.

Binary arithmetic examples

```
  111
+ 101
-------
  100
```

Another example.

In unsigned decimal, this is 7 + 5 = 4.
In 2's complement decimal, this is -1 + -3 = -4.

Binary arithmetic examples

```
  011
+ 011
-------
  110
```

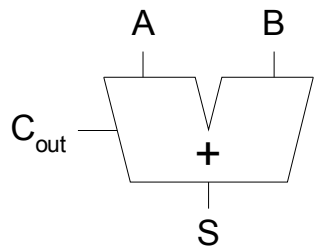
Another example.

In unsigned decimal, this is 3 + 3 = 6.
In 2's complement decimal, this is 3 + 3 = -2.

Adders

1-Bit Adders

Half Adder

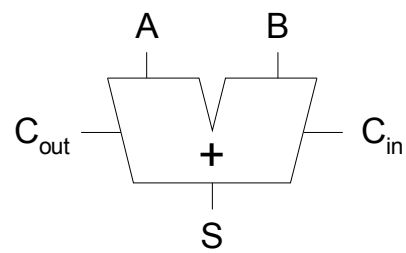


A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S = A \oplus B$

$C_{out} = AB$

Full Adder

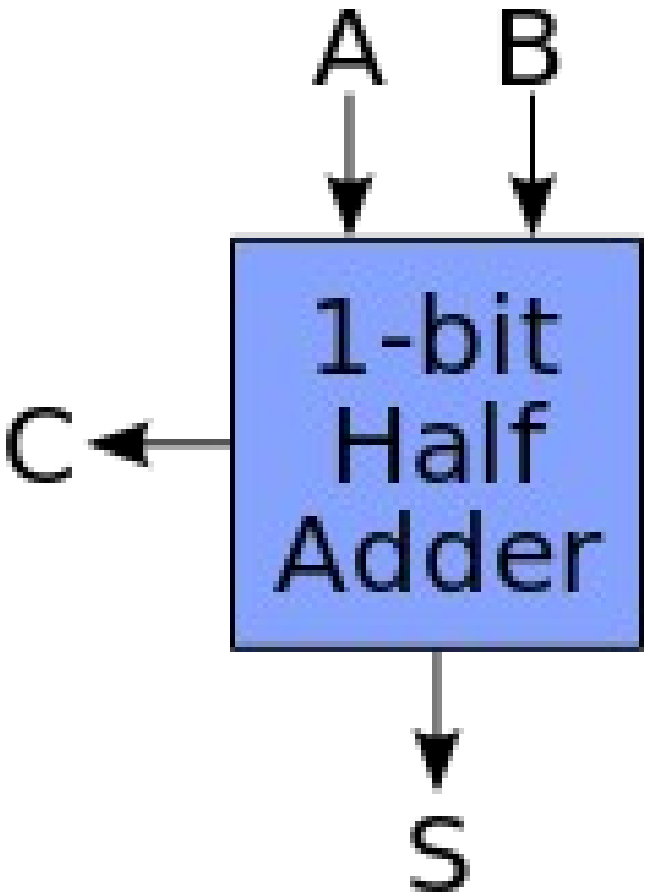


C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

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

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

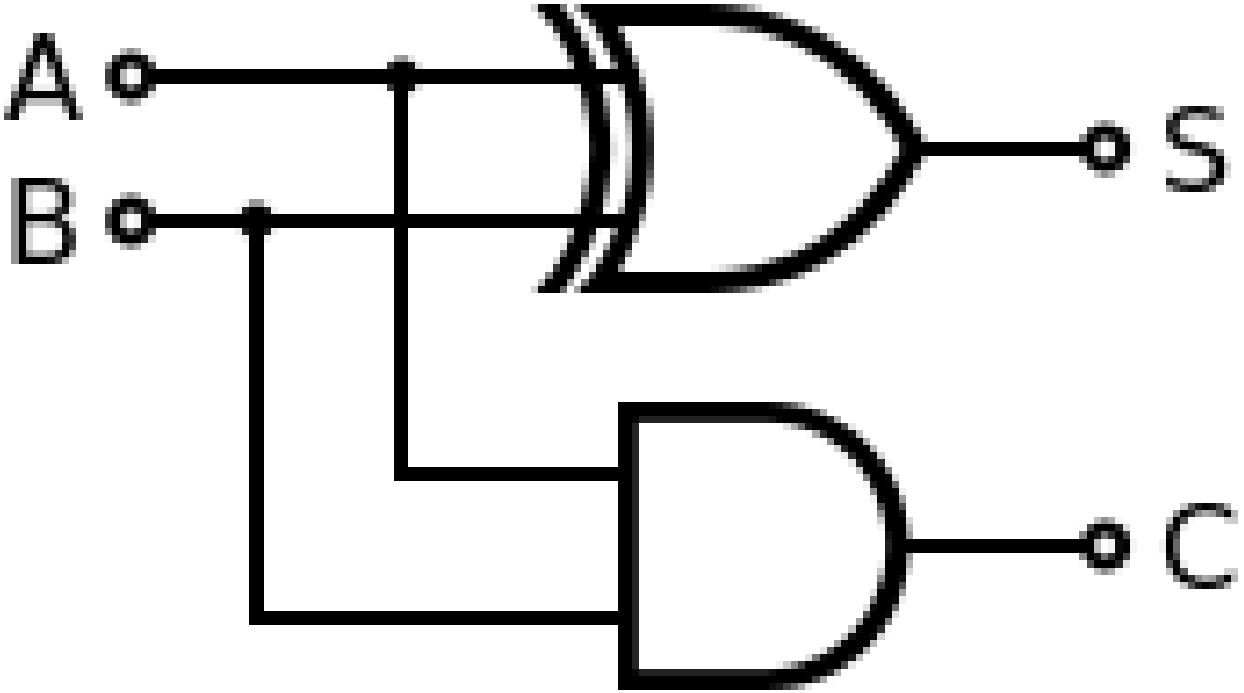
Half adder



Half adder: truth table

Inputs		Outputs	
A	B	C	S
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	0

Half adder: schematic

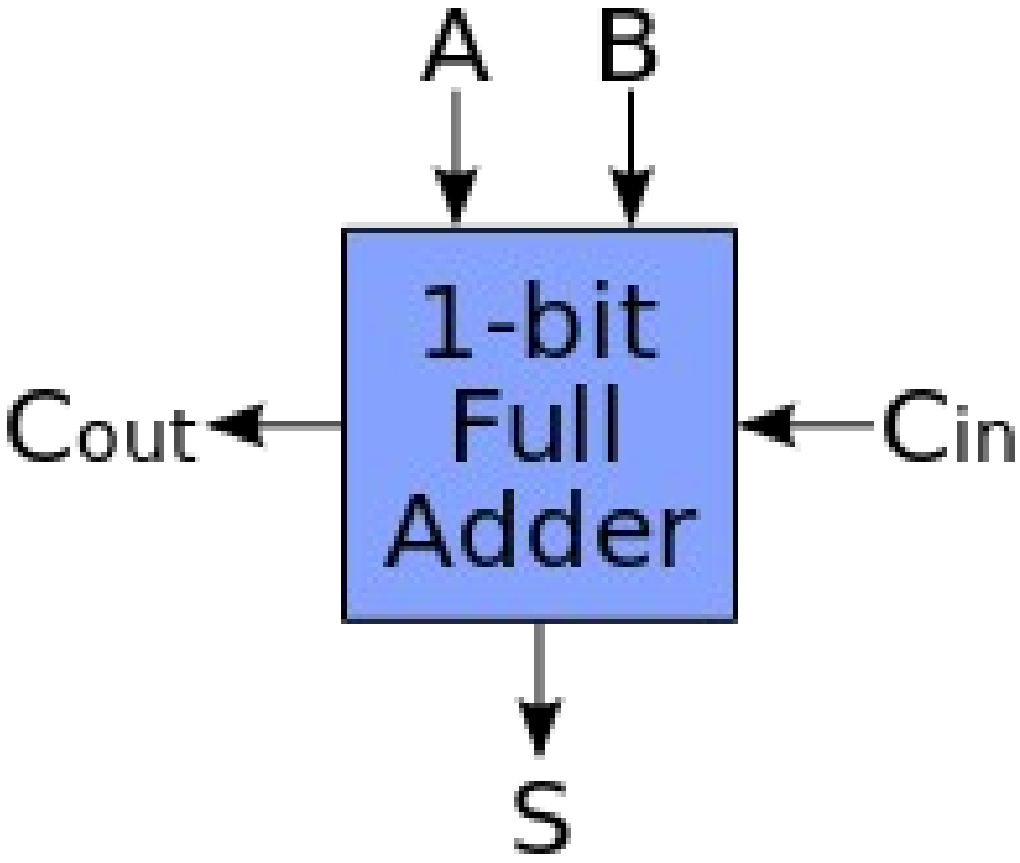


Half adder equation

S = A xor B

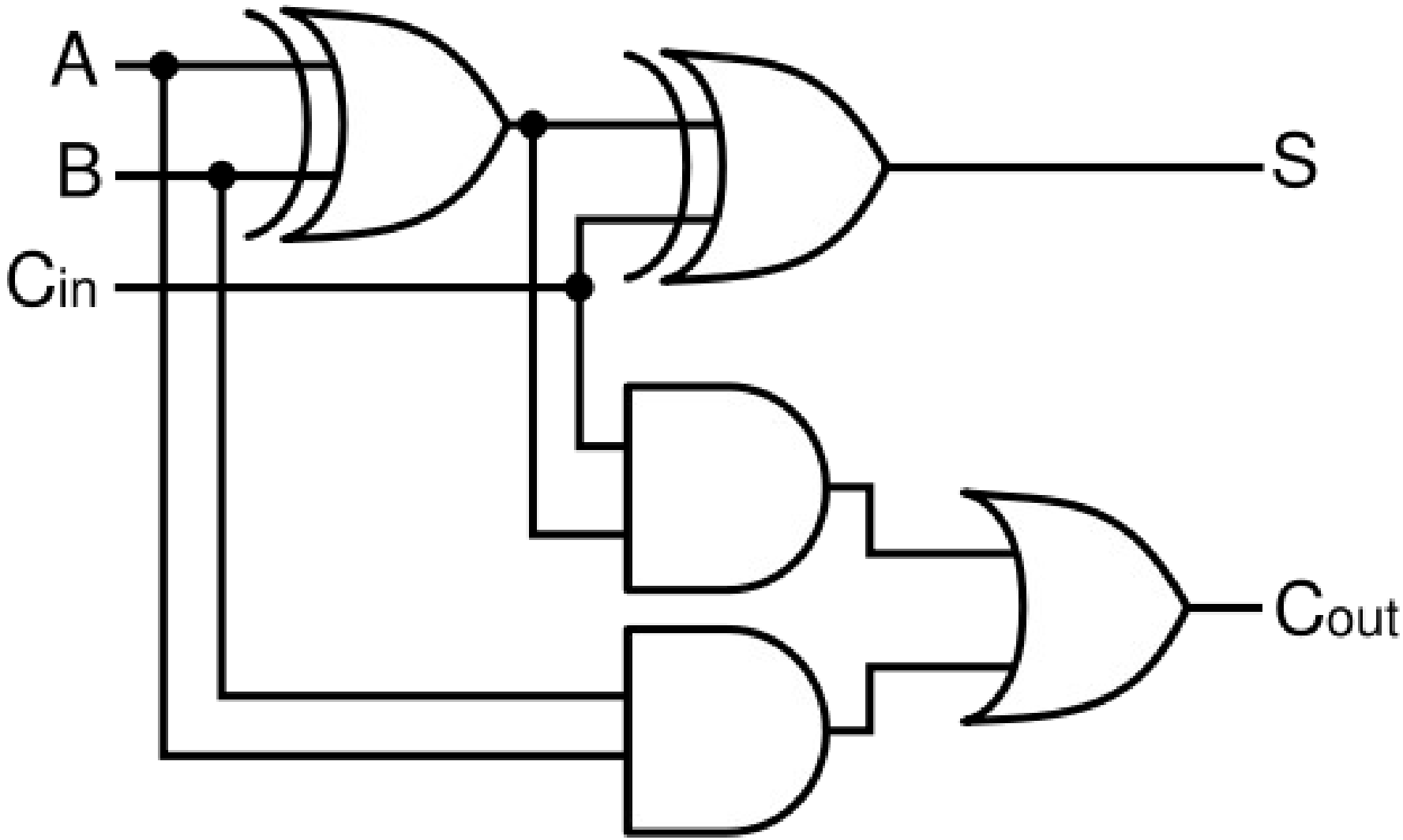
C = A and B

Full adder



Full adder: truth table

Inputs			Outputs	
A	**B**	**C_{in}**	**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



Schematic for Full Adder

Full adder equation

S = A xor B xor Cin
Cout = (A and B) or (A and Cin) or (B and Cin)

equivalently:

Cout = (A and B) or (Cin and (A or B))

Synthesis

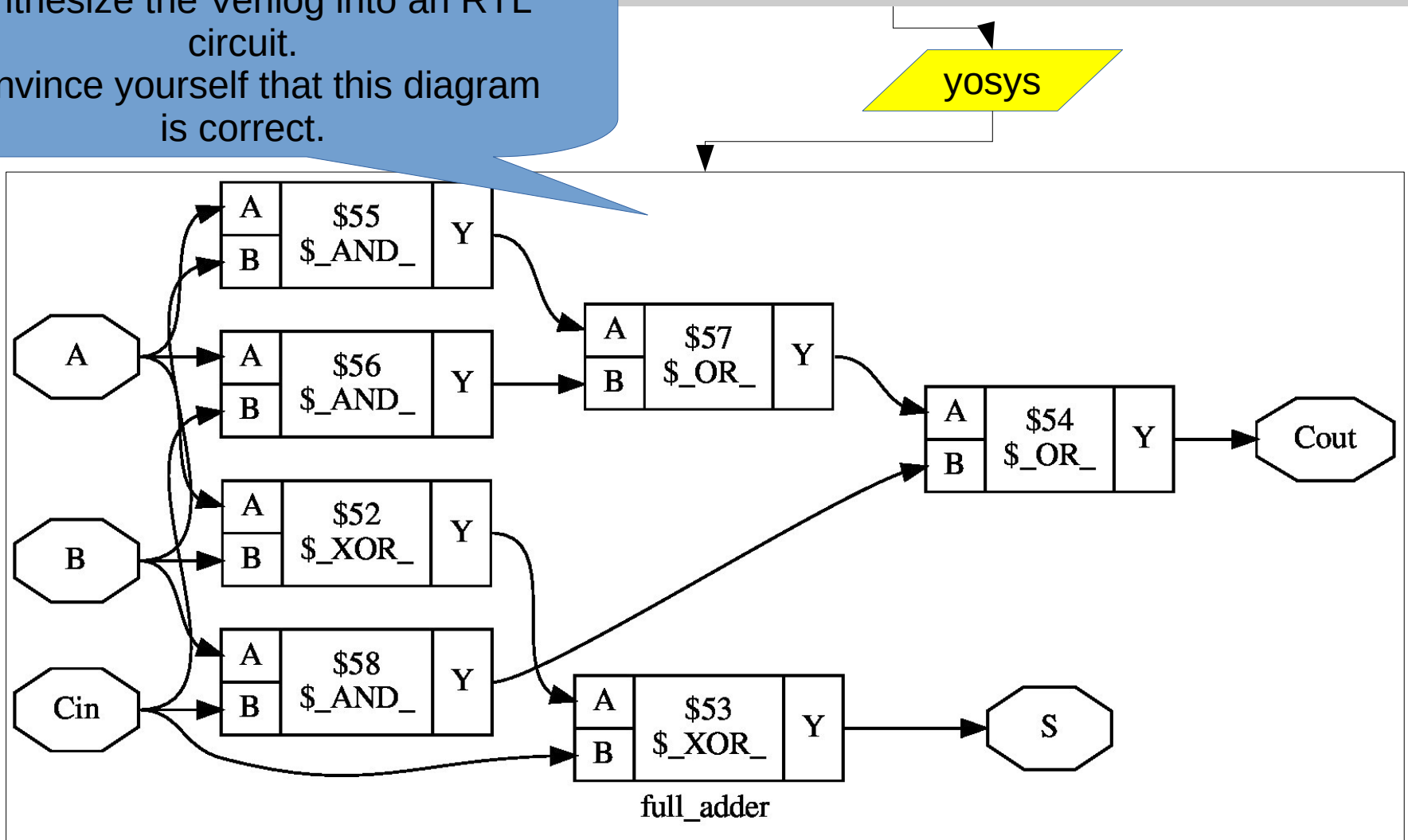
```
module full_adder(A, B, Cin, S, Cout);

    input A, B, Cin;
    output S, Cout;

    assign S = A ^ B ^ Cin;
    assign Cout = (A & B) | (A & Cin) | (B & Cin);

endmodule
```

The formula for the full adder can be easily expressed in Verilog.

Synthesis

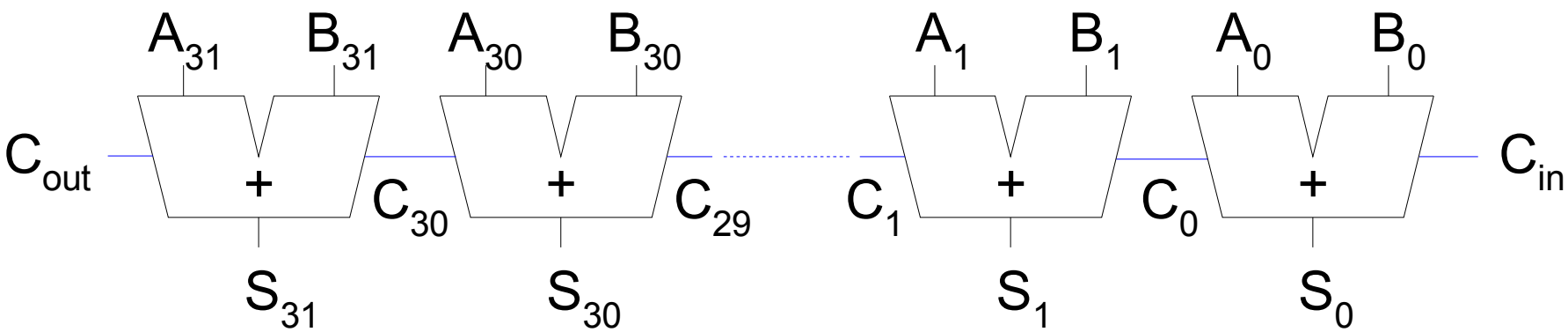
```
module full_adder(A, B, Cin, S, Cout);

    input A, B, Cin;
    output S, Cout;

    assign S = A ^ B ^ Cin;
    assign Cout = (A & B) | (A & Cin) | (B & Cin);
```

As before, we can use yosys to synthesize the Verilog into an RTL circuit.
Convince yourself that this diagram is correct.



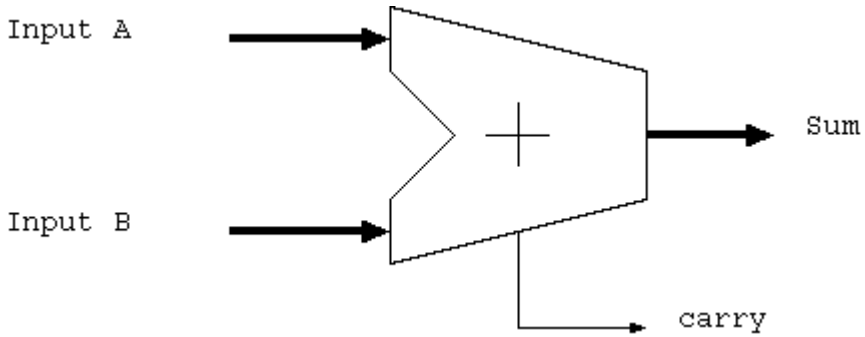
Ripple-Carry Adder

- Chain 1-bit adders together
- Carry ripples through entire chain
- Disadvantage: **slow**

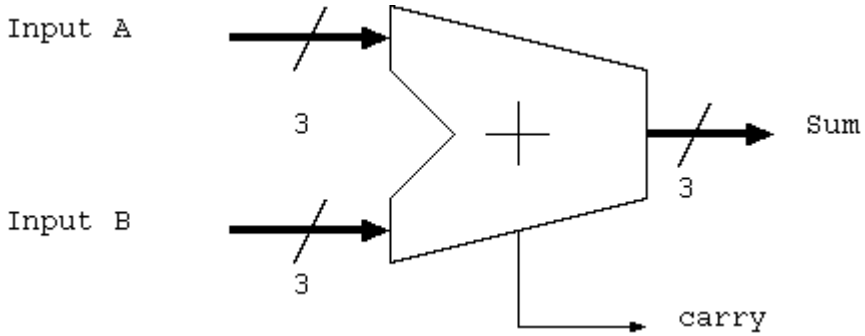


Delay equals N times delay of a Full Adder
Size equals N times size of a Full Adder

1-bit adder

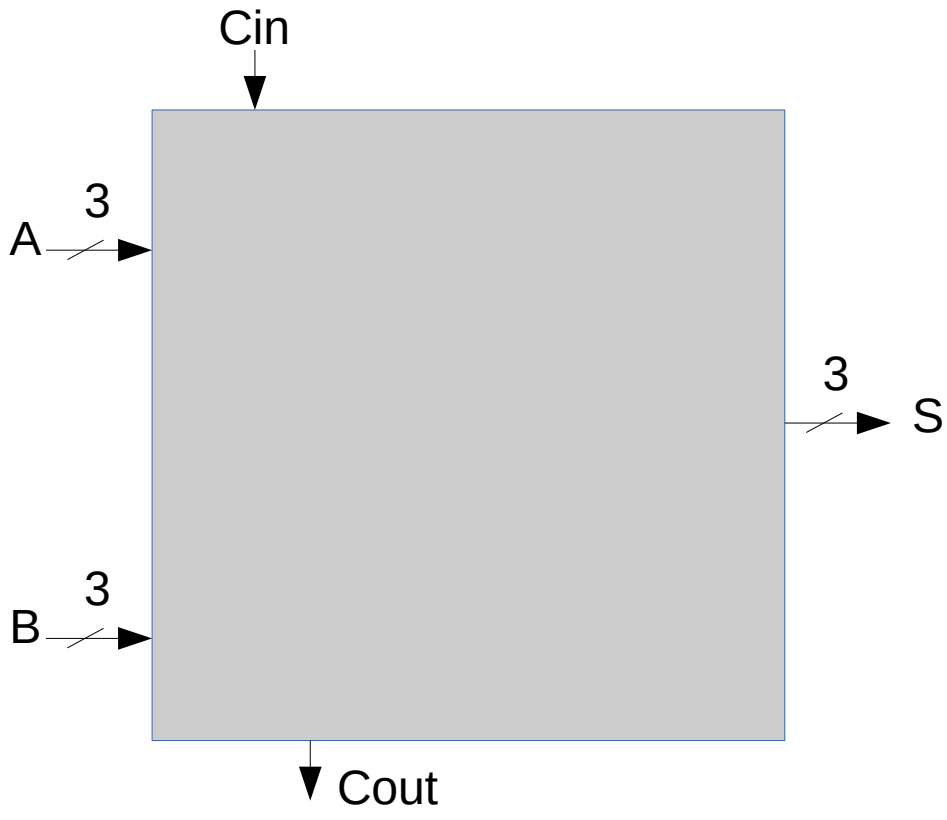


3-bit adder



Let's make a 3-bit adder!

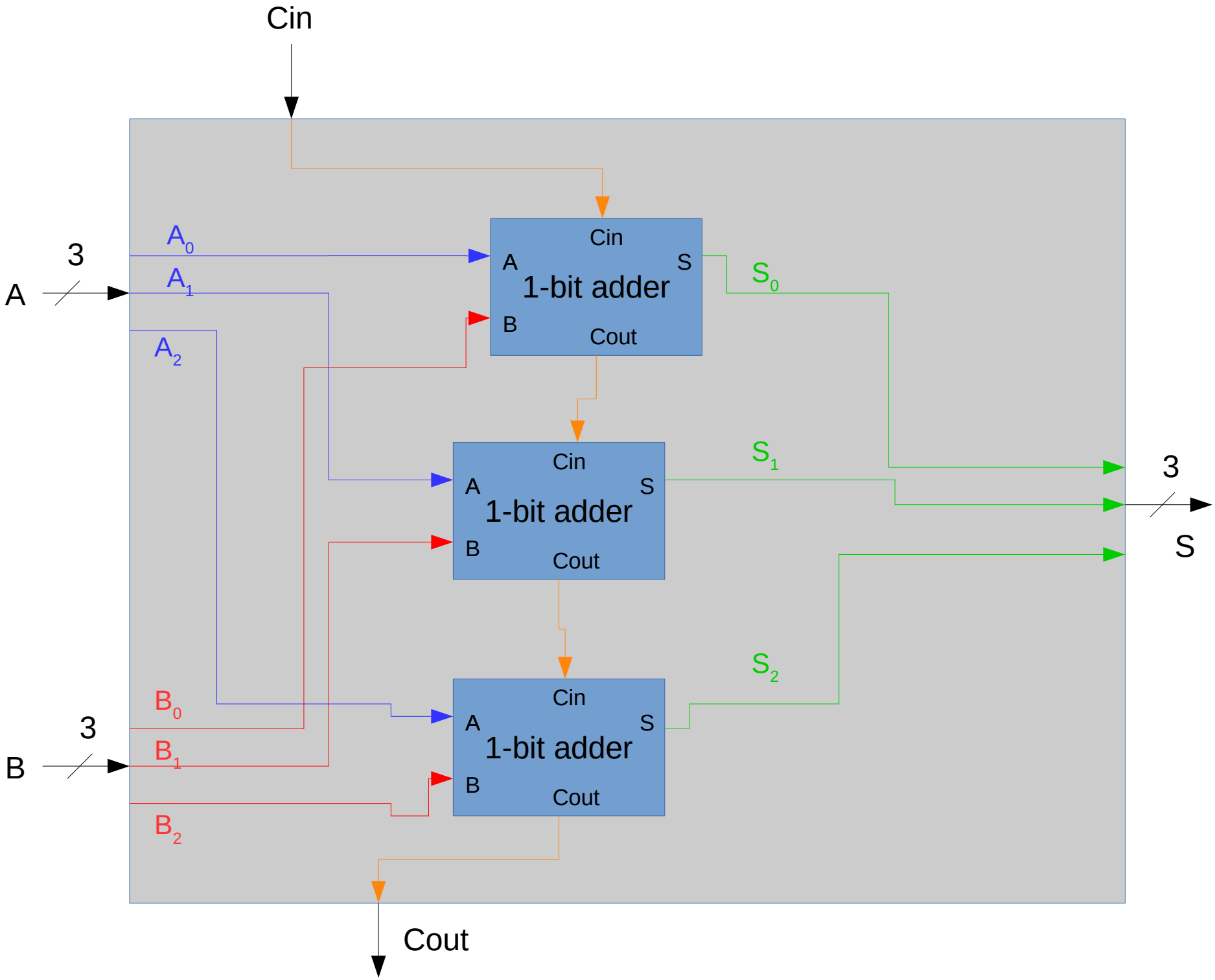
Inputs: A (3-bit), B (3-bit), Cin (1-bit)
Outputs: S (3-bit), Cout (1-bit)



Examples:
A=001, B=011, Cin=0; S=100, Cout=0
A=011, B=011, Cin=1; S=111, Cout=0
A=101, B=011, Cin=0; S=000, Cout=1

A=001, B=011, Cin=0; S=100, Cout=0

$$\begin{array}{r} 001 \\ +\ 011 \\ \hline 100 \end{array} \quad \Longleftrightarrow \quad \begin{array}{r} A_2A_1A_0 \\ +\ B_2B_1B_0 \\ \hline S_2S_1S_0 \end{array}$$


Cin
3
A
A_0
A_1
A_2
Cin
A
S
1-bit adder
B
Cout
S_0
Cin
A
S
1-bit adder
B
Cout
S_1
3
S
Cin
A
S
1-bit adder
B
Cout
S_2
B_0
3
B
B_1
B_2
Cout

Synthesis

```
`include "full_adder.v"
module threebit_adder(A, B, Cin, S, Cout);

    input [2:0] A;
    input [2:0] B;
    input Cin;
    output[2:0] S;
    output Cout;

    wire temp0, temp1;

    full_adder first(A[0], B[0], Cin, S[0], temp0);
    full_adder second(A[1], B[1], temp0, S[1], temp1);
    full_adder third(A[2], B[2], temp1, S[2], Cout);
endmodule
```

Let's describe our three-bit adder in Verilog.
We use our full_adder module, three times.

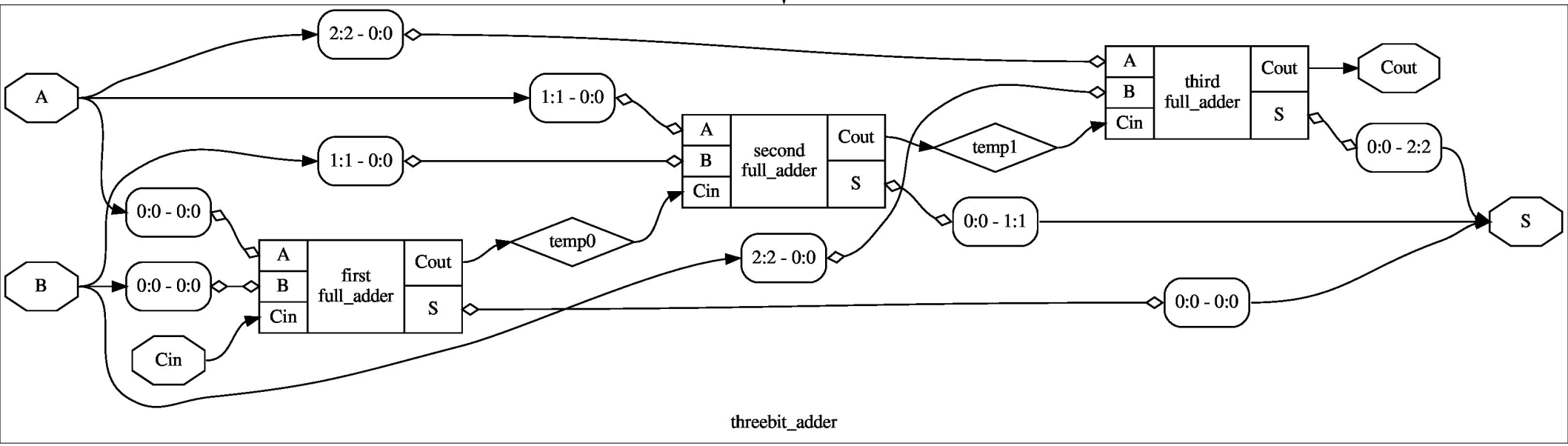
This code corresponds directly to the previous diagram. Convince yourself of this.

```
odule full_adder(A, B, Cin, S, Cout);

    input A, B, Cin;
    output S, Cout;

    assign S = A ^ B ^ Cin;
    assign Cout = (A & B) | (A & Cin) | (B & Cin);

endmodule
```

Synthesis

```
`include "full_adder.v"
module threebit_adder(A, B, Cin, S, Cout);

    input [2:0] A;
    input [2:0] B;
    input Cin;
    output[2:0] S;
    output Cout;

    wire temp0, temp1;

    full_adder first(A[0], B[0], Cin, S[0], temp0);
    full_adder second(A[1], B[1], temp0, S[1], temp1);
    full_adder third(A[2], B[2], temp1, S[2], Cout);
endmodule
```

yosys



Synthesis

```
module threebit_adder_highlevel(A, B, S);

    input [2:0] A;
    input [2:0] B;
    output[2:0] S;

    assign S = A + B;
endmodule
```

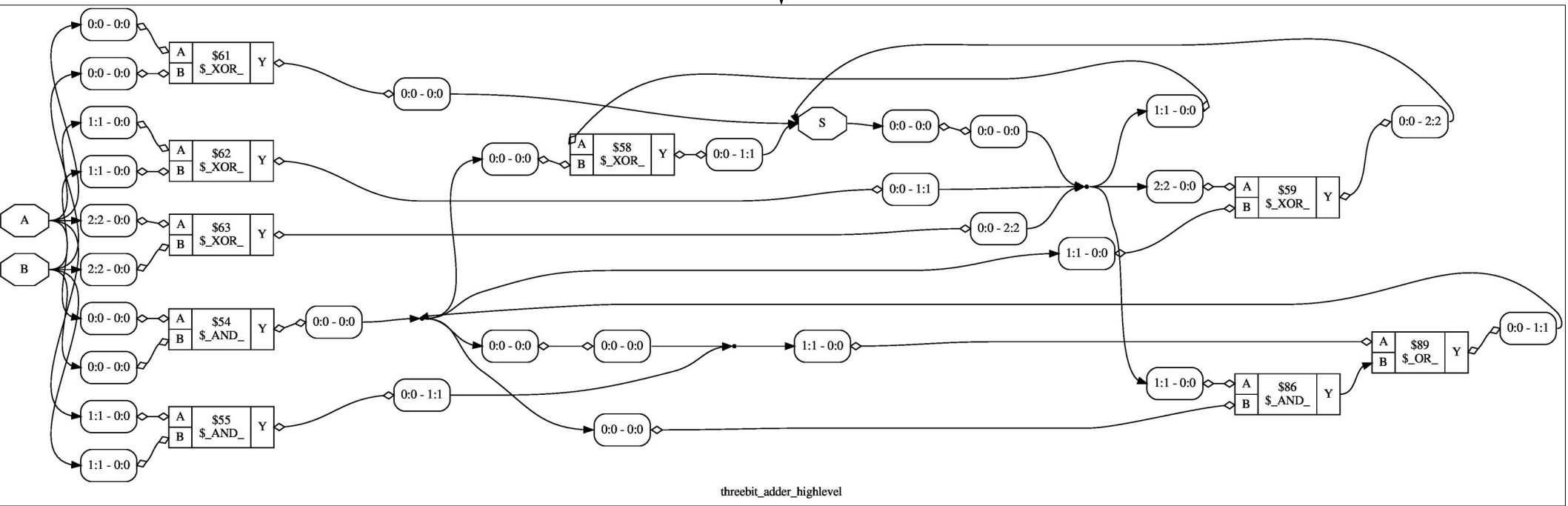
Verilog also supports high-level arithmetic operations like +, -, *, etc. These are easier for the programmer, and work for inputs of arbitrary size, **but**...

Synthesis

```
module threebit_adder_highlevel(A, B, S);

    input [2:0] A;
    input [2:0] B;
    output[2:0] S;

    assign S = A + B;
endmodule
```

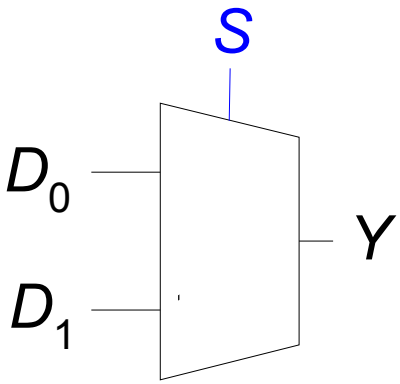
yosys

The high-level operations are ultimately synthesized into low-level gates. Some homework assignments will ask you to avoid the high-level operations (+, -, *, etc).



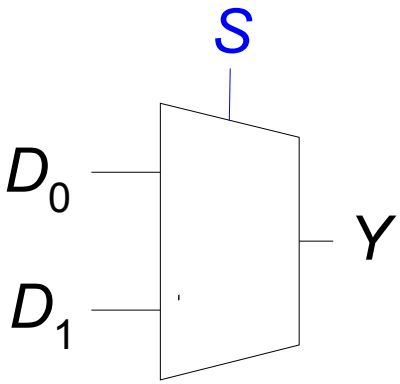
Other useful circuit components

What is happening here?



S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

What is happening here?

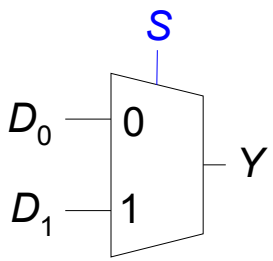


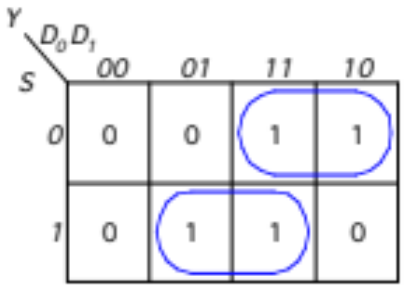
S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

S	Y
0	D_0
1	D_1

Multiplexer (Mux)

- Selects between one of *N* inputs to connect to output
- $\log_2 N$-bit select input – control input
- **Example:** 2:1 Mux

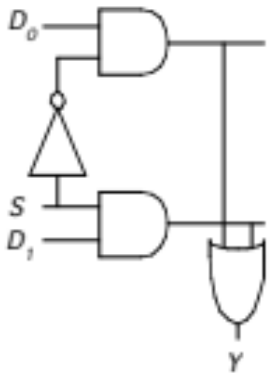




$Y = D_0\overline{S} + D_1S$

S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

S	Y
0	D_0
1	D_1



Multiplexer (Mux) in Verilog

```
module multiplexer2(D0, D1, S, Q);

    input D0, D1;    // the two incoming data lines
    input  S;        // the selector, a 1-bit value
    output Q;        // the output

    assign Q = (D0 & ~S) | (D1 & S);

endmodule
```

Think of the logic this way. There are two ways this multiplexer will output 1:

- *either* S is 0, and D0 is 1
- *or* S is 1, and D1 is also 1.

Otherwise the multiplexer outputs 0.

Multiplexer (Mux) in Verilog

Conditional (aka Ternary) operator

```
module multiplexer2(D0, D1, S, Q);

    input D0, D1;    // the two incoming data lines
    input  S;        // the selector, a 1-bit value
    output Q;        // the output

    assign Q = (S==0) ? D0 : D1;

endmodule
```

We can express a multiplexor in Verilog using the **?:** syntax. The same syntax exists in C and C++.

```
condition ? iftrue : iffalse
```

In other words: *if* S is 0, *then* the output is the same as D0; *otherwise*, the output is the same as D1.

More about the Conditional (aka Ternary) operator

This operator works like an **if** statement, but syntactically it's an expression.

```
condition ? iftrue : iffalse
```

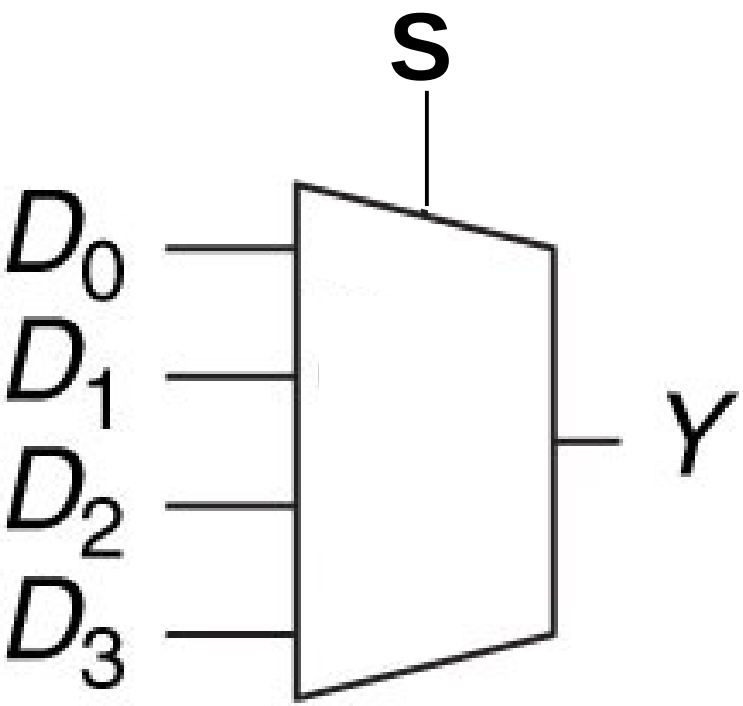
The following two fragments of C++ code produce the same result.

```
if (x % 2 == 1)
   cout << "odd";
else
   cout << "even";
```

```
cout << ((x % 2 == 1) ? "odd" : "even");
```

If the **condition** part is true, the operator returns the **iftrue** part; otherwise, it returns the **iffalse** part.

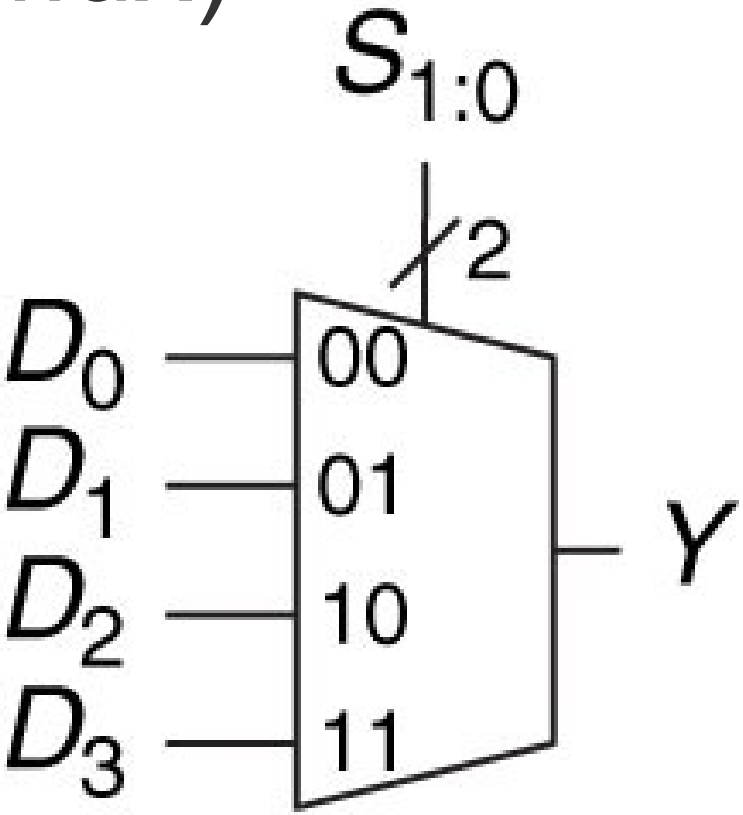
Multiplexer (Mux)



What if we want to select one out of *four* possible inputs?

4:1 multiplexer

Multiplexer (Mux)



What if we want to select one out of *four* possible inputs? **Then we will need a 2-bit selector.** A 2-bit signal has 4 possible values.

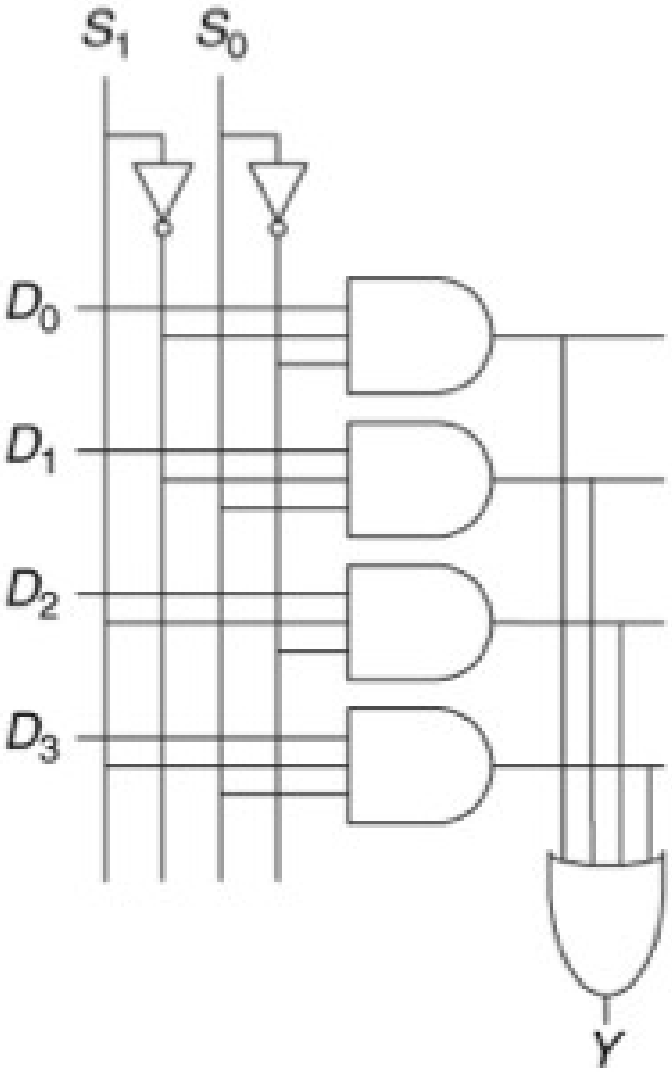
4:1 multiplexer

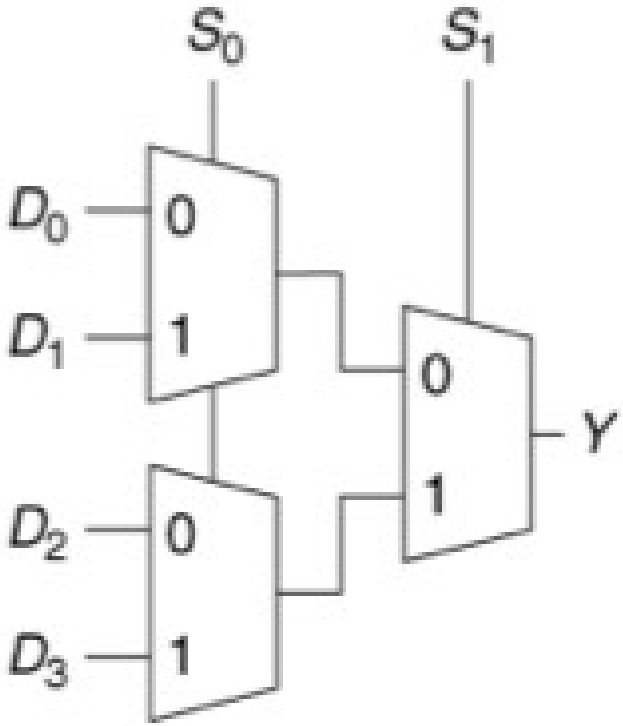
Generally, if we have *n* inputs, we need *ceil(log$_2$(n))* bits in our selector.

Multiplexer (Mux)

```
module multiplexer4(D0, D1, D2, D3, S, Q);

    input D0, D1, D2, D3;  // the four incoming data lines
    input [1:0] S;         // the selector, a 2-bit vaue
    output Q;              // the output

    assign Q = (S==0) ? D0 : (S==1) ? D1 : (S==2) ? D2 : D3;

endmodule
```

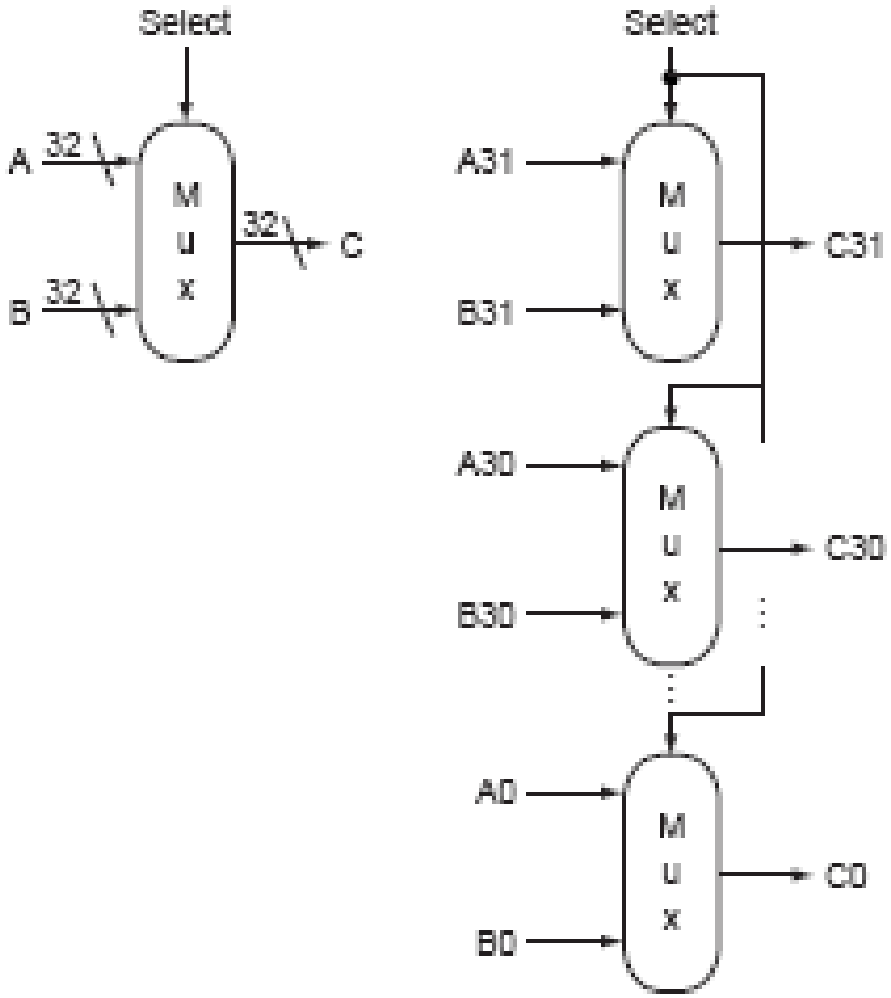
Multiplexer (Mux)





Two implementations of 4:1 multiplexor

Multiplexer (Mux)

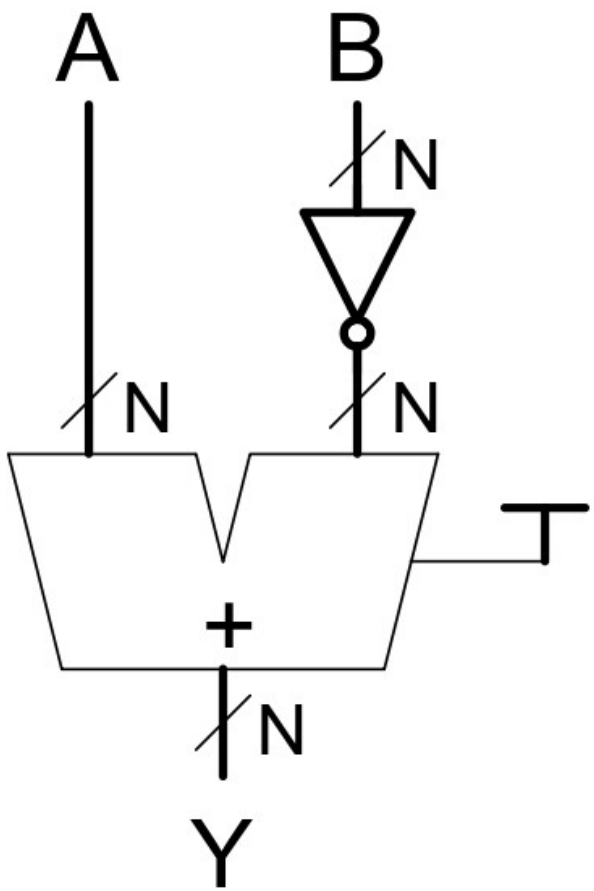


a. A 32-bit wide 2-to-1 multiplexor

b. The 32-bit wide multiplexor is actually an array of 32 1-bit multiplexors

FIGURE B.3.6 A multiplexor is arrayed 32 times to perform a selection between two 32-bit inputs. Note that there is still only one data selection signal used for all 32 1-bit multiplexors.

What does this circuit do?



There are two *n*-bit inputs, A and B.

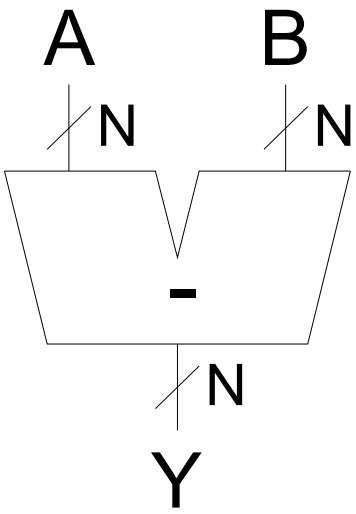
A is passed into an *n*-bit adder. B is passed into a NOT gate. then into the adder.

The Cin input of the adder is connected to a wire carrying a fixed high voltage, i.e. a 1. The T symbol indicates a True value.

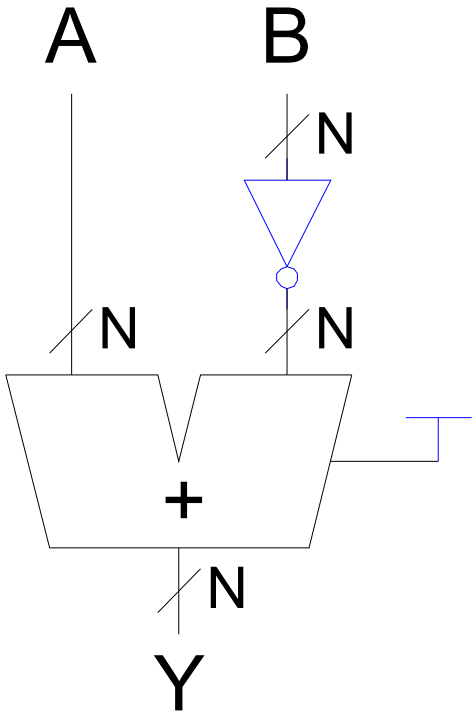
What is Y, in terms of A and B?

Subtracter

Symbol



Implementation



Y = A - B

Subtracter

How does the subtracter work? It uses the adder to add a negative.

A - B → A + (-B)

How do we calculate -B? 2's complement tells us that to calculate the negative of a number, we just flip all the bits and add one.

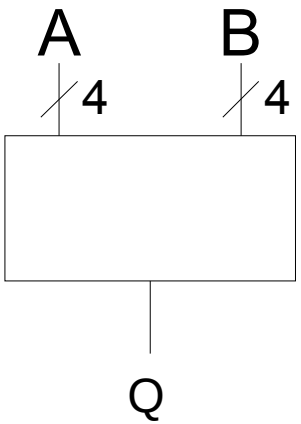
-B → (NOT B) + 1

The flipping of the bits is done by the NOT gate. Adding one is done by ensuring that the Cin of the adder is always 1.

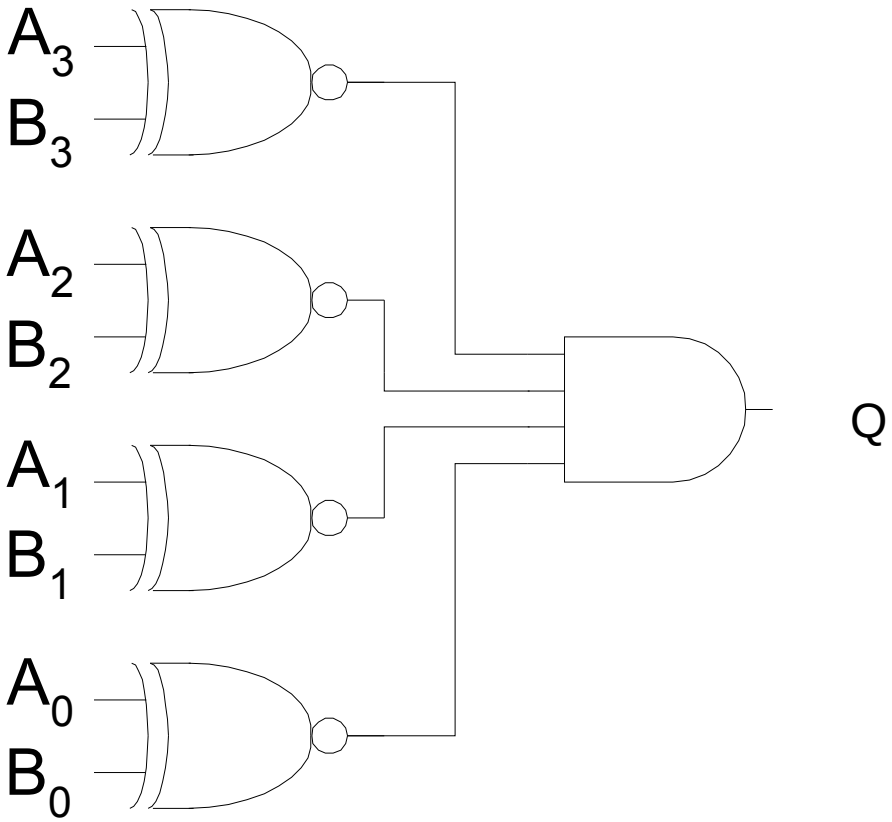
A - B → A + (-B) → A + (NOT B) + 1

What does this circuit do?

Symbol

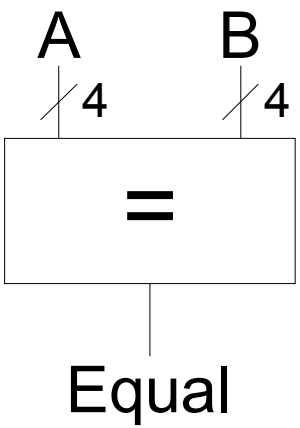


Implementation

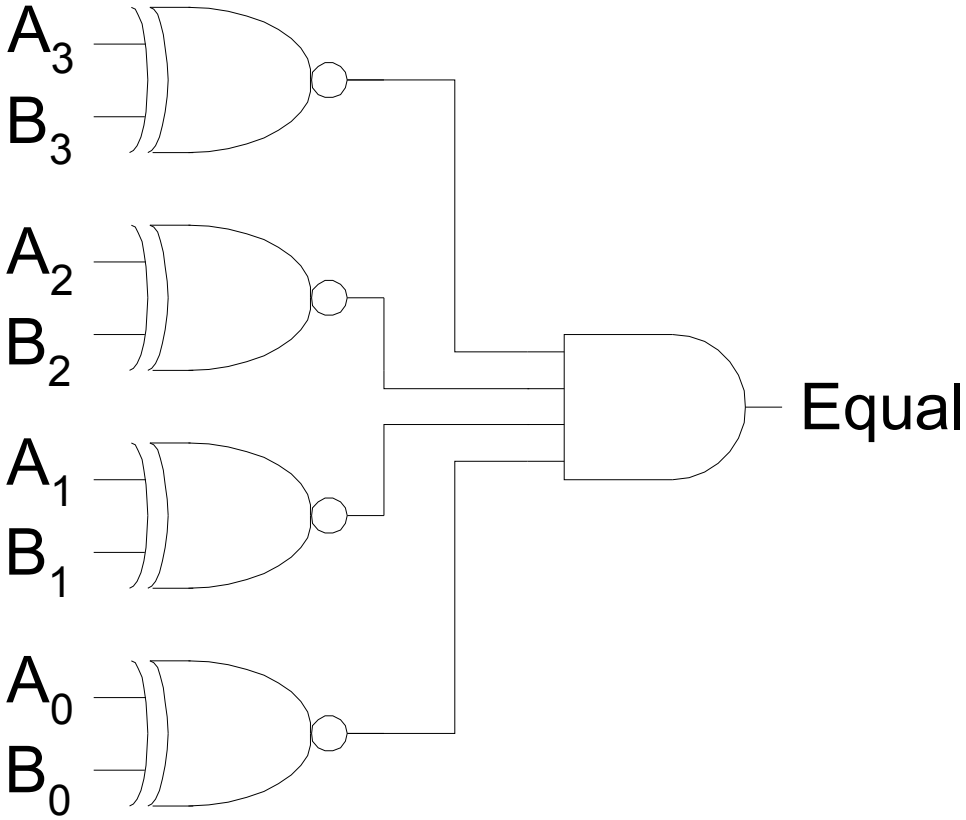


Comparator: Equality

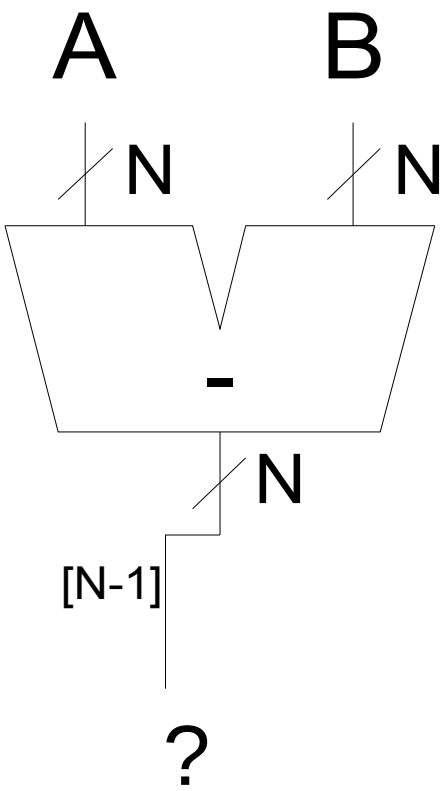
Symbol



Implementation

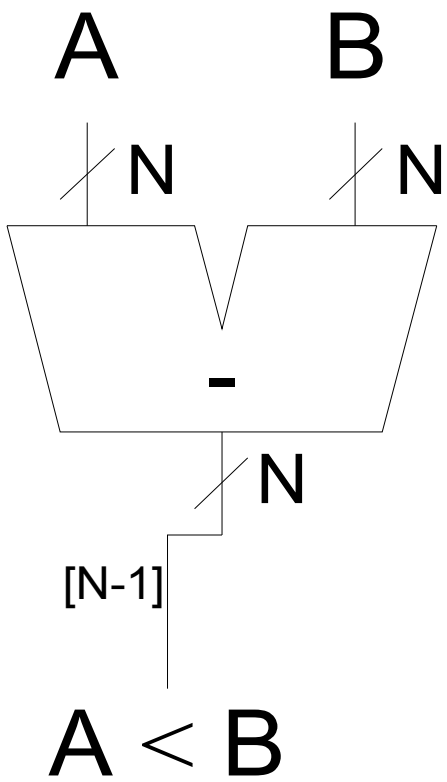


What does this circuit do?



Note that the [N-1] notation means that we are taking the N-1 bit from the N-bit value. That is, if the output of the subtractor is 8 bits, then we are looking at bit 7. In other words, [N-1] examines the **most significant bit**.

Comparator: Less Than



A is less than B exactly when A-B is negative, i.e., the m.s.b. is 1. A is greater or equal to B otherwise and m.s.b. is 0.

Shifters

- **Logical shifter:**
  - Ex: 11001 >> 2 = 00110
  - Ex: 11001 << 2 = 00100
- **Arithmetic shifter:**
  - Ex: 11001 >>> 2 = **11**110
  - Ex: 11001 <<< 2 = 00100
- **Rotator:**
  - Ex: 11001 ROR 2 = 01110
  - Ex: 11001 ROL 2 = 00111

Arithmetic significance of shifting

3 << 1 = 6
3 << 2 = 12
3 << 3 = 24

51 >> 1 = 25
51 >> 2 = 12
51 >> 3 = 6

Shift implementation

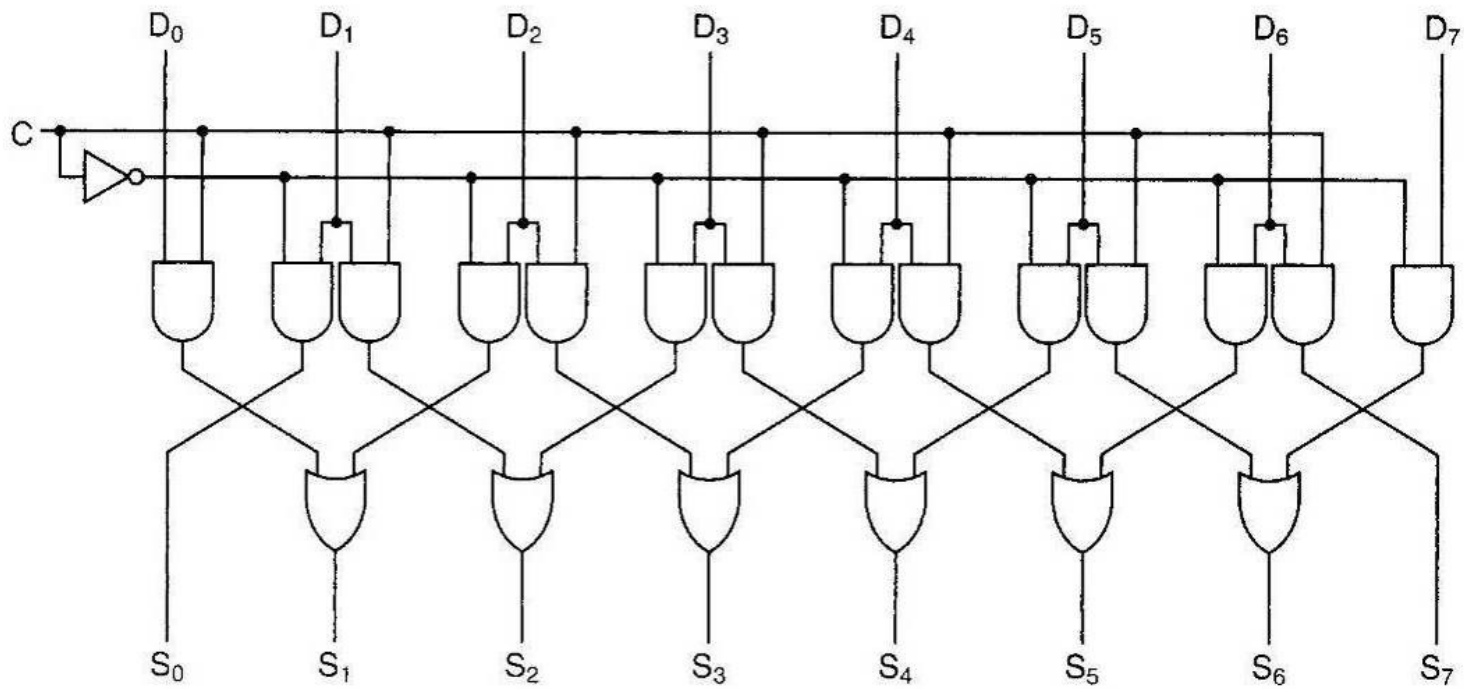
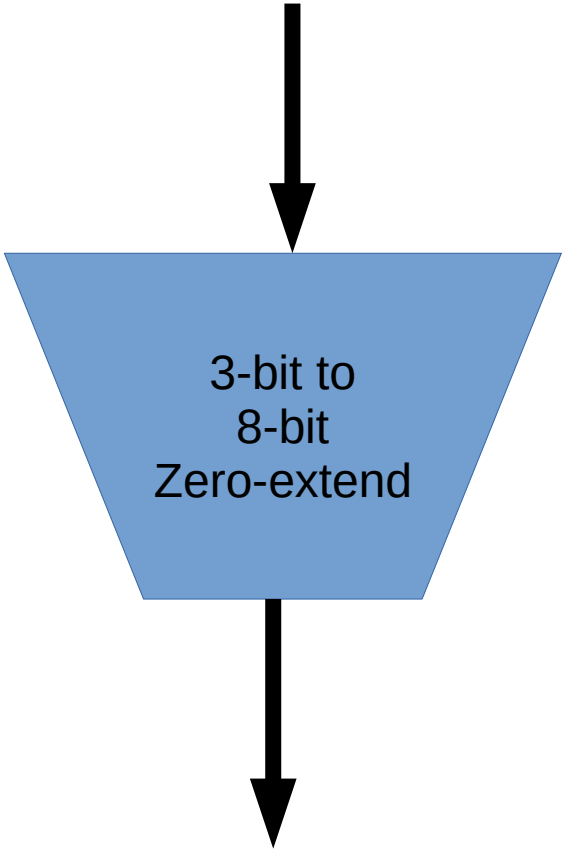


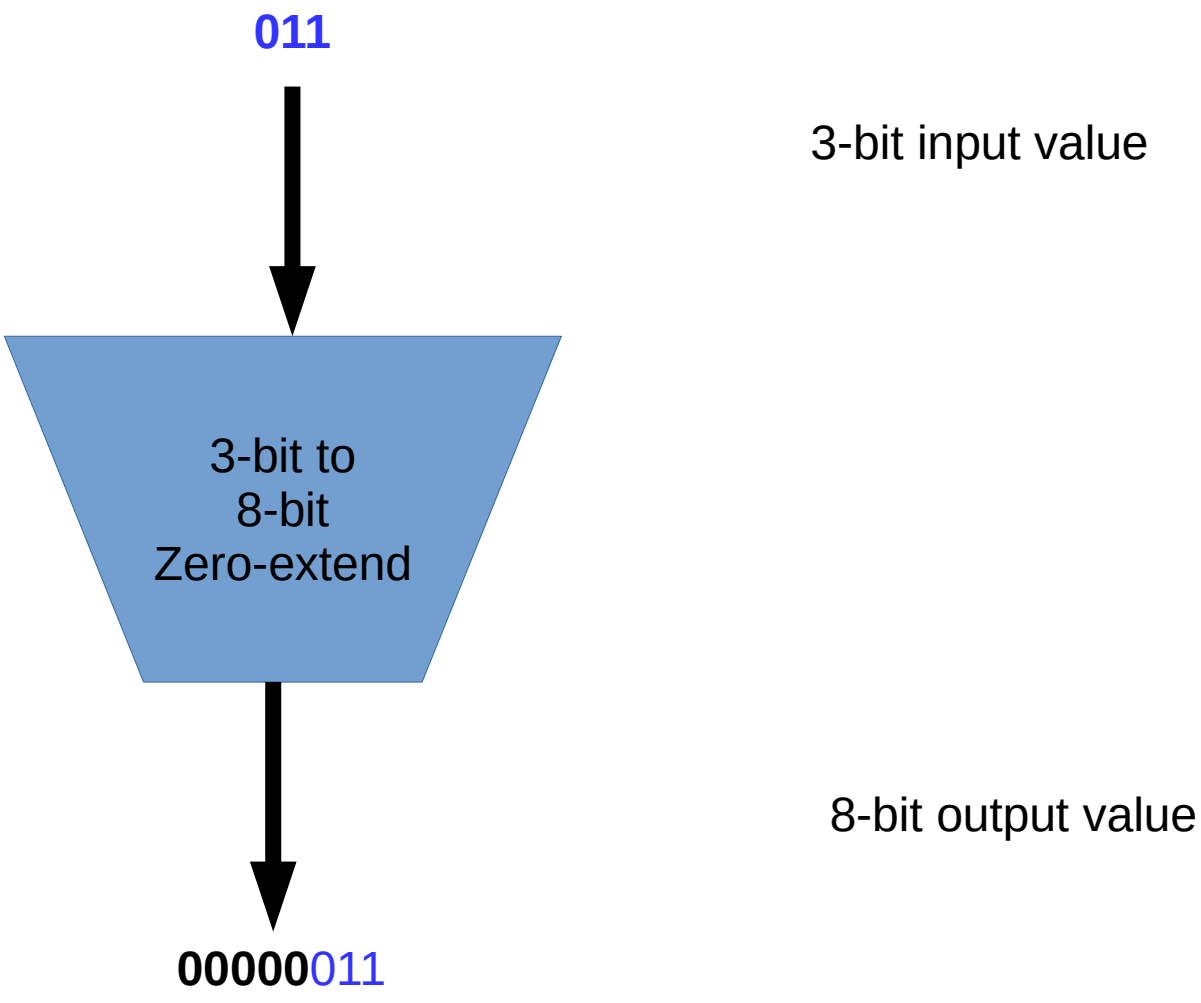
Figure 3-16. A 1-bit left/right shifter.

Shift by a single bit left or right (C = 0,1)

Zero-extend and Sign-extend

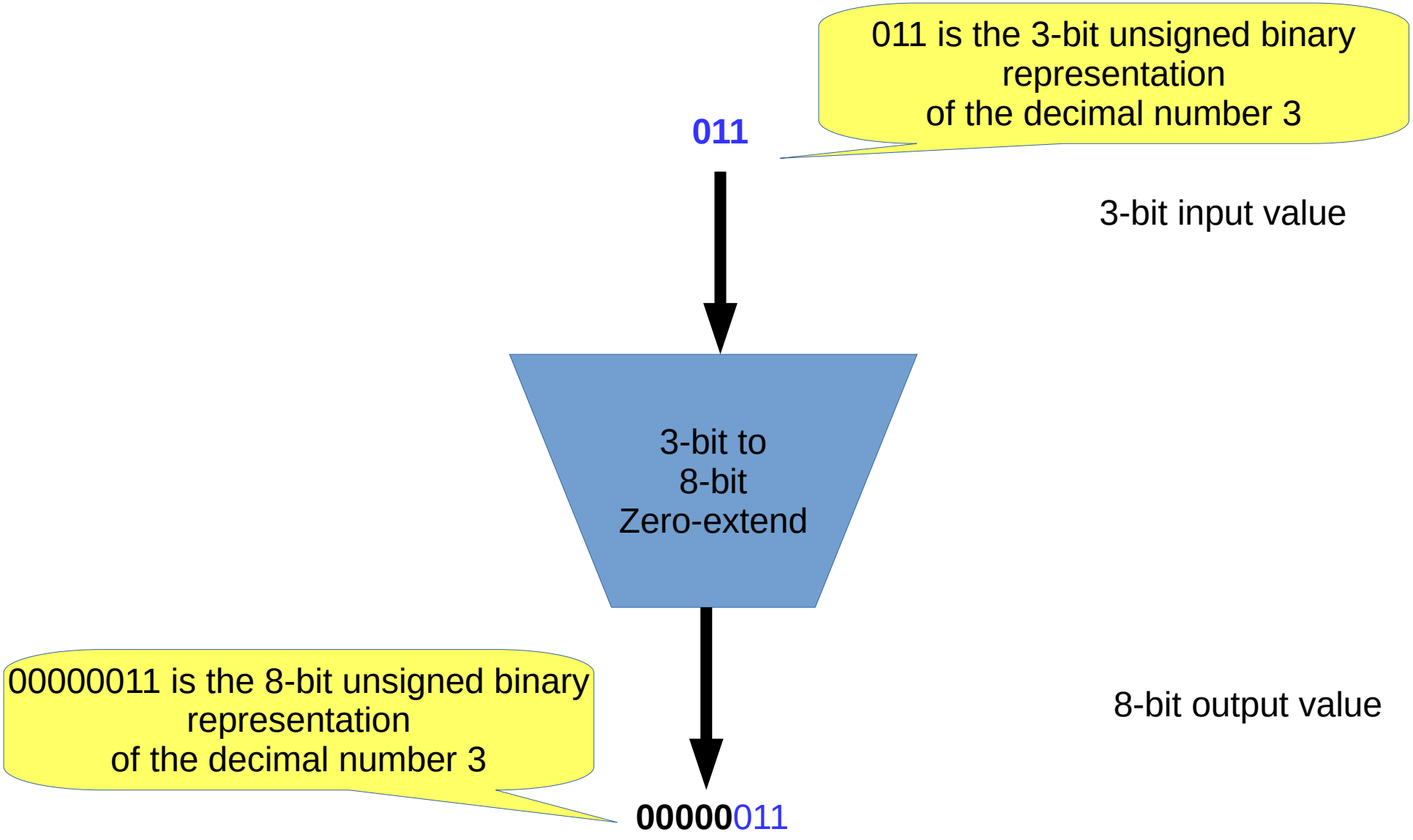


Zero-extend and Sign-extend



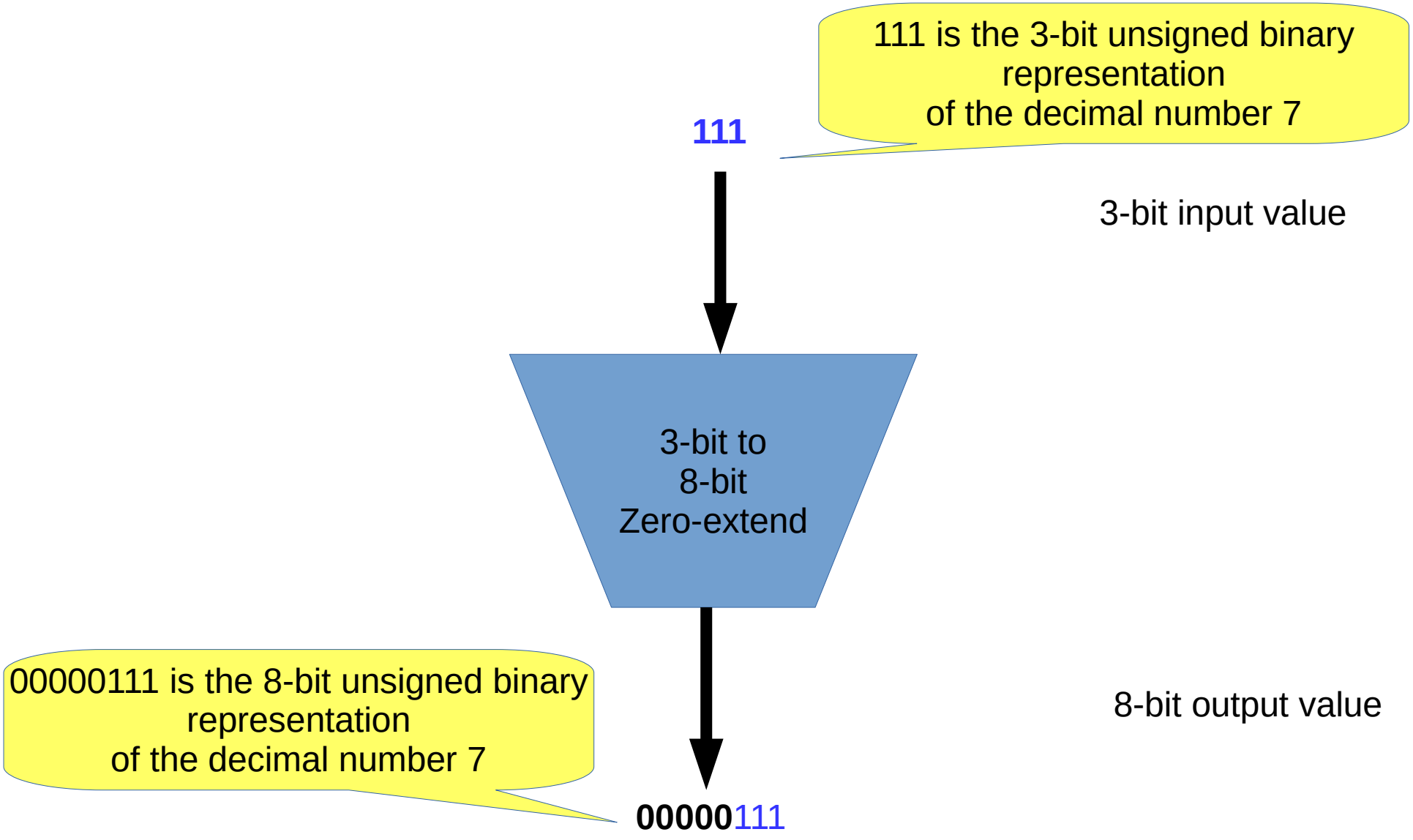
Zero-extend takes a n-bit input and produces an m-bit output, by adding m-n additional zero bits. Assuming the input is an unsigned integer, zero-extend will change the *size*, but not the *value*, of its input.

Zero-extend and Sign-extend



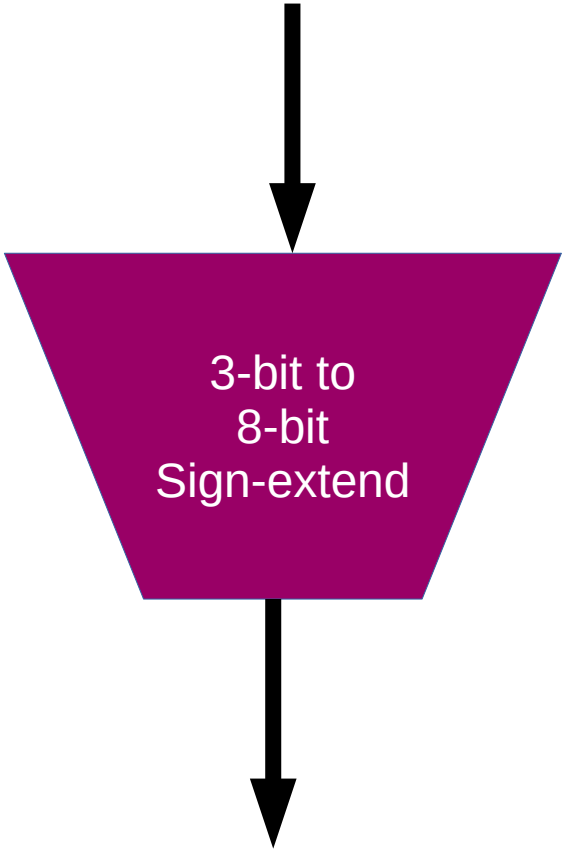
Zero-extend takes a n-bit input and produces an m-bit output, by adding m-n additional zero bits. Assuming the input is an unsigned integer, zero-extend will change the *size*, but not the *value*, of its input.

Zero-extend and Sign-extend



Zero-extend takes a n-bit input and produces an m-bit output, by adding m-n additional zero bits. Assuming the input is an unsigned integer, zero-extend will change the *size*, but not the *value*, of its input.

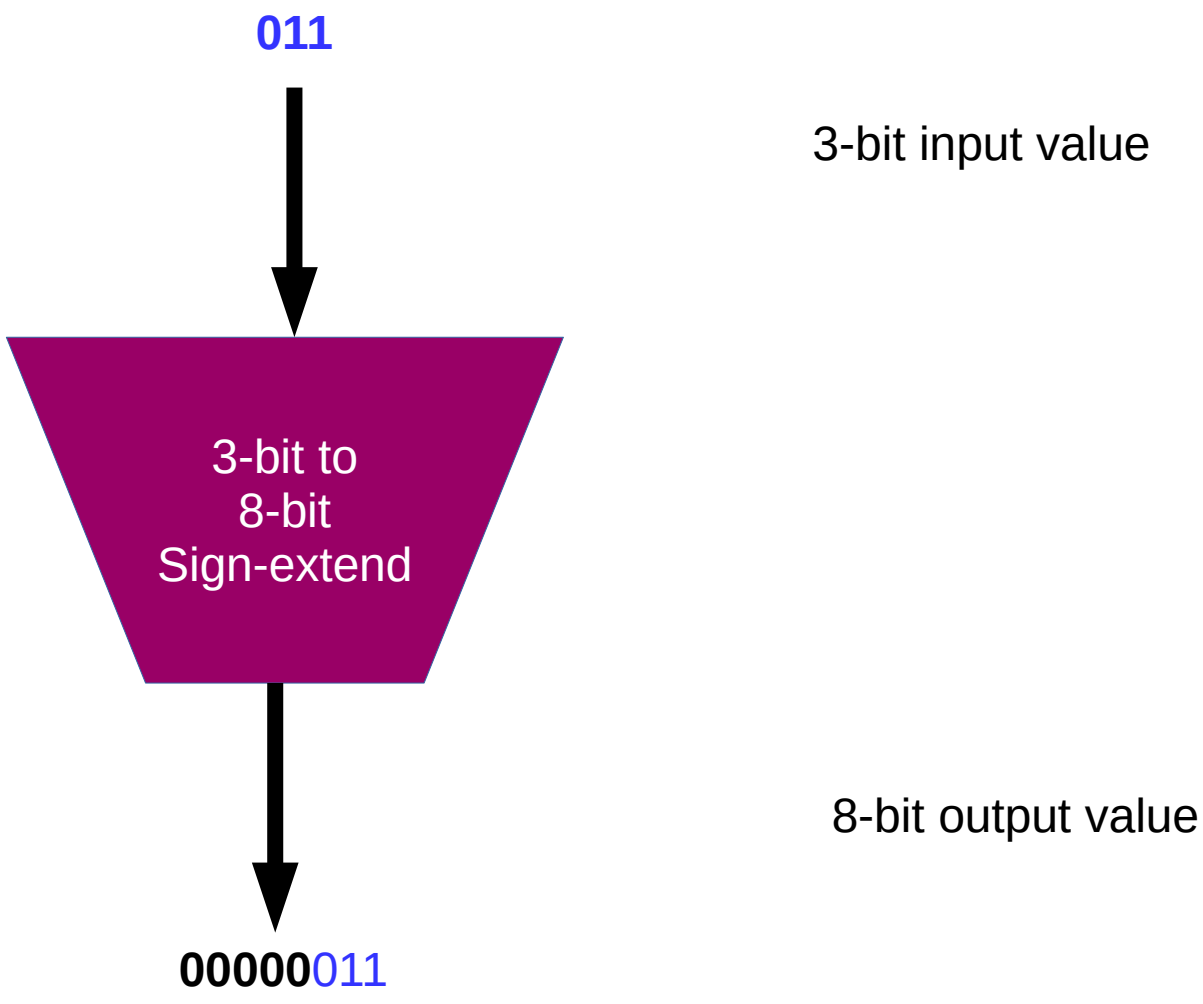
Zero-extend and Sign-extend



3-bit input value

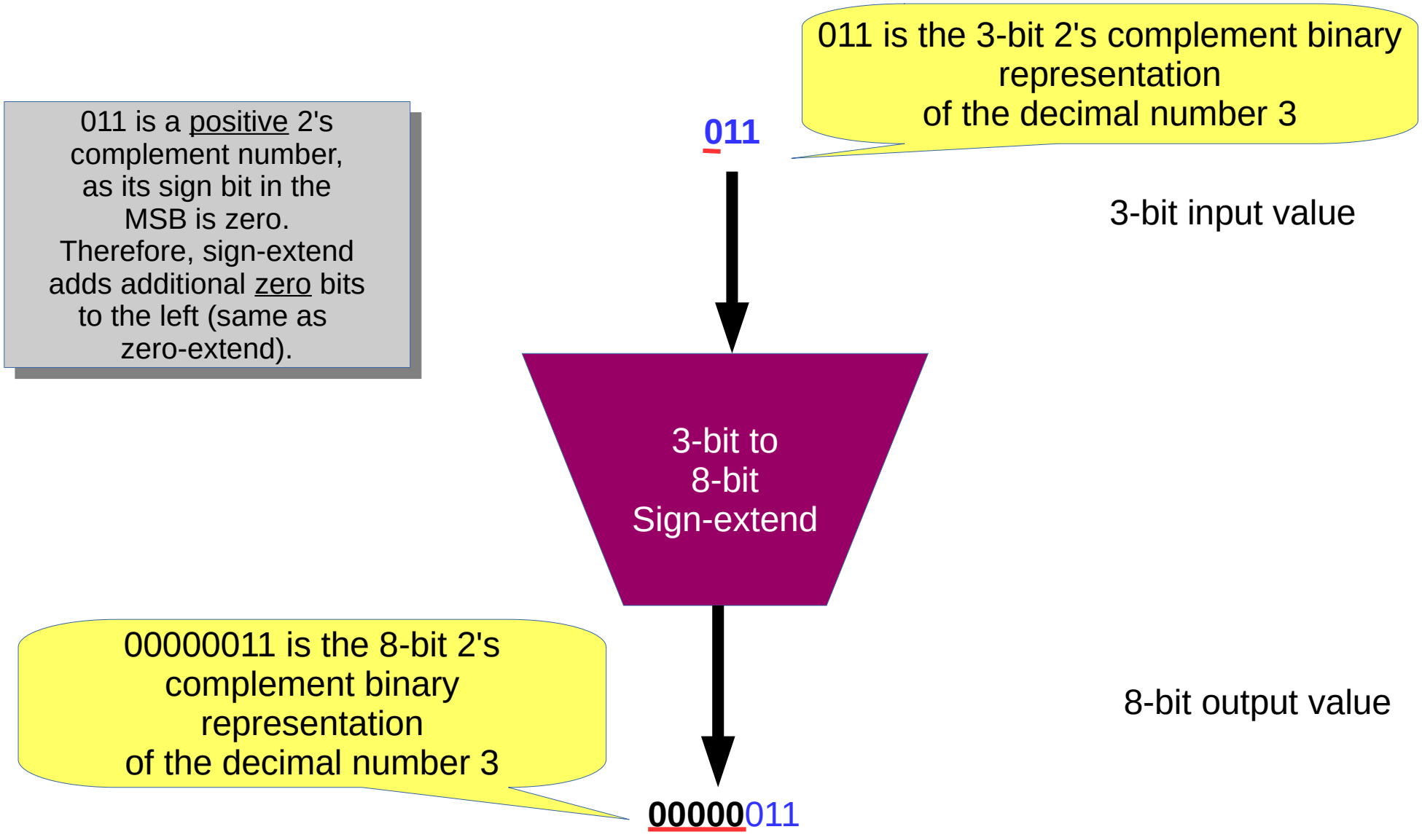
8-bit output value

Zero-extend and Sign-extend



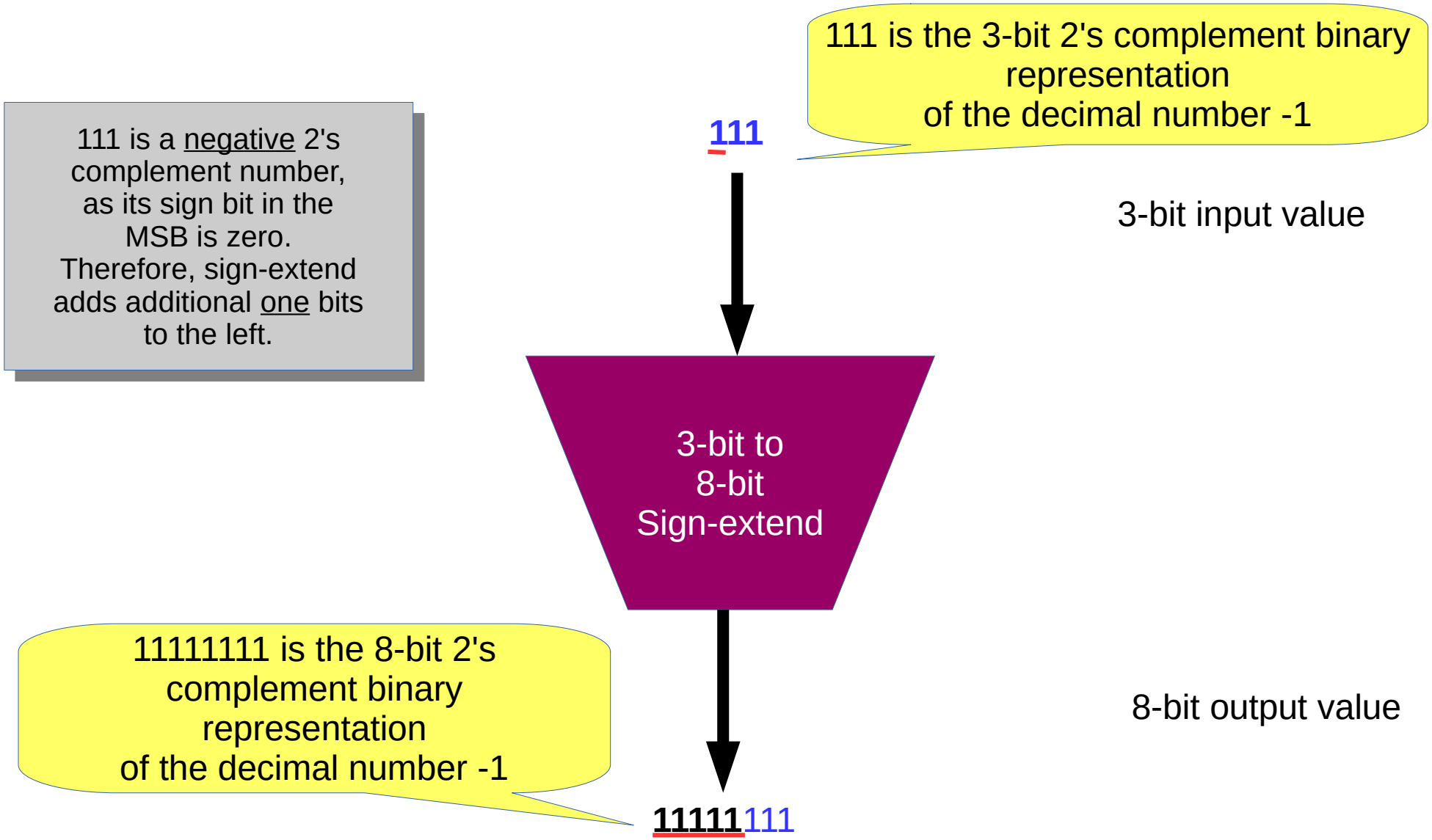
Sign-extend takes a n-bit input and produces an m-bit output, by adding m-n additional bits equal to the most significant bit of the input (i.e. the *sign* bit). Assuming the input is a 2's complement integer, sign-extend will change the *size*, but not the *value*, of its input.

Zero-extend and Sign-extend



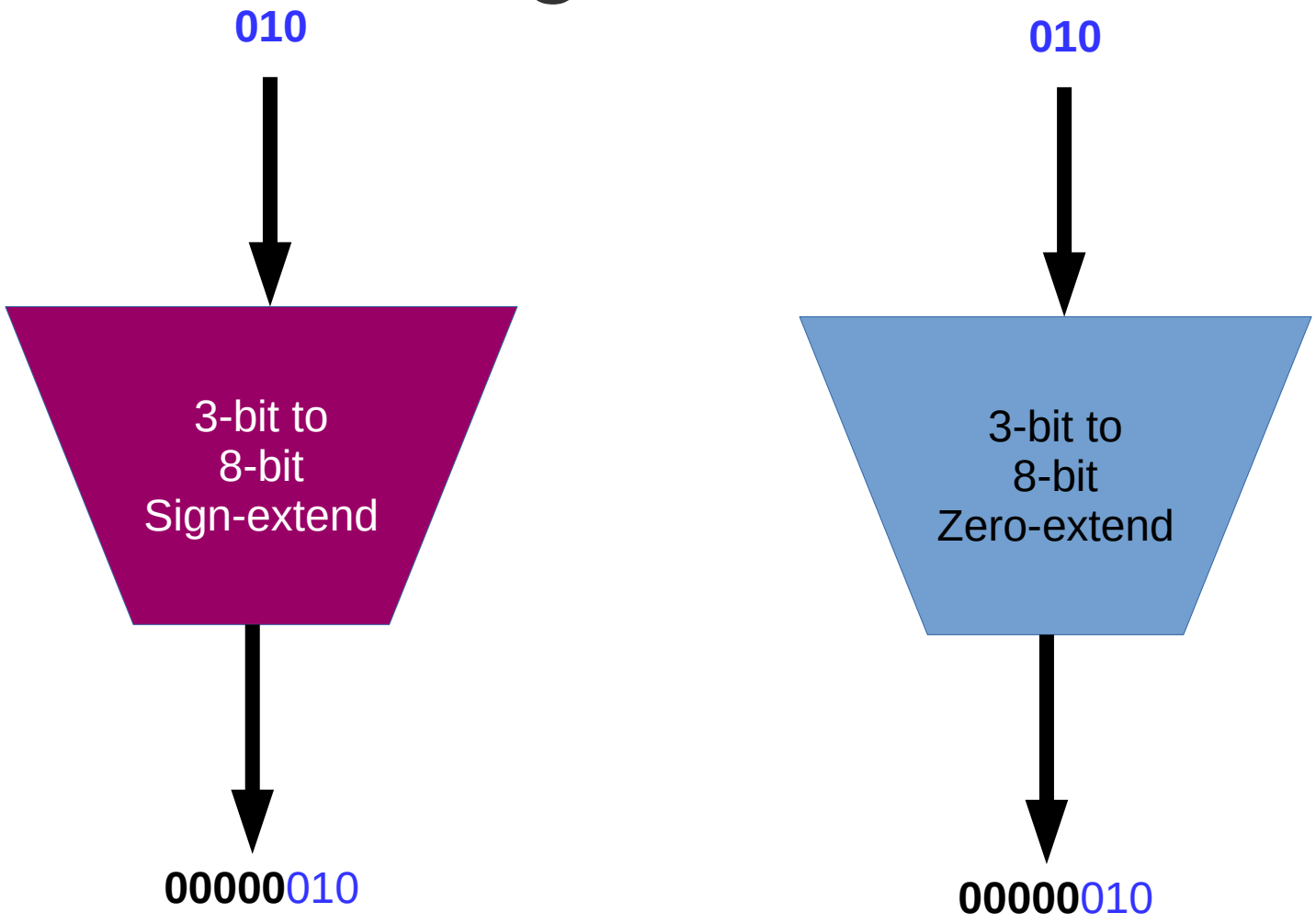
Sign-extend takes a n-bit input and produces an m-bit output, by adding m-n additional bits equal to the most significant bit of the input (i.e. the *sign* bit). Assuming the input is a 2's complement integer, sign-extend will change the *size*, but not the *value*, of its input.

Zero-extend and Sign-extend



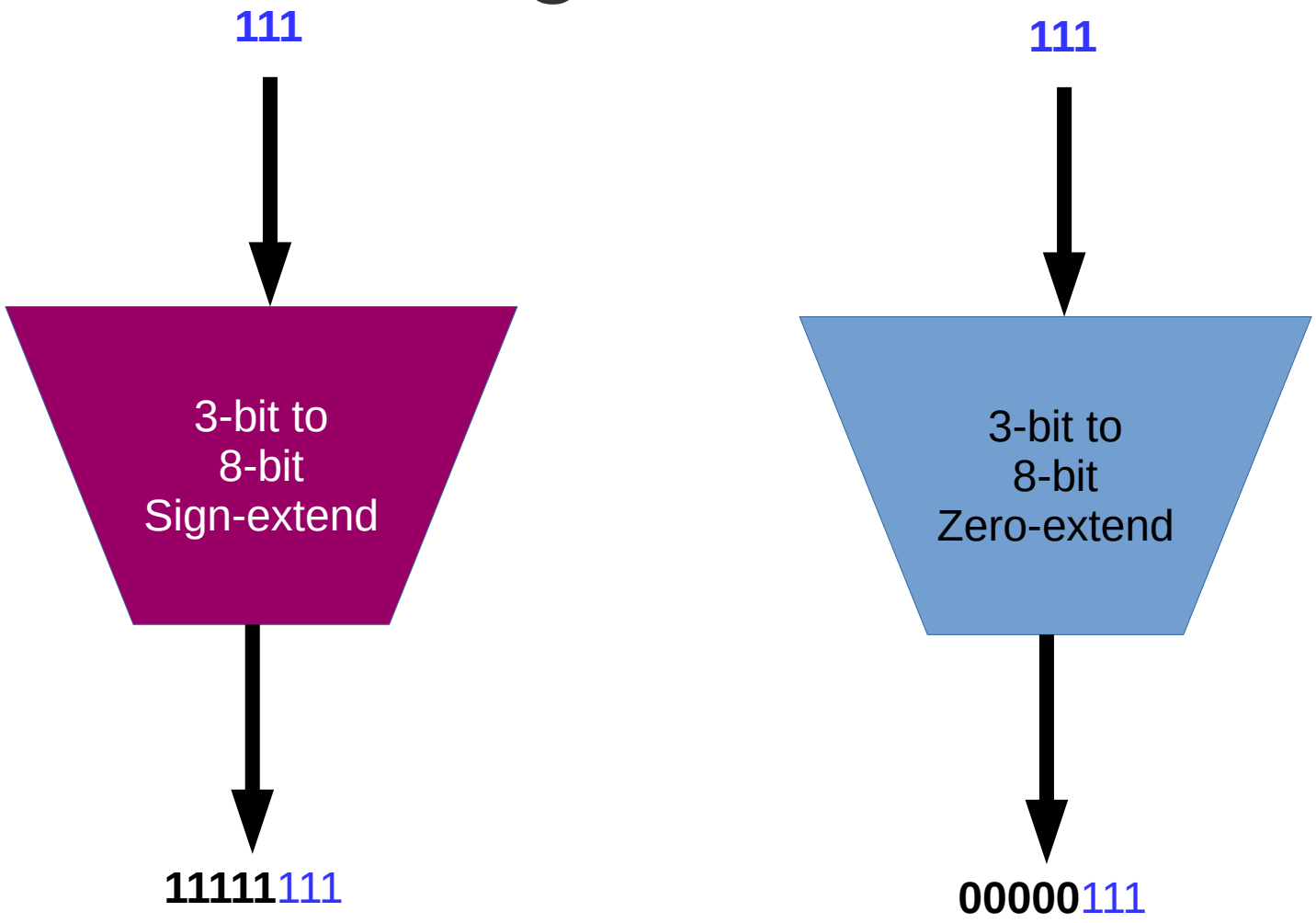
Sign-extend takes a n-bit input and produces an m-bit output, by adding m-n additional bits equal to the most significant bit of the input (i.e. the *sign* bit). Assuming the input is a 2's complement integer, sign-extend will change the *size*, but not the *value*, of its input.

Zero-extend and Sign-extend



When the MSB of the input is zero, zero-extend and sign-extend produce the same output.

Zero-extend and Sign-extend



If we know the input is 2's complement, we probably want to use sign-extend. If we know the input is unsigned, we probably want to use zero-extend. If we use the wrong one, we will accidentally change the value. For example:

111 (binary) = -1 (2's complement decimal)

zero-extended

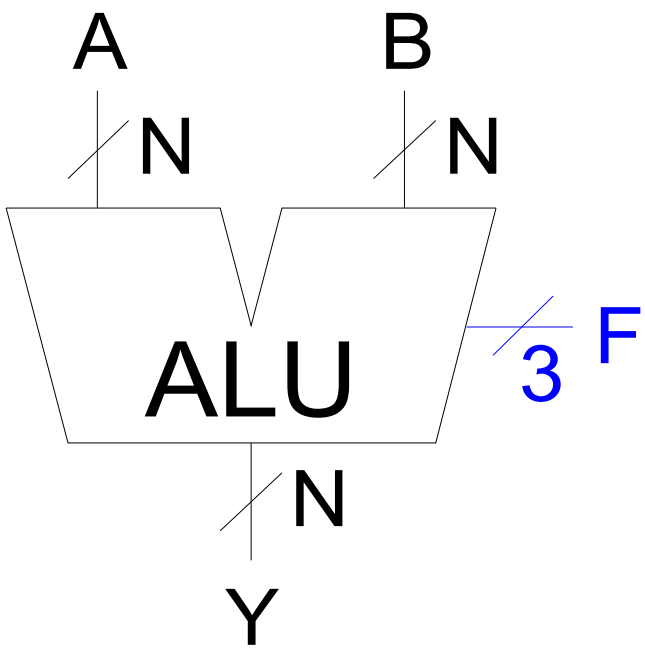
00000111 (binary) = 3 (2's complement decimal)

Zero-extend and Sign-extend in Verilog

```
module zero_extend_3_to_8(a, b);
    input [2:0] a;
    output [7:0] b;
    assign b = {5'b0,a};
endmodule
```

```
module sign_extend_3_to_8(a, b);
    input [2:0] a;
    output [7:0] b;
    assign b = {a[2],a[2],a[2],a[2],a[2],a};
endmodule
```

Arithmetic Logic Unit (ALU)

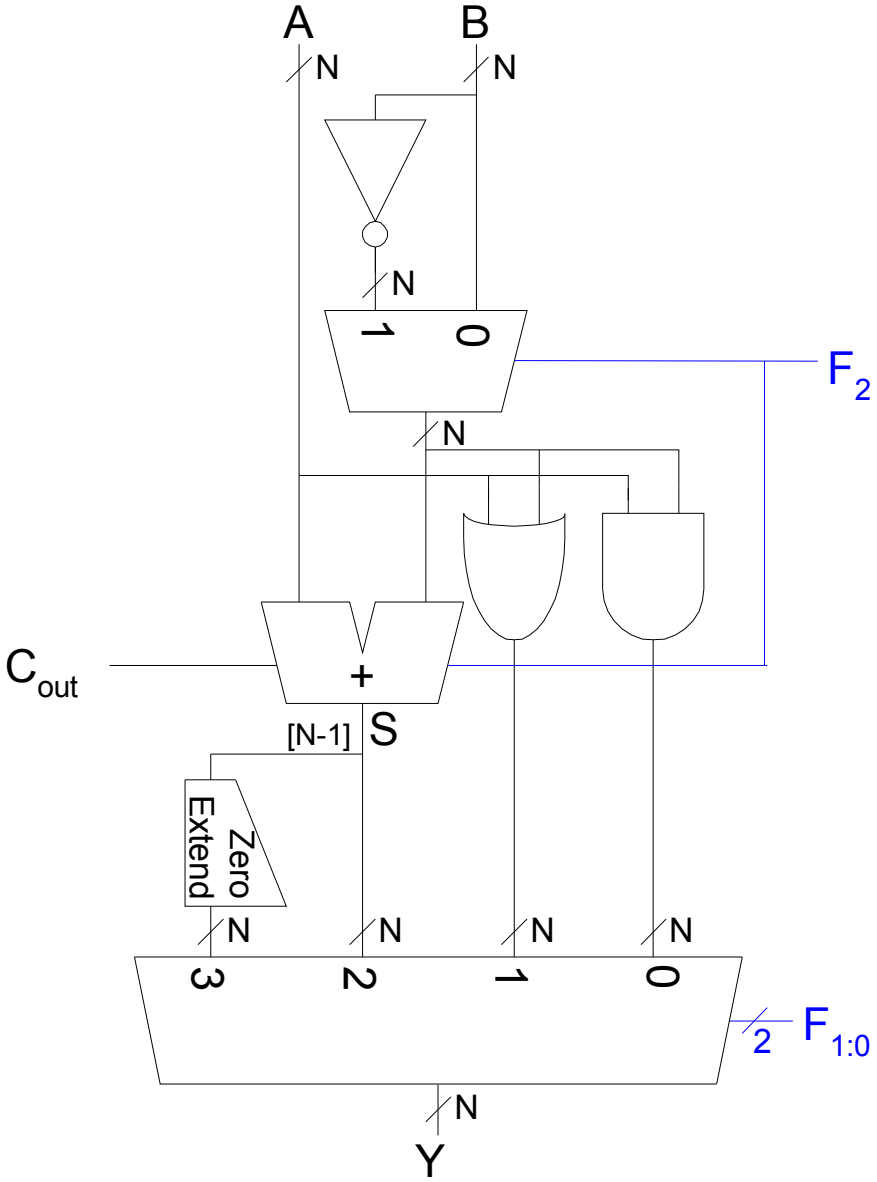


$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

~B=Not B

SLT: Set on Less Than
Y= 1 if A<B; 0 otherwise

ALU Design



$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT