Lesson: Building a Computer Pillar: Computer Architecture

In previous lessons that dealt with computer architecture, many fundamental building blocks of computers were covered; for example:

- The layers of a computer system (e.g., operating system, applications, and so on);
- Fundamentals of digital logic (circuits and switches);
- Logic gates (*and*, *or*, and *not*);
- Boolean algebra;
- Combinational circuits (*xor*, comparators, and adders);
- The binary and hexadecimal number systems;
- Converting between number systems;
- Binary arithmetic (addition and multiplication); and
- Half and full adders, and chaining full adders together to add larger binary numbers.

In this lesson, we will first discuss how numbers and characters are represented in computers. So far, all of the numbers that we have looked at have been *unsigned* (i.e., assumed to be positive). For example, the number 1001001_2 is equivalent to 73_{10}. However, in order to be able to handle negative numbers, we need a way to represent the sign of a number. Subsequently, we will cover several more combinational circuits that prove quite useful in the design of computing systems. In addition, several important sequential circuits that serve as the building blocks for computer memory will be discussed. Finally, we will show how a Central Processing Unit (CPU) is built from the primitive components discussed in the lessons on computer architecture.

Signed numbers

An early attempt to handle signed numbers was to add a special sign *bit* to the left of each number. A zero in the sign bit was used for positive numbers, and a one in the sign bit was used for negative numbers. The representations of +5 and -5 in **signed magnitude notation** are shown below. These representations assume that three bits are reserved for the magnitude of a number and one bit for its sign:

Signed magnitude binary				Decimal
Sign	Magnitude			
0	1	0	1	+5
1	1	0	1	-5

Signed magnitude notation works, provided that the sign bit is treated separately from the magnitude of the number and guides how arithmetic operations are performed. However, arithmetic in this notation is difficult and results in some anomalies. For example, there are two representations for zero (+0 is 0000 and -0 is 1000). To illustrate the difficulties in arithmetic, let's look at the sum of +5 and -5 (which should be 0):

	Binary					Decimal
Carry	1	1	0	1		
1st number		0	1	0	1	+5
2nd number		1	1	0	1	-5
Sum	1	0	0	1	0	2?

Note that the sum (10010) does not equal 0 (0000 or 1000) using signed magnitude notation (even if we ignore the leftmost bit). Arithmetic becomes easier, however, if we write a negative number as the complement of the corresponding positive number. The **complement** of a binary number is formed by writing a 0 wherever there is a 1 in the original number, and a 1 wherever there is a 0 in the original number. As with signed magnitude notation, the leftmost bit still indicates the sign of the number: 0 for positive numbers, and 1 for negative numbers. Complementing a binary number in this way to represent signed values is sometimes referred to as **one's complement notation**. The one's complement representation of +5 and -5 are shown below:

One's complement binary				Decimal
Sign	Magnitude			
0	1	0	1	+5
1	0	1	0	-5

Although the arithmetic for one's complement numbers is much easier than for signed magnitude numbers, the anomaly of two representations for zero (0000 for +0 and 1111 for -0) still exists. Let's see how +5 and -5 can be added using one's complement notation:

	Binary				Decimal
Carry	0	0	0		
1st number	0	1	0	1	+5
2nd number	1	0	1	0	-5
Sum	1	1	1	1	0

Indeed, the result (1111) is 0 (well, one variation of it: -0). **Two's complement notation** is a variation of one's complement notation that only includes a single representations for 0. To change the sign of a two's complement binary number, we perform the following three steps:

(1) Write down the binary representation of the original number;
(2) Complement all of the bits (i.e., replace 1's with 0's and 0's with 1's); and
(3) Add one to the result.

Here are +5 and -5 written in two's complement notation:

Two's complement binary				Decimal
0	1	0	1	+5
1	0	1	1	-5

Let's see how +5 and -5 can be added using two's complement notation:

	Binary				Decimal
Carry	1	1	1	1	
1st number	0	1	0	1	+5
2nd number	1	0	1	1	-5
Sum	*1*	0	0	0	0

Notice that while we get a correct answer (0000), this problem also generated a carry bit (the italicized leftmost 1). We will discuss this anomaly later. While the idea of two's complement notation may seem a little strange at first, it has a unique representation for zero (0000, given four bits) and straightforward arithmetic operations. Notice that, for positive numbers, all three representations (signed magnitude, one's complement, and two's complement) have the same pattern as unsigned numbers.

The following table presents the sixteen signed numbers that can be represented using two's complement notation and four bits of storage. Notice that the numbers range from negative eight to positive seven. In two's complement notation, the range of numbers that can be represented given a fixed number of bits will always include one more negative value than there are positive values. This is because the representation of zero includes a 0 in the leftmost bit position, thus taking up one of the “positive” slots.

Two's complement binary	**Decimal**
0111	+7
0110	+6
0101	+5
0100	+4
0011	+3
0010	+2
0001	+1
0000	0
1111	-1
1110	-2
1101	-3
1100	-4
1011	-5

Two's complement binary	Decimal
1010	-6
1001	-7
1000	-8

For *n* bits of storage the range of numbers that can be represented in two's complement notation extends from -2^{n-1} to $+2^{n-1}-1$. In the table above, four bits of storage were used; therefore, the range of numbers was from $-2^{4-1} = -2^3 = -8$ to $2^{4-1}-1 = 2^3-1 = 7$. Most modern computers use 32 bits to represent integers. Thus, they are capable of representing values in the range $-2^{31} = -2{,}147{,}483{,}648$ to $2^{31}-1 = 2{,}147{,}483{,}647$.

What do you suppose happens if 1 is added to 7? In other words, how does the computer handle 0111 + 0001? Performing binary addition, the result is 1000. In the table, this is -8! Does this mean that 7 + 1 = -8? If the computer stored integers using only four bits, then indeed this would be the case! Counting essentially wraps around the table. That is, 7 + 1 = -8 and -8 – 1 = 7. The four bits of storage only allow numbers in this range.

In order to understand why two's complement is the preferred method for representing signed numbers at the machine level, we will look at a number of addition problems involving both positive and negative numbers. As we will see, what makes two's complement so great is that the sign of a number can essentially be ignored when performing addition (since positive and negative numbers are treated in an identical manner). An added bonus is that we get the subtraction operations for "free" once we have addition: a problem such as X – Y can be recast as the following two step process:

(1) Swap the sign of Y; and
(2) Add X and Y.

Let's begin by examining the summation of numbers with opposite signs. First, the sum of +2 and -3 (which should sum to -1). Assuming four bits of storage, the two's complement binary representation of +2 is 0010. The two's complement representation of -3 is 1101. Why? To obtain -3, we begin with +3 (0011), complement the bits (1100), and add one (1101). Here's the addition of +2 and -3:

	Binary				Decimal
Carry	0	0	0		
1st number	0	0	1	0	+2
2nd number	1	1	0	1	-3
Sum	1	1	1	1	-1

The 1 in the leftmost column of the result tells us that the number is negative. Its magnitude can be determined by complementing each bit (0000) and adding one (0001). Thus, the addition of +2 and -3 is 1111 (or -1).

Next, let's look at the addition of -2 and +3. The two's complement binary representation of -2 is 1110 (start with +2 which is 0010, complement the bits to obtain 1101, and add one to obtain 1110). The

representation of +3 is 0011. Adding these two values together gives +1 or 0001 in binary two's complement, as is illustrated below:

	Binary				Decimal
Carry	1	1	1	0	
1st number		1	1	1	0	-2
2nd number		0	0	1	1	+3
Sum	*1*	0	0	0	1	+1

Note that this particular addition of two four-bit numbers results in a carry to a fifth bit. We saw this before (when adding +5 and -5). This carry is ignored. In order to reinforce the fact that this bit is discarded, it is shown in italics.

Let's now turn our attention to addition problems involving numbers of the same sign. Here is an illustration of the addition of two positive numbers: +2 (0010) and +3 (0011):

	Binary				Decimal
Carry	0	1	0		
1st number	0	0	1	0	+2
2nd number	0	0	1	1	+3
Sum	0	1	0	1	+5

To show that the system handles the addition of negative numbers properly, consider adding -2 (0010 → 1101 → 1110) and -3 (0011 → 1100 → 1101). The result should be -5 (0101 → 1010 → 1011):

	Binary				Decimal
Carry	1	1	0	0	
1st number		1	1	1	0	-2
2nd number		1	1	0	1	-3
Sum	*1*	1	0	1	1	-5

The result is indeed 1011. The 1 in the leftmost bit indicates that the result is a negative number. We can determine its magnitude by implementing the two's complement operations on it: complement the bits 1011 to obtain 0100, and add one to obtain 0101.

Notice that while we get a correct answer, this problem also generated a carry bit that is discarded. One problem that can arise when representing numeric values via a fixed number of bits is the problem of overflow.

Definition: ***Overflow*** *occurs when the value that is to be stored is outside the range of permissible values (i.e., the value is too large to fit in the available space).*

The only way around overflow is to add more bits to the representation, thus increasing the range of permissible values. The best we can do in place of this is to detect when overflow occurs.

Two's complement notation makes it easy to spot when overflow occurs: when two numbers of the same sign are added together and the result has the opposite sign. Here are two examples; first, the sum of +4 and +5 (which are both positive numbers):

	Binary				Decimal
Carry	1	0	0		
1st number	0	1	0	0	+4
2nd number	0	1	0	1	+5
Sum	1	0	0	1	-7?

Clearly, the result is not -7. Note that the result (-7) has a different sign than the two numbers (+4 and +5). Here's another example:

	Binary				Decimal
Carry	1	0	0	0	
1st number		1	1	0	0	-4
2nd number		1	0	1	1	-5
Sum	*1*	0	1	1	1	+7?

Overflow in two's complement can be spotted when the sign bits of both numbers being added are the same, yet the sign bit of the answer is different. Note that overflow can only occur when adding numbers of like sign. It can never occur when numbers of opposite signs are added.

When an overflow is detected, the answer is incorrect, as shown above by the question marks, and must be discarded. There is no way to get the correct answer if the number of bits available does not allow us to express that answer. There is no way to express +9 or -9 (the correct results to the examples above) using a four-bit two's complement number, since both are outside the range of permissible values.

Characters

We now turn our attention to representing characters at the machine level. One of the most popular methods of representing character data in a computer is to use the ASCII character set. The basic idea behind **ASCII** (American Standard Code for Information Interchange), and in fact all other character sets, is to associate particular bit patterns with individual characters.

There are 128 symbols in the standard ASCII character set. These are numbered from 0 to 127. ASCII characters are grouped together in the following way:

Character group	Range	
	Decimal	Hexadecimal
Control characters	0 – 31	0 – 1F
Punctuation	32 – 47	20 – 2F
Digits	48 – 57	30 – 39
More punctuation	58 – 64	3A – 40
Uppercase letters	65 – 90	41 – 5A
More punctuation	91 – 96	5B – 60
Lowercase letters	97 – 122	61 – 7A
More punctuation	123 – 126	7B – 7E
One final control character	127	7F

Control characters refer to *nonprinting* characters (e.g., return/enter, linefeed, delete, and so on). Punctuation refers to any printable character that is not a digit or letter (e.g., [, {, |, \, and so on).

The following table contains the hexadecimal representations of all of the printable ASCII characters. These range from the space character (with an ASCII value of $20_{16} = 32_{10}$) to the tilde (with an ASCII value of $7E_{16} = 126_{10}$). The ASCII value for a character may be found in the table below by locating the row and column in which the character appears. The row specifies the high-order (leftmost) hexadecimal digit, and the column specifies the low-order (rightmost) hexadecimal digit. For example, the ASCII value for the uppercase letter H is 48_{16}, since it appears in row 4 and column 8. To determine the decimal version of the ASCII value, you must perform the conversion from hexadecimal to decimal (e.g., $48_{16} = 72_{10}$).

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	[space]	!	“	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	I	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

While any assignment of bit patterns to characters would accomplish the primary task of enabling computers to store character data, a lot of thought went into the design of ASCII. For example, uppercase and lowercase characters differ in ASCII by exactly one bit (e.g., $A = 41_{16} = 01000001_2$, and $a = 61_{16} = 01100001_2$). Therefore, any lowercase character can be changed to upper case by simply setting bit six to 0. Conversion from uppercase to lowercase is just as easy: set bit six to 1!
The concept of mapping characters to bit patterns is logical; however, characters typed on a keyboard don't show up on the screen as ASCII values. This is because ASCII values are hidden from the computer user. When you type “patootie” on your keyboard, you first hit a key marked “p”. This

causes the keyboard circuitry to generate the ASCII code 70_{16}. Next, you strike the key marked "a", causing the keyboard circuitry to generate the ASCII code 61_{16}. This process occurs for the entire word. The display circuitry built into your computer's video card then displays the characters of the word as it receives the ASCII values for each character. Since the internal circuitry of both the keyboard and video card handle the translation and processing of ASCII values, you never see the ASCII values.

It is interesting to note what happens when the shift key is held down as, for example, the word "patootie" is typed. What the shift key actually does (at least for the keys marked with letters of the alphabet) is to set bit six to 0. Thus, the combination of *shift* and *p* (which is $70_{16} = 01110000_2$) produces $50_{16} = 01010000_2$ (which is the ASCII value for *P*).

Floating point numbers

Most modern personal computers are based on a bus size of thirty-two bits. Using two's complement notation, they can conveniently manipulate numbers in the range of approximately -2 billion to +2 billion. Many times, we humans need our computers to work with numbers far outside of this relatively narrow range. For example, scientific applications often need to represent very small or very large quantities, such as Avogadro's number: $6.02 * 10^{23}$ molecules/mole. Even a program designed to make projections about the national debt of the United States, which is measured in trillions of dollars, will be forced to work with numbers outside of the range provided by the 32-bit two's complement representation. In addition to the problem of being limited to a relatively narrow range of values, the two's complement binary notation we studied cannot represent fractions. This makes it impossible to express concepts like one-half and 75%.

Floating point notation, which is closely related to exponential notation, addresses both of these problems. As you will recall from math or science classes, exponential notation is often used to express very small or very large numbers. For example, the speed of light can be expressed as $3.0 * 10^8$ meters/second.

The two components necessary to express a number in exponential notation are called the **mantissa** (3.0 in the above number representing the speed of light) and the **exponent** (8 in the above number representing the speed of light). These values are expressed in base ten, so the exponent is expressed in terms of base ten as well. Essentially, the exponent tells us the number of positions that the decimal point should be moved to the right or left. Positive exponents cause the decimal point to move to the right; negative exponents cause the decimal point to move to the left. Hence, the representation for the speed of light translates to a 3 followed by eight 0s (i.e., three hundred million meters per second). Fractions can also be expressed in this system; for example, $1/1000 = 1.0 * 10^{-3} = 0.001$.

Floating point notation is the machine level equivalent of exponential notation. Floating point numbers include signed representations of both the mantissa and the exponent. Since the representations of both will be in base two, it will be considered the base of the exponent.

As an example, here is the number 56.0 expressed using one possible 32-bit floating point representation:

+ -	Exponent								Mantissa																							
0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

The leading bit (bit 32) of this representation signifies the sign of the number (0 for positive, 1 for negative). Specifically, it represents the sign of the mantissa. The next eight bits (bits 31 to 24) represent the exponent in two's complement notation (remember the exponent can be positive or negative). The final 23 bits are the mantissa expressed as an unsigned number. The number represented in this example (in binary form) is $111 * 2^{11}$ which in decimal form becomes $7 * 2^3 = 7 * 8 = 56.0$.

Another way of thinking about this example is to view the exponent as specifying the number of 0s that are to follow the bit pattern supplied by the mantissa. In this case, we have 111 followed by three 0s, or $111000_2 = 56$.

Here is an example of a number that is outside the range of values that can be stored using a 32-bit two's complement notation. The number (in binary form) is $11 * 2^{100000}$, which in decimal form is $3 * 2^{32} = 12{,}884{,}901{,}888$:

+ -	Exponent								Mantissa																							
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

Another way of thinking about the number represented by this bit pattern is 11 followed by thirty-two 0s. Now, consider the representation of -0.75 as a floating point number:

+ -	Exponent								Mantissa																							
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

The 1 in the leftmost bit signifies that the number is negative. The exponent, 11111110_2 is also negative due the 1 in its leftmost bit (remember that the exponent is represented in two's complement notation). The magnitude of the exponent is 2 (complement the bits to get 00000001_2, and add one to get $00000010_2 = 2_{10}$). The magnitude of the mantissa is $11_2 = 3_{10}$. Therefore, the value is:

$$-3 * 2^{-2} = -3 * \frac{1}{2^2} = -3 * \frac{1}{4} = -3 * 0.25 = -0.75$$

Another way of thinking about the number in this example is to view it (in binary form) as -0.11. The columns to the right of the decimal point are the 1/2's column, the 1/4's column, the 1/8's column, and so on. The number -0.11 would then be interpreted as having a 0 in the one's place, a 1 in the 1/2's place, and a 1 in the 1/4's place, producing -3/4 (or -0.75). Using this interpretation, the mantissa always has an implied decimal point immediately to its right. In the example above, this is viewed as 11. (note the decimal point to the right). The exponent then specifies the number of places that the

decimal point should be moved (to the right for positive exponents, or to the left for negative exponents). In the example above, the exponent moves the decimal point two places to the left giving 0.11. The binary number is then interpreted in the following way:

8	4	2	1		1/2	1/4
0	0	0	0	.	1	1

To work backwards (i.e., to convert -0.75 to binary), we must first break the number down into two parts, each separated by the decimal point. The part of the number that is to the *left* of the decimal point is easily converted using the remainder method for unsigned numbers shown in a previous lesson. Recall, that we repeatedly divide the decimal number by two and build its binary representation by using the remainders generated, from right-to-left. In this case, however, it's simple in that $0_{10} = 0_2$.

The part of the number that is to the *right* of the decimal point is effectively done in reverse when compared to the remainder method. That is, it is repeatedly *multiplied* by two. The binary representation is built by using the bits generated to the left of the decimal point, from left-to-right. Specifically, we multiply the decimal number by two and separately record both parts of the result to the left and right of the decimal point. We then repeat this process with the part to the right of the decimal point, while keeping track of the parts to the left. This is repeated until the part to the right of the decimal point is 0. The binary equivalent of the original number (remember that it's the part to the right of the decimal point in the original number) is subsequently given by listing the parts to the left of the decimal point in the order of their derivation (i.e., from the first identified to the last).

This may be a bit confusing, so let's try this method on the decimal number -0.75. As shown above, the 0 to the left of the decimal point is easily converted to binary: 0. We'll use the following table to convert the part to the right of the decimal point, .75:

Value	Times	Two	Equals	Product	Left part	Right part
.75	*	2	=	1.5	1	.5
.5	*	2	=	1.0	1	.0

Since the part to the right of the decimal point is 0, the process is finished. Combining the "left part" results produces 11 as the part to the right of the decimal point in the final answer. To complete the conversion, we concatenate the parts to the left and right of the decimal point. Therefore, the binary representation of -0.75_{10} is -0.11_2. From this point, we could generate the table shown earlier that breaks -0.75 down into a sign bit, an exponent, and a mantissa. We know that the sign bit is 1:

+ -	Exponent								Mantissa																							
1																																

The mantissa is obtained by moving the decimal point all the way to the right: .11 → 1.1 → 11. The mantissa is therefore 11. We also note that it was moved two decimal places:

+ -	Exponent								Mantissa																	
1									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

Lastly, we know that the exponent is -2 (since we'll need to move the decimal point 2 places to the left in the mantissa). We know how to convert -2 to binary by starting with the binary representation of 2, complementing the bits, and finally adding one: 00000010 → 11111101 → 11111110_2:

+ -	Exponent								Mantissa																	
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

Let's try another example and convert +5.5625 to binary. First, the 5 to the left of the decimal point: $5_{10} = 101_2$. Next, we'll convert the part to the right of the decimal point using the method described earlier:

Value	Times	Two	Equals	Product	Left part	Right part
.5625	*	2	=	1.125	1	.125
.125	*	2	=	0.25	0	.25
.25	*	2	=	0.5	0	.5
.5	*	2	=	1.0	1	.0

Therefore, $5.5625_{10} = 101.1001_2$. Similarly, we can fill the table below, specifying the sign bit, exponent, and mantissa. First, the number is positive; therefore, the sign bit is 0:

+ -	Exponent								Mantissa																	
0																										

Next, the mantissa is 1011001 (we shifted the decimal point all the way to the right):

+ -	Exponent								Mantissa																	
0									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

Finally, the exponent is -4 (since the decimal point needs to be moved four places to the left). We convert -4 to binary using two's complement notation: 00000100 → 11111011 → 11111100_2:

+ -	Exponent								Mantissa																	
0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

Working in reverse, we can actually recreate the decimal representation of the number. We know that the number is positive because the sign bit is 0. We can convert the exponent using two's complement notation: 11111100 → 00000011 → 00000100_2 = -4. And we can convert the mantissa: 1011001_2 = 89_{10}.

We then combine the components into exponential notation (using two as the base for the exponent):

$$89 * 2^{-4} = 89 * \frac{1}{2^4} = 89 * \frac{1}{16} = 89 * 0.0625 = 5.5625$$

Let's try another example, converting -16.125 to binary. First, the left part: $16_{10} = 10000_2$. Next, the right part:

Value	Times	Two	Equals	Product	Left part	Right part
.125	*	2	=	0.25	0	.25
.25	*	2	=	0.5	0	.5
.5	*	2	=	1.0	1	.0

Therefore, $-16.125_{10} = 10000.001_2$. As before, we can fill the floating point table. First, the number is negative; therefore, the sign bit is 1:

+ -	Exponent								Mantissa																			
1																												

Next, the mantissa is 10000001 (we shifted the decimal point all the way to the right):

+ -	Exponent								Mantissa																			
1									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Finally, the exponent is -3 (since the decimal point needs to be moved three places to the left). We convert -3 to binary using two's complement notation: 00000011 → 11111100 → 11111101_2:

+ -	Exponent								Mantissa																			
1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Again, we can recreate the decimal representation of the number. We know that the number is negative because the sign bit is 1. We can convert the exponent: 11111101 → 00000010 → 00000011_2 = -3. And we can convert the mantissa: $10000001_2 = 129_{10}$. We then combine the components:

$$-129 * 2^{-3} = -129 * \frac{1}{2^3} = -129 * \frac{1}{8} = -129 * 0.125 = -16.125$$

Let's try one last (perhaps more complicated) example, converting -2337.7109375 to binary. First, the left part: $-2337_{10} = 100100100001_2$. Now, the right part:

Value	Times	Two	Equals	Product	Left part	Right part
.7109375	*	2	=	1.421875	1	.421875
.421875	*	2	=	0.84375	0	.84375
.84375	*	2	=	1.6875	1	.6875
.6875	*	2	=	1.375	1	.375
.375	*	2	=	0.75	0	.75
.75	*	2	=	1.5	1	.5
.5	*	2	=	1.0	1	.0

Therefore, $-2337.7109375_{10} = -100100100001.1011011_2$. Again, we can fill the floating point table. First, the number is negative; therefore, the sign bit is 1:

+ -	Exponent								Mantissa																			
1																												

Next, the mantissa is 10010010000110110 11 (we shifted the decimal point all the way to the right):

+ -	Exponent								Mantissa																			
1									0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	0	1	1	0	1	1

Finally, the exponent is -7 (since the decimal point needs to be moved seven places to the left). We convert -7 to binary using two's complement notation: $00000111 \rightarrow 11111000 \rightarrow 11111001_2$:

+ -	Exponent								Mantissa																			
1	1	1	1	1	1	0	0	1	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	0	1	1	0	1	1

Again, we can recreate the decimal representation of the number. We know that the number is negative because the sign bit is 1. We can convert the exponent: $11111001 \rightarrow 00000110 \rightarrow 00000111_2 = -7$. And we can convert the mantissa: $10010010000110110_2 = 299227_{10}$. We then combine the components:

$$-299227 * 2^{-7} = -299227 * \frac{1}{2^7} = -299227 * \frac{1}{128} = -299227 * 0.0078125 = -2337.7109375$$

Did you know?

It is interesting to note that some fractions that can be expressed precisely in decimal notation, such as 1/10 = 0.1, do not have an exact floating point representation. This is due to the fact that floating point numbers are represented using base two. One-tenth can be approximated in binary as 0.0001100110011001... (where 1001 repeats indefinitely). It can never be represented exactly. This result should not be surprising. After all, many factions (e.g., 1/3) cannot be represented exactly as decimal values. This is one reason why computed results that involve fractions are not always 100% accurate. They sometimes suffer from round off error as a result of the base ten to base two conversion process.

Combinational circuits

Recall that combinational circuits are digital circuits that do not involve any kind of feedback. In other words, the output of a combinational circuit cannot be fed back into that circuit as input. Previously, only three types of combinational circuits were covered: the *xor* gate, comparators (for equality, less than, and greater than), and adders (half adder, full adder, chaining full adders). There are several more types of combinational circuits that are important, particularly in the design of computers.

Decoders and encoders

Definition: *A **decoder** is a type of circuit that takes in a number (in unsigned binary form) and generates a 1 (high) on the output line that corresponds to the input number. All other output lines are set to 0 (low).*

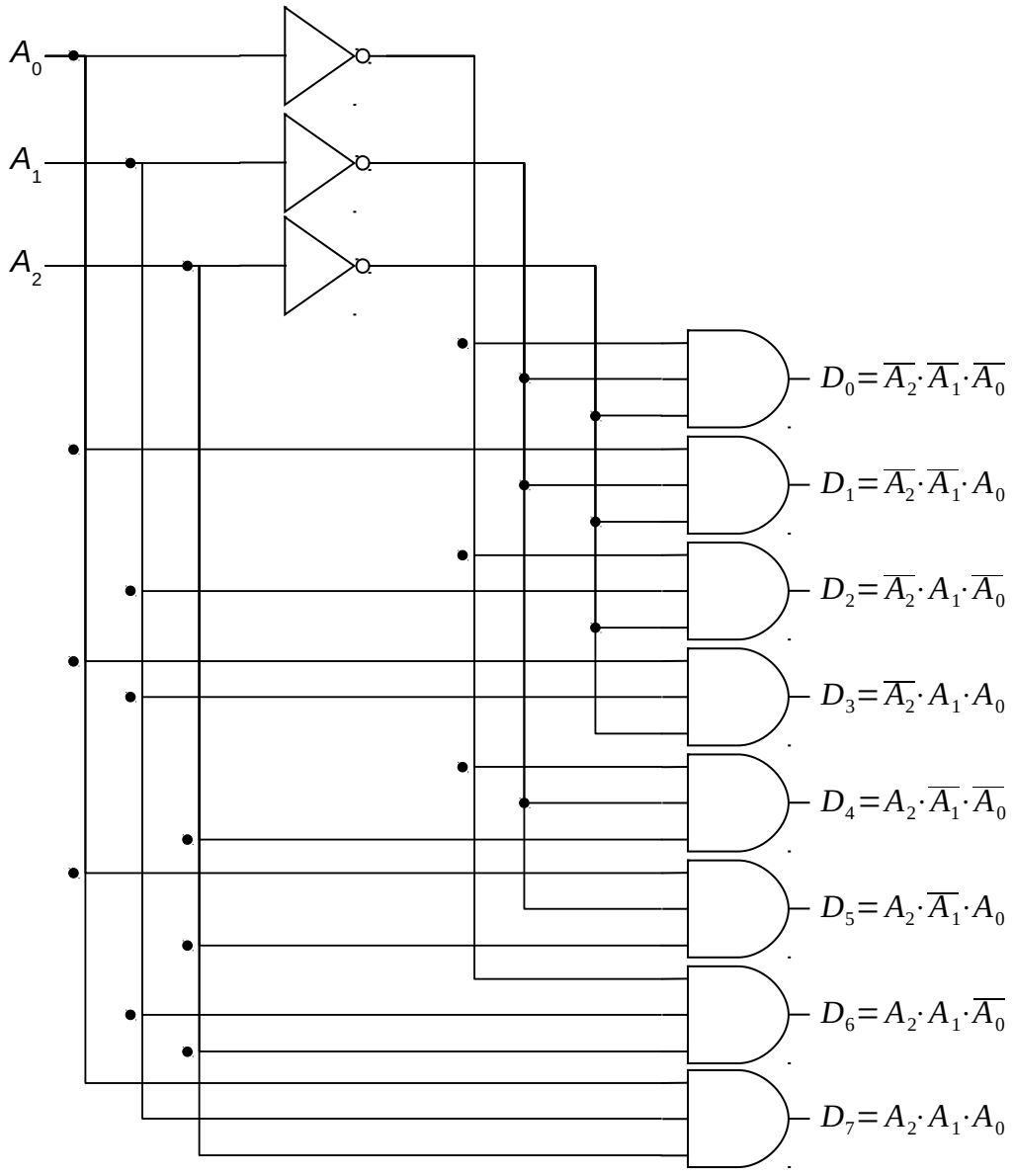
For example, given an input of 00_2, a 1 would be generated on output line zero. Likewise, given 01_2 as input, a 1 would be placed on output line one. You will see later how decoders form an integral part of memory and register addressing circuitry.

Every decoder with *n* input lines will have exactly 2^n output lines. Therefore, a *two-to-four* decoder will have *two* input lines and *four* output lines, while a *three-to-eight* decoder will have *three* input lines and *eight* output lines. Both the input and output lines of decoders are numbered, with the *n* inputs ranging from 0 to *n*-1, and the 2^n outputs ranging from 0 to $2^n - 1$. Here is the truth table for a three-to-eight decoder:

A_2	A_1	A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

The inputs are labeled A_0 to A_2 and represent the bits of a three-bit unsigned binary number. The outputs are labeled D_0 through D_7. As the table shows, an input number (such as $110_2 = 6_{10}$) results in the corresponding data line (D_6 in this case) being set to 1, with all other lines held at 0.

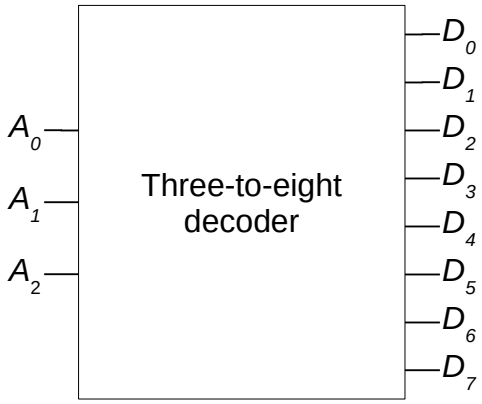
The following circuit illustrates an implementation of a three-to-eight decoder. The circuit diagram uses eight three-input *and* gates (one for each data line), together with a total of three *not* gates. If you don't happen to have access to three-input *and* gates, remember that they can easily be constructed from two standard two-input *and* gates.



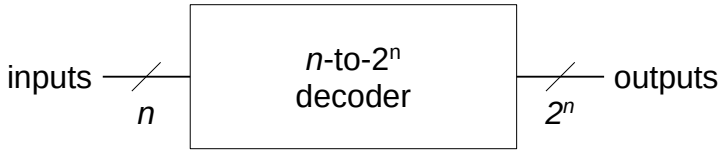
Carefully tracing the lines in the circuit diagram, we see that all three of the inputs to the D_0 *and* gate are negated. Thus, when inputs A_2, A_1, and A_0 each have a value of 0, D_0's three-input *and* gate receives three 1s (i.e., *not* A_2, *not* A_1, and *not* A_0) and generates a 1 as output. The result is that a 1 (high) is placed on output line zero when the number 000_2 is given as input to the circuit.

Let's continue to the *and* gate for D_1. Note that two of the three inputs (A_2 and A_1) are negated, but the third input (A_0) is not. Thus, when inputs A_2 and A_1 are 0, but A_0 is 1 (corresponding to the input number 001_2 – or 1_{10}) the *and* gate will receive three 1's and generate a 1 on line D_1. Skipping ahead to the final case, note that the three-input *and* gate for D_7 receives all of its inputs directly from A_2, A_1, and A_0 without negation. Thus, when A_2, A_1, and A_0 each contain 1 (corresponding to the number 111_2 – or 7_{10}) the gate will generate a 1 on data line D_7. The other cases are handled in a similar manner.

The three-to-eight decoder may be encapsulated using a "black box" (as you've seen before) as follows:



In general, since an *n* input decoder will always have 2^n outputs, its "black box" can be expressed in the



The following symbol (along with a number or variable), is frequently used as a shorthand in circuit design to represent the indicated number of lines without actually drawing them. Since modern computers are based on 32-bit buses, this compact representation is very important.

It is natural to wonder at this point if there is a circuit that does the exact opposite of a decoder. Of course there is!

Definition: *An **encoder** is a type of circuit with 2^n input lines, only one of which can be high at any given time, that produces an n-bit unsigned binary number that corresponds to the raised input line.*

A four-to-two encoder is defined by the following truth table:

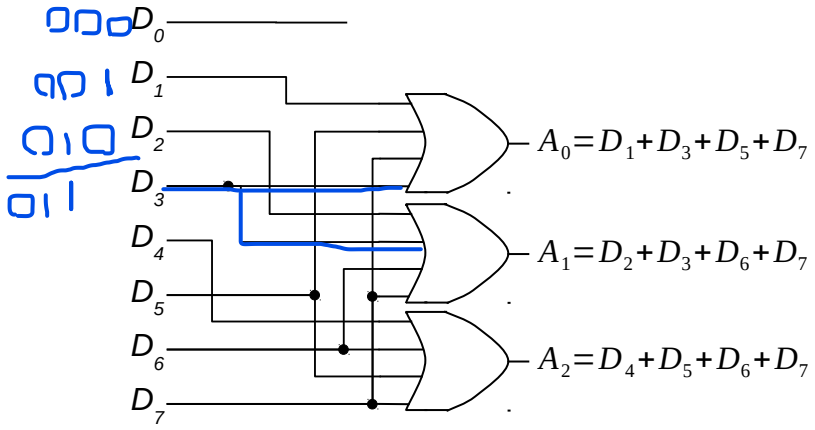
D_3	D_2	D_1	D_0	A_1	A_0
0	0	0	0	?	?
0	0	0	1	0	0
0	0	1	0	0	1

D_3	D_2	D_1	D_0	A_1	A_0
0	0	1	1	?	?
0	1	0	0	1	0
0	1	0	1	?	?
0	1	1	0	?	?
0	1	1	1	?	?
1	0	0	0	1	1
1	0	0	1	?	?
1	0	1	0	?	?
1	0	1	1	?	?
1	1	0	0	?	?
1	1	0	1	?	?
1	1	1	0	?	?
1	1	1	1	?	?

Note that twelve of the sixteen rows of the truth table are invalid. These rows represent input configurations that are disallowed because either no input line is high, or multiple input lines are high. In these cases, there is no single high input line for the circuit to return the corresponding binary number of.

How are we to deal with these *undefined* configurations when constructing the corresponding circuit? One way is to simply ignore the disallowed input configurations. This approach will work fine as long as we can be *sure* that the illegal states can never occur.

The following presents an implementation of an eight-to-three encoder that assumes that disallowed states will never be encountered. The circuit works fine as long as this limitation is respected. For example, placing a 1 on line D_6, while holding all other lines low, causes the number 110_2 (or 6_{10}), to be generated. An invalid configuration of inputs generates an erroneous output. For example, setting both D_1 and D_2 to 1 causes the circuit to output 011_2 (or 3_{10}).

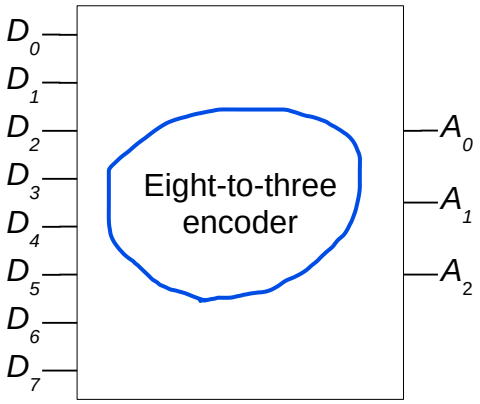


One odd feature of this circuit is that input line D_0 is not connected to anything. It is, in a sense, ignored. Although this may seem strange at first, it is precisely what we want the circuit to do. Setting line D_0 to 1 (and all other input lines to 0) is supposed to cause all of the output lines to be set to 0 in order to represent the number 000_2.

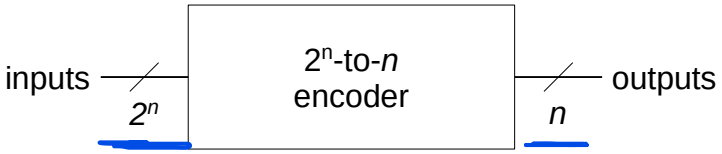
The truth table for the eight-to-three encoder would be similar to the one shown above for the four-to-two encoder. However, the full truth table would consist of 256 rows (since it has eight input lines and therefore $2^8 = 256$ possible input configurations). All but eight of these input configurations would be disallowed. In order for the circuit's truth table to fit on a page, only the allowed rows are presented below:

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_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

As you can see, this truth table is the exact inverse of the truth table for the three-to-eight decoder that was presented earlier. The eight-to-three encoder can be encapsulated into a "black box" as follows:



In general, a 2^n to n encoder can be drawn as follows:



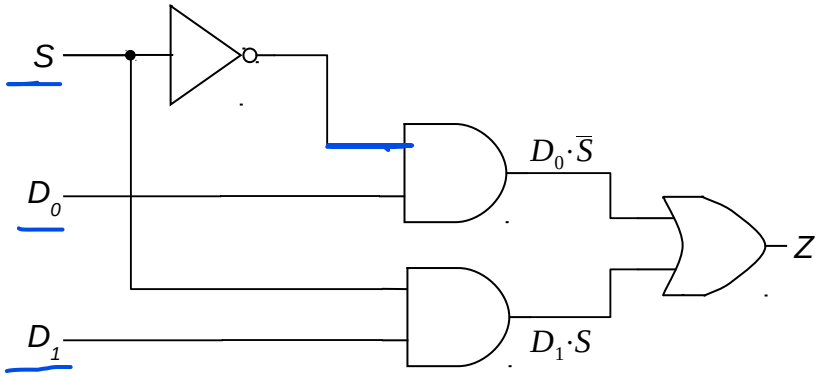
Multiplexers and demultiplexers

Definition: ***Multiplexers*** *are circuits that are used to transfer the contents of an input data line to an output line based on the value of one or more input selector lines.*

While that definition sounds a bit intimidating, multiplexers really aren't that complex. Essentially, a multiplexer is a kind of switch. It has two types of inputs: data lines and selector lines; it has a single output line. The purpose of a multiplexer is to transfer the contents of *one* of the input lines to the circuit's single output line. The values placed on the selector lines determine which data line will have its contents transferred to the output line. Thus, the selector lines allow the multiplexer circuit to switch its "attention" between the various input data lines when determining the value to be output.

Generally, a multiplexer (MUX) will have 2^n data lines for n selector lines. The data lines are numbered 0 to $2^n - 1$, and the selector lines are numbered 0 to $n - 1$. The bit pattern placed on the selector lines, when interpreted as an unsigned binary number, determines the active data line. Multiplexers are often referred to in terms of their number of data lines. Therefore, an eight-input MUX will have eight data lines and three selector lines, while a two-input MUX will have two data lines and a single selector line. Remember that, regardless of the number of input data and selector lines, every MUX has only one output line.

The following presents an implementation of a two-input multiplexer. It has two data lines, one selector line, and a single output. When the selector line, S, is set to 0, the current value of D_0 (either a 1 or a 0) becomes the output of the circuit, regardless of the value of line D_1. The circuit is, in a sense, *listening* to D_0 and ignoring D_1. When S is 1, the opposite situation exists: the output of the multiplexer becomes the current value of D_1, and D_0 is ignored.

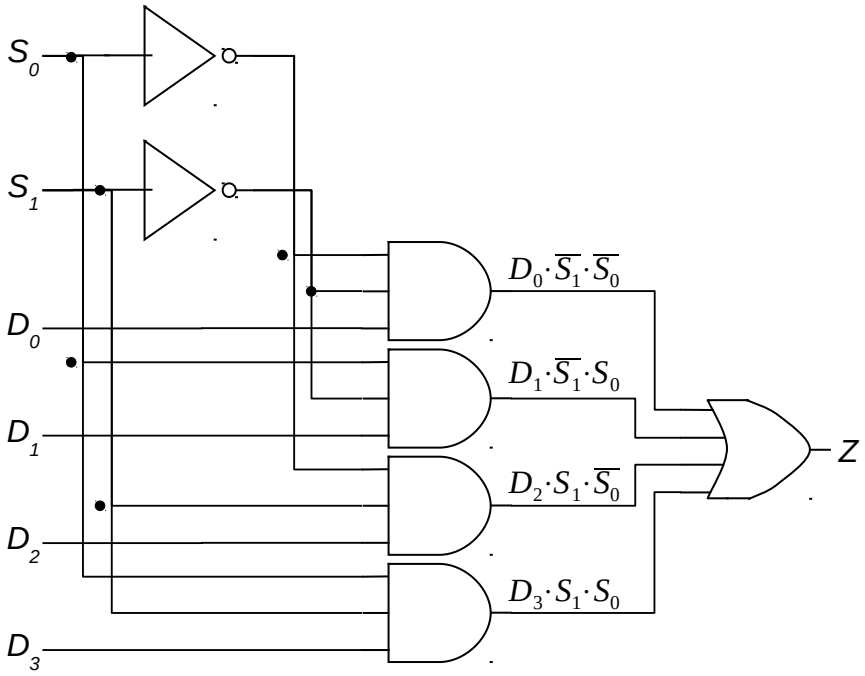


The behavior of the two-input multiplexer is summarized in the following truth table:

S	D_1	D_0	Z
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

The following illustrates a somewhat more complex multiplexer, a four-input MUX. It has two selector lines, four data lines, and a single output line. Placing a 0 on both S_1 and S_0, corresponding to 00_2, causes the value of D_0 to be transferred to the output line. Setting S_1 to 0 and S_0 to 1, corresponding to 01_2, causes D_1 to be transferred to the output line. Likewise, setting S_1 to 1 and S_0 to 0, corresponding to 10_2, causes D_2 to be transferred to the output line. Finally, setting S_1 to 1 and S_0 to 1, corresponding to 11_2, causes D_3 to be transferred to the output line:



An inspection of the circuit diagram for the four-input MUX reveals that it contains four three-input *and* gates, two single-input *not* gates, and one four-input *or* gate. Each *and* has one of the data lines running into it, plus two selector signals. Some of the selector signals have been routed directly from the selector input lines, while others have been negated before being sent on to the *and* gates. The outputs of all four of the *and* gates are routed into the four-input *or*. The output of this *or* becomes the output of the MUX circuit. If any one of the *and* gates generate a 1, the circuit will output a 1. If all of the *and* gates produce 0, the circuit will output a 0.

In order to better understand the behavior of this multiplexer, let's examine the conditions under which each of the *and* gates could fire. A three-input *and* gate can produce a 1 only if all of its inputs are 1. Thus, the two selector signals and the data value reaching the *and* gate must be 1 for that gate to generate a 1.

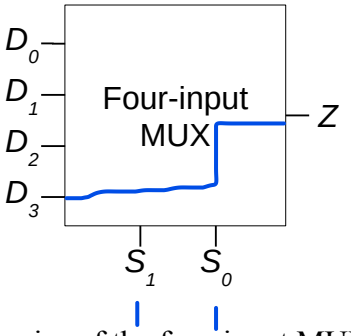
The selector signals reaching the gate that is connected to D_0 are *not* S_1 and *not* S_0. This gate can produce a 1 only when $S_1 = 0$, $S_0 = 0$ and $D_0 = 1$. Under any other circumstances (e.g., if D_0 is 0 or if either S_1 or S_0 is 1) this *and* gate produces a 0. Hence, this part of the circuit faithfully transfers the

value of D_0 when the S_1,S_0 bit pattern is 00. Just as importantly, this *and* gate stays low when the selector bit pattern is not 00, regardless of the value of D_0.

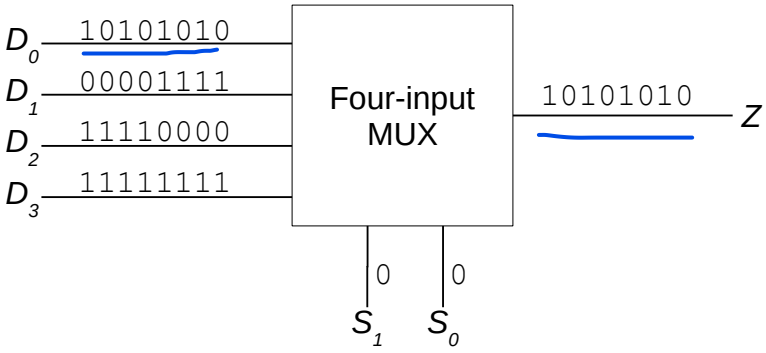
The other *and* gates act in a similar manner. The gate for D_1 is attached to *not* S_1 and S_0; therefore, it only generates a 1 when $S_1 = 0$, $S_0 = 1$, and $D_1 = 1$. Other selector bit patterns keep it low. The *and* gate that receives the D_2 signal is connected to S_1 and *not* S_0. This gate produces a 1 only when $S_1 = 1$, $S_0 = 0$, and $D_2 = 1$. It generates a 0 at all other times. Finally, the *and* gate for D_3 is attached directly to S_1 and S_0. Thus, it generates a 1 when $S_1 = 1$, $S_0 = 1$, and $D_3 = 1$.

Because of the way the selector signals are routed to the various *and* gates, it is impossible for more than one of them to produce a 1 at the same time. For this reason, the results of the *and* gates can be safely combined via an *or* gate without worrying that signals from multiple data lines will be accidentally combined.

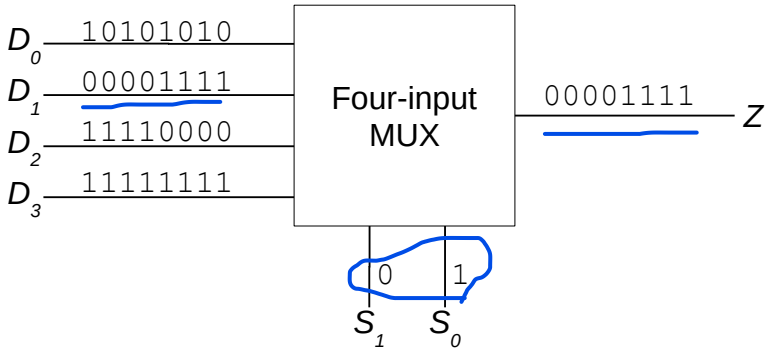
The four-input multiplexer can be represented by a "black box" as follows:



The following figures illustrate the behavior of the four-input MUX over time. The input lines and output lines are labeled with the data streams flowing down them. During the period of time illustrated in the following figure, selector lines S_1 and S_0 are both held at zero, making D_0 the *active* data channel:



In the following figure, S_1 is held at zero and S_0 at one, thereby activating D_1:



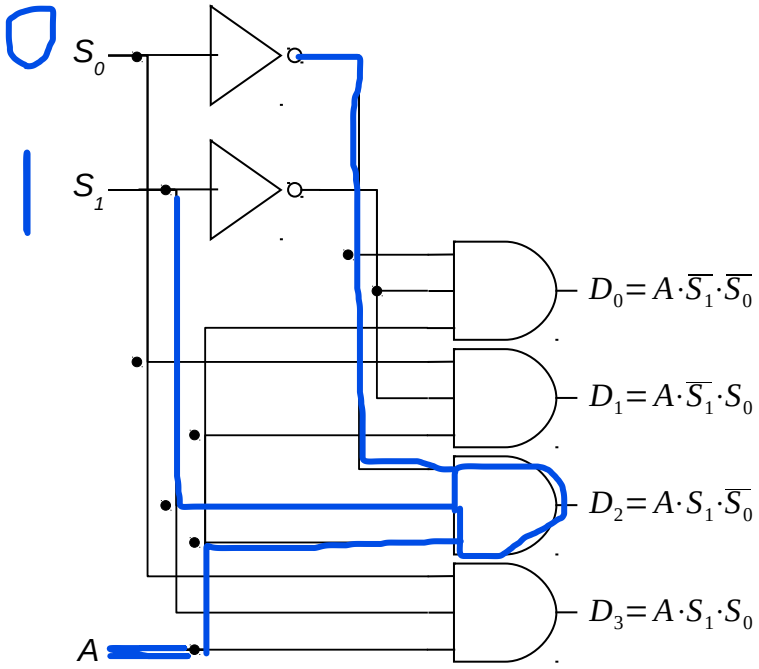
While the selector lines are held steady, the current state of each of the data lines varies over time as data moves across the lines (from left-to-right; therefore, the data on the lines is read from right-to-left). For example, in both of the previous figures, line D_0 first contains a 0, then its value changes to 1, then back to 0, then to 1, then 0, then 1, and so on. Line D_1 begins by broadcasting four consecutive 1s followed by four consecutive 0s.
Notice that when D_0 is active (i.e., $S_1 = 0$, $S_0 = 0$), its bit pattern, 01010101, is copied to the output channel. Likewise, when D_1 is active (i.e., $S_1 = 0$, $S_0 = 1$), its bit pattern, 11110000, is copied to the output channel.

Just as with the decoder above, the multiplexer has an exact opposite circuit: the demultiplexer (DEMUX).

Definition: *A **demultiplexer** has a single data input line, n selector input lines, and 2^n output data lines. Demultiplexers generate a copy of their input data value on the output data line specified by their selector lines*.

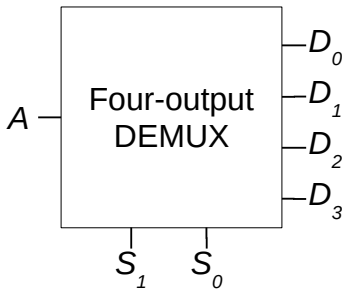
As was the case with multiplexers, the *n* selector lines are numbered from 0 to $n - 1$, and the 2^n output data lines from 0 to $2^n - 1$.

Demultiplexers are often referred to using their number of output lines. The following illustrates an implementation of a four-output demultiplexer:

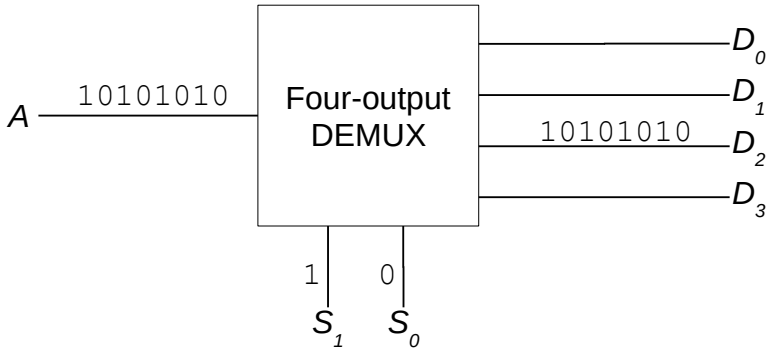


The value of the input data line labeled A is transferred to one of the output data lines, D_0 through D_3, based on an interpretation of the bit pattern in S_1,S_0 as a two-bit unsigned binary number. For example, if $S_1 = 1$ and $S_0 = 0$, corresponding to 10_2 (or 2_{10}), then the current state of the input line would be transferred to D_2. The design of this circuit is quite similar to the one used for the decoder circuit shown earlier. The only difference is that a copy of the input data value is routed to each of the *and* gates so that, instead of simply setting the selected output line high, its value will instead be determined by the input data value.

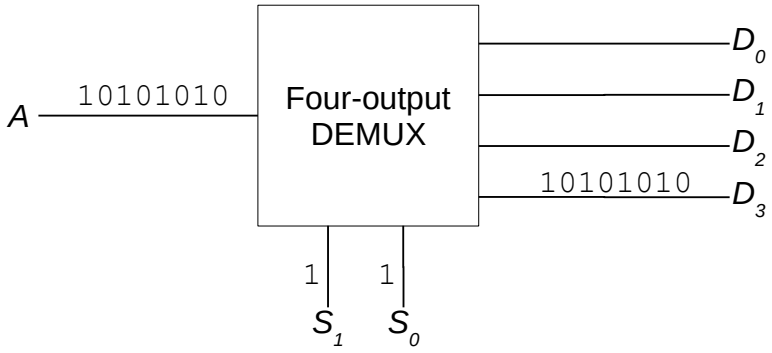
The following is a "black box" representation of the four-output demultiplexer:



The following figures illustrate the behavior of the four-input DEMUX over time. The input lines and output lines are labeled with the data streams flowing down them. During the period of time illustrated in the following figure, selector line S_1 is held at one and S_0 is held at zero, making D_2 the active data channel.



In the following figure, S_1 and S_0 are both held at one, activating D_3:



These examples illustrate how a demultiplexer can *broadcast* an input data stream down one of many different output channels. Changing S_1,S_0 changes the broadcast to a different output channel.

Sequential circuits

All of the circuits discussed up to this point have been combinational circuits whose outputs depended solely on their inputs. Such circuits do not incorporate feedback and have no "memory" of their previous state. In order to construct a general-purpose computer, circuits capable of remembering instructions, data, and the results of computations are needed.

It is possible to design circuits that exhibit memory by incorporating feedback, in the sense that the outputs of a circuit can be made to depend not only on the circuit's current inputs, but also on its past outputs as well. Such circuits are referred to as **sequential circuits**, since their output may be viewed as a function of a sequence of past inputs.

Definition: ***Sequential circuits*** *have memory because one or more of their outputs are fed back to serve as input. Therefore, a sequential circuit's next output will, in a sense, be a function of its present inputs and its previous outputs.*

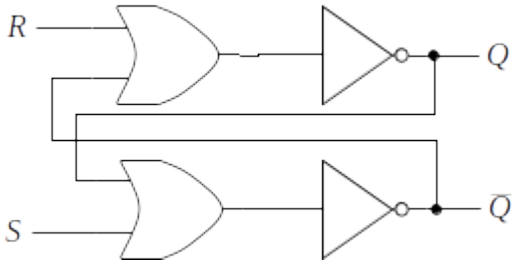
We will limit our study of sequential circuits to one, very important, type of circuit: the flip-flop.

Definition: *The term* ***flip-flop*** *is a generic term applied to devices having two stable states.*

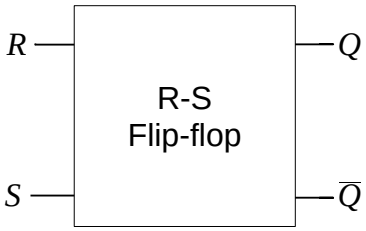
The primary function of a flip-flop is to store a binary digit, 0 or 1. Therefore, a flip-flop can be used to implement the most basic unit of memory, the bit. A flip-flop is implemented as a set of logic gates that make use of feedback to remain in one of two stable states, thereby *remembering* a binary digit.

The R-S flip-flop

An R-S flip-flop can be constructed from two interconnected *nor* gates. The following figure illustrates such an R-S flip-flop. As discussed in a previous lesson, a *nor* gate is simply an *or* gate followed by a *not* gate. For clarity, the following figures shows the two *nor* gates of the flip-flop as having been decomposed into their underlying *or* and *not* gates:



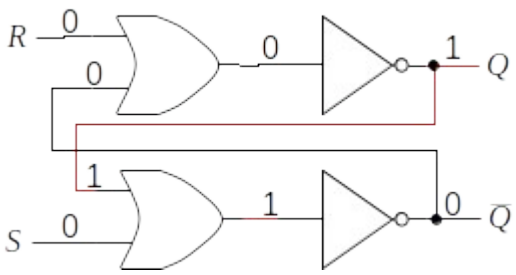
Here's a "black box" representation of the R-S flip-flop:



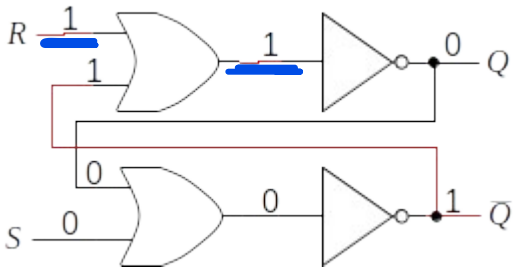
R-S flip-flops have two inputs and two outputs. The inputs are usually labeled R for **Reset** and S for **Set**. The outputs are traditionally labeled Q and $\overline{Q}$ (*not Q*). $\overline{Q}$ is the complement (or opposite) of Q; therefore, if Q is 1 then $\overline{Q}$ should be 0 (and vice versa).

The Q output of the flip-flop determines its state. In other words, examining the Q output is the same as seeing what is stored in the bit. The state can be a binary 1 (Set) or a binary 0 (Reset). When a flip-flop like the R-S is in one of its states (Set or Reset), it will remain unchanged until an appropriate signal or pulse is applied to one of its input lines. The ability of a flip-flop to hold the Q output constant until a signal is given to change the value of Q is why flip-flops are considered basic memory devices.

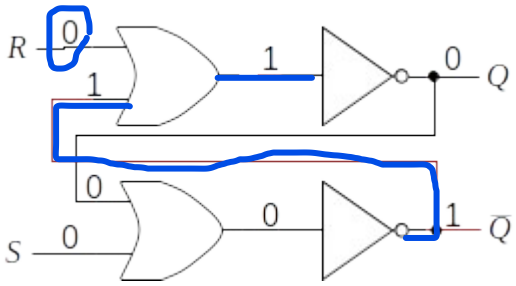
We begin our detailed analysis of the R-S flip-flop assuming that Q is initially high (1), and both R and S are low (0), as shown in the following figure. Since both of the inputs to the top *or* gate are low, its output is low, and the output of the top *not* gate (the Q signal) is high. This high output is fed back into the input of the bottom *or* gate making its output high and the output of the *not* gate (the $\overline{Q}$ signal) low. This is a stable circuit. While R and S remain low, Q will remain high and $\overline{Q}$ low. The bit is thus holding a 1.



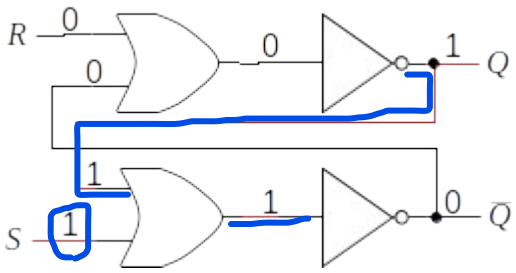
If we apply a 1 to R in order to place a 0 into the bit (Q), the following events occur: (1) the output of the top *nor* gate (the Q signal) goes low; (2) this is fed back to the input of the bottom *nor* gate, which causes its output (the $\overline{Q}$ signal) to go high; and (3) this signal is fed back to the top *nor* gate, but since it already has another high input, there is no change in the output (i.e., it remains low). This is a stable circuit, as shown in the following figure (note that the flip-flop has been reset to 0):



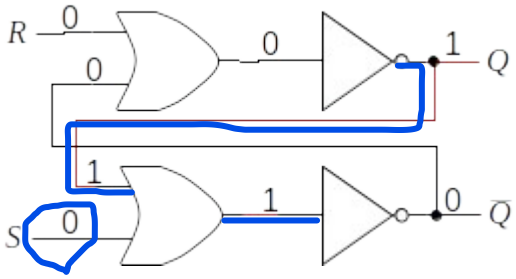
If R is set to low, the flip-flop remains in the Reset (0) state since there is still a 1 input to the top *nor* gate from the $\overline{Q}$ input. This is shown in the following figure. Note that this is exactly the behavior we desire. In order for the flip-flop to be useful, it must be able to *remember* that it has been reset to 0 even after the reset signal is removed.



Next, we apply a high signal to the S input in order to set the bit to 1. This action causes the following events to occur: (1) the output of the bottom *nor* gate becomes 0 since one of its inputs is now high; (2) this low signal is applied as an input to the top *nor* gate, where the R input is also low; and (3) the output of the top *nor* is forced high, and this high output is fed back into the bottom *nor* gate, which does not change its state (since its other input was already high). The flip-flop has been set to 1, as can be seen in the following figure:

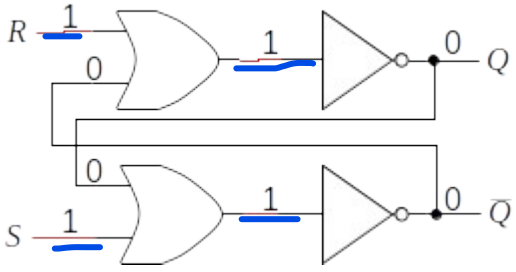


Next, we remove the Set signal. In other words, we change the S input to low. This action does not change the output of the bottom *nor* gate since its other input is already high and the circuit remains in the set state. This is shown in the following figure. Thus, once an R-S flip-flop receives a pulse (or 1) down its set line, it will continue to hold that 1 until a reset signal (i.e., 1 down the reset line) is received.



Notice that state of the flip-flop shown above is identical to its original state (shown at the very beginning of these examples). They both represent the flip-flop in its 1 state, with 0 on both input lines. Comparing the figures to one another illustrates another feature of flip-flops: the value output by the circuit is not solely dependent on its current inputs. When the R and S inputs are both 0, the value output on the Q line depends on whether the most recent 1 input was on the set line or the reset line.

There is one other possible configuration of inputs we have not yet considered. What happens if both the R and S inputs are set to high at the same time (corresponding to an attempt to simultaneously store both a 0 and a 1 into a single bit)? This is pictured in the following figure. The outputs do remain constant, but the R-S flip-flop is not in a valid state because Q and $\overline{Q}$ have identical values, yet they are always supposed to be the opposite of one another.



To conclude the introduction to the R-S flip-flop, here is its truth table:

S	R	Q	$\overline{Q}$
0	0	no change	
0	1	0	1
1	0	1	0
1	1	**disallowed**	

The clocked R-S flip-flop

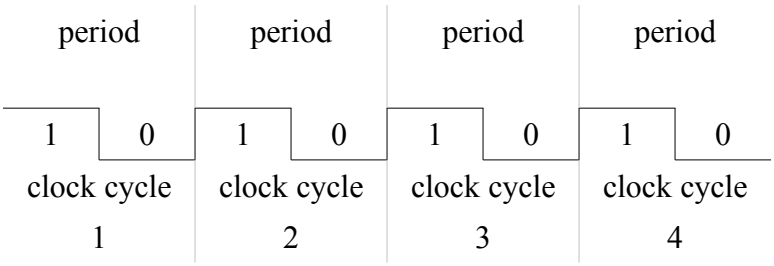
Definition: *A **clock** is a device that generates a signal that periodically cycles between a high state and a low state.*

Clocks ensure that the operations performed by a computer proceed in an orderly manner. They do so by enabling certain operations to occur only at specific points in time.

Clocks divide time into **cycles** that consist of two phases: a high phase and a low phase.

Definition: *A **clock cycle** is defined as the interval of time beginning when the clock goes to a high state, lasting through the return to a low state, and ending with the start of the transition back to the high state again.*

The following figure illustrates four complete cycles of a clock:



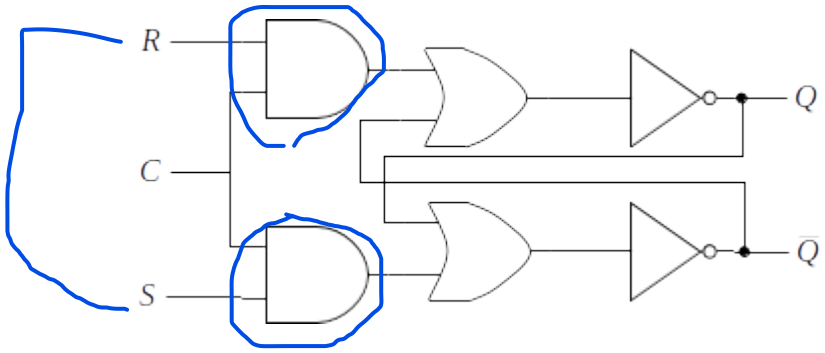
Each clock cycle lasts for only a brief instant of time. The CPU of a modern PC, for example, runs at billions of clock cycles per second (or gigahertz). The clock speeds of other components, such as the system bus, are usually somewhat slower but still in the range of hundreds of millions of clock cycles per second (or megahertz). The various operations that a computer can perform require one or more clock cycles to complete. The exact number of cycles depends upon the complexity of the particular operation.

As mentioned earlier, flip-flops can be used to implement the most basic unit of storage, the bit. Memory devices based on R-S flip-flops perform read operations by retrieving the contents of the Q outputs of a number of selected bits. Similarly, the write operation stores bit patterns into memory by placing 1s on either the S (Set) or R (Reset) inputs of various bits.

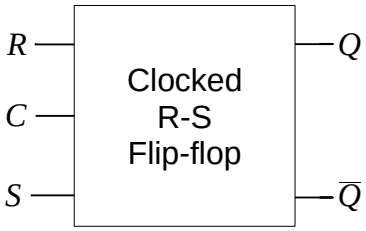
The clock ensures that these operations happen in an orderly manner. During one phase of the clock cycle (e.g., low) the contents of memory can be examined but not modified. During the opposite phase of the cycle (e.g., high) the contents of memory can be updated. This sort of timing is critical for the

reliable operation of a computer because it allows time after a *write* operation for the flip-flops to settle into their stable configurations before *read* operations can be attempted.

The following figure presents the circuit diagram of a clocked R-S flip-flop. In addition to the R and S inputs, these circuits also receive the clock signal. In the clocked R-S flip-flop, the Q output will be unaffected by any change in R or S as long as the clock (C) is 0. That is, during the *read* phase of the clock cycle, the contents of memory cannot be changed. When the clock input goes to 1, designating the *write* phase of the clock cycle, the Q output will change depending upon the values of R and S.



Lastly, the following is a "black box" representation of the clocked R-S flip-flop:



Building a simple computer

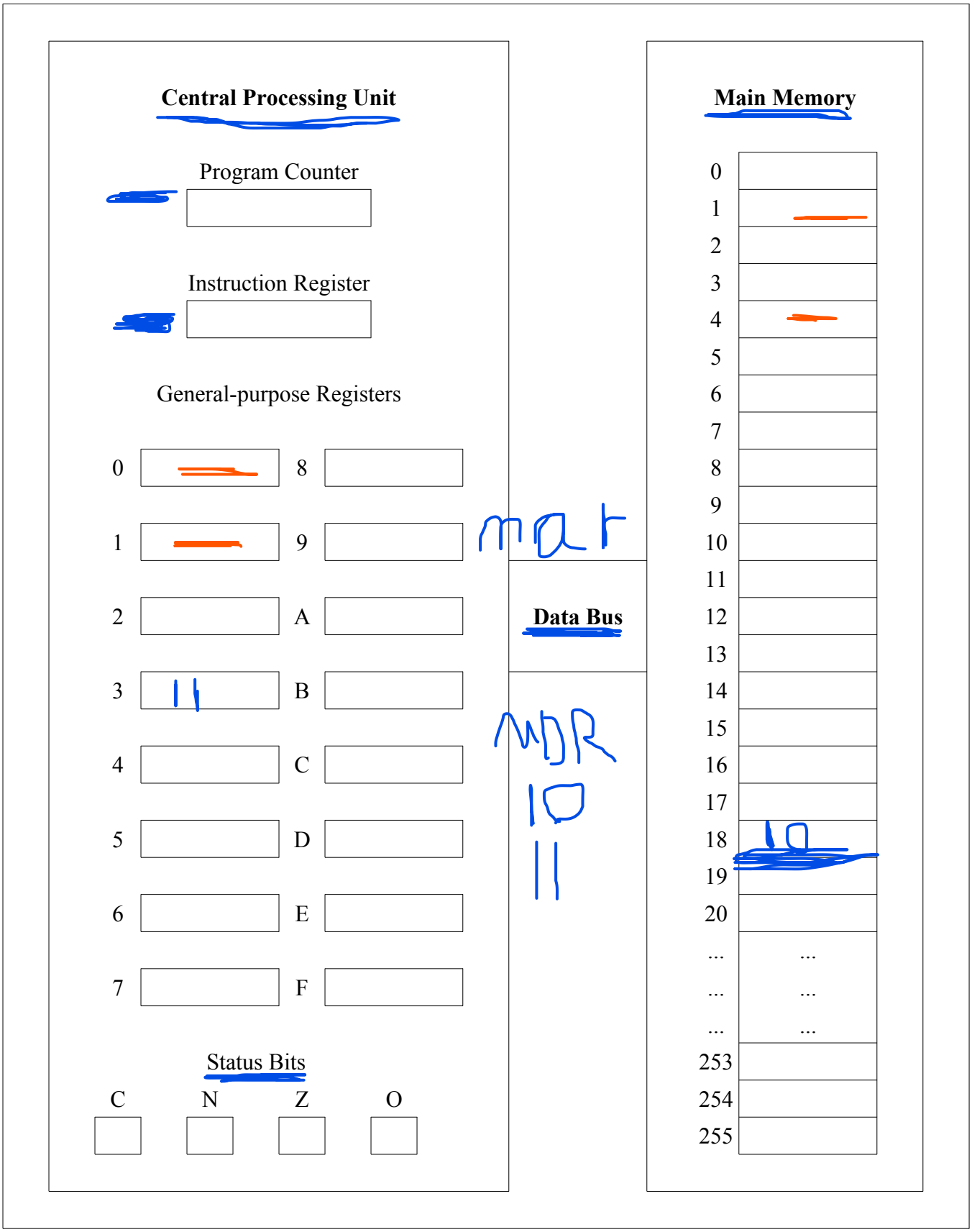
At this point, we have all of the building blocks required to design a simple microcomputer. Primarily, we will focus on two of the higher-level components, the central processing unit (CPU) and main memory, and how they can be constructed from low-level circuits. In order to have a context in which to frame this discussion, we need to briefly discuss register-based machines.

Definition: *The general organization of a machine is often referred to as its **architecture**.*

Over the years, computer scientists and engineers have explored a wide variety of computer architectures. The register-based machine is the most popular architecture.

Definition: *A **register** is similar to memory in that it can store data; however, it is much more quickly accessible because it is located within the CPU.*

Although we will use a "toy" computer, designed solely for the purpose of instruction, it embodies most of the major features of register-based machines. The sample (virtual) machine that we will use is shown in the figure below. It consists of two major parts: main memory and a CPU. These two components are connected together via the data bus, which allows information to be copied between the CPU and main memory.


Central Processing Unit
Program Counter
Instruction Register
General-purpose Registers
0
1
2
3
4
5
6
7
8
9
A
B
C
D
E
F
Status Bits
C
N
Z
O
Data Bus
Main Memory
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
...
...
...
...
...
...
253
254
255

The purpose of main memory is to store computer programs and the data on which they act. The main memory of the virtual machine can be visualized as a list (or array) of 256 storage locations, numbered 0 to 255. Each of these locations is individually addressable. In other words, the main memory unit can be given an address and told to retrieve a value from that location, or be given an address and told to write a value to that location.

The purpose of the CPU is to perform the arithmetic and logic functions specified by the instructions of a computer program. CPUs are complex devices composed of many functional units. One of the most prominent features of the CPUs of register-based machines is a large collection of general-purpose registers. The virtual machine contains sixteen registers, labeled 0 through 9 and A through F.

Registers can be directly manipulated by the arithmetic and logic units of the CPU. Main memory locations cannot. In order to perform any kind of arithmetic or logic operation on values stored in main memory, it is necessary to copy those values into CPU registers, manipulate the contents of the registers in a desired manner, and then copy the computed results back to main memory. For example, in order to add two numbers stored in main memory locations, the virtual machine must:

(1) Copy the values of both numbers from main memory into separate CPU registers;
(2) Add the contents of those registers, placing the result into yet another register; and
(3) Copy the result of the addition operation back into a main memory location.

In addition to the general-purpose registers, there are a number of special-purpose registers important to the operation of the CPU. These registers include the instruction register, the program counter, and the status bits.

Definition: *The **instruction register** holds a copy of the program instruction that the CPU is currently executing. The **program counter** contains the address of the next instruction to be executed. The **status bits** are used to hold information about the most recently performed computation (e.g., was a carry generated from some binary addition, was the result zero, was some result negative, did an overflow occur, and so on).*

More detail about these special-purpose registers is beyond the scope of this lesson.

Real world computers are similar to the virtual machine in that they contain main memory, a CPU, and a data bus. The virtual machine differs from real machines in that it has no I/O devices (such as keyboards, display screens, mice, and so on), nor any long-term storage devices (such as hard disk drives). The size of its memory is also very small, containing only 256 memory locations. Modern computers typically have hundreds of millions of memory locations. Technically, these memory locations are known as words.

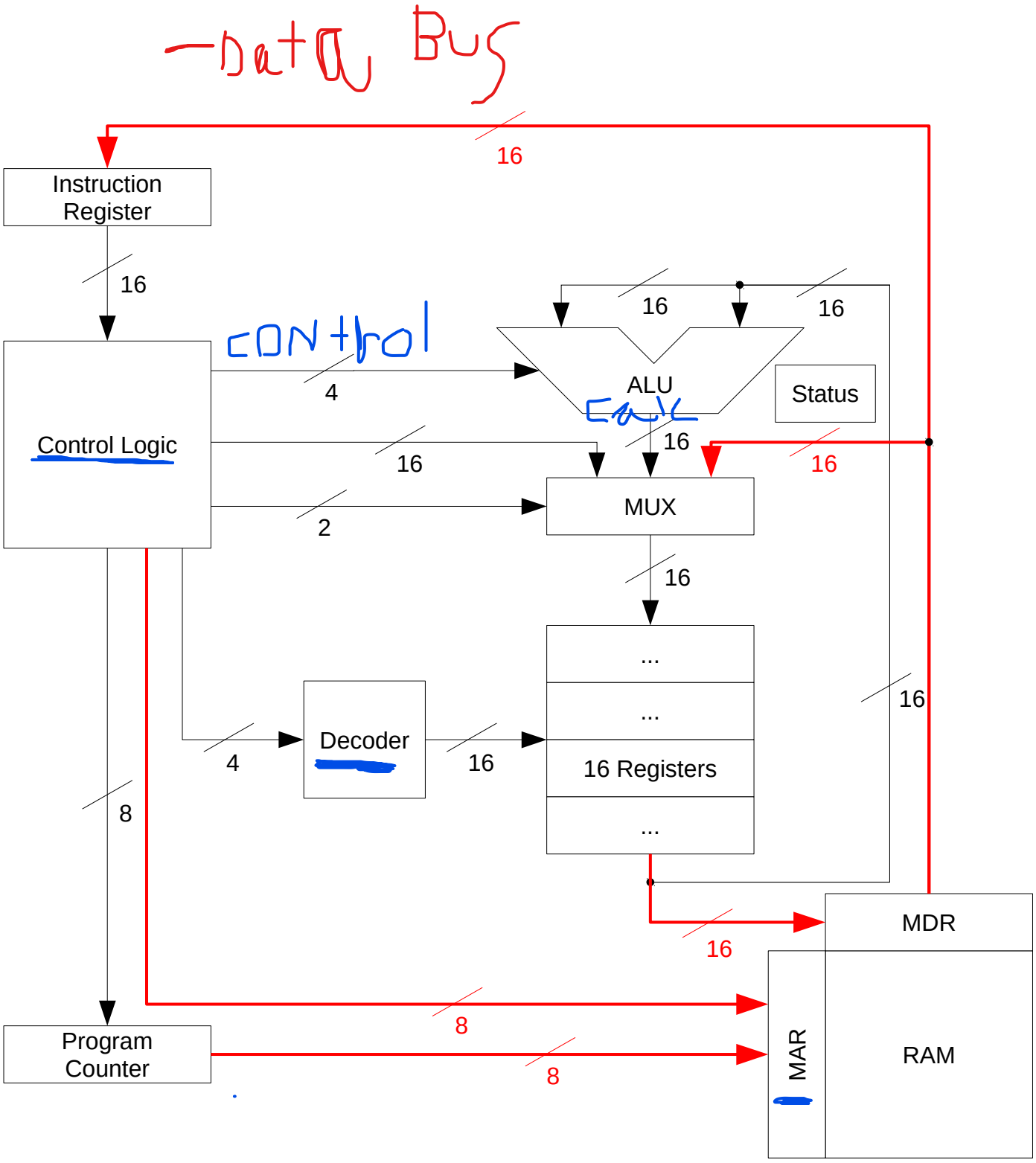
Definition: *A **word** is the base unit of data that is supported by a CPU.*

Today's computers usually have word sizes that are 32 or 64 bits. The registers are typically word-sized. There are other differences between the virtual machine and real-world computers, such as the size of the numbers it can manipulate and the limited number of commands in its machine language.

The main memory of the virtual machine is composed of 256 words, each 16 bits wide. The interface to main memory (RAM) consists of two registers: the memory address register (MAR), and the memory

data register (MDR). During both “read” and “write” operations, the MAR is used to hold the address of the memory location to be accessed. Since there are 256 words of memory in the virtual machine (numbered 0 to 255), the MAR is eight bits wide, enabling it to hold addresses in the range 00000000_2 to 11111111_2 (i.e., 0 to 255). During “write” operations, the 16-bit value to be written into memory will be placed in the MDR. During “read” operations the value of the memory location to be read will be output to the MDR. Thus, the memory data register can function both in an input role (for “write” operations) and in an output role (for “read” operations). The MAR, on the other hand, always functions in an input role, specifying the location to be accessed.

The following figure primarily details the virtual machine CPU. Special emphasis is given to illustrating the major control and data lines interconnecting the various components:


Data Bus
16
Instruction Register
16
Control Logic
control
4
16
2
16
16
16
ALU
Status
16
16
MUX
16
...
...
16 Registers
...
16
Decoder
4
16
8
MDR
16
8
8
Program Counter
MAR
RAM

We now address the final piece of the puzzle concerning machine language programs by illustrating how the machine actually goes about executing a program. At the machine level, all a computer ever does is perform the following five tasks (collectively known as the **instruction cycle**) over and over:

(1) Fetch the next instruction from memory. To do so, load into the instruction register the bit pattern found at the address held in the program counter.
(2) Increment the program counter by one so that it points to the next instruction in the current sequence.
(3) Decode the current instruction. This involves correctly identifying the various parts of an instruction based on its op-code. The **op-code** is the part of a machine language instruction that defines what operation to perform (e.g., addition). The other parts of a machine language instruction are called **operands** on which the op-code is applied.
(4) Execute the instruction. Operand values are routed to the appropriate arithmetic and logic hardware. Results are computed and routed to their appropriate destinations. Appropriate destinations include the general-purpose registers, special purpose registers such as the program counter, and main memory.
(5) Return to Step 1.

That's all a computer ever does! There is no magic; no smoke and mirrors. Just a simple, but very fast machine running through the instruction cycle over and over, tens or hundreds of millions of times each second.

Virtual machine overview

The control logic, or control unit, is the component of the virtual machine that is responsible for implementing the instruction cycle. It does so by generating the signals necessary to direct the other components of the machine to carry out their tasks in an orderly manner. For this reason, the control unit is sometimes referred to as the "traffic cop" of the CPU. Input to the control unit is from the instruction register, since it must have access to the bit pattern that represents the instruction for it to do its job. Because of the central role played by the control unit, its outputs are connected by various data and signal lines to most of the other components of the machine.

The ALU is responsible for the math and logic operations performed by the machine. If the control unit is the "traffic cop", then the ALU is the "calculator", responsible for performing addition, subtraction, and the various logic operations. The ALU of the virtual machine receives two 16-bit input values from the general-purpose registers and one 4-bit control code from the control unit. The 16-bit values represent the operands. The control code, derived from the op-code of the current instruction, specifies which arithmetic or logic operation is to be performed on those operands. Output from the ALU is directed to a multiplexer that is responsible for routing data to the general-purpose registers. This makes sense, because the results of arithmetic and logic operations must end up in one of the registers.

Taking a closer look at the multiplexer, we see that it has three 16-bit data inputs, one from the control unit, one from the ALU, and one from the MDR. There is also a two-bit selector signal sent from the control unit to the multiplexer in order to tell it which input data values should be passed on.

The reason for the three data inputs into the multiplexer has to do with the sources that can generate register values. Arithmetic and logic instructions (like addition), require that the registers be able to receive input from the ALU in order to place the result of the operation into the destination register. Other instructions require that registers be able to receive data from main memory or from "immediate"

values, which are part of the instruction (and are therefore located in the instruction register). The two-bit selector signal, under the direction of the control unit, specifies which of these three sources should be routed to the registers.

Building this multiplexer is a straightforward exercise: we simply arrange sixteen of the four-input multiplexers shown earlier in parallel. The reason for using four-input multiplexers, even though we only have three input sources, is that multiplexers only come in powers of two: two input, four input, eight input, etc. Therefore, one of the inputs on each of the sixteen standard four-input multiplexers will be unused. While this is may seem a tad wasteful, it won't cause any operational difficulties.

The reason for needing sixteen copies of the "standard" four-input multiplexers is that our data values are sixteen bits wide. Thus, each of the "standard" four-input multiplexers will pass only one bit of the 16-bit data value to an output line. By arranging sixteen of these "one-bit wide" multiplexers in parallel, we can build a multiplexer capable of routing all 16 bits of the selected data value simultaneously.

The final component of the CPU that we will look at is the four-input decoder, located between the control unit and the bank of general-purpose registers. The task of this decoder is to select one of the sixteen registers for access, either for receiving a value from the MUX or for sending a register value to main memory or the ALU. This decoder would be similar to the three-to-eight decoder shown earlier; however, it would be extended to handle four input lines and sixteen output lines.

In addition to the memory and CPU, the virtual machine also contains a data bus. This bus was illustrated earlier simply as a path connecting the CPU and memory. In the virtual machine, the representation of the data bus is much more detailed. It is presented as a number of separate data and address lines (highlighted in red in the figure above) that connect components of the CPU to RAM. In the figure above, there are two sets of 8-bit address lines, both leading into the MAR. There are also two sets of 16-bit data lines connected to the MDR, one set for input and one set for output.

The memory location to be retrieved (and therefore the address to be loaded into the MAR) can originate from two separate CPU components: the control unit or the program counter. The address comes from the program counter when the machine is fetching the next instruction from memory. The address comes from the control unit when the machine is fetching an operand, such as a variable, from memory.

The input data lines leading to the MDR originate from the block of sixteen general-purpose registers. This makes sense, because in order to store a value into memory, the value must first be in a register.

Looking at the output data lines originating from the MDR, we see that a 16-bit memory value can be transferred to either the instruction register or the multiplexer that routes values to the general-purpose registers. If the machine is performing an instruction fetch, the bit pattern being retrieved from memory represents an instruction and therefore must be sent to the instruction register. If the machine is in the process of fetching an operand, that operand should end up in one of the sixteen general-purpose registers and is thus sent to the multiplexer.

Even though the virtual machine is a very simple microcomputer by today's standards, it is still much too complex for us to go through a full design of its components. The development of its components, however, is constructed from the combinational and sequential circuits discussed throughout this

curriculum. We have arrived at the bare machine that lies underneath the many layers of software. There is no magic; no smoke and mirrors.