8: Digital Logic

We have so far built up what a microprocessor does using high-level hardware components. In this chapter we will describe how exactly these high-level components are built going all the down to the physical manufacturing of these entities and end with two of the most well known “laws” of computer systems.

8.1: Gates

First we will look at a level of abstract called the logic gate and show how any circuit-block, register, or memory can be constructed using logic gates alone. Recall that we already showed in Chapter 7 an entire computer can be constructed by combining and connecting together circuit blocks, registers, and memories. In this chapter we also get into the details of how the state exactly the state machine controller is implemented with logic gates. So let's get started.

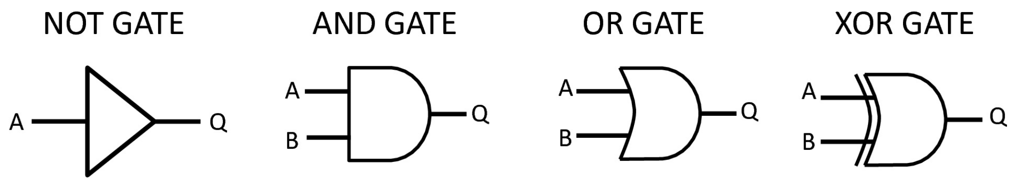
This section will be structured as follows. First we will describe five simple logic gates, we will (re)introduce the idea of a truth table and how it specifies the functionality of what is desired. Next we will show look at arbitrary truth-tables and show how one can construct a circuit-block using just the aforementioned logic gates to implement any functionality – this is the magic we need to implement our state machine controller. Third, we will look at how the logic gates can be combined to implement a register. Finally, we will use logic gates to implement memories. Voila – all components implemented with logic gates.

Remember that the rule from the previous chapter about why and how a computer works even though it has various components: “Every module implements its interface and thus becomes independent of anything else happening in computer!” This is the idea of abstraction stated another way. In this section we are taking this to another level lower – implement every module using only logic gates. In the next section, we will show how any gate can be implemented using transistors connected in a various ways, and physically show how a transistor can be built using silicon crystals!

8.1.1: Simple Logic Gates

The first and simplest logic gate we can build is called the NOT gate. The NOT gate is defined as a hardware module whose output is 0 when input is 1 and output is 1 when input is 0. The standard notation to specify the functionality of gates is the truth table. The text symbol for a NOT gate is ~. Sometimes a prefix of ! in front of a Boolean variable is also used. In this course we will use ~x to denote NOT(x). The Figure below (left-most column) shows the symbol for the NOT gate, its corresponding truth-table, and the notation.

The Figure also shows three other gates that are commonly used: AND gates, OR gates, and XOR gates. The names for the first two are reasonably intuitive .The AND gate sets the output to 1, when both input A and input B are 1. The OR gate sets the output to 1, input A or input B are 1. The XOR gate (stands for exclusive OR), sets the output to 1, when exclusively one of input A or input B is 1, but not both.



A	B	Q
0	0	1
0	1	0

A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

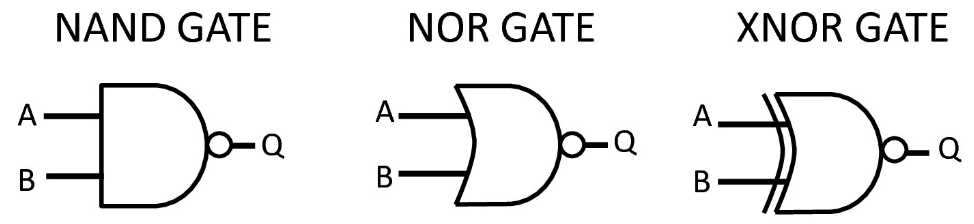
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

Text notation: ~ Text notation: · Text notation: + Text notation: ⊕

Figure 8.1.1.1: The symbols, truth tables, and text notation for the four basic logic gates.

Another set of logic gates that are commonly used are NAND, NOR, and XNOR--these are simply the AND, OR, and XOR gates with their outputs inverted. The Figure below shows their symbols, truth-table, and also shows how one could construct them using the NOT gate and the 3 gates we have already looked at. It is important that you get familiar with these symbols.



A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

A	B	Q
0	0	1
0	1	0
1	0	0
1	1	1

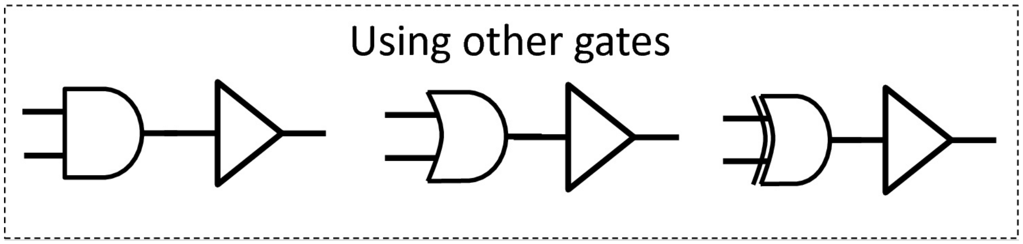


Figure 8.1.1.2: The symbols and truth tables for three further logic gates, as well as how to construct them using the basic gates.

We can also define 3-input gates, 4-input gates, and more generally N-input gates using the intuitive definitions for AND or OR gates. (There is also n-input XOR gate, but its definition is not very intuitive, so we will restrict ourselves to only 2-input XOR gates.)

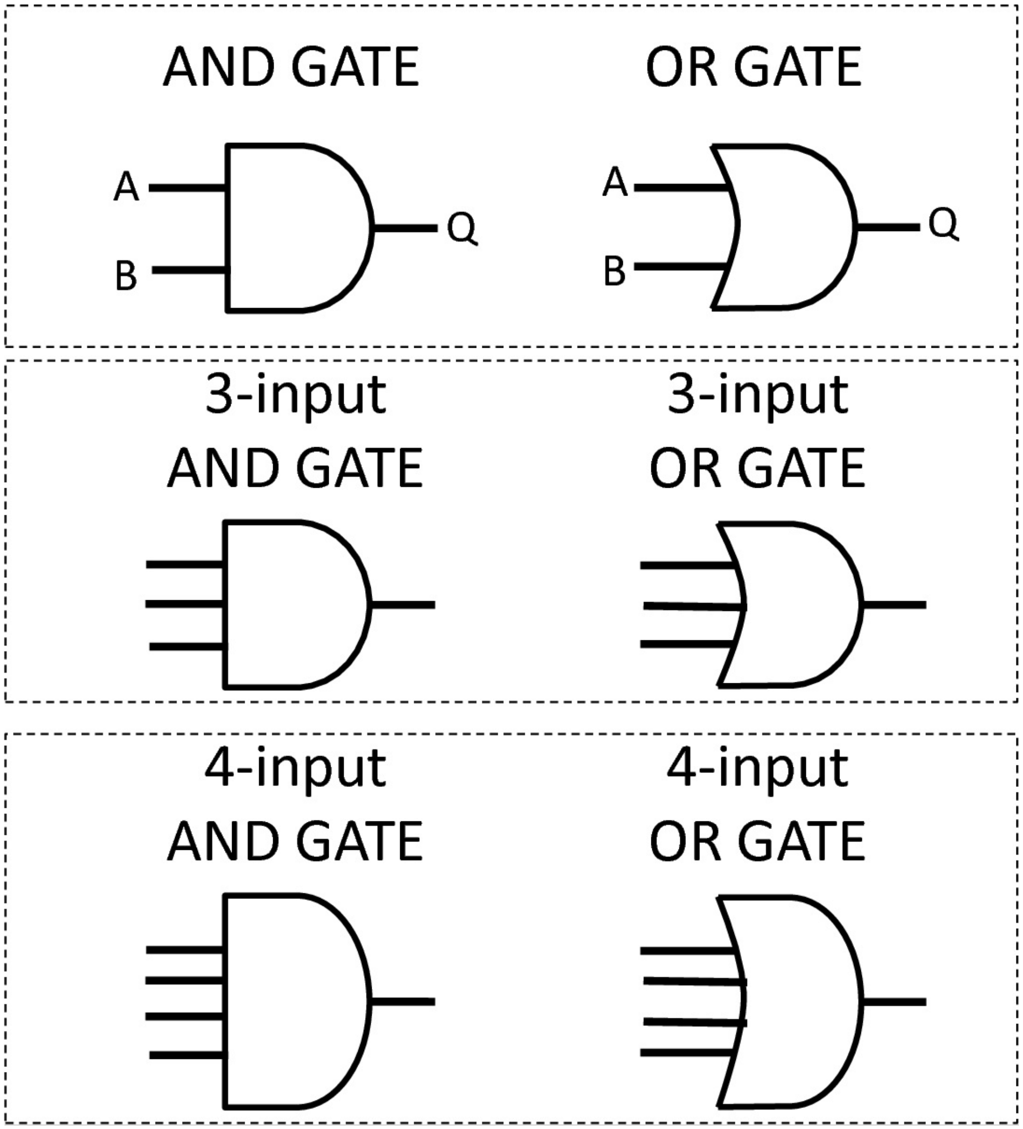


Figure 8.1.1.3: AND and OR gates with various numbers of inputs.

In general, the N-input AND gate will output 1 if all its inputs are 1, and 0 otherwise, whereas the N-input OR gate will output 1 if at least one of its inputs is 1, and 0 otherwise.

8.1.2: Combining Logic Gates

As we have already mentioned in the previous chapter, circuit blocks can be combined by connecting the output of one gate to another. We can do just that with logic gates (they are simply a very primitive type of circuit block). Let us look at a simple example of connecting two AND gates to an OR gate as shown below:

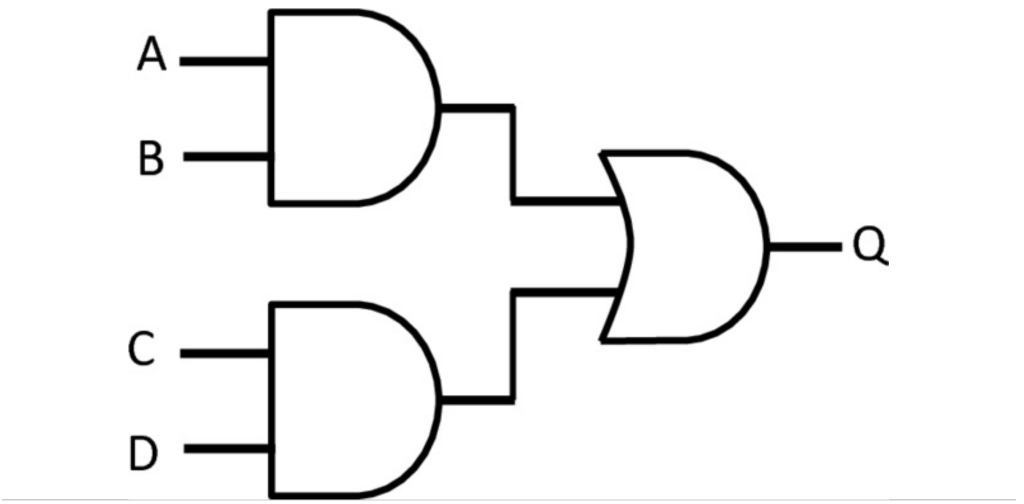
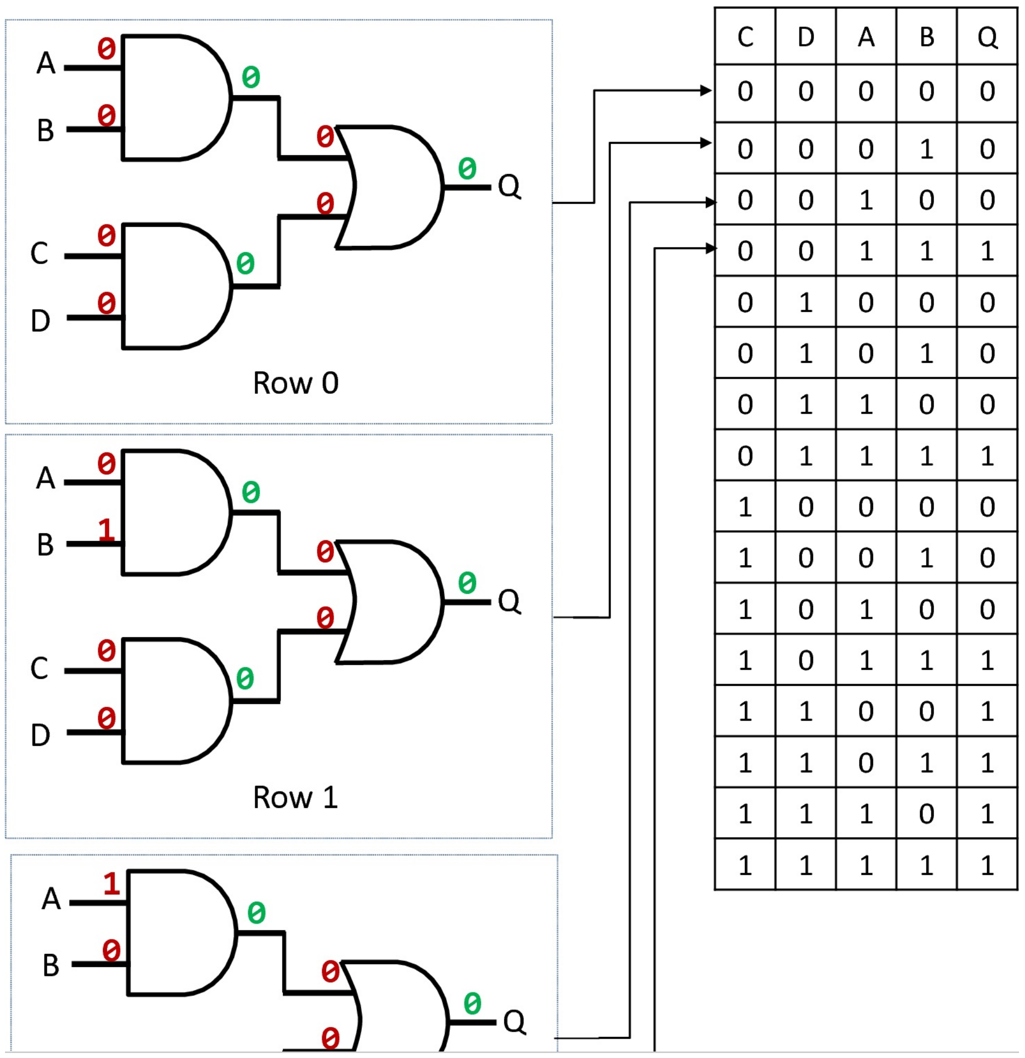


Figure 8.1.2.1: Connecting the outputs of a pair of AND gates to the inputs of an OR gate.

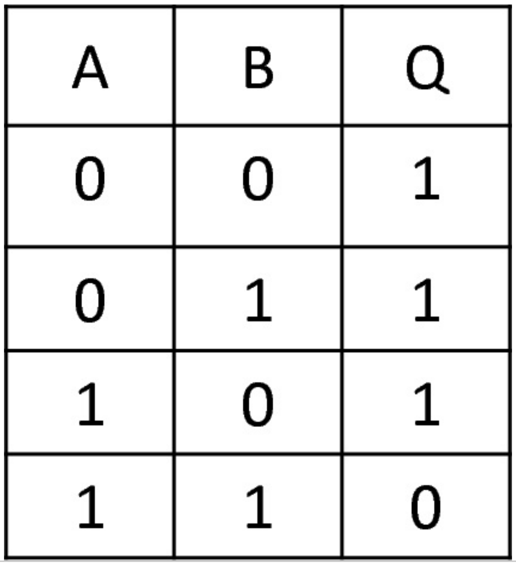
We need to understand or specify what this circuit block computes. We can do that by completing a truth-table. Notice that this circuit block has 4 inputs – and each of the inputs is a Boolean variable. Hence we have a total of 16 possible combinations of inputs. Hence our truth table will have 16 lines as shown in the Figure below (right). Let's populate this one row at a time. This can be done by looking up the truth-table of each gate, starting at the inputs, and tracing the values at the output of each gate and onward into the next gate. Let’s do just that. The Figure shows an example for the first 4 rows. We have denoted the values that are outputs in green. In terms of drawing style, notice that we have written the value on some wires twice--the wires which connect an output to an input of another gate--and yes, the value that travels on a wire cannot change.



C	D	A	B	Q
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Figure 8.1.2.2: The above circuit in various situations, and the truth table describing all possible combinations of input (A, B, C, D) and corresponding output (called Q).

Essentially the art of building a computer is to combine gates in such ways to build useful things. A more fun and interesting exercise is designing the gate implementation, when given the problem specification in terms of truth tables. Let us look at a simple example of the truth table below. We have two inputs A and B and one output Q as described by this truth table.



A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

Figure 8.1.2.3: A random truth table with two inputs and one output.

Our job is to convert this into a circuit built only with AND, OR, XOR, and NOT gates. As a rule for this course, we will not really use the XOR gate for such conversions and restrict ourselves to AND, OR, and NOT. So how do we do this? We do this by following two rules:

1. First construct a set of separate circuits that implement the logic for each ROW whose output is 1
2. Combine all of these using an OR gate – since an OR gate outputs a 1 when any of its inputs is 1

Let’s do this with text notation to keep it concise.

Step 1:

- Let's consider the zeroth row, A = 0, B = 0, Q = 1. What this means is, we need to construct some circuit which takes A and B as input and produces an output of 1, when both A and B are 0. Intuitively this can be achieved by ~A and ~B.
- Let's consider the 1st row, A = 0, B = 1. This can be achieved by ~A and B.
- Considering the 2nd row, A = 1, B = 0. This can be achieved by A and ~B.
- Considering the 3rd row, A = 1, B = 1, Q = 0. Notice Q is 0 so we don’t need to do anything.

Step 2: Now, we combine the previously constructed circuits with one OR gate. So we have the final output Q for the entire truth table as: Q = (~A and ~B) OR (~A and B) OR (A and ~B). That’s it we are done! The schematic or gate-level circuit corresponding to this is below:

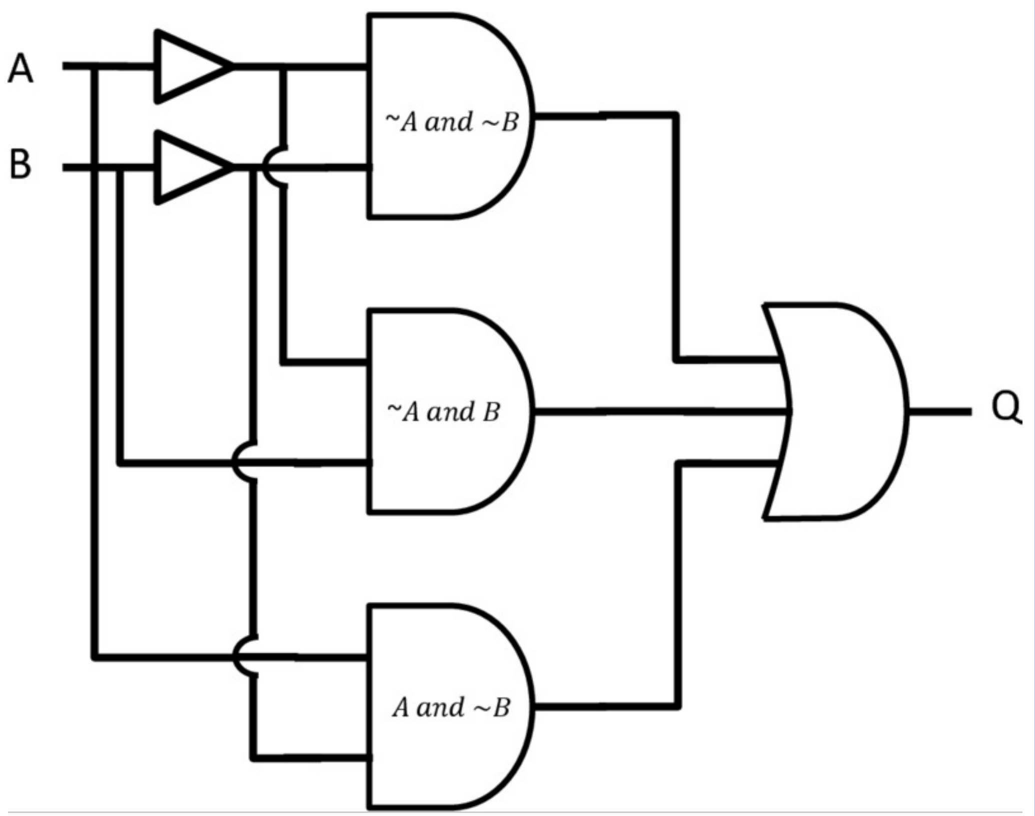
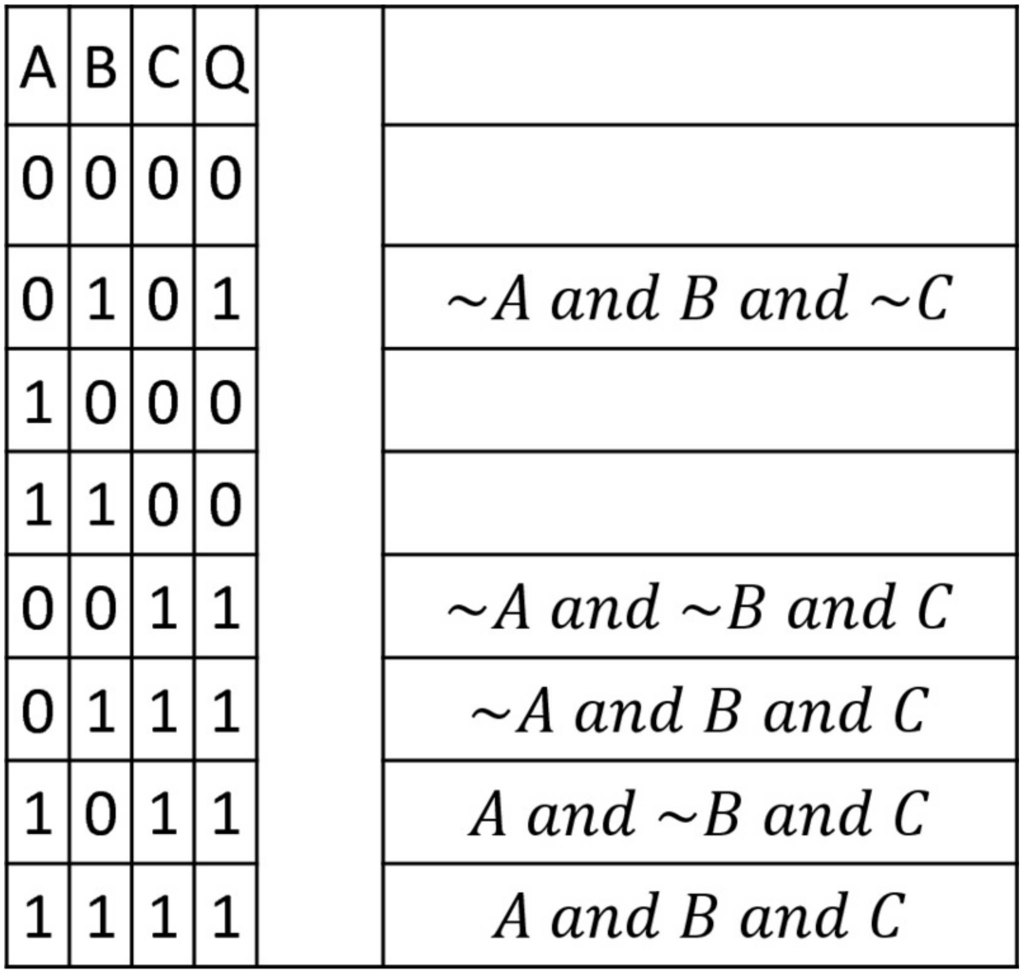


Figure 8.1.2.4: A circuit implementing the above truth table.

Diving down another level, what we intuitively did for step 1 for each row, can be converted into an algorithm. If a value is 0 negate that input variable, else leave as is. We have essentially specified the entire process of circuit construction into Boolean equations as an algorithm!

Let’s do an example with 3 input variables as shown in the table below. In the last column we can see the equations for the rows whose inputs are 1. The final circuit simply ORs all these together. In terms of a Boolean equation, we can write Q as: Q= (~A and B and ~C) OR (~A and ~B and C) OR (~A and B and C) OR (A and ~B and C) OR (A and B and C).



A	B	C	Q		
0	0	0	0		
0	1	0	1		$\sim A \text{ and } B \text{ and } \sim C$
1	0	0	0		
1	1	0	0		
0	0	1	1		$\sim A \text{ and } \sim B \text{ and } C$
0	1	1	1		$\sim A \text{ and } B \text{ and } C$
1	0	1	1		$A \text{ and } \sim B \text{ and } C$
1	1	1	1		$A \text{ and } B \text{ and } C$

Figure 8.1.2.5: A random truth table for a circuit with 3 inputs and 1 output, annotated with formulas describing the inputs that correspond to an output of 1.

Recall, state machine tables from Chapter 7? It was in the end a table with inputs and a set of ouputs. Essentially once we constructed the table, we can use exactly this mechanism to create an implementation in terms of logic gates.

Aside: There are many complex techniques called **Karnaugh maps** and **logic minimization** that can help in implementing circuits for truth tables using fewer gates than our simple method here. We will defer that for a different course.

The takeaway from this subsection is simple: Given a truth-table you can follow the method we have developed to build an implementation of it using logic gates. We will now apply this to build all the circuit blocks we are aware of using logic gates.

Remember, we introduced NAND and NOR gates. These are useful in a practical way because they can be oddly be built out of fewer transistors than AND or OR gates. De Morgan's Law is a law about Boolean logic that allows conversion of circuits from AND and OR gates to NAND and NOR gates. It is a simple law that states: “The negation of the sum of two variables is equal to the product of the compliment of each variable.” In terms of equations:

```
~(A AND B) = ~A OR ~B
~(A OR B) = ~A AND ~B
```

Or, put another way:

```
A NAND B = ~A OR ~B
A NOR B = ~A AND ~B
```

8.1.3: Datapath Modules

The non-storage circuit blocks we have used so far are multiplexers and adder. We will build those using logic gates now.

A **multiplexer** can be considered as a 3-input circuit block, A, B, and S, whose output is defined by S. The truth table below specifies its functionality. The corresponding equations for each non-zero term and the gate-level implementation are shown in the figure below.

A	B	S	Q		
0	0	0	0		
0	1	0	0		
1	0	0	1		*A and ~B and ~S*
1	1	0	1		*A and B and ~S*
0	0	1	0		
0	1	1	1		*~A and B and S*
1	0	1	0		
1	1	1	1		*A and B and S*

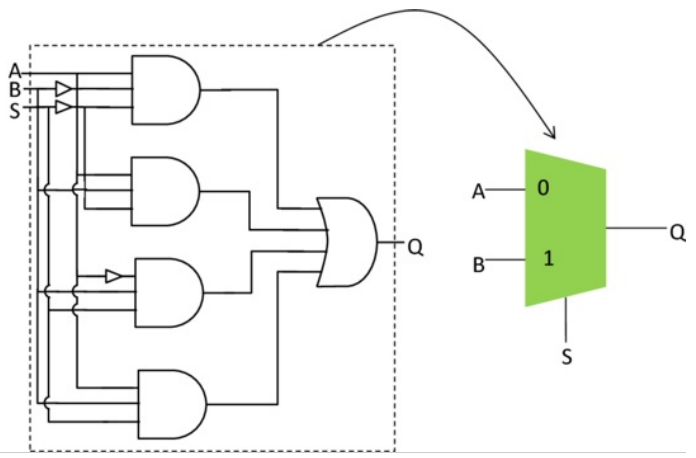


Figure 8.1.3.1:

An **adder** is a simply a circuit that has 3 inputs, A, B, and Cin (carry-in) and produces as output a single-bit sum (S) and a single bit carry-out (Cout). This is also referred to as a Full-Adder. The truth table and implementation for it can be constructed using the techniques we know. Since an adder is used often in various circuits, it is common to develop an optimized implementation with as few gates as possible. Alongside the truth-table we have one implementation that uses the XOR gates and implements an optimized adder.

A	B	Cin	S	Cout
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

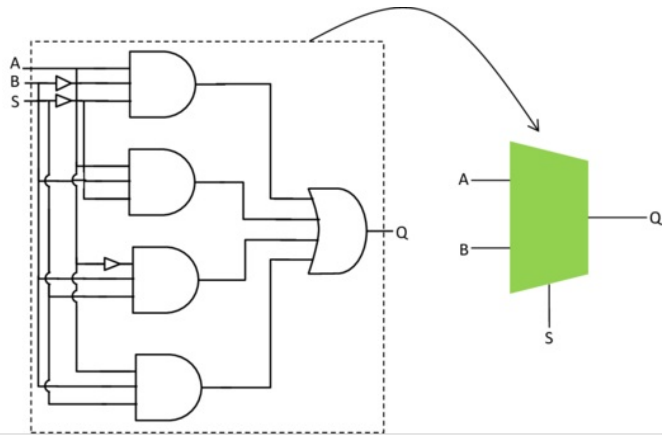


Figure 8.1.3.2:

Now that we have a circuit that can add two single bit inputs and produce a single-bit output, we can build an adder that will add 8-bit values that chains 8 such full-adders as shown below which produces an 8-bit out and a final carry.

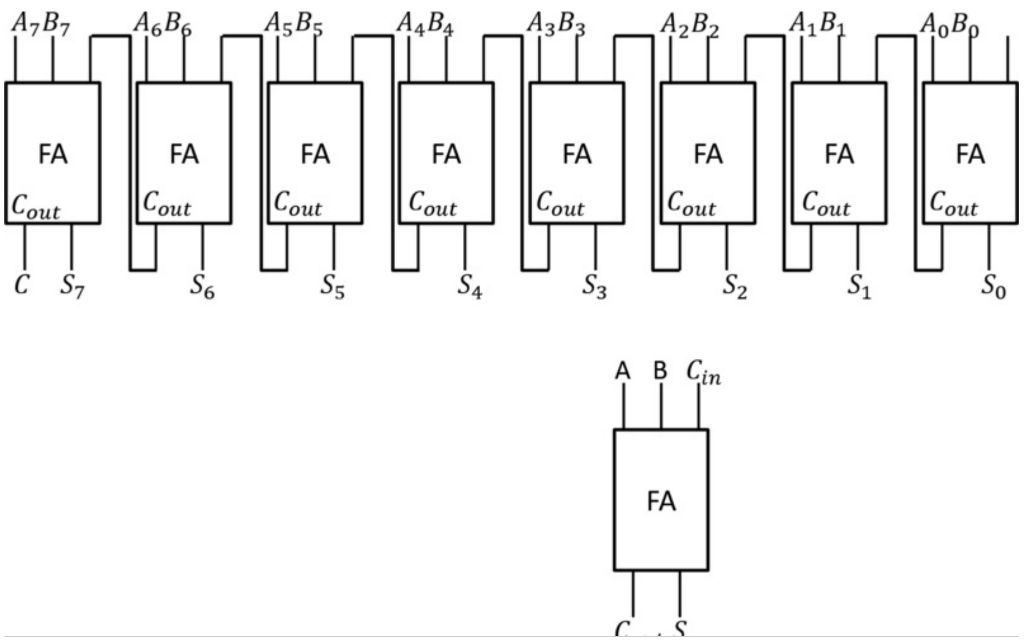


Figure 8.1.3.3:

With these elements we can build another module that can perform addition or subtraction based on an ALUop input as shown in the Figure below. We need to augment this to handle the computation of C flags and Z flags to develop the complete ALU we used in the previous chapter. We leave out the details of the complete ALU.

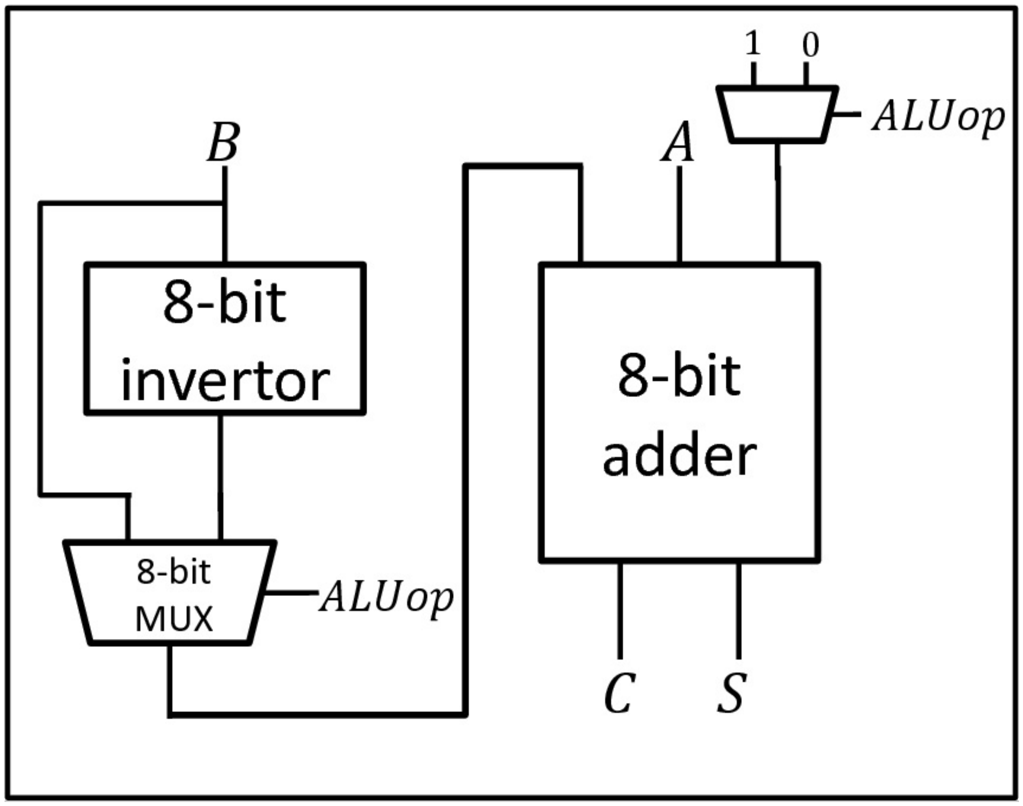


Figure 8.1.3.4:

Two other circuits that are commonly used are what are called **decoders** and **encoders**. An n-bit decoder takes as input, an n-bit input and produces 2^n outputs of which exactly one output is 1, based on the value of n. Such an output format is also referred to as one-hot encoding. An encoder does the converse. It takes as input 2^n one-hot values and converts it into an n-bit output.

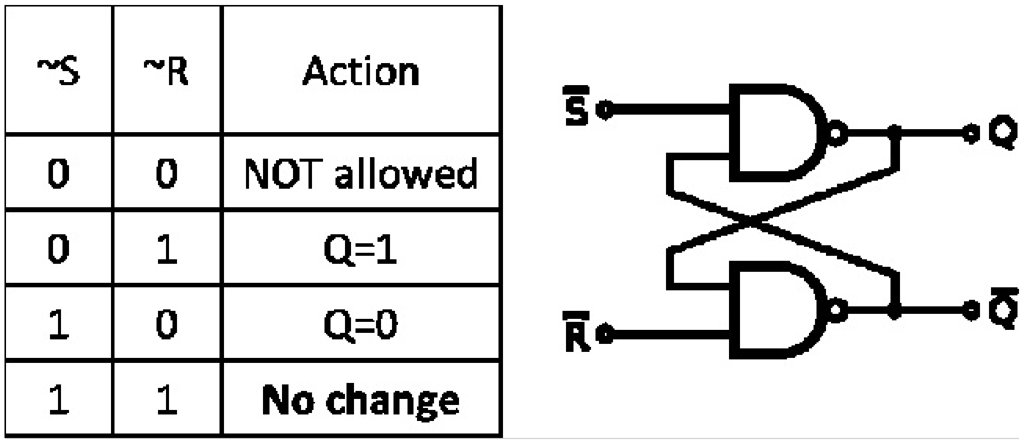
[TODO: Decoder and encoder truth-table and diagram here]

8.1.4: Registers and Sequential Elements

The circuits we have constructed thus far are referred to as combinational circuits. They produce an output that is independent of any kind of clock signal and produces a new output simply based on the input. Creating a register which only changes its value based on a clock signal is a little tricky. These types of circuits are called sequential circuits and in this sub-section we will build two types of sequential elements – something that can store a value and hold it unmodified and a D-flip-flop register which will capture the value on the rising clock edge.

First, a brief note the clock. In modern processors, the clock signal is a value that toggles between 0 and 1 at a clock period. It is used to build sequential elements and allows the construction of circuit that operates at a fixed frequency.

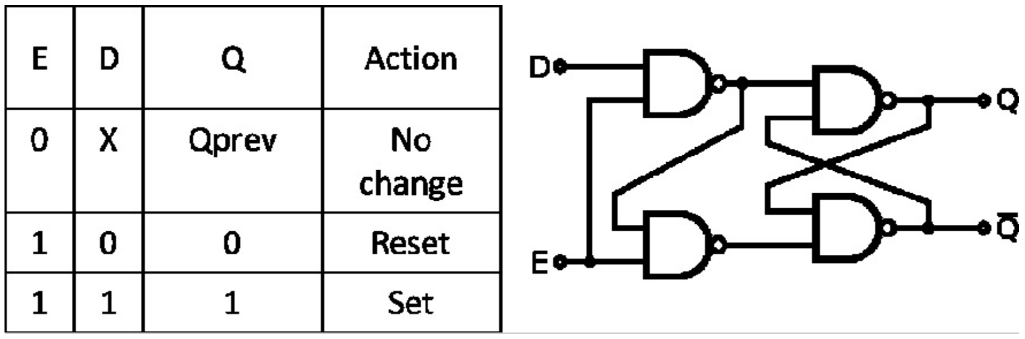
First, we will build an RS latch which is a type of circuit which can hold its value and can thus be used to build memories. The basic circuit an RS latch is shown below. Its operation and truth table are a little unusual and are described in the table alongside it. Due to historical design reasons, its inputs are ~S and ~R. The basic essence of this circuit is that, when both ~S and ~R are held at 1, the circuit retains its old value and serves as a memory cell. Setting ~R to 1 writes 1 into the cell. Setting ~S to 1 writes 0 into the cell.



~S	~R	Action
0	0	NOT allowed
0	1	Q=1
1	0	Q=0
1	1	**No change**

Figure 8.1.4.1:

This can be modified slightly by adding another set of gates controlling ~S and ~R to prevent the disallowed state from ever occurring. This design is called the gated-D latch and is shown below.



E	D	Q	Action
0	X	Qprev	No change
1	0	0	Reset
1	1	1	Set

Figure 8.1.4.2:

Even this doesn't quite give us the capability of being able to modify values only at the clock-edge. We accomplish by construction what is called a master-slave D-flip-flop circuit using two gated D-latches as shown below. This gated D latch is basically a single-bit register.

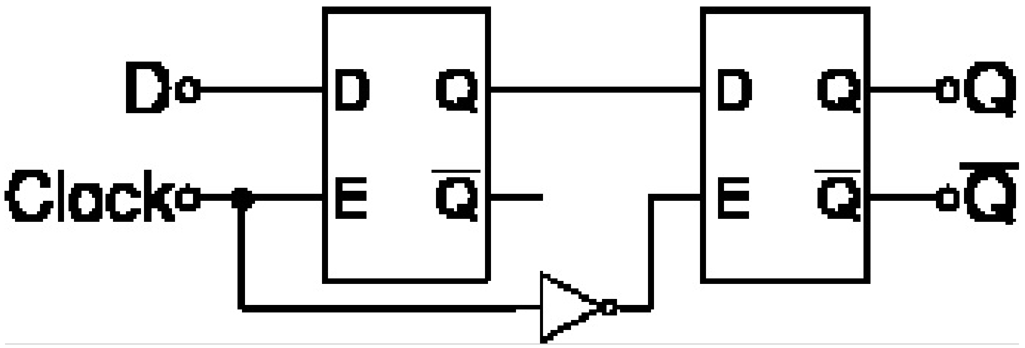


Figure 8.1.4.3:

We can enhance this slightly to get a single-bit register with a write-enable like we need for all our auxiliary registers. What we do is AND the “we” wire to the clock wire as shown below.

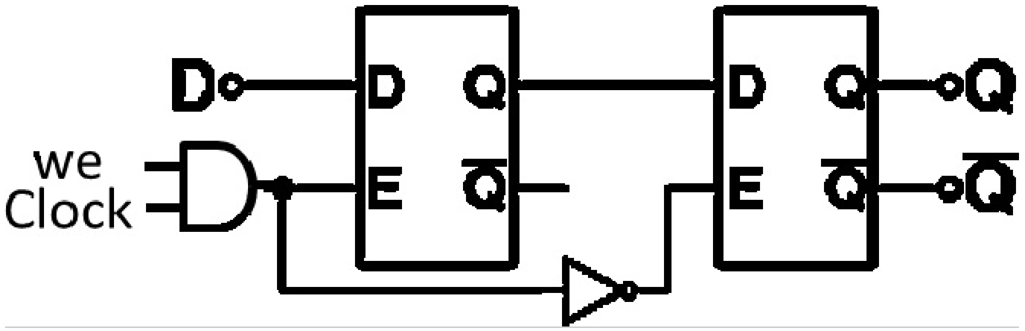


Figure 8.1.4.4:

8.1.5: Memories

The final module we will examine is how to build memories. Recall memories are constructed to have a certain width, a certain depth and are accessed with an addr point, a write-enable, and produce data on the data port. We can first construct a row of a memory by simply arranging multiple gated D-latches together. We can then arrange multiple rows to create the entire memory. We can connect all the rows to a multiplexer whose select-line is connected to the addr port to select the correct row. To write the memory, we connect the addr into a decoder to produce a one-hot signal which is 1 at the row that is being written to. This can be ANDed with the we input and accomplishes the task of writing to the memory.

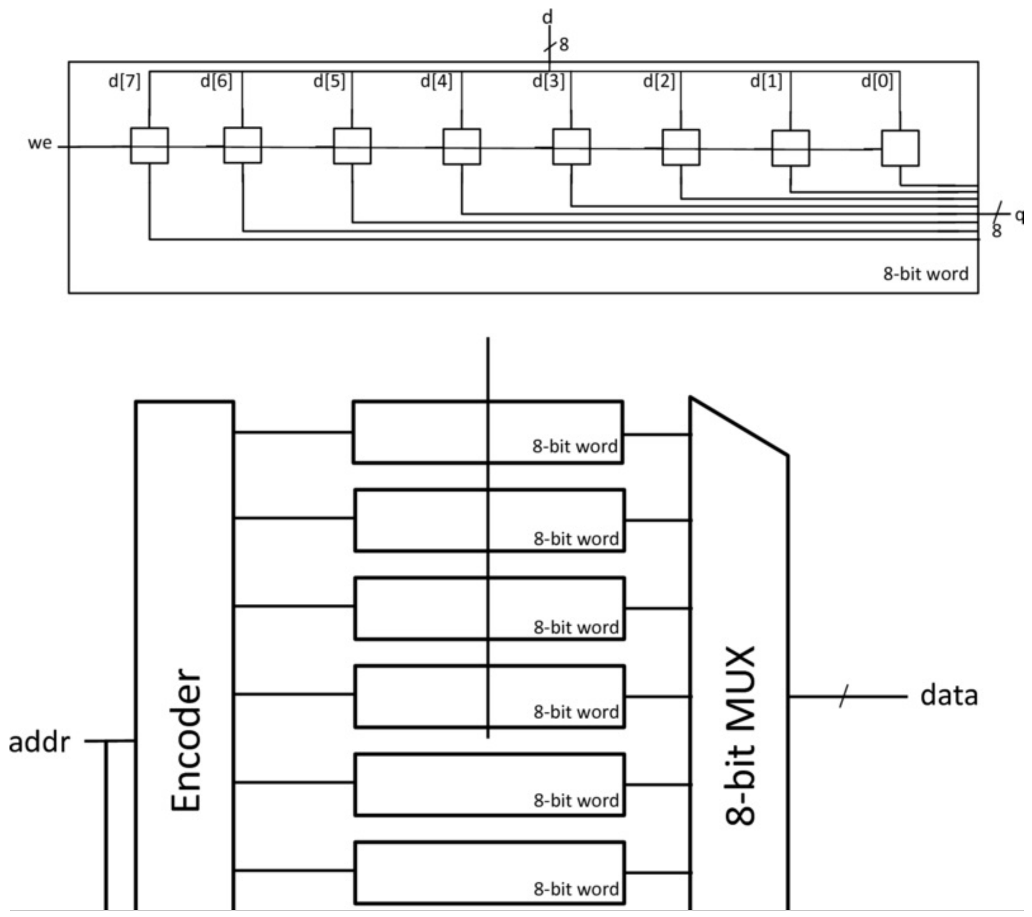


Figure 8.1.5.1:

What we have now is an 8-wide, 8-deep memory. We can combine four of these to create the 32-entry register file memory.

We can combine 8192 such blocks to create the full 65536 byte memory we need. Or build this more hierarchically. Combine four 32-entry memories to create a 128-entry memory. Then combine four 128-entry memories to create a 512-entry memory. Combine four 512-entry memories to create a 2048-entry memory. And so on.

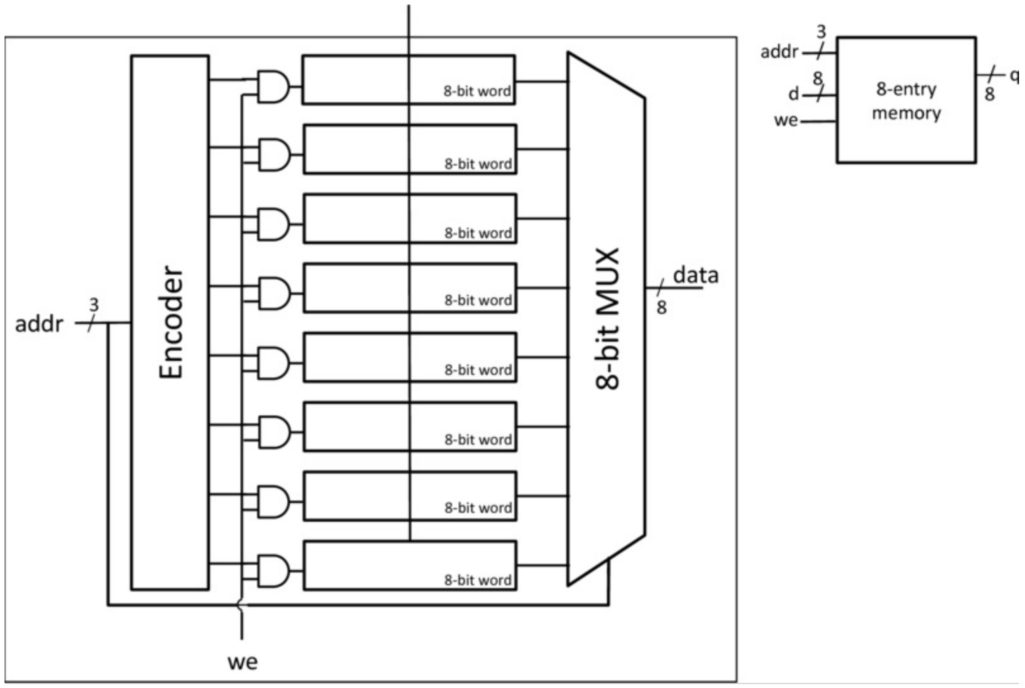


Figure 8.1.5.2:

8.2: Transistors

We will now get down to the lowest level of abstraction – the mighty transistor. A transistor is the fundamental building block of all computing devices and in simple terms a transistor is nothing but a switch. Historical computers were built out of switches made from mechanical devices, and later vacuum tubes before transistors became the standard – more about this evolution later. A transistor has three terminals called a source and drain. These are fancy terms that simply refer to two ends of the circuit that is going to be connected or left open. See diagram below. In a mechanical switch for an electric light, when the switch is one position (manually set by a human), the circuit is closed and current flows turning the light on.

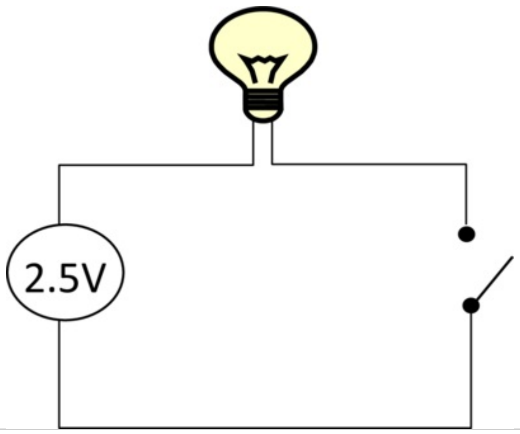


Figure 8.2.1:

For modern electronic devices, having such a moving part to turn the switch on or off would be undesirable – since mechanical things break. Instead the 3rd terminal called the gate is simply controlled by a voltage level. See diagram below. When we supply high-voltage the transistor is closed and current flows from source to drain! We call this type of transistor and n-type transistor (exactly why is quite complicated). The short answer is that this type of transistor creates an electrical connection between source and drain by allowing negatively-charged particles to move from the source side to the drain side. A p-type transistor is turned on when the gate is provided with low-voltage (or negative terminal of a battery) and it operates by allowing positively-charged particles to move from source to drain. Furthermore, for n-type transistors the source terminal is connected to low-voltage. And for p-type transistors the source-terminal is connected to high-voltage. When the switch is open, the output is undefined – electrically there is high impedance and we sometimes represent by saying the value is Z.

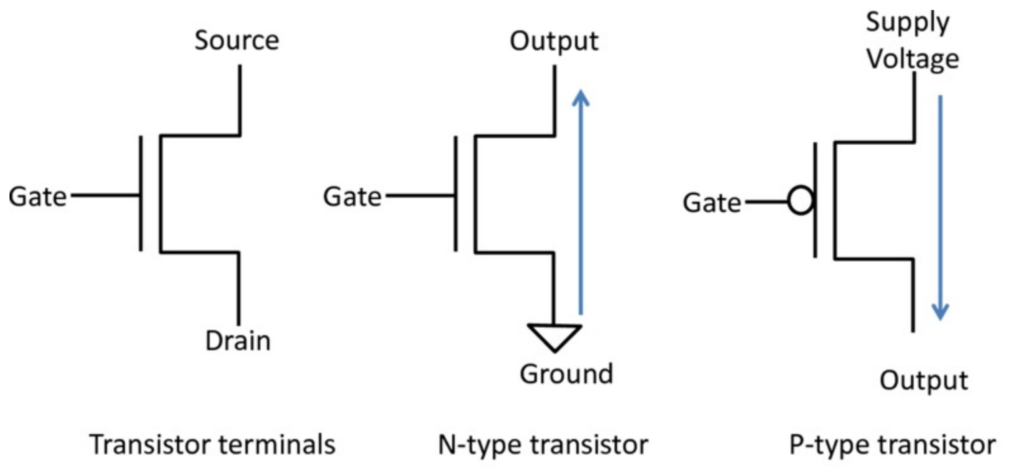
Some simple electrical rules of transistors we must adhere to:

1. N-type transistors are ON when the gate is 1.
2. N-type transistors must be connected in such a way that when they are ON, the output ONLY has a path to ground (Boolean zero).
3. P-type transistors are ON when the gate is 0.
4. P-type transistors must be connected in such a way that when they are ON, the output ONLY has a path to the voltage supply (Boolean 1).

What we have accomplished with a transistor is something simple, yet profound. Using purely electrical control we can open or close a connection. We can connect transistors to each other and create many fancy things. In fact, the microprocessor you have in your laptop is merely a collection of transistors – depending on exactly the laptop you have it may have up to 1 billion of them squeezed into an area no bigger than a quarter. Since we are moving down layers of abstraction, all we need to show is that all the logic gates can be constructed out of transistors – and with that we are done!

You might see the connection between the transistor and the concept of Boolean logic. Essentially high-voltage we will treat as logical 1 and low-voltage we will treat as logical 0. Voila--with that we have built what we need for a digital computer.

First, let us look at how to build a NOT gate using transistors as shown below. The middle and left-most diagrams show the operation and how we are able to accomplish the NOT operation using two transistors.



Gate	Behavior
1	Closed Output=0
0	Open Output=Z

Gate	Behavior
0	Closed Output=1
1	Open Output=Z

Figure 8.2.2: The implementation of a NOT gate using transistors.

Remember NAND and NOR gates and how we said they are easier to implement. We’ll look at just why now. The Figure below shows the implementation of NAND and NOR gates – and as you can see we need only 4 transistors for each. We can combine the NAND with NOT to get an AND gate, and similarly NOR with NOT to get an OR gate.

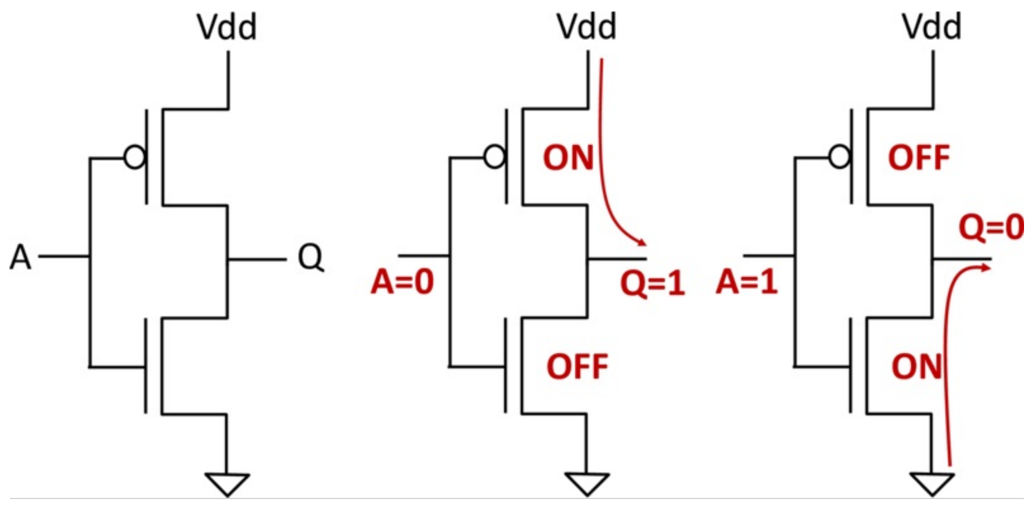


Figure 8.2.3: The implementation of a NAND and NOR gate using transistors.

Finally we can build an XOR gate by getting very creative with the operation of transistors as shown below. Note we are assuming that ~A and ~B are available, which can be accomplished with a NOT gate, which we already saw how to constuct.

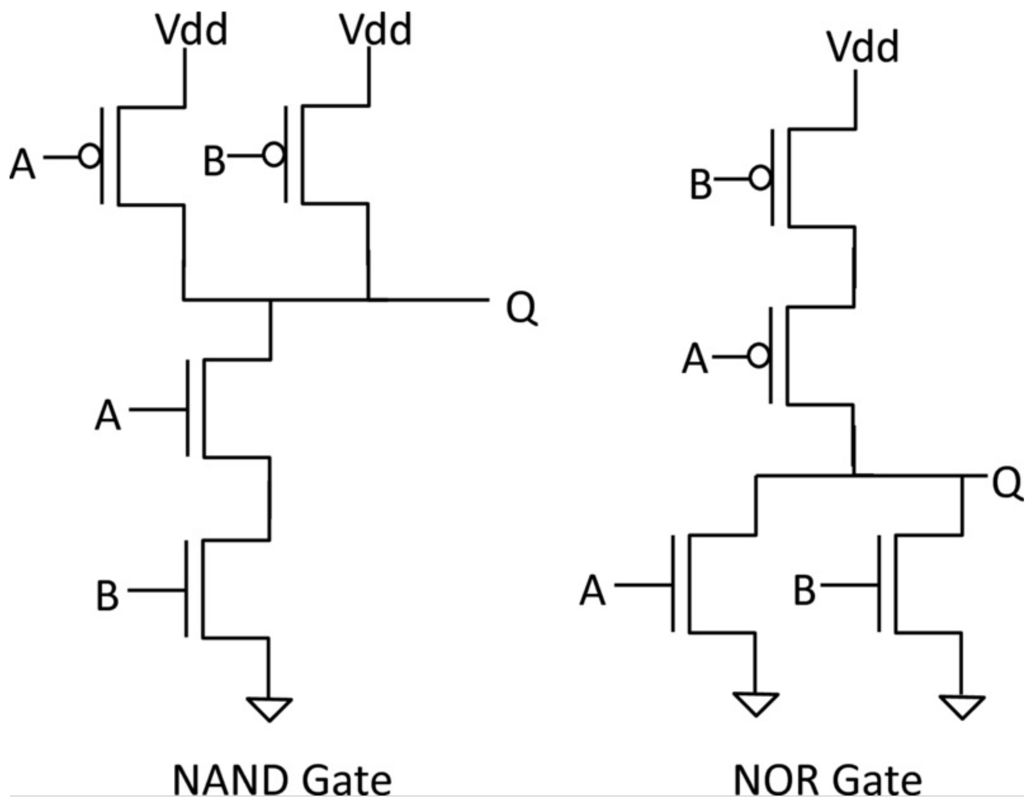


Figure 8.2.4: The implementation of an XOR gate using transistors.

Just like what we did with logic gates, given a transistor-level circuit we can compute what it does by tracing the operation of each transistor as being on or off.

8.3: Electrical Operation and Physical Manufacturing

We will now briefly touch on physical manufacturing by looking at how transistors are constructed. A single transistor (n-type) comprises of four regions all made largely out of silicon. It consists of a substrate (Silicon), the source and drain terminals (which are doped with electrons and form what are referred to as n+ regions), a polysilicon crystal which acts as a gate, and a capacitive material (gate oxide) which allows the gate to set up an electrical field for electrons to flow. The length of the transistor (L) is also referred to as the channel length – its value is what is referred to as technology node. When people say a 22nm transistor they are referring to the distance between the source and drain!

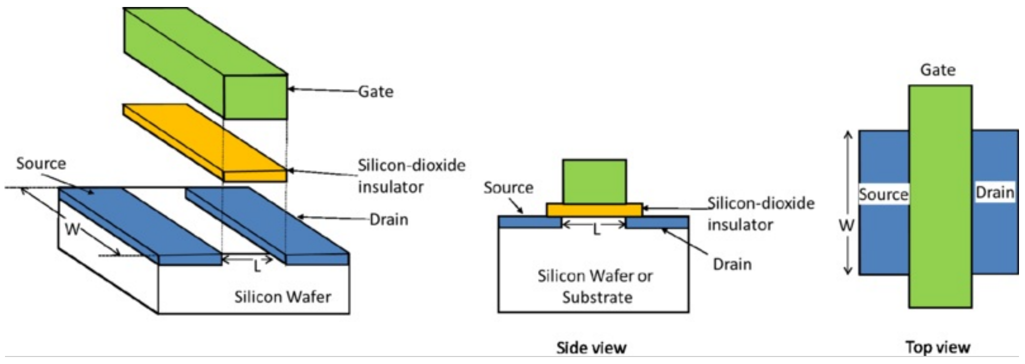

Gate
Silicon-dioxide insulator
Source
Drain
W
L
Silicon Wafer
Source
Silicon-dioxide insulator
L
Drain
Silicon Wafer or Substrate
Side view
Gate
Source
Drain
W
Top view

Figure 8.3.1:

8.3.1: Electrical Operation of a Transistor

We will consider a simplified schematic of a transistor to explain its electrical operation and how it provides the logical abstraction of 0 and 1. First let's revisit the Periodic Table and some Chemistry. Transistors are made of Silicon and Silicon has an interesting property – it is a poor conductor because all of its valence electrons are involved in chemical bonds. However it can be easily "doped" with impurities and made to behave like a conductor. When a silicon crystal region is doped with Arsenic (which has 5 valence electrons), an Arsenic atom will take the place of a Silicon atom in the lattice structure and leave a free electron – allowing conduction. Conversely when it is doped with a group III atom like Boron, a Boron atom will take the place of some Silicon atoms and leave a neighbor Silicon atom short by one electron – we call this a hole. And just like how electrons can help current flow, a hole can also help current flow. Doping a part of Silicon with Group V atoms creates n-type regions and doping a part of Silicon with Boron atoms creates p-type regions. Creating two n-type regions, with a gate-oxide and a polysilicon gates creates an nMOS transistor as shown below and described previously. Silicon-di-oxide is an insulator and hence will prevent current flowing from the gate down into the transistor – this is a very important property. We want our control signal (gate) to allow flow of current from source to drain – logically reflect the value of what is in source into drain, and not be polluted by the value of the gate.

An n-type transistor consists of a lightly doped p-type substrate onto which two heavily doped n-type regions are created. Let's look at how it operates. When a high-voltage is applied to the gate, it attracts electrons from the substrate which reach the Si and SiO2 boundary and stay there. Because SiO2 is an insulator they cannot enter the gate. This essentially creates a channel filled with electron carriers. The source and the drain have excess electrons which use the channel as a conducting path to enable current to flow from source to drain – thereby "closing" the switch and having current flow. When the voltage at the gate is 0, there are not electrons in the channel, thereby closing the switch. In simple terms, that is driving mechanism behind all digital logic.

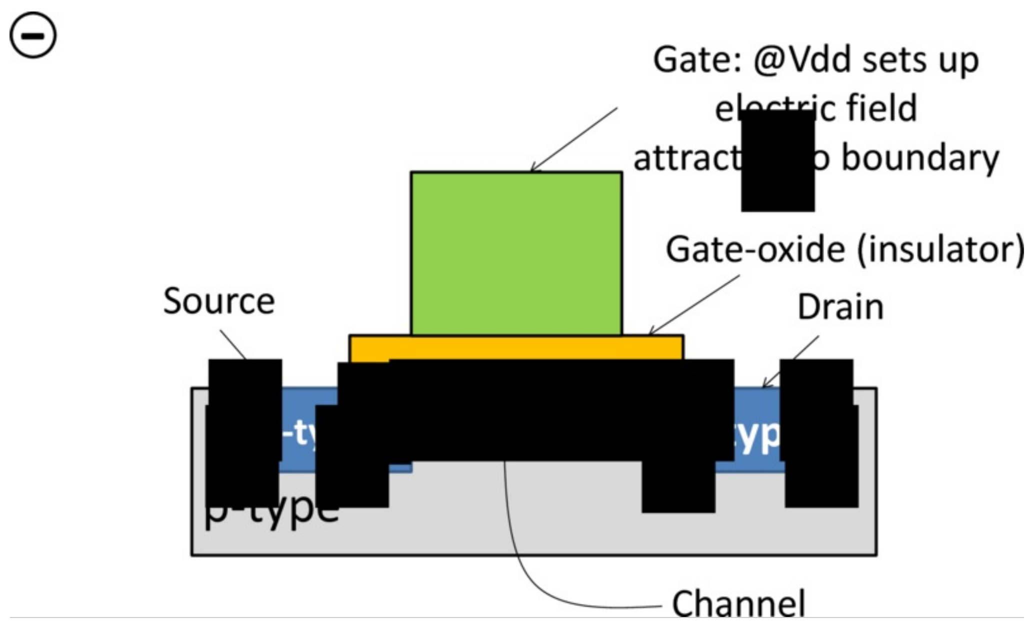


Figure 8.3.1.1:

A p-type transistors behavior is the converse and we will skip its operation details.

8.3.2: Physical Manufacturing of a Transtistor

The manufacture of transistor is a complicated process that involves many steps. The cross section layout of the transistor gives a hint on how the manufacturing is done. While complicated the manufacture process in essence reduces to the following. Take a Silicon crystal, create as many n-type or p-type regions based on the circuit being designed. Form a gate oxide and a gate for each transistor. Finally, metal wires are drawn connecting the requisite inputs to outputs. Often there is not enough space to route the wires in just one level – to overcome this problem, multiple levels of metals are used and “vias” are physical columns that make connections across layers. It is common to have 10 metal levels in modern chips. Instead of making all this for one chip at a time, these are done for 100s to 1000s of chips at a time by doing these operations on an entire silicon wafer (typically 8 inches or 12 inches in diameter). The number of chips on a wafer depends on the size of each chip (these are referred to as dies).

8.3.3: Transistor scaling

Over the past 40 years, scientists have discovered ways to successively reduce the Length and width of transistor every 2 to 3 years. Gordon Moore observed this was possible many years ago and postulated:

Moore's Law: the number of transistors in a dense integrated circuit will double approximately every two years.

A companion to Moore’s Law is another phenomenon called **Dennard scaling**, which states that the supply voltage for a transistor can be reduced linearly with its size. Combining these two, if we shrink the length by 1.41X, the width by 1.41X, we get a transistor that is 2X smaller. We also reduce its voltage supply by 1.41X. The power consumption of a transistor is proportional to its length (because of capacitance of the electric field), and proportional to the (supply voltage)^2. Combining all of these when we go from one generation of transistor to another, we get 2X reduction in size and cubic reduction in 2.7X reduction in its power consumption.