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 abstraction 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 exactly the state machine controller is implemented with logic gates. So let's get started.

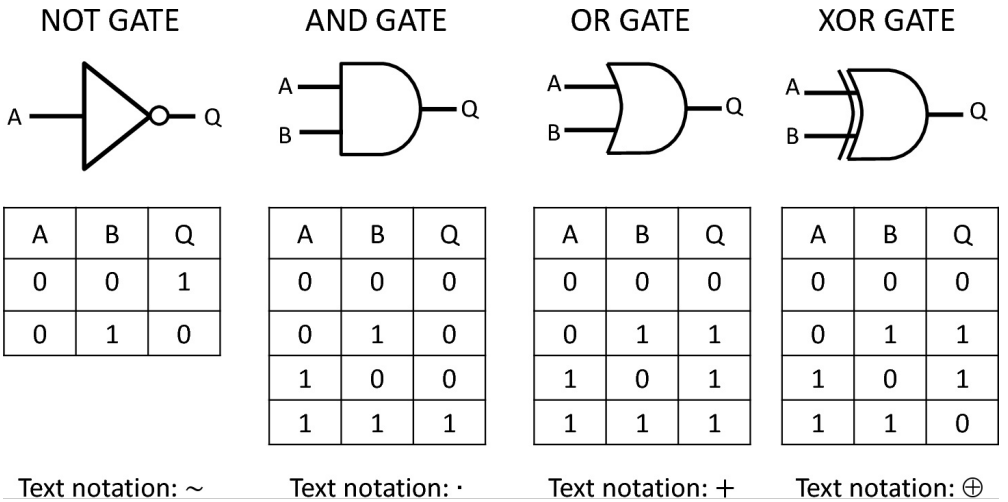
This section will be structured as follows. First we will describe five simple logic gates by (re)introducing the idea of a truth table and how it specifies the functionality of what is desired. Second, we will look at arbitrary truth-tables and show how one can construct a circuit-block using just the 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 logic gates can be combined to implement a register. Finally, we will use registers 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 !, ~ or ' (the apostrophe symbol) in front of a Boolean variable is also used. In this course we will mostly use ~x to denote NOT(x). The left-most column in the Figure below 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

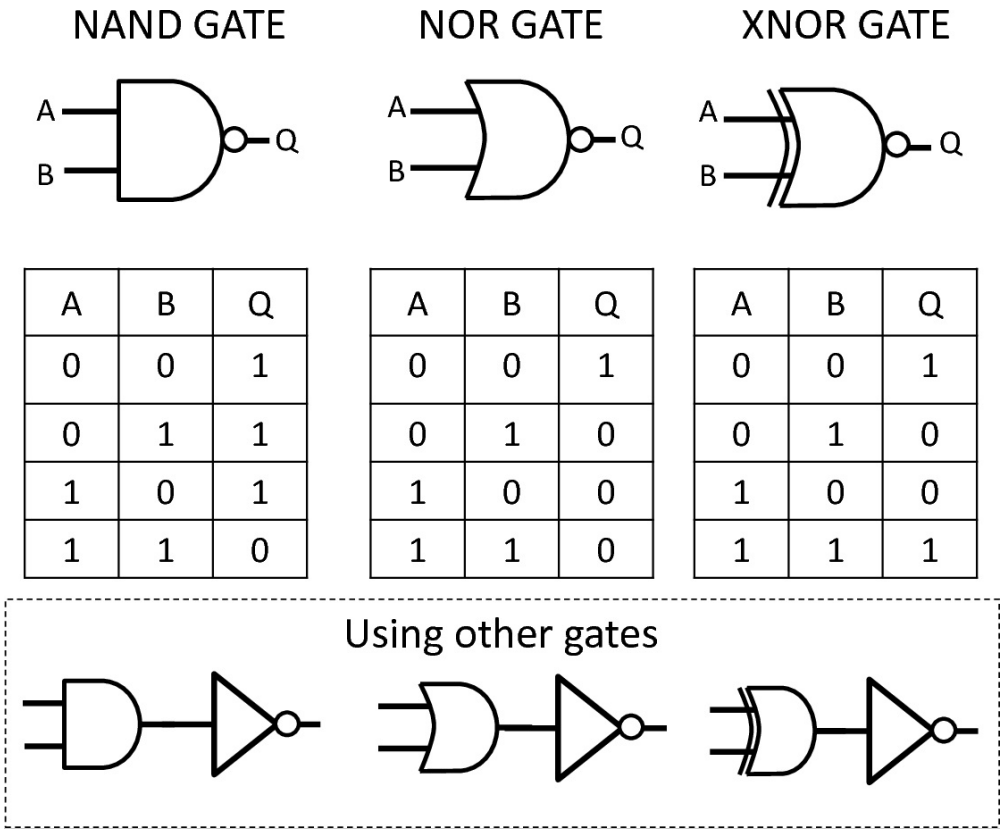
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

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

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 respectively 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. The reason we are looking at these gates is because, in a somewhat counter-intuitive way they are easier to build of transistors than the usual AND and OR gates.



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.)

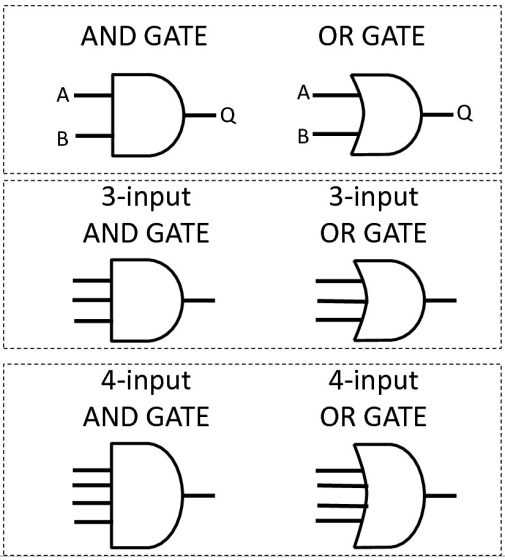


Figure 8.1.1.3: AND and OR gates with various numbers of inputs.

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:

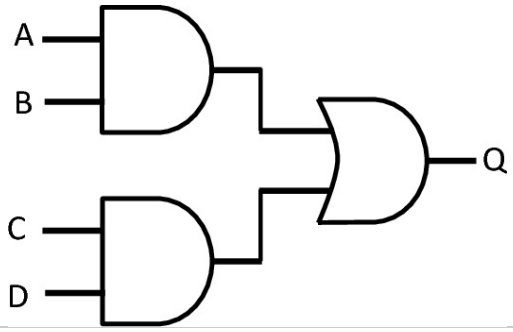
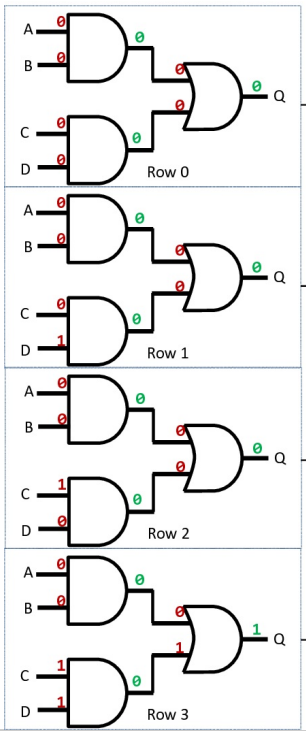


Figure 8.1.2.1: Connecting the outputs of a pair of AND gates to the inputs of an OR gate.

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.

It is important to note here that, when we consider each row, we want to create a boolean function whose output is 1 **ONLY** for that row - this can be achieved by ANDing and considering the complement or orginal version of each variable. Specifically suppose we consider the 1st row, A = 0, B = 1. One could consider the boolean function A OR B which will also produce 1, which is the desired output for this row. However the A OR B function will produce an output of 1 even when A is 1. For this specific truth-table although that works out to be ok, in the general case it will not. The rule to remember is that we want to create a function which is 1 ONLY for that row - and this can be achieved by ANDing the variables together. From the basic AND gate's truth-table we know that can AND will produce a 1 under exactly one condition.

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:

These terms which are 1 are also referred to as **min-terms**. Conversely, one could consider the rows that are 0 (referred to as max-terms) and also construct the boolean equation - we will need to use a different technique and is omitted in this book. That technique is called product of sums.

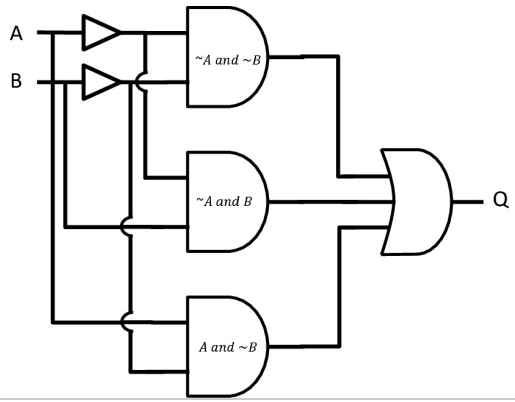


Figure 8.1.2.4: A circuit implementing the above truth table.

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.

A brief note on precedence before we continue further. Just like in regular algebra we have the notion of precedence in Boolean Algebra. Terms within parenthesis are evaluated first. For the operators, the

order of precedence is NOT (~), followed by AND (.), followed by (+).

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.

8.1.3: Laws of Boolean Algebra

Just like basic algebra has some simple laws for algebraic equations, we have simple rules for Boolean Algebra. They are listed below. Their meaning is intuitive and proofs are straight-forward and hence omitted.

This entire sub-section on laws of Boolean Algebra you need to only skim for the exam - you will not be tested on it.

Commutative law	`A + B = B + A` `AB = BA`
Associative law	`(A + B) + C = A + (B + C)` `(AB)C = A(BC)`
Distributive law	`A(B + C) = AB + AC` `A + (BC) = (A+B)(A+C)`
Idempotent law	`A + A = A` `AA = A`
Redundance law	`A + AB = A` `A(A + B) = A`
De Morgan's law	`~(AB) = ~A + ~B` `~(A + B) = (~A)(~B)`

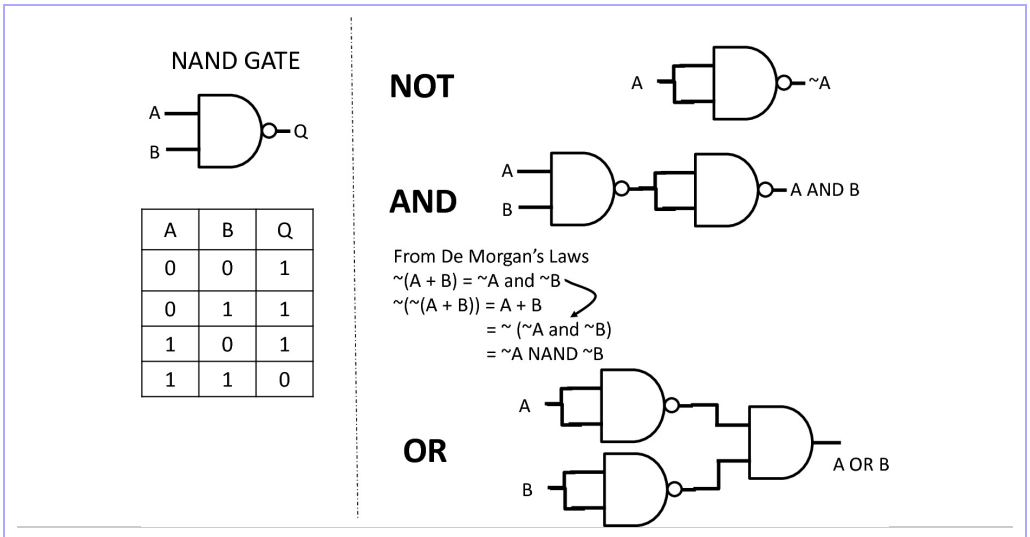
We can apply the first 5 laws and get the following simple results that become handy in simplyfying boolean equations.

Operations on 0	`0 + A = A` `0(A) = 0`
Operations on 1	`1 + A = A` `1(A) = A`
Operations on complement	`~A + A = 1` `~A(A) = 0`
Applying distributive law (a)	`AB + A(~B) = A` `(A + B)(A + ~B) = A`

Applying distributive law (b)	A + ~AB = A + B A(~A + B) = AB

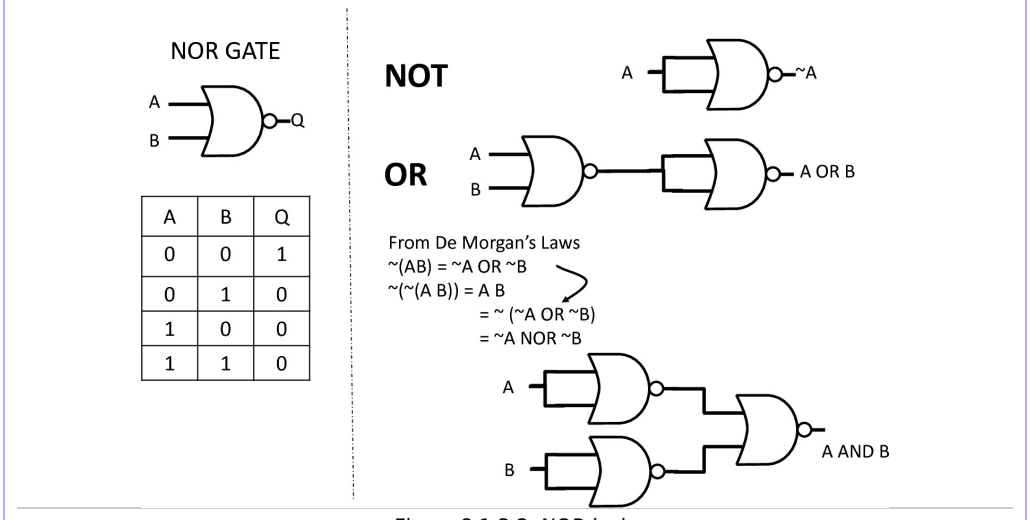
De Morgan's Laws are intruiging and very powerful and we will briefly discuss them in detail. Remember, we introduced NAND and NOR gates. These are useful in a practical way because they can be oddly be built out of fewer transistors than AND or OR gates. De Morgan's Law is a law about Boolean logic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

In the Figure below we show how it can applied to build AND, OR, and NOT gates using only NAND gates. The next figure shows the same using only NOR gates. The NOT gate is a straight-forward application of the truth-table of a NAND gate when both inputs are the same. The AND gate can be construct by inverting the output of a NAND gates, using the NAND-based invertor we just built. The figure shows how we apply De Morgan's law and build an OR gate.



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

Figure 8.1.3.1: NAND logic



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

Figure 8.1.3.2: NOR logic

Aside: In addition to applying these equations, 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.

Recall blocks like the ALU and mux and state machine tables from Chapter 7? We will now look at how these can be constructed using logic gates next.

8.1.4: Datapath Modules

In this section we will build various components in the processor that perform combination logic work and don't do any storage. These are muxes, the ALU, and the incrementor block.

First, we will introduce some notation to interpret the logic diagrams we will draw. The diagram below shows different ways to indicate wires criss-crossing and their meaning. Also, it shows the convention for extracting individual bits out of a multi-bit signal and combining multiple single-bit wires into one multi-bit signal.

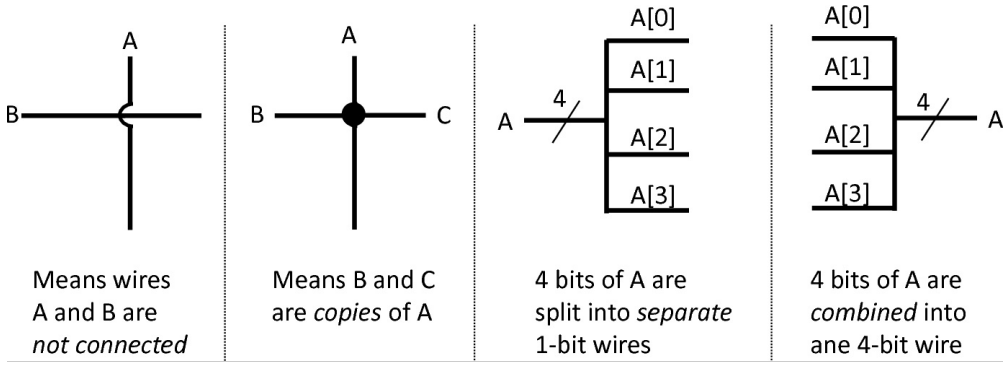


Figure 8.1.4.1: Drawing conventions used.

The simplest thing we can do is take individual gates and combine them to produce blocks that operate on multiple bits. We can thus build an 8-bit invertor and 8-bit AND gate-block as shown below. Indeed this idea can be extended for all the basic gates.

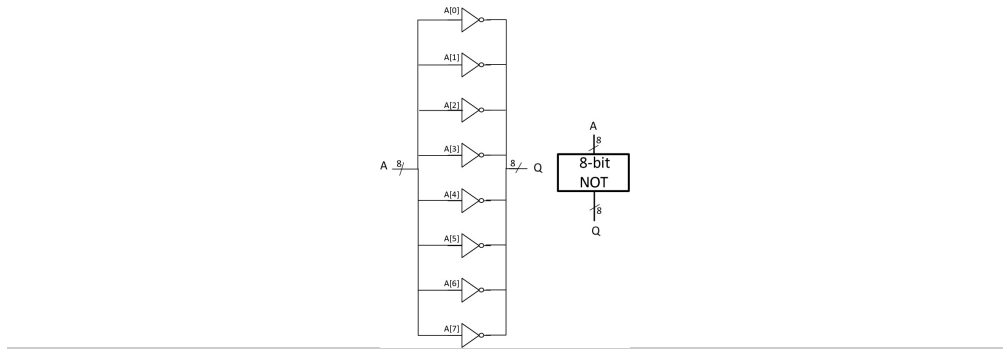


Figure 8.1.4.2: 8-bit invertor

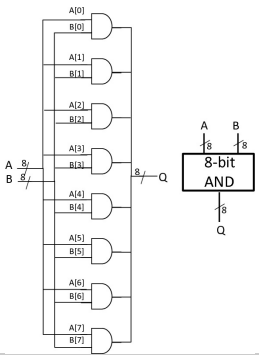
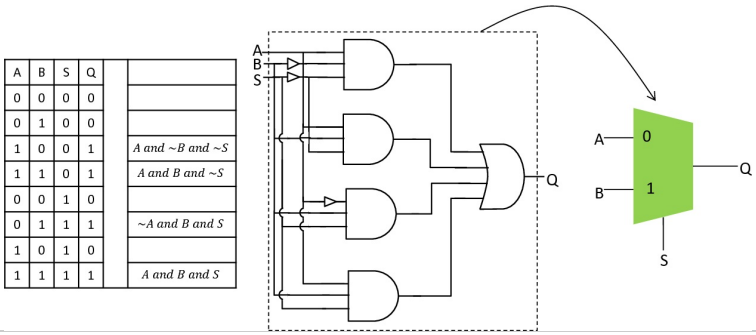


Figure 8.1.4.3: 8-bit AND

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 8.1.4.4: 1-bit Multiplexor

Just like what we did with the 8-bit NOT and AND gates, we can combine eight single-bit muxes to form an 8-bit MUX. Note in the diagram, that the same single-bit Select signal (S) is sent to all the muxes, since we want to do the samel selection for each bit.

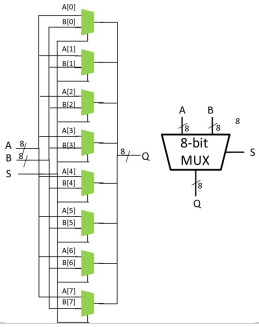


Figure 8.1.4.5: 8-bit Multiplexor

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. In fact, more optimized implementations are possible for the other blocks as well. But in this book, we will look at optimizations for just the adder. Using some of the Boolen Algebra laws from above, we have simplified the Cout implementation. From the truth-table, Cout = ABCin' + A'BCin + AB'Cin + ABC. This can be simplified down as follows:

```
Cout = ABCin' + A'BCin + AB'Cin + ABCin

[Applying Idempotent Law for ABC] (we add two copies of ABCin)
     = ABCin' + A'BCin + ABCin + AB'Cin + ABCin + ABCin + ABCin
```

```
[Applying Associative Law for ABC]
     = ABCin' + ABCin + A'BCin + ABCin + AB'Cin + ABCin

[Applying Distributive Law for ABC]
     = AB(Cin' + Cin) +  BCin(A' + A) + ACin(B' + B)

[Since A + A' = 1]
     = AB + BCin + ACin
```

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

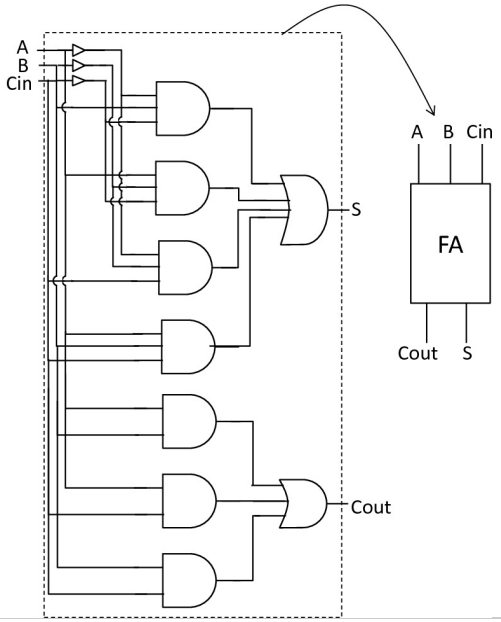


Figure 8.1.4.6: 1-bit Full Adder

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.

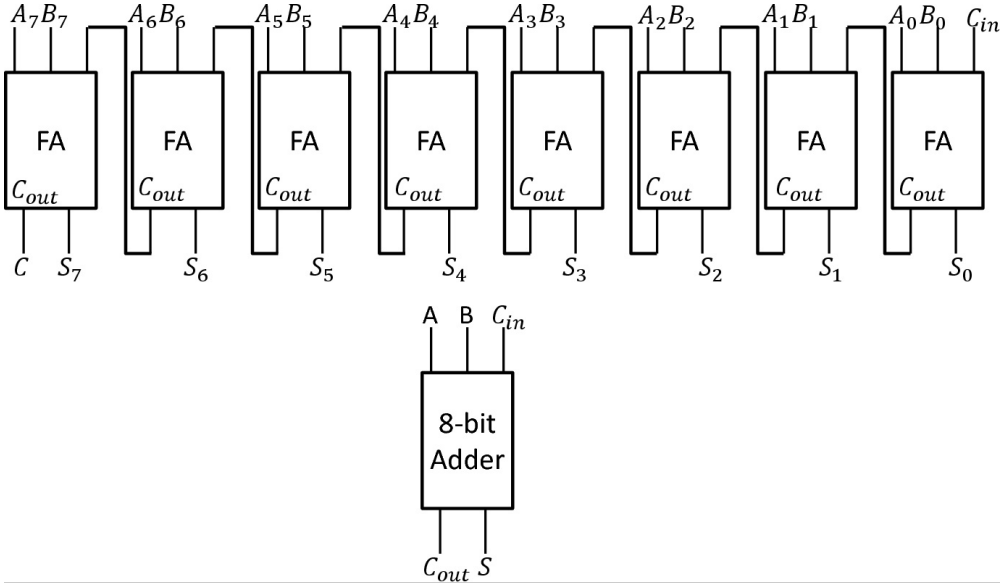


Figure 8.1.4.7: 8-bit Adder.

Aside: This adder we have built is called a ripple-carry adder as the carry-bit ripples through each adder. If a single full-adder circuit takes x nano-seconds, then the an 8-bit ripple-carry adder will take 8x nano-seconds. A 32-bit adder will take 32x nano-seconds and so on. When we have to build 64-bit adders as is necessary in today's modern microprocessors, this delay becomes too large. To reduce this delay there are many optimizations possible. These include carry-look-ahead adders, carry-select adders, and tree-adders.

We need to build another module that **can detect if the zero-flag must be set**. Let us call this a CHECKZ module. From the definition of the Z-flag, we want to set it to 1 if all bits are zero. It can be implemented using AND gates by ANDing together all the bits (after first inverting them) and producing one ouput bit as shown below. To show some hierarchy, we have used two 4-input AND gates fed to another 2-input AND gate to build this block. We could have used a single 8-input AND gate as well.

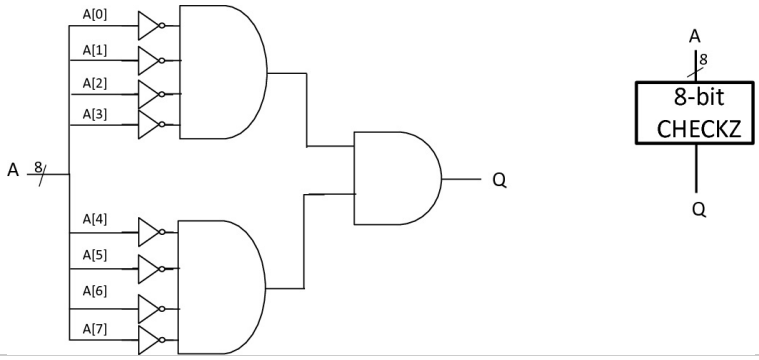


Figure 8.1.4.8: CHECK If ZERO block.

We can combine these above two modules and the MUX to build the ALU block that can perform addition or subtraction based on an ALUop input as shown in the Figure below. Note that the inversion of B's bit and setting the Cin to 1 accomplishes the conversion of B to -B to perform subtraction by doing addition.

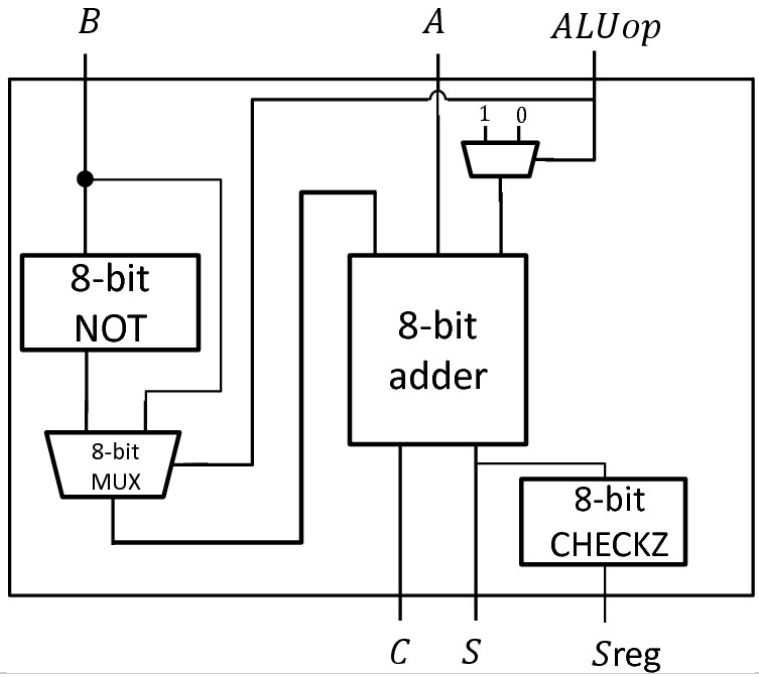
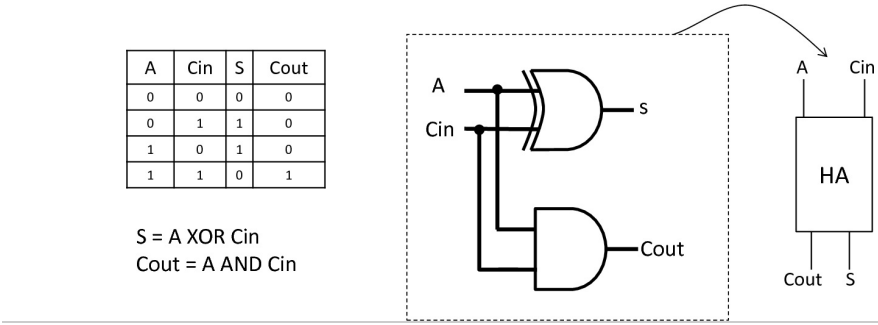


Figure 8.1.4.9: 8-bit complete ALU

To increment the PC by 1 we also used a special block whose only functionality is to increment by 1. While, we could do that with the ALU we just designed, it is useful to contemplate if a simpler circuit

can suffice. Indeed one can build a simpler circuit. To build an **incrementor**, we will start with what is called a **half-adder**. A half-adder circuit takes two inputs and produces a sum and carry output.



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

Figure 8.1.4.10: 1-bit half adder

We can combine many of these just like we did for the 8-bit Full adder. As shown in the Figure below, we can connect together multiple such half-adders and create an 8-bit incrementor.

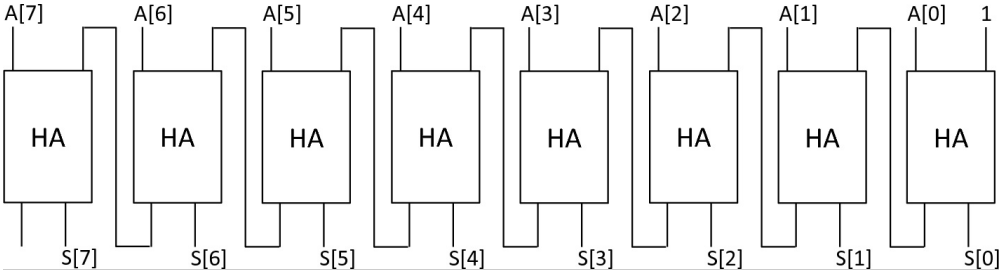


Figure 8.1.4.11: 8-bit Incrementor

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. The basic truth-table of an 8-bit encoder is shown below. It is common to prefix the decoder with the number of inputs and outputs. So the decoder we have shown below is called a 3-to-8 Decoder. The columns in orange are the outputs and the red-boxes denote the rows when the output is 1.

A[2]	A[1]	A[0]	d0	d1	d2	d3	d4	d5	d6	d7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Figure 8.1.4.12: 3-to-8 Decoder truth table

On this truth table you will notice that for any given output variable all outputs are 1 in exactly one row - which thus simplifies quite a bit the logic implementation. The Figure below shows how the truth table can be written in a stylized for with one boolean equation for each row where exactly one output variable is 1. The corresponding logic diagram is shown below the truth-table

A[2]	A[1]	A[0]	Output	
0	0	0	d0=1	*d0=a[2]'a[1]'a[0]'*
0	0	1	d1=1	*d1=a[2]'a[1]'a[0]*
0	1	0	d2=1	*d2=a[2]'a[1]a[0]'*
0	1	1	d3=1	*d3=a[2]'a[1]a[0]*
1	0	0	d4=1	*d4=a[2]a[1]'a[0]'*
1	0	1	d5=1	*d5=a[2]a[1]'a[0]*
1	1	0	d6=1	*d6=a[2]a[1]a[0]'*
1	1	1	d7=1	*d7=a[2]a[1]a[0]*

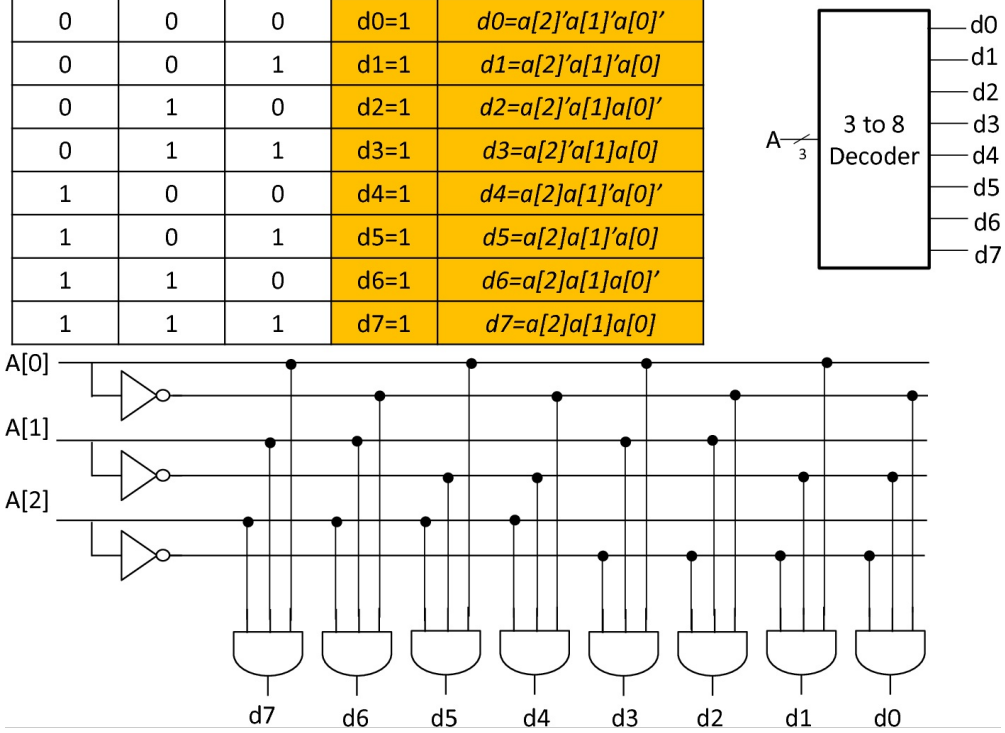


Figure 8.1.4.13: 3-to-8 Decoder Implementation

An **encoder** does the converse. It takes as input 2^n one-hot values and converts it into an n-bit output. Just like decoder naming, they are also prefixed with the number of inputs and outputs. A 8-to-3 Encoder's truth-table and implementation is shown below.

d7	d6	d5	d4	d3	d2	d1	d0	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**

$$A[0] = d1.d3.d5.d7$$
$$A[1] = d2.d3.d6.d7$$
$$A[2] = d4.d5.d6.d7$$

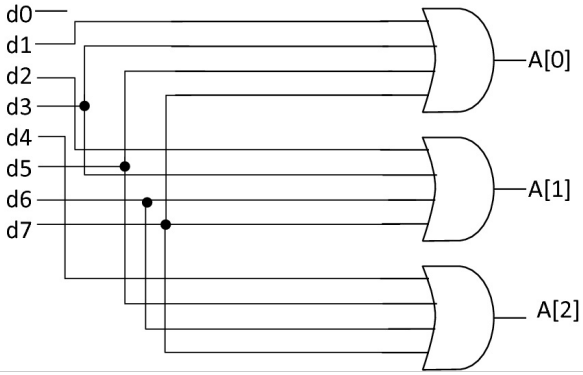


Figure 8.1.4.14: 8-to-3 Encoder Implementation

8.1.5: State machine controller using Logic gates

Revisting our processor from Chapter 7, the one piece that uses logic gates and is not memory is the state-machine controller. We will look at its implementation in detail in this section.

The Figure below shows the high-level overview of the state-machine controller. It internally has a 5-bit STATE register (whose implementation we will look at in the next section, since it is a storage element). A set of control signals are generated each cycle based on the value in the state register. The next-state for the next cycle is computed by the "State update unit" based on the state and the instruction (specifically only the opcode bits). The output of this unit is a 5-bit signal that will get written into the STATE register at the beginning of the next clock cycle (or equivalently the end of the current clock cycle).

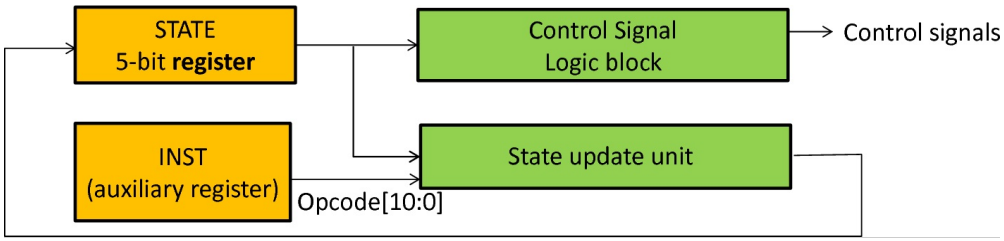


Figure 8.1.5.1: State-machine controller overview

The upshot of all this is, we need to design the logic gates based implementation of the two boxes in green. So let us proceed. First we will look at the control signal logic block.

The large truth table from Chapter 7 is reproduced in the Figure below with the output columns ordered slightly differently. This big table is simply a truth-table with 22 outputs and 5 inputs (each bit of the state register). Using the rules of logic gates we can thus create the logic circuit for each output variable as shown in the Figure. While simple or perhaps intimidating, the control-logic block of the processor is simply these set of logic gates! Note that signals which are one in exactly one row can be computed by just one 5-input AND gate with the original or negated value of the relevant inputs. Signals (like REG_we) which have multple rows that are 1, use an OR gates which is fed by multiple AND gates.

State number	INST_we	REG_we	REG_sel	OFF_sel	OFF_we	VAL_we	VAL1_we	VAL2_we	RF_sel	ADDR_we	REG2_we	A_sel	ALU_op	SREG_we	B_sel	VAL2_sel	RAM_we	RF_we	PC_sel	PC_we	VAL_sel1	VAL_sel0
00000	1	0			0	0	0	0		0	0			0			0	0		0		
00001	0	1	0		0	0	0	0		0	0			0			0	0		0		
00010	0	1	1		0	0	0	0		0	0			0			0	0		0		
00011	0	0		1	1	0	0	0		0	0			0			0	0		0		
00100	0	0		0	1	0	0	0		0	0			0			0	0		0		
00101	0	0			0	1	0	0		0	0			0			0	0		0	1	1
00110	0	0			0	1	0	0	0	0	0			0			0	0		0	1	0
00111	0	0			0	0	1	0	0	0	0			0			0	0		0		
01000	0	0			0	0	0	1	1	0	0			0		0	0	0		0		
01001	0	0			0	0	0	0		1	0			0			0	0		0		
01010	0	0			0	0	0	0		0	1			0			0	0		0		
01011	0	0			0	1	0	0		0	0	1	0	0	0		0	0		0	0	0
01100	0	0			0	1	0	0		0	0	1	1	0	0		0	0		0	0	0
01101	0	0			0	0	0	0		0	0			1			0	0		0		
01110	0	0			0	1	0	0		0	0	0	0	0	1		0	0		0	0	0
01111	0	0			0	0	0	1		0	0			0		1	0	0		0		
10000	0	0			0	1	0	0		0	0			0			0	0		0		
10001	0	0			0	0	0	0		0	0			0			1	0		0		
10010	0	0			0	0	0	0	0	0	0			0			0	1		0		
10011	0	0			0	0	0	0		0	0			0			0	0	1	1		
10100	0	0			0	0	0	0		0	0			0			0	0	0	1		

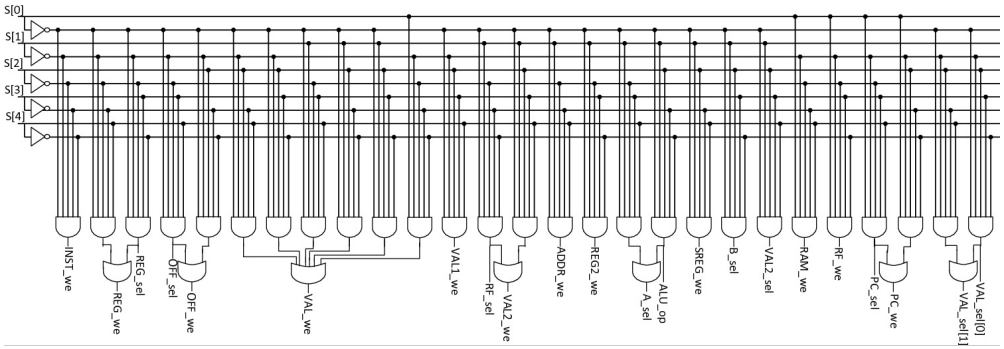


Figure 8.1.5.2: Control signals: truth-table and logic-gate implementation.

Let us look at this in a little bit more detail. The Figure below shows three (randomly) selected control signals, and the subset of the truth table - removing all rows that 0 for all three outputs.

State#	Name	PC_sel	PC_we	INST_we
00000	INST=PM[PC]	0	0	1
10011	PC = PC+1	1	1	0
10100	PC = VAL	0	1	0

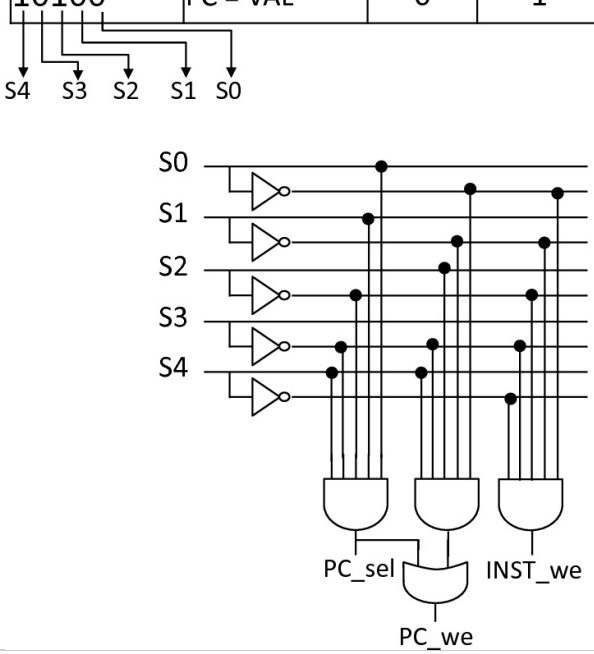


Figure 8.1.5.3: Control signals: truth-table and logic-gate zoomed in view of 3 signals.

Considering PC_sel, it is 1 when the state-number is 10011. This can be expressed as:

```
PC_sel = S[0] AND S[1] AND ~S[2] AND ~S[3] AND S[4]
```

Considering PC_we, it is 1 when the state-number is 10011 or 10100. This can be expressed as:

```
PC_we = (S[0] AND S[1] AND ~S[2] AND ~S[3] AND S[4]) OR
        (~S[0] AND ~S[1] AND S[2] AND ~S[4] AND S[4])
```

Considering INST_we, it is 1 when the state-number is 10011 or 10100. This can be expressed as:

```
PC_we = (S[0] AND S[1] AND ~S[2] AND ~S[3] AND S[4]) OR
        (~S[0] AND ~S[1] AND S[2] AND ~S[4] AND S[4])
```

The circuit for the entire truth-table essentially does such an implementation for each and every control signal. Note also that the 2-bit signal VAL_sel (which controls a 4-input MUX) is represented as two separated 1-bit signals VAL_sel1 and VAL_sel0.

The next piece is the state update unit. Its truth table implementation is "long" but straight-forward. For every opcode, it contains a set of rows that show the transition from one state to another as that instruction is executed. If you recall the microarchitecture trace diagrams, we had a set of states each opcode would sequence through. A truth-table version that shows this by combining all opcodes and their state transitions is shown below. Note here that we have taken all the 4-bit, 6-bit, and 8-bit opcodes and zero-padded them all to be 11-bit signals since that is the longest opcode we have. We are showing the most significant 4 bits grouped together since most of the opcodes are 4-bits and it makes the table easier to follow.

	Opcode	Curr State	Next State
ldi	1110 0000 000	00000	00001
	1110 0000 000	00001	00101
	1110 0000 000	00101	10010
	1110 0000 000	10010	10011
	1110 0000 000	10011	00000
subi	0101 0000 000	00000	00001
	0101 0000 000	00001	00111
	0101 0000 000	00111	01111
	0101 0000 000	01111	01100
	0101 0000 000	01100	01101
	0101 0000 000	01101	10010
	0101 0000 000	10010	10011
	0101 0000 000	10011	00000
cpi	0011 0000 000	00000	00001
	0011 0000 000	00001	00111
	0011 0000 000	00111	01111
	0011 0000 000	01111	01100
	0011 0000 000	01100	01101
	0011 0000 000	01101	10011
	0011 0000 000	10011	00000
ld	1001 0001 100	00000	00010
	1001 0001 100	00010	01001
	1001 0001 100	01001	10000
	1001 0001 100	10000	10010
	1001 0001 100	10010	10011
	1001 0001 100	10011	00000
st	1001 0011 100	00000	00010
	1001 0011 100	00010	01001
	1001 0011 100	01001	00110
	1001 0011 100	00110	10001
	1001 0011 100	10001	10011
	1001 0011 100	10011	00000

	Opcode	Curr State	Next State
add	0000 1100 000	00000	00010
	0000 1100 000	00010	00111
	0000 1100 000	00111	01010
	0000 1100 000	01010	01000
	0000 1100 000	01000	01011
	0000 1100 000	01011	01101
	0000 1100 000	01101	10010
	0000 1100 000	10010	10011
	0000 1100 000	10011	00000
sub	0001 1000 000	00000	00010
	0001 1000 000	00010	00111
	0001 1000 000	00111	01010
	0001 1000 000	01010	01000
	0001 1000 000	01000	01011
	0001 1000 000	01011	01101
	0001 1000 000	01101	10010
	0001 1000 000	10010	10011
	0001 1000 000	10011	00000
cp	0001 0100 000	00000	00010
	0001 0100 000	00010	00111
	0001 0100 000	00111	01010
	0001 0100 000	01010	01000
	0001 0100 000	01000	01100
	0001 0100 000	01100	01101
	0001 0100 000	01101	10011
	0001 0100 000	10011	00000
breq	1111 0000 001	00000	00011
	1111 0000 001	00011	01110
	1111 0000 001	01110	10100
	1111 0000 001	10100	10011
	1111 0000 001	10011	00000
rjmp	1100 0000 000	00000	00100
	1100 0000 000	00100	01110
	1100 0000 000	01110	10100
	1100 0000 000	10100	10011
	1100 0000 000	10011	00000

Figure 8.1.5.4: State update unit: truth-table.

Given this truth-table, we can observe that we have 16 inputs (11 opcode bits and 5 state-number bits) and we have 5 output bits. Just like we have done the previous big truth table and all other truth-tables, we can compute the Sum-of-Products implementation. It gets quite unwieldy here. For each row, we first need a 16-input AND gate (since there are 16 inputs in this truth-table). For bit-0, there are 32 rows whose outputs are 1 and hence we need a 32-input OR gate. The figure below shows the detailed implementation - which is mostly illegible, but conveys to you that the implementation is possible. And you can do it yourself!

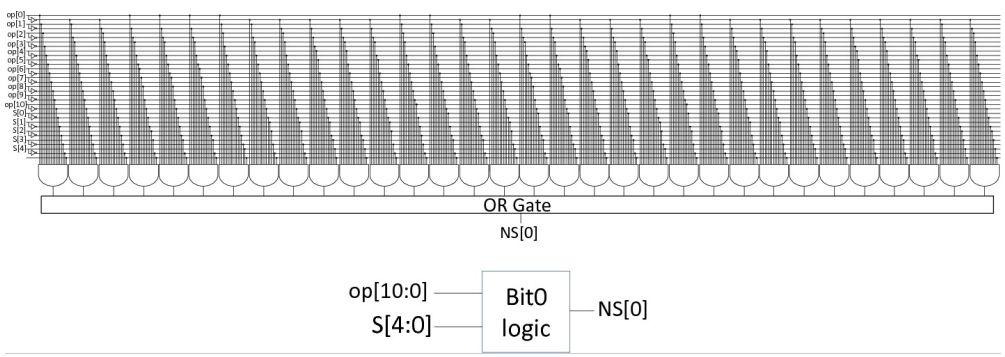


Figure 8.1.5.5: State update unit: implementation for state-bit 0.

Applying the ideas of sum-of-products representation, we can implement all the 5 bits to complete the state-update-unit logic block shown stylistically below hiding away the implementation details.

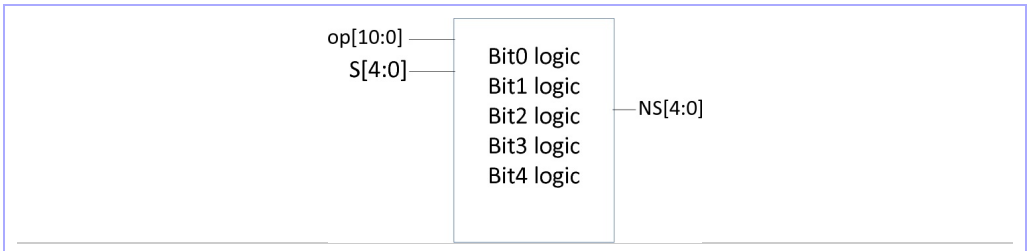


Figure 8.1.5.6: State update unit: full implementation.

With this we have (almost) completely specified the state-machine-controller using purely logic gates. The only piece left in it is the STATE register and the INST register which are sequential elements which we will look at next.

8.1.6: 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. Specifically sequential logic is defined as types of logic blocks whose output depends on the previous input that has been stored in them. 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. We will need the clock-signal very soon when we lock at a block called the D flip-flop.

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.

$\sim S$	$\sim R$	Action
0	0	Not allowed or undefined
0	1	Q = 1
1	0	Q = 0
1	1	**No change**

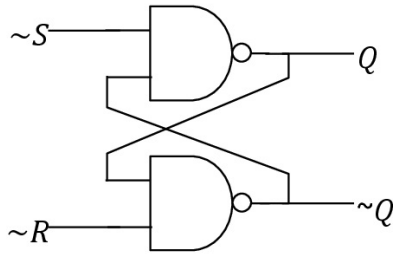


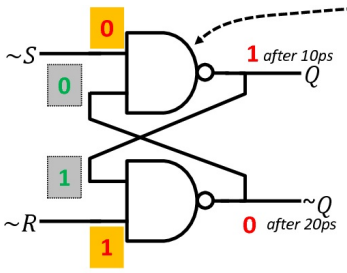
Figure 8.1.6.1: Basic RS Latch

Let us look at the behavior in more detail and how we arrive at that truth table for it. The two inputs we have control over in the RS latch are ~S and ~R. The output is Q and ~Q (i.e. one of the outputs must be inverse of the other). Two rules come in handy while trying to understand the behavior of and RS-latch.

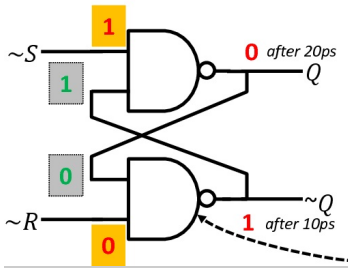
- The NAND truth-table says that if any one input to a NAND gate is 0, the output is guaranteed to be 1. It doesn't matter what the other input is.
- All gates have some finite delay before they can produce their output. We need to dig one level deeper into what gates are made of. We will look at this in detail in the next section, but for now, we will take it that these gates can be made out of transistors. And each transistor has a certain delay to turn on or off, effectively inducing a delay for each logic gate. Assume that each of these gates has 10 pico-seconds (ps) of delay. It does not matter what this number is - we know it is non-zero.

  The output will be produced only after that amount of delay. In the intermediate time window of 0 to 10ps, the output can be anything -- 0 or 1.

We can now methodically look at all 4 combinations of inputs. The immediate Figure below shows the behavior for 0/1 and 1/0 inputs to ~S and ~R and we elaborate on them next.



	0 ps		10ps	...	20ps	
~S	0	0	0	0	0	0
~R	1	1	1	1	1	1
Q	0 or 1	0 or 1	1	1	1	1
~Q	0 or 1	0 or 1	0 or 1	0 or 1	0	0



	0 ps	...	10ps	...	20ps	
~S	1	1	1	1	1	1
~R	0	0	0	0	0	0
Q	0 or 1	0 or 1	0 or 1	0 or 1	0	0
~Q	0 or 1	0 or 1	1	1	1	1

Figure 8.1.6.2: RS Latch Behavior for 0/1 and 1/0 inputs.

Given ~S = 0 and ~R = 1, we know for sure that the output of the top NAND gate will be 1 (because there is one 0 input). We get Q = 1. After 10 more ps (total 20ps), we can derive the value of ~Q from the second NAND which has both inputs set to 1, and hence outputs 0. This latch will then stabilize and hold onto its value so long as ~S and ~R are held at 0 and 1.

We can repeat this process for ~S = 1 and ~R = 0 to get ~Q = 1 (@10ps), Q = 0 (@20ps).

When both inputs are set to 1, then based on whether we previously stored a 0 or a 1, the latch will continue producing that same output. Looking at the Figure below we can deduce the behavior. For either the top or bottom NAND gate (actually the gate whose output was previously 1), one input has to be zero, and it will produce a 1 immediately (no change). The other gate will then produce a 0 (also no change).

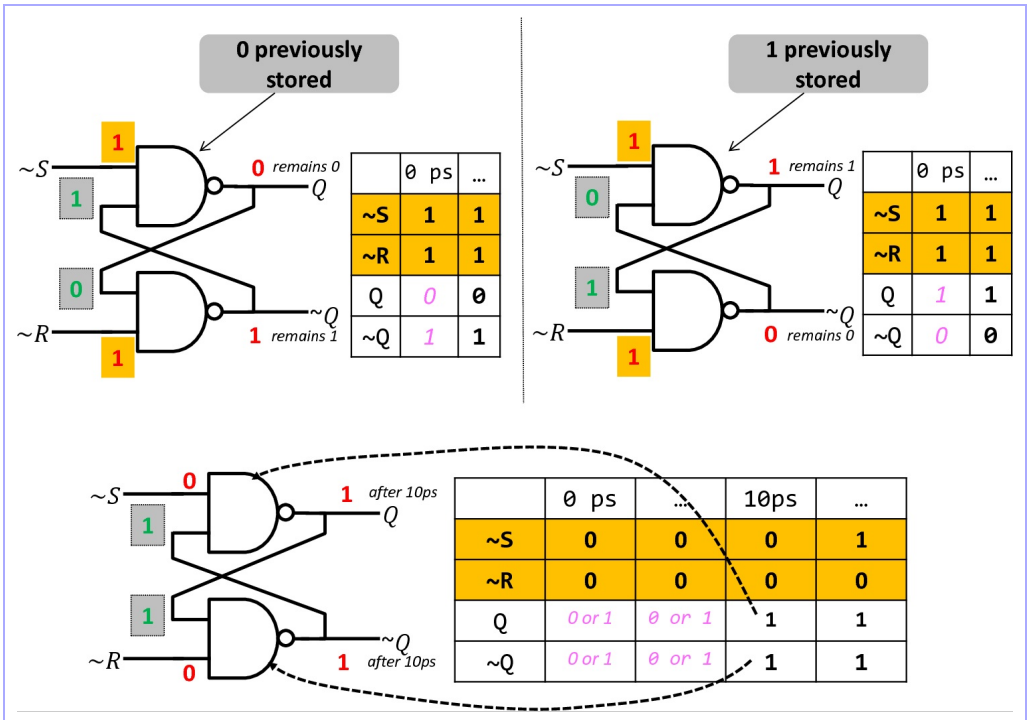


Figure 8.1.6.3: RS Latch Behavior for 1/1 and 0/0 input.

Now suppose that if ~S = 0 and ~R = 0. Q and ~Q would then become 1 (after the 10 ps second and remain that way). Since, we want Q to be the inverse of ~Q, we would like to disallow this scenario from occuring and call this the disallowed state. To disallow this state from ever occurring, we are going to modify the RS latch.

This desired modification can be achieved 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. It basically adds two NAND gates fed by a D and E input whose outputs are connected to the RS latch.

The Figure below shows the design of a gated D-latch. The behavior can be deduced by first looking at E.

- When E is 0, then the new gates will always output 1, generating the hold-state of 1/1 inputs for the RS latch.
- When E is 1, then the D inputs becomes effectively the value we want to write into the RS latch.
  - When D is 0, the top-gate's (the new one we added - shown in red) output becomes 1, and the bottom gate's output becomes 0, generating the 1/0 input combination for the RS latch.
  - When D is 1, the top-gate's (the new one we added - shown in red) output becomes 0, and bottom gate's output becomes 1, generating the 0/1 input combination for the RS latch.

E	D	Q	Action
0	X	Q(previous)	**No change**
1	0	0	Reset
1	1	1	Set

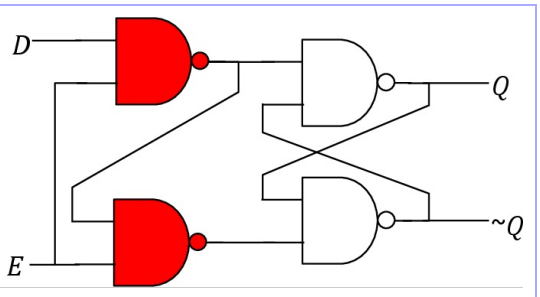


Figure 8.1.6.4: Gated D-Latch

Even this doesn't quite give us the capability of being able to modify values only at the clock-edge. We accomplish by constructing 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.

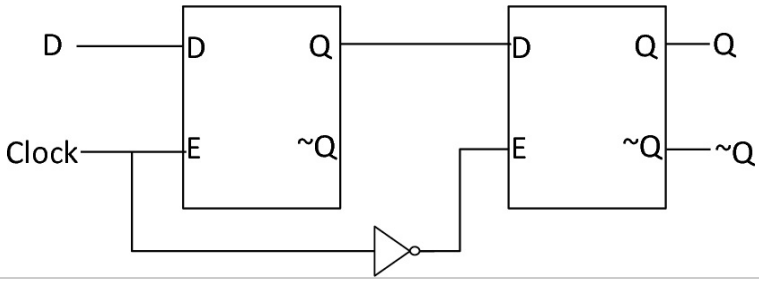


Figure 8.1.6.5: Master-Slave D Flip-Flop

We explain the detailed behavior using the Figure below using color for exposition. Even though the values of Din can only be 0 or 1, for exposition, let us assume the Din input can be various colors at different times: blue, salmon, green, and violet. The hatched yellow regions denote any other value and indicate that the value can be changing (or anything in that time). We also have colored the master D-latch green and the slave D-latch orange. Q0 indicates the output of the master and Q1 indicates the output of the slip (which is the final output). The clock signals for each are shown in green and orange respectively, and note they are complements of each other. In the explanation below we will use the terms green latch and organe latch to aid in exposition.

The basic behavior is that at the end of the first half of its clock cycle, the green latch will enter the hold state and for the rest of that clock-cycle (green clock = 0), it will produce as its output whatever value was available at that clock edge. Until the clock-edge, the output will be identical to whatever Din is. In clock-cycle 0, this will be blue, in the next cycle it will be salmon, next green, and next violet.

Now, consider the orange latch. It is controlled by a clock-signal driving its E input that is phase-shifted compared to the green latch. During its write-stage (which happens during the hold-stage of the green D-latch), it will produce the value provided to it by the green latch. In the Figure, you can observe this by looking at the the first half of every orange clock cycle and indicate by the cyan arrow, which denotes the Q0 to D wire in the flip-flip circuit. In the second-half of the orange clock cycle, the orange latch will enter its hold state (since E is zero). At this point, it will continue holding whatever was available for the rest of that clock cycle -- shown by retaining the same color for the rest of that clock cycle. Effectively holding for an entire clock cycle (both 0 and 1 phase), the Din value that was available at the beginning of the orangle clock-cycle -- the rising edge (or equivalently middle of the green clock-cycle - its falling edge).

On cycle 1, the same thing repeats. In the Figure we denote this value at the clock-edge with the salmon color and notice how that is also held steady for any entire clock cycle as the output, regardless of how Din changes in the intervening time.

Using such an edge-triggered flip-flop we can performing combination logic work for an entire clock-cycle, store the result in an auxiliary register, and proceed to the next cycle using this computed value.

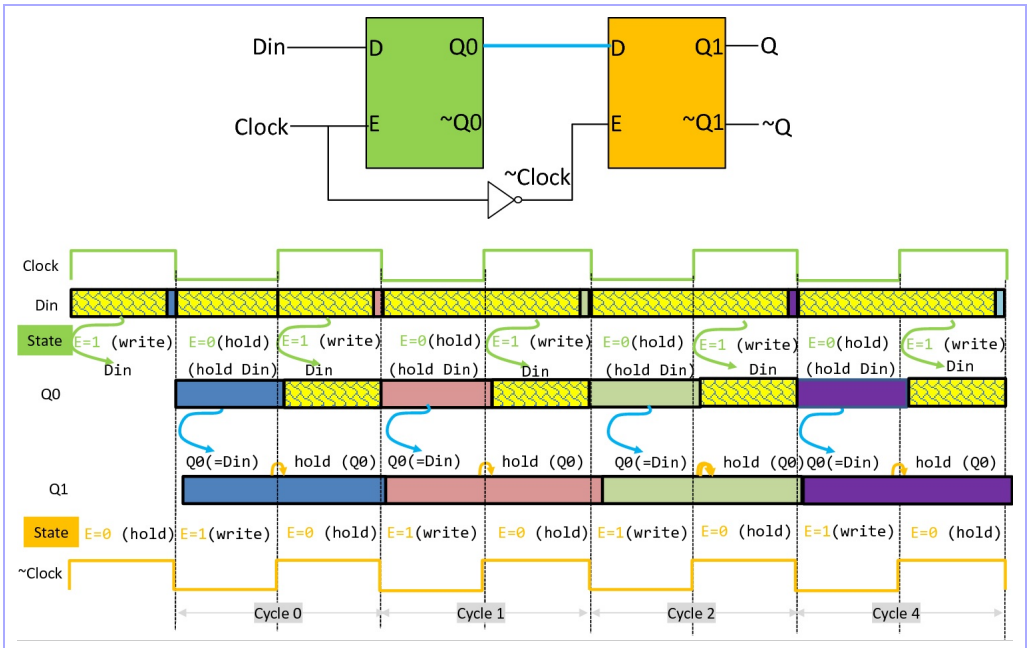


Figure 8.1.6.6: Master-Slave D Flip-Flop Behavior

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.

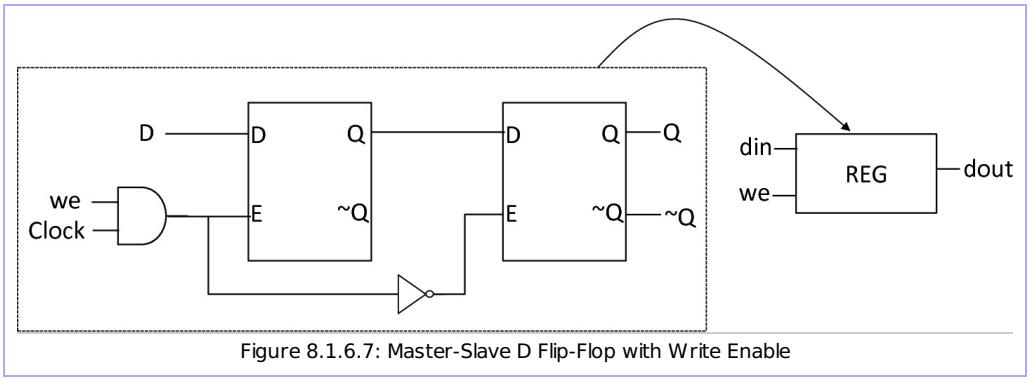


Figure 8.1.6.7: Master-Slave D Flip-Flop with Write Enable

Just like we did the NOT gates and muxes we can combine 8 such flip-flops to form an 8-bit block which can serve as the auxliary register we used to build our processor.

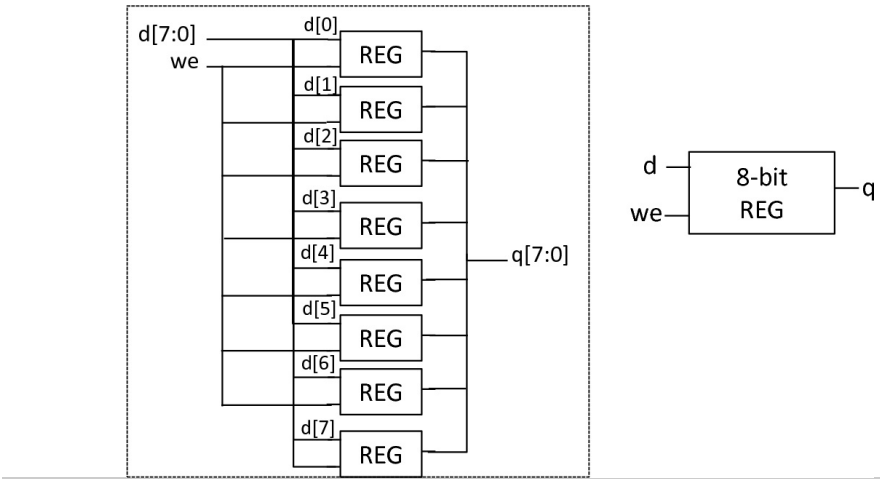


Figure 8.1.6.8: 8-bit REG. Used to implement all Auxiliary registers in processor.

8.1.7: 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. This is just the 8-bit auxiliary register we discussed in the previous paragraph. 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. What we have now is an 8-wide, 8-deep memory as shown in the Figure below.

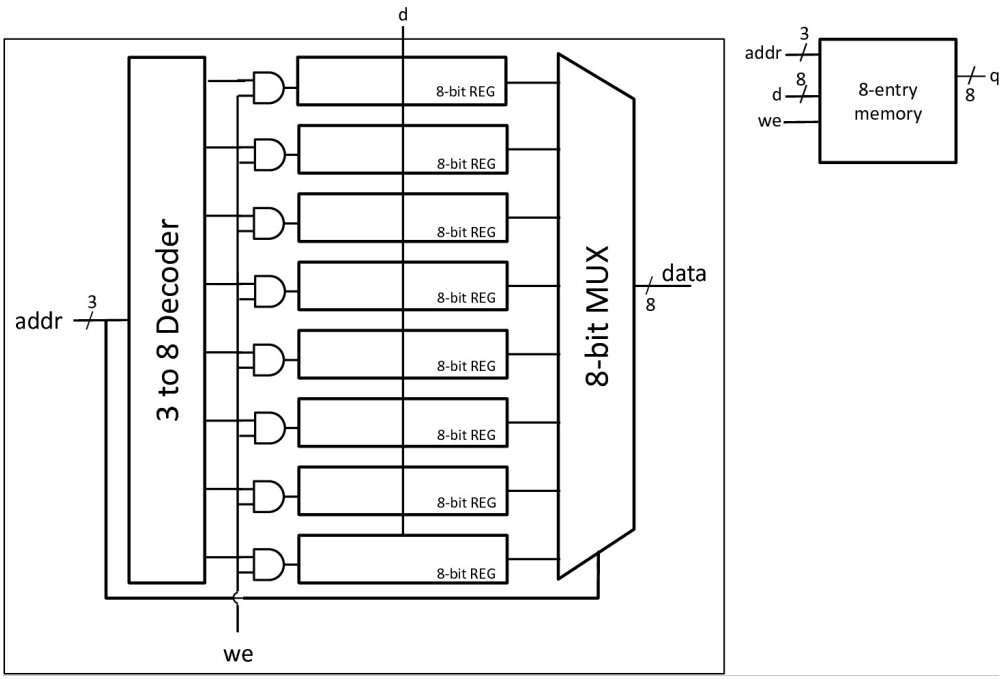


Figure 8.1.7.1: 8-entry memory with 8-bits per entry.

We can combine four of these to create the 32-entry register file memory as shown below. Note in the figure below the wires drawn in red and blue. The red wire now use bits 4 and 3 to select which of 8-entry memories to choose from to produce as the final outtput. Furthermore each 8-entry memory is fed bits [2:0] from the address to select a specific row internally. We also need a little bit more additional to logic to make sure we write to only one of the 8-entry memories (which will in-turn write to exactly one row within that memory). To accomplish that, we use a 2-to-4 Decoder that uses bits

[4:3] to generate a one-hot signal each of which is ANDed with the write-enable signal and sent as input into each 8-entry memory.

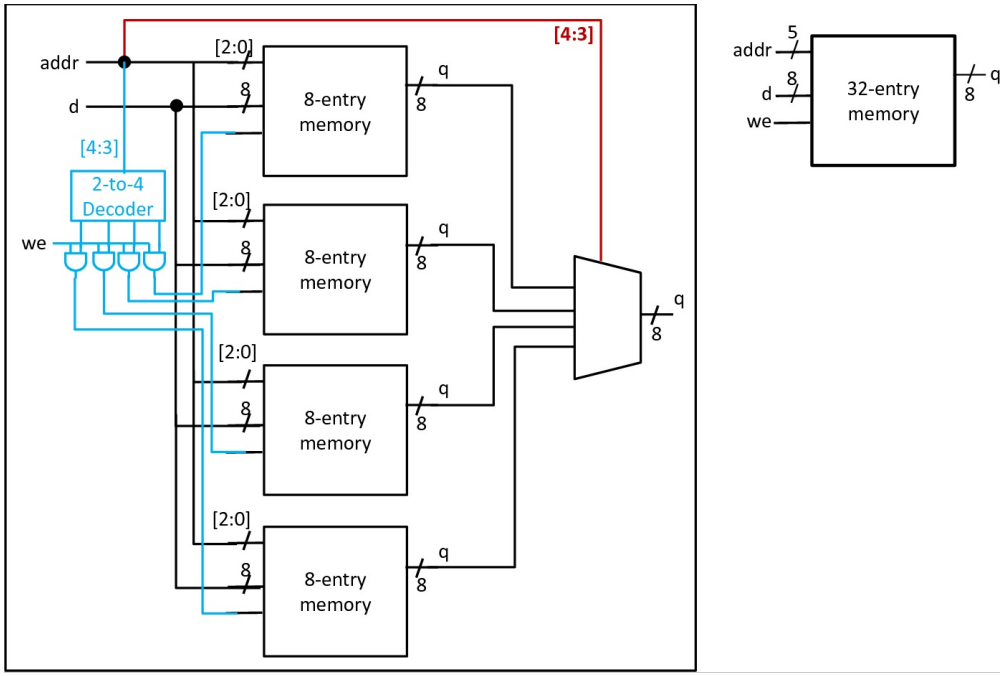


Figure 8.1.7.2: 32-entry 8-bit memory (Used for Register File).

We can now combine 4 of these again to create a 128-entry memory. As is evident the only change will be that we now send bits [4:0] as the addr bits into each 32-entry memory which will serve as our building block. The final selection mux and the write-decoders use bits [6:5] now. This Figure is shown below.

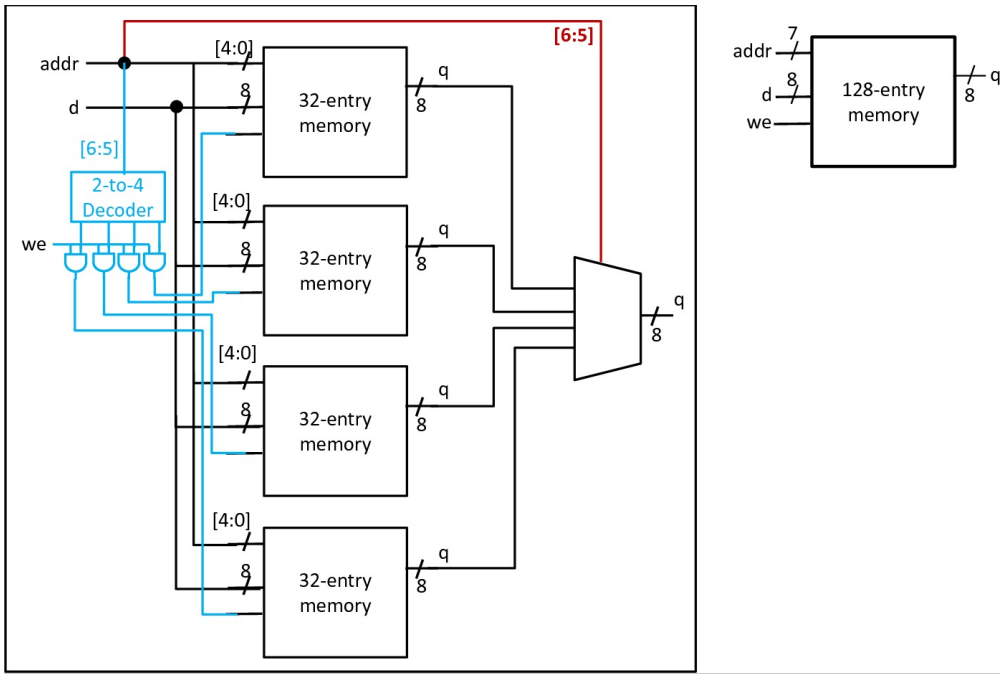


Figure 8.1.7.3: 32-entry 8-bit memory (Used for Register File).

This idea can be generalized to use an N-entry memory and combine R of them to create an N*R capacity memory * 8 bits. A stylized Figure representing this circuit is shown below.

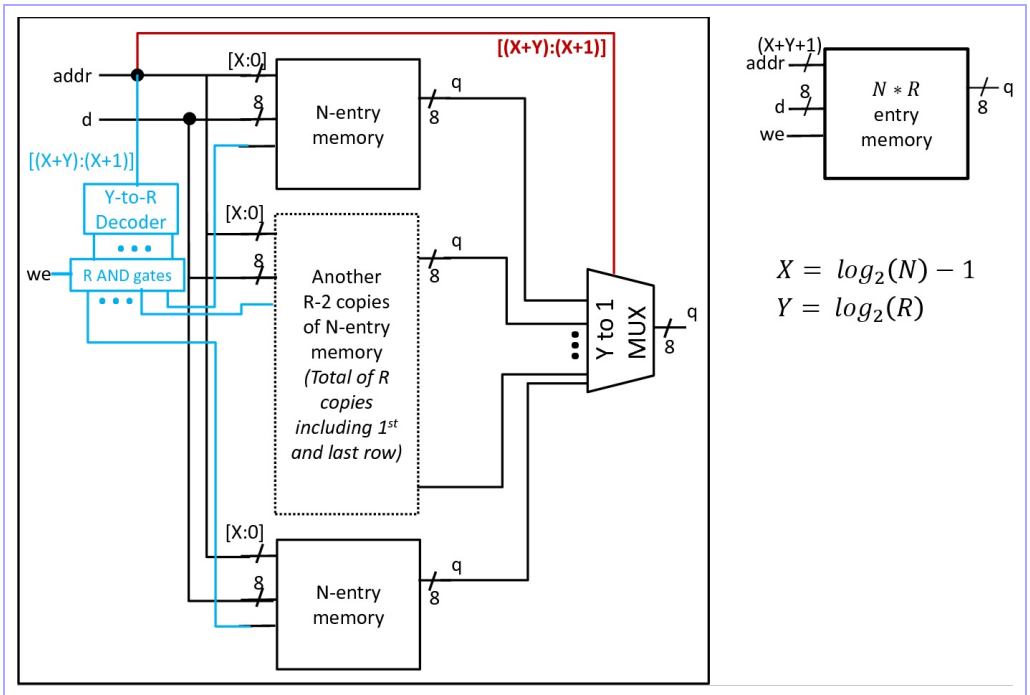


Figure 8.1.7.4: General memory organization.

The Table below shows how we can hierarchicaly combine and build larger memories. Any combination of N and R are allowed. Due to vagaries of how transistors are manufactured microprocessor manufactures prefer certain combinations -- that is well beyond the scope of this book. In the table we show one hierarchy of combining (16 X 32-entry memories) to first form a 512-entry memory. We can then combine (16 x 512-entry memories) to form a 8192-entry memory. We can combine two of these to form a 16384-entry memory. We can combine (4 x 16384-entry memories) to form a 65536-entry memory, which we can use for our processor's program memory and data-memory. Not that our program memory needs 16-bits per entry. For that memory, we can use a 16-bit register in the first 8-entry memory block.

N	R	Capacity	# addr bits
8	4	32 bytes	5
32	16	512 bytes	9
512	16	8192 bytes	13
8192	2	16384 bytes	14
16384	4	65536 bytes	16

Using these building blocks we can construct the register file, program memory and data-memory (RAM). In the previous section, we built the state-machine controller, and in the one prior to that we built MUXes and the ALU. Thus we have seen how all the elements of the processor can be built out of logic gates.

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 named source, drain, and gate. The source and drain are fancy terms refer to two ends of the circuit that is going to be connected or left open. The gate decides whether the circuit should be closed or 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.

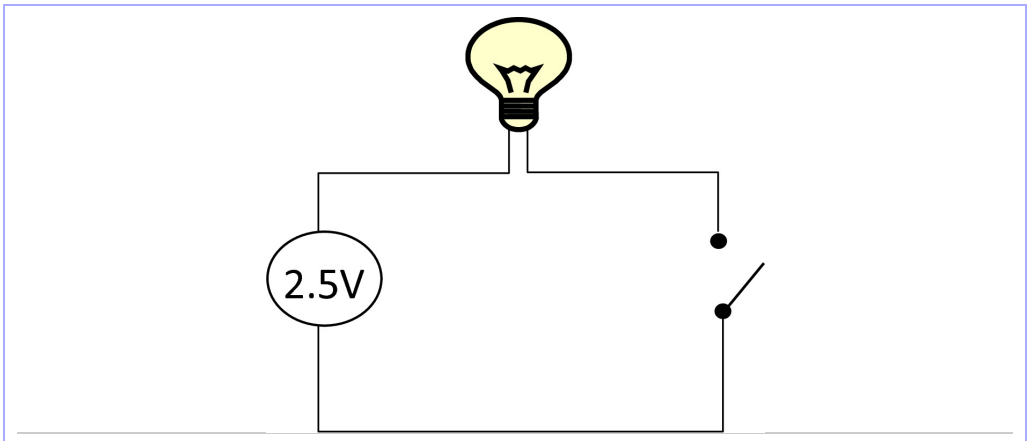
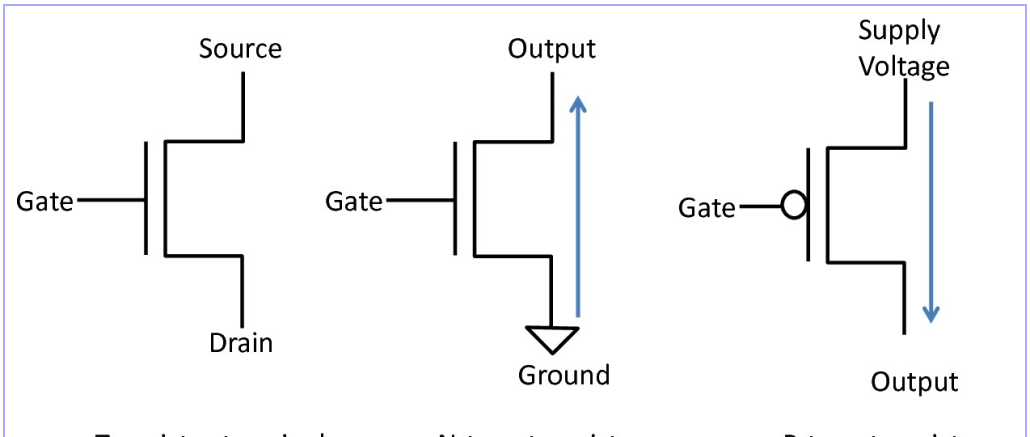


Figure 8.2.1:

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.



Transistor terminals

N-type transistor

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

P-type transistor

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

Figure 8.2.2: Basic transistor behavior.

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.

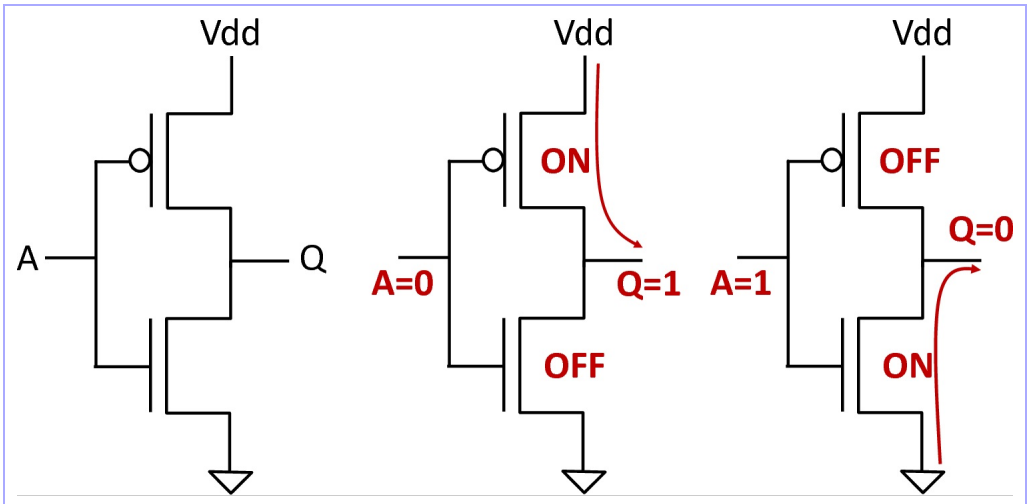


Figure 8.2.3: The implementation of a NOT gate using transistors.

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.

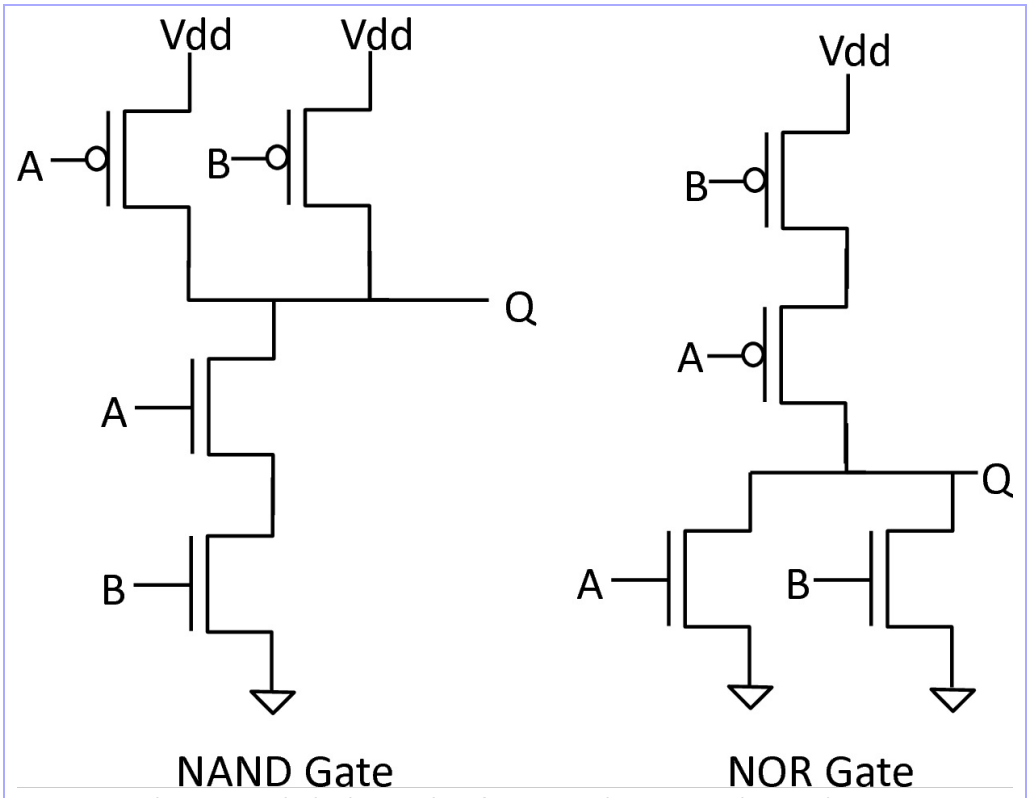


Figure 8.2.4: The implementation of a NAND and NOR gate using transistors.

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 construct.

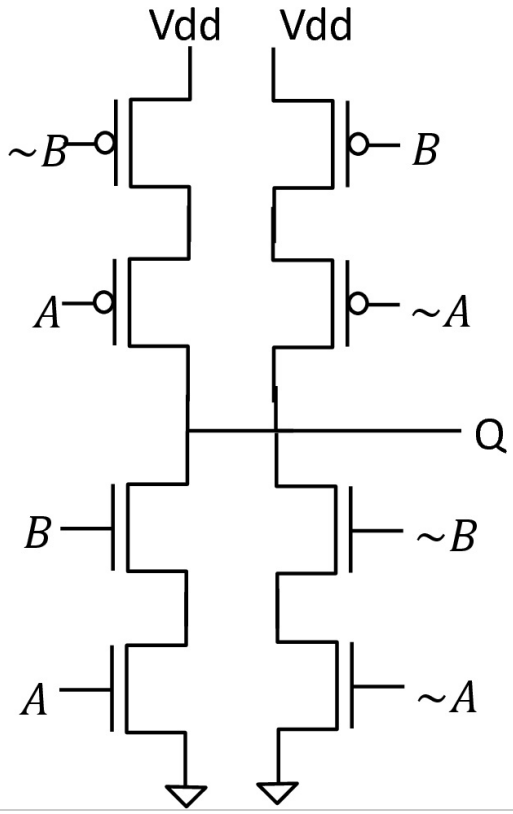


Figure 8.2.5: The implementation of an XOR gate using transistors.

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.

The table below summarizes the number of transistors required for simple gates and for the entire processor we built. In the calculation below, we approximate the control signals block as comprising for 22 5-input AND gates + 5 invertors. We also approximate the state-update block as 32 16-input AND gates, 16 invertors, and 5 32-input OR gates. For the memories we are ignoring the number of transistors needed for the various decoders and the various muxes at the different levels of hierarchy.

Block	# Transistors
Basic elements	
NAND2	4
NOR2	4
NOT	2
AND2	6
OR2	6
AND (n inputs)	2n+2
OR (n inputs)	2n+2
MUX	40
Combined logic gates	
FA 1-bit	74
HA 1-bit	6
1-bit REG	40

3-to-8 Decoder	68
8-to-3 Encoder	30
Datapath modules	
FA 8-bit	592
CHECKZ 8-bit	32
NOT 8-bit	16
MUX 8-bit	320
8-bit REG	320
Microarchitecture blocks	
ALU	1000
Incrementor	48
State machine controller (control sigs)	274
State machine controller (next state)	1,450
Total AUX regs (9 8-bit REGS)	2,880
RF (32 8-bit REGs)	10,240
PM (16384 1-bit REGs)	655,360
RAM (16384 1-bit RGEs)	655,360
Total	**1,326,612**

The processor we have designed needs more than 1 million transistors - these tiny things quickly add up! If we removed all the memories, we are left with a measly 15892 transistors. To put this in perspective, Intel's first microprocessor (the 4004) was introduced in 1971 had merely 2300 transistors. You can find its detailed transistor-level schematic here. and all about it at the 4004 page. Its high-level schematic can be found here. That processor had no transistors devoted for memories inside the processor chip - it was a separate chip.

It is true that the processor we have designed here is more complex than that first processor. So Kudos!

> **Aside:** Modern microprocessors have 100 million to 10 billion transistors on a single chip (the SPARC M7 processor announced in 2015). It is probably quite evident to you know that most of those transistors are devoted to storage structures of some sort. In modern processor these storage structure may be memories or could be auxiliary storage structures like prediction tables, and caches.

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!

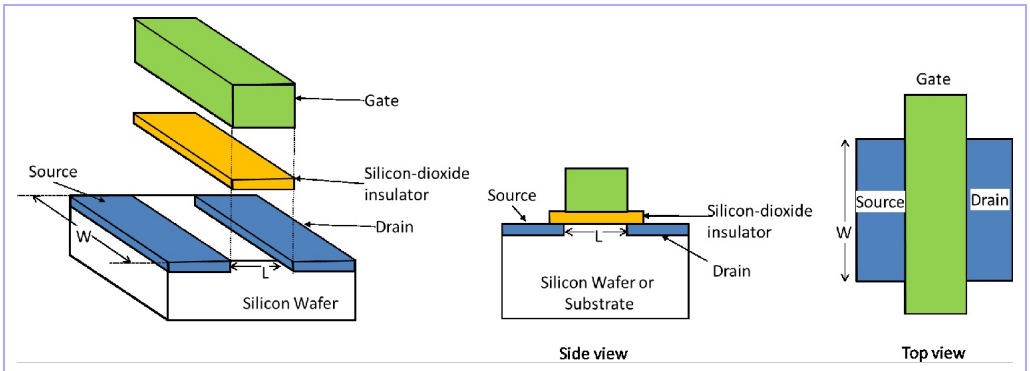


Figure 8.3.1: Components and Physical Organization of a Transistor

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.

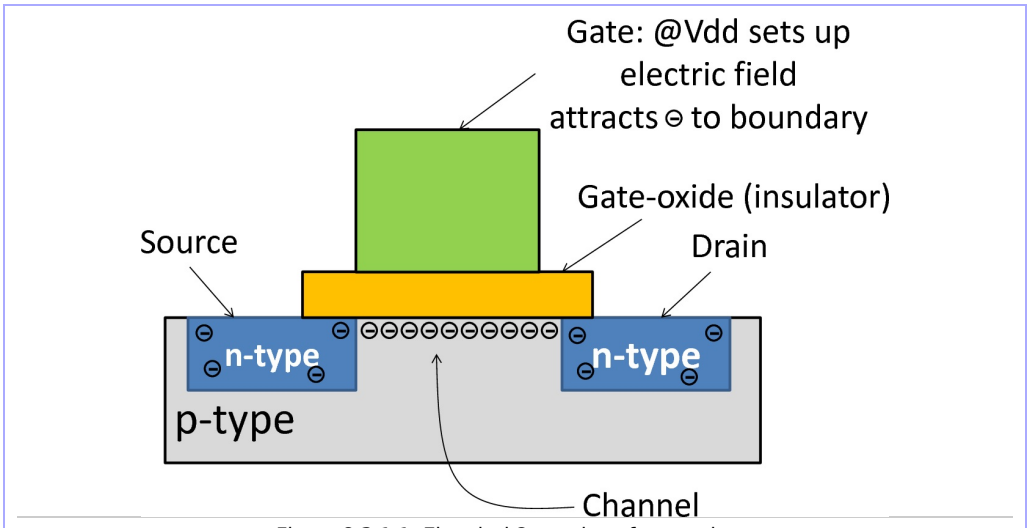


Figure 8.3.1.1: Electrical Operation of a transistor

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

8.3.2: Physical Manufacturing of a Transistor

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 channel length. 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 area and cubic reduction in its power consumption.

8.4: Conclusion

We hope with this book you are now left with a better understanding of computer science, how computers operate, and how they can be programmed. Today's computers are made of millions to billions of transistors - but their essence of operation is based on the transistor behavior we learned in this chapter, gates, microarchitecture, and ISA principles in previous chapters. There are 100s of programming languages, but the foundational principles of almost all programming languages are based on the ideas we covered early on.

You now know more about computers. You also probably feel there is a lot more you don't know about than you thought you didn't know about when you started!

The basic ideas in this book and course should equip you on your journey to build future computers, systems, languages, and software. Good luck!