7

Logic Gates, Registers, and Memories

We are ready to design a real computer now. Before we start, let us quickly take a glance at some of the main requirements and constraints for designing a computer as described in the last few chapters.

Way Point 3

- *A computer needs a central processing unit, set of registers, and a large amount of memory.*
- *A computer needs to support a complete, concise, generic, and simple instruction set.*
- *SimpleRisc is a representative instruction set. To implement it, we need to primarily have support for logical operations, arithmetic computations, register and memory accesses.*

Figure 7.1 shows a plan for the next few chapters. In this chapter, we shall look at designing simple circuits for logical operations, registers, and basic memory cells. We shall consider arithmetic units such as adders, multipliers, and dividers in Chapter 8, and subsequently combine the basic elements to form advanced elements such as the central processing unit, and an advanced memory system in Chapters 9, 10, and 11.

Before, we proceed further, let us warn the reader that this chapter is meant to give a brief introduction to the design and operation of logic circuits. This chapter takes a cursory look at digital logic, and focuses on introducing the broad ideas. A rigorous treatment of digital logic is beyond the scope of this book. The interested reader is referred to seminal texts on digital logic [Taub and Schilling, 1977, Lin, 2011, Wakerly, 2000]. This chapter is primarily meant

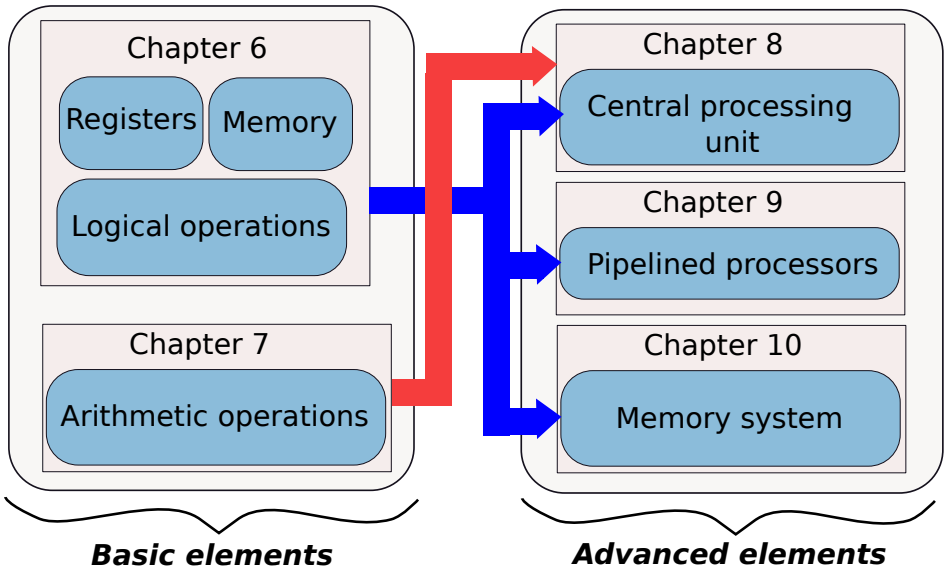

Figure 7.1: Plan for the next few chapters

for two types of readers. The first type of readers are expected to have taken an introductory course on digital logic, and they can use this chapter to refresh their knowledge. The readers in the second category are presumed to have little or no background in digital electronics. We provide enough information for them to appreciate the nuances of digital circuits, and their operation. They can use this knowledge to understand the circuits required to perform computer arithmetic, and implement complex processors.

For implementing logical operations such as bitwise AND, OR, shifts, and register/memory cells, we typically use silicon based circuits today. Note that this was not always the case. The earliest computers in the 19^{th} century were made from mechanical parts. Till the sixties, they were made of vacuum tubes. It is only after the discovery of the transistor and integrated circuit technology that computer processors started getting manufactured using silicon. However, this might be a passing trend. It is perfectly possible in the future that we will have computers made of other materials.

7.1 Silicon based Transistors

Silicon is the 14^{th} element in the periodic table. It has four valence electrons and belongs to the same group as carbon and germanium. However, it is less reactive than both.

Over 90% of the earth's crust consists of silicon based minerals. Silicon dioxide is the primary constituent of sand, and quartz. It is abundantly available, and is fairly inexpensive to manufacture.

Silicon has some interesting properties that make it the ideal substrate for designing circuits and processors. Let us consider the molecular structure of silicon. It has a dense structure, where each silicon atom is connected to four other silicon atoms, and the tightly connected set

of silicon atoms are bound together to form a strong lattice. Other materials notably, diamond, have a similar crystalline structure. Silicon atoms are thus more tightly packed than most metals.

Due to the paucity of free electrons, silicon does not have very good electrical conduction properties. In fact, it is midway between a good conductor, and an insulator. It is thus known as a *semiconductor*. It is possible to slightly modify its properties by adding some impurities in a controlled manner. This process is called *doping*.

Definition 44
A semiconductor has an electrical conductivity, which is midway between a good conductor and an insulator.

7.1.1 Doping

Typically, two types of impurities are added to silicon to modify its properties: *n-type* and *p-type*. N-type impurities typically consist of group V elements in the periodic table. Phosphorus is the most common n-type dopant. Arsenic is also occasionally used. The effect of adding a group V dopant with five valence electrons is that an extra electron gets detached from the lattice, and is available for conducting current. This process of doping effectively increases the conductivity of silicon.

Likewise, it is possible to add a group III element such as boron or gallium to silicon to create p-type doped silicon. This produces the reverse effect. It creates a void in the lattice. This void is also referred to as a *hole*. A hole denotes the absence of an electron. Like electrons, holes are free to move. Holes can also help in conducting current. Electrons have a negative charge, and holes are conceptually associated with a positive charge.

Now that we have created two kinds of semiconductor materials – n-type and p-type. Let us see what happens if we connect them to form a *p-n junction*.

Definition 45

- *An n-type semiconductor has group V impurities such as phosphorus and arsenic. Its primary charge carriers are electrons.*
- *A p-type semiconductor has group III impurities such as boron and gallium. Its primary charge carriers are holes. Holes have an effective positive charge.*
- *A p-n junction is formed when we place a p-type and n-type semiconductor side by side.*

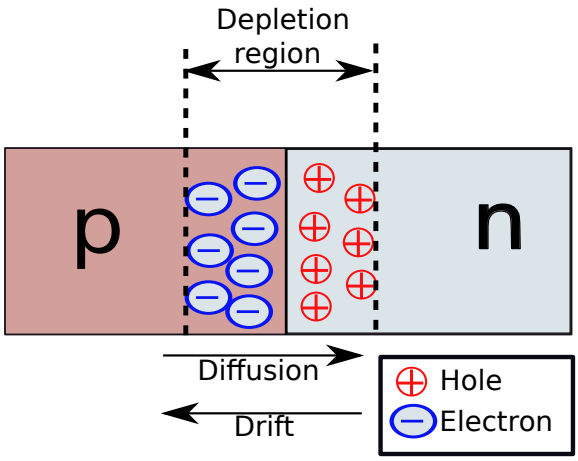

Figure 7.2: A P-N junction

7.1.2 P-N Junction

Let us consider a p-n junction as shown in Figure 7.2. The p-type region has an excess of holes and the n-type region has an excess of electrons. At the junction, some holes cross over and move to the n region because they are attracted by electrons. Similarly, some electrons cross over and get amassed on the side of the p region. This migration of holes and electrons is known as *diffusion*. The area around the junction that witnesses this migration is known as the *depletion region*. However, due to the migration of electrons and holes, an electric field is produced in the opposite direction of migration in the depletion region. This electric field induces a current known as the *drift* current. At steady state, the drift and diffusion currents balance each other, and thus there is effectively no current flow across the junction.

Now, let us see what will happen if we connect both sides of the p-n junction to a voltage source such as a battery. If we connect the p side to the positive terminal, and the n side to the negative terminal, then this configuration is known as *forward bias*. In this case, holes flow from the p side of the junction to the n side, and electrons flow in the reverse direction. The junction thus conducts current.

If we connect the p side to the negative terminal and the n side to the positive terminal, then this configuration is known as *reverse bias*. In this case, holes and electrons are pulled away from the junction. Thus, there is no current flow across the junction and the p-n junction in this case does not conduct electricity.

A simple p-n junction as described is known as a *diode*. It conducts current in only one direction, i.e., when it is in forward bias.

Definition 46
A diode is an electronic device typically made of a single p-n junction that conducts current in only one direction.

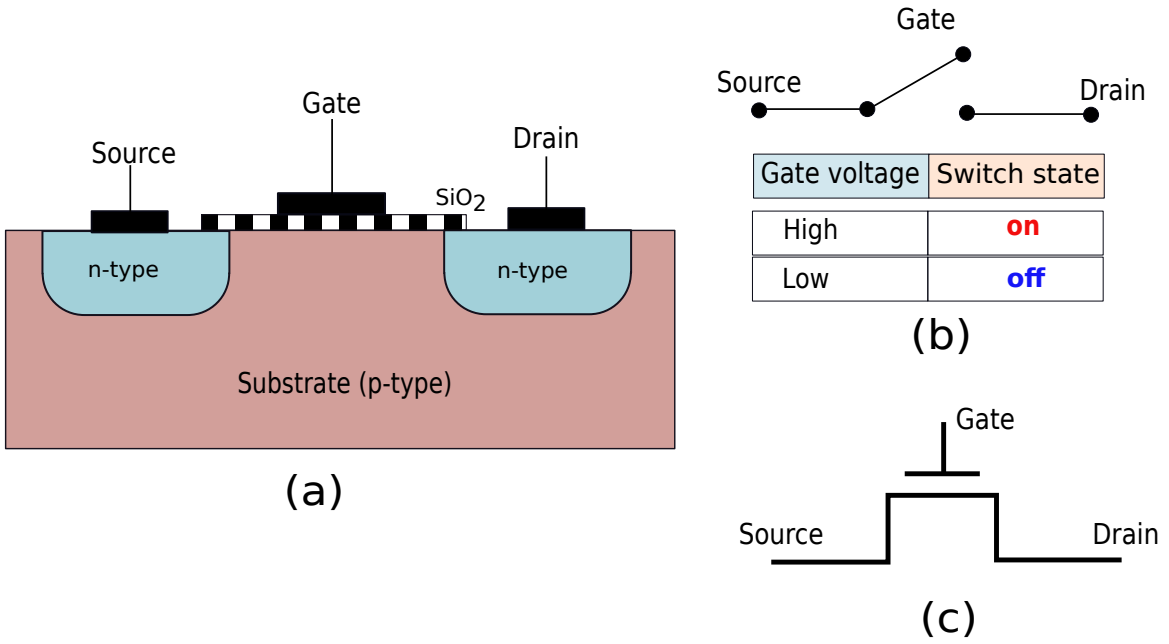

Figure 7.3: NMOS transistor

7.1.3 NMOS Transistor

Now, let us connect two p-n junctions to each other as shown in Figure 7.3(a). This structure is known as an NMOS (Negative Metal-Oxide-Semiconductor) transistor. In this figure there is a central substrate of p type doped silicon. There are two small regions on both sides that contain n type doped silicon. These regions are known as the *drain* and *source* respectively. Note that since the structure is totally symmetric, any of these two regions can be designated as the source or the drain. The region in the middle of the source and drain is known as the *channel*. On top of the channel there is a thin insulating layer typically made of silicon dioxide(SiO_2), and it is covered by a metallic or polysilicon-based conducting layer. This is known as the *gate*.

There are thus three terminals of a typical NMOS transistor – source, drain and gate. Each of them can be connected to a voltage source. We now have two options for the gate voltage – logical 1 (V_{dd} volts) or logical 0 (0 volts). If the voltage at the gate is logical 1 (V_{dd} volts), then the electrons in the channel get attracted towards the gate. In fact, if the voltage at the gate is larger that a certain *threshold voltage* (typically 0.15 V in current technologies), then a low resistance conducting path forms between the drain and the source due to the accumulation of electrons. Thus, current can flow between the drain and the source. If the effective resistance of the channel is $R_{channel}$, then we have $V_{drain} = IR_{channel} + V_{source}$. If the amount of current flow through the transistor is low, then V_{drain} is roughly equal to V_{source} because of the low channel resistance ($R_{channel}$). We can thus treat the NMOS transistor as a switch (see Figure 7.3(b)). It is turned on when the voltage of the gate is 1.

Now, if we set the voltage at the gate to 0, then a conducting path made up of electrons cannot form in the channel. Hence, the transistor will not be able to conduct current. It will

be in the *off* state. In this case, the switch is turned off.

The circuit symbol for a NMOS transistor is shown in Figure 7.3(c).

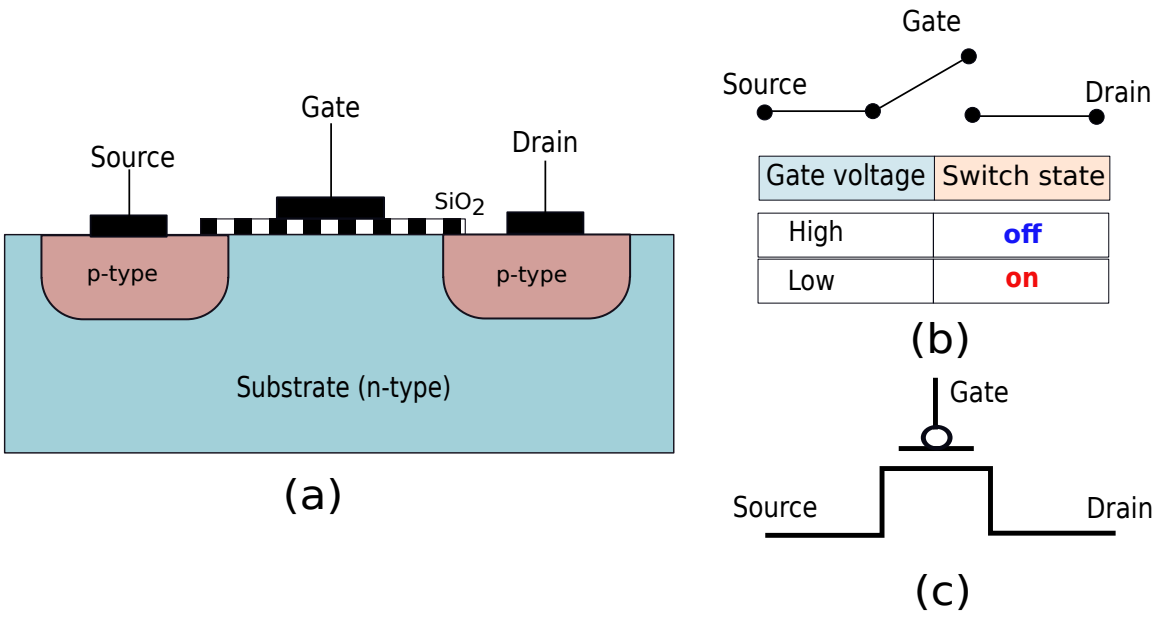

Figure 7.4: PMOS transistor

7.1.4 PMOS Transistor

Like the NMOS transistor, we can have a PMOS transistor as shown in Figure 7.4(a). In this case, the source and drain are regions made up of p type silicon. The logic for the operation of the transistor is exactly the reverse of that of the NMOS transistor. In this case, if the gate is at logical 0, then holes get attracted to the channel and form a conducting path. Whereas, if the gate is at logical 1, then holes get repelled from the channel and do not form a conducting path.

The PMOS transistor can also be treated as a switch (see Figure 7.4(b)). It is turned on, when the gate voltage is 0, and it is turned off when the voltage at the gate is a logical 1. The circuit symbol of a PMOS transistor is shown in Figure 7.4(c).

7.1.5 A Basic CMOS based Inverter

Now, let us construct some basic circuits using NMOS and a PMOS transistors. When a circuit uses both these types of transistors, we say that it uses *CMOS* (combined mos) logic. The circuit diagram of an inverter using CMOS logic is shown in Figure 7.5. In this circuit, an NMOS transistor is connected between the ground and the output, and a PMOS transistor is connected between V_{dd} and the output. The input is fed to the gates of both the transistors.

If the input is a logical 0, then the PMOS transistor is switched on, and the NMOS transistor is switched off. In this case, the output is equal to 1. Likewise, if the input is a logical 1, then

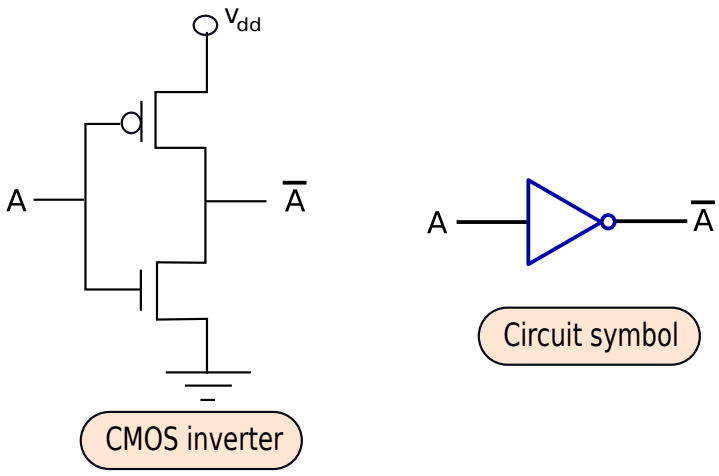

Figure 7.5: CMOS inverter

the NMOS transistor is switched on and the PMOS transistor is switched off. In this case the output is a logical 0. We thus see that this simple circuit inverts the value at the input. It can thus be used to implement the NOT operation.

The benefits of CMOS technology are manifold. Note that during steady state one of the transistors is in the off state. It thus does not conduct any current. A little amount of current can still leak through the transistors or flow through the output terminal. However, this is minimal. Hence, we can conclude that the power dissipation of a CMOS inverter at steady state is vanishingly small since power is equal to current multiplied by voltage. However, some power is dissipated, when the transistor switches its input value. In this case, both the transistors are on for a small amount of time. There is some current flow from V_{dd} to ground. Nonetheless, as compared to competing technologies, the power dissipated by a CMOS inverter is significantly lower and is thus amenable for use by today's processors that have more than a billion transistors.

7.1.6 NAND and NOR Gates

Figure 7.6 shows how to construct a NAND gate in CMOS technology. The two inputs, A and B, are connected to the gates of each NMOS-PMOS pair. If both A and B are equal to 1, then the PMOS transistors will switch off, and both the NMOS transistors will conduct. This will set the output to a logical 0. However, if one of the inputs is equal to 0, then one of the NMOS transistors will turn off and one of the PMOS transistors will turn on. The output will thus get set to a logical 1.

Note that we use the '.' operator for the AND operation. This notation is very widely used in representing Boolean formulae. Likewise for the OR operation, we use the '+' sign.

Figure 7.7 shows how to construct a NOR gate. In this case, the two inputs, A and B, are also connected to the gates of each NMOS-PMOS pair. However, as compared to the NAND gate, the topology is different. If one of the inputs is a logical 1, then one of the NMOS

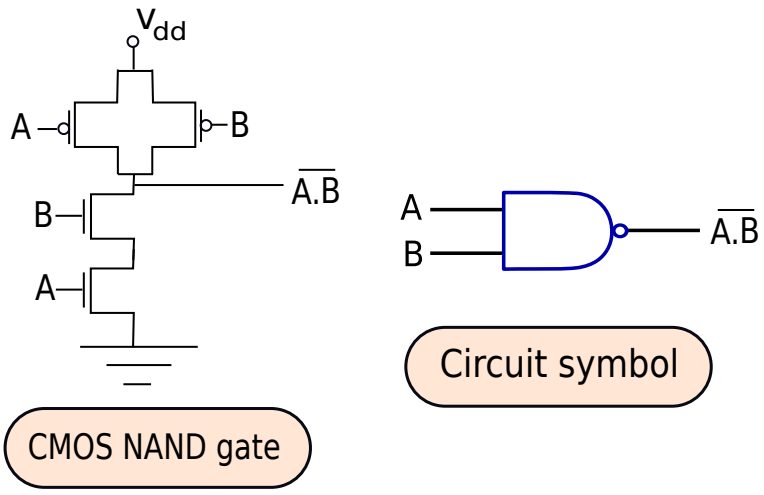

Figure 7.6: CMOS NAND gate

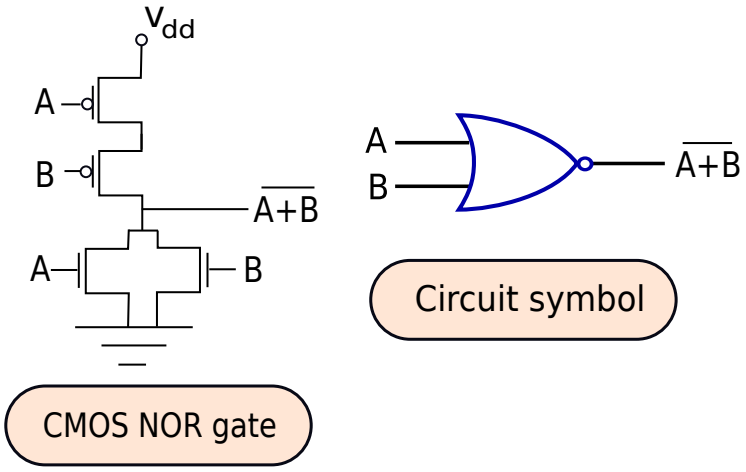

Figure 7.7: CMOS NOR gate

transistors will turn on and one of the PMOS transistors will turn off. The output will thus get set to 0. If both the inputs are equal to 0, then both the NMOS transistors will shut off, and both the PMOS transistors will turn on. The output in this case will be equal to a logical 1.

Now, that we have constructed a basic inverter, a NAND, and NOR gate using CMOS logic; we have the tools to construct any type of logic gate. This is because NAND and NOR gates are known as universal gates (see [Kohavi and Jha, 2009]). They can be used to construct any kind of logic gate and implement any logic function. In our circuits, we shall implement complex logic gates using AND, OR, NAND, NOR, XOR, and NOT gates. Other than AND and OR gates, we have described the construction of the rest of the four gates in this section. We can

construct an AND gate by connecting the output of a NAND gate to a NOT gate. Similarly, we can construct an OR gate by connecting the output of a NOR gate to a NOT gate.

In the next section, we shall look at structures that compute complex functions on a set of Boolean input bits. We call such structures *combinational logic* structures because they decide if a certain set of input boolean bits belong to a set containing restricted combinations of bits. For example, a XOR gate produces 1 if the input bits are either 01 or 10. In this case the set $\mathcal{S}$ contains the combinations: $\{01, 10\}$. A XOR logic structure decides if the two input bits are in the set $\mathcal{S}$. If they are in the set, then it produces an output equal to 1, otherwise it produces 0.

7.2 Combinational Logic

7.2.1 XOR Gate

Let us implement the logic function for exclusive or (XOR). The truth table is shown in Table 7.1. We shall use the $\oplus$ operator for the XOR operation. An exclusive or operation returns a 1 if both the inputs are unequal, otherwise it returns a 0.

A	B	A $\oplus$ B
0	0	0
1	0	1
0	1	1
1	1	0

Table 7.1: The XOR operation

We observe that $A \oplus B = A.\overline{B} + \overline{A}.B$. The circuit for implementing a XOR gate is shown in Figure 7.8.

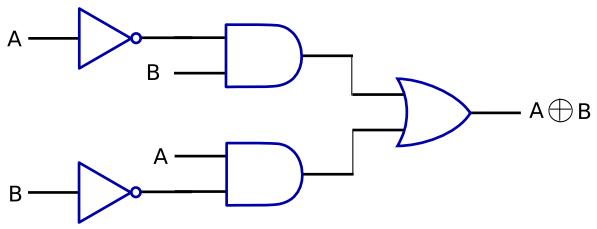

Figure 7.8: XOR gate

7.2.2 Decoder

A decoder takes as input a $log(n)$-bit binary number and has n outputs. Based on the input it sets one of the outputs to 1.

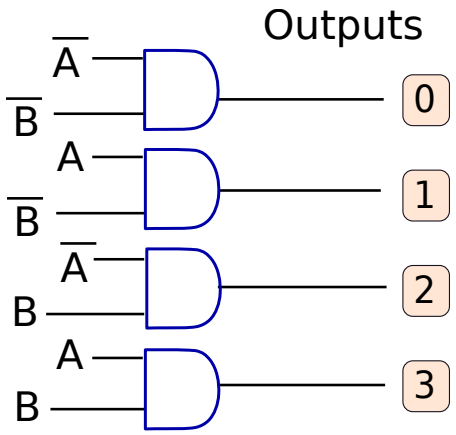

Figure 7.9: Design of a 2×4 decoder

Let us assume that n is a power of 2. Now any value from 0 to $n-1$ can be encoded using $log(n)$ bits. Let us thus treat the input as a Boolean representation of one of the outputs $(0 \ldots (n-1))$. For example, if the value of the $log(n)$-bit input is equal to i, then the i^{th} output is set to 1. The rest of the outputs are set to 0. Decoders are extensively used in the design of memory cells and other combinational elements such as multiplexers and demultiplexers.

The design of the decoder is shown in Figure 7.9. We show the design of a 2×4 decoder that has two inputs and four outputs. Let the inputs be A and B. We generate all possible combinations: $\overline{A}\overline{B}$, $\overline{A}B$, $A\overline{B}$, and AB. These Boolean combinations are generated by computing a logical NOT of A and B and then routing the values to a set of AND gates.

Let us now explain with an example. Assume that the input is equal to 10. This means that $B = 1$ and $A = 0$. We need to set the 2^{nd} output line to 1, and the rest to 0. The reader can verify that this is indeed happening (note that we are counting from 0). The AND gate corresponding to the 2^{nd} output needs to compute $\overline{A}B$. In this case, it will be the only condition that evaluates to 1.

On similar lines, we can create a generic $log(n) \times n$ decoder.

7.2.3 Multiplexer

The block diagram of a multiplexer is shown in Figure 7.10. It takes n input bits and $log(n)$ select bits, and based on the value of the select bits, chooses one input as the output (refer to the line with arrows in Figure 7.10). Multiplexers are heavily used in the design of processors, where we need to choose one output out of a set of inputs. A multiplexer is also known as a *mux*.

To choose 1 input out of n inputs, we need to specify the identifier of the input. Note that it takes $\lceil log(n) \rceil$ bits to identify any input uniquely (see Section 4.4). We can number each input using $\lceil log(n) \rceil$ binary bits. Each input thus has a unique representation. Now, if the select bits match the binary encoding of an input, then the input gets reflected at the output. For example, if the value of the select bits is i, then the value of the output is equal to the value of the i^{th} input.

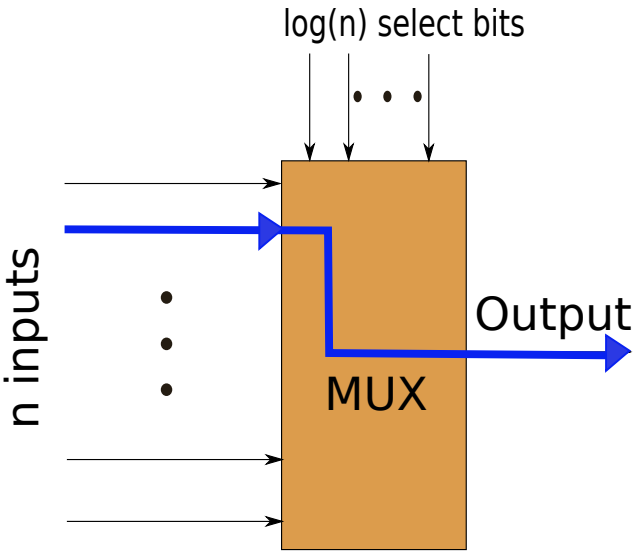

Figure 7.10: Block diagram of a multiplexer

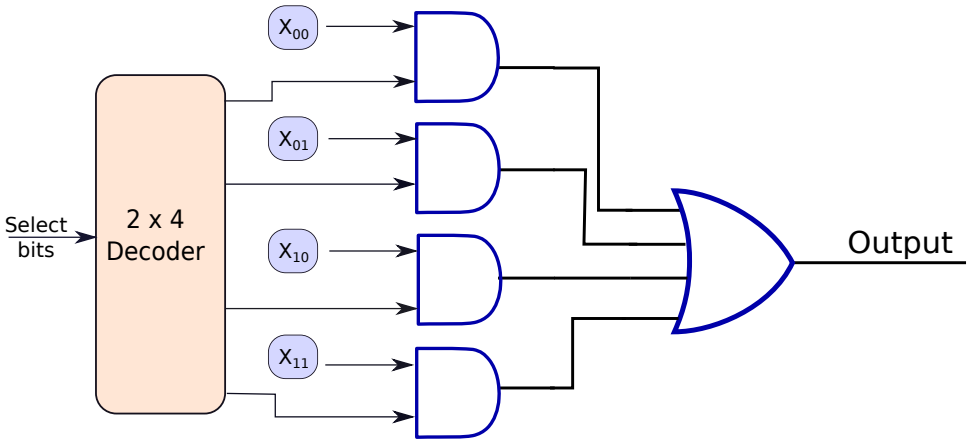

Figure 7.11: Design of a 4-input multiplexer

A multiplexer consists of three stages (see Figure 7.11). The first stage is a decoder that takes the $log(n)$ select bits as its input and sets one of the n output lines to a logical 1. Each output line is connected to an AND gate (second stage). Since only one of the output lines is set to 1, only the AND gate corresponding to that output is *enabled*. This means that the output of this gate is equal to the value of the other input.

In the example in Figure 7.11, the multiplexer has four inputs: X_{00}, X_{01}, X_{10}, and X_{11}. Each input is connected to an AND gate. If the select bits are equal to 01, then the AND gate corresponding to the input X_{01} is enabled by the decoder. Its output is equal to X_{01}. The outputs of the rest of the AND gates are a logical 0 because they are not enabled by the

decoder: one of their inputs is a logical 0.

Finally, in the third stage, an OR gate computes the logical OR of all the outputs of the second stage. Note that for the OR gate, the inputs are $n-1$ zeros and X_{BA}, where B and A are the values of the select bits, respectively. The final output is thus the value of the input that was selected, X_{BA}.

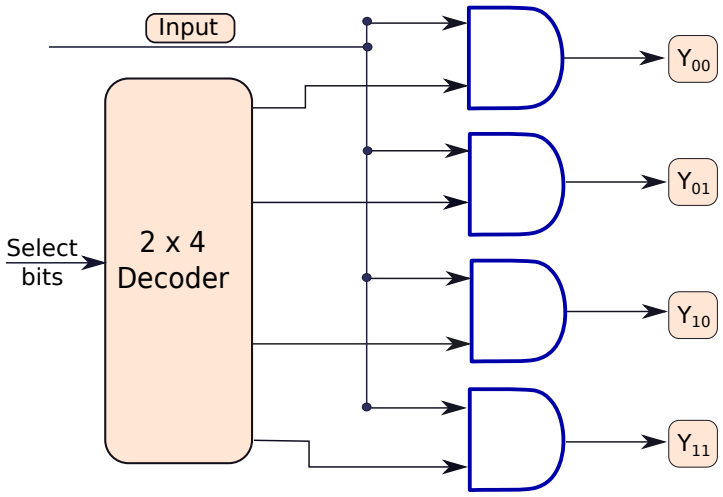

Figure 7.12: Design of a demultiplexer

7.2.4 Demultiplexer

A demultiplexer takes as input a $log(n)$-bit binary number, a 1-bit input, and transfers the input to one of n output lines. The block diagram is shown in Figure 7.12. Demultiplexers are used in the design of memory cells, where it is necessary for the input to reflect in exactly one of the output lines.

The operation is similar to that of a multiplexer. Instead of having multiple inputs, we have just 1 input. Like the case of the multiplexer, we first enable the appropriate AND gate with a decoder. Then the second stage consists of an array of AND gates, where each gate reflects the input bit at its output if it is *enabled* by the decoder. Recall that an AND gate is *enabled* if one of its inputs is a logical 1; in this case the output is equal to the value of the other input.

7.2.5 Encoder

Let us now consider a circuit with the reverse logic as that of a decoder. Its block diagram is shown in Figure 7.13. The circuit has n inputs, and $log(n)$ outputs. One of the n inputs is assumed to be 1, and the rest are assumed to be 0. The output bits provide the binary encoding of the input that is equal to 1. For example in an 8-input, 3-output encoder, if the fifth line is equal to 1, then the output is equal to 100 (count starts from 0).

Let us construct a simple 4-2 encoder (4 inputs, 2 output bits). Let us number the bits X_0, X_1, X_2, and X_3. If bit X_A is equal to 1, then the output should be equal to the binary encoding

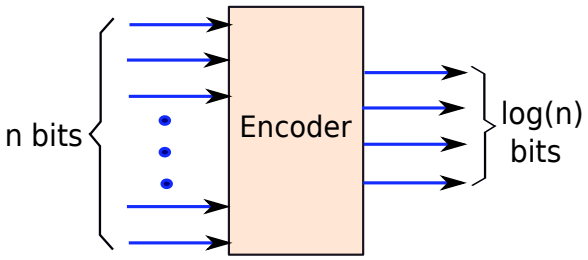

Figure 7.13: Block diagram of an n bit encoder

of A. Let us designate the output as Y, with two bits Y_0, and Y_1. We have the following equations for Y_0, and Y_1.

$$Y_0 = X_1 + X_3 \tag{7.1}$$
$$Y_1 = X_2 + X_3 \tag{7.2}$$
$$\tag{7.3}$$

The intuition behind these equations is that the LSB of a 2-bit number is equal to 1, when it is equal to 01(1), or 11(3). The MSB is equal to 1, when it is equal to 10(2), or 11(3). We can extend this logic to create a generic (n log(n))-bit encoder. The circuit diagram of a 4-2-bit encoder is shown in Figure 7.14.

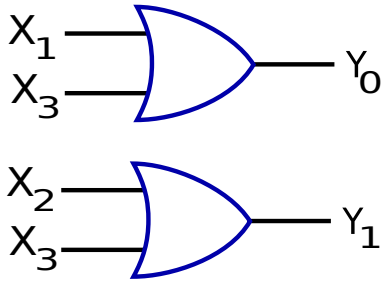

Figure 7.14: Circuit diagram of a 4-2-bit encoder

Example 119
Write the equations for an 8-3-bit encoder. Assume that the inputs are $X_0 \ldots X_7$, and the outputs are Y_0, Y_1, and Y_2.

Answer:

$$Y_0 = X_1 + X_3 + X_5 + X_7 \tag{7.4}$$
$$Y_1 = X_2 + X_3 + X_6 + X_7 \tag{7.5}$$
$$Y_3 = X_4 + X_5 + X_6 + X_7 \tag{7.6}$$

7.2.6 Priority Encoder

Let us now assume that we do not have the restriction that only one input line can be equal to 1. Let us assume that more than one inputs can be equal to 1. In this case, we need to report the binary encoding of the input line that has the highest index (priority). For example, if lines 3 and 5 and on, then we need to report the binary encoding of the 5^{th} line. The block diagram remains the same as Figure 7.13.

However, the equations for computing the output change. For a 4-2-bit priority encoder, the equations are as follows.

$$Y_0 = X_1.\overline{X_2} + X_3 \tag{7.7}$$
$$Y_1 = X_2 + X_3 \tag{7.8}$$

Let us consider Y_0. If $X_3 = 1$, then $Y_0 = 1$, because X_3 has the highest priority. However, if $X_1 = 1$, then we cannot take a decision based on its value, because X_2, and X_3 might also be equal to 1. If $X_3 = 1$, then there is no issue, because it also sets the value of Y_0. However, if $X_3 = 0$, and $X_2 = 1$, then we need to disregard X_1. Hence, we need to compute $X_1.\overline{X_2} + X_3$ for Y_0. The equation for Y_1 remains the same (the reader should try to find the reason). The circuit diagram of a 4-2-bit encoder is shown in Figure 7.15.

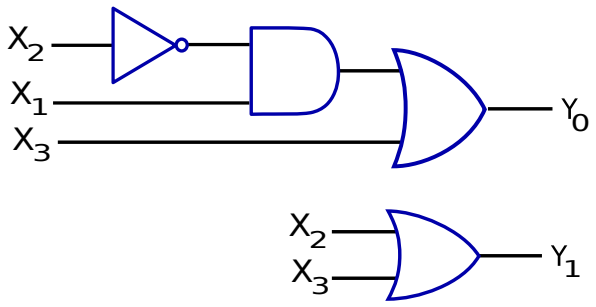

Figure 7.15: Circuit diagram of a 4-2-bit priority encoder

Example 120
Write the equations for an 8-3-bit priority encoder. Assume that the inputs are $X_0 \dots X_7$, and the outputs are Y_0, Y_1, and Y_2.
Answer:

$$Y_0 = X_1.\overline{X_2}.\overline{X_4}.\overline{X_6} + X_3.\overline{X_4}.\overline{X_6} + X_5.\overline{X_6} + X_7 \quad (7.9)$$

$$Y_1 = X_2.\overline{X_4}.\overline{X_5} + X_3.\overline{X_4}.\overline{X_5} + X_6 + X_7 \quad (7.10)$$

$$Y_3 = X_4 + X_5 + X_6 + X_7 \quad (7.11)$$

7.3 Sequential Logic

We have looked at combinational logic circuits that compute different functions on bits. In this section, we shall look at saving bits for later use. These structures are known as sequential logic elements because the output is dependent on past inputs, which came earlier in the *sequence* of events. The basic idea in a logic gate was to modify the input values to get the desired outputs. In a combinational logic circuit, if the inputs are set to 0, then the outputs also get reset. To ensure that a circuit stores a value and maintains it for as long as the processor is powered on, we need to design a different kind of circuit that has some kind of "built-in memory". Let us start with formulating a set of requirements.

1. The circuit should be self-sustaining, and should maintain its values after the external inputs are reset. It should not rely on external signals to maintain its stored elements.
2. There should be a method to read the stored value without destroying it.
3. There should be a method to set the stored value to either 0 or 1.

The best way to ensure that a circuit maintains its value is to create a feedback path, and connect the output back to the input. Let us take a look at the simplest logic circuit in this space: an SR latch.

7.3.1 SR Latch

Figure 7.16 shows the SR latch. There are two inputs S(set) and R (reset). There are two outputs Q, and its complement $\overline{Q}$. Let us now analyze this circuit that contains two cross-coupled NAND gates. Note that if one of the inputs of a NAND gate is 0, then the output is guaranteed to be 1. However, if one of the inputs is 1, and the other input is A, then the output is $\overline{A}$.

Let us consider the case when, S=1 and R=0. One of the inputs($\overline{S}$) to the top NAND gate is 0. Thus, Q=1. The bottom NAND gate has two inputs $\overline{R} = 1$, and $Q = 1$. Thus, the output, $\overline{Q} = 0$. Similarly, if S=0 and R=1, then Q=0, and $\overline{Q} = 1$. The S input sets the bit in the latch, and the R bit resets it to 0. Let us now consider the case when both S and R are 0. In this case one of the inputs to both the NAND gates is 1. The top NAND gate's output is $\overline{\overline{Q}} = Q$, and

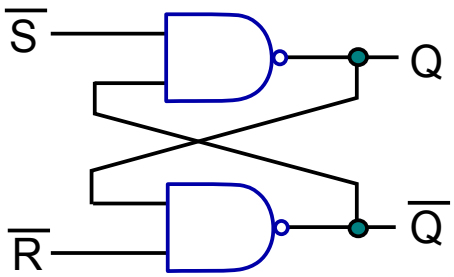

Figure 7.16: The SR latch

the bottom NAND gate's output is $\overline{Q}$. Thus, the value is maintained, and we have effectively achieved the objective of storing a bit.

Now, let us see what happens if we set both S and R to 1. In this case, $\overline{S} = 0$ and $\overline{R} = 0$. Thus, Q and $\overline{Q}$ are both equal to 1. In this case, $\overline{Q}$ is not the logical complement of Q. Now, let us say that S is set to 0. Then Q will become 0, and $\overline{Q}$ will become 1. Likewise, if R is set to 0, then Q will become 1, and $\overline{Q}$ will become 0. However, if both S and R simultaneously become 0, then we cannot predict the state of the latch before hand. This is because in practice, signal transitions are never perfectly simultaneous. A non-zero time lag between the transitions of both the inputs is almost always there. Hence, the circuit can see the following sequence of transitions in the SR bits: $11 \rightarrow 10 \rightarrow 00$, or $11 \rightarrow 01 \rightarrow 00$. For the former sequence, Q will be set to 1, and for the latter sequence Q will be set to 0. This is known as a *race* condition and causes unpredictable behavior. Thus, we do not want to set both S and R to 1.

S	R	Q	$\overline{Q}$	Action
0	0	Q_{old}	$\overline{Q}_{old}$	maintain
1	0	1	0	set
0	1	0	1	reset
1	1	?	?	indeterminate

Table 7.2: State transition table for an SR latch

Table 7.2 shows the state transition table for the SR latch. Q_{old} and $\overline{Q}_{old}$ are the old values of Q and $\overline{Q}$ respectively. The main feature is that setting SR=00 maintains the value stored in the latch. During this period, we can read the value of the outputs infinitely often.

The main issues with the SR latch are as follows.

- S=1 and R=1 is an invalid input.
- We do not have a method of synchronizing the transitions in the input and the output. Whenever the inputs change, the outputs also change. As we shall see in the next section, this is not desired behavior.

7.3.2 The Clock

A typical processor contains millions or possibly billions of logic gates and thousands of latches. Different circuits take different amounts of time. For example, a multiplexer might take 1 ns, and a decoder might take 0.5 ns. Now, a circuit is ready to forward its outputs when it has finished computing them. If we do not have a notion of global time, it is difficult to synchronize the communication across different units, especially those that have variable latencies. Under such a scenario it is difficult to design, operate, and verify a processor. We need a notion of time. For example, we should be able to say that an adder takes two units of time. At the end of the two units, the data is expected to be found in latch X. Other units can then pick up the value from latch X after two units of time and proceed with their computation.

Let us consider the example of a processor that needs to send some data to the printer. Let us further assume that to communicate data, the processor sends a series of bits over a set of copper wires, the printer reads them, and then prints the data. The question is, when does the processor send the data? It needs to send the data, when it is done with the computation. The next question that we can ask is, "How does the processor know, when the computation is over?" It needs to know the exact delays of different units, and once the total duration of the computation has elapsed, it can write the output data to a latch and also set the voltages of the copper wires used for communication. Consequently, the processor does need a notion of time. Secondly, the designers need to tell the processor the time that different sub-units take. Instead of dealing with numbers like 2.34 ns, and 1.92 ns, it is much simpler to deal with integers such as 1, 2, and 3. Here, 1, 2, and 3, represent units of time. A unit of time can be any number such as 0.9333 ns.

Definition 47

clock signal *A periodic square wave that is sent to every part of a large circuit or processor.*

clock cycle *The period of a clock signal.*

clock frequency *The inverse of the clock cycle period.*

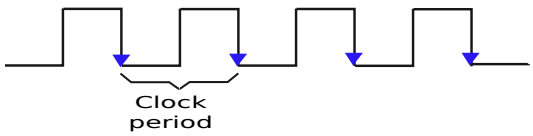

Figure 7.17: A Clock Signal

Hence, most digital circuits synchronize themselves with a *clock signal* that sends a periodic pulse to every part of the processor at exactly the same time. A clock signal is a square wave as shown in Figure 7.17; most of the time the clock signal is generated externally by a dedicated

unit on the motherboard. Let us consider the point at which the clock signal transitions from 1 to 0 (downward/negative edge) as the beginning of a *clock cycle*. A clock cycle is measured from one downward edge of the clock to the next downward edge. The duration of a clock cycle is also known as a *clock period*. The inverse of a clock period is known as the *clock frequency*.

Important Point 8
A computer, laptop, tablet, or mobile phone typically has a line listing the frequency in its specifications. For example, the specifications might say that a processor runs at 3 GHz. Now we know that this number refers to the clock frequency*.*

The typical model of computation is as follows. The time required to perform all basic actions in a circuit is measured in terms of clock cycles. If a producer unit takes n clock cycles, then at the end of n clock cycles, it writes its value to a latch. Other consumer units are aware of this delay, and at the beginning of the $(n+1)^{th}$ clock cycle, they read the value from the latch. Since all the units explicitly synchronize with the clock, and the processor is aware of the delays of every unit, it is very easy to sequence the computation, communicate with I/O devices, avoid race conditions, debug and verify circuits. We can see that our simple example in which we wanted to send data to a printer can be easily solved by using a clock.

7.3.3 Clocked SR Latch

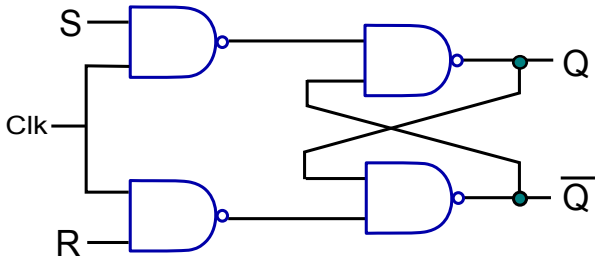

Figure 7.18: The Clocked SR latch

Figure 7.18 shows the SR latch augmented with two NAND gates that have the clock as one of the inputs. The other two inputs are the S and R bits respectively. If the clock is 0, then both the inputs to the cross-coupled NAND gates are 1. This maintains the previous value. If the clock is 1, then the inputs to the cross-coupled NAND gates are $\overline{S}$ and $\overline{R}$ respectively. These are the same inputs as the basic SR latch. The rest of the operation follows Table 7.2. Note that a clocked latch is typically referred to as a *flip-flop*.

Definition 48
Flip-flop : It is a clocked latch that can save a bit (0 or 1).

By using the clock, we have partially solved the problem of synchronizing inputs and outputs. In this case, when the clock is 0, the outputs are unaffected by the inputs. When the clock is 1, the outputs are affected by the input. Such a latch is also called a *level sensitive latch.*

Definition 49
A level sensitive latch is dependent on the value of the clock signal – 0 or 1. Typically, it can read in new values, only when the clock is 1.

In a level sensitive latch, circuits have half a clock cycle to compute the correct outputs (when the clock is 0). When the clock is 1, the outputs are visible. It would be better to have one full clock cycle to compute the outputs. This would require an edge sensitive latch. An *edge sensitive* latch reflects the inputs at the output, only at the downward edge of the clock.

Definition 50
An edge sensitive latch reflects the inputs at the output only at a fixed clock edge, such as the downward edge (transition from 1 to 0).

Let us try to create an edge sensitive SR latch.

7.3.4 Edge Sensitive SR Flip-flop

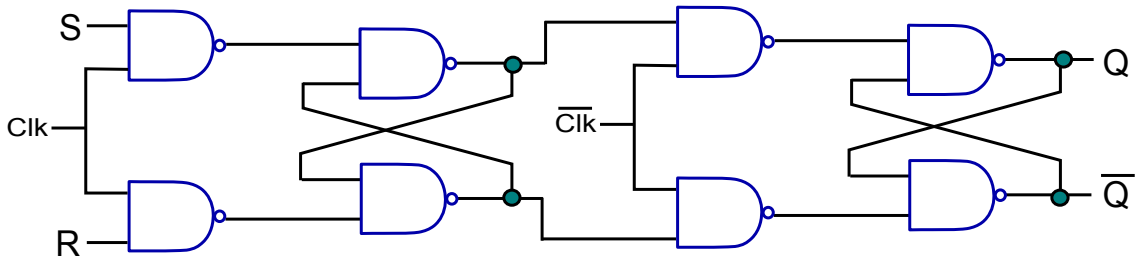

Figure 7.19: The clocked edge sensitive SR flip-flop

Figure 7.19 shows a clocked edge sensitive SR flip-flop. We connect two clocked SR flip-flops together. The only difference is that the second SR flip-flop uses the complement of the clock signal, $\overline{CLK}$. The first flip-flop is known as the *master*, and the second flip-flop is known as the *slave*. This flip-flop is also known as a master-slave SR flip-flop. Here, is how this circuit works.

Definition 51
A master-slave flip-flop contains two flip-flops that are connected to each other. The master's output is the slave's input. Typically, the slave uses a clock signal that is the logical complement of the master's clock.

Whenever the clock signal is high (1), the inputs (S and R) are read into the first SR flip-flop. When the clock signal becomes low (0), then the first flip-flop stops accepting new data; however, the second flip-flop takes the output of the first flip-flop and sets its output accordingly. Thus, new data arrives at the output terminals Q and $\overline{Q}$ when the clock transitions from 1 to 0 (downward clock edge). We thus have created a flip-flop that is edge sensitive.

However, some problems still remain. If both S and R are 1, then there might be a race condition, and the output can be unpredictable. This problem needs to be fixed. Let us first try to look at a complex solution that augments the clocked edge sensitive SR flip-flop, and then look at simpler solutions.

7.3.5 JK Flip-flop

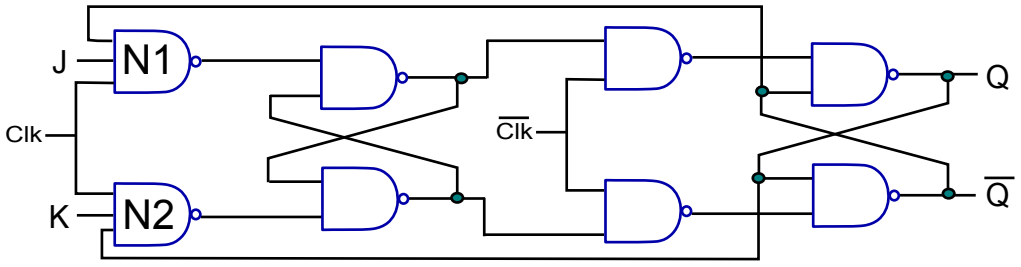

Figure 7.20: The JK flip-flop

Figure 7.20 shows a JK flip-flop. There are two minor differences as compared to the master-slave SR flip-flop. The first is that the inputs are now J and K, instead of S and R. The second is that Q and $\overline{Q}$ are now inputs to the input NAND gates ($N1$ and $N2$).

Let us do a case by case analysis. Assume that the clock is high. If J and K are both 0, then the outputs of $N1$ and $N2$ are both 1 and the case is same as that for the master-slave SR flip-flop. The outputs are maintained. If J=1 and K=0, then we need to consider the value of $\overline{Q}$. If $\overline{Q} = 1$, then Q = 0, and the output of NAND gate N1 (see Figure 7.20) is 0. The outputs of the master flip-flop are therefore 1 and 0 respectively. The output of the slave after the downward/ negative clock edge will therefore be: $Q = 1, \overline{Q} = 0$.

Now, assume that $\overline{Q} = 0$, and Q=1. In this case, the outputs of both N1 and N2 are 1 and thus all the values are maintained. Hence, after the negative clock edge we have: Q=1, and $\overline{Q} = 0$. We can thus conclude, that if J=1 and K=0, Q=1, and $\overline{Q}$=0.

Similarly, if J=0 and K=1, we can prove that Q=0, and $\overline{Q}$=1.

Let us now consider the case when both J and K are 1. In this case the output of N1 is Q and the output of N2 is $\overline{Q}$. The output of the master flip-flop is equal to $\overline{Q}$ and Q respectively. After the negative clock edge the outputs will be: Q = $\overline{Q}_{old}$ and $\overline{Q}$ = Q_{old}. Thus, the outputs will get toggled. We will not have a race condition anymore. Table 7.3 shows the state transition table for the JK flip-flop.

J	K	Q	$\overline{Q}$	Action
0	0	Q_{old}	$\overline{Q}_{old}$	maintain
1	0	1	0	set
0	1	0	1	reset
1	1	$\overline{Q}_{old}$	Q_{old}	toggle

Table 7.3: State transition table for a JK flip-flop

7.3.6 D Flip-flop

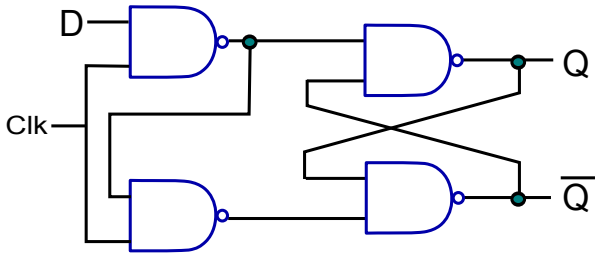

Figure 7.21: D flip-flop

Instead of having a dedicated S(set) and R (reset) signal, we can make our life easy by making one the complement of the other. However, in this case, we will not have a method of maintaining the value. The input will get reflected at the output at every negative clock edge. In a lot of cases, this is sufficient, and we do not need dedicated logic to either maintain or toggle the values. In this case, we can use the simplistic D flip-flop as shown in Figure 7.21. It is basically a SR flip-flop where R = $\overline{S}$.

Note that the second input (to the lower NAND gate) is equal to $\overline{D \wedge Clk}$. When Clk is equal to 1, the second input is equal to $\overline{D}$. When Clk is 0, the flip-flop maintains the previous values and does not accept new data.

7.3.7 Master-slave D Flip-flop

Akin to the JK flip-flop we can have a master-slave version of the D flip-flop.

Figure 7.22 shows a master-slave version of the D flip-flop. We connect one D flip-flop to a SR flip-flop. Here, we do not need wires connecting the inputs with Q and $\overline{Q}$ because we are not interested in toggling the state. Secondly, we have avoided race conditions by not having the evil (1,1) input.

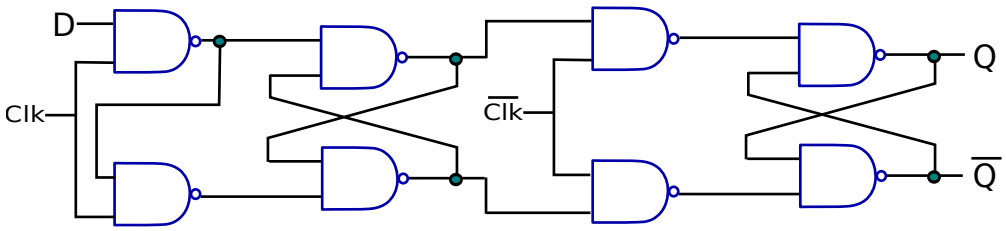

Figure 7.22: Master-slave D flip-flop

A master-slave flip-flop uses 8 NAND gates and 1 inverter for the clock. A master slave D flip-flop requires 34 transistors, and a master-slave JK flip-flop requires 38 transistors. This is a large expenditure for saving just 1 bit! We should be able to do better. We shall see in later sections that we can store a single bit with even 1 transistor. But the circuit will become extremely slow. We can thus conclude that flip-flops are inefficient as far as power and the transistor budget are concerned; however, they are very fast. If we need to save a lot of data, and we are willing to sacrifice on time, then we should opt for SRAM and DRAM memories (described in Section 7.4).

7.3.8 Metastability

Up till now we have assumed that at the negative edge of a clock the input instantaneously reflects at the output. This high level assumption is however not strictly correct. Readers need to appreciate the fact that every digital circuit is at its core an analog circuit. Quantities like current and voltage take time to reach their optimal levels, and the circuit is sensitive to the voltage levels, and the timing of the inputs. The readers can refer to standard text books [Taub and Schilling, 1977] on digital design for a thorough explanation.

In this section, we shall take a look at one particular aspect of the analog nature of flip flops known as *metastability*. If there is a change in the input close to the negative clock edge, then the output becomes non-deterministic, and might even fluctuate or oscillate for some period of time. This phenomenon is known as *metastability*. To avoid such behavior it is necessary to ensure that the input is stable (does not change) for t_{setup} units of time before the clock edge, and is also stable for t_{hold} units of time after the clock edge. This means that there is a window of time around the clock edge in which the input to the flip flop needs to remain stable. Only, in this case, we can guarantee the correct operation of a flip flop. This window of time in which we want the inputs to be stable is known as the *keep out* region.

Let us now display these concepts graphically in Figure 7.23. We need to note that in practice, the setup time, and the hold time are small fractions of the total cycle time ($< 10\%$). Designers take special precautions to ensure that there are no transitions in the input in the keep out region. We shall see that this phenomenon has important implications when we discuss I/O circuits. These circuits have sophisticated delay elements that delay signals to keep transitions out of the keep out region.

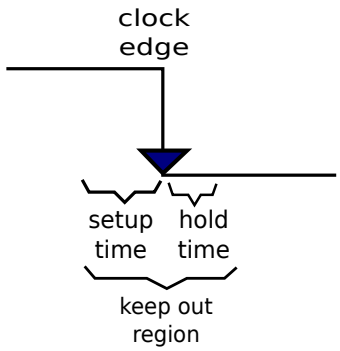

Figure 7.23: Setup time, hold time, and the keep out region

7.3.9 Registers

Parallel In–Parallel Out

We can store n bit data by using a set of n master slave D flip flops. Each D flip flop is connected to an input line, and its output terminal is connected to an output line. Such an n bit structure is known as an n bit register. Here, we can load n bits in parallel, and also read out n bits in parallel at every negative clock edge. Hence, this structure is known as a parallel in–parallel out register. Its structure is shown in Figure 7.24.

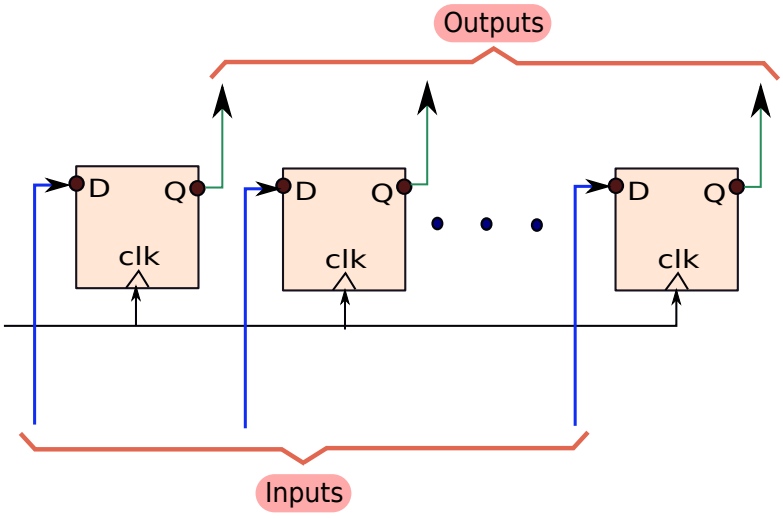

Figure 7.24: A parallel in–parallel out register

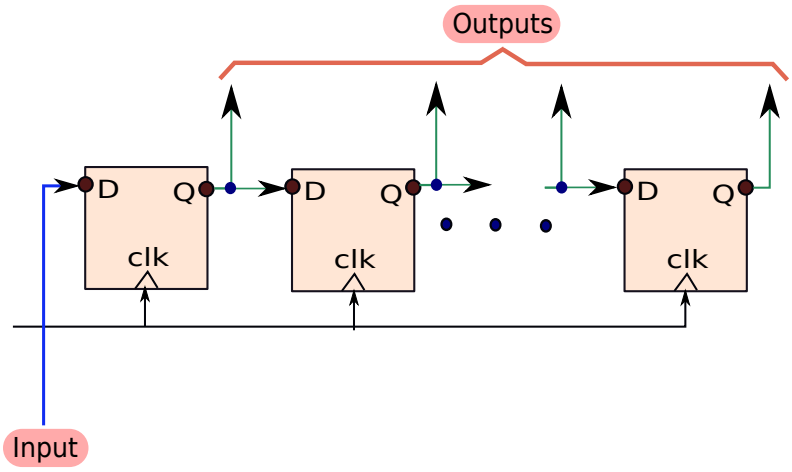

Figure 7.25: A serial in–parallel out register

Serial In – Parallel Out

Let us now consider a serial in–parallel out register as shown in Figure 7.25. Here, we have a single input that is fed into the leftmost D flip flop. Every cycle, the input moves to the adjacent flip flop on the right. Thus, to load n bits it will take n cycles. The first bit will get loaded into the leftmost flip flop in the first cycle, and it will take n cycles for it to reach the last flip flop. By that time, the rest of the $n-1$ flip flops will get loaded with the rest of the $n-1$ bits. We can then read out all the n bits in parallel (similar to the parallel in–parallel out register). This register is also known as a *shift register* and is used for implementing circuits used in high speed I/O buses.

7.4 Memories

7.4.1 Static RAM (SRAM)

SRAM Cell

SRAM refers to static random access memory. A basic SRAM cell contains two cross-coupled inverters as shown in Figure 7.26. In comparison, a basic SR flip-flop or a D flip-flop contains cross-coupled NAND gates. The design is shown in Figure 7.26.

The core of the SRAM cell contains 4 transistors (2 in each inverter). This cross-coupled arrangement is sufficient to save a single bit (0 or 1). However, we need some additional circuitry to read and write values. At this point, the reader might be wondering if it is a bad idea to have cross-coupled inverters in a latch. They after all require fewer transistors. We shall see that the overheads of implementing the circuitry for reading and writing an SRAM cell are non-trivial. The overheads do not justify making a latch with an SRAM cell as its core.

The cross coupled inverters are connected to transistors on each side (W1, W2). The gates

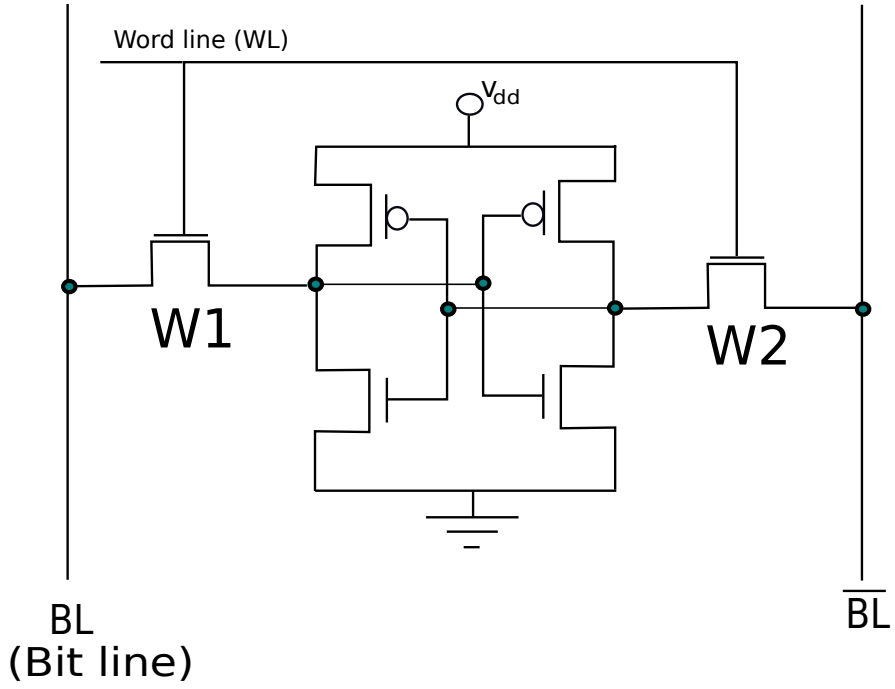

Figure 7.26: A 6 transistor SRAM cell

of W1 and W2 are connected to the same signal, known as the *word line*. The four transistors in the two inverters, W1, and W2, comprise the SRAM cell. It has six transistors in total. Now, if the voltage on the word line is low, then W1 and W2 are off. It is not possible to read or write the SRAM cell. However, if the signal on the word line is high, then W1 and W2 are on. It is possible to access the SRAM cell.

Now, the transistors, $W1$, and $W2$, are connected to copper wires on either side known as the bit lines. The bit lines are designed to carry complementary values. One of them is BL and the other is $\overline{BL}$. To write a value into the cell it is necessary to set the values of BL and $\overline{BL}$ to A and $\overline{A}$ respectively, where A is the value that we intend to write. To read a value, we need to turn the word line on and read the voltages of the bit lines.

Let us now delve slightly deeper into the operation of the SRAM cells. Note that SRAM cells are not solitary units like latches. They exist as a part of an array of SRAM cells. We need to consider the array of SRAM cells in entirety.

Array of SRAM Cells

Figure 7.27 shows a typical SRAM array. SRAM cells are laid out as a two dimensional matrix. All the SRAM cells in a row share the word line, and all SRAM cells in a column share a pair of bit lines. To activate a certain SRAM cell it is necessary to turn its associated word line on. This is done by a decoder. It takes a subset of address bits, and turns the appropriate word line on. A single row of SRAM cells might contain 100+ SRAM cells. Typically, we will

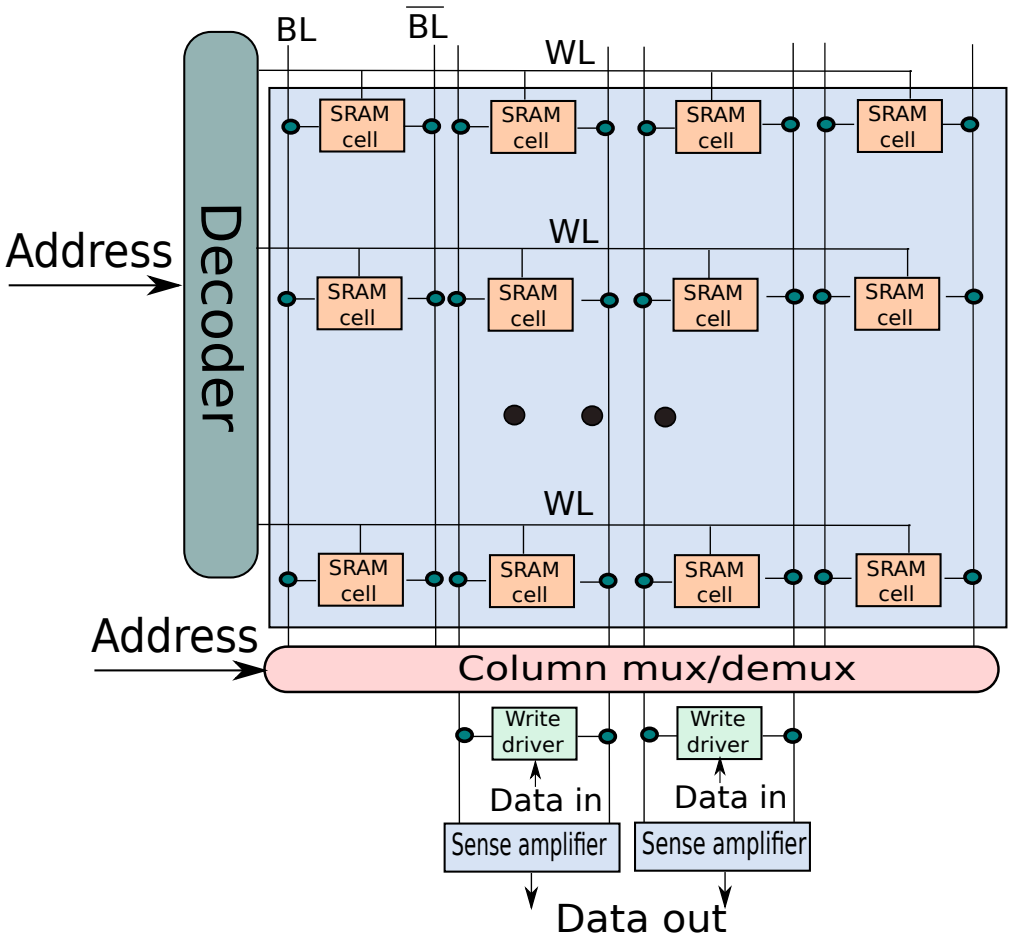

Figure 7.27: Array of SRAM Cells

be interested in the values of 32 SRAM cells (on a 32-bit machine). In this case the column mux/demux selects the bit lines belonging to the SRAM cells of interest. It uses a subset of the bits in the address as the column select bits. This design methodology is also known as 2.5D memory organization.

As the size of the array grows it may become more asymmetric. This needs to be avoided, otherwise the capacitive loading on the word lines or bit lines will become prohibitive. Hence, columns need to become wider and the column mux/demux structure needs to be driven by a large column decoder.

The process of writing is easy. The strong write drivers need to set the values of BL and $\overline{BL}$. To write 1, BL needs to be driven to 1, and $\overline{BL}$ needs to be driven to 0. However, reading a value is slightly more difficult. The reason is that an SRAM cell needs to charge an entire bit line to the stored value such that it can be read. Since hundreds of SRAM cells are typically connected to a bit line, the bit line has a very high capacitance. Consequently, it will take a long time to charge/discharge the bit line to logical 0 or 1.

Hence, something smarter needs to be done. The read operation is divided into two phases. In the first phase, BL and $\overline{BL}$ are precharged to $V_{dd}/2$ volts. If the supply voltage is equal to 1 volt, then the bit lines are charged to 0.5 volts. This step is known as *pre-charging*. Subsequently, the SRAM cells of interest are accessed by setting the corresponding word line. The sense amplifiers simply monitor the difference in voltage between BL and $\overline{BL}$. The moment the absolute value of the difference exceeds a threshold, the result can be inferred. For example, if we are reading a logical 1, we need not wait for BL to reach 1, and $\overline{BL}$ to reach 0. If the voltage difference between BL and $\overline{BL}$ exceeds a threshold, then we can declare the result to be 1.

This method is very fast because of the following reasons. Pre-charging bit lines is very fast because there are dedicated pre-charge circuits that can pump a large amount of current into the bit lines to enable faster charging or discharging. After pre-charging inferring the stored value from the voltage swing between BL and $\overline{BL}$ is also very fast. This is because the threshold for the voltage swing is much lower than the supply voltage. Given the high capacitance of bit lines, the time to charge/discharge bit lines is very crucial. Hence, if we reduce the amount of the voltage swing that is required to infer the value stored in the SRAM cell, it makes a significant difference.

We can justify the overhead of pre-charge circuits, write drivers, and sense amplifiers, if we have a large number of SRAM cells. Hence, SRAMs are suitable for structures such as register files and on-chip memories. They should not be used for storing a few bits; flip-flops are better choices.

7.4.2 Content Addressable Memory (CAM)

CAM Cell

In this section, we shall look at a special type of memory cell called a CAM (Content Addressable Memory) cell. First, consider an SRAM array. A typical SRAM array is a matrix of SRAM cells. Each row contains a vector of data. A given row is addressed or located by using a specific set of address bits. However, it is possible to locate a row using a different method. We can address a row by its content. For example, if each row contains 32 SRAM cells, then we can think of the contents of a row as a 32-bit number. We wish to address the row this 32-bit number. For example, if a row contains 0x AB 12 32 54, then we should be able to find the index of the row that contains this value. Such a memory is known as a CAM memory, and each basic cell is known as a CAM cell. A CAM memory is typically used to implement a hashtable(see [Cormen et al., 2009]) in hardware. A hashtable saves key-value pairs such as (name and address). We address a hashtable by its key, and read the value. It is a more flexible data structure than an array, because we are not limited to integer indices. We shall use CAM arrays to implement some memory structures in Chapter 11.

Let us take a look at a 10 transistor CAM cell in Figure 7.28. If the value stored in the SRAM cell, V, is not equal to the input bit, A_i, then we wish to set the value of the *match* line to 0. In the CAM cell, the upper half is a regular SRAM cell with 6 transistors. We have 4 extra transistors in the lower half. Let us now consider transistor $T1$. It is connected to a global *match* line, and transistor $T2$. $T1$ is controlled, by the value, V, which is stored in the SRAM cell, and $T2$ is controlled by $\overline{A_i}$. Let us assume that $V = \overline{A_i}$. If both of them are 1, then transistors $T1$ and $T2$ are in the ON state, and there is a direct conducting path between the

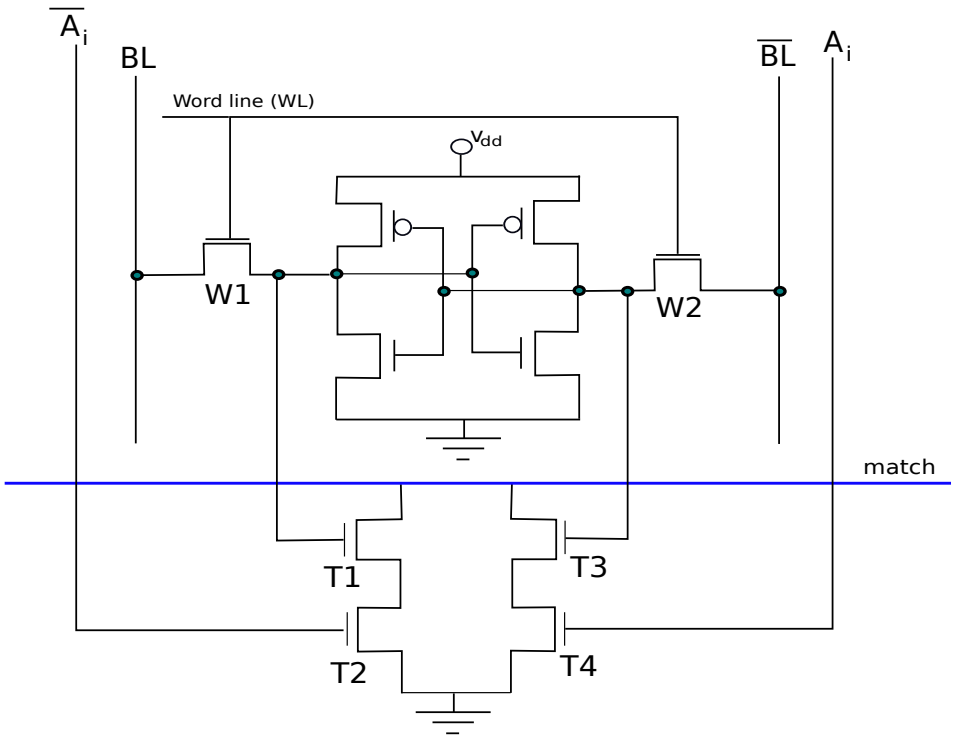

Figure 7.28: A 10 Transistor CAM cell

match line and ground. Thus, the value of the match line will get set to 0. However, if V and $\overline{A_i}$ are both 0, then the path through $T1$ and $T2$ is not conducting. But, in this case, the path through $T3$, and $T4$ becomes conducting, because the gates of these transistors are connected to $\overline{V}$, and A_i respectively. The input to both the gates is a logical 1. Thus, the match line will be pulled down to 0. The reader can conversely verify that if $V = A_i$, no conducting path is formed. Thus, a CAM cell drives the match line to a logical 0, if there is a mismatch between the value stored and the input bit, A_i.

Array of CAM Cells

Figure 7.29 shows an array of CAM cells. The structure is mostly similar to an SRAM array. We can address a row by its index, and perform a read/write access. Additionally, we can compare each row of the CAM cell with the input, A. If any row matches the input, then the corresponding match line will have a value of 1. We can compute a logical OR of all the match lines, and decide if we have a match in the CAM array or not. Additionally, we can connect all the match lines of the CAM array to a priority encoder to find the index of the row that matches the data.

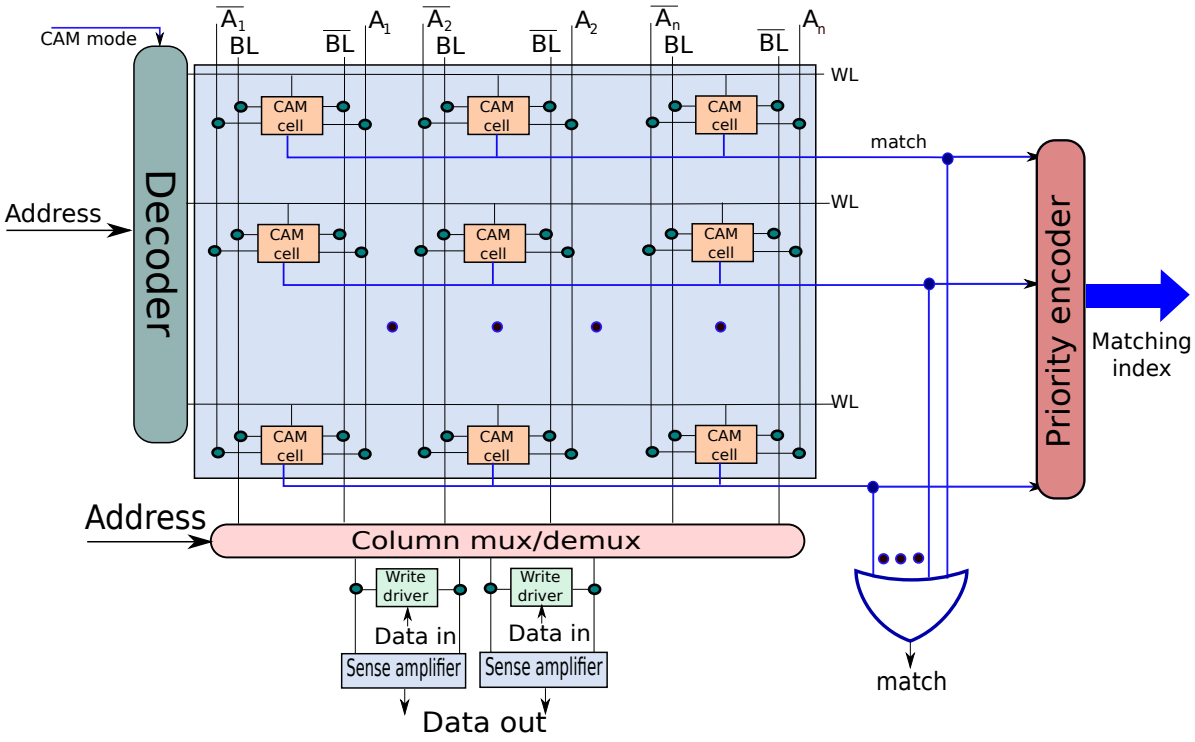

Figure 7.29: Array of CAM Cells

7.4.3 Dynamic RAM (DRAM)

Let us now take a look at a memory technology that just uses one transistor to save a bit. It is thus very dense, area, and power efficient. However, it is also much slower than SRAMs and latches. It is suitable for large off-chip memories.

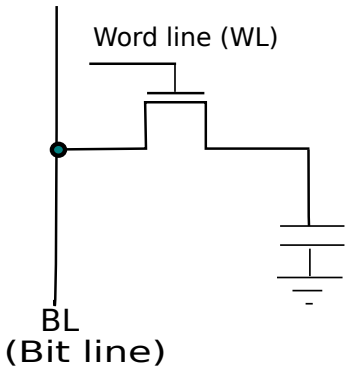

Figure 7.30: A DRAM cell

A basic DRAM(dynamic RAM) cell is shown in Figure 7.30. The gate of the single transistor

is connected to the word line, which enables or disables it. One of the terminals is connected to a capacitor that stores charge. If the bit stored is a logical 1, then the capacitor is fully charged. Otherwise, it is not charged.

Thus, reading and writing values is very easy. We need to first set the word line such that the capacitor can be accessed. To read the value we need to sense the voltage on the bit line. If it is at ground potential, then the cell stores 0, else if it is close to the supply voltage, then the cell stores 1. Similarly, to write a value, we need to set the bit line (BL) to the appropriate voltage, and set the word line. The capacitor will get charged or discharged accordingly.

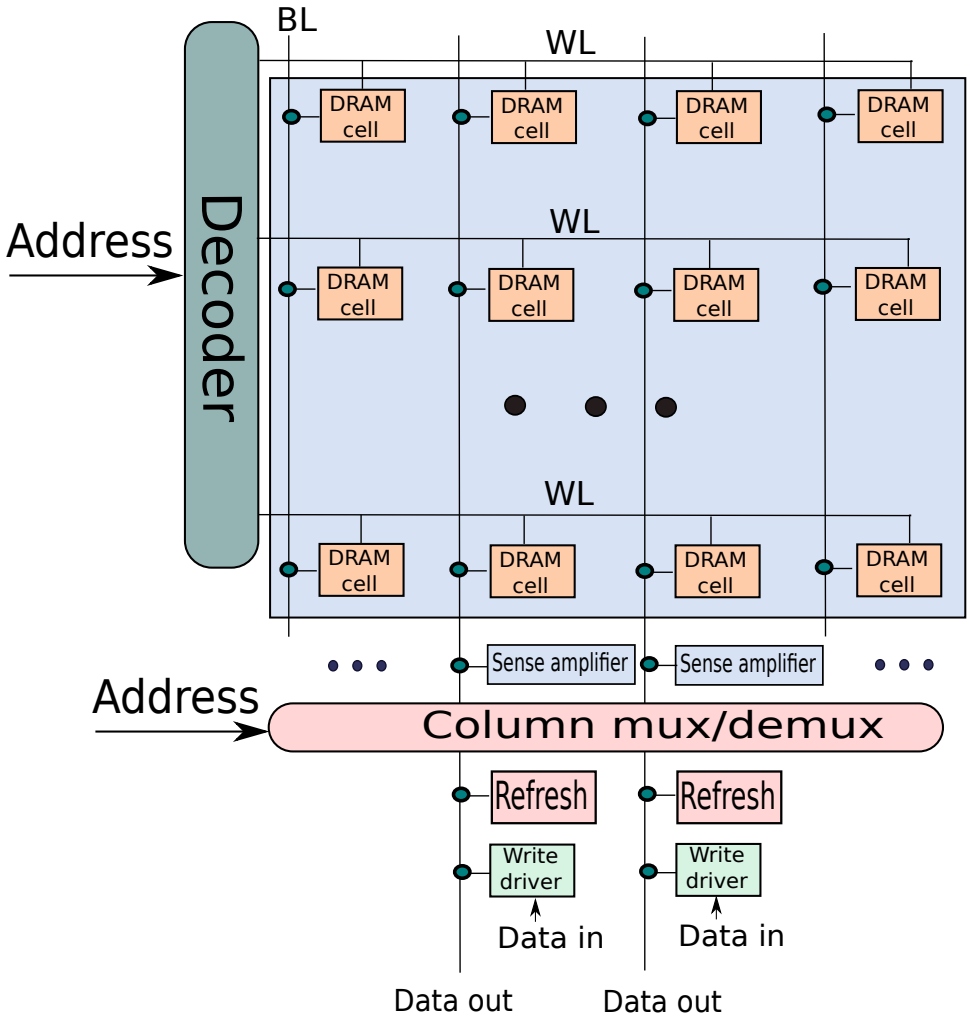

Figure 7.31: Array of DRAM cells

However, in the case of DRAMs, everything does not come for free. Let us assume that the capacitor is charged to a voltage equal to the supply voltage. In practice, the capacitor will gradually leak some charge through the dielectric, and the transistor. This current is very small, but the total loss of charge over a large duration of time can be significant and can

ultimately discharge the capacitor. To prevent this, it is necessary to periodically refresh the value of a DRAM cell. We need to read it and write the data value back. This also needs to be done after a read operation because the capacitor loses some charge while charging the bit line. Let us now try to make an array of DRAM cells.

Array of DRAM Cells

We can construct an array of DRAM cells (see Figure 7.31) the same way that we created an array of SRAM cells. There are three differences. The first is that there is one bit line instead of two. Second, we also have a dedicated refresh circuit connected to the bit lines. This is used after read operations, and is also invoked periodically. Finally, in this case the sense amplifiers appear before the column mux/demux. The sense amplifiers additionally cache the data for the entire DRAM row (also called a DRAM *page*). They ensure that subsequent accesses to the same DRAM row are fast because they can be serviced directly from the sense amplifiers.

Let us now briefly discuss the timing aspects of modern DRAMs. In the good old days, DRAM memory was accessed asynchronously. This means that the DRAM modules did not make any timing guarantees. However, nowadays, every DRAM operation is synchronized with a system clock. Hence, today's DRAM chips are synchronous DRAM chips (SDRAM chips). Figure 13.26 in Chapter 13 shows the timing diagram of a simple SDRAM chip for a read access.

Synchronous DRAM memories typically use the DDR4 or DDR5 standards as of today. DDR stands for *Double Data Rate*. Devices using the earliest standard, DDR1, send 8-byte data packets to the processor on both the rising and falling edges of the clock. This is known as *double pumped* operation. The peak data rate of DDR1 is 1.6 GB/s. Subsequent DDR generations extend DDR1 by transferring data at a higher frequency. For example, DDR2 has twice the data rate as DDR1 devices (3.2 GB/s). DDR3 further doubles the peak transfer rate by using a higher bus frequency, and has been in use since 2007 (peak rate of 6.4 GB/s).

7.4.4 Read Only Memory (ROM)

Let us now consider simpler memory structures that are read-only in nature. We require read only memories that save data that should never be modified. This can include security information in smartphones, BIOS chips in the motherboards of processors, or the programs in some microcontrollers. We desire read only memory such that it is not possible for users to maliciously change the information stored in them. It turns that we can make simple modifications to our DRAM cell, and construct read only memories.

ROM Memories

The capacitor in a DRAM cell stores a logical bit. If it stores a logical 1, then the charge across the capacitor is equal to the supply voltage V_{dd}, and if it stores a logical 0, then the charge across the capacitor is 0V. Instead of having a capacitor, we can directly connect one end of the transistor to either ground or V_{dd} depending upon the bit that we wish to store. This can be done at the time of designing and manufacturing the chip. The ROM memory cell as shown in Figure 7.32 replaces the capacitor by a direct connection to V_{dd} or ground. A ROM array is similar to a DRAM array. However, it does not have write drivers and refresh circuitry.

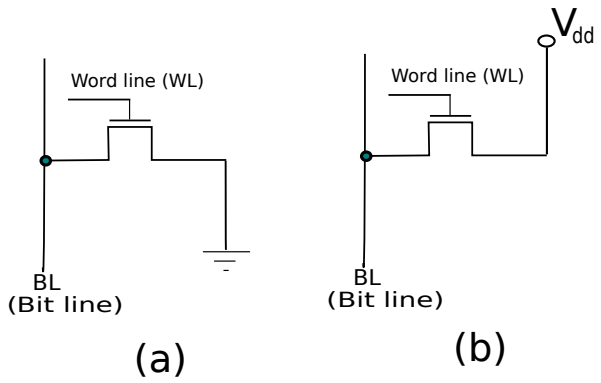

Figure 7.32: (a) ROM cell storing a logical 0, (b) ROM cell storing a logical 1

PROM (Programmable ROM) Memories

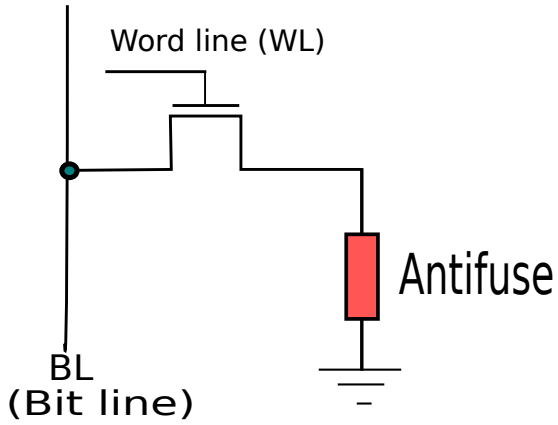

Figure 7.33: PROM Cell

Let us now look at a programmable ROM (PROM) cell that can be programmed only once to store either a 0 or 1. Typically, vendors of PROM cells release PROM arrays to the market. They are then programmed by microprocessor vendors to hold a given set of data values. Note that once data has been written, we cannot make any further modifications. Hence, it acts like read only memory. A PROM cell is shown in Figure 7.33. We have a connection between the transistor and ground through an element known as an *antifuse*. An *antifuse* is the reverse of a

conventional fuse. By default, an antifuse is a very weak conductor of current. However, when we transfer a large amount of current through the antifuse, it changes its structure and forms a conducting path. In the former case, the transistor is left floating, and it does not have the ability to drive the bit line to a logical 0. Hence, we can infer that the cell stores a logical 1. Once, the antifuse becomes conducting, the transistor (if enabled by the word line) can drive the bit line to a logical 0. Thus, in this case, the transistor stores a logical 0. A PROM cell based array is similar to an array of ROM cells. Each bit line is initially precharged. After enabling the word line, if the voltage at the sense amplifiers does not increase, then we can infer a logical 1. However, if the voltage keeps decreasing towards 0 V, then we can infer a logical 0.

We can build antifuses with a variety of materials. Dielectric antifuses use a thin layer of a sensitive material (silicon oxides) between two conducting wires. This thin layer breaks down and forms a conducting path upon the application of a large current pulse. In place of dielectrics we can use other materials such as amorphous silicon.

Read only memories are useful in a limited set of scenarios since their contents cannot be modified. We have two related sets of devices called EPROM (Erasable PROM), and EEPROM (Electrically Erasable PROM). Typically, these memories are made of special transistors that allow charge storage in a structure known as a *floating gate*. EPROM memories could be erased by applying a strong ultraviolet light pulse to the floating gate. However, such memories are not used today, and have been superseded by flash memories that can be read, written, and erased electrically. Flash memories are explained in detail in Section 13.8.4.

7.4.5 Programmable Logic Arrays

It turns out that we can make a combinational logic circuit out of a memory cell similar to a PROM cell very easily. Such devices are known as *programmable logic arrays*, or PLAs. PLAs are used to implement complex logic functions consisting of tens or hundreds of minterms in practice. The advantage of a PLA over a hardwired circuit made up of logic gates is that a PLA is flexible. We can change the Boolean logic implemented by the PLA at run time. In comparison, a circuit made in silicon can never change its logic. Secondly, designing and programming a PLA is simpler, and there are a lot of software tools to design and work with PLAs. Lastly, a PLA can have multiple outputs, and it can thus implement multiple Boolean functions very easily. This additional flexibility comes at a cost, and the cost is **performance**.

Let us now consider an example. Let us assume that we wish to implement the Boolean function $(\overline{ABC} + AB)$ using a PLA. Let us break the Boolean expression into a set of minterms (see Section 2.1.6). We thus have:

$$\overline{ABC} + AB = \overline{ABC} + ABC + AB\overline{C} \tag{7.12}$$

Since, we have three variables (A, B, and C) here, we have 8 possible minterms. Let us thus have a PLA with 8 rows that generates the values of all the possible minterms. Let each row correspond to a minterm. Let us design the PLA in such a way that a row has an output equal to 1 if the corresponding minterm evaluates to 1. We can compute the result of the entire Boolean expression by computing the logical OR of the values of all the minterms that we are interested in. For this specific example $(\overline{ABC} + AB)$, we are interested in 3 out of 8 minterms.

Hence, we need a mechanism to filter these 3 minterms and compute a logical OR of their values.

Let us start out with the design of a PLA cell.

PLA Cell

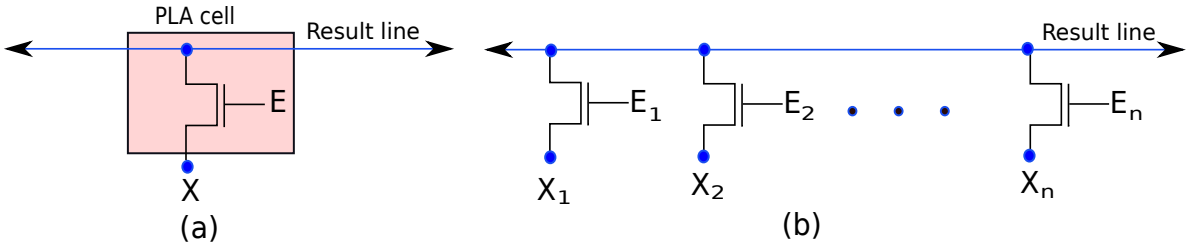

Figure 7.34: A PLA cell

The PLA cell shown in Figure 7.34(a) is in principle similar to a basic PROM cell. If the value (E) at the gate is equal to 1, then the NMOS transistor is in the ON state. As a result of this, the voltage difference between the source and drain terminals of the NMOS transistor is very small. In other words, we can simplistically assume that the voltage of the result line is equal to the voltage of the signal, X. If ($E = 0$), the NMOS transistor is in the OFF state. The result line is floating, and it maintains its precharged voltage. In this case, we propose to infer a logical 1.

Let us now construct a row of PLA cells where each PLA cell is connected to an input wire at its source terminal as shown in Figure 7.34(b). The inputs are numbered $X_1 \ldots X_n$. The drains of all the NMOS transistors are connected to the result line. The gates of the transistors of the PLA cells are connected to a set of enable signals, $E_1 \ldots E_n$. If any of the enable signals is equal to 0, then that specific transistor is disabled, and we can think of it as being logically removed from the PLA array.

Let us consider all the PLA cells that are enabled (gate voltage is equal to a logical 1). If any of the source inputs ($X_1 \ldots X_n$) is equal to 0, then the voltage of the result line will be driven to 0. We assume that we precharge the result line to a voltage corresponding to logical 1 at the beginning. Now, if none of the input voltages is equal to 0, then the value of the result line will be equal to a logical 1. We can thus conclude that the Boolean function computed by a row of PLA cells is equal to $X_1.X_2 \ldots .X_n$ (assuming that all the cells are enabled). For example, if we want to compute the value of the minterm, $ABCD$, then we need to set $X_1 = A$, $X_2 = B$, $X_3 = C$, and $X_4 = D$. The Boolean value represented by the result line will be equal to $ABCD$.

In this manner, we can evaluate the values of all the minterms by connecting the source terminals of PLA cells to the input bits. Up till now we have not considered the case of Boolean variables in minterms in their complemented form such as $\overline{A}BC\overline{D}$. For the minterm, $\overline{A}BC\overline{D}$, we need to make the following connections. We need to connect 4 PLA cells to the result line,

where their source terminals are connected to the signals $\overline{A}$, B, C, and $\overline{D}$ respectively. We need to generate, $\overline{A}$, and $\overline{D}$ by complementing the values of A and D using inverters.

Array of PLA Cells

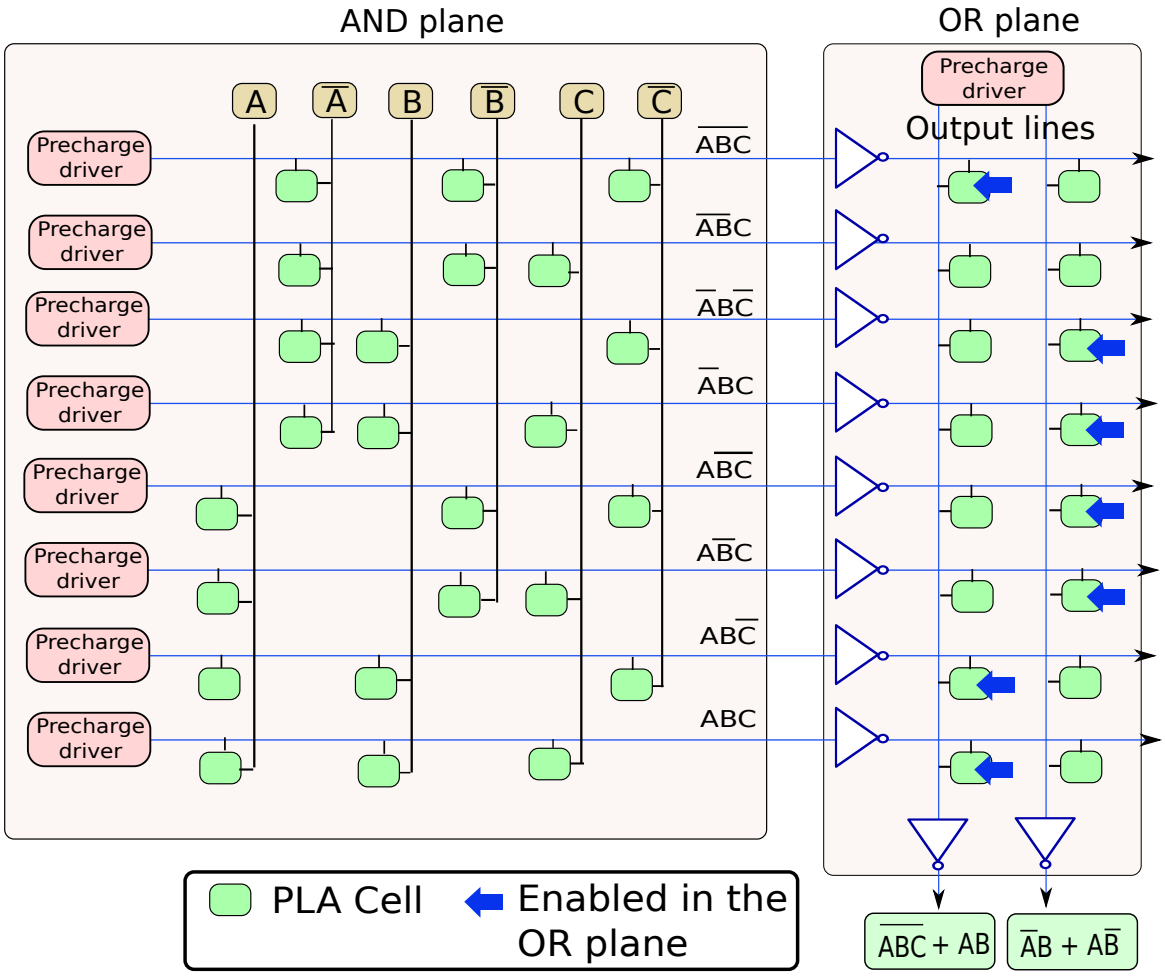

Figure 7.35: Array of PLA cells

Let us now create an array of PLA cells as shown in Figure 7.35. Each row corresponds to a minterm. For our 3 variable example, each row consists of 6 columns. We have 2 columns for each variable (original and complemented). For example, the first two columns correspond to A and $\overline{A}$ respectively. In any row, only one of these two columns contains a PLA cell. This is because A and $\overline{A}$ cannot both be true at the same time. In the first row, we compute the value of the minterm, $\overline{ABC}$. Hence, the first row contains PLA cells in the columns corresponding to $\overline{A}$, $\overline{B}$, and $\overline{C}$. We make similar connections in the rest of the rows for the remaining minterms. This part of the PLA array is known as the AND plane because we are computing a logical AND of the values (original or complemented) of variables. The AND plane of the PLA array is

independent of the Boolean functions that we wish to compute. Given the inputs, it calculates the value of all possible minterms.

Now, we need to compute the logical OR of the minterms that we are interested in. For example, in Equation 7.12, we are interested in the logical OR of 3 minterms. To compute the OR function, we use another PLA array known as the OR plane. However, there is a small problem here. A row of PLA cells is designed to compute a logical AND of all the inputs. We can use DeMorgan's theorem to compute the OR of inputs using a PLA array. Let us use the following relationship:

$$(X_1 + X_2 + \ldots X_n) = \overline{\overline{X_1 + X_2 + \ldots X_n}} = \overline{\overline{X_1}.\overline{X_2}.\ldots \overline{X_n}}$$

Thus, to compute the logical OR of $(X_1, X_2, \ldots X_n)$, we need to complement each input, compute the logical AND of all the complemented inputs, and compute the complement of the result. A similar computation needs to be performed in the OR plane of the PLA array. In Figure 7.35, we have inverters to compute the logical negation of each minterm. Then, we compute their logical AND using a column of PLA cells (similar to a row of PLA cells). In this case, only the PLA cells that correspond to minterms in the Boolean expression need to be enabled (shown as an arrow in Figure 7.35). The rest of the PLA cells in each column are disabled. Finally, we compute the logical negation of the result line, to get the value of the Boolean function.

Note that we can have multiple output lines in the OR plane, and thus we can compute the values of multiple Boolean functions in parallel. In Figure 7.35 we also compute the value of $A \oplus B = A.\overline{B} + \overline{A}.B$ in the second output line. For the result lines, and the output lines, we assume an array of sense amplifiers that perform appropriate voltage conversions. For the sake of simplicity, we do not show them in the figure.

The important point to remember here is that the OR plane is programmable. As shown in Figure 7.35, we compute the logical OR of a set of result lines by setting the gate voltage of the connecting PLA cell to a logical 1. At any point of time, we can change the connections to the output lines by changing the gate voltages, and we can thus change the Boolean expression that is computed by the PLA.

Way Point 4

- *We have assembled the arsenal required to implement circuits to perform complex arithmetic operations.*
- *Using our set of logic gates, flip-flops, memories, and arithmetic circuits(to be studied in Chapter 8), we are ready to implement a full fledged processor.*

7.5 Summary and Further Reading

7.5.1 Summary

Summary 7

1. *Modern processors use silicon based transistors. Silicon is a semi-conductor. However, its properties can be modified by adding impurities (known as doping) from the III, and V groups of the periodic table. If we add group III elements such as boron (p type), we remove a charge carrier from the silicon atom lattice, and thus create a hole. Similarly, if we add a group V element such as phosphorus (n type), then we introduce an additional electron to the lattice. In either case, we increase the conductivity of silicon.*

2. *We can create a p-n junction by juxtaposing p-type and n-type silicon structures. A p-n junction conducts when it is forward biased (the p side is connected to the higher voltage), and stops conducting when it is reverse biased.*

3. *We can create an NMOS transistor by having two wells of n-type silicon in a p-type substrate. These wells are connected to electrical terminals, and are known as the* source *and* drain*. The area between the wells is known as the channel, and is separated from an electrical terminal (known as the* gate*) by a thin layer of silicon dioxide. When we apply a positive voltage (> threshold voltage) at the gate, the NMOS transistor can conduct current. Otherwise, it acts like an open circuit.*

4. *In comparison, the PMOS transistor has two p-type wells in an n-type substrate. It forms a conducting path, when the voltage at the gate is 0V.*

5. *We can use NMOS and PMOS transistors to implement a basic inverter, NAND, NOR, AND and OR gates.*

6. *We can use these basic gates to create a host of complex structures such as the XOR gate, multiplexer, decoder, encoder, and priority encoder.*

7. *By creating a feedback path between NAND or NOR gates, we can save a logical bit. This structure is known as a* latch*. If we enable and disable access to a latch based on the value of a clock signal, then the latch is known as a* flip flop*.*

8. *We have considered SR, JK, and D flip flops in this chapter. The SR flip flop does not have deterministic outputs for all combinations of input transitions.*

9. *We use a master slave design for JK, and D flip flops because we can make the inputs appear at the outputs almost instantaneously at the negative clock edge.*

10. *A set of flip flops can be used to make registers.*

11. *We use a cross coupled pair of inverters to make an SRAM cell. It is connected by two transistors (enabled by a word line) to a pair of bit lines. A sense amplifier monitors the difference in voltages across the pair of bit lines. Based on the sign of the difference, we can infer a logical 0 or 1.*

12. *In comparison, a DRAM cell uses a single transistor and a capacitor. It is a high density memory technology. However, we need to regularly refresh the value of DRAM cells to keep it operational.*

13. *We can modify the basic structure of a DRAM cell by creating hardwired connections between the transistor and the supply rails to implement read only memory. Programmable ROM (PROM) cells use an antifuse to create an alterable connection to V_{dd}.*

14. *We can create programmable logic arrays for computing the values of Boolean expressions by arranging PLA cells (similar to PROM cells) in an array. In the AND plane, each row computes the value of a minterm. In the OR plane, we compute a logical OR of all the minterms that form a part of the Boolean expression.*

7.5.2 Further Reading

Semiconductors and electronic devices are thoroughly studied in advanced courses on semiconductor device physics. Readers can refer to the books by Sze [Sze and Ng, 2006], and Streetman [Streetman and Banerjee, 2005] for a deeper discussion on the physics of semiconductor devices. In advanced courses on semiconductor device physics, students typically study the basics of the operations of diodes, transistors, and other semiconductor devices from the point of view of quantum mechanics. After introducing semiconductor devices, we introduced combinational logic gates and sequential logic elements. The simple structures that we introduced in this chapter, are very commonly used. Students should however take a look at the following books [Lin, 2011, Wakerly, 2000, Dally and Poulton, 1998] for getting a thorough understanding of the devices, including their behavior from the perspective of analog electronics. We lastly talk about memories. The book, "Introduction to VLSI Systems", by Ming Bo Lin [Lin, 2011] has a fairly in-depth coverage of memory structures. Memory technology, especially DRAM technology, is advancing very quickly. A host of standards, and design styles have evolved over the last decade. A lot of these trends are discussed in the book on memory systems by Jacob, Ng, and Wang [Jacob et al., 2007]. This book is also very useful for professionals who are looking to build commercial systems with state-of-the-art memory technology.

Exercises

Transistors and Logic Gates

Ex. 1 — Describe the operation of a p-n junction.

Ex. 2 — Define drift and diffusion current.

Ex. 3 — Describe the operation of a NMOS transistor?

Ex. 4 — Draw the circuit of a CMOS inverter, NOR gate, and NAND gate.

Ex. 5 — Implement the AND, OR and NOT gates using NAND gates only.

Ex. 6 — Implement the AND, OR and NOT gates using NOR gates only.

Ex. 7 — Implement XOR and XNOR gates using NAND gates only.

Ex. 8 — Implement the following Boolean functions by using only AND, OR and NOT gates.

(a) $\overline{A}.B + A.\overline{B} + \overline{A}.\overline{B}.\overline{C}$

(b) $\overline{(A+B+C)}.(\overline{A}.\overline{B}+C)$

(c) $\overline{(A+B)}.\overline{(\overline{C}+\overline{D})}$

(d) $\overline{A}.\overline{B}.\overline{C} + A.B.C + \overline{D}$

Ex. 9 — Answer Question 8 by using only NAND gates.

Combinational Logic and Sequential Logic

Ex. 10 — Draw the circuit diagram of a 3×8 decoder.

Ex. 11 — Draw the circuit diagram of a 8-3 bit encoder.

Ex. 12 — Draw the circuit diagram of a 8-3 bit priority encoder.

Ex. 13 — Suppose a poll has to be conducted with three entities A, B and C, each of which can either vote a 'yes' (encoded as 1) or a 'no' (encoded as 0). The final output is equal to the majority opinion. Draw a truth table of the system, simplify the function, and implement it using logic gates.

* **Ex. 14** — Most circuits in modern computers are built using NAND and NOR gates, because they are easy to build using CMOS technology. Suppose another technology in invented in the near future, which implements a new gate, X, very efficiently. X takes 3 inputs A, B and C and computes: $X(A,B,C) = A.B + \overline{C}$. Using only X gates and NOT gates, how will you implement the following function: $f(A,B,C) = A + B + C$?

** **Ex. 15** — Implement the following logic functions using a 4 to 1 multiplexer, and a single NOT gate.

(a) $AB + BC + AC$

(b) $\overline{A} + \overline{B} + \overline{C}$

(c) $A.\overline{B} + A.B.\overline{C}$

** **Ex. 16** — Is it possible to implement every 3 variable Boolean function with a 4 to 1 multiplexer, and a single NOT gate? Prove your answer.

Sequential Logic

Ex. 17 — What is the difference between a flip-flop and a latch?

Ex. 18 — Define the following terms:

i)Metastability

ii)*Keep out* region

iii)Setup time

iv)Hold time

Ex. 19 — Why do we wish to avoid the indeterminate state in an SR flip-flop?

Ex. 20 — What is the advantage of an edge sensitive flip-flop?

* **Ex. 21** — What is the fundamental advantage of a JK flip-flop over a D flip-flop?

Ex. 22 — Describe the design of registers in your own words.

Ex. 23 — An edge sensitive toggle flip-flop (or T flip-flop) has a single input T and toggles its state on a negative clock edge if $T = 1$. If $(T = 0)$, then it maintains its state. How will you construct an edge sensitive T flip-flop form an edge sensitive J-K flip-flop?

* **Ex. 24** — Can you create a negative edge triggered D flip-flop using 2 multiplexers, and a NOT gate?

Ex. 25 — Design a SR flip-flop with NOR gates.

* **Ex. 26** — Using two edge triggered D flip-flops, design a circuit that divides the frequency of the clock signal by 4.

** **Ex. 27** — Counters are essential components of any complex digital circuit. They are essentially sequential circuits which loop through a specific set of states. Design a counter which generates a sequence of numbers (in binary form) from 0 to 7 and cycles back again to 0. This is called a MOD 8 counter.

** **Ex. 28** — Using D flip-flops and logic gates, design a circuit, which generates the following sequence of numbers:

$$001 \rightarrow 100 \rightarrow 010 \rightarrow 101 \rightarrow 110 \rightarrow 111 \rightarrow 011 \rightarrow 001$$

Assume that the circuit never generates 000. This circuit can be used to generate pseudo-random numbers.

Memories

Ex. 29 — Compare the power, area and time of an SRAM, DRAM, and latch.

Ex. 30 — Propose a design for the column mux/demux circuit.

Ex. 31 — What is the role of the *match* line in a CAM array?

Ex. 32 — What is the role of the *refresh logic* in a DRAM array?

Ex. 33 — Describe the design of a ROM and PROM cell.

Ex. 34 — Design a single PLA array to compute al the following Boolean functions:

a)$A.B + B.C + C.A$

b)$A.\overline{B}.\overline{C} + \overline{A}.B.C$

c)$\overline{A + B}$

Design Problems

Ex. 35 — Design the following circuits using a circuit simulator such as Spice and verify their operation:

a)NOT gate

b)NAND gate

c)D flip-flop

d)SRAM cell

Ex. 36 — Prepare a report on novel memory technologies such as phase change memory, Ferro-electric RAM, and magneto-resistive RAM.