Chapter 2
Basics of Digital Logic

Scientists build to learn; Engineers learn to build.

– F. Brooks

Abstract In the previous chapter we made some statements regarding various features of digital logic systems; for example, we stated that they could easily represent the binary digits zero and one because these could be mapped onto low and high electrical signals. The goal of this chapter is to expand on these statements by showing how they are satisfied using basic physics. Since most people learn about physics in school and an in-depth discussion would require another book, our goal is not rigorous accuracy but simply a rough recap and overview of the pertinent details. Using this starting point, we build higher level digital logic components capable of representing meaningful values and performing useful operations on them. We then look at how simple storage elements can be constructed and how sequences of operations can be performed using state machines and clocks.

2.1 Switches and Transistors

2.1.1 Basic Physics

Everything in the universe is composed from primitive building blocks called **atoms**. An atom in turn is composed from a group of subatomic particles: a group of nucleons, either **protons** or **neutrons**, in a central core or **nucleus** and a cloud of **electrons** which orbit the nucleus. The number of protons translates into the **atomic number** of the atom which roughly dictates what chemical element or material it is. Silicon has atomic number fourteen, for example, since there are fourteen protons in the nucleus. The arrangement of electrons around the nucleus is called the **electron configuration**; electrons can orbit the nucleus in one of several levels or **shells**.

Subatomic particles carry an associated electrical **charge** which is characteristic of their type; electrons carry a negative charge, protons have a positive charge and neutrons have neutral charge. An **ion** is an atom with a non-neutral charge overall. A negatively charged ion has more electrons in the electron cloud than protons in the nucleus, a positively charged ion has fewer electrons than protons. Electrons can be displaced from an electron cloud, using some energy, in a process called **ionisation**. How much energy is required to perform this process relates to how tightly coupled the electrons are to the nucleus and is dictated in part by the type of atom. For example, metals generally have loosely coupled electrons while plastics generally have tightly coupled electrons. We usually term these types of material **conductors** and **insulators** respectively.

Ionisation aside, electrons repel each other but are attracted by **holes** or space in an electron cloud. So given the right conditions, electrons may move around between atoms. In particular, if we apply a potential difference between two points on a conductor then electrons, and hence their associated charge, will move and create an electrical current between the points. This is a useful starting point but depends highly on the materials we are using and their electron configurations: if we do not have a material with the right number of electrons or holes, it will not behave as required. However, we can manipulate a material to have the required properties via a process called **doping**. We take the starting material and dope it, or mix it, with a second material called the **donor**. Depending on the properties of the donor, the new material will have a different number of electrons or holes but otherwise retain very similar characteristics. As an example, consider pure silicon which has an electron cloud of four electrons (only about half full): it is more or less an insulator. Doping with a boron or aluminium donor creates extra holes while doping with phosphor or arsenic creates extra electrons. We call the new materials P-type and N-type **semiconductors** respectively; the material is not a conductor or an insulator but somewhere in between.

In summary, we can more or less create materials with atomic characteristics of our choosing. The basic idea in building digital logic components from these materials is to sandwich together wafers of the materials in different ways. For example, electrons can flow from N-type to P-type semiconductors but not in the other direction; we can use this fact to build atomic scale **switches** called **transistors**.

2.1.2 Building and Packaging Transistors

Although transistors can perform a variety of functions, for example the amplification of signals, we will use them almost exclusively as switches. That is, a component that conducts, or allows charge to flow, between two terminals when we turn on the switch and not conduct, or be a resistor, when we turn off the switch. There are several different types of transistor but we will concentrate on one of the simplest, the **Field Effect Transistor (FET)** which was originally invented in 1952 by William Shockley, an engineer at Bell Labs. In this context we term the two tran-

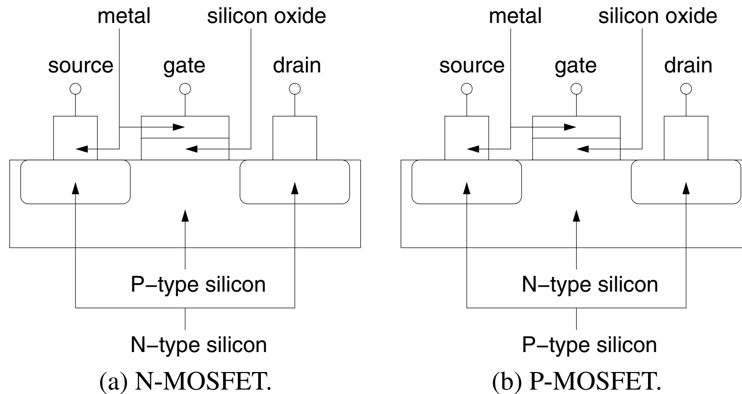


Figure 2.1 Implementation of N-MOSFET and P-MOSFET transistors.

sistor terminals the **source** and **drain**, and call the switch that controls the flow of current the **gate**.

To actually create the switch, we combine and layer materials with different properties. Again there are several ways to approach this: we concentrate on the most common design which is called a **Metal Oxide Semiconductor Field-Effect Transistor (MOSFET)**, invented in 1960 by Martin Atalla, also at Bell Labs. Figure 2.1 shows how a typical MOSFET device is built from N-type and P-type semiconductor materials and layers of metal; the two different types are termed N-MOSFET and P-MOSFET devices. The rough idea is that by applying a potential difference between the gate and source, an electric field is created which offers a channel between the source and drain through which current can flow. The type of channel depends on the types of semiconductor material used. Changing the potential difference between gate and source changes the conductivity between source and drain, and hence regulates the current: if the potential difference is small, the flow of current is small and vice versa. Thus, the gate acts like a switch that can regulate the current between source and drain.

For an N-MOSFET the types of semiconductor material used mean that when there is a high potential difference between gate and source, the conduction channel is open and current flows: the switch is on. When there is a low potential difference between gate and source, the channel closes and the switch is off. A P-MOSFET reverses this behaviour by swapping around the types of semiconductor material: when the potential difference is high the switch is off, when it is low the switch is on. Although we say the switch is off, the transistor still allows a small level of conductivity between source and drain because of the imperfect nature of the materials involved; this effect is termed **leakage**.

Transistors are seldom used in isolation and are more commonly packaged into larger components before use in larger digital circuits: these components form what are often called **logic styles**. The most frequently used style, and one of the rea-

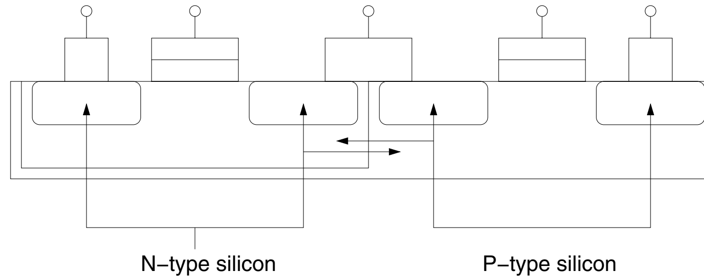
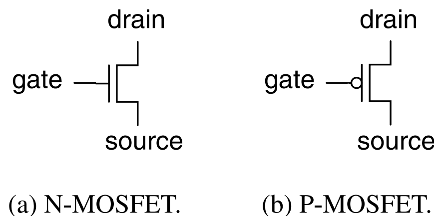


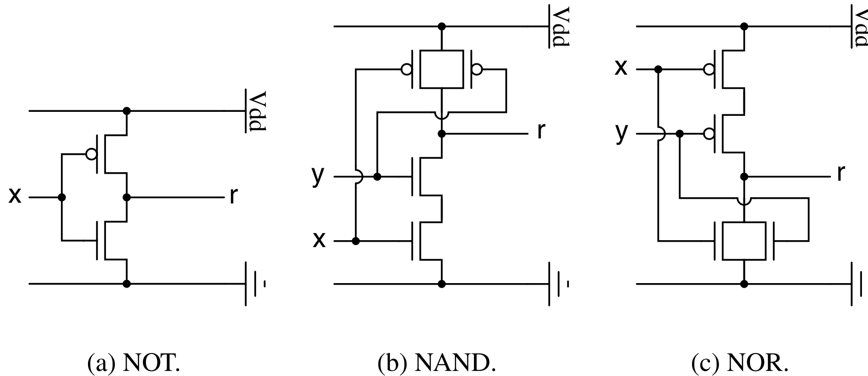
Figure 2.2 A CMOS cell built from N-MOSFET and P-MOSFET transistors.



(a) N-MOSFET. (b) P-MOSFET.

Figure 2.3 Symbolic representations of N-MOSFET and P-MOSFET transistors.

sons behind the popularity of MOSFET type transistors, is called **Complementary Metal-Oxide Semiconductor (CMOS)**; it was invented in 1963 by Frank Wanlass at Fairchild Semiconductor. The idea of CMOS is to pair each N-MOSFET transistor with a P-MOSFET partner into a component sometimes termed a **cell**. Figure 2.2 details a CMOS cell; the two transistors are arranged in a complementary manner such that whenever one is conducting, the other is not. This arrangements allows one to form circuits which have two key features: a **pull up network** of P-MOSFET transistors which sit between the positive power rail (that supplies the V_{dd} voltage level) and the output, and a **pull down network** of N-MOSFET transistors which sit between the negative power rail (which supplies the GND voltage level) and the output. Only one of the networks will be active at once, so aside from leakage a CMOS cell only consumes power when the inputs are **toggled** or **switched**. Since we commonly aim to build circuits with many very small transistors in close proximity to each other, this feature provides some significant advantages. In particular, it means that the use of CMOS reduces power consumption and heat dissipation which in turn aids reliability and enables size reduction. Thus, CMOS can be characterised as a low-power alternative to competitors such as **Transistor-Transistor Logic (TTL)** which is usually built from **Bipolar Junction Transistors (BJT)** and is operationally faster. It can be common to mix the characteristics of CMOS and TTL in one circuit, a technology termed bipolar-CMOS (BiCMOS).



(a) NOT. (b) NAND. (c) NOR.

Figure 2.4 Implementations of NOT, NAND and NOR logic gates using transistors.

x	y	r = NOT x	r = x NAND y	r = x NOR y
GND	GND	V_{dd}	V_{dd}	V_{dd}
GND	V_{dd}	V_{dd}	V_{dd}	GND
V_{dd}	GND	GND	V_{dd}	GND
V_{dd}	V_{dd}	GND	GND	GND

Table 2.1 Truth tables for transistor implementations of NOT, NAND and NOR.

When constructing higher-level components that utilise these behaviours, we use the symbols detailed in Figure 2.3 to represent different transistor types. We term the higher-level components we are aiming for **logic gates**, the idea being that they implement some function related to those we saw at the end of Chapter 1 and whose behaviour is described by Table 2.1.

For example, consider building a component which inverts the input; we call this a NOT gate. That is, when the input x is V_{dd} the component outputs GND and vice versa. We can achieve this using the arrangement of transistors in Figure 2.4a. To see that this works as required, consider the different states the input can be in and the properties of N-MOSFET and P-MOSFET transistors. When the input x is connected to GND the bottom transistor will be closed while the top one will be open: the output r will be connected to V_{dd}. When input x is connected to V_{dd} the bottom transistor will be open while the top one will be closed: the output will be connected to GND.

We can build two further components in a similar way; these are the NAND (or NOT AND) and NOR (or NOT OR) gates in Figure 2.4b-c. We can reason about their behaviour in a similar way as above. For example, consider the NAND gate. When input x is connected to GND the bottom transistor will be closed while the top left transistor will be open: the output r will be connected to V_{dd} no matter what input y is connected to. Likewise, when input y is connected to GND the middle

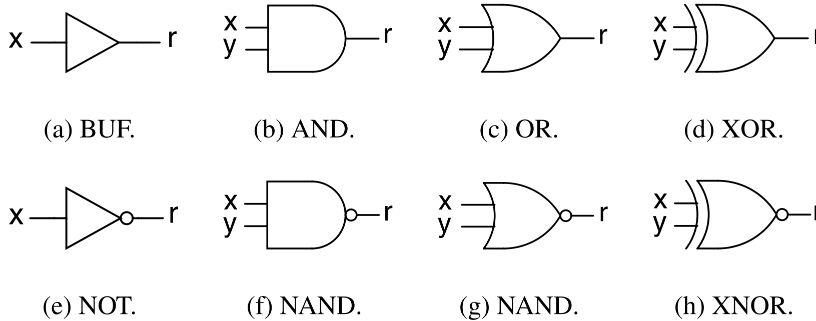


Figure 2.5 Symbolic representation of standard logic gates.

transistor will be closed while the top right transistor will be open: the output will be connected to V_{dd} no matter what input x is connected to. However, when both x and y are connected to V_{dd} the top two transistors will be closed but the bottom two will be open; this means the output is connected to *GND*. A similar reasoning applies to the operation of the NOR gate.

Clearly this is a good step toward building useful circuits in that we now have devices, constructed from very simple components, that perform meaningful operations. Specifically, we can relate the functions they implement to many of the concepts previously encountered. That is, we have a means of actually building devices to compute Boolean functions that up until now have been discussed abstractly.

2.2 Combinatorial Logic

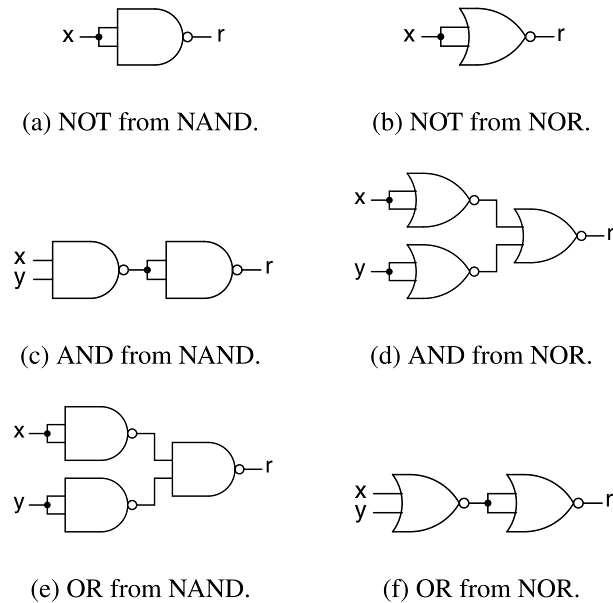
2.2.1 Basic Logic Gates

It is somewhat cumbersome to work with transistors because they represent such a low-level of detail. In order to make our life easier, we commonly adopt a more abstract view of logic gates by taking two steps: we forget about the voltage levels *GND* and V_{dd}, abstractly calling them 0 and 1, then we forget about the power rails and just draw each gate using a single symbol with inputs and outputs. The goal is that by viewing the gates more abstractly, we can focus on their properties as Boolean logic functions rather than their underlying implementation as transistors or via another technology. By composing the gates together we can build all manner of operations which are loosely termed **combinatorial logic**; the gate behaviours combine to compute the function continuously, the output is always updated as a product of the inputs.

Figure 2.5 defines symbols for each of the NOT, NAND and NOR gates built above and also the AND, OR and XOR operations described at the end of the previ-

	BUF	AND	OR	XOR
x y	$r = x$	$r = x \wedge y$	$r = x \vee y$	$r = x \oplus y$
0 0	0	0	0	0
0 1	0	0	1	1
1 0	1	0	1	1
1 1	1	1	1	0
	NOT	NAND	NOR	XNOR
x y	$r = \neg x$	$r = x \overline{\wedge} y = \neg(x \wedge y)$	$r = x \overline{\vee} y = \neg(x \vee y)$	$r = x \overline{\oplus} y = \neg(x \oplus y)$
0 0	1	1	1	1
0 1	1	1	0	0
1 0	0	1	0	0
1 1	0	0	0	1

Table 2.2 Truth tables for standard logic gates.

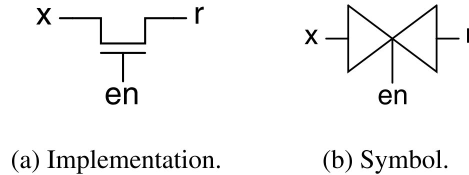


(a) NOT from NAND. (b) NOT from NOR.

(c) AND from NAND. (d) AND from NOR.

(e) OR from NAND. (f) OR from NOR.

Figure 2.6 Identities for standard logic gates in terms of NAND and NOR.



(a) Implementation. (b) Symbol.

Figure 2.7 An implementation and symbolic description of a 3-state enable gate.

ous chapter; these symbols represent functions with behaviour described as a truth table in Table 2.2. Note the use of an **inversion bubble** on the output of a gate to invert the value. Using this notation, a buffer (or BUF) is simply a gate that passes the input straight through to the output and a NOT gate just inverts the input to form the output. Also note that for completeness we have included the XNOR (or NXOR) symbol which has the obvious meaning but is seldom used in practise. We use $\overline{\wedge}$, $\overline{\vee}$ and $\overline{\oplus}$ as a short hand to denote the NAND, NOR and XNOR operations respectively. Finally, for two input gates such as AND, OR and XOR we often use a notational short hand and draw the gates with more than two inputs; this is equivalent to making a tree of two-input gates since, for example, we have that

$$(w \wedge x \wedge y \wedge z) = (w \wedge x) \wedge (y \wedge z).$$

The natural question to ask after this is why have we built NAND and NOR when AND and OR seem to be more useful ? The answer is related to the cost of implementation; it is simply more costly to build an AND gate from transistors, the cheapest way being to compose a NAND gate with a NOT gate. Since NAND and NOR are the cheapest meaningful components we can build, it makes sense to construct a circuit from these alone. This is possible because NAND and NOR are both **universal** in the sense that one can implement all the other logic gates using them alone; one simply substitutes gates using the identities in Figure 2.6 to perform the translation. The reason for wanting to use just one component to implement an entire circuit is related to how the circuit is fabricated: one can typically build much more reliable and compact circuits if they are more regular.

2.2.2 3-state Logic

Roughly speaking, one can freely connect the output of one logic gate to the input of another. While it does not really make sense to connect two inputs together since *neither* will drive current along the wire, connecting two outputs together is more dangerous since *both* will drive current along the wire. The outcome depends on a number of factors but is seldom good: the transistors which are wired to fight against each other typically melt and stop working.

x	en	r
0	0	Z
1	0	Z
Z	0	Z
0	1	0
1	1	1
Z	1	Z

Table 2.3 A truth table for a 3-state enable gate.

We can mitigate this fact by introducing a slightly different style of logic called 3**-state logic**; we do not go into a great deal of depth about how such logic gates are constructed but instead focus on their behaviour. The central component in 3-state logic is a switch which is sometimes called an **enable gate**. The single transistor implementation of the gate and the more common symbolic description are shown in Figure 2.7. The gate has some interesting characteristics in terms of its behaviour which is shown as a truth table in Table 2.3.

Notice that we have introduced a new logic state, hence the name 3-state, called Z or **high impedance**. When the enable signal en is set to 0 the gate does not pass anything through to the output r, it is not driven with anything so in a sense some other device can be connected to and drive the same wire. However, when en is set to 1, the gate passes through the input x to the output r: nothing else should be driving a value along this wire or we are back to the situation which caused the original melted transistor. In summary, the high impedance signal acts as a sort of "empty" value allowing any other signal to overpower it without causing damage to the surrounding transistors. Thus, careful use of enable gates allows two or more normal logic gate outputs to be connected to the same wire: as long as only one of them is enabled at a time, and hence driving a signal along the wire, there will be no disasters.

2.2.3 Designing Circuits

The next issue to address is how one combines simple Boolean logic gates to create higher-level functions. Fortunately we have already covered a lot of material which enables us to do this in a fairly mechanical form. In particular, given a truth table that describes the function required, one can perform a simple sequence of steps to extract the corresponding sum of products (SoP) expression built from a number of minterms:

1. Imagine there are n inputs numbered $1 \ldots n$, and one output:
   - Use $I_{i,j}$ to denote the value of input j in row i.

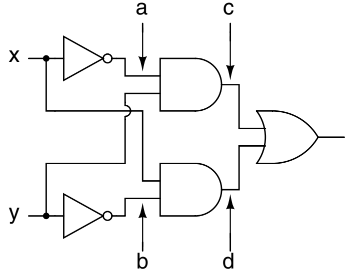


Figure 2.8 An implementation of an XOR gate.

- Use O_i to denote the value of the output in row i.

2. Find all the rows, a set R, for which given $r \in R$ we have that $O_r = 1$. That is, R is all row numbers where the output is 1.
3. For each such $r \in R$, find
   - A set H_r such that for $h \in H_r$, $I_{r,h} = 1$.
   - A set L_r such that for $l \in L_r$, $I_{r,l} = 0$.
4. The expression is then given by

$$O = \bigvee_{r \in R} \left(\bigwedge_{i \in H_r} I_{r,i} \wedge \bigwedge_{j \in L_r} \neg I_{r,j} \right).$$

The large $\wedge$ and $\vee$ symbols used here are similar to the summation and product symbols (Σ and Π). They basically AND and OR many terms together in a similar way to Σ and Π which add and multiply many terms together.

Consider the example of constructing an XOR gate whose truth table can be found in Table 2.2. Setting $x = I_1$ and $y = I_2$ while numbering the rows of the table $1 \ldots 4$, we find that we want a 1 from our expression in rows 2 and 3 and a 0 in rows 1 and 4. This is expressed using the set $R = \{2,3\}$ such that $H_2 = \{2\}$, $L_2 = \{1\}$, $H_3 = \{1\}$ and $L_3 = \{2\}$. As a result, our expression is

$$(I_2 \wedge \neg I_1) \vee (I_1 \wedge \neg I_2)$$

or rather

$$(\neg x \wedge y) \vee (x \wedge \neg y)$$

which we can verify as being correct. To build the circuit diagrammatically we simply replace each of the operators with a gate and connect the inputs and outputs together as one would expect; for our XOR expression above this is shown in Figure 2.8.

w	x	y	z	r
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Table 2.4 An example 4-input function.

2.2.4 Simplifying Circuits

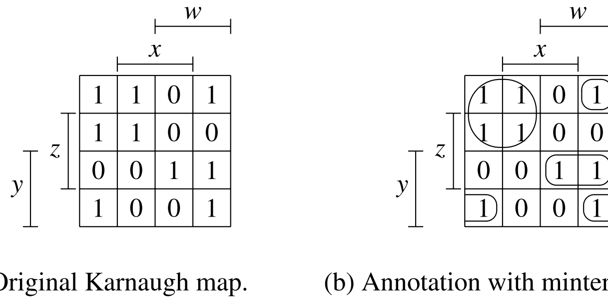
Again using the theory developed in the previous chapter, we can manipulate our XOR expression from above using the axiomatic laws of Boolean logic. For example, we can construct the product of sums (PoS) expression built from a number of maxterms

$$(x \lor y) \land (\neg x \lor \neg y)$$

or the functionally equivalent expression

$$(x \lor y) \land \neg(x \land y).$$

We know that if there were more than one output required, say m outputs for example, we can simply construct m such expressions using x and y which each produce one of the outputs. This raises an important issue in designing circuits: if we can simplify our expressions in the sense that they use less operators, we use less transistors and hence less space. This is a valuable step and is coupled to the issue that if we have multiple expressions, we can potentially share common blocks between the two; for example, if two expressions include some term $x \land y$, we only need one gate to produce the result, not two.



(a) Original Karnaugh map. (b) Annotation with minterms.

Figure 2.9 Karnaugh maps for an example 4-input function.

2.2.4.1 Karnaugh Maps

Consider the truth table in Table 2.4 describing some function r, and how one might derive a SoP expression for r to implement it. By applying the method from the previous section we get

$$\begin{aligned} r = {} & (\neg w \wedge \neg x \wedge \neg y \wedge \neg z) \vee \\ & (\neg w \wedge \neg x \wedge \neg y \wedge z) \vee \\ & (\neg w \wedge \neg x \wedge y \wedge \neg z) \vee \\ & (\neg w \wedge x \wedge \neg y \wedge \neg z) \vee \\ & (\neg w \wedge x \wedge \neg y \wedge z) \vee \\ & (w \wedge \neg x \wedge \neg y \wedge \neg z) \vee \\ & (w \wedge \neg x \wedge y \wedge \neg z) \vee \\ & (w \wedge x \wedge y \wedge z) \end{aligned}$$

which is verbose to say the least ! The **Karnaugh map**, a method of simplifying such expressions, was invented in 1953 by Maurice Karnaugh while working as an engineer at Bell Labs. The idea is to first provide a concise way to write down lengthy truth tables, and secondly a mechanism to easily create and simplify logic expressions from said tables without resorting to manipulation by logical axioms.

The first step is to encode the truth table on a grid as shown for our example in Figure 2.9a. The encoding used is not the natural binary ordering one might expect. For example, writing the inputs as vectors of the form (w, x, y, z) we might expect the order to follow the original truth table and produce the sequence

$$
\begin{array}{c}
(0,0,0,0) \\
(0,0,0,1) \\
(0,0,1,0) \\
(0,0,1,1) \\
(0,1,0,0) \\
(0,1,0,1) \\
(0,1,1,0) \\
(0,1,1,1) \\
\cdots
\end{array}
$$

such that moving from, for example, $(0,0,1,1)$ to $(0,1,0,0)$ flips the three bits associated with x, y and z. Instead, we use what is called a **Gray code**, named after Frank Gray who patented the idea in 1953 while also working as a researcher at Bell Labs; such codes had been known and used for a couple of hundred years before that. The basic idea is that between neighbouring lines in the sequence only one bit should change; thus we get a sequence such as

$$
\begin{array}{c}
(0,0,0,0) \\
(0,0,0,1) \\
(0,0,1,1) \\
(0,0,1,0) \\
(0,1,1,0) \\
(0,1,1,1) \\
(0,1,0,1) \\
(0,1,0,0) \\
\cdots
\end{array}
$$

such that moving from $(0,1,1,1)$ to $(0,1,0,1)$ flips only the bit associated with y. In fact, moving from any line to the previous or next flips only one bit. Looking at the Karnaugh map, the left most column represents the values where $(w,x) = (0,0)$ and reading from top to bottom where $(y,z) = (0,0)$, $(0,1)$, $(1,1)$, $(1,0)$. The next columns are similar for y and z but for the cases where $(w,x) = (0,1)$, $(1,1)$ and $(1,0)$.

Use of this technique to encode the Karnaugh map grid allows us to easily simplify the function in question. The basic idea is that by locating minterms which differ in only one input, we can simplify the resulting expression by eliminating this input. In our example, the minterms $(1,0,1,0)$ and $(1,0,1,1)$ offer a good demonstration of this fact. We could implement these minterms using the expressions

$$w \wedge \neg x \wedge y \wedge \neg z$$

and

$$w \wedge \neg x \wedge y \wedge z.$$

This is clearly wasteful however; it does not matter what value z takes since the overall function is still 1 as long as $w = 1$, $x = 0$ and $y = 1$. Thus we can eliminate

x	y	z	r
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	?
1	1	0	1
1	1	1	?

Table 2.5 An example 3-input function.

the z term from our expressions above and simply use the single and more simple minterm

$$w \wedge \neg x \wedge y$$

to cover both cases.

The next step in realising this simplification from our Karnaugh map is to gather cells in the grid whose entry is 1 into rectangular groups of size 2^n for some integer n. We can thus form groups of 1 cell, 2 cells, 4 cells and so on: the key thing is that the groups are rectangular. Due to the properties of the Gray code, these groups are allowed to wrap around the sides of the grid; this is valid since moving right from $(w,x,y,z) = (0,0,0,0)$ in the top left-hand corner to $(1,0,0,0)$ in the top right-hand corner still only flips one bit. Additionally, groups are allowed to overlap if they want as long as they are still rectangular. The annotated Karnaugh map in Figure 2.9b demonstrates these concepts by forming four groups.

Each of the final groups represents one term we need to implement in order to implement the overall function. The bigger the group, the less variables we need to include in each of the terms. Consider, for example, the group of four cells in the top left-hand corner. Looking at how the variables are assigned values in the grid, it does not matter what value x takes as long as $w = 0$ and it does not matter what value z takes as long as $y = 0$ because we still get a 1 as output. We can implement this term as $\neg w \wedge \neg y$ to cover all four cells in that group; this is much simpler than the four separate corresponding terms in our original implementation. Applying the same technique to the function as a whole, we find that

$$\begin{aligned} r = & (\neg w && \wedge \neg y &&) \vee \\ & (w \wedge \neg x \wedge \neg y \wedge \neg z) \vee \\ & (\neg x \wedge y \wedge \neg z) \vee \\ & (w \wedge y \wedge z) . \end{aligned}$$

Clearly this is significantly simpler than our initial effort and does not require tedious application of logical axioms. However, it is not the most simple description of the function; we leave this issue open to explore in the next section.

(a) Grouped without don't care state. (b) Grouped with don't care state.

Figure 2.10 Karnaugh maps for an example 3-input function.

Karnaugh maps can represent functions with any number of inputs but drawing them for more than eight or so becomes tricky. More importantly, there is no reason why they need to be square. For the three input function r in Table 2.5 we can draw the grid in Figure 2.10a. Another important feature is detailed in this example in that some of the outputs are neither 0 nor 1 but rather ?. We call ? the **don't care state** in the sense it can take any value we want; it is not that we do not know the value, we just do not care what the value is. One can rationalise this by thinking of a circuit whose output just does not matter given some combination of input, maybe this input is invalid and so the result is never used. Either way, we can use these don't care states to our advantage during simplification. By regarding them as 1 values when we want to include them in our groupings and 0 otherwise, we can further simplify our implementation. This can be seen in Figure 2.10b where without the ? state in the middle we are forced to implement two groups but by including it we can group four cells together into one: remember, less groups and larger groups will typically mean a simpler implementation.

2.2.4.2 Quine-McCluskey

Although the Karnaugh map method for simplification is attractive, it falls down when either the number of inputs grows too large or it needs to be implemented in software. In these cases we prefer **Quine-McCluskey minimisation**, a method developed independently by Willard Quine [55] and Edward McCluskey [39] in the mid 1950s. As an example of applying the method, we re-simplify the function r used above and detailed in Table 2.4.

The first step is to extract all the minterms from the truth table, i.e., all the combinations of input that result in a 1 in the output. We assign a number to each minterm and write them, using vectors of the form (w,x,y,z) in this case, within a table; this is shown in the top, first phase section of Table 2.6. Placed in this form, each entry is called an **implicant**. We place the implicants into groups according to the number of 1 values in their vector representation. For example, implicant 0 corresponding to $(0,0,0,0)$ has no 1 values and is placed in group 0; implicants 1, 2, 4 and 8 all have one 1 value and are all placed in group 1.

Phase 1

0	$(0,0,0,0)$	$\surd$
1	$(0,0,0,1)$	$\surd$
2	$(0,0,1,0)$	$\surd$
4	$(0,1,0,0)$	$\surd$
8	$(1,0,0,0)$	$\surd$
5	$(0,1,0,1)$	$\surd$
10	$(1,0,1,0)$	$\surd$
11	$(1,0,1,1)$	$\surd$
15	$(1,1,1,1)$	$\surd$

Phase 2

$0+1$	$(0,0,0,-)$	$\surd$
$0+2$	$(0,0,-,0)$	$\surd$
$0+4$	$(0,-,0,0)$	$\surd$
$0+8$	$(-,0,0,0)$	$\surd$
$1+5$	$(0,-,0,1)$	$\surd$
$4+5$	$(0,1,0,-)$	$\surd$
$2+10$	$(-,0,1,0)$	$\surd$
$8+10$	$(1,0,-,0)$	$\surd$
$10+11$	$(1,0,1,-)$	
$11+15$	$(1,-,1,1)$	

Phase 3

$0+1+4+5$	$(0,-,0,-)$	
$0+2+8+10$	$(-,0,-,0)$	
$0+4+1+5$	$(0,-,0,-)$	duplicate
$0+8+2+10$	$(-,0,-,0)$	duplicate

Table 2.6 Step one of a Quine-McCluskey-based simplification: extraction of prime implicants.

Recall from our simplification using Karnaugh maps that we were able to apply a rule to cover the pair of minterms

$$w \wedge \neg x \wedge y \wedge \neg z$$

and

$$w \wedge \neg x \wedge y \wedge z$$

with one simpler minterm

$$w \wedge \neg x \wedge y$$

because the value of z has no influence on the result. We use a similar basic approach here. We compare each member of the i-th group with each member of the $(i+1)$-th group and look for pairs which differ in only one place; there is no point in comparing members of the i-th group with other groups since we know they cannot

	0	1	2	4	8	5	10	11	15
$0+1+4+5$	$\surd$	$\surd$		$\surd$		$\surd$			
$0+2+8+10$	$\surd$		$\surd$		$\surd$		$\surd$		
$10+11$							$\surd$	$\surd$	
$11+15$								$\surd$	$\surd$

Table 2.7 Step two of a Quine-McCluskey-based simplification: the prime implicants table.

satisfy this criterion since they have different numbers of 1 values. So for example, we compare implicant 0 from group 0 with implicants 1, 2, 4 and 8 from group 1 and find that in each case the pairs differ in just one place. We write these pairs in a new table; this is detailed in the middle, second phase section of Table 2.6. In the new table, we replace the differing value with a $-$ so that, for example, combining implicants 0 and 1, represented by $(0,0,0,0)$ and $(0,0,0,1)$, produces implicant $0+1$ which is represented by $(0,0,0,-)$. Furthermore, when we have used an implicant from the original table to produce a combined one in the new table we place a tick next to it; all of implicants 0, 1, 2, 4 and 8 are ticked as a result of our initial comparisons between groups 0 and 1. This process is repeated for other groups, so groups 1 and 2, 2 and 3 and 3 and 4 in this case. We then iterate the whole procedure to construct further tables until we can no longer combine any implicants. In this case we terminate after three phases as shown in Table 2.6. However, notice that in the third phase we start to introduce duplicate implicants; these duplicates are deleted from further consideration upon detection.

Implicants in any table which are unticked, i.e., have not been combined and used to produce further implicants, are called **prime implicants** and are the ones we need to consider when constructing the final expression for our function. In our example, we have four prime implicants

$$
\begin{array}{ll}
0+1+4+5 & (0,-,0,-) \\
0+2+8+10 & (-,0,-,0) \\
10+11 & (1,0,1,-) \\
11+15 & (1,-,1,1)
\end{array}
$$

To further simplify the resulting expression, the Quine-McCluskey method uses a second step. A so-called prime implicant table is constructed which lists the prime implicants along the side and the original minterms along the top. In each cell where the prime implicant includes the corresponding minterm, we place a $\surd$; this is shown in Table 2.7 for our example. The goal now is to select a combination of the prime implicants which includes, or covers, all of the original minterms. For example, the implicant $0+1+4+5$ covers the prime implicants 0, 1, 4 and 5 so selecting this as well as implicant $10+11$ will cover 0, 1, 4, 5, 10 and 11 in total.

Before we start however, we can identify so-called **essential prime implicants** as being those which are the only cover for a given minterm. Minterm 15 in our example demonstrates this case since it is only covered by prime implicant $11+15$;

it is essential to include this implicant in our final expression as a result. Starting with the essential prime implicants, we can draw a line through the associated row in the prime implicants table. Each time the line goes through a $\surd$, we also draw a line through that column. The intuition behind this is that the resulting lines show which minterms are currently covered by prime implicants we have selected for inclusion in our final expression. Following this procedure in our example, we find that using only implicants $11+15$, $0+1+4+5$ and $0+2+8+10$ we can cover all the original minterms. Looking at the associated vectors, we have

$$\begin{array}{ll} 0+1+4+5 & (0,-,0,-) \\ 0+2+8+10 & (-,0,-,0) \\ 11+15 & (1,-,1,1). \end{array}$$

Treating the entries with $-$ as don't care, we thus implement the final expression for the function r as

$$\begin{array}{rl} r = & (\neg w \wedge \neg y) \vee \\ & (\neg x \wedge \neg z) \vee \\ & (w \wedge y \wedge z) \end{array}$$

which is simpler than our original attempt using Karnaugh maps; we only have three terms in this case ! The reason is obvious when one looks at the original Karnaugh map: we formed a group of 1 cell in the top right-hand corner and a group of 2 cells from the bottom left and right-hand corners. We can easily merge these into a single group of 4 cells which covers all 4 corner cells of the map and overlaps with the group of 4 cells already in the top left-hand corner. Since less groups and larger groups mean simpler implementation, we get a simpler expression: the two terms

$$w \wedge \neg x \wedge \neg y \wedge \neg z$$

and

$$\neg x \wedge y \wedge \neg z$$

have been merged into the single term

$$\neg x \wedge \neg z$$

which covers both.

Hopefully it is clear that the Quine-McCluskey method offers something that Karnaugh maps do not. Although Karnaugh maps are a useful tool for simplifying functions with a small number of inputs by hand, Quine-McCluskey is easy to automate and hence ideal for implementation on a computer which can deal with many more inputs as a result. However, the complexity of the algorithm dictates that the upper limit in terms of number of inputs is quickly reached; logic simplification is classed as an NP-complete problem, and the complexity of Quine-McCluskey is exponential in the number of inputs.

2.2.5 *Physical Circuit Properties*

2.2.5.1 Propagation Delay

CMOS-based logic gates do not operate instantaneously because of the way transistor switching works at the physical level; there is a small delay between when the inputs are supplied and when the output is produced called **propagation delay**. The problem of propagation delay in gates, so-called **gate delay**, is compounded by the fact that there is also a delay associated with the wires that connect them together. A **wire delay** is typically much smaller than gate delay but the same problem applies: a value takes an amount of time proportional to the wire length to travel from one end to the other. Although such delays are extremely small, when many gates are placed in series or wires are very long the delays can add up and produce the problem of circuit **metastability**.

Consider for example the XOR circuit developed previously. So far we have taken a static view of the circuit in the sense that we set some inputs and by assuming that all gates operate instantly, compute the output. The picture changes when we take a dynamic view that includes time; in particular, we see that depending on which time period we sample, the circuit can be in a metastable state which does not correctly represent the required final result. Given the labelled XOR implementation in Figure 2.8, Figure 2.11 highlights this effect. We start at time $t = 70$ns when the circuit is in the correct state given inputs of $x = 0$ and $y = 1$. The inputs are then toggled to $x = 1$ and $y = 1$ at $t = 80$ns but the result does not appear instantly given example propagation delays for NOT, AND and OR gates as 10ns, 20ns and 20ns respectively. In particular, we can examine points in the time-line and show that the final and intermediate results are wrong. For example, it takes until $t = 90$ns before the NOT gates produce the correct outputs and the final result does not change to the correct value until $t = 130$ns; before then the output is wrong in the sense that it does not match the inputs.

One can try to equalise the delay through different paths in the circuit using dummy or **buffer** gates. A buffer simply copies the input to the output and can also amplify the input in some cases; essentially it is performing the identity function. Since the buffer takes some time to operate just like any other gate yet performs no change in input, one can place them throughout the circuit in order to ensure values arrive at the same time. An example is shown in Figure 2.12. Although the circuit may still enter metastable states, we have tamed the problem to some extent by ensuring that values passing into the two AND gates arrive at the same time.

However, even when the effects of propagation delay are minimised it produces a secondary feature that acts as a limit. This feature is the **critical path** of the circuit, the longest sequential sequence of gates in the circuit that dominates the time taken for it to produce a result. In the XOR circuit above, the critical path goes through a NOT gate, an AND gate and then an OR gate: the path has a total delay of 50ns. We will see later how this feature limits the performance of our designs. For now it is enough to believe that the shorter the critical path is, the quicker we can compute

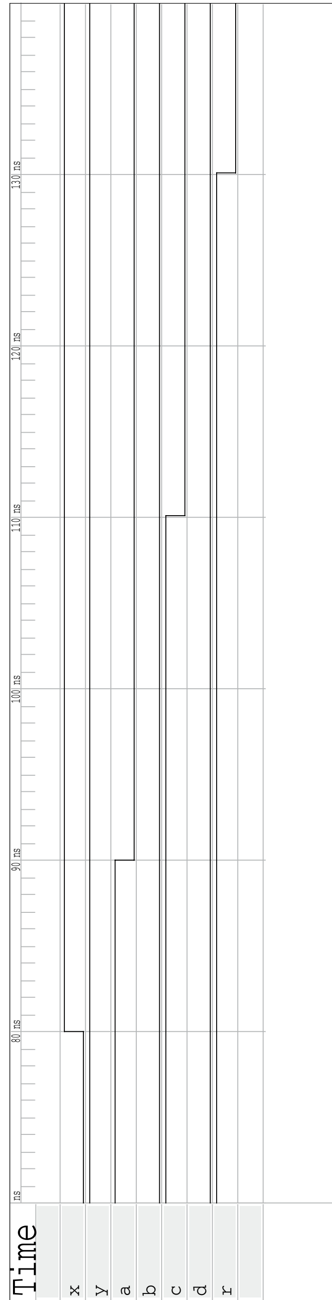


Figure 2.11 A behavioural time-line demonstrating the effects of propagation delay on the XOR implementation.

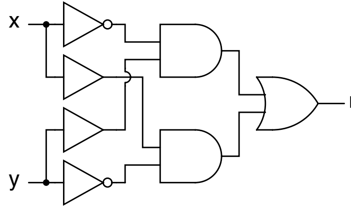


Figure 2.12 Buffering inputs within the XOR gate implementation to equalise propagation delay.

results with our circuit; hence it is advantageous to construct or simplify a design so as to minimise the critical path length.

2.2.5.2 Fan-in and Fan-out

Fan-in and **fan-out** are properties of logic gates relating to the number of inputs and outputs. The problem associated with fan-in and fan-out is that a given logic gate will have physical constraints on the amount of current which can be driven through it. For example, it is common for the output of a source gate to be connected to inputs of several target gates. Unless the source gate can drive enough current onto the output, errors will occur because the target gates will not receive enough of a share to operate correctly; fan-out is roughly the number of target gates which can be connected to a single source. CMOS-based gates have quite a high fan-out rating, perhaps 100 target gates or more can be connected to a single source.

2.2.6 Basic Building Blocks

2.2.6.1 Half and Full Adders

Ultimately, our goal in constructing digital circuits is to do some computation or, more specifically, arithmetic on numbers. The most basic key component one can think of in this context is a 1-bit adder which takes two 1-bit values, say x and y, and adds them together to product a sum and a carry, say s and co. This is commonly termed a **half-adder**, the truth table is shown in Table 2.8a. The truth table makes intuitive sense: if we add $x = 0$ to $y = 1$, the sum is 1 and there is no carry-out; if we add $x = 1$ to $y = 1$, the result is 2 which we cannot represent in 1-bit so the sum is 0 and the carry-out is 1. The half-adder is easily implementable as shown by Figure 2.13a. Essentially we use the same techniques as developed above to extract an expression for each of the outputs in terms of x and y:

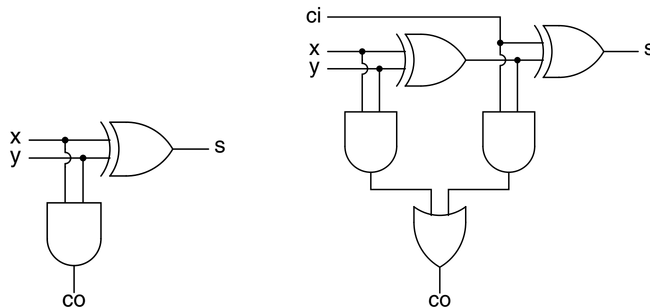
(a) Half-adder

x	y	co	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(b) Full-adder

ci	x	y	co	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 2.8 Truth tables for 1-bit adders.



(a) Half-adder (b) Full-adder

Figure 2.13 Implementation of a 1-bit half-adder and full-adder.

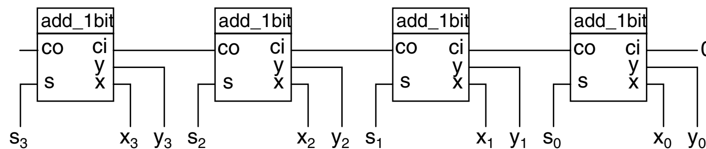
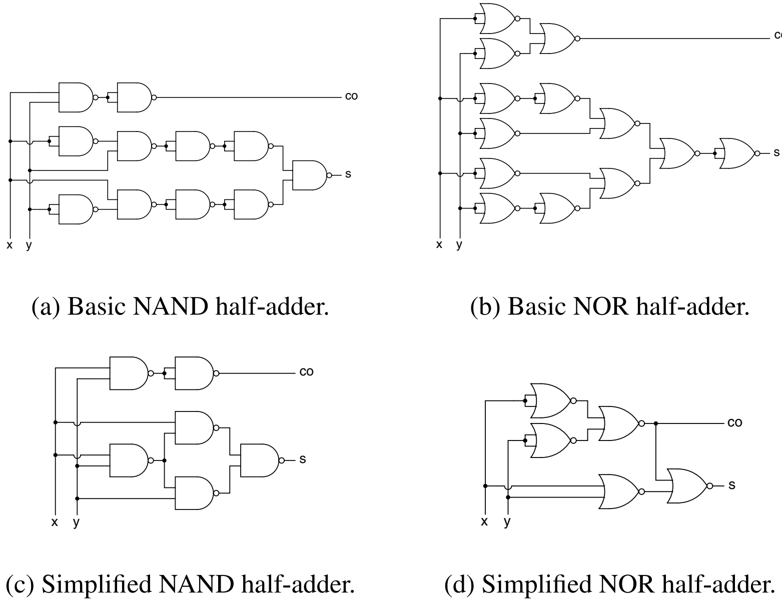


Figure 2.14 A chain of full-adders used to add 4-bit values.



(a) Basic NAND half-adder.

(b) Basic NOR half-adder.

(c) Simplified NAND half-adder.

(d) Simplified NOR half-adder.

Figure 2.15 NAND and NOR gate implementations of a half-adder.

$$co = x \wedge y$$
$$s = x \oplus y.$$

However, the half-adder is only partly useful since in order to add together numbers larger than 1-bit we need a **full-adder** circuit that can accept a carry-in as well as produce a carry-out. The truth table for a full-adder is shown in Table 2.8b. Again extracting an expression for each of the outputs, this time in terms of x, y and ci we find

$$co = (x \wedge y) \vee ((x \oplus y) \wedge ci)$$
$$s = x \oplus y \oplus ci$$

which are a little more complicated. To simplify these expressions we note that there is a shared term $x \oplus y$ which we only need one set of logic for. This produces the circuit shown in Figure 2.13b. Note that the full-adder is essentially two half-adders joined together. Using such components, we will see later that one can build a chain of n 1-bit full-adders that is capable of adding together n-bit values. The basic structure follows that in Figure 2.14 where each of the i full-adders takes bits i of the two inputs x and y and produces bit i of the result; the carry-out of the i-th full-adder is fed into the carry-in of $(i+1)$-th full-adder.

Finally, as an example of how to implement a circuit using only NAND or NOR gates consider translating the half-adder implementation above. Essentially we just use the identities discussed previously in place of each gate in the circuit; this is shown in Figure 2.15a-b. However, once this translation is completed we can start

x	y	equ
0	0	1
0	1	0
1	0	0
1	1	1

(a) Equality.

x	y	lth
0	0	0
0	1	1
1	0	0
1	1	0

(b) less than.

Table 2.9 Truth tables for 1-bit comparators.

to simplify the equations, eliminating and sharing logic. The results are shown in Figure 2.15c-d. We can re-write the expressions for the NOR version of the half-adder as follows:

$$\begin{aligned} co &= \neg x \overline{\vee} \neg y \\ s &= \neg((\neg\neg x \overline{\vee} \neg y) \overline{\vee} (\neg x \overline{\vee} \neg\neg y)) \\ &= \neg((x \overline{\vee} \neg y) \overline{\vee} (\neg x \overline{\vee} y)) \\ &= (x \overline{\vee} y) \overline{\vee} (\neg x \overline{\vee} \neg y) \end{aligned}$$

from which we can then share the term $\neg x \overline{\vee} \neg y$ between co and s. This reduces the number of NOR gates used from thirteen in the original, naive translation to five in the simplified version.

2.2.6.2 Comparators

In the same way as one might want to perform arithmetic on numbers, comparison of numbers is also a fundamental building block within many circuits. With our adder circuits, we looked at the idea of addition of 1-bit numbers with a view to adding n-bit numbers later on. We take the same approach to comparison of numbers by looking at some very simple circuits for performing 1-bit equality and less than comparisons; it turns out we can produce all other comparisons from these two.

Table 2.9 shows the truth tables for 1-bit **equality** and **less than** comparison between two numbers x and y. Both operations produce a 1-bit true or false result, named equ and lth respectively. The tables are self explanatory although maybe a little odd given we are dealing with 1-bit numbers: for example, reading the table for less than we have that 0 is not less than 0, 0 is less than 1, 1 is not less than 0 and 1 is not less than 1. Using the truth tables we can easily produce the following expressions for the two operations:

$$\begin{aligned} equ &= \neg(x \oplus y) \\ lth &= \neg x \wedge y. \end{aligned}$$

s_0	i_1	i_0	r
0	?	0	0
0	?	1	1
1	0	?	0
1	1	?	1

(a) 2-way, 1-bit multiplexer.

s_1	s_0	i_3	i_2	i_1	i_0	r
0	0	?	?	?	0	0
0	0	?	?	?	1	1
0	1	?	?	0	?	0
0	1	?	?	1	?	1
1	0	?	0	?	?	0
1	0	?	1	?	?	1
1	1	0	?	?	?	0
1	1	1	?	?	?	1

(b) 4-way, 1-bit multiplexer.

Table 2.10 Truth tables for 2-way and 4-way, 1-bit multiplexers.

2.2.6.3 Multiplexers and Demultiplexers

A **multiplexer** is a component that takes n control inputs and uses them to decide which one of 2^n inputs is fed through to the single output. A **demultiplexer** does the opposite, it still takes n control inputs but uses them to decide which one of 2^n outputs is fed with the single input. We say that a multiplexer or demultiplexer is n-way if it accepts or produces n inputs or outputs, and m-bit if those inputs and outputs are m bits in size.

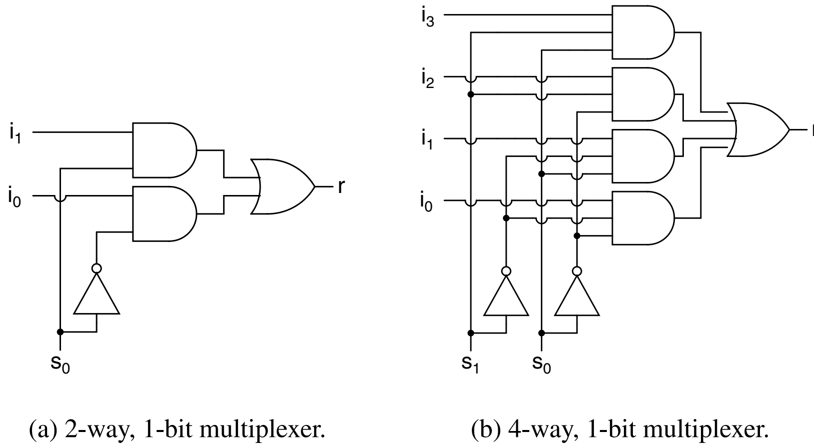
Table 2.10a shows the truth table for a 2-way, 1-bit multiplexer with inputs i_0 and i_1, a control signal s_0 and an output r. The table shows that when $s_0 = 0$ we feed the value of i_0 through to the output, i.e., $r = i_0$, while if $s_0 = 1$ we have that $r = i_1$. This behaviour is easily implemented using the expression

$$\begin{aligned} r = (\ \neg s_0 \wedge i_0\ ) \vee \\ (\ s_0 \wedge i_1\ ) \end{aligned}$$

which is shown diagrammatically in Figure 2.16a.

A natural question arises given this simple starting point, namely how do we extend the design to produce n-way, m-bit alternatives ? Building an m-bit multiplexer is easy, we simply have m separate 1-bit multiplexers where multiplexer i takes the i-th bit of each input and produces the i-th bit of the output. This technique is the same no matter how many inputs we have and is sometimes termed **bit-slicing** since we chop up the work into bit-sized chunks.

We can produce a multiplexer design with double the number of inputs using two techniques: either produce a large truth table along the same lines as Table 2.10a but with four inputs and two control signals, or re-use our 2-way multiplexer. Considering the truth table for a 4-way multiplexer in Table 2.10b, we can derive an expression for r as



(a) 2-way, 1-bit multiplexer.

(b) 4-way, 1-bit multiplexer.

Figure 2.16 Implementations for 2-way and 4-way, 1-bit multiplexers.

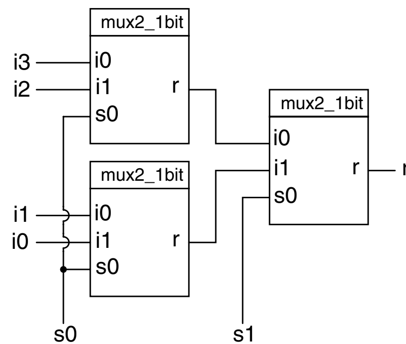


Figure 2.17 Using cascading to build a 4-way multiplexer from 2-way building blocks.

$$
\begin{aligned}
r = & (\neg s_0 \wedge \neg s_1 \wedge i_0) \vee \\
& (s_0 \wedge \neg s_1 \wedge i_1) \vee \\
& (\neg s_0 \wedge s_1 \wedge i_2) \vee \\
& (s_0 \wedge s_1 \wedge i_3) .
\end{aligned}
$$

This is more complicated than our 2-way multiplexer but still offers easy and efficient implementation as shown in Figure 2.16b; hopefully it is clear that one can take this general pattern and extend it to even more inputs. However, in doing so the truth table and corresponding implementation become more and more complicated to the point of being unmanageable. To address this issue we can simply re-use multiplexers with smaller numbers of inputs to build components with more inputs using a technique called **cascading**. The idea is to split the large decisional task of

s_0	i	r_1	r_0
0	0	?	0
0	1	?	1
1	0	0	?
1	1	1	?

(a) 2-way, 1-bit demultiplexer.

s_1	s_0	i	r_3	r_2	r_1	r_0
0	0	0	?	?	?	0
0	0	1	?	?	?	1
0	1	0	?	?	0	?
0	1	1	?	?	1	?
1	0	0	?	0	?	?
1	0	1	?	1	?	?
1	1	0	0	?	?	?
1	1	1	1	?	?	?

(b) 4-way, 1-bit demultiplexer.

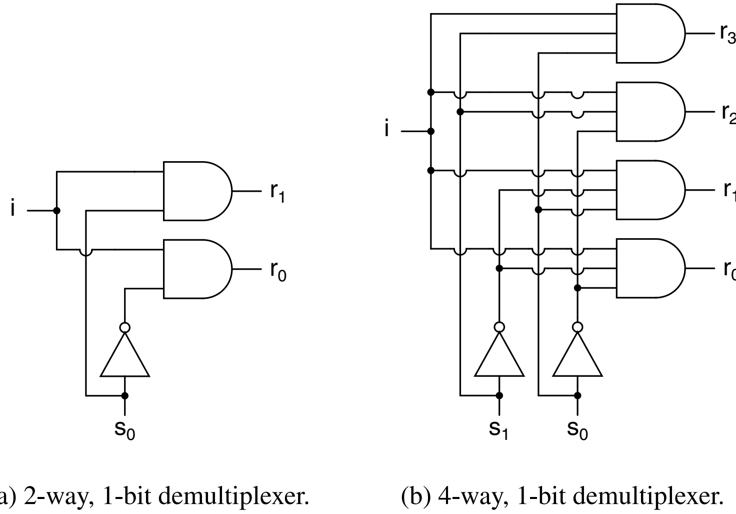
Table 2.11 Truth tables for 2-way and 4-way, 1-bit demultiplexers.

which input to feed through into smaller tasks. Figure 2.17 demonstrates this idea by building a 4-way multiplexer from 2-way multiplexer building blocks. The control signal s_0 is now used to manage the first two multiplexers; the top one produces i_3 if $s_0 = 1$ or i_2 or $s_0 = 0$, the bottom one produces i_1 if $s_0 = 1$ or i_0 or $s_0 = 0$. These outputs are fed into a final multiplexer which uses s_1 to select the appropriate input. For example, if $s_0 = 0$ and $s_1 = 1$ the top multiplexer produces i_2 while the bottom one produces i_0; the final multiplexer selects i_2 and feeds it to the output; this is what we expect for the given control signals. By using a similar approach one can construct multiplexers with even more inputs, for example a 16-way component using 4-way building blocks.

Having discussed multiplexers, demultiplexers are somewhat trivial since they are essentially just multiplexers in reverse; Table 2.11 details the truth tables for the 2-way and 4-way, 1-bit components which are implemented in Figure 2.18. Note that the same techniques of bit-slicing and cascading can be applied to demultiplexers as to multiplexers in order to produce n-way, m-bit components.

2.2.6.4 Encoders and Decoders

What we normally called **encoder** and **decoder** devices are, roughly speaking, like specialisations of the multiplexer and demultiplexer devices considered earlier. Strictly speaking we usually say decoders translate, or map, an n-bit input into one of 2^n possible output values; encoders perform the converse by translating one of 2^n possible input values into an n-bit output. Of course, some inputs or outputs might be ignored in order to relax this restriction; more generally we say an encoder or decoder is an ***n*-to-*m*** device if it has n inputs and m outputs, a 4-to-2 encoder for example. Some decoders (resp. encoders) demand that exactly one bit of their output (resp. input) is 1 at a time, these are normally called **one-of-many** devices although the term is often dropped when the assumption is that all devices are one-of-many.



(a) 2-way, 1-bit demultiplexer. (b) 4-way, 1-bit demultiplexer.

Figure 2.18 Implementations for 2-way and 4-way, 1-bit demultiplexers.

i_1	i_0	r_3	r_2	r_1	r_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

(a) 2-input one-of-many decoder.

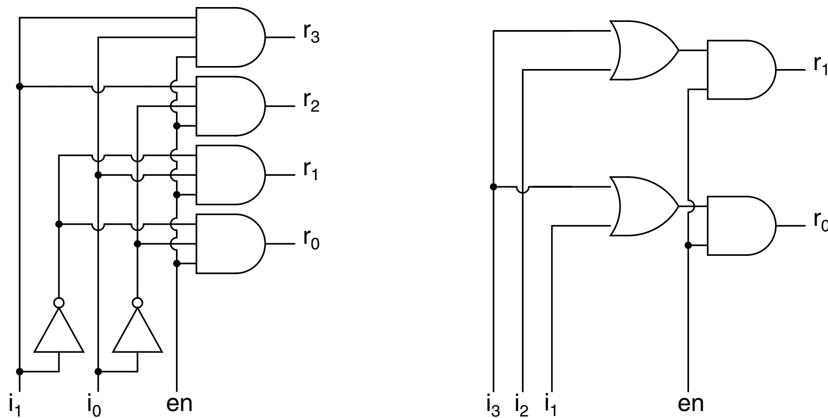
i_3	i_2	i_1	i_0	r_1	r_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

(b) 2-output one-of-many encoder.

Table 2.12 Truth tables for a simple 2-input decoder and 2-output encoder.

Table 2.12a details the truth table for a simple one-of-many decoder circuit; notice that the structure is similar to a demultiplexer with the inputs tied to suitable values. This type of decoder is often used to control other circuits. Consider the case of having say four circuits in a system, only one of which should be active at a time depending on some control value. A decoder could implement this behaviour by taking the control value as input and driving 1 onto an output to activate the associated circuit. The decoder drives 1 onto the output number associated with the inputs i_0 and i_1. That is, if one considers the 2-bit value $i = (i_1, i_0)$, the decoder drives 1 onto output number i and 0 onto all other outputs.

Considering this example again, there may be occasions where no circuit should be active; it is therefore typical to equip the decoder with an **enable** input which, when set to 0, forces all outputs to 0 thus disabling any control behaviour. This is easily realised by adding an extra input *en* to the circuit which is ANDed with the outputs. Expressions for $r_0 \ldots r_3$ can be written as



(a) 2-input one-of-many decoder. (b) 2-output one-of-many encoder.

Figure 2.19 Implementation of a simple 2-input decoder and 2-output encoder.

$$
\begin{aligned}
r_0 &= en \wedge \neg i_0 \wedge \neg i_1 \\
r_1 &= en \wedge i_0 \wedge \neg i_1 \\
r_2 &= en \wedge \neg i_0 \wedge i_1 \\
r_3 &= en \wedge i_0 \wedge i_1
\end{aligned}
$$

with the resulting implementation shown in Figure 2.19a.

Table 2.12a details the truth table for the corresponding encoder circuit which is implemented in Figure 2.19a. The idea is to perform the converse translation: take the inputs $i = \langle i_3, i_2, i_1, i_0 \rangle$ and decide which value the decoder would have mapped them to. This is somewhat easier to construct because the many-of-one property ensures exactly one of $i_0 \ldots i_3$ is equal to 1; expressions for r_0 and r_1 can be written as

$$
\begin{aligned}
r_0 &= en \wedge (i_1 \vee i_3) \\
r_1 &= en \wedge (i_2 \vee i_3)
\end{aligned}
$$

with the resulting implementation shown in Figure 2.19b.

In this context, it is also useful to consider what is termed a **priority encoder**. In a one-of-many encoder, exactly one input bit is allowed to be 1 but in practise this is hard to ensure. To cope with this fact, we might like to give some priority to the various possibilities. For example, if both bits $i_0 = 1$ and $i_1 = 1$ then the normal one-of-many encoder fails, we might like to say for example that we give priority to i_1 in this case and set the output accordingly. More generally, we are saying that input bit i_{j+1} has a higher priority that input bit i_j (although other schemes are obviously possible). Reading the truth table detailed in Table 2.13 enforces this relationship.

i_3	i_2	i_1	i_0	r_1	r_0
0	0	0	?	0	0
0	0	1	?	0	1
0	1	?	?	1	0
1	?	?	?	1	1

Table 2.13 Truth table for a simple 2-output priority encoder.

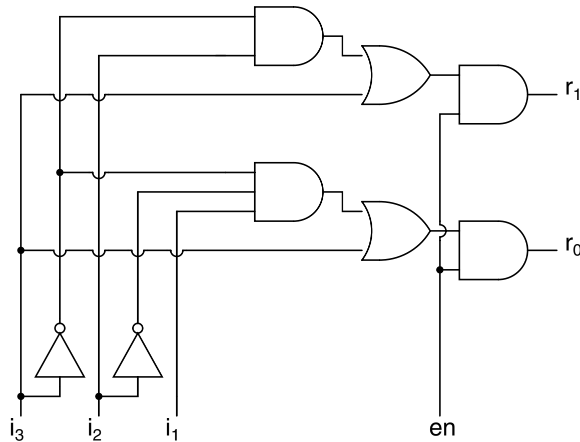


Figure 2.20 Implementation of 2-output priority encoder.

Take the bottom row for example: although potentially $i_0 = 1$, $i_1 = 1$ or $i_2 = 1$ the output gives priority to i_3. The resulting implementation of the expressions

$$\begin{aligned} r_0 &= (i_1 \wedge \neg i_2 \wedge \neg i_3) \vee (i_3) \\ r_1 &= (i_2 \wedge \neg i_3) \qquad\quad \vee (i_3) \end{aligned}$$

is shown in Figure 2.20.

As a concrete example, consider the task of controlling a 7-segment LED as found in many calculators and shown in Figure 2.21. The device has seven segments labelled $r_0 \ldots r_6$ which are controlled individually in order to create a suitable image: in our case we deal only with images related to decimal digits $0 \ldots 9$. We need $n = 4$ bits to represent a decimal digit. Therefore a decoder to translate from a digit to the corresponding image will have a 4-bit input and $2^4 = 16$ outputs; an encoder to perform the reverse translation from an image to the corresponding digit will have a 16-bit input and 4-bit output.

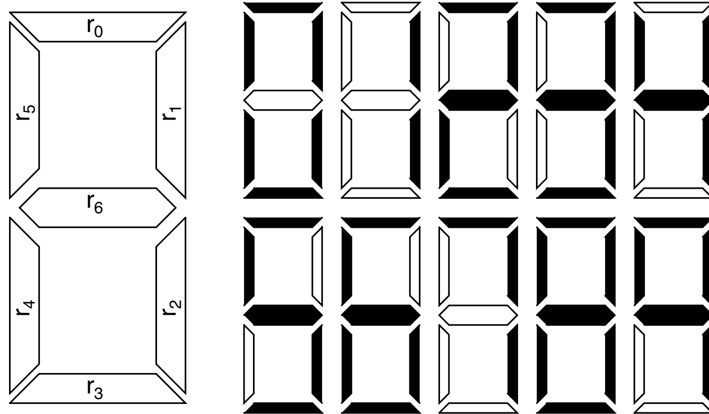


Figure 2.21 A 7-segment LED.

i_3	i_2	i_1	i_0	r_6	r_5	r_4	r_3	r_2	r_1	r_0
0	0	0	0	0	1	1	1	1	1	1
0	0	0	1	0	0	0	0	1	1	0
0	0	1	0	1	0	1	1	0	1	1
0	0	1	1	1	0	0	1	1	1	1
0	1	0	0	1	1	0	0	1	1	0
0	1	0	1	1	1	0	1	1	0	1
0	1	1	0	1	1	1	1	1	0	1
0	1	1	1	0	0	0	0	1	1	1
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	0	0	1	1	1

Table 2.14 Truth table for a 7-segment LED decoder.

Table 2.14 describes a truth table for a decoder which could be used to control the LED. It leaves out unused input and output combinations for brevity. For example, we do not use the possible digits $10 \ldots 15$ and some of the potential segment numbers are not used; there are no segments $r_7 \ldots r_{15}$. In reality, these relate to don't care states. To implement the decoder, one can derive a logical expression for each of the outputs via a Karnaugh map or similar technique

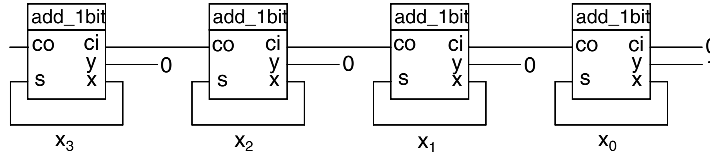


Figure 2.22 A flawed, nonsensical example of a 4-bit counter where we neither initialise the value *x or* allow it to settle before incrementing it.

$$
\begin{aligned}
r_0 &= i_3 && \lor i_1 && \lor (i_0 \land i_2) && \lor (\neg i_0 \land \neg i_2)\\
r_1 &= i_3 && \lor \neg i_2 && \lor (i_0 \land i_1) && \lor (\neg i_0 \land \neg i_1)\\
r_2 &= i_3 && \lor \neg i_1 && \lor (\neg i_3 \land i_2) && \lor (i_0 \land i_1)\\
r_3 &= (\neg i_0 \land \neg i_2) && \lor (\neg i_0 \land i_1) && \lor (i_0 \land \neg i_1 \land i_2) && \lor (i_0 \land i_1 \land \neg i_2)\\
r_4 &= (\neg i_0 \land \neg i_2) && \lor (\neg i_0 \land i_1) && &&\\
r_5 &= i_3 && \lor (\neg i_0 \land \neg i_1) && \lor (\neg i_1 \land i_2) && \lor (\neg i_0 \land i_2)\\
r_6 &= (\neg i_0 \land i_1) && \lor (\neg i_2 \land i_3) && \lor (\neg i_1 \land i_2) && \lor (i_1 \land \neg i_2 \land \neg i_3)
\end{aligned}
$$

Implementation of the decoder is then simply implementation of the expressions with suitable sharing of common terms to reduce the number of gates required.

2.3 Clocked and Stateful Logic

Combinatorial logic is continuous in the sense that the output is always being computed, subject to propagation delay, as a product of the inputs. One of the limitations of this sort of component is that it cannot store state, it has no memory. As an example of why this could be an issue, consider the problem of implementing a counter that repeatedly computes $x \leftarrow x+1$ for some 4-bit value x. That is, it takes x, increments it and stores the result back into the value x ready to start again. Given we already have a 1-bit adder component, a naive first attempt at realising such an operation might be to build the component shown in Figure 2.22. Essentially we take the value x, feed it into the adder and loop the adder output back into the input ready to start again.

There are two *major* problems with this approach. Firstly, we have not given x and initial value so it is unclear what the input to the adder will be when we start. Secondly and more importantly, we never actually store the value x. The value fed to the input of the adder is continuously being updated by the output; as the output changes the input changes and so the output changes again, the circuit is never able to settle into a consistent state. These problems highlight a big difference between designing hardware and writing software. In a language such as C, we are given variables which can store or remember values for us; sequences of statements are guaranteed to execute in order with each only being executed once previous ones

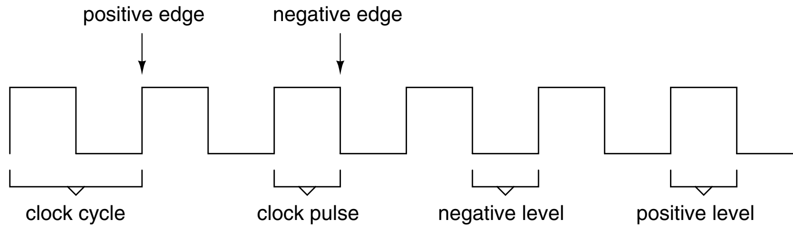


Figure 2.23 Features of a clock signal.

have completed. This section aims to address this imbalance by showing how we can build components to store values and how to perform sequences of operations in a controlled manner.

2.3.1 Clocks

In order to execute a sequence of operations correctly, we need some means of controlling and synchronising them. The concept of a **clock** is central to achieving this. When we say clock in digital circuit design we usually mean a signal that oscillates between 0 and 1 in a regular fashion rather than something that tells the time. This sort of clock is more like a metronome; ticks of the clock act as synchronising events that keep other components in step with each other.

A **clock signal** is just the same as any other value within a logic design. As such, there are several items of terminology that relate to general values but are best introduced in the context of clocks. Figure 2.23 details these features. Firstly, we assume the clock signal approximates a square wave so the signal is either 0 or 1. The physical characteristics of these components mean this is dubious in reality (the corners of the signal can be "rounded"), but suffices for the discussion here. Transitions of the clock from 0 to 1 are called positive **clock edges**, transitions from 1 to 0 are negative clock edges. At any point while the clock is 1 we say it is at a positive **level**, while the clock is 0 it is at a negative level. The interval between one positive or negative edge and the next positive or negative edge respectively is termed a **clock cycle**; the time taken for one clock cycle to happen is the **clock period** and the number of clock cycles that occur each second is the **clock frequency** or **clock rate**. The region between a positive and negative clock edge is termed a **clock pulse**. Note that the time the clock signal spends at positive and negative levels need not be equal; the term **duty cycle** is used to describe the ratio between these times. The clocks we use will typically have a duty cycle of a half meaning the signal is at a positive level for the same time it is at a negative level.

The basic idea in using clocks to synchronise events within a circuit is that on a clock edge we present values to our combinatorial logic in order to do some com-

C_3	C_2	C_1	C_0
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

Table 2.15 Values of bits in a 4-bit counter as used for clock division.

putation with them. These values are only updated on clock edges, as a result the combinatorial logic has one clock cycle to settle into a stable state; note that this differs from the flawed example above where values are updated instantaneously. As such, the faster our clock frequency is, the faster we perform computation. However, there are two limits to how a given circuit can be clocked. Firstly, since the combinatorial logic must compute a new result within one clock cycle, the clock period must be greater than the critical path of that logic. Recall that the critical path dominates the time taken to compute a result, if it is longer than the clock period then the result required to update our value with will not be ready. This is a crucial fact: to perform computation at high speeds it is vital that the critical path of the combinatorial components is minimised so the clock frequency can be maximised. Secondly, we must consider the effects of propagation delay in wires to prevent a phenomenon called **clock skew**. Clock skew is simply the fact that if a clock signal arrives at one point along a slightly longer wire than another point, the times of arrival will differ slightly; the clocks will be skewed or unsynchronised. Depending on the situation this might present a problem because the circuit components are no longer perfectly in step.

Although the clock is just any other value, we need it to oscillate in a regular and reliable manner. As such, clock signals are usually generated externally to a given circuit and fed into it. A common method for generating the required behaviour is to use a piezoelectric crystal which, when a voltage is applied across it, oscillates according to a natural frequency related to the physical characteristics of the crys-

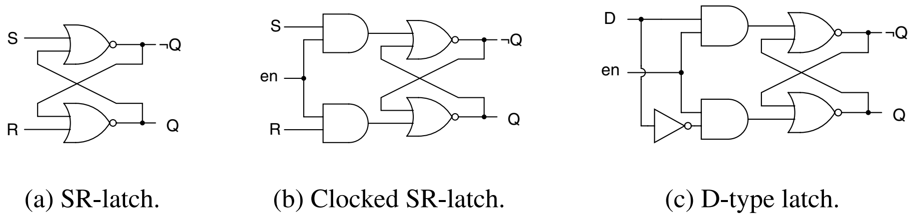


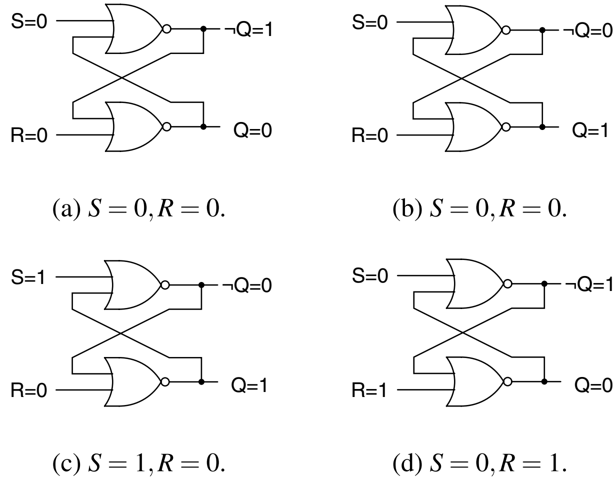
Figure 2.24 Three variations of SR-latch.

tal. Roughly speaking, one can use the resulting electrical field generated by this oscillation as the clock signal.

Multiplying a reference clock, or increasing the frequency, is best done using a dedicated component and is somewhat beyond the scope of discussion here. Dividing a reference clock, or decreasing the frequency, is achieved much more easily. Imagine that on each positive edge of our original clock, a 4-bit counter C is incremented. Table 2.15 shows the progression of the counter with row i representing the value of bits in C after i clock edges have passed. The value of bit C_0 oscillates from 0 to 1 at half the same speed as the original clock; the value of bit C_1 takes twice as long again, it oscillates from 0 to 1 at a quarter of the speed as the original clock. Thus by using our counter and examining specific bits, we have created a component which can divide the clock frequency by powers-of-two. Specifically, if we want to divide the clock by 2^j (with $j > 0$), we use a j-bit counter and examine the $(j-1)$-th bit.

2.3.2 Latches

The first step toward building a component which can remember or store values is somewhat counter-intuitive. We start by looking at the Set-Reset latch, usually called an **SR-latch**, as shown in Figure 2.24a. The circuit has two inputs called S and R which are the set and reset signals, and two outputs Q and $\neg Q$ which are always the inverse of each other. The circuit design seems a little odd compared to what we have seen so far because we have introduced a sort of loop; the outputs of each NOR gate are wired to the input of the other. The result of this odd design is that the outputs of the circuit are not uniquely defined by the inputs. For example, when we set the inputs to $S = 0$ and $R = 0$, either of the two situations in Figure 2.25a-b is valid; the circuit is in a stable state in both cases. When we set the inputs to $S = 1$ and $R = 0$, we force the latch into the state shown in Figure 2.25c; Q is set to 1 and $\neg Q$ to 0 whatever they were before because the top NOR gate will have to output 0. Conversely, when we set the inputs to $S = 0$ and $R = 1$ we force the latch into the state shown in Figure 2.25d; Q is set to 0 and $\neg Q$ to 1 because the bottom NOR gate



(a) $S = 0, R = 0$. (b) $S = 0, R = 0$.

(c) $S = 1, R = 0$. (d) $S = 0, R = 1$.

Figure 2.25 Stable states for the SR-latch.

		current		*next*	
S	R	Q	$\neg Q$	Q	$\neg Q$
0	0	0	1	0	1
0	0	1	0	1	0
0	1	?	?	0	1
1	0	?	?	1	0
1	1	?	?	?	?

(a) SR-latch

		current		*next*	
J	K	Q	$\neg Q$	Q	$\neg Q$
0	0	0	1	0	1
0	0	1	0	1	0
0	1	?	?	0	1
1	0	?	?	1	0
1	1	0	1	1	0
1	1	1	0	0	1

(b) JK-latch

Table 2.16 Truth tables for an SR-latch and a JK-latch.

will have to output 0. The final issue is what happens if we set the inputs to $S = 1$ and $R = 1$. In this case, the circuit enters a contradictory or undefined state: both outputs should become 0, yet we know they must also be the inverse of each other.

Defining the metastable state $S = 1, R = 1$ as unknown, the behaviour of the SR-latch can be described by Table 2.16a. Note that this differs somewhat from previous truth tables because there is some notion of state in Q and $\neg Q$. Their value when they are next latched depends partly on their current value. Essentially we have a component that we can set into a defined state using the set and reset inputs: if we want the output Q to be 1 we set $S = 1, R = 0$ for a while; if we want the output to be 0 we set $S = 0, R = 1$ for a while. The key thing to notice is that after waiting a while, when we return the inputs to $S = 0, R = 0$ the circuit remembers which of S

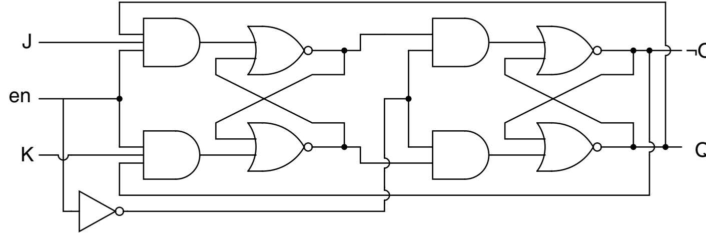


Figure 2.26 Implementation of a master-slave JK-latch.

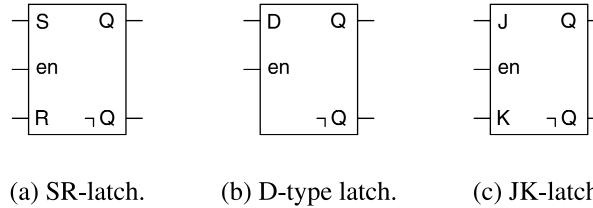
and *R* was set last. It offers a means of storing or latching a 1-bit value provided we control it correctly.

One problem with the basic SR-latch design is that it cannot currently be synchronised with other components in the system, i.e., the latched value can be changed at *any* time. It is often convenient to constrain when the value can be changed; as a result we introduce the variation shown in Figure 2.24b. The basic idea is to AND an enable signal *en* with the *S* and *R* inputs to provide the latch inputs. By doing so, the latch is only sensitive to changes in *S* and *R* during positive levels of *en*; when $en = 0$ it does not matter what *S* and *R* are, the AND gates simply output 0. Since *en* will often be a clock signal, but is not required to be, we call this component a **clocked SR-latch**. In this context, the latched value can only change value during the positive phase of a clock cycle.

As a side note, we usually call components which are sensitive to inputs during positive or negative levels of an enable signal **active high** and **active low** respectively. The choice of which to use is usually dictated by cost of implementation and ease of integration with other components: there is no real advantage in using one over the other from any other point of view.

Using either an active high or active low approach, a further problem with our clocked SR-latch is the potential to enter an unattractive, metastable state when $S = 1$ and $R = 1$. The easiest way to avoid this situation is to ensure it never occurs; we can do this by allowing one input *D* and using $S = D$ and $R = \neg D$ as inputs to a clocked latch. This approach is shown in Figure 2.24c. Clearly we can now never have that $S = 1$ while $R = 1$ and so avoid the potential problem. However, we can also not have that $S = 0$ while $R = 0$ which was the state which remembered the input. This design, termed a **D-type latch**, is a component which operates such that when $D = 1$ and $en = 1$ the output $Q = 1$, while if $D = 0$ and $en = 1$ the output $Q = 0$. If $en = 0$ the D-type latch simply maintains the current state. We now have a reliable 1-bit memory. When we want to latch a new value we place it on the input *D* and send a clock pulse into *en*; the currently stored value is always available on the output *Q*.

An alternative approach to solving the metastable state problem is to use the input $S = 1$, $R = 1$ for some other purpose. The **JK-latch** is an example of this



(a) SR-latch. (b) D-type latch. (c) JK-latch.

Figure 2.27 Symbolic representations of different latches.

which uses the state to toggle the outputs Q and $\neg Q$; when $S = 1$ and $R = 1$ the outputs swap over as described in Table 2.16b. Although other methods exist, the classic implementation of a JK-latch uses two SR-latches in series: the first is called the master while the second is called the slave, an arrangement termed a master-slave JK-latch and is shown in Figure 2.26. By inverting *en* and feeding the result to the slave SR-latch, we enable the master SR-latch during positive levels of the enable signal while disabling the slave. When the enable signal changes, the value held in the master SR-latch is transferred to the slave. Roughly speaking, since there is no longer a feedback path directly from one latch into itself, the metastable state is eliminated. Or, by feeding Q and $\neg Q$ back into the input AND gates we can never feed $S = 1$ and $R = 1$ because at least one of Q and $\neg Q$ will be 0 in the slave latch.

We typically use the symbols in Figure 2.27 to represent the different latches presented here so as to simplify the description of large designs. It is common to further simplify the symbols by omitting $\neg Q$ unless it is explicitly required.

2.3.3 Flip-Flops

Latches are **level triggered** in the sense that their value can be changed when the enable signal is at a positive or negative level depending on whether then are active high or low. This partly solves the problem of synchronisation but it is often attractive to constrain things even further and say that the value should only be able to change at a positive or negative *edge* rather than at any time during the positive or negative *level*. Altering our basic latch component so that it is **edge triggered** produces a variation called a **flip-flop**.

A basic approach for changing a latch into a flip-flop is to construct a pulse generator circuit which remains at a high-level for a very short period of time after the enable signal goes high. We use this pulse generator to drive a level triggered latch; the reasoning is that if the pulse is short enough, the active period of the latch is essentially limited to the positive edge of the enable signal. Figure 2.28 shows an implementation of such a device while Figure 2.29 details the resulting behaviour assuming propagation delays for NOT and AND gates are again 10ns and 20ns respectively. Propagation in the NOT gate is what makes the circuit work, when *en*

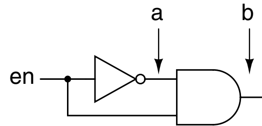


Figure 2.28 Implementation of a simple pulse generator.

is set to 1 at time $t = 10$ns there is a 10ns delay before a changes. This delay means that for a short time equal to the propagation delay of the NOT gate, both en and a are equal to 1 meaning that b is also 1. The change in b is delayed by 20ns but the short pulse we are looking for is still evident; the pulse width is equal to the propagation delay of the NOT gate, it is skewed by a distance equal to the propagation delay of the AND gate. Notice that when en is set to 0 again at time $t = 70$ns we do not get a similar pulse: this generator produces positive edge triggered devices when coupled to it. Thus, as long as the propagation delay of the NOT gate is small enough, we can alter, for example, a standard D-type latch to produce an edge triggered D-type flip-flop shown in Figure 2.30.

We again use symbols, shown in Figure 2.31, to represent the different flip-flop types. The only difference from the symbol for the corresponding latch is the inclusion of a vee-shaped marker on the en input which denotes edge triggering rather than level triggering.

2.3.4 State Machines

Definition 31. A **Finite State Machine (FSM)** is formally defined by a number of parts:

- Σ, an input alphabet.
- Q, a finite set of states.
- $q_0 \in Q$, a start state.
- $A \subseteq Q$, a set of accepting states.
- A function $\delta : Q \times \Sigma \to Q$ which we call the transition function.

More simply an FSM is just a directed graph where moving between nodes (which represent each state) means consuming the input on the corresponding edge. As an example, consider a simple state machine that describes a vending machine that sells chocolate bars. The machine accepts tokens worth 10 or 20 units but does not give change. When the total value of tokens entered reaches 30 units it delivers a chocolate bar but it does not give change; the exact amount must be entered otherwise an error occurs, all tokens are ejected and we start afresh.

From this description we can start defining the state machine components. The input alphabet is simply the possible input values, in this case the different tokens

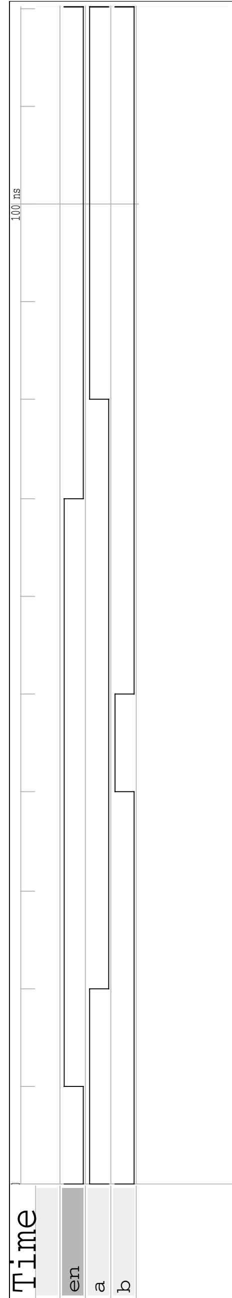


Figure 2.29 Behavioural time-line for a simple pulse generator.

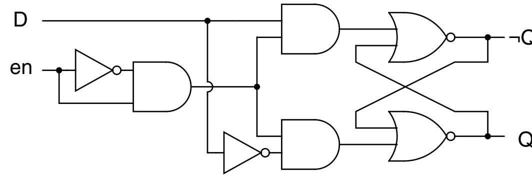
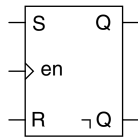
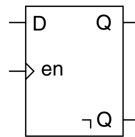


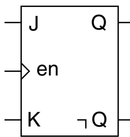
Figure 2.30 Implementation of a D-type flip-flop.



(a) SR flip-flop.



(b) D-type flip-flop.



(c) JK flip-flop.

Figure 2.31 Symbolic representations of different flip-flops.

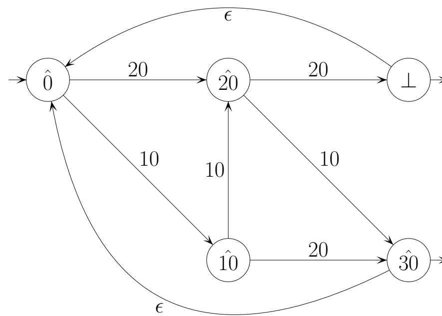


Figure 2.32 A graph describing the FSM for a chocolate vending machine.

such that $\Sigma = \{10, 20\}$. The set of states the machine can be in is easy to imagine: it can either have tokens totalling 0, 10, 20 or 30 units in it or be in the error state which we denote by $\perp$. Thus, using a hat over numbers to differentiate between states and input token values, we have that $Q = \{\hat{0}, \hat{10}, \hat{20}, \hat{30}, \perp\}$ where $q_0 = \hat{0}$ since we start with 0 tokens in the machine. The accepting state is $\hat{30}$ since this represents the state where the user has entered a valid sequence of tokens and obtains a chocolate bar. Finally we need to define the transition function δ which is best described diagrammatically as in Figure 2.32. From this we see that, for example, if we are in state $\hat{20}$ and accept a 10 unit token we end up in state $\hat{30}$, while if we accept a 20 unit token we end up in the error state. Note that the input marked ε is the empty

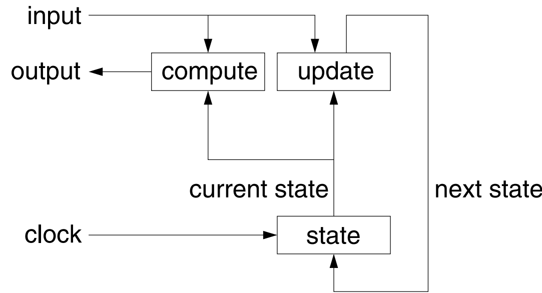


Figure 2.33 The general structure of an FSM in digital logic.

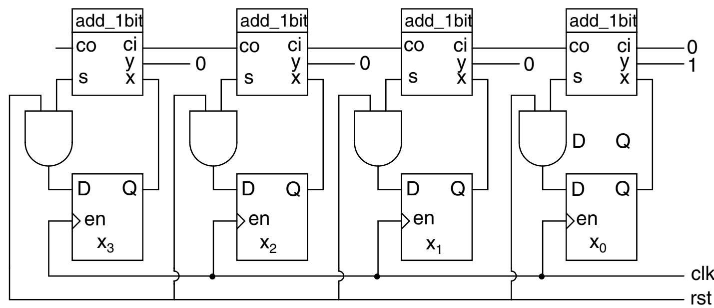


Figure 2.34 A corrected example of a 4-bit counter.

input; that is, with no input we can move between the accepting or error states back into the start state thus resetting the machine.

From this description it should start to become obvious why state machines play a big part in digital logic designs. Given we now have components that allow us to remember some state, as long as we can construct combinatorial logic components to compute the transition function δ, and a clock to synchronise components, we can think of a design that allows us to step through sequences of operations. A general structure for digital logic state machines is shown in Figure 2.33. A block of edge triggered flip-flops is used to store the current state; this is fed into two combinatorial components whose task it is to update the state by applying the transition function and compute any outputs based on the state we are in. On each positive clock edge our flip-flops take the next state and store it, overwriting the current state. Thus the design steps into a new state under control of the clock.

Using this general structure we can finally solve the problem outlined at the beginning of the section: how to build a 4-bit counter. Clearly we need four flip-flops to store the 4-bit state of our counter; these constitute the state portion of the general structure. We do not require any additional output from the circuit, simply that it cy-

State	Next State	Main Road	Annex Road
0	1	green	red
1	2	amber	red
2	3	red	amber
3	4	red	green
4	5	red	amber
5	0	amber	red

Table 2.17 Description of the traffic light controller state machine.

S_2	S_1	S_0	S'_2	S'_1	S'_0	M_r	M_a	M_g	A_r	A_a	A_g
0	0	0	0	0	1	0	0	1	1	0	0
0	0	1	0	1	0	0	1	0	1	0	0
0	1	0	0	1	1	1	0	0	0	1	0
0	1	1	1	0	0	1	0	0	0	0	1
1	0	0	1	0	1	1	0	0	0	1	0
1	0	1	0	0	0	0	1	0	1	0	0

Table 2.18 Truth table for the traffic light controller state machine.

cles through values in turn. Thus we only need a simple update portion built from a 4-bit adder as before. Figure 2.34 demonstrates the final design. When the reset signal *rst* is 0, the counter is set to 0 on the following positive clock edge. When the reset signal is 1, the counter is updated on each positive clock edge by feeding the current value through the adder. Since there is a whole clock cycle between presenting the current value and storing the new value, the adder can settle in a stable state and we get the correct result.

As a more complete example of state machines in this context, consider the problem of controlling a set of traffic lights which prevent motorists on a main road crashing into those turning on to it from an annex road. The idea is that while traffic is set flowing on the main road via a green light or go sign, the annex road drivers should be shown a red light or stop sign. Eventually drivers on the annex road get a turn and the roles are reversed. An amber light is added to the system to offer a period of changeover where both roads are warned that the lights are changing.

Table 2.17 offers a description of the problem. It describes a state machine with six states and, given a current state, which state we should move into next: there is no input in this machine, each of the transitions in the function δ being ε type and based only on the state we are currently in. Furthermore, it shows what lights should be on given the current state. So if we are in state 0, for example, we should move to state 1 next and the main and annex road lights should be green and red respectively. We could draw a graph of the state machine but it would be a bit boring since it simply cycles through the six inputs in sequence.

The states can be represented as 3-bit integers since there are six states and $2^3 > 6$. We use S_0, S_1 and S_2 to denote the current state bits and S'_0, S'_1 and S'_2 to denote the next state bits. There are nine outputs from our combinatorial logic, three outputs for the next state and six outputs to represent the red, amber and green lights on each of the roads. We use M_r, M_a and M_g to denote red, amber and green lights on the main road and A_r, A_a and A_g to denote red, amber and green lights on the annex road.

As a side note, the encoding of our states is important in some cases; we have simply used the binary representation of each state number but more involved schemes exist. For example, one could use a Gray code, mentioned earlier when we introduced Karnaugh maps. Using a Gray code to represent states means that only one bit changes between adjacent states: this might reduce transistor switching and thus improve the power efficiency of our design. Alternatively one can consider a technique called **one-hot coding**. Essentially this uses one flip-flop for each state; when the *i*-th flip-flop stores 1, we are in state *i*. The state machine is reset by setting all flip-flops to 0 except the first flip-flop which is set to 1 to store the hot bit. The flip-flops are chained together so that on each clock edge the hot bit is passed to the next flip-flop and we transfer from state to state. One advantage of this approach is that decoding which state we are in is made much easier; one disadvantage is that we typically use more logic to store the state.

From this tabular description of the traffic light problem we can derive a truth table, shown in Table 2.18, for each of our outputs given the current state. Note that for each light we assume a signal of 1 turns the light on while a signal of 0 turns it off. Using this truth table we can derive expressions for the next state in terms of the current state:

$$\begin{aligned}
S'_0 &= (\neg S_0) \\
S'_1 &= (S_0 \wedge \neg S_1 \wedge \neg S_2) \vee \\
&\quad (\neg S_0 \wedge S_1 \wedge \neg S_2) \\
S'_2 &= (S_0 \wedge S_1 \wedge \neg S_2) \vee \\
&\quad (\neg S_0 \wedge \neg S_1 \wedge S_2) \ .
\end{aligned}$$

Note that if our state machine transition function depended on some input as well as the current state, our expressions above would have to include that input in order to produce the correct next state given the current one. Either way, we can do a similar thing to control the traffic light colours on each road:

$$
\begin{aligned}
M_r &= (\neg S_0 \wedge S_1 \wedge \neg S_2) \vee \\
&\quad (S_0 \wedge S_1 \wedge \neg S_2) \vee \\
&\quad (\neg S_0 \wedge \neg S_1 \wedge S_2) \\
M_a &= (S_0 \wedge \neg S_1 \wedge \neg S_2) \vee \\
&\quad (S_0 \wedge \neg S_1 \wedge S_2) \\
M_g &= (\neg S_0 \wedge \neg S_1 \wedge \neg S_2) \\
\\
A_r &= (\neg S_0 \wedge \neg S_1 \wedge \neg S_2) \vee \\
&\quad (S_0 \wedge \neg S_1 \wedge \neg S_2) \vee \\
&\quad (S_0 \wedge \neg S_1 \wedge S_2) \\
A_a &= (\neg S_0 \wedge S_1 \wedge \neg S_2) \vee \\
&\quad (\neg S_0 \wedge \neg S_1 \wedge S_2) \\
A_g &= (S_0 \wedge S_1 \wedge \neg S_2) \ .
\end{aligned}
$$

Thus the current state drives everything; we feed S_0, S_1 and S_2 into two components, one of which produces the next state we need to move into and one which produces signals to control the lights. Clearly by sharing terms between the state update and light components we can reduce the cost of our design significantly.

2.4 Implementation and Fabrication Technologies

Designing a digital circuit is roughly akin to writing a program: once we have written a program, one usually intends to use it to perform some real task by executing it on a computer. The same is true of circuits, except that we first need to realise them somehow using physical components in order to use them. Although this topic is somewhat beyond the scope of this book, it is useful to understand mechanisms by which this is possible: even if that understanding is at a high-level, it can help to connect theoretical concepts with their practical realisation.

2.4.1 Silicon Fabrication

2.4.1.1 Lithography

The construction of semiconductor-based circuitry is very similar to how pictures are printed; essentially it involves controlled use of chemical processes. The act of printing pictures onto a surface is termed **lithography** and has been used for a couple of centuries to produce posters, maps and so on. The basic idea is to coat the surface, which we usually call the **substrate**, with a photosensitive chemical. We then expose the substrate to light projected through a negative, or **mask**, of the required image; the end result is that a representation of the original image is left on the substrate where the light reacts with the photosensitive chemical. After

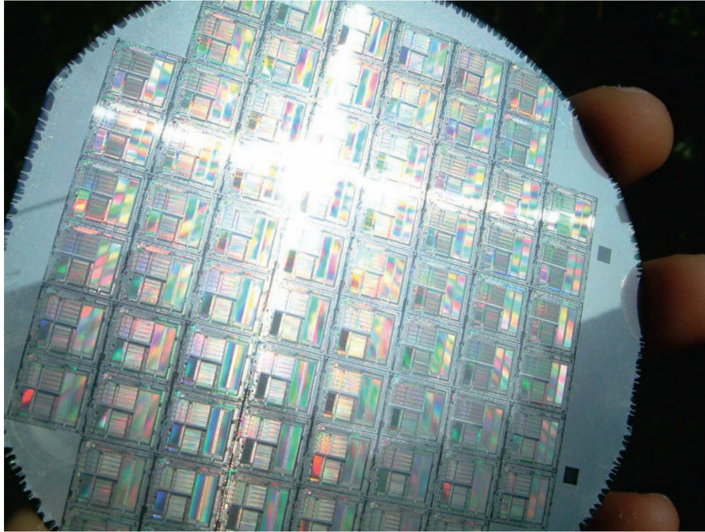


Figure 2.35 A silicon wafer housing many instances of an INMOS T400 processor design; see finger tips for scale ! (Copyright remains with photographer, photographer: James Irwin, source: personal communication)

washing the substrate, one can treat it with further chemicals so that the treated areas representing the original image are able to accept inks while the rest of the substrate cannot.

The analogous term in the context of semiconductors is **photo-lithography** and roughly involves similar steps. We again start with a substrate, which is usually a wafer of silicon. The wafer is circular because of the process of machining it from a synthetic ingot, or boule, of very pure silicon. After being cut into shape, the wafer is polished to produce a surface suitable for the next stage. We can now coat it with a layer of base material we wish to work with, for example a doped silicon or metal.

We then coat the whole thing with a photosensitive chemical, usually called a **photo-resist**. Two types exist, a positive one which hardens when hidden from light and a negative one which hardens when exposed to light. By projecting a mask of the circuit onto the result, one can harden the photo-resist so that only the required areas are covered with a hardened covering. After baking the result to fix the hardened photo-resist, and **etching** to remove the surplus base material, one is left with a layer of the base material only where dictated by the mask. A further stage of etching removes the hardened photo-resist completes the process for this layer. Many layers are required to produce the different layers required in a typical circuit; for example, we might need layers of N-type and P-type semiconductor and a metal layer to produce transistors. Using photo-lithography to produce a CMOS-based circuit, for example, thus requires many rounds of processing.

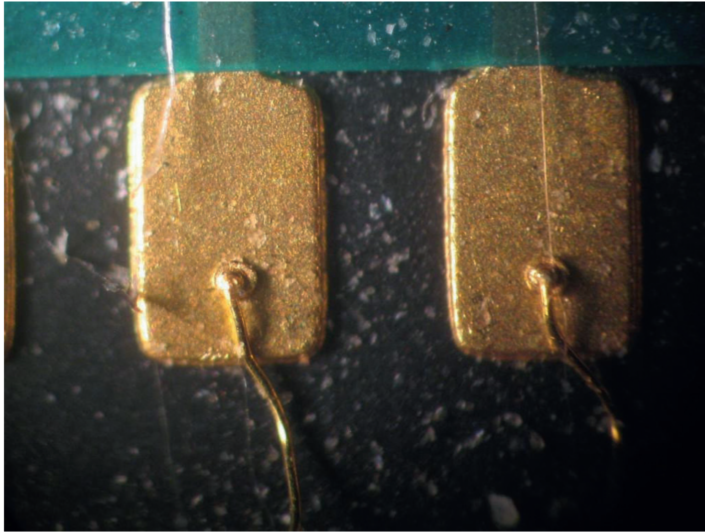


Figure 2.36 Bonding wires connected to a high quality gold pad. (Public domain image, photographer: Unknown, source: `http://en.wikipedia.org/wiki/Image:Wirebond-ballbond.jpg`)

2.4.1.2 Packaging

A major advantage of lithography is that one can produce accurate results across the whole wafer in one go. As such, one would typically produce many individual but identical circuits on the same wafer; each circuit instance is called a **die**. The next step in manufacture is to subject the wafer to a series of tests to ensure the fabrication process has been successful. Some dies will naturally be defective due to the circular shape of the wafer and typically rectangular shape of the dies; some dies will only represent half a circuit ! More importantly, since the manufacturing process is imperfect even complete dies can be operationally defective. Some circuit designs will incorporate dedicated test circuitry to aid this process. The ratio of intended dies to those that actually work is called the **yield**; clearly a high yield is attractive since defective circuits represent lost income.

The final step is to cut out each die from the wafer and **package** the working components. Packaging involves mounting the silicon circuit on a plastic base and connecting the tiny circuit inputs and outputs to externally accessible **pins** or **pads** with high quality **bonding wires**. The external pins allow the circuit to be interfaced with the outside world, the final component can thus be integrated within a larger design and forms what we commonly call a **microchip** or simply a **chip**. Large or power-hungry microchips are often complemented with a heat sink and fan which

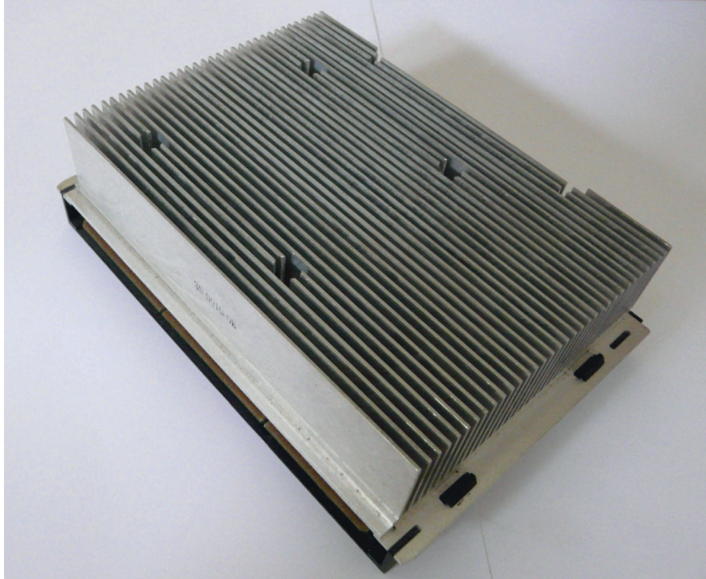


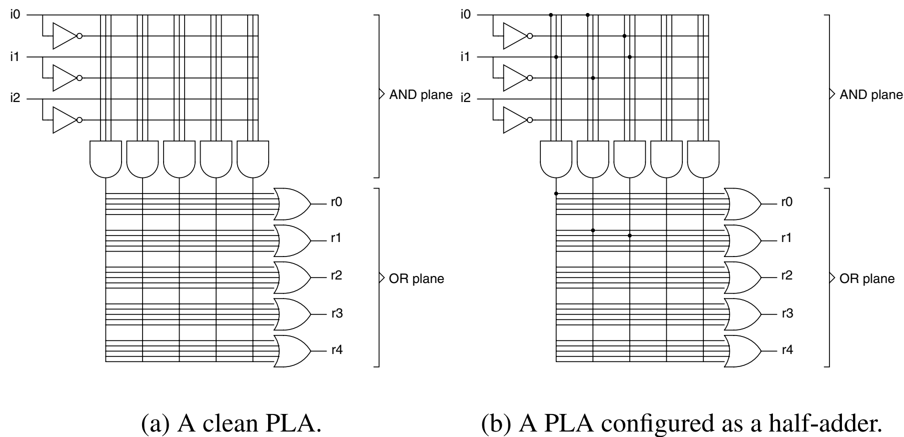
Figure 2.37 A massive heatsink (roughly 8" × 4" × 1.5") sat on top of a 667 MHz DEC Alpha EV67 (21264A) processor; such aggressive cooling allowed the processor to be run at higher clock speeds than suggested ! (Copyright remains with photographer, photographer: James Irwin, source: personal communication)

draws heat away and allow it to more easily operate within physical limits of the constituent materials.

2.4.2 Programmable Logic Arrays

A **Programmable Logic Array (PLA)** allows one to implement most combinatorial logic functions from a single, generic device. The basic idea is that the PLA forms a network of AND and OR gates. Considering an AND-OR type PLA as an example, the first network, or **plane**, allows one to implement minterms using a set of AND gates. These minterms are fed into a second plane of OR gates which implement the SoP expression for the required function. An OR-AND type PLA reverses the ordering of the planes of gates and thus allows implementation of PoS expressions.

More than one function that uses the same set of inputs can be implemented by the same PLA in either case. However, the limit in terms of both numbers of input and output and design complexity, is governed by the number of gates and wires one can fit into each plane. Figure 2.38a shows a diagram of a clean AND-OR type PLA



(a) A clean PLA. (b) A PLA configured as a half-adder.

Figure 2.38 Implementations of a clean and a configured PLA.

device with three inputs and five outputs. Notice how we construct the inverse of each input and feed wires across the plane so that they can be connected to produce the required logic function.

To program or configure a PLA so that it computes the required function, one starts with a generic, clean device with no connections on the wires. For an AND-OR type device, connection of the wires, by filling in the black dots on the diagram, forms the minterms in the AND plane and collects them in the OR plane. From a physical perspective, this is done by manipulating components placed at the connection points. Consider the placement of a fuse at each connection point or junction. Normally this component would act as a conductive material somewhat like a wire; when the fuse is blown using some directed energy, it becomes a resistive material. Thus, to form the right connections on a PLA one simply blows all the fuses where no connection is required. Alternatively, one can consider an antifuse which acts in the opposite way to a fuse: normally it is a resistor but when blown it is a conductor. Using antifuses at each junction means the configuration can simply blow those antifuses at the junctions where a connection is required. Figure 2.38b shows the originally clean PLA configured to act as a half-adder. We have that i_0 and i_1 are the inputs while r_0 is the carry-out and r_1 is the sum.

PLA devices based on fuses and antifuses are termed one-time-programmable since once the components at each junction are blown, they cannot be unblown: the device can only be configured once. However, the advantage of a PLA is the generality it offers. That is, one can take a single PLA component and easily implement most combinatorial functions: no costly silicon fabrication is required.

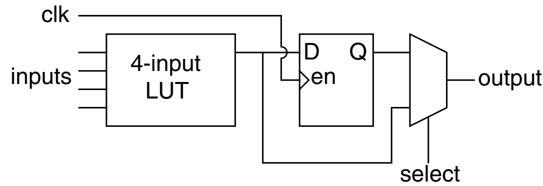


Figure 2.39 Implementation of a basic FPGA logic block.

2.4.3 Field Programmable Gate Arrays

The problems of one-time-programmable PLA devices are clear: prototyping logic designs can be costly since any mistake is terminal and the device can only ever be used for one task which, for large designs, means more cost when redundant devices are left idle. A **Field Programmable Gate Array (FPGA)** is one type of device that acts to solve this problem. Essentially, the idea of an FPGA is to offer a many-time-programmable device; the FPGA can be configured with one design and then re-configured with another design at a later date.

Roughly, an FPGA is built from a mesh of logic blocks. Each block can be configured to perform a specific, although limited, range of logic functions. Typical logic blocks might contain an n-input, 1-output **Look-Up Table (LUT)**. The LUT operation is simple: one forms an integer index i from the n inputs, looks-up item i in a table and returns this as the result. By filling the table with the required outputs for each input, the LUT can implement any n-input, 1-output logic function. Along side each LUT we typically find one or more flip-flops to enable devices with memory to be constructed. Most FPGAs also offer a small number of specialist logic blocks which incorporate high-performance versions of common circuits such as adders or even small processors. Figure 2.39 details a simple logic block consisting of a 4-input LUT and a D-type flip-flop; this is typical of many FPGA architectures.

Connections between the logic blocks in the mesh are formed in a similar way to a PLA in the sense that a component is placed at each connection junction. However, the FPGA junctions are re-configurable switches that can be turned on and off as required rather than blown in a one-off act. The idea is that by configuring each logic block to perform the right task and configuring connections between them, one can have the mesh compute both combinatorial and stateful logic circuits. For example, with our simple logic block above the configuration would fill the LUT with correct values and initialise the select input of the multiplexer with the right value so the block performs the right role. Connections between the logic blocks are then configured so that they interconnect and form the required circuit.

FPGA devices typically operate more slowly than custom silicon and use more power. As such, they are often used as a prototyping device for designs which will eventually be fabricated using a more high-performance technology. Further, in applications where debugging and updating hardware is as essential as software the

FPGA can offer significant advantages. Use of FPGA devices in mission-critical applications such as space exploration is commonplace: it turns out to be exceptionally useful to be able to remotely fix bugs in ones hardware rather than write off a multi-million pound satellite which is orbiting Mars and hence out of the reach of any local repair men !

2.5 Further Reading

- T.L. Floyd.
  Digital Fundamentals.
  Prentice-Hall, 2005. ISBN: 0-131-97255-3.
- D. Harris and S. Harris.
  Digital Design and Computer Architecture: From Gates to Processors.
  Morgan-Kaufmann, 2007. ISBN: 0-123-70497-9.
- C. Petzold.
  Code: Hidden Language of Computer Hardware and Software.
  Microsoft Press, 2000. ISBN: 0-735-61131-9.
- C. Roth.
  Fundamentals of Logic Design.
  Brooks-Cole, 2003. ISBN: 0-534-37804-8.
- A.S. Tanenbaum.
  Structured Computer Organisation.
  Prentice-Hall, 2005. ISBN: 0-131-48521-0.

2.6 Example Questions

5. a. Describe how N-type and P-type MOSFET transistors are constructed using silicon and how they operate as switches.

b. Draw a diagram to show how N-type and P-type MOSFET transistors can be used to implement a NAND gate. Show your design works by describing the transistor states for each input combination.

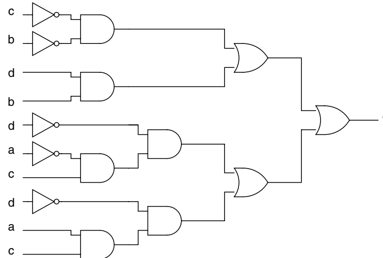
6. Given that ? is the don't care state, consider the following truth table which describes a function p with four inputs (a, b, c and d) and two outputs (e and f):

a b c d	e f
0 0 0 0	0 0
0 0 0 1	0 1
0 0 1 0	1 0
0 0 1 1	? ?
0 1 0 0	0 1
0 1 0 1	1 0
0 1 1 0	0 0
0 1 1 1	? ?

a b c d	e f
1 0 0 0	1 0
1 0 0 1	0 0
1 0 1 0	0 1
1 0 1 1	? ?
1 1 0 0	? ?
1 1 0 1	? ?
1 1 1 0	? ?
1 1 1 1	? ?

a. From the truth table above, write down the corresponding sum of products (SoP) equations for e and f.
b. Simplify the two SoP equations so that they use the minimum number of logic gates possible. You can assume the two equations can share logic.

7. Consider the following circuit where the propagation delay of logic gates in the circuit are NOT= 10 ns; AND= 20 ns; OR= 20 ns; XOR= 60 ns.



a. Draw a Karnaugh map for this circuit and derive a sum of products (SoP) expression for the result.
b. Describe advantages and disadvantages of your SoP expression and the dynamic behaviour it produces.
c. If the circuit is used as combinatorial logic within a clocked system, what is the maximum clock speed of the system ?

8. You are tasked with building a device for a casino based gambling game. The device must check when three consecutive heads are generated by a random coin flipper. Design a hardware circuit which will output 1 when three consecutive heads are generated and 0 otherwise. You can assume the random coin flipper circuit is already built or, for extra marks, design one yourself.

9. Imagine you are asked to to build a simple DNA matching hardware circuit as part of a research project. The circuit will be given DNA strings which are sequences of tokens that represent chemical building blocks. The goal is to search a large input sequence of DNA tokens for a small sequence indicative of some feature.

The circuit will receive one token per clock cycle as input; the possible tokens are adenine (A), cytosine (C), guanine (G) and thymine (T). The circuit should, given the input sequence, set an output flag to 1 when the matching sequence ACT is found

somewhere in the input or 0 otherwise. You can assume the inputs are infinitely long, i.e., the circuit should just keep searching forever and set the flag when the match is a success.

a. Design a circuit to perform the required task, show all your working and explain any design decisions you make along the way.
b. Now imagine you are asked to build two new matching circuits which should detect the sequences *CAG* and TTT respectively. It is proposed that instead of having three separate circuits, they are combined into a single circuit that matches the input sequence against one matching sequence selected with an additional input. Describe **one** advantage and **one** disadvantage you can think of for the two implementation options.

10. NAND is a universal logic gate in the sense that the behaviour of NOT, AND and OR gates can be implemented using only NAND. Show how this is possible using a truth table to demonstrate your solution.

11. A revolutionary, ecologically sound washing machine is under development by your company.

When turned on, the machine starts in the *idle* state awaiting input. The washing cycle consists of the three stages: *fill* (when it fills with water), *wash* (when the wash occurs), *spin* (when spin dying occurs); the machine then returns to *idle* when it is finished. Two buttons control the machine: pressing B_0 starts the washing cycle, pressing B_1 cancels the washing cycle at any stage and returns the machine to *idle*; if both buttons are pressed at the same time, the machine continues as normal as if neither were pressed.

a. You are asked to design a logic circuit to control the washing machine behaviour. Draw a diagram that shows the washing machine states and the valid transitions between them.
b. Draw a state transition table for a state machine that could be used to implement your diagram.
c. Using the Karnaugh map technique, derive logic expressions which allow one to compute the next state from the current state in your table; note that because the washing machine is ecologically sound, minimising the number of logic gates is important.