ANDREW KENNINGS

INTRODUCTION TO DIGITAL LOGIC DESIGN

Contents

1 Preface

I prepared this book based off a set of handwritten notes which I had created when tasked with teaching a freshman course on digital logic design. I hope that the contents of this book are easy to follow and are direct in explaining different concepts.

I have specifically not included any problem sets in these notes in order to keep things a bit shorter. I have also specifically chosen to not include the use of Hardware Descriptive Languages (**HDL**s) in this book. I made that decision based on the fact that there are many different **HDL**s available and different people might have access to different tools and languages. I saw no point in including pages of material that might not be useful as a consequence of selecting the "wrong" **HDL**. Finally, I think that a lot of insight can be gained by sometimes building things by hand rather that through the use of an **HDL**.

I have divided this book into three sections. The first section addresses combinational logic. The second section introduces storage elements including latches and flip flops and then focuses on sequential circuit design. The last section briefly covers asynchronous circuits and design with latches.

I hope that everyone finds the explanation of the material contained with this book useful.

Part I

Combinational logic

2 The basics

2.1 Binary variables and functions

A binary variables is a variable that take on only two discrete values 0 and 1. A binary logic function produces an output as an expression of its input variables. Its input variables are binary variables and/or other binary logic functions. A binary logic function evaluates to either 0 or 1 depending on the value of its input variables.

2.2 Truth tables

A truth table is *one way* to express a logic function in a tabular form. The truth table specifies the value (output) of the logic function for each possible setting of inputs → one row for each input combination. A logic function with n input variables therefore requires 2^n rows in its truth table (since each input variable can be set to 0 or 1).

Figure 2.1 is an example of a truth table for a 3-input function.

x_0	x_1	x_2	f
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Figure 2.1: Example of a truth table for a 3-input function $f = f(x_0, x_1, x_2)$.

Notice that this truth table has $2^3 = 8$ rows; each row corresponds to a different input combination. We can refer to rows of the truth table as "row 0", "row 1", etc.

2.3 Logic functions

We can also write a logic function with as a logical expression. For example, the function f described by the truth table in Figure 2.1

can be written as $f = \bar{x}_2\bar{x}_1\bar{x}_0 + \bar{x}_2x_1\bar{x}_0 + \bar{x}_2x_1x_0 + x_2x_1x_0$. The logic expression provides the same information as the truth table. To manipulate or evaluate logic expressions, we need to define some *logic operations*.

2.4 Logic operators

There are three basic logic operators, namely **AND**, **OR** and **NOT**. These three operators have symbols given in Table 2.1. The behaviour of each operation is defined via a truth table. Operators also have precedence: **NOT**, then **AND** then **OR**. We can use parentheses is to clarify precedence when required.

Operation	Symbol	Example
AND	$\bullet$, "nothing"	$f = x_1 \bullet x_0, f = x_1x_0$
OR	$+$	$f = x_1 + x_0$
NOT	$!, ', \neg, overbar$	$f =!x, f = x^{'}, f = \neg x, f = \bar{x}$

Table 2.1: Basic logic operators.

2.4.1 Logic operator – AND

The **AND** operator generates an output of 1 when *all* inputs are 1, otherwise 0. The truth table for a 2-input **AND** is shown in Figure 2.2.

x_0	x_1	$f = x_1x_0$
0	0	0
0	1	0
1	0	0
1	1	1

Figure 2.2: Truth table for a 2-input **AND**.

The **AND** operator can have any number of inputs (i.e., it generalizes to n inputs). The truth table for an n-input **AND** is shown in Figure 2.3.

x_0	x_1	$\cdots$	x_{n-2}	x_{n-1}	$f = x_{n-1}x_{n-2}\cdots x_1x_0$
0	0	$\cdots$	0	0	0
0	0	$\cdots$	0	1	0
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
0	1	$\cdots$	1	1	0
1	0	$\cdots$	0	0	0
1	0	$\cdots$	0	1	0
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
1	1	$\cdots$	1	1	1

Figure 2.3: Truth table for a n-input **AND**.

It is useful to have a schematic symbol for the **AND** operator

(referred to as a **gate**). The symbol for the **AND** gate is shown in Figure 2.4.

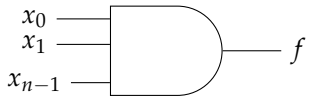


Figure 2.4: Schematic symbol for the **AND** operator.

2.4.2 *Logic operator – **OR***

The **OR** operator generates an output of 1 when *any* of its inputs are 1, otherwise 0. The truth table for a 2-input **OR** is shown in Figure 2.5.

x_0	x_1	$f = x_1 + x_0$
0	0	0
0	1	1
1	0	1
1	1	1

Figure 2.5: Truth table for a 2-input **OR**.

The **OR** operator can have any number of inputs (i.e., it generalizes to n inputs). The truth table for an n-input **OR** is shown in Figure 2.6.

x_0	x_1	$\cdots$	x_{n-2}	x_{n-1}	$f = x_{n-1} + \cdots + x_1 + x_0$
0	0	$\cdots$	0	0	0
0	0	$\cdots$	0	1	1
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
0	1	$\cdots$	1	1	1
1	0	$\cdots$	0	0	1
1	0	$\cdots$	0	1	1
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
1	1	$\cdots$	1	1	1

Figure 2.6: Truth table for a n-input **OR**.

It is useful to have a schematic symbol for the **OR** operator (referred to as a **gate**). The symbol for the **OR** gate is shown in Figure 2.7.

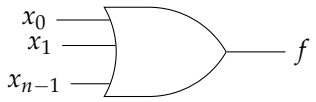


Figure 2.7: Schematic symbol for the **OR** operator.

2.4.3 *Logical operator – **NOT***

The **NOT** operator, which is also sometimes referred to as the **INV** (inversion) operator, takes a single input and "flips" or "inverts" the input value. The truth table defining the **NOT** operator is shown in Figure 2.8. The schematic symbol for the **NOT** operator is shown in Figure 2.9.

x	$f = !x$
0	1
1	0

Figure 2.8: Truth table for the **NOT** operator.

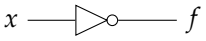


Figure 2.9: Schematic symbol for the **NOT** operator.

2.5 Truth tables from logic expressions

Given a logic function f, we can find its truth table by evaluating f for every possible input combination. Consider the 2-input logic function $f = !x_1!x_0 + x_1x_0$; it's truth table is calculated in Figure 2.10.

x_1	x_0	f
0	0	$f = !0 \bullet !0 + 0 \bullet 0 = 1 + 0 = 1$
0	1	$f = !0 \bullet !1 + 0 \bullet 1 = 0 + 0 = 0$
1	0	$f = !1 \bullet !0 + 1 \bullet 0 = 0 + 0 = 0$
1	1	$f = !1 \bullet !1 + 1 \bullet 1 = 0 + 1 = 1$

$\rightarrow$

x_1	x_0	f
0	0	1
0	1	0
1	0	0
1	1	1

Figure 2.10: Finding the truth table from the logic expression for $f = !x_1!x_0 + x_1x_0$. The logic expression is simply evaluated for every row of the truth table.

2.6 Circuit diagrams

We can draw circuit diagrams to represent/illustrate logic functions. For example, consider the logic expression $f = !x_2!x_0 + x_1x_0$. The schematic diagram for this function is shown in Figure 2.11.

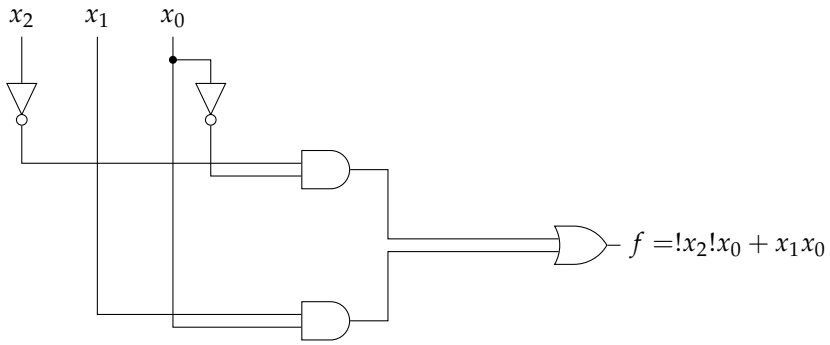


Figure 2.11: Schematic diagram for $f = !x_1!x_0 + x_1x_0$. Given a logic expression, to draw a schematic diagram we simply draw the necessary logic gates and then use wire to connect the gates as appropriate.

The circuit diagram (schematic) can be seen as a *third* way to represent a logic function. We can clearly write down a logic expression given a circuit diagram (and visa-versa).

2.7 Summary

There are several ways to represent logic functions. We can use truth tables, logic expressions or schematic diagrams. We should be able to convert from one to another (although it's not yet clear how to write down a logic expression from a truth table — this comes later). There are three basic logic operators, namely **AND**, **OR** and **NOT** which are important to understand.

3 Boolean algebra

Boolean algebra was introduced in 1854 by George Boole and shown in 1938 to be useful for manipulating Boolean logic functions by C. E. Shannon.

The algebra provides postulates and theorems that are useful to simply logic equations, demonstrate equivalence of expressions, etc.

3.1 Postulates and theorems

We have a set of elements **B** and two binary operators $+$ and $\bullet$ that satisfy the following postulates:

1. Closure with respect to: (a) $+$ and (b) $\bullet$

2. Identify elements with respect to: (a) $+$, designated by 0 and (b) $\bullet$, designated by 1

3. Commutative with respect to: (a) $+$ and (b) $\bullet$

4. Distributive for: (a) $\bullet$ over $+$ and (b) $+$ over $\bullet$

5. For each element $x \in B$, $\exists !x \in B$ such that (a) $x+!x = 1$ and (b) $x\bullet !x = 0$

6. There exists at least two elements $x, y \in B$ such that $x \neq y$.

Axioms are truths and do not require proof. Fortunately, our definitions of **AND**, **OR** and **NOT** satisfy these axioms.

We can list these postulates in a more convenient way along with some very useful theorems. Note that the theorems must be proven (from either truth tables or from the postulates and/or previously proven theorems).

Postulate 2	(a)	$x+0=x$	(b)	$x \bullet 1 = x$	identity
Postulate 3	(a)	$x+y=y+x$	(b)	$x \bullet y = y \bullet x$	commutative
Postulate 4	(a)	$x \bullet (y+z) = x \bullet y + x \bullet z$	(b)	$x + (y \bullet z) = (x+y)(x+z)$	distributive
Postulate 5	(a)	$x+!x = 1$	(b)	$x \bullet !x = 0$	
Theorem 1	(a)	$x+x=x$	(b)	$x \bullet x = x$	
Theorem 2	(a)	$x+1=1$	(b)	$x \bullet 0 = 0$	
Theorem 3		$(x')' = x$			involution
Theorem 4	(a)	$x+(y+z) = (x+y)+z$	(b)	$x \bullet (y \bullet z) = (x \bullet y) \bullet z$	associative
Theorem 5	(a)	$(x+y)' = x' \bullet y'$	(b)	$(x \bullet y)' = x' + y'$	DeMorgan
Theorem 6	(a)	$x + x \bullet y = x$	(b)	$x \bullet (x+y) = x$	absorption

Note the *duality* if we interchange $+$ with $\bullet$ and 0 with 1 in all of the postulates and theorems!

As an example of how we can use this information, we can consider proving Theorem 1b ($x \bullet x = x$); the proof is shown in Figure 3.1 using postulates and in Figure 3.2 using truth tables.

$$\begin{array}{rcll} x & = & x \bullet 1 & \text{P2b} \\ & = & x \bullet (x+!x) & \text{P5a} \\ & = & x \bullet x + x \bullet !x & \text{P4a} \\ & = & x \bullet x + 0 & \text{P5b} \\ & = & x \bullet x & \text{P2a} \end{array}$$

Figure 3.1: Proof of Theorem 1b ($x \bullet x = x$) using Boolean algebra and postulates.

x	x	$x \bullet x$
0	0	0
1	1	1

Figure 3.2: Proof of Theorem 1b ($x \bullet x = x$) using truth tables; comparing columns 1 and 3 prove $x = x \bullet x$.

As an example of another proof, we can consider the consensus theorem which states $xy + x'z + yz = xy + x'z$ —the proof of this theorem is shown in Figure 3.3.

$$\begin{array}{rcl} xy + x'z + yz & = & xy + x'z + (x+x')yz \\ & = & xy + x'z + xyz + x'yz \\ & = & xy + x'z \end{array}$$

Figure 3.3: Proof of the consensus theorem.

Finally, another example would be the theorem $xb + x'a = (x + a)(x' + b)$ — the proof of this theorem is shown in Figure 3.4 and uses the consensus theorem in the last step of the proof.

We can also use Boolean algebra to perform *simplification* of logic expressions. Simplification means to find a simpler expression for a logic expression. Consider the following: Find a simpler expression for for $f = ab + cd + a+!(!(cd) + a)$. The solution is provided on Figure 3.5. Note that when simplifying expressions with Boolean

$$\begin{aligned}(x+a)(x'+b) &= xx'+xb+x'a+ab \\ &= xb+x'a+ab \\ &= xb+x'a\end{aligned}$$

Figure 3.4: Proof of $xb+x'a = (x+1)(x'+b)$.

$$\begin{aligned}f &= ab+cd+a+!(!(cd)+a) \\ f &= ab+a+cd+(cd)!a \\ f &= a(1+b)+cd(1+!a) \\ f &= a+cd\end{aligned}$$

Figure 3.5: Use of Boolean algebra to perform simplification of a logic expression.

algebra, it might be hard to know that you have the absolute simplest expression.

3.2 Circuit cost

We previously used Boolean algebra to obtain a simpler expression for a logic function f. But how should we define simpler? We can define the *cost* of a function (or circuit). There can be many different ways to define cost depending on our overall objective. For now (unless otherwise stated), let us define the cost of a circuit as follows:

1. Input variables are available both complemented and uncomplemented (therefore any inversions of the inputs is free);
2. Every logic gate costs 1 (so more gates are bad);
3. Every logic gate input costs 1 (so larger gates are bad).

Cost defined in this way tends to result in circuits that require less overall *area* (less and smaller gates).

As an example, the cost of $f = a+cd+ab+!(!(cd)+a)$ is shown as follows:

$$f = \underbrace{a+\underbrace{cd}_{1+2=3}+\underbrace{ab}_{1+2=3}+\underbrace{!}_{2}\overbrace{(\underbrace{!}_{2}\underbrace{(cd)}_{1+2=3}+a)}^{1+2=3}}_{1+4=5}$$

The total cost is **21** (3, 2-input **AND**s; 1, 2-input **OR**; 1, 4-input **OR**; 2 non-trivial **NOT**).

Now, consider the cost of $f = cd+a$ which is computed as follows:

$$f = \underbrace{\overbrace{cd}^{1+2=3}+a}_{1+2=3}$$

So the total cost is **6** (1, 2-input **AND**; 1, 2-input **OR**).

Since this was our previous simplification example, we can see that Boolean algebra has allowed us to reduce the cost of implementing f (reduced the total number of gates and the size of the gates).

3.3 Postive and negative literals

There is some terminology that you might hear: Let x be a binary variable. Depending on the situation, we might write x (variable *not complemented*) or $!x$ (variable *complemented*).

- The uncomplemented version of x is called the "positive literal" of variable x.
- The complemented version of x is called the "negative literal" of variable x.

3.4 Summary

Boolean algebra allows us to manipulate logic expressions. This can be useful, for example, to simply logic expressions and obtain something which is "cheaper" to implement. Typically, by "cheap" we mean a circuit performs its function using "few" gates and that the number of inputs required by any gate is "reasonable". Depending on our objective, however, there can be different definitions of cost.

4 Other types of logic gates

Although **AND**, **OR**, and **NOT** gates are sufficient to implement any circuit, there are other useful types of logic gates. By "useful", we mean that we can implement things more efficiently using these types of gates.

4.1 Logical operator — NAND

NAND gates perform the "NOT-AND" operation (i.e., AND then NOT) which generates an output of 0 when all inputs are 1, otherwise 1. The truth table for a 2-input **NAND** gate is shown in Figure 4.2. The **NAND** can have any number of inputs. The truth table for a n-input **NAND** gate is shown in Figure **??**.

x_0	x_1	$f = \overline{x_1 x_0}$
0	0	1
0	1	1
1	0	1
1	1	0

Figure 4.1: Truth table for a 2-input **NAND**.

x_0	x_1	$\cdots$	x_{n-2}	x_{n-1}	$f = \overline{x_{n-1} \cdots x_1 x_0}$
0	0	$\cdots$	0	0	1
0	0	$\cdots$	0	1	1
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
0	1	$\cdots$	1	1	1
1	0	$\cdots$	0	0	1
1	0	$\cdots$	0	1	1
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
1	1	$\cdots$	1	1	0

Figure 4.2: Truth table for a 2-input **NAND**.

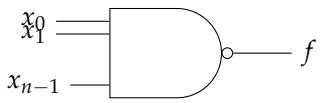


Figure 4.3: Schematic symbol for **NAND**.

The symbol for a **NAND** gate is shown in Figure 4.3. Note that, unlike **AND**, **NAND** is *not* an associative operator — you cannot make larger **NAND** gates using smaller **NAND** gates.

4.2 *Logical operator – NOR*

NOR gates perform the "NOT-OR" operation (i.e., OR then NOT) which generates an output of 0 when any inputs are 1, otherwise 1. The truth table for a 2-input **NOR** gate is shown in Figure 4.4.

The **NOR** can have any number of inputs. The truth table for a *n*-input **NOR** gate is shown in Figure 4.5.

NOR with 2-inputs...

x_0	x_1	$f = \overline{x_1 + x_0}$
0	0	1
0	1	0
1	0	0
1	1	0

Figure 4.4: Truth table for a 2-input **NOR**.

x_0	x_1	$\cdots$	x_{n-2}	x_{n-1}	$f = \overline{x_{n-1} + \cdots + x_1 + x_0}$
0	0	$\cdots$	0	0	1
0	0	$\cdots$	0	1	0
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
0	1	$\cdots$	1	1	0
1	0	$\cdots$	0	0	0
1	0	$\cdots$	0	1	0
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
1	1	$\cdots$	1	1	0

Figure 4.5: Truth table for a *n*-input **NOR**.

The symbol for a **NOR** gate is shown in Figure 4.6. Note that, unlike **OR**, **NOR** is *not* an associative operator and you cannot make a larger **NOR** gate from smaller **NOR** gates.

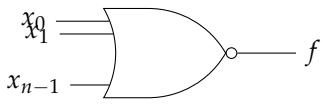


Figure 4.6: Schematic symbol for **NOR** .

4.3 *Logical operator – XOR*

The **XOR** operator occur often in circuits designed for algebraic operations such as adder circuits, multipler circuits, and so forth. With 2 inputs, the **XOR** operator generates a 1 when the inputs are *different*, otherwise 0. The truth table is shown in Figure 4.7.

x_0	x_1	$f = x_1 \oplus x_0$
0	0	0
0	1	1
1	0	1
1	1	0

Figure 4.7: Truth table for a 2-input **XOR**.

With $\geq$ 3 inputs, the **XOR** operator generates a 1 when the number of inputs which are 1 is *odd*, otherwise 0. The truth table for a 3-input **XOR** gate (as an example of an **XOR** with more than 2 inputs) is shown in Figure 4.8. Notice that the **XOR** utilizes a new symbol ($\oplus$) to denote its operation. The schematic symbol for the **XOR** operator is shown in Figure 4.9. The **XOR** operator is associative.

x_0	x_1	x_2	$f = x_2 \oplus x_1 \oplus x_0$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Figure 4.8: Truth table for a 3-input **XOR**.

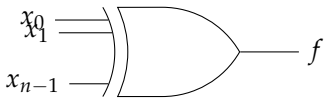


Figure 4.9: Schematic symbol for **XOR**.

4.4 *Logical operator – NXOR*

The **NXOR** gate perform the "NOT-XOR" operation. With 2-inputs, the **NXOR** generates a 1 when the inputs are *equal*, otherwise 0. This is shown in Figure 4.10.

With $\geq$ 3 inputs, the **NXOR** generates a 1 when the number of inputs which are 1 is *even*, otherwise 0. This is shown in Figure 4.11 for a 3-input **NXOR** as an example of a **NXOR** with more than 2 inputs. Note that the **NXOR** operator is not associative. The schematic symbol for the **NXOR** gate is shown in Figure 4.12.

x_0	x_1	$f = \overline{x_1 \oplus x_0}$
0	0	1
0	1	0
1	0	0
1	1	1

Figure 4.10: Truth table for a 2-input **NXOR**.

x_0	x_1	x_2	$f = \overline{x_2 \oplus x_1 \oplus x_0}$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Figure 4.11: Truth table for a 3-input **NXOR**.

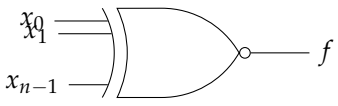


Figure 4.12: Schematic symbol for **NXOR**.

4.5 *Logical operator* – **BUF**

We might also see the **BUF** operator. This operator does nothing logically — it takes a single input and produces an output which is equal to its input. Buffers are intended to solve a practical problem — they serve to increase the "driving strength" of a signal. The truth table for a **BUF** operator is shown in Figure 4.13 and its schematic symbol is shown in Figure 4.14.

x	$f = x$
0	0
1	1

Figure 4.13: Truth table for **BUF**.

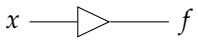


Figure 4.14: Schematic symbol for **BUF**.

4.6 *Tristate buffers*

The tristate buffer is a buffer with an *enable* signal. When the enable signal is 1, the output f equals the input x; i.e., $f = x$ just like a regular buffer. However, when the enable signal is 0, the output f is *disconnected* from the input x — basically an open circuit. This behaviour is described in 4.15. The schematic symbol for a tristate buffer is shown in Figure 4.16.

x	en	$f = x$
0	1	f disconnected from x
1	1	f disconnected from x
0	1	0
1	1	1

Figure 4.15: Behaviour for a tristate buffer.

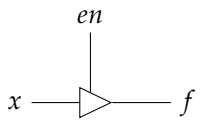


Figure 4.16: Schematic symbol for a tristate buffer.

Every wire in a circuit can only have one *source* that drives the wire. However, it might be the case we might want to have multiple sources that can drive the same wire, but *at different times*. This is exactly the purpose for the tristate buffer — if we have multiple sources that want to drive a wire, we can use tristate buffers to ensure that only one source maximum is connected to the wire at any given time. This is illustrated in Figure 4.17.

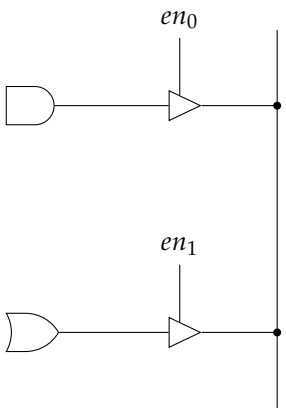


Figure 4.17: Example of two tristate buffers to prevent different logic gates from driving a wire at the same time; this can be prevented by avoiding $en_0 = 1$ and $en_1 = 1$. The gates can "take turns" driving the same wire or both can be disconnected from the wire.

4.7 *Summary*

In addition to **AND**, **OR** and **NOT**, there are other types of logic operators. This includes **NAND**, **NOR**, **XOR**, **NXOR** and **BUF**. These other operators can prove useful in different scenarios.

5 Minterms, maxterms and two-level representations

Given a truth table, we might want to derive a logical expression for a logic function (which, for example, can then be manipulated or simplified using Boolean algebra). There are two simple and standard ways to derive a logical expression for a function from its truth table — one method will yield a *Sum-Of-Products* (SOP) representation for the logic function while the other method will yield a *Product-Of-Sums* (POS) representation for the logic function.

5.1 Minterms

Consider a truth table for an n-input function with 2^n rows. For *each* row, create an **AND** of *all* inputs according to the following rule:

- If the input variable has value 1 in the current row, then include the variable uncomplemented (the positive literal);
- If the input variable has value 0 in the current row, then include the variable complemented (the negative literal).

The result of applying the above rule yields the so-called "minterm" for each row of the truth table. An n-input function has 2^n minterms (one for each row) and all are different. We are particularly interested in the minterms for which the function is required to be 1. Figure 5.1 shows the minterms for a 3-input function $f = f(x, y, z)$.

Minterms are denoted by the lower-case letter "m_i" to represent the minterm for the i-th row of the truth table. Minterms exhibit a "useful" property: A minterm evaluates to 1 if and only if the corresponding input pattern appears, otherwise it evaluates to 0.

5.2 Canonical Sum-Of-Products

Given the truth table for a logic function, we can *always* write down a logic expression by taking the **OR** of the *m*interms for which the

x	y	z	minterm	
0	0	0	$!x!y!z$	$= m_0$
0	0	1	$!x!yz$	$= m_1$
0	1	0	$!xy!z$	$= m_2$
0	1	1	$!xyz$	$= m_3$
1	0	0	$x!y!z$	$= m_4$
1	0	1	$x!yz$	$= m_5$
1	1	0	$xy!z$	$= m_6$
1	1	1	xyz	$= m_7$

Figure 5.1: Minterms for a 3-input function $f = f(x, y, z)$.

function is 1. The resulting expression is "canonical" (unique) and is called the Canonical Sum-Of-Products (SOP) or Sum-Of-Minterms representation for the function. An example is shown in Figure 5.2.

We can write $f(x, y, z) = m_1 + m_4 + m_7 = !x!yz + x!y!z + xyz$.

x	y	z	f	
0	0	0	0	
0	0	1	1	$!x!yz = m_1$
0	1	0	0	
0	1	1	0	
1	0	0	1	$x!y!z = m_4$
1	0	1	0	
1	1	0	0	
1	1	1	1	$xyz = m_7$

Figure 5.2: Finding a Sum-Of-Minterms given a truth table. We are interested in the **OR** of the minterms for those input combinations (rows) where $f = 1$.

Shortcut notation exists. We can use a summation sign to indicate those minterms that must be included in f. As an example,

$$f(x, y, z) = m_1 + m_4 + m_7 = \underbrace{\sum m(1, 4, 7)}_{\text{minterms required for } f}.$$

Note that using minterms to write down an expression for f follows the idea of including minterms to decide when f needs to be set to 1 and turning f "on" for the correct inputs.

When implemented as an SOP, a function f always has the same "structure"; a "plane" of **NOT**, followed by a "plane" of **AND** followed by a single **OR**. This is illustrated in Figure 5.3. Note that SOP implementations of f composed of minterms are not cheap since each **AND** gate has a maximum number of inputs and the **OR** gate (can) also have a large number of inputs if the function f involves many minterms.

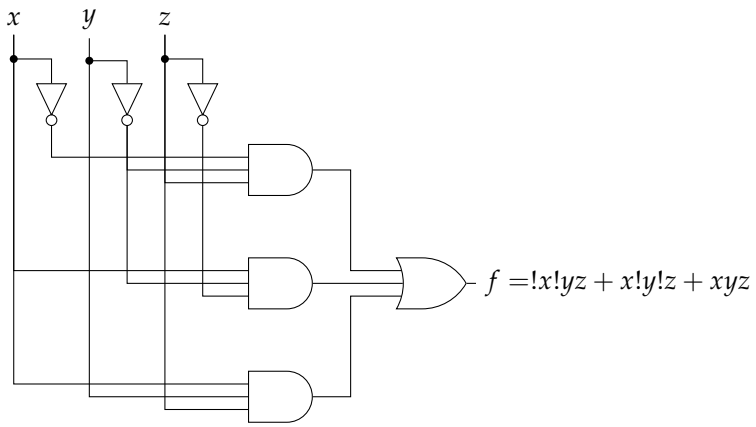


Figure 5.3: Illustration of a Sum-Of-Products (in this case a Sum-Of-Minterms).

5·3 *Maxterms*

Consider a truth table for an n-input function with 2^n rows. For *each* row, create an **OR** of *all* inputs according to the following rule:

- If the input variable has value 0 in the current row, then include the variable uncomplemented (the positive literal);
- If the input variable has value 1 in the current row, then include the variable complemented (the negative literal).

The result of applying the above rule yields the so-called "maxterm" for each row of the truth table. An n-input function has 2^n maxterms (one for each row) and all are different.

Figure 5.4 illustrates the maxterms for a 3-input function $f = f(x, y, z)$. Maxterms are denoted by the upper-case letter "M_i" to

x	y	z	minterm	
0	0	0	$x + y + z$	$= M_0$
0	0	1	$x + y + !z$	$= M_1$
0	1	0	$x + !y + z$	$= M_2$
0	1	1	$x + !y + !z$	$= M_3$
1	0	0	$!x + y + z$	$= M_4$
1	0	1	$!x + y + !z$	$= M_5$
1	1	0	$!x + !y + z$	$= M_6$
1	1	1	$!x + !y + !z$	$= M_7$

Figure 5.4: Maxterms for a 3-input function $f = f(x, y, z)$.

represent the maxterm for the i-th row of the truth table. Maxterms exhibit a "useful" property: A maxterm evaluates to 0 if and only if the corresponding input pattern appears, otherwise it evaluates to 1.

5·4 *Canonical Product-Of-Sums*

Given the truth table for a logic function, we can *always* write down a logic expression by taking the **AND** of the *maxterms* for which the function is 0. The resulting expression is "canonical" (unique) and is called the Canonical Product-Of-Sums (POS) or Product-Of-Maxterms representation for the function. An example is shown in Figure 5.5. We can write $f(x, y, z) = M_0 M_2 M_4 M_5 M_6 = (x + y + z)(x + !y + z)(x + !y + !z)(!x + y + !z)(!x + !y + z)$ Shortcut notation exists. We can use a product sign to indicate which maxterms that must be included in f. As an example

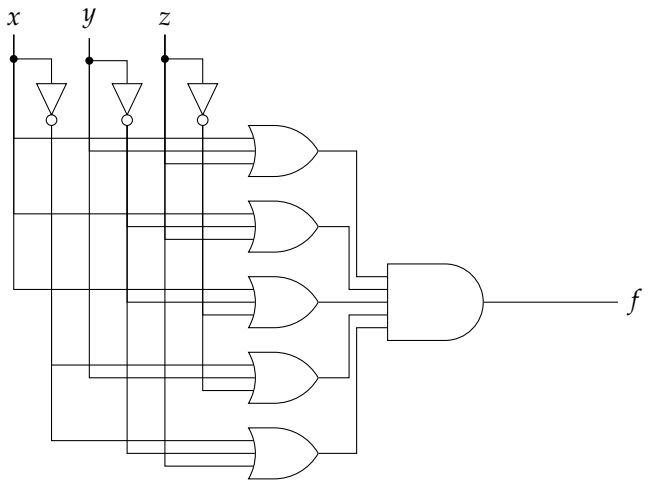
$$f(x, y, z) = M_0 M_2 M_4 M_5 M_6 = \underbrace{\Pi M(0, 2, 4, 5, 6)}_{\text{maxterms required for } f} .$$

x	y	z	f	
0	0	0	0	$x+y+z = M_0$
0	0	1	1	
0	1	0	0	$x+!y+z = M_2$
0	1	1	0	$x+!y+!z = M_3$
1	0	0	1	
1	0	1	0	$!x+y+!z = M_5$
1	1	0	0	$!x+!y+z = M_6$
1	1	1	1	

Figure 5.5: Finding a Sum-Of-Maxterms given a truth table. We are interested in the **AND** of the maxterms for those input combinations (rows) where $f = 0$.

Note that using maxterms to write down an expression for f follows the idea of including maxterms to decide when f needs to be set to 0 and turning f "off" for the correct inputs.

When implemented as an POS, a function f always has the same "structure"; a "plane" of **NOT**, followed by a "plane" of **OR** followed by a single **AND**. This is illustrated in Figure 5.6. Note



$$f = (x+y+z)(x+!y+z)(x+!y+!z)(!x+y+!z)(!x+!y+z)$$

Figure 5.6: Illustration of a Product-Of-Sums (in this case a Product-Of-Maxterms).

that POS implementations of f composed of minterms are not cheap since each **OR** gate has a maximum number of inputs and the **AND** gate (can) also have a large number of inputs if the function f involves many maxterms.

5.5 *Additional comments*

SOP and POS representations of a logic function f are often referred to as the so-called "*two-level*" implementations of f. The is because if we ignore the inverters (e.g., because we assume that input variables are available either complemented or un-complemented), f is the

output of two levels of logic gates (AND the OR for SOP; OR then AND for POS).

5.6 Standard Sum-Of-Product (SOP) implementations

A function f implemented as a canonical SOP using minterms is by no means minimal. Let any AND of input literals be called a **product term**. We can then express any function f as a Sum-Of-Products where, instead of using minterms, we use product terms. The resulting SOP is typically simpler and of lower cost than the canonical SOP. This simplification can, for example, be obtained by Boolean algebra.

As an example, consider the 3-input function $f = f(x, y, z)$ defined by the truth table in Figure 5.7. We can write $f(x, y, z) = !xyz +$

x	y	z	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Figure 5.7: Example truth table to demonstration the implementation of logic function f using product terms rather than minterms.

$x!yz + xy!z + xyz$. However, Boolean algebra can be used to show that $f(x, y, z) = xy + xz + yz$ which is simpler and composed of 3 product terms (none of which is a minterm). Note that while a minterm is a product term, a product term is *not necessarily a minterm*.

5.7 Standard Product-Of-Sum (POS) implementations

A function f implemented as a canonical POS using maxterms is by no means minimal. Let any OR of input literals be called a **sum term**. We can then express any function f as a Product-Of-Sums where, instead of using maxterms, we use sum terms. The resulting POS is typically simpler and of lower cost than the canonical POS. This simplification can, for example, be obtained by Boolean algebra.

As an example, consider the 3-input function $f = f(x, y, z)$ defined by the truth table in Figure 5.8. We can write $f(x, y, z) = (x + y + z)(x+!y+z)(x+!y+!z)(!x+y+!z)(!x+!y+z)$. However, Boolean algebra can be used to show that $f(x, y, z) = (x + z)(x+!y)(!y + z)(!x+y+!z)$ which is simpler and composed of 4 sum terms (one

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Figure 5.8: Example truth table to demonstration the implementation of logic function f using sum terms rather than maxterms.

of which IS a maxterm). Note that while a maxterm is a sum term, a sum term is *not necessarily a maxterm*.

5.8 Conversion and equality of representations

Minterms and maxterms are "*duals*" of each other; $m_i = !M_i$ (and $M_i = !m_i$). You can always convert from one canonical represention to the other — We can convert from minterms to maxterms by changing $\sum$ to Π and list those terms missing from the original list. The reverse (maxterms to minterms) works the same way. An example is the conversion of $f(x,y,z) = \sum m(1,4,7)$ to maxterms. The solution is

$$\begin{aligned} f &= \sum m(1,4,7) \\ &= m_1 + m_4 + m_7 \\ &= !(!f) \\ &= !(m_0 + m_2 + m_3 + m_5 + m_6) \\ &= !m_0 !m_2 !m_3 !m_5 !m_6 \\ &= M_0 M_2 M_3 M_5 M_6 \\ &= \Pi M(0,2,4,5,6) \end{aligned}$$

We can also convert from a SOP to a POS and visa-versa even if a logic function is not expressed in a canonical form. Consider the 3-input logic function $f = xy + xz + yz$ which is a minimized SOP expression for f. We can write this as $f = ((xy + xz + yz)')'$ since double inversion does not change the function. We first eliminate the innermost inversion and rearrange to get the expression written as the inversion of a SOP. Finally, we use the outermost inversion to convert the SOP to a POS. Specifically, $f = ((xy + xz + yz)')' = ((xy)'(xz)'(yz)')' = ((x' + y')(x' + z')(y' + z'))' = ((x' + y'z')(y' + z'))' = (x'y' + x'z' + y'z')' = (x + y)(x + z)(y + z)$ which is a POS.

5.9 Summary

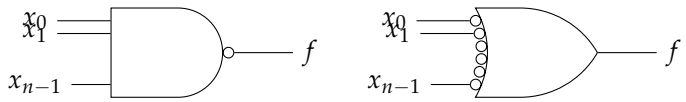
Given a truth table, we can always write a logic expression for a function using minterms to obtain a Sum-Of-Minterms or maxterms to

obtain a Sum-Of-Maxterms. These two implementations are considered to be 2-level logic implementations due to their structure. These implementations, however, are by no means low in cost since they have large gates (i.e., gates with many inputs). Boolean algebra can be applied to simply these expresions and still yield 2-level circuit implementations in the form of SOP or POS depending on what we are interested in.

6 *Circuits implemented with only **NAND** and/or **NOR***

We can implement any circuit with **AND**, **OR**, and **NOT** gates, but we can implement any circuit using *only* **NAND** or **NOR** (or a combination of the two types). We might do this for certain considerations — e.g., due to technology considerations where **NAND** and **NOR** gates are cheaper to implement. The ability to use only **NAND** or **NOR** follows from the observation that we can make **AND**, **OR**, and **NOT** gates with only **NAND** or **NOR** gates.

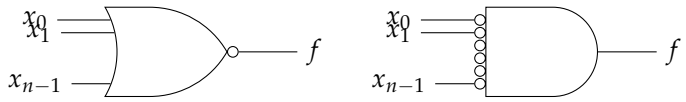
It is useful to note that an **NAND** gate performs the same operation as an **OR** gate that has inverted inputs. This is illustrated in Figure 6.1.



$$f = \overline{x_{n-1} \cdots x_1 x_0} = \overline{x_{n-1}} + \cdots + \overline{x_1} + \overline{x_0}$$

Figure 6.1: Illustration of **NAND** performing the same operation as an **OR** gate with inverted inputs.

It is also useful to note that an **NOR** gate performs the same operation as an **AND** gate that has inverted inputs. This is illustrated in Figure 6.2.



$$f = \overline{x_{n-1} + \cdots + x_1 + x_0} = \overline{x_{n-1}} \bullet \cdots \bullet \overline{x_1} \bullet \overline{x_0}$$

Figure 6.2: Illustration of **NOR** performing the same operation as an **AND** gate with inverted inputs.

6.1 *Converting 2-level SOP and POS expressions*

Given a SOP obtaining a NAND only implementation is trivial — apply a double inversion and use the DeMorgan theorem being careful of where we "leave" the inversions in the final expression. This is illustrated in the example shown in Figure 6.3.

$$
\begin{aligned}
f &= x1!x2+!x1x2+x3 \\
&= !(!f) \\
&= !(!(x1!x2+!x1x2+x3)) \\
&= !(!(x1!x2)!(!x1x2)!(x3)) \\
&= \underbrace{!(\overbrace{!(x1!x2)}^{NAND}\overbrace{!(!x1x2)}^{NAND}\overbrace{!(x3)}^{INV})}_{NAND}
\end{aligned}
$$

Figure 6.3: Example of obtaining a **NAND** only circuit implementation from a SOP.

Similarly, Given a POS obtaining a **NOR** only implementation is trivial — apply a double inversion and use the DeMorgan theorem being careful of where we âĂIJleaveâĂİ the inversions in the final expression. This is illustrated in the example shown in Figure 6.4.

$$
\begin{aligned}
f &= (x1+x2)(x3+x4)(x5) \\
&= !(!f) \\
&= !(!((x1+x2)(x3+x4)(x5))) \\
&= !(!(x1+x2)+!(x3+x4)+!(x5)) \\
&= \underbrace{!(\overbrace{!(x1+x2)}^{NOR}+\overbrace{!(x3+x4)}^{NOR}+\overbrace{!(x5)}^{INV})}_{NOR}
\end{aligned}
$$

Figure 6.4: Example of obtaining a **NOR** only circuit implementation from a POS.

6.2 *Multi-level circuit conversions*

For circuits which are not SOP or POS we can still do the conversion (note that this also works for SOP or POS). We insert "double inversions" where necessary to convert gates appropriately. We might end up with some left-over inverters, but that's okay.

6.2.1 *Conversion to **NAND***

The approach to convert a circuit to only **NAND** gates is as follows:

1. Insert double inversions prior to each **OR**;
2. Insert double inversions after every **AND** gate;
3. Convert gates and cancel out any remaining double inversions.

The purpose of inserting double inversions is to avoid changes to the logic operation performed by the circuit. An example is illustrated in Figures 6.5 through 6.7.

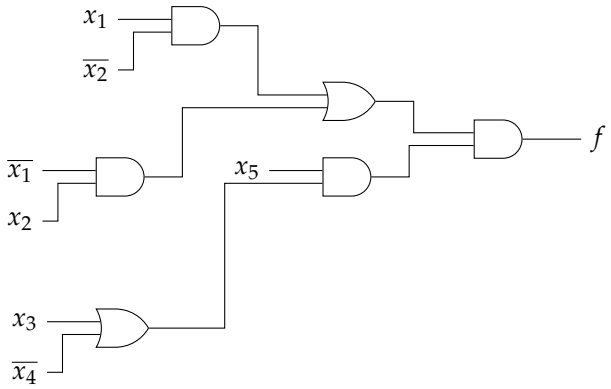


Figure 6.5: The original circuit which we would like to convert to **NAND**-only logic gates.

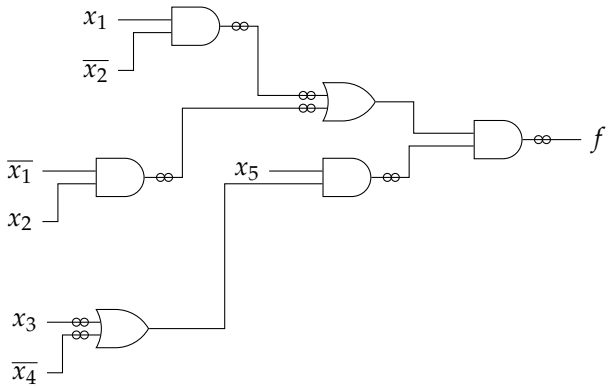


Figure 6.6: Insertion of double inversions to facilitate the conversion of the existing logic gates to **NAND** gates.

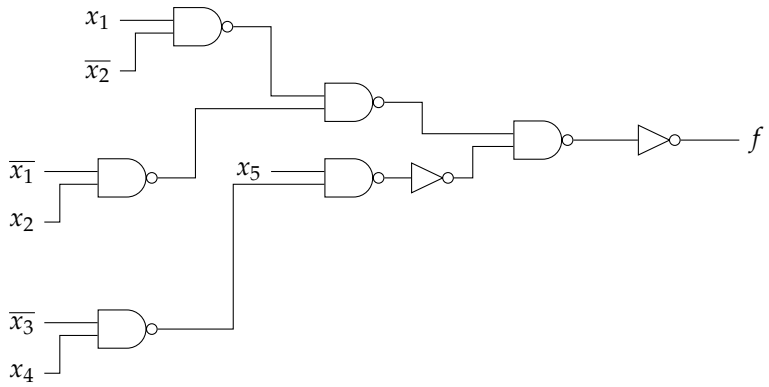


Figure 6.7: Final circuit converted to only **NAND** gates.

6.2.2 *Conversion to NOR*

The approach to convert a circuit to only **NOR** gates is as follows:

1. Insert double inversions prior to each **AND**;
2. Insert double inversions after every **OR** gate;
3. Convert gates and cancel out any remaining double inversions.

Note that there might be some left-over inverters. Note that the use of double inversion ensures that the circuit doesn't change its operation. An example is illustrated in Figures 6.8 through 6.10.

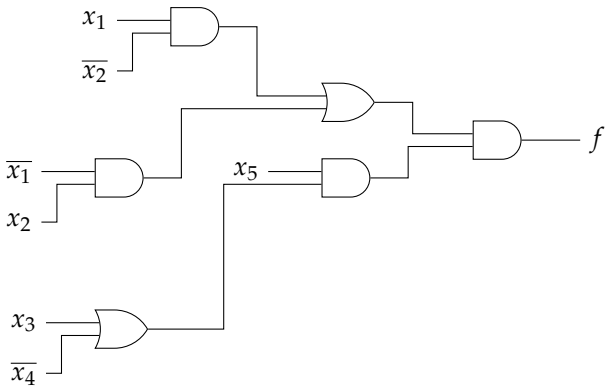


Figure 6.8: The original circuit which we would like to convert to **NOR**-only logic gates.

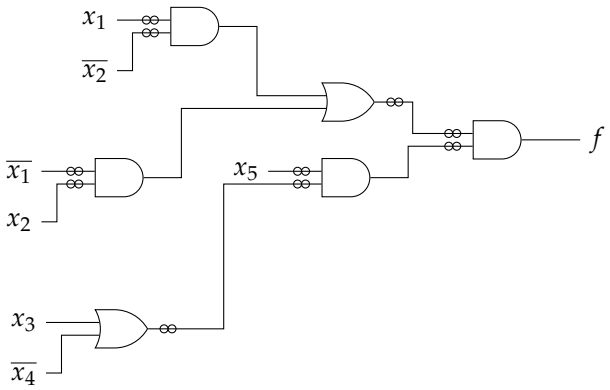


Figure 6.9: Insertion of double inversions to facilitate the conversion of the existing logic gates to **NAND** gates.

6.3 *Summary*

The **NAND** and **NOR** are interesting operators because they can be used to implement any circuit exclusively. Two level circuits can be converted using Boolean algebra. For multi-level circuits, we can use inverters to help convert gates as required.

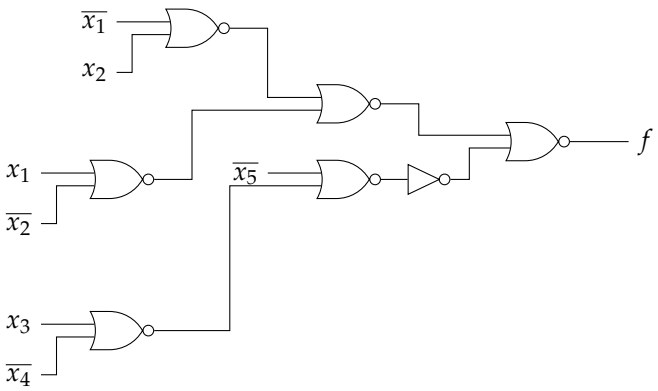


Figure 6.10: Final circuit converted to only **NOR** gates.

7 Karnaugh maps

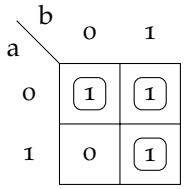
Karnaugh maps are an alternative way (compared to a truth table) to describe logic functions; they are useful for displaying logic functions with up to 4 or 5 inputs maximum. Whereas truth tables describe a function in a tabular format, Karnaugh maps describe a function in a *grid-like* format. The real benefit of Karnaugh maps lies in their ability to allow one to minimize logic functions graphically — for functions with only a few inputs, this tends to be much simpler compared to using Boolean algebra. We will consider Karnaugh maps for finding minimized SOP and POS expressions.

7.1 Two-variable Karnaugh maps for minimum SOP

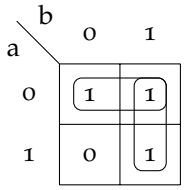
Consider the logic function shown in Table 7.1. The canonical SOP for this function is $f = a'b' + a'b + ab$. Figure 7.1 shows the Karnaugh map for this function. It is a 2×2 grid of entries. One input variable is used to index the rows of the Karnaugh map while the other input variable is used to index the columns of the Karnaugh map. The entries in the Karnaugh map correspond to the required values of f with the input variables are set to particular values. Each 1×1 rectangle in the Karnaugh map corresponds to a minterm. We can see that adjacent minterms differ by a change in only one variable. Therefore, adjacent rectangles can be merged together — the result is no longer a minterm, but rather a product term which encloses (includes) all minterms contained within. This is also shown in Figure 7.1 along with the resulting Boolean algebra. Therefore, we can see that Karnaugh maps provide a useful way to: 1) decide which terms should be duplicated; and 2) which terms should be grouped and factored to simply the Boolean expression. The minimum SOP in

a	b	f
0	0	1
0	1	1
1	0	0
1	1	1

Table 7.1: Example 2-input logic function.



$$f = a'b' + a'b + ab$$



$$f = a'b' + a'b + a'b + ab$$
$$f = a'(b' + b) + (a' + a)b = a' + b$$

Figure 7.1: Example Karnaugh map for 2 variables. Adjacent minterms (rectangles) can be merged together to form product terms (larger rectangles). A minterm can be duplicated as required. The merging of rectangles corresponds to Boolean algebra performed on the logic function.

this example is $f = a' + b$.

7.2 *Three-variable Karnaugh maps for minimum SOP*

Consider the logic function shown in Table 7.2. The canonical SOP for this function is $f = a'b'c + ab'c + abc' + abc$. Figure 7.2 shows

a	b	c	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Table 7.2: Example 3-input logic function.

the Karnaugh map for this function. It is a table with 8 entries — one for each possible value of the function. One variable is used to index the rows while two variables are used to index the columns. Notice that the numbering of columns is not consecutive. The numbering is done to ensure that between any two adjacent columns, only one variable changes its value. We can merge adjacent rectangles in the Karnaugh map to determine that the minimum SOP is $f = b'c + ab$. Some additional three variable Karnaugh maps are shown in Figure 7.3. Notice that in these examples, we can obtain larger rectangles. We should now be able to deduce the technique for finding minimum SOP implementations using Karnaugh maps. We identify rectangles of 1s in the Karnaugh map. The rectangles should be as large as possible without enclosing any 0s of the function — larger rectangles correspond to product terms with fewer inputs.

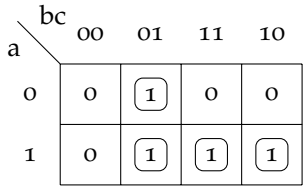
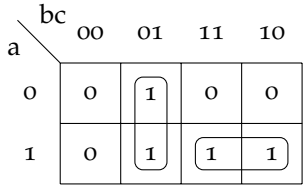
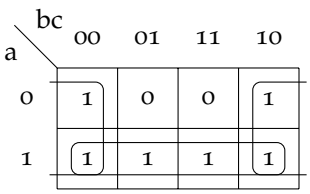


Figure 7.2: Example Karnaugh map for 3 variables. Adjacent minterms (rectangles) can be merged together to form product terms (larger rectangles). A minterm can be duplicated as required. The merging of rectangles corresponds to Boolean algebra performed on the logic function.

$$f = a'b'c + ab'c + abc' + abc$$
$$f = (a' + a)b'c + ab(c' + c)$$
$$f = b'c + ab$$

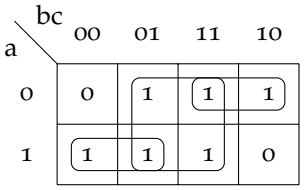


Larger rectangles are formed by starting with minterms and merging adjacent rectangles. Therefore, rectangles **must always be doubling in size — the area of a rectangle is always a power of** 2. Finally, we select the fewest rectangles (product terms) as required in order to make sure all the 1s of the logic function are included (covered) in the final SOP.



$$f = a'b'c' + a'bc' + ab'c' + ab'c + abc' + abc$$
$$f = a'b'c' + a'bc' + ab'c' + ab'c' + ab'c + abc' + abc' + abc$$
$$f = (a' + a)b'c' + (a' + a)bc' + ab'(c' + c) + ab(c' + c)$$
$$f = b'c' + bc' + ab' + ab$$
$$f = (b' + b)c' + a(b' + b)$$
$$f = c' + a$$

Figure 7.3: More examples of 3 variable Karnaugh maps.



$$f = a'b'c + a'bc' + a'bc + ab'c' + ab'c + abc$$
$$f = a'b'c + a'bc' + a'bc + a'bc + ab'c' + ab'c + ab'c + abc$$
$$f = (a' + a)b'c + a'b(c' + c) + (a' + a)bc + ab'(c' + c)$$
$$f = b'c + a'b + bc + ab'$$
$$f = (b' + b)c + a'b + ab'$$
$$f = c + a'b + ab'$$

7.3 *Four-variable Karnaugh maps for minimum SOP*

Figure 7.4 shows the Karnaugh map for a four input function. Here, we use two inputs to label the rows on the Karnaugh map and two input variables to label the columns of the Karnaugh map. Notice the numbering of the rows and columns — it is not sequential, but rather done to ensure that adjacent rows (or columns) different by only one changing variable. Figure 7.4 also shows the minimization of the logic function using the Karnaugh map which yields $f = c' + a'd' + bd'$.

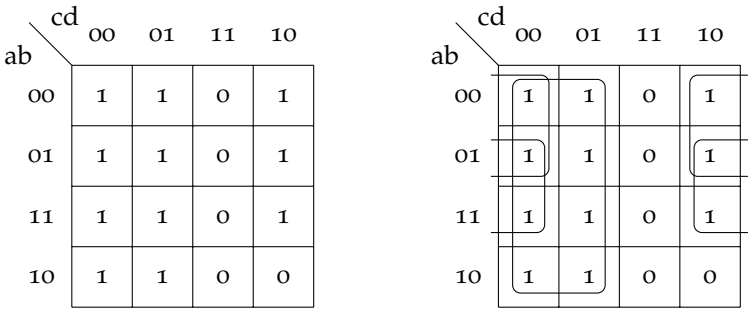


Figure 7.4: Example of a 4 variable Karnaugh map. Also shown is its minimization.

When optimizing logic functions, we often have choices and can obtain equally good (but different) implementations of a function f. This is shown in Figure 7.5 which shows two possible (and minimal) implementations of f. Both f_1 and f_2 implement f and have equal cost. It can be proven that $f_1 = f_2$.

We should also be careful to watch out for corners in four variable maps. Recall that the left and right columns are adjacent and that the top and bottom rows are adjacent. Therefore, the corners are also adjacent. This is shown in Figure 7.6.

$$f_1 = a'c' + cd + ac$$

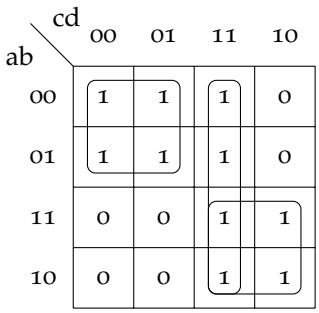


Figure 7.5: Two different implementations of a 4 variable function. Both functions are equally good in terms of cost.

$$f_1 = a'c' + a'd + ac$$

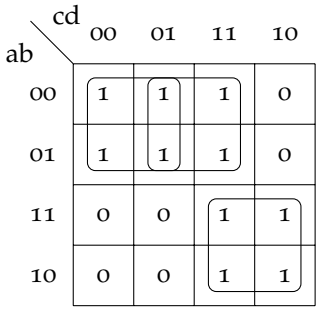
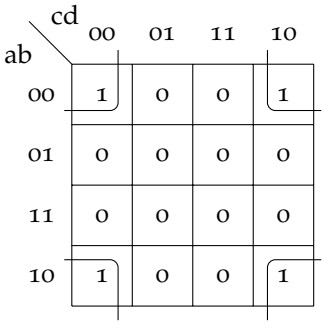


Figure 7.6: In four variable Karnaugh maps, the corners can be grouped.

$$f_1 = a'b'c'd' + a'b'cd' + ab'c'd' + ab'cd'$$
$$f_1 = a'b'(c' + c)d' + ab'(c' + c)d'$$
$$f_1 = a'b'd' + ab'd'$$
$$f_1 = (a' + a)b'd'$$
$$f_1 = b'd'$$



7.4 *Larger Karnaugh maps for minimum SOP*

A five variable Karnaugh map can be drawn by using two, four variable maps. If we pick one input, then each four variable Karnaugh map corresponds to one of the two possible values of the selected input. Figure 7.7 shows a five variable Karnaugh map in which input a has been selected; the left four variable map corresponds to $a = 0$ while the right four variable map corresponds to $a = 1$. We can proceed to minimize each four variable map, but should realize that identical rectangles in each map can also be merged removing the dependence on the last variable (in this case a). In Figure 7.7,

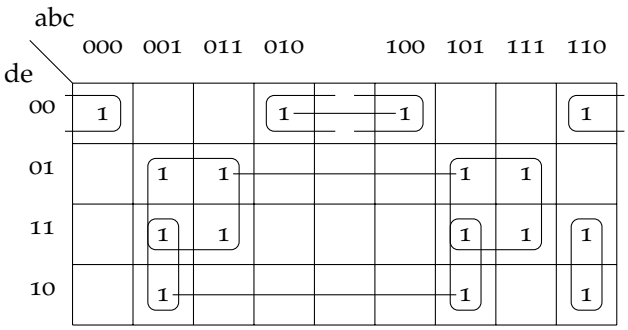


Figure 7.7: Example of a 5 variable Karnaugh map composed of two 4 variable maps. One can visualize one Karnaugh map "on top" of the other. The horizontal lines drawn between the two 4-variable maps indicate that they can be merged together to remove the dependence on the controlling variable which, in this case, is a.

we can compute f by considering each four variable map which gives

$$f = \overbrace{a'ce + a'b'cd + a'c'd'e'}^{\text{when } a=0} + \overbrace{ace + ab'cd + ac'd'e' + abc'd}^{\text{when } a=1} = \overbrace{ce + b'cd + c'd'e'}^{\text{identical terms in each map}} + \underbrace{abc'd}_{\text{unique terms in each map}}.$$

Without going into details (or an example), a six variable Karnaugh map can be constructed by using four, 4 variable Karnaugh maps. In this case, the four maps are arranged in a 2×2 grid. Karnaugh maps for functions with more than 6 variables become impractical to draw.

7.5 *Karnaugh maps for minimum POS*

The procedure for finding minimum POS implementations of logic functions using Karnaugh maps is more or less the same as with SOP, but rather than focusing on the 1s of the function and writing product terms, we must concentrate on the 0s of the function and write sum terms. A simple example of POS minimization with the associated Boolean algebra is shown in Figure 7.8.

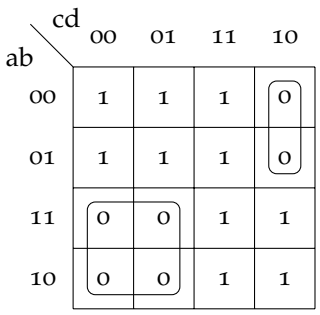


Figure 7.8: Finding minimum POS expressions using Karnaugh maps. The example is a 4-input logic function.

$$f = (a + b + c' + d)(a + b' + c' + d)(a' + b' + c + d)(a' + b' + c + d')(a' + b + c + d)(a' + b + c + d')$$
$$f = [(a + c' + d) + (bb')][(a' + b' + c) + (dd')][(a' + b + c) + (dd')]$$
$$f = (a + c' + d)[(a' + b' + c)(a' + b + c)]$$
$$f = (a + c' + d)[(a' + c) + (b'b)]$$
$$f = (a + c' + d)(a' + c)$$

7.6 *Karnaugh maps and don't cares*

Sometimes we might be given a logic function and will be told that certain inputs are guaranteed not to appear. In such cases, we do not really care what the output of the logic function will be since it will never happen. We might be tempted to set the output to either 1 or 0, but if we consider the output to be a don't care (denoted by and "X"), then we can exploit the fact that this output can be either 0 or 1 when we minimize the function — depending on how we set the don't cares, we might end up with a more simplified expression. Figure 7.9 shows a four variable function for which the function output does not matter for three different input patterns. Figure 7.10 shows the optimization of the function when all the don't cares are forced to be 0 and then forced to be 1. If the don't cares are forced to be all 0, then we find that $f = a'b'd + c'd'$. Similarly, if the don't cares are forced to be all 1, then we find that $f = a'b' + cd + a'd$.

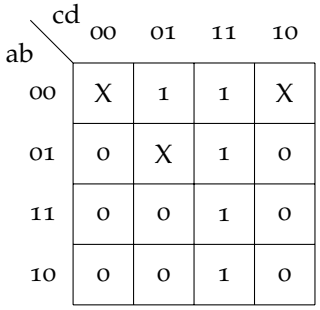


Figure 7.9: Example Karnaugh map for a four variable function with three don't cares.

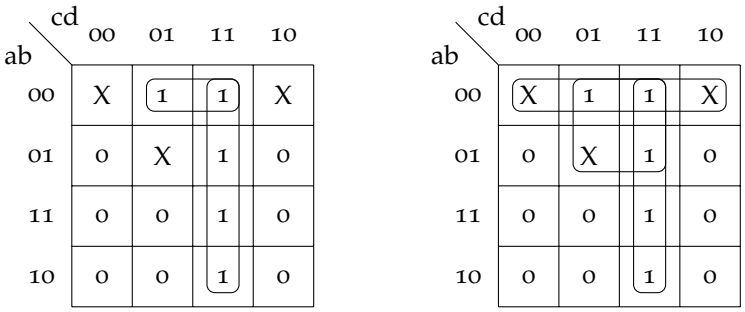


Figure 7.10: Minimization of f from Figure 7.9 if all the don't cares are considered to be 0 and 1.

The cost of these two implementations are 10 and 13, respectively. However, if we set some don't cares to be 0 while others to be 1, then we can obtain the implementation $f = a'd + cd$ which has cost 9. This implementation is shown in Figure 7.11. Here, those don't cares not enclosed in a rectangle are set to 0 while those don't cares which are enclosed in a rectangle are set to 1. The procedure for figuring out how to properly set don't cares stems from the observation that larger rectangles correspond to product terms with fewer inputs. Therefore, if a rectangle can be enlarged by enclosing a don't care, then we should do so. In Figures 7.9 and 7.11, we can see this — the product term $a'b'd$ can be enlarged to $a'd$ is one don't care is forced to be 1. In other words, the **optimization procedure tells us how to set the don't cares** to arrive at a minimized expression.

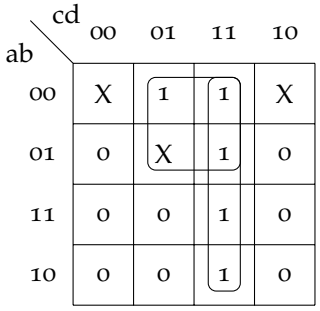


Figure 7.11: The minimum SOP implementation of f exploiting don't cares.

7.7 Logical and algebraic equivalence

Often when optimization a logic function with don't cares, we can end up with two different (but equally good) implementations which exploit the don't cares in different ways. An example of this is illustrated in Figure 7.12 which shows the Karnaugh map for a function f. In Figure 7.12, both f_1 and f_2 are implementations of f. Both

$f_1 = a'd + cd$

ab \ cd	00	01	11	10
00	X	1	1	X
01	0	X	1	0
11	0	0	1	0
10	0	0	1	0

$f_2 = a'b' + cd$

ab \ cd	00	01	11	10
00	X	1	1	X
01	0	X	1	0
11	0	0	1	0
10	0	0	1	0

Figure 7.12: Two different implementations of a 4 variable function which exploit don't cares differently.

f_1 and f_2 produce produce the proper value for f when required — f_1 and f_2 only differ in how the don't care situations are treated. Therefore, both f_1 and f_2 **are functionally equivalent** — they both implement f as specified. However, f_1 and f_2 **are not algebraically equivalent**. In other words, in is impossible to show that f_1 is equal to f_2 because they are not.

7.8 Multiple output functions and product term sharing

It is often the case that we have multiple logic functions which share the same set of inputs. This is shown in Figure 7.13 where we are given two functions f and g which are both 4 variable functions of inputs a, b, c and d. If we want to obtain an optimized circuit, we might consider the optimization of f and g separately. This is

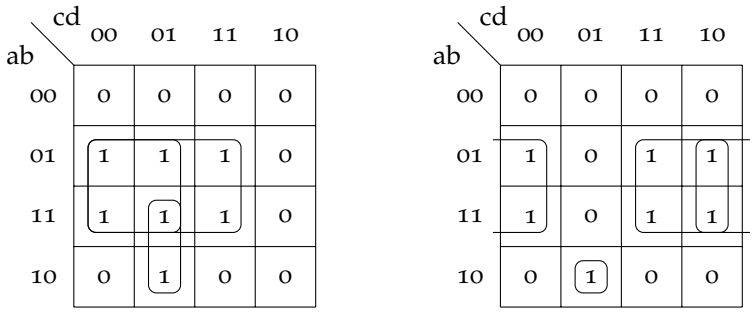


Figure 7.13: Two 4-variable functions f and g.

also shown in Figure 7.13 and results in $f = bc' + bd + ac'd$ and $g = bd' + bc + ab'c'd$. The total cost of f is 14 and the total cost of g is 15. By inspection, f and g do not share any common logic and therefore the cost of the total circuit (obtained by minimizing f and g separately) is 29.

However, if we consider **both** functions f and g **at the same time**, then we can do better. Consider the Karnaugh maps shown in Figure 7.17. In Figure 7.17 the identified product terms to imple-

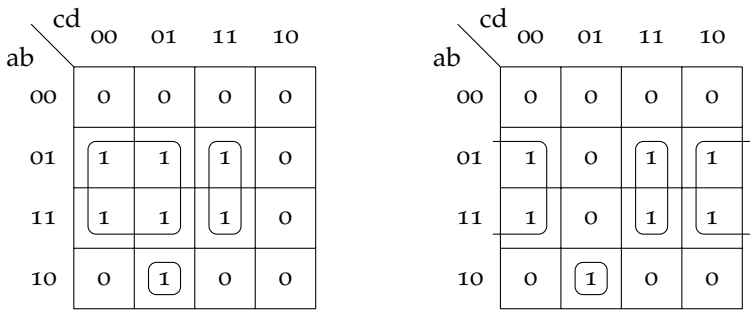


Figure 7.14: Two 4-variable functions f and g.

ment f and g are not necessarly the best for minimization of f and g separately. In fact, using the product terms in Figure 7.17, we get $f = bc' + bcd + ab'c'd$ and $g = bd' + bcd + ab'c'd$. If the functions are considered separately, we find the cost of f is 16 and the cost of g is 16. However, we can observe that both f and g require the product terms bcd and $ab'c'd$ — these product terms are referred to as shared product terms. Hence, we do not need to count them twice. The actual cost of the circuit is only 23 if we share product terms be-

tween f and g. The circuit implementing both f and g is shown in Figure 7.15. Notice that several product terms are used by both f and g.

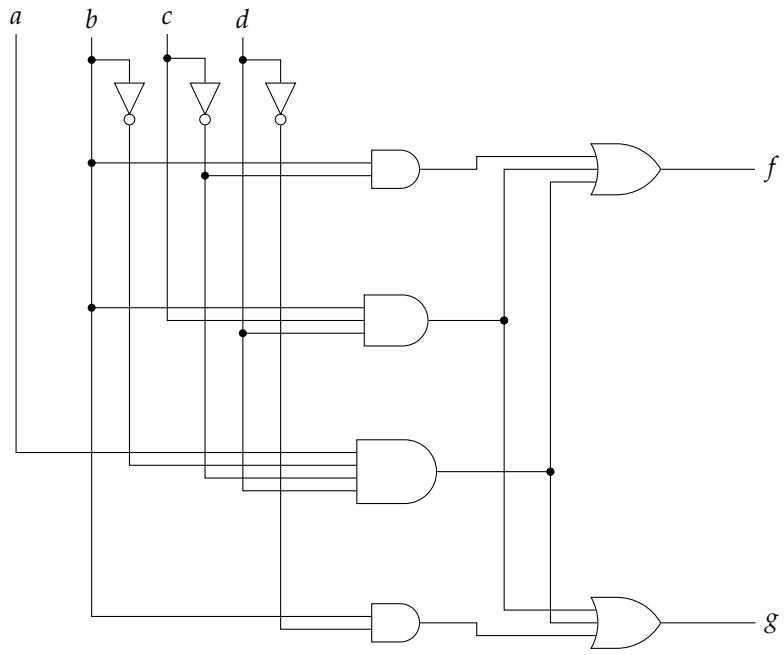


Figure 7.15: Circuit implementing two functions with shared product terms.

The conclusion that we can reach is that **if posed with the problem of optimizing a circuit with multiple outputs, it can often be better to accept non-minimal implementations of individual functions if product terms can be shared between multiple functions.**

7.9 *Implicants, prime implicants, essential implicants and covers*

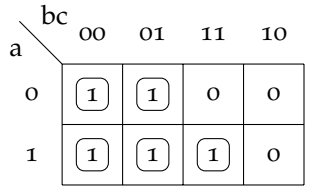
There is some terminology that we encounter when dealing with logic functions. This terminology also helps explain the minimization procedure so it is worth understanding. It is most common to talk about this terminology in terms of SOP function implementations.

If we are given an n variable logic function f and are told that the output of f is 0 or 1 for every 2^n possible input patterns, then f is said to be **completely specified**. If some outputs of f are don't cares, then f is said to be **incompletely specified**. For any function f we can split the minterms into three different sets. The set of minterms for which the function is 1 is called the **on set**. The set of minterms for which the function is 0 is called the **off set**. Finally, the set of minterms for which the function is a don't care is called the **don't care set**.

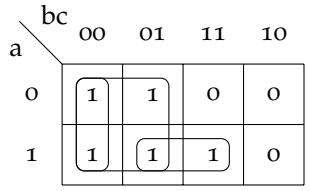
A product term is called an **implicant** of a function if the function

is 1 for all minterms contained within the product term. **In terms of a Karnaugh map, all rectangles which include only** 1**s correspond to implicants of a function**. An implicant of a function is called a **prime implicant** if the removal of any input from the implicate results in a product term which is **no longer** an implicant. **In terms of a Karnaugh map, a prime implicant corresponds to a rectangle of** 1**s which cannot be made any larger (e.g., by merging with another rectangle) without enclosing a** 0. An **essential prime implicant** is a prime implicant which includes a minterm that is not found in any other prime implicant. Finally, a **cover** of a function is a collection of enough implicates to account for all situations where a logic function is a 1. **We can always form a cover of a function using only prime implicants if we so chose.** A cover that consists of only prime implicants **must** include all of the essential prime implicants — if not, then some minterms for which f produces a 1 would not be included and the implementation would be wrong.

An illustration of this terminology is illustrated in Figure 7.16 in terms of a Karnaugh map. To form an implementation of f, we must select enough implicants to cover all the situations for which f produces a 1. This can be accomplished by selecting the two prime implicants giving $f = b' + ac$.



The minterms $a'b'c', a'b'c, ab'c', ab'c, abc$ are implicants.
None of the minterms are prime implicants.



The product terms $b'c', b', ac$ are implicants.
However, only b' and ac are prime implicants.
Both b' and ac also happen to be essential prime implicants.

Figure 7.16: Illustration of minterms, implicants and prime implicants.

Implementations of functions which use only prime implicants tend to have low cost. This is because prime implicants tend to have less inputs. Therefore, a reasonable procedure to optimize a function

is as follows:

1. Generate all prime implicates;
2. Identify the essential prime implicants;
3. Include all the essential prime implicants in the cover for f. If the function is completely covered, then stop.
4. Otherwise, include as few non-essential prime implicants as possible to complete the cover of f.

This is precisely what we try to do with Karnaugh maps: We attempt to find rectangles which are as large as possible — this corresponds to identifying prime implicants. We then select the rectangles which includes minterms not included by other rectangles (we include the essential prime implicants). Finally, we include as few additional rectangles as possible to properly implement f. An example which more explicitly shows this procedure is illustrated in Figure 7.17. From Figure 7.17, we can see two different minimal

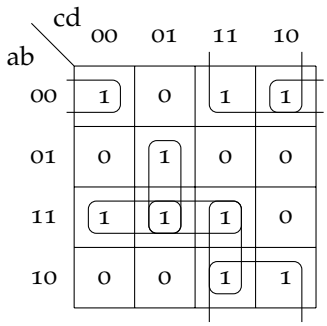


Figure 7.17: Four variable Karnaugh map illustrating the minimization procedure in terms of prime implicants.

implementations for f which are equally as good. Specifically, $f = a'b'd' + b'c + abc' + bc'd + abd$ or $f = a'b'd' + b'c + abc' + bc'd + acd$ both implement f and have equal cost. Both options include the product terms $a'b'd', b'c, abc'$ and $bc'd$ which are essential prime implicants. However, with only the essential prime implicants, we do not have a complete implementation of f since we have not covered the minterm $abcd$. We have two options to cover $abcd$ — we can either pick abd or acd which are both prime implicants (but not essential); we also only need to include one of them. This yields are two, equally optimized implementations of f.

7.10 *Karnaugh maps for **XOR** operators*

Turns out that **XOR** gates are useful in arithmetic circuits (e.g., to build adders) and other sorts of circuits. Such circuits are "special-

ized" and the **XOR** operations that occur are easy to see. Sometimes, we can "discover" a **XOR** gate "buried" or "hidden" inside of a general logic expression. If we can find it, it can help to reduce the implementation cost. Recall that **XOR** and **NXOR** operators perform the "odd" and "even" function, respectively.

K-Maps are not "pretty" in terms of optimization. The K-Maps for 4-input **XOR** and **NXOR** are illustrated in Figure 7.18. Implementing an **XOR** or **NXOR** using **AND**, **OR** and **NOT** would be a disaster.

ab \ cd	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	1	0	1
10	1	0	1	0

$f = a \oplus b \oplus c \oplus d$

ab \ cd	00	01	11	10
00	1	0	1	0
01	0	1	0	1
11	1	0	1	0
10	0	1	0	1

$f = \overline{a \oplus b \oplus c \oplus d}$

Figure 7.18: Karnaugh maps for 4-input **XOR** and **NXOR** operators. Clearly, these operators *can* be implemented using **AND**, **OR** and **NOT** gates since we can write either an SOP or a POS, but there is no optimization possible.

Sometimes we might just have an expression which yields a K-Map that sort of looks like an **XOR**. This is illstrated in Figure 7.19. We can perform Boolean algebra as follows:

ab \ cd	00	01	11	10
00	0	1	0	0
01	1	0	1	1
11	0	1	0	0
10	1	0	1	1

Figure 7.19: Karnaugh maps for 4-input function that is not quite a **XOR**, but looks similar.

$$\begin{aligned} f &= a'b'c'd + a'bc'd + a'bcd + a'bcd' + abc'd + ab'c'd' + ab'cd + ab'cd' \\ &= a'b'c'd + abc'd + a'bc + ab'c + a'bd' + ab'd' \end{aligned}$$

The amount of algebra allowed results in a minimum SOP which has a cost of 33.

However, we can continue to perform Boolean algebra (the K-Map is no longer of help since its for SOP and POS) as follows:

$$\begin{aligned}
f &= a'b'c'd + a'bc'd + a'bcd + a'bcd' + abc'd + ab'c'd' + ab'cd + ab'cd' \\
&= a'b'c'd + abc'd + a'bc + ab'c + a'bd' + ab'd' \\
&= (a'b' + ab)(c'd) + (a'b + ab')c + (a'b + ab')d' \\
&= \overline{(a \oplus b)}(c'd) + (a \oplus b)c + (a \oplus b)d' \\
&= \overline{(a \oplus b)}(c'd) + (a \oplus b)(c + d') \\
&= \overline{(a \oplus b)}(c'd) + (a \oplus b)\overline{(c'd)} \\
&= (a \oplus b) \oplus (c'd) \\
&= a \oplus b \oplus (c'd)
\end{aligned}$$

If we assume an 3-input **XOR** gate costs the same as any other 3-input gate, then this implementation costs 7. Although **XOR** are more expensive (in reality), this implementation is most certainly cheaper than the minimum SOP!

Figure 7.20 illustrates another K-Map which begins to look like there might be an **XOR** operator involved. In Figure 7.20, we can observe that the input a is acting like a "gate" — when $a = 0$, f is an **XOR** and when $a = 1$, $f = 0$. In other words... $f = \overline{a}(b \oplus c \oplus d)$ which (again) is likely much cheaper than the minimum SOP implementation. In general, finding **XOR** operations hidden in general logic expressions can be quite difficult, but it's cool.

ab \ cd	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	0	0	0
10	0	0	0	0

Figure 7.20: Karnaugh maps for 4-input function that is not quite a **XOR**, but looks similar.

7.11 Summary

Karnaugh maps provide an alternative way (compared to truth tables) to present logic functions. They are practical for functions up to an including about 5 inputs, but become impractical to draw (compared to the tabular nature of a truth table) for a larger number of inputs.

The real benefit of Karnaugh maps for smaller functions is that they permit the minimization of logic functions to be performed graphically rather than via Boolean algebra. Rectangles in a Karnaugh map which enclose the 1s of a function correspond to product terms which can be used to implement a minimized SOP. Conversely, rectangles of 0s in the Karnaugh map correspond to sum terms which can be used to implement a minimized POS. It is important to remember that there is no magic taking place. Rectangles in the Karnaugh map are simply a graphical way of performing Boolean algebra — in the case of an SOP, including a minterm in multiple rectangles corresponds to duplicating the minterm. Merging two adjacent rectangles of 1s corresponds to taking two product terms, factoring out the common part and then simplifying. Karnaugh maps are useful tools when dealing with logic minimization of smaller functions by hand.

8 Quine-McCluskey optimization

Rather than using Karnaugh maps which is a graphical method to optimize logic functions with only a few inputs, we can chose to use a tabular method instead. This tabular method is commonly referred to as the *Quine-McCluskey method*. This method will work for logic functions with any number of inputs.

The method is based on using a table to generate all of the prime implicants for a logic function and, subsequently, using a matrix to decide on which prime implicants to select to cover the function. The Quine-McCluskey method will be illustrated through an example. Consider the 4-input logic function given by $f = f(a,b,c,d) = \sum m(0,2,3,5,10,11,12,13,15)$. These minterms are listed as shown in Figure 8.1. There are a couple of important points to note

0	0000
2	0010
3	0011
5	0101
12	1100
10	1010
11	1011
13	1101
15	1111

Figure 8.1: The first step of the Quine-McCluskey method which involves listing all of the minterms and rows of the truth table grouped according to the number of variables which are equal to 1. At this point all terms involved all input variables.

about Figure 8.1. Minterms are *grouped* according to the number of positive literals (or, equivalently, the number of 1s in that row of the truth table). This is because the Quine-McCluskey method relies on the Boolean operation $x_i x_j + x_i x_j^{'} = x_i$. This Boolean operation means that we are looking for situations in which rows (minterms) are identical *except for a single variable* which is 0 in one row and 1 in another row. By arranging the minterms as shown in Figure 8.1, we only need to compare a minterm with minterms if the immediately preceding group.

We now start to perform pairwise comparisons of minterms looking for minterms which differ by only the value of a single variable.

Performing this task leads to the table shown in Figure 8.2. In

0, 2	00X0
2, 3	001X
2, 10	X010
3, 11	X011
10, 11	101X
5, 13	X101
12, 13	110X
11, 15	1X11
13, 15	11X1

Figure 8.2: Next step of the Quine-McCluskey method which involves comparing minterms to identify variables which can be removed. The result is a set of product terms in which one variable has been removed.

Figure 8.2, the removed variables which result from the pairwise comparison of minterms are marked with an X. We now proceed to perform the same pairwise comparisons on the product terms in Figure 8.2. Specifically, we look compare all product terms in a group with all product terms in the immediately preceeding group. Further, the variables marked with an X must match — when comparing product terms we must make sure to only compare product terms involving the same set of input variables. Performing these comparisons leads to the table in Figure 8.3. In this step, we find only

2, 3, 10, 11	X01X

Figure 8.3: Next step of the Quine-McCluskey method which involves comparing product terms to identify more variables which can be removed. The result is a set of product terms in which two variable has been removed.

one new product term can be generated which combines minterms 2, 3, 10 and 11.

The next step of the Quine-McClusky algorthm is to look at all the product terms generated and to mark which product terms were successfully paired when generating the next table — since these product terms have been combine into other product terms, they cannot be prime implicants and are therefore not required to implement f (since f can be generated using only prime implicants). The tables of product terms are repeated in Figure 8.4 and those product terms (minterms) which have been combine are checked. As marked in Figure 8.4, none of the minterms will be required to implement f as they have all been included in larger product terms. Similarly, several of the product terms themselves have been included in a larger product term. In summary, the Quine-McCluskey method has indicated that the prime implicants for f are the product terms $a'b'c'$, $bc'd$, abc', acd, abd, and $b'c$.

The final step of the Quine-McCluskey method is to compose a matrix in which the rows of the matrix show the prime implicants and the columns show the minterms. A matrix entry has a check mark if the corresponding minterm is included by the prime impli-

0	0000	✓	0, 2	00X0				
2	0010	✓	2, 3	001X	✓			
3	0011	✓	2, 10	X010	✓			
5	0101	✓	3, 11	X011	✓			
12	1100	✓	10, 11	101X	✓	2, 3, 10, 11	X01X	
10	1010	✓	5, 13	X101				
11	1011	✓	12, 13	110X				
13	1101	✓	11, 15	1X11				
15	1111	✓	13, 15	11X1				

Figure 8.4: All product terms generated by applying the Quine-McCluskey method. Those product terms which are **not** required are checked.

cant. This is shown in Figure 8.5. We now consider the matrix

Prime Implicant	**Minterm** 0	2	3	5	10	11	12	13	15
$0, 2 = 00X0$	✓	✓							
$5, 13 = X101$				✓				✓	
$12, 13 = 110X$							✓	✓	
$11, 15 = 1X11$						✓			✓
$13, 15 = 11X1$								✓	✓
$2, 3, 10, 11 = X01X$		✓	✓		✓	✓			

Figure 8.5: Matrix showing the minterms and prime implicants.

and look at the columns of the matrix. If we see a column with only a single checkmark, then we **must** include the appropriate product term to ensure that minterm is included in the final expression for f. This is equivalent to determining which of the prime implicants are essential prime implicants. In our example, it is clear that we must include product term $(0, 2)$ to cover minterm 0, product term $(2, 3, 10, 11)$ to cover minterms 3 and 10, product term $(5, 13)$ to cover minterm 5, and and product term $(12, 13)$ to cover minterm 12.

By including these minterms, we can generate a smaller matrix showing the remaining prime implicants and any remaining minterms which have not already been covered. This is shown in Figure 8.6. From Figure 8.6 we can see that we must still select a

Prime Implicant	**Minterm** 15
$11, 15 = 1X11$	✓
$13, 15 = 11X1$	✓

Figure 8.6: Matrix showing remaining minterms to cover after removing the essential prime implicants and all minterms included in the essential prime implicants.

prime implicant to cover minterm 15. It turns out that we have two choices and we can select either option. The final implementation of

f is given by

$$f = f(a,b,c,d) = a'b'd' + bc'd + abc' + b'c + \begin{cases} acd \\ abd \end{cases}$$

We will not do an example, but don't cares are also easily handled. For those rows of the truth table which are don't cares, we include them in the initial table along with the minterms and use the don't cares to aid in the generation of prime implicants. However, when we form the matrix we do **not** include the don't cares as columns in the matrix. This is because we are not required to cover the don't cares in the final implementation of f.

9 *Multi-level circuits*

Although we can obtain simplified 2-level SOP and POS expressions, it might be preferable to implement a function as a *multi-level* circuits. As the name implies, the "distance" from the output of the logic function back to its inputs is more that 2 logic gates. The reason multi-level circuits might be preferable is that, despite being optimized, a 2-level circuit implementation might require logic gates with more inputs than are available. Similarily, if we have multiple functions to implement we might find by some common terms shared amoung the different logic functions (beyond what might be accomplished by sharing product terms).

One simple example might be as follows. Consider the 4-input logic function $f = f(w,x,y,z) = x'y' + w'y' + wxyz + wx'z'$ which is a minimized SOP expression for f. The implementation of this circuit requires 2, 2-input **AND** gates, 1, 3-input **AND** gate, 1, 4-input **AND** gate, and 1, 4-input **OR** gate. The total cost is 20. However, by applying some simple factoring, we can see that f can be rewritten as $f = y'(w' + x') + wxyz + wx'z'$. This implementation is no longer 2-level and requires 1, 2-input **OR** gate, 1, 2-input **AND** gate, 1, 3-input **AND** gate, 1, 4-input **AND** gate, and 1, 3-input **OR** gate. The total cost is 19 which is slightly cheaper. Further, although we still require a 4-input **AND** gate we no longer require a 4-input **OR** gate. If we are restricted to only 3-input gates or less, we could consider additional Boolean algebra. We could write $f = y'(w' + x') + w(x + z')(x' + yz)$ which would require 4, 2-input **OR** gates, 2, 2-input **AND** gates, and 1, 3-input **AND** gate. The cost of this implementation is 22 which is larger than our original SOP but only requires a single 3-input gate. Similarly, we could factor differently and find $f = w'(x' + y') + w(xyz + x'z')$ which would require 3, 2-input **OR** gates, 3, 2-input **AND** gates, and 1, 3-input **AND** gate which also costs 22 and requires only a single 3-input **AND** gate. The last of these implementations is shown in Figure 9.1. It is interesting to note from Figure 9.1 that the circuit is also structured such that conversion to a **NAND** only implementation is possible. This is because during our factoring, we *just happened* to end up with a result that is a bunch

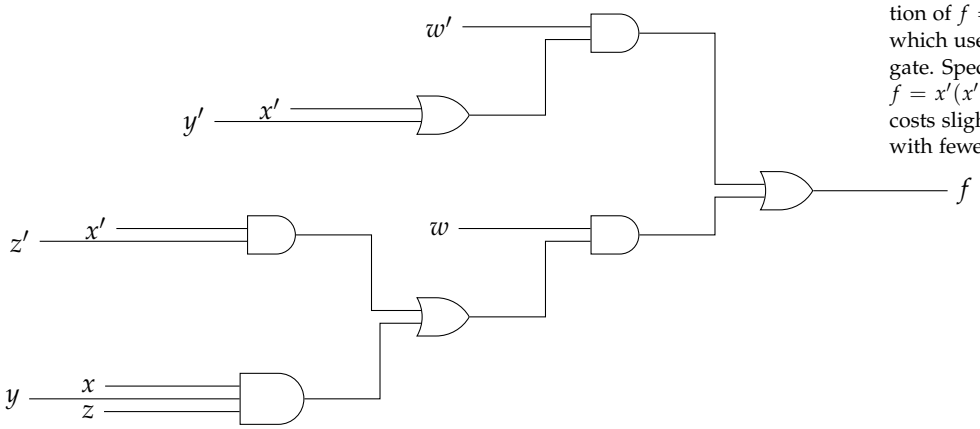


Figure 9.1: One multi-level implementation of $f = x'y' + w'y' = wxyz + wx'z'$ which uses at most a 3-input logic gate. Specifically, f is implemented as $f = x'(x' + y') + w(xyz + x'z')$ which costs slightly more but uses logic gates with fewer inputs.

of SOP sub-circuits connected together to implement f. We can see this by examining the structure of the implementation of f as follows:

$$f = \underbrace{w' \underbrace{(w' + y')}_{SOP} + w \underbrace{(xyz + x'z')}_{SOP}}_{SOP}$$

Since SOP circuits convert easily to **NAND**, the entire circuit in Figure 9.1 by simply replacing every logic gate with a **NAND** gate.

An an additional example, we can consider the the 4-input logic function $f = f(a, b, c, d) = (a + b + c')(a' + b' + d')(a + c + d')(a' + b' + c')$ which is a minimized POS implementation of f. The cost of this circuit is **21** and consists of 4, 3-input **OR** gates and 1, 4-input **AND** gate. However, assume that we only have 2-input logic gates available. If we simply used the associtive property of **OR** and **AND** gates, we could insert parentheses to get $f = f(a, b, c, d) = (((a + b) + c')((a' + b') + d'))(((a + c) + d')((a' + b') + c'))$ which is a poor implementation and would cost **33** with a circuit depth of 4. We can again consider the use of Boolean algebra to try and manipulate the expression which gives $f = (a + b + c')(a' + b' + d')(a + c + d')(a' + b' + c') = [(a + b)(a' + b') + c'][(a' + b')(a + c) + d']$. This expression only requires 2-input logic gates. The cost of this circuit is 24 and consists of 5, 2-input **OR** gates and 3, 2-input **AND** gates. The circuit diagram for this multilevel implementation is shown in Figure 9.2.

This second example is also interesting in that the problem could have been stated as find an implementation of f which uses only 2-input **NOR** gates in which no **NOR** gate is used as an inverter. The factoring has resulted in an implementation for f which consists of sub-circuits all implemented as POS which is seen as follows:

$$
\begin{aligned}
f &= (a+b+c')(a'+b'+d')(a+c+d')(a'+b'+c') \\
f &= \underbrace{[\underbrace{(a+b)(a'+b')}_{POS}+c'][\underbrace{(a'+b')(a+c)}_{POS}+d']}_{POS}
\end{aligned}
$$

Every logic gate in Figure 9.2 could therefore be replaced with a **NOR** gate which would result in a **NOR** only implementation of f.

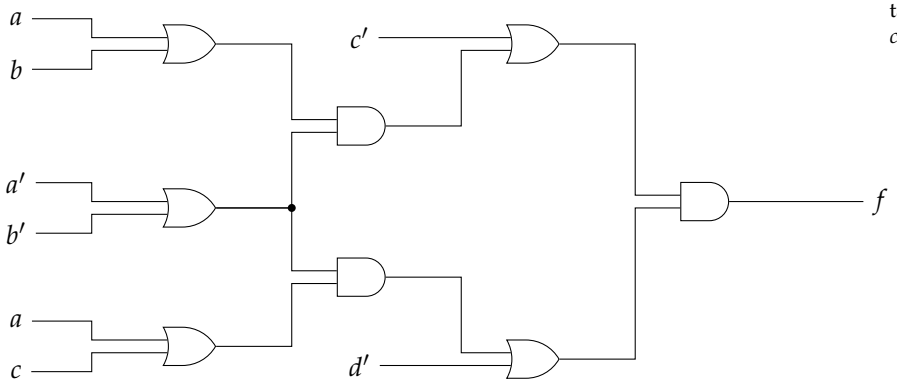


Figure 9.2: Multilevel circuit implementation of $f = f(a,b,c,d) = (a+b+c')(a'+b'+d')(a+c+d')(a'+b'+c')$.

In each of these cases, it turned out that we could easily convert the resulting multilevel circuits into either **NAND** only or **NOR** only circuits. This might not always be the case. It could be that other structures result when we apply Boolean algebra to manipulate a logic function to remove gates with a large number of inputs. Further, we might end up requiring inverters within the circuit or perhaps we will discover a **XOR** function which is best implemented with an **XOR** gate. Regardless, multilevel circuits provide useful and practical alternative implementations compared to using strictly minimized POS and SOP expressions.

10 Number representations

In this part, we will focus on how to represent numbers. In particular, we will focus on how we can represent both *unsigned integers* and *signed integers* using 0s and 1s. This, of course, is necessary in order for us to design circuits that can operate with values (e.g., building a circuit to perform addition).

We will also consider one way to represent numbers which have fractional parts.

10.1 Unsigned number representations

Here we will consider how to represent unsigned integers in different bases; in particular we are interested in representing values in the binary number system.

Numbers have a *value*. Values have *representations* in different *bases* using *digits*. In base-r, we have digits from $0, 1, \cdots, r-1$.

Popular bases and digits are

Base	**Digits**	
Base-10	$0, 1, \cdots, 9$	$\leftarrow$ *decimal*
Base-2	$0, 1$	$\leftarrow$ *binary*
Base-8	$0, 1, \cdots, 7$	$\leftarrow$ *octal*
Base-16	$0, 1, \cdots, 9, A, B, C, D, E, F$	$\leftarrow$ *hexadecimal*

The base-16 (hexadecimal) uses $A, \cdots, F$ for digits 10 through 15. In binary, the digits are often called *bits*.

10.1.1 Positional number representation

Consider any unsigned integer with value V. In base-r (r sometimes called the radix) using n digits (sometimes called coefficients), V is represented by $D = (d_{n-1}d_{n-2}\cdots d_1 d_0)_r$. Digit d_0 is the least significant digit and d_{n-1} is the most significant digit. This is called *positional number representation*. Note that the value of a unsigned integer V "looks the same" as its representation in base-10.

Given a representation D of some value V in base r using n digits,

we can compute the value V using

$$V = d_{n-1} \times r^{n-1} + d_{n-2} \times r^{n-2} + \cdots d_1 \times r^1 + d_0 \times r^0$$

Two examples are shown in Figure 10.6 and 10.7.

$$\begin{aligned}(219)_{10} &= 2 \times 10^2 + 1 \times 10^1 + 9 \times 10^0 \\ &= 200 + 10 + 9 \\ &= 219\end{aligned}$$

Figure 10.1: Computing the value of a number represented in base-10 using 3 digits.

$$\begin{aligned}(11011011)_2 &= 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 \\ &\quad +1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ &= 128 + 64 + 16 + 8 + 2 + 1 \\ &= 219.\end{aligned}$$

Figure 10.2: Computing the value of a number represented in base-2 using 8 bits.

A value's representation in base r can be computed using recursive division by r:

$$V = d_{n-1} \times r^{n-1} + d_{n-2} \times r^{n-2} + \cdots d_1 \times r^1 + d_0 \times r^0$$

Division by r gives

$$\frac{V}{r} = \underbrace{d_{n-1} \times r^{n-2} + d_{n-2} \times r^{n-3} + \cdots d_1 \times r^0}_{quotient} + \underbrace{\frac{d_0}{r}}_{remainder}$$

Note that one of the digits pops out as the remainder. We can repeat the division using the quotient to get the next digit. We stop once the quotient is 0 (which means all further "leading digits" will also be zero). An example of determining the representation of a value in base-2 is shown in Figure 10.3. It might be worth while to

		Quotient	**Remainder**	
$\frac{53}{2}$	=	26	1	$\rightarrow d_0 = 1$
$\frac{26}{2}$	=	13	0	$\rightarrow d_1 = 0$
$\frac{13}{2}$	=	6	1	$\rightarrow d_2 = 1$
$\frac{6}{2}$	=	3	0	$\rightarrow d_3 = 0$
$\frac{3}{2}$	=	1	1	$\rightarrow d_4 = 1$
$\frac{1}{2}$	=	0	1	$\rightarrow d_5 = 1$

Figure 10.3: Representing the value 53 in base-2. We stop when we do since the quotient becomes 0 and $d_6 = 0$, $d_7 = 0$, and so forth. The representation of 53 in base-2 is 110101_2.

consider the representations in base-2 of unsigned values as we count — this can be related to and help explain why we write the rows of truth tables in the order that we do. This is shown in Figure 10.4.

Value/Decimal Representation	**Binary Representation**
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
...	...

Figure 10.4: "Counting" in base-2.

Finally, we can also figure out the *range* of values we can represent in base-2 using n bits; We can represent the values $0 \cdots 2^n - 1$ in n bits. For example, with 4 bits we can represent the values $0 \cdots 15$. With 8 bits we can represent the values $0 \cdots 255$.

10.1.2 *Conversion between bases*

Conversion from base-a to base-b is easy. We first find the value of the base-a representation (the result is also the base-10 representation). Then, we use successive division to convert the value to base-b. An example is shown in Figure 10.5. Conversion from base-2 to

- **Step 1:** $110101_2 = 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^2 + 1 \times 2^0 = 32 + 16 + 4 + 1 = 53 = (53)_{10}$.
- **Step 2:** Successive division...

		Quotient	**Remainder**	
$\frac{53}{6}$	=	8	5	$\rightarrow d_0 = 5$
$\frac{8}{6}$	=	1	2	$\rightarrow d_1 = 2$
$\frac{1}{6}$	=	0	1	$\rightarrow d_2 = 1$

  and $53 = (125)_6$.
- **Step 3:** Check... $125_6 = 1 \times 6^2 + 2 \times 6^1 + 5 \times 6^0 = 36 + 12 + 5 = 53$.

Figure 10.5: Conversion of $(110101)_2$ to base-6.

base-8 and base-16 is easy since they are all powers of two — this allows us to simply group bits:

1. Converting binary to octal requires converting groups of 3-bits;
2. Converting binary to hexadecimal requires converting groups of 4-bits.

If the binary number is not a multiple of the group size, then we can arbitrarily add leading zeros if required. An example of conversion by grouping bits in given in Figure 10.1.2. Conversion from base-

- $1100111100101_2 = \underbrace{0001}_{4bits;leadingzeros}\ \underbrace{1001}_{4bits}\underbrace{1110}_{4bits}\underbrace{0101}_{4bits} = 19E5_{16}$.
- $1100111100101_2 = \underbrace{001}_{3bitsleadingzeros}\ \underbrace{100}_{3bits}\ \underbrace{111}_{3bits}\ \underbrace{100}_{3bits}\ \underbrace{101}_{3bits} = 14745_8$.

8 or base-16 to base-2 is simple; base-8 digits become groups of 3 bits and base-16 digits become groups of 4 bits in base-2, respectively.

10.2 Unsigned addition

Addition of unsigned numbers in any base is done just like we would in base-10 — We add digits (for base-2 we add bits) to get sums and to generate carries. Since this course is on digital circuits, we care mostly about numbers represented in binary. Some examples are shown in Figure 10.6. **NOTE** that we can get a non-zero carry out

$$\begin{array}{cccccccccc} & \overset{0}{\curvearrowleft} & \overset{0}{\curvearrowleft} & \overset{1}{\curvearrowleft} & \overset{0}{\curvearrowleft} & \overset{0}{\curvearrowleft} & \overset{0}{\curvearrowleft} & \overset{0}{\curvearrowleft} & \overset{0}{\curvearrowleft} & \\ & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 170 \\ + & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & +48 \\ \hline 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 218 \end{array}$$

$$\begin{array}{cccccccccc} & \overset{1}{\curvearrowleft} & \overset{0}{\curvearrowleft} & \overset{1}{\curvearrowleft} & \overset{0}{\curvearrowleft} & \overset{0}{\curvearrowleft} & \overset{0}{\curvearrowleft} & \overset{0}{\curvearrowleft} & \overset{0}{\curvearrowleft} & \\ & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 170 \\ + & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & +176 \\ \hline 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 346 \end{array}$$

Figure 10.6: Examples of unsigned addition of binary numbers. One example demostrates *unsigned overflow*.

which is non-zero from the addition of the most significant bits. If we are limited to n digits, then a carry out of 1 signifies *unsigned overflow*. The result of the addition is too large to represent in the available number of bits.

10.3 Unsigned subtraction

Subtraction of unsigned numbers in any base is done just like we would in base-10 — We subtract digits (in base-2 bits) to get differences and use borrows as required. Since this course is on digital circuits, we care mostly about numbers represented in binary. Some examples are shown in Figure 10.7. **NOTE** that in the second

$$\begin{array}{cccccccccc} & \overset{0}{\curvearrowright} & \overset{2}{\curvearrowright} & \overset{2}{\curvearrowright} & \overset{2}{\curvearrowright} & \overset{0}{\curvearrowright} & \overset{0}{\curvearrowright} & \overset{0}{\curvearrowright} & \overset{0}{\curvearrowright} & \\ & \not{1}0 & \not{\phi}\not{2}1 & \not{1}0 & 0 & 1 & 0 & 1 & 0 & 170 \\ - & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & -48 \\ \hline & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 122 \end{array}$$

$$\begin{array}{cccccccccc} & \overset{2}{\curvearrowright} & \overset{2}{\curvearrowright} & \overset{2}{\curvearrowright} & \overset{2}{\curvearrowright} & \overset{0}{\curvearrowright} & \overset{0}{\curvearrowright} & \overset{0}{\curvearrowright} & \overset{0}{\curvearrowright} & \\ & \not{1}\not{\phi}2 & \not{\phi}\not{2}1 & \not{1}0 & 0 & 1 & 0 & 1 & 0 & 170 \\ - & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & -176 \\ \hline & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & -6 \end{array}$$

Figure 10.7: Examples of unsigned subtaction of binary numbers. One example demonstrates *unsigned underflow*.

case, we get garbage. For unsigned arithmetic, we cannot subtract a larger number from a smaller number otherwise we get *unsigned underflow*. This is indicated by the need to generate a borrow at the most-significant digit.

10.4 Signed number representations

We might also want to represent signed integers in different bases. One simple way to do this would be to use a *sign bit*; Given the repre-

sentation of a number in n digits, we could add another digit:

- If the extra digit is 0, the number is positive;
- If the extra digit is 1, the number is negative.

Figure 10.8 illustrates an example of representing negative numbers with a sign bit. The problem with using a sign bit is that we

- The value 9 with 8 bits is 000010012 .
- The value +9 would be represented as 0000010012 . Note the extra leading bit with value 0.
- The value âĹŠ9 would be represented as 1000010012 . Note the extra leading bit with value 1.

Figure 10.8: Representation of 9 and âĹŠ9 using 8 digits in base-2 using an extra sign bit.

introduce the concept of "+0" and "âĹŠ0". Also (not evident now) is that when we build circuits to perform numerical operations, the need for a sign bit complicates the circuitry unnecessarily.

10.4.1 *Radix complements*

The solution to our dilemma of representing negative numbers is to consider the idea of r-complements (or radix complements). The r-complement of a positive value V in base-r using n digits is defined as

$$\begin{array}{rl} r^n - V & \text{if } V \neq 0 \\ 0 & \text{if } V = 0 \end{array}$$

For example, consider representing the value 546700 in base-10 assuming only 7 digits are available — the result 0546700_{10}. The 10s complement of 0546700_{10} is $10^7 - 0546700 = 10000000 - 0546700 = 0453300_{10}$. As another example, consider representing the value 88 in binary assuming only 8 bits are available — the result is 01011000_2. The 2s complement of 01011000_2 is $2^8 - 01011000 = 100000000 - 01011000 = 10101000_2$.

In base-2, the 2s complement of a number can be found very quickly in other ways (i.e., other than performing a subtraction):

1. You can simply "flip the bits" and add 1.
2. You can copy bits right to left until the "first" 1 is encountered and then start flipping bits.

These alternative techniques in binary are useful.

10.5 Signed numbers and 2s complements

It turns out that representing negative numbers using radix complements is a really good idea rather than using a sign bit. In other words, if we want to represent a negative value in base-r, we find the rs complement of its absolute value and whatever we get represents the negative number.

As an example, the representation of -546700 in base-10 using 7 digits would be the 10s complement of $|\text{-}546700|$ which we previously found to be 0453300_{10}. Therefore, -546700 will be represented by 0453300_{10}. Another example would be the representation of -88 in binary — We previously found that the 2s complement if $|-88| = 88$ was 01011000_2. Therefore, -88 will be represented by 10101000_2.

Note that, in base-2 a positive value will always have *leading zeros* while a negative value will always have *leading ones*. If we are given a number in base-2 and we are told that it represents a signed integer *and* and the leading bits are 0, we can simply find the (+ve) value it represents. If we are given a number in base-2 and we are told that it represents a signed integer *and* and the leading bits are 1, we need to first find the 2s complement and then add a negative sign to figure out the value. An example of determining the value of a number in binary is shown in Figure 10.9.

The representation 11101010_2 is 8 bits and has leading 1s. Therefore, we know the number is negative. The 2s complement of 11101010_2 is 00010110_2 in 8 bits. The representation 00010110_2 equals a value of 22. This means that the absolute value of the number is 22 and, consequently, that 11101010_2 must be the representation of -22.

Figure 10.9: Example of determining the value of a negative number represented with rs complement.

We should consider the range of numbers we can represent in n bits in base-2. Figure 10.10 shows the numbers represented if we have 3 bits.

0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	-4
1	0	1	-3
1	1	0	-2
1	1	1	-1

Figure 10.10: Signed numbers which can be represented with 3 bits.

With n bits, we can represent the numbers $-2^{n-1} \cdots 2^{n-1} - 1$. Note we have only a single representation of the value 0.

10.6 Signed addition

It turns out that if negative numbers are represented using *r*s complement, then addition works! We perform the addition and ignore the carry out. Several examples of signed addition using 2s complement to represent negative numbers are shown in Figure 10.11.

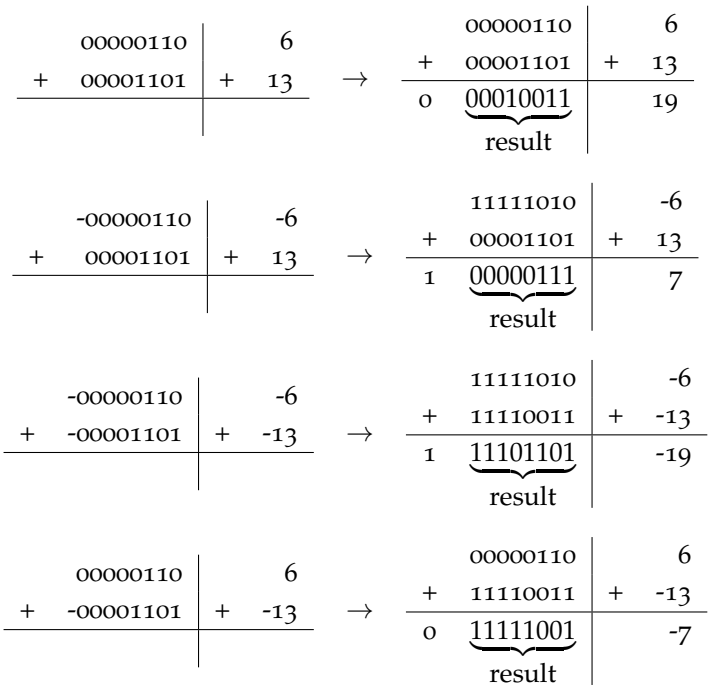


Figure 10.11: Examples of signed addition using complements.

10.7 Signed subtraction

If we represent negative numbers using complements, then subtraction becomes much easier. We avoid borrows and simply use the idea of "adding the opposite" or $(\pm M) - (\pm N) = (\pm M) + (\mp N)$. In other words, instead of subtracting the subtrahead N from the minuend M, we add the 2's complement of the subtrahend N to the minuend M. Since we know that addition works, so to will subtraction. An example is shown in Figure 10.12.

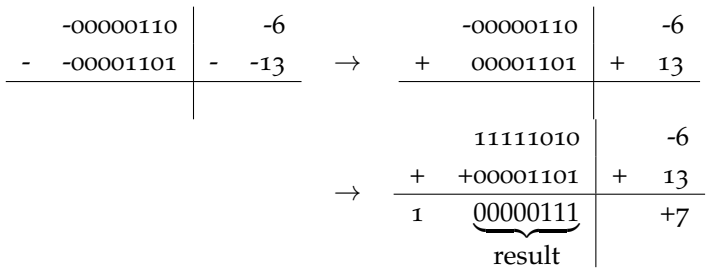


Figure 10.12: Example of signed subtraction using complements and addition.

10.8 *Overflow and signed addition*

We need to be concerned about *overflow* and *underflow* when performing signed addition. This can only happen if we are adding either:

1. Two positive numbers that exceed the upper limit of what we can represent; or
2. Two negative numbers that exceed the lower limit of what we can represent.

These two examples of signed overflow are shown in Figure 10.13. From Figure 10.13, we see the value 150 is too positive to represent

$$\begin{array}{cccccccccc} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 0} & {\scriptstyle 0} & {\scriptstyle 0} & {\scriptstyle 0} & {\scriptstyle 0} & {\scriptstyle 0} & \\ & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 70 \\ + & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & +80 \\ \hline & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 150 \end{array}$$

$$\begin{array}{cccccccccc} & {\scriptstyle 1} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 0} & {\scriptstyle 0} & {\scriptstyle 0} & {\scriptstyle 0} & \\ & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & -70 \\ + & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & -80 \\ \hline & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & -150 \end{array}$$

Figure 10.13: Examples of signed overflow.

in 8-bits. Notice that the result is *obviously wrong* since it indicates the result of the addition is a negative number even though we are adding two positive numbers. Similarly, from Figure 10.13 we see the value -150 is too negative to represent in 8-bits. Notice that the result is *obviously wrong* since it indicates the result of the addition is a postive number even though we are adding two negative numbers.

It turns out that an observation can be made: If the carry in to the most significant digit is *different* than the carry out from the most significant digit, then either overflow or underflow has occurred.

- This is true for signed arithmetic using 2s complement representation of negative numbers.
- If there is no overflow (e.g., consider adding a +ve number and a -ve number where overflow cannot happen), the carry in and out at the most significant bit will *always* be the same.

The conclusion is as follows. For unsigned addition, a carry out from the most significant bit indicates overflow. For signed addition, if the carry in and out at the most significant bit are different, this indicates overflow.

10.9 *Fixed point representations*

If we have values with fractional parts, one way to represent them is to consider using a **fixed point representation**. In this representation,

we allow a certain number of digits prior to the fixed point (the unsigned integer part of the value), and a certain number of digits after the fixed point (the fractional part of the value). The unsigned integer part of the number will be represented the same as with positional number notation. Therefore, we only need to figure out how to deal with the fractional part.

Consider a positive value V which has only a fractional part. In base-r using k digits after the fixed point (not called a decimal point if not base-10), V is represented by $D = (0.d_1 d_2 \cdots d_{-k})_r$. Therefore, any positive value V which has both an whole part (using positional notation) and a fractional part can be represented in base-r as

$$D = (\underbrace{d_{n-1}d_{n-2}\cdots d_1 d_0}_{\text{whole part}} \underbrace{.}_{\text{fixed point}} \underbrace{d_1 d_2 \cdots d_{-k}}_{\text{fractional part}})_r$$

This is called fixed point notation (fixed because we define the number of digits n in front of the fixed point and the number of digits k after the fixed point).

The value of V can be computed using the expression

$$\begin{aligned} V &= d_{n-1} \times r^{n-1} + d_{n-2} \times r^{n-2} + \cdots + d_1 \times r^1 + d_0 \times r^0 \\ &\quad + d_{-1} \times r^{-1} + d_{-2} \times r^{-2} + \cdots + d_{-k} \times r^{-k} \end{aligned}$$

Some examples are shown in Figures 10.14 and 10.15.

$$\begin{aligned} (213.78)_{10} &= 2 \times 10^2 + 1 \times 10^1 + 3 \times 10^0 + 7 \times 10^{-1} + 8 \times 10^{-2} \\ &= 200 + 10 + 3 + 7/10 + 8/100 \\ &= 200 + 10 + 3 + 0.70 + 0.08 \\ &= 213.78 \end{aligned}$$

Figure 10.14: Computing the value of a fixed point number represented in base-10 assuming 2 digits after the fixed point.

$$\begin{aligned} (1321.312)_4 &= 1 \times 4^3 + 3 \times 4^2 + 2 \times 4^1 + 1 \times 4^0 \\ &\quad + 3 \times 4^{-1} + 1 \times 4^{-2} + 2 \times 4^{-3} \\ &= 64 + 48 + 8 + 4 + 0.75 + 0.0625 + 0.031250 \\ &= 121.84375 \end{aligned}$$

Figure 10.15: Computing the value of a fixed point number represented in base-4 assuming 5 digits after the fixed point.

Given a fractional value, its representation in base r can be computed using successive multiplication by r. Consider

$$V = 0.d_{-1} \times r^{-1} + d_{-2} \times r^{-2} + \cdots d_{-k} \times r^{-k}$$

and then multiplication by r which gives

$$\frac{V}{r} = \underbrace{d_{-1}}_{\text{one of the digits}} . \underbrace{d_{-2} \times r^{-1} + \cdots + d_{-k} \times r^{-k+1}}_{\text{remaining fractional part}}$$

One of the digits pops out in front of the fixed point. We record this digit and repeat the multiplication with the remaining fractional part.

We stop once everything is 0... (as all subsequent digits will be zero). As example is given in Figure 10.16.

$$\begin{aligned} 0.625 \times 2 &= 1.250 \quad \rightarrow d_{-1} = 1 \\ 0.250 \times 2 &= 0.500 \quad \rightarrow d_{-2} = 0 \\ 0.500 \times 2 &= 1.000 \quad \rightarrow d_{-3} = 1 \end{aligned}$$

Figure 10.16: Representation of the value 0.625 in base-2. We stop when we do since all further bits would be 0. The representation of 0.625 in base-2 is 0.101000_2 assuming 6 bits after the fixed point.

To determine the representation for a value V that has both a whole part and a fractional part, we convert the whole part and the fractional part separately. The whole part is converted just like with positional number representation (i.e., through repeated division). The fractional part is converted as just discussed (i.e., through repeated multiplication). Converting fixed point representations from base-a to base-b is the same as before: First determine the value of the representation in base-a and then convert the value to base-b. Note that fast conversion between base-2, base-8 and base-16 can still be done by grouping bits.

For fixed point notation, it we need to represent a negative number, then we will use a *sign bit* and deal with the consequences of doing it this way.

11 Arithmetic circuits

Since we can represent numbers in base-2, we can design various types of circuits to perform arithmetic operations. We will consider a few different circuits.

11.1 Half adder circuit

A half adder (HA) circuit takes two bits x and y and adds the bits together to produce $x + y$. The circuit should produce two outputs — a sum bit s and a carry out bit c_{out}. The truth table for the half adder is shown in Figure 11.1. From the truth tables, we see that

x	y	s	c_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 11.1: Truth table for half adder (HA).

$s = x \oplus y$ and $c_{out} = xy$. The circuit which implements the half adder is shown in Figure 11.2.

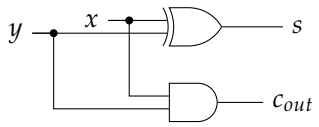


Figure 11.2: Circuit implementing a half adder.

11.2 Full adder circuit

If we want to add numbers which consist of multiple bits, then we need a circuit that "supports" a carry in c_{in} bit; the carry in bit comes from the addition of the previous bits. A full adder (FA) circuit takes inputs x, y and c_{in} and produces the outputs s and c_{out}. The truth table for a full adder is shown in Figure 11.3. From the truth tables, we see that $s = x \oplus y \oplus c_{in}$ and $c_{out} = xy + c_{in}(x \oplus y)$.

x	y	c_{in}	s	c_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Figure 11.3: Truth table for the full adder.

We can see that a full adder can be made using half adders as shown in Figure 11.4.

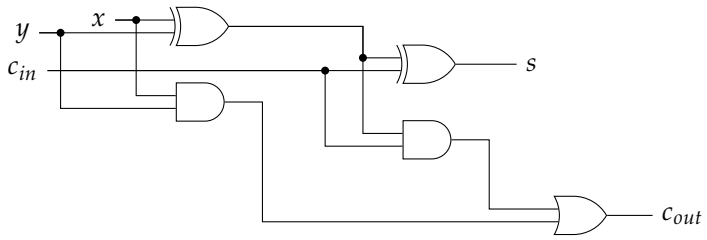


Figure 11.4: Circuit implementing a full adder.

11.3 Ripple adder

Say we want to add two n-bit numbers $x = (x_{n-1}, x_{n-2}, \cdots, x_1 x_0)$ and $y = (y_{n-1}, y_{n-2}, \cdots, y_1 y_0)$. We can accomplish this task using one half adder and $n-1$ full adders as shown in Figure 11.5. Can

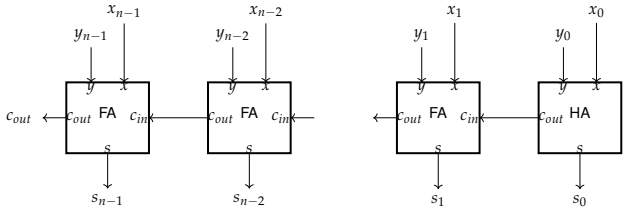


Figure 11.5: Implementation of an n-bit adder using $n-1$ full adders and 1 half adder.

use only full adders if we force the carry in of the least significant bit to be 0. This alternative is shown in Figure 11.6.

11.3.1 Performance of the ripple adder

Recall that, after a logic gate's input change, the gate output takes a bit of time to change due to delay inside of the gate. We can ask ourselves how long must we wait for the output of a ripple adder to

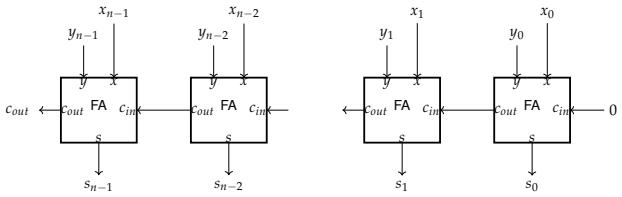


Figure 11.6: Implementation of an n-bit adder using only full adders.

be correct (i.e., outputs stop changing). To determine how long we must wait, we need to find the *longest combinational path* from any input to any output. For the n-bit ripple adder, let us assume that the inputs are all present at time 0. We can see that bit i cannot correctly compute its sum and carry out until bit $i-1$ has correctly computed *its carry out*. Further, note that the carry out from the i-bit depends on all the inputs from i down to 0. Therefore, the longest potential path through the ripple adder is from either a_0 or b_0 to the ultimate carry cout c_{out} from bit $n-1$.

Let us assume that we are using the ripple adder consisting of only full adders.

- For the 0-th bit, from either a_0 or b_0 to the carry out in the worst case signals must propagate through 1 **XOR**, 1 **AND** and 1 **OR**.
- For the i-th bit, from the carry in to the carry out, in the worst case signals must propagate through 1 **AND** and 1 **OR**.

Therefore if we measure delay in terms of the number of gates through which signals must propagate and cause potential changes in values, the worst possible delay of the circuit is

$$(1\mathbf{XOR} + 1\mathbf{AND} + 1\mathbf{OR}) + (n-1) * (1\mathbf{AND} + 1\mathbf{OR})$$

If the delay of all gates is the same, then the delay is a total of $2n+1$ gate delays. The performance of the ripple adder is extremely bad for large n due to the need for values to *ripple* down the *carry chain*.

11.4 Half subtractor circuit

A half subtractor (HS) takes two bits x and y and performs the subtraction $x - y$. The circuit produces two outputs, namely the difference d and a borrow b_{out}. The borrow output b_{out} is 1 when the subtraction would *require a borrow*; this is the case when $x = 0$ and $y = 1$. The truth table for a half subtractor is shown in Figure 11.7.

From the truth tables, we see that $s = x \oplus y$ and $b_{out} = \bar{x}y$. The circuit implementing the half subtractor is shown in Figure 11.8. Note the similarity of this circuit to the half adder circuit.

x	y	d	b_{out}	
0	0	0	0	
0	1	1	1	← we would require a borrow
1	0	1	0	
1	1	0	0	

Figure 11.7: Truth table for a half subtractor.

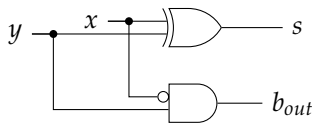


Figure 11.8: Circuit for half subtractor.

11.5 Full subtractor

If we want to subtract numbers (which consist of multiple bits), then we need a circuit that "supports" a borrow in b_{in}. The b_{out} is 1 when the circuit would *require a borrow*. When b_{in} is 1, it means the circuit would *supply a borrow*. The truth table for the full subtractor (FS) operation is shown in Figure 11.9. From the truth tables, we see

x	y	b_{in}	d	b_{out}
0	0	0	0	0
0	1	0	1	1
1	0	0	1	0
1	1	0	0	0
0	0	1	1	1
0	1	1	0	1
1	0	1	0	0
1	1	1	1	1

Figure 11.9: Truth table for the full subtractor.

that $s = x \oplus y \oplus b_{in}$ and $b_{out} = \bar{x}y + b_{in}\overline{(x \oplus y)}$. Note that there is similarity with a full adder circuit. The circuit implementing the full subtractor is shown in Figure 11.10.

11.6 Ripple subtractor

Say we want to subtract two n-bit numbers $x = (x_{n-1}, x_{n-2}, \cdots, x_1 x_0)$ and $y = (y_{n-1}, y_{n-2}, \cdots, y_1 y_0)$ and get $x - y$. We can accomplish this task using one half subtractor and $n - 1$ full subtractors as shown in Figure 11.11.

We can use only full subtractors if we force the borrow in of the least significant bit to be 0 and this circuit is shown in Figure 11.12.

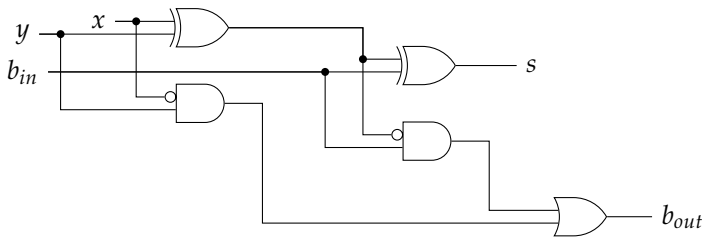


Figure 11.10: Circuit for the full subtractor.

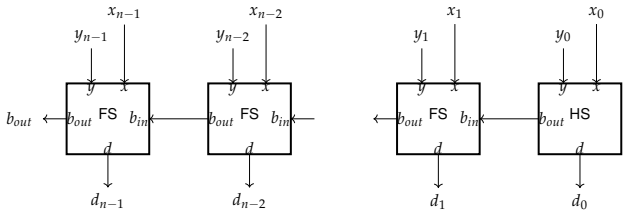


Figure 11.11: Implementation of an n bit subtraction circuit using $n-1$ full subtractors and 1 half subtractor.

11.6.1 *Performance of the ripple subtractor*

Similar to the ripple adder. Need to follow down the *borrow chain* from the least significant inputs x_0 or y_0.

11.7 *Addtion and subtraction together*

Subtraction is equivalent to adding the 2s complement of the subtrahend to the minuend. Therefore we can perform subtraction with an adder circuit rather than having to worry about the creation of a subtractor. Further, taking the 2s complement of a binary number is equivalent to flipping the bits and adding 1. We can therefore implement subtraction using the circuit shown in Figure 11.13.

Since, from Figure 11.13, subtraction can be performed using an adder, we can combine the operations of addition and subtraction into a single circuit. We can introduce a *control line* called $\overline{add}/sub$. When $\overline{add}/sub = 0$, the circuit performs the addition operation. When $\overline{add}/sub = 1$, the circuit performs subtraction. Such a circuit is shown in Figure 11.14. Note the creative use of **XOR** to perform inversion when doing subtraction and the connection of the control

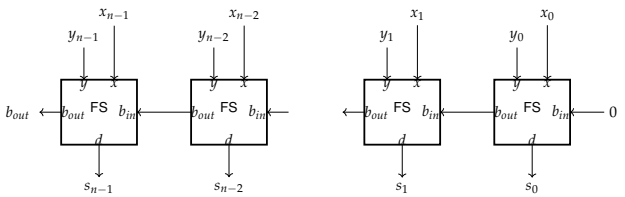


Figure 11.12: Implementation of an n bit subtraction circuit using only full subtractors.

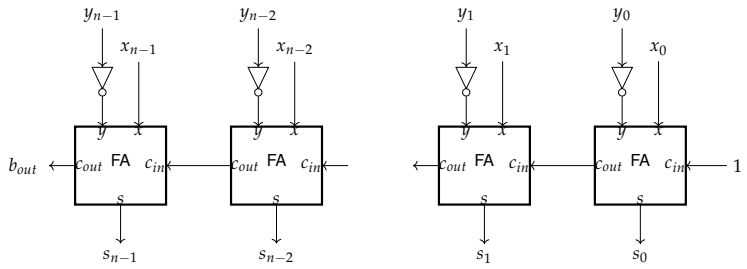


Figure 11.13: Subtraction performed using an adder circuit by taking the 2s complement of the subtrahend and adding it to the minuend.

line to the carry in of the adder to faciliate the 2s complement of the subtrahend when performing subtraction.

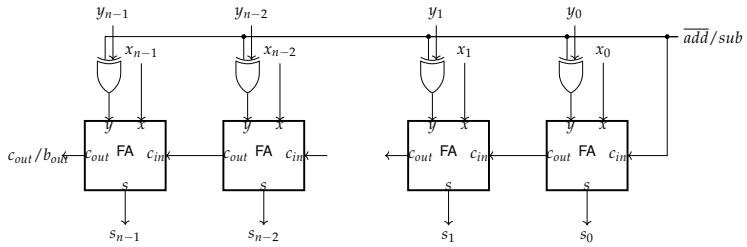


Figure 11.14: Single circuit to perform both addition and subtraction using a control line.

11.8 *Carry lookahead adders (CLA)*

We can make adders faster using *carry lookahead* circuitry. Let $p = x \oplus y$ be the *propagate* signal and let $g = xy$ be the *generate* signal. Then, we can redraw the full adder circuit to identify the p and g signals as shown in Figure 11.15. We can then write the carry out

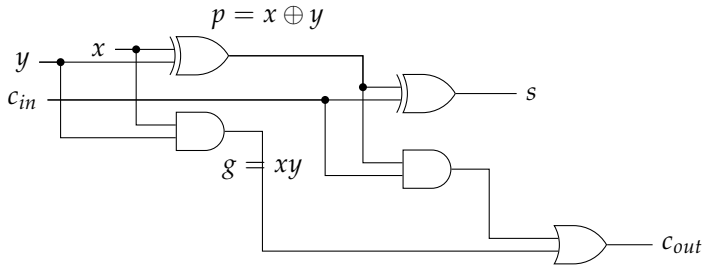


Figure 11.15: Indentification of the *propagate* and *generate* signals inside of full adder circuit.

in terms of p and g as $c = g + c_{in}p$.

The ripple adder is slow because high bits must "wait" for carries to be generated. Note that the carries are produced as signals propagate through many "levels of gates". To speed up computation of the carries, we can first generate all the p and g for all bits (p and g only depend on x and y). Then, we can compute all carries at the *same time*

by "collapsing" the multiple levels of logic; e.g.,

$$\begin{aligned} c_1 &= g_0 + p_0 c_0 \\ c_2 &= g_1 + p_1 c_1 \\ &= g_1 + p_1(g_0 + p_0 c_0) \\ &= g_1 + p_1 g_0 + p_1 p_0 c_0 \\ c_3 &= g_2 + p_2 c_2 \\ &= g_1 + p_2(g_1 + p_1 g_0 + p_1 p_0 c_0) \\ &= g_1 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0 \\ \cdots &= \cdots \end{aligned}$$

Note that in removing the parentheses "(" and ")" we are, in effect, removing levels of gates from the equations and collapsing the carry signals back to 2-level SOPs!

11.8.1 *Performance of carry lookahead adders (CLA)*

It is sufficient to consider how fast the carry signals can be generated. Note that all p and g are generated from x and y after 1 gate delay. Therefore, **all** carry signals are generated via SOP expressions 2 gate delays after the p and g. Consequently, all carries are available at the same time after 3 gate delays. Regardless of the number of bits n that we are adding, the delay of a carry lookahead adder is fixed. Of course, this is impractical in reality. **Why**? It's impractical because we see that the carry lookahead circuitry requires larger and larger gates (i.e., more inputs) as we more to higher order carry signals. Totally impractical.

11.8.2 *Compromise between ripple adder and CLA*

We can compromise to build a reasonably fast adder. We can use a combination of ideas from carry lookahead adders and ripple adders — we build a reasonably sized carry lookahead adder and then connect multiple carry lookahead adders in a chain just like a ripple adder. An example of is shown in Figure 11.16.

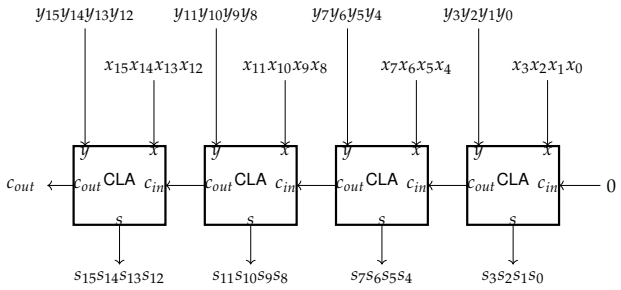


Figure 11.16: Example of a 16-bit adder built using multiple 4-bit carry lookahead adders connected together in a chain.

.

11.9 *Array multipliers*

Multiplication of binary numbers works just like in decimal. Example multiplication of 2, 3-bit numbers $x = (x_2x_1x_0)$ and $y = (y_2y_1y_0)$ to get $x \times y = p$ where $p = (p_5p_4p_3p_2p_1p_0)$. Note that multiplication requires twice the number of bits for the result. The details of the a multiplication is shown in Figure 11.17. A numerical example is

$$
\begin{array}{rccccc}
 & & x_2 & x_1 & x_0 & \leftarrow \text{multiplicand} \\
 & \times & y_2 & y_1 & y_0 & \leftarrow \text{multiplier} \\
\hline
 & & y_0x_2 & y_0x_1 & y_0x_0 & \leftarrow \text{partial product} \\
+ & y_1x_2 & y_1x_1 & y_1x_0 & & \\
\hline
 & pp_4^1 & pp_3^1 & pp_2^1 & pp_1^1 & pp_0^1 \leftarrow \text{partial product} \\
+ & y_2x_2 & y_2x_1 & y_2x_0 & & \\
\hline
pp_5^2 & pp_4^2 & pp_3^2 & pp_2^2 & pp_1^2 & pp_0^2 \leftarrow \text{partial product}
\end{array}
$$

Figure 11.17: Multiplication of two, 3-bit numbers to yield a 6-bit product.

given in Figure 11.18. Note that multiplication of digits in binary

$$
\begin{array}{rccccc}
 & & & 1 & 1 & 1 \\
 & & \times & 1 & 1 & 0 \\
\hline
 & & & 0 & 0 & 0 \\
 & + & 1 & 1 & 1 & \\
\hline
 & 0 & 1 & 1 & 1 & 0 \\
+ & 1 & 1 & 1 & & \\
\hline
1 & 0 & 1 & 0 & 1 & 0
\end{array}
$$

Figure 11.18: Numerical example of unsigned binary multiplication of 7×6 (or 111×110). The result is Result is 101010 or 42.

is simply the **AND** operator.

Since bit multiplication is just **AND** we can make an array multiplier from **AND** gates and 1-bit full adders. We first build the circuit in Figure 11.19. Using the circuit block from Figure 11.19, we can can build an *array multipler*. An example is shown in Figure 11.20.

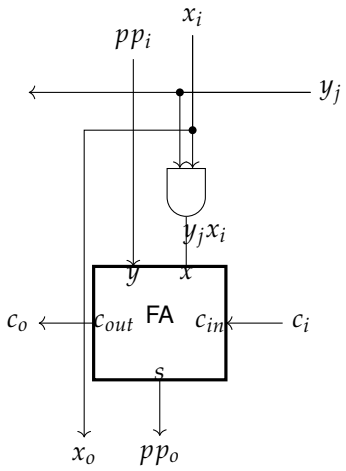


Figure 11.19: Building block circuit required to build a binary multiplier.

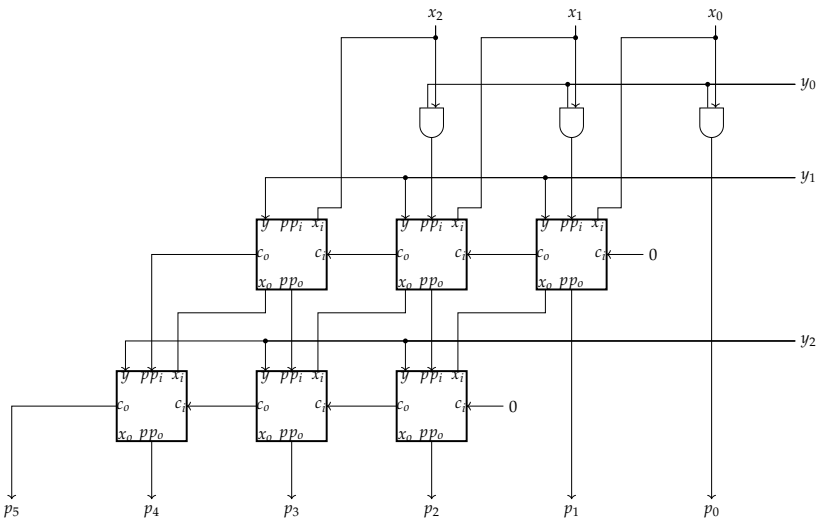


Figure 11.20: Multiplier circuit for 3-bit unsigned numbers.

12 Common circuit blocks

There are a number of operations in additon to arithmetic circuits that occur frequently. Therefore, we should investigate some additional combinational circuit blocks and see what we can do with them. circuits.

12.1 Comparators

It's common to want to compare two n-bit unsigned numbers $A = a_{n-1}a_{n-2}\cdots a_1a_0$ and $B = b_{n-1}b_{n-2}\cdots b_1a_0$. to determine if $A > B$, $A < B$ or $A = B$. Such a circuit would have 3 outputs $f_{A>B}$, $f_{A=B}$ and $f_{A<B}$ such that:

1. $f_{A>B} = 1$ when the magnitude of A is larger than B,
2. $f_{A=B} = 1$ when the magnitude of A is equal to B, and
3. $f_{A<B} = 1$ when the magnitude of A is smaller than B.

We can consider the three different situations to obtain a final circuit.

12.1.1 Equality — $A = B$

Equality is straightforward — we compare each pair of digits of A and B and if a_i and b_i for all bits, then the numbers are equal. Equality of individual pairs of bits a_i and b_i is done via $e_i = a_i'b_i' + a_ib_i = \overline{a_i \oplus b_i}$ which is just the **NXOR** operation shown in Figure 12.1.

Then, once equality is checked on all bits, we can determine

a_i	b_i	$a_i = b_i$?
0	0	1
0	1	0
1	0	0
1	1	1

Figure 12.1: Testing equality of two bits in the **NXOR** operator.

equality via $f_{A=B} = e_{n-1}e_{n-2}\cdots e_1e_0$. The equality circuit is shown in Figure 12.2 for the case when A and B are 2-bit numbers.

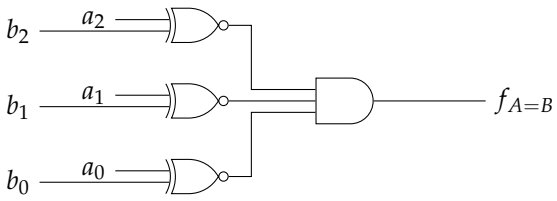


Figure 12.2: Circuit to compare two 3-bit numbers for equality.

12.1.2 *Greater than — $A > B$*

To determine if $A > B$ requires consideration of the *algorithm* used to compare two numbers. First, consider comparing two bits a_i and b_i; a_i is larger than bit b_i when $a_i b_i'$. This is shown in figure 12.3.

a_i	b_i	$a_i > b_i$?
0	0	0
0	1	0
1	0	1
1	1	0

Figure 12.3: Testing $a_i > b_i$.

To compare numbers A and B, we would start at the most significant bits a_{n-1} and b_{n-1} and work our way to the leat significant bits a_0 and b_0. At bits a_i and b_i, we can declare $A > B$ if all bit pairs are equal for larger bit pairs and $a_i > b_i$. All of these checks can be done at the same time. This results in the expression

$$\begin{aligned} f_{A>B} \quad = \quad & a_{n-1}b'_{n-1} + e_{n-1}a_{n-2}b'_{n-2} + \\ & \cdots + e_{n-1}e_{n-2}\cdots e_2 a_1 b'_1 + e_{n-1}e_{n-2}\cdots e_2 e_1 a_0 b'_0. \end{aligned}$$

The circuit for comparing two 3-bit numbers A and B is shown in Figure 12.4.

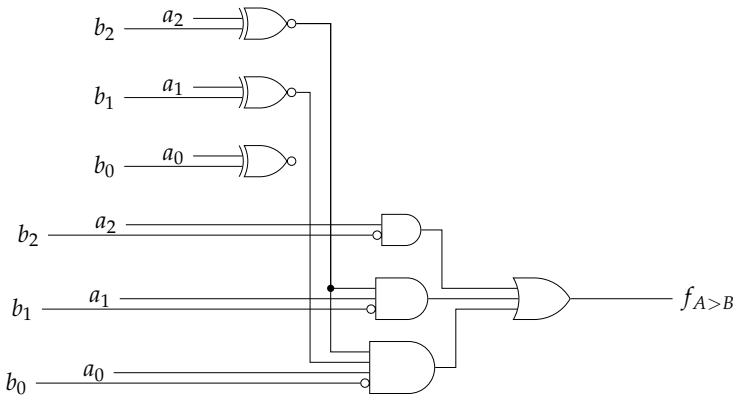


Figure 12.4: Circuit for comparing two, 3-bit numbers for $A > B$.

12.1.3 *Less than — $A < B$*

To determine if $A < B$ we have something very similar to $A > B$. If we consider only a pair of bits a_i and b_i, we find that $a_i < b_i$ when $a_i' b_i$. This is shown in Figure **??**. Using the same idea as for finding $A > B$, we can obtain the expression for $f_{A<B}$ from

a_i	b_i	$a_i < b_i$?
0	0	0
0	1	1
1	0	0
1	1	0

Figure 12.5: Testing $a_i < b_i$.

$$f_{A<B} = a'_{n-1}b_{n-1} + e_{n-1}a'_{n-2}b_{n-2} + \cdots + e_{n-1}e_{n-2}\cdots e_2 a'_1 b_1 + e_{n-1}e_{n-2}\cdots e_2 e_1 a'_0 b_0.$$

The circuit to compare two, 3-bit numbers for $A < B$ is shown in Figure 12.6. Compared to the circuit for $A > B$ all we have done is "reverse the roles" of A and B by switching the location of some inverters. If we already have a circuit for $A = B$ and $A > B$, then we can compute $A < B$ with less circuitry. We know $A < B$ if A is not equal to B and A is not greater than B. Logically, this can be expressed as $f_{A<B} = \overline{f_{A=B} + f_{A>B}}$ and the circuit is a simple **NOR** gate as shown in Figure 12.7.

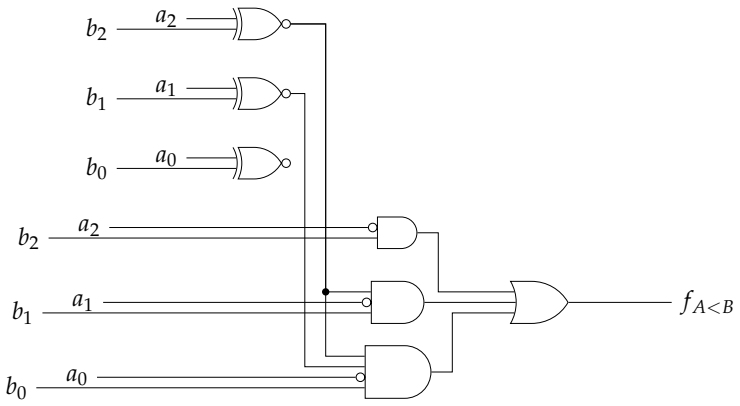


Figure 12.6: Circuit for comparing two, 3-bit numbers for $A < B$.

12.1.4 *Hierarchical or iterative comparator design*

Circuits for $A = B$, $A > B$ or $A < B$ as previously designed only require a few levels of logic. However, as the number of bits n in A and B increases, we require gates with a larger number of inputs. Gates with a significant number of inputs are prohibitive. For example, to

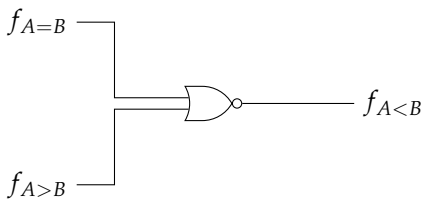


Figure 12.7: Alternative circuit to compute $A < B$ given that the circuits for $A = B$ and $A > B$ already exist.

.

test $A = B$ for n-bits requires an n-input **AND** gate and for large n, such an **AND** gate might not be practical.

We can design an iterative comparator by designing a single smaller circuit and then copying this smaller circuit n times. This idea is shown in Figure 12.8. Such a circuit is called an *iterative comparator*.

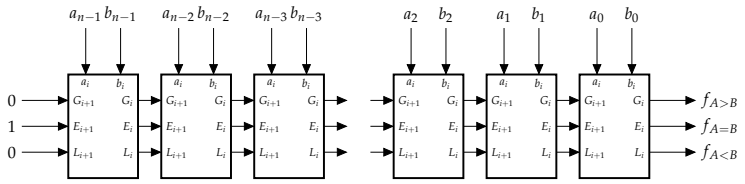


Figure 12.8: Block diagram for an n-bit iterative comparator.

In Figure 12.8, each block i is identical internally so our n-bit comparator is made from n identical copies of the same sub-circuit. Each sub circuit has 5 inputs (a_i, b_i, E_{i+1}, G_{i+1} and L_{i+1}) and produces 3 outputs (E_i, G_i and L_i). E_i means that $A = B$ if only the i-th to $(n-1)$-th bits are considered. G_i means that $A > B$ if only the i-th to $(n-1)$-th bits are considered. Finally, L_i means that $A < B$ if only the i-th to $(n-1)$-th bits are considered. The purpose if the i-th sub-circuit is therefore to decide how A compares to B based on the comparison of the higher order bits and the current bits a_i and b_i. With consideration, the logic for E_i, G_i and L_i can be found to be given as $G_i = G_{i+1} + E_{i+1}a_i b_i'$, $E_i = E_{i+1}\overline{a_i \oplus b_i}$, and $L_i = L_{i+1} + E_{i+1}a_i' b_i$. In other words (in the case of G_i), when comparing bits a_i and b_i we know that $A > B$ if: 1) higher order bits have indicated that $A > B$; *or* 2) higher order bits are all equal and the current bits a_i and b_i allow use to deduce that $A > B$. The replicated sub-circuit that we require is illustrated in Figure 12.8. Note the inputs to the sub-circuit comparing the MSB; since we have not "compared" any bits higher than the MSB (since there aren't any), then inputs to this block indicate that the higher bits are all equal. This is equivalent to assuming the numbers are been prefixed with an infinite number of zeros (which does not change the magnitude of unsigned numbers!). Finally, the

performance of this circuit will be less since there are multiple levels of logic involved. However, the largest gate required regardless of the number of bits to be compared is a 3-input `AND` gate.

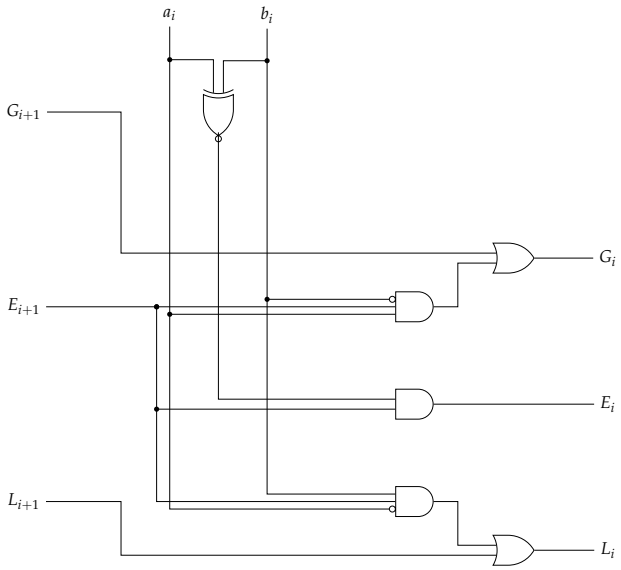


Figure 12.9: Sub-circuit required for building an iterative comparator.

12.2 *Multiplexers*

Combinational circuit that has data inputs, select lines and a single output. One input is passed through to the output based on the control lines. For n data inputs, we need $\lceil \log(n) \rceil$ control lines.

12.2.1 2-*to*-1 *multiplexer*

Two data inputs x_0 and x_1. One select line s. One output f. Truth table for a 2-to-1 multiplexer is shown in Figure 12.10. Imple-

x_0	x_1	s	f
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1
0	0	1	0
0	1	1	1
1	0	1	0
1	1	1	1

$\rightarrow$

s	f
0	x_0
1	x_1

Figure 12.10: Table explaining the operations of a 2-to-1 multiplexer.

ments (basically) the "if-else" function in hardware. We can write an

equation easily:

$$f = \bar{s}x_0 + sx_1$$

Multiplexers have their own symbol which is shown in Figure 12.11.

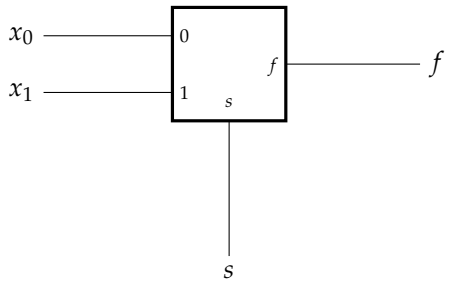


Figure 12.11: Symbol for a 2-to-1 multiplexer. **NOTE: THIS SYMBOL IS NOT QUITE RIGHT... I AM STILL TRYING TO FIGURE OUT HOW TO DRAW IT USING THE PROGRAM THAT I USE...**

12.2.2 4-*to*-1 *multiplexer*

A 4-to-1 multiplexer has 4 inputs, 2 select lines and 1 output. Its truth table is shown in Figure 12.12. The logic expression describing its

s_0	s_1	f
0	0	x_0
0	1	x_1
1	0	x_2
1	1	x_3

Figure 12.12: Table explaining the operation of a 4-to-1 multiplexer.

behaviour is given by

$$f = s_0' s_1' x_0 + s_0' s_1 x_1 + s_0 s_1' x_2 + s_0 s_1 x_3$$

The symbol for a 4-to-1 multiplexer is shown in Figure 12.13. We

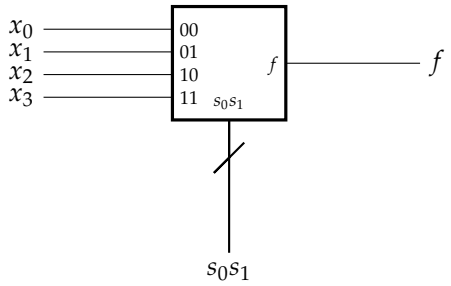


Figure 12.13: Symbol for a 4-to-1 multiplexer. textbfNOTE: THIS SYMBOL IS NOT QUITE RIGHT... I AM STILL TRYING TO FIGURE OUT HOW TO DRAW IT USING THE PROGRAM THAT I USE...

have similar symbols for larger multiplexers — we just add additional inputs and more select lines as required.

12.2.3 *Multiplexer trees*

We can build larger multiplexers from smaller multiplexers. An example of a 4-to-1 multiplexer built from 2-to-1 multiplexers is shown in Figure 12.14. We can check the mathematics for the

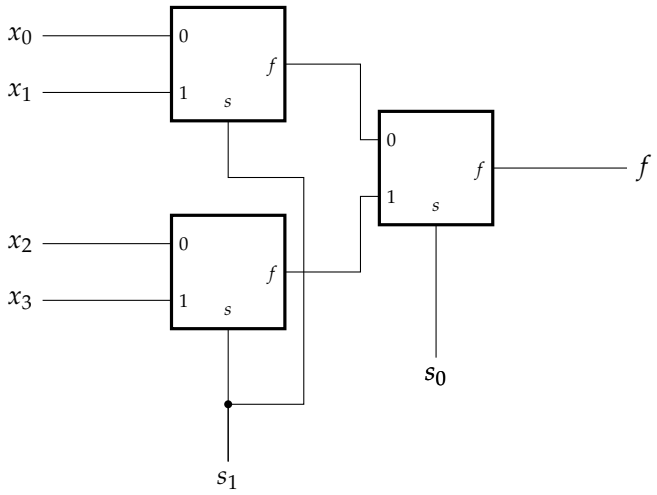


Figure 12.14: Example of a multiplexer tree used to build a 4-to-2 multiplexer from several 2-to-1 multiplexers.

circuit in Figure 12.14 and compare it to the equation for a 4-to-2 multiplexer to show that it works as follows:

$$\begin{aligned} f &= s_0'(s_1'x_0 + s_1x_1) + s_0(s_1'x_2 + s_1x_3) \\ &= s_0's_1'x_0 + s_0's_1x_1 + s_0s_1'x_2 + s_0s_1x_3 \end{aligned}$$

12.2.4 *Function implementation with multiplexers*

Rather than using logic gates, we can actually implement logic functions using only multiplexers. There are severable ways to do this depending on the multiplexers you have available. Here, we will consider two cases.

You can implement an n-input function with a single multiplexer as long as the multiplexer has $n-1$ select lines. Consider the 3-input function given by

$$f = x_0'x_1 + x_0x_1'x_2 + x_0x_1x_2'.$$

For this 3 input function, we require a multiplexer with 2 select lines; i.e., a 4-to-1 multiplexer. This sort of implementation is best done with a truth table; the truth table for f is shown in Figure 12.15. We use the $n-1$ leftmost inputs (leftmost columns of the truth table) to connect to the select lines and divide the truth table into a *number of pieces*; the number of pieces equals the number of inputs on the multiplexer. For each piece of the truth table, we compare the value

x_0	x_1	x_2	f
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	1	0	1
1	1	1	0

Figure 12.15: Implementation of a 3 input function using a multiplexer with 2 select lines. The first two variables x_0 and x_1 partition the truth table into four separate pieces.

of the right most input to the value of f and connect the correct value to the appropriate input of the multiplexer. The value to connect will always be one of four choices: 0, 1, x_{n-1} or $\overline{x_{n-1}}$.

Following this strategy, we get the circuit shown in Figure 12.16 as the implementation of f as shown in Figure 12.15.

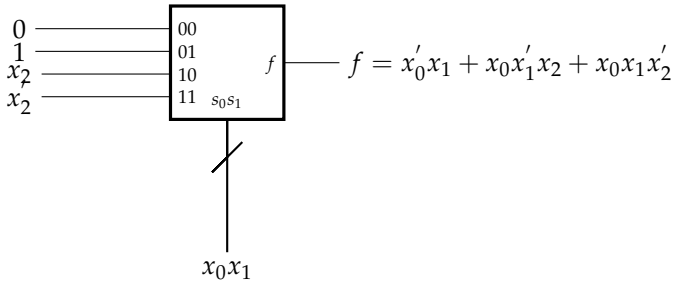


Figure 12.16: Circuit implementing the 3 input function from Figure 12.15 using a multiplexer with 2 control lines.

If we don't have a large enough multiplexer, we can work with smaller multiplexers. Consider having only 2-to-1 multiplexers. We can *decompose* algebraically any logic function such that it can be implemented with only 2-to-1 multiplexers. Say we have a function f with n inputs. Pick any input variable — say the i-variable x_i. Then, we can always write f as

$$\begin{aligned} f \quad = \quad & x_i' f(x_0, \cdots, x_{i-1}, 0, x_{i+1}, \cdots, x_{n-1}) \\ & + x_i f(x_0, \cdots, x_{i-1}, 1, x_{i+1}, \cdots, x_{n-1}) \end{aligned}$$

In other words, we can collect everything involving x_i' together and everything involving x_i together and then factor out x_i' and x_i, respectively. This is known as *cofactoring* Note the "structure" of f after cofactoring... It has the structure of a 2-to-1 multiplexer in which x_i is connected to the select line.

We can consider how to implement a logic function f using cofactoring. Consider the function $f = x_0'x_1 + x_0x_1'x_2 + x_0x_1x_2'$. Using cofactoring, we can write

$$
\begin{aligned}
f &= x_0'(x_1) + x_0(x_1'x_2 + x_1x_2') \\
&= x_0'(x_1) + x_0(x_1'(x_2) + x_1(x_2')) \qquad \leftarrow \\
&= x_0'(x_1'(0) + x_1(1)) \\
&\quad + x_0(x_1'(x_2'(0) + x_2(1)) + x_1(x_2'(1) + x_2(0))) \qquad \leftarrow
\end{aligned}
$$

which shows several different steps of cofactoring depending on how "far we wish to go". By "how far we wish to go", we mean to say whether or not we want to connect function inputs to only select lines or to both select lines and data inputs on the multiplexers. For example, in the first step of the above equation, we see that we have a situation where the mathematics corresponds to 2-to-1 multiplexers, but would require connecting inputs to the both the multiplexer control lines and data inputs — this solution is shown in Figure 12.17.

The last line of the previous mathematics, however, consists of additional cofactoring which would allow us to connect only the constant values 0 and 1 to the data inputs of the multiplexers and the input variables are restricted to being connected only to the multiplexer select lines — this solution is shown in Figure 12.18.

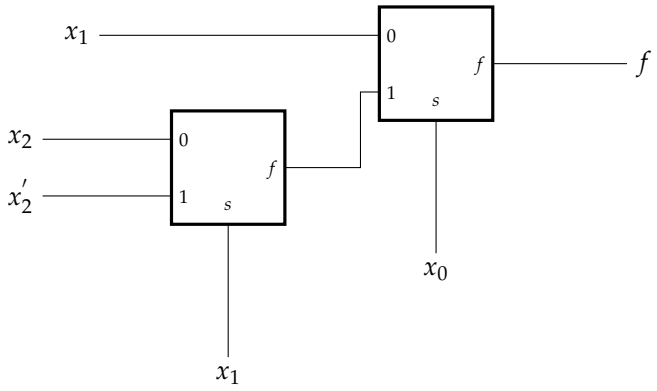


Figure 12.17: Implementation of $f = x_0'(x_1) + x_0(x_1'x_2 + x_1x_2')$ using 2-to-1 multiplexers. Here, we connect input variables to both the multiplexer select lines and the multiplexer data inputs.

Note that the order in which you apply cofactoring matters. If you cofactor using a different variable ordering you can get a different circuit which may be more or less complex in terms of the number of multiplexers required. For example, in our example, we we cofactored with respect to x_0 first, then x_1 and finally x_2. We could have cofactored using a different selection of input variables at each step.

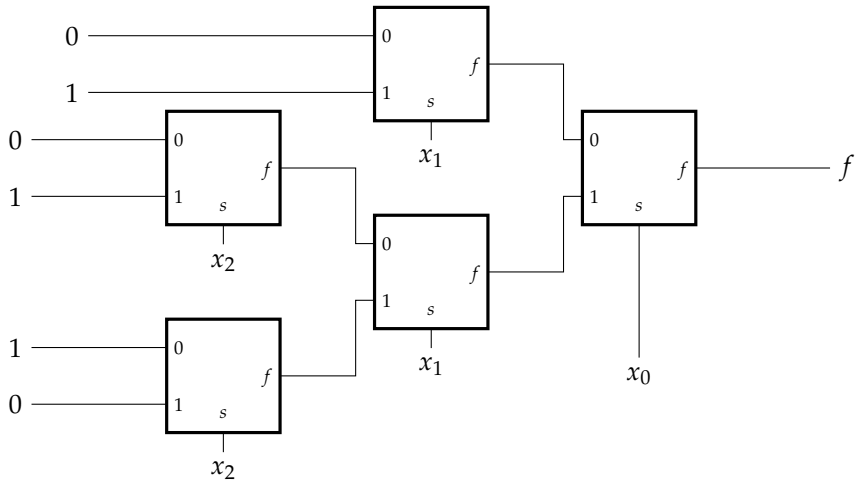


Figure 12.18: Implementation of $f = x_0^{'}(x_1) + x_0(x_1^{'}x_2 + x_1x_2^{'})$ using 2-to-1 multiplexers. Here, we connect input variables to only the multiplexer select lines. The data inputs for the multiplexers have only constant 0 or 1 connected.

13 Encoders and decoders

13.1 Decoders

A decoder accepts n inputs and has 2^n outputs. The purpose of a decoder is to recognize (or "decode") the binary input pattern and set the corresponding output. Often, a decoder will have an *enable* signal. When the enable signal is low, all the outputs are 0. When the enable signal is high, the decoder performs its task. An table explaning the operation of a 2-to-4 decoder is shown in Figure 13.1 and its corresponding circuit is shown in Figure 13.2.

x_0	x_1	en	d_0	d_1	d_2	d_3
X	X	0	0	0	0	0
0	0	1	1	0	0	0
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	1

Figure 13.1: Table describing the operation of a 2-to-4 decoder.

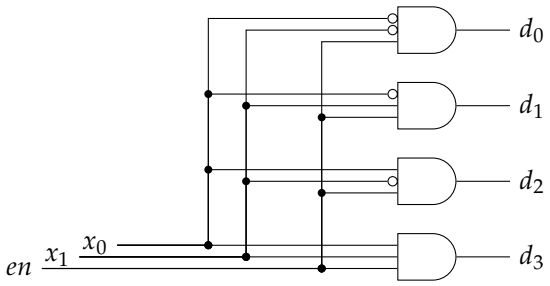


Figure 13.2: Circuit implementing a 2-to-4 decoder.

13.1.1 Decoder trees

We can make larger decoders from smaller decoders. For example, we can consider implementing a 4-to-16 decoder built from 2-to-4 decoders as follows:

- Of the 4 inputs x_0, x_1, x_2 and x_3, the two most significant inputs

x_0 and x_1 are used with one decoder to *enable* the appropriate decoder in the second stage.

- The two least significant inputs x_2 and x_3 are used to generate the correct output.

The circuit for a 4-to-16 decoder built from smaller decoders is shown in Figure 13.3.

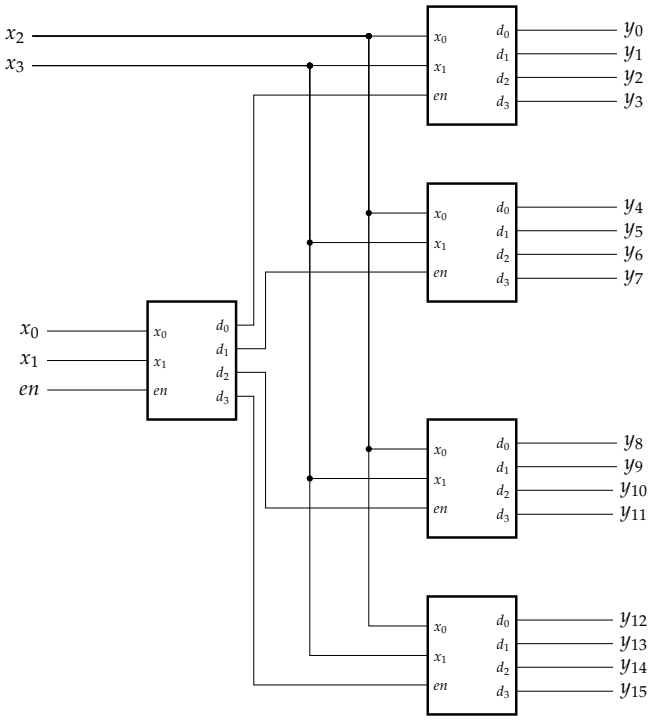


Figure 13.3: Example of a decoder tree. This shows a 4-to-16 decoder built from several 2-to-4 decoders.

13.1.2 *Function implementation with decoders*

Consider that a decoder is basically decoding the input pattern corresponding to a minterm. Consequently, if we have a decoder with n inputs and an **OR** gate, we can implement any n input function. Figure 13.4 shows the implementation of a small 2-input function $f = f(a, b) = \sum(0, 2, 3)$ using a decoder.

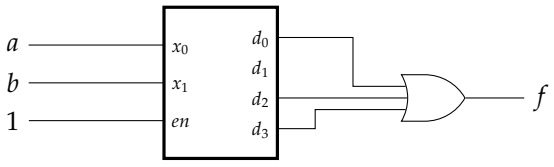


Figure 13.4: Example of implementing a 2 input function using a 2-to-4 decoder. Implementation of an n input function requires a n-to-2^n decoder. Note the *enable* is fixed at 1.

13.2 *Encoders and priority encoders*

An *encoder* performs the inverse operation of a decoder — the encoder has 2^n inputs and generates n outputs. The output is a binary encoding of the input line which is set. The table describing the operation of an an 8-to-3 encoder is given in Figure 13.5. It is easy

d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	x_0	x_1	x_2
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

Figure 13.5: Table explaining the operation of a 8-to-3 encoder.

to see that the outputs are easily computed using **OR** gates. For the encoder in Figure 13.5, $x_0 = d_4 + d_5 + d_6 + d_7$, $x_1 = d_2 + d_3 + d_6 + d_7$ and $x_2 = d_1 + d_3 + d_5 + d_7$.

The simple encoder is a silly circuit because it does not account for the following situations:

1. What if no inputs is set?

2. What if multiple inputs are set?

The solution to the first problem is to introduce another output, *valid*, which is set when any input is set, otherwise 0. This tells us when we have encoded an input value vs. having no input value. The solution to the second problem is to have *priority* and to design a so-called *priority encoder*. The output the circuit produces should be the encoding of the "highest indexed" (highest priority) input. The operation of a 4-to-2 priority encoder is shown in Figure 13.6. It is straightfor-

d_0	d_1	d_2	d_3	x_0	x_1	*valid*	
0	0	0	0	0	0	0	← output not valid
1	0	0	0	0	0	1	
X	1	0	0	0	1	1	
X	X	1	0	1	0	1	
X	X	X	1	1	1	1	

Figure 13.6: Table explaining the operation of a 4-to-2 priority encoder.

ward to write down unoptimized equations for a priority encoder outputs. For the 4-to-2 priority encoder in Figure 13.6 we get the circuit in Figure 13.7.

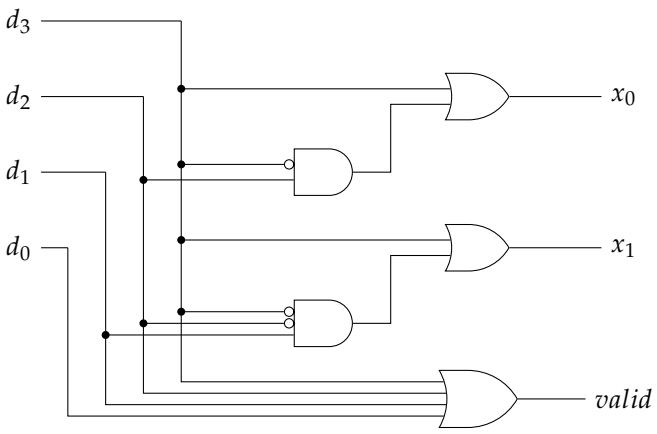


Figure 13.7: Circuit implementing a 4-to-2 priority encoder.

13.3 *Hierarchical priority encoders**

Large priority encoders can require gates with a large number of inputs. For example, a priority encoder with n inputs requires an n input **OR** gate to determine its *valid* signal. It is therefore of interest to consider whether or not we can construct larger priority encoders from smaller ones. We can if we use some additional multiplexers. Figure 13.8 show the table explaining the operation of a 2-to-1 priority encoder (the smallest The logic equations for this small

d_0	d_1	x_0	$valid$
0	0	0	0
1	0	0	1
X	1	1	1

Figure 13.8: Table describing the operation of a 2-to-1 priority encoder.

priority encoder are simply $valid = d_0 + d_1$ and $x_0 = d_1$.

We can now proceed to construct a 4-to-2 priority encoder using 2-to-1 priority encoders and multiplexers. The largest gate required in this implementation equals the largest gate in any of the sub-circuits (which, in this case, is 2 inputs). The block diagram for the 4-to-2 priority encoder using circuit sub-blocks is shown in Figure 13.9. All gates inside of these blocks are 2-input gates.

We can go even further — we can design an 8-to-3 priority encoder from 4-to-2 priority encoders (plus some multiplexers). This also means we can design an 8-to-3 priority encoder using only 2-to-1 priority encoders. There is a pattern which emerges and it should be clear that we can continue the procedure to build any sized 2^n-to-n priority encoder from smaller encoders (and multiplexers). The block diagram of a 8-to-3 priority encoder using only 2-to-1 priority encoders and some multiplexers is shown in Figure 13.10.

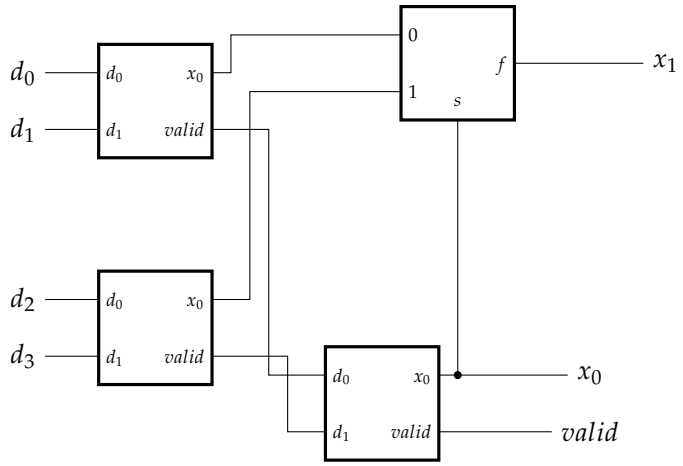


Figure 13.9: Implementation of a 4-to-2 priority encoder using smaller 2-to-1 priority encoders and some extra multiplexers.

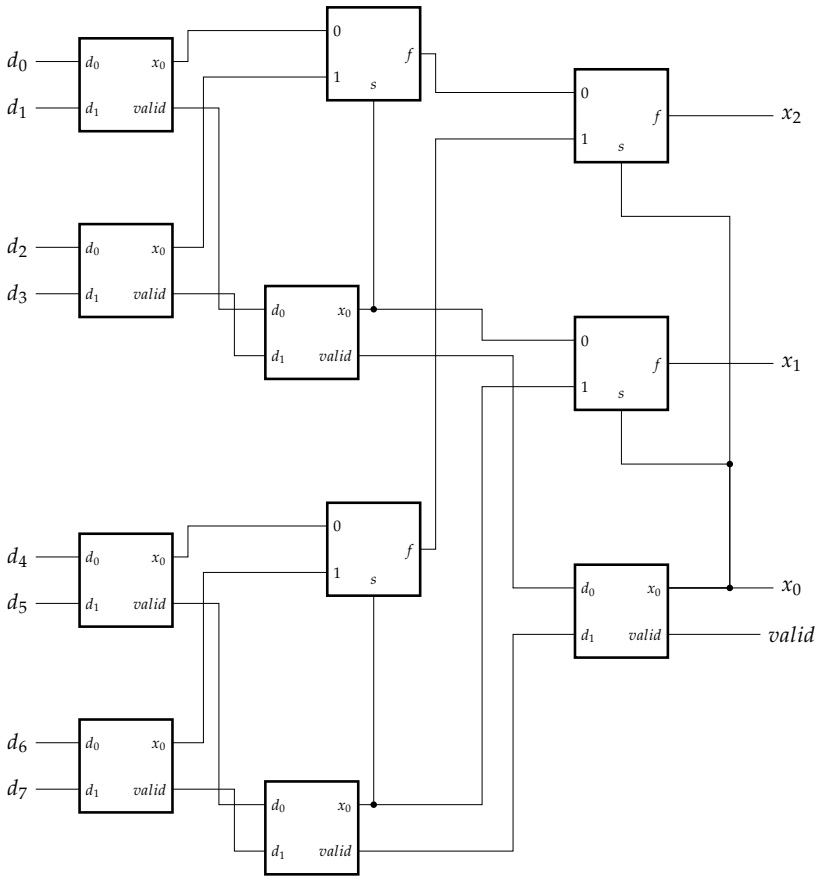


Figure 13.10: Implementation of a 8-to-3 priority encoder using smaller 2-to-1 priority encoders and some extra multiplexers.

13.4 *Demultiplexers*

A demultiplexer does the opposite operation to a multiplexer; it "switches" a single data input line onto one of several output lines depending on the setting of some control lines. A demultiplexer can be implemented identically to a decoder by changing the interpretation of the signals.

1. The decoder enable becomes the data input;
2. The decoder data inputs become the control signals.

Depending on how the decoder inputs (now the control lines), the selected output will appear to follow the value on the enable input (now the data input).

Part II

Sequential logic

14 Latches

The latch is one type of *storage element*. It's a storage element because it exhibits the concept of *memory*.

14.1 SR latch

We want a circuit in which we can force one output to 1, force the output to 0 and "hold" the output. Further, the circuit will have 2 outputs which are always *complements* of each other. The SR latch is shown in Figure 14.1. There are 3 different cases to consider to

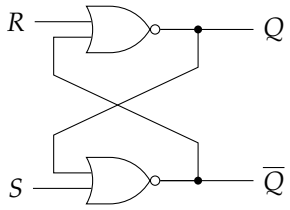


Figure 14.1: Circuit for the SR latch.

see if this circuit does what we want and to figure out how it does it.

- Case I: Set $S = 1$ and $R = 0$. Then, change S to $1 \rightarrow 0$. This is illustrated in Figure 14.2. By considering how a **NOR** gate

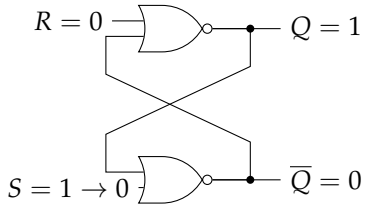


Figure 14.2: Case I for the SR latch behaviour.

  works, setting $S = 1$ causes the bottom **NOR** to produce $\overline{Q} = 0$. The top **NOR** gate has inputs 00 and therefore produces $Q = 1$. We will call this the *set* state. Next, S is changed to 0. The bottom **NOR** gate now has inputs 10 which means that $\overline{Q} = 0$ still. The top **NOR** gate remains unchanged and $Q = 1$. We will call this the *memory* or *hold* state. In summary we find the following

$$\begin{array}{cccccc} S=1 & R=0 & \rightarrow & Q=1 & \overline{Q}=0 & \leftarrow \textit{set} \\ & & \downarrow & & & \\ S=0 & R=0 & \rightarrow & Q=1 & \overline{Q}=0 & \leftarrow \textit{hold} \end{array}$$

- Case II: Set $S = 0$ and $R = 1$. Then, change R to $1 \rightarrow 0$. This is illustrated in Figure 14.3. By considering how a **NOR** gate

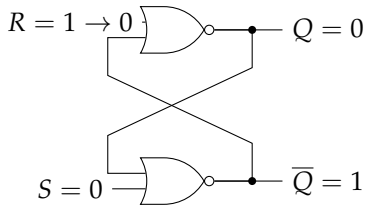


Figure 14.3: Case II for the SR latch behaviour.

works, setting $R = 1$ causes the top **NOR** to produce $Q = 0$. The bottom **NOR** gate has inputs 00 and therefore produces $\overline{Q} = 1$. We will call this the *reset* state. Next, R is changed to 0. The top **NOR** gate now has inputs 01 which means that $Q = 0$ still. The bottom **NOR** gate remains unchanged and $\overline{Q} = 1$. We will call this the *memory* or *hold* state. In summary we find that

$$\begin{array}{cccccc} S=0 & R=1 & \rightarrow & Q=0 & \overline{Q}=1 & \leftarrow \textit{reset} \\ & & \downarrow & & & \\ S=0 & R=0 & \rightarrow & Q=0 & \overline{Q}=1 & \leftarrow \textit{hold} \end{array}$$

- Case III: Set $S = 1$ and $R = 1$ and let $S = 1 \rightarrow 0$ and $R = 1 \rightarrow 0$. This is the last case to consider and is illustrated in Figure 14.4.

  With both $S = 1$ and $R = 1$ we see that both $Q = 0$ and

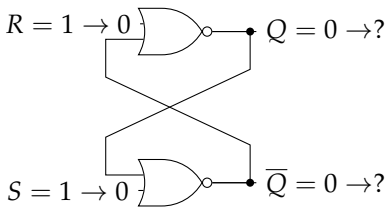


Figure 14.4: Case III for the SR latch behaviour.

$\overline{Q} = 0$ which should be troubling — the outputs are no longer complements of each other. We will call the situation with $S = 1$ and $R = 1$ the *restricted* or *undesireable* state. Now assume that both S and R change to 0. Different things could happen. Because both **NOR** gates have inputs 00 which means their outputs need to change. However, we can assume things change at different times.

  - If we assume that Q changes first from $Q = 0 \rightarrow 1$, this means the bottom **NOR** gate will inputs 10 and so $\overline{Q} = 0$. In this case,

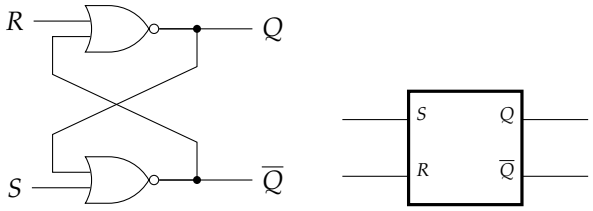
the different circuit values are $S = 0$, $R = 0$ and $Q = 1$ and $\overline{Q} = 0$.

- If we assume that $\overline{Q}$ changes first from $\overline{Q} = 0 \rightarrow 1$, this means the top **NOR** gate will inputs 01 and so $Q = 0$. In this case, the different circuit values are $S = 0$, $R = 0$ and $Q = 0$ and $\overline{Q} = 1$.
- If we assume that both Q and $\overline{Q}$ changes at the same time from $Q = 0 \rightarrow 1$ and $\overline{Q} = 0 \rightarrow 1$, this means that both **NOR** gates will now have inputs 01 so there outputs should change back to $Q = 0$ and $\overline{Q} = 0$. This situation could cause *oscillations*.

The bottom line is that $S = 1$ and $R = 1$ is an input combination that should be avoided.

We can avoid $S = 1$ and $R = 1$ by adding additional logic at the inputs such that we never get this input. If we add logic such that $S = 1$ and $R = 1$ causes the outputs to become $Q = 1$ and $\overline{Q} = 0$, we have a *set dominated latch*. If we add logic such that $S = 1$ and $R = 1$ causes the outputs to become $Q = 0$ and $\overline{Q} = 1$, we have a *reset dominated latch*. We could do something else like have $S = 1$ and $R = 1$ cause the latch to *flip* or *toggle* its outputs. In this case we will get a *JK latch*.

The behaviour of the SR latch is described in Figure 14.5. If



S	R	Q	$\overline{Q}$	**Action**
1	0	1	0	*set*
0	1	0	1	*reset*
0	0	1	0	*memory or hold* after $S = 1$, $R = 0$
0	0	0	1	*memory or hold* after $S = 0$, $R = 1$
1	1	?	?	*restricted*

Figure 14.5: The behaviour of the SR latch summarized.

we let $Q(t+1)$ represent the next value of Q and $Q(t)$ represent the current value of Q, then the SR latch can also be described via the equation given by

$$Q(t+1) = Q(t)\overline{R} + S\overline{R}$$

14.2 $\overline{S}\,\overline{R}$ latch

We can also build a similar circuit using **NAND** gates, but the operation is a little bit different. The circuit is shown in Figure 14.6

and is called the $\overline{S}\,\overline{R}$ latch. We could do analysis similar to the

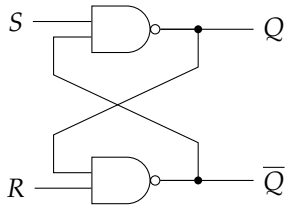
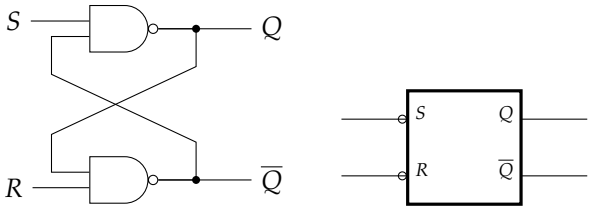


Figure 14.6: The $\overline{S}\,\overline{R}$ latch.

SR latch and conclude there is a *set*, *reset*, *hold* and *restricted* case. The behaviour of the $\overline{S}\,\overline{R}$ latch is described in Figure 14.7. As an

The $\overline{S}\,\overline{R}$ latch:



S	R	Q	$\overline{Q}$	**Action**
1	0	0	1	*reset*
0	1	1	0	*set*
1	1	0	1	*memory or hold* after $S = 1$, $R = 0$
1	1	1	0	*memory or hold* after $S = 0$, $R = 1$
0	0	?	?	*restricted*

Figure 14.7: The behaviour of the $\overline{S}\,\overline{R}$ latch summarized.

equation, this latch is described by

$$Q(t+1) = Q(t)R + \overline{S}R$$

Note the bubbles on the S and R inputs; this tends to be described as "active low" inputs.

14.3 Gated D latch

The gated D latch is shown in Figure 14.8. The D input is called the *data* input and the G input is called the *gate* input. There are several cases to consider to figure out the behaviour of this latch.

- Case I: If $G = 1$ and $D = 1$, then the output is *set* and this is shown in Figure 14.9.
- Case II: If $G = 1$ and $D = 0$, then the output is *reset* and this is shown in Figure 14.10.

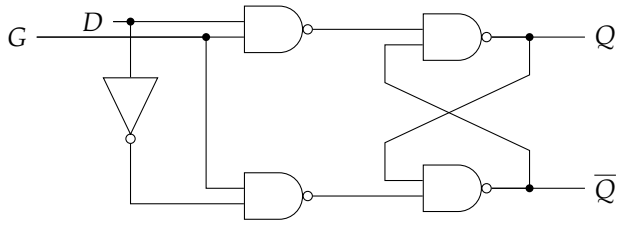


Figure 14.8: The D latch.

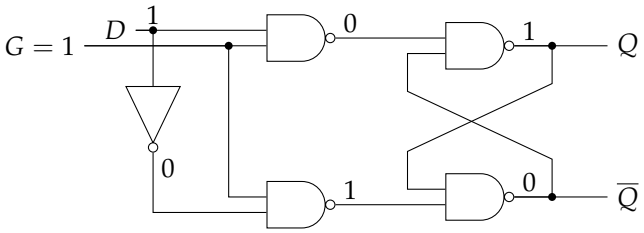


Figure 14.9: The D latch when *set*.

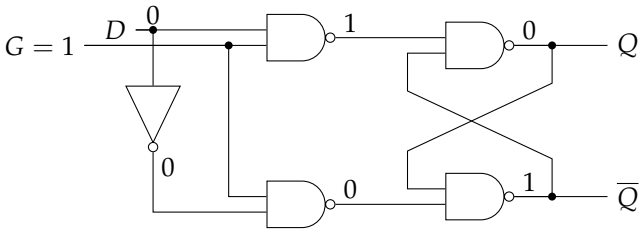


Figure 14.10: The D latch when *reset*.

- Case III: If $G = 0$ the value of D is not relevant; the outputs remain at their previous value and *hold* and this is shown in Figure 14.11.

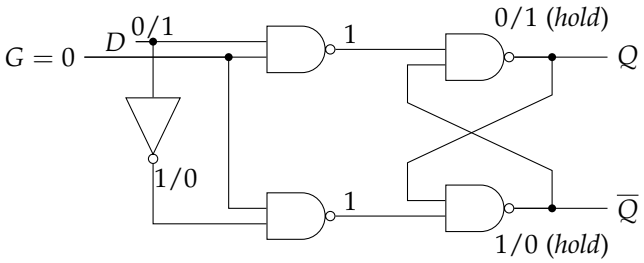


Figure 14.11: The D latch when in *hold*.

The behaviour of the D latch is summarized in Figure 14.12 and its symbol is shown in Figure 14.13. Note that we don't need to

D	G	Q	$\overline{Q}$	**Action**
1	1	1	0	$\leftarrow$ *set*
0	1	0	1	$\leftarrow$ *reset*
0/1	0	*previous value*		$\leftarrow$ *hold*

Figure 14.12: Summary of the D latch behaviour.

worry about any "*restricted*" cases like with the SR latch or the $\overline{S}\,\overline{R}$ latch. As an equation, we can describe the D latch using the

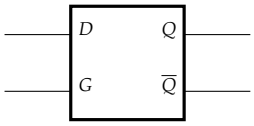


Figure 14.13: Symbol for D latch.

equation given by

$$Q(t+1) = DG + Q(t)\overline{G}$$

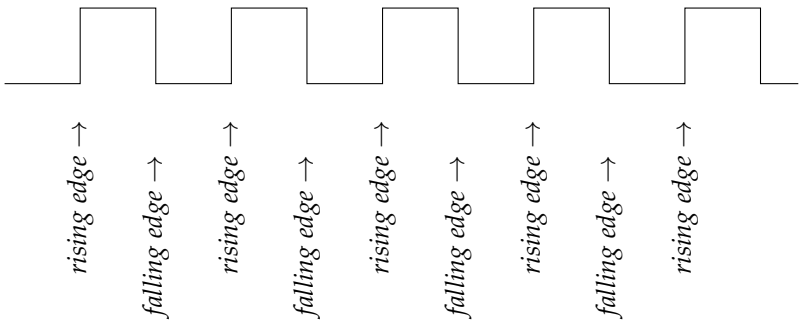
14.4 Summary

There are other configurations and types of latches, but these ones are enough for our purposes. We won't do too much with latches depending on time, but they are useful for a number of things. One last comment: Latches are an example of a *level sensitive* storage element — the outputs are *set*, *reset* or *hold* depending on the logic levels/values of the inputs.

15 Flip flops

A *flip flop* is another type of storage element which exhibits the concept of memory. Flip flops differ from latches in that flip flops are *edge triggered* devices. The output of a flip flop *changes* depending on the values of the flip flop inputs *and* when a *clock* input changes from $0 \rightarrow 1$ (or $1 \rightarrow 0$). If the output changes when the clock transitions from $0 \rightarrow 1$, the flip flop is *positive edge triggered*. If the output changes when the clock transitions from $1 \rightarrow 0$, then the flip flop is *negative edge triggered*.

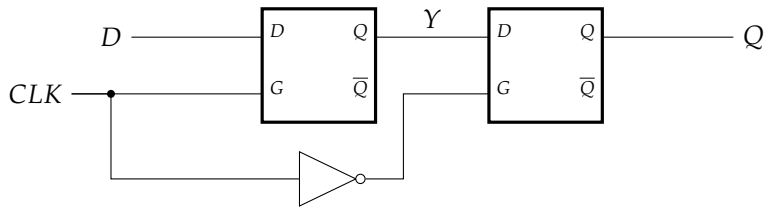
Flip flops are generally controlled by a *clock signal*. A clock signal is a periodic signal. By using flip flops, we can restrict when flip flop outputs change to discrete instances in time — either on the *rising edge* ($0 \rightarrow 1$) of the clock, the *falling edge* ($1 \rightarrow 0$) of the clock, or both.



This is the real beauty of flip flops — we can control precisely in time when the outputs change based on computation taken during the clock period to generate the flip flop inputs.

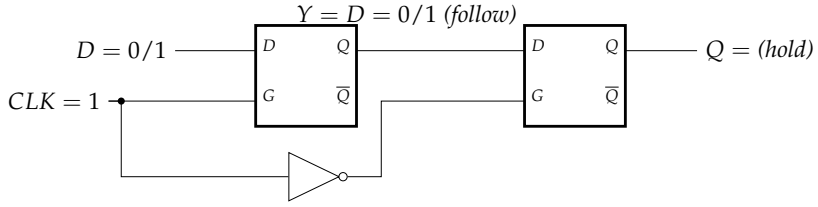
15.1 The master-slave flip flop

Can build a flip flop from two latches to get an idea of how a flip flop works using a previously understood circuit (the latch). The master-slave flip flop:

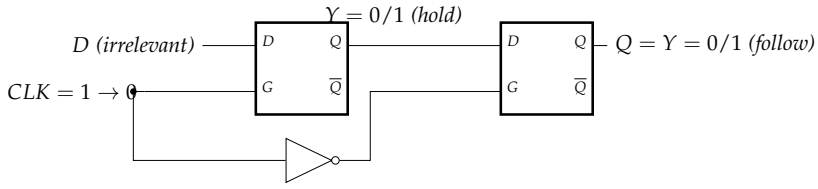


The first latch is called the *master* and the second latch is called the *slave*.

The behaviour of a master-slave flip flop is as follows: Assume $CLK = 1$. The output of the master latch will *follow* the input D so $Y = D$. The slave latch, however, is in *hold* so the output Q will not change and it will be at its previous value.

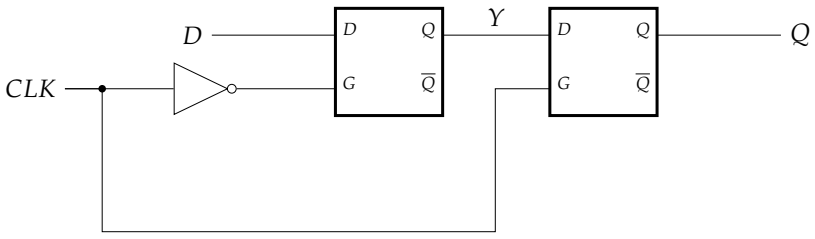


Now, assume CLK changes from $1 \rightarrow 0$. The output of the master latch will *hold* — it will be held at *the value of D just prior to the clock changing*. The slave latch, however, will have its output Q equal to its input $Q = Y$.



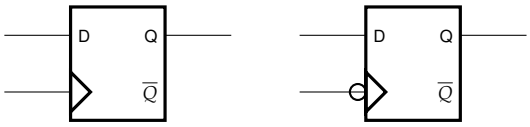
The net result is that it appears that the value of the D input just prior to the CLK changing from $1 \rightarrow 0$ gets transferred to the output Q. This exhibits the behaviour of the definition of a flip flop — the output changes according to the inputs based on the transition of a clock signal. In this particular explanation, we have built an example of a *negative edge triggered D type flip flop* using the concept of master and slave latches.

We can make a positive edge triggered master-slave flop-flop by moving the position of the inverter:



15.2 The DFF

The master-slave flip flop is one way to make the D type flip flop, or DFF. Symbols for both a *positive edge triggered* DFF and a *negative edge triggered* DFF.



Typical to describe a flip flop using a table — the table should show what the flip flop output will become given the current flip flop inputs after the clock makes its active edge transition. This is the *characteristic table*.

D	$Q(t)$	$Q(t+1)$
0	0	0
0	1	0
1	0	1
1	1	1

or

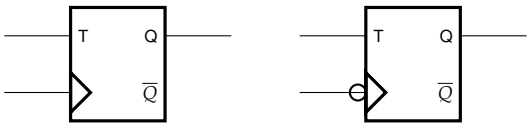
D	$Q(t+1)$
0	0
1	1

Can also use *characterstic equation*:

$$Q(t+1) = D$$

15.3 The TFF

Another type of flip flop that behaves differently compared to the DFF; known as a TFF or *toggle flip flop*. Symbols for both a *positive edge triggered* TFF and a *negative edge triggered* TFF.



The *characteristic table* for the TFF.

T	$Q(t)$	$Q(t+1)$
0	0	0
0	1	1
1	0	1
1	1	0

or

T	$Q(t+1)$
0	$Q(t)$
1	$\overline{Q(t)}$

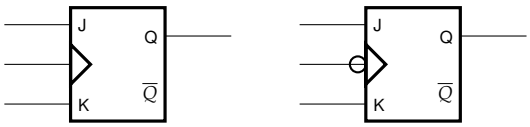
Can also use *characterstic equation*:

$$Q(t+1) = Q(t)\overline{T} + \overline{Q(t)}T = Q(t) \oplus T$$

The *TFF* flips or *toggles* the output if input $T = 1$ when the active clock edge arrives, otherwise the output does not change.

15.4 The JKFF

Another type of flip flop that behaves differently compared to the *DFF* and the *TFF* known as a *JKFF*. Symbols for both a *positive edge triggered JKFF* and a *negative edge triggered JKFF*.



The *characteristic table* for the *JKFF*.

J	K	$Q(t)$	$Q(t+1)$	
0	0	0	0	(hold)
0	1	0	0	(reset)
1	0	0	1	(set)
1	1	0	1	(toggle)
0	0	1	1	(hold)
0	1	1	0	(reset)
1	0	1	1	(set)
1	1	1	0	(toggle)

or

J	K	$Q(t+1)$	
0	0	$Q(t)$	(hold)
0	1	0	(reset)
1	0	1	(set)
1	1	$\overline{Q(t)}$	(toggle)

Can also use *characterstic equation*:

$$Q(t+1) = J\overline{Q(t)} + \overline{K}Q(t)$$

15.5 Control signals — sets, resets and enables

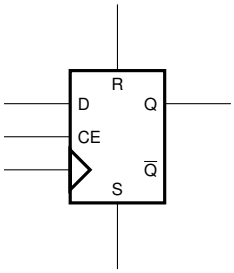
We might see additional input pins on flip flops. Such additional signals might include *set*, *reset* and *enable* inputs.

- A *set* signal is a signal which forces $Q = 1$ regardless of the flip flop input values and the behaviour of the flip flop;
- A *reset* signal is a signal which forces $Q = 0$ regardless of the flip flop input values and the behaviour of the flip flop;

Signals like sets and resets can either be *synchronous* with the clock (change happens at the active clock edge) or *asynchronous* with the clock (change happens immediately). An *enable* is a signal that

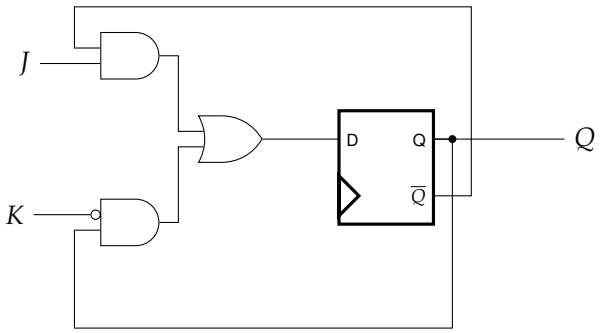
prevents the clock signal from causing a change in Q (it effectively "blocks" the clock from the flip flop).

Example of a DFF with additional input/control pins — set, reset and enable.



15.6 Constructing flip flops from other types of flip flops

By considering the characteristic equations for each flip flop type, we can always build one type of flip flop from another type of flip flop. As an example, we can construct a $JKFF$ using a DFF plus some additional logic.



For the DFF, $Q(t+1) = D$ (characteristic equation), but that $D = J\overline{Q(t)} + \overline{K}Q(t)$. Therefore, the DFF plus logic acts exactly like a $JKFF$.

15.7 Timing parameters for flip flops

There are some *restrictions* with respect to when signals can change in order to ensure that a flip flop behaves *correctly*. Observing these parameters is necessary to ensure that circuits involving flip flops work properly.

Two timing parameters represent a relationship between the data inputs of the flip flop and the clock input of the flip flop:

- Setup time (T_{SU}): The *setup time* of a flip flop is defined as the amount of time that the data inputs need to be held stable (not

changing) prior to the arrival of the active clock edge.

- Hold time (T_H): The *hold time* of a flip flop is defined as the amount of time that the data inputs need to be held stable (not changing) after the arrival of the active clock edge.

Both *setup* and *hold* times need to be obeyed in order to guarantee that the output of the flip flop changes to the correct value when the clock edge arrives based on the values of the data inputs.

Finally, there is a relationship between the data output of the flip flop and the clock input of the flip flop. The *clock to output time* (T_{CO}) of a flip flop is defined as the amount of time it takes for the output to change and become stable (reliable) after the arrival of the active clock edge. The output of a flip flop cannot be trusted until after an amount of time equal to the clock to output time.

16 Registers

A register is nothing more than n flip flops together in a group in order to perform some task. Flip flops in a register all use the same clock. It's common that registers are created with DFFs. A simple 3-bit register is shown in Figure 16.1. This simple register which will load new data on every active clock edge.

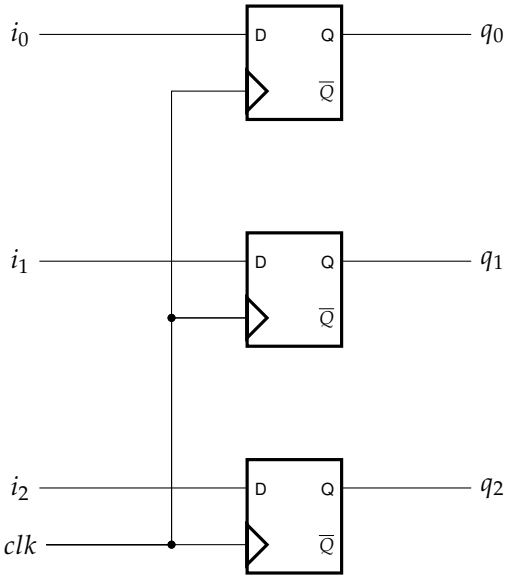


Figure 16.1: Example of a simple 3-bit register. The register uses DFFs and all DFFs are clocked by the same signal.

16.1 Register with parallel load and hold

Registers often need to do more than just load new data every active clock edge. For example, we can consider a register which has the ability to load new data as well as hold current data at the flip flop outputs. Such a circuit is shown in Figure 16.2 for a 3-bit register.

The idea is to "get the correct value to the input of each flip flop depending on the operation to be performed". In this circuit, the operation is controlled by the ld signal. When $ld = 1$, the register loads new data. When $ld = 0$, the register holds its current value.

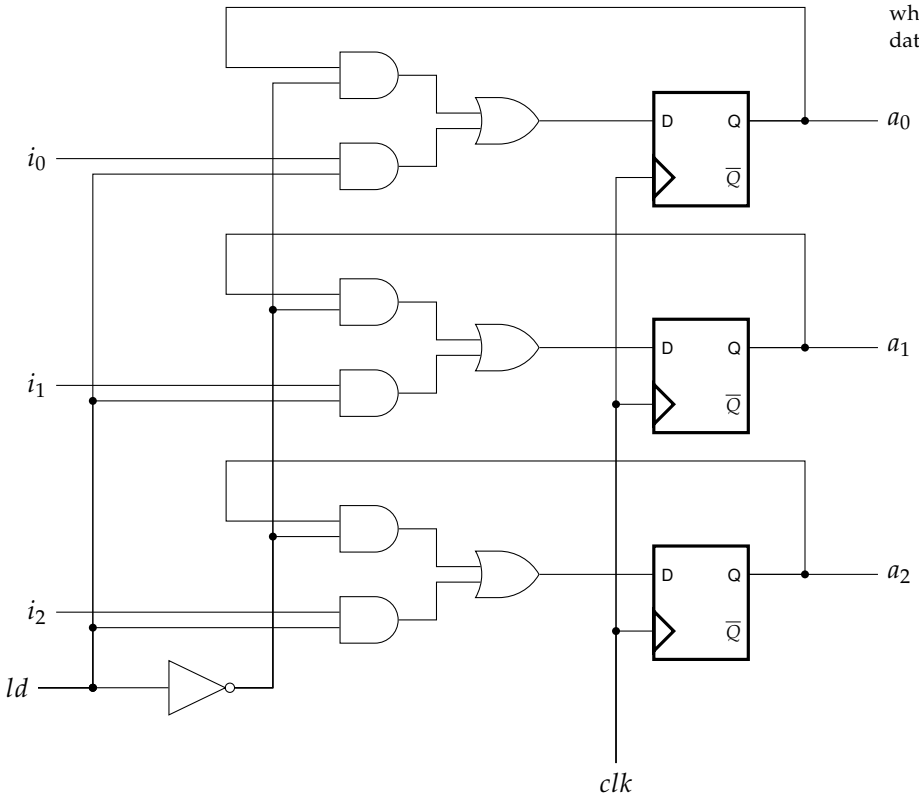


Figure 16.2: Example of a 3-bit register which has the capability to load new data as well as hold existing data.

The logic placed in front of the *DFF*s is nothing more than a multiplexer with *ld* as a select line and the appropriate value connected to each data input of the multiplexer.

16.2 Shift registers

Another common register is the *shift register*. The purpose of a shift register is to accept input and to shift it one bit over on every active clock edge. A shift register, for example, can be used to take "serial data" and convert it to "parallel data" or visa versa (assuming we have a shift register with parallel load). An example of a simple 3-bit shift register is shown in Figure 16.3. With an n-bit shift register,

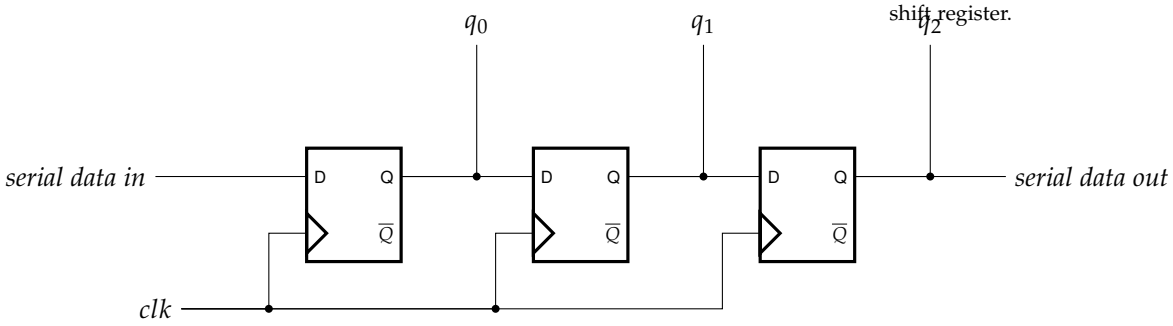


Figure 16.3: Example of a simple 3-bit shift register.

as the active clock edge arrives, data present at the *serial data in* gets transferred towards the *serial data out*; so the data gets shifted to the right each time an active clock edge arrives and data appears to "move" from q_0 towards q_{n-1}.

16.3 Universal shift registers

As our last example of a register, we will consider a *universal shift register*. A universal shift register is simply a shift register augmented to perform additional operations. For example, we can consider making a shift register that can shift from q_0 towards q_{n-1} (shift up), can shift from q_{n-1} towards q_0 (shift down), hold data or load new data. Such a register would require some control signals and we can specify how such a circuit show behave by considering the table shown in Figure 16.4. By *shift up*, we mean that data moves from q_0 towards q_{n-1}. By *shift down*, we mean that data moves from q_{n-1} towards q_0.

A universal 3-bit shift register that behaves according to the table in Figure 16.4 is shown in Figure 16.5.

Inputs			Action
load	*up*	*dn*	
0	0	0	*hold*
0	0	1	*shift down*
0	1	X	*shift up*
1	X	X	*load*

Figure 16.4: Table describing the behaviour of a universal shift register capable of shift up, shift down, parallel load and hold operations.

The extra circuitry (in addition to the *DFF*s) simply ensures that the correct input reaches every flip flop depending on the operation to be performed. In effect, this circuitry is simply performing a multiplexing operation which we can see by considering the mathematics of the circuitry in front of every flip flop data input as shown in Figure 16.6.

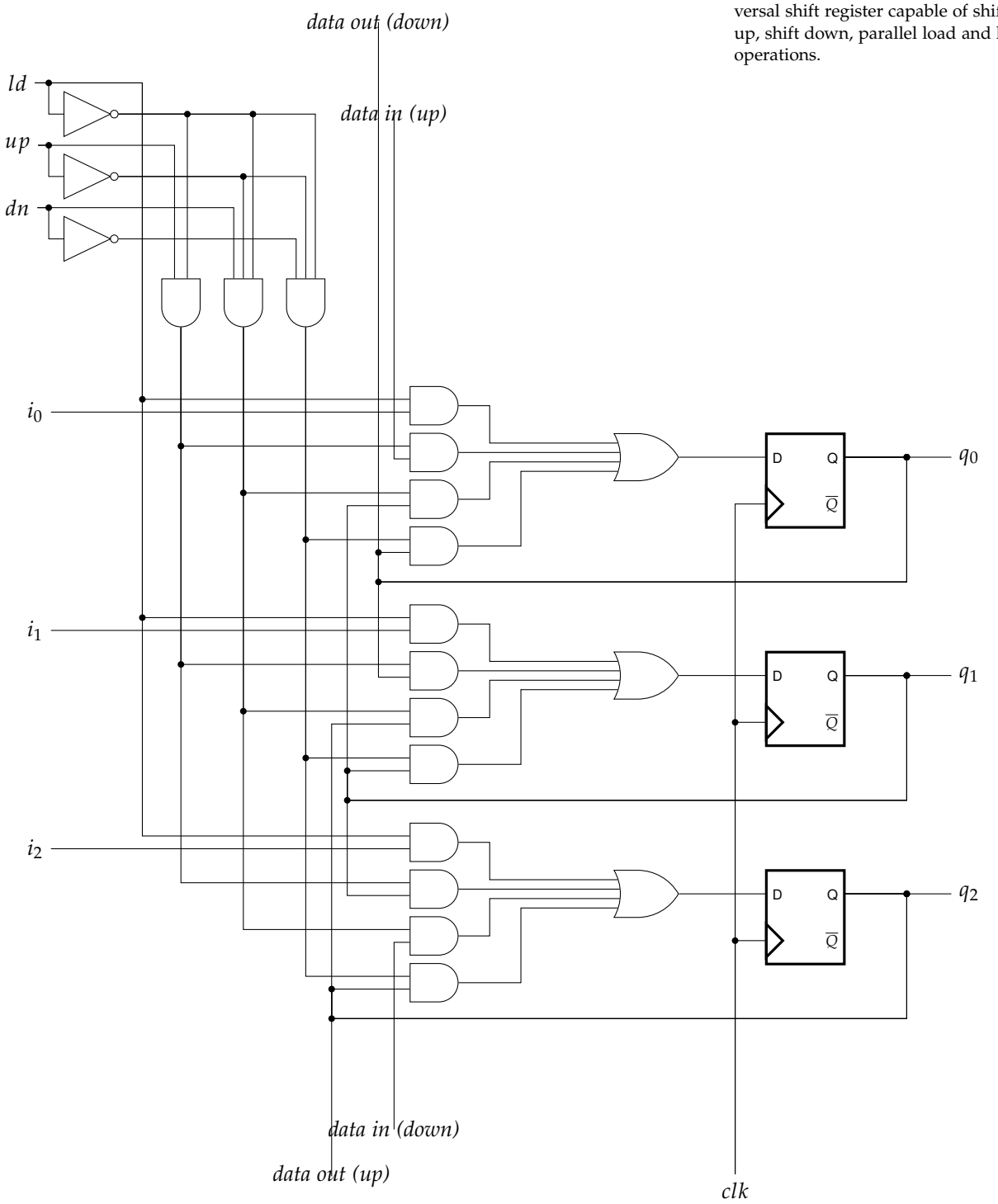


Figure 16.5: Example of a 3-bit universal shift register capable of shift up, shift down, parallel load and hold operations.

$$
\begin{aligned}
d_j &= \underbrace{(ld)i_j}_{\text{load}} + \underbrace{(\overline{ld})upq_{i-1}}_{\text{shift up}} + \underbrace{(\overline{ld})(\overline{up})(dn)q_{i+1}}_{\text{shift down}} + \underbrace{(\overline{ld})(\overline{up})(\overline{dn})q_i}_{\text{hold}} \\
&= (ld)(i_j) + +(\overline{ld})((up)q_{i-1} + +(\overline{up})(dn)q_{i+1} + (\overline{up})(\overline{dn})q_i) \\
&= (ld)(i_j) + +(\overline{ld})((up)(q_{i-1}) + (\overline{up})((dn)q_{i+1} + (\overline{dn})q_i)) \\
&= (ld)(i_j) + +(\overline{ld})((up)(q_{i-1}) + (\overline{up})(\underbrace{(dn)q_{i+1} + (\overline{dn})q_i}_{\text{MUX}}))
\end{aligned}
$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{MUX}}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{MUX}}$$

Figure 16.6: The algebra to explain the logic in front of every flip flop data input in the universal shift register. The logic is simply performing a multiplexing operation to ensure that, based on the values of the control signals ld, up and dn.

17 Counters

A counter is a circuit whose outputs go through a prescribed sequence of values and repeats over and over. The outputs of the counter are held at the outputs of flip flops. The change from one value to the next value happens on the arrival of the active edge of some signal; e.g., a clock signal. Counters come in two varieties: *asynchronous* and *synchronous* counters. We will consider mainly synchronous counters, but will still consider an asynchronous counter.

17.1 Asynchronous counters — the binary ripple adder

Asynchronous counters are characterized by the observation that different signals drive different clock inputs on different flip flops. This means that the outputs of each flip flop *do not change at the same time are not synchronized with some master clock.*

To demonstrate the design of an synchronous counter, we will consider the binary ripple counter. We will consider onyl 3 bits and design a 3-bit counter that counts $0, 1, \cdots 7$ and repeats. The circuit should have a signal *en* such that $en = 1$ the circuit should count upon the arrival of the active clock edge. When $en = 0$, the circuit should not count (hold its current value). The sequence of numbers we want to produce (ignore the *en* input for the moment and assume the counter is always counting) is shown in Figure 17.1. We can

current count			next count			
a_2	a_1	a_0	a_2	a_1	a_0	
0	0	0	0	0	1	
0	0	1	0	1	0	
0	1	0	0	1	1	
0	1	1	1	0	0	
1	0	0	1	0	1	
1	0	1	1	1	0	
1	1	0	1	1	1	
1	1	1	0	0	0	← *repeat*

Figure 17.1: Sequence of count values for a 3-bit binary counter.

make some observations with respect to when and how individual bits in the count sequence need to change: i) a_0 always toggles; ii) a_1 toggles when $a_0 = 1 \rightarrow 0$; iii) a_2 toggles when $a_1 = 1 \rightarrow 0$. In other words, the i-th bit toggles when the $(i-1)$-th bit makes a transition from $1 \rightarrow 0$.

The observation that outputs need to toggle motivates the use of *TFF*s. The observation that things toggle when a signal changes $1 \rightarrow 0$ motivates using these signals as inputs to flip flop clock inputs. Our required circuit is shown in Figure 17.2. Note the creative

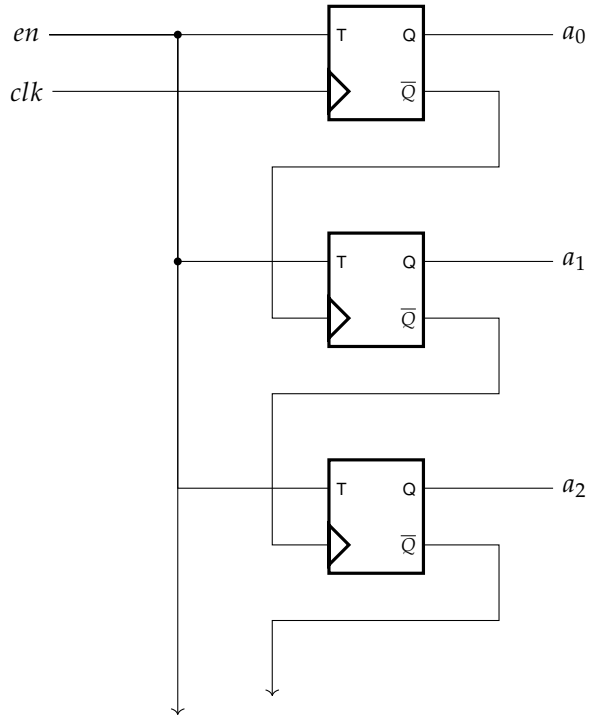


Figure 17.2: Circuit for a 3-bit binary ripple counter.

use of the enable signal... If $en = 0$ all inputs to the toggle flip flops is 0 and the flip flops will not toggle.

If we don't care about how fast the circuit works, then there is really no problem with this sort of counter. However, it can be *slow* if the number of bits is very large. Here's why: Consider the number of bits n to be very large (e.g., 128 or something like that). Further, consider all of the flip flops to be 1; the next count value is 0 and *all flip flop outputs need to toggle*. See the problem? Output a_0 must toggle (and it takes T_{co} for a_0 to change...), then a_1 will toggle (and it takes another T_{co} for a_1 to change), then a_2 will toggle (and it takes another T_{co} for a_2 to change), and so on... Because every flip flop is clocked by a different clock and the clocks form a *chain*, this counter can be quite slow. It is better to create a circuit in which all flip flop outputs

change at the same time.

17.2 Synchronous counters — the binary up counter

Synchronous counters are characterized by the observation that a single clock signal is used to clock every flip flop. This means that all the flip flop outputs *change simultaneously and are synchronized with the clock*.

We can again consider the design a 3-bit synchronous binary up counter with an enable input en. When $en = 0$, the counter should not count. When $en = 1$, the circuit should count upon the arrival of the active clock edge. The output sequence always increases by 1 and repeats over and over. The count sequence is shown in Figure 17.3 assuming $en = 1$. We can make some observations. However, un-

current count			next count			
a_2	a_1	a_0	a_2	a_1	a_0	
0	0	0	0	0	1	
0	0	1	0	1	0	
0	1	0	0	1	1	
0	1	1	1	0	0	
1	0	0	1	0	1	
1	0	1	1	1	0	
1	1	0	1	1	1	
1	1	1	0	0	0	← *repeat*

Figure 17.3: Count sequence for a binary up counter.

like when we designed a binary ripple counter, we will **not** consider how *changes* in signals cause other signals to change. Rather, we will consider the logic values of other signals when other signals need to change. That said, we can make the following observations. i) a_0 always toggles; ii) a_1 toggles when the current value of $a_0 = 1$; iii) a_2 toggles when the current value of $a_1a_0 = 11$.

In other words, a_i should toggle when bits a_{i-1} down to a_0 are all 1. This would apply for a binary up counter with any number of bits. The observation that bits need to toggle motivates the use of TFF. Considering the logic required for bits to change, we find the circuit shown in Figure 17.4. Note the creative use of the enable signal... If $en = 0$ all inputs to the TFF will be forced to 0 and they will not toggle. Otherwise, the inputs to the TFF are the conditions required for the flip flops to toggle appropriately. We could use set/reset control signals to initialize the counter to any *initial* state.

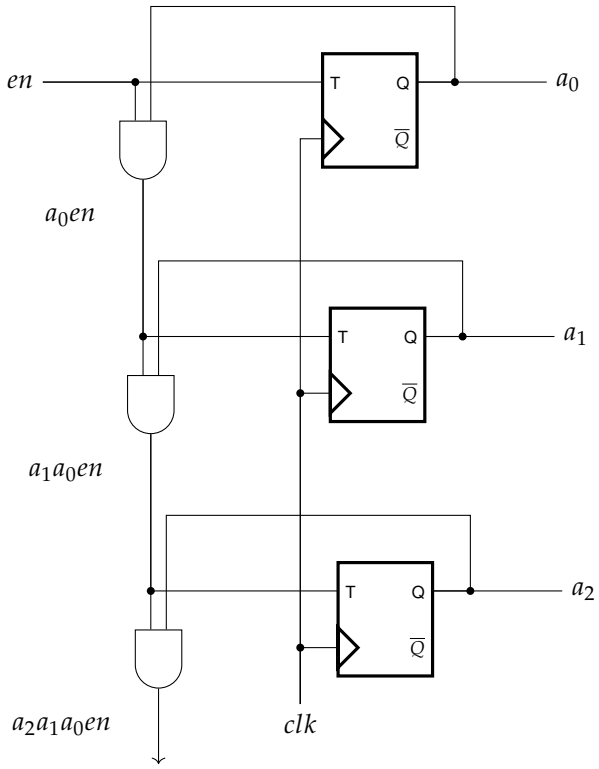


Figure 17.4: Circuit implementing a 3-bit synchronous binary up counter.

17.3 *Synchronous counters — binary down counter*

We can design a synchronous binary down counter following the same ideas as the up counter. For example, consider the design of a 3-bit synchronous binary down counter with an enable input en. When $en = 0$, the counter should not count. When $en = 1$, the circuit should count upon the arrival of the active clock edge. The count should always decrease by 1 and repeats over and over.

The required count sequence is shown in Figure 17.5 assuming that $en = 1$. We can make the following observations: i) a_0 always

current count			next count			
a_2	a_1	a_0	a_2	a_1	a_0	
1	1	1	1	1	0	
1	1	0	1	0	1	
1	0	1	1	0	0	
1	0	0	0	1	1	
0	1	1	0	1	0	
0	1	0	0	0	1	
0	0	1	0	0	0	
0	0	0	1	1	1	← *repeat*

Figure 17.5: Count sequence for a 3-bit synchronous binary down counter.

toggles; ii) a_1 toggles when the current value of $a_0 = 0$; iii) a_2 toggles when the current value of $a_1a_0 = 00$. In other words, a_i should toggle when bits a_{i-1} down to a_0 are all 0. This would apply for a binary down counter with any number of bits.

The observation that bits need to toggle motivates the use of TFF. The circuit for a binary down counter is shown in Figure 17.6.

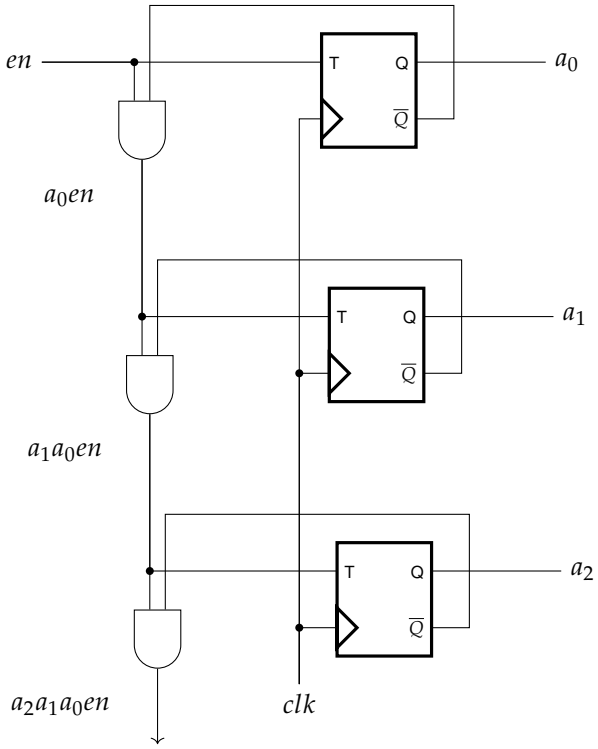


Figure 17.6: Circuit for a 3-bit synchronous binary down counter.

Note the creative use of the enable signal — if $en = 0$ all inputs to the TFF will be forced to 0 and they will not toggle. Otherwise, the inputs to the TFF are the conditions required for the flip flops to toggle appropriately. We could use set/reset control signals to initialize the counter to any *initial* state.

17.4 *Synchronous counters — binary up/down counter*

We can combine different operations together. For example, we could create a circuit that can count both up or down, or hold the current count value. To do this, we require some control signals and a table to explain the operation of the counter based on the value of the control signals. As an example, we will design a 3-bit counter that can count up, down or hold its current value according to the table shown in Figure 17.7. We can take the approach of using circuits

Inputs		Action
up	*down*	
0	0	*don't count*
0	1	*count down*
1	X	*count up*

Figure 17.7: Table explaining the operation of the control signals for a binary counter capable of counting up, down or holding its current count value.

we *previously built* and adding the additional circuitry we require to control the behaviour of the circuit. Specifically, we need to use the control signals to *get the correct value to the input of each flip flop so that the flip flop changes its output appropriately when the active clock edge arrives.*

The resulting circuit is shown in Figure 17.8. The extra gates

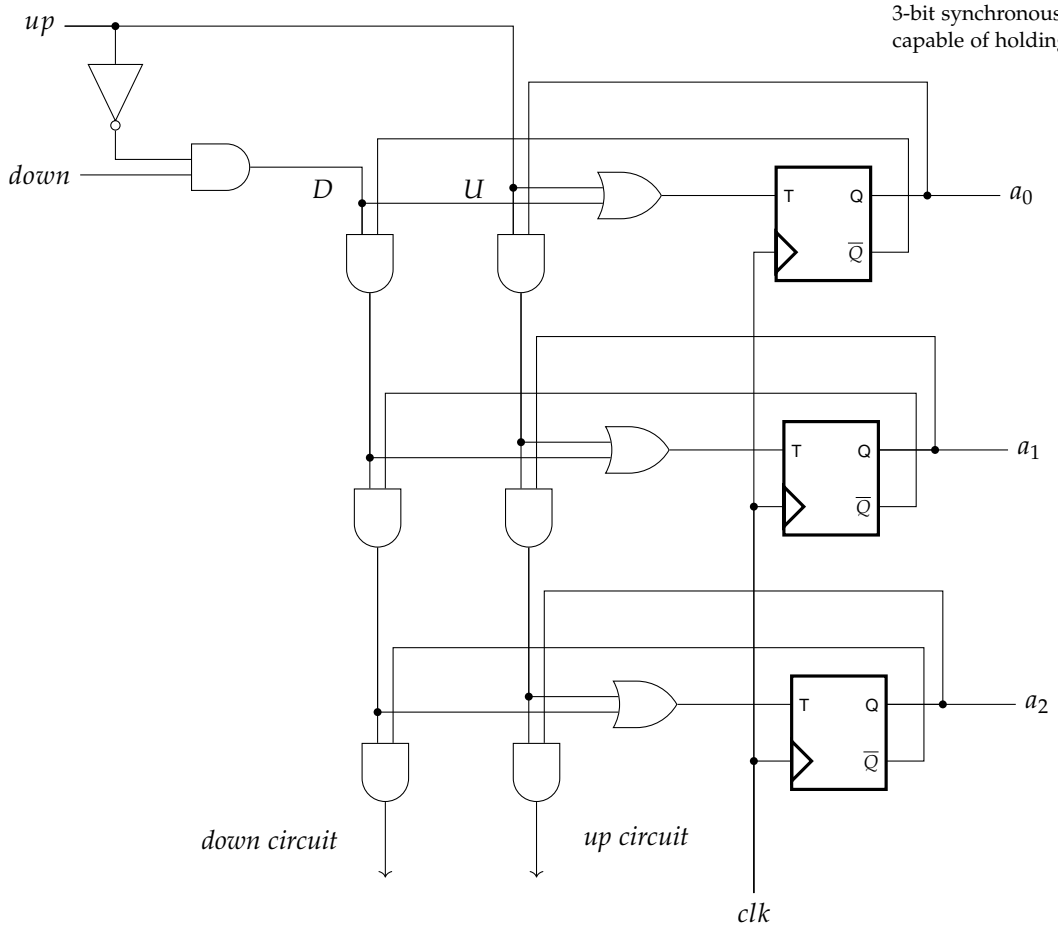


Figure 17.8: Circuit implementing a 3-bit synchronous up/down counter capable of holding its current value.

and control logic are effectively performing a multiplexer operation to get the correct signal to the input of the flip flop. Consider the signals D and U in the circuit — these are the enable signals for the

down counter and up counter, respectively... Think about it...

17.5 *Synchronous counters — binary up/down counter with parallel load*

We can extend to behaviour of the previous binary up/down counter to allow for the loading of an arbitrary value to start the count sequence. In other words, we want to design another circuit with some control signals that behaves according to the table shown in Figure 17.9. The circuit is shown in Figure 17.10. The extra gates

Inputs			**Action**
load	*up*	*down*	
0	0	0	*hold*
0	0	1	*count down*
0	1	X	*count up*
1	X	X	*load*

Figure 17.9: Table describing the behaviour of a binary up/down counter which is capable of holding its current value *and* loading an arbitrary value to start counting from.

and control logic are effectively performing a multiplexer operation to get the correct signal to the input of the flip flop. **Question:** What is the purpose of the **XOR** gates? **Answer:** Since we are using *TFF*, we need to *compare* i_j and a_j to decide how to load (do we need to invert a_j or not?).

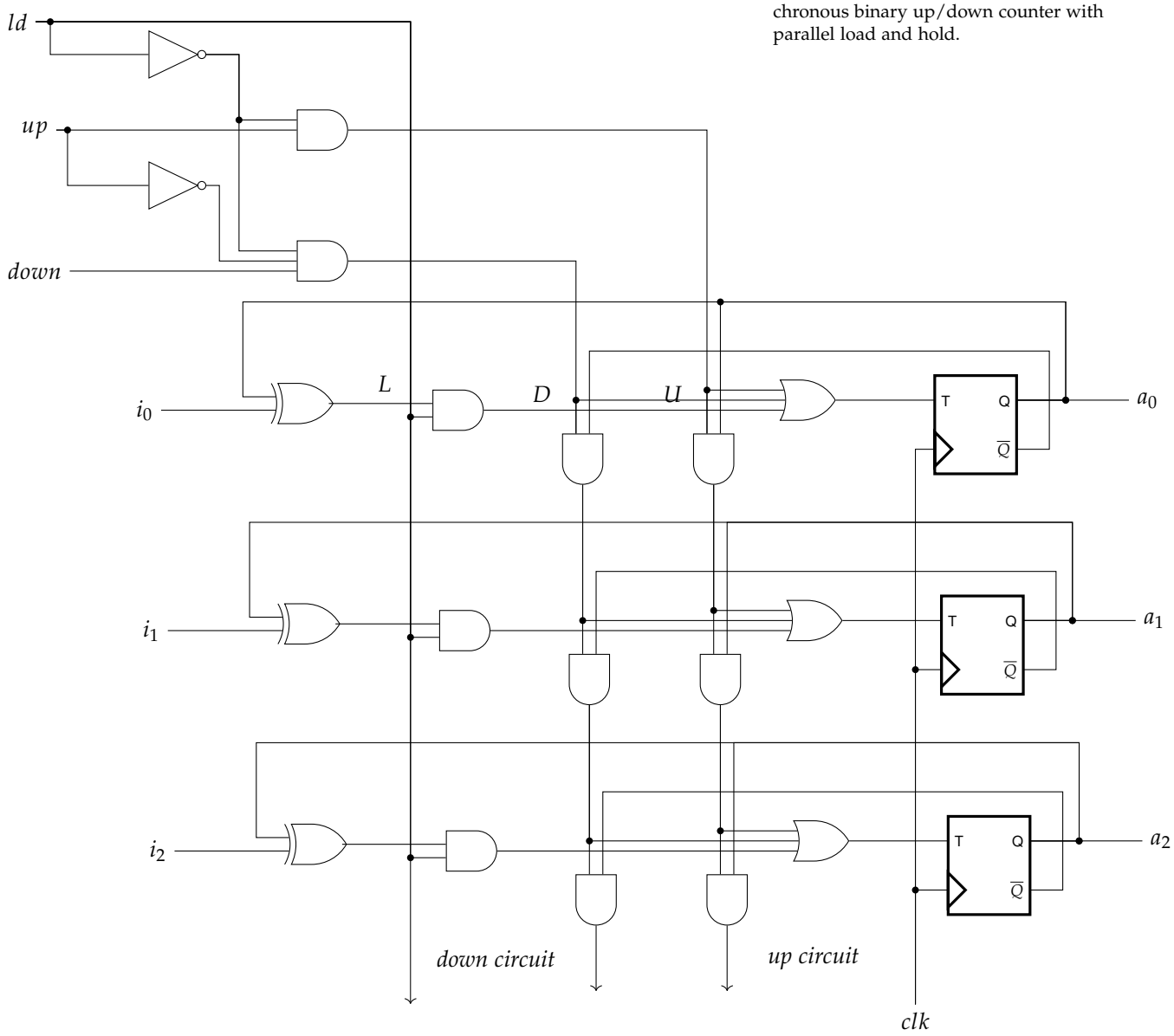


Figure 17.10: Circuit for a 3-bit synchronous binary up/down counter with parallel load and hold.

17.6 Counter symbols

Premade counters would be represented with a schematic symbol and likely accompanied by a table explaining its operation, including the purpose of each control signal, which signal takes priority over another, and so forth. An example of a counter symbol is shown in Figure 17.11 along with a table explaining its operation in Figure 17.12.

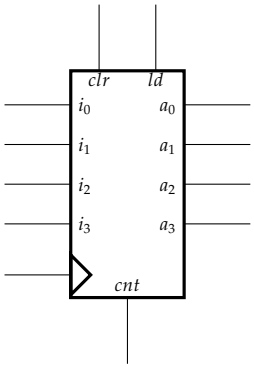


Figure 17.11: An example of a schematic symbol for a counter.

clr	*ld*	*cnt*	*clk*	Function
0	X	X	X	Output to zero
1	1	X	↑	Parallel load
1	0	1	↑	Count up
1	0	0	↑	Hold

Figure 17.12: Table explaining the operation of the counter shown in Figure 17.11.

17.7 Modulo counters

If we don't want to count through the entire binary sequence, then we might want to build a *modulo counter*. A modulo-n counter would count $0, 1, \cdots n-1$ and repeat. We could also decide to start the count sequence at some value other than 0. Such a counter could be designed from scratch, or we could modify an existing binary up counter to perform the task. To construct a modulo counter from an existing binary up counter, we simply need to add additional logi to detect when we have reached our maximum count. Then, upon the next clock cycle, do a parallel load to restart the count sequence.

For example, consider designing a synchronous counter that counts $2, 3, \cdots, 9$ and then repeats. Do the design using an existing 4-bit binary up counter. The solution is shown in Figure 17.13. When this circuit hits 9, the output of the **AND** gate will be 1 and the

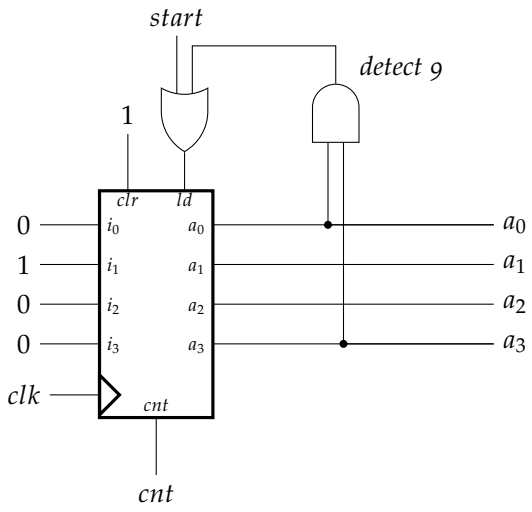


Figure 17.13: Design of a modulo-10 counter using a pre-existing binary up counter.

ld signal will be active. At the next active clock edge, the circuit will load the binary value of 2 and start counting correctly. We've also added a *start* signal to be able to reset the counter and start counting from 2.

17.8 Generic counter design

Counter design can also be generic — we can design a counter which counts in any sequence we want. Further, we can use any type of flip flop. To do this, we can start by drawing a version of a *state diagram* to illustrate the count sequence. We will illustrate via an example. Consider designing a counter which counts in the sequence $3 \rightarrow 6 \rightarrow 0 \rightarrow 1 \rightarrow 7$ and repeats.

The state diagram for this count sequence is shown in Figure 17.14.

It's useful to redraw the state diagram and, instead of showning

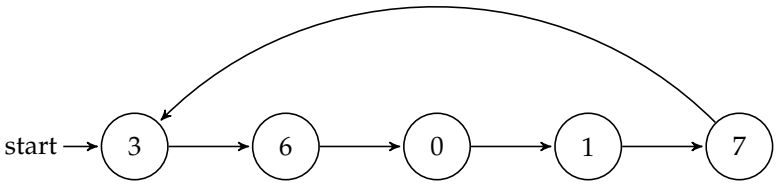


Figure 17.14: State diagram for a counter circuit.

the decimal values of the count sequence, to show the values in their binary form. This is illustrated in Figure 17.15.

Each *bubble* in the state diagram represents a *state*. Each *edge* in the state diagram tells us how we transition from one state to another

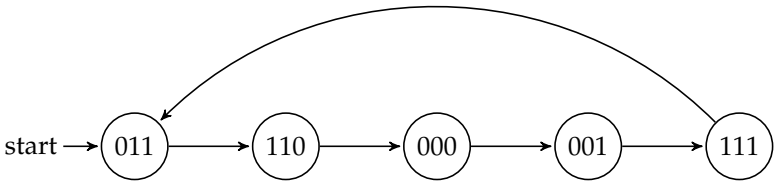


Figure 17.15: State diagram for a counter circuit with binary values shown instead of decimal values.

state when the active clock edge arrives. The *values* shown inside of the state bubbles in Figure 17.15 are the required outputs of the circuit which depend only on the state. In this example, the number of output bits also tells us how many flip flops we require for the counter — in this case, we need 3 flop flips, one for each bit.

We can convert this diagram to a table which is a version of a *state table*. The state table shows the *current state* and the *next state*. More or less, the *current state* represents the current output of the flip flops (so the current count value) and the *next state* is what the flip flop outputs need to change to (so the next count value). The state table is shown in Figure 17.16. the state table tells us how the flip flop

Current State			Next State		
a_2	a_1	a_0	a_2	a_1	a_0
0	1	1	1	1	0
1	1	0	0	0	0
0	0	0	0	0	1
0	0	1	1	1	1
0	0	1	0	1	1

Figure 17.16: State table for the state diagram shown in Figure 17.15.

outputs need to change when the active clock edge arrives.

17.8.1 *Generic counter design — implementation with DFF*

Assume we want to use *DFF* to build the counter. We should recall how *DFF*s work — the behaviour of a *DFF* is shown in Figure 17.17.

We can augment the state table with additional columns that show the required inputs of each *DFF* so that each output bit will change appropriately. This is shown in Figure 17.18. Using the information in Figure 17.18, we can use the *DFF* inputs to write down equations for each of the *DFF* inputs (we could use Karnaugh maps to get these equations). The resulting set of equations is

$$\begin{aligned} d_2 &= a_2' a_0 \\ d_1 &= a_0 \\ d_0 &= a_1' + a_2 a_0 \end{aligned}$$

D	$Q(t+1)$
0	0
1	1

Figure 17.17: Behaviour of a *DFF* repeated here to remind ourselves how they work and how the outputs change based on the data input.

or

D	$Q(t) \rightarrow Q(t+1)$
0	$0 \rightarrow 0$
0	$1 \rightarrow 0$
1	$0 \rightarrow 1$
1	$1 \rightarrow 1$

Current State			Next State			Inputs for *DFF*		
a_2	a_1	a_0	a_2	a_1	a_0	d_2	d_1	d_0
0	1	1	1	1	0	1	1	0
1	1	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	1
0	0	1	1	1	1	1	1	1
0	0	1	0	1	1	0	1	1

Figure 17.18: A state table for a counter augmented with additional columns showing the necessary input values to *DFF*s so that outputs from the counter change as required when the active clock edge arrives.

Finally, we can draw a circuit that implements our required counter using *DFF*s; the circuit is shown in Figure 17.19. What is not shown in Figure 17.19 is that we should use the set and reset inputs on the *DFF*s in order the ensure that we can start the count sequence at 3.

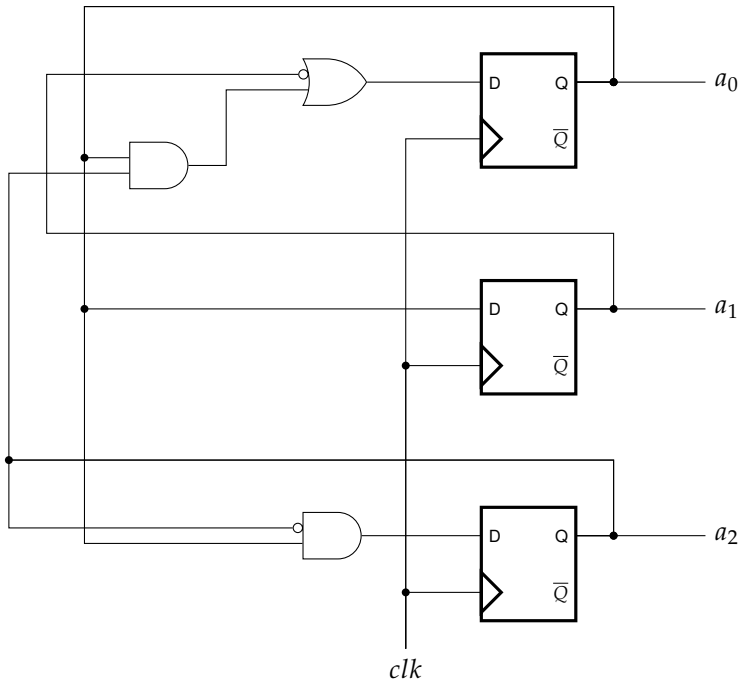


Figure 17.19: Circuit implementing our counter designed using *DFF*s.

17.8.2 *Generic counter design — implementation using TFFs*

Assume we want to use *TFF*... We can recall how a *TFF* works as shown in Figure 17.20.

T	$Q(t+1)$
0	$Q(t)$
1	$\overline{Q(t)}$

Figure 17.20: Behaviour of a *TFF* repeated here to remind ourselves how they work and how the outputs change based on the data input.

or

T	$Q(t) \rightarrow Q(t+1)$
0	$0 \rightarrow 0$
0	$1 \rightarrow 1$
1	$0 \rightarrow 1$
1	$1 \rightarrow 0$

We can augment the state table with additional columns that show the required inputs of each *DFF* so that each output bit will change appropriately. This is shown in Figure 17.21.

Current State			Next State			Inputs for *TFF*		
a_2	a_1	a_0	a_2	a_1	a_0	t_2	t_1	t_0
0	1	1	1	1	0	1	0	1
1	1	0	0	0	0	1	1	0
0	0	0	0	0	1	0	0	1
0	0	1	1	1	1	1	1	0
0	0	1	0	1	1	0	1	0

Figure 17.21: A state table for a counter augmented with additional columns showing the necessary input values to *TFF*s so that outputs from the counter change as required when the active clock edge arrives.

We can use the *TFF* inputs to write down equations for each of the *TFF* inputs... We could use Karnaugh maps to get these equations...

$$\begin{aligned} t_2 &= a_1 + a_0 \\ t_1 &= a_1 \oplus a_0 \\ t_0 &= a_2' a_0' + a_2' a_1 = a_2'(a_0' + a_1) \end{aligned}$$

The resulting circuit is shown in Figure 17.22.

What is not shown is that we should use the set and reset inputs on the *TFF*s in order the ensure that we can start the count sequence at 3.

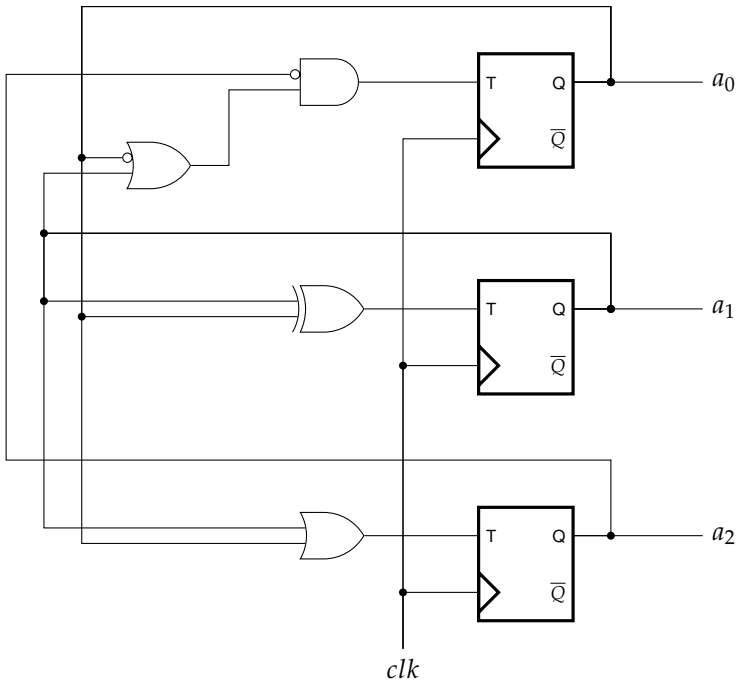


Figure 17.22: Circuit implementing our counter designed using *TFF*s.

17.8.3 *Generic counter design — implementation using JKFFs*

Assume we want to use *JKFF*... We can recall how a *JKFF* works as shown in Figure 17.8.3.

J	K	$Q(t+1)$
0	0	$Q(t)$
0	1	0
1	0	1
1	1	$\overline{Q(t)}$

or

J	K	$Q(t) \rightarrow Q(t+1)$
0	X	$0 \rightarrow 0$
X	0	$1 \rightarrow 1$
1	X	$0 \rightarrow 1$
X	1	$1 \rightarrow 0$

We can augment the state table with additional columns that show the required inputs of each *DFF* so that each output bit will change appropriately. This is shown in Figure 17.8.3.

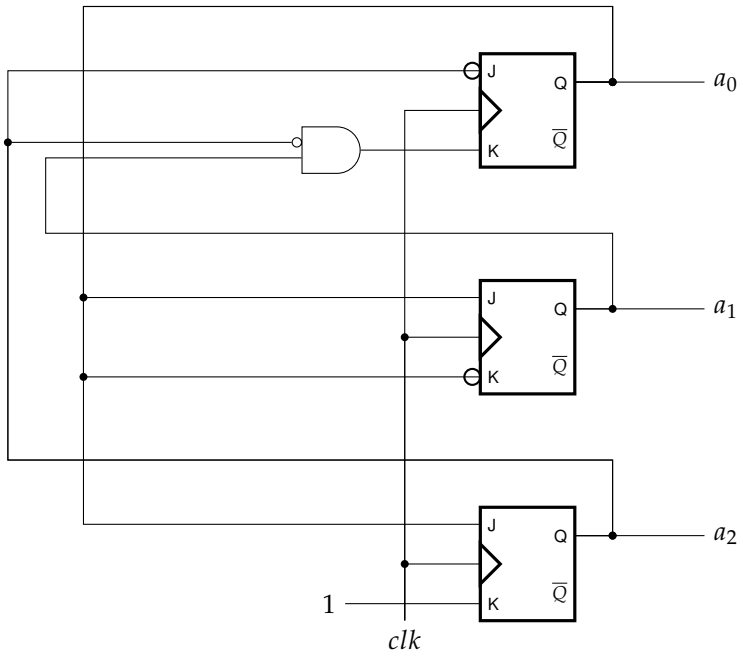
Current State			Next State			Inputs for *JKFF*		
a_2	a_1	a_0	a_2	a_1	a_0	j_2k_2	j_1k_1	j_0k_0
0	1	1	1	0	0	1X	X1	X1
1	0	0	1	1	1	X0	1X	1X
1	1	1	0	0	1	X1	X1	X0
0	0	1	0	0	0	0X	0X	X1
0	0	0	0	1	1	0X	1X	1X

Using the augmented state table, we can write down equations for each of the inputs to the *JKFF*s. These equations are

$$\begin{aligned} j_2 &= a_0 \\ k_2 &= 1 \\ j_1 &= a_0 \\ k_1 &= a_0' \\ j_0 &= a_2' \\ k_0 &= a_2' a_1 \end{aligned}$$

The resulting circuit implementing the required counter using *JKFF*s is shown in Figure 17.8.3

What is not shown is that we should use the set and reset inputs on the *JKFF*s in order the ensure that we can start the count sequence at 3.



17.8.4 *Generic counter design – summary*

The type of flip flop we select has an impact on the *complexity* of the circuit we end up with (that is, how much logic gates we need to produce the inputs to the flip flops). In this example, using DFF wasn't any more or less complex than using TFF. This might not always be the case! However, using $JKFF$ resulted in almost no extra logic gates. $JKFF$ are a bit more difficult to work with, but often result in less combinational logic — it's because $JKFF$ has a lot of don't care situations when considering the required inputs to cause the outputs to change as required.

18 Synchronous sequential circuits

Synchronous circuits are characterized by the presence of flip flops and clock signals. We've seen a few examples already — registers and counters are examples of synchronous, or clocked, sequential circuits. We will consider both the *analysis* and *design* of synchronous sequential circuits.

18.1 State diagrams

The *state diagram* is a useful idea for the pictorial representation of how a synchronous sequential circuit should operate to perform a task. The state diagram illustrates several things about a circuit:

1. The states of the circuit;
2. Transitions between states which depend on the circuit inputs;
3. Required circuit output values which depend on the state and, possibly, the circuit inputs.

The states of a sequential circuit are represented by *state bubbles* in the state diagram. The transitions from state to state are represented by *directed edges* or *arcs* in the state diagram. Circuit output values are labeled either inside of the state bubbles or on the arcs — it depends whether or not the outputs depend on only the state or on the state and the circuit inputs. The state diagram might also illustrate a *initial state* or *reset state* which is the desired start state for the circuit.

An example of a state diagram is shown in Figure 18.1. The state diagram in Figure 18.1 is for a design that requires 4 states denoted S_0, S_1, S_2 and S_4. The initial state of the design is S_0 denoted by the arc labeled "start" and pointing at S_0 from "nowhere". The circuit has one input X and one output Z — this is deduced by examining the edges where we see notation like $X = value/Z = value$.

When the circuit outputs depend on both the state and the circuit inputs, their values are labeled on the *arcs*. Such state diagrams are often called *Mealy state diagrams*.

Figure 18.2 shows another example of a state diagram. The

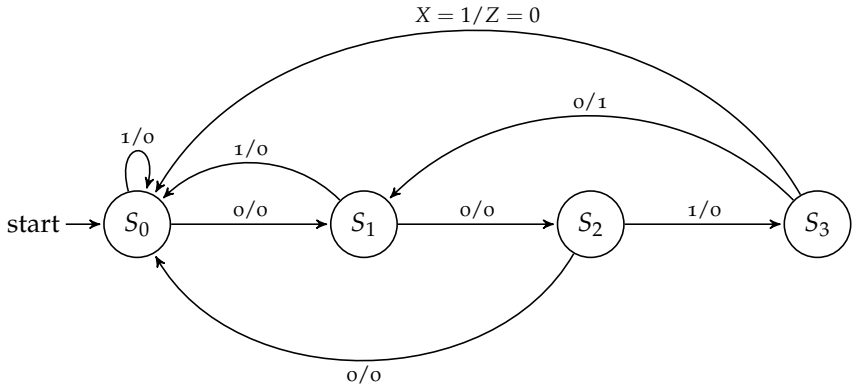


Figure 18.1: Example of a state diagram.

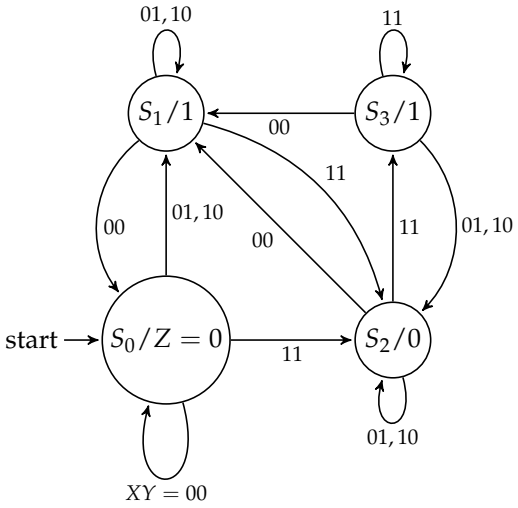


Figure 18.2: Another example of a state diagram. In this state diagram (compared to Figure 18.1) the outputs are only a function of the state.

state diagram in Figure 18.2 is for a design with requires 4 states: S_0, S_1, S_2 and S_4. Initial state is S_0 denoted by the arc labeled "start" and pointing at S_0. The state diagram has two input X and Y which deduced by looking at the arcs. No outputs labeled on the arcs! The state diagram has one output Z and this is deduced by looking *inside* state bubbles where we see notation like $state/Z = value$. Outputs labeled inside of a state bubble only depend on the current state. When the circuit outputs depend only on the state and **not** on circuit inputs their values are labeled inside the *state bubbles*. Such state diagrams are often called *Moore state diagrams*.

18.2 State tables

The *state table* describes the behaviour of a sequential circuit in tabular form. It shows the same information as a state diagram. A state diagram is shown in Figure 18.3 and its corresponding state table is shown in Figure 18.4.

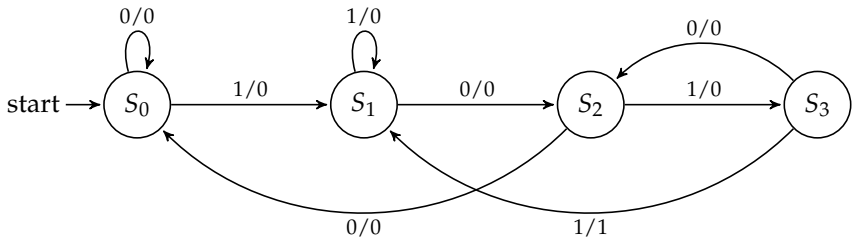


Figure 18.3: The state diagram of Figure 18.1 repeated.

Current State	**Next State**		**Output** Z	
	$X=0$	$X=1$	$X=0$	$X=1$
S_0	S_0	S_1	0	0
S_1	S_2	S_1	1	1
S_2	S_0	S_3	0	0
S_3	S_2	S_1	0	1

Figure 18.4: The state table for the state diagram in Figure 18.3. The state table shows exactly the same information as the state diagram, but in a tabular form.

From the state diagram there is clearly one input — we've called it X. There is clearly one output — we've called it Z. This state table shows states *symbolically*; e.g., they have names like S_0, S_1, and so forth.

18.3 Structure of a sequential circuit

Synchronous sequential circuits have a structure to them which is shown in Figure 18.5. The **current state** is represented as a binary

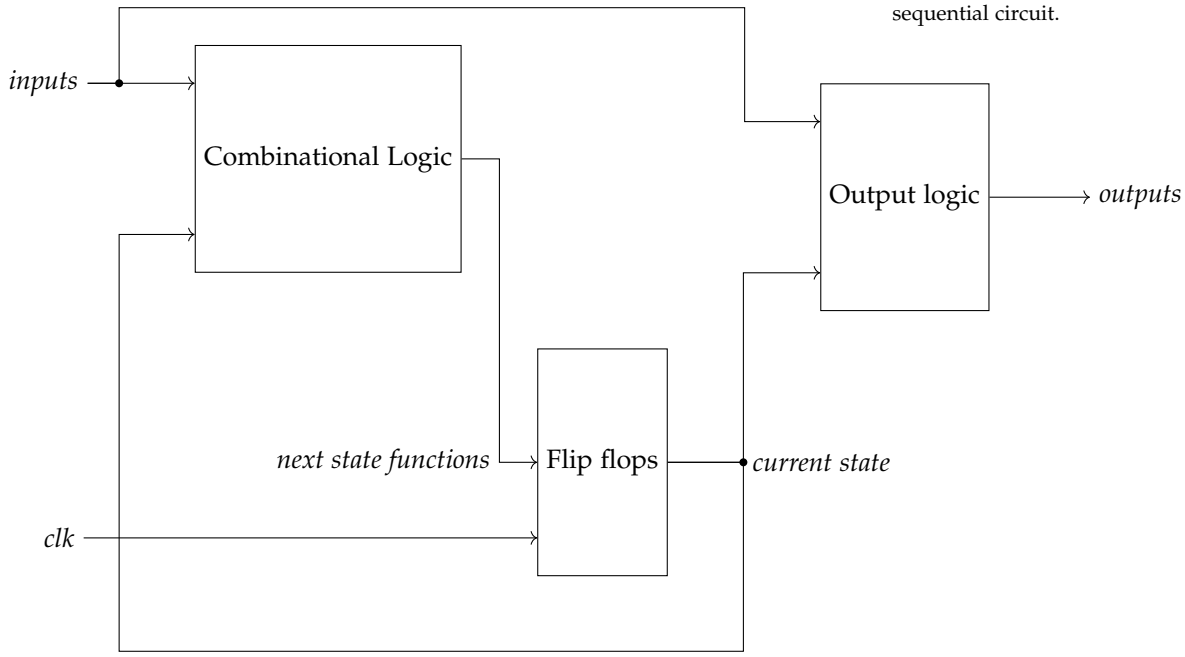


Figure 18.5: Structure of a synchronous sequential circuit.

pattern at the output of the flip flops. Based on the circuit inputs and the current state, we have **next state functions** (combinational logic gates) which are connected to flip flop inputs. These functions determine the **next state** upon the arrival of the active clock edge. We have output logic (combinational logic gates) which based on the state and (possibly) determine the circuit outputs.

18.4 *Sequential circuit analysis*

Circuit analysis implies figuring out what a circuit is doing. We might want to do this for a few reasons. Perhaps we are given a circuit and we simply want to know what it is doing. Another example of when we might want to do analysis is that we are given a circuit and we are told it performs a certain operation. Analysis would help us figure out if the circuit will operate correctly. Finally, we might want to "abstract out" the particular implementation of a circuit so that we could reimplement the same functionality in a different way. For a synchronous sequential circuit, analysis implies determining a symbolic state table or symbolic state diagram given a circuit drawing.

The analysis procedure involves the following steps:

- Identifying the flip flops. This tells us the potential number of states.

- Identifying the logic that produces the circuit outputs and writing down the output equations.
- Identifying the next state functions and writing down their logic equations. This tells us how the circuit transitions from state to state.
- Obtaining a state table and/or state diagram.
- Abstracting away anything which is particular to the implementation. For example, how the states have been encoded.

Again, our goal is to obtain the state table or state diagram since that's what specifies the behaviour of the circuit.

Consider the circuit shown in Figure 18.6. From the circuit in Figure 18.6, we can make some immediate observations. The circuit as two inputs a and b and one output z. The circuit has two flip flops which implies that — at most — this circuit is implementing a state diagram that has a maximum of 4 states. Because we are given a circuit and don't know the *description* of the states, we can know that there is a maximum of 4 states and are represented in the circuit using the binary values 00, 01, 10 and 11.

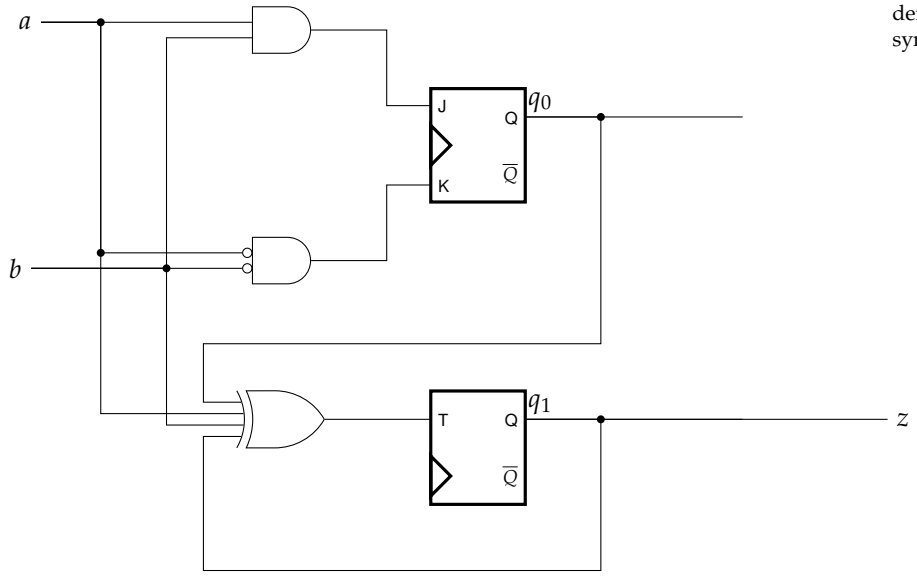


Figure 18.6: Example circuit used to demonstrate the analysis procedure for synchronous sequential circuits.

To analyze the circuit, we write logic equations for the circuit output in terms of the flip flop outputs (the *current state*) and the *circuit inputs*. This gives

$$z = q_1$$

Next, we write logic equations for the flip flop inputs (the *next state functions* in terms of the flip flop outputs (the *current state*) and the *circuit inputs*. This gives

$$\begin{aligned} j_0 &= ab \\ k_0 &= a'b' \end{aligned}$$

$$t_1 = a \oplus b \oplus q_0 \oplus q_1$$

We need these equations so we can figure out the transitions from one state to another state when the clock edge arrives.

Because the circuit is using flip flops, we should recall how the different flip flops work. This helps us determine how the flip flop outputs will change given the current flip flop outputs and the circuit inputs upon arrival of the active clock edge. This circuit uses a *TFF* and a *JKFF* — the behaviour of these flip flops is given in Figure 18.7.

J	K	$Q(t+1)$
0	0	$Q(t)$
0	0	0
0	0	1
1	1	$\overline{Q}(t)$

T	$Q(t+1)$
0	$Q(t)$
1	$\overline{Q}(t)$

Figure 18.7: Behaviour of the *JKFF* and *TFF* in case it has been forgotten.

Next, using the flip flop input equations and the behaviour of the different flip flops, we can write down a table showing what the flip flop inputs will be for each possible state and each possible setting of the circuit inputs. This is shown in Figure 18.8

current state	t_1			
q_1q_0	$ab=00$	$ab=01$	$ab=11$	$ab=10$
00	0	1	0	1
01	1	0	1	0
10	1	0	1	0
11	0	1	0	1

current state	j_0k_0			
q_1q_0	$ab=00$	$ab=01$	$ab=11$	$ab=10$
00	01	00	10	00
01	01	00	10	00
10	01	00	10	00
11	01	00	10	00

Figure 18.8: Flip flop input values for the circuit in Figure 18.6 for all possible states and input values.

Finally, based on the information computed in Figure 18.8, we can determine how the flip flop outputs will change upon the arrival of the active clock edge. This is shown in Figure 18.9. Note that

the table in Figure 18.9 is almost a complete state table — the only thing missing is the value of the circuit output. The table from

current state	*next state* (q_1q_0)			
q_1q_0	$ab = 00$	$ab = 01$	$ab = 11$	$ab = 10$
00	00	10	01	10
01	10	01	11	01
10	00	10	01	10
11	10	01	11	01

Figure 18.9: The next state derived from the current state and the circuit inputs. This table was found using the information about the flip flop inputs in Figure 18.8.

Figure 18.9 is repeated in Figure 18.10, but with the circuit output shown. The table in Figure 18.10 is the complete state table. The

current state	*next state* (q_1q_0)				*outputs* (z)			
q_1q_0	$ab = 00$	$ab = 01$	$ab = 11$	$ab = 10$	$ab = 00$	$ab = 01$	$ab = 11$	$ab = 10$
00	00	10	01	10	0	0	0	0
01	10	01	11	01	0	0	0	0
10	00	10	01	10	1	1	1	1
11	10	01	11	01	1	1	1	1

Figure 18.10: Complete state table for the analysis example. The complete state table shows the next state and the circuit outputs for all possible current state and input value combinations.

final state diagram is shown in Figure 18.12 and simply shows the information from Figure 18.10 in a more pictorial way. Notice that an initial state is not identified — this is because, for example, the original circuit in Figure 18.6 did not have any *reset* signal connected to the flip flops to set and initial state. Therefore, it is unclear what the initial state should be.

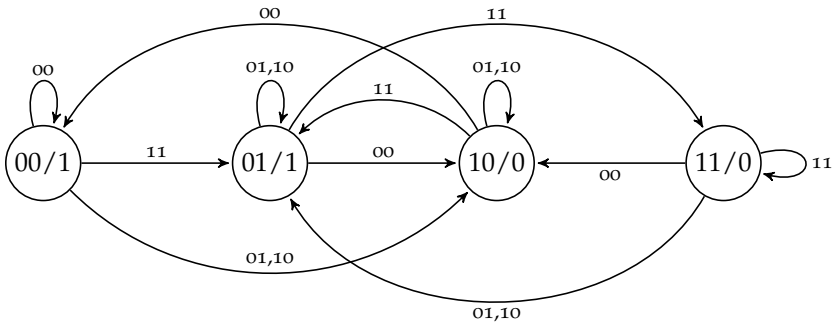


Figure 18.11: State diagram for the analysis example.

One last step we might consider is to "abstract out" the actual binary patterns used to represent the states. For example, rather than having state 00, we could call it S_0 and so forth. If you knew more about what this circuit was suppose to implement, you might even be

able to assign more meaningful names to the states such as *start state*, *stop state* and so forth. The abstracted state table and state diagram are shown in Figure 18.13 and Figure **??** respectively assuming state names of s_0, s_1, s_2 and s_4.

current state	*next state* (q_1q_0)				*outputs* (z)			
q_1q_0	$ab = 00$	$ab = 01$	$ab = 11$	$ab = 10$	$ab = 00$	$ab = 01$	$ab = 11$	$ab = 10$
s_0	s_0	s_2	s_1	s_2	0	0	0	0
s_1	s_2	s_1	s_3	s_1	0	0	0	0
s_2	s_0	s_2	s_1	s_2	1	1	1	1
s_3	s_2	s_1	s_3	s_1	1	1	1	1

Figure 18.12: The state table with the binary values assigned to represent the states "abstracted out".

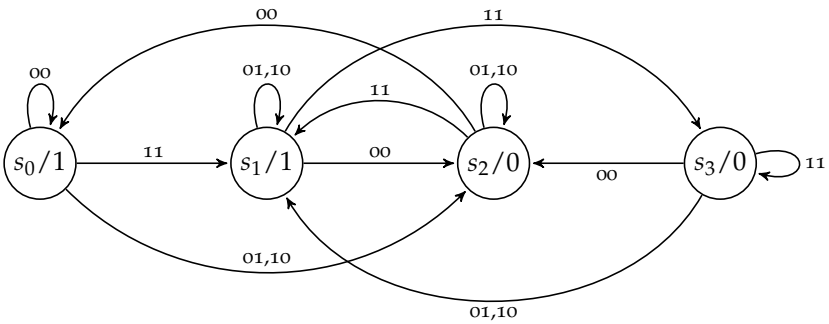


Figure 18.13: The state diagram with the binary values assigned to represent the states "abstracted out".

18.5 *Sequential circuit design*

Circuit design implies figuring out from a verbal problem description how to implement a circuit to accomplish what is described. The procedure involves a few steps.

1. Understand the verbal problem description.
2. Figure out what states are required and how we transition from state to state based on circuit inputs. Figure out what the circuit outputs need to be — this will lead to a state diagram and/or a state table. Note that things might be *symbolic* at this point; the states might have *names* rather than binary numbers to represent them.
3. Perform *state reduction*. It could be that our state diagram and/or state table, while valid and correct, uses more states than what is required. Often we can find *equivalencies* between states and get a smaller state diagram and/or state table that will also work. This could lead to a smaller circuit.

4. Perform *state assignment*. This means giving each state a binary number to represent it.
5. Select a flip flop in order to store the state information. The number of flip flops depends on how many bits are required to represent the states.
6. Derive output logic equations and next state (flip flop input) equations.
7. Draw the resulting circuit.

We have already seen an example of synchronous sequential circuit design when we talked about generic counter design. Recall for generic counter design that:

1. We required a number of states equal to the values that our counter needed to produce; (e.g., count $1 \rightarrow 7 \rightarrow 3 \rightarrow 2 \rightarrow 5 \rightarrow 6$ and repeat requires 6 states).
2. The circuit outputs are the binary values of the count sequence;
3. An active clock edge was all that was required to move from one state to the next state;
4. Each state was given a name; (e.g., we have state "1", "7", "3", and so forth);
5. Each state was "assigned" or represented in binary using the binary representation of the number. Here, we'd need 3 bits and the states would be represented as $1 \rightarrow 001$, $7 \rightarrow 111$, $3 \rightarrow 011$, and so forth.
6. We used all different types of flip flops;
7. The outputs were equal to the state for a counter.

Compared to counter design, however, we can implement more complicated types of circuits.

18.5.1 *Sequence detector example*

Sequence detectors are a good sort of circuit to consider when trying to learn sequential circuit design. We will do a few examples. Consider the verbal problem description as follows:

> Design a circuit that has one input x and one output z. The input x is assumed to change only between active clock edges. When any input sequence 101 is detected, the circuit should output a 1, otherwise 0. A sample input and output sequence might be as follows:
>
> x : 00110101000100010100...
> z : 00000101000100000100...

A useful way to design such circuits is to start with an initial state where no input has been received. Then, we can start adding states to follow the sequence which requires a 1 as output. Finally, we can fill in any remaining edges (and add additional states if required). Note that at this point, we don't know how many states are required, how many flip flops we will need, and so forth.

We will start our circuit in state S_0. Since we are looking for the pattern 101, we can remain in this state until we at least see a 1. Therefore, this state represents seeing a sequence of either nothing or a sequence of 0s which we can denote as ...0. When an input of 1 is received, we move to a new state S_1 which represents string of at least one 1 or ...1. This state also represents the start of a new pattern. If in state S_1 and we receive a 0, we have seen the first 2 bits of our pattern which we can denote as ...10 and we can move to another state s_2 to represent the fact we have reached ...01. Now, once in state S_2 if we see a 1 we can output a 1. This is the first part of the state diagram (the detection of the sequence) as is shown in Figure 18.14. It is clear from Figure 18.14 that we have more to do because there are clearly edges missing from the diagram as well as an edge leaving S_2 for which we haven't figured out its end point yet.

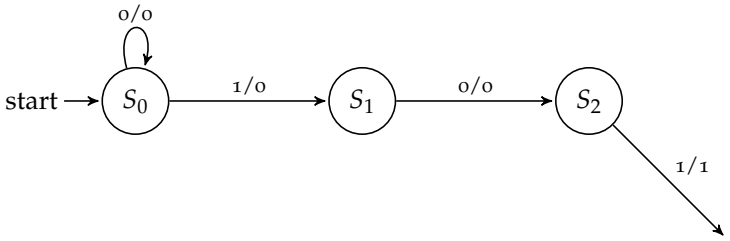


Figure 18.14: A partial state diagram for a sequence detector example.

We can now fill in the missing edges as well as figure out if we need any more states. If in S_1, we have already ...1. If we see additional 1s we can remain in S_1. If in S_2 and we receive a 0, we have seen ...100 and this breaks our pattern. We can return to S_0 and record that we have seen ...0. Finally, if in S_2 and we receive a 1, we output a 1 and can return to S_1 since this 1 not only represents the end of our pattern, but the potential start of the next occurance of the pattern. The complete state diagram is shown in Figure 18.15.

In the state diagram in Figure 18.15 we have chosen to implement the circuit as a Mealy state diagram. This is perfectly fine, but we should make note of when the output is *valid* to avoid reading *false* outputs. The output is valid when we are in S_2 and after the output has changed to a 1, but the next clock edge has not yet arrived.

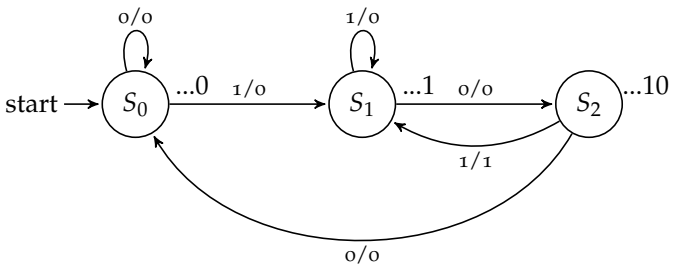


Figure 18.15: Complete state diagram for our sequence detector example.

This is illustrated in Figure 18.16 assuming the positive clock edge is the active clock edge. Timing with Mealy state diagrams can be somewhat problematic.

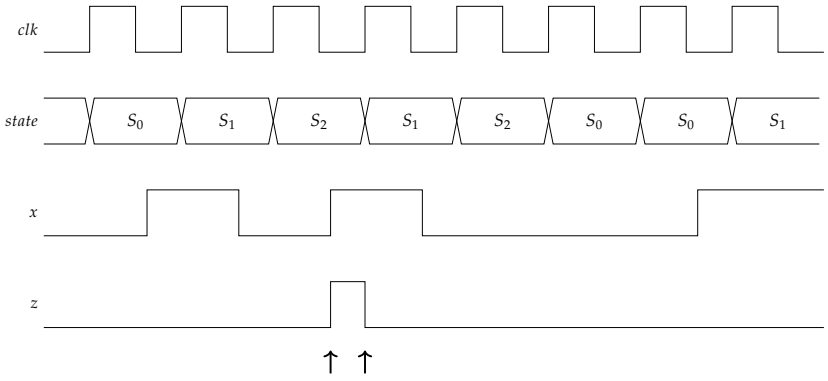


Figure 18.16: Timing in a Mealy state diagram. We need to be careful to read the circuit output(s) at the correct time to avoid reading *false outputs*.

The state diagram in Figure 18.15 leads to the state table in Figure 18.17.

Current State	**Next State**		**Output (z)**	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
S_0	S_0	S_1	0	0
S_1	S_2	S_1	0	0
S_2	S_0	S_1	0	1

Figure 18.17: State table for the state diagram in Figure 18.15.

Given the state diagram and/or the state table, the next step in the design procedure would normally be to try and reduce the number of required states — it could be the case that although our state diagram (or table) implements the desired functionality as specified, that we might have used more states than required. Obviously, more states will potentially result in a larger and potentially more complicated circuit. However, this problem is pretty simple, it's the first example and the problem doesn't actually permit any state reduction.

So, for now we will just continue and skip the state reduction. We will consider state reduction later.

Next, we need to do *state assignment*. This means assigning a binary pattern to represent each state. In this example, we have three states and therefore need *at least* enough bits to represent three different values. Of course, we can do this with 2 bits. We can choose S_0 is 00, S_1 is 01 and S_2 is 10. Note that 11 is unused. The state table with binary values replacing the symbolic states is shown in Figure 18.18. Note that the current state is held at the output of flip flops. Therefore, we will require 2 flip flops since we are using 2 bits for the state representation. Call these bits q_1 and q_0 and this is exactly what has been done in Figure 18.18.

Current State (q_1q_0)	**Next State (q_1q_0)**		**Output (z)**	
	$x=0$	$x=1$	$x=0$	$x=1$
00	00	01	0	0
01	10	01	0	0
10	00	01	0	1

Figure 18.18: State table for our first synchronous sequential circuit design when state assignment performed.

We are now at the point where we need to consider the actual implementation of a circuit and this requires selection of a flip flop type. We can use *DFF*, *TFF* or *JKFF* or even some combination of different types of flip flops. We don't know in advance which type of flip flop will be best. Further, because it is our first example, we will consider each type of flip flop. Given a that a type of flip flop has been selected, we need to figure out the necessary flip flop inputs in order to get the desired changes in state as a function of the current state and the circuit inputs. If there are *unused current states* (potential binary patterns which we did not need to use), then these result in don't cares for the flip flop inputs. In this example, there is no state for pattern 11 and therefore if the current state was somehow 11, the flip flop inputs are all don't cares.

18.5.2 *Sequence detector example — implementation with DFF*

The necessary *DFF* inputs to obtain the desired state transitions are provided in Figure 18.19. This leads to the flip flop input

Current State (q_1q_0)	**Next State (q_1q_0)**		**DFF inputs (d_1d_0)**	
	$x=0$	$x=1$	$x=0$	$x=1$
00	00	01	00	01
01	10	01	10	01
10	00	01	00	01

Figure 18.19: Flip flop input values required for our sequence detector to obtain the required state transitions assuming the current state is held at the outputs of *DFF*s.

equations $d_1 = \bar{x}q_0$ and $d_0 = x$ (these equations are available using

a Karnaugh map). We can also find (using a Karnaugh map) that the output $z = q_1 x$.

18.5.3 *Sequence detector example — implementation with TFFs*

The necessary *TFF* inputs to obtain the desired state transitions are provided in Figure 18.20. This leads to the flip flop input equations $t_1 = q_1 + \overline{x} q_0$ and $t_0 = \overline{x} q_0 + x \overline{q}_0 = x \oplus q_0$ (these equations are available using a Karnaugh map). We can also find (using a Karnaugh map) that the output $z = q_1 x$. Compared to the *DFF*, both t_1 and t_0 are more complicated than d_1 and d_0 so, thus far, *DFF*s are a better choice.

Current State ($q_1 q_0$)	**Next State** ($q_1 q_0$)		**TFF inputs** ($t_1 t_0$)	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
00	00	01	00	01
01	10	01	11	00
10	00	01	10	11

Figure 18.20: Flip flop input values required for our sequence detector to obtain the required state transitions assuming the current state is held at the outputs of *TFF*s.

18.5.4 *Sequence detector example — implementation with JKFFs*

The necessary *JKFF* inputs to obtain the desired state transitions are provided in Figure 18.21. This leads to the flip flop input equations $j_1 = \overline{x} q_0$, $k_1 = 1$, $k_0 = x$ and $j_0 = \overline{x}$ (equations from Karnaugh maps). We can also find (using a Karnaugh map) that the output $z = q_1 x$. Compared to the *DFF*, the circuit using *JKFF* also looks more complicated so, thus far, *DFF* are a better choice.

The final circuit using *DFF*s is shown in Figure 18.22.

Current State ($q_1 q_0$)	**Next State** ($q_1 q_0$)		**JKFF inputs** ($j_1 k_1$)		**JKFF inputs** ($j_0 k_0$)	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$	$x = 0$	$x = 1$
00	00	01	$0X$	$0X$	$0X$	$1X$
01	10	01	$1X$	$0X$	$X1$	$X0$
10	00	01	$X1$	$X1$	$0X$	$1X$

Figure 18.21: Flip flop input values required for our sequence detector to obtain the required state transitions assuming the current state is held at the outputs of *JKFF*s.

18.6 *Sequence detector example — implementation as a Moore state diagram*

What if implemented as a Moore state diagram? Recall that in a Moore state diagram, the circuit outputs can only be a function of the current state. The resulting state diagram is shown in Figure 18.23.

Notice that the Moore state diagram requires one additional state

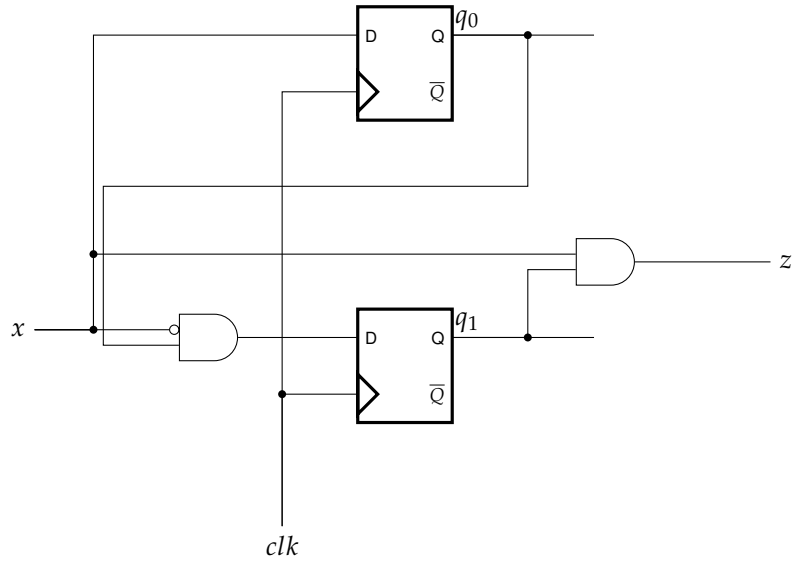


Figure 18.22: Final circuit implementation for our sequence detector design example. This circuit uses *DFF*s which proved to be the best choice compared to *TFF*s or *JKFF*s.

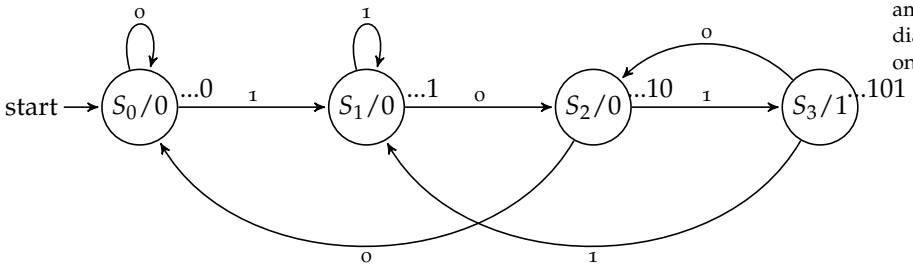


Figure 18.23: Sequence detector example implemented as a Moore state diagram in which outputs depend only on the current state.

compared to the Mealy state diagram because we need an additional state, S_3 to represent the detection of the entire pattern.

The timing diagram showing when the output is valid in a Moore state diagram is shown in Figure 18.24. Notice that the output is

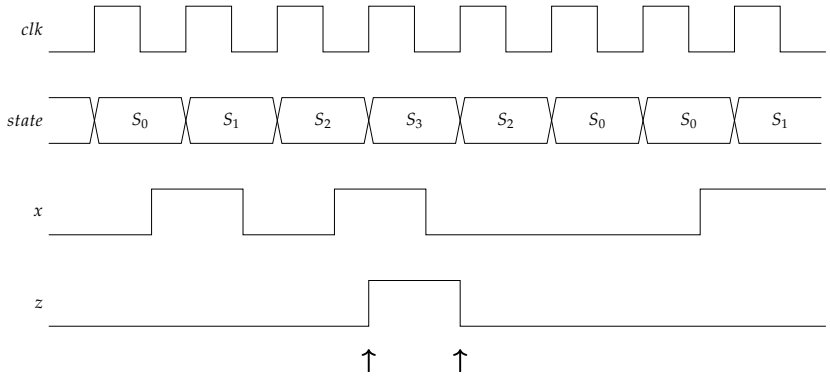


Figure 18.24: Diagram demonstrating when outputs are valid for a Moore state diagram. Outputs are valid for the entire duration of a state and therefore somewhat less problematic compared to outputs in a Mealy state diagram.

valid for as long as we remain in state S_3. Further, the output is valid during entire clock periods which occur between active clock edges. Moore state diagrams typically require more states compared to Mealy state diagrams, but the timing and validity of circuit outputs is easier to understand and therefore less problematic.

18.6.1 *Another sequence detection example*

We can consider one more example to demonstrate and practice figuring out how to draw a state diagram. Here, we will only figure out the state diagram — the other steps required to obtain an actual circuit are left up to you.

Consider the following verbal problem description:

> Design a Mealy state diagram for a circuit that has one input x and one output z. The input x is assumed to change only between active clock edges. The circuit output z should be 1 when exactly two of the last three inputs are 1.

In this case, it is useful to think of two things. First, we need to account for the start of the sequence when we have not yet received at least 3 bits. We can have a state "no input", we can have a state "first 0" and a state "first 1". Then, we can have further states, but we will define these states as the "last two bits are...". This leads us to draw the partial state diagram shown in Figure 18.25. It should now be clear that, based on the next input, we will simply move between S_3 to S_6 generating either $z = 1$ or $z = 0$ depending on the next input. Filling in the remaining edges yields the complete state diagram in

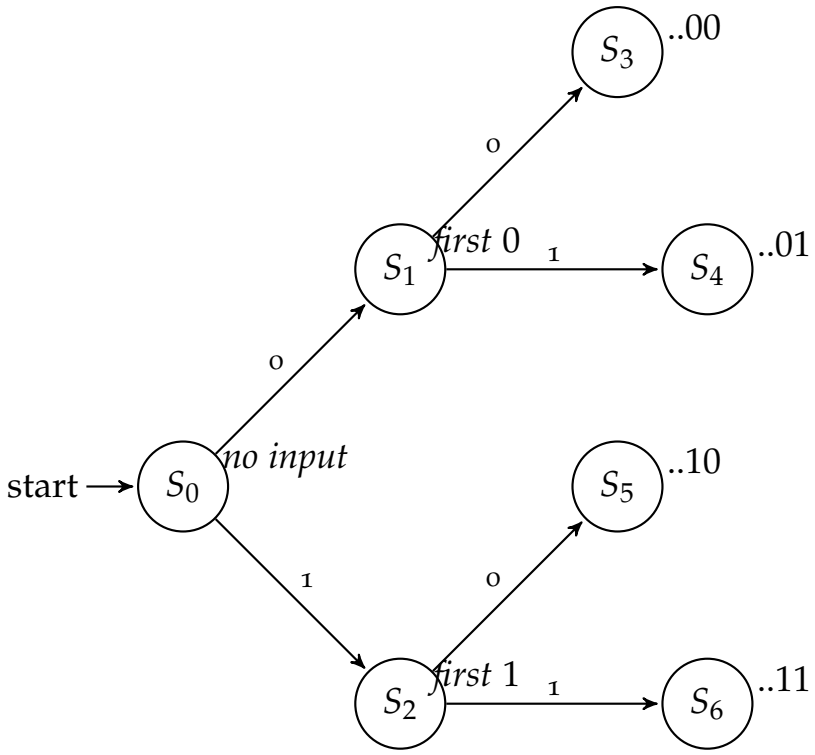


Figure 18.25: Partial state diagram for a second design example.

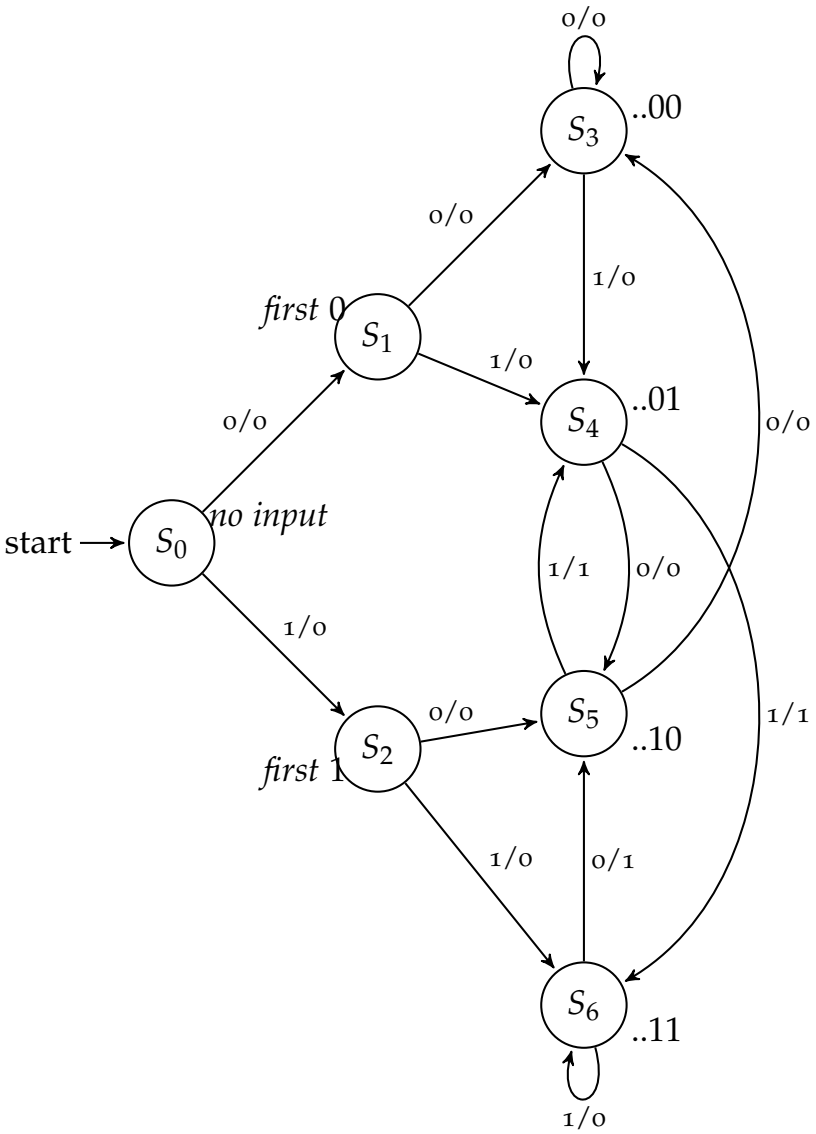


Figure 18.26: Complete state diagram for our second design example.

Figure 18.26. It should now be clear that, based on the next input, we will simply move between S_3 to S_6 generating either $z = 1$ or $z = 0$ depending on the next input. As previously mentioned, we could continue with this example if we wanted to obtain an actual circuit implementation.

18.7 State reduction

Sometimes when we derive a state diagram or a state table we might end up with more states than are really required. Since the number of flip flops and the complexity of the circuit is largely dependent on the number of states, we should always ask ourselves if there is a solution to our problem that requires *fewer states*. The best way to reduce the number of states is through *state reduction* which helps us identify whether or not states are *equivalent* to each other — equivalent states can be combine together into a single state.

A pair of states is said to be equivalent if:

1. For every possible circuit input combination, the states give exactly the same circuit output; and
2. For every possible circuit input combination, the states transition to the same *next state* or the same *equivalent state*.

In effect, what these conditions mean is that, for a pair of states which are equivalent, they will produce the same outputs (all the time) and will transition to the same set of next states (all the time) for all possible input combinations — the states "look" and "behave" the same so they can be merged into a single state.

State reduction can be accomplished using a *implication chart* and a *merger diagram*. The implication chart helps us identify whether or not pairs of states are equivalent. The merger diagram helps us decide and check whether or not a particular merging of states is correct. Consider the state diagram in Figure 18.27 which shows

Current State	**Next State**		**Output (z)**	
	$a = 0$	$a = 1$	$a = 0$	$a = 1$
s_0	s_3	s_2	1	1
s_1	s_0	s_4	0	0
s_2	s_3	s_0	1	1
s_3	s_1	s_3	0	0
s_4	s_2	s_1	0	0

Figure 18.27: Example state table used to explain state reduction by finding equivalent states.

a state table for a problem that requires one input a, produces one output z and required 5 states. With 5 states, we would require a minimum of 3 flip flops.

To perform state reduction, we start by creating an implication chart. The implication chart will tell us: which states are *definitely not equivalent*, which states are *definitely equivalent* and which states are *equivalent under conditions*. The implication chart looks like the lower triangle of a matrix. Each entry (i, j) tells us about the equivalency of state i and state j. Figure 18.28 shows a blank (empty) implication chart for our state table in Figure 18.28 We need to fill in the

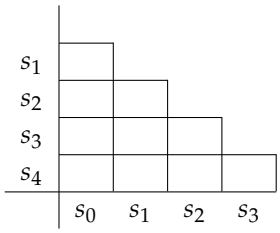


Figure 18.28: Empty implication chart for the state table in Figure 18.27.

implication chart appropriately.

The first step in completing the implication chart is to pick pairs of states and compare their outputs *for every possible input combination*. If, for *any* input, the outputs are different, the states *cannot* be equivalent — this makes sense because to be equivalent, the outputs must be the same for every input combination. If not equivalent, we mark the appropriate entry in the implication chart with an X.

For example, if we compare the outputs for s_0 and s_1, we see that they must produce different output values when $a = 0$ (and also when $a = 1$). Therefore, this pair of states cannot be equivalent. If we compare states s_0 and s_2, we see that the output values are the same for $a = 0$ and $a = 1$ (in other words, for all input combinations). We cannot yet claim s_0 and s_2 are equivalent (we haven't checked the next state conditions), but they might be. Comparing outputs only allows us to determine when states cannot be equivalent.

The updated implication chart showing states which are *not* equivalent due to their output values is shown in Figure 18.29.

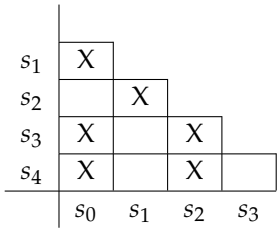


Figure 18.29: Updated implication chart for the state table in Figure 18.27. The pairs of states marked "X" cannot be equivalent due to having different output values for at least one input value.

The next step is to mark entries that are *definitely* equivalent because they go to same next state under every input combination. Note that if comparing two states i and j... If i goes to j and j goes to i under some input condition, this is obviously okay since we are

comparing i and j. We don't need to consider states that have previously been marked as not equivalent.

For example, if we compare the outputs for s_0 and s_2, we see that when $a = 0$, they both transition to s_3. When $a = 1$, s_0 goes to s_2 and s_2 goes to s_0. Therefore, states s_0 and s_2 are definitely equivalent.

The updated implication chart is shown in Figure 18.30.

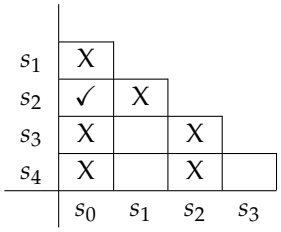


Figure 18.30: Implication chart for the state table in Figure 18.27 showing equivalencies due to transitions to the same next state under every input condition.

Next, we should mark entries that have *conditions* for equivalency. A condition is along the lines of "i and j are equivalent if k and l are equivalent".

For example, if we compare s_1 and s_4, we see that when $a = 0$, s_1 transitions to s_0 and s_4 transitions to s_2. When $a = 1$, s_0 transitions to s_4 and s_4 transitions to s_1. Therefore, s_1 and s_4 are equivalent **assuming** that s_0 and s_2 are equivalent (which we may or may not know yet).

The updated implication chart is shown in Figure 18.31.

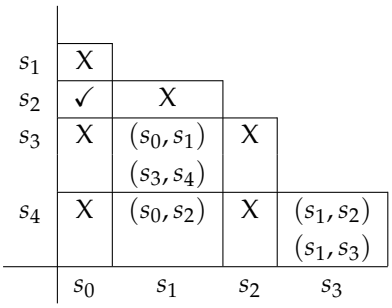


Figure 18.31: Implication chart for the state table in Figure 18.27 showing equivalencies subject to other pairs of states begin equivalent.

Finally, we must review the implication chart and attempt to decide whether or not conditions are true or false. When any condition fails, the the associated states cannot be equivalent. We might need to make more that one pass through the chart.

For example, s_0 and s_1 are not equivalent. Therefore, the condition (s_0, s_1) is false. This means that s_1 and s_3 cannot be equivalent — we don't even need to check the additional condition (s_3, s_4). Conversely, s_0 and s_2 are equivalent so the condition (s_0, s_2) is true. This means that s_1 and s_4 can be equivalent.

The final implication chart for the state table in Figure 18.27 is given in Figure 18.32.

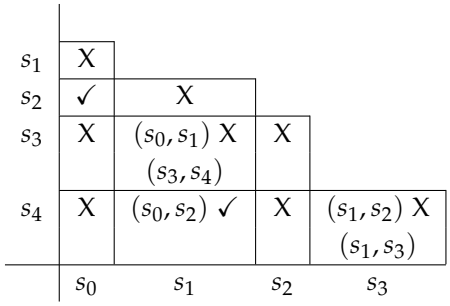


Figure 18.32: Final implication chart for the state table in Figure 18.27.

In summary, the implication chart for a state table or state diagram is generated by

1. Decide which states are definitely not equivalent due to differing output values;
2. Decide which states are definitely equivalent due to having the same outputs and the same next states (always);
3. Decide which states are equivalent subject to other states begin marked as equivalent. These become conditions for equivalence;
4. Check the conditions to decide whether or not conditions can be met. If not, then mark the states subject to the conditions as not equivalent.

Given an implication chart, we can proceed to draw a merger diagram. The purpose of the merger digram is to help us visualize which equivalent states should be merged into a single state. It also allows use to check any conditions (whether or not they are satisfied) if we merge states with conditions.

The merger diagram for the final implication chart in Figure 18.32 is shown in Figure 18.33. To decide what states to merge together,

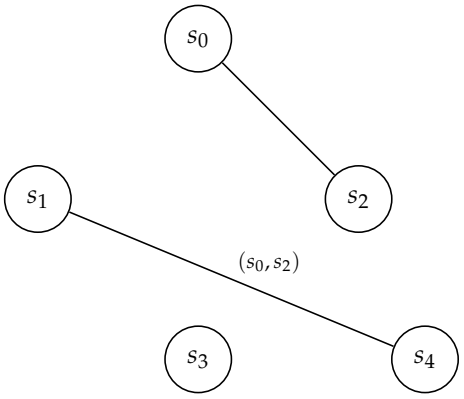


Figure 18.33: The merger diagram corresponding to the implication chart in Figure 18.32.

we look for *cliques* of states and then check any conditions to ensure

they are valid. A clique of states is a set of states in which every state has an edge connecting it to every other state in the set. For example, a clique of 3 states i, j and k would have an edge between i and j, i and k and j and k (a triangle).

In the merger diagram in Figure 18.33, there are two cliques: s_0 and s_2 as well as s_1 and s_4. However, the edge between s_1 and s_4 has a condition (s_0, s_2) which means s_1 and s_4 can only be merged if out solution also merges s_0 and s_2 — in this case, the condition is satisfied.

Therefore, we can merge s_0 and s_2 into a single state — we will use s_0 to represent the merged state. We can also merge s_1 and s_4 can be merged into a single state (because we've merged s_0 and s_2 thereby satisfying the necessary condition to merge s_1 and s_4 — we will use s_1 to represent the merged state.

This leads to the reduced state table in Figure 18.34 (which also shows the original state table from Figure 18.27 prior to state reduction. Whatever problem the original state table was designed to

Current State	**Next State**		**Output (z)**	
	$a=0$	$a=1$	$a=0$	$a=1$
s_0	s_3	s_2	1	1
s_1	s_0	s_4	0	0
s_2	s_3	s_0	1	1
s_3	s_1	s_3	0	0
s_4	s_2	s_1	0	0

$\rightarrow$

Current State	**Next State**		**Output (z)**	
	$a=0$	$a=1$	$a=0$	$a=1$
s_0	s_3	s_0	1	1
s_1	s_0	s_1	0	0
s_3	s_1	s_3	0	0

Figure 18.34: An original state table from Figure 18.27 and its reduced state table.

solve can actually be done with 3 states and not 5 — this reduction would require less flip flops and less additional logic and is therefore likely leads to a more efficient implementation.

18.8 *State assignment*

Recall that state assignment is when we assign binary values to represent symbolic states in a state diagram or state table. We need to do this prior to designing a circuit implementation. Since circuit outputs and next state functions depend on the current state, how we do state assignment can impact the complexity of our circuit. Further, since we are assigning binary patterns, there is nothing restricting us from using more flip flops than required. We will consider a few different ways to perform state assignment; each approach will have some *pros and cons*.

State assignment can be explained with an example. Consider the state diagram shown in Figure 18.35 for a problem that has 3

inputs R,B and W. The problem has 4 outputs A, B, C and D. Finally, the problem requires 7 states denoted as S_0 through S_6. Note that conditions on edges are shown as logic equations; if none of the conditions to leave a state are true, it's assumed that one remains in the current state. This is done to reduce the "clutter" in the state diagram. We will consider several ways to assign binary patterns to the states S_0 through S_6.

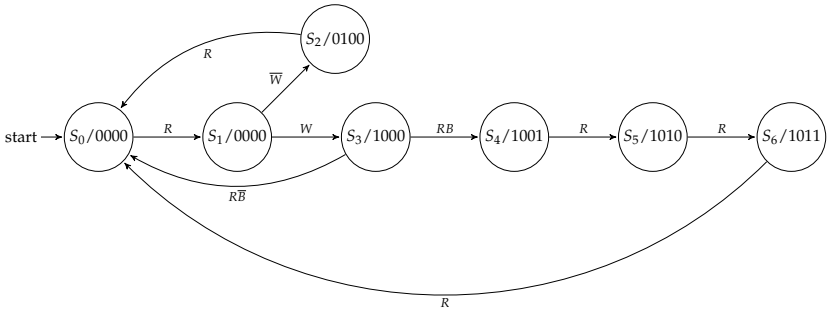


Figure 18.35: State diagram used to illustrate different ways to perform state assignment.

18.8.1 *State assignment — minimum flip flop count*

One way to perform state assignment is to try and use the *minimum number of flip flops*. For n states, we require *at least* $\lceil \log n \rceil$ bits (or flip flops), but no more than that.

In the problem from Figure 18.35, we have 7 states and therefore require $\lceil \log 7 \rceil = 3$ flip flops. With 3 flip flops, we have 8 different patterns available, although we only need 7 of them. The obvious assignment is to do the assignment sequentially which would give the state assignment $S_0 = 000$, $S_1 = 001$, $S_2 = 010$, $S_3 = 011$, $S_4 = 100$, $S_5 = 101$, and $S_6 = 110$. We could have assigned binary patterns differently (for example, we could have assigned $S_3 = 111$ instead of $S_3 = 011$). A different assignment of binary patterns to states will change the flip flop input equations and the output equations. Unfortunately, it is difficult to say which binary pattern should be assigned to represent each state to reduce the final complexity of the circuit. We can neither predict the complexity of the next state functions nor the circuit output equations. The only benefit and thing we know for sure is that this approach requires the minimum number of flip flops.

18.8.2 *State assignment — output encoding*

Sometimes we can try to encode the states such that the outputs are equal to the current state. This was exactly what we did when we were designing counters — the current count value was not only the output of the circuit, but also the current state of the circuit. The clear benefit of output encoding is that *there is no output logic required — the outputs of the circuit are connected directly to the flip flop outputs*. Unfortunately, we may not be able to achieve output encoding with a minimum number of flip flops.

For the example in Figure 18.35, we know the circuit can be implemented with as few as 3 flip flops. However, our circuit requires 4 output values. Hence, we must use more than the minimum number of flip flops, but how many flip flops we need must be carefully considered. Figure 18.36 shows the states for the state diagram from Figure 18.35 along with the required circuit output values. Since

Current state	**Output**			
	A	B	C	D
S_0	0	0	0	0
S_1	0	0	0	0
S_2	0	1	0	0
S_3	1	0	0	0
S_4	1	0	0	1
S_5	1	0	1	0
S_6	1	0	1	1

Figure 18.36: States and required circuit outputs for the state diagram in Figure 18.35.

we have 4 circuit outputs, we should consider whether or not we should implement the circuit using 4 flip flops and use the required circuit outputs to represent the states; if this was the case, then there is no output logic and the circuit outputs are equal to the current state. Since 4 flip flops is larger than the minimum required, it is certainly possible to implement the circuit with 4 flip flops. As tempting as this is, it will not work since it would require both S_0 and S_1 to be assigned the binary pattern 0000 according to Figure 18.36 — two different states cannot be represented with the same binary pattern.

We don't have to give up. What we need to achieve is the ability to get our circuit outputs by looking at the state *and ensuring that every state has a different binary pattern*. In this example, we can accomplish this by introducing a 5-th flop flop as shown in Figure 18.37. Notice that, in Figure 18.37, the additional flip flop is only required to differentiate between states S_1 and S_2; for other states its value is a dont' care. So, we could use the state assignment given by $S_0 = 00000$, $S_1 = 00001$, $S_2 = 01000$, $S_3 = 10000$, $S_4 = 10010$, $S_5 = 10010$, and $S_6 = 10011$. Again, this encoding is suitable because every

Current state	Output				Extra bits
	A	B	C	D	E
S_0	0	0	0	0	0
S_1	0	0	0	0	1
S_2	0	1	0	0	X
S_3	1	0	0	0	X
S_4	1	0	0	1	X
S_5	1	0	1	0	X
S_6	1	0	1	1	X

Figure 18.37: Use of an extra flip flop to allow for output encoding to be used during the implementation of the state diagram in Figure 18.35.

state is assigned a different pattern. This solution requires 5 flip flops which is 2 more than the minimum. Similar to the use of the minimum number of flip flops, we cannot predict the complexity of the next state (flip flop input) functions. However, *we can predict the complexity of the output logic — there isn't any*. The outputs for this circuit are taken directly from the first 4 flip flops. In other words, the output logic is simply wire.

18.8.3 *State assignment — one hot encoding*

One hot encoding is specifically intended for use with DFF. In one hot encoding, the output of **only one** output can be 1 or "hot" at any given time. In terms of the number of required flip flops, this condition clearly implies that we will need one flip flop for each state. This can result in a very large number of flip flops. In our example, we would require 7 flip flops. We would use the following state assignment: $S_0 = 0000001$, $S_1 = 0000010$, $S_2 = 0000100$, $S_3 = 0001000$, $S_4 = 0010000$, $S_5 = 0100000$, and $S_6 = 1000000$. Obviously, one hot encoding requires a lot of flip flops which is a potential disadvantage. Therefore, we should investigate further to determine the benefits offered by one hot encoding.

In a circuit implemented with one hot encoding, we can always tell which state we are in simply by looking to see if the DFF representing that state has an output of 1 which might be advantageous. Another advantage of one hot encoding (not necessarily apparent in this discussion) is that one hot encoding tends to lead to very simple expressions for the next state (flip flop input) functions. The output logic required to generate circuit outputs also tends to be very simple.

One hot encoding also makes deriving the next state functions and output functions very simple. Consider the state table in Figure 18.38 for a circuit with one input a, one output z and three states s_0, s_1 and s_2. We can use one hot encoding and do the assignment $s_0 = 001$, $s_1 = 010$ and $s_2 = 100$ which implies that we will need 3 DFFs in

Current State	Next State		Output (z)	
	$a=0$	$a=1$	$a=0$	$a=1$
s_0	s_2	s_0	1	1
s_1	s_0	s_1	0	0
s_2	s_1	s_2	1	0

Figure 18.38: Simple state table to further illustrate the use of one hot encoding.

order to implement the circuit. The state table using this state assignment is shown in Figure 18.39. When using DFF, the required

Current State	Next State		Output	
	$a=0$	$a=1$	$a=0$	$a=1$
$q_2q_1q_0$	$q_2q_1q_0$	$q_2q_1q_0$	z	z
001	100	001	1	1
010	001	010	0	0
100	010	100	1	0

Figure 18.39: State table from Figure 18.38 using one hot encoding.

flip flops inputs are equal to the next state. Therefore, we can examine Figure 18.39 to try to derive the next state functions. Normally, we would consider using Karnaugh maps to write down the logic equations. However, using one hot encoding makes it possible to *write down next state functions directly.* This is also true for the output functions. To write down the next state function for the i-th flip flop — which corresponds to state i, it is simply necessary to consider when the system will enter into state i. From the state table in Figure 18.39, it becomes clear that the conditions to enter into each state are given by the equations

$$\begin{aligned} d_0 &= \bar{a}q_1 + aq_0 \\ d_1 &= \bar{a}q_2 + aq_1 \\ d_2 &= \bar{a}q_0 + aq_2 \end{aligned}$$

and are the next state functions. Similarily, to write down the output functions it becomes necessary to consider when the output must be 1. This leads to the output equation given by

$$z = q_0 + \bar{a}z_2$$

So not only are the next state functions and output functions simple in form, they are also simple to write down directly from the state table. In fact, it was not even required to do the state assignment — we know that we need one DFF for each state and the DFF for state i is labeled with output q_i and its input is d_i.

You can also write the equations down directly if given a state diagram. Consider the state diagram in Figure 18.40 which happens to correspond to the state table in Figure 18.38. Just as with the state table, to write down the next state functions from the state diagram,

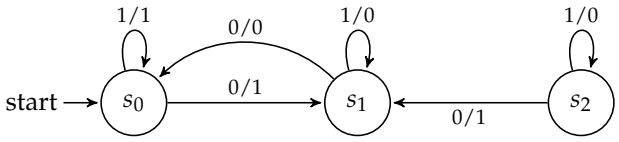


Figure 18.40: State digram for the state table in Figure 18.38.

you simply need to consider the conditions on when you enter into each state. The output functions are similarly written down by considering when each output function needs to be 1. Doing this with the state diagram leads to the same equations as perviously found. One important thing to note if trying to perform one hot encoding from the state diagram is to be careful that no information is *missing* in order to reduce the "clutter" in the state diagram. For example, sometimes in a state diagram, one might leave our certain edges such as self loops on individual states (as it might be *implied* that if there is no condition to leave a state is true, that you remain in the current state). However, if you ignore such implied information, you need to be sure that *all edges entering each state* are shown otherwise your equations will be missing necessary logic.

19 Algorithmic state machines

An algorithmic state machine (ASM) is simply an alternative to the state diagram which looks more like a flow chart. One finds three types of boxes in an ASM:

1. State boxes;
2. Decision boxes;
3. Conditional output boxes.

The state box is rectangular and is equivalent to the state bubble in a state diagram. A state box is illusrated in Figure 19.1. The box has the state name or its binary encoding listed above the Any outputs that are 1 and that depend only on the state are labeled inside of the box. The state box has one entry point and one exit point. Note

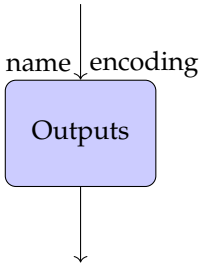


Figure 19.1: State box in an algorithmic state machine.

that a convention says that *outputs are only shown in the ASM when they are* 1.

The decision box is a diamond shaped box with one entry point and two exist points and is shown in Figure 19.2. The inside of the decision box is labeled with an arbitrary logic expression which evaluates to 0 or 1; the evaluation will cause one the exit points to be chosen.

The conditional output box is an ellipsoid shapes box and has one entry and one exit. An example is shown in Figure 19.3. It specifies an output that occurs when a transition takes place. Conditional output boxes are required if we have a circuit with Mealy outputs.

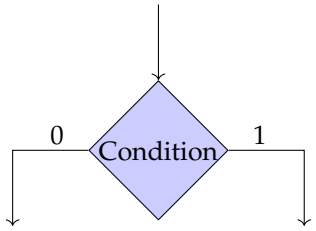


Figure 19.2: Decision box in an algorithmic state machine.

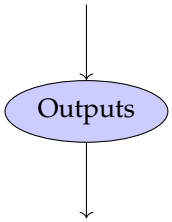


Figure 19.3: Conditional output box in an algorithmic state machine.

Once again, convention indicates that *outputs are only shown in the ASM when they are* 1.

Designing a circuit given an ASM is exactly the same as if one was given a state diagram. For example, the number of states equals the number of state boxes. The next state transitions are based on the state boxes and the decision boxes. Circuit outputs are based on state boxes and conditional output boxes. This will lead to a completed state table and the rest of the design is the same as before (do state assignment, find logic equations, etc.)

An example ASM is illustrated in Figure 19.4. The ASM in Figure 19.4 clearly has 3 states S_0, S_1 and S_2 since there are 3 state boxes. There are 3 inputs g, w and z since there are 3 decision boxes and these variables are labeled inside the decision boxes. Finally there is clearly 1 output a which is labeled inside of the state box S_0 and in a conditional output box. The output a is 1 when in state S_0 or in state S_1 and $w = 1$, otherwise 0.

The corresponding state table (written in a slightly different form due to the number of don't care situations in the decision boxes and the large number of inputs) is shown in Figure 19.5. Given the state table in Figure 19.5, design would continue as usual, including state assignment, flip flop selection, determination of next state functions and output functions, and so forth.

Since the ASM and a state diagram convey the same information, it is possible to convert from an ASM to a state diagram and vis versa. For example, the state diagram for the ASM diagram in Figure 19.4 is shown in Figure 19.6. Note that is state diagram is sort of a hybrid — sometimes output is shown inside the state bubble and sometimes on the edges.

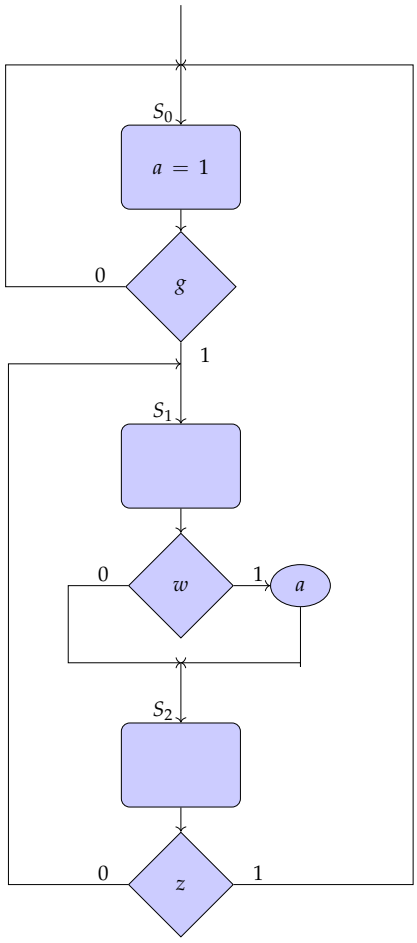


Figure 19.4: Example of an algorithmic state machine.

Current state	Input			Next state	Output
	g	w	z		a
S_0	0	X	X	S_0	1
S_0	1	X	X	S_1	1
S_1	X	0	X	S_2	0
S_1	X	1	X	S_2	1
S_2	X	X	0	S_1	0
S_2	X	X	1	S_0	0

Figure 19.5: State table derived from the ASM in Figure 19.4.

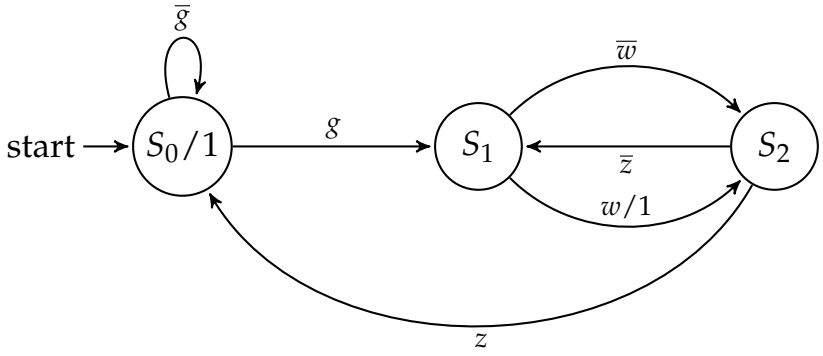


Figure 19.6: State diagram for the ASM shown in Figure 19.4.

19.0.4 *Algorithmic state machines — one hot encoding*

It is interesting and particularly easy to consider the use of one hot encoding when given an ASM. This is because every part of an ASM has a *corresponding circuit implementation*. This implies that given an ASM, it is possible to draw a circuit directly from the ASM.

Figure 19.7 shows the hardware equivalent to the state box — it is simply a *DFF*. Figure 19.8 shows the hardware equivalent to a

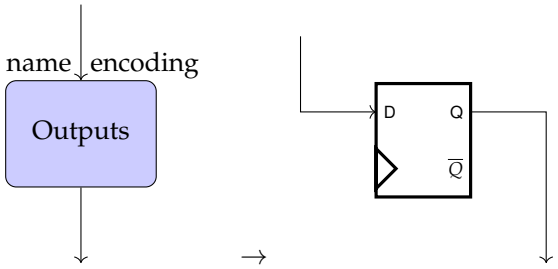


Figure 19.7: Hardware equivalent to a state box in an algorithmic state machine.

decision box. The hardware equivalent for a conditional output

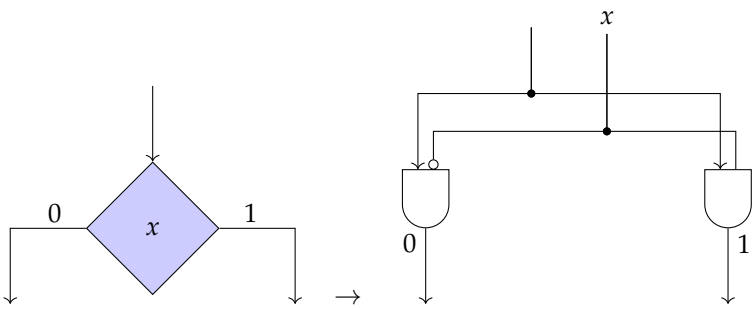


Figure 19.8: Hardware equivalent to a decision box in an algorithmic state machine.

box is shown in Figure 19.9. We simply "tap off" from the wire passing through the conditional output box.

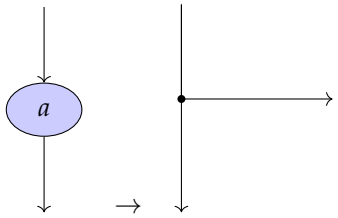


Figure 19.9: Hardware equivalent to a conditional output box in an algorithmic state machine.

Finally, there are situations in an ASM where different paths connect together. This also has a hardware equivalent which is shown in Figure 19.10 which indicates that places where paths join in an ASM are equivalent to an **OR** gate with a sufficient number of inputs.

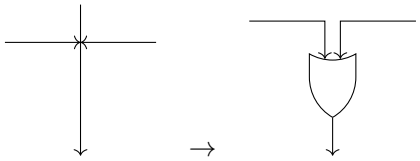


Figure 19.10: Hardware equivalent for joining different paths in an algorithmic state machine.

Given this knowledge, we can simply draw a circuit corresponding to an ASM. For the ASM in Figure 19.4, the resulting circuit is shown in Figure 19.11. Of course, you can look at the ASM and write down equations for the DFF inputs and circuit outputs too.

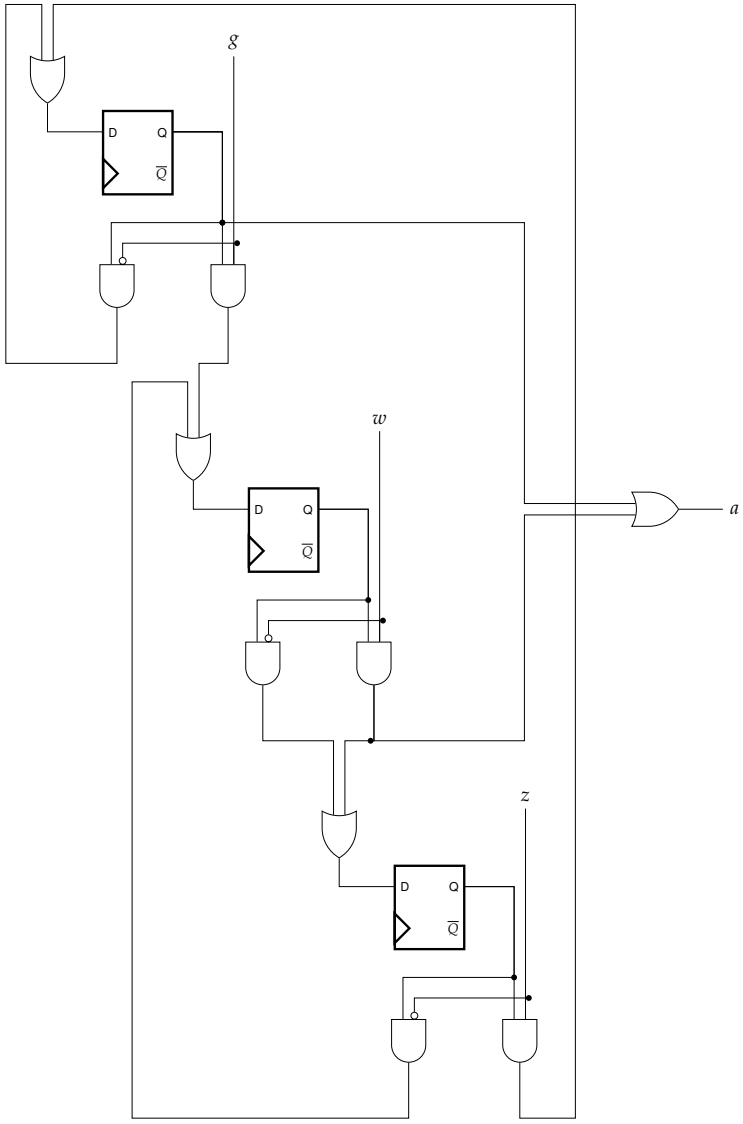


Figure 19.11: Circuit implementation of the ASM in Figure 19.4 drawn directly from the ASM using hardware equivalent circuits for each component of the ASM.

Part III

Asynchronous logic

20 *Asynchronous circuits*

We have seen combinational circuits. These are circuits which are logic functions which are written in terms of input variables and described by logic functions or truth tables. We have see clocked sequential circuits. These are circuits which involve combinational logic and flip flops. The behaviour of the circuit is controlled by a clock. There is another type of circuit that we might encounter, namely the *asynchronous circuit*. An asynchronous circuit can be defined as a circuit which exhibits the concept of *state* or *memory* much like a clocked sequential circuit. However, an asynchronous circuit *has no flip flops and no clock*. The concept of memory in an asynchronous circuit is a result of the combinational feedback paths (the loops) in the circuit and the delays involved in the circuit. In fact, asynchronous circuits were used prior to clocked sequential circuits. However, in many cases were replaced since their design and analysis is more difficult.

We've seen an example already. For example the SR latch shown in Figure 20.1 is an example of an asynchronous circuit. The SR latch which we already know has the concept of memory or state, but no clock. We need to figure out how to deal with such circuits including their analysis and their design. Asynchronous circuits are identified by the presence of latches and/or the presence of combinational feedback paths in a circuit diagram.

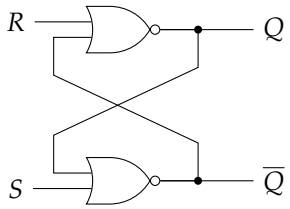


Figure 20.1: An SR latch as an example of an asynchronous circuit.

A conceptual block diagram for a asynchronous circuit to try and identify things we can talk about is shown in Figure 20.2. The next state variables Y are often called the excitation variables. The current state variables y are often called the secondary variables. Basically, the excitation and secondary variables are the values at

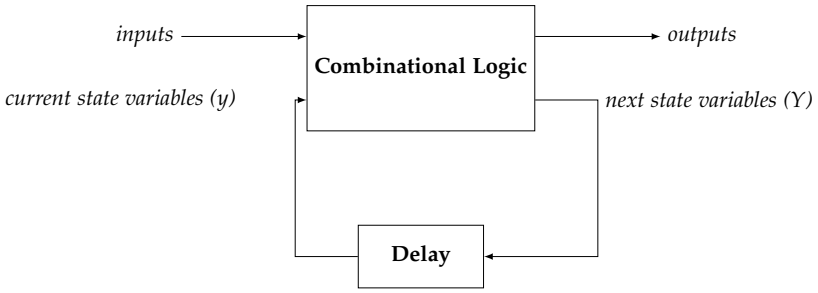


Figure 20.2: Conceptual block diagram of an asynchronous circuit which is useful to identify different sets of variables used during the design and analysis of an asychronous circuit.

opposite ends of the same wire (which loops back in the circuit). However, they are separated by a delay and we can think of them as separate variables (equal to each other after a certain amount of delay) if we conceptually break the feedback loops. With excitation and secondary variables we can describe asynchronous circuits using tables similar to state tables (with minor modifications).

20.1 *Asynchronous circuit analysis*

Analysis of an asynchronous circuit is similar to clocked sequential circuit analysis in that our objective is to derive tables that describe the behaviour of the circuit. The tables that we derive, however, are not called state tables although they still show transitions between states and circuit output values. Rather, we will derive **transition tables** and **flow tables**. The difference between a transition table and a flow table is that the transition table shows binary state assignments while the flow table is purely symbolic. For asynchronous circuits, when we derive these tables, we should indicate **stable states** and this is one difference between the transition table and flow table and the state tables we previously derived (the concept of a stable state was not really required for clocked sequential circuits). We require the definition of stability. For a given set of circuit input values, the circuit is **stable** if the circuit reaches **steady state** in which the excitation variables and secondary variables are equal to each other and unchanging, otherwise the circuit is unstable.

As an example of asynchronous circuit analysis and to figure out the procedure involved, consider the circuit shown in Figure 20.3.

In the circuit in Figure 20.3, there are no obvious latches present. However, there are certainly feedback paths in the circuit. We hypothetically break these paths in order to label the excitation and secondary variables. This circuit therefore has one input x, one output z, two excitation variables Y_2Y_1 and two secondary variables y_2y_1.

We can write down equations for the excitation variables. These

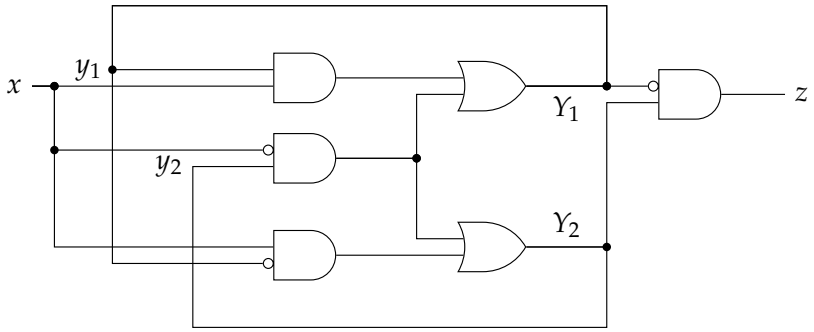


Figure 20.3: Sample circuit to illustrate the asynchronous analysis procedure.

equations should be written in terms of the circuit inputs and the secondary variables (compared to clocked sequential design, this is equivalent to writing the next state functions in terms of the current state and circuit input variables):

$$\begin{aligned} Y_2 &= xy_1 + x'y_2 \\ Y_1 &= xy_1' + x'y_2 \end{aligned}$$

We can also write an equation for the circuit output z. Again, we want to write this equation in terms of the secondary variables and circuit inputs (despite the fact that the circuit diagram makes it look like the output is a function of the excitation variables):

$$z = y_1'y_2$$

Using these equations we can determine the **transition table** for this circuit which is shown in Figure 20.4. Much like the state table, the **transition table** shows what the excitation variables (next state) will be given the secondary variables (current state) and the circuit inputs. It also shows the circuit outputs as a function of the secondary variables (our current state) and the circuit inputs.

Current state	**Next state** (Y_2Y_1)		**Outputs** (z)	
y_2y_1	$x=0$	$x=1$	$x=0$	$x=1$
00	[00]	10	0	0
01	00	[01]	0	0
11	[11]	01	0	0
10	11	[10]	1	1

Figure 20.4: The transition table derived for the circuit in Figure 20.2.

One significant difference between an asynchronous transition table and a state table is that in the transition table we **circle stable states**. For example, if we had $x = 0$ and $y_2y_1 = 00$, then we would find that $Y_2Y_1 = 00$ as well — the outputs of all gates in the circuit would be constant and not changing. However, if we changed $x = 0 \rightarrow 1$,

we would find that Y_2Y_1 are required to change from 00 to 10 which means that y_2y_1 will also change to 10 (after some amount of delay). According to the transition table, we would end up in another stable situation in which $x = 1$ and $y_2y_1 = 10$ (and $Y_2Y_1 = 10$).

If we know the circuit always reaches a stable state when an input variable is changed, we can also observe that the output really only needs to be specified when the circuit is in a stable state. Therefore, we *could* consider rewriting the transition table as shown in Figure 20.5. This indicates that when the circuit is unstable, the

Current state	**Next state** (Y_2Y_1)		**Outputs** (z)	
y_2y_1	$x = 0$	$x = 1$	$x = 0$	$x = 1$
00	$\boxed{00}$	10	0	–
01	00	$\boxed{01}$	–	0
11	$\boxed{11}$	01	0	–
10	11	$\boxed{10}$	–	1

Figure 20.5: The transition table derived for the circuit in Figure 20.2, but with the outputs marked as unspecified when the state is unstable.

output z is not relevant and can potentially change whenever it wants to just as long as it reaches a correct value upon entering another stable state. There could, however, be reasons why the value of the output z is specified in certain unstable situations such as to guarantee that output glitches do not occur.

Because we had two feedback paths, we had the potential for 4 states represented by the binary values 00, 01, 10 and 11. We can "undo" this binary assignment to get the flow table which is the symbolic version of the transition table. For our example, the flow table is shown in Figure 20.6 Note that stable states are still boxed

Current state	**Next state** (Y_2Y_1)		**Outputs** (z)	
y_2y_1	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	$\boxed{a}$	d	0	–
b	a	$\boxed{b}$	–	0
c	$\boxed{c}$	b	0	–
d	c	$\boxed{d}$	–	1

Figure 20.6: Flow table for our asynchronous analysis example.

in the flow table. If we wanted, we could also draw a state diagram.

The asynchronous analysis procedure is summarized as follows:

- Identifying the feedback paths, hypothetically breaking these paths (turning the circuit — at least hypothetically — into a combinational circuit);
- Labelling the excitation and secondary variables;
- Deriving equations for the excitation variables and output variables in terms of the circuit inputs and secondary variables;

- Deriving a transition table being careful to box the stable states;
- Deriving a flow table by undoing the state assignment from the transition table.

This, of course, assumes that we are given an asynchronous circuit in which we are unable to identify the presence of latches.

It might be that we can easily identify latches in a circuit and, further, that it is the latch outputs that feedback around in the circuit to the latch inputs — in some ways, this makes the circuit look much more like a clocked sequential circuit. An example of such as circuit is illustrated in Figure 20.7. This circuit has two input x_2x_1, two

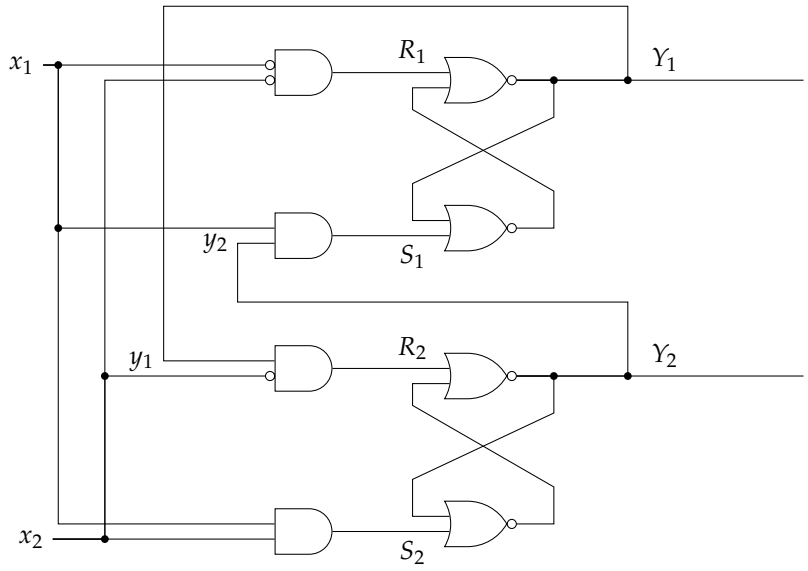


Figure 20.7: Example of an asychronous circuit in which we can clearly identify latches.

excitation variables Y_2Y_1 (latch outputs) and two secondary variables y_2y_1. Since we have identified two latches (in this case SR latches), we label the latch inputs.

Since latches are clearly identified, it is more straight forward to first write down the logic equations for the latch inputs in terms of the circuit inputs and secondary variables. For our circuit, we get the equations given by

$$\begin{aligned} S_1 &= x_1y_2 \\ R_1 &= x_1'x_2' \\ S_2 &= x_1x_2 \\ R_2 &= x_2'y_1 \end{aligned}$$

The excitation variables are the latch outputs. We can therefore use equations for the latch outputs written in terms of the latch inputs to

ultimately write the latch outputs in terms of the secondary variables and circuit inputs which are given by

$$Y_1 = R_1'(S_1 + y_1) = (x_1' x_2')'(x_1 y_2 + y_1) = x_1 y_1 + x_2 y_1 + x_1 y_2$$

$$Y_2 = R_2'(S_2 + y_2) = (x_2' y_1)'(x_1 x_2 + y_2) = x_1 x_2 + x_2 y_2 + y_1' y_2$$

The transition table is then obtained by evaluating the excitation and output equations for all possible input variable and secondary variable values. For our example, this yields the transition table shown in Figure 20.8. When working with latches, we might

Current state	**Next state** $(Y_2 Y_1)$			
$y_2 y_1$	$x_2 x_1 = 00$	$x_2 x_1 = 01$	$x_2 x_1 = 11$	$x_2 x_1 = 10$
00	00	00	10	00
01	00	01	11	01
11	00	11	11	01
10	10	10	11	11

Figure 20.8: Transition table dervived for the analysis example in which latches are clearly identified.

want to check to determine if the latches avoid the restricted state (for a SR latch is $SR = 11$). For our current example, we see that $S_1 R_1 = x_1 y_2 x_1' x_2' = 0$ and $S_2 R_2 = x_1 x_2 x_2' y_1 = 0$. Therefore, these latches will not have both set and reset inputs at 1.

20.2 *Asynchronous circuit design*

Similar the clocked sequential design, we can follow a sequence of steps to design a circuit from a verbal problem description. There are, however, some additional complexities that arise during asynchronous design. In addition, there are some other concepts we should exploit to make the design easier. When designing, we should make the **fundamental mode assumption**. An asynchronous circuit is said to be operating in fundamental mode if two conditions are true:

- One **one circuit input** can change at any time;
- The circuit is required to reach a stable state **prior** to changing a circuit input.

When we design using the fundamental mode assumption, we should also derive what is called a **primitive flow table**. A primitive flow table is one in which there is **exactly one stable state per row**. Examples of a non-primitive and a primitive flow table are given in Figure 20.9.

The design procedure for an asynchronous circuit (while pointing out some potential differences when compared to clock sequential

Current state	**Next state** (Y_2Y_1)			
	$x_2x_1 = 00$	$x_2x_1 = 01$	$x_2x_1 = 11$	$x_2x_1 = 10$
a	$\boxed{a}$	$\boxed{a}$	$\boxed{a}$	b
b	a	a	$\boxed{b}$	$\boxed{b}$

(a)

Current state	**Next state** (Y_2Y_1)	
	$x = 0$	$x = 1$
a	$\boxed{a}$	b
b	c	$\boxed{b}$
c	$\boxed{c}$	d
d	a	$\boxed{d}$

(b)

Figure 20.9: Example of a (a) non-primitive flow table and a (b) primitive flow table.

circuit design) can be described as follows given a verbal problem description:

- Derive a state diagram (if desired) and a primitive flow table using the fundamental mode assumption;
- Perform state reduction to obtain a smaller flow table with less states;
- Perform state assignment to obtain a transition table. For asynchronous circuits, state assignment needs to be performed while avoiding critical races;
- Derive equations for the excitation variables (next state) in terms of the secondary variables (current state) and circuit inputs. For asynchronous circuits, we should try to avoid combinational hazards;
- Derive output equations in terms of the secondary variables and circuit inputs. For asynchronous circuits, we should try to avoid output glitches;
- Draw the circuit.

This procedure will result in an asynchronous circuit in which the presence of latches is not obvious. If so desired, we can amend the procedure to design explicitly using latches. In this case, rather that deriving equations for the excitation variables, we can derive equations for latch inputs in order to obtain the desired transitions from stable state to stable state.

We will consider an example to illustrate the procedure. Consider a circuit with two inputs D and G. The circuit has one output Q. When $G = 0$, the output Q maintains its current value, but when $G = 1$, the output Q is equal to the input D. Design an asynchronous circuit that exhibits this behaviour.

To design this circuit, we should proceed in a certain way which might make it easier.

- We should assume fundamental mode since this means from any stable state we have less transitions to consider (only need to consider the transitions caused by changing each input variable);
- We should figure out an (any) initial stable state and begin from there. Let an input change and transition to another stable state. Continue this procedure until you return to a previously created state (this is sort of like following a timing diagram with single inputs changing, the circuit reaching a stable state followed by another input changing).
- Fill in any additional transitions required.

For this problem, we can see that one stable state would be the situation in which $D = 0$, $G = 0$ and $Q = 0$.

Figure 20.10 shows an incomplete state diagram with only the initial state. We can see that this initial state has output $Q = 0$ and is stable (due to the self-loop) in the diagram as long as $DG = 00$.

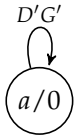


Figure 20.10: Incomplete state diagram for our design example showing only the initial stable state.

We can consider changing each of the inputs. For example, consider changing $G = 0 \rightarrow 1$. The problem specification says that Q should be equal to D (so $Q = 0$ since $D = 0$). According to our recommended procedure, since an input changed we should transition to another stable state. The result is shown in Figure 20.11 which shows that we have introduced a new stable state — the new stable state has a self loop which implies that it is a stable state so long as the inputs do not change. Referring to Figure 20.11, we can also observe

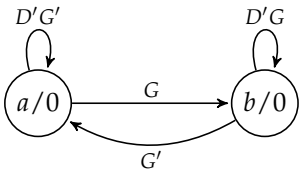


Figure 20.11: Incomplete state diagram for our design example showing an initial transition from the initial stable state to another stable state due to a changed input.

that if we were in stable state b and the input $G = 1 \rightarrow 0$. We would go to stable state a so we can add that edge now.

We can continue from stable state b changing one input variable at a time and eventually end up with the (still) incomplete state diagram shown in Figure 20.12. We are not done, but we need to

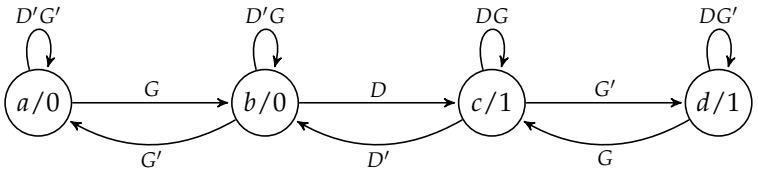


Figure 20.12: Incomplete state diagram for our design example showing additional transitions between stable states.

think carefully about how to proceed. For example, in stable state d we have $Q = 1$ and $DG = 10$. In other words, our circuit is "holding an output value of 1". If we change $D = 0 \rightarrow 1$ we will now have inputs $DG = 00$ and we have a stable state a in which $DG = 00$. However, stable state a produces output 0. But, from stable state d, if $D = 1 \rightarrow 0$, the output should still be 1. Therefore, we need at least one more stable state.

The final (and complete) state diagram for our example is shown in Figure 20.13. We can consider the diagram in Figure 20.13 to

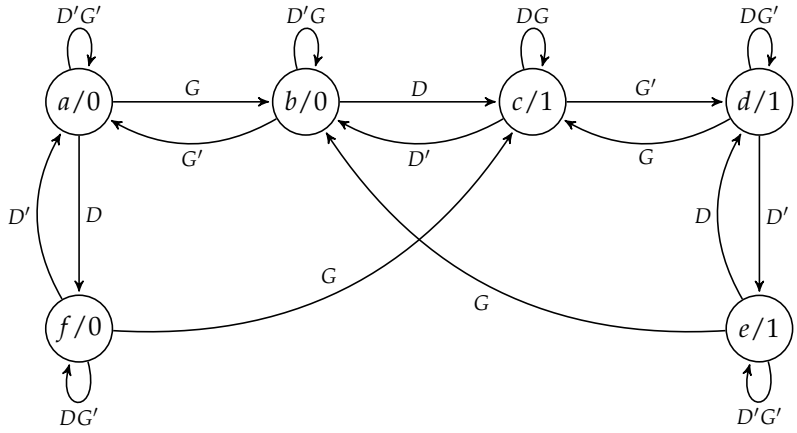


Figure 20.13: Complete state diagram for our design example showing all transitions between stable states.

see if it makes sense compared to our verbal problem description. States a and f represent the scenario where $G = 0$ and $Q = 0$, but the input D is "flipping around". Similarly, states d and e represent a similar scenario, but $Q = 1$ despite D "flipping around". Finally, states b and c indicate the scenario where $G = 1$ and the output Q changes so that is always equals the value of D.

The primitive flow table for the state diagram in Figure 20.13 is shown in Figure 20.15. Again, we end up with a primitive flow table due to the fundamental mode assumption we used when deriving the state diagram. This flow table has a lot of unspecified entries.

Figure 20.14: Primitive flow table for our design example.

Current state	Next state				Output (Q)			
	$DG=00$	$DG=01$	$DG=11$	$DG=10$	$DG=00$	$DG=01$	$DG=11$	$DG=10$
a	$\boxed{a}$	b	–	f	0	–	–	–
b	a	$\boxed{b}$	c	–	–	0	–	–
c	–	b	$\boxed{c}$	d	–	–	1	–
d	e	–	c	$\boxed{d}$	–	–	–	1
e	$\boxed{e}$	b	–	d	1	–	–	–
f	a	–	c	$\boxed{f}$	–	–	–	0

This happens in the transition to the next state since we have assumed fundamental mode operation where inputs cannot change at the same time. This also happens in the value of the output since we only need to specify the value of the output when we are in a stable state.

The flow table should be reduced by merging equivalent states. The procedure for doing this is similar to state minimization with clocked sequential circuits (implication charts and merger diagrams). The major difference we encounter with asynchronous circuits is the large number of unspecified entries in the flow table. Succinctly stated, we can match unspecified entries with anything. The reduced flow table (we can merge a, b and f into one state and c, d and e into another state) is shown in Figure 20.16. The reduced flow table is

Figure 20.15: Reduced flow table for our design example.

Current state	Next state				Output (Q)			
	$DG=00$	$DG=01$	$DG=11$	$DG=10$	$DG=00$	$DG=01$	$DG=11$	$DG=10$
a	$\boxed{a}$	$\boxed{a}$	c	$\boxed{a}$	0	0	–	0
c	$\boxed{c}$	a	$\boxed{c}$	$\boxed{c}$	1	–	1	1

no longer a primitive flow table.

The next step is state assignment. Recall, when performing state assignment, we should avoid race conditions. There are no races here and we will therefore skip the consideration of races at this point in time. Let state a be represented by 0 and let state c be represented by 1 — this yields the transition table shown in Figure 20.16.

Figure 20.16: Transition table for our design example.

Current state (y)	Next state (Y)				Output (Q)			
	$DG=00$	$DG=01$	$DG=11$	$DG=10$	$DG=00$	$DG=01$	$DG=11$	$DG=10$
0	$\boxed{0}$	$\boxed{0}$	1	$\boxed{0}$	0	0	–	0
1	$\boxed{1}$	0	$\boxed{1}$	$\boxed{1}$	1	–	1	1

Since we only need 1-bit to represent the state, we have one excitation

variable Y and one secondary variable y. The excitation variable equation is $Y = DG + G'y$ and the output variable equation is $Q = y$. In deriving the excitiation equations, we should consider whether or not our circuit will exhibit hazards. Further, we should consider whether or not our output equations will exhibit glitches. We won't worry about these considerations right now.

Using these equations, we can draw the final circuit which is the implementation of our verbal problem description. The circuit is shown in Figure 20.18.

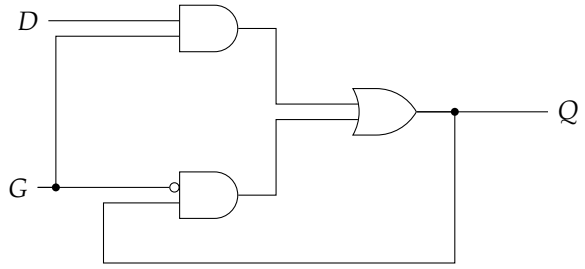


Figure 20.17: Circuit for our design example.

We could also consider using latches to hold the excitation variable values. Rather than deriving an equation for Y, we could derive equation for the latch inputs to obtain the desired state transitions. Assume we want to use $S'R'$ latches constructed from **NAND** gates (Our design procedure would be similar if using SR latches designed with **NOR** gates). A $S'R'$ latch and its table of operation are illustrated in Figures 20.19 and 20.20, respectively. Notice in Figure 20.20, the table is drawn a bit differently to help express how the latch output *changes* given the latch inputs.

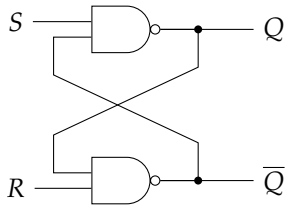


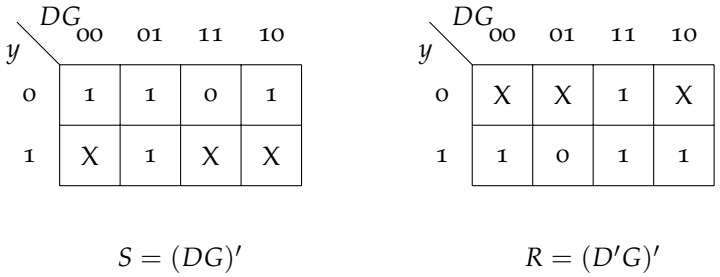
Figure 20.18: The $S'R'$ latch redrawn to refresh our memory.

S	R	$Q(t) \rightarrow Q(t+1)$	
1	X	$0 \rightarrow 0$	*hold or reset*
1	0	$1 \rightarrow 0$	*reset*
0	1	$0 \rightarrow 1$	*set*
X	1	$1 \rightarrow 1$	*hold or set*

Figure 20.19: The $S'R'$ behaviour described differently to emphasize how the output of the latch changes given the latch input values.

If we reconsider our design example and refer back to the transition table shown in Figure 20.16 we can use this transition table to

determine the necessary latch inputs in order to obtain the necessary transitions $y \rightarrow Y$ — the required S and R inputs are shown (via Karnaugh maps) in Figure 20.20. Note that we are using a



$S = (DG)'$ $R = (D'G)'$

Figure 20.20: Required latch inputs (assuming **NAND** latch) to obtain the transitions in the transition table from Figure 20.16.

NAND latch; we should see if we properly avoid the situation where $SR = 00$. This is accomplished by writing a logic expression for $S + R = (DG)' + (D'G)' = D' + G' + D + G' = 1$. Since $S + R$ is never equal to 0 we know that SR cannot be zero at the same time. The final circuit designed using a latch is shown in Figure 20.21.

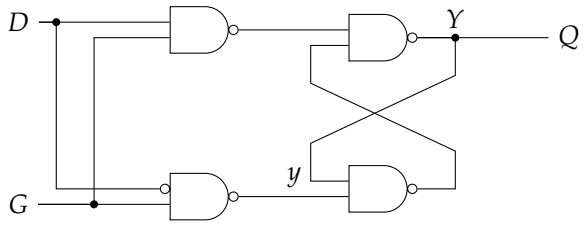


Figure 20.21: Our design example implemented as a circuit which involves the use of a latch.

21 Races

A **race condition** occurs in an asynchronous circuit when two or more state variables are required to change at the same time in response to a change in a circuit input. Unequal circuit delays (due to delay through logic gates and along wires) can imply that the state variables might not change at the same time and this can potentially cause problems. Let us assume that several state variables are required to change:

- If the circuit reaches the correct final stable state regardless of the order in which the state variables change then the race is **not critical**.
- If the circuit reaches can reach different final stable states depending on the order in which the state variables then the race is **critical**.

Therefore, one issue that we should address when designing an asynchronous circuit is to *avoid designs which have critical races*. Fortunately, we can fix and avoid critical races.

A circuit that exhibits a critical race is shown in Figure 21.1. The corresponding transition table is shown in Figure 21.2.

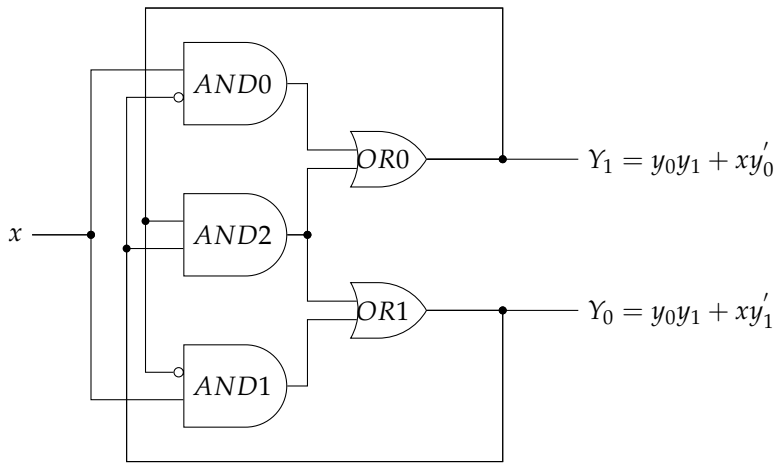


Figure 21.1: Example circuit with a critical race.

Current state	**Next state** (Y_1Y_0)	
y_1y_0	$x=0$	$x=1$
00	00	11
01	00	01
11	11	11
10	00	10

Figure 21.2: Transition table for the circuit in Figure 21.1.

Assume the input x is 0 and the circuit is currently in the stable state with $y_2y_1 = 00$. Then, assume that the input changes from $x = 0 \rightarrow 1$. From the transition table, we see that we should transition to stable state $y_1y_0 = 11$ when $x = 1$. We can, however, consider a few different situations and show that this *might not be the case* depending on the circuit delays.

We can first assume that all the delays in the circuit are equal. If this is the case, we would see the signals change as shown in Figure 21.3. So if we looking at the outputs Y_1Y_0 it would they

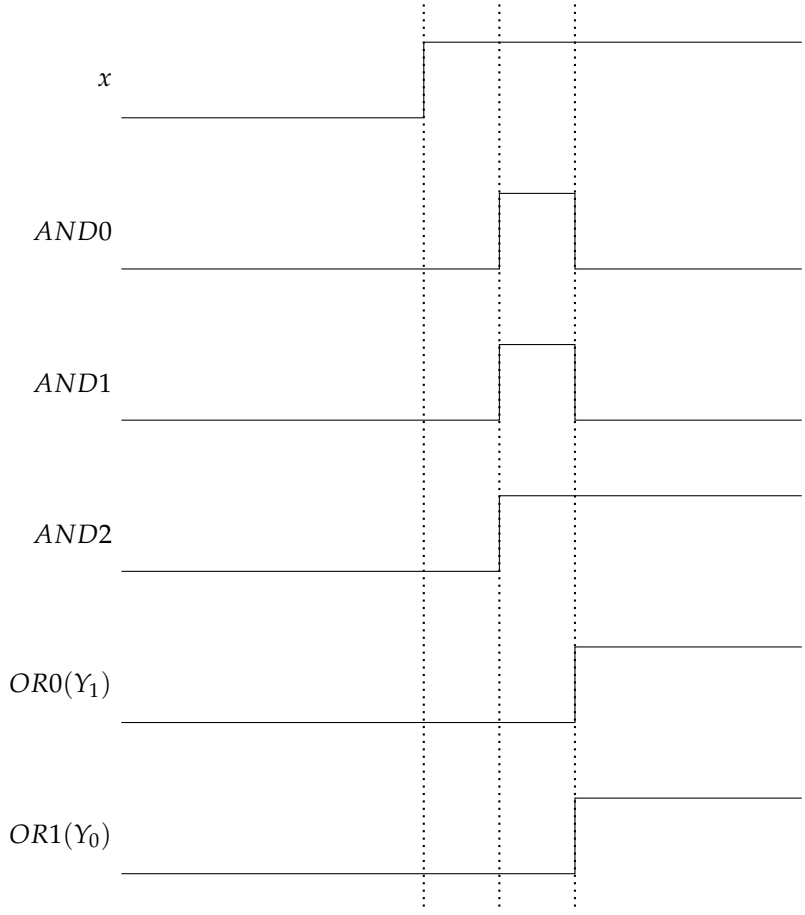


Figure 21.3: Signal transitions assuming equal circuit delays.

would appear to have changed $\boxed{00} \rightarrow \boxed{11}$ which is expected.

Next, we can assume that the delay through $AND0$ and $OR0$ is very fast. This implies that Y_1 will change very quickly. The various waveforms are shown in Figure 21.4. Output Y_1 will change very

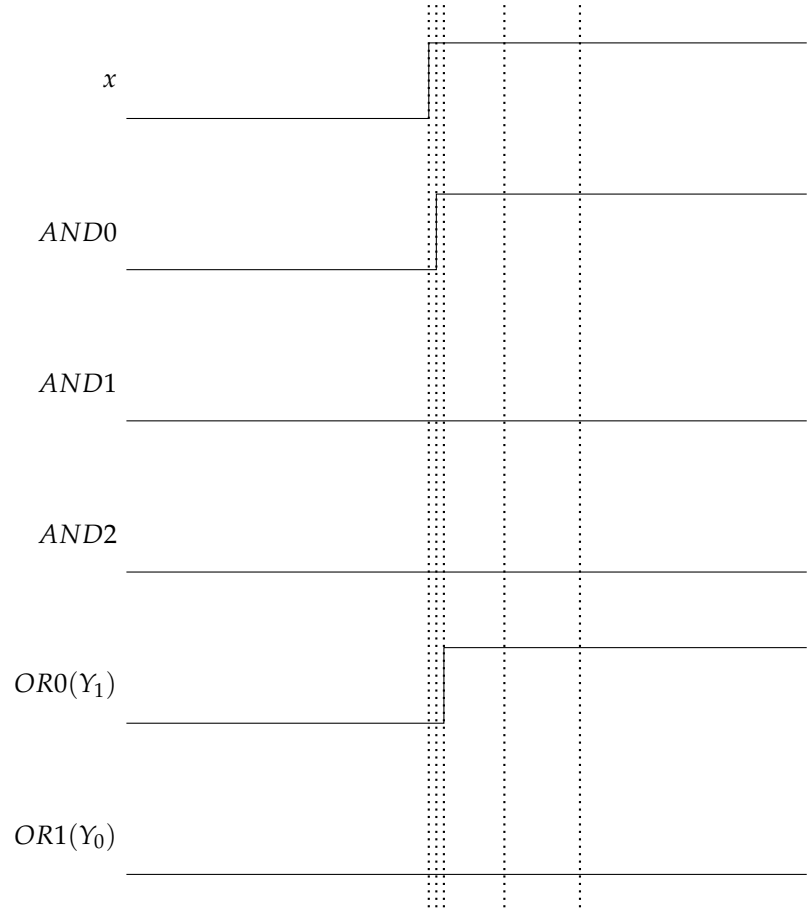


Figure 21.4: Signal traces assuming gates $AND0$ and $OR0$ are very fast compared to other gates. We can observe that Y_1 changes very quickly as x changes from $0 \rightarrow 1$.

quickly from $0 \rightarrow 1$. The value of 1 will propagate back around to the inputs of $AND1$ (inverted) and $AND2$. The value at the inverted input of $AND1$ will prevent $AND1$ from changing its output which means that Y_0 will not change due to x changing from $0 \rightarrow 1$. So if we looking at the outputs Y_1Y_0 it would they would appear to have changed $\boxed{00} \rightarrow \boxed{10}$ **which is the wrong behaviour according to the transition table.** which is expected.

A similar situation exists if we assume that the gates $AND1$ and $OR1$ are both very fast. In this case, we will see signals change as shown in Figure 21.5. Output Y_0 will change very quickly from $0 \rightarrow 1$. The value of 1 will propagate back around to the inputs of $AND0$ (inverted) and $AND2$. The value at the inverted input of $AND0$ will prevent $AND0$ from changing its output which means

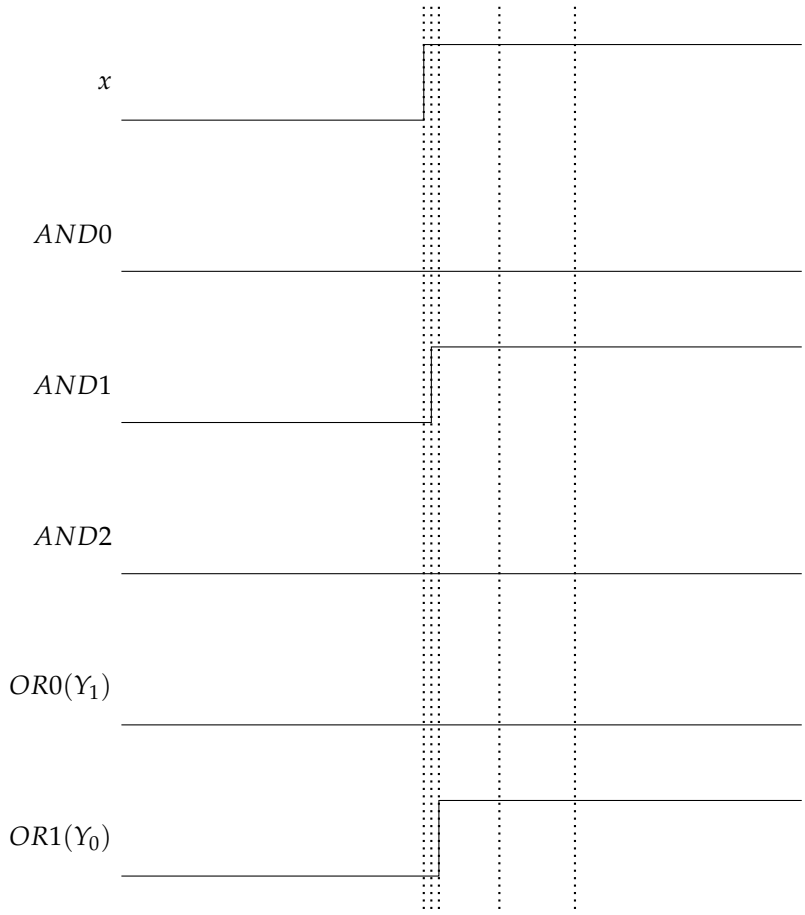


Figure 21.5: Signal traces assuming gates $AND1$ and $OR1$ are very fast compared to other gates. We can observe that Y_0 changes very quickly as x changes from $0 \rightarrow 1$.

that Y_1 will not change due to x changing from $0 \rightarrow 1$. So if we looking at the outputs Y_1Y_0 it would they would appear to have changed $\boxed{00} \rightarrow \boxed{01}$ **which is the wrong behaviour according to the transition table.** which is expected.

While we should do something to prevent or avoid critical races, we do not need to be concerned about non-critical races. Figure 21.6 shows a transition table that exihibits a non-critical race. Assume

Current state	**Next state** (Y_1Y_0)	
y_1y_0	$x = 0$	$x = 1$
00	$\boxed{00}$	11
01	11	$\boxed{01}$
11	$\boxed{10}$	01
10	10	11

Figure 21.6: Transition table exhibiting a non-critical race.

that $x = 0$ and we are in the stable state $y_1y_0 = 00$. Now assume that x changes $0 \rightarrow 1$. According to the transition table in Figure 21.6, we could see the following things are the output: $\boxed{00} \rightarrow \boxed{01}$, $\boxed{00} \rightarrow 11 \rightarrow \boxed{01}$ or $\boxed{00} \rightarrow 10 \rightarrow 11 \rightarrow \boxed{01}$. However, regardless of how the state variables change, we always end up in stable state 01 which is the expected behaviour. This race is therefore non-critical because the circuit will behave correctly and is insensitive to the delays in the circuit.

Races are a direct result of multiple state variables having to change due to a change in an input variable. When examining races we used transitions tables and not flow tables. Races **do not exist** in flow tables because there is no binary state assignment in a flow table (and consequently multiple state variables do not change in a flow table due to a change in an input variable). Consequently, races **are a result of state assignment**. Therefore, we can **avoid** races by being more careful during state assignment. Note that avoiding races is a **preemptive** strategy — races which are non-critical are okay. However, by proper state assignment we can avoid all races and be secure that we will not have any issues due to race conditions. If we are already given a transition table we can consider "fiddling" with the transition table slightly in order to try and remove the critical races (and we can ignore the non-critical races).

The **transition diagram** helps to visualize when a race can potentially occur. The purpose of the transition diagram is to illustrate transitions between stable states. For small problems, the transition diagram is best drawn as a multi-dimensional cube. States are labeled on the corners of the cube while transitions between stable states are drawn as arrows between stable states. Consider the flow table in Figure 21.7. One transition diagram for this flow table is

Current state	Next state $x=0$	Next state $x=1$
a	$\boxed{a}$	b
b	c	$\boxed{b}$
c	$\boxed{c}$	d
d	a	$\boxed{d}$

Figure 21.7: Flow table for illustrating transition diagrams.

shown in Figure 21.8. States are labelled and arrows are drawn

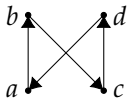


Figure 21.8: One transition diagram for the flow table in Figure 21.7.

between stable state trajectories. For example, if $x = 0$ and we are in stable state a and x changes to 1, we transition to stable stat b. Therefore, we include an arrow from $a \rightarrow b$.

If we consider the corners of the transition diagram to have coordinates, we then have a state assignment. The arrows in the transition diagram will then immediately tell us if there will be a race condition — *diagonal edges in the transition diagram require geq*2 *state variables to change at the same time which indicates a race*. Therefore, the transition diagram in Figure 21.8, as drawn, will exhibit races if we select our state assignment by using the coordinates of the states in the transition diagram. That is, the state assignment $a = 00$, $b = 01$, $c = 10$ and $d = 11$ is will have races.

We can redraw the transition diagram by moving around the states. This will change the appearance of the arrows and possibly avoid diagonal arrows as shown in Figure 21.9.

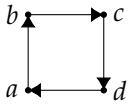


Figure 21.9: Another transition diagram for the flow table in Figure 21.7.

From the transition diagram in Figure 21.9, we can see that the state assignment $a = 00$, $b = 01$, $c = 11$ and $d = 10$ will not have races.

21.1 *Avoiding races during state assignment — use of transition diagrams*

Races can be avoided by using a transition diagram — the objective is to assign states to the corners of a cube and then adjust the flow

table to move from state to state along the edges of the cube rather than along diagonals in the cube. This approach is might require the addition of temporary unstable states. For example, consider the flow table in Figure 21.10. If you construct the transition diagram for

Current state	Next state $x_1x_0 = 00$	$x_1x_0 = 01$	$x_1x_0 = 11$	$x_1x_0 = 10$
a	$\boxed{a}$	$\boxed{a}$	c	b
b	a	$\boxed{b}$	d	$\boxed{b}$
c	$\boxed{c}$	b	$\boxed{c}$	d
d	c	a	$\boxed{d}$	$\boxed{d}$

Figure 21.10: Flow table to demonstrate removal of races using extra (always) unstable states.

this flow table which is shown in Figure 21.11, you will clearly see that it is impossible to even consider rearranging the states on the corners of the cube to remove races (diagonals).

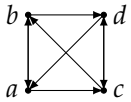


Figure 21.11: Transition diagram for the flow table in Figure 21.10.

This does not mean that we cannot remove the races. It simply means that we might require additional states as previously mentioned. In this case, we can consider drawing a three dimensional cube and attempt to label states on the corners of the cube. A partial solution showing only the original states and a subset of the transitions required is shown in Figure 21.12. In Figure 21.12, it is

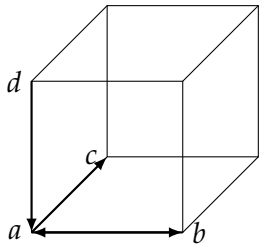


Figure 21.12: Partial transition diagram for the flow table in Figure 21.10.

impossible to add the rest of the transitions between stable states without introducing a diagonal edge and therefore we require additional unstable states. A completed transition diagram showing all transitions without diagonals but with (required) extra states is shown in Figure 21.13. In Figure 21.13 The extra states are e, f and g. State e allows the transition from $b \rightarrow d$. State f allows the transition from $c \rightarrow b$. Finally, state g allows the transitions from $c \rightarrow d$ and $d \rightarrow c$. The final flow table and state assignment (so there-

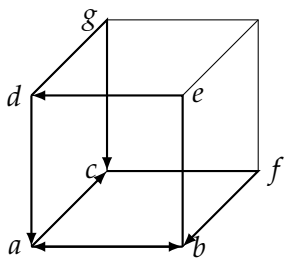


Figure 21.13: Complete transition diagram for the flow table in Figure 21.10 with additional unstable states to avoid diagonal edges. The state assignment obtained from this transition diagram is race free.

fore a transition table) is shown in Figure 21.14 and is completely race free.

	Current state	**Next state** $x_1x_0 = 00$	$x_1x_0 = 01$	$x_1x_0 = 11$	$x_1x_0 = 10$
000	a	$\boxed{a}$	$\boxed{a}$	c	b
100	b	a	$\boxed{b}$	e	$\boxed{b}$
010	c	$\boxed{c}$	f	$\boxed{c}$	g
001	d	g	a	$\boxed{d}$	$\boxed{d}$
101	e	–	–	d	-
110	f	–	b	-	-
011	g	c	–	-	d

Figure 21.14: Race free transition table for the flow table in Figure 21.10. Note that this is referred to as a transition table because the state assignment is given.

21.2 *Avoiding races during state assignment — one hot encoding*

The idea of one-hot encoding can be used to get a race free state assignment. Assume a change in an input variable requires moving from stable state i to stable state b. Assume that every state k is encoded as $0...0 \underbrace{1}_{k\text{-th bit}} 0...0$. Only a single bit is a 1. To move from stable state i to stable state j requires *exactly* 2 bits to change (bit i must change from 1 to 0 and bit j must change from 0 to 1. This causes a race, but we can *always avoid the race* by introducing a new unstable state in which both bits i and j are 1; we are effectively forcing the bits i and j to change one *after* the other. To be clear, if we require a transition from stable state i to stable state j requires the following bits to change:

$$0...0 \underbrace{1}_{\text{bit } i} 0...0 \underbrace{0}_{\text{bit } j} 0...0 \rightarrow 0...0 \underbrace{0}_{\text{bit } i} 0...0 \underbrace{1}_{\text{bit } j} 0...0$$

Instead, however, we go through an intermediate unstable state as follows:

$$0...0\underbrace{1}_{\text{bit } i}0...0\underbrace{0}_{\text{bit } j}0...0 \rightarrow 0...0\underbrace{1}_{\text{bit } i}0...0\underbrace{1}_{\text{bit } j}0...0 \rightarrow 0...0\underbrace{0}_{\text{bit } i}0...0\underbrace{1}_{\text{bit } j}0...0$$

In effect, we are *forcing* bit j to change *before* bit i.

To demonstrate race free state assignment using one hot encoding, consider the flow table shown in Figure 21.15. Note that a

Figure 21.15: Flow table to demonstrate race free state assignment using one hot encoding.

State assignment	**Current state**	**Next state** $x_2x_1 = 00$	$x_2x_1 = 01$	$x_2x_1 = 11$	$x_2x_1 = 10$
0001	a	$\boxed{a}$	$\boxed{a}$	c	b
0010	b	a	$\boxed{b}$	d	$\boxed{b}$
0100	c	$\boxed{c}$	b	$\boxed{c}$	d
1000	d	c	a	$\boxed{d}$	$\boxed{d}$

one hot state assignment is shown and that this state assignment does exhibit races using this state assignment. New stable states are introduced where appropriate to yield the flow table with state assignment shown in Figure 21.16. From Figure 21.16, we see that

Figure 21.16: Race free state assignment with additional unstable states.

State assignment	**Current state**	**Next state** $x_2x_1 = 00$	$x_2x_1 = 01$	$x_2x_1 = 11$	$x_2x_1 = 10$
0001	a	$\boxed{a}$	$\boxed{a}$	e	f
0010	b	f	$\boxed{b}$	g	$\boxed{b}$
0100	c	$\boxed{c}$	h	$\boxed{c}$	i
1000	d	i	j	$\boxed{d}$	$\boxed{d}$
0101	e	–	–	c	-
0011	f	a	–	–	b
1010	g	–	–	d	-
0110	h	–	b	–	-
1100	i	c	–	–	d
1001	j	–	a	–	-

entries have been changed to use the unstable states. Further, the unstable states permit the original transitions as follows: e permits transitions between states a and c; f permits transitions between states a and b; g permits transitions between states b and d; h permits transitions between states b and c; i permits transitions between states c and d; and j permits transitions between states a and d.

21.3 *Avoiding races during state assignment — state duplication*

State duplication works for small flow tables in which the number of states is ≤ 4. It works by duplicating each state into a pair of equivalent states. How this method works is easy to visualize if you draw a cube and label the corners of the cube with their coordinates. Assume we have states a, b, c and d and we will duplicate each state; e.g., a becomes a_1 and a_0, and so forth. This is shown in Figure 21.17.

Two things can be noticed from Figure 21.17 and how the equivalent states have been labeled:

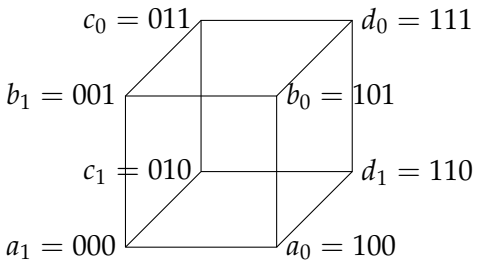


Figure 21.17: Method of state duplication for obtaining race free state assignments for small flow tables.

1. One can always move between equivalent states by changing only 1 bit because all equivalent states are adjacent to each other;
2. Each original state is adjacent to every other state once we take into account equivalent states and can therefore get from any state to any other state by changing only 1 bit.

To demonstrate the state duplication approach consider the flow table in Figure 21.18. With state duplication and some minor

Current state	**Next state**			
	$x_2x_1 = 00$	$x_2x_1 = 01$	$x_2x_1 = 11$	$x_2x_1 = 10$
a	$\boxed{a}$	$\boxed{a}$	c	b
b	a	$\boxed{b}$	d	$\boxed{b}$
c	$\boxed{c}$	b	$\boxed{c}$	d
d	c	a	$\boxed{d}$	$\boxed{d}$

Figure 21.18: Flow table to demonstrate race free state assignment using state duplication

.

modifications to the flow table we get a new flow table with a state assignment shown in Figure 21.19. Note in Figure 21.19, that some unstable entries are changed as required to sometimes first pass through the equivalent state rather than move directly to the desired state. An example would be stable state c_1 with $x_2x_1 = 00 \rightarrow 01$ — rather than trying to go directly to state b_1 or b_0 we temporarily pass through state c_0 (which then goes to b_1).

State assignment	Current state	Next state $x_2x_1 = 00$	$x_2x_1 = 01$	$x_2x_1 = 11$	$x_2x_1 = 10$
000	a_1	$\boxed{a_1}$	$\boxed{a_1}$	c_1	b_1
100	a_0	$\boxed{a_0}$	$\boxed{a_0}$	a_1	b_0
001	b_1	a_1	$\boxed{b_1}$	b_0	$\boxed{b_1}$
101	b_0	a_0	$\boxed{b_0}$	d_0	$\boxed{b_0}$
010	c_1	$\boxed{c_1}$	c_0	$\boxed{c_1}$	d_1
011	c_0	$\boxed{c_0}$	b_1	$\boxed{c_0}$	d_0
110	d_1	c_1	a_0	$\boxed{d_1}$	$\boxed{d_1}$
111	d_0	c_0	d_1	$\boxed{d_0}$	$\boxed{d_0}$

Figure 21.19: Race free state assignment using state duplication.

22 *Hazards*

Hazards are not unique to asynchronous circuits — they happen in any circuit. However, we tend to discuss hazards once we start talking about asynhronous circuits because hazards can cause chaos in an asynchronous circuit.

So what is a hazard? A *hazard* is a momentary *unwanted* switching transient at the output of a logic function. In other words, a hazard produces a *glitch* at the output of a circuit. Hazards result due to an unequal propagation delay along different paths in a combinational circuit from an input to the logic function's output. Hazards may be either *static* or *dynamic* and are classified by the type of glitch they may produce. Static hazards themselves are categorized as either *static*-0 *hazards* or *static*-1 *hazards*.

A static-0 hazard is shown in Figure 22.1. In Figure 22.1, we

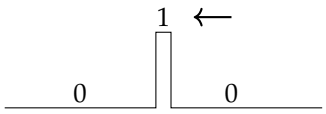


Figure 22.1: Illustration of a static-0 hazard.

see the output of a circuit in which the output should not change (due to a change in an input signal). However, we see a brief period in time when the output is 1. The "width" or amount of time the logic function output is 1 is very narrow, but the function should have remained constant at 0 nevertheless.

A static-1 hazard is shown in Figure 22.2. In Figure 22.2, we

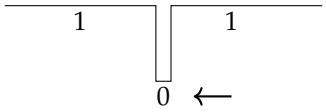


Figure 22.2: Illustration of a static-0 hazard.

see the output of a circuit in which the output should not change (due to a change in an input signal). However, we see a brief period in time when the output is 0. The "width" or amount of time the logic function output is 0 is very narrow, but the function should

have remained constant at 1 nevertheless.

Finally, a dynamic hazard occurs when inputs change and the circuit output is expected to change from $0 \rightarrow 1$ or from $1 \rightarrow 0$. However, in making the transition, the logic function output "flips" one or more times. This is shown in Figure 22.3.

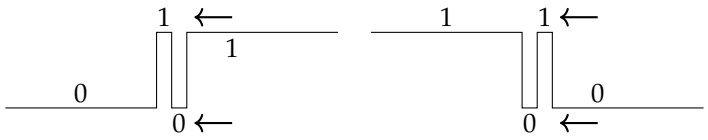


Figure 22.3: Example of dynamic hazards for a signal changing $0 \rightarrow 1$ and $1 \rightarrow 0$.

In a sequential circuit, hazards are not really a concern since we have a clock period for signals to stabilize prior to the active clock edge causing the flip flops to update. However, in an asynchronous circuit hazards can cause terrible problems — the glitch caused by the hazard can propagate back in the circuit and cause the circuit to malfunction (e.g., enter into an incorrect state). In some cases, we can mask or prevent hazards from causing glitches.

22.1 *Illustration of a static hazard*

To illustrate a static hazard, we will consider the circuit shown in Figure 22.4 which is a sum-of-products implementation of the function $f = ab + \bar{b}c$. We assume that every logic gate incurs one unit of delay. Finally, we assume the circuit inputs are at the values shown in Figure 22.4 and that b changes from $0 \rightarrow 1$ at time 0.

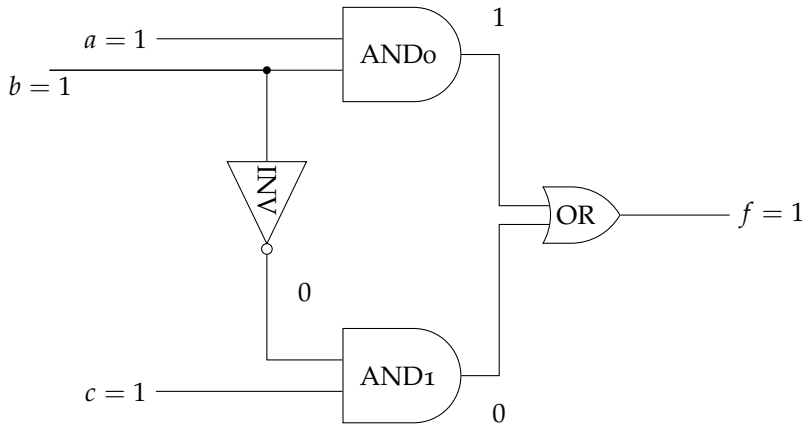


Figure 22.4: Sample circuit to show an example of a static-1 hazard. The output values of all logic gates are shown given that the circuit inputs are $a = 1$, $b = 1$ and $c = 1$.

Figure 22.5 shows how this outputs of the logic gates change at time 0 when b changes from $1 \rightarrow 0$. In Figure 22.5, the outputs

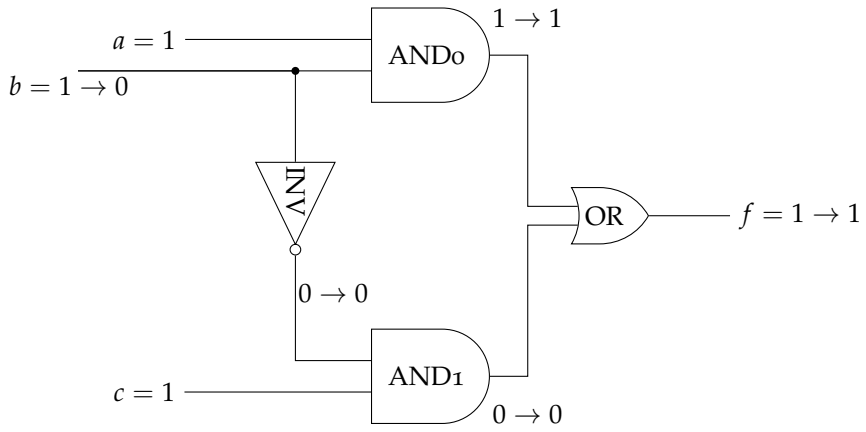


Figure 22.5: Values at gate outputs at time 0.

of INV and ANDo do **not change immediately** due to delay even though logically they will change.

Figure 22.6 shows gate outputs at time 1. In Figure 22.6 the

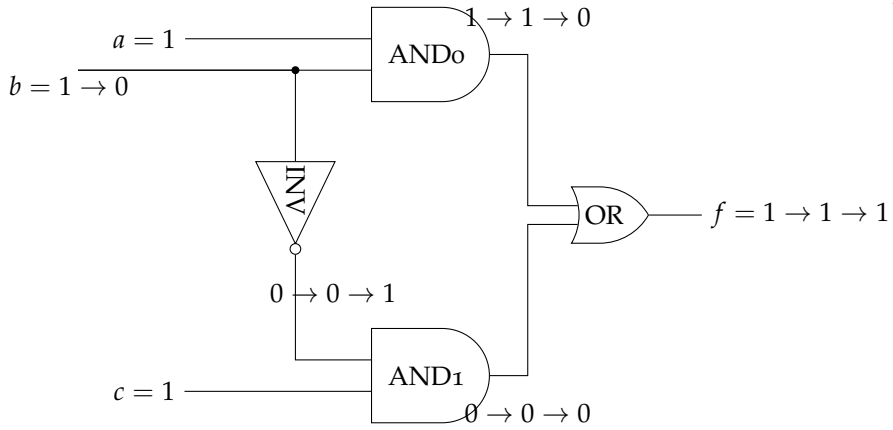


Figure 22.6: Values at gate outputs at time 1.

outputs of INV and ANDo have changed at time 1. However, the output of AND1 has **not yet changed** due to delay even though logically it will change.

Figure 22.7 shows the gate outputs at time 2. In Figure 22.7 the output of AND1 has changed. It is also the case that the output of OR changed to zero at time 2. This was because at time 1, both its inputs were 0. Note now, however, that the inputs to OR imply that its output should be 1, but the output does **not yet change** due to delay.

Finally, Figure 22.8 shows the gate outputs at time 3. The

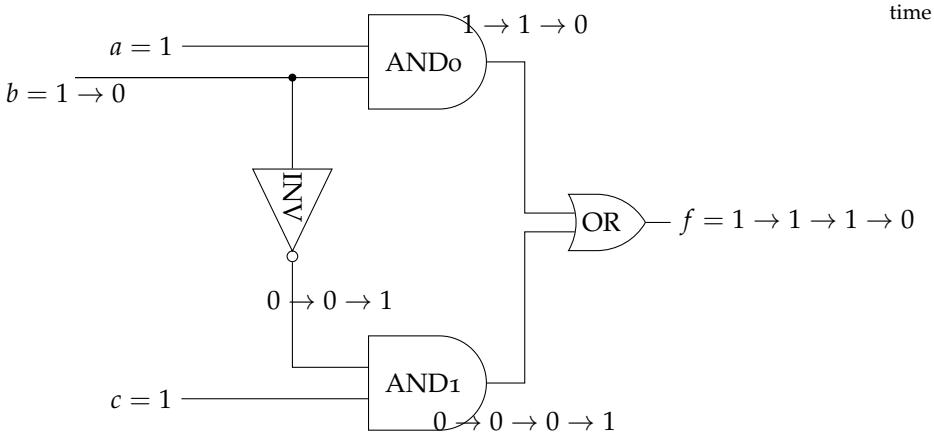


Figure 22.7: Values at gate outputs at time 2.

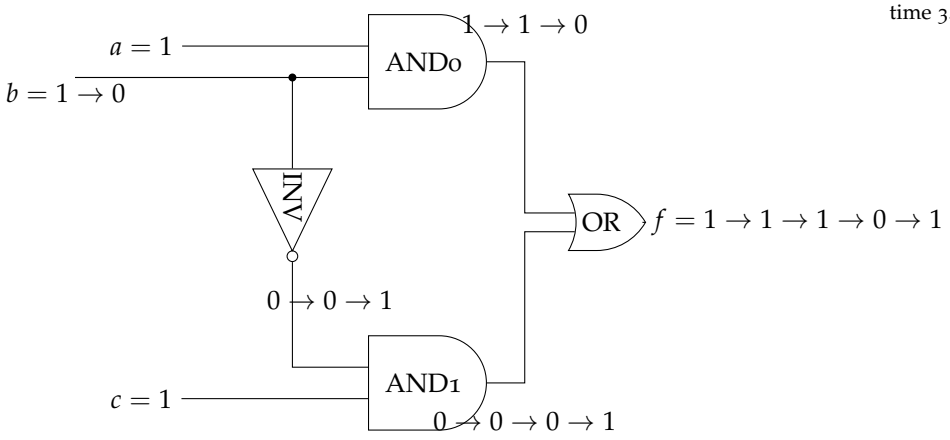


Figure 22.8: Values at gate outputs at time 3.

output of OR has changed to 1. It should be clear that the OR gate output which was initially 1 should have remained 1, but temporarily change to 0. **So we have encountered a static 1 hazard.**

Figure 22.9 shows the logic gate outputs from Figure 22.4 through Figure 22.8 as a timing diagram which clearly shows when logic values in the circuit change.

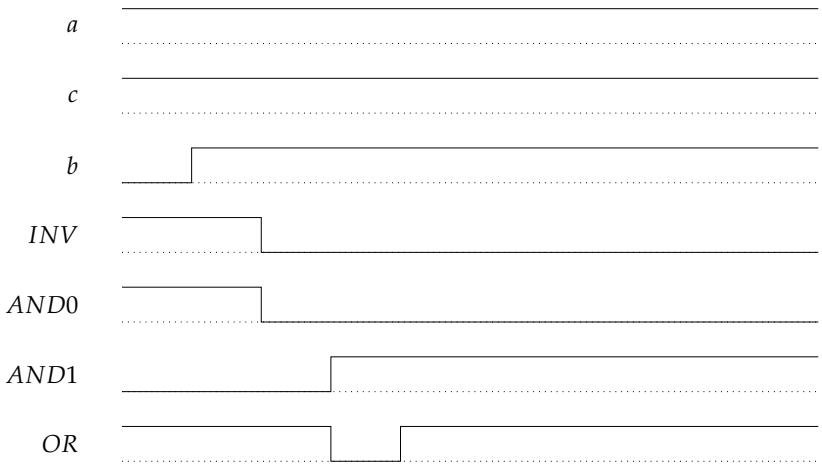


Figure 22.9: Timing diagram for the circuit values shown in Figure 22.4 through Figure 22.8. The output of the OR gate clearly shows a static-1 hazard due to the change in input b.

22.2 *Basic static-1 hazards*

We will consider static-1 hazards caused due to a change in a single input variable. Static-1 hazards are characterized by the hypothetical circuit shown in figure 22.10.

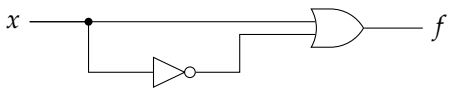


Figure 22.10: Hypothetical circuit to demonstrate a static-1 hazard. The idea is that, at the input of an OR gate we get both x and $\overline{x}$, but, due to the realities of actual circuits, the values x and $\overline{x}$ do not arrive at the same time. There is potential for both inputs to the OR gate to be 0 even though that cannot happen logically.

In Figure 22.10 we have a circuit in which there are two parallel paths from input x and one of these paths is inverted. The paths re-converge at an OR gate. The OR gate function is $f = x + \overline{x}$. Logically, this would result if $f = 1$ always. But, due to circuit delay, we cannot rely on both x and $\overline{x}$ arriving at the input of the OR gate at the same time — we sort of need to consider x and $\overline{x}$ as *separate signals*. Therefore, it is possible that OR gate will see 0 at both inputs for a brief moment in time which will potenially cause a static-1 hazard.

A more elaborate example involving more logic gates is shown in Figure 22.11. The constant values at the inputs of the logic gates in Figure 22.11 are intended to be input variables which are held constant at those values. Then, if we consider only input x, we can

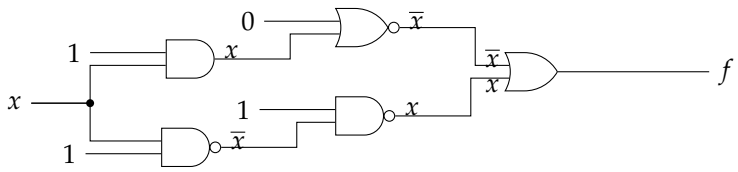


Figure 22.11: A more complicated situation in which a static-1 hazard exists.

see that there are two paths from x which reconverge at the OR gate. This means that, if the delays along the paths are not equal, the OR gate will see x and $\overline{x}$ which could both be 0 causing a static-1 hazard.

22.3 *Basic static-0 hazards*

Static-0 hazards are characterized by the hypothetical circuit in Figure 22.12.

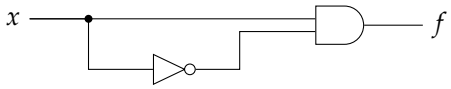


Figure 22.12: Hypothetical circuit to demonstrate a static-0 hazard. The idea is that, at the input of an AND gate we get both x and $\overline{x}$, but, due to the realities of actual circuits, the values x and $\overline{x}$ do not arrive at the same time. There is potential for both inputs to the AND gate to be 1 even though that cannot happen logically.

In Figure 22.12 have a circuit in which there are two parallel paths from input x and one of these paths is inverted. The paths reconverge at an AND gate. The function $f = x\overline{x}$ butdue to delay x and $\overline{x}$ need to be considered as separate signals. Therefore, it is possible that AND gate will see 1 at both inputs for a brief moment in time.

Figure 22.13 shows a more elaborate example of a static-0 hazard embedded in a circuit.

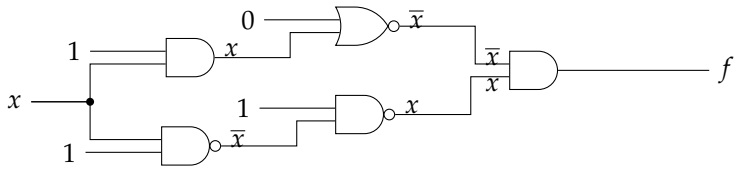


Figure 22.13: A more elaborate example of a static-0 hazard.

In Figure 22.13, we still see the potential for x and $\overline{x}$ to appear at the input to the final AND gate if other inputs are set appropriately. This means there is a static-0 hazard.

22.4 *Static-1 hazards and sum-of-products (redundant terms)*

Sum-of-products expressions may potentially have static-1 hazards due to one input variable changes. Sum-of-products expressions will not have static-0 hazards or dynamic hazards. Masking hazards in sum-of-products expressions due to single input variable changes

is pretty straightforward. Consider the logic function $f = ab + \overline{b}c$ whose Karnaugh map is shown in Figure 22.14. The product terms ab and $\overline{b}c$ are identified in the Karnaugh map. We can observe that

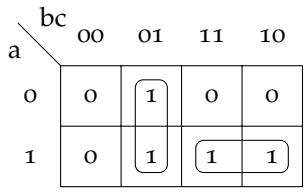


Figure 22.14: Karnaugh map for $f = ab + \overline{b}c$.

when b changes from $1 \rightarrow 0$ that we "jump" from one product term to another. Further, at any given time, there is only 1 product term responsible for holding $f = 1$.

We should introduce another product term into the implementation of f which is independent of the variable b. This is shown in Figure 22.15. Now we can write $f = ab + \overline{b}c + ac$ which is a

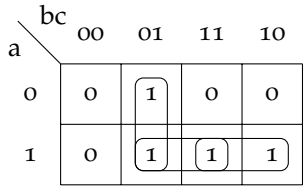


Figure 22.15: Karnaugh map for $f = ab + \overline{b}c$ with an extra redundant product term identified to mask a hazard.

valid (although not minimal) implementation of f. The additional (redundant) product term is independent of input b changing. This additional product term will be 1 while $b = 1 \rightarrow 0$ and will hold $f = 1$.

The final SOP implementation of f in which all static-1 hazards due to a single input variable chaning are masked is shown in Figure 22.16. To fix static-1 hazards in sum-of-products expressions we add redundant product terms to ensure that a single input variable changing does not cause the input variable and its complement to be the only inputs to the OR gate. The redundant product term "holds" the output at 1 and prevents the static-1 hazard.

22.5 *Static-0 hazards and product-of-sums (redundant terms)*

Product-of-sums expressions may potentially have static-0 hazards due to one input variable changes. Product-of-sums expressions will

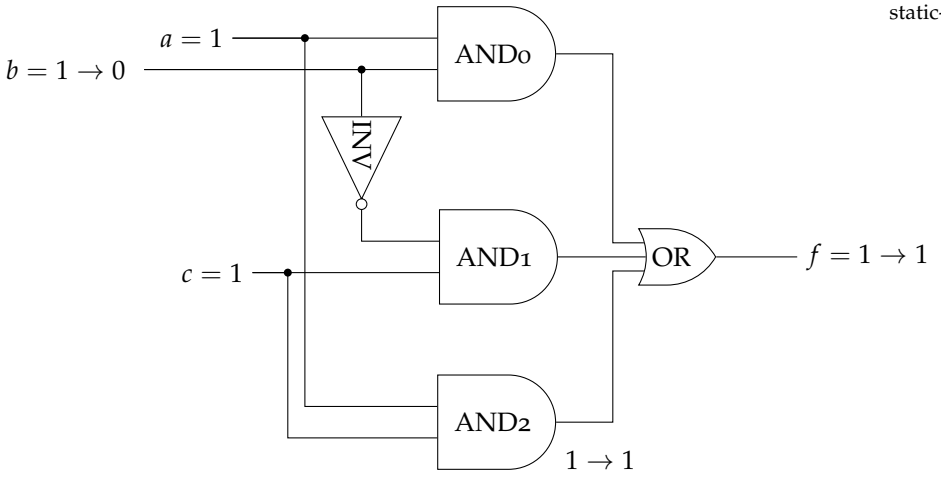


Figure 22.16: Circuit for f in which all static-1 hazards are masked.

not have static-1 hazards or dynamic hazards. Masking hazards in product-of-sums expressions due to single input variable changes is pretty straightforward. Add redundant sum terms which are independent of the single input variable that causes the hazard. As an example, consider the POS implementation of $f = (a + \overline{b})(b + c)$. This implementation has a static-0 hazard when $a = 0$, $c = 0$ and b changes from $1 \to 0$. This can be confirmed by drawing the Karnaugh map for f and identifying the two sum terms used by f. This hazard is masked by adding the redundant sum term $(a + c)$ and use $f = (a + \overline{b})(b + c)(a + c)$.

To fix static-0 hazards in product-of-sums expressions we add redundant sum terms to ensure that a single input variable changing does not cause the input variable and its complement to be the only inputs to the AND gate. The redundant sum term "holds" the output at 0 and prevents the static-0 hazard.

22.6 *Illustration of a dynamic hazard*

Dynamic hazards occur in multilevel circuits in which there are ≥ 3 paths from a circuit input to the circuit output and the paths have unequal delays. We will not consider how to mask or prevent dynamic hazards, but rather just examine a circuit to see that such problems can exist. Figure 22.17 shows and example of a multilevel circuit in which there are multiple paths from input a to the circuit output f. Assume that every gate has 1 unit of delay and further assume the circuit inputs are initially the values shown in Figure 22.17. Finally, assume that a changes from $0 \to 1$ at time 0. Figure 22.17 shows how the outputs of each of the logic gates change over time; these

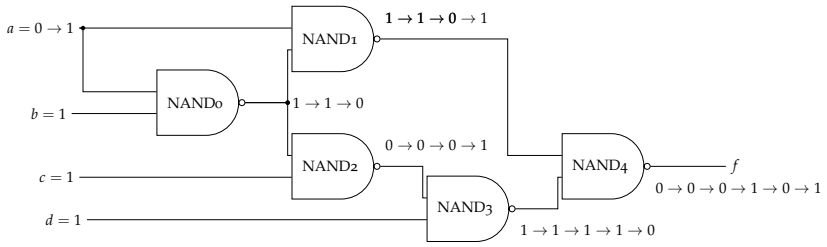


Figure 22.17: Circuit to illustrate a dynamic hazard.

signals are shown in time in Figure 22.18. As previously stated,

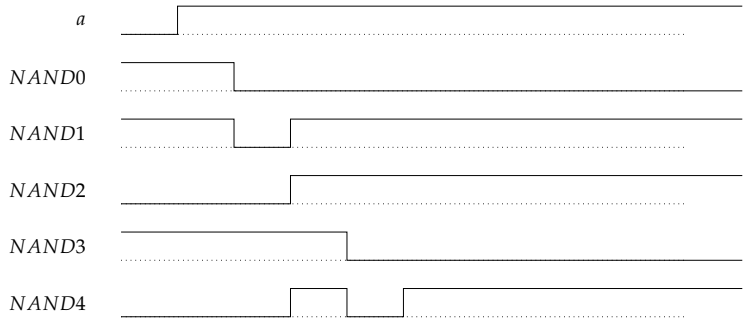


Figure 22.18: Timing diagram for the circuit in Figure 22.17. The dynamic hazard due to changes in input a are visible at the output of the logic gate $NAND4$ which is also the circuit output.

dynamic hazards are more difficult to mask so we won't consider it. However, if we did need to mask the dynamic hazards, we could do so with what we already know — if we wanted or needed something in which hazards were masked, we we could always implement f as a 2-level SOP or POS (so no dynamic hazards) and then proceed to remove static hazards.