ECE241: Digital Systems

Arnav Patil

University of Toronto

Contents

1 Introduction

1.1 Digital Hardware

Until the 1960s, individual transistors and registers had to assembled to form logic circuits. The invention of the integrated circuit allowed for multiple transistors to be placed together; the number of transistors has roughly doubled every two years in a phenomenon called *Moore's Law*. Integrated circuits are manufactured on a silicon wafer, which is cut to make individual chips. For many digital hardware products, it is necessary to build logic circuits; we can use three main types of chips: standard chips, programmable logic devices, and custom chips.

1.1.1 Standard Chips

Each standard chip contains a small amount of circuitry (usually <100 transistors) and performs a very simple function. Smaller chips can be interconnected to create larger logic circuits. These chips are no longer popular as they take up valuable printed circuit board (PCB) space for more capable chips and their functionality is fixed.

1.1.2 Programmable Logic Devices

We may also construct chips with programmable switches that allow the user to implement their own logic circuits. This way, the designer can achieve the functionality they require; these chips are known as *programmable logic devices (PLDs)*. The most common type of PLD is called a *field-programmable gate array, or FPGA*. Because of their capability and flexibility, FPGAs are very popular and widely in use today.

1.1.3 Custom-Designed Chips

FPGAs may not be able to achieve performance or cost objectives because they consume valuable chip area, which further reduces operating speed. Thus, some designers may choose to build a custom chip that meets their requirements, which is then fabricated and put into use. These chips are known as *application-specific integrated circuits (ASICs)*.

1.2 The Design Process

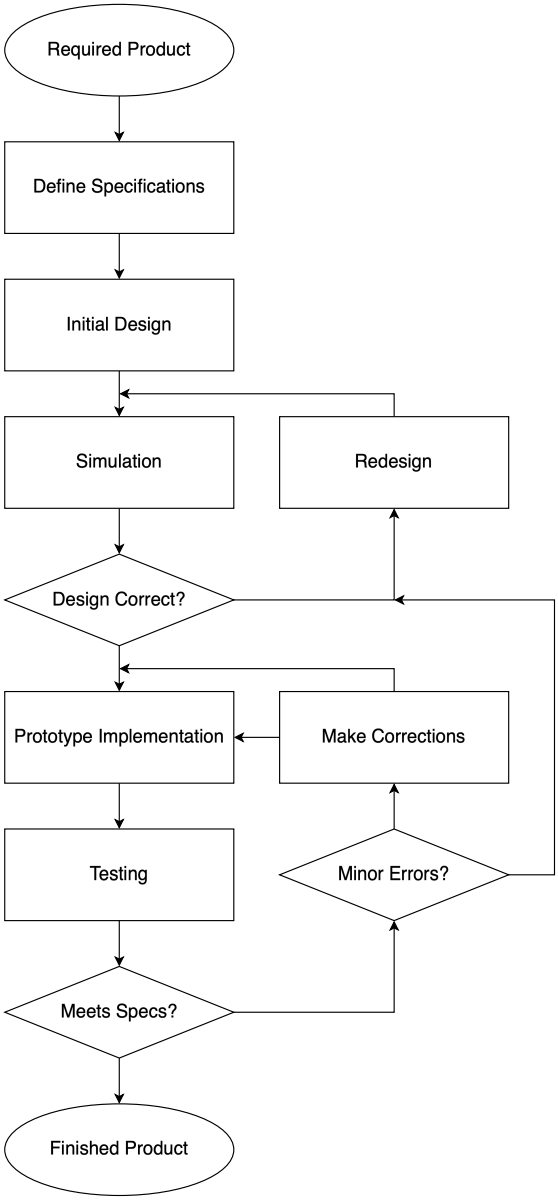


Figure 1: The Design Process.

1.3 Structure of a Computer

A computer consits of a number of PCBs, all of which are connected to a 'main PCB' called a motherboard. The motherboard holds several integrated circuit chips and provides slots for connections to other PCBs.

Each chip contains a number of subcircuits, which are interconnected and are logic circuits. Each logic circuit consists of a number of interconnected logic gates, which performs a very simple function. The text on which these notes are based is concerned with the design of these logic circuits. We will focus on building these circuits for speed or cost, and how they can be tested. We will also explore how transistors work and how they are constructed on silicon chips.

1.4 Logic Circuit Design in This Book

Section discussing Verilog HDL and Quartus II software.

1.5 Digital Representation of Information

Each digit is allowed to take on only one of two values: 0 or 1. 0 usually represents 0 V or ground, while 1 represents the logic circuit voltage (somewhere between 1-5 V).

To convert a decimal number into a binary number, the easiest method is to procedurally divide by two (note down the reminders); though keep in mind this method gives the digits from the least significant bit (right most digit) to the most significant bit (left most digit).

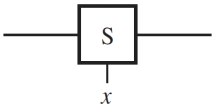
2 Introduction to Logic Circuits

Any circuit in which signals are limited to some number of discrete values is called a **logic circuit**.

2.1 Variables and Functions

One of the simplest binary elements is the switch which has only two states: on and off.

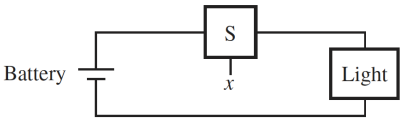
(a) Two states of a switch



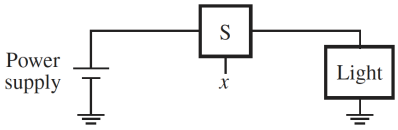
(b) Symbol for a switch

Figure 2: A simple binary switch

We can attach these switches into circuits to introduce control.



(a) Simple connection to a battery



(b) Using a ground connection as the return path

Figure 3: Simple circuit with switch

In the above example, let us denote the light using the letter L. If the light is on, we say $L = 1$, and if it's off, we say $L = 0$. Since the light is controlled by the switch, we can denote L as a function of x, which we would write as $L(x) = x$. This simple logic expression describes the output as a function of the input. We can say that $L(x) = x$ is a logic function, and that x is a logic input.

Now we will consider what happens if we use two switches to control the light. If these switches are placed in series, then both must be turned on in order for the light to turn on. This is a logical AND function. The

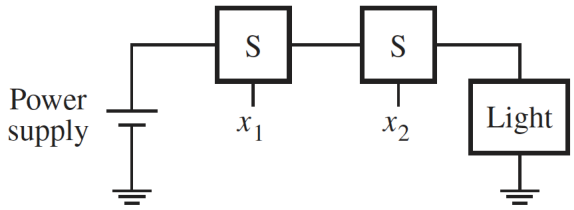


Figure 4: Logical AND function

behaviour of the AND function is described by:

$$L(x_1, x_2) = x_1 \cdot x_2$$

although we more commonly ignore the dot and instead write x_1x_2.

If, on the other hand, the switches were placed in parallel, then only one needs to be turned on in order for the light to turn on (note that the light will be on if both switches are turned on as well). This describes a logical OR function.

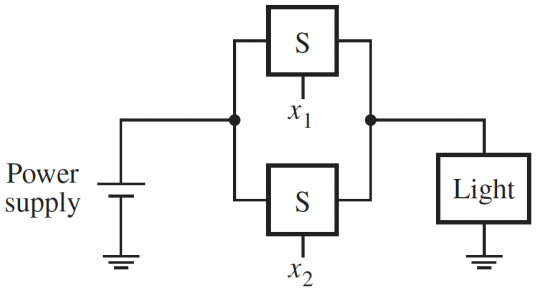


Figure 5: Logical OR function

The behaviour of the OR function is given by:

$$L(x_1, x_2) = x_1 + x_2$$

2.2 Inversion

Now we will consider a circuit where a positive action takes place when a switch is opened rather than closed. Consider the following inverting circuit, which described a NOT function. The reason this gate works is because of the fundamental circuit property that current will take the path of least resistance. When the switch is closed, it acts as a short circuit, meaning it will travel through the switch instead of the light (which has a given resistance), not turning it on. When the switch is opened, the current will be forced to go through the light, thereby turning it on. We describe the behaviour of the NOT function by:

$$L(x) = \overline{x}$$

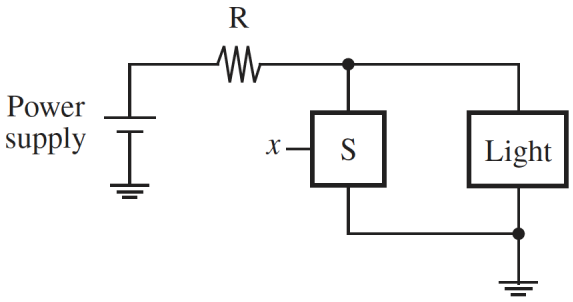


Figure 6: Logical NOT function

2.3 Truth Tables

We can visualize the behaviour of logic functions using a truth table, which gives the outcome for every possible input provided to the logic circuit. For example, here is a truth table visualizing both the AND and OR operations for two inputs.

x_1 x_2	$x_1 \cdot x_2$	$x_1 + x_2$
0 0	0	0
0 1	0	1
1 0	0	1
1 1	1	1
	AND	OR

2.4 Logic Gates and Networks

Each logic operation can be implemented using transistors to form what's known as a logic gate. We often draw schematics to describe a logic function's inputs and outputs. The basic gates are given below:

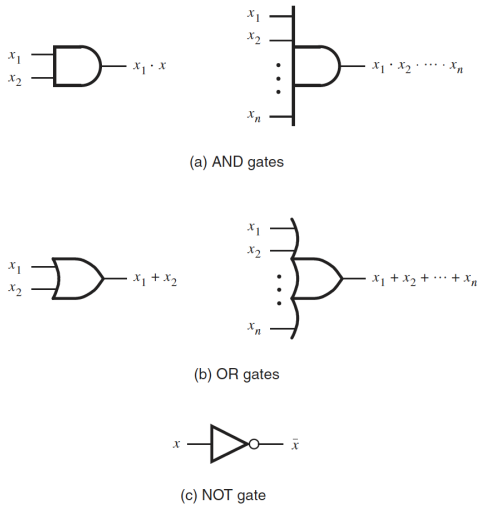


Figure 7: Basic Logic Gates

We can connect gates in a network to create a larger circuit. Since cost is a direct function of the complexity

of any network, we try to find ways to describe a given logic function as simply as possible.

2.4.1 Analysis of a Logic Network

There are two parts to the digital systems design process: analysis (figuring out what the logic network does) and synthesis (building the actual circuit that does the function). The simplest method remains to build a truth table for the logic function, though that can make it difficult to visualize how exactly a given output is produced.

Timing Diagram

Timing diagrams are a way to visualize the changes in signal at different points in the network in a graphical form. Idealized waveforms rely on the assumption that logic gates can respond to changes to their inputs instantly, something that practical logic circuits do not do. However, we will study this behaviour in later chapters.

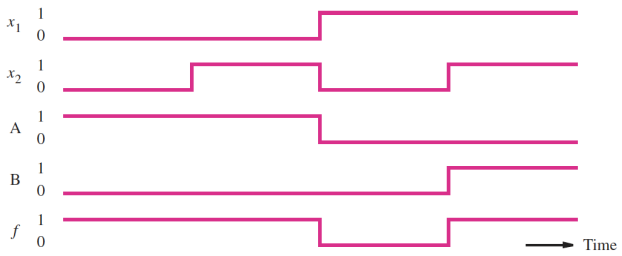


Figure 8: Timing Diagram for $f = \overline{x}_1 + x_1 \cdot x_2$

Functionally Equivalent Networks

Consider the function from the above timing diagram, $f = \overline{x}_1 + x_1 \cdot x_2$. The function $g = \overline{x}_1 + x_2$ will have the same diagram, as if we look closely, $f(x_1, x_2) = g(x_1, x_2)$. However, g is much simpler to implement, and it follows that one should implement g over f.

Example: Hallway Light Switch
Let us consider an example of a unique circuit, say one that controls the light in a hallway using two switches. When one is turned off and the other is turned on, the light is on. However, if both switches are on or off, the light is turned off. Essentially, we can define a function $L(x_1, x_2)$ where $L = 0$ when $x_1 = x_2$. The truth table for this function can be given by:

x y	L
0 0	0
0 1	1
0 0	1
1 1	0

Table 1: Truth Table for $L(x)$

This function is actually very common, so much so that it has a name: it's called the exclusive-OR (XOR) function. We write it as $L = \overline{x} \cdot y + x \cdot \overline{y} = x \oplus y$. The symbol is given by:

Figure 9: XOR gate symbol

Example: Two-Bit Adder

The XOR gate allows us to build a circuit to add binary numbers, known as an adder circuit. Since we are considering only two digits in the sum, $S = s_1 s_0$, we more specifically call it a two-bit adder.

a	b	s_1	s_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

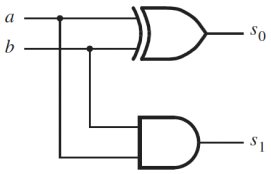


Figure 10: Truth Table and Logic Network for a Two-Bit Adder

2.5 Boolean Algebra

George Boole published the algebraic description of operations involved in logical thought and reasoning in 1849; this became known as Boolean Algebra. In the late 1930s, Claude Shannon showed that Boolean Algebra can describe circuits built with switches.

Axioms of Boolean Algebra

Like any other algebra, Boolean algebra is founded on axioms, or rules that we derive from basic assumptions. First we assume that Boolean algebra involves elements that take on only two values: 0 and 1. Then, we assume the following axioms are true:

1. $0 \cdot 0 = 0$
2. $1 + 1 = 1$
3. $1 \cdot 1 = 1$
4. $0 + 0 = 0$
5. $0 \cdot 1 = 1 \cdot 0 = 0$
6. $1 + 0 = 0 + 1 = 1$
7. If $x = 0$, then $\overline{x} = 1$
8. If $x = 1$, then $\overline{x} = 0$

Single-Variable Theorems

From the above axioms we can set some rules to deal with single variable expressions. These rules are more commonly called theorems. Therefore, if x is a Boolean variable, then the following theorems will hold true:

1. $x \cdot 0 = 0$
2. $x + 1 = 1$
3. $x \cdot 1 = x$
4. $x + 0 = x$
5. $x \cdot x = x$
6. $x + x = x$
7. $x \cdot \overline{x} = 0$
8. $x + \overline{x} = 1$
9. $\overline{\overline{x}} = x$

These theorems may be easily proved by showing that they hold true for $x = 1$ and $x = 0$.

Duality

Given a logic expression, its dual is obtained by replacing all AND operators with OR operators and vice versa. The importance of this concept will become apparent later in this chapter, when we show that every logic function can be expressed in at least two different ways.

Two- and Three-Variable Properties

1. $x \cdot y = y \cdot x$
2. $x + y = y + x$
3. $x \cdot (y \cdot z) = (x \cdot y) \cdot z$
4. $x + y \cdot z = (x + y) \cdot (x + z)$
5. $x + x \cdot y = x$ – Absorption
6. $x \cdot (x + y) = x$
7. $x \cdot y + x \cdot \overline{y} = x$ – Combining
8. $(x + y) \cdot (x + \overline{y}) = x$
9. $\overline{x \cdot y} = \overline{x} + \overline{y}$ – DeMorgan's Theorem
10. $\overline{x + y} = \overline{x} \cdot \overline{y}$
11. $x + \overline{x} \cdot y = x + y$
12. $x \cdot (\overline{x} + y) = x \cdot y$
13. $x \cdot y + y \cdot z + \overline{x} \cdot z = x \cdot y + \overline{x} + z$ – Consensus
14. $(x + y) \cdot (y + z) \cdot (\overline{x} + z) = (x + y) \cdot (\overline{x} + z)$

Example

We will prove the validity of the equation:

$$(x_1 + x_3) \cdot (\overline{x_1} + x\overline{x_3}) = x_1 \cdot \overline{x_3} + \overline{x_1} \cdot x_3$$

We can manipulate the left side using the distributive property as follows:

$$LHS = (x_1 + x_3) \cdot x_1 + (x_1 + x_3) \cdot \overline{x_3}$$

Applying the distributive property again:

$$LHS = x_1 \cdot \overline{x_1} + x_3 \cdot \overline{x_1} + x_1 \cdot \overline{x_3} + x_3 \cdot \overline{x_3}$$

By SVT 7, we have:

$$LHS = 0 + x_3 \cdot \overline{x_1} + x_1 \cdot \overline{x_3} + 0$$

Now we have:

$$LHS = x_1 \cdot \overline{x_3} + \overline{x_1} \cdot x_3 = RHS$$

2.5.1 Precedence of Operations

We use parentheses to indicate the order in which operations are performed. To limit the use of parentheses we establish a precedence of operations: NOT, AND, then OR.

2.6 Synthesis Using AND, OR, and NOT Gates

We will not try to implement some functions using the three gates. Suppose we want to synthesize a function that follows this truth table:

x_1	x_2	$f(x_1, x_2)$
0	0	1
0	1	1
1	0	0
1	1	1

Table 2: A Function to be Synthesize

We can find an AND expression for each line of the table, and connect them using OR operators. Thus, we can realize f as:

$$\begin{aligned} f(x_1, x_2) &= x_1x_2 + \overline{x_1}\overline{x_2} + \overline{x_1}x_2 \\ &= x_1x_2 + \overline{x_1}\overline{x_2} + \overline{x_1}x_2 + +\overline{x_1}x_2 \\ &= (x_1 + \overline{x_1})x_2 + \overline{x_1}(\overline{x_2} + x_2) \\ &= 1 \cdot x_2 + \overline{x_1} \cdot 1 \\ &= x_2 + \overline{x_1} \end{aligned}$$

This example highlights two things. First, a straightforward implementation is obtained by using the product term for each row of the truth table where the function equals 1. Second, there are multiple networks that can realize a given function, and some are much simpler and easier to implement than others.

2.6.1 Sum-of-Products and Products-of-Sums Forms

With an introduction into the synthesis process, we can now present it in more formal terminology. This is also where we'll explore how duality applies in the synthesis process.

Minterms

For a function of n variables, a product term in which each of the n variables appears once is called a minterm. These variables may be either complemented or uncomplemented, it only matters that they're present in the minterm.

Row	$x_1\ x_2\ x_3$	Minterm	Maxterm
0	0 0 0	$m_0 = \overline{x_1}\overline{x_2}\overline{x_3}$	$M_0 = x_1 + x_2 + x_3$
1	0 0 1	$m_1 = \overline{x_1}\overline{x_2}x_3$	$M_1 = x_1 + x_2 + \overline{x_3}$
2	0 1 0	$m_2 = \overline{x_1}x_2\overline{x_3}$	$M_2 = x_1 + \overline{x_2} + x_3$
3	0 1 1	$m_3 = \overline{x_1}x_2x_3$	$M_3 = x_1 + \overline{x_2} + \overline{x_3}$
4	1 0 0	$m_4 = x_1\overline{x_2}\overline{x_3}$	$M_4 = \overline{x_1} + x_2 + x_3$
5	1 0 1	$m_5 = x_1\overline{x_2}x_3$	$M_5 = \overline{x_1} + x_2 + \overline{x_3}$
6	1 1 0	$m_6 = x_1x_2\overline{x_3}$	$M_6 = \overline{x_1} + \overline{x_2} + x_3$
7	1 1 1	$m_7 = x_1x_2x_3$	$M_7 = \overline{x_1} + \overline{x_2} + \overline{x_3}$

Table 3: Three-variable minterms and maxterms

Sum-of-Products Form

A function f can be represented as a sum of minterms that correspond to each row in its truth table for which $f = 1$. The resulting implementation is functionally correct, but is not necessarily the lowest-cost implementation. A logic expression consisting of product (AND) terms that are summed (OR'd) together is in its sum-of-products form. If each of the OR'd terms is a minterm, then the expression is in a **canonical sum-of-products form**.

Minterms with row-number scripts can be used to specify a given function in a more concise form. For example, we can describe a function as:

$$f(x_1, x_2, x_3) = \sum(m_1, m_4, m_5, m_6)$$

or even more simply as:

$$f(x_1, x_2, x_3) = \sum m(1, 4, 5, 6)$$

Maxterms

The principle of duality says that it is possible to synthesize a function f by considering the rows of a truth table for which $f = 0$. This approach considers maxterms, which are the complement of minterms.

Product-of-Sums Form

If a function f is specified by a truth table, then it's complement $\overline{f}$ can be represented by a sum of minterms for which $\overline{f} = 1$, or, where $f = 0$.For example, consider:

$$\begin{aligned}\overline{f}(x_1, x_2) &= m_2 \\ &= x_1\overline{x_2}\end{aligned}$$

We can use De Morgan's theorem to complement this expression, the result is:

$$\begin{aligned}\overline{\overline{f}} = f &= \overline{x_1\overline{x_2}} \\ &= \overline{x_1} + x_2\end{aligned}$$

A logic expression consisting of sum (OR) terms that are factors of a product (AND'd together) is called a product-of-sum form. If each of the OR terms is a maxterm, then the expression is called a canonical product-of-sums.

We can use a shorthand notation as an alternative way of specifying a sample function:

$$f(x_1, x_2, x_3) = \Pi(M_0, M_1, M_2, M_7)$$

or more simply:

$$f(x_1, x_2, x_3) = \Pi M(0, 1, 2, 7)$$

The Π sign denotes a logical product operation.

2.7 NAND and NOT Logic Networks

The NAND and NOR functions are obtained by complementing the output of the AND and OR functions respectively. We place a bubble on the output side of the AND and OR gate symbols to get the new gate symbols, see below. Below is DeMorgan's theorem in terms of logic gates.

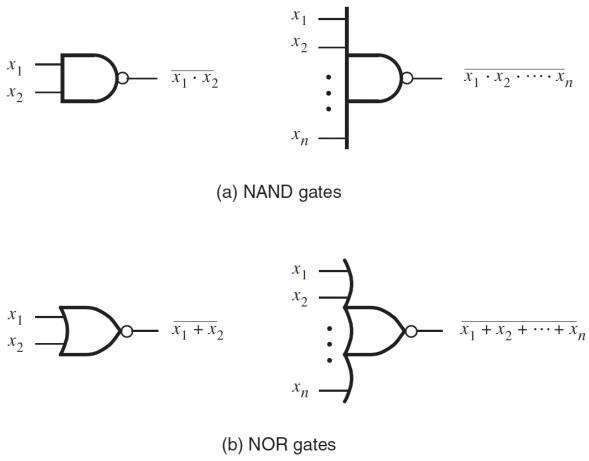


Figure 11: NAND and NOR gates

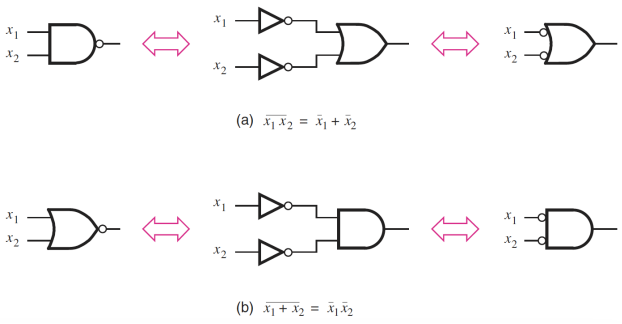


Figure 12: DeMorgan's theorem in terms of logic gates

2.8 Design Examples

2.8.1 Three-Way Light Control

We have a large room with three doors and a switch by each door. Flipping any one switch will turn the light on and off. We'll first define x_1, x_2, x_3 as our input variables that denote the state of each light switch. The light will be on if exactly one switch is closed, and off if two (or zero) switches are closed. If the light is off when two switches are closed, then it must be possible to turn it on by closing the third switch. The canonical sum-of-products expression for the specified function is:

$$\begin{aligned} f &= m_1 + m_2 + m_4 + m_7 \\ &= \overline{x_1}\overline{x_2}x_3 + \overline{x_1}x_2\overline{x_3} + x_1\overline{x_2}\overline{x_3} + x_1x_2x_3 \end{aligned}$$

Unfortunately, this circuit cannot be simplified into a lower-cost SOP expression. However, we can also write this function in product-of-sums form. The expression would be:

$$\begin{aligned} f &= M_0 \cdot M_3 \cdot M_5 \cdot M_6 \\ &= (x_1 + x_2 + x_3)(x_1 + \overline{x_2} + \overline{x_3})(\overline{x_1} + x_2 + \overline{x_3})(\overline{x_1} + \overline{x_2} + x_3) \end{aligned}$$

2.8.2 Multiplexer Circuit

Sometimes, we want to design a system that chooses to output data from one of multiple sources; this is a circuit that outputs the same as either input x_1 or x_2, dependent on a third input s. We will assume that the circuit should output x_1 when $s = 0$ and x_2 when $s = 1$. We can derive the canonical sum of products:

$$f(s, x_1, x_2) = \overline{s}x_1\overline{x_2} + \overline{s}x_1x_2 + s\overline{x_1}x_2 + sx_1x_2$$

This equation may simplified to:

$$\begin{aligned} f &= \overline{s}x_1(\overline{x_2} + x_2 + s(\overline{x_1} + x_1)x_2 \\ &= \overline{s}x_2 \cdot 1 + s \cdot 1x_2 \\ &= \overline{s}x_1 + sx_2 \end{aligned}$$

2.8.3 Number Display

In this example, we want to design a logic circuit to drive a seven-segment display, that will allow us to show S as a decimal number, 0, 1, or 2.

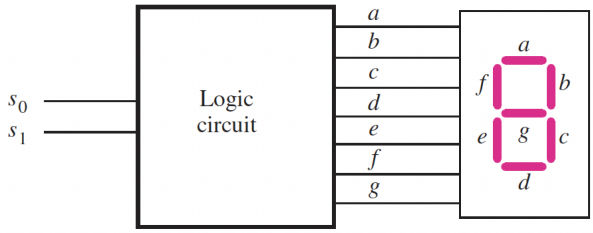


Figure 13: Logic Circuit to Seven-Segment Display

	s_1	s_0	a	b	c	d	e	f	g
0	0	0	1	1	1	1	1	1	0
1	0	1	0	1	1	0	0	0	0
2	1	0	1	1	0	1	1	0	1

Figure 14: Truth Table for Display

Logic expressions for each of the seven functions:

$$\begin{aligned} a &= d = e = \overline{s_0} \\ b &= 1 \\ c &= \overline{s_1} \\ f &= \overline{s_1 s_0} \\ g &= s_1\overline{s_0} \end{aligned}$$

2.9 Introduction to CAD Tools

Logic circuits found in complex systems cannot be designed manually – they are designed using CAD tools that implement synthesis techniques.

2.9.1 Schematic Capture

A CAD tool used to enter a designed circuit this way is called a schematic capture tool; a schematic refers to a diagram of a circuit in which circuit elements are depicted as graphical symbols and connections are drawn as lines.

A schematic capture tool allows the user to draw a schematic diagram. The tool provides a collection of symbols, called a library, to represent various gates and elements. Designers often employ hierarchical design, which means creating a circuit that includes smaller circuits inside of it as a method of design.

2.9.2 Hardware Description Language

An HDL tool is similar to a computer programming language except that HDLs are used to describe hardware rather than a program to be executed on a computer. There are two HDLs that are endorsed as IEEE standards, Verilog HDL, and VHDL (Very High Speed Integrated Circuit Hardware Description Language). We will use Verilog in these notes.

2.10 Introduction to Verilog

Verilog was created as part of an effort to develop standard design practices for digital circuits. It was originally intended for simulation and verification of digital circuits.

2.10.1 Structural Specification of Logic Circuits

A gate is represented by indicating its functional name, outputs, and inputs. For example, a two-input AND gate with output y and inputs x_1 and x_2 is denoted as:

$$\textbf{and}\ (y, x_1, x_2);$$

A four-input OR gate is denoted by:

$$\textbf{or}\ (y, x_1, x_2, x_3, x_4);$$

Keywords **nand** and **nor** define the NAND and NOR gates respectively. The NOT gate is given by:

$$\textbf{not}\ (y, x);$$

A logic circuit is defined as a **module**, which contains the statements that define the circuit.

```
1    module example1 (x1, x2, s, f);
2        input x1, x2, s;
3        output f;
4
5        not (k,s);
6        and (g, k, x1);
7        and (h, s, x2);
8        or (f, g, h);
9
10   endmodule
```

Verilog Syntax

Names of modules and signals in Verilog must start with a letter, and can contain any letter and number, including the "$" and "_" characters, and Verilog is case sensitive. We use semicolons to indicate end of statements, and whitespaces are ignored. Comments begin with // and continue to the end of the line.

2.10.2 Behavioural Specification of Logic Circuits

We can also use "&" and "—" (vertical bar) as AND and OR operators. The tilde "~" negates the signal immediately following it. The **assign** keyword provides a continuous assignment for a signal. Whenever any signal on the right side changes, the value of the signal on the left will be re-evaluated and will change as well. We also have **if-else** statements:

```
    if (s==0)
        f=x1;
    else
        f=x2;
```

Verilog syntax requires that procedural statements (like if-else statements) be contained within **always** blocks. Always blocks evaluate statements in the order specified in the code, in contrast to the continuous assignment statements. If a signal is being assigned a value using procedural statements, we must declare it as a variable using the **reg** keyword.

```
    module example5 (input x1, x2, s, output reg f);

        always@(x1, x2, s)
            if (s == 0)
                f = x1;
            else
                f = x2;

    endmodule
```

2.10.3 Hierarchical Verilog Code

For larger designs, it is often convenient to create a hierarchical structure in the Verilog code, in which there is a top-level module that includes multiple instances of lower-level modules.

See any lab past Lab 2 for examples of hierarchical code.

2.11 Minimization and Karnaugh Maps

In general, it is not obvious how to go from a SoP or PoS implementation of a function to a simpler, reduced method. This section introduces Karnaugh maps, which are a more manageable approach to produce a minimum-cost logic expression. The key to this approach is that it allows the application of the Combining property as judiciously as possible.

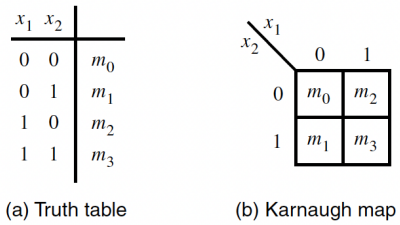


Figure 15: Location of Two-Variable Minterms

The advantage of the Karnaugh map is that it allows us to easily recognize minterms that can be minimized using the Combining property. Minterms in any two adjacent cells can be combined. The Karnaugh map can be used to directly derive a minimum-cost circuit for a logic function.

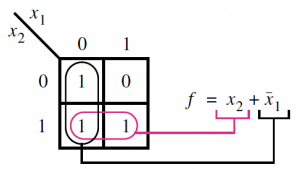


Figure 16: How to Read a Two-Variable Map

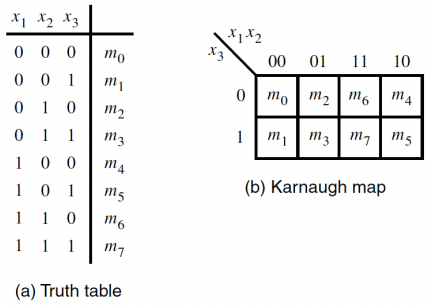


Figure 17: How to Read a Three-Variable Map

The Karnaugh map is a simple mechanism to generate the product terms that are used to implement a given function. A product term must include only those variables that have the same value for all cells in the group represented by that term. If that variable is 0, then it's complemented.

We can also build a four-variable Karnaugh map.

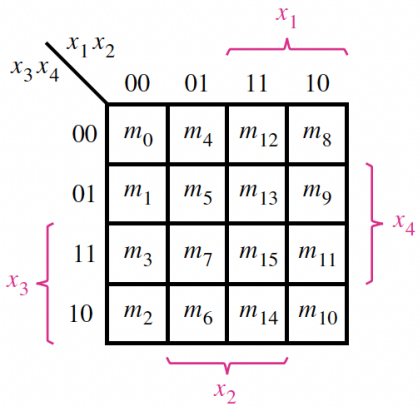


Figure 18: How to Read a Four-Variable Karnaugh Map

2.12 Strategy for Minimization

Our strategy in the previous section was to find few and large groups of variables that cover all cases where the function equals 1. For larger logic circuits, however, we should have a more organized method to derive a minimum-cost implementation.

2.12.1 Terminology

Lots of research has gone into developing techniques for synthesis of logic functions. Certain terminology has evolved to address the need to repeatedly describe highly-used phrases.

Literal – Each appearance of a variable in a product term, either complemented or not, is called a literal.

Implicant – A product term that indicates the input which equals 1 for a function is called an implicant.

Prime Implicant – A prime implicant is one that cannot be combined into another implicant with fewer literals. Another way to put it is that we cannot get an equal implicant if we remove any literals from the minterm.

Essential Prime Implicant – If a prime implicant includes a minterm for which $f = 1$ that is not included in another prime implicant, then it must be included in any cover, and is thus called an essential prime implicant.

Cover – A collection of implicants that account for all possible inputs for which a function will equal 1. The cover consisting of all prime implicants will lead to the lowest-cost implementation.

Cost – The sum of the number of gates and the total number of inputs to all gates in the circuit.

2.12.2 Minimization Procedure

1. Generate all prime implicants for the given function.
2. Find the set of essential prime implicants.
3. If the set of prime implicants covers all inputs for which $f = 1$, then the set is a good cover for f. If not, add the non-essential prime implicants to make a complete cover.

2.12.3 Incompletely Specified Functions

In digital systems it oftentimes occurs that certain input conditions can never occur. If we have two interlocked switches, x_1 and x_2, then the input valuations $(x_1, x_2) = 00, 01, 10$ can occur, but 11 is guaranteed to never occur in real life. Then, we can say that $(x_1, x_2) = 11$ is a don't care condition, meaning that a circuit with these inputs can be designed by ignoring this condition. A function with don't care conditions is known as incompletely specified. We can represent these in shorthand by:

$$f(x_1, ..., x_4) = \sum m(2, 4, 5, 6, 10) + D(12, 13, 14, 15)$$

where D is the set of don't cares.

Although don't care values can be assigned arbitrarily, such an assignment may not lead to a minimum-cost implementation of a given function. With k don't cares, there are 2^k ways to assign 0 or 1 values to them.

3 Number Representation and Arithmetic Circuits

3.1 Positional Number Representation

Trivial, see Chapter 1.

3.2 Addition of Unsigned Numbers

One-bit addition entails four possible combinations for two inputs, and we need two bits to represent the result of this addition. The right bit is called the sum, s, and the left bit is called the carry, c (think about the carry over in decimal addition).Such a circuit is called a **half-adder**.

When we deal with addition with multiple bits involved, for each bit position i, we need to include the carry in from position $i-1$. The optimal sum-of-products realization for the carry-out function is:

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

For the sum, the sum-of-products realization is:

$$s_i = \overline{x}_i y_i \overline{c}_i + x_y \overline{y}_i \overline{c}_i + \overline{x}_i \overline{y}_i c_i + x_i y_i c_i$$

To make things far more simple, we can just use an XOR operation:

$$s_i = x_i \oplus y_i \oplus c_i$$

At this point, we will notice that the operation $\overline{x \oplus y}$ appears quite frequently, so much so that we will give it a distinct name XNOR. We use the symbol $\odot$ to represent this operation. We sometimes call it the coincidence operation because it returns 1 when its inputs coincide in value (both 0s or both 1s).

3.2.1 Decomposed Full-Adder

We can construct a full-adder by compounding two half-adders. See the decomposition below:

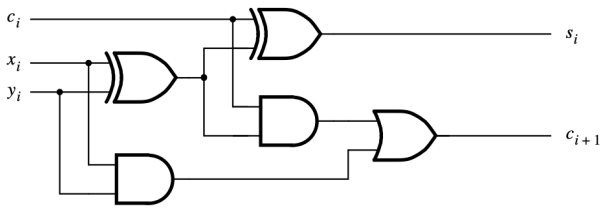


Figure 19: Gate-Level Implementation of a Full-Adder

3.2.2 Ripple-Carry Adder

Start at the least-significant digit and add pairs of digits. If we produce a carry in bit position i, we add it to the operands in $i+1$. We use a full-adder for each bit position, and call it the "ripple-carry" adder because the carry ripples from the least to most significant bits.

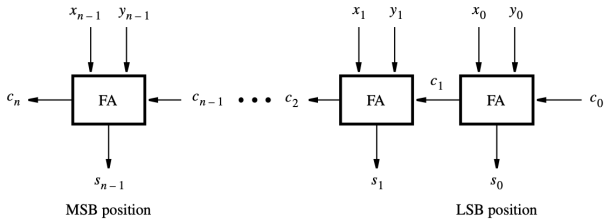


Figure 20: An n-bit Ripple Carry Adder

4 Combinational-Circuit Building Blocks

4.1 Multiplexers

A multiplexer circuit has several inputs, one or more select inputs, but only one output.

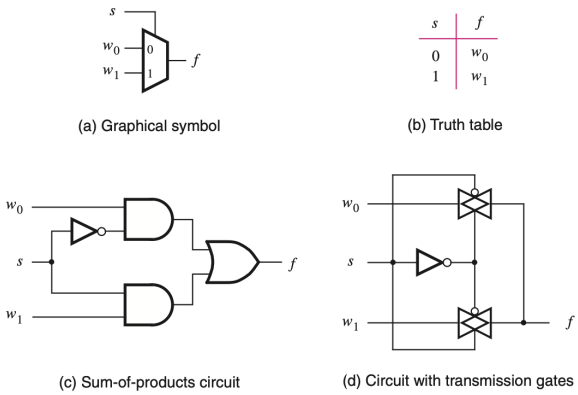


Figure 21: 2-to-1 Multiplexer

A multiplexer with n data inputs requires $\log_2 n$ select inputs.

4.1.1 Synthesis of Logic Functions Using Multiplexers

Section mainly describes how one can realize several logic functions using multiplexers. Two particularly interesting examples are included below.

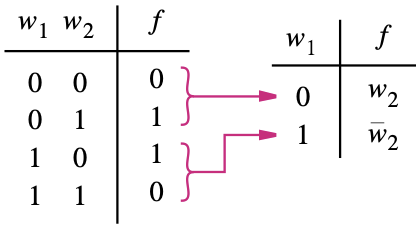


Figure 22: 2-input 4-to-1 Multiplexer

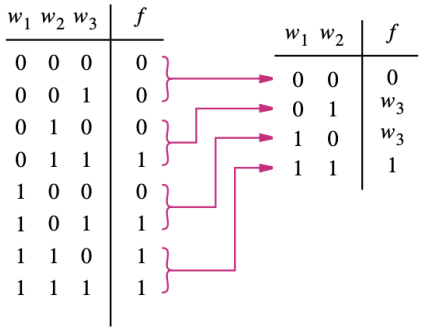


Figure 23: Majority Function Using a 3-input 4-to-1 Multiplexer

4.1.2 Multiplexer Synthesis Using Shannon's Expansion

Consider the circuit given by the following diagram: We may realize the sum-of-products form of this circuit

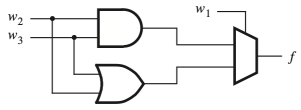


Figure 24: Sample Circuit

as:

$$f = \overline{w}_1 w_2 w_3 + w_1 \overline{w}_2 w_3 + w_1 w_2 \overline{w}_3 + w_1 w_2 w_3$$

which can be manipulated into:

$$\begin{aligned} f &= \overline{w}_1 (w_2 w_3) + w_1 (\overline{w}_2 w_3 + w_2 \overline{w_3} + w_2 w_3) \\ &= \overline{w}_1 (w_2 w_3) + w_1 (w_2 + w_3) \end{aligned}$$

Shannon's Expansion Theorem – Any Boolean function $f(w_1, ..., w_n)$ can be written in the form:

$$f(w_1, w_2, ..., w_n) = \overline{w}_1 \cdot f(0, w_2, ..., w_n) + w_1 \cdot f(1, w_2, ..., w_n)$$

We may also expand this theorem for expressions of multiple variables.

4.2 Decoders

Not covered as of the midterm assessment.

4.3 Encoders

Not covered as of the midterm assessment.

4.4 Code Converters

Not covered as of the midterm assessment.

4.5 Arithmetic Comparison Circuits

Not covered as of the midterm assessment.

4.6 Verilog for Combinational Circuits

4.6.1 The Conditional Operator

For simple implementation of choosing between multiple signals, Verilog has a conditional operator (?:) which assigns one of two values depending on a conditional expression.

```
1    conditional_expression ? true_expression: false_expression
```

Here is a more concrete example:

```
1    A= (B < C) ? (D + 5) : (D + 2);
```

We can implement the conditional operator in both continuous assignment statements and in procedural argument statements using the **always** block. Here is some example code:

```
module mux_2to1 (
    input w0, w1, s,
    output reg f);

    always @ (w0, w1, s)
        f = s ? w1 : w0;

endmodule
```

4.6.2 The If-Else Statement

We have already seen the if-else statement. Remember to delineate a block of statements using **begin** and **end**.

4.6.3 The Case Statement

When there are many combinations to implement in an if-else statement, we may use the Verilog **case** statement.

```
case (expression)
    alternative1 : statement;
    alternative2 : statement;
    alternative3 : statement;
    ...
    [default : statement;]
endcase
```

When there are don't cares present in the logic of the expression, we should always include the default clause.

4.6.4 The For Loop

Not covered as of this midterm.

4.6.5 Verilog Operators

Operator type	Operator symbols	Operation performed	Number of operands
Bitwise	~	1's complement	1
	&	Bitwise AND	2
	|	Bitwise OR	2
	^	Bitwise XOR	2
	~^ or ^~	Bitwise XNOR	2
Logical	!	NOT	1
	&&	AND	2
	||	OR	2
Reduction	&	Reduction AND	1
	~&	Reduction NAND	1
	|	Reduction OR	1
	~|	Reduction NOR	1
	^	Reduction XOR	1
	~^ or ^~	Reduction XNOR	1
Arithmetic	+	Addition	2
	-	Subtraction	2
	-	2's complement	1
	*	Multiplication	2
	/	Division	2
Relational	>	Greater than	2
	<	Less than	2
	>=	Greater than or equal to	2
	<=	Less than or equal to	2
Equality	==	Logical equality	2
	!=	Logical inequality	2
Shift	>>	Right shift	2
	<<	Left shift	2
Concatenation	{,}	Concatenation	Any number
Replication	{{}}	Replication	Any number
Conditional	?:	Conditional	3

Figure 25: Verilog Operators

4.6.6 The Generate Construct

Not covered as of this midterm.

4.6.7 Tasks and Functions

Not covered as of this midterm.

5 Flip-Flops, Registers, and Counters

Till this point, we have considered combinational circuits where the value of each output depends solely on the values of the inputs. However, we can construct circuits where the output values depend on the present and past behaviour of the circuit. When input values change, the circuit is left as-is, or it is changed to a new output state. These circuits are called sequential circuits.

We will finish this preface with a motivating example. Suppose we want to control an alarm system which responds to the control input. The alarm is meant to turn on when the sensor generates a positive voltage, *Set*, in response to some undesirable event. Once triggered, the alarm must stay active even after the sensor reads zero. We will require a *Reset* input to turn off the alarm manually, and the circuit requires a memory element to keep the alarm active until the *Reset* signal is applied.

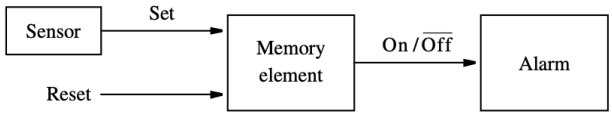


Figure 26: Control Flow of the Alarm System

5.1 Basic Latch

We will construct a circuit with two cross-connected NOR gates, called a **basic latch**. Understand the behaviour of the circuit at different values of input signals.

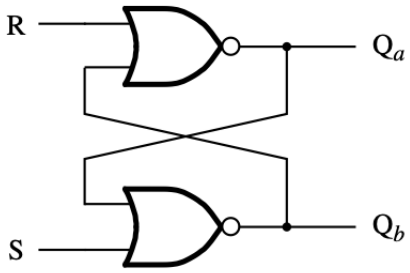


Figure 27: Basic Latch Circuit Diagram

We can construct a truth table-style representation of this circuit; we call it a **characteristic table** over a truth table because the values of the output may be determined by the past states of the circuit, not just present states.

S	R	Q_a	Q_b	
0	0	0/1	1/0	(no change)
0	1	0	1	
1	0	1	0	
1	1	0	0	

Figure 28: Basic Latch Characteristic Table

Now, let us study the timing diagram:

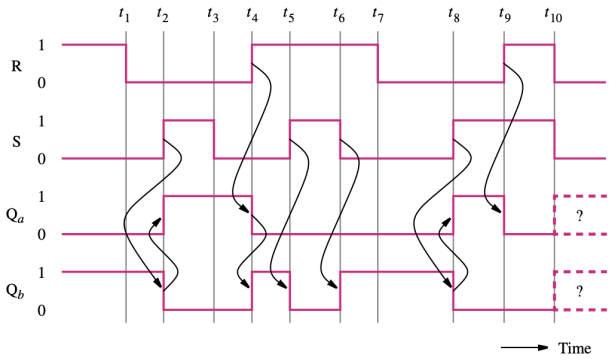


Figure 29: Basic Latch Timing Diagram

Note that if the Set and Reset signals are both made inactive at the same time, we will have an interesting situation where $Q_a = Q_b = 1$. This forces $Q_a = Q_b = 0$, which in turns forces the former; we will enter into a state where both signals are oscillating indefinitely in response to the other.

5.2 Gated SR Latch

In the above section, we were able to create a circuit that remembers its state when both S and R are 0. We can add functionality by adding a control input, which we will call Enable. When this input is disabled, changing Set will not cause the output state to change. In this textbook, we will instead call this signal Clock because such circuits are used in situations where the user wants to allow the changes to output only at specified time intervals. We will abbreviate the Clock signal as Clk.

Circuits employing a control signal are called gated latches, and since the circuit we are interested in also includes a Set and Reset capability, we can classify it further as a gated SR latch. We see that as with the

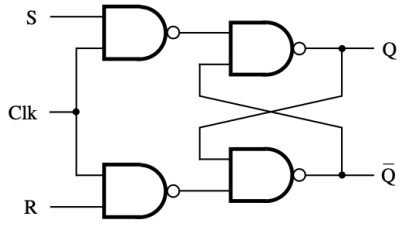


Figure 30: Gated SR Latch Circuit Diagram with NAND Gates

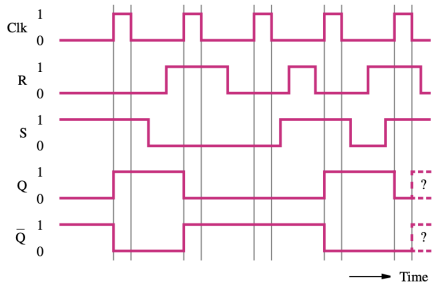


Figure 31: Timing Diagram for a Gated SR Latch

SR Latch, when both Set and Reset are made 1, the circuit enters an undefined state. In this circuit, we must avoid the possibility that both S and R are 1 when the Clk is changed from 1 to 0, or the circuit will enter an undefined state.

5.3 Gated D Latch

In this section, we will further increase the practicality of our circuit by implementing a single data input D and a Clk.

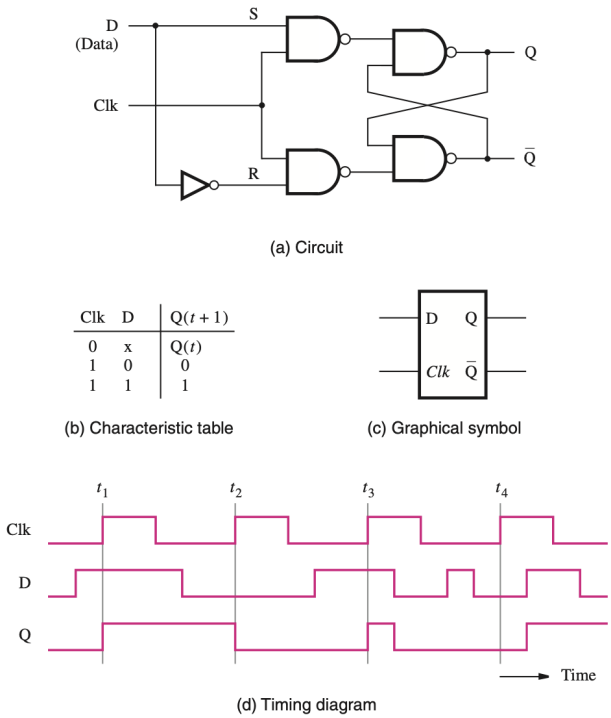


Figure 32: Relevant Information for a Gated D Latch

To state simply the behaviour of this circuit, we can say that while Clk is on, the output tracks the input D, and stores the last tracked value of D when Clk is off.

Up to this point, all of the circuits in this Chapter have been **level sensitive**, that is, the output is controlled by the level of the Clk. We will soon discuss circuits where the output is changed when the clock changes from one value to another; these circuits are called **edge triggered**.

5.3.1 Effects of Propagation Delays

We have previously neglected the effects of propagation delays, and we will continue to do so for now. We will discuss this in more details in Chapter 7.

5.4 Edge-Triggered D Flip-Flops

5.4.1 Master-Slave D Flip-Flop

Consider a circuit with two gated D latches in series, where the output of the first latch is the input of the second. The first latch, called the **master**, changes its state when Clk = 1, and the second, called the **slave**, changes its state when Clk = 0.

When the clock is high, the master tracks the value of D and the slave remains constant. When the clock flips to 0, the master stops tracking changes to D, and the slave changes the output by tracking the output of the master instead.

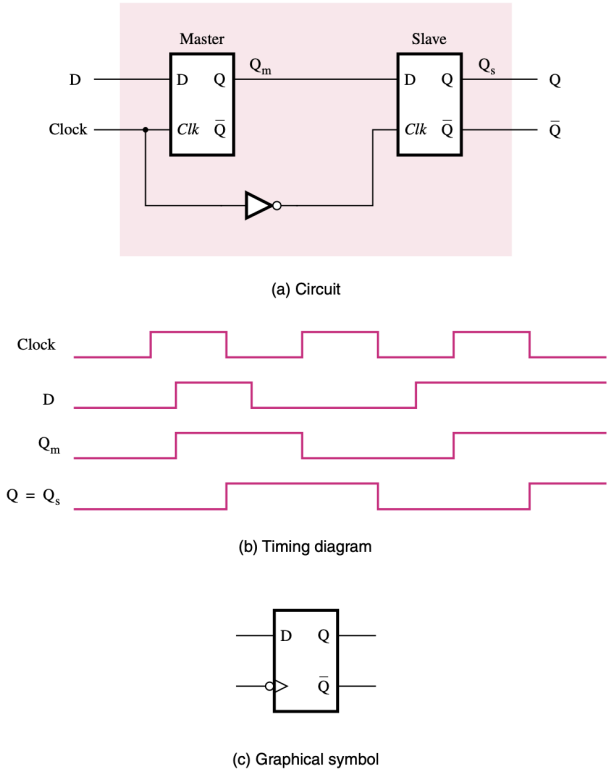


Figure 33: Master-Slave D Flip-Flop

5.4.2 Other Types of Edge-Triggered D Flip-Flops

We may also reverse the Clk input to the flip-flop so we have a negedge triggered flip-flop instead.

5.4.3 D Flip-Flops with Clear and Reset

We will start this section with a circuit diagram. One way to add the clear and preset capability is to add

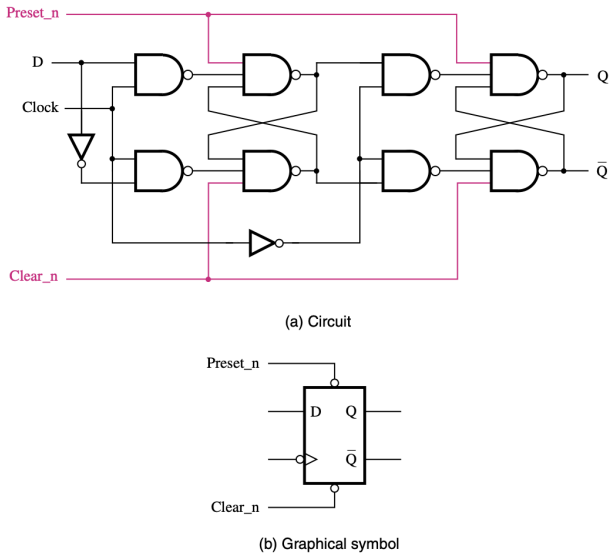


Figure 34: Master-Slave D Flip-Flop with Clear and Preset

an extra input to each NAND gate in the cross-coupled latches. Both signals are low-active, meaning active when their value is 0, so we append _n. When $Clear_n = 0$, we will force the FF into the state $Q = 0$, and when $Clear_n = 1$, it will have no impact on the NAND gates. Conversely, $Preset_n = 0$ forces the flip-flop into the state $Q = 1$, while $Preset_n = 1$ has no effect.

In the above example, the $Clear_n$ signal clears the flip-flop regardless to the clock signal; this is known as an **asynchronous clear**. In practice, however, we like to clear the flip-flop on the active edge of the clock. We can implement a **synchronous clear** using the circuit given below:

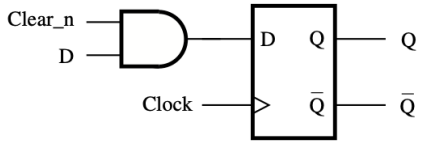


Figure 35: A Flip-Flop with Synchronous Clear

5.5 Toggle Flip-Flop

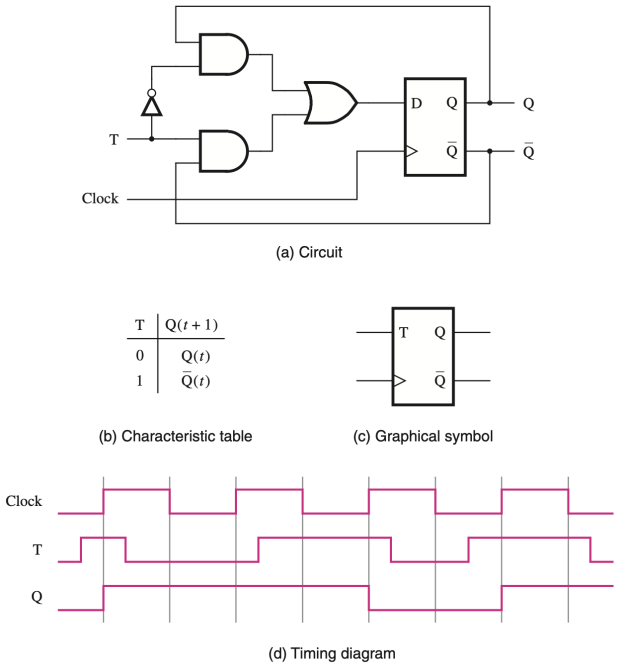


Figure 36: Toggle Flip-Flop

We may further modify the D flip-flop by adding a level of control to the input. D will equal either Q or $\overline{Q}$ depending on the value of T. If $T = 0$ then we have $D = Q$ and conversely, $T = 1$ gives us $D = \overline{Q}$.

5.6 JK Flip-Flop

Not covered as of the midterm.

5.7 Summary of Terminology

A **basic latch** is a feedback connection or two NOR or NAND gates, which stores a bit of information. It's set to 1 using S and reset to 0 using R.

A **gated latch** is a basic latch which includes input gating and a control input. We can change its state when the Clk is 1. We have discussed two types of gates latches:

1. A **gated SR latch** uses S and R input signals to set it to 1 or 0 respectively.

2. A **gated D latch** uses the D to force a latch into a state that has the same logic value as the D input.

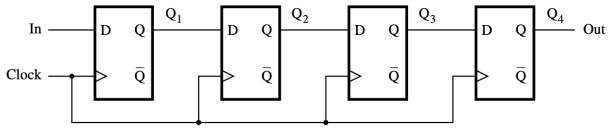
A **flip-flop** is a storage element that can change its output only on the edge of the Clk signal.

5.8 Registers

A flip-flop stores only 1 bit of information, we can use n FFs to store an n-bit number. Thus, we have a register.

5.8.1 Shift Register

In an earlier section, we saw that we can multiply a number by 2 if we shift each bit position to the left by one, and insert a 0 in the now-empty right-most bit position. This type of register, with the ability to shift its contents, is called a **shift register**. We must use flip-flops to implement a shift register, not level-sensitive gated latches, because a change in the input would propagate through multiple latches when the Clk is 1.



(a) Circuit

	In	Q_1	Q_2	Q_3	Q_4 = Out
t_0	1	0	0	0	0
t_1	0	1	0	0	0
t_2	1	0	1	0	0
t_3	1	1	0	1	0
t_4	1	1	1	0	1
t_5	0	1	1	1	0
t_6	0	0	1	1	1
t_7	0	0	0	1	1

(b) A sample sequence

Figure 37: A Simple Shift Register

5.8.2 Parallel-Access Shift Register

We can transfer n bits at once using n separate wires, which would be a parallel transfer. This is in contrast to the serial transfer that is given above.

The 2-to-1 multiplexer on the D input allows each FF to be connected to two sources: the preceding flip-flop, which is needed for the shift-register operation, and the external input that corresponds to the bit to be loaded into the FF as part of the parallel-load operation. We use $\overline{Shift}/Load$ to select the mode of the operation. If $\overline{Shift}/Load = 0$, the circuit operates as a shift-register, and if $\overline{Shift}/Load = 1$, the parallel input data is loaded into the register. Note that irrespective of the mode of operation, the action will take place at the positive edge of the clock.

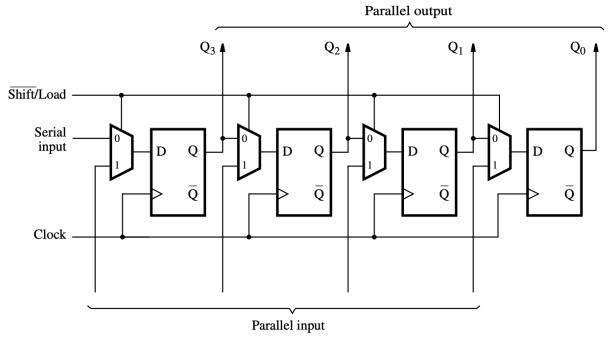


Figure 38: Parallel-Access Shift Register

5.9 Counters

5.9.1 Asynchronous Counters

The simplest counter circuit can be built using multiple T flip-flops because the toggle is well-suited to implement the counting operation.

Up-Counter with T Flip-Flops

The clock input of the three flip-flops are connected in cascade, meaning the $\overline{Q}$ output of each FF becomes

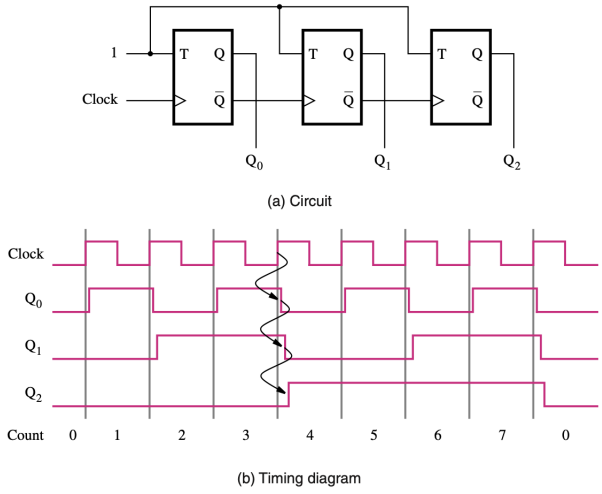


Figure 39: 3-Bit Up Counter

the Clk input of the next (except for the first FF). The value of Q_0 toggles once each per clock cycle, specifically, on the posedge of Clk. Since the second FF is clocked by $\overline{Q}_0$, the signal Q_1 is toggled by the negedge of Q_0. Lastly, Q_2 is toggled by the negedge of Q_1. Because of the rippling behaviour of the circuit in the diagram above, we often refer to it as an asynchronous counter or a **ripple counter**.

Down-Counter with T Flip-Flops

We can slightly modify the circuit above such that the output Q of each FF becomes the clock of the next; this gives us a down-counter.

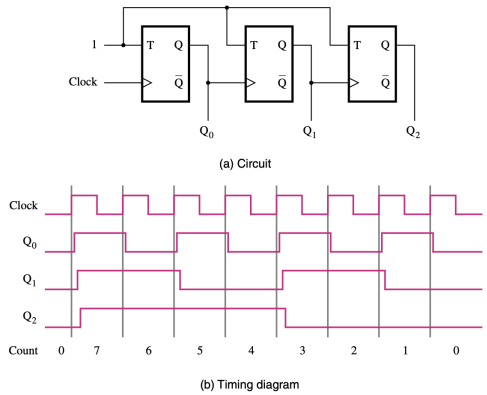


Figure 40: 3-Bit Down Counter

5.9.2 Synchronous Counters

We can build a faster counter by clocking all the flip-flops at the same time.

Synchronous Counter with T Flip Flops

For an n-bit up-counter, a given FF only changes its state when every preceding FF has output 1. So, it follows that if we use toggle FFs to realize our counter, we can define the T inputs as:

$$T_0 = 1$$
$$T_1 = Q_0$$
$$T_2 = Q_0 Q_1$$

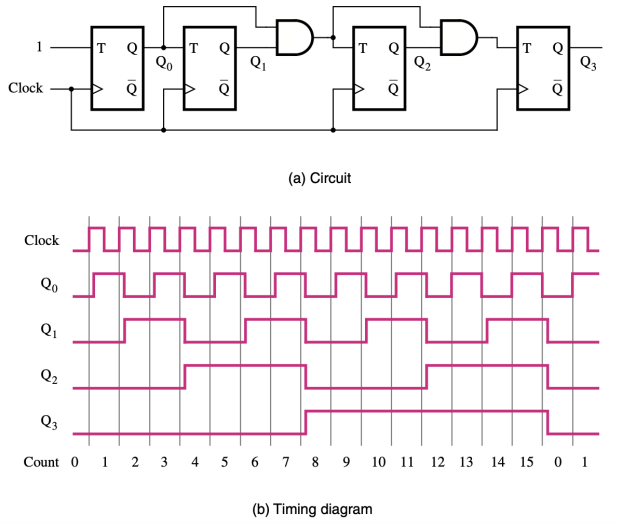


Figure 41: A 4-Bit Synchronous Counter

Clearly, we see that all changes take place with the same propagation delay (after the Clk signal). Thus, we have a synchronous counter.

Enable and Clear Capability

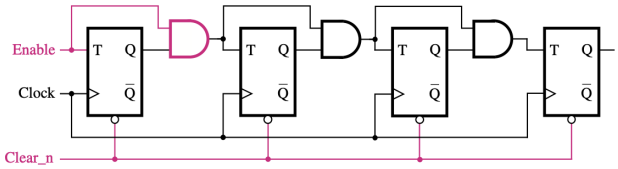


Figure 42: A Synchronous Counter with Enable and Clear Capability

Synchronous Counters with D Flip Flops

Assume we want to build a counter with FF outputs $Q_3Q_2Q_1Q_0$. Assuming we set $Enable = 1$, we can find the D inputs of the FFs as:

$$D_0 = Q_0 \oplus 1 = \overline{Q}_0$$
$$D_1 = Q_1 \oplus Q_0$$
$$D_2 = Q_2 \oplus Q_1Q_0$$
$$D_3 = Q_3 \oplus Q_2Q_1Q_0$$

and we can generalize from there.

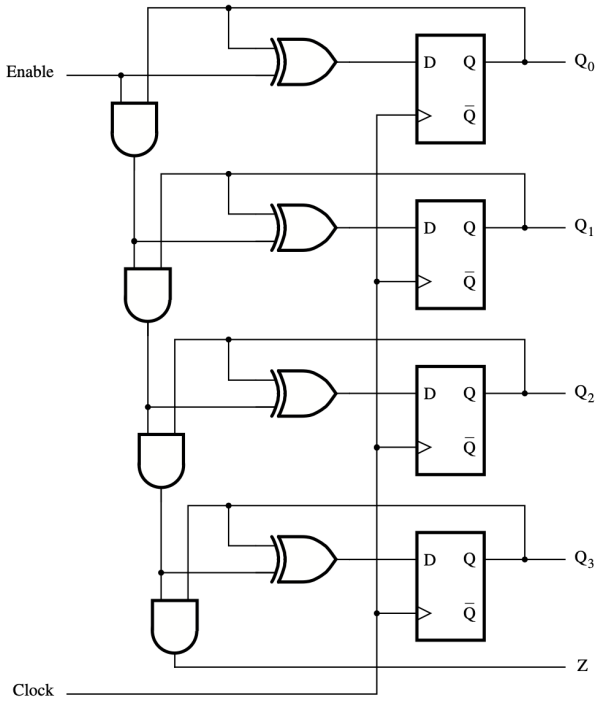


Figure 43: A 4-Bit Counter with D Flip Flops

6 Synchronous Sequential Circuits

Circuits in which the outputs depend on the past behaviour of the circuit are called sequential circuits. Where a clock is used to control its operation, these circuits are called **synchronous sequential circuits**, else they are **asynchronous sequential circuits**.

Sequential circuits are also called finite state machines (FSMs) in technical literature; this name is derived from the fact that these circuits can be represented in a number of finite states.

6.1 Basic Design States

Sequential circuits are used to control the operation of physical systems.

6.1.1 State Diagram

The first step is to determine how many states are needed, and what transitions are possible between states. While there is no set procedure for how to do this, a good way is to select a starting state, which is the state when the machine is turned on or a Reset signal is applied. From there, we consider each next state depending on what input is applied.

Once we have defined what transitions are possible between the different states, we can construct what's known as a state diagram.

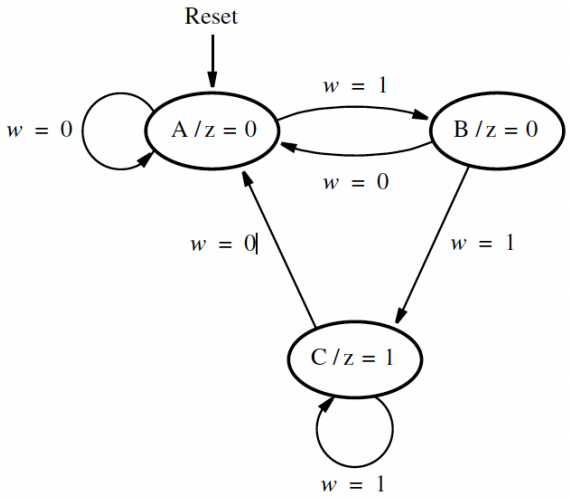


Figure 44: Sample State Diagram for a Sequential Circuit

6.1.2 State Table

A state table indicates all transitions from a present state to the next state for different input values.

Present state	Next state		Output
	$w = 0$	$w = 1$	z
A	A	B	0
B	A	C	0
C	A	C	1

Figure 45: State Table for the Above State Diagram

6.1.3 State Assignment

Each state is represented by a particular valuation of **state variables**. Each state variable is implemented as a flip-flop. We can make a variety of choices for the number of FFs used. Y_1 and Y_2 are next-state

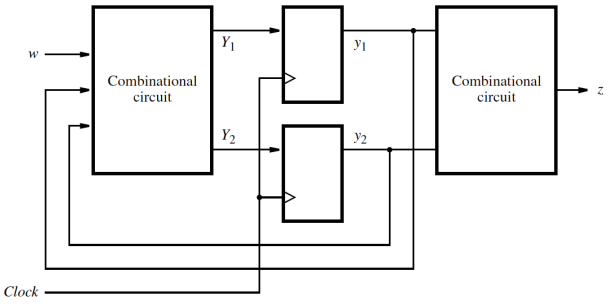


Figure 46: General Sequential Circuit with Two FFs

	Present state	Next state		Output
		$w = 0$	$w = 1$	
	$y_2 y_1$	$Y_2 Y_1$	$Y_2 Y_1$	z
A	00	00	01	0
B	01	00	10	0
C	10	00	10	1
	11	*dd*	*dd*	*d*

Figure 47: State-Assigned Table

variables and y_1 and y_2 are present-state variables.

6.1.4 Summary of Design Steps

1. Obtain the specification of the desired circuit.
2. Derive the states for the machine starting with an initial state. Consider each input valuation and create new states for the machine to respond appropriately.
3. Create a state table from the state diagram.
4. Minimize the number of states if necessary.
5. Decide on the number of state variables needed to represent all states and perform the state assignment.
6. Choose the type of flip-flops to be used in the circuit.
7. Implement the circuit as indicated by the logic expressions.