Department of Engineering
Australian National University

ENGN3213

Digital Systems & Microprocessors

Part I: Digital System Design and Synthesis

V3.0
Copyright 2008 ANU Engineering
3.6.2 Noise Margin ........................................... 37
3.6.3 Fan Out ............................................... 38
3.6.4 Transition Time ..................................... 39
3.6.5 Propagation Delays ................................ 39
3.6.6 Power .................................................. 40
3.6.7 Speed-Power Product ................................. 40
3.6.8 Another Example of a CMOS Logic Gate ............. 40
3.7 Programmable Logic ..................................... 41
   3.7.1 Field Programmable Gate Arrays .................. 43
   3.7.2 Applications of FPGAs ............................. 43
   3.7.3 FPGA Architecture ................................. 44
4 Combinational Logic - Review ............................ 46
   4.1 Combinational Logic ................................. 46
   4.2 Boolean Algebra ...................................... 46
      4.2.1 Boolean Variables ............................... 46
      4.2.2 Boolean Functions .............................. 47
      4.2.3 Gate Outputs .................................. 50
      4.2.4 Unused Gate Inputs .............................. 51
      4.2.5 Theorems of Boolean Algebra .................... 51
      4.2.6 De Morgan’s Laws in Terms of Gates ............. 54
   4.3 Boolean Functions and Digital Circuits ............... 54
      4.3.1 Universality of NAND and NOR .................. 54
      4.3.2 Asserted Levels ................................ 56
      4.3.3 Alternative Logic Gate Representations ........... 57
      4.3.4 Canonical Forms ................................ 57
   4.4 Simplification and Implementation of Boolean Functions .. 60
      4.4.1 Direct Implementation ........................... 60
      4.4.2 Karnaugh Maps (K-Maps) ......................... 61
      4.4.3 Generating Gray Code for n Bits .................. 66
      4.4.4 SOP and POS representations ...................... 66
6.1.8 Additional Inputs ........................................... 106
6.1.9 D Flip-Flops with Clear and Preset .......................... 106
6.1.10 T flip flops ........................................... 108
6.1.11 JK Flip-Flops ........................................ 109
6.1.12 Triggering ........................................... 111
6.2 A Practical Examples of the Use of Flip Flops .............. 111
6.2.1 A Parallel in Parallel Out Register ....................... 111
6.2.2 Serial in Serial Out Shift Register ....................... 112
6.2.3 Parallel-Access Shift Register ............................ 115
6.2.4 Counters .......................................... 118
6.3 Timing and Metastability .................................. 118
6.4 Summary of Latch and Flip Flop Truth Tables .............. 124

7 Finite State Machines ........................................ 125
7.1 What is a Sequential State Machine? Examples ............... 125
  7.1.1 Examples on the ENGN3213 Web Page .................... 125
7.2 Diagrams ................................................ 126
7.3 Formal Definition of Finite State Machines ................ 126
7.4 State Diagrams and Tables ................................ 128
  7.4.1 State Diagrams ........................................ 128
  7.4.2 Next State Table ...................................... 130
  7.4.3 Timing Diagrams ...................................... 131
7.5 Implementing Finite State Machines using Flip Flops .......... 131
  7.5.1 Characteristic Equations ................................ 131
  7.5.2 Flip-flops as Finite State Machines .................... 132
7.6 Synthesis of Sequential Machines .......................... 135
  7.6.1 Canonical Implementation .............................. 135
  7.6.2 Design Steps ......................................... 135
  7.6.3 High Level Representations and Binary Coding .......... 136
7.7 Example - Up/Down/Stop Counter ........................... 136
7.8 Example - Vending Machine Controller ...................... 139
7.8.1 Vending Machine Circuit Schematic .................................. 141
7.9 Dealing with unused states ............................................. 143
7.10 Equivalence and Redundancy .......................................... 144
7.11 Using VERILOG for State Machine Design .......................... 144
7.12 Some Applications of State Machines in Telecommunications .... 145
  7.12.1 The Convolutional Encoder ....................................... 145
  7.12.2 Analysis of the Convolutional Encoder .......................... 146
  7.12.3 Synthesis of the Convolutional encoder ........................ 148
  7.12.4 Some Convolutional Encoder Terminology ...................... 151
  7.12.5 Convolutional Encoder Timing Diagram ......................... 152
  7.12.6 The Ethernet (IEEE 802.3) MAC State Machine ............... 153
  7.12.7 The WiFi (IEEE 802.11b) MAC State Machine ................. 155
7.13 Practical Aspects of Sequential Machine Design ................... 156
  7.13.1 Asynchronous Inputs ............................................. 157
  7.13.2 Clock Skew .................................................... 158
  7.13.3 Switch Debouncing .............................................. 159
  7.13.4 Initialisation and Reset ....................................... 161

A Appendix: VERILOG Syntax .............................................. 165
  A.1 Lexical Tokens ................................................... 165
     A.1.1 White Space .................................................. 165
     A.1.2 Comments ..................................................... 165
     A.1.3 Verilog Keywords ........................................... 165
  A.2 Data Types ........................................................ 165
     A.2.1 Wire .......................................................... 166
     A.2.2 Reg ............................................................ 167
     A.2.3 Input, Output, Inout ....................................... 167
     A.2.4 Integers ..................................................... 168
     A.2.5 Supply0, Supply1 ............................................ 168
     A.2.6 Time .......................................................... 168
     A.2.7 Parameter ................................................... 168
A.3 Number Specification .................................................. 169
  A.3.1 Negative numbers .............................................. 170
A.4 Operators ................................................................. 171
  A.4.1 Arithmetic Operators ........................................... 171
  A.4.2 Relational Operators ........................................... 171
  A.4.3 Bit-wise Operators ............................................... 172
  A.4.4 Logical Operators ............................................... 172
  A.4.5 Reduction Operators ........................................... 173
  A.4.6 Shift Operators ................................................... 173
  A.4.7 The Concatenation Operator .................................. 173
  A.4.8 The Replication Operator .................................... 174
  A.4.9 The Conditional Operator: "?" ............................... 174
A.5 Operands ................................................................. 174
  A.5.1 Literals ............................................................. 174
  A.5.2 Bit-Selects "x[3]" and Part-Selects "x[5:3]" ............... 175
A.6 Gates ................................................................... 175
  A.6.1 Basic Gates ......................................................... 175
  A.6.2 buf, not Gates ..................................................... 176
A.7 Behavioural VERILOG Syntax ......................................... 177
  A.7.1 Delay in Assignment (not for synthesis) ..................... 177
  A.7.2 Blocking Assignments .......................................... 178
  A.7.3 Nonblocking (RTL) Assignments ............................... 178
  A.7.4 begin...end ........................................................... 181
  A.7.5 For loops ............................................................ 182
  A.7.6 while Loops ........................................................ 183
  A.7.7 forever Loops ...................................................... 183
  A.7.8 disable .............................................................. 184
  A.7.9 if ... else if ... else .......................................... 184
  A.7.10 case ............................................................... 185
A.8 Timing Controls ....................................................... 186
  A.8.1 Delay Control (Not synthesizable) ......................... 186
A.8.2 Event Control, @ .................................. 187
A.9 Procedures ............................................ 187
  A.9.1 Always Blocks .................................... 187
  A.9.2 Initial Blocks (not for synthesis) ............. 188
A.10 Functions ............................................. 189
  A.10.1 Function Return Value ......................... 190
  A.10.2 Function Call .................................... 190
  A.10.3 Function Rules .................................. 190
A.11 Compiler Directives ................................. 190
  A.11.1 Time Scale ...................................... 190
  A.11.2 Macro Definitions ............................... 191
  A.11.3 Include Directive ............................... 191
A.12 System tasks and functions ....................... 191
  A.12.1 $display, $monitor ............................. 192
  A.12.2 $time, $stime, $realtime ...................... 192
1 Introduction

1.1 General

The course will cover the design of sequential digital systems using FPGA (programmable logic devices) and microprocessors. The languages Verilog and C will be used. Students will have the opportunity to exercise their creativity in some of the lab design assignments and to gain experience in the implementation of state of the art digital processing systems used in telecommunications.

Digital systems are widely used, and their use is ever-increasing. Witness the explosion in the uses of computers, and the development of digital communications systems, etc. Modern digital design involves the use of a range of devices, from discrete logic chips (NAND, NOR, etc), to more complex logic chips (adders, counters, flip-flops, etc), to hardware programmable logic devices and memory arrays (PLD, ROM, etc), and ultimately to software programmable microprocessors and microcontrollers. This course will teach you the basics of how digital hardware and microprocessors work and how to design digital systems to solve engineering problems. You will learn to use CAD software, and to implement a design in hardware and software. The course will teach you about microprocessors and programmable logic devices (Field Programmable Gate Arrays), how to write software for them, and to gain first hand experience in their use in telecoms. As a result, you will also gain exposure to certain telecom signal processing techniques. After completing this course, you will be in a position to decide on the best way to solve a problem, using hardware and/or software.

This course is considerably more design-oriented than most of the other Engineering courses you will have taken to date. In particular, the course involves hands-on design lab assignments. You will find it lots of fun!

A student may reasonably ask “why am I being required to do this course?” There are several reasons:

- The vast majority of electronic systems nowadays contain digital electronics. All digital computers naturally use this technology. In this course you will be exposed to the fundamentals of digital systems, sufficient for you to understand the approach used in designing and integrating digital systems. Examples of such systems include mobile phones, wireless LANs, modern telephone exchanges, digital video broadcast systems, CD players, robotic systems, traffic control systems, mp3 players, fax machines, GPS and telemetry systems etc.

- The subject contains design lab assignments, which expose you to the synthesis part of engineering (rather than analysis). Much of the material you cover in a university engineering course is analysis, whereas when you graduate you will be called upon to design and redesign systems, as well as analyse them.
• You will learn how to think clearly about sequential processes (finite state automata), and develop facility with manipulating formal representations of them. This is important, because any task that has more than a couple of states is very hard to reason about correctly without the appropriate conceptual tools.

• You will also be exposed to and use a real-world computer aided design tools. In particular will be concentrating on some free and open source tools.

The course reflects current industry practice and recent trends in using hardware description languages for hardware design, simulation and sophisticated EDA software, programmable logic ICs, microprocessors and microcontrollers, and indeed, the increasingly blurred distinction between hardware and software in modern digital systems. In this course the focus will be on hardware design, software design, and combining them to build digital systems.

Hardware/software used in the course:

• A simple PIC programmer for MICROCHIP PICS.
• The Small Device C Compiler (SDCC) for embedded microprocessors.
• Xilinx FPGA board (programmable logic).
• Xilinx WebPACK ISE 7.1i.
• ICARUS open source Verilog simulator.

This particular reading brick covers the material on digital systems, including the basics, combinational and sequential logic, finite state machines, VERILOG HDL and programmable logic devices. If time permits there will be a second reading brick which treats the microprocessor and C programming aspects of the course.

1.2 Course Objectives

The most important outcome for this course is for students to be able to analyse digital circuits and to synthesise this in hardware to meet stated specifications.

Student will also,

• Master fixed point and binary number arithmetic.
• Analyse finite state machines and register transfer language systems.
• Learn to design circuits that implement finite state machines in programmable logic.
• Obtain experience in dealing with complex digital systems in programmable logic.
• Translate system requirements into a practical digital design.
• Learn how to program in C and VERILOG HDL.
• Learn about the resources and EDA software available to the open source community.
• Learn practical electronics testbench skills and the ability to communicate appropriately via a lab notebook.

1.3 Timetable

The course consists of two lectures per week, one computer lab and one hardware lab. The computer labs will also serve as tutorials where quizzes will be set and assessed.

<table>
<thead>
<tr>
<th>Class</th>
<th>M</th>
<th>T</th>
<th>W</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL1</td>
<td>12-14</td>
<td></td>
<td></td>
<td>CHEM-G51B</td>
</tr>
<tr>
<td>CL2</td>
<td></td>
<td>11-13</td>
<td>CHEM-G51B</td>
<td></td>
</tr>
<tr>
<td>CL3</td>
<td></td>
<td></td>
<td>12-14</td>
<td>CHEM-G51B</td>
</tr>
<tr>
<td>LA</td>
<td>9-10</td>
<td></td>
<td></td>
<td>ENGN-T</td>
</tr>
<tr>
<td>LB</td>
<td></td>
<td>9-10</td>
<td></td>
<td>ENGN-T</td>
</tr>
<tr>
<td>HL1</td>
<td></td>
<td>14-17</td>
<td></td>
<td>IRLAB-103</td>
</tr>
<tr>
<td>HL2</td>
<td></td>
<td>14-17</td>
<td></td>
<td>IRLAB-103</td>
</tr>
<tr>
<td>HL3</td>
<td></td>
<td></td>
<td>14-17</td>
<td>IRLAB-103</td>
</tr>
</tbody>
</table>

1.4 Resources

It is clear from the above list of open source or free software items that engineers need only consider open source software to work in digital design. As a result, you will be free to work on digital design and implementation projects in your leisure without the need for University infrastructure. A CDROM containing the above software items and examples can be made available for loan to enable students to install software on their home PCs but unable to download it from the web.

The course has a web page:

http://engnet.anu.edu.au/Dcourses/engn3213/
1.5 Assessment and Work Policies

Assessment for this course is as follows.

<table>
<thead>
<tr>
<th>Assessment</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tute Quizzes</td>
<td>10%</td>
</tr>
<tr>
<td>Lab Notebooks</td>
<td>30%</td>
</tr>
<tr>
<td>Exam</td>
<td>20%</td>
</tr>
<tr>
<td>Project</td>
<td>40%</td>
</tr>
</tbody>
</table>

There will be a computer lab every week in which half the time (approx) will be spent answering a short quiz for marking and half the time discussing course questions and computer software exercises. The sum total of these quizzes is 10% of the course and is the best indicator of the style of question to be expected in the exam.

In the labs there will be design, implementation and test exercises. These are assessed by marking of the lab notebook each week. Lab demonstrators will not trapse through untidy work looking for your working. Scrappy lab books will be marked down (if marked at all).

It should be clear that absence from computer and hardware labs is not an option.

A large part of the assessment for this course will involve a project. You will have all semester to complete the project and you may do it with other students. All projects offered will require deep knowledge of the course material and will almost certainly introduce new ideas outside the scope of the course. Assessment will be in the form of a thirty page project report document. More information on the project will arrive shortly.

To solve the tute quizzes, labs and make a sensible attempt at the project, you are encouraged to work with other students in the ENGN3213 class ONLY.

You must, however, document any of the help you receive in the form of comments directly on your homework paper or in your lab notebook as appropriate. No comments mean you are submitting the item as totally your own work; my assumption will always be that you are an honorable person unless you cause me to believe otherwise. Simply copying another person’s work is not allowed.
2 Number Systems

Binary digits are alien to real-life systems where most problems are based on either real numbers or no numbers at all. For example in digital telecommunications where radiofrequency (and therefore analogue) signals are recorded by a wireless receiver, digital signal processing must be performed to recover the data stream that is being transmitted. The signal processing that performs this task must at all stages use fixed point arithmetic. Floating point arithmetic of the type that you are used to in MATLAB, can only be performed on special DSP devices that may not meet the cost or performance requirements of the application. In fact, the lowest cost and highest performance devices are always the simplest in architecture. These are the Field Programmable Gate Arrays (FPGAs and other Complex Programmable Logic Devices (CPLDs)) which consist of little more than arrays of logic gates. Digital designers must be able to perform signal processing tasks such as Fourier Transformations in fixed point arithmetic.

In general, the digital designer needs to be able to establish some correspondence between binary digits or boolean logic and real world problems. In this chapter we review how numerical quantities can be represented and manipulated in digital systems.

2.1 Positional Number Systems

The decimal number system that we routinely use is a positional number system. In such a system an arbitrary real number can be approximated by,

$$X = d_p r^{p-1} + d_{p-2} r^{p-2} + ... + d_0 + d_{-1} r^{-1} ... + d_n r^{-n}$$  \hspace{1cm} (1)

or more succinctly,

$$X = d_p d_{p-2} ... d_0 . d_{-1} ... d_n$$ \hspace{1cm} (2)

where $0 \leq d_k < r$ are positive integers and $r > 1$ is the base or radix of the number system. The point is referred to as the radix point with whole numbers to the left and fractions to the right. If no radix point is shown then we assume that it is located at the extreme right. In this case, there are no fractional quantities. For the case of a binary radix, the leftmost digit is the high order or most significant bit (MSB) and the rightmost digit is the low order or least significant bit (LSB). In a decimal number ($r = 10$) the radix point is referred to as the decimal point and in a binary number ($r = 2$) the radix point is referred to as the binary point.
The following are some examples. Note the subscript indicating the radix.

\[10011_2 = 1 \times 16 + 0 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1 = 19_{10}\]

\[100010_2 = 1 \times 32 + 0 \times 16 + 0 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 = 34_{10}\]

\[101.001_2 = 1 \times 4 + 0 \times 2 + 1 \times 1 + 0 \times 0.5 + 0 \times 0.25 + 1 \times 0.0125 = 5.125_{10}\]

(3)

### 2.2 Octal and Hexadecimal Numbers

Radix 10 is obviously important. Radix 2 is the basis of binary which is important in computing. Octal (radix 8) and hexadecimal (radix 16) also widely used in digital systems. Table 1 shows the binary integers 0 to 1111 and their octal, decimal and hexadecimal equivalents.

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
<th>Octal</th>
<th>3-Bit String</th>
<th>Hexadecimal</th>
<th>4-Bit String</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>001</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>2</td>
<td>010</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>3</td>
<td>011</td>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
<td>4</td>
<td>100</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>101</td>
<td>5</td>
<td>5</td>
<td>101</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>110</td>
<td>6</td>
<td>6</td>
<td>110</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>111</td>
<td>7</td>
<td>7</td>
<td>111</td>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>10</td>
<td>--</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>11</td>
<td>--</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>12</td>
<td>--</td>
<td>A</td>
<td>1010</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>13</td>
<td>--</td>
<td>B</td>
<td>1011</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>14</td>
<td>--</td>
<td>C</td>
<td>1100</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>15</td>
<td>--</td>
<td>D</td>
<td>1101</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>16</td>
<td>--</td>
<td>E</td>
<td>1110</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>17</td>
<td>--</td>
<td>F</td>
<td>1111</td>
</tr>
</tbody>
</table>

The octal system needs 8 digits so it uses the digits 0-7. The hexadecimal system needs 16 digits and so uses the numerals 0-9 and the letters A-F. Because their radices are powers of two, octal and hexadecimal numbers are used to conveniently aggregate binary numbers. Any string of three binary digits (i.e. 1’s or 0’s) can represent 8 possible
numbers. Thus a 3-bit string can be represented uniquely by an octal digit. Likewise, a 4-bit string can be represented by one hexadecimal digit.

Thus it is very easy to convert a binary number to octal or hexadecimal. Consider the case of hexadecimal. Consider first the case where the binary point is at the extreme right (and is therefore not written at all). Start at the extreme right and group the binary digits into 4-strings. Translate each 4-string into a hex digit according to the above table. For example consider,

\[ 100011001110_2 = 8CE_{16} \]  

(4)

Note that sometimes you will have to add zeros to the left in the case of Octal to have an integer multiple of three digits and in the case of Hex to have an integer multiple of four digits.

**Question** What do you need to do if there the numbers are not integers?

### 2.3 Addition and Subtraction of Non-Decimal Numbers

Addition and subtraction of non-decimal numbers occurs in exactly the same way as for decimal numbers. Addition/subtraction table 2 shows the case of binary numbers. In the table \( C_{in} \) and \( B_{in} \) are the carry in and borrow in bits for addition and subtraction respectively, the operation being performed is \( x \pm y \), \( s \) and \( d \) are the sum and difference respectively and \( c_{out} \) and \( b_{out} \) are respectively the carry and borrow out bits.

<table>
<thead>
<tr>
<th>( C_{in} ) or ( B_{in} )</th>
<th>x</th>
<th>y</th>
<th>( C_{out} )</th>
<th>s</th>
<th>( B_{out} )</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Exercise.** Verify the addition/subtraction table and perform the following arithmetic operations: (a) \( 10111110 + 10001101 \) and \( 11100101 - 00101110 \).
2.4 Representation of Negative Numbers

2.4.1 Signed Magnitude Representation

In everyday situations we use the signed magnitude representation of negative numbers, wherein a minus sign precedes a number to indicate that it is negative. Such a system could be used in the case of the binary number system as well, but we need some way to represent a minus "−" sign on a computer. The sign is represented by adding one more bit to the left (i.e. on the high order end) of a binary number. Traditionally a "1" bit is used to represent minus and a "0" bit is used to represent a plus. This approach has the advantage that the number system has an equal number of positive and negative numbers. It has the major disadvantage that extra digital logic is required to process the signs before performing arithmetic.

2.4.2 The Radix-Complement Representation: Introduction

Whereas the signed-magnitude system negates a number by changing its sign, a complement number system negates a number by taking its complement as defined by the number system at hand. Such a methodology for producing signed numbers exploits the “modulo” nature of a fixed bit length representation of a number. Taking the complement is more difficult than simply negating a number as it produces a completely different looking number for the negative version of a positive number, however subtraction operations become easier to implement. We will concentrate on a complement method called the radix complement.

Assume that we have a fixed number of bits and assume that the radix is $r$ and that numbers have the form:

$$X = d_{p-1}d_{p-2}...d_0$$ (5)

We will assume that $X$ is an integer because most of the arithmetic that we will be doing will be with integers. In a radix complemented system, the complement of a $p$-digit number is obtained by subtracting it from $r^p$. In the decimal number system, the radix complement is referred to as 10's complement. Some examples for the case of 5 decimal integers where $10^5 = 10000$ are shown in table 3.

The radix complement of an $p$-digit number $X$ is obtained by subtracting it from $r^p$. If $X$ is between 1 and $r^p - 1$, then so too is $r^p - X$. If $X = 0$ then the radix complement is $r^p$ which has the form 100...00. We throw away the high order bit so that there is only one representation of zero in a radix complement system.

The usefulness of the radix complement representation lies in how we deal with zero. Since $r^p = 0$, it follows that $X + (r^p - X) = 0$: thus $r^p - X$ is an additive inverse of $X$ in the number system.
2.4.3 The Radix-Complement Representation: Another look

One problem with the above definition of the radix complement negative number is that a subtraction operation is involved. We can avoid this subtraction by rewriting \( r^p \) as \( (r^p - 1) + 1 \) and \( r^p - X \) as \( ((r^p - 1) - X) + 1 \). Since \( r^p - 1 \) has the form \( mmm...mm \) where \( m = r - 1 \) is the maximum digit in the number system, \( r^p - X \) is therefore a string of the complements of each digit of \( X \). The radix complement of \( X \) can therefore be obtained by complementing each digit of \( X \) and adding 1. Here is an example. Consider the radix complement of the four digit number, 7610 in base 10. The number \( r^p - 1 \) is 9999. The complement of each digit of 7610 is 2389. The radix complement of 7610 is therefore 2390. Check it out: 7610 + 2390 = 10000 = 0.

Table 4 lists the digit complements for binary, octal, decimal and hexadecimal numbers.

2.4.4 Two’s-Complement Representation

For binary numbers the radix complement is called the two’s complement. A number is negative iff its MSB (Most Significant Bit) is unity. In order to compute the decimal equivalent of a two’s complement number we proceed in the usual fashion if the MSB is 0 (positive number). When the MSB is a 1, we weight it as \(-2^{p-1}\) keeping the usual weight for the remaining digits. For example consider \( 17_{10} = 00010001_2 \) in 8-bit binary. To find \(-17_{10}\) in two’s complement binary, complement the bits to obtain \(11101110_2\) and add 1 to obtain, \(11101111_2\). To convert the latter back consider first the bits that represent the digits (the last 7 bits). These give the number 111. The weight of the sign bit is \(-2^7 = -128\) and \(-128 + 111 = -17\).

The special case is that of \( 0_{10} = 00000000_2 \). The complement is 11111111 and adding 1 we obtain 10000000. In this case we ignore the carry out to obtain 00000000.

Note that as a result of all this there is one more negative number \(-2^{p-1}\) than there are positive numbers. The number \(-2^{p-1}\) does not have a positive counterpart: VIZ \( 2^{p-1} \) does not exist. Check this out for \( p = 8 \) (8-bit binary). \( 2^7 = 128 = 10000000 \) which is \(-0\).

Table 3: Some examples of 10’s complement numbers.

<table>
<thead>
<tr>
<th>Number</th>
<th>10s Complement</th>
<th>9s Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1849</td>
<td>8151</td>
<td>8150</td>
</tr>
<tr>
<td>2067</td>
<td>7933</td>
<td>7932</td>
</tr>
<tr>
<td>100</td>
<td>9900</td>
<td>9899</td>
</tr>
<tr>
<td>7</td>
<td>9993</td>
<td>9992</td>
</tr>
<tr>
<td>8151</td>
<td>1849</td>
<td>1848</td>
</tr>
<tr>
<td>0</td>
<td>10000</td>
<td>9999</td>
</tr>
</tbody>
</table>
\(-128 = 0111111 + 1 = 10000000\). The minimum and maximum integer in 8-bit binary respectively are \(-128\) and \(127\) and \(p\)-bit integers range between \(-2^{p-1}\) and \(2^{p-1} - 1\).

Most computers use the two’s complement system to represent negative numbers and this is what we will use for designs in this course.

### 2.4.5 Addition and Subtraction in Two’s Complement

The main advantage of two’s complement over signed magnitude and other systems (for example one’s complement which I do not treat here) is that one can start with the minimum number \(-2^{p-1}\), and obtain all the numbers in the system in sequence by adding 1 provided only that we ignore carries beyond the MSB. That is two’s complement numbers can be added by ordinary binary addition, ignoring any carries beyond the MSB. The result will always be correct so long as the range of the number system is not exceeded.

Some examples of addition follow in table 5. Be careful to note where the carry out has occurred.

*Overflow* is said to occur if an operation produces a result which is outside the range of the number system. For 8-bit binary overflow occurs when we count past +7. *Addition of two numbers of different sign can never produce an overflow*, but addition of two numbers of like sign can as shown in the examples of overflow under binary addition in table 9.

Table 4: Digit complements for binary, octal, decimal and hexadecimal numbers.

<table>
<thead>
<tr>
<th>Digit</th>
<th>Binary</th>
<th>Octal</th>
<th>Decimal</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>9</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>6</td>
<td>8</td>
<td>E</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>5</td>
<td>7</td>
<td>D</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>4</td>
<td>6</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>3</td>
<td>5</td>
<td>B</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>2</td>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>1</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>0</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>A</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>5</td>
</tr>
<tr>
<td>B</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 5: Examples of two’s complement binary addition

| +3   | 0011 | -2   | 1110 |
| +4   | 0100 | -6   | 1010 |
| -     |      | -     |      |
| +7   | 0111 | -8   | 1000 |
| +6   | 0110 | +4   | 0100 |
| -3   | 1101 | -7   | 1001 |
| -     |      | -     |      |
| +3   | 0011 | -3   | 1101 |

Table 6: Examples of overflow in binary addition.

| -3   | 1101 | +5   | 0101 |
| -6   | 1010 | +6   | 0110 |
| -     |      | -     |      |
| -9   | 1011=+7 | -8   | 1011=-5 |

| -8   | 1000 | +7   | 0111 |
| -8   | 1000 | +7   | 0111 |
| -     |      | -     |      |
| -16  | 0000=+0 | +14  | 1110=-2 |
In addition, overflow has occurred \textit{if the addends’ signs are the same but the sum’s sign is different from the addends’}. Equivalently, addition overflow occurs if the carry bits $c_{in}$ into and $c_{out}$ out of the sign position are different.

Subtraction in two’s complement is not done on a computer using a rule analogous to the way humans do decimal subtraction. It is simpler to proceed as follows:

(a) Perform a bit by bit complement of the subtrahend.

(b) Add the subtrahend to the minuend with an initial carry of 1 instead of 0.

Overflow in subtraction cannot occur if the two numbers have the same sign to start with. On the other hand, if one of them is negative, then the same test as for addition can be applied. Compare the minuend and the complemented subtrahend. They should now have the same sign.

Examples are shown in table 7.

\begin{table}[h]
\begin{center}
\begin{tabular}{ccc}
+4 & 0100 & 0100 \\
+3 & 0011 & 1100 \\
--- & --- & --- \\
+1 & 0001 & -1 \\
\end{tabular}
\end{center}
\caption{Examples of two’s complement binary subtraction.}
\end{table}

\begin{table}[h]
\begin{center}
\begin{tabular}{ccc}
+3 & 0011 & 0011 \\
-4 & 1100 & 0011 \\
--- & --- & --- \\
+7 & 0111 & +1 \\
\end{tabular}
\end{center}
\caption{Examples of two’s complement binary subtraction.}
\end{table}

\subsection*{2.5 Binary Multiplication}

Binary multiplication is performed by adding a list of shifted multiplicands in the same way as you already know for for decimal arithmetic.

Instead of listing all the shifted multiplands and then adding, in a digital system it is more convenient to add each shifted multiplicand as it is created to a \textit{partial product}. Applying this procedure to the above example we obtain,
Table 8: Binary multiplication.

<table>
<thead>
<tr>
<th></th>
<th>11</th>
<th>1011</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td></td>
<td>33</td>
<td>1011</td>
</tr>
<tr>
<td></td>
<td>110</td>
<td>00000</td>
</tr>
<tr>
<td></td>
<td>143</td>
<td>101100</td>
</tr>
<tr>
<td></td>
<td>143</td>
<td>1011000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10001111</td>
</tr>
</tbody>
</table>

Table 9: Binary multiplication with partial products.

<table>
<thead>
<tr>
<th></th>
<th>11</th>
<th>1011</th>
<th>multiplicand</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>13</td>
<td>1101</td>
<td>multiplier</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0000</td>
<td>partial product</td>
</tr>
<tr>
<td></td>
<td>1011</td>
<td>0000</td>
<td>shifted multiplicand</td>
</tr>
<tr>
<td></td>
<td>01011</td>
<td>00000</td>
<td>partial product</td>
</tr>
<tr>
<td></td>
<td>001011</td>
<td>00000</td>
<td>shifted multiplicand</td>
</tr>
<tr>
<td></td>
<td>011011</td>
<td>101100</td>
<td>shifted multiplicand</td>
</tr>
<tr>
<td></td>
<td>00011111</td>
<td>101000</td>
<td>partial product</td>
</tr>
<tr>
<td></td>
<td>10001111</td>
<td>101000</td>
<td>shifted multiplicand</td>
</tr>
<tr>
<td></td>
<td>10001111</td>
<td>101000</td>
<td>product</td>
</tr>
</tbody>
</table>
In general, when we multiply an $n$-bit number by an $m$-bit number, the resulting product requires at most $n + m$ bits to express. This should be borne in mind we start doing fixed point arithmetic in VERILOG HDL.

The methodology for multiplying negative numbers is similar. We can also convert each number to positive two’s complement numbers first and then put the sign back at the end.
3 Physics of CMOS Devices

In this chapter we review the physics of semiconductor devices (esp. Complementary Metal Oxide Semiconductor devices or CMOS) in order to obtain an understanding of the basics of Metal Oxide Semiconductor (MOS) technology. We will glance at:

- Transistor manufacture.
- Transistor operation (we leave a discussion of the logic function of CMOS until the next chapter).
- The different varieties of programmable technology.

3.1 Background

An integrated circuit is made up of a tiny fragile but highly chemically processed silicon chip built on a silicon die. The chip is enclosed in a package that provides protection for the die and an array of interface pins that are bonded to electrodes on the die by wires. Figure 1 shows some typical packages that you are surely familiar with. In this course we will be looking at DIL (dual in line, also DIPs) packages for PICs \(^1\) such as the PIC16F84. DIL PICs can be easily inserted into ZIF sockets (Figure 2) for programming and IC sockets in case you don’t want to solder them into a PCB (print circuit board). You will obtain experience at soldering a complete PCB PIC programmer in the labs.

In addition in the course we will be using FPGAs (esp. the XC2S50 Xilinx FPGA that appears on the PEGASUS FPGA development board by Digilent Corporation). This is a Plastic Quad Flat Package (PQFP) surface mount device (SMD) with 208 pins and 0.5 mm lead spacing. Although one can obtain sockets for these, they are usually soldered into circuit.

3.2 Transistor Manufacture

The manufacturing process of the die from a silicon wafer is referred to as front-end processing while packaging and bonding is referred to as back end processing. A wire bonded die is shown in Figure 3 [6].

The processing starts with a circular wafer of raw silicon usually about 650 microns thick. Diameters are usually 125, 150 or 200 mm. The wafer is cut from a long cylindrical monocrystalline ingot (Figure 4) [1].

\(^1\)A PIC is a brand name for a very common type of microprocessor manufactured by MicroChip, the world’s second largest manufacturer of computer chips. Some say that PIC stands for “Programmable Interface Controller” but this is in fact just a nick name. PICs are the dominant species in the low power end of computer chips referred to as “microcontrollers.”
Figure 1: Some common IC packages. Note that the Dual in line (DIL) and PLCC packages can be mounted in sockets other packages need to be soldered [6].

Figure 2: A ZIF socket allows simple insertion and release of microcontroller chips from programmer circuits

Because Silicon is an insulator in pure crystalline form, processing has to be performed to introduce controlled conduction and insulation layers as well as metal wire tracks and contacts to connect the layers to each other. Producing these is the goal of front-end processing and usually involves numerous steps that consist of successive application and
reapplication of the following processes.

(1) *Photomasking* produces an imprint of the component traces to be engraved or coated on the silicon. The process is also referred to as *photolithography* and is very close to the same process used in the manufacture of PCBs where a photoresist is baked on the substrate and is exposed to ultraviolet light through the mask. Photomasking produces a hard coating on areas in the shadow of the UV and a soft resist crust in areas that can be etched away.

(2) *Etching* removes thin film material in order to exposed a wanted layer buried under the film. Two etching techniques are chemical or wet etching and plasma aided or dry etching.

(3) *Diffusion* is the process by which dopants (impurities) are introduced into the silicon to enhance its conductivity in a controlled way or to add an oxide (insulating) layer.
Diffusion is a thermal process involving gaseous impurities in a high temperature oven containing the silicon target.

(4) *Ion implantation* injects dopants into the silicon with monochromatic energy so that they arrive at a predetermined depth inside the silicon. Ion implantation is a more controlled process than diffusion.

(5) Metal deposition allows the realisation of metal connections.
At the end of processing, the die has to be passivated (coated) in order to prevent unwanted contamination from external sources.

The dopants are purposely injected into the silicon lattice to change its conductivity. In order to introduce negative carriers one uses Phosphorous or Arsenic. To introduce positive carriers, one frequently uses Boron. Cartoons of metal deposition, doping and etching processes are shown in Figure 5.

Figure 5: Doping, ion-implantation, metal deposition (diffusion) and etching processes [6].
3.3 MOS Transistor Operation

In this section we touch on the basics of CMOS technology. The presentation is very introductory, but hopefully it will go some way to demystifying the inner workings of CMOS devices.

3.3.1 MOS Technology

There are three major MOS technology families: PMOS (also referred to as p-channel devices: see Wakerly), NMOS (also referred to as n-channel: see Wakerly) and CMOS. The names refer to the channel type of the MOS transistors.

- **PMOS Technologies** implement p-channel transistors by diffusing p-type dopants into an n-type silicon substrate to form the source and the drain. See figure 6.

- **NMOS Technologies** implement n-channel transistors by diffusing n-type dopants into a p-type silicon substrate to form the source and the drain. See figure 6.

- **CMOS** (Complementary MOS) is a MOS technology in which both p-channel and n-channel devices are fabricated on the same die. Note from the figure 6 that a deep well of n-type material must be defined to allow fabrication of the complementary transistor type.

![Figure 6: The MOS technologies](image)

Figure 6 shows a close up view of an NMOS and a CMOS transistor. This figure should make clear where the n-channel is with respect to the p-substrate in an NMOS device for example.
3.3.2 How is a MOS transistor fabricated?

Manufacture of a CMOS device is a complex process. To accommodate both NMOS and PMOS transistors, special regions must be created in which the semiconductor type is opposite to the substrate type. These regions are called wells or tubs. A detailed cross-sectional view of a CMOS transistor is shown in Figure 8. A p-well is created in an n-type substrate or alternatively, an n-well is created in a p-type substrate. In the simple n-well CMOS, fabrication technology, the NMOS transistor is created in the p-type substrate and the PMOS transistor is created in the n-well, which is itself built into the type substrate.

In summarised form the sequence of processes for the fabrication of CMOS integrated circuits on a p-type silicon substrate is,

- Fabrication starts with the creation of the n-well regions for the PMOS transistors, by impurity implantation (doping) into a p-type substrate.
- A thick oxide is grown in the regions that surround the NMOS and PMOS active regions.
3.3.3 How does a MOS transistor work?

The most basic element in the design of a large scale integrated circuit is the transistor. We only need to explain how an NMOS transistor works, remembering that CMOS is just a combination of both NMOS and PMOS technologies.

In order to understand how transistors work we need a bit of physics. In a metal or an n-type semiconductor, electrons are located in the conduction band and are free to move in the presence of an electric field. In a p-type material there are gaps or bubbles in the valence band. Current conduction occurs either by electrons flowing in the conduction band of a metal or an n-type semiconductor or by electrons falling into the bubbles in the valence band of a p-type semiconductor (conduction by holes). Having conductive materials of both types is crucial to the operation of transistors as amplifiers as we will now explain.

Before looking at NMOS transistors, let’s revise how a JFET (Junction Field Effect Transistor) works. Figure 9 shows the structure of an n-channel JFET. The gate is made of p-type material. The output terminals of a JFET are the drain (high voltage) and the source (low voltage). The JFET operates by application of a negative voltage to the gate with respect to the source. This produces a reverse bias across the PN junction between the gate and the source. As in a junction diode, reverse biasing a pn-junction causes electrons in the n-region to be repelled away from the junction. In an n-channel JFET, the controlled quantity is the number of n-type carriers (electrons) in the channel region between the drain and source. The larger the negative voltage on the gate, the lower the electron density in the n-channel and the higher the resistance of the channel. Thus
the voltage supply driving the channel between the drain and the source sees a higher resistance. This is the basis of operation of the JFET. A JFET can act as an amplifier because the input impedance at the gate is high. A low power signal on the gate can control a large current in the channel (drain current) and a higher voltage across the channel.

Figure 9: The cross-sectional view and operation of a JFET transistor.

A cross-sectional view of an NMOS transistor during operation is shown in Figure 10. We first recognise that the p-type material in its substrate consists of holes. These are bubbles waiting for electrons to fall in. At the interface between the p-type substrate and the n-type drain and source regions, diffusion of electrons into the holes gives rise to the formation of a shallow reverse electric field that points from the drain/source to the channel. This means that the drain and source are at a higher potential than the channel and electron flow eventually stops. The field at each PN junction opposes any further diffusion. Basically the p-channel substrate of the NMOS device remains in tact with no electrons in its bulk: the drain to source resistance is high. If a voltage is applied from the drain to the source then the p-channel continues to act like an insulator because it has no free carriers (electrons).

If now a positive voltage is applied to the gate, electrons near the junctions are drawn into the entire channel region near the gate. This results in an electron conduction path which can carry current in response to changes in the drain to source voltage. Because the current increases with a positive voltage applied to the gate, we call this an enhancement mode MOSFET.

The NMOS transistor differs from the JFET (Figure 9) in two ways. Firstly, the gate electrode is placed on top of a very thin oxide layer which means it isn’t in direct electrical contact with the rest of the transistor. Secondly, the device is made without
manufacturing an n-type doped channel between the source and drain. Instead, there are a pair of small n-type regions just under the drain and source electrodes. As a result, when the gate source voltage is zero, the drain to source current is zero no matter what drain-source voltage we apply. In the JFET there is an n-type channel all the way through the substrate and the effect of the negative voltage on the gate is to pinch off the channel by repulsion of electrons in the n-type substrate.

![Figure 10: The cross-sectional view and operation of an NMOS transistor.](image)

In addition to the enhancement mode MOSFET, there is also a depletion mode family normally manufactured in NMOS on p-channel substrates. This device is doped with n-type material (same as at the gate and drain regions) just near the gate so that drain-source current normally flows in the absence of gate voltage. If a negative voltage is applied to the gate, then the negative charge of the doped region of the substrate is repelled and the transistors conducts less drain-source current. This is more similar to a JFET.

*We donot deal with depletion mode devices in CMOS or this course.*

In addition, to the NMOS devices, there are the PMOS devices. Their operation can be described in similar terms to the NMOS device above with some obvious differences. First of all, a PMOS device is a p-channel devices built on an n-type substrate. It has p-type source and drain terminals. To make it conduct, one needs to apply a *negative* voltage to the gate so that holes are attracted into the channel from the source p-type terminal.

*CMOS transistors are always enhancement mode devices.*

32
3.4 Digital Gates and Logic

Now that we understand how CMOS transistors are manufactured, we are in a position to learn how they can be used to perform digital operations.

Transistors can be made to switch between the extreme operating points, essentially from ON to OFF and vice versa. This forms the basis for the design of digital circuits, which only use two voltage states, L and H. Here, L stands for a range of voltages close to zero, while H stands for voltages near $V_{CC}$ (or $V_{DD}$).

First we will first look at how Bipolar Junction Transistors (BJTs) can be used for logic operations. Let’s interpret the BJT switch in digital terms. Suppose $V_{BB}$ is H (say 5 V). Then the transistor is turned on and will operate at the point $Q_{sat}$, so that $V_{CE}$ is L (i.e. 0.2 V). Now if $V_{BB}$ is L (say 0 V), the transistor is off, operates at $Q_{cutoff}$, so $V_{CE}$ is H (i.e. 5 V). If we regard $V_{BB}$ as the input and $V_{CE}$ as the output then the circuit is behaving like a digital inverter. This is summarised in the table.

<table>
<thead>
<tr>
<th>INPUT</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>L</td>
</tr>
</tbody>
</table>

The voltage levels L and H can be interpreted in terms of logic levels 0 and 1, or TRUE and FALSE.

In positive logic, we have the correspondence

<table>
<thead>
<tr>
<th>Physical (voltage) level</th>
<th>logic level (0 or 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>0</td>
</tr>
<tr>
<td>H</td>
<td>1</td>
</tr>
</tbody>
</table>

whereas in negative logic, we have

<table>
<thead>
<tr>
<th>Physical (voltage) level</th>
<th>logic level (0 or 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>1</td>
</tr>
<tr>
<td>H</td>
<td>0</td>
</tr>
</tbody>
</table>

In mixed logic, parts of the circuit use positive logic, while other parts use negative logic.

Therefore the transistor switch is an implementation of a logical inverter, as shown in the table in positive logic.

<table>
<thead>
<tr>
<th>INPUT</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
The FET switch has a similar inverter function.

The inverter is an example of a logic gate. A logic gate is a circuit which has one or more inputs and an output and performs logical operations (assuming we interpret the voltages as logic levels).

There are a number of different digital integrated circuit technologies, and two of the most common are transistor-transistor logic (TTL) and complementary metal-oxide semiconductor (CMOS). In this course we will focus on CMOS, since it is currently the most prominent technology and is relatively easy to understand.

3.5 CMOS Logic

As we saw in the previous section, there are two basic types of CMOS logic gates - NMOS and PMOS. The idealised behaviour of these gates is shown in figure 11 below.

![Symbols and idealised behaviour of NMOS and PMOS CMOS gates.](image)

Figure 11: Symbols and idealised behaviour of NMOS and PMOS CMOS gates.

The gate voltage controls the resistance between source and drain. For an NMOS transistor, a 'low' gate voltage results in a high resistance (essentially no connection between source and drain) while a 'high' gate voltage results in a low resistance conducting path. The transistors can also be represented using the symbols in figure 12. The circle on the PMOS transistor indicates inversion, since a 'low' gate voltage turns the device 'on', resulting in conduction between source and drain.
The two types of transistor can be combined to create logic gates. An inverter is shown in figure 13.

Typically, $V_{DD}$ is 5V and corresponds to the 'high' voltage. When $V_{in}$ is low (0V) the upper (p channel) transistor is 'on' and the lower (n channel) transistor is 'off', pulling the output voltage to $V_{DD}$. When $V_{in}$ is high (5V), the situation is reversed and $V_{out}$ is pulled low (0V).

3.6 Physical Structure and Nonideal Characteristics of MOS Transistors

3.6.1 Currents and Voltages

In the last section we treated MOS transistors as ideal devices with zero resistance in the on state and infinite resistance in the off state. Furthermore, an ideal device would have no current flows. One of the major advantages of MOS technology is that it comes close to these ideals in many ways. Nevertheless, the real characteristics must be considered to ensure proper circuit operation.

It is important to understand over what range of voltages a particular device (such as an inverter or a logic gate) will function as desired, since in practice, $V_{in}$ is never exactly equal to either 0V or $V_{DD}$. This range of voltages is specified by the device manufacturer as follows:
• $V_{IH(min)}$ The minimum voltage level at the input to be recognised as H.
• $V_{IL(max)}$ The maximum voltage level at the input to be recognised as L.
• $V_{OH(min)}$ The minimum voltage at the output when in state H.
• $V_{OL(max)}$ The maximum voltage at the output when in state L.

Thus any voltage greater than $V_{IH(min)}$ applied to an input will be interpreted as H, and any voltage less than $V_{IL(max)}$ will be interpreted as L. For unambiguous communication of logic level from one gate to another we need

$$V_{OH(min)} \geq V_{IH(min)},$$
$$V_{OL(max)} \leq V_{IL(max)}$$

Typical specifications are:

• $V_{IH(min)} = 0.7V_{CC}$
• $V_{IL(max)} = 0.3V_{CC}$
• $V_{OH(min)} = V_{CC} - 0.1V$
• $V_{OL(max)} = 0.1V$

The maximum currents that can flow into the device are also specified:

• $I_{IH}$ The maximum current flowing into the input when a specified H voltage is applied.
• $I_{IL}$ The maximum current flowing into the input when a specified L voltage is applied.

For CMOS devices, these currents are small (typically $1\mu A$ for a simple logic gate).

CMOS transistors have finite resistances in the ON state, and this limits the amount of current that can flow through the transistor. Below are shown equivalent circuits for an inverter with a high or low voltage applied, and with an assumed supply voltage of 5V.

When $V_{in} = 5V$, the NMOS transistor has to sink current from the circuit connected to the output; this current will push up $V_{out}$ due to the non-zero resistance of the NMOS transistor. Similarly, when $V_{in} = 0V$, the PMOS transistor has to source current which will pull $V_{out}$ down. The maximum current that can be sunk or sourced depends on the
values for acceptable LOW and HIGH state voltages. Note also that the transistors are often not symmetric in the sense that they have different resistances in the on state.

The limits on the output current become an important issue particularly when the output is connected to something other than another CMOS circuit - for example, when the output is connected to a TTL (transistor - transistor logic) device. Such devices have a much higher current requirements. We define:

- $I_{OH}$ The maximum current that can be sourced in the HIGH state while maintaining the output voltage within specified limits
- $I_{OL}$ The maximum current that can be sunk in the LOW state while maintaining the output voltage within specified limits

When $V_{in}$ is between 0 and 5V, the values for the resistances of the transistors will be between the on and off values and the output voltage will be between 0 and 5V.

3.6.2 Noise Margin

The noise margin is a measure of the extent to which a logic circuit can tolerate noise or unwanted spurious signals. The high state noise margin is defined as

$$V_{NH} = V_{OH(min)} - V_{IH(min)}$$

and the low state noise margin is

$$V_{NL} = V_{IL(max)} - V_{OL(max)}$$

Thus $V_{NH}$ and $V_{NL}$ give the maximum noise voltage that will not cause incorrect or ambiguous communication of logic level in the H and L states, respectively.
3.6.3 Fan Out

In logic circuits, a gate output often has to drive a number of other gate inputs. However, it can only supply a limited amount of current. The fan out is the maximum number of gates that can be driven reliably, i.e. without incorrect or ambiguous voltage levels.

Fan out depends on both the driving and driven devices, since different devices require different input currents.

Fan out is given by

\[
\text{FANOUT} = \min \left( \frac{|I_{OH}|}{|I_{IH}|}, \frac{|I_{OL}|}{|I_{IL}|} \right).
\]

The current values are available on data sheets. Loading a device beyond its specifications may result in an output voltage which does not meet specifications, circuit delays which increase beyond the specification and overheating and possible device failure.
### 3.6.4 Transition Time

Transition time is the amount of time that the output of a logic circuit takes to change from one state to another. The output does not change state instantaneously since the transistors have a nonzero resistance in their on state, and since the output circuit always has a capacitance associated with it. This output capacitance has a number of sources, including the circuit wiring and the input circuits (the gates of the transistors connected to the input must be charged or discharged).

### 3.6.5 Propagation Delays

It takes a non-zero amount of time for a transistor to change state, due to the physics of transistor devices. The *propagation delay* of a signal path is the amount of time that it takes for a change in the input signal to produce a change in the output signal:

- $t_{PLH}$ Time taken to go from L to H.
- $t_{PHL}$ Time taken to go from H to L.
- The average propagation delay is 
  \[ t_{P(\text{ave})} = \frac{t_{PLH} + t_{PHL}}{2}. \]

The speed of a circuit is limited by both the transition time and propagation delay.
3.6.6 Power

Every digital IC consumes power, and this is an important parameter in digital system design, especially in portable products.

Power consumption varies significantly between logic families. Static power consumption refers to power consumption in steady-state. Dynamic power consumption occurs during transitions from one state to another. CMOS is characterised by very low static power consumption, since the gate leakage current is small. Two sources of dynamic power consumption in CMOS devices are:

- During transitions, there is a period when an input voltage between the LOW and HIGH state voltages is applied. This leads to both n and p channel transistors being partially on, and significant current can flow from the high voltage supply rail to ground.
- The capacitive load on the output leads to power dissipation in the transistors which have to sink or source the required current.

3.6.7 Speed-Power Product

Since it is desirable to have digital IC which are fast and low power, a useful figure of merit is the speed-power product

\[ t_{P(ave)} \times P_{D(ave)} \]

which we would like to be as small as possible.

3.6.8 Another Example of a CMOS Logic Gate

An example of a slightly more complex logic gate, a 2 input AND gate, is shown below in Figure 17 and a 2 input NAND in 18.

Exercise: Draw up a table listing which of the transistors Q1-Q6 in 17 is on for each possible combination of the inputs, and hence deduce the output Z. Note that the AND gate consists of two parts - Q1-Q4 comprise a 2 input NAND gate, while Q5 and Q6 form an inverter.

Exercise: Try to draw the circuits of a 3 input AND gate and of a 2 input OR gate.

The simplest gates to implement in CMOS (and therefore also the fastest gates) are actually NAND and NOR gates, since the inversion come 'for free'.
3.7 Programmable Logic

In addition to the fixed logic ASICs that we have learnt about in this chapter, there are more complex classes that are in fact the main subject matter of this course. These include the complex programmable logic devices. A very brief summary of these is given here.

A list of these devices and some of their most important properties follows [2].

- Field-Programmable Device (FPD) – a general term that refers to any type of integrated circuit used for implementing digital hardware, where the chip can be configured by the end user to realise different designs. Programming of such a device often involves placing the chip into a special programming unit, but some chips can also be configured “in-system”. Another name for FPDs is programmable logic devices (PLDs).

- PLA – a Programmable Logic Array (PLA) is a relatively small FPD that contains two levels of logic, an AND-plane and an OR-plane, where both levels are programmable.

- PAL – a Programmable Array Logic (PAL) is a relatively small FPD that has a programmable AND-plane followed by a fixed OR-plane

- CPLD – a more Complex PLD that consists of an arrangement of multiple logic blocks on a single chip.
Figure 18: A 2 input NAND gate.

- FPGA – a Field-Programmable Gate Array is an FPD featuring a general structure that allows very high logic capacity. Whereas CPLDs feature logic resources with a wide number of inputs (AND planes), FPGAs offer more narrow logic resources. FPGAs also offer a higher ratio of flip-flops to logic resources than do CPLDs. FPGAs are field programmable. This means that you can program them in situ to take on new digital designs. These designs are generally volatile.

- Interconnect – the wiring resources in an FPD.

- Logic Block – a relatively small circuit block that is replicated in an array in an FPD. When a circuit is implemented in an FPD, it is first decomposed into smaller sub-circuits that can each be mapped into a logic block. The term logic block is mostly used in the context of FPGAs, but it could also refer to a block of circuitry in a CPLD. Hence the term Configurable Logic Block (CLB) in an FPGA.

- Logic Capacity – the amount of digital logic that can be mapped into a single FPD. This is usually measured in units of “equivalent number of gates in a traditional gate array”. In other words, the capacity of an FPD is measured by the size of gate array that it is comparable to. In simpler terms, logic capacity can be thought of as "number of 2-input NAND gates".

- Logic Density – the amount of logic per unit area in an FPD.
• Speed-Performance – measures the maximum operable speed of a circuit when implemented in an FPD. For combinational circuits, it is set by the longest delay through any path, and for sequential circuits it is the maximum clock frequency for which the circuit functions properly.

3.7.1 Field Programmable Gate Arrays

The historical roots of FPGAs are in complex programmable logic devices (CPLDs) of the early to mid 1980s. Ross Freeman, Xilinx cofounder, invented the field programmable gate array in 1984. CPLDs and FPGAs include a relatively large number of programmable logic elements. CPLD logic gate densities range from the equivalent of several thousand to tens of thousands of logic gates, while FPGAs typically range from tens of thousands to several million.

The primary differences between CPLDs and FPGAs are architectural. A CPLD has a somewhat restrictive structure consisting of one or more programmable sum-of-products logic arrays feeding a relatively small number of clocked registers. The result of this is less flexibility, with the advantage of more predictable timing delays and a higher logic to interconnect ratio. The FPGA architectures, on the other hand, are dominated by interconnect. This makes them far more flexible (in terms of the range of designs that are practical for implementation within them) but also far more complex to design for.

Another notable difference between CPLDs and FPGAs is the presence in most FPGAs of higher level embedded functions (such as adders and multipliers) and embedded memories. A related, important difference is that many modern FPGAs support full or partial in system reconfiguration, allowing their designs to be changed "on the fly" either for system upgrades or for dynamic reconfiguration as a normal part of system operation. Some FPGAs have the capability of partial reconfiguration that lets one portion of the device be reprogrammed while other portions continue running.

3.7.2 Applications of FPGAs

Applications of FPGAs include DSP, software defined radio, aerospace and defense systems, ASIC prototyping, medical imaging, computer vision, speech recognition, cryptography, bioinformatics, computer hardware emulation and a growing range of other areas. FPGAs originally began as competitors to CPLDs and competed in a similar space, that of glue logic for PCBs. As their size, capabilities, and speed increased, they began to take over larger and larger functions to the state where some are now marketed as full systems on chips (SOC).

FPGAs especially find applications in any area or algorithm that can make use of the massive parallelism offered by their architecture. One such area is code breaking, in particular bruteforce attack, of cryptographic algorithms. FPGAs are increasingly used in conventional High Performance Computing applications where computational kernels
such as FFT or Convolution are performed on the FPGA instead of a microprocessor. The use of FPGAs for computing tasks is known as Reconfigurable Computing.

The inherent parallelism of the logic resources on the FPGA allows for considerable compute throughput even at a sub 500MHz clock rate. For example, the current (2007) generation of FPGAs can implement around 100 single precision floating point units, all of which can compute a result every single clock cycle. The flexibility of the FPGA allows for even higher performance by trading off precision and range in the number format for an increased number of parallel arithmetic units. This has driven a new type of processing called Reconfigurable Computing, where time intensive tasks are offloaded from software to FPGAs. The adoption of FPGAs in high performance computing is currently limited by the complexity of FPGA design compared to conventional software and the extremely long turnaround times of current design tools, where 48 hours wait is necessary after even minor changes to the source code. In this course, the FPGA concerned (the XC2S50 by Xilinx) can be programmed in a matter of seconds for our lab tasks.

3.7.3 FPGA Architecture

The typical basic architecture consists of an array of configurable logic blocks (CLBs) and routing channels. Multiple I/O pads may fit into the height of one row or the width of one column in the array. Generally, all the routing channels have the same width (number of wires). An application circuit must be mapped into an FPGA with adequate resources.

A classic FPGA logic block consists of a 4 input lookup table (LUT), and a flipflop, as shown in Figure ???. In recent years, manufacturers have started moving to 6 input LUTs in their high performance parts, claiming increased performance.

In addition there are peripheral arrays of Input Output Blocks. A table comparing all the FPGAs in the same product class (SPARTAN-II Xilinx FPGAs) as the XC2S50 which we use this semester in the labs is shown in Figure ???

For more detail on this and other FPGAs please see [4] and [3].
Figure 19: A classic standard FPGA CLB.

<table>
<thead>
<tr>
<th>Device</th>
<th>Logic Cells</th>
<th>System Gates (Logic and RAM)</th>
<th>CLB Array (R x C)</th>
<th>Total CLBs</th>
<th>Maximum Available User I/O^{(1)}</th>
<th>Total Distributed RAM Bits</th>
<th>Total Block RAM Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>XC2S15</td>
<td>432</td>
<td>15,000</td>
<td>8 x 12</td>
<td>96</td>
<td>86</td>
<td>6,144</td>
<td>16K</td>
</tr>
<tr>
<td>XC2S30</td>
<td>972</td>
<td>30,000</td>
<td>12 x 18</td>
<td>216</td>
<td>92</td>
<td>13,824</td>
<td>24K</td>
</tr>
<tr>
<td>XC2S50</td>
<td>1,728</td>
<td>50,000</td>
<td>16 x 24</td>
<td>384</td>
<td>176</td>
<td>24,576</td>
<td>32K</td>
</tr>
<tr>
<td>XC2S100</td>
<td>2,700</td>
<td>100,000</td>
<td>20 x 30</td>
<td>600</td>
<td>176</td>
<td>38,400</td>
<td>40K</td>
</tr>
<tr>
<td>XC2S150</td>
<td>3,888</td>
<td>150,000</td>
<td>24 x 36</td>
<td>864</td>
<td>260</td>
<td>55,296</td>
<td>48K</td>
</tr>
<tr>
<td>XC2S200</td>
<td>5,292</td>
<td>200,000</td>
<td>28 x 42</td>
<td>1,176</td>
<td>284</td>
<td>75,264</td>
<td>56K</td>
</tr>
</tbody>
</table>

Figure 20: The Spartan II FPGA product class.
4 Combinational Logic - Review

4.1 Combinational Logic

In ENGN2211 (or elsewhere) you learned about combinational logic, i.e. digital logic circuits without memory. Combinational circuits consist of an interconnected network of standard logic gates such as AND, OR, NAND, XOR, etc. You have learnt about canonical SOP forms, K-maps, and other techniques for combinational circuit representation and design, and you are probably familiar with digital schematic diagrams. You also looked at combinational modules such as ALUs, multiplexers, decoders, etc.

This chapter contains some basic information about combinational logic. See also Wakerly, Chapters 3, 4 and 6 [7].

4.2 Boolean Algebra

Boolean algebra is the underlying mathematics of digital or logical circuits. In this section we look at digital circuits in terms of abstract Boolean algebra, and do not consider details of physical implementation.

4.2.1 Boolean Variables

Digital circuits manipulate boolean variables, $A, B, X, Y, \text{ etc}$

which take values in the set $B = \{0, 1\}$

where

$0 = \text{"false"}$

$1 = \text{"true"}$

In boolean algebra, we consider expressions such as

$X = A \text{ AND } B$

which means:

In order for $X$ to be true, both $A$ and $B$ must be true.

This is just like in common sense. Boolean algebra is mathematically precise, and has rules and theorems concerning the manipulation of boolean expressions involving the boolean functions AND, OR, NOT.
4.2.2 Boolean Functions

NOT.

\[ \overline{A} = \text{NOT } A \]

So \( \overline{A} \) is the complement of \( A \), defined by the truth table

<table>
<thead>
<tr>
<th>( A )</th>
<th>( \overline{A} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

A truth table defines the boolean function by specifying the boolean output value associated with each boolean input value.

The circuit symbol for the NOT function is the inverter, Figure 21. The bubble denotes inversion (complement).

![Figure 21: NOT gate.](image)

OR.

\[ X = A \text{ OR } B \]
\[ = A + B \]

Here + denotes OR, defined by:

<table>
<thead>
<tr>
<th>( A )</th>
<th>( B )</th>
<th>( X = A + B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

OR is like two switches \( A, B \) is parallel; one or both open lets current flow. The circuit symbol is shown in Figure 22.

![Figure 22: OR gate.](image)
**AND.**

\[ X = A \text{ AND } B \]
\[ = A \cdot B \]
\[ = AB \]

Here \( \cdot \) denotes AND, defined by:

<table>
<thead>
<tr>
<th>( A )</th>
<th>( B )</th>
<th>( X = A \cdot B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

AND is like two switches \( A, B \) is series; both must be open for current to flow. The circuit symbol is shown in Figure 23.

![AND gate](image)

Figure 23: AND gate.

These three are the most basic logic functions, and define the Boolean algebra on the set \( B \).

Also important are the following functions.

**NAND.**

\[ X = \text{NOT}(A \text{ AND } B) \]
\[ = \overline{A \cdot B} \]
\[ = \overline{AB} \]

NAND is defined by:

<table>
<thead>
<tr>
<th>( A )</th>
<th>( B )</th>
<th>( X = AB )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

NAND is the complement of AND. The circuit symbol is shown in Figure 24.

**NOR.**

\[ X = \text{NOT}(A \text{ OR } B) \]
\[ = \overline{A + B} \]

NOR is defined by:
NAND and NOR are very important building blocks.

**XOR.**

\[ X = A \oplus B \]
\[ = \overline{AB} + A\overline{B} \]

XOR is defined by:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>X = A \oplus B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

XOR is true if one and only one of \( A, B \) is true; hence the term *exclusive OR*. The circuit symbol is shown in Figure 26.

XOR is useful in

- arithmetic circuits
- controlled inverter
- parity generation and checking
EQV is defined by:

\[ X = A \oplus B = A \overline{B} + AB \]

EQV is true only if both \( A \) and \( B \) are the same, and so is the complement of XOR. It is called *equivalence* or *exclusive NOR*. The circuit symbol is shown in Figure 27.

EQV is useful for detecting equality.

### 4.2.3 Gate Outputs

In general, gate outputs should not be tied together unless the output circuitry is specifically designed to do so. The basic problem is that each gate may have different output states, and these can be in conflict, e.g. gate 1 may be H and gate 2 may be L. The output in this case is difficult to predict and may be in the undefined voltage region. This situation can also cause excessive heating, resulting in device failure.

One type of gate which can have its outputs tied together (if properly used!) is the tri state gate, shown in figure 28.

The tri state gate has the following truth table:
High Z means high output impedance, in other words, the output is effectively isolated from the input. Tri state gates are very useful, for example, when several devices are connected to the same data bus, as we will see later. In that case, tri state gates can be used to ensure that only device has control of the bus at any one time and can place its data on the bus. Tri state gates are usually designed so that the delay involved in turning the gate on (resulting in the input signal being transferred to the output) is slightly longer than the delay involved in turning the gate off, to ensure that at no time more than one device has ‘access’ to the bus.

4.2.4 Unused Gate Inputs

At times, a gate has to be used which has more inputs than required. In this case, the question arises how to deal with the unused inputs. It is not good practice to leave unused inputs floating - due to the high input impedance of MOS devices, it is easy for charge to accumulate on the input, creating an effective HIGH signal. One way of dealing with unused inputs it to tie two inputs together, as shown in fig. 29.

4.2.5 Theorems of Boolean Algebra

Here are some of the most important theorems of boolean algebra.
1. $X \cdot 0 = 0$
2. $X \cdot 1 = X$
3. $X \cdot X = X$
4. $X \cdot \overline{X} = 0$
5. $X + 0 = X$
6. $X + 1 = 1$
7. $X + X = X$
8. $X + \overline{X} = 1$
9. $X + Y = Y + X$
10. $X \cdot Y = Y \cdot X$
11. $X + (Y + Z) = (X + Y) + Z$
12. $X(YZ) = (XY)Z$
13. $X(Y + Z) = XY + XZ$
14. $(W + X)(Y + Z) = WY + XY + WZ + XZ$
15. $X + XY = X$
16. $X + \overline{XY} = X + Y$

**De Morgan’s laws:**

17. $\overline{X + Y} = \overline{X} \cdot \overline{Y}$
18. $\overline{X \cdot Y} = \overline{X} + \overline{Y}$

These theorems can be proved in a straightforward way: write down the truth tables for the left and right sides of each asserted equality and check that they are the same.

These theorems help us simplify boolean expressions, such as

$$X = \overline{(A + C)} \cdot (B + D)$$  \hspace{1cm} (7)

We would like to reduce this to an expression involving only the variables and complements, as follows:

$$X = \overline{(A + C)} + (B + D)$$
$$= \overline{A} \cdot \overline{C} + \overline{B} \cdot \overline{D}$$
$$= A \cdot \overline{C} + \overline{B} \cdot D$$
Exercise. Determine which theorems are being applied in each step.

Principle of Duality: Any theorem or identity in switching algebra remains true if 0 and 1 are swapped and · and + are swapped throughout.

Mathematically this can be written as $F^D (X_1, X_2, ..., X_n, +, ·) = F^D (X_1, X_2, ..., X_n, ·, +)$ where $F^D$ denotes the dual.

Example 1: Consider the logic statement,

$$Z = X \cdot Y$$

The dual of this statement is,

$$\overline{Z} = \overline{X} + \overline{Y}$$

Example 2: Consider the following rule (distributivity).

$$X \cdot (Y + Z) = X \cdot Y + X \cdot Z$$

The right hand side can be obtained by multiplying out the left hand side. The dual of this expression is,

$$X + Y \cdot Z = (X + Y) \cdot (X + Z)$$

Which can be proven by adding out the left hand side.

Operator precedence: Just as in ‘normal’ algebra, · has precedence over +. It is important to remember this when applying the principle of duality!

Examples:
Consider the following statements:

$$X + (X \cdot Y) = X$$
$$X + X \cdot Y = X$$

These statements are obviously the same. Operator precedence (superficially and speciously) renders the parentheses superfluous. Now apply the principle of duality to each statement by inverting the ·’s and +’s,

$$X \cdot (X + Y) = X$$
$$X \cdot X + Y = X$$

In the first case we have a true statement thanks to the parentheses. However operator precedence by itself in the second case (without the parentheses) has led to an absurd result.
4.2.6 De Morgan’s Laws in Terms of Gates

Interestingly De Morgan’s laws can be expressed in terms of logic gates, Figure 30.

The idea is to pull bubble through and swap AND and OR.

4.3 Boolean Functions and Digital Circuits

Combinational logic circuits can be constructed to implement boolean functions.

Consider the boolean function $X$ defined by

$$X = AB + C$$

Here, $A$, $B$ and $C$ are the independent boolean variables. In terms of gates we have the circuit of Figure 31.

The algebraic expression and the circuit are equivalent representations of the same boolean function.

4.3.1 Universality of NAND and NOR

We know that all boolean functions can be expressed in terms of AND, OR and NOT. In fact, all boolean functions can be expressed in terms of either
• NAND gates only, or
• NOR gates only.

To see this, let’s show that NAND gates can be used to implement NOT, AND and OR. Algebraically, we have

\[(A \cdot 1) = \bar{A}\]

so NAND can implement NOT. Next,

\[(\overline{A \cdot B} \cdot 1) = AB\]

giving AND.

Exercise. Using Boolean algebra, show how to obtain \(A + B\) from NAND gates.

The universality of NAND is illustrated in Figure 32.

![Figure 32: Universality of NAND.](image)

Exercise. Show how NOR gates can be used to implement AND, OR, and NOT.

Example. Implement the logic function

\[X = AB + CD\]

using a minimum number of NAND gates only.
We first implement as using AND and OR as shown in Figure 33.
This implementation requires two ICs. Next, we draw bubbles as shown in Figure 34.
The AND level clearly becomes a NAND level. But by De Morgan, the OR gate has become also a NAND gate. This NAND gate implementation requires only one 74LS00 chip.
The circuit of Figure 34 is identical to Figure 35, where we have explicitly drawn the second level NAND gate.

4.3.2 Asserted Levels

Logic signals are often used to initiate actions.
A signal is asserted if it is active.
A signal is unasserted if it is inactive.
The labeling of signals reflects this, for example:
Figure 35: NAND implementation.

\[
\begin{align*}
\text{LOAD} & \quad \text{active H} \\
\text{RESET} & \quad \text{active L} \\
\text{RD/WR} & \quad \text{bistate}
\end{align*}
\]

where bistate means

\[
\begin{align*}
H \rightarrow RD \text{ asserted} \\
L \rightarrow WR \text{ asserted}
\end{align*}
\]

4.3.3 Alternative Logic Gate Representations

Figure 36 list some useful representations of some standard logic gates.

Active H - no bubble

Active L - bubble

4.3.4 Canonical Forms

There are two standard or canonical ways of expressing boolean functions:


\[
X = A + BC + ABC
\]

2. Product-of-sums (POS) E.g.

\[
X = (A + D)(C + \overline{D})(\overline{B} + C)
\]

These representations are useful for
Figure 36: Alternative logic gate representations.

- direct implementation, and
- starting logic function minimization.

We will focus on SOP.

Consider

$$f(A, B, C) = A + \overline{B}C + \overline{A}BC$$

where

- product terms $A, \overline{B}C, \overline{A}BC$
- minterms $\overline{A}BC$

A minterm is any ANDed term containing all of the variables (perhaps complemented).

Let’s look at the truth table which corresponds to this function:
A B C f(A, B, C) \\
0 0 0 0 \\
0 0 1 1 \\
0 1 0 0 \\
0 1 1 1 \\
1 0 0 1 \\
1 0 1 1 \\
1 1 0 1 \\
1 1 1 1 \\

Each row of the truth table corresponds to one of the \(2^n = 8\) possible minterms in \(n = 3\) variables.

\[ m_i = m_i(A, B, C), \quad i = 0, \ldots, 2^3 - 1 \]

E.g.

\[ m_3 = \overline{A}BC = 011 \]

Actually, the truth table specifies the function as a sum of minterms:

\[ f(A, B, C) = m_1 + m_3 + m_4 + m_5 + m_6 + m_7 \]

This is called the canonical SOP representation of the function \(f\).

The minterm code for \(n = 3\) is as follows:

<table>
<thead>
<tr>
<th>(m_i)</th>
<th>(A)</th>
<th>(B)</th>
<th>(C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(m_0)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(m_1)</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(m_2)</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(m_3)</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(m_4)</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(m_5)</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(m_6)</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(m_7)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Complemented variables correspond to 0 and uncomplemented variables correspond to 1.

The function \(f(A, B, C) = A + \overline{B}C + \overline{A}BC\) can be put into canonical SOP form algebraically as follows:

\[ \overline{BC} = (\overline{A} + A)\overline{BC} = \overline{A}BC + A\overline{BC} = m_1 + m_5 \]

\[ A = (B + \overline{B})A = \ldots = m_4 + m_5 + m_6 + m_7 \]

(fill in the missing steps!) and so on combining we get \(f(A, B, C) = \sum(1, 3, 4, 5, 6, 7)\) as before.

Any Boolean function can be expressed in canonical SOP form.
4.4 Simplification and Implementation of Boolean Functions

Boolean functions can be implemented in hardware in a number of ways. For instance, standard discrete TTL or CMOS ICs could be used, in which case it is useful to find the simplest expression for the function being implemented. Or if programmable devices are to be used, then a more direct representation of the function may be useful.

4.4.1 Direct Implementation

Consider the function
\[ f(A, B, C) = \sum m(2, 3, 6, 7) \]
expressed in canonical SOP form. Then assuming all variables and their complements are available we can implement this function with the AND-OR circuit of Figure 37.

![Figure 37: AND/OR implementation.](image)

This implementation is not minimal in general (i.e. can realize \( f \) with fewer gates).

This representation is direct and is useful when implementing with programmable logic devices (PLD). To illustrate, consider functions \( f = f(A, B) \) of two variables \( (n = 2, \ 2^n = 4) \). A PLD schematic is shown in Figure 38.

This PLD can program any given function \( f(A, B) \) by breaking appropriate links.
4.4.2 Karnaugh Maps (K-Maps)

Karnaugh or K-maps are useful tool for boolean function minimization, and for visualization of the boolean function. In brief,

- K-maps provide a graphical method for minimizing boolean functions via pattern recognition for up to about $n = 6$ variables.
- For larger numbers of variables, there are computer algorithms which can yield near-minimal implementations.
- K-maps are a way of expressing truth tables to make minimization easier. They are constructed from minterm codes.

Consider the boolean function

$$f(A, B) = \sum m(0, 1, 2)$$

The truth table is
The K-map is shown in Figure 39. The essence of the K-map is the two dimensional representation of $f$, which is equivalent to the truth table but more visual.

To minimize $f$, we loop out logical adjacencies, Figure 39.

$$
\overline{AB} + \overline{AB} = \overline{A(B + \overline{B})} = \overline{A}
$$

$$
\overline{AB} + A\overline{B} = (A + B)\overline{B} = \overline{B}
$$

Figure 39: K-map showing looped-out terms and also corresponding minterms.

Therefore

$$
f = \overline{A} + \overline{B}
$$

This is less complex than $f$ in canonical SOP form.

Note. Looping out logical adjacencies is a graphical alternative to algebraic calculations.
Unit distance code (Gray code.) For two bits, the Gray code is:

\[\begin{array}{cccc}
00 & 01 & 11 & 10 \\
\end{array}\]

Only one bit changes as you go from left to right. This code preserves logical adjacencies.

The \textit{K-map method} is to loop out groups of \(2^n\) logically adjacent minterms. Each looped out group corresponds to a product term in a minimal SOP expression.

1. Loop out single 1s \((n = 0)\) which have no logical adjacencies.
2. Loop out all pairs of 1s \((n = 1)\) which cannot be included in a larger group.
3. Loop out all quads of 1s \((n = 2)\) which cannot be included in a larger group.
4. Etc.

\textbf{Example.} \(f(A, B, C) = \sum m(0, 2, 3, 4, 6)\) The K-map is shown in Figure 40.

\[\begin{array}{c|cccc}
A & 00 & 01 & 11 & 10 \\
\hline
0 & 1 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 & 1 \\
\end{array}\]

\[\text{Figure 40: K-map for } f(A, B, C) = \sum m(0, 2, 3, 4, 6).\]

Moving left to right or up to down in the K-map changes only one digit in the minterm code. Note the wrap-around at the ends: because of logical adjacency, the top and bottom are joined, and the left and right are joined.

\(n = 0\): none
\(n = 1\): \(m_2 + m_3 = \overline{AB}\)
\(n = 2\): \(m_0 + m_2 + m_4 + m_6 = \overline{C}\)
Therefore the minimal SOP representation is
\[ f = \overline{A}B + C \]

**Example.** \( f(A, B, C, D) = \sum m(0, 1, 4, 8, 9, 12, 14, 15) \) The K-map is shown in Figure 41.

![K-map for \( f(A, B, C, D) = \sum m(0, 1, 4, 8, 9, 12, 14, 15) \).](image)

Therefore the minimal SOP representation is
\[ f = ABC + \overline{C}.D + B.\overline{C} \]

*Don’t cares.* In some applications it doesn’t matter what the output is for certain input values. These are called *don’t cares.*

For instance, in the *Binary Coded Decimal* code, not all input values occur:
The decimal numbers are those in the range 0, 1, . . . , 9, and a minimum of 4 bits is needed to encode these. The remaining numbers 10, 11, . . . , 15 correspond to code values which are not used in BCD.

*We shall use the symbols φ or X to denote don’t cares.*

Don’t care can be exploited to help minimize boolean functions.

**Example.** \( f(A, B, C) = \sum m(0, 1, 5, 7) + \phi(2, 4) \) The K-map is shown in Figure 42.

![K-map for \( f(A, B, C) = \sum m(0, 1, 5, 7) + \phi(2, 4) \).](image)

The minimal SOP representation is

\[
f = \overline{B} + AC
\]
4.4.3 Generating Gray Code for n Bits

To generate gray code for n bits, a simple algorithm can be used:

1. Write down the gray code for n bits
2. Append a '0' to all the code words
3. Write the code words again in reverse order, with a 1 rather than a 0 appended

For example, to generate 3 bit gray code:

1. Write 00, 01, 11, 10
2. Append 0 ⇒ 000, 001, 011, 010
3. Repeat in reverse order, and append 1 rather than 0 ⇒ 000, 001, 011, 010, 110, 111, 101, 100

4.4.4 SOP and POS representations

Karnaugh maps allow us to come up with a 2 level realization of logic functions. In general, if we want to find the lowest cost realization (in the sense of requiring the fewest number of gates with the fewest number of inputs) we must consider both SOP and POS.

As a trivial example of the process, consider the function

\[ f(A, B) = \sum m(1, 2, 3) \]

Using a Karnaugh map we can come up with the SOP representation:

\[ f = A + B \]

To come up with the POS representation, we first draw the K-map for the function \( \overline{f} \). This gives us

\[ \overline{f} = \overline{A} \overline{B} \]

Therefore

\[ f = \overline{A} \overline{B} = A + B \]

For this example, both SOP and POS yield the same implementation. However, this is not always the case.
4.4.5 Entered-Variable K-Maps

A generalization of the k-map method is to introduce variables into the k-map squares. These are called *entered variable k-maps*. This is useful for functions of large numbers of variables, and can generally provide a clear way of representing Boolean functions.

An entered variable k-map is shown in Figure 43.

Note the variable $C$ in the top left square. It corresponds to $\overline{A}.\overline{B}.C$.

It can be looped out with the 1, since $1 = 1 + C$, and we can loop out the two terms $\overline{A}.\overline{B}.C$ and $A.\overline{B}.C$ to get $\overline{B}.C$.

The remaining term $A.\overline{B}.C$ needs to be added to the cover, or more simply, just loop out the 1. The outcome is $f = A.\overline{B} + \overline{B}.C$.

Think about this!

Figure 44 shows another EV k-map, with four entered variables $C_0$, $C_1$, $C_2$, $C_3$. Each of these terms are different and must be looped out individually to get

$$f = \overline{A}.\overline{B}.C_0 + A.B.C_1 + A.\overline{B}.C_2 + A.B.C_3.$$
4.4.6 Computer Methods

Modern electronics CAD packages have algorithms for Boolean function minimisation, which help designers considerably by doing the tedious reduction calculations automatically. ESPRESSO is a basic package of function minimization algorithms.

4.5 Hazards

A *glitch* is an unwanted short duration pulse. Such pulses can cause circuit malfunction, and so they are *hazards*.

In combinational circuits, hazards are not so critical, but in sequential circuits glitches can initiate unwanted actions.

The following two types of hazards commonly occur in combinational circuits:

- *Static Hazards*. A static hazard is the possibility of a circuit’s output producing a temporary glitch (either a 0 or a 1) when it should remain steady. A static-1 hazard is the possibility of a circuit’s output producing a temporary 0 when the output would be expected to remain at 1; a static-0 hazard is the possibility of a circuit’s output producing a temporary 1 when the output would be expected to remain at 0.

- *Dynamic Hazards*. The possibility of the output changing more than once as a result of a single input transition.

Hazards are caused by asymmetries in signal paths with different propagation delays.
As an example of a static hazard, consider the combinational logic circuit of Figure 45 for a function $F$.

The K-map is shown in Figure 46.

Now $F = X Y + Y Z$. Let’s set $X = Z = 1$. Then we have $F = 1 \cdot Y + Y \cdot 1 = 1$, for any value of $Y$ ideally. Now consider a transition for $Y$ from 1 to 0 - this will produce a glitch in $F$, because there will be a short time during which $a = b = 0$ (these should be complementary). This is illustrated in Figure 47.

This hazard can be removed with the aid of a hazard cover. This is achieved by doing an extra loop in the K-map which overlaps the existing loops in Figure 46; this is shown in Figure 48. Algebraically, we are expressing $F$ in the non-minimal form

$$F = X \overline{Y} + Y Z + X Z$$

where $X Z$ is the extra term corresponding to the hazard cover. This term is redundant logically, but has the effect of removing the glitch.
4.6 Standard MSI Combinational Components

There is a wide range of standard combinational (and sequential) IC circuits which perform useful functions. A designer can make use of standard IC packages instead of having to build everything from scratch.

4.6.1 Decoders and Encoders

A decoder is an $n$-input $2^n$ output device which activates one and only one of its outputs depending on the unique input pattern, Figure 49.

$$Y_i = m_i \cdot EN,$$ where $m_i$ is the $i$th minterm (out of $2^n$ total outputs). $Y$ is active L.

$EN$ is ENABLE (active L).

$A_1, \ldots, A_n$ are $n$ inputs

When $EN$ is active, one and only one $Y_i$ is active; all other outputs are inactive.

When $EN$ is inactive, all outputs are inactive.

**Example.** Consider the 2-4 ($n = 2$) decoder specified in the truth table:

<table>
<thead>
<tr>
<th>$EN$</th>
<th>$A_1$</th>
<th>$A_0$</th>
<th>$Y_3$</th>
<th>$Y_2$</th>
<th>$Y_1$</th>
<th>$Y_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$X$</td>
<td>$X$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
The output functions are given by

\[ Y_0 = \overline{A_1A_0}EN \]
\[ Y_1 = \overline{A_1A_0}EN \]
\[ Y_2 = A_1\overline{A_0}EN \]
\[ Y_3 = A_1A_0EN \]

A circuit implementation is shown in Figure 50.
Example. Decoders can be used to implement combinational logic functions. Consider the canonical SOP expression

\[ F(A, B, C) = \sum m(1, 3, 4, 7). \]

A 3-8 decoder circuit implementing this function is shown in Figure 51.

Encoders perform the function opposite to that of decoders. Encoders have up to \(2^n\) inputs and \(n\) outputs, and generate an \(n\)-bit word for each of the inputs.

In priority encoders, any number of the inputs can be active, so a priority scheme selects a particular input line.

An \(n\)-m encoder block is shown in Figure 52.

Here, \(m \leq 2^n\), \(EI\) denotes input enable, \(OE\) denotes output enable, which together with \(GS\) is used for cascading.

A truth table indicating how the priority scheme is implemented is:
Figure 51: Decoder \((n = 3)\) used to implement the logic function \(F = \sum m(1, 3, 4, 7)\).

\[
\begin{array}{cccccccc}
 E_1 & I_2 & I_1 & I_0 & G_0 & Y_1 & Y_0 & E_O \\
 0 & X & X & X & 0 & 0 & 0 & 0 \\
 1 & 1 & X & X & 1 & 1 & 1 & 0 \\
 1 & 0 & 1 & X & 1 & 0 & 0 & 0 \\
 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\end{array}
\]

\(I_2\) has the highest priority, with code 11; second is \(I_1\), with code 10; and lowest is \(I_0\), with code 01. When the device is inactive, the code is 00.

4.6.2 Multiplexers and Demultiplexers

Multiplexers and demultiplexers are used in serial data transmission systems where a single transmission line can be used to transmit information in a large number of data lines by letting each data line have access to the transmission line on a time-share basis, Figure 53.

Multiplexers selectively connect one and only one of \(2^n\) input lines \(I_i\) to the transmission line \(Y\).

Example. Here is a truth table defining a 4-1 MUX:

\[
\begin{array}{cccc}
 E_N & S_1 & S_0 & Y \\
 0 & X & X & 0 \\
 1 & 0 & 0 & I_0 \\
 1 & 0 & 1 & I_1 \\
 1 & 1 & 0 & I_2 \\
 1 & 1 & 1 & I_3 \\
\end{array}
\]

\(Y = \overline{S}_1 \overline{S}_0 I_0 E_N + S_1 \overline{S}_0 I_1 E_N + \overline{S}_1 S_0 I_2 E_N + S_1 S_0 I_3 E_N\)
Example. Multiplexers can be used to implement combinational logic functions. Consider the canonical SOP expression

\[ F(A, B, C) = \sum m(1, 3, 4, 7). \]

This function is implemented by a MUX in Figure 54.

There are no specific demultiplexers, since the DMUX function can be implemented using a decoder.

Exercise. Show how a DMUX can be built from a decoder.

4.6.3 Other

Some other examples of combinational MSI devices are listed below.

- *Comparators* compare two inputs and determine whether the logic levels are the same or different.

- *Code converters* of various types perform a conversion of an input word in one code into a second word in another code.

- *Arithmetic-logic units (ALU)* perform a range of arithmetic and logical operations.
Figure 53: Multiplexer-demultiplexer time-share arrangement.

Figure 54: MUX used to implement the logic function $F = \sum m(1, 3, 4, 7)$. 
5 Introduction to VERILOG

5.1 VERILOG HDL

VERILOG HDL is one of the two most popular hardware definition languages for describing the behaviour and structure of electronic circuits. The other is VHDL. Hardware Definition Languages (HDLs) are somewhat like computer procedural programming languages except that, instead of programming a machine of fixed architecture to execute instructions, HDLs are used to design the hardware itself. The Verilog Hardware Description Language is an IEEE standard (IEEE Std. 1364-2001). This standard is supported by Xilinx’s MODELSIM simulator and the ICARUS VERILOG compiler.

HDLs are used to design Application Specific Integrated Circuits (ASICs). The problem with ASICs is that their manufacture is a very expensive process with little chance to redress design errors. For devices such as the Complex Programmable Logic Devices (CPLDs), HDLs allow the designer to design and manufacture complex digital systems. This involves a few simple steps culminating in the synthesis of a design in software and the implementation of the design in real hardware by the loading of a fuse file into the CPLD via a programmer. Sophisticated digital designs can be implemented in CPLDs without an expensive manufacturing process.

This chapter teaches VERILOG by example. I will not give a thorough description of VERILOG syntax. For this I have written an appendix to the reading brick which you should consult for the details as you read. Your VERILOG learning will not cease with this chapter either. We are learning VERILOG now so that when we introduce sequential circuits and Finite State Machines in the next few chapters, we will be able to describe the hardware implementations of these devices in an alternative exact form to other exact methods such as circuit schematics and state diagrams. The main difference with VERILOG is that, given a Complex Programmable Logic Device such as a Field Programmable Gate Array, you can move straight to circuit construction and testing.

The Field Programmable Gate Arrays (FPGAs) are a special case of CPLD in which the hardware is also reconfigurable. FPGAs are usually SRAM based devices. This means that they can be repeatedly reprogrammed with different designs. The configurations are also volatile. Volatile means that the configuration is lost when the FPGA is powered down. At each power up the fuse file has to be reloaded into the FPGA. The FPGA can be reprogrammed with a different design each time.

“Programming an FPGA” or any CPLD is a simple process, superficially akin to the programming of a microprocessor and is performed with an electronic programming device. The reconfigurability of FPGAs implies that failed implementations can be erased and reimplemented an arbitrary number of times at zero cost. Nowadays, FPGAs with millions of gates and clock speeds of hundreds of MHz are readily available at low cost. As one can imagine, FPGA based hardware designs can also achieve computational efficiencies by implementing large scale parallel computer architectures. For some tasks,
even a low end FPGA can out-perform the latest INTEL processor. As a result, FPGAs are rapidly replacing ASICs in many digital applications.

HDLs such as VERILOG differ from procedural programming languages in the following aspects,

(1) The concept of simulation time,

(2) Multiple threads or concurrently running procedures as one would expect in a system of parallel gates.

(3) The concepts of wired connections and primitive gates.

In this chapter we take a first look at VERILOG HDL. If you know how to program in C and you understand basic digital design then learning VERILOG will be easy. However it is important to keep in mind that the job of VERILOG is to wire gates and in this respect it is very different to C.

Throughout this chapter we will be referring to sequential circuits as well as combinational circuits. Sequential circuits are circuits that contain feedback so as to produce memory. These circuits respond to changing input signals by evolving through a sequence of states. They are in fact Finite State Machines. As such they superficially behave in much the same way that procedural programs work and it should therefore not be surprising that one can use a language such as procedural or behavioural VERILOG, to describe this kind of hardware.

5.2 A Quick look at VERILOG

In this section, we look at several simple VERILOG programs and study their behaviour. Don't worry if you don't understand all the details: we will examine things a bit more closely in the remainder of the chapter and elsewhere in the course. The ultimate aim of this course is to become comfortable with VERILOG and efficiently apply it to digital design.

First of all VERILOG can be used for several stages of the design process including,

(1) Hardware definition: the wiring of gates.

(2) Simulation of circuit behaviour. Check that the response of the circuit to externally imposed stimuli is that which was intended.

(3) Synthesis: the elaboration of a design as a system of interconnected gates. This can be performed by the VERILOG compiler.
Notice that the first and third appear to be the same thing. In fact they are not. As a digital designer using VERILOG, you can choose to design the circuit at the gate level so that the VERILOG synthesiser leads to a predictable circuit schematic. Alternatively, you could design it at the procedural level, where the hardware is specified as some interconnection of gates that carries out the “instructions” of a program written in procedural VERILOG. This is sort of like designing computer given a particular C-program that you want to run on it. Sounds strange? Let’s look at this in more detail.

5.2.1 The Different Levels of Abstraction in VERILOG

VERILOG supports design at different levels of abstraction.

(i) Procedural or algorithmic level (much like c code with if, case and loop statements). This level describes a system by concurrent algorithms (Behavioral). Each algorithm itself is sequential, that means it consists of a set of instructions that are executed one after the other. Functions, Tasks and Always blocks are the main elements. There is no regard to the structural realisation of the design.

(ii) Register transfer level (RTL uses registers connected by Boolean equations). Designs using the Register-Transfer Level specify the characteristics of a circuit by operations and the transfer of data between the registers. An explicit clock is used. RTL design contains exact timing bounds: operations are scheduled to occur at certain times. In fact RTL code is any code that is synthesisable.

(iii) Gate level (interconnected AND, NOR etc.). Within the logic level the characteristics of a system are described by logical links and their timing properties. All signals are discrete signals. They can only have definite logical values (‘0’, ‘1’, ‘X’, ‘Z’). The usable operations are predefined logic primitives (AND, OR, NOT etc gates). Using gate level modeling might not be a good idea for complex logic design.

(iv) Switch level (the switches are MOS transistors inside gates). The language also defines constructs that can be used to control the input and output of simulation. We are not concerned with this level in this course.

For example,

- Structural, which is a verbal wiring diagram without storage.

  ```verilog
  assign a=b & c | d;  /* "|" is a OR */
  assign d = e & (~c);
  ```

- Behavioural or Procedural refers to (i) above and is used for circuits with storage, or as a convenient way to write conditional logic.
$always @(posedge clk)$ // Execute the next statement on every rising clock edge.
$count <= count+1;$

Of course VERILOG is a unified language and any of the above design levels may appear in the same code. However there are some important things to look out for.

- Behavioural VERILOG may not be synthesisable. This is not a problem as long as we recognise that the language facilitates simulation of the behaviour of a circuit. We should be careful to distinguish the simulation-oriented aspects of VERILOG those aspects that are not synthesisable.

- Gate and RTL level VERILOG synthesises to a well defined circuit schematic. However you have to understand how to build the circuit you want in the first place.

### 5.2.2 ICARUS VERILOG

For our simulations (particularly when working off-campus on home computers) we will use a free, open source VERILOG compiler/simulator called ICARUS VERILOG that can be downloaded from,

http://www.icarus.com/eda/verilog/

ICARUS VERILOG is a convenient tool as it can be run as a simulator without the need for Xilinx ISE WebPACK or any hardware. We can refine our designs at the computer terminal in the early design phase without the overhead of an IDE.

ICARUS VERILOG is easy to use and has few command-line options. In the next section we will install and use it to run examples. You should try ICARUS right away and use it to test designs before working with Xilinx ISE WebPACK.

### 5.3 Structural VERILOG

In this and the next section we take a glimpse at the two most basic programming methodologies in VERILOG: **structural** and **behavioural** VERILOG.

#### 5.3.1 Gate Level Modelling

Primitive logic gates are part of the VERILOG language. Two properties can be specified, drive-strength and delay. Drive-strength specifies the strength at the gate outputs. The strongest output is a direct connection to a source, next comes a connection through a conducting transistor, then a resistive pull-up/down. The drive strength is usually not
specified, in which case the strength defaults to strong1 and strong0. Delays: If no delay is specified, then the gate has no propagation delay; if two delays are specified, the first represent the rise delay, the second the fall delay; if only one delay is specified, then rise and fall are equal.

**Delays are ignored in synthesis. Only used for simulation**

In the first example we look at structural modelling. Structural modelling can be done either with gates as described above or pure Boolean logic: they are equivalent for synthesis. In this example we use Boolean logic.

For more details on the syntax of statements used in structural modelling, see the appendix.

5.3.2 Example 1

Consider the digital circuit of Figure 55.

![Figure 55: A digital circuit.](image)

This circuit represents the Boolean function

\[ X = AB + C \]

Gate level VERILOG code that describes this circuit is shown below,

```verilog
// A simple Verilog program that implements the logic (A AND B) OR (C).
//
//This is the main program
module MyCircuit(A, B, C, O);
    input A, B, C;
    output O;
    wire w;

    MYAND MA(A, B, w);
    MYOR MO(C, w, O);
```

80
Verilog programs consist of modules. A module is the basic programming unit in VERILOG HDL. Each module is akin to a function in C: that is modules are lower level routines which taken together form the main program. It is generally good practice to use a separate module for each function in the circuit and to use a separate appropriately named file per module.

The first line of a module declaration specifies the name and port list (arguments). These port values represent the inputs and outputs of the gates. A module does not need to have any input and output ports. For example it is possible to write code which describes a process and can simulate the process without the need for inputs and outputs. The next few lines in each module consist of declarations of I/O type. Notice that in general we do not provide values for A, B and C. This is because they are inputs and outputs from an external circuit (or simulation block). Very often one of these parameters will be the system clock: a necessary requirement for a sequential circuit. More on this in a moment.

All the above program does is to wire a circuit. Although one could write an analogous C-program that appears to do the same operations; in VERILOG HDL, these “operations” are not really being performed at all. VERILOG is a circuit builder. This is the fundamental difference between HDL and procedural programming languages such as C. You should think of all inputs and outputs as physical wires. The wire declaration is specifically for device inter-connects.

Let us consider signals to be binary digital numbers. They’re wires that go between logic gates. They carry information without performing any computation. Signals are very important for transporting data, and they underlie everything one does in digital design.

The most basic kind of signal is the wire. The wire is used for continuous assignment. That is, if you have an expression, you can associate the result of that expression with
a named wire, and then you can use that wire name in another expression. It’s a lot like a variable, but you cannot reassign it once you have assigned it. Think of this as the same problem as resoldering the wire in a real circuit! The value of the wire is directly a function of what’s assigned to it, and you can only assign one thing. Continuous assignment is used to assign a value onto a wire in a module. It is the normal assignment outside of always or initial blocks (we’ll talk about these in a moment). Continuous assignment is done with an explicit assign statement or by assigning a value to a wire during its declaration. Note that continuous assignment statements are concurrent and are continuously executed during simulation. The order of assign statements does not matter. Any change in any of the right-hand-side inputs will immediately change a left-hand-side output. The assign statement causes the left-hand side to be updated any time the right hand side changes. This agrees with the simple notion of wiring a circuit. Wiring is not a causal process. It happens prior to simulation. It is for this reason that assign is referred to as continuous assignment. Continuous assignment is a foreign concept to procedural languages. The inputs and outputs are wires, although an output can be reassigned as a reg.

A reg is assigned in behavioral code, which we’ll explain in a moment. A reg will be much like a wire for behavioral code that is combinational, or it will correspond to a physical register (a flip-flop or group of flip-flops) in synchronous logic.

Examples of continuous assignment.

```verilog
wire [1:0] a = 2'b01; // assigned on declaration
assign b = c & d;   // using assign statement
assign d = x | y;   /* The order of the assign statements does not matter. */
```

The final thing to notice is the way that the module MyCircuit instantiates the MYAND and MYOR modules. Instantiation is in fact a kind of declaration. We will see another example of this in a minute. Instantiation allows us to decouple our modules so that our VERILOG coding can be modular. We will look at another way of decoupling code in a moment using functions.

5.3.3 Simulation: Test Benches

The lines of code presented above do nothing by themselves. In order for it to “do” something we need to simulate.

We can execute the functions defined in the circuit by providing stimuli. To simulate the circuit we write a non-synthesisable module we call a test bench. The module is non-synthesisable because it will have no manifestation in hardware. This allows the VERILOG simulator to produce some visible output to test your design.

```verilog
//The test bench
```
module My_TB;
reg A, B, C;
wire O;

// instantiate your circuit
MyCircuit MCCT(A, B, C, O);
//Behavioural code block generates stimuli to test MyCircuit

initial begin
A = 0; B = 0; C = 0;
#50 $display("A = %b, B = %b, C = %b, Output O = %b",A,B,C,O);

A = 1; B = 0; C = 0;
#50 $display("A = %b, B = %b, C = %b, Output O = %b",A,B,C,O);

A = 0; B = 1; C = 0;
#50 $display("A = %b, B = %b, C = %b, Output O = %b",A,B,C,O);

A = 0; B = 1; C = 0;
#50 $display("A = %b, B = %b, C = %b, Output O = %b",A,B,C,O);

A = 1; B = 1; C = 0;
#50 $display("A = %b, B = %b, C = %b, Output O = %b",A,B,C,O);

A = 0; B = 0; C = 1;
#50 $display("A = %b, B = %b, C = %b, Output O = %b",A,B,C,O);
end
endmodule

The test bench provides initial values for the simulation and then instantiates the circuit. Simulation begins with the initial statement. The variables A,B,C and O are local but must have the same type as the corresponding variables in the main module. Note that there is also a port inout which is a bidirectional I/O. As noted in the appendix, inputs, inouts and outputs are mostly wires.

The initial statement is executed once at startup. When there is more than one line in the initial block then these are straddled by the begin...end lines. The $display keyword is simply a command to dump the parameters in the specified format to the console. The %b indicates bit format and will cause $display to dump variables as a string of 1’s and 0’s. Notice the similarity of this syntax to the following printf syntax in C:

printf("A = %d, B = %d, C = %d, Output O = %b",A,B,C,O);

83
where I have opted to use the \%d format specifier which applies to integers in C, because C does not have a \%b format. VERILOG does however have a \%d format specifier which means 'decimal'.

Finally note the \#50. This simply tells the simulator to execute this \$display dump on the 50th cycle of the system CLOCK.

In order to see this circuit in action, download and install ICARUS VERILOG. ICARUS VERILOG compiles VERILOG code into a NETLIST in much the same way as a C-compiler such as GCC or SDCC compiles C-source code into an executable binary (".exe" under WINDOWS). ICARUS VERILOG runs at the command prompt. The \$display command dumps variables to STDOUT for viewing in much the same way as a terminal (DOS-like) C-program would do using the \printf() command. After installing ICARUS VERILOG download example1.v (the above example) from the web. You can compile your design .v-file into a NETLIST from the command prompt (WINDOWS CMD) as follows,

`iverilog example1.v -o example1`

The same syntax applies if you are running LINUX in a console or X-terminal. Note that example1.v is the VERILOG source file. The NETLIST example1 is the simulator output which can be run at the command prompt as follows,

`.example1`

If all goes well you should see an output like the following,

```
|>/example1
A = 0, B = 0, C = 0, Output O = 0
A = 1, B = 0, C = 0, Output O = 0
A = 0, B = 1, C = 0, Output O = 0
A = 0, B = 1, C = 0, Output O = 0
A = 1, B = 1, C = 0, Output O = 1
A = 0, B = 0, C = 1, Output O = 1
```

Notice that in the initial statement, there are only six Display commands. Hence there are only six lines of output. As you may appreciate, certain sequential circuits such as error correction decoders, computer ALUs,... respond synchronously to the system clock and may run forever. The above circuit does not behave in this way. In fact the above simulation was somewhat simplistic. All we had was some combinational logic which we provided inputs to. The initial block explored a few combinations and then terminated. In the next section, we will introduce the always statement which will allow us to do sequential logic.
This concludes our introduction to gate level VERILOG. This was rather trivial as the circuit is **combinational**. The details of VERILOG syntax can be found in the appendix which is an edited of Peter Nyasulu’s Introduction to Verilog [5]. The appendix clarifies the notions that have just been introduced and rounds out your knowledge to a more advanced level. Please consult the appendix while reading the next section.

## 5.4 Behavioural Modelling

VERILOG procedural statements are used to model a design at a higher level of abstraction. However small changes in coding methods can cause large changes in the hardware generated. Procedural VERILOG is a very powerful technique for implementing complex designs, but care has to be taken that you understand the hardware you produce.

Procedural statements can only be used in VERILOG procedures and not in structural modules treated above. In order to clarify, let us continue with the previous example. In example 2 we are going take the combinational circuit of example 1 and give it a procedural block to do simulation.

### 5.4.1 Example 2

Replace the above testbench with that shown in the following program snippet. In the last section we introduced one of the two most important blocks in behavioural VERILOG; the **initial** block. Below we introduce the other important block; the **always** block. Like continuous assignment, the **always** block is a concurrent statement that is continuously executed during simulation.

The main new thing to notice is the introduction of a new port: the **clk** input. The input **clk** is the system clock input. In an FPGA it is provided by a system fed from an external crystal or an electronic oscillator.

```verilog
//The test bench
module My_TB;
reg clk;
reg A, B, C;
wire O;

// instantiate your circuit
MyCircuit MCCT(clk, A, B, C, O);
//Behavioural code block generates stimuli to test MyCircuit

initial begin
clk=0;
```

85
A = 0; B = 0; C = 1;
$display("A = %b, B = %b, C = %b, Output O = %b",A,B,C,O);
forever #1 clk=~clk;
end

always@(posedge clk) begin
  #1 A = ~A;
  #1 B = B&C;
  #1 $display("A = %b, B = %b, C = %b, Output O = %b",A,B,C,O);
end
endmodule

In order to simulate sequential logic, the \texttt{clk} input has to be provided by the test bench. The \texttt{forever} statement gives the clock its oscillatory characteristic. The statement reads "forever at the current step of the system clock, set the variable \texttt{clk} to \texttt{clk}". VERILOG uses the clock variable as the means by which the simulation is executed.

The Event Control \texttt{@} causes a statement or begin-end block to be executed only at the occurrence of specified events. An event is a change in a variable, and the change may be: a positive edge, a negative edge, or either (a level change), and is specified by the keyword \texttt{posedge}, \texttt{negedge}, or no keyword respectively. Several events can be combined with the \texttt{or} keyword. Event specification begins with the character \texttt{@} and are usually used in \texttt{always} statements. In this example, the clock has been set to oscillate by the \texttt{forever} statement in the \texttt{initial} block, and the \texttt{always} block continuously executes on each of its positive edges. Note that for synthesis one cannot combine level and edge changes in the same list, as this does not make sense electronically.

Consider the following examples. For flip-flop and register synthesis, the standard list contains only a clock and an optional reset. For synthesis to give combinational logic, the list must specify only level changes and must contain all the variables appearing in the right-hand-side of statements in the block, so that the left-hand-sides are updated virtually instantaneously. Normally however we try to put combinational logic circuits into structural blocks.

The \texttt{always@} statement indicates that the lines in the block are executed at the event of a positive edge trigger on the input clock, \texttt{clk}. The \#1's indicate that the respective commands should be executed at the respective delays. Note that while the \texttt{clk} can be supplied electronically and therefore makes sense in synthesis, the \# delays are NOT synthesizable. They are in the code for simulation purposes only.

This test bench is very different to that of example 1 in that it executes forever. Download example 2 and run it using ICARUS VERILOG.
5.4.2 Behavioural VERILOG

Procedural or behavioural VERILOG consists of the code contained predominantly inside the `always` blocks. It is used to execute sequential logic. Sequential logic is basically combinational logic that is sensitive to a clock edge rather than signal inputs. This is how we get flip-flops (registers) that can store signals. In combinational logic, the computations in the always block are "instantaneous" with respect to the change in any input clock or whatever signal is used to trigger updates. With sequential logic, the inputs may change arbitrarily, but they only matter at the time of the clock edge, after which the signals are assigned.

An example is that in the code snippet above (but this is just a simulator). Notice that it consists of a sequence of assignments with Boolean logic to the right of an "=" sign and assignment targets to the left. Each assignment is prefixed by the # time delay operator. Procedural VERILOG runs inside the (always and initial blocks). In addition to the strong similarity between the operators and keywords of the C-programming language, VERILOG is most similar to C in its procedural syntax. Only reg variables (and their bit/part-selects and concatenations) can be placed left of the "=" in procedures. The right hand side of the assignment is an expression which may use any of the operator types described in the appendix.

Please refer to the appendix for detailed explanations of the meanings of the operators and operands used in the procedural code.

5.4.3 Example 3

Example 3 introduces a few new ideas including synthesisable procedural code (always block), initialisation by reset, signed numbers, temporal execution and the use of functions. Each of these things is explained in detail in the appendix.

The example performs addition and multiplication of signed 8-bit numbers. Again the code is divided into a synthesizable module and a test bench module. The test bench will never be synthesised and can therefore use all the VERILOG commands (you do not necessarily need to of course).

```verilog
//Procedural VERILOG
module Behavioural_Example(clk, reset_, A, B, C, D);

input clk, reset_;
    input signed [7:0] A, B;
    output signed [7:0] C;
    output signed [15:0] D;

    /* Declare sum initial values */
```
reg signed [7:0] X;
reg signed [7:0] Y;
reg signed [7:0] P = 3;

assign C = X;
assign D = Y;

function [7:0] signed_add; // signed function
    input signed [7:0] X1, X2;
signed_add = X1 + X2; // signed conversion
endfunction

function [15:0] signed_mult;
    input signed [7:0] X1, X2; // signed input port declaration
signed_mult = X1 * X2;
endfunction

always@(posedge clk) begin
    if(!reset_) begin
        X <= A;
        Y <= B;
    end else begin
        X <= signed_add(X, P);
        Y <= signed_mult(X, P);
    end
end

//The test bench
module My_TB;
reg CLOCK;
reg RESET;
reg signed [7:0] Ai, Bi;
wire signed [7:0] Co;
wire signed [15:0] Do;

// instantiate your circuit
Behavioural_Example BE(CLOCK, RESET, Ai, Bi, Co, Do);

initial begin
    CLOCK=0;
    RESET=0;
end
Verilog-2001 provides for signed as well as unsigned arithmetic. A `reg` variable, a net (`wire`), or a function output (see below) can be declared as signed by including the keyword `signed` in the declaration, for example, "`reg signed [15:0] A`". Likewise a module’s ports, may be declared as signed, for example, "`output signed [15:0] A`". A numerical literal is signed if the letter "s" is included after the base, for example, `8’sb11111111` is an 8-bit 2’s complement number, -1.

Operations and comparisons on signed variables and nets follow the rules for 2’s complement arithmetic, in both simulation and synthesis. However for signed arithmetic to be simulated and synthesised, all operands in an expression must be signed, that is, there is no up-casting as in MATLAB.

The first few lines in the module are the familiar declarations apart from the `signed` keyword. By running the above code, you can confirm that the decimal format `%d` is a signed magnitude whereas the bit format `%b` is a 2’s complement number. As with multiplication and division, care should be taken with synthesis.

Next we declare some initial registers for the `always` block. You may wonder, why use the variables X and Y at all? Can’t we just compute the outputs C and D in the `always` block without introducing the variables X and Y? Recall that only `regs` can appear on the left hand side of `=` and `<=` in procedural blocks, so make sure that you declare
C and D to be regs first. Also the output variables are not readable at all unless they are declared as regs. On the web I have two versions of the above program. The listing example3a.v is the above listing. In example3b.v I show how to provide direct assignment of C and D.

I should summarise this point as in Wakerly’s book (see Wakerly p. 298). You have basically two mechanisms to produce output from a module.

1. If for any reason the port must be declared as a wire (remember that VERILOG will assume an output port to be a wire unless otherwise specified: after all modules represent circuits and circuits are connected by nets or wires not by registers or memories), define an “output” reg variable (X,Y in the above example), to the module output net (C,D respectively in the above example).

2. Declare the output port to be of type reg and use the always block (procedural code) to assign values to it directly. (example3b.v)

Let’s move on. In this example I wanted to show you a way to provide and initial state for your system. This should seem a bit odd given that we already have the inputs A and B which ought to be used to stimulate the system. In digital electronics, circuits often have reset buttons to reset the state of a digital device to a known value. What I show is a trick which allows the reset to be used in synthesis. As you can see, the reset is the first thing we test for with the 'if'. The 'else' is for the main process. This is a coding convention that synthesiser heuristics are designed to look for. For real hardware, resets and other external signals are all you have to set initial values for your registers. There is no such thing as 'initialisation' in hardware or VERILOG, other than what you create explicitly based on signals.

RTL (non-blocking) assignments (<=), that follow each other in the code, are done in parallel whereas the blocking operator ('=' operator) is executed sequentially. Let’s discuss in detail blocking versus non-blocking assignments in procedures.

Blocking assignment (using the '=' operator) are called ”blocking” because, conceptually, the logic pauses to finish the computation inside the entire block and all concurrent blocks and make the assignment before moving on. That is, they block the execution of subsequent procedural statements in the same always block until the assignment of the current line has been made. Blocking statements are what C-code would do.

Non-blocking assignments (using the '<=' operator (only available in always blocks)) post the computation to be assigned an infinitesimal time later; conceptually, the logic runs past the assignment statement (does not block), setting it up but not completing it immediately.

Let us look at a few examples. Procedural code with blocking assignments (=) are done sequentially in the order the statements are written. A second assignment is not started until the preceding one is complete. Consider an example of blocking code. Let us suppose that the variable in has been supplied with an initial value of 3.
wire [3:0] in;
reg [3:0] x, y, z;
always @(posedge clock) begin
    x = in + 1;
    y = x + 1;
    z = y + 1;
end

The following output would occur,

**time: values**
0: in=3, x='hx, y='hx, z='hx
1: in=3, x=4, y=5, z=6

The time t=0 is before the first clock edge. In the case of blocking assignment, all expressions are evaluated to produce the final result. Obviously it can contain no loops: it must be combinational logic. However the above is a clumsy implementation because x and y will not be synthesised. The synthesis only need involve *in* and *z*.

Consider the following example of non-blocking code.

wire [3:0] in;
reg [3:0] x, y, z;
always @(posedge clock) begin
    x <= in + 1;
    y <= x + 1;
    z <= y + 1;
end

When this is executed the following will be produced,

**time: values**
0: in=3, x='hx, y='hx, z='hx
1: in=3, x=4, y='hx, z='hx
2: in=3, x=4, y=5, z='hx
3: in=3, x=4, y=5, z=6

At t=0, x, y and z have no values and hence have the unknown value *x* assigned to them. On the next step at delay t=1, x has the value 4 while y and z are still *x*. Notice how all statements are executed concurrently, but assignment occurs only after the statements are executed. The values at the *n*th step are those that resulted from the execution at the *(n-1)*th step. Hence an execution at the *n*th step, uses values computed at the *(n-1)*th step.
What happens if we change the order in which the code lines are written. In the case of the non-blocking code nothing will change: order does not matter.

For the blocking code, let us consider the following case.

```verilog
wire [3:0] in;
reg [3:0] x, y, z;
x = in + 1;
z = y + 1;
y = x + 1;
end
```

The following output would occur,

```
time: values
0: in=3, x='hx, y='hx, z='hx
1: in=3, x=4, y=5, z=x
2: in=3, x=4, y=5, z=6
```

The key idea behind using non-blocking assignments is that you can take advantage of the time delta delay in the assignment, making your order of assignments generally irrelevant. The only time the order matters is when you assign to the same signal. For instance, this:

```verilog
reg z;
always @(posedge clock) begin
  z <= 0; // by default, z is zero
  if (in_signal) begin
    z <= 1; // unless the input signal says otherwise
  end
end
```

Assignments like this are a really good way to set a default value that is followed by complex logic that may or may not assign something else to the signal.

To summarise there are some rules. We should use blocking assignments ('=' ) with combinational code, and non-blocking ('<=') assignments with sequential code. Moreover as a general rule, do not mix the use of blocking and non-blocking assignments in the same `always` block. Also do not make assignments to the same variable in two different `always` blocks. For example, when synthesising to flip-flops as in an `@ (posedge... ) procedure, always use non-blocking. Without that you will be racing with the flip-flops in the other modules.

Finally we have the test bench. Once again I have been careful to formulate this example so that the test bench as a non-synthesisable VERILOG is in a separate module.
The first thing to note is the way that I terminate the run after 20 time steps with the
\$finish keyword. The **always** block in the test bench does two non-synthesisable tasks. Firstly it defines a reset back to a high after the fifth time step. This is an artificial stimulus that would be replaced by a reset voltage in an FPGA. Finally there is the \$display directive that prints out some relevant data.

Once again download example3.v and run it in ICARUS VERILOG.

### 5.5 Conclusion

This concludes our short introduction to VERILOG. I believe that you know all you need to if you understand this chapter. For practice, I have posted a huge number of VERILOG simulations of different sequential circuits on the web which I downloaded from

http://asic.co.in/Index_files/verilogexamples.htm

Write test benches and try them with ICARUS VERILOG.
6 Sequential Logic

We have already heard mention of sequential circuits in the last chapter on VERILOG HDL. We needed some definition there in order to introduce the minimum VERILOG necessary for this chapter. We now treat sequential circuits in detail. Sequential circuits are the key building blocks of complex digital systems. They include circuits such as flip flops, counters, registers... The approach taken is to first understand sequential circuits in detail and to see how to implement these in VERILOG. This will allow us to build systems from latches, flip flops, registers, counters etc. We will learn in the labs how to implement these in FPGA. In this chapter we will make the connection between the sequential circuits and the Finite State Machines of the next chapter. With the ability to design state machines, you will be able to implement quite complex systems in FPGAs. The Xilinx XC2S50 FPGA on the Pegasus boards used in the labs has 1,536 Configurable logic block (CLB) Flip flops and 176 user IO. One can therefore implement quite complex state machine designs in this FPGA.

6.1 Storage Elements

Thus far we have looked at combinational circuits where the output values depend only on the input signals applied. Before moving onto Complex Programmable Logic Devices which will significantly enhance our capability of designing very complex digital systems such as mp3 codecs and Digital Video Broadcast transmitters, we are going to take a look at the key enabling technology for the digital age: sequential circuits.

Sequential circuits have the property that the outputs depend not only on the current inputs, but also on the past behaviour of the circuit. Essentially the useful function served by these circuits depends on how they remember their previous states in response to a clock or other input that evolves their state. Such circuits include storage elements that store the values of logic signals. These circuits are so important yet so simple that their study has been abstracted by the mathematical notion of a finite state machine which we discuss in this chapter.

In a sequential circuit, the contents of the storage elements represent the state of the circuit. The input value changes may leave the circuit in the same state or cause it to change to a new state. With time, the circuit changes through a sequence of states as a result of changes in the input values. We motivate the need for sequential circuits by looking at a simple alarm control system.

6.1.1 Alarm Control System

As an example of a practical need for circuits that have memory, consider a controller for an alarm circuit shown in the block diagram of Figure 56. The desired operation is that the alarm fires when the sensor generates a positive voltage in response to some
undesirable event. This trigger signal is referred to as a set signal. Once the alarm is triggered it has to remain on to maximise the chance that someone will hear it. The alarm circuit must remember its state when the signal that triggered the set condition is removed (returns to zero). The alarm is turned off manually by a reset signal. The circuit for an alarm needs a memory element to remember that the alarm has been triggered.

![Alarm control circuit](image)

Figure 56: Alarm control circuit.

### 6.1.2 Simple bistable memory element

The memory behaviour can be achieved by providing a feedback path. The simplest demonstration of this is the **Schmitt Trigger** which you should have already seen. The **Schmitt Trigger** is an analogue input device that has memory. In a Schmitt Trigger the output remembers the last extreme limit breeched by an input analogue signal. A Schmitt trigger is essentially a comparator with feedback as shown in Figure 57.

![Schmitt trigger circuit](image)

Figure 57: A Schmitt trigger circuit.

The bistable element shown in Figure 58 can be used as the basis for memory circuits in digital applications. It consists of two cross-connected inverters. If the voltage at A is
a 1, then the voltage at B is a 0, which is again consistent with the 1 at A. If A changes to a 0 then B flips to a 1. Thus the circuit has two possible states. Despite this bistability which is crucial to the operation of a memory element, this circuit has no inputs and no way of changing its state.

Figure 58: A bistable latch circuit.

The circuit of Figure 59 is an extension of the bistable element. In fact this figure was lifted directly from the datasheet of the 74HC/HCT373 Octal D-type transparent latch. It uses two transmission gates (TGs) to control which parts of the bistable element are connected together. One gate is used to connect the Data (D) input terminal and the other to connect the feedback between the inverters. When the control signal (LE) is a 1, the D signal is connected to the input of the lower inverter. At the same time the feedback loop is disconnected (open circuit). When LE changes to 0, the feedback path is closed and the memory element will retain its state so long as LE remains low.

Unfortunately, this circuit cannot work as an alarm controller but is a very useful circuit in many applications such as the 74HC373 Octal D-type transparent latch.

For later reference, the data sheet of the 74HC373 is quite detailed and provides examples of all the important physical timing parameters we learn in this course. Please download it from [http: www.datasheetcatalog.comdatasheets,df74HC74HC373.shtml](http: www.datasheetcatalog.comdatasheets,df74HC74HC373.shtml) and read it. As you are reading it, I advise you to pay attention to the next few sections of this chapter, so that you understand how this device operates as a memory element. It is actually not clear from the datasheet. The 74HC373 latch may be used as a cell in a static RAM (SRAM) as shown in Figure 60 of an SRAM (static RAM) cell. By referring to Figure 59, can you understand how the SRAM cell of Figure 60 works?

### 6.1.3 The Basic SR Latch

The circuit shown in Figure 61 is a basic memory element built with NOR gates. With \( R = S = 0 \) the circuit remains in its current state by virtue of the feedback path. When
Figure 59: A controlled memory element that is the basis of the 74HC/HCT373 Octal D-type transparent latch.

Figure 60: An SRAM cell consists of the same type of latch as in Figure 59.

$S = 1$ and $R = 0$, the circuit is **set** into a state where $Q = 1$ and $\overline{Q} = 0$. When $S = 0$ and $R = 1$, the circuit is **reset** into a state where $Q = 0$ and $\overline{Q} = 1$. This circuit is a
basic SR latch.

![Basic SR latch diagram](image)

Figure 61: Basic SR latch.

Note that for the desired operation of this circuit, $Q$ and $\overline{Q}$, will always be inverses of each other. Despite this, you should be aware that this circuit has a special state in which this does not occur. When $S = 1$ and $R = 1$, the latch is forced into a state where $Q = \overline{Q} = 0$. This state is unstable. To see this, consider the timing diagram of Figure 62. If both inputs are initially in the 1 state and at least one remains in this state at the next transition to a 0 then the states of $Q$ and $\overline{Q}$ are stable as expected (e.g. $t_5 - t_6$). However if both inputs are in the 1 state and they both attempt to transition to the 0 state at the same time, then a race condition ensues. This occurs at time $t_{10}$ in Figure 62. The race condition arises because $R = 1$ and $S = 1$ force the two outputs to be zero. Assuming that the input high to low transitions occur simultaneously, then we are retransitioning to the hold or memory state.

Clearly the actual output state that ensues when both $R$ and $S$ go low at $t_{10}$ depends on the propagation delays of the gates. The gate with the shortest delay will hit zero first and win the race. Suppose that $S$ wins the race. This will cause the $Q$ output to go low because the $R$ input is still high. When $R$ finally goes low, $Q$ will be latched in the hold state. In actual fact of cause the outcome is unpredictable because gates on a VLSI latch will have nominally the same propagation delay. The only way to deal with this is to either disallow the $(1, 1)$ state in the design or use another more complicated circuit that avoids it. Both approaches are used in practice.

Make sure that you understand this discussion.

The truth table summarises the behaviour of the basic SR latch.
The circuit in Figure 61 can play the role of the alarm controller in Figure 56, by connecting the set signal to the \( S \) input and the reset input to the \( R \) input. The \( Q \) outputs provide the desired alarm indicator. To initialise the operation of the alarm system, the latch is reset. When the sensor generates the logic value 1, the latch is set and \( Q \) becomes equal to 1. This fires the alarm. If the sensor output returns to 0, the latch retains its state where \( Q = 1 \); hence the alarm remains turned on. The only way to turn off the alarm is by resetting the latch, which is accomplished by making the reset input equal to 1.

### 6.1.4 The Gated SR Latch

The basic SR latch above is said to be a transparent device because any change of its inputs is immediately evident at its outputs. Can you understand why the 74HC373 is described as a transparent latch? In some cases however it is desirable to add an enable signal to the basic SR latch that allows control over when the circuit can change its state. Such a circuit is referred to as a gated SR latch and is shown in Figure 63.

The modified circuit includes two AND gates that provide the desired control. When the control signal Clk is equal to 0, the \( S' \) and \( R' \) inputs to the latch will be 0, regardless of the values of signals \( S \) and \( R \). Hence the latch will maintain its existing state as long as
as Clk = 0. When Clk changes to 1, the S' and R' signals will be the same as the S and R signals, respectively. Therefore, in this mode the latch will behave as we previously described.

We use the term clock (or Clk) for the control signal that allows the latch to be set or reset, rather than call it the enable signal. We should be aware that in gated latches, the clock controls the device by levels. This is to be distinguished from flip flops which are edge controlled as we are about to see.

Figure 63: Gated SR latch.

The truth table summarises the behaviour of the gated SR latch.

<table>
<thead>
<tr>
<th>$CLK$</th>
<th>$S$</th>
<th>$R$</th>
<th>$Q(t + 1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$Q(t)$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$Q(t)$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$\times$</td>
</tr>
</tbody>
</table>

The timing diagram for the gated SR latch is shown in Figure 64.

Finally note that the ambiguity can be eliminated by making sure that the (1, 1) never occurs. Figure 65 shows how this can be done using a JK latch.

6.1.5 The Gated SR Latch with NAND gates

Figure 66 shows a similar gated SR latch circuit based on NAND gates.

Note the reversal of the definitions of S and R to produce the same $Q$ and $\bar{Q}$. The truth table of this gated SR latch is shown below.
Figure 64: The gated SR latch timing diagram

Figure 65: The JK latch.

Figure 66: The gated SR latch with NAND gates

<table>
<thead>
<tr>
<th>$CLK$</th>
<th>$S$</th>
<th>$R$</th>
<th>$Q(t+1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>×</td>
<td>×</td>
<td>$Q(t)$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$Q(t)$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>×</td>
</tr>
</tbody>
</table>
6.1.6 The Gated D Latch

In the last sections we saw that the SR latch could serve the purpose of the alarm control circuit. However we often require a latch which has a single input, called D, whose value it stores. Such a circuit is referred to as a D latch and is shown in Figure 67(a). The latch can be thought of as a storage element that stores a single bit. Even more useful is a D latch controlled by a clock input as shown in Figure 67(b). This device is referred to as a gated D latch.

The gated D latch is useful in multiplier or adder circuits where carries have to be stored from one operation to the next. The truth table of the gated D latch is,

![D latch and truth table](image)

Figure 67: (a) The D latch and (b) the gated D latch and (c) Symbol.

<table>
<thead>
<tr>
<th>$Clk$</th>
<th>$D$</th>
<th>$Q(t+1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>×</td>
<td>$Q(t)$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
In this circuit it is important to notice that the undesirable situation where $R = S = 1$ does not arise. In the **gated D latch**, the output $Q$ merely tracks the value of the input $D$ while $Clk = 1$. As soon as $Clk$ goes to 0, the state of the latch is frozen until the next time the clock signal goes to 1. Therefore, the gated D latch stores the value of the $D$ input at a time determined by the clock.

We note that the **gated D latch** of Figure 67 behaves in the same way as the circuit of one of the latches in the 74HC373 Octal D-type transparent latch of Figure 59 where the clock is replaced by $LE$. The advantage of the latch in the 74HC373 chip however is that it can be built with fewer transistors.

The VERILOG code for the **gated D latch** is.

```verilog
module gated_D_latch (D, Clk, Q);
  input D, Clk;
  output Q;
  reg Q;
  always @(D or Clk)
    if (Clk)
      Q <= D;
endmodule
```

**Exercise** Implement this code and verify the timing diagram of Figure 68.

![Timing Diagram](image)

Figure 68: The gated SD latch timing diagram

### 6.1.7 Master-Slave D Flip-Flops

Up until now we have looked at latches. These are level triggered memory devices that can change state (i.e. are **transparent**) when the clock is **active** (equal to a 1 in the
examples). The active edge is that which triggers the change of state. Such devices can change their state at any time when the clock is active. There is also a need for devices that cannot change their states more than once when the clock is active. These edge-triggered devices are referred to as flip-flops.

The circuit of Figure 69 utilises two D latches. The first is the master that changes its state when the clock = 1. The second is the slave and because of the inverter it can only change its state when the clock = 0. When the clock is high, the master D latch transparently latches the input signal D into P and Q remains unchanged. When the clock goes low, the P signal ceases to follow D. Due to the inverter, the slave transparently latches the current value of P into Q. Note that the slave can undergo at most one change of state per clock cycle. From the point of view of the external observer, the system changes its state at the negative edge of the clock. Thus, to the user, the active edge of the clock is the negative edge. Note that there is also a master slave D flip flop that triggers with a positive edge. Its symbol is also shown in Figure 69.

This can be confirmed by inspecting the timing diagram of Figure 69(b). Notice the

Figure 69: (a) The Master Slave D Flip-Flop, (b) Timing (c) Symbol.
small amount of artistically introduced timing jitter and propagation delay drawn into the clock traces. Timing jitter refers to the slightly random times at which the clock transitions actually occur. Timing jitter is caused by phase noise in crystal oscillators. Although this is usually not a problem in digital electronics (computer clocks usually have disgraceful timing jitter performance), should it turn out that you need accurate timing for ps-scale applications such as some GPS or wireless applications, then a good quality clock such as a temperature controlled crystal oscillator (TCXO) or an oven controlled crystal oscillator (OCXO) might be a better choice. These have timing jitters more than a hundred times lower than ordinary crystal oscillators.

The VERILOG code for the master slave D flip-flop is.

```verilog
module master_slave_D_flipflop (D, Clock, Q);
    input D, Clock;
    output Q;
    reg Q;
    always @(negedge Clock)
        Q <= D;
endmodule
```

**Exercise** Implement this code and verify the timing diagram of Figure 69.

We conclude that the master-slave D flip-flop has the property that its output $Q$ will only change state at the negative edge of the clock. In general, a flip flop denotes a storage element that changes its output at the active edge of the clock. In the symbol of Figure 69(c), the > sign denotes the clock input and the bubble o indicates that the clock active edge is negative as shown in Figure 70.

![Figure 70: D FF symbols.](image)

The output of the master-slave D flip-flop in Figure 69 responds to the negative edge of the clock signal. The circuit can be changed to respond to the positive clock edge by connecting the slave stage directly to the clock and the master stage to the complement of the clock.
6.1.8 Additional Inputs

Often, flip-flops feature additional inputs for added functionality. For example, it is often useful to have an additional input to force the flip flop into a known state when the circuit is first powered up, or when a 'reset' button is pressed. These additional inputs can be synchronous or asynchronous. Synchronous inputs are 'set' and 'reset' (which force the flip flop output to 1 and 0, respectively, while the corresponding asynchronous inputs are 'preset' and 'clear').

6.1.9 D Flip-Flops with Clear and Preset

Later in this chapter we will be looking at counters and registers. These devices will need to be cleared and preset at regular intervals. For example, an up-down counter may be required to count the number of samples entered into a wireless PHY layer processor (digital receiver) from an Analogue to Digital Converter (ADC). The ADC digitises the raw analog baseband signals obtained from the radiofrequency or analogue part of the receiver. A fixed number of samples may, for example, be fed to a Costa’s style carrier recovery loop for the elimination of carrier offsets in digital hardware. The Carrier offset refers to the difference in the frequency $\Delta f$ of the local oscillators of the radio transmitter and the receiver as shown in the simplified radio link in Figure 71. In order to guarantee that the same number of samples are calculated each time, the counter has to be cleared and preset for each radio packet received off the air.

![Figure 71: Counters must be cleared and preset. Here the sample counter must count a certain number of samples from zero every time a new radio packet arrives](image)

The circuit of Figure 72(a) shows an implementation of the circuit in Figure 69 in NAND gates. The master stage is just the gated D latch of Figure 67(b). The slave stage is the gated SR latch of Figure 66. A simple way of providing the clear and preset functionality is to add extra inputs to the NAND gates in the cross coupled latches. Placing a 00 on the clear will force both latched into the $Q = 0$ state. Placing a 0 on the preset will force both latched into the $Q = 1$ state.

A clear and preset both of 1 will have no effect. To indicate that the clear and the preset are active when 0 we place an overbar on their names in the circuit schematic.
It should be clear that clear and preset cannot be active at the same time. The circuit symbol is shown in Figure 72(b).

![Circuit Diagram](attachment:image.jpg)

(a)

(b)

Figure 72: A master slave D flip-flop with clear and preset.

In Figure 72(b) the effect of the 0 on either the clear or preset input is immediate. For example, if clear = 0 the flip-flop goes into the same state $Q = 0$ immediately, regardless of the value of the clock signal. In such a circuit, where the clear signal is used to clear a flip-flop without regard to the clock signal, we say that the clear is an asynchronous clear. In practice it is often desirable to clear the flip-flops on the active edge of the clock. This is referred to as synchronous clear and an example is demonstrated in Figure 73.

The VERILOG code for a D flip-flop with asynchronous preset is as follows.

```verilog
module flipflop (D, Clock, Presetn, Q);
  input D, Clock, Presetn;
  output Q;
  reg Q;
  always @(negedge Presetn or negedge Clock)
    if (!Presetn)
```

107
Figure 73: **Synchronous reset** for a D flip-flop.

![Synchronous reset](image)

Exercise Rewrite this code to include *asynchronous clear* functionality.

The VERILOG code for a D flip-flop with *synchronous reset* is as follows.

```verilog
module flipflop (D, Clock, Resetn, Q);
  input D, Clock, Resetn;
  output Q;
  reg Q;
  always @(negedge Clock)
    if (!Resetn)
      Q <= 0;
    else
      Q <= D;
endmodule
```

Notice that the only difference to the asynchronous code is that the reset signal is only activated at the CLOCK edge.

### 6.1.10 T flip flops

The D flip-flop is a useful building block in certain storage applications. By including some simple logic circuitry to drive its input, the D flip-flop can play a different role. An interesting modification is shown in Figure 74. This circuit uses a positive edge triggered D flip-flop. The *feedback* connections make the input signal D equal to either the value...
of \( Q \) or \( \overline{Q} \) under the control of the \( T \) signal. On each positive edge of the clock, the flip flop may change its state \( Q(t) \). If \( T = 0 \) then \( D = Q \) and the new state will be the same, \( Q(t + 1) = Q(t) \). If \( T = 1 \) then \( D = \overline{Q} \) and the new state will be \( Q(t + 1) = \overline{Q}(t) \). Therefore the overall action of the circuit is that it remains in its present state if \( T = 0 \), and it reverses the state if \( T = 1 \). The operation of the circuit is described in the truth table below.

<table>
<thead>
<tr>
<th>( T )</th>
<th>( Q(t + 1) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( Q(t) )</td>
</tr>
<tr>
<td>1</td>
<td>( \overline{Q}(t) )</td>
</tr>
</tbody>
</table>

![Figure 74: The T flip-flop.](image)

Any circuit that implements this truth table is referred to as a \( T \) flip-flop. The name \( T \) flip-flop derives from the name of the circuit with toggles the input state when \( T = 1 \).

There are several circuit implementations of the \( T \) flip-flop. Wakerly (Ch 7.2.11) discusses two types of \( T \) flip flops. The first is a flip flop with only one input (see figure 75(a) below). This flip flop toggles on every tick of the clock. The second type of \( T \) flip flop is referred to in Wakerly as a '\( T \) flip flop with enable' and has the symbol shown in figure 75(b). This is the \( T \) flip flop referred to in the above table.

**Exercise** Write and simulate the VERILOG code for a \( T \) flip-flop.

### 6.1.11 JK Flip-Flops

Instead of using one control input \( T \) we can use two inputs \( J \) and \( K \), as shown in Figure 76. For this circuit, the input \( D \) is given by \( D = J\overline{Q} + \overline{K}Q \). The corresponding truth table is,
Figure 75: Two types of T flip flops. (a) T flip flop, (b) T flip flop with enable

<table>
<thead>
<tr>
<th>$J$</th>
<th>$K$</th>
<th>$Q(t+1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$Q(t)$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$\overline{Q(t)}$</td>
</tr>
</tbody>
</table>

Figure 76: The JK flip-flop.

The circuit is referred to as a JK flip-flop and combines the operation of $SR$ latches and $T$ flip-flops. It behaves like a **gated SR latch** where $J=S$ and $K=R$ for all input values except $J = K = 1$. In the latter case which had to be avoided for the SR flip-flop, JK flip-flop toggles its state like the T flip-flop. The JK flip-flop is a versatile circuit. It
can be used for straight storage purposes, just like the D and SR latches. But it can also serve as a T flip-flop by connecting the J and K inputs together.

6.1.12 Triggering

Triggering is a key issue; the memory element may be

1. clocked (synchronous) or unclocked (asynchronous)
2. level or edge triggered

We have seen some examples of triggering. Figure 77 illustrates some common types of triggering modes. This applies to flip flops. The triggering mode of a state machine will be inherited from the triggering mode of the memory elements used to construct it.

![Triggering modes diagram]

Figure 77: Triggering modes.

6.2 A Practical Examples of the Use of Flip Flops

6.2.1 A Parallel in Parallel Out Register

The circuit of Figure 78 is a simple data register that sends the input data to the output on the positive edge of the clock and sets the output to zero when the reset is held high.

Since registers of different sizes are often needed in logic circuits, it is advantageous to define a register module for which the number of flip-flops can be easily changed.
Figure 78: Parallel in parallel out three-bit register with asynchronous clear.

The VERILOG code for an n-bit register with asynchronous clear is shown below. The parameter n specifies the number of flip-flops in the register. By changing this parameter, the code can represent a register of any size.

```verilog
module regn (D, Clock, Resetn, Q);
    parameter n = 16;
    input [(n-1):0] D;
    input Clock, Resetn;
    output [(n-1):0] Q;
    reg [(n-1):0] Q;
    always @(negedge Resetn or posedge Clock)
        if (!Resetn)
            Q <= 0;
        else
            Q <= D;
endmodule
```

### 6.2.2 Serial in Serial Out Shift Register

Figures 80 and 81 below show a serial in, serial out shift register. Its purpose is to temporarily store data. At each clock tick, data present at the input to FF1 is read by FF1 and stored. At each subsequent clock tick, the data advances one flip flop, until it is presented at the output of FF3.

Assume the outputs of all the flip flops are initially '0', and that a '1' is presented at the input to FF1. At the next rising edge of the clock R1, the '1' is read by FF1 and, after some time delay, appears at the output Q1.

At R1, a '0' should be read by FF2, since the data should propagate through only one flip flop for each rising edge of the clock. For proper operation, the time delay before the
signal appears at the output Q1 must be neither too long nor too short, otherwise the setup or hold time requirements for FF2 will be violated.

The output must change at some time in the shaded interval shown in the figure. Similar considerations apply for FF3. The output of FF3 will go high some time after R3. A sample sequence is shown in the following table.
Let us write the VERILOG code for the serial in serial out shift register. To make it more interesting let us consider the case of eight stages. The task is greatly simplified if we use hierarchical VERILOG to repeatedly implement eight instantiations of the master slave D flip-flop of Figure 69. The master slave flip-flop module is exactly as before save the fact from Figure 80 that we need a positive edge triggered D flip-flop (hence the `posedge`) as indicated by the following code.

```verilog
data = 4'b1010;  // Example data

module master_slave_D_flipflop (D,Clock,Q);
    input D, Clock;
    output Q;
    reg Q;
    always @(posedge Clock)
        Q <= D;
endmodule
```

<table>
<thead>
<tr>
<th>Data</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>t1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>t2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>t3</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>t4</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>t5</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>t6</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>t7</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 81: Timing diagram for the serial in, serial out shift register.
Notice that the master slave D flip flop module has been reused. Please use such modular coding techniques in your work.

In hierarchical VERILOG, we express the concatenation of D flip-flops by eight different instantiations with the appropriate inputs as follows.

```verilog
module serial_in_serial_out_shift_register (D,CLOCK,Q);
  input D;
  input CLOCK;
  output [7:0] Q;
  wire [7:0] Q;
  master_slave_D_flipflop s0(D, CLOCK,Q[0]);
  master_slave_D_flipflop s1(Q[0],CLOCK,Q[1]);
  master_slave_D_flipflop s2(Q[1],CLOCK,Q[2]);
  master_slave_D_flipflop s3(Q[2],CLOCK,Q[3]);
  master_slave_D_flipflop s4(Q[3],CLOCK,Q[4]);
  master_slave_D_flipflop s5(Q[4],CLOCK,Q[5]);
  master_slave_D_flipflop s6(Q[5],CLOCK,Q[6]);
  master_slave_D_flipflop s7(Q[6],CLOCK,Q[7]);
endmodule
```

**Exercise.** Develop a test bench to exercise the serial in serial out shift register VERILOG code.

### 6.2.3 Parallel-Access Shift Register

In computer systems it is often necessary to transfer n-bit data sequences. This may be done by transmitting all bits at once using n separate wires: that is in parallel. But it is also possible to transfer bits using a single wire, by performing the transfer one bit at a time, in n consecutive clock cycles. We refer to this scheme as serial transfer. To transfer an n-bit data item serially, we can use a shift register that can be loaded with all n bits in parallel (in one clock cycle). Then during the next n clock cycles, the contents of the register can be shifted out for serial transfer. The reverse operation is also needed. If bits are received serially, then after n clock cycles the contents of the register can be accessed in parallel as an n-bit item.

Figure 82 shows a four-bit shift register that allows the parallel access. There are two sources for each flip-flop. One source is the preceding flip-flop, which is needed for the shift register operation. The other source is the external input that corresponds to the bit that is to be loaded. The shift/load signal is used to select the mode of operation. If shift/load = 0, then the circuit operates as a shift register. If shift/load = 1, then the
Figure 82: The parallel access shift register can be used for both serial to parallel and parallel to serial conversion.

Parallel input data are loaded into the register. In both cases the action takes place on the positive edge of the clock.

In Figure 82 we have chosen to label the flip-flops outputs as $Q_3, ..., Q_0$ because shift registers are often used to hold binary numbers. The contents of the register can be accessed in parallel by observing the outputs of all flip-flops. The flip-flops can also be accessed serially, by observing the values of $Q_0$ during consecutive clock cycles while the contents are being shifted. A circuit in which data can be loaded in series and then
accessed in parallel is called a series-to-parallel converter. Similarly, the opposite type of circuit is a parallel-to-series converter. The circuit in Figure 82 can perform both of these functions.

Assume that we wish to write Verilog code that represents the four-bit parallel-access shift register in Figure ???. One approach is to write hierarchical code that uses four subcircuits. Each subcircuit consists of a D flip-flop with a 2-to-1 multiplexer connected to the D input. The following code defines the module named muxdff, which represents this subcircuit.

```verilog
module muxdff (D0, D1, Sel, Clock, Q);
    input D0, D1, Sel, Clock;
    output Q;
    reg Q;
    always @(posedge Clock)
        if (!Sel)
            Q <= D0;
        else
            Q <= D1;
endmodule
```

The two data inputs are named D0 and D1, and they are selected using the Sel input. The if-else statement specifies that on the positive clock edge if Sel = 0, then Q is assigned the value of D0; otherwise, Q is assigned the value of D1. The module below defines the four-bit parallel access shift register. The module Stage3 instantiates the left most flip-flop, which has the output Q3, and the module Stage0 instantiates the right-most flip-flop, Q0. When L = 1, the register is loaded in parallel from the R input; and when L = 0, shifting takes place in the left to right direction. Serial data is shifted into the most-significant bit, Q3, from the w input.

```verilog
module shift4 (R, L, w, Clock, Q);
    input [3:0] R;
    input L, w, Clock;
    output [3:0] Q;
    wire [3:0] Q;
    muxdff Stage3 (w, R[3], L, Clock, Q[3]);
    muxdff Stage2 (Q[3], R[2], L, Clock, Q[2]);
    muxdff Stage1 (Q[2], R[1], L, Clock, Q[1]);
    muxdff Stage0 (Q[1], R[0], L, Clock, Q[0]);
endmodule
```
6.2.4 Counters

In this section we look at a couple of examples of counters. These are devices which produce an output code in response to the counts of a clock. Here we look at an asynchronous up counter and a synchronous counter with enable. A useful application is clock frequency division, where chains of mod-\(p\) counters can be used to produce clock waveforms of reduced frequencies. E.g. a cascade of two decimal (or decade) counters driven by a 1000 Hz signal can produce (among others) waveforms of frequencies 100 Hz and 10 Hz.

The asynchronous up counter produces a binary output that counts the input positive transitions of the clock. Figure 83 shows such a counter based on T flip-flops. T flip-flops with their memorised toggle action are a natural choice for counters. As you can see, the circuit counts the number of positive clock edges. If you look at the timing diagram it is easy to confirm that the outputs \(\{Q_0, Q_1, Q_2\}\) are indeed the binary representation of the count. This circuit is also referred to as a ripple counter. Because it uses three bits it is a modulo eight counter.

The counter in Figure 83 has three stages. Only the first stage responds directly to the Clock signal; we say that this stage is synchronized to the clock. The other two stages respond after an additional delay. As a result, the asynchronous counter is not very fast. If a counter with a larger number of bits is constructed in this manner, then the delays caused by the cascaded clocking scheme may become too long to meet the desired performance requirements. We can build a faster counter by clocking all flip-flops at the same time, using the following approach.

An example of a synchronous four-bit counter (modulo 16) is given in Figure 84. In this case, all flip-flops change their states after a propagation delay from the positive edge of the clock. Note that a change in the value of \(Q_0\) may have to propagate through several AND gates to reach the flip-flops in the higher stages of the counter, which requires a certain amount of time. This time must not exceed the clock period. Actually, it must be less than the clock period minus the setup time for the flip-flops. Notice also that this clock has a built in enable and clear.

Figure 84 gives the timing diagram. It shows that the circuit behaves as a modulo-16 up-counter. Because all changes take place with the same delay after the active edge of the Clock signal, the circuit is called a synchronous counter.

In the next chapter we will learn how to design an up/down counter.

Exercise Write VERILOG code and test benches to simulate the synchronous and asynchronous counters of Figures 83 and 84.

6.3 Timing and Metastability

Before continuing with this chapter, we are going to look at timing issues in some detail. Timing is a very important issue in state machine design. In order for a carefully sequenced
design to work correctly, various timing conditions must be met. See Figure 85.

- $t_{su}$ - set up time: the minimum time period immediately prior to an active clock edge during which the input data must remain stable.

- $t_h$ - hold time: the minimum time period immediately after an active clock edge during which the input data must remain stable.

- $t_w$ - pulse width: width of the clock pulse.

- $t_p$ - propagation delay: time taken for the output to stabilize after each active clock edge.

- The sampling interval is the sum $t_{su} + t_h$, and data should be stable for this time period.

Flip flop data sheets specify the values for these (and other) parameters which the designer must take into account. For instance, set up and hold times must be met when
applying data to a flip flop. If the setup and hold time requirements are not met, then correct operation of the memory element and hence of the state machine cannot be guaranteed. For instance, suppose $t_{su} = 20$ ns, be the data arrives only 10 ns before the active clock edge. Then the device can enter a metastable state, Figure 86. The late arriving data can push $Q$ half-way from LOW to HIGH, i.e. somewhere in the invalid range of voltages. This can cause problems and errors in circuits connected to $Q$. Moreover, $Q$ can go either LOW or HIGH, so the next state is unpredictable. We have already seen how this situation can arise in the discussion of Figure 62. Metastable states must be avoided.

As a practical example consider once again the 74HC373 Octal tristate D latch. Its logic diagram is shown in Figure 87. The purpose of this device is to store the data on an eight bit bus as shown in the figure. It is called an octal latch because it can be used to construct memories with a 3 bit address space. It is a three state latch because in addition to the possibility of a 1 or a 0 at the output of the latch, the latch can also go into a high impedance or disconnect mode. The ”373” consists of eight D-type transparent latches.
with 3-state outputs. When LE is HIGH, data at the \( D_n \) inputs enter the latches. In this condition the latches are transparent, i.e. a latch output will change state each time its corresponding D-input changes. When LE is LOW the latches store the information that was present at the D-inputs a set-up time preceding the HIGH-to-LOW transition of LE. When OE is LOW, the contents of the 8 latches are available at the outputs. When OE
is HIGH, the outputs go to the high impedance OFF-state. Operation of the OE input does not affect the state of the latches. The reason for the high impedance state is so that the latch can be used in **bus oriented** applications.

Figure 87: The 74HC/HCT373 Octal D-type transparent latch logic diagram

<table>
<thead>
<tr>
<th>OPERATING MODES</th>
<th>INPUTS</th>
<th>INTERNAL LATCHES</th>
<th>OUTPUTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>enable and read register</td>
<td>OE: L</td>
<td>D_n: L</td>
<td>Q_0 to Q_7: L</td>
</tr>
<tr>
<td>(transparent mode)</td>
<td>LE: H</td>
<td></td>
<td>L</td>
</tr>
<tr>
<td>latch and read register</td>
<td>LE: L</td>
<td>I</td>
<td>L</td>
</tr>
<tr>
<td>and disable outputs</td>
<td>OE: H</td>
<td>X</td>
<td>Z</td>
</tr>
<tr>
<td></td>
<td>LE: X</td>
<td>X</td>
<td>Z</td>
</tr>
</tbody>
</table>

**Notes**

1. H = HIGH voltage level
   h = HIGH voltage level one set-up time prior to the HIGH-to-LOW LE transition
   L = LOW voltage level
   I = LOW voltage level one set-up time prior to the HIGH-to-LOW LE transition
   X = don’t care
   Z = high impedance OFF-state

Figure 88: 74HC373 function or truth table
### AC CHARACTERISTICS FOR 74HC

GND = 0 V; \( I_{1} = I_{2} = 6 \text{ ns} \); \( C_{L} = 50 \text{ pF} \)

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>PARAMETER</th>
<th>( T_{amb} ) (°C)</th>
<th>UNIT</th>
<th>TEST CONDITIONS</th>
<th>WAVEFORMS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>-25</td>
<td>-40 to -85</td>
<td>-40 to -125</td>
<td>( V_{CC} ) (V)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>min.</td>
<td>typ.</td>
<td>max.</td>
<td>min.</td>
</tr>
<tr>
<td>( t_{PHL} / t_{PLH} )</td>
<td>propagation delay ( D_{n} ) to ( Q_{n} )</td>
<td>41</td>
<td>150</td>
<td>190</td>
<td>225</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15</td>
<td>30</td>
<td>38</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12</td>
<td>26</td>
<td>33</td>
<td>38</td>
</tr>
<tr>
<td>( t_{PHL} / t_{PLH} )</td>
<td>propagation delay ( LE ) to ( Q_{n} )</td>
<td>50</td>
<td>175</td>
<td>220</td>
<td>265</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18</td>
<td>35</td>
<td>44</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14</td>
<td>30</td>
<td>37</td>
<td>45</td>
</tr>
<tr>
<td>( t_{PZH} / t_{PLZ} )</td>
<td>3-state output enable time ( \overline{OE} ) to ( Q_{n} )</td>
<td>44</td>
<td>150</td>
<td>190</td>
<td>225</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>30</td>
<td>38</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13</td>
<td>26</td>
<td>33</td>
<td>38</td>
</tr>
<tr>
<td>( t_{HZ} / t_{LZ} )</td>
<td>3-state output disable time ( \overline{OE} ) to ( Q_{n} )</td>
<td>47</td>
<td>150</td>
<td>190</td>
<td>225</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17</td>
<td>30</td>
<td>38</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14</td>
<td>26</td>
<td>33</td>
<td>38</td>
</tr>
<tr>
<td>( t_{NL} / t_{NL} )</td>
<td>output transition time</td>
<td>14</td>
<td>60</td>
<td>75</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>12</td>
<td>15</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>10</td>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td>( t_{W} )</td>
<td>LE pulse width ( HIGH )</td>
<td>80</td>
<td>17</td>
<td>100</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>6</td>
<td>20</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14</td>
<td>5</td>
<td>17</td>
<td>20</td>
</tr>
<tr>
<td>( t_{SU} )</td>
<td>set-up time ( D_{n} ) to ( LE )</td>
<td>50</td>
<td>14</td>
<td>65</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>5</td>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>4</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>( t_{H} )</td>
<td>hold time ( D_{n} ) to ( LE )</td>
<td>5</td>
<td>–8</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>–3</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>–2</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

**Figure 89:** Timing parameters for the 74HC373.

Its function table to be compared with Figure 87 is shown in Figure 88.

The timing parameters for the HC device are reproduced in Figure 89 and are from page 5 of the datasheet. First at the top there are two propagation delays: one from the data input to the latched data output and the other from the enable (clock) input to the latched data output. The set-up time \( t_{SU} \) is the minimum time that the data \( D_{n} \), must be established before the falling edge (active edge) of the \( LE \) clock (transitioning to the hold state) for the state to be remembered. The hold time \( t_{H} \), is the minimum time that the \( D_{n} \) input must remain stable when the \( LE \) is in the low state for the required state to be remembered. These are the usual definitions.

In addition there are certain time parameters that arise through the addition of the \( \overline{OE} \) line. For example, the 3-state output enable time \( t_{PZH} \) is the time required for the 74HC373 latch to transition from the high Z to a binary output HIGH-state when the \( \overline{OE} \) (output enable) signal goes from HIGH to LOW. To understand this table have a look at the timing diagrams on page 7 of the datasheet. For precise definitions of the
parameters you will also need to look at the Philips HCMOS family characteristics ([?]).

6.4 Summary of Latch and Flip Flop Truth Tables

These tables will be useful for state machine design.

**SR**

<table>
<thead>
<tr>
<th>$S$</th>
<th>$R$</th>
<th>$Q_{t+}$</th>
<th>$Q_t \rightarrow Q_{t+}$</th>
<th>$S$</th>
<th>$R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$Q_t$</td>
<td>hold</td>
<td>0</td>
<td>$\phi$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>reset</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>set</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>invalid</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**D**

<table>
<thead>
<tr>
<th>$D$</th>
<th>$Q_{t+}$</th>
<th>$Q_t \rightarrow Q_{t+}$</th>
<th>$D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>reset</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>set</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 → 0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 → 1</td>
<td>1</td>
</tr>
</tbody>
</table>

**T**

<table>
<thead>
<tr>
<th>$T$</th>
<th>$Q_{t+}$</th>
<th>$Q_t \rightarrow Q_{t+}$</th>
<th>$T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$Q_t$</td>
<td>hold</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>$\overline{Q_t}$</td>
<td>toggle</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 → 0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 → 1</td>
<td>0</td>
</tr>
</tbody>
</table>

**JK**

<table>
<thead>
<tr>
<th>$J$</th>
<th>$K$</th>
<th>$Q_{t+}$</th>
<th>$Q_t \rightarrow Q_{t+}$</th>
<th>$J$</th>
<th>$K$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$Q_t$</td>
<td>hold</td>
<td>0</td>
<td>$\phi$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>reset</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>set</td>
<td>1</td>
<td>$\phi$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$\overline{Q_t}$</td>
<td>toggle</td>
<td>1</td>
<td>$\phi$</td>
</tr>
</tbody>
</table>

124
7 Finite State Machines

7.1 What is a Sequential State Machine? Examples

In the last chapter we introduced various types of latches and flip-flops and briefly touched on their applications in register and counter circuits. We also saw some VERILOG HDL in action. This code could be used to build these memory elements and more complicated circuits in CPLDs. We could take the view that digital system design could proceed using latches and flip-flops and connecting them into more complex structures. However although such an approach might work for a simple processor, such an approach would be very limited because not all digital systems obviously look like combinations of sequential circuits. We need a more general approach. In this chapter we adopt a slightly more formal approach and learn how to design finite state machines based on memory elements.

Sequential machines (or state machines) carry out operations in a sequence of steps.

Synchronous sequential machines are those whose operation is governed by a system clock signal; such machines move from state to state (undergo state transitions) only at times determined by the clock. Sequential machines that are not synchronous are called asynchronous. In these machines the state transitions are governed by external inputs that can occur with a continuous time variable.

When a sequential machine has only a finite number of states, it is called a finite state machine (FSM).

Unlike combinational systems, state machines have memory. The state is remembered. State machines can be effected in digital logic using interconnected memory elements (flip-flops). The systems so constructed can be used to emulate state machines that perform particular tasks. The flip-flops are all connected to a common clock signal that causes them to change state at each active tick of the clock. We have seen this at work in the last chapter.

7.1.1 Examples on the ENGN3213 Web Page

There are some interactive state machine examples on the ENGN3213 web page. You are encouraged to experiment with them to help familiarise yourself with the operation of state machines.

- State Machine Demo 1 - The Alarm Control System
- State Machine Demo 2 - Up/Down/Stop Counter
- State Machine Demo 3 - Vending Machine
### 7.2 Diagrams

State machines can be described with the aid of diagrams:

- *Block diagrams* are used to represent the modular architecture of the machine.
- *State diagrams* describe the relationships between the states in terms of input signals and state transitions, and describe the output signals.
- *Timing diagrams* illustrate the values of signals as time evolves.

### 7.3 Formal Definition of Finite State Machines

We know that combinational logic circuits are described formally using Boolean algebra, a mathematical formalism. State machines are also described in mathematical terms, as follows.

**Finite sets:**

- $X$ - input values
- $Z$ - output values
- $S$ - states

These sets can be *symbolic*, not necessarily binary. E.g.

$$X = \{ \text{up, down, stop} \}$$

These values or symbols reflect the meaning of the state machine specification. For binary implementation we would use a *binary encoding.*

**Functions:**

- $f : S (X) \rightarrow S$ - next state function
- $h : S \rightarrow Z$ - Moore-type output function, or $h : S (X) \rightarrow Z$ - Mealey-type output function

The function $f$ is called the *next state function*, and $h$ is called the *output function.*

State machines may use positive-edge triggered D flip-flops for their state memory, in which case a tick occurs at each rising edge of the clock. Most state machines are designed nowadays using CPLDs with positive edge triggered D flip-flops.
Equations. Moore machine:

\[ s(t + 1) = f(s(t), x(t)) \quad \text{— state equation} \]
\[ z(t) = h(s(t)) \quad \text{— output equation} \] (8)

Equations. Mealey machine:

\[ s(t + 1) = f(s(t), x(t)) \quad \text{— state equation} \]
\[ z(t) = h(s(t), x(t)) \quad \text{— output equation} \] (9)

In these equations \( t \) indicates \textit{time}, and so the equations specify how the value of the state \( s = s(t + 1) \) at time \( t + 1 \) is determined by the state value \( s = s(t) \) at time \( t \) and the input \( x = x(t) \) at time \( t \). The equations are a description of the sequential operation of the machine.

The outputs are specified in either of two ways, corresponding to the names \textit{Moore machine} and \textit{Mealey machine}. The only difference is that a Mealey output may depend explicitly on the current input value \( x(t) \) whereas a Moore output depends only on the current state \( s(t) \).

The FSM black-box model is illustrated in Figure 90.

![Figure 90: Finite state machine (FSM).](image)

Example. For the alarm control system there are two inputs \textit{trip} and \textit{Reset}, two states: \textit{Fired} (s1) and \textit{Armed} (s2) and one outputs \textit{Ring}. The complement of the output is the idle state.

\[ X = \{ \text{Trip, Reset} \} \]
\[ S = \{ s1, s2 \} \]
\[ Z = \{ \text{Ring} \} \]

Example. For the Up/Down/Stop counter, we have

\[ X = \{ \text{up, down, stop} \} \]
\[ S = \{ s1, s2, s3, s4 \} \]
\[ Z = \{ \text{odd, even} \} \]
Example. For the vending machine controller, we have

\[ X = \{ \text{coins}, \text{return}, \text{change} \leftarrow \text{available}, \sum < 75, \sum = 75, \sum > 75 \} \]
\[ S = \{ \text{s0, s1, s2, s3, s4, s5} \} \]
\[ Z = \{ \text{release} \leftarrow \text{candy}, \text{return} \leftarrow \text{all} \leftarrow \text{coins}, \text{return} \leftarrow \text{change} \} \]

7.4 State Diagrams and Tables

State machines are specified by using various types of diagrams and tables. These determine the next state function \( f \) and the output function \( h \).

The creation of tables and diagrams form an important part of the state machine design procedure.

7.4.1 State Diagrams

State diagram form an important part of the analysis of FSMs. They consist of,

- States are symbolised by a circle or a box.
- Lines or branches represent state transitions with the logical condition for the transition written on it.
- For the general case of the Mealey machines the outputs are written underneath the logical conditions on the branches.
- Moore Outputs can be specified inside the states.

As noted in the last chapter, there are basically two types of logic device: latches and flip-flops. Accordingly transitions between the states of an FSM can occur either synchronously with a clock or in response to a change in an input signal asynchronous.

The state diagrams for the alarm controller, the up/down/stop counter and the vending machine are shown in Figures 91 to 93. In this chapter we study the latter two in some detail and we look at some telecom related state machines as more practical examples. Note from the state diagrams that these are all Moore machines. Another thing to note is that not every state necessarily corresponds to a visible output. Thirdly note that they are drawn here to be asynchronous.

Take a careful look at these figures and make sure that you understand what is going on.

Exercise Implement these systems in VERILOG. Take two approaches using a system clock to make a synchronous machine and an external input system to make an asynchronous machine.
Generally in this course we will not consider asynchronous systems as they are more complex to design. For example, FPGAs will always have system clocks. Nonetheless
most practical systems will have an external (asynchronous) reset in order to set them to a fiducial state.

7.4.2 Next State Table

One representation of the FSM is the next state table (also sometimes called next state/output table). In this table the left-most column gives the state of the system. Recall that the state is a collection simply of a given set of all internal and external parameters. The next columns give the state to which a transition must occur for the stated set of input signals. The final column are the outputs.

Let us consider the case of the alarm control circuit. Most circuits are in fact like the alarm circuit in that they do not have a clock. The alarm responds directly to signals from the outside world. Its next state table is drawn below.

<table>
<thead>
<tr>
<th>Present state</th>
<th>next state</th>
<th>next state</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>s0 if trip=0 or reset=1</td>
<td>s1 if trip=1 and reset = 0</td>
<td>0</td>
</tr>
<tr>
<td>s1</td>
<td>s0 if trip=0 and reset=1</td>
<td>s1 if trip=1 and reset = 0</td>
<td>1</td>
</tr>
</tbody>
</table>
Here is the state/output table for the vending machine example.

<table>
<thead>
<tr>
<th>Present state</th>
<th>next state</th>
<th>next state</th>
<th>next state</th>
<th>release candy</th>
<th>return all coins</th>
<th>return change</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>s0 if coin=0 and return=0</td>
<td>s1 if coin=1 and return = 0</td>
<td>s5 if coin = 0 and return = 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s1</td>
<td>s0 if sum&lt;75</td>
<td>s2 if sum=75</td>
<td>s3 if sum&gt;75</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s2</td>
<td>s0</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s3</td>
<td>s4 if change available = 1</td>
<td>s5 if change available = 0</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s4</td>
<td>s2</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>s5</td>
<td>s0</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

### 7.4.3 Timing Diagrams

Timing is central to the function of state machines. Pay particular attention to the sequence of states that is followed in response to the clock signal, as determined by the input values. In general you can generate a timing diagram from a state diagram. What if there is no clock?

Given an initial state and input sequence, determine the resulting sequence of states and outputs.

You should practice doing this with examples.

### 7.5 Implementing Finite State Machines using Flip Flops

We will now look at how to implement finite state machines using flip flops. The first step is to formalise our notation for sequential circuits.

#### 7.5.1 Characteristic Equations

The functional behaviour of a latch or flip flop can be described using characteristic equations. These specify the device’s next state as a function of its current state and input(s). The characteristic equations for the devices we have considered are as follows:

1. **SR latch:** \( Q^* = S + \overline{R}.Q \)
2. **D Flip Flop:** \( Q^* = D \)
3. **T (toggle) Flip Flop:** \( Q^* = \overline{Q} \)
4. T (toggle) Flip Flop with enable: \[ Q^* = EN\overline{Q} + \overline{EN}.Q \]

5. JK Flip Flop: \[ Q^* = J\overline{Q} + \overline{K}.Q \]

The asterisk indicates the next state of the output. See Wakerly page 545.

### 7.5.2 Flip-flops as Finite State Machines

The fundamental concept is that since the output from a flip flop can be either a zero or a one, each flip flop can code for two states, and a collection of \( n \) flip flops can code for up to \( 2^n \) states. For example, the finite state machine of figure 93 has 6 states, so we need at least 3 flip flops to code for these 6 states. If we call the 3 flip flops A, B and C then we could (arbitrarily) assign states as follows:

<table>
<thead>
<tr>
<th>State</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>S2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>S3</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>S4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S5</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

It is important to be clear about what this means. It means that whenever the outputs of the flip flops are, e.g., \( ABC = 010 \), then the machine is considered to be in state S2. The state of the system only changes when at least one of the outputs of the flip flops changes. Note also that we may choose to use more than the minimum number of flip flops to implement the finite state machine, as we will see later.

Consider the clocked synchronous state machine shown in figure 94 below. The system has two T flip flops with enable. Note that it is a Mealy machine since the output depends on the current input as well as the current state. We wish to analyse it in order to understand its behaviour.

The first step is to determine the **excitation equations** for the two flip flops. Excitation equations express the flip flop inputs (excitations) as a function of the current state (i.e., flip flop outputs) and the current input. In our example, these equations are:

\[
T1 = Y \\
T2 = \overline{X}.Y.Q1
\]

Next, we make use of the flip flop **characteristic equations** to derive the **transition equations** which describe the next state of the system as a function of the current state and the inputs. The characteristic equations for the T flip flops with enable are:
Figure 94: A clocked synchronous state machine

$$Q_1^* = T_1 \bar{Q}_1 + \bar{T}_1.Q_1$$
$$Q_2^* = T_2.Q^*_2 + \bar{T}_2.Q_2$$

Substituting the excitation equations into these equations gives:

$$Q_1^* = Y.Q_1 + \bar{Y}.Q_1$$
$$Q_2^* = \bar{X}.Y.Q_1.Q^*_2 + \bar{X}.Y.Q_1.Q_2$$
$$Q_2^* = \bar{X}.Y.Q_1.Q^*_2 + (X + \bar{Y} + \bar{Q}_1).Q_2$$

where the asterisk denotes the next state. The next step is to determine the transition/output table from the transition equations, by considering all the possible combinations of inputs and current outputs. This gives the following:

<table>
<thead>
<tr>
<th>XY</th>
<th>Q1</th>
<th>Q2</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00,1</td>
<td>10,1</td>
</tr>
<tr>
<td>01</td>
<td>01,0</td>
<td>11,0</td>
</tr>
<tr>
<td>10</td>
<td>10,1</td>
<td>01,1</td>
</tr>
<tr>
<td>11</td>
<td>11,0</td>
<td>00,0</td>
</tr>
</tbody>
</table>

We can now assign states to the various output combinations to produce a state/output table:
Finally, we can use the state/output table to draw up a state diagram for the system. This is left as an exercise.

Note that, because this is a Mealy machine, it is necessary to specify the system output on the arrows indicating the transition rather than inside the states. To illustrate this point, figure 95 below shows two different systems, each having two states A and B, one input X and one output Z. The first system is a Moore machine. The second system is a Mealy machine, and its output has to be indicated on the transition arrows.

![State Diagrams](image-url)

Figure 95: Examples of (a) a Moore and (b) a Mealy machine

Most practical systems will contain provisions to allow the system to be reset to a predictable state. Also, in many practical systems, there are more output combinations than
there are states of the system. In other words, some of the possible output combinations will, or should, never occur in practice.

### 7.6 Synthesis of Sequential Machines

We now turn to the design of synchronous state machines using flip flops as memory elements. While software tools automate many aspects of the design process, it is nevertheless important to have an understanding of some of the underlying principles. Further, there are some practical aspects of state machine design to consider.

#### 7.6.1 Canonical Implementation

In Section 7.3 we looked at the general mathematical description of a state machine, of Moore and Mealy types. Figure 96 gives corresponding block diagrams, referred to as the canonical implementation of state machines.

![Canonical state machine implementations](image)

See Wakerly, Section 7.3 for additional information.

#### 7.6.2 Design Steps

1. **Design specification.** Understand the problem specification, and refine it as necessary. Often design tasks are not 100% specified to begin with.
2. Design synthesis.

(a) Draw block diagrams, timing diagrams, flow charts, and modularise as appropriate.
(b) Draw state diagrams, showing all inputs and outputs. Refine the design as appropriate.
(c) Decide on memory elements (how many flip-flops) and triggering (can it be done as a synchronous machine with perhaps a global asynchronous reset).
(d) Draw as a Moore or Mealey machine, so that you can separately examine the input and output combinational circuits. These are the circuits that connect all inputs (X) and feedback wires ($Q_k$) from the inputs to the flip-flops and the circuits connecting the flip-flop outputs to the outputs (Z) of the machine. In the case of the Mealey machine, the input (X) will also be an input to the output combinational circuit that produces Z.
(e) Use K-maps to obtain next state and output combinational logic.
(f) Simulate your design and check for correct operation. Revise as appropriate.


5. Revise. As appropriate.

You should also be aware that there may be more than one output Z, so that it will be simplest to do a separate design for each output. This procedure is also discussed in Wakerly, Section 7.4. Synthesis using VERILOG is very similar, though some steps are automated by the software.

7.6.3 High Level Representations and Binary Coding

We have seen that it is often convenient to use high level descriptions of state machines, where the inputs, outputs and states are symbolic and not necessarily binary. When we come to synthesise or implement a state machine we need to encode any symbols in a binary form. We will see how this can be done in the examples to follow.

7.7 Example - Up/Down/Stop Counter

The up/down/counter state diagram is shown in Figure 92. In this section we will implement it in synchronous form. Here is the state/output table for the up/down/stop counter example:
there are four states, so we need a minimum of two flip flops for the memory, which we label as $AB$. There are three input values, which we code using two bits $SM$, and two output values, coded with one bit labeled $y$.

Let’s use the following binary coding:

<table>
<thead>
<tr>
<th>state</th>
<th>$AB$</th>
<th>input</th>
<th>$SM$</th>
<th>output</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>00</td>
<td>up</td>
<td>00</td>
<td>odd</td>
<td>0</td>
</tr>
<tr>
<td>s2</td>
<td>01</td>
<td>down</td>
<td>01</td>
<td>even</td>
<td>1</td>
</tr>
<tr>
<td>s3</td>
<td>11</td>
<td>stop</td>
<td>1X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s4</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

There are many other possibilities; it is important to know that the choice of coding can have a significant impact on performance and reliability.

With this coding, the state/output table becomes:

<table>
<thead>
<tr>
<th>Present state</th>
<th>next</th>
<th>state</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AB$</td>
<td>$S = 0, M = 0$</td>
<td>$S = 0, M = 1$</td>
<td>$S = 1$</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>10</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>01</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

The general form of the circuit diagram for the state machine is shown in Figure 97. Here we have selected rising edge triggered D flip flops with (asynchronous) clear as the memory elements.

The RESET signal is set up to force the state machine into a specified state, viz. $s1 = 00$.

Our job now is to work out the combinational logic required for the next state and output logic. In order to do this, we need to determine the excitation signal that we need to present at the inputs to the flip flops in order to get the desired transition, summarized in the state/output table. The flip flop application tables come in useful here. They allow...
Figure 97: Up/down/stop counter circuit diagram.

us to derive the excitation table, which lists the required excitations for each combination of current states and inputs.

<table>
<thead>
<tr>
<th>SM</th>
<th>AB 00 01 11 10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00 01 10 00 00</td>
</tr>
<tr>
<td></td>
<td>01 11 00 01 01</td>
</tr>
<tr>
<td></td>
<td>11 10 01 11 11</td>
</tr>
<tr>
<td></td>
<td>10 00 11 10 10</td>
</tr>
<tr>
<td></td>
<td>D_A, D_B</td>
</tr>
</tbody>
</table>

Notice how the last two columns are the same. This is a simple technique for dealing with unused states (more on this later)

For the case of D flip flops, the excitation table is the same as the state/output table, since for a D flip flop, the value of the input has to be equal to the next value we desire at the output. However, this is not the case for other flip flops.

So looking at Figure 98 for the input logic, we have to find the combinational logic that produces D_A and D_B from A,B,S,M. To do this, we first write down the truth table in SOP form and separate D_A and D_B outputs. The following table is just a rewrite of
the above table, but note that we will have to restore the Gray Code ordering to do the
K-map (remember that Gray Coding is necessary in order to loop out logical adjacencies).

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>S</th>
<th>M</th>
<th>D_A</th>
<th>D_B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

We can then draw up k-maps for $D_A$, $D_B$ and the output $y$, Figure 98.

Remember that in each group (including cyclic loop backs) we combine the A,B,S and
M in OR groups for each group of $2^n$ where in this case, $n=0,1,2$. There are two logical
groups of one, one of two and one of four for the case of $D_A$. For example in the case of
$D_A$ we have,

For $n = 0$ : $\overline{A}.\overline{B}.\overline{S}.M + A.\overline{B}.\overline{S}.M = \overline{B}.\overline{S}.M$

For $n = 1$ : $A.B.S.M + A.B.S.M = B.S.M$

For $n = 2$ : $A.B.S.M + A.B.S.M + A.\overline{B}.S.M + A.\overline{B}.S.M = A.S$

The final simplified result is:

$$D_A = \overline{S}.M.\overline{B} + \overline{S}.\overline{M}.B + S.A$$
$$D_B = \overline{S}.\overline{M}.\overline{A} + \overline{S}.M.A + S.B$$
$$y = A \oplus B$$

7.8 Example - Vending Machine Controller

The state diagram and next state/output table for the vending machine controller were
given earlier in Section 7.4.
A common, simple, though not minimal binary coding of states is called *one hot*, where one flip flop is used for each state. One hot coding is suitable for the FPGA architecture, which is rich in flip-flops. (You may have noticed the one hot option in the Xilinx compilation options). In this example we have six states, and with one hot coding we use six flip flops (a minimal binary coding would require three flip flops).

Using state variables $ABCDEF$ the one hot coding is as follows:

<table>
<thead>
<tr>
<th>state</th>
<th>$ABCDEF$</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>000001</td>
</tr>
<tr>
<td>s1</td>
<td>000010</td>
</tr>
<tr>
<td>s2</td>
<td>000100</td>
</tr>
<tr>
<td>s3</td>
<td>001000</td>
</tr>
<tr>
<td>s4</td>
<td>010000</td>
</tr>
<tr>
<td>s5</td>
<td>100000</td>
</tr>
</tbody>
</table>

We use a similar coding for the inputs and outputs, (although there are many valid possibilities) with the following abbreviations:
<table>
<thead>
<tr>
<th>input</th>
<th>abbreviation</th>
<th>output</th>
<th>abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>coin</td>
<td>$CN$</td>
<td>release-candy</td>
<td>$RC$</td>
</tr>
<tr>
<td>return</td>
<td>$RTN$</td>
<td>return-change</td>
<td>$RCH$</td>
</tr>
<tr>
<td>sum&lt;75</td>
<td>$SM0$</td>
<td>return-all-coins</td>
<td>$RAC$</td>
</tr>
<tr>
<td>sum=75</td>
<td>$SM1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sum&gt;75</td>
<td>$SM2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>change-available</td>
<td>$CA$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

So, e.g., “release-candy” is ACTIVE when $RC = 1$, and inactive when $RC = 0$.

With this coding, the state/output table reads:

<table>
<thead>
<tr>
<th>Present state</th>
<th>next state</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ABCDEF$</td>
<td>$000001$</td>
<td>$000001$</td>
</tr>
<tr>
<td></td>
<td>$000010$</td>
<td>$000010$</td>
</tr>
<tr>
<td></td>
<td>$000100$</td>
<td>$000100$</td>
</tr>
<tr>
<td></td>
<td>$010000$</td>
<td>$010000$</td>
</tr>
<tr>
<td></td>
<td>$100000$</td>
<td>$000001$</td>
</tr>
<tr>
<td></td>
<td>$000001$</td>
<td>$000001$</td>
</tr>
<tr>
<td></td>
<td>$000010$</td>
<td>$000010$</td>
</tr>
<tr>
<td></td>
<td>$000100$</td>
<td>$000100$</td>
</tr>
<tr>
<td></td>
<td>$010000$</td>
<td>$010000$</td>
</tr>
<tr>
<td></td>
<td>$100000$</td>
<td>$100000$</td>
</tr>
</tbody>
</table>

Note: We assume that the input signals $SM0, SM1, SM2$ are mutually exclusive, and one of them is active at any time (the vending machine hardware supplying these signals determines this).

<table>
<thead>
<tr>
<th>Present state</th>
<th>$RC$</th>
<th>$RCH$</th>
<th>$RAC$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ABCDEF$</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$000001$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$000010$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$000100$</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$010000$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$010000$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$100000$</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The state diagram can also be re-drawn as in Figure 99. This may help you see what the coding means (do this yourself for the up/down/stop counter).

### 7.8.1 Vending Machine Circuit Schematic

As we can see, each state is associated with exactly one flip flop. The vending machine circuit diagram is shown in Figure 100.
Figure 99: Vending machine controller state diagram - one hot binary coding.

We again use rising edge triggered D flip flops as shown, with asynchronous preset and clear. Activation of the RESET signal will force the state machine into state $s_0 = 000001$. Then it is quite straightforward to obtain the next state logic:

$$
D_A = F.C.N.RTN + C.CA \\
D_B = C.CA \\
D_C = E.SM2 \\
D_D = E.SM1 + B \\
D_E = F.C.N.RTN \\
D_F = F.C.N.RTN + E.SM0 + D + A 
$$

To see this, for $D_A$ we look at the state labeled by $A$ ($s_5$) and for each branch entering this state, we AND the state label ($F$, $C$) for the originating state with the condition for the branch (to get $F.C.N.RTN$ and $C.CA$), and then OR them all together to get finally $D_A = F.C.N.RTN + C.CA$). Keep in mind the application table for the D flip flop.

The output logic is also straightforward to obtain:

$$
RC = D \\
RCH = B \\
RAC = A
$$
7.9 Dealing with unused states

Often, not all of the possible combinations of the outputs from the memory elements code for meaningful states. For example, if a finite state machine has 5 states, then (at least) 3 flip flops are needed and 3 of the 8 output combinations should never occur in practice. Two ways of dealing with these unused states are:

- Minimal cost: output combinations that should never occur are treated as ‘don’t cares’ when determining the next state logic. This results in the simplest implementation.

- Minimal risk: output combinations that should never occur are treated such that,
should they ever occur, the system will be forced back into a known state - such as the initialisation state.

**Example.** The one-hot approach is a minimal cost strategy, since unused states are not considered.

### 7.10 Equivalence and Redundancy

Other issues to consider in state machine design and analysis include the possibility of redundant states and equivalence of state machines.

Two state machines $M_1$ and $M_2$ are *input-output equivalent* if they have the same input-output behaviour, i.e.

If the same input sequence is applied to both machines, then the output sequences are identical. This should hold for all input sequences.

Equivalent machines need not have the same number of states. Indeed, some states may be redundant as far as input-output equivalence is concerned.

**Example.** The four state up/down/stop counter is input-output equivalent to the state machine of Figure 101.

![Input-output equivalent to up/down/stop counter](image)

Figure 101: Input-output equivalent to up/down/stop counter.

### 7.11 Using VERILOG for State Machine Design

In this chapter we have looked at methods for designing sequential state machines "by hand", as well as some practical issues. These methods will help gives us at least some appreciation of what the VERILOG compilers do in taking a VERILOG source file and producing an implementation, and certainly we now have the ability to construct state machines using discrete components.

The general approach is to code the design using the programming style used in Wakerly and by the Xilinx software, as discussed earlier. This style is closely related to the next state/output tables, and is used in many examples. The Xilinx state machine editor is very useful for this approach. However, you should know how to do it by hand.
7.12 Some Applications of State Machines in Telecommunications

In this section we look at some telecom related state machines. It should soon become clear that state machines and their implementations in FPGA are basic technologies in modern telecommunications. In this section, we confine our attention to,

- An example of channel coding. Channel coding refers to the use of digital signal processing to alleviate the deleterious effects of physical channels on the information being sent over a link. We look at the channel coding technology which first made deep space communications a reality: convolutional coding. In this chapter we study the convolutional encoder which is the subsystem that resides in the transmitter. In a class project we will also look at the optimum convolutional decoder. The decoder resides at the receiver and is responsible for error free recovery the information that transmitted across the link. As you might expect, decoding is tougher than encoding and should therefore lead to a more complex state machine.

- A couple of Network protocols. Network protocols are standard methods of signalling and communication understood by a wide class of devices. Many network protocols have already been implemented as state machines in FPGAs. We look at two well known examples: Ethernet and WiFi.

Not covered here are the ever increasing number of applications that used to be in the domain of analogue electronics but are now better solved using digital techniques. Two examples are frequency division multiplexing and modulation. More and more we find that the digital “back end” of telecommunications systems is encroaching into the analogue front end. In some applications, little more than the radio antenna will survive the march of digital technology. Perhaps the best known new technology in wireless telecommunications is Software Defined Radio. As an example consider cell phones. Currently cell phones operate on only one type of network such as GSM, CDMA or UMTS. In the not-too-distant future, cell phones will be software radios capable of functioning on more than one network with little more than a software reconfiguration.

7.12.1 The Convolutional Encoder

In this section, we introduce one example of an encoder for the communication of information bits over a communications channel. In telecommunications, a bit stream generated from a source of information, is transmitted over a channel. The channel could be a wire such as a telephone line or a wireless channel after being radiated from an antenna. The bit stream propagates from the transmitter to the receiver. Unfortunately these signals are corrupted by noise from the environment. This means that the bitstream received by the receiver is a corrupted version of that at the transmitter. The degree of corruption can
quantified by a quantity known as the **Bit Error Rate (BER)** for uncoded reception. The BER is simply defined as the ratio of the number of bits in error to the total number of bits. Uncoded BER is simply the BER the receiver generates by making **hard decisions** about the bits without doing any special processing. The maximum average BER for the worst possible case of corruption is 50% (Why?). The aim of the **convolutional decoder** is to remove these errors.

In telecomms it is clear that the bitstream produced by a typical information source is pretty much random. This means that the sequence of bits has no memory: one cannot tell anything about the recent history of received bits from the present received bits. On the other hand if we could build a special device which imparted memory to the bit stream, then it might be possible to devise a **forward error correction** device at the receiver which could determine the correct sequence of bits from the corrupted received sequence. This special device is a decoder capable of recovering the transmitted bit sequence **despite** the errors.

In this chapter we have already seen that sequential circuits such as flip-flops and latches have memory. This also means that a random sequence of bits clocked into these devices produces an output stream which has memory. It is not too surprising that a convolutional encoder turns out to be a pretty simple looking state machine. The state transition diagram for such an encoder is shown in Figure 102. There are four states $S_0$, $S_1$, $S_2$, $S_3$. These have been labelled (00), (01), (10), (11) respectively. Joining these are the allowable state transitions labelled with the input bit on top and the output state just before the clock transition that moves the encoder into the next state. These devices are Mealey machines and therefore we have had to adopt the more complex labelling convention.

### 7.12.2 Analysis of the Convolutional Encoder

The convolutional encoder has four states and can therefore be built with two flip-flops. Just like Figure 94, the Convolutional encoder is a Mealy machine (Why?). Therefore we can following the previous example. The next state diagram is shown in the following transition/output table.

<table>
<thead>
<tr>
<th>State</th>
<th>Input X</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00, (00)</td>
<td>10, (11)</td>
</tr>
<tr>
<td>01</td>
<td>00, (11)</td>
<td>10, (00)</td>
</tr>
<tr>
<td>10</td>
<td>01, (10)</td>
<td>11, (01)</td>
</tr>
<tr>
<td>11</td>
<td>01, (01)</td>
<td>11, (10)</td>
</tr>
<tr>
<td>State*, $(Z_1, Z_2)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We can now assign states to the various output combinations to produce a state/output table:

<table>
<thead>
<tr>
<th>State</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00, (00)</td>
</tr>
<tr>
<td>01</td>
<td>00, (11)</td>
</tr>
<tr>
<td>10</td>
<td>01, (10)</td>
</tr>
<tr>
<td>11</td>
<td>01, (01)</td>
</tr>
</tbody>
</table>

We can now assign states to the various output combinations to produce a state/output table:
Figure 102: State diagram of the convolutional encoder. The states are in the bubbles with 2-bit binary labels. The transitions are labelled with the current bit and output state.

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>S₀</td>
<td>S₀,(00) S₂,(11)</td>
</tr>
<tr>
<td>S₁</td>
<td>S₀,(11) S₂,(00)</td>
</tr>
<tr>
<td>S₂</td>
<td>S₁,(10) S₃,(01)</td>
</tr>
<tr>
<td>S₃</td>
<td>S₁,(01) S₃,(10)</td>
</tr>
</tbody>
</table>

State*, (Z₁, Z₂)
7.12.3 Synthesis of the Convolutional encoder

The state diagram of the convolutional encoder is shown in Figure 102. Our job now is to work out the combinational logic required for the input, next state and output logic. Since there are four states in the convolutional encoder we know that there must be two flip-flops. We also know that we are dealing with a Mealy machine because the output states are directly dependent on the input. This is evident from the state diagram because the outputs (there are two) appear under the inputs in the state transitions. In a Moore machine, there would only be one pair of outputs for each state. In the convolutional encoder, the output depends on how we transition into the state. For example, in the $S_0 = (0, 0)$ state, the output could either be a $(0, 0)$ if the input is a 0 or a $(1, 1)$ if the input is a 1.

As a result one can draw the Mealy machine of Figure 103.

Figure 103: Block diagram of the convolutional encoder generalised circuit

Let us synthesise the input circuit. The input circuit is to the left of the broken line in the figure. It has three inputs, $X, Q_1$ and $Q_2$ and two outputs (to each flip-flop) $D_1$ and $D_2$. An important question is: What are $D_1$ and $D_2$? Since the flip-flops are assumed to be D flip-flops then we can presume that $D_1$ and $D_2$ are the next state outputs of the flip-flop. This is so because on the active edge of the clock (not shown, but obviously

148
connected into the flip-flops), \( D_1 \) and \( D_2 \) are transferred respectively to \( Q_1 \) and \( Q_2 \) and \( Q_1 \) and \( Q_2 \) take on new values. We can therefore write the following table of outputs and inputs for the input combinational circuit.

<table>
<thead>
<tr>
<th>Input X</th>
<th>( Q_1, Q_2 )</th>
<th>( 0 )</th>
<th>( 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( Q_1' ), ( Q_2' )</td>
<td>00</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>( Q_1', Q_2' )</td>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>( Q_1', Q_2' )</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>( Q_1', Q_2' )</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

From this we can group the three inputs into standard canonical form as follows.

\[
\begin{array}{cccc}
X & Q_1 & Q_2 & D_1 \ D_2 \\
0 & 0 & 0 & 0 \ 0 \\
0 & 0 & 1 & 0 \ 0 \\
0 & 1 & 0 & 0 \ 1 \\
0 & 1 & 1 & 0 \ 1 \\
1 & 0 & 0 & 1 \ 0 \\
1 & 0 & 1 & 1 \ 0 \\
1 & 1 & 0 & 1 \ 1 \\
1 & 1 & 1 & 1 \ 1 \\
\end{array}
\]

From this we readily produce the input block K-maps shown in Figure 104: one for each of \( D_1 \) and \( D_2 \). Notice how the Gray code has been observed.

The input combinational circuit can be readily determined.

\[
D_1 = X \\
D_2 = \overline{X}.Q_1\overline{Q}_2 + X.Q_1.Q_2 + \overline{X}.Q_1.Q_2 + X.Q_1.\overline{Q}_2 = \overline{Q}_1
\]

Thus the input circuit can be redrawn as shown in Figure 105.

Next we turn our attention to the output circuit shown in the right of the right most vertical broken line. This combinational circuit has inputs \( Q_1, Q_2 \) and \( X \) and outputs \( Z_1 \) and \( Z_2 \). Once again using the state diagram of Figure 102, we obtain the following input/output table.
Figure 104: The convolutional encoder input block K-map

Figure 105: Block diagram of the convolutional encoder input block
From this we can group the three inputs into standard canonical form as follows.

\[
\begin{array}{c|cc}
\text{X} & Q_1 & Q_2 \\
\hline
00 & 00 & 11 \\
01 & 11 & 00 \\
10 & 10 & 01 \\
11 & 01 & 10 \\
\hline
Z_1, Z_2
\end{array}
\]

From this we readily produce the input block K-maps shown in Figure 106: one for each of \( Z_1 \) and \( Z_2 \). Again be careful that the Gray coding is used.

The input combinational circuit can be readily determined. For \( Z_1 \) there are no logical adjacencies except for \( n = 0, \ (2^n = 1) \).

\[
\begin{align*}
Z_1 &= X \overline{Q_1} \overline{Q_2} + X \overline{Q_1} Q_2 + X Q_1 Q_2 + X \overline{Q_1} Q_2 = X \oplus Q_1 \oplus Q_2 \\
Z_2 &= X Q_1 \overline{Q_2} + X \overline{Q_1} Q_2 + X \overline{Q_1} Q_2 + X Q_1 Q_2 = X \overline{Q_2} + \overline{X} Q_2 = X \oplus Q_2
\end{align*}
\]

This leads to the output circuit as shown in Figure 107.

Finally Figure 108 shows the circuit implementation of the convolutional encoder.

### 7.12.4 Some Convolutional Encoder Terminology

The above device is referred to as a \( K = 3, (7, 5) \) convolutional code. The octal numbers 7 and 5 represent the code generator polynomials, which when read in binary (111\(_2\) and 101\(_2\)) correspond to the shift register connections to the upper and lower XOR gates, respectively. Have a look at the figure to see how this occurs. Note that at the output two bits have to be multiplexed at twice the input data rate.
This code has been determined to be the "best" code for rate 1/2, K = 3. At the output a toggle switch alternately selects bits from each half of the encoder at twice the rate of the incoming bit stream. It is because there are two bits produced for every input bit that this is called a rate 1/2 code. Notice that the price we pay for having error correcting capability is a loss of a factor of two in bandwidth. This is entirely to be expected of forward error correction. It is called a K=3 encoder because each input bit has an effect on three successive output symbols.

### 7.12.5 Convolutional Encoder Timing Diagram

Fig 109 shows a timing diagram. Make sure that you understand what is going on.

In this course we will be looking at how to decode the output bit stream (which is that sent over the communications channel) and their will be anoptional project linked to this. We will introduce the process of maximum likelihood sequence estimation (MLSE). We will look at how to implement the decoder based on the Viterbi Algorithm which is a well known highly efficient technique for MLSE decoding. The Viterbi Algorithm is a very important FEC scheme that has been used radio receivers for wireless communications since the earliest deep space missions. It is currently used in UMTS mobile phone and Digital Video Broadcast (DVB) receivers.
7.12.6 The Ethernet (IEEE 802.3) MAC State Machine

Figure 110 shows a simplified state description of the ethernet (IEEE 802.3) transmitter. As you may know, ethernet is the dominant Local Area Network (LAN) wireline standard.
This state machine is termed the Media Access Controller (MAC). You may have heard of the term “MAC address” in your PC travels. This refers to the unique hardware address of your computer’s ethernet network interface card (NIC).

The MAC’s job is to coordinate channel access on the LAN ethernet communications. The channel here refers to the ethernet cable. As can be seen from the figure the MAC goes about its job by sensing the channel for a carrier transmitted from another station. If the channel is free, the transmitter transmits a packet. If carrier is sensed from another station during transmission, then transmission is halted and the controller goes into a wait state. This is termed a collision. After waiting a certain time, the transmitter continues to sense the channel and if the channel becomes clear, attempts retransmission. The controller counts the number of unsuccessful transmission attempts and if these exceed 16, the packet is dropped. This strategy for avoiding collisions is called Collision sensing multiple access with collision detection (CSMA/CD).

In addition the controller also deals with packet reception. The “network” referred to in the figure is the network stack above the ethernet layer. In practice, ethernet is a bit more complicated than this. The standard also includes detailed specifications on how to build devices to allow communication over wires. This is the so called PHY layer specification. All the details are in the document 802.32002-2002.pdf which can be downloaded from the IEEE Standards Association. The most interesting chapters to read are 3, 4 and 14.

There is an ethernet project that students may choose to do. In this project a portion of the IEEE 802.3 specification will be implemented. In this regard it should be mentioned that ethernet over 10base-T cable, may be full duplex. This means that the MACs can transmit and receive at the same time such that collisions are no longer an issue.
7.12.7 The WiFi (IEEE 802.11b) MAC State Machine

Figure 111 shows the WiFi (802.11b) Distributed Interframe Space wireless MAC. Notice that it is similar to ethernet except that it has a much more complex form of collision detection called Collision sensing multiple access with collision avoidance (CSMA/CA). In this protocol short clear to send (CTS) and request to send (RTS), packets are sent out before transmitting the much longer data packets proper. This is to minimise disturbance in the event of a collision. It also allows stations to communicate with a base station or access point even if the client stations cannot hear each other. This latter is referred to as the hidden transmitter problem. Full duplex communications is not possible in 802.11b.
7.13 Practical Aspects of Sequential Machine Design

There are many practical issues which must be taken into account when designing state machines. These include:

- **Asynchronous inputs.** These are inputs which can arrive at any time, and are not synchronised to the clock signal.
  - metastability, improper detection of signal
  - pulse catching, stretching and synchronisation
  - state selection
- switch debouncing

- **Output glitches.** Due mainly to
  - hazards in output logic
  - race conditions

May not be a problem, but if so, then
  - eliminate, or
  - filter out

- **State code assignment.** Careful selection of states and state code can minimise circuitry and reduce problems associated with glitches and asynchronous inputs.
  - States having the same next state for a given input condition should have logically adjacent state code assignments.
  - States that are next states of a single state should have logically adjacent state code assignments.

- **Reset and initialization.** On power up, you cannot predict the state a flip flop will go into. A circuit, sometimes called a *sanity circuit*, is used to force the flip flops into a known or specified state. This is often done using PRESET or PRECLEAR (or other) features of the flip flops.

- **Clock skew.** Differences in the arrival time of the clock signal at different devices can cause circuit malfunction.

We will look further into some of these points in the following sections.

### 7.13.1 Asynchronous Inputs

Asynchronous inputs can be difficult to handle since data can arrive at unpredictable times, and may not meet the setup and hold time requirements. Various techniques have been developed to deal with this.

One approach is to carefully craft the state diagram. Branching dependencies on more than one asynchronous input should be avoided. Logical adjacency principles could be used when possible. The go-no go configuration of Figure 112 is useful.

Here, \(X\) is the asynchronous input, and \(a\) and \(b\) are logically adjacent states. The machine will stay in state \(a\) while \(X\) is inactive, and move to state \(b\) when \(X\) is activated. This is a commonly used and reliable scheme.

It may even be useful to add in extra states, as the enhanced reliability would more than compensate for the cost of the additional states.
In some situations the asynchronous input might be of the form of a short pulse, and circuitry is required to catch the pulse, stretch it as needed, and synchronise it with the clock signal before passing it on to the state machine, Figure 113.

Figure 113: Pulse catching, stretching, and synchronisation.

7.13.2 Clock Skew

If more than one device is connected to the same clock, the clock signal will not arrive at exactly the same time at each device due to propagation delays. The difference between...
arrival times of the clock signal at different devices is called clock skew. Excessive clock skew can result in incorrect circuit operation, as shown in figure 114. In order to minimize clock skew, differences in clock signal path length to different devices should be minimized.

![Figure 114: Illustration of clock skew.](image)

7.13.3 Switch Debouncing

When a switch is thrown, there is often not a sharp transition from ON to OFF (or vice versa), and instead there may be oscillations called switch bounce, Figure 115. Bouncing
may last for something like 10 ms, which is a long time for a digital system. These oscillations may cause problems unless handled correctly.

![Figure 115: Switch bounce.](image)

A *debouncing circuit* to filter out the switch bounce is shown in Figure 116.

![Figure 116: Switch debouncing circuit.](image)

The Schmitt trigger inverter has an input-output characteristic as shown in figure 117. As a result of the hysteresis in the characteristic, it is possible to obtain a clean output signal when the input signal is noisy. In the circuit shown above, $R_1$, $R_2$ and $C$ effectively integrate the pushbutton signal, leading to input and output signals at the Schmitt trigger inverter that look something like shown in figure 118. Typically, $R_1 \gg R_2$, and $R_1 C \approx 10$ ms.
7.13.4 Initialisation and Reset

As we have already seen, memory elements like flip flops often have additional features such as PRESET or CLEAR which can override their normal operation and force the element into a set or cleared state.

_Sanity circuits_ may make use of these features to force a state machine into a specified initial state on power up. Figure 119 shows one such sanity circuit.

The SANITY signal is active for a time period until the capacitor voltage reaches a certain level, and then becomes inactive. While SANITY is active, the state machine is reset to the desired initial state.

**Some practical issues.** Output _race glitches_ may occur in the output signals, e.g. _HALT_. Consider the transition

\[ AB = 11 \rightarrow AB = 00 \]

which involves two flip flops changing state “simultaneously”, a _race condition_. See Figure 120.

Since we can’t predict which path the machine will take, we may or may not get a glitch as the figure shows.

**Exercise.** Examine other outputs for possible glitches (either due to races, or hazards in the output logic).

**Question?** Will glitches cause problems?

This depends on what the signals are used for.
There should be no problems with glitches in the next state logic, since the glitch will be over well before the next active clock edge.

If an outputs signal drives LEDs, e.g., no problems should arise (too fast to see).

However, if the output signal is an input to further logic, then glitches may pose a potential problem if unwanted actions are initiated.

How to avoid problems with glitches (if necessary).

- **State code assignment.** Do this in a way which minimises or avoids races by ensuring at most one flip flop changes (if possible).

  A better choice of code in this example would be the Gray code:
**Figure 119:** Sanity circuit for power-up initialisation.

**Figure 120:** K-map for *HALT* illustrating race condition.

<table>
<thead>
<tr>
<th>State</th>
<th>old code</th>
<th>new (Gray) code</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>s1</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>s2</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>s3</td>
<td>11</td>
<td>163</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>
Use of this new coding will result in a design free of races, and race glitches. Sometimes it might be necessary to add additional states called fly states to achieve this, Figure 121.

Figure 121: Fly states.

- Remove any static hazards in the output logic.
- Filter out the glitch. Figure 122 shows a circuit sometimes used to filter out glitches; here a finite state machine (negative edge triggered) has an output \( HALT \) with glitches, while \( HALT' \) will be glitch-free, but delayed by half a clock cycle.

Figure 122: Filtering out race glitch.

- Be careful with asynchronous inputs appearing in conditional outputs (these can introduce glitches).
Appendix: VERILOG Syntax

The following compilation of VERILOG Syntax is a subset of that provide by [5].

A.1 Lexical Tokens

Verilog source text files consist of the following lexical tokens:

A.1.1 White Space

Separates words and can contain spaces, tabs, new-lines and form feeds. Thus a statement can extend over multiple lines without special continuation characters.

A.1.2 Comments

Can be specified in two ways (exactly the same way as in C/C++):

1. Single line comments begin with double slashes (/\). All following text to the end of the line will be ignored by the VERILOG compiler.

2. To write comments extending more than one line, enclose comments between the characters /* and */. This is good for "commenting out" many lines of code, or for in-line comments.

A.1.3 Verilog Keywords

These are words that have special meaning in VERILOG. Some examples are assign, case, while, wire, reg, and, nand, and module. They should not be used as identifiers.

A.2 Data Types

VERILOG consists of only four basic values. Almost all Verilog data types store all these values:

0 logic zero, or false condition
1 logic one, or true condition
x unknown logic value
z high impedance state
The values x and z have limited use for synthesis and you should be wary if your simulation unexpectedly produces these values at an output.

In this respect VERILOG differs significantly from procedural computer languages which always have some kind of non-trivial number system such as integers and floating point numbers. Having said this, VERILOG does have integers and reals: but you may consider these as only useful for simulation purposes.

A.2.1 Wire

The `wire` keyword represents a physical wire in a circuit and is used to connect gates or modules. The value of a wire can be read or assigned inside a `module` but never inside procedural code such as `initial` and `always` blocks. A wire does not store its value but must be driven by a continuous assignment statement `assign` or by connecting it to the output of a gate or `module`. A wire cannot be an integer. It must be a bit or a bus.

Other specific types of wires include:

- **wand** (wired-AND): the value of a wand depend on logical AND of all the drivers connected to it.
- **wor** (wired-OR): the value of a wor depend on logical OR of all the drivers connected to it.
- **tri** (three-state): all drivers connected to a tri must be z, except one (which determines the value of the tri).

Syntax.

```verilog
wire [msb:lsb] wire_variable_list;
wend [msb:lsb] wand_variable_list;
wor [msb:lsb] wor_variable_list;
tri [msb:lsb] tri_variable_list;
```

Examples.

```verilog
wire c // simple wire
assign c = a || b; // a or b
wire [9:0] A; // a cable (vector) of 10 wires.
```
A.2.2 Reg

A reg (register) is a data object that holds its value from one procedural assignment to the next. They are used only in functions and procedural blocks such as always and initial. A reg is a VERILOG variable type and does not necessarily imply a physical register. In multi-bit registers, data is stored as unsigned numbers and no sign extension is done for what the user might have thought were two’s complement numbers. To do two’s complement arithmetic you need to declare the regs as signed (see below).

Syntax

\[ \text{reg [msb:lsb] reg\_variable\_list;} \]

Example.

\begin{align*}
\text{reg a;} & \quad // \text{single 1-bit register variable} \\
\text{reg [7:0] tom;} & \quad // \text{an 8-bit vector; a bank of 8 registers.} \\
\text{reg [5:0] b, c;} & \quad // \text{two 6-bit variables}
\end{align*}

A.2.3 Input, Output, Inout

These keywords declare input, output and bidirectional ports of a module or task. Input and inout ports are of type wire. An output port can be configured to be of type wire, reg, wand, wor or tri. The default is wire.

Examples.

\begin{align*}
\text{module sample(b, e, c, a);} & \quad // \text{See "Module Instantiations" on p. 10} \\
\text{input a;} & \quad // \text{An input which defaults to wire.} \\
\text{output b, e;} & \quad // \text{Two outputs which default to wire} \\
\text{output [1:0] c;} & \quad /* \text{A two-it output. One must declare its} \\
& \quad \quad \quad \quad \quad \text{type in a separate statement. */} \\
\text{reg [1:0] c;} & \quad // \text{The above c port is declared as}
\end{align*}

Here are a couple of simplified rules for matching the structural data type to the type of a port.

1. Use regs as the outputs of behavioural blocks. If you use a wire then the value will never be seen by the other blocks.

2. Use wire for all inputs, inouts and most outputs of structural elements (i.e. modules representing combinational logic).
A.2.4 Integers

Integers are general-purpose variables. For synthesis they are used mainly as loop indicies, parameters, and constants. They are implicitly of type reg. However they store data as signed numbers whereas explicitly declared reg types store them as unsigned. If they hold numbers which are not defined at compile time, their size will default to 32-bits. If they hold constants, the synthesizer adjusts them to the minimum width needed at compilation.

There is also a real data type. Both integers and reals elaborate into a large number of gates and there is no reason to use them at all in VERILOG.

A.2.5 Supply0, Supply1

Supply0 and supply1 define wires tied to logic 0 (ground) and logic 1 (power), respectively.

A.2.6 Time

Time is a 64-bit quantity that can be used in conjunction with the $time system task to hold simulation time. Time is not supported for synthesis and hence is used only for simulation purposes. For example, you may wish to know exactly when (at least theoretically in terms of the system clock cycles if not in nanoseconds) when the FFT of an input has been calculated.

Syntax.

time time_variable_list;

Example.

time c;
c = $time;    //c = current simulation time
%

A.2.7 Parameter

A parameter defines a constant that can be set when you instantiate a module. This allows customization of a module during instantiation.

Syntax.

parameter par_1 = value,
par_2 = value, ......;
parameter [range] parm_3 = value
Example.

parameter n = 4;
...
reg [n-1:0] harry; /* A 4-bit register whose length is set by parameter n above. */

A.3 Number Specification

There are two types of number specification in VERILOG: sized and unsized. Sized numbers are represented as `<size> <base format> <number>`. `<size>` is written only in decimal and specifies the number of bits in the number. Legal base formats are decimal (`'d` or `'D`), hexadecimal (`'h` or `'H`), binary (`'b` or `'B`) and octal (`'o` or `'O`). The number is specified as consecutive digits from 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f. Only a subset of these digits is legal for a particular base. Uppercase letters are legal for number specification.

The default size of a constant is one bit wide, but as expected, we can have constants that span multiple bits. Some common examples are,

```
8'b00000101; // represents an 8 bit value.
8'b101; // leading zeros are omitted in this representation.
8'h2C; // hexadecimal number that is 8 bits wide.
8'b0001001110; // equivalent to the above hex number.
4'bz; // A four bit wide high impedance (tristate)
4'b1111 // This is a 4-bit binary number
12'habc // This is a 12-bit hexadecimal number
16'd255 // This is a 16-bit decimal number.
```

Numbers specified without a `<base format>` specification are decimal numbers by default. Numbers that are written without a `<size>` specification have a default number of bits that is simulator and machine specific (must be at least 32).

```
23456 // This is a 32-bit decimal number by default
'hc3 // This is a 32-bit hexadecimal number
'o21 // This is a 32-bit octal number
```

VERILOG has two symbols for unknown and high impedance values. These values are very important for modeling real circuits. An unknown value is denoted by an x. A high impedance value is denoted by z.
12'h13x // This is a 12-bit hex number; 4 least significant bits unknown
6'hx // This is a 6-bit hex number
32'bz // This is a 32-bit high impedance number

An x or z sets four bits for a number in the hexadecimal base, three bits for a number in the octal base, and one bit for a number in the binary base. If the most significant bit of a number is 0, x, or z, the number is automatically extended to fill the most significant bits, respectively, with 0, x, or z. This makes it easy to assign x or z to whole vector. If the most significant digit is 1, then it is also zero extended.

A.3.1 Negative numbers

can be specified by putting a minus sign before the size for a constant number. Size constants are always positive. It is illegal to have a minus sign between < base format > and < number >. An optional signed specifier can be added for signed arithmetic.

-6’d3 // 8-bit negative number stored as 2’s complement of 3
-6’sd3 // Used for performing signed integer arithmetic
4’d-2 // Illegal specification

Wire and reg variables are positive numbers. Thus (-3'b001) = 3'b111 and (-3d001)>3d110. On the other hand, for integers -1 < 6.

The sign of a wire or reg can be altered by means of the signed keyword. In bit format the sign is represented as two’s complement.

reg [7:0] X; //Unsigned reg X
reg signed [7:0] Xs; //Signed reg X

To see how this works consider the following program. Try varying the values of X and Xs to see the effect of signed.

// Verilog program to test signed registers
module RegSigned;
reg clk;
reg [7:0] X;
reg signed [7:0] Xs;

initial begin
clk=0;
X = -127;
Xs = -127;
A.4 Operators

Operators are of three types: unary, binary, and ternary. Unary operators precede the operand. Binary operators appear between two operands. Ternary operators have two separate operators that separate three operands.

\[ a = \sim b; \] \hspace{1cm} // \sim \text{ is a unary operator. } b \text{ is the operand}  
\[ a = b \&\& c; \] \hspace{1cm} // \&\& \text{ is a binary operator. } b \text{ and } c \text{ are operands}  
\[ a = b ? c : d; \] \hspace{1cm} // ?: \text{ is a ternary operator. } b, c \text{ and } d \text{ are operands}  

A.4.1 Arithmetic Operators

The + and - can be used as either unary (+1, -2) or binary (x+y=2) operators.

\[ + \] \hspace{1cm} (addition)  
\[ - \] \hspace{1cm} (subtraction)  
\[ * \] \hspace{1cm} (multiplication)  
\[ / \] \hspace{1cm} (division)  
\[ \% \] \hspace{1cm} (modulus)  

A.4.2 Relational Operators

compare two operands and return a single bit 1 or 0. These operators synthesise into comparators.

\[ < \] \hspace{1cm} (less than)  
\[ <= \] \hspace{1cm} (less than or equal to)  
\[ > \] \hspace{1cm} (greater than)  
\[ >= \] \hspace{1cm} (greater than or equal to)  
\[ == \] \hspace{1cm} (equal to)  
\[ != \] \hspace{1cm} (not equal to)
Example.

```plaintext
if (x == y) e = 1;
else e = 0;
```

// Compare in 2’s compliment; a>b
```plaintext
reg [3:0] a,b;
if (a[3]== b[3]) a[2:0] > b[2:0];
else b[3];
```

### A.4.3 Bit-wise Operators

do a bit-by-bit comparison between two operands. However see "Reduction Operators" below.

```
~   (bitwise NOT)
&   (bitwise AND)
|   (bitwise OR)
^   (bitwise XOR)
^^ or ^~(bitwise XNOR)
```

Example.

```plaintext
module and2 (a, b, c);
  input [1:0] a, b;
  output [1:0] c;
  assign c = a & b;
endmodule
```

### A.4.4 Logical Operators

return a single bit 1 or 0. They are the same as bit-wise operators only for single bit operands. They can work on expressions, integers or groups of bits, and treat all values that are nonzero as "1". Logical operators are typically used in conditional (if ... else) statements since they work with expressions.

```
! (logical NOT)
&& (logical AND)
|| (logical OR)
```

Example.
wire[7:0] x, y, z;  // x, y and z are multibit variables.
reg a;

... 

if ((x == y) && (z)) a = 1;  // a = 1 if x equals y, and z is nonzero.
else a = !x;  // a =0 if x is anything but

A.4.5 Reduction Operators

operate on all the bits of an operand vector and return a single-bit value. These are the unary (one argument) form of the bit-wise operators above.

&  (reduction AND)
|  (reduction OR)
~&  (reduction NAND)
~|  (reduction NOR)
^  (reduction XOR)
~^ or ^~  (reduction XNOR)

A.4.6 Shift Operators

shift the first operand by the number of bits specified by the second operand. Vacated positions are filled with zeros for both left and right shifts (There is no sign extension).

<<  (shift left)
>>  (shift right)

A.4.7 The Concatenation Operator

combines two or more operands to form a larger vector.

{ }(concatenation)

Example.

wire [1:0] a, b; wire [2:0] x; wire [3:0] y, Z;

assign x = {1'b0, a};  // x[2]=0, x[1]=a[1], x[0]=a[0]
assign y = {a, b};  /* y[3]=a[1], y[2]=a[0], y[1]=b[1],
                      y[0]=b[0] */
assign {cout, y} = x + Z;  // Concatenation of a result
A.4.8 The Replication Operator

makes multiple copies of an item.

\{n\{item\}\} (n fold replication of an item)

Example.

wire [1:0] a, b; wire [4:0] x;
assign x = {2\{1'b0\}, a}; //Equivalent to x = {0,0,a }
assign y = {2\{a\}, 3\{b\}}; //Equivalent to y =

A.4.9 The Conditional Operator: ")?"?

is as in C/C++. It evaluates one of the two expressions based on a condition. It will synthesize to a multiplexer (MUX).

(cond) ? (result if cond true):
(result if cond false)

Example.

assign a = (g) ? x : y;
assign a = (inc = = 2) ? a+1 : a-1; /* if (inc), a = a+1, else a = a-1 */

Operator precedence should not be relied upon. Instead control precedence by the use of parentheses

A.5 Operands

A.5.1 Literals

Literals are constant-valued operands that can be used in Verilog expressions. The two common Verilog literals are:

(a) String: A string literal is a one-dimensional array of characters enclosed in double quotes (" ").

(b) Numeric: constant numbers specified in binary, octal, decimal or hexadecimal.

Number Syntax.
n’Fddd..., where
n - integer representing number of bits
F - one of four possible base formats:
   b (binary), o (octal), d (decimal),
   h (hexadecimal). Default is d.
dddd - legal digits for the base format

Example.

"time is" // string literal
267 // 32-bit decimal number
2'b01 // 2-bit binary
20'hB36F // 20-bit hexadecimal number
'o62 // 32-bit octal number

A.5.2 Bit-Selects ”x[3]” and Part-Selects ”x[5:3]”

Bit-selects and part-selects are a selection of a single bit and a group of bits, respectively, from a wire, reg or parameter vector using square brackets "[ ]". Bit-selects and part-selects can be used as operands in expressions in much the same way that their parent data objects are used.

Syntax.

variable_name[index]
variable_name[msb:lsb]

Example.

reg [7:0] a, b;
reg [3:0] ls;
reg c;

c = a[7] & b[7]; // bit-selects
ls = a[7:4] + b[3:0]; // part-selects

A.6 Gates

A.6.1 Basic Gates

implement the basic logic gates. They have one output and one or more inputs. In the gate instantiation syntax shown below, GATE stands for one of the keywords and, nand, or, nor, xor, xnor.

Syntax.
Example.

and c1 (o, a, b, c, d); // 4-input AND called c1 and
c2 (p, f g); // a 2-input AND called c2.

or #(4, 3) ig (o, a, b); /* or gate called ig (instance name);
  rise time = 4, fall time = 3 */

xor #(5) xor1 (a, b, c); // a = b XOR c after 5 time units
xor (pull1, strong0) #5 (a,b,c); /* Identical gate with pull-up
  strength pull1 and pull-down

A.6.2 buf, not Gates

These implement buffers and inverters, respectively. They have one input and one or
more outputs. In the gate instantiation syntax shown below, GATE stands for either the
keyword buf or not

Syntax.

GATE (drive_strength) # (delays)
  instance_name1(output_1, output_2, ...., output_n, input),
  instance_name2(out1, out2, ..., outN, in);

Example.

not #(5) not_1 (a, c); // a = NOT c after 5 time units
buf c1 (o, p, q, r, in); // 5-output and 2-output buffers
c2 (p, f g);

Delays are ignored in synthesis. Only used for simulation
A.7 Behavioural VERILOG Syntax

Verilog has four levels of modelling:

1) The switch level which includes MOS transistors modelled as switches. This is not discussed here.
2) The gate level.
3) Register transfer level (RTL uses registers connected by Boolean equations).
4) The Behavioral or procedural level described below.

VERILOG procedural statements are used to model a design at a higher level of abstraction than the other levels. They provide powerful ways of doing complex designs. However small changes in coding methods can cause large changes in the hardware generated. Procedural statements can only be used in procedures. VERILOG procedures are described later in "Procedures: Always and Initial Blocks" and "Functions".

A.7.1 Delay in Assignment (not for synthesis)

In a delayed assignment, $\Delta t$ time units pass before the statement is executed and the left-hand assignment is made. With intra-assignment delay, the right side is evaluated immediately but there is a delay of $\Delta t$ before the result is placed in the left hand assignment. If another procedure changes a right-hand side signal during $\Delta t$, it does not affect the output. Delays are not supported by synthesis tools.

Syntax.

Procedural Assignment.

```
variable = expression
```

Delayed assignment.

```
#t variable = expression;
```

Intra-assignment delay

```
variable = #t expression;
```

Example.

```
reg [6:0] sum; reg h, ziltch;
ziltch = #15 ckz&h;  /* ckz&a evaluated now; ziltch changed after 15 time units. */

#10 hat = b&c;  /* 10 units after ziltch changes, b&c is evaluated and hat changes. */
```
A.7.2 Blocking Assignments

Procedural (blocking) assignments (=) are done sequentially in the order the statements are written. A second assignment is not started until the preceding one is complete.

Syntax.

Blocking

\[ \text{variable} = \text{expression;} \]
\[ \text{variable} = \#t \text{ expression;} \]
grab inputs now, deliver answer later.

\[ \#t \text{ variable} = \text{expression;} \]
grab inputs later, deliver answer later

Example 1. For simulation.

initial
begin
\[ a=1; \ b=2; \ c=3; \]
\[ \#5 \ a = b + c; \] // wait for 5 units, and execute \( a = b + c = 5 \).
\[ d = a; \] // Time continues from last line, \( d = 5 = b+c \) at \( t=5 \).

Example 2. For synthesis.

always @( posedge clk)
begin
\[ Z=Y; \ Y=X; \] // shift register
\[ y=x; \ z=x; \] // parallel flip flop.

Figure 123 show the synthesisable logic corresponding to the assignment in example 2. Note that there is no particular order in which we need to write the \( Z=Y; Y=X; \) assignments. When the clock triggers the flip flops, \( Y \) in its initial state is transferred to \( Z \) and \( X \) in its initial state is transferred to \( Y \).

A.7.3 Nonblocking (RTL) Assignments

RTL (nonblocking) assignments (\(<=\)), which follow each other in the code, are done in parallel. The right hand side of nonblocking assignments is evaluated starting from the completion of the last blocking assignment or if none, the start of the procedure. The transfer to the left hand side is made according to the delays. A delay in a non-blocking statement will not delay the start of any subsequent statement blocking or non-blocking. A good habit is to use "\(<=\)" if the same variable appears on both sides of the equal sign (Example 1 below).
Non-Blocking

\[
\begin{align*}
\text{variable} & \leq \text{expression}; \\
\text{variable} & \leq \#t \text{ expression}; \\
\#t \text{ variable} & \leq \text{expression};
\end{align*}
\]

It is recommended that blocking and nonblocking assignments not be mixed in the same always block. Syntax.

Having described the behavior of nonblocking assignments, it is important to understand why they are used in digital design. They are used as a method to model several concurrent data transfers that take place after a common event. Consider the following example where three concurrent data transfers take place at the positive edge of clock.

```verilog
always @(posedge clock)
begin
    reg1 <= #1 in1;
    reg2 <= @(negedge clock) in2 ^ in3;
    reg3 <= #1 reg1; //The old value of reg1
end
```

At each positive edge of clock, the following sequence takes place for the nonblocking assignments.

1. A read operation is performed on each right-hand-side variable, in1, in2, in3, and reg1, at the positive edge of clock. The right-hand-side expressions are evaluated, and the results are stored internally in the simulator.

2. The write operations to the left-hand-side variables are scheduled to be executed at the time specified by the intra-assignment delay in each assignment, that is, schedule "write" to reg1 after 1 time unit, to reg2 at the next negative edge of clock, and to reg3 after 1 time unit.
3. The write operations are executed at the scheduled time steps. The order in which
the write operations are executed is not important because the internally stored
right-hand-side expression values are used to assign to the left-hand-side values.
For example, note that reg3 is assigned the old value of reg1 that was stored after
the read operation, even if the write operation wrote a new value to reg1 before the
write operation to reg3 was executed.

Example 1. For simulation

```verbatim
initial
begin
    #3 b <= a;  /* grab a at t=0 Deliver b at t=3.
    #6 x <= b + c;  // grab b+c at t=0, wait and assign x at t=6.
    x is unaffected by b's change. */
```

Example 2. For synthesis

```verbatim
always @( posedge clk)
begin
    Z<=Y; Y<=X;  //shift register
    y<=x; z<=y;  //also a shift register.
```

Example 2 is illustrated in Figure 124.

![Figure 124: Top. A series flip flop implements the blocking assignment pair Z<=Y;Y<=X; (shift register). Bottom. A series flip flop implements the blocking assignment pair y<=x;z<=y; (also a shift register)](diagram)

Example 3. Use <= to transform a variable into itself.

```verbatim
reg G[7:0];
always @( posedge clk)
    G <= { G[6:0], G[7]};  // End around rotate 8-bit register.
```
Example 4. for simulation only.

initial begin
    a=1; b=2; c=3; x=4;
    #5 a = b + c; // wait for 5 units, then grab b,c and execute a=2+3.
    d = a; // Time continues from last line, d=5 = b+c at t=5.
    x <= #6 b + c; // grab b+c now at t=5, don’t stop, make x=5 at t=11.
    b <= #2 a; /* grab a at t=5 (end of last blocking statement).
              Deliver b=5 at t=7. previous x is unaffected by
    b change. */
    y <= #1 b + c; // grab b+c at t=5, don’t stop, make x=5 at t=6.
    #3 z = b + c; // grab b+c at t=8 (#5+#3), make z=5 at t=8.
    w <= x // make w=4 at t=8. Starting at last blocking

Rules to remember for synthesis.

(1) One must not mix "<=" or "=" in the same procedure.

(2) "<=" best mimics what physical flip-flops do; use it for "always @(posedge clk ..)"
type procedures.

(3) "=" best corresponds to what c/c++ code would do; use it for combinational pro-
    cedures.

A.7.4 begin...end

begin/end block statements are used to group several statements for use where one state-
ment is syntactically allowed. Such places include functions, always and initial blocks,
if, case and for statements. Blocks can optionally be named. These blocks can include
register and parameter declarations.

Syntax.

begin : block_name
  reg [msb:lsb] reg_variable_list;
  integer [msb:lsb] integer_list;
  parameter [msb:lsb] parameter_list;
  ... statements ...
end

Example.

initial // specify simulation waveforms
begin
  a = 2’b00; // at time = 0, a = 00
  #50 a = 2’b01; // at time = 50, a = 01
  #50 a = 2’b10; // at time = 100, a = 10
end

A.7.5 For loops

Similar to for loops in C/C++, they are used to repeatedly execute a statement or block
of statements. If the loop contains only one statement, the begin ... end statements may
be omitted.

Syntax.

for (count = value1;
     count </=/>/>= value2;
     count = count +/- step)
begin
  ... statements ...
end

Example.

for (j = 0; j <= 7; j = j + 1)
begin
  c[j] = a[j] & b[j];
  d[j] = a[j] | b[j];
end
A.7.6 while Loops

The while loop repeatedly executes a statement or block of statements until the expression in the while statement evaluates to false. To avoid combinatorial feedback during synthesis, a while loop must be broken with an @ (posedge/negedge clock) statement. For simulation a delay inside the loop will suffice. If the loop contains only one statement, the begin ... end statements may be omitted.

Syntax.

\[
\text{while (expression)} \\
\text{begin} \\
\hspace{1em} \text{... statements ...} \\
\text{end}
\]

Example.

\[
\text{while (!overflow)} \text{ begin} \\
\hspace{1em} @\text{(posedge clk)}; \\
\hspace{2em} a = a + 1; \\
\text{end}
\]

A.7.7 forever Loops

The forever statement executes an infinite loop of a statement or block of statements. To avoid combinatorial feedback during synthesis, a forever loop must be broken with an @ (posedge/negedge clock) statement. For simulation, a delay inside the loop will suffice. If the loop contains only one statement, the begin ... end statements may be omitted.

Syntax.

\[
\text{forever} \\
\text{begin} \\
\hspace{1em} \text{... statements ...} \\
\text{end}
\]

Example.

\[
\text{forever begin} \\
\hspace{1em} @\text{(posedge clk)}; \quad \text{// or use a= #9 a+1;} \\
\hspace{2em} a = a + 1; \\
\text{end}
\]
A.7.8 disable

Execution of a disable statement terminates a block and passes control to the next statement after the block. It is like the C break statement except it can terminate any loop, not just the one in which it appears. Disable statements can only be used with named blocks.

Syntax.

```
disable block_name;
```

Example.

```
begin: accumulate
forever
  begin
    @ (posedge clk);
    a = a + 1;
    if (a == 2'b0111) disable accumulate;
  end
end
```

A.7.9 if ... else if ... else

The if ... else if ... else statements execute a statement or block of statements depending on the result of the expression following the if. If the conditional expressions in all the if’s evaluate to false, then the statements in the else block, if present, are executed.

There can be as many else if statements as required, but only one if block and one else block. If there is one statement in a block, then the begin .. end statements may be omitted.

Both the else if and else statements are optional. However if all possibilities are not specifically covered, synthesis will generated extra latches.

Syntax.

```
if (expression)
  begin
    ... statements ...
  end
else if (expression)
  begin
    ... statements ...
  end
```
... more else if blocks ...
else
    begin
    ... statements ...
end

Example.

if (alu_func == 2'b00)
    aluout = a + b;
else if (alu_func == 2'b01)
    aluout = a - b;
else if (alu_func == 2'b10)
    aluout = a & b;
else // alu_func == 2'b11
    aluout = a | b;
if (a == b) // This if with no else will generate
begin // a latch for x and ot. This is so they
    x = 1;
    ot = 4'b1111;
end

A.7.10 case

The case statement allows a multipath branch based on comparing the expression with
a list of case choices. Statements in the default block executes when none of the case
choice comparisons are true (similar to the else block in the if ... else if ... else). If
no comparisons, including default, are true, synthesisers will generate unwanted latches.
Good practice says to make a habit of putting in a default whether you need it or not. If
the defaults are dont cares, define them as ‘x’ and the logic minimiser will treat them as
don’t cares. Case choices may be a simple constant or expression, or a comma-separated
list of same.

Syntax.

case (expression)
    case_choice1:
        begin
            ... statements ...
        end
    case_choice2:
        begin

185
... statements ...
end
... more case choices blocks ...
default:
begin
... statements ...
end
endcase
\begin{verbatim}
Example.
\begin{verbatim}
case (alu_ctr)
  2'b00: aluout = a + b;
  2'b01: aluout = a - b;
  2'b10: aluout = a & b;
  default: aluout = 1'bx; // Treated as don't cares for
endcase // minimum logic generation.

Example.
case (x, y, z)
  2'b00: aluout = a + b; //case if x or y or z is 2'b00.
  2'b01: aluout = a - b;
  2'b10: aluout = a & b;
  default: aluout = a | b;
endcase
\end{verbatim}

A.8 Timing Controls

A.8.1 Delay Control (Not synthesizable)

This specifies the delay time units before a statement is executed during a simulation. A
delay time of zero can also be specified to force the statement to the end of the list of
statements to be evaluated at the current simulation time.

Syntax.

#delay statement;

Example.

#5 a = b + c; // evaluated and assigned after 5 time units
#0 a = b + c; // very last statement to be evaluated
A.8.2 Event Control, @

This causes a statement or begin-end block to be executed only after specified events occur. An event is a change in a variable, and the change may be: a positive edge, a negative edge, or either (a level change), and is specified by the keyword posedge, negedge, or no keyword respectively. Several events can be combined with the or keyword. Event specification begins with the character @ and are usually used in always statements. For synthesis one cannot combine level and edge changes in the same list. For flip-flop and register synthesis the standard list contains only a clock and an optional reset. For synthesis to give combinational logic, the list must specify only level changes and must contain all the variables appearing in the right-hand-side of statements in the block.

Syntax.

@ (posedge variable or
    negedge variable) statement;
@ (variable or variable . . ) statement;

Example.

always
  @(posedge clk or negedge rst)
    if (rst) Q=0; else Q=D;  // Definition for a D flip-flop.
    // re-evaluate if a or b or e changes.
  @(a or b or e);
    sum = a + b + e;  // Will synthesize to a combinational adder.

A.9 Procedures

A.9.1 Always Blocks

The always block is the primary construct in RTL modeling. Like the continuous assignment, it is a concurrent statement that is continuously executed during simulation. This also means that all always blocks in a module execute simultaneously. This is very unlike conventional programming languages, in which all statements execute sequentially. The always block can be used to imply latches, flip-flops or combinational logic. If the statements in the always block are enclosed within begin ... end, the statements are executed sequentially. If enclosed within the fork ... join, they are executed concurrently (simulation only).

The always block is triggered to execute by the level, positive edge or negative edge of one or more signals (separate signals by the keyword or). A double-edge trigger is
implied if you include a signal in the event list of the always statement. The single edge-triggers are specified by posedge and negedge keywords. Procedures can be named. In simulation one can disable named blocks. For synthesis disable is mainly used as a comment.

Syntax 1

always @(event_1 or event_2 or ...)
begin
  ... statements ...
end

Syntax 2

always @(event_1 or event_2 or ...)
begin: name_for_block
  ... statements ...
end

Example.

always @(a or b)  // level-triggered; if a or b changes levels
always @(posedge clk);  // edge-triggered: on +ve edge of clk

A.9.2 Initial Blocks (not for synthesis)

The initial block is like the always block except that it is executed only once at the beginning of the simulation. It is typically used to initialize variables and specify signal waveforms during simulation. Initial blocks are not supported for synthesis.

Syntax.

initial
begin
  ... statements ...
end

Example.

initial
begin
  // variables initialized at
  clr = 0;
// beginning of the simulation

clk = 1;
end

initial // specify simulation waveforms
begin
  // at time = 0, a = 00
  a = 2'b00;
  // at time = 50, a = 01
  #50 a = 2'b01;
  // at time = 100, a = 10
  #50 a = 2'b10;
end

A.10 Functions

Functions are declared within a module, and can be called from continuous assignments, always blocks or other functions. In a continuous assignment, they are evaluated when any of its declared inputs change. In a procedure, they are evaluated when invoked.

Functions describe combinational logic, and by do not generate latches. Thus an if without an else will simulate as though it had a latch but synthesize without one. This is a particularly bad case of synthesis not following the simulation. It is a good idea to code functions so they would not generate latches if the code were used in a procedure. Functions are a good way to reuse procedural code, since modules cannot be invoked from a procedure.

Syntax. Function Declaration

function [msb:lsb] function_name;
  input [msb:lsb] input_arguments;
  reg [msb:lsb] reg_variable_list;
  parameter [msb:lsb] parameter_list;
  integer [msb:lsb] integer_list;
  ...
  statements ...
endfunction

Example.

function [7:0] my_func; // function return 8-bit value
  input [7:0] i;
  reg [4:0] temp;
  integer n;
  temp = i[7:4] | (i[3:0]);
  my_func = {temp, i[[1:0]]}; // Concatenation
endfunction
A.10.1 Function Return Value

When you declare a function, a variable is also implicitly declared with the same name as the function name, and with the width specified for the function name (The default width is 1-bit). This variable is ”my_func” in the above example. At least one statement in the function must assign the function return value to this variable.

A.10.2 Function Call

A function call is an operand in an expression. A function call must specify in its terminal list all the input parameters.

A.10.3 Function Rules

The following are some of the general rules for functions:

- Functions must contain at least one input argument.
- Functions cannot contain an inout or output declaration.
- Functions cannot contain time controlled statements (#, @).
- Functions must contain a statement that assigns the return value to the implicit function name register.

A.11 Compiler Directives

Compiler directives are special commands, beginning with ‘that affect the operation of the VERILOG simulator. Many VERILOG simulators parse and ignore compiler directives, and hence can be included even in synthesisable models.

A.11.1 Time Scale

‘timescale specifies the time unit and time precision. A time unit of 10 ns means a time expressed as say #2.3 will have a delay of 23.0 ns. Time precision specifies how delay values are to be rounded off during simulation. Valid time units include s, ms, ns, ps, fs. Only 1, 10 or 100 are valid integers for specifying time units or precision. It also determines the displayed time units in display commands like $display.

Syntax

‘timescale time_unit / time_precision;
Example.

`timescale 1 ns/1 ps // unit = 1ns, precision = 1/1000ns
'timescale 1 ns /100 ps // time unit = 1ns; precision = 1/10ns;

A.11.2 Macro Definitions

A macro is an identifier that represents a string of text. Macros are defined with the directive `define, and are invoked with the quoted macro name as shown in the example.

Syntax.

`define macro_name text_string;
... `macro_name ...

Example.

`define add_lsb a[7:0] + b[7:0]
assign 0 = `add_lsb; // assign 0 = a[7:0] + b[7:0];

A.11.3 Include Directive

Include is used to include the contents of a text file at the point in the current file where the include directive is. The include directive is similar to the C/C++ include directive.

Syntax.

`include

Example.

module x;
'include "dclr.v"; // contents of file "dclr.v" are put here

A.12 System tasks and functions

These are tasks and functions that are used to generate input and output during simulation. Their names begin with a dollar sign ($). Many synthesis tools parse and ignore system functions, and hence can be included even in synthesisable models. System tasks that extract data, like $monitor need to be in an initial or always block.
A.12.1 $display, $monitor

These commands have the same syntax, and display their values as text on the screen during simulation. They are much less convenient than waveform display tools like MOD-ELSIM. $display displays once every time they are executed, whereas $monitor displays every time one of its parameters changes. The format string is like that in C/C++, and may contain format characters. Format characters include %d (decimal), %h (hexadecimal), %b (binary), %c (character), %s (string) and %t (time). Append b, h, o to the task name to change default format to binary, octal or hexadecimal.

Syntax.

$display ("format_string",
   par_1, par_2, ... );

Example.

initial begin
   $displayh (b, d); // displayed in hexadecimal
   $monitor ("at time=%t, d=%h", $time, a);
end

A.12.2 $time, $stime, $realtime

These return the current simulation time as a 64-bit integer, a 32-bit integer, and a real number, respectively.
References


Index

74HC/HCT373 latch, 122
active, 104
ALU, 74
AND, 48
asserted, 56
asynchronous, 125
    inputs, 157
asynchronous inputs, 156
base, 13
BCD, 64
binary coding, 137, 140
bistate, 57
block diagram, 126
Boolean algebra, 46
boolean variables, 46
bus, 122
canonical, 57
canonical implementation, 135
canonical SOP representation, 59
clear, 106
clock skew, 157, 158
clocked, 125
CMOS, 34
code converter, 74
combinational circuit, 94
combinational logic, 54
comparator, 74
Counters, 118
D flip flop, 124
D latch, 102
debouncing, 159
decoder, 70
demultiplexer, 73
design
    state machine, 135
don’t cares, 64
edge triggered, 111
edge triggered, 100, 104
encoder, 72
    priority, 72
EQV, 50
exclusive NOR, 50
exclusive OR, 49
fan out, 38
finite state machine, 125
finite state machine, 125
Flip Flop, 100
flip flop, 105
gated D latch, 102
gated SR latch, 99, 100
glitch, 68
race, 161
    how to avoid, 162
go-no go, 157
Gray code, 63
hazard, 68
    cover, 69
dynamic, 68
    static, 68
hold state, 98
hold time, 119
implementation, 60
direct, 60
initialization, 161
inverter, 33, 47
JK flip-flops, 109
JK flip flop, 124
K-map, 61
    entered variable, 67
level triggered, 111
level triggered, 100, 104
logic gate, 34