# Digital System Design with SystemVerilog

Mark Zwoliński

July 28, 2008

# Contents


**1 Introduction** **5**
1.1 Modern Digital Design . . . . . 5
1.2 Designing with Hardware Description Languages . . . . . 6
1.2.1 Design Automation . . . . . 6
1.2.2 What is SystemVerilog? . . . . . 6
1.2.3 What is VHDL? . . . . . 7
1.2.4 Simulation . . . . . 7
1.2.5 Synthesis . . . . . 8
1.2.6 Reusability . . . . . 9
1.2.7 Verification . . . . . 9
1.2.8 Design flow . . . . . 11
1.3 CMOS technology . . . . . 12
1.3.1 Logic Gates . . . . . 12
1.3.2 ASICs and FPGAs . . . . . 15
1.4 Programmable logic . . . . . 21
1.5 Electrical Properties . . . . . 26
1.5.1 Noise Margins . . . . . 26
1.5.2 Fan-Out . . . . . 28
Summary . . . . . 29
Further Reading . . . . . 30
Exercises . . . . . 30

**2 Combinational Logic Design** **33**
2.1 Boolean Algebra . . . . . 33
2.1.1 Values . . . . . 33
2.1.2 Operators . . . . . 33
2.1.3 Truth Tables . . . . . 34
2.1.4 Rules of Boolean Algebra . . . . . 36
2.1.5 De Morgan's Law . . . . . 37
2.1.6 Shannon's expansion theorem . . . . . 37
2.2 Logic Gates . . . . . 37

2.3 Combinational Logic Design . . . . . 39
2.3.1 Logic Minimization . . . . . 40
2.3.2 Karnaugh Maps . . . . . 42
2.4 Timing . . . . . 47
2.5 Number Codes . . . . . 50
2.5.1 Integers . . . . . 52
2.5.2 Fixed Point Numbers . . . . . 52
2.5.3 Floating Point Numbers . . . . . 53
2.5.4 Alphanumeric Characters . . . . . 53
2.5.5 Gray Codes . . . . . 53
2.5.6 Parity Bits . . . . . 55

**3 Netlists and Structural SystemVerilog 59**
3.1 Basic Gate Models . . . . . 59
3.2 A Simple Netlist . . . . . 60
3.3 Delays . . . . . 61
3.4 Logic Values . . . . . 64
3.5 Logic Strengths . . . . . 64
3.6 Wired Logic . . . . . 67
3.7 Three-state Primitives . . . . . 69
3.8 User-Defined Primitives (UDPs) . . . . . 69

**4 Combinational Building Blocks 73**
4.1 Multiplexers . . . . . 73
4.1.1 2 to 1 Multiplexer . . . . . 73
4.1.2 4 to 1 Multiplexer . . . . . 75
4.2 Decoders . . . . . 76
4.2.1 2-4 Decoder . . . . . 76
4.2.2 $N - 2^N$ Decoder . . . . . 78
4.2.3 Seven-segment decoder . . . . . 79
4.3 Priority Encoder . . . . . 81
4.3.1 Don’t cares and uniqueness . . . . . 81
4.4 Adders . . . . . 82
4.4.1 Functional Model . . . . . 82
4.4.2 Ripple Adder . . . . . 84
4.5 Parity Checker . . . . . 85
4.6 Three state Buffers . . . . . 87
4.6.1 Multi-Valued Logic . . . . . 87

**5 Sequential logic blocks 91**
5.1 Latches . . . 91
5.1.1 SR latch . . . 91
5.1.2 D latch . . . 93
5.2 Flip-flops . . . 94
5.2.1 Edge-triggered D flip-flop . . . 94
5.2.2 Asynchronous set and reset . . . 95
5.2.3 Synchronous set and reset and clock enable . . . 96
5.3 JK and T flip-flops . . . 98
5.4 Registers and shift registers . . . 100
5.4.1 Multiple bit register . . . 100
5.4.2 Shift registers . . . 101
5.5 Counters . . . 104
5.5.1 Binary counter . . . 104
5.5.2 Johnson counter . . . 107
5.5.3 Linear feedback shift register . . . 109
5.6 Memory . . . 113
5.6.1 ROM . . . 113
5.6.2 Static RAM . . . 114
5.6.3 Synchronous RAM . . . 115
5.7 Sequential multiplier . . . 116

**6 Synchronous Sequential Design 121**
6.1 Synchronous sequential systems . . . 121
6.2 Models of synchronous sequential systems . . . 122
6.2.1 Moore and Mealy machines . . . 122
6.2.2 State registers . . . 123
6.2.3 Design of a three-bit counter . . . 124
6.3 Algorithmic state machines . . . 126
6.4 Synthesis from ASM charts . . . 131
6.4.1 Hardware implementation . . . 131
6.4.2 State assignment . . . 136
6.4.3 State minimization . . . 141
6.5 State machines in SystemVerilog . . . 146
6.5.1 A first example . . . 146
6.5.2 A sequential parity detector . . . 149
6.5.3 Vending machine . . . 151
6.5.4 Storing data . . . 153

**7 Complex Sequential Systems** **159**
7.1 Linked state machines . . . . . 159
7.2 Datapath/controller partitioning . . . . . 163
7.3 Instructions . . . . . 166
7.4 A simple microprocessor . . . . . 168
7.5 SystemVerilog model of a simple microprocessor . . . . . 172

**8 Writing Testbenches** **181**
8.1 A First Example . . . . . 182
8.2 Clock Generation . . . . . 183
8.3 Reset and other deterministic signals . . . . . 184
8.4 Random timing . . . . . 185
8.5 Synchronised signals . . . . . 185
8.6 Monitoring Responses . . . . . 186
8.6.1 Printing output data . . . . . 187
8.6.2 Comparing responses . . . . . 188

**9 SystemVerilog Simulation** **191**
9.1 Races . . . . . 193
9.2 Avoiding Races . . . . . 194
9.3 Delay Models . . . . . 195

**10 SystemVerilog Synthesis** **199**
10.1 RTL synthesis . . . . . 201
10.1.1 Non-synthesizable SystemVerilog . . . . . 202
10.1.2 Inferred flip-flops and latches . . . . . 202
10.1.3 Combinational logic . . . . . 207
10.1.4 Summary of RTL synthesis rules . . . . . 213
10.2 Constraints . . . . . 213
10.2.1 Attributes . . . . . 214
10.2.2 Area and structural constraints . . . . . 214
10.2.3 full_case and parallel_case attributes . . . . . 217
10.3 Synthesis for FPGAs . . . . . 219
10.4 Verifying synthesis results . . . . . 222

**11 Testing Digital Systems** **225**
11.1 The need for testing . . . . . 225
11.2 Fault Models . . . . . 226
11.2.1 Single-Stuck Fault Model . . . . . 227
11.2.2 PLA Faults . . . . . 227
11.3 Fault-oriented Test Pattern Generation . . . . . 228

11.3.1 Sensitive Path Algorithm . . . . . 230
11.3.2 Undetectable Faults . . . . . 231
11.3.3 The D Algorithm . . . . . 232
11.3.4 PODEM . . . . . 235
11.3.5 Fault Collapsing . . . . . 236
11.4 Fault simulation . . . . . 236
11.4.1 Parallel fault simulation . . . . . 238
11.4.2 Concurrent fault simulation . . . . . 239
11.5 Fault Simulation in Verilog . . . . . 242

**12 Design for Testability 249**
12.1 Ad hoc Testability Improvements . . . . . 250
12.2 Structured Design for Test . . . . . 251
12.3 Built-In Self-Test . . . . . 253
12.3.1 Example . . . . . 256
12.3.2 Built-In Logic Block Observation (BILBO) . . . . . 260
12.4 Boundary Scan (IEEE 1149.1) . . . . . 263

**13 Asynchronous Sequential Design 277**
13.1 Asynchronous Circuits . . . . . 277
13.2 Analysis of Asynchronous Circuits . . . . . 280
13.2.1 Informal Analysis . . . . . 280
13.2.2 Formal Analysis . . . . . 283
13.3 Design of Asynchronous Sequential Circuits . . . . . 285
13.4 Asynchronous state machines . . . . . 294
13.5 Setup and Hold Times and Metastability . . . . . 297
13.5.1 The Fundamental Mode Restriction and Synchronous Circuits . . . . . 297
13.5.2 Random Pulse Generator . . . . . 298
13.5.3 SystemVerilog Modelling of Setup and Hold Time Violations . . . . . 298
13.5.4 Metastability . . . . . 299

**14 Interfacing with the Analogue World 305**
14.1 Digital to Analogue Converters . . . . . 306
14.2 Analogue to Digital Converters . . . . . 307
14.3 Verilog-AMS . . . . . 310
14.3.1 Mixed-Signal Modelling . . . . . 310
14.4 Phased-Locked Loops . . . . . 312
14.5 Verilog-AMS simulators . . . . . 316

# Chapter 1

# Introduction

In this chapter we will review the design process, with particular emphasis on the design of digital systems using Hardware Description Languages such as SystemVerilog. The technology of CMOS integrated circuits will be briefly revised and programmable logic technologies will be discussed. Finally, the relevant electrical properties of CMOS and programmable logic are reviewed.

## 1.1 Modern Digital Design

Electronic circuit design has traditionally fallen into two main areas: analogue and digital. These subjects are usually taught separately and electronics engineers tend to specialize in one area. Within these two groupings there are further specializations, such as radio frequency analogue design; digital integrated circuit design and, where the two domains meet, mixed-signal design. In addition, of course, software engineering plays an increasingly important role in embedded systems.

Digital electronics is ever more significant in consumer goods. Cars have sophisticated control systems. Most homes now have personal computers. Products that used to be thought of as analogue, such as radio, television and telephones are or are becoming digital. Digital Compact Discs have entirely replaced analogue LPs for recorded audio. With these changes, the lifetimes of products have lessened. In a period of less than a year, new models will probably have replaced all the digital electronic products in your local store.

## 1.2 Designing with Hardware Description Languages

### 1.2.1 Design Automation

To keep pace with this rapid change, electronics products have to be designed extremely quickly. Analogue design is still a specialized (and well-paid) profession. Digital design has become very dependent on Computer-Aided Design (CAD) – also known as Design Automation (DA) or Electronic Design Automation (EDA). The EDA tools allow two tasks to be performed: *synthesis*, in other words the translation of a specification into an actual implementation of the design; and *simulation* in which the specification, or the detailed implementation can be exercised in order to verify correct operation.

Synthesis and simulation EDA tools require that the design be transferred from the designer's imagination into the tools themselves. This can be done by drawing a diagram of the design using a graphical package. This is known as *schematic capture*. Alternatively, the design can be represented in a textual form, much like a software program. Textual descriptions of digital hardware can be written in a modified programming language, such as C, or in a *Hardware Description Language* (HDL). Over the past thirty years, or so, a number of HDLs have been designed. Two HDLs are in common usage today: Verilog and VHDL (VHSIC Hardware Description Language, where VHSIC stands for Very High Speed Integrated Circuit). Standard HDLs are important because they can be used by different CAD tools from different tool vendors. In the days before Verilog and VHDL, every tool had its own HDL, requiring laborious translation between HDLs, for example to verify the output from a synthesis tool with another vendor's simulator.

### 1.2.2 What is SystemVerilog?

*SystemVerilog* is a hardware description language (HDL). In many respects, an HDL resembles a software programming language, but HDLs have several features not present in languages such as C.

*Verilog* was first developed in the early 1980s. It is based on Hilo-2, which was a language (and simulator) from Brunel University. The company who first developed Verilog, Gateway Design Automation, was bought out by Cadence. In the early 1990s Cadence put the language into the public domain and in 1995, Verilog became an IEEE (Institute of

Electrical and Electronics Engineers, Inc., USA) standard – 1364. In 2001, a new version of the standard was agreed, with many additional features. Work is continuing to extend Verilog to system-level modelling. This new language is known as SystemVerilog, the latest version of which is 3.1a (the number assumes the 1995 version of Verilog was version 1.0 and the 2001 revision was 2.0). The language became an IEEE standard, 1800, in 2005.

Verilog has also been extended to allow modelling of analogue circuits (Verilog-A) and again for mixed-signal modelling (Verilog-AMS).

### 1.2.3 What is VHDL?

During the same period of time, another HDL – VHSIC (Very High Speed Integrated Circuit) HDL or VHDL was developed for the US Department of Defense and was also standardised by the IEEE as standard 1076. There have been three versions of IEEE 1076, in 1987, 1993 and 2002. There have been other HDLs – e.g. Ella, UDL/I, but now Verilog and VHDL are dominant. Ironically, given their pedigrees, VHDL is widely used in Europe, while Verilog is predominant on the West Coast of the USA. Each language has its champions and detractors. Objectively (if it's possible to take a truly unbiased view), both languages have weaknesses and it's futile getting into arguments about which is best.

### 1.2.4 Simulation

Another HDL is, however, worthy of note. SystemC uses features of C++ to allow modelling of hardware. At this time it is not possible to predict whether SystemC might supersede SystemVerilog or VHDL. It is, however, worth noting that many of the design style guidelines refer to all three languages.

An HDL has three elements that are seldom present in a programming language – concurrency, representation of time and representation of structure.

Hardware is intrinsically parallel. Therefore an HDL must be able to describe actions that happen simultaneously. C (to choose a typical and widely-used programming language) is sequential.

Actions in hardware take a finite time to complete. Therefore mechanisms are needed to describe the passage of time.

The structure of hardware is significant. A C program can consist of functions that are called and having completed their task, no longer retain

any sense of internal state. On the other hand, gates or other hardware structures persist and have a state even when they appear to be doing nothing.

SystemC allows these features to be described in a C-like language. SystemVerilog (and VHDL) have these features built in.

Concurrency, time and structure lead to another significant difference between an HDL and a programming language. A C program can be compiled and executed on a PC or workstation. SystemVerilog can be compiled, but needs a simulator to execute. The simulator handles the interactions between concurrent elements and models the passage of time. The simulator also keeps track of the state of each structural element. A number of SystemVerilog simulators are available. It is essential that you obtain a simulator to complete this course. A list of Web sites is included at the end of this chapter.

Advocates of Verilog often argue that it is an easier language to learn than VHDL. I would disagree with this (I think they're about the same) for one reason. VHDL has a very well-defined simulation model. Two different VHDL simulators are (almost) guaranteed to produce exactly the same simulations. The Verilog simulation model is more loosely defined. Unless you are very careful, two different SystemVerilog simulators may produce *different* simulations. For this reason, this book does not attempt to cover every detail of the SystemVerilog language. Instead, the intention is to show how to write models of hardware that will simulate and synthesise with predictable behaviour.

### 1.2.5 Synthesis

SystemVerilog is a hardware *description* language – not a hardware *design* language. In 1983, digital simulation was a mature technology; automatic hardware synthesis was not. (This argument applies equally to VHDL.) It is possible to write models in SystemVerilog that do not and cannot correspond to any physically realisable hardware. Only a subset of the language can be synthesised using current RTL (Register Transfer Level) synthesis tools. Moreover, RTL synthesis tools work by recognising particular patterns in the code and use those patterns to infer the existence of registers. (RTL synthesis tools also optimise combinational logic – but not much more.) Therefore the style of SystemVerilog coding is important. An IEEE standard – 1364.1 – was agreed in 2002. This defines a subset of Verilog, and the meaning of that subset, of the 2001 revision of Verilog.

The hardware models in this course will conform to an application of

the 1364.1-2002 RTL synthesis standard to the SystemVerilog language. In other words, we will use the synthesis standard as a style guide.

### 1.2.6 Reusability

The electronics industry is currently very keen on the idea of reuse. Integrated circuits are so large and complex that it is almost impossible for one team to create a design from scratch. Instead, it is expected that more and more of a design will consist of parts reused from earlier projects or bought in from specialised design companies. Clearly, if a design is to be reused, it has to be versatile. It has either to be so common that everyone will want to use it, or adaptable, such that it can be used in a variety of designs.

At a simple level, imagine that you have been asked to design a four-bit multiplier. This can be done by setting the widths of the inputs and outputs to 4. You would also need to set the widths of some internal registers. At a later date, you might be asked to design a thirteen-bit adder. At a functional level (or RTL for a synthesisable design), the difference between the two designs is simply a change of input, output and registrer widths. Both the new and original designs would have needed simulating and synthesising. It's possible you might make a mistake in changing your design from 4 to 13 bits. This would require debugging effort. Imagine instead, that you had designed an "n-bit" multiplier. This would be debugged once. When asked to produce the 13-bit multiplier, you would simply plug the parameter "13" into the model and take the rest of the day off! The idea of producing parameterisable designs is therefore very attractive. We will, as far as is possible, design parameterisable, reusable components.

We will also show how to write models that are likely to behave the same way in different simulators and that synthesise with the same results with different synthesis tools. Related to this is a need to ensure that the behaviour after synthesis is the same as the behaviour before synthesis.

### 1.2.7 Verification

How do we know that a model accurately describes the hardware that we want to build? Formal verification tools exist, but they are somewhat specialised and difficult to use. Simulation is a much more common technique. In simulation, we try to give a design as wide a range of inputs as possible in order to cover everything that could happen to that design. This approach can appply to each individual part and to the system as a whole.

As the hardware model gets larger, the range of possible behaviours increases. Therefore it gets harder to exhaustively test the model and the simulation time gets greater. This is a disadvantage of using simulation as a verification tool.

In this course a number of examples are given. You are encouraged to investigate these models by running simulations. To do this, you will need to provide test stimuli. One of factors that has made SystemVerilog so important is the ability to use the language itself to describe these test stimuli. This may seem obvious – in 1983, this was an innovation; hardware modelling and test definitions were usually done using entirely different languages. Latterly, Hardware *Verification* Languages, such as Vera and $e$ were developed, but many of their features have been absorbed into SystemVerilog.

In the jargon, the test stimuli are defined in a "testbench". A testbench is a piece of SystemVerilog code that (conceptually) surrounds the model and defines the universe as seen by that model. Therefore a testbench has no inputs or outputs (and can be identified by this feature). Within a testbench, you might write a description of a clock generator and define a sequence of inputs. You might also check the responses from the model. Within the simulator, you can display the waveforms of signals, even signals deep within the design.

Writing testbenches requires a different coding style to hardware modelling. A testbench does not represent a piece of real hardware. Indeed, you should never attempt to synthesise a testbench – you'll just get pages of warning messages. Again, the SystemVerilog simulation problem arises here. A testbench may behave differently for different simulators. We'll try to minimise this problem, but it's a less precise art than writing portable RTL models.

Simulation can help to ensure that your design implements the specification as accurately as is humanly possible (and humans are never capable of perfection). We can, with a bit of luck, assume that the synthesis process correctly translates a SystemVerilog description into gates and flip-flops. When thousands or millions of the final integrated circuit are manufactured, it is inevitable that defects will occur in a small (we hope) number of chips. These defects can be caused by, for example, dirt or imperfections in the silicon. If these defects cause the electrical behaviour of the circuit to change, the circuit will not work correctly. Such faulty circuits need to be detected at the time of manufacture. *Fault simulation* allows potential faults in a circuit to be modelled and rapidly simulated. Another testbench or set of testbenches and a fault simulator are needed to determine a minimal set of test vectors to uncover all possible faults.

### 1.2.8 Design flow

Most digital systems are sequential, that is they have states, and the outputs depend on the present state. Some early designs of computer were asynchronous; in other words, the transition to a new state happened as soon as inputs had stabilized. For many years, digital systems have tended to be synchronous. In a synchronous system, the change of state is triggered by one or more clock signals. In order to design reliable systems, formal design methodologies have been defined. The design of a (synchronous sequential) digital system using discrete gates would therefore proceed as follows.

1. Write a specification.
2. If necessary, partition the design into smaller parts and write a specification for each part.
3. From the specification draw a state machine chart. This shows each state of the system and the input conditions that cause a change of state, together with the outputs in each state.
4. Minimize the number of states. This is optional and may not be useful in all cases.
5. Assign Boolean variables to represent each state.
6. Derive next state and output logic.
7. Optimize the next state and output logic to minimize the number of gates needed.
8. Choose a suitable placement for the gates in terms of which gates share integrated circuits and in terms of where each integrated circuit is placed on the printed circuit board.
9. Design the routing between the integrated circuits.

In general, steps 1 and 2 cannot be avoided. This is where the creativity of the designer is needed. Most books on digital design concentrate on steps 3 to 7. Steps 8 and 9 can be performed manually, but placement and routing was one of the first tasks to be successfully automated. It is possible to simulate the design at different stages if it is converted into a computer-readable form. Typically, in order to perform the placement and routing, a schematic capture program would be used at around step

7, such that the gate-level structure of the circuit would be entered. This schematic could be converted to a form suitable for a logic simulator. After step 9 had been completed, the structure of the circuit, including any delays generated by the resistance and capacitance of the interconnect could be extracted and again simulated.

The implementation of digital designs on ASICs or FPGAs therefore involves the configuration of connections between predefined logic blocks. As noted, we cannot avoid steps 1 and 2, above and steps 8 and 9 can be done automatically. The use of an HDL, here SystemVerilog, means that the design can be entered into a CAD system and simulated at step 3 or 4, rather than step 7. So-called register transfer level (RTL) synthesis tools automate steps 6 and 7. Step 4 still has to be done by hand. Step 5 can be automated, but now the consequences of a particular state assignment can be assessed very quickly. Behavioural synthesis tools are starting to appear that automate the process from about step 2 onwards. Figure 1.1 shows the overall design flow for RTL synthesis-based design.

Because of this use of Electronic Design Automation (EDA) tools to design ASICs and FPGAs, a course such as this can concentrate on higher-level aspects of design, in particular the description of functional blocks in an HDL. Many books on digital design describe multiple output and multi-level logic minimization, including techniques such as the Quine–McCluskey algorithm. Here, we assume that a designer may occasionally wish to minimize expressions with a few variables and a single output, but if a complex piece of combinational logic is to be designed a suitable EDA tool is available that will perform the task quickly and reliably.

## 1.3 CMOS technology

### 1.3.1 Logic Gates

The basic building blocks of digital circuits are *gates*. A gate is an electronic component with a number of inputs and, generally, a single output. The inputs and the outputs are normally in one of two states: logic 0 or logic 1. These logic values are represented by voltages (for instance, 0 V for logic 0 and 3.3V for logic 1) or currents. The gate itself performs a logical operation using all of its inputs to generate the output. Ultimately, of course, digital gates are really analogue components, but for simplicity we tend to ignore their analogue nature.

It is possible to buy a single integrated circuit containing, say, four identical gates, as shown in Figure 1.2. (Note that two of the connections are

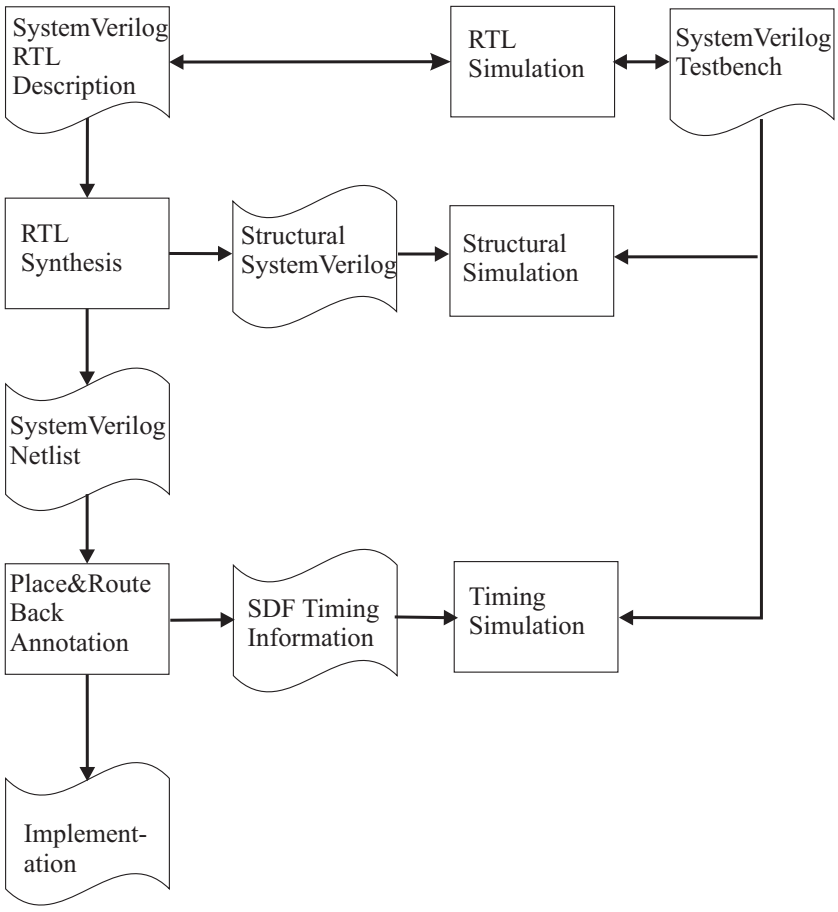



Figure 1.1: RTL synthesis design flow

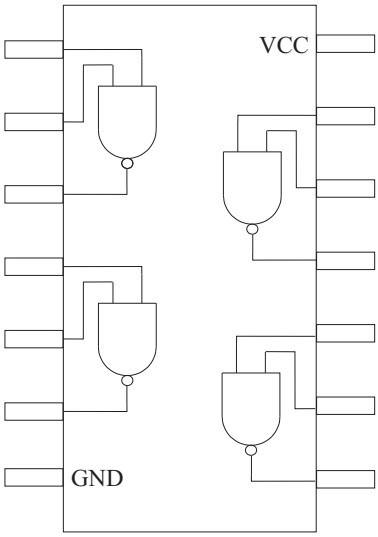



Figure 1.2: Small-scale integrated circuit

for the positive and negative power supplies to the device. These connections are not normally shown in logic diagrams.) A digital system could be built by connecting hundreds of such devices together – indeed many systems have been designed in that way. Although the individual integrated circuits might cost as little as 10 cents each, the cost of designing the printed circuit board for such a system and the cost of assembling the board are very significant and this design style is no longer cost effective.

Much more complicated functions are available as mass-produced integrated circuits, ranging from flip-flops through to microprocessors. With increasing complexity comes flexibility – a microprocessor can be programmed to perform a near-infinite variety of tasks. Digital system design therefore consists, in part, of taking standard components and connecting them together. Inevitably, however, some aspect of the functionality will not be available as a standard device. The designer is then left with the choice of implementing this functionality from discrete gates or of designing a specialized integrated circuit to perform that task. While this latter task may appear daunting, it should be remembered that the cost of a system will depend to a great extent not on the cost of the individual components but on the cost of connecting those components together.

### 1.3.2 ASICs and FPGAs

The design of a high-performance, full-custom integrated circuit (IC) is, of course, a difficult task. In full-custom IC design, *everything*, down to and including individual transistors may be designed (although libraries of parts are, of course, used). For many years, however, it has been possible to build semi-custom integrated circuits using *gate arrays*. A gate array, as its name suggests, is an integrated circuit on which an array of logic gates has been created. The design of an *Application Specific Integrated Circuit* (ASIC) using a gate array therefore involves the definition of how the gates in the array should be connected. In practical terms, this means that one or two layers of metal interconnect must be designed. Since an integrated circuit requires seven or more processing stages, all the processing steps other than the final metalization can be completed in advance. Because the uncommitted gate arrays can be produced in volume, the cost of each device is relatively small.

The term ASIC is often applied to full-custom and semi-custom integrated circuits. Another class of integrated circuit is that of *programmable logic*. The earliest programmable logic devices (PLDs) were *Programmable Logic Arrays* (PLAs). Like gate arrays, these consist of arrays of uncommitted logic, but unlike *mask-programmable* gate arrays, the configuration of the array is determined by applying a large (usually negative) voltage to individual connections. The general structure of a PLA is shown in Figure 1.3. The PLA has a number of inputs (A, B, C) and outputs (X, Y, Z), an AND-plane and an OR-plane. Connections between the inputs and the product terms (P, Q, R, S) and between the product terms and outputs are shown; the remaining connections have been removed as part of the programming procedure. Some PLAs may be reprogrammed electrically, or by restoring the connections by exposing the device to ultra-violet light. PALs (Programmable Array Logic) extend the idea of PLAs to include up to 12 flip-flops. In recent years, programmable devices have become much more complex and include CPLDs (Complex PLDs) and FPGAs (*Field Programmable Gate Arrays*). FPGAs are described in more detail in section 1.4.

Even digital gates can be thought of as analogue circuits. The design of individual gates is therefore a circuit design problem. Hence there exist a wide variety of possible circuit structures. Very early digital computers were built using vacuum tubes. These gave way to transistor circuits in the 1960s and 1970s. There are two major types of transistor: bipolar junction transistors (BJTs) and field effect transistors (FETs). Logic families such as TTL (transistor–transistor logic) and ECL (emitter–collector logic) use

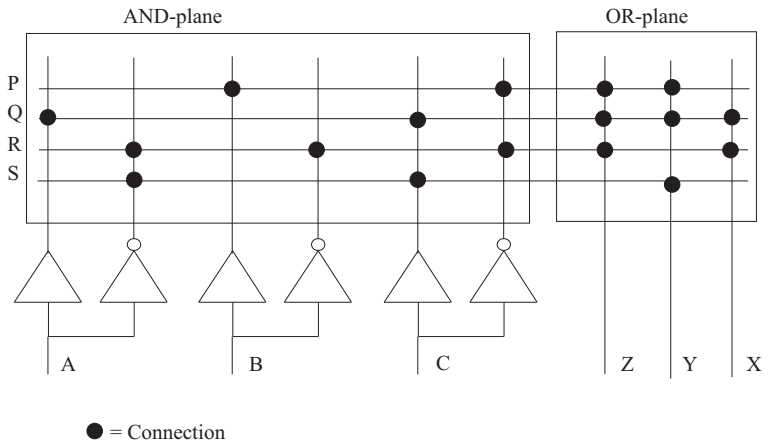



Figure 1.3: PLA structure

BJTs. Today, the dominant (but not exclusive) technology is CMOS, which uses FETs. CMOS derives its name from the particular type of FET used – the MOSFET (metal oxide semiconductor FET). CMOS therefore stands for complementary MOS, as two types of MOS device are used. MOS is, in fact, a misnomer; a better term is IGFET (insulated gate FET).

The structure of an n-type (NMOS) MOS transistor is shown in Figure 1.4, which is not drawn to scale. The substrate is the silicon wafer that has been doped to make it p-type. The thickness of the substrate is therefore significantly greater than the other transistor dimensions. Two heavily doped regions of n-type silicon are created for each transistor. These form the source and drain. In fact, the *source* and *drain* are interchangeable, but by convention the drain–source voltage is usually positive. Metal connections are made to the source and drain. The polycrystalline silicon (polysilicon) gate is separated from the rest of the device by a layer of silicon dioxide insulator. Originally the gate would have been metal – hence the name MOS was derived from the structure of the device (metal oxide semiconductor).

When the gate voltage is the same as the source voltage, the drain is insulated from the source. As the gate voltage rises, the gate–oxide–semiconductor sandwich acts as a capacitor, and negative charge builds up on the surface of the semiconductor. At a critical *threshold voltage* the charge is sufficient to create a channel of n-type silicon between the source and drain. This acts as a conductor between the source and the drain. Therefore the NMOS transistor can be used as a switch that is open when the gate voltage is low and closed when the gate voltage is high.

A PMOS transistor is formed by creating heavily doped p-type drain

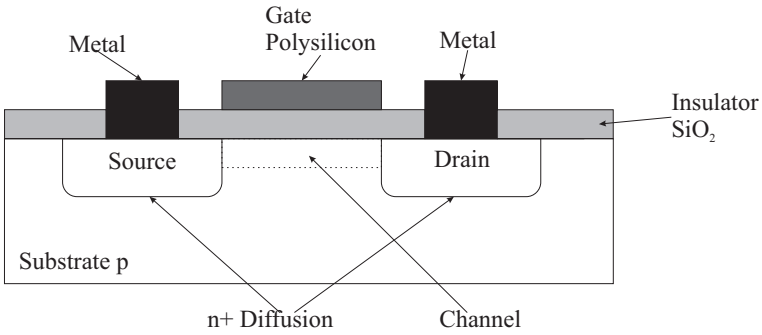



Figure 1.4: NMOS transistor structure

and source regions in an n-type substrate. A PMOS transistor conducts when the gate voltage is low and does not conduct when the gate voltage is high.

Symbols for NMOS transistors are shown in Figure 1.5(a) and (b). The substrate is also known as the *bulk*, hence the symbol B. In digital circuits, the substrate of NMOS transistors is always connected to ground (logic 0) and hence can be omitted from the symbol, as shown in Figure 1.5(b). Symbols for PMOS transistors are shown in Figure 1.5(c) and (d). Again the bulk connection is not shown in Figure 1.5(d), because in digital circuits the substrate of a PMOS transistor is always connected to the positive supply voltage (logic 1).

A logical inverter (a NOT gate) can be made from an NMOS transistor and a resistor, or from a PMOS transistor and a resistor, as shown in Figure 1.6(a) and (b), respectively. VDD is the positive supply voltage (3.3 V to 5 V); GND is the ground connection (0 V). The resistors have a reasonably high resistance, say 10 k$\Omega$. When IN is at logic 1 (equal to the VDD voltage), the NMOS transistor in Figure 1.6(a) acts as a closed switch. Because the resistance of the NMOS transistor, when it is conducting, is much less than that of the resistor, OUT is connected to GND, giving a logic 0 at that node. In the circuit of Figure 1.6(b), a logic 1 at IN causes the PMOS transistor to act as an open switch. The resistance of the PMOS transistor is now much greater than that of the resistance, so OUT is connected to GND via the resistor. Again a logic 0 is asserted at OUT.

A logic 0 at IN causes the opposite effects. The NMOS transistor becomes an open switch, causing OUT to be connected to VDD by the resistor; the PMOS transistor becomes a closed switch with a lower resistance than the resistor and again OUT is connected to VDD.

Figure 1.6(c) shows a CMOS inverter. Here, both PMOS and NMOS

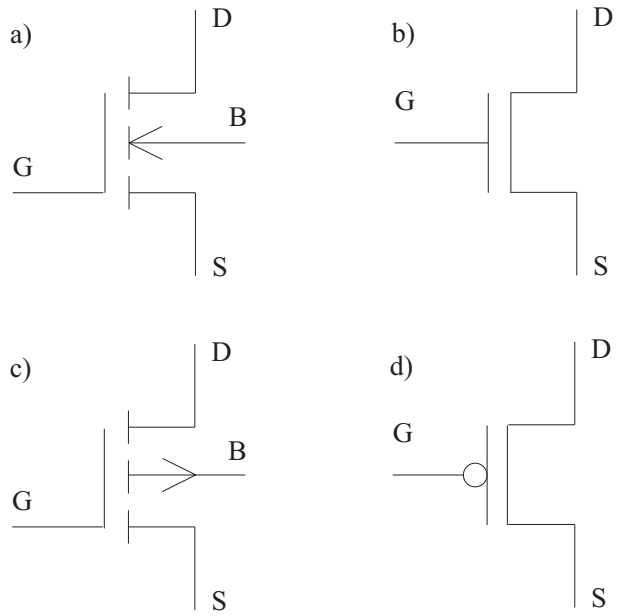



Figure 1.5: MOS transistor symbols: (a), (b) NMOS, (c), (d) PMOS

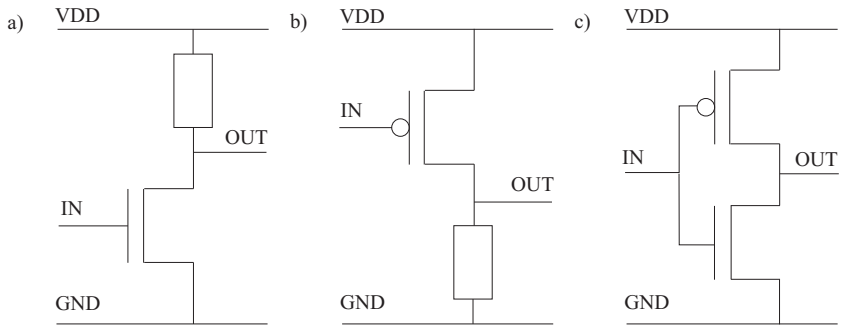



Figure 1.6: MOS inverters: (a) NMOS, (b) PMOS, (c) CMOS

transistors are used. A logic 1 at IN will cause the NMOS transistor to act as a closed switch and the PMOS transistor to act as an open switch, giving a 0 at OUT. A logic 0 will have the opposite effect: the NMOS transistor will be open and the PMOS transistor will be closed. The name CMOS comes from complementary MOS – the NMOS and PMOS transistors complement each other.

Current flows in a semiconductor as electrons move through the crystal matrix. In p-type semiconductors it is convenient to think of the charge being carried by the absence of an electron, a 'hole'. The mobility of holes is less than that of electrons (i.e. holes move more slowly through the crystal matrix than electrons). The effect of this is that the gain of a PMOS transistor is less than that of the same-sized NMOS transistor. Thus to build a CMOS inverter with symmetrical characteristics, in the sense that a 0 to 1 transition happens at the same rate as a 1 to 0 transition, requires that the gain of the PMOS and NMOS transistors be made the same. This is done by varying the widths of the transistors (assuming the lengths are the same) such that the PMOS transistor is about 2.5 times as wide as the NMOS transistor. As will be seen, this effect is compensated for in CMOS NAND gates, where similarly sized NMOS and PMOS transistors can be used. CMOS NOR gates, however, do require the PMOS transistors to be scaled. Hence, NAND gate logic is often preferred for CMOS design.

Two-input CMOS NAND and NOR gates are shown in Figure 1.7(a) and (b), respectively. The same reasoning as used in the description of the inverter may be applied. A logic 1 causes an NMOS transistor to conduct and a PMOS transistor to be open; a logic 0 causes the opposite effect. NAND and NOR gates with three or more inputs can be constructed using similar structures. Note that in a NAND gate all the PMOS transistors must have a logic 0 at their gates for the output to go high. As the transistors are working in parallel, the effect of the lower mobility of holes on the gain of the transistors is overcome.

Figure 1.8 shows a CMOS AND–OR–Invert structure. The function $\overline{A.B + C.D}$ can be implemented using eight transistors compared with the 14 needed for three NAND/NOR gates and an inverter.

A somewhat different type of structure is shown in Figure 1.9(a). This circuit is a three-state buffer. When the *EN* input is at logic 1, and the $\overline{EN}$ input is at logic 0, the two inner transistors are conducting and the gate inverts the *IN* input as normal. When the *EN* input is at logic 0 and the $\overline{EN}$ input is at logic 1, neither of the two inner transistors is conducting and the output floats. The $\overline{EN}$ input is derived from *EN* using a standard CMOS inverter. An alternative implementation of a three-state buffer is shown in Figure 1.9(b). Here a transmission gate follows the CMOS inverter. The

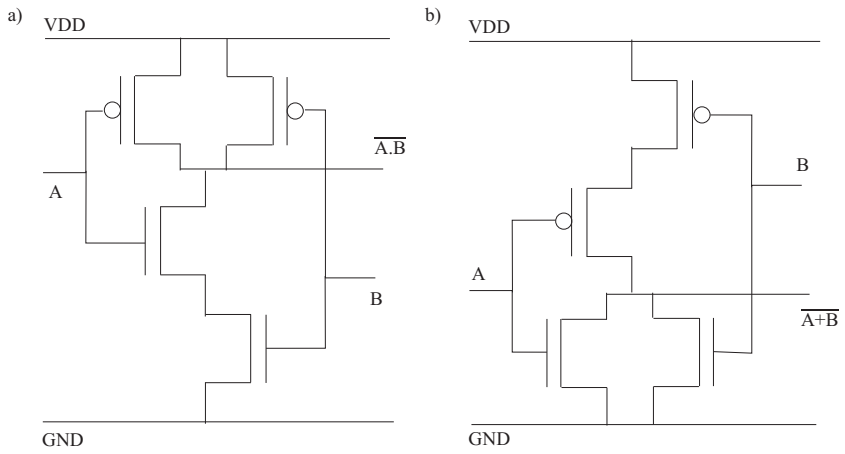



Figure 1.7: (a) CMOS NAND; (b) CMOS NOR

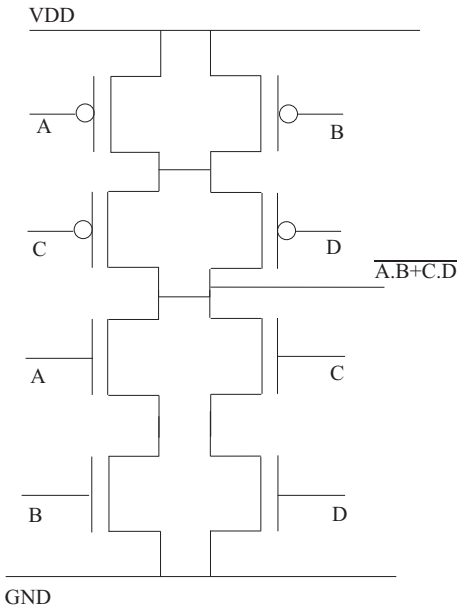



Figure 1.8: CMOS AND-OR-INVERT

NMOS and PMOS transistors of the transmission gate are controlled by complementary signals. When *EN* is at logic 1 and $\overline{EN}$ is at logic 0, both transistors conduct; otherwise both transistors are open circuit.

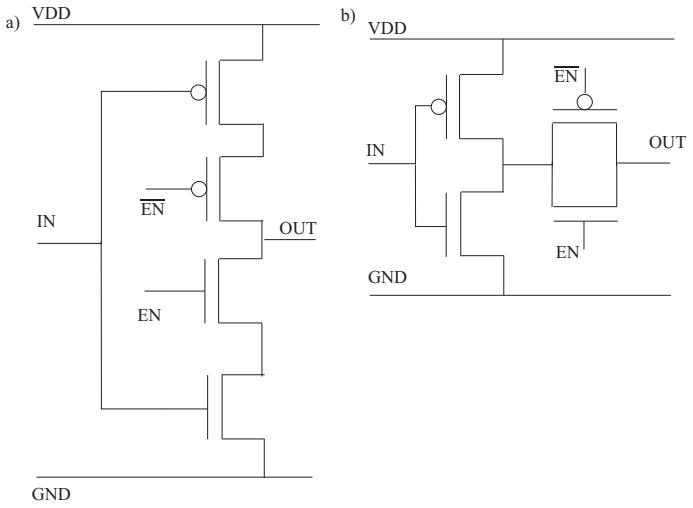



Figure 1.9: CMOS three-state buffer

Figure 1.10(a) shows a two-input multiplexer constructed from transmission gates while Figure 1.10(b) and (c) show an exclusive OR gate and a D latch, respectively, that both use CMOS transmission gates. All these circuits use fewer transistors than the equivalent circuits constructed from standard logic gates. It should be noted, however, that the simulation of transmission gate circuits can be problematic. VHDL, in particular, is not well suited to this type of transistor-level modelling, and we do not give any examples in this book, other than of general three-state buffers.

## 1.4 Programmable logic

While CMOS is currently the dominant technology for integrated circuits, for reasons of cost and performance, many designs can be implemented using programmable logic. The major advantage of *programmable logic* is the speed of implementation. A programmable logic device can be configured on a desktop in seconds, or at most minutes. The fabrication of an integrated circuit can take several weeks. The cost per device of a circuit built in programmable logic may be greater than that of a custom integrated circuit, and the performance, in terms of both speed and functionality, is likely to be less impressive than that of CMOS. These apparent

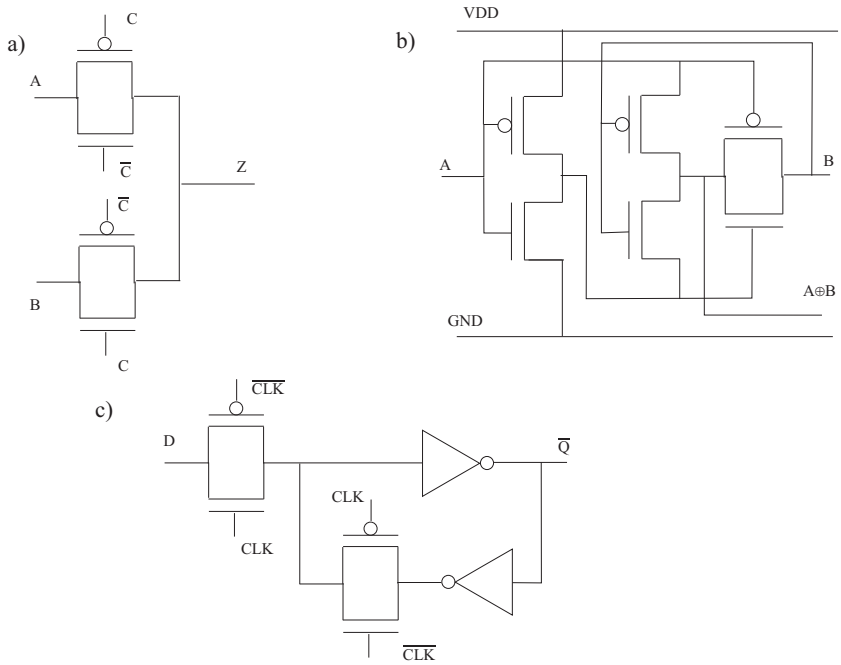



Figure 1.10: CMOS transmission gate circuits. (a) Multiplexer; (b) XOR; (c) D latch

disadvantages are often outweighed by the ability to rapidly produce working integrated circuits. Thus programmable logic is suited to prototypes, but also increasingly to small production volumes.

One recent application of programmable devices is as *reconfigurable logic*. A system may perform different functions at different points in time. Instead of having all the functionality available all the time, one piece of hardware may be reconfigured to implement the different functions. New functions, or perhaps better versions of existing functions, could be downloaded from the Internet. Such applications are likely to become more common in future.

There are a number of different technologies used for programmable logic by different manufacturers. The simplest devices, *programmable logic arrays* (PLAs), consist of two programmable planes, as shown in Figure 1.3. In reality, both planes implement a NOR function. The device is programmed by breaking connections. Most simple programmable devices use some form of floating gate technology. Each connection in the programmable planes consists of a MOS transistor. This transistor has two gates – one is connected to the input, while the second, between the first gate and the channel, floats. When the appropriate negative voltage

is applied to the device, the floating gate can have a large charge induced on it. This charge will exist indefinitely. If the charge exists on the floating gate, the device is disabled; if the charge is not there, the device acts as a normal transistor. The mechanisms for putting the charge on the device include *avalanche* or *hot electron injection* (EPROM) and *Fowler–Nordheim tunnelling* (EEPROM and Flash devices). These devices can be reprogrammed electrically.

PALs have a programmable AND plane and a fixed OR plane, and usually include registers, as shown in Figure 1.11. More complex PLDs (CPLDs) consist effectively of a number of PAL-like macrocells that can communicate through programmable interconnect, as shown in Figure 1.12

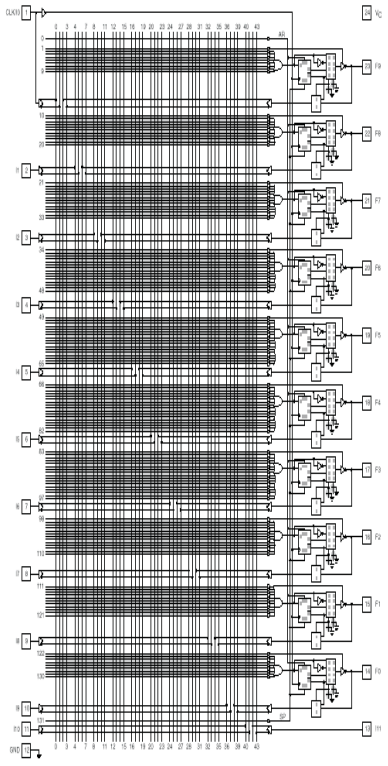


Figure 1.11: PAL structure (Lattice Semiconductor Corporation)

More complex still are *field programmable gate arrays* (FPGAs). FPGAs have a different type of architecture to CPLDs and are implemented in different technologies. Each FPGA vendor tends to have its own architecture – we will discuss two particular architectures here. Actel FPGAs consist of an array of combinational and sequential cells as shown in Figure 1.13. The combinational and sequential cells are shown in Figure

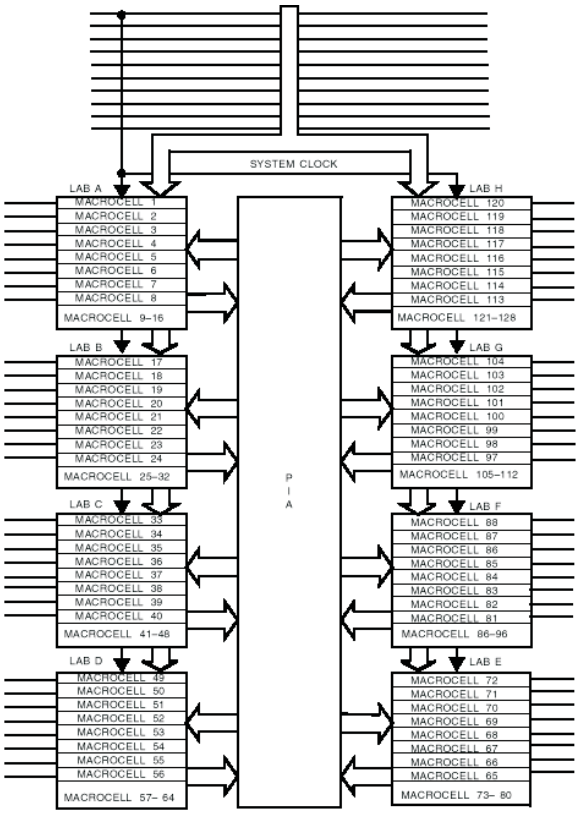



Figure 1.12: CPLD structure (Cypress Semiconductor Corporation)

1.14(a) and (b), respectively. Actel FPGAs are configured using an antifuse technology. In other words, a connection is normally open circuit, but the application of a suitably large voltage causes a short circuit to be formed. This configuration is not reversible, unlike EPROM or Flash technology. Once made, a short circuit has a resistance of around 50 $\Omega$, which limits the fan-out, as described below.

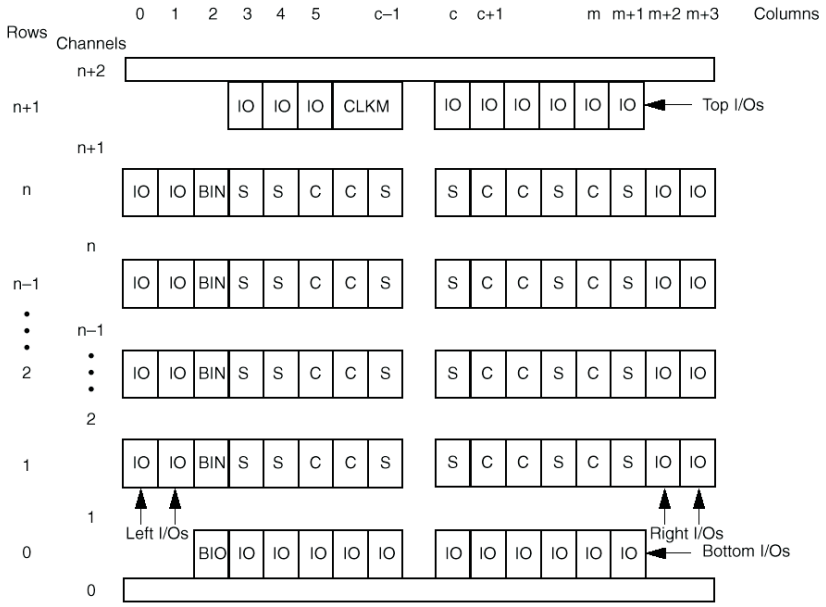



Figure 1.13: Actel FPGA (Actel Corporation)

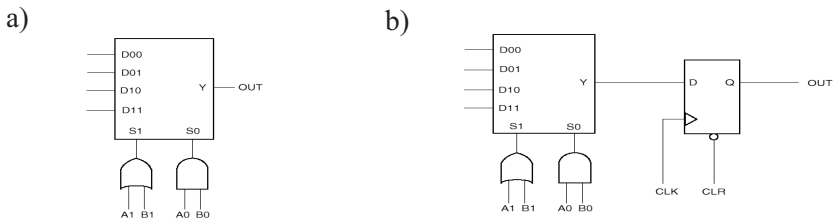



Figure 1.14: Actel FPGA cells. (a) Combinational; (b) sequential (Actel Corporation)

Xilinx FPGAs are implemented in static RAM technology. Unlike most programmable logic, the configuration is therefore volatile and must be restored each time power is applied to the circuit. Again, these FPGAs consist of arrays of logic cells. One such cell is shown in Figure 1.15 Each of these cells can be programmed to implement a range of combinational

and sequential functions. In addition to these logic cells, there exists programmable interconnect, including three-state buffers.

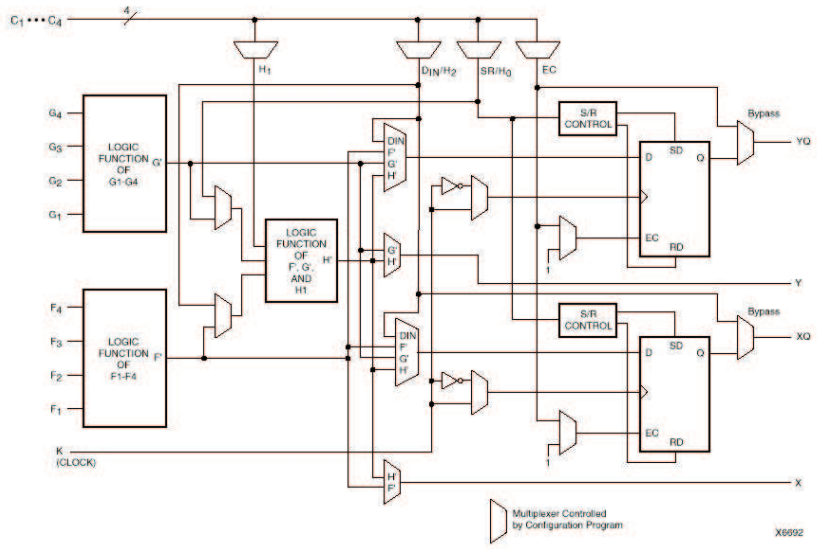



Figure 1.15: Xilinx FPGA logic cell (Xilinx, Inc.)

## 1.5 Electrical Properties

### 1.5.1 Noise Margins

Although it is common to speak of a logic 1 being, say, 2.5 V and a logic 0 being 0 V, in practice a range of voltages represent a logic state. A range of voltages may be recognised as a logic 1, and similarly one voltage from a particular range may be generated for a logic 1. Thus we can describe the logic states in terms of the voltages shown in Table 1.1.

The transfer characteristic for a CMOS inverter is illustrated in Fig. 1.16. The *noise margin* specifies how much noise, from electrical interference, can be added to a signal before a logic value is misinterpreted. From Table 1.1, it can be seen that the maximum voltage that a gate will generate to represent a logic 0 is 0.75 V. Any voltage up to 1.05 V is, however, recognised as a logic 0. Therefore there is a “spare” 0.3 V and any noise added to a logic 0 within this band will be accepted. Similarly, the difference between the minimum logic 1 voltage generated and the minimum recognised is 0.4 V. The noise margins are calculated as:

Table 1.1: Typical voltage levels for CMOS circuits with a supply voltage of 2.5V

| Parameter | Description | Typical CMOS Value |
|---|---|---|
| $V_{IHmax}$ | Maximum voltage recognised as a logic 1 | 2.5 V |
| $V_{IHmin}$ | Minimum voltage recognised as a logic 1 | 1.35 V |
| $V_{ILmax}$ | Maximum voltage recognised as a logic 0 | 1.05 V |
| $V_{ILmin}$ | Minimum voltage recognised as a logic 0 | 0.0 V |
| $V_{OHmax}$ | Maximum voltage generated as a logic 1 | 2.5 V |
| $V_{OHmin}$ | Minimum voltage generated as a logic 1 | 1.75 V |
| $V_{OLmax}$ | Maximum voltage generated as a logic 0 | 0.75 V |
| $V_{OLmin}$ | Minimum voltage generated as a logic 0 | 0.0 V |

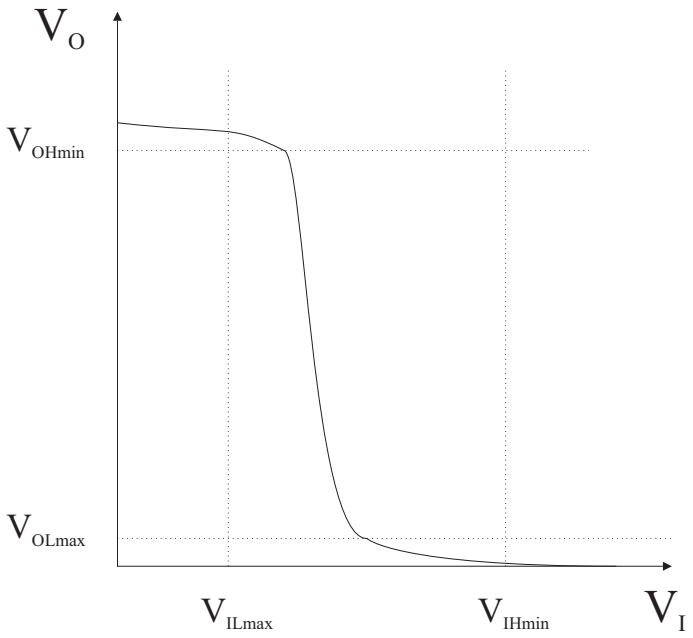



Figure 1.16: Transfer characteristic of CMOS inverter

$$\begin{aligned} NM_L &= V_{ILmax} - V_{OLmax} \\ NM_H &= V_{OHmin} - V_{IHmin} \end{aligned}$$

In general, the bigger the noise margin, the better.

### 1.5.2 Fan-Out

The fan-out of a gate is the number of other gates that it can drive. Depending on the technology, there are two ways to calculate the fan-out. If the input to a gate is resistive, as is the case with TTL or anti-fuse technology, the fan-out is calculated as the ratio of the current that a gate can output to the amount of current required to switch the input of a gate. For example, 74ALS series gates have the input and output currents specified in Table 1.2.

Table 1.2: Input and output currents for 74ALS series TTL gates

| | | |
|---|---|---|
| $I_{IHmax}$ | Maximum logic 1 input current | 20 $\mu$A |
| $I_{ILmax}$ | Maximum logic 0 input current | -100 $\mu$A |
| $I_{OHmax}$ | Maximum logic 1 output current | -400 $\mu$A |
| $I_{OLmax}$ | Maximum logic 0 output current | 8 mA |

Two fan-out figures can be calculated:

$$\begin{aligned} Logic1 fan-out &= \frac{I_{OH\max}}{I_{IH\max}} = \frac{400\mu A}{20\mu A} = 20 \\ Logic0 fan-out &= \frac{I_{OL\max}}{I_{IL\max}} = \frac{8mA}{100\mu A} = 80 \end{aligned}$$

Obviously the smaller of the two figures must be used.

CMOS gates draw almost no DC input current because there is no DC path between the gate of a transistor and the drain, source or substrate of the transistor. Therefore it would appear that the fan-out of CMOS circuits is very large. A different effect applies in this case. Because the gate and substrate of a CMOS gate form a capacitor, it takes a finite time to charge that capacitor and hence the fan-out is determined by how fast the circuit is required to switch. In addition, the interconnect between two gates has a capacitance. In high performance circuits, the effect of the interconnect can dominate that of the gates themselves. Obviously, the interconnect

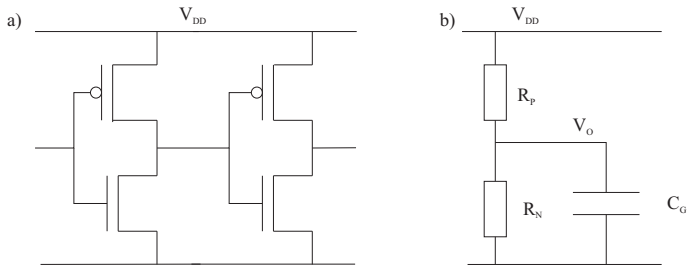



Figure 1.17: (a) CMOS inverter driving CMOS inverter; (b) equivalent circuit

characteristics cannot be estimated until the final layout of the circuit has been completed.

Figure 1.17(a) shows one CMOS inverter driving another. Figure 1.17(b) shows the equivalent circuit. If The first inverter switches from a logic 1 to a logic 0 at t=0, and if we assume that the resistance of NMOS transistor is significantly less than the resistance of the PMOS transistor, $V_O$ is given by:

$$V_O = V_{DD} e^{-t/R_N C_G}. \tag{1.1}$$

From Table 1.1, above, the minimum value of $V_O$ that would be recognised as a logic 1 is 1.35V and the maximum value of $V_O$ that would be recognised as a logic 0 is 1.05 V. For example, if $V_{DD}$ is 2.5 V, $R_N$ is 100 $\Omega$ and $C_G$ is 100 pF, we can see that the time taken for $V_O$ to drop from 1.35 V to 1.05 V is given by:

$$\begin{aligned} t &= -100 \times 100 \times 10^{-12} \times \ln \frac{1.05}{2.5} + 100 \times 100 \times 10^{-12} \times \ln \frac{1.35}{2.5} \\ &= 2.5 \text{ ns} \end{aligned}$$

.

If two inverters are driven, the capacitive load doubles, so the switching time doubles. Therefore, although a CMOS gate can drive an almost unlimited number of other gates at a fixed logic level, the fan-out is limited by the speed required of the circuit.

# Summary

Digital design is no longer a matter of taking small-scale integrated circuits and connecting them together. Programmable logic devices are an im-

portant alternative to full-custom integrated circuits. A number of different technologies exist for PLDs. These different technologies impose different constraints on the designer.

## Further Reading

The best source of information about different families of programmable logic is the manufacturers themselves. The entire data books are now available on the Web. These generally include electrical information, design advice and hints for programming using VHDL. In general, it is easy to guess the Web addresses, e.g. Xilinx are at http://www.xilinx.com; Actel are at http://www.actel.com.

## Exercises

1.1 Find examples of the following components in a 74LS/74HC data book (or on the Web):

- 4 bit Universal Shift Register
- 4 bit binary counter
- 8 bit priority encoder
- 4 bit binary adder
- 4 bit ALU

1.2 Find examples of PLDs, CPLDs and FPGAs from manufacturers' data books or from the Web. Compare the following factors:

- technologies
- performance
- cost
- programmability (e.g. use of SystemVerilog)
- testability

1.3 How is SystemVerilog used in the design process?

1.4 FPGAs are available in a number of sizes. Given that smaller FPGAs will be cheaper, what criteria would you use to estimate the required size of an FPGA, prior to detailed design?

1.5 A digital system may be implemented in a number of different technologies. List the main types available and comment on the advantages and disadvantages of each option. If you were asked to design a system with about 5000 gates and which was expected to sell about 10000 units, which hardware option would you choose and why?

# Chapter 2

# Combinational Logic Design

Digital design is based on the processing of binary signals. In this chapter, we will review the principles of Boolean algebra and the minimization of Boolean expressions. Hazards and basic numbering systems will also be discussed.

## 2.1 Boolean Algebra

### 2.1.1 Values

Digital design uses a two-value algebra. Signals can take one of two values that can be represented by

ON and OFF, or

TRUE and FALSE, or

1 and 0.

### 2.1.2 Operators

The algebra of two values, known as Boolean algebra, after George Boole (1815-1864), has five basic operators. In decreasing order of precedence (i.e. in the absence of parentheses, operations at the top of the list should be evaluated first) these are:

- NOT
- AND

- OR
- IMPLIES
- EQUIVALENCE

The last two operators are not normally used in digital design. These operators can be used to form expressions. For example:

$$\begin{aligned} A &= 1 \\ B &= C \text{ AND } 0 \\ F &= \overline{(A + B.C)} \\ Z &= \left(\bar{A} + B\right).\left(A + \bar{B}\right) \end{aligned}$$

The symbol "+" means "OR", "." means "AND" and the overbar, e.g. "$\bar{A}$" means "NOT $A$".

### 2.1.3 Truth Tables

The meaning of an operator or expression can be described by listing all the possible values of the variables in that expression, together with the value of the expression in a *truth table*. The truth tables for the three basic operators are given below.

| $A$ | NOT $A$ ($\bar{A}$) |
|---|---|
| 0 | 1 |
| 1 | 0 |

| $A$ | $B$ | $A$ AND $B$ ($A.B$) |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

In digital design, three further operators are commonly used, NAND (Not AND), NOR (Not OR) and XOR (eXclusive OR).

The XNOR ($\overline{A \oplus B}$) operator is also used occasionally. XNOR is the same as EQUIVALENCE.

| $A$ | $B$ | $A$ OR $B$ ($A + B$) |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| $A$ | $B$ | $A$ NAND $B$ $\overline{(A.B)}$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

| $A$ | $B$ | $A$ NOR $B$ $\overline{(A + B)}$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

| $A$ | $B$ | $A$ XOR $B$ ($A \oplus B$) |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

### 2.1.4 Rules of Boolean Algebra

There are a number of basic rules of Boolean Algebra that follow from the precedence of the operators.

1. Commutivity

$$\begin{aligned} A + B &= B + A \\ A.B &= B.A \end{aligned}$$

2. Associativity

$$\begin{aligned} A + (B + C) &= (A + B) + C \\ A.(B.C) &= (A.B).C \end{aligned}$$

3. Distributivity

$$A.(B + C) = A.B + A.C$$

In addition, some basic relationships can be observed from the truth tables above.

$$\bar{\bar{A}} = A$$

$$\begin{aligned} A.1 &= A & A + 0 &= A \\ A.0 &= 0 & A + 1 &= 1 \\ A.A &= A & A + A &= A \\ A.\bar{A} &= 0 & A + \bar{A} &= 1 \end{aligned}$$

The right hand column can be derived from the left-hand column, by applying the *Principle of duality*. The principle of duality states that if each AND is changed to an OR, each OR to an AND, each 1 to 0 and each 0 to 1, the value of the expression remains the same.

### 2.1.5 De Morgan's Law

There is a very important relationship that can be used to rewrite Boolean expressions in terms of NAND or NOR operations: de Morgan's Law. This is expressed as

$\overline{(A.B)} = \bar{A} + \bar{B}$ or $\overline{(A+B)} = \bar{A}.\bar{B}$

### 2.1.6 Shannon's expansion theorem

Shannon's expansion theorem can be used to manipulate Boolean expressions.

$$\begin{aligned} F(A,B,C,D,\ldots) &= A.F(1,B,C,D,\ldots) + \bar{A}.F(0,B,C,D,\ldots) \\ &= (A + F(0,B,C,D,\ldots)).\left(\bar{A} + F(1,B,C,D,\ldots)\right) \end{aligned}$$

$F(1,B,C,D,...)$ means that all instances of $A$ in $F$ are replaced by a logic 1.

## 2.2 Logic Gates

The basic symbols for one and two input logic gates are shown in Figure 2.1. Three and more inputs are shown by adding extra inputs (but note that there is no such thing as a three input XOR gate). The ANSI/IEEE symbols can be used instead of the traditional "spade"-shaped symbols, but are "not preferred" according to IEEE Standard 91-1984. As will be seen in the next chapter, IEEE notation is useful for describing complex logic blocks, but simple sketches are often clearer if done with the traditional symbols. A circle shows logic inversion. Note that there are two forms of the NAND and NOR gates. From de Morgan's law, it can be seen that the two forms are equivalent in each case.

In drawing circuit diagrams, it is desirable, for clarity, to choose the form of a logic gate that allows inverting circles to be joined. The circuits of Figure 2.2 are identical in function. If the circuit of Figure 2.2(a) is to be implemented using NAND gates, the diagram of Figure 2.2(b) may be preferable to that of Figure 2.2(c), because the function of the circuit is clearer.

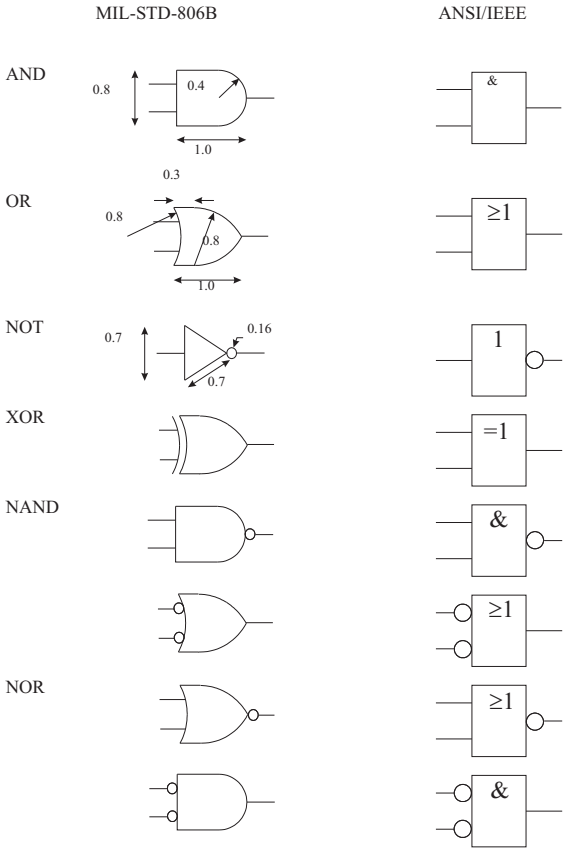



Figure 2.1: Logic symbols

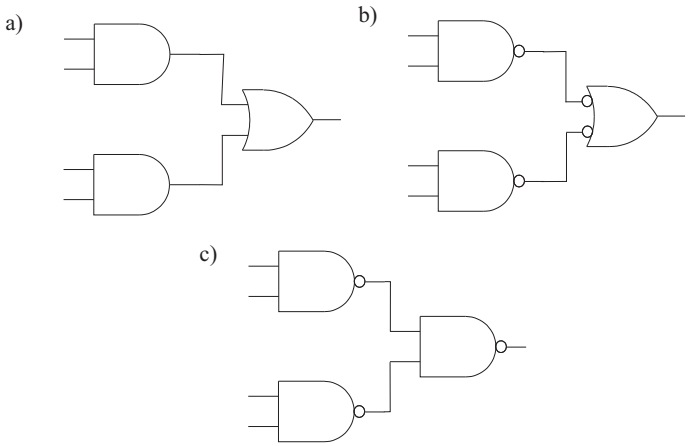



Figure 2.2: Equivalent circuit representations

## 2.3 Combinational Logic Design

The values of the output variables of combinational logic are dependent only on the input values and are independent of previous input values or states. Sequential logic, on the other hand, has outputs that depend on the previous states of the system. The design of sequential systems is described in later chapters.

The major design objective is usually to minimize the cost of the hardware needed to implement a logic function. That cost can usually be expressed in terms of the number of gates, although for technologies such as programmable logic, there are other limitations, such as the number of terms that may be implemented. Other design objectives may include testability (discussed in detail in Chapter 11) and reliability.

Before describing the logic design process, some terms have to be defined. In these definitions, it is assumed that we are designing a piece of combinational logic with a number of input variables and a single output.

A *minterm* is a Boolean AND function containing exactly one instance of each input variable or its inverse. A *maxterm* is a Boolean OR function with exactly one instance of each variable or its inverse. For a combinational logic circuit with $n$ input variables, there are $2^n$ possible minterms and $2^n$ possible maxterms. If the logic function is true at row $i$ of the standard truth table, that minterm exists and is designated by $m_i$. If the logic function is false at row $i$ of the standard truth table, that maxterm exists and is designated by $M_i$. For example, the following truth table defines a logic function. The final column shows the minterms and maxterms for the function.

| $A$ | $B$ | $C$ | $Z$ | |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | $m_0$ |
| 0 | 0 | 1 | 1 | $m_1$ |
| 0 | 1 | 0 | 0 | $M_2$ |
| 0 | 1 | 1 | 0 | $M_3$ |
| 1 | 0 | 0 | 0 | $M_4$ |
| 1 | 0 | 1 | 1 | $m_5$ |
| 1 | 1 | 0 | 0 | $M_6$ |
| 1 | 1 | 1 | 1 | $m_7$ |

The logic function may be described by the logic OR of its minterms:

$$Z = m_0 + m_1 + m_5 + m_7$$

A function expressed as a logical OR of distinct minterms is in *Sum of Products* form.

$$Z = \bar{A}.\bar{B}.\bar{C} + \bar{A}.\bar{B}.C + A.\bar{B}.C + A.B.C$$

Each variable is inverted if there is a corresponding 0 in the truth table and not inverted if there is a 1.

Similarly, the logic function may be described by the logical AND of its maxterms.

$$Z = M_2.M_3.M_4.M_6$$

A function expressed as a logical AND of distinct maxterms is in *Product of Sums* form.

$$Z = \left(A + \bar{B} + C\right).\left(A + \bar{B} + \bar{C}\right).\left(\bar{A} + B + C\right).\left(\bar{A} + \bar{B} + C\right)$$

Each variable is inverted if there is a corresponding 1 in the truth table and not inverted if there is a 0.

An *implicant* is a term that covers at least one true value and no false values of a function. For example, the function $Z = A + \bar{A}.\bar{B}$ is shown in the following truth table.

| $A$ | $B$ | $Z$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

The implicants of this function are $A.B$, $A$, $\bar{B}$, $\bar{A}.\bar{B}$, $A.\bar{B}$. The non-implicants are $\bar{A}$, $B$, $\bar{A}.B$.

A *prime implicant* is an implicant that covers one or more minterms of a function, such that the minterms are not all covered by another single implicant. In the example above, A, $\bar{B}$, are prime implicants. The other implicants are all covered by one of the prime implicants. An *essential prime implicant* is a prime implicant that covers an implicant, not covered by any other prime implicant. Thus, A, $\bar{B}$ are essential prime implicants.

### 2.3.1 Logic Minimization

The function of a combinational logic circuit can be described by one or more Boolean expressions. These expressions can be derived from the

specification of the system. It is very likely, however, that these expressions are not initially stated in their simplest form. Therefore, if these expressions were directly implemented as logic gates, the amount of hardware required would not be minimal. Therefore, we seek to simplify the Boolean expressions and hence minimize the number of gates needed. Another way of stating this is to say that we are trying to find the set of prime implicants of a function that is necessary to fully describe the function.

It is in principle possible to simplify Boolean expressions by applying the various rules of Boolean algebra described in section 2.1. It doesn't take long, however, to realize that this approach is slow and error prone. Other techniques have to be employed. The technique described here, *Karnaugh maps*, is a graphical method, although it is effectively limited to problems with 6 or fewer variables. The *Quine-McCluskey* algorithm is a tabular method, which is not limited in the number of variables and which is well suited to tackling problems with more than one output. Quine-McCluskey can be performed by hand, but it is generally less easy than the Karnaugh map method. It is better implemented as a computer program. Logic minimization belongs, however, to the *NP-complete* class of problems. This means that as the number of variables increases, the time to find a solution increases exponentially. Therefore, heuristic methods have been developed that find acceptable, but possibly less than optimal solutions. The *Espresso* program implements heuristic methods that reduce to the Quine-McCluskey algorithm for small problems. Espresso has been used in a number of logic synthesis systems. Therefore, the approach adopted here is to use Karnaugh maps for small problems with a single output and up to six inputs. In general, it makes sense to use an EDA program to solve larger problems.

The Karnaugh map (or K-map, for short) method generates a solution in sum-of-products or product-of-sums form. Such a solution can be implemented directly as two-level AND-OR or OR-AND logic (ignoring the cost of generating the inverse values of inputs). AND-OR logic is equivalent to NAND-NAND logic and OR-AND logic is equivalent to NOR-NOR logic. Sometimes, a cheaper (in terms of the number of gates) can be found by factorizing the two-level, minimized expression to generate more levels of logic – two-level minimization must be performed before any such factorization. Again, we shall assume that if such factorization is to be performed, it will be done using an EDA program, such as *SIS*.

### 2.3.2 Karnaugh Maps

A Karnaugh map is effectively another way to write a truth table. For example, the Karnaugh map of a general 2 input truth table is shown in Figure 2.3.

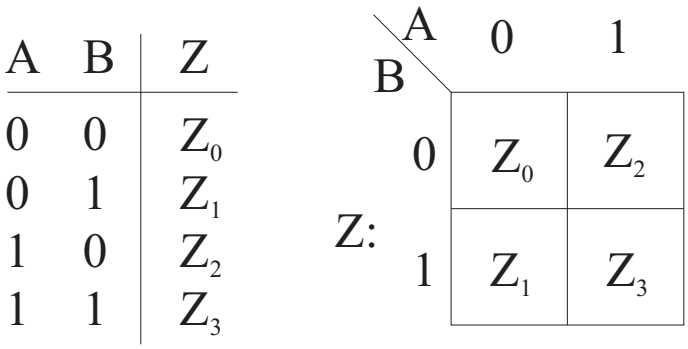



Figure 2.3: Two-input Karnaugh map

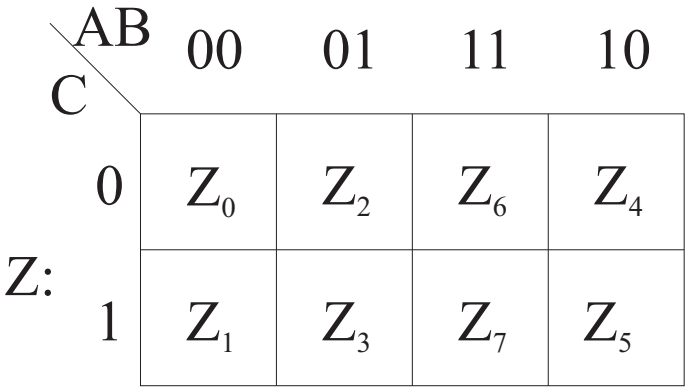



Figure 2.4: Three-input Karnaugh map

Similarly, 3 and 4 input Karnaugh maps are shown in Figures 2.4 and 2.5, respectively. Note that along the top edge of the three variable Karnaugh map and along both edges of the four variable map, only one variable changes at a time – the sequence is 00, 01, 11, 10, not the normal binary counting sequence. Hence, for example, the columns in which $A$ is true are adjacent. Therefore the left and right edges, and top and bottom in the 4 variable map, are also adjacent – $B$ is false in the leftmost and rightmost columns. The 3 variable map is therefore really a tube and the 4 variable map is a torus, as shown in Figure 2.5. Of course, the maps are drawn as squares for convenience!

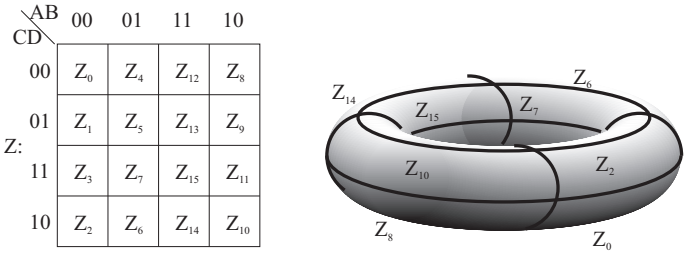



Figure 2.5: Four-input Karnaugh map

A five variable Karnaugh map is drawn as two four variable maps, one representing the truth table when the fifth variable, $E$, is false, the other when $E$ is true. Squares at the same co-ordinates on both maps are considered to be adjacent. Similarly, a six variable Karnaugh map is drawn as four four-variable maps corresponding to $\bar{E}.\bar{F}$, $\bar{E}.F$, $E.\bar{F}$ and $E.F$, respectively. For this to work, the Karnaugh maps have to be arranged themselves in the pattern as the entries in the two-variable map. Hence, squares at the same location in adjacent maps can be considered adjacent. In practice, therefore, it is not feasible to consider Karnaugh maps with more than six variables.

Implicants can be read from Karnaugh maps by circling groups of 1, 2, 4, 8, ... $2^n$ true values. For example, the function $Z = \bar{A}.\bar{B} + \bar{A}.B$ can be expressed as the following truth table.

| $A$ | $B$ | $Z$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

The corresponding Karnaugh map is shown in Figure 2.6. We can now circle the two adjacent 1s as shown. This grouping represents the function $Z = \bar{A}$, because it lies in the column $A = 0$, and because within the grouping, $B$ takes both 0 and 1 values and hence we don't care about its value. Therefore, by grouping patterns of 1s, logic functions can be minimized. Examples of 3 and 4 variable Karnaugh maps are shown in Figures 2.7 and 2.8. In both cases, by considering that the edges of the Karnaugh maps are adjacent, groupings can be made that include 1s at two, or four, edges.

The rules for reading *prime implicants* from a Karnaugh map are as follows.

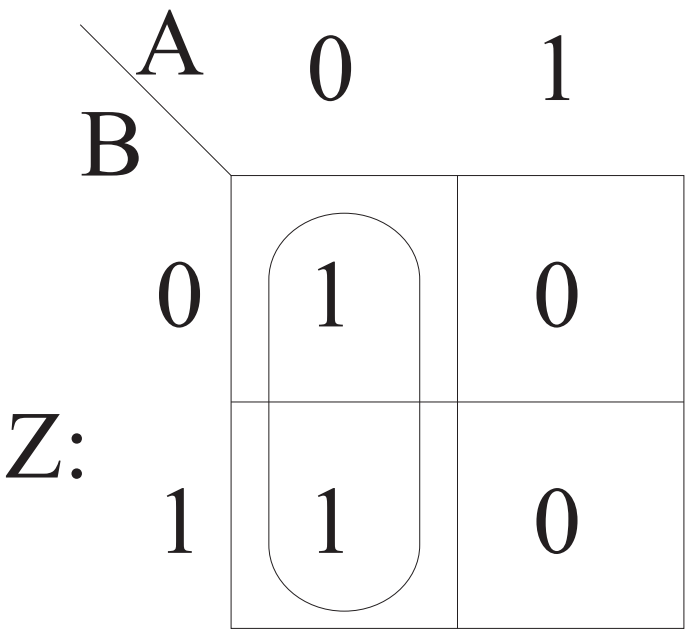



Figure 2.6: Karnaugh map for 2-input function

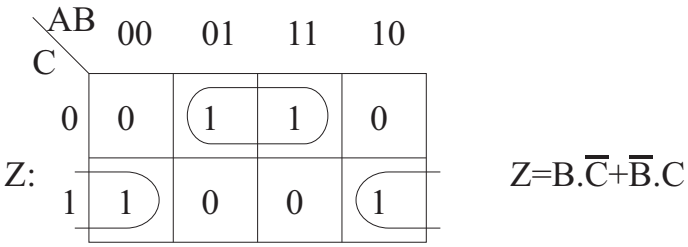



Figure 2.7: Groupings on three-input Karnaugh map

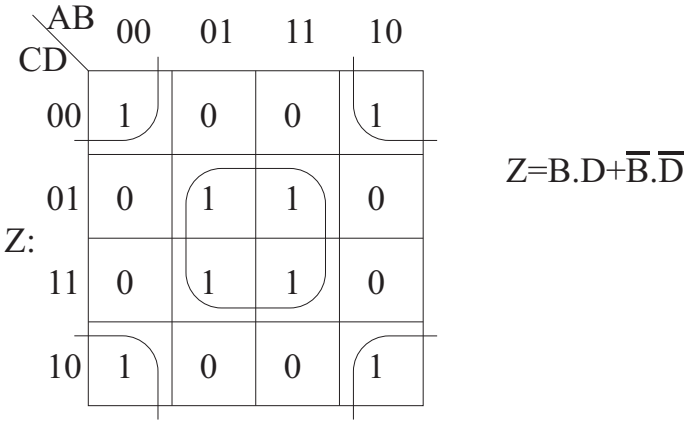



Figure 2.8: Groupings on four-input Karnaugh map

Circle the largest possible groups.
Avoid circles inside circles (see the definition of a prime implicant).

- Circle 1s and read the sum of products for $Z$.
- Circle 0s and read the sum of products for $\bar{Z}$.
- Circle 0s and read the product of sums for $Z$.
- Circle 1s and read the product of sums for $\bar{Z}$.

Diagonal pairs, as shown in Figure 2.9, correspond to XOR functions.

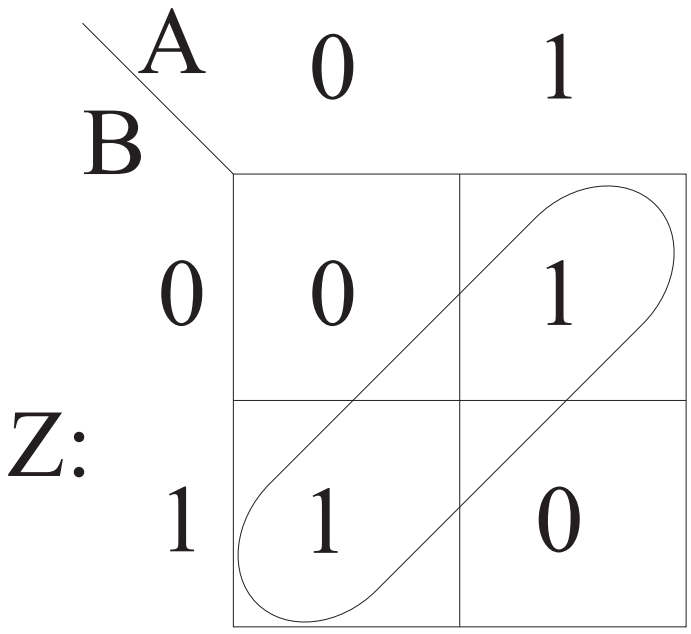



Figure 2.9: Exclusive OR grouping on Karnaugh map

The Karnaugh map of Figure 2.10 has three prime implicants circled. The function can be read as $Z = B.\bar{C}.D + \bar{A}.C.D + \bar{A}.B.D$. The vertical grouping, shown with a dashed line covers 1s covered by the other groupings. This grouping is therefore *redundant* and can be omitted. Hence, the function can be read as $Z = B.\bar{C}.D + \bar{A}.C.D$.

Assuming that all the prime implicants have been correctly identified, the minimal form of the function can be read by selecting all the essential prime implicants (i.e. those circles that circle 1s – or 0s – not circled by any other group), together with sufficient other prime implicants needed to cover all the 1s (or 0s). Redundant groupings can be ignored, but under some circumstances it may be desirable to include them.

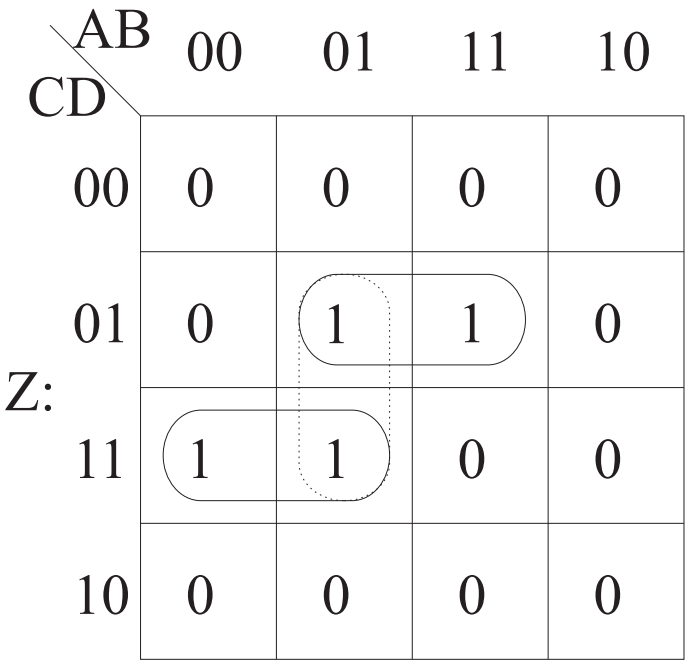



Figure 2.10: Redundant grouping on Karnaugh map

| A | B | Z |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | - |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

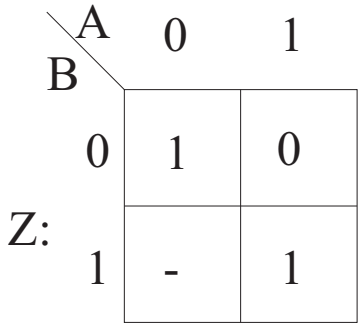



Figure 2.11: Redundant grouping on Karnaugh map

Incompletely specified functions have “don’t cares” in the truth tables. These don’t cares correspond to input combinations that will not (or should not) occur. For example, consider the truth table of Figure 2.11.

The don’t care entries can be included or excluded from groups as convenient, in order to get the largest possible groupings, and hence the smallest number of implicants. In the example, we could treat the don’t care as a 0 and read $Z = \bar{A}.\bar{B} + A.B$, or treat the don’t care as a 1 and read $Z = \bar{A} + B$.

## 2.4 Timing

The previous section dealt with minimizing Boolean expressions. The minimized Boolean expressions can then be directly implemented as networks of gates or on programmable logic. All gates have a finite delay between a change at an input and the change at an output. If gates are used, therefore, different paths may exist in the network, with different delays. This may cause problems.

To understand the difficulties, it is helpful to draw a *Timing Diagram*. This is a diagram of the input and output waveforms as a function of time. For example, Figure 2.12 shows the timing diagram for an inverter. Note the stylised (finite) rise and fall times. An arrow shows causality, i.e. the fact that the change in the output results from a change in the input.

A more complex circuit would implement the function

$$Z = A.C + B.\bar{C}$$

The value of $\bar{C}$ is generated from $C$ by an inverter. A possible implementation of this function is therefore given in Figure 2.13. In practice, the delay through each gate and through each type of gate would be slightly different. For simplicity, however, let us assume that the delay through each gate is 1 unit of time. To start with, let $A = 1$, $B = 1$. The output, $Z$, should be at 1 irrespective of the value of $C$. Let us see, by way of the timing diagram in Figure 2.14, what happens when $C$ changes from 1 to 0. One unit of time after $C$ changes $\bar{C}$ and $D$ change to 1. In turn, these changes cause $E$ and $Z$ to change to 0 another unit of time later. Finally, the change in $E$ causes $Z$ to change back to 1 a further unit of time later. This change in $Z$ from 1 to 0 and back to 1 is known as a *hazard*. A hazard occurs as a result of delays in a circuit.

Figure 2.15 shows the different types of hazard that can occur. The hazard in the circuit of Figure 2.13 is a Static 1 hazard. Static 1 hazards

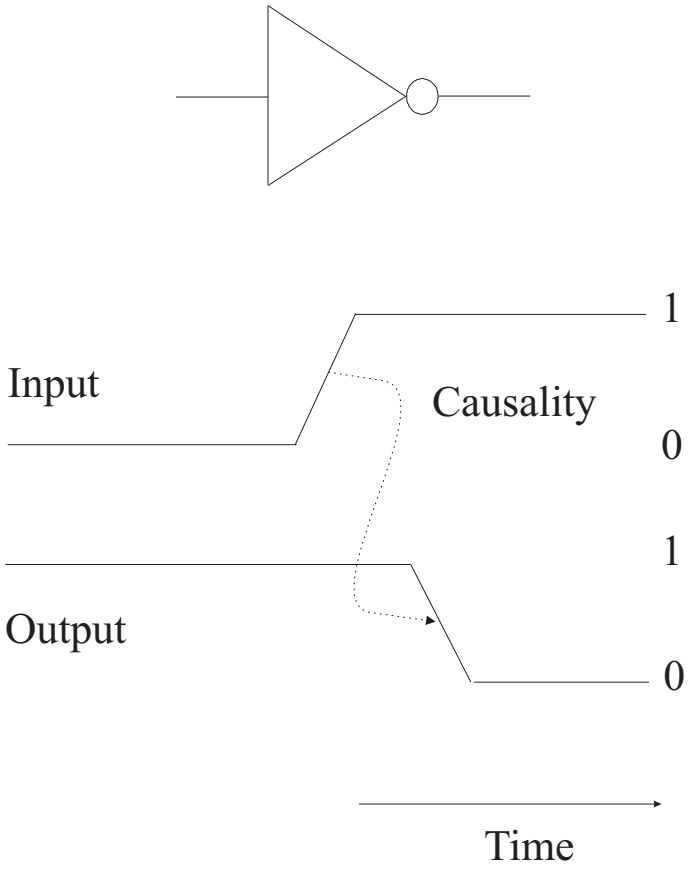



Figure 2.12: Timing diagram for inverter

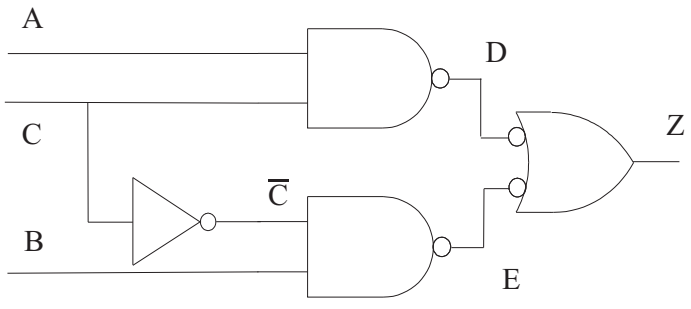



Figure 2.13: Circuit with Static 1 hazard

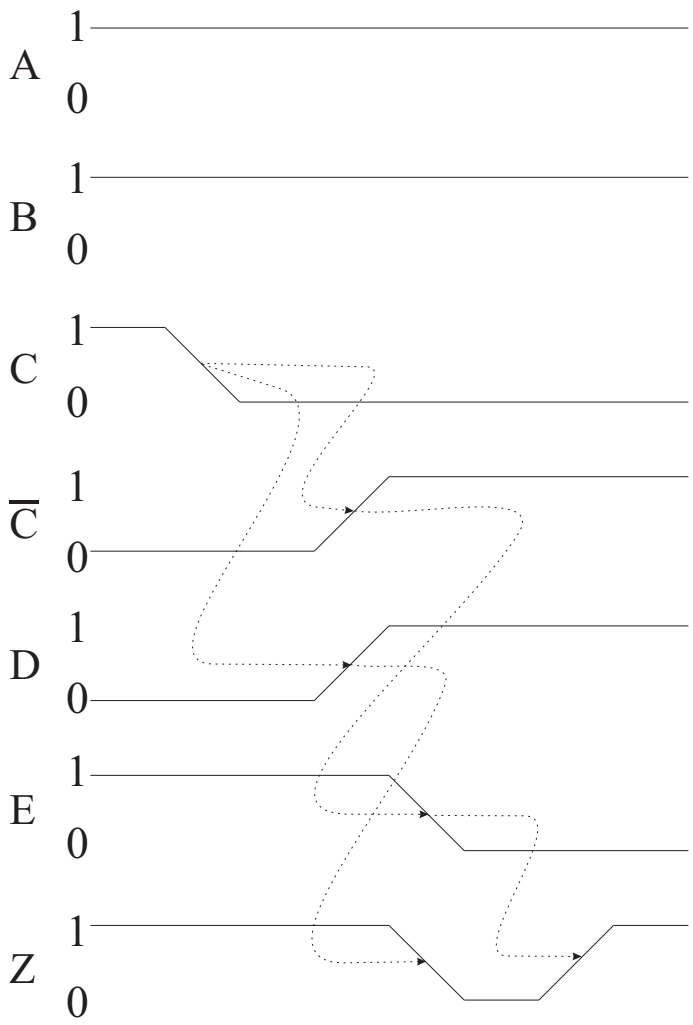



Figure 2.14: Timing diagram for circuit of Figure 2.13

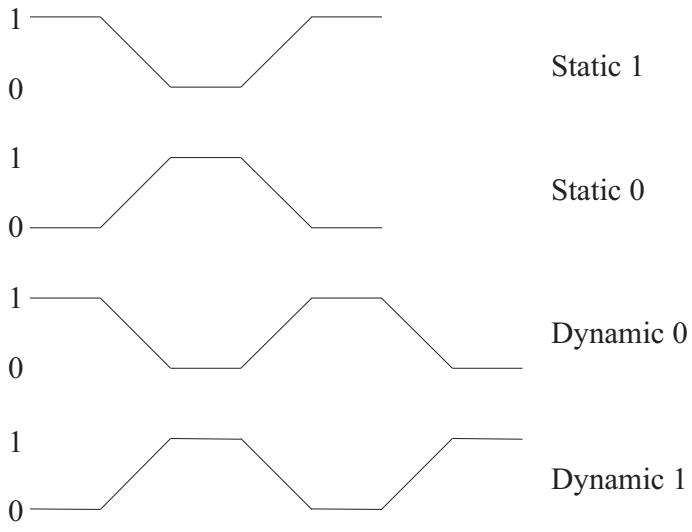



Figure 2.15: Types of hazard

can occur in AND-OR or NAND-NAND logic. Static 0 hazards can occur in OR-AND or NOR-NOR logic. Dynamic hazards do not occur in two-level circuits. They require three or more unequal signal paths. Dynamic hazards are often caused by poor factorization in multi-level minimization.

Static hazards, on the other hand, can be avoided by designing with redundant logic. For example, the Karnaugh map of the circuit function of Figure 2.13 is shown in Figure 2.16. The redundant prime implicant is shown as a dashed circle. The redundant gate corresponding to this prime implicant can be introduced to eliminate the hazard. The circuit function is therefore

$$Z = A.C + B.\bar{C} + A.B$$

The circuit is shown in Figure 2.17. Now, $F$ is independent of $C$. If $A = B = 1$, $F = 0$. $F$ stays at 0 while $C$ changes, therefore $Z$ stays at 1.

## 2.5 Number Codes

Digital signals are either control signals of some kind, or information. In general, information takes the form of numbers or characters. These numbers and characters have to be coded in a form suitable for storage and manipulation by digital hardware. Thus, one integer or one character may be represented by a set of bits. From the point of view of a computer or other digital system, no one system of coding is better than another. There

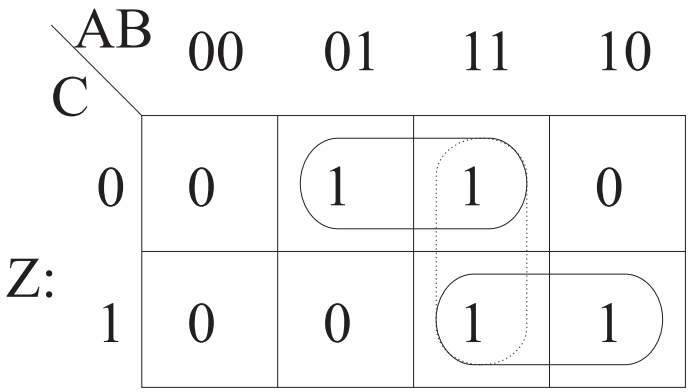



Figure 2.16: Redundant term on Karnaugh map

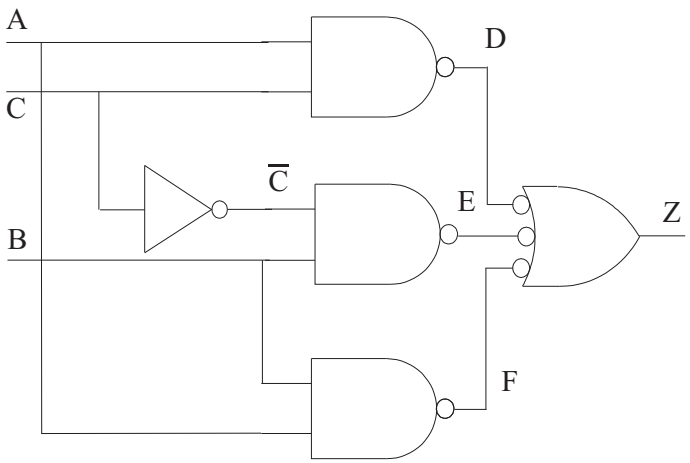



Figure 2.17: Hazard-free circuit

do, however, need to be standards, so that different systems can communicate. The standards that have emerged are generally also designed such that a human being can interpret the data if necessary.

### 2.5.1 Integers

The simplest form of coding is that of positive integers. For example, a set of three bits would allow us to represent the decimal integers 0 to 7. In base 2 arithmetic, $000_2$ represents $0_{10}$, $011_2$ represents $3_{10}$ and $111_2$ represents $7_{10}$. As with decimal notation, the most significant bit is on the left.

For the benefit of human beings, strings of bits may be grouped into sets of three or four and written using *octal* (base 8) or *hexadecimal* (base 16) notation. For example, $66_8$ is equal to $110\ 110_2$ or $54_{10}$. For hexadecimal notation, the letters A to F represent the decimal numbers 10 to 15. For example, $EDA_{16}$ is $1110\ 1101\ 1010_2$ or $7332_8$ or $3802_{10}$.

The simple translation of a decimal number into bits is sufficient for zero and positive integers. Negative integers require additional information. The simplest approach is to set aside one bit as a sign bit. Therefore, $0\ 110_2$ might represent $+6_{10}$, while $1\ 110_2$ would represent $-6_{10}$. While this makes translation between binary and decimal numbers simple, the arithmetic operations of addition and subtraction require that the sign bits be checked before an operation can be performed on two numbers. It is common, therefore, to use a different notation for signed integers: *two's complement*. The principle of two's complement notation is that the code for $-b$, where $b$ is a binary number represented using n bits, is the code given by $2^n - b$. For example, $-6_{10}$ is represented by $10000_2 - 0110_2$, which is $1010_2$. The same result is obtained by inverting all the bits and adding 1: $-6_{10}$ is $1001_2 + 1 = 1010_2$.

The advantage of two's complement notation is that addition and subtraction may be performed using exactly the same hardware as for unsigned arithmetic; no sign checking is needed. The major disadvantage is that multiplication and division become much more complicated. Booth's algorithm, described in Section 6.7, is a technique for multiplying two's complement numbers.

### 2.5.2 Fixed Point Numbers

For many applications, non-integer data needs to be stored and manipulated. The binary representation of a *fixed-point* number is exactly the

same as for an integer number, except that there is an implicit "decimal" point. For example, 6.25 is equal to $2^2 + 2^1 + 2^{-2}$ or $110.01_2$. Instead of representing the point, the number $11001_2$ ($25_{10}$) is stored, with the implicit knowledge that it and the results of any operations involving it have to be divided by $2^2$ to obtain the true value. Notice that all operations, including two's complement representations are the same as for integer numbers.

### 2.5.3 Floating Point Numbers

The number of bits that have been allocated to represent fractions limits the range of fixed point numbers. *Floating point* numbers allow a much wider range of accuracy. In general, floating point operations are only performed using specialized hardware, because they are very computationally expensive. A typical *single precision* floating point number has 32 bits, of which 1 is the sign bit ($s$), 8 are the exponent ($e$) in two's complement form, and the remaining 23 are the mantissa ($m$), such that a decimal number is represented as

$$(-1)^s \times 1.m \times 2^e$$

The IEEE standard, 754-1985, defines formats for 32, 64 and 128 bit floating point numbers, with special patterns for $\pm\infty$ and the results of invalid operations, such as $\sqrt{-1}$.

### 2.5.4 Alphanumeric Characters

Characters are commonly represented by seven or eight bits. The ASCII code is widely used. Seven bits allow the basic Latin alphabet in upper and lower cases, together with various punctuation symbols and control codes to be represented. For example, the letter A is represented by 1000001. For accented characters eight bit codes are commonly used. Manipulation of text is normally performed using general purpose computers rather than specialized digital hardware.

### 2.5.5 Gray Codes

In the normal binary counting sequence, the transition from 0111 ($7_{10}$) to 1000 ($8_{10}$) causes three bits to change. In some circumstances, it may be undesirable that several bits should change at once, because the bit changes may not occur at exactly the same time. The intermediate values

might generate spurious warnings. A *Gray* code is one in which only one bit changes at a time. For example a three bit Gray code would count through the following sequence (other Gray codes can also be derived):

000

001

011

010

110

111

101

100

Note that the sequence of bits on a K-map is a Gray code. Another application of Gray codes is as a position encoder on a rotating shaft, as shown in Figure 2.18.

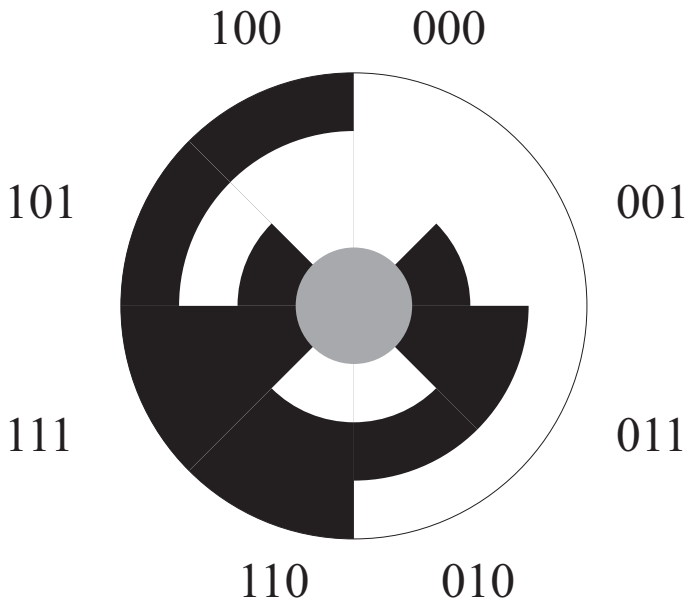



Figure 2.18: Gray code as shaft encoder

### 2.5.6 Parity Bits

When data is transmitted either by wire or by using radio communications, there is always the possibility that noise may cause a bit to be misinterpreted. At the very least it is desirable to know that an error has occurred and it may be desirable to transmit sufficient information to allow any error to be corrected.

The simplest form of error detection is to use a parity bit with each word of data. For each eight bits of data, a ninth bit is sent that is 0 if there is an even number of ones in the data word (even parity) or 1 otherwise. Alternatively odd parity can be used; in which case the parity bit is inverted. This is sufficient if the chances of an error occurring are low. We cannot tell which bit is in error, but knowing that an error has occurred means that the data can be transmitted again. Unfortunately, if two errors occur, the parity bit might appear to be correct. A single error can be corrected by using a two-dimensional parity scheme, in which every ninth word is itself a set of parity bits, as shown below. If a single error occurs, both the row parity and column parity will be incorrect, allowing the erroneous bit to be identified and corrected. Certain multiple errors are also detectable and correctable.

| | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Parity |
|---|---|---|---|---|---|---|---|---|---|
| Word 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| Word 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| Word 2 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| Word 3 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| Word 4 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| Word 5 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| Word 6 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| Word 7 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| Parity | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

By using a greater number of parity bits, each derived from part of the word, multiple errors can be detected and corrected. The simplest forms of such codes were derived by Hamming in 1948. Better codes were derived by Reed and Solomon in 1960.

## Summary

Digital design is based on Boolean algebra. The rules of Boolean algebra allow logical expressions to be simplified. The basic logical operators can be implemented as digital building blocks – gates. Graphical methods, Karnaugh maps, are a suitable tool for finding the minimal forms of Boolean expressions with fewer than 6 variables. Larger problems can be tackled with computer-based methods. Gates have delays, which means that non-minimal forms of Boolean expressions may be needed to prevent timing problems, known as hazards. Data can be represented using sets of bits. Different types of data can be encoded to allow manipulation. Error detecting codes are used when data is transmitted over radio or other networks.

## Further Reading

The principles of Boolean algebra and Boolean minimization are covered in many books on digital design. Recommended are those by Wakerly, Mano and Hill and Peterson. De Micheli describes the Espresso algorithm, which sits at the heart of many logic optimization software packages. Espresso may be downloaded from http://www-cad.eecs.berkeley.edu/ .

Error detection and correction codes are widely used in communications systems. Descriptions of these codes can be found in, for example, Hamming.

## Exercises

2.1 Derive Boolean expressions for the circuits of Figure 2.19; use truth tables to discover if they are equivalent.

2.2 Minimise

(a) $Z = m_0 + m_1 + m_2 + m_5 + m_7 + m_8 + m_{10} + m_{14} + m_{15}$

(b) $Z = m_3 + m_4 + m_5 + m_7 + m_9 + m_{13} + m_{14} + m_{15}$

2.3 Describe two ways of representing negative binary numbers. What are the advantages and disadvantages of each method?

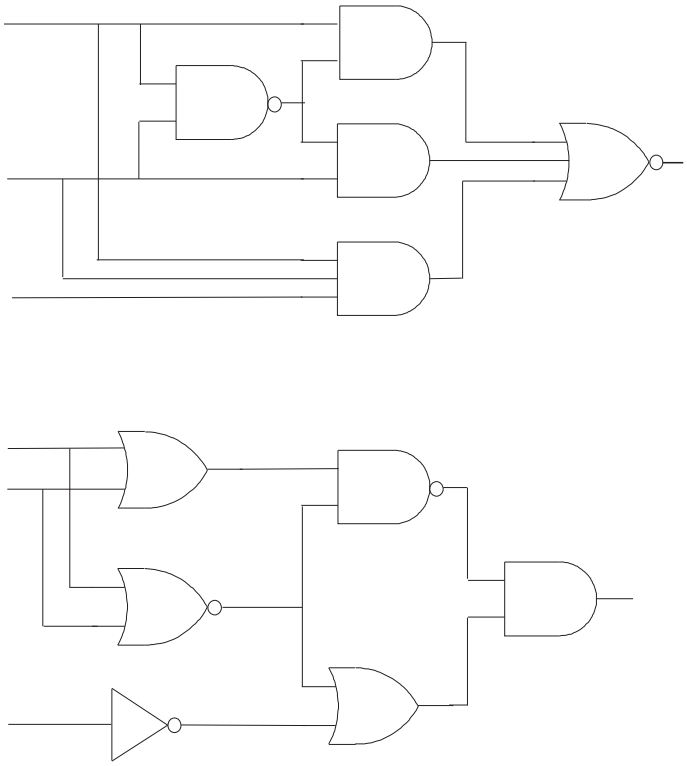


Figure 2.19: Circuits for Exercise 2.1

2.4 A floating-point decimal number may be represented as:

$$(-1)^s \times 1.m \times 2^{e-127}$$

Explain what the binary numbers $s$, $m$ and $e$ represent. How many bits would typically be used for $s$, $m$ and $e$ in a single precision floating-point number?

# Chapter 3

# Netlists and Structural SystemVerilog

In the last section we saw some of the basic structures of CMOS ASIC and FPGA circuits. An HDL should, at the very least, be able to model the functionality of such structures and to describe how such structures are connected together. In this section, we will see how networks of basic elements can be described in SystemVerilog. Networks of the basic elements can be simulated and can be used as inputs to synthesis or layout tools. This does not really justify the existence of full HDL, that functionality will be explained in later chapters. We will however note in passing that in SystemVerilog we can describe circuits at a very low level. This can be useful for verification. We will wait until the next chapter to show how stimuli can be generated for verifying these models.

## 3.1 Basic Gate Models

Built into SystemVerilog are a number of low-level gate primitives. These include:

**and**, **or**, **nand**, **nor**, **xor**, **xnor**, **not**, **buf**.

These are keywords, which will be shown in bold font. SystemVerilog is case sensitive; keywords are always lower case. The meaning of the gates is probably self-evident: **xnor** is an exclusive NOR gate (in other words the output is true if the inputs are equal); **buf** is a non-inverting buffer. There are several other primitives, some of which we will discuss later.

To distinguish one instance of a gate from another, a label follows the gate primitive name – see below.

A gate is connected to one or more nets. These nets are listed in parentheses. The convention is that the output comes first and is followed by the input(s). A NAND gate with inputs a and b and output y, might appear in a piece of SystemVerilog as:

**nand** g1 (y, a, b);

where g1 is the label for that gate. Note the semicolon (;) at the end of the instance. Whitespace is not important, so this description could be split over two or more lines, or formatted to line up with other statements.

It is possible to have more than one gate instance declared at the same time:

**nand** g1 (y, a, b), g2 (w, c, d);

This describes two gates (g1 and g2). This can be split over two or more lines. While legal, this is not really recommended because it can make circuit descriptions difficult to read.

There are two further pieces of information that can be declared with a gate: the signal strength and the delay. In CMOS circuits, signal strength is not usually modelled – we will return to this later. We will discuss delay modelling after we have looked at the structure of a netlist description.

## 3.2 A Simple Netlist

A netlist is a list of nets (!) and the gates (or other elements) that connect them. Let us see how a simple combinational logic circuit can be described.

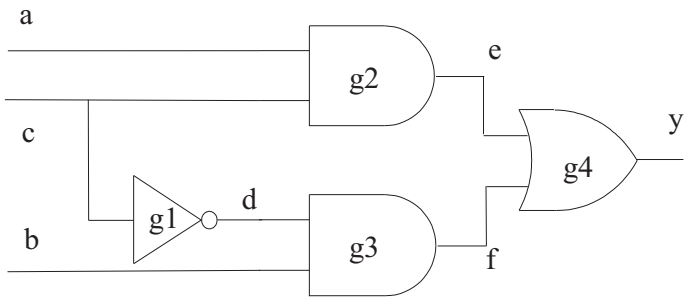



Figure 3.1: Simple Combinational Circuit

Figure 3.1 shows a simple combinational logic circuit, with one output (y), three inputs (a, b, c) and three internal nodes (d, e, f). The gates and inverter are labelled (g1, g2, g3, g4). This is a Verilog description of this circuit:

```
module ex1 (output wire y;
                  input wire a, b, c);
  wire d, e, f;
  not g1 (d, c);
  and g2 (e, a, c);
  and g3 (f, d, b);
  or g4 (y, e, f);
endmodule
```

In SystemVerilog, the description of a circuit component begins with the keyword **module**, followed by a name. The inputs and outputs are then listed. We will follow the convention used in gate primitives, and list the output(s) before the input(s). It is also possible to have bidirectional connections, declared with the **inout** keyword. All the inputs and outputs are declared to be *nets* with the keyword **wire**. In fact, this keyword is not needed, but it is *strongly recommended*, however, that you declare all nets using the **wire** keyword (or the **logic** keyword, as we will see in later chapters).

The second line declares the internal nodes of the circuit. Again, the declaration is not strictly needed because once a net is used in a gate description, it is automatically declared. Again, it is recommended that you declare all nets for clarity.

The next four lines are the gate declarations, which we have already discussed. Finally, the end of the description is marked by the keyword endmodule. Note there is no semicolon!

## 3.3 Delays

While it is possible to design circuits at the gate level, this does make the use of an HDL like SystemVerilog a little pointless. Indeed, it could be argued that writing a netlist by hand is a waste of time. If you are working at that level, you will probably have had to sketch the circuit diagram. So why not use a schematic capture program to draw the circuit, and then generate the netlist from the schematic automatically?

This does not mean you will never encounter a netlist. Another way of generating a netlist is from a synthesis tool or by extracting the circuit *after* physical layout of the circuit. In both these cases, you will probably wish to verify your design by simulation. You can simply verify the logical functionality of the circuit, but it is often more important to verify that the circuit will work correctly at the normal operating speed. To verify the timing, the sim-

ulation model must include timing information. The simplest way to include this information is to model the delay through each gate. For example, a delay of 10 units through a NAND gate would be written as:

**nand** #10 g1 (y, a, b);

The hash symbol (#) is used to denote a parameter. We will see further examples of parameters in later chapters. Notice that the delay parameter is placed between the type of gate (**nand**) and the name of the instance (g1).

In the example above, there is one delay parameter. In the case of a NAND gate, the output is at logic 1 if either or both inputs is at logic 0. Therefore, the output will only go to logic 0 after the second of the two inputs has gone to 1. This change will be delayed by 10 time units:

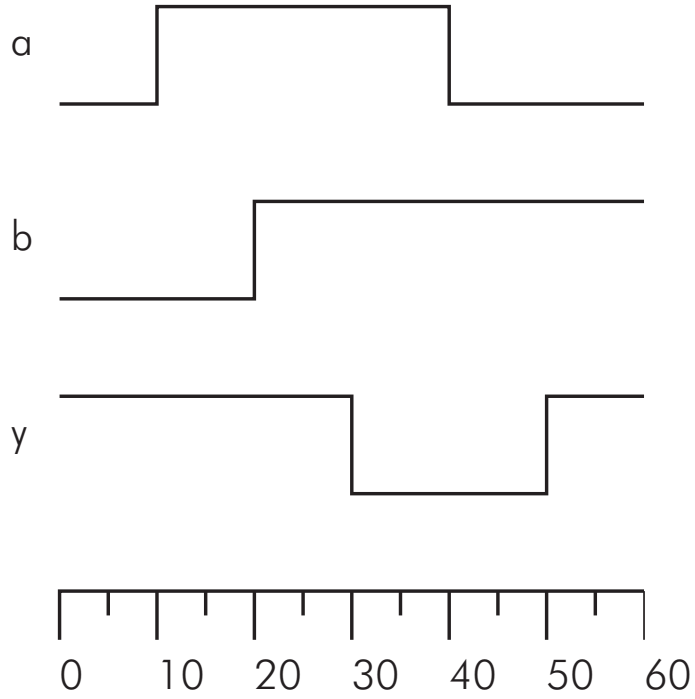



Figure 3.2: NAND function with delay

In Figure 3.2, signal b goes to 1 at time 20; signal a goes back to 0 at time 40. Therefore the pulse on y is 20 units wide, delayed by 10 units.

Suppose that a changes back to 0 at time 35. This would suggest that a pulse 5 units wide would appear at y, again delayed by 10 units. In fact, the delay has a second meaning: any pulse less than 10 units wide is suppressed:

This is known as an *inertial delay*. Hence, a pulse is supressed by

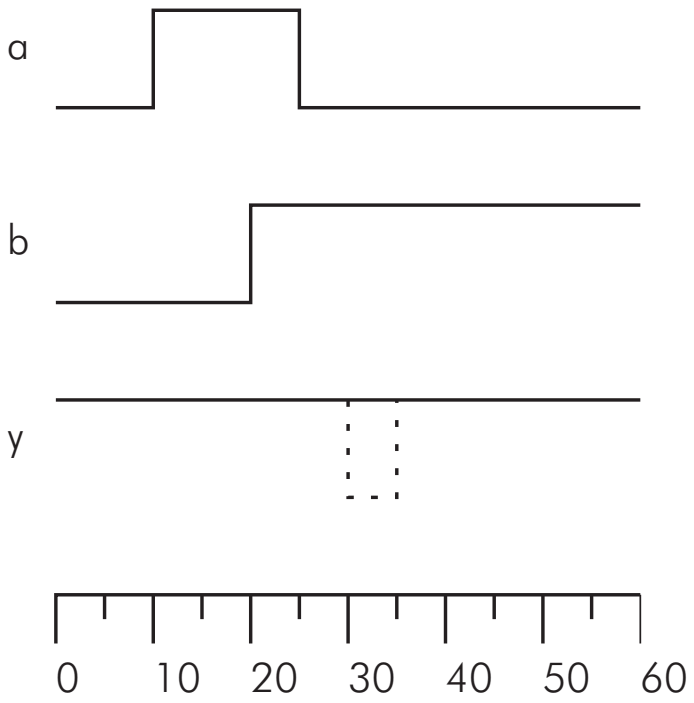



Figure 3.3: NAND function with inertial cancellation

*inertial cancellation*.

This delay model assumes that the delay through a gate is the same for a 0 to 1 transition on the output as for a 1 to 0 transition. This assumption is probably valid for CMOS technologies, but may not be true in other cases. If the 0 to 1 and 1 to 0 delays differ, the two values may be specified. For example,

**nand** #(10, 12) g1 (y, a, b);

describes a NAND gate that has a 10 unit delay when the output changes to 1 (rise delay) and a 12 unit delay when the output changes to 0 (fall delay). It is also possible to specify a third delay for the case when the output changes to a high-impedance state.

We can take delay modelling one step further to describe uncertainty. When a circuit is extracted from a silicon layout, it is not possible to exactly predict the delay through each gate or between each gate, because of process variations. It is reasonable, however, to expect that the minimum, typical and maximum delays through a gate can the minimum, typical and maximum delays respectively, e.g.

**nand** #(8:10:12, 10:12:14) g1 (y, a, b);

describes a NAND gate that has a minimum rise delay of 8 units, a typical

rise delay of 10 units and a maximum rise delay of 12 units. Similarly, the fall delay has three values. A simulation can therefore be performed using the minimum, typical or maximum delays for all gates. In principle, the functionality of the circuit can therefore be verified under extremes of fabrication.

## 3.4 Logic Values

In the preceding description, we have mentioned logic values and referred briefly to a high impedance state. SystemVerilog allows wires to take 4 possible values: 0, 1, x (unknown) and z (high impedance). In general, logic gates are designed to generate 0 or 1 at the outputs. x usually indicates some kind of anomalous situation – perhaps an uninitialised flip-flop or a wire that is being driven to two different values by two gates simultaneously.

The high impedance state, z, is used to model the output of three-state buffers. The purpose of three-state buffers is to allow the outputs of gates to be connected together to form buses, for example. We have already noted that the x state is generated when the outputs of two gates are connected together. We would expect, however, that a 1 and a z (or a 0 and a z) driving the same wire would resolve to a 1 (or a 0). Clearly, therefore, not all logic values are equal.

The unknown and high-impedance states can be written as lower case (x and z) or upper case (X and Z) characters. The question mark (?) can be used as an alternative to the high-impedance state (but note that “?” has a different meaning in a UDP table, section 2.8, below).

## 3.5 Logic Strengths

A z state is assumed to be “weaker” than a 0 or a 1. SystemVerilog has eight strengths, in descending order: supply, strong, pull, large, weak, medium, small and high impedance. (Beware, however, not every SystemVerilog simulator supports strength modelling, and even those that do may not support all strengths.) Supply, strong, pull and weak may be used to model gate outputs; large, medium and small are used to model three-state outputs. The table below shows the full range of SystemVerilog strength names, together with equivalent numerical values.

If two signals of unequal strength drive a wire, the wire takes the value of the stronger signal. The rules for combining signal strengths and re-

| Strength Name | Value |
|---|---|
| supply0 | 7 |
| strong0 | 6 |
| pull0 | 5 |
| large0 | 4 |
| weak0 | 3 |
| medium0 | 2 |
| small0 | 1 |
| highz0 | 0 |
| highz1 | 0 |
| small1 | 1 |
| medium1 | 2 |
| weak1 | 3 |
| large1 | 4 |
| pull1 | 5 |
| strong1 | 6 |
| supply1 | 7 |

solving ambiguities are, in fact, a lot more complex than this. For CMOS gates and for higher levels of modelling, we can ignore signal strengths completely. Indeed, if we attempt to specify strengths for the purposes of synthesis, they will be ignored. We will, however, illustrate the use of strength modelling, and introduce some further primitives, with one example.

Figure 3.4 shows the transistor-level model of an XOR gate. This circuit uses 6 transistors, significantly fewer than would be needed if the gate were built up from NAND gates and inverters. The operation of the gate relies on the behaviour of pass transistor logic. P2, N2, P3 and N3 are all bidirectional transistors (as, of course, are all transistors). SystemVerilog contains primitives to model both bidirectional and unidirectional pass transistors.

Bidirectional transistors can be modelled by **tranif0** and **tranif1** (“transmit if control is 0” or “1”, respectively). Unidirectional MOS transistors are modelled by **pmos** and **nmos**. Four further primitives, **rtranif0**, **rtranif1**, **rpmos** and **rnmos** reduce the strength of any signal passing through them.

If the XOR circuit is modelled by bidirectional transistors, it is difficult for the simulator to resolve the values of each wire in the circuit[1]. It is

[1]In fact, I have seen two simulators crash without warning when presented with this

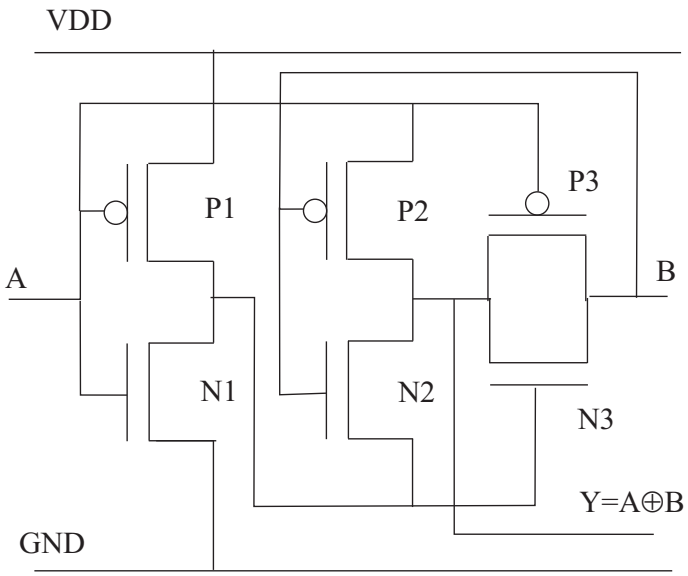



Figure 3.4: Pass transistor implementation of XOR gate

necessary to use unidirectional transistors. If simulated using pmos and nmos, the output is as shown in Figure 3.5. The strengths are shown (St1, St0, StX are short for strong 1, strong 0 and strong x). It can be seen from this figure that the XOR behaviour is not correctly modelled.

The netlist for the circuit using **rpmos** and **rnmos** is shown below. Note that the supplies are modelled using pullup and pulldown elements. The supplies are given the strengths **strong1** and **strong0**. (At least one common SystemVerilog simulator does not recognise **supply1** or **supply0**, which would be the obvious strengths to use.)

```
module swxor (output wire y, input wire a, b);
  wire nota, vdd, gnd;

  pullup (strong1, strong0) (vdd);
  pulldown (strong1, strong0) (gnd);

  rpmos p1 (nota, vdd, a);
  rnmos n1 (nota, gnd, a);

  rpmos p2 (y, a, b);
  rnmos n2 (y, nota, b);
```

circuit modelled with bidirectional transistors.

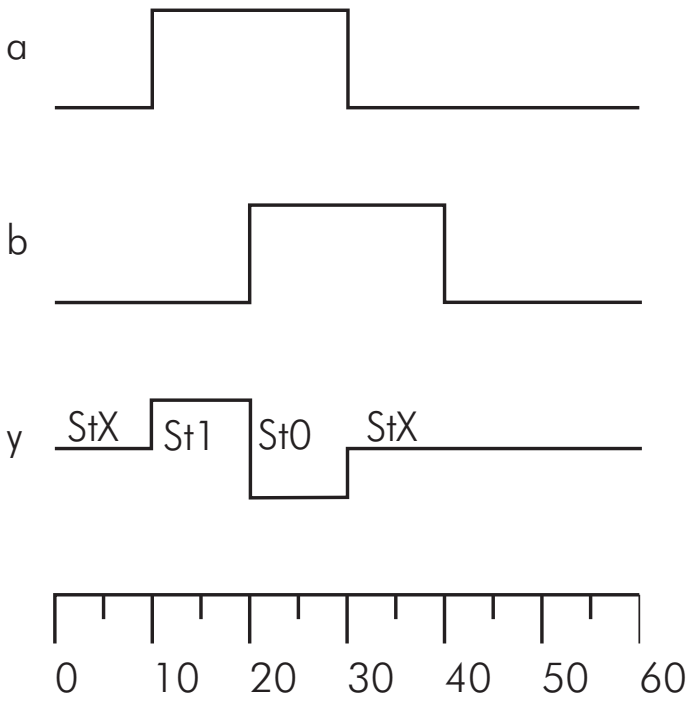



Figure 3.5: Incorrect simulation of pass transistor XOR gate

```
   rpmos p3 (b, y, a);
   rnmos n3 (b, y, nota);
endmodule
```

A simulation of the circuit is shown in Figure 3.6. The signal strengths are shown – Pu0, Pu1, We0 and We1 stand for **pull0**, **pull1**, **weak0** and **weak1** respectively.

## 3.6 Wired Logic

In the netlist above, **pullup** and **pulldown** are primitive elements. The strengths are listed in parentheses after the primitive name. In the case of **pullup** and **pulldown**, listing two strengths is a little redundant, as only one is used in each instance, but in a more general case the output strengths of any gate can be listed. For example, we could declare an OR gate with strengths (and delays for completeness):

**or** (**strong0**, **weak1**) #(10, 12) g1 (y, a, b);

If we connect the outputs of two such gates together, we can write a “truth table” describing how the output wire is resolved.

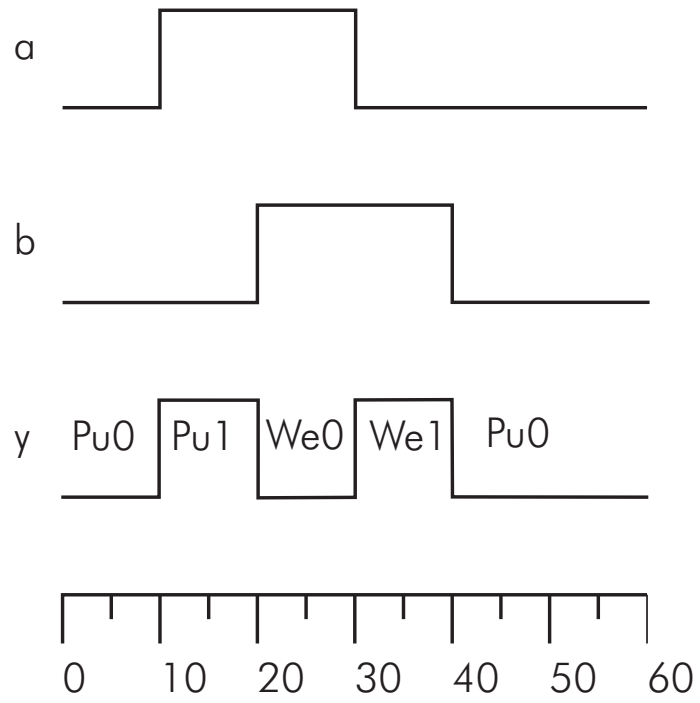



Figure 3.6: Correct simulation of pass transistor XOR gate

**or** (**strong0**, **weak1**) #(10, 12) g1 (y, a, b);
**or** (**strong0, weak1**) #(10, 12) g2 (y, c, d);

| **g1** | **g2** | **z** |
|---|---|---|
| strong0 | strong0 | strong0 |
| strong0 | weak1 | strong0 |
| weak1 | strong0 | strong0 |
| weak1 | weak1 | weak1 |

The output wire, y, behaves like an AND gate.

This is known as wired-AND behaviour. In some technologies, this is a legitimate way of building logic circuits. It is not relevant to CMOS or FPGA technologies. For this reason, we will not discuss strengths further. We will also note in passing that the **wire** declaration can be replaced by one of the following to indicate how signal resolution is performed: **tri**, **wor**, **wand**, **trior**, **triand**, **trireg**, **tri1**, **tri0**, **supply1**, **supply0**. Again, we will not discuss these declarations further.

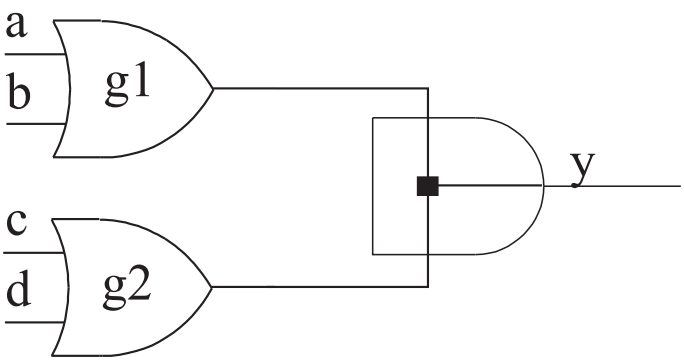



Figure 3.7: Wired-AND

## 3.7 Three-state Primitives

In CMOS technologies, three-state[2] logic is used to allow multiple gates to drive the same piece of wire. Implicitly, the high-impedance state, z, has a lower strength than 1, 0 or x. SystemVerilog has four primitives to model three state buffers: **notif0**, **notif1**, **bufif0**, **bufif1**. As the names imply, these four primitives invert or buffer a signal according to the state of a control signal, or if the control signal is not asserted, the out put is in the high-impledance state.

In all four cases, three wires are connected to the primitive: the output, the input and the control, in that order. For example:

**bufif1** (outa, inb, controlc);

These primitives may also take a third delay parameter: the delay for the output to change to the high-impedance state.

As we will see in the next Chapter, there are easier ways to model three-state buses, than with collections of three-state primitives.

## 3.8 User-Defined Primitives (UDPs)

In the next Chapter we will look at more abstract ways to describe combinational circuits. While, however, we are looking at low-level modelling, we will briefly consider how we could define our own primitives to augment the built-in primitives. The advantage of using UDPs is that exactly the same style of delay and strength modelling can be used as for the built-in primitives. The UDP model simply has to describe the logical function. Other

[2]The term “tristate” is commonly used to denote three-state logic. In fact, “tristate” is a trademark, so we will avoid its use here.

modelling styles require delays to be explicitly included in the model. The disadvantage of UDPs is that they are not synthesisable. So this section is only relevant if you find yourself needing to write models of low-level cells for verification.

As an example of a UDP, consider a four input AND-OR-INVERT gate. This can be built from three two-input NAND gates and an inverter (14 transistors) or from 8 transistors as shown below.

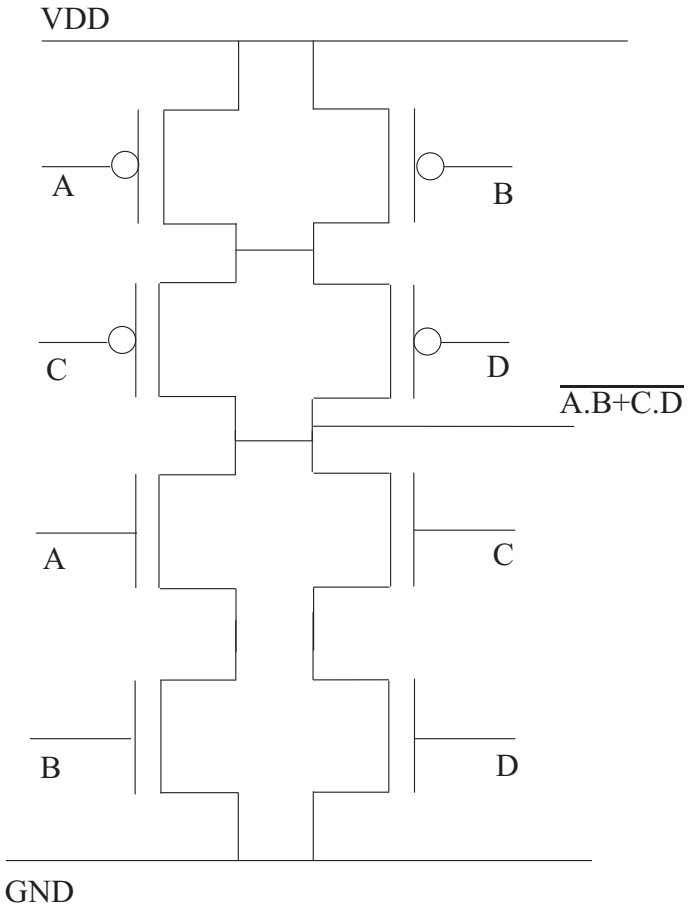



Figure 3.8: AND-OR-Invert cell

The truth table for this is:

If you draw the Karnaugh map for this table, you can see that the function can be described by a small number of minterms or maxterms. A SystemVerilog primitive model for this gate is:

**primitive** aoi4 (**output** y, **input** a, b, c, d);

**table**

| A | B | C | D | Y |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 |

```
//abcd : y
11?? : 0;
??11 : 0;
0?0? : 1;
0??0 : 1;
?00? : 1;
?0?0 : 1;
endtable
endprimitive
```

The symbol ? is a “don’t care” that stands for 1, 0 or x. All combinations not listed in the table (i.e. those that include some other combination of xs) result in an x output. It is not possible to have a z in an input (it will be converted to an x) or to specify a high-impedance output.

# Summary

In this section we have discussed the modelling of logic circuits as netlists of primitive components.

A circuit block begins with the keyword module and ends with the keyword endmodule.

Inputs and outputs are listed in parentheses after the module keyword and are then declared as inputs, outputs or inouts on subsequent lines.

Internal wires may optionally be declared. It is good practice to do this as it increases the readability of a design.

A number of gate primitives exist in SystemVerilog. Generally, the connections to these gates are in the order: output(s), input(s), control.

Primitives may have one or two (or in the case of three-state primitives, three) delay parameters. The delays are listed following a hash (#). Delays are inertial. Delays may be specified with minimum, typical and maximum values.

SystemVerilog signals can take four values: 1, 0, x or z. z has a lower strength than the other values.

Signal strengths can also be included for switch-level or wired logic modelling. Signal strengths are not used for CMOS modelling or in logic synthesis.

# Exercises

3.1 Write a description of a three-input NAND gate with a delay of 5 units.

3.2 A full adder has the following truth table for its sum (S) and carry (Co) outputs, in terms of its inputs, A, B and carry in (Ci).

| **A** | **B** | **Ci** | **S** | **Co** |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

Derive expressions for S and Co using only AND and OR operators. Hence write a SystemVerilog description of a full adder as a netlist of AND and OR gates and inverters. Do not include any gate delays in your models.

3.3 Modify the gate models of Exercise 3.2 such that each gate has a delay of 1 unit. What is the maximum delay through your full adder?

# Chapter 4

# Combinational Building Blocks

While it is possible to design all combinational (and indeed sequential) circuits in terms of logic gates, in practice this would be extremely tedious. It is far more efficient, in terms of both the designer's time and the use of programmable logic resources, to use higher level building blocks. If we were to build systems using TTL or CMOS integrated circuits on a printed circuit board, we would look in a catalogue and choose devices to implement standard circuit functions. If we use SystemVerilog and programmable logic, we are not constrained to using just those devices in the catalogue, but we still think in terms of the same kinds of circuit functions. In this chapter we will look at a number of combinational circuit functions. As we do so, various features of SystemVerilog will be introduced. In addition, the IEEE dependency notation will also be introduced, allowing us to describe circuits using both graphical and textual representations.

## 4.1 Multiplexers

### 4.1.1 2 to 1 Multiplexer

A multiplexer can be used to switch one of many inputs to a single output. Typically multiplexers are used to allow large, complex pieces of hardware to be reused. The IEEE symbol for a 2 to 1 multiplexer is given in Figure 4.1. G is a select symbol. If G is true, the input labelled 1 is connected to the output; if G is false the input labelled $\bar{1}$ is chosen.

A SystemVerilog model of this multiplexer is given below.

```
module mux2 (output logic y,
             input logic a, b, s);
```

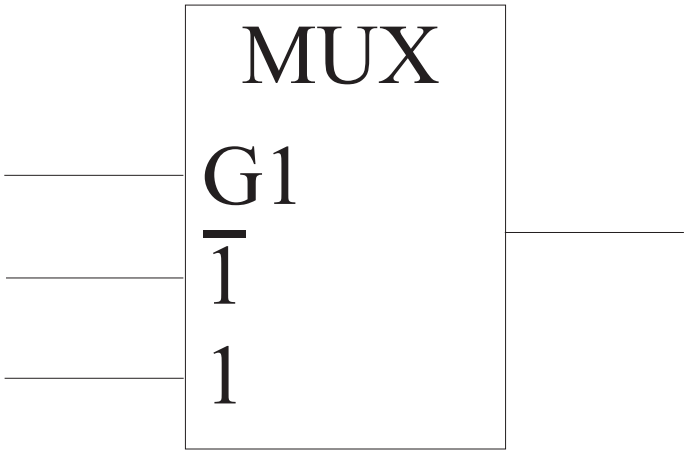



Figure 4.1: 2 to 1 multiplexer

```
always_comb
  if (s)
    y = b;
  else
    y = a;

endmodule
```

**always_comb** is a SystemVerilog variant on the general purpose Verilog **always** construct. We will see other variants in later chapters. A procedural **always** block allows various procedural programming constructs to be used. **always_comb** indicates that the block models purely combinational logic at the register transfer level. In a general purpose Verilog **always** block, every input used by that block must be listed. For an **always_comb** block, the inputs are derived automatically. Before SystemVerilog, two of the commonest errors made in writing RTL Verilog were the accidental creation of sequential logic and the accidental omission of input signals, resulting in a mis-match between simulated and synthesized behaviour. Therefore it is very strongly recommended that combinational logic should, with a small number of specific exceptions, always be modelled using the **always_comb** construct.

The **always_comb** block here contains exactly one statement. If more than one statement is needed, they should be grouped using **begin** and **end**. The **if** statement here is self-explanatory: if the select input, s, is true, the logic value at input b is assigned to output y; if false, the value at a is assigned. An **if** statement must occur within a procedural block. Note

that the assignment is indicated using a single equals sign (=). This is known as a *blocking* assignment. The significance of this will be explained later. It is sufficient to note here that combinational logic should always be modelled using blocking assignments.

### 4.1.2 4 to 1 Multiplexer

The symbol for a 4 to 1 multiplexer is shown in Figure 4.2. As before, G is a select symbol. $\frac{0}{3}$ is not a fraction, but means 0-3. Therefore the binary value on the top two inputs is used to select one of the inputs 0-3.

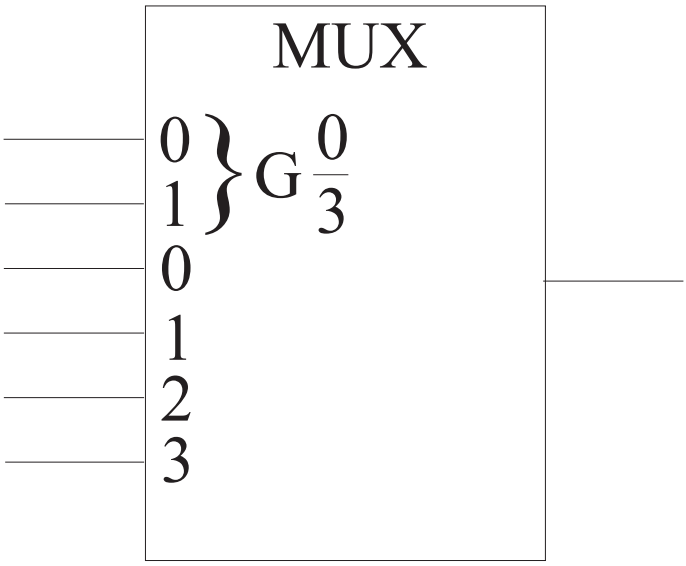



Figure 4.2: 4 to 1 multiplexer

The 2 to 1 multiplexer model can be extended to a 4 to 1 multiplexer by nesting if statements.

```
module mux4 (output logic y,
             input logic a, b, c, d, s0, s1);

always_comb
  if (s0)
    if (s1)
      y = d;
    else
      y = c;
  else
```

```
    if (s1)
       y = b;
    else
       y = a;

endmodule
```

## 4.2 Decoders

### 4.2.1 2-4 Decoder

A decoder converts data that has previously been encoded into some other form. For example, $n$ bits can represent $2^n$ distinct values. The truth table for a 2-4 decoder is given below.

| Inputs | | Outputs | | | |
|---|---|---|---|---|---|
| **A1** | **A0** | **Z3** | **Z2** | **Z1** | **Z0** |
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |

The IEEE symbol for a 2 to 4 decoder is shown in Figure 4.3. BIN/1-OF-4 indicates a binary decoder in which one of four outputs will be asserted. The numbers give the "weight" of each input or output.

We could choose to treat each of the inputs and outputs separately, but as they are obviously related, it makes sense to treat the input and output as two vectors of size 2 and 4 respectively. Vectors can be described using an array of variables, for example:

```
logic [3:0] four_bit_array;
```

The 2-4 decoder can be modelled using a **case** statement:

```
module decoder (output logic [3:0] y,
                input logic [1:0] a);

always_comb
  case (a)
    0 : y = 1;
```

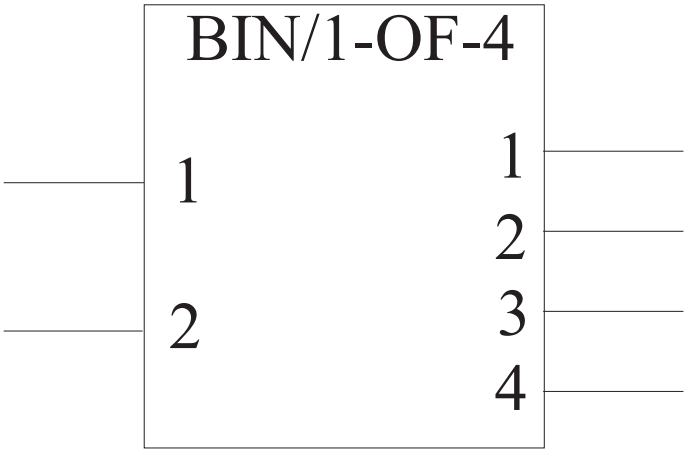



Figure 4.3: 2 to 4 decoder

```
    1 : y = 2;
    2 : y = 4;
    3 : y = 8;
    default : y = 'x;
  endcase
endmodule
```

Depending on the numerical value of a, one of the branches in the case statement is selected. There is, however, a sleight of hand here. SystemVerilog is not a strongly typed language. The input, a, is declared to be a two-bit variable of type logic, but it is interpreted as an integer. This is acceptable in SystemVerilog, but would be completely illegal in many other HDLs and programming langauges. This is ability to interpret bit patterns automatically is very powerful, but can be dangerous. The bit pattern could have been interpreted as a signed number and there is no protection against mixing such interpretations. So be careful! Similarly, the output is assigned an integer value that is automatically reinterpreted as four bits.

The fifth alternative is a default. At first glance, this seems redundant as two bits give four values, as specified. The control input, a, is of type logic, however. Therefore its bits can take x or z values. So, in effect, there are 16 possible values for a. The default line assigns an x to the output if any of the input bits is not a true binary value. This line will not be synthesized, but it is good practice to include it if you want to see unusual behaviour in simulation.

### 4.2.2 $N - 2^N$ Decoder

We have seen two ways to describe a 2-4 decoder. The same structures could easily be adapted to model a 3-8 decoder or a 4-16 decoder. Although these devices are clearly more complex than the 2-4 decoder, conceptually there is little difference. It would be convenient to have a general $N-2^N$ decoder that could be described once, but used for any application. We can't, of course, write a case statement with an indeterminate number of branches. Another approach is needed. One way to think about this is based on the following observation. The output can be described as a single 1 shifted leftwards by a number of places given by the input number. The bit pattern of the input, a, is again interpreted as an integer number.

```
y = 1'b1 << a;
```

We specify a single bit with the notation 1'b, followed by the value of that bit.

Similarly, an array of size $2^N$ can be declared as [(1<<N)−1:0]. If N takes the default value of 3, the width of the output vector is given by 1 (note that this can be an integer value, not a bit value) shifted left by three places to give the bit pattern $1000_2$, which is 8 in decimal. To get 8 bits in total, we make the range (8-1) down to 0.

We saw in the previous chapter that parameters can be used to pass values, such as delays, to SystemVerilog models. We can similarly use a parameter to define the size of a structure.

```
module decoderN #(parameter N = 3)
  (output logic [(1<<N)-1:0] y, input logic [N-1:0] a);

  always_comb
    y = 1'b1 << a;

endmodule
```

There is, of course, another way to describe the decoder - as a $\log_2(N)-N$ decoder. In fact, we need the *ceiling* of the function, in other words the result is rounded up to the next highest integer. This function, clog2, can be implemented by shifting and adding. In SystemVerilog, a *constant function* can be used to determine such values as array sizes. Constant functions are evaluated at compile time and hence are a little more limited that regular functions. An example of constant function is given below. It is left as an exercise for the reader to understand how the clog2 function works!

```
module decoderlogN #(parameter N = 8)
  (output logic [N-1:0] y,
   input logic [clog2(N)-1:0] a);

  function int clog2(input int n);
    begin
    clog2 = 0;
    n--;
    while (n > 0)
      begin
          clog2++;
          n >>= 1;
          end
    end
  endfunction


  always_comb
    y = 1'b1 << a;

endmodule
```

### 4.2.3 Seven-segment decoder

Sometimes, several input patterns might give the same output. There are two alternatives to the **case** statement that allow don't care values.

- **casez** allows z values in the case branches to be treated as don't cares. A ? can be used instead of z.
- **casex** allows z and x to be treated as don't cares.

If more than one pattern should give the same output, the patterns can be listed. For example the following model describes a seven-segment decoder that displays the digits '0' to '9'. If the bit patterns corresponding to decimal values '10' to '15' are fed into the decoder, an 'E' (for "Error") is displayed. If the inputs contain 'X's or other invalid values, the display is blanked. These patterns are shown in Figure 4.4. (But be careful, there are many different ways to encode seven-segment displays. This example will need to be changed if the segments are differently numbered or if the logic is active low.)

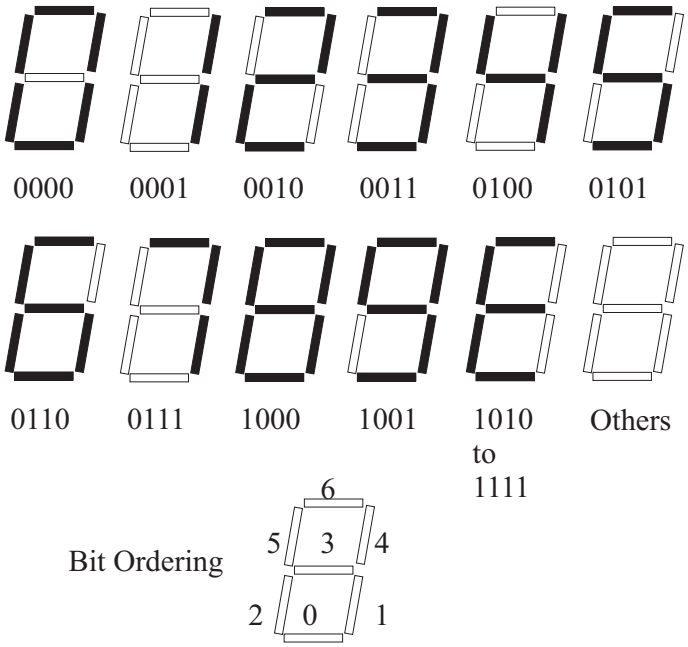



Figure 4.4: Seven-segment display

```
module sevenseg(output logic [6:0] data,
                input logic [3:0] address);

always_comb
  casez (address)
    4'b0000 : data = 7'b1110111;
    4'b0001 : data = 7'b0010010;
    4'b0010 : data = 7'b1011101;
    4'b0011 : data = 7'b1011011;
    4'b0100 : data = 7'b0111010;
    4'b0101 : data = 7'b1101011;
    4'b0110 : data = 7'b1101111;
    4'b0111 : data = 7'b1010010;
    4'b1000 : data = 7'b1111111;
    4'b1001 : data = 7'b1111011;
    4'b101?,
    4'b11?? : data = 7'b1101101;
    default : data = 7'b0000000;
  endcase
endmodule
```

## 4.3 Priority Encoder

### 4.3.1 Don't cares and uniqueness

An encoder takes a number of inputs and encodes them in some way. The difference between a decoder and an encoder is therefore somewhat arbitrary. In general, however, an encoder has fewer outputs than inputs. A priority encoder attaches an order of importance to the inputs. Thus if two inputs are asserted, the most important input takes priority. The symbol for a priority encoder is shown in Figure 4.6. There are three outputs. The lower two are the encoded values of the four inputs. The upper output indicates whether the output combination is valid. An OR function ($\geq 1$) is used to check that at least one input is 1. Z is used to denote an internal signal. Thus Z10 is connected to 10. This avoids unsightly and confusing lines across the symbol.

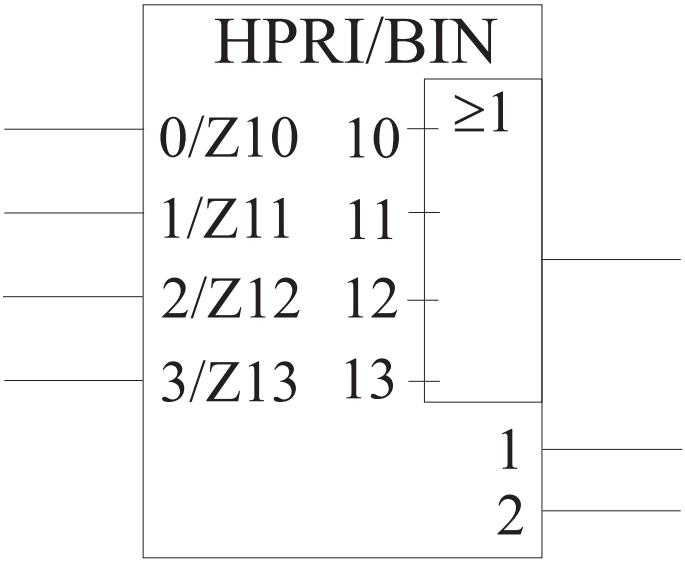



Figure 4.5: 4 to 2 priority encoder

An example of a priority encoder is given in the truth table below. The 'Valid' output is used to signify whether at least one input has been asserted and hence whether the outputs are valid.

We can code this directly in SystemVerilog, using the **casez** statement. We need to be a little careful when using don't cares. It would be very easy to write two or more lines that overlapped. In other words, a pattern might match two or more case branches. For example, the pattern

| Inputs | | | | Outputs | | |
|---|---|---|---|---|---|---|
| **A3** | **A2** | **A1** | **A0** | **Y1** | **Y0** | **Valid** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | - | 0 | 1 | 1 |
| 0 | 1 | - | - | 1 | 0 | 1 |
| 1 | - | - | - | 1 | 1 | 1 |

4'b0110 would match both 4'b0?10 and 4'b01?0. If both these alternatives were in a **casez** statement, the one occurring first would be selected in simulation. If the design were synthesized, however, there would be an ambiguity and the synthesis tool might attempt to impose its own priority. To avoid any ambiguity, it is good practice to qualify a **casez** statement with the **unique** modifier. If there is an overlap, an error would be flagged during compilation.

We can reproduce the structure of the truth table, by making one assignment to y and valid simultaneously. Curly braces { } are used to concatenate variables.

```
module encoder (output logic [1:0] y, logic valid,
                input logic [3:0]a);

always_comb
  unique casez (a)
    4'b1??? : {y,valid} = 3'b111;
    4'b01?? : {y,valid} = 3'b101;
    4'b001? : {y,valid} = 3'b011;
    4'b0001 : {y,valid} = 3'b001;
    default : {y,valid} = 3'b000;
  endcase

endmodule
```

## 4.4 Adders

### 4.4.1 Functional Model

The IEEE Symbol for a 4-bit adder is shown in Figure 4.6. The $\sum$ symbol denotes an adder. P and Q are assumed to be the inputs to the adder. CI and CO are carry in and carry out, respectively.

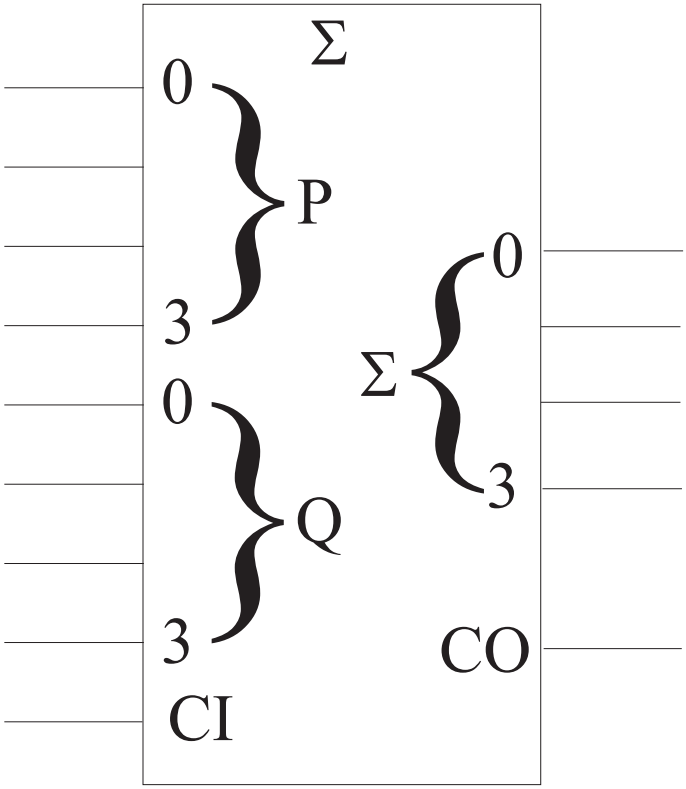



Figure 4.6: 4-bit adder

The addition of two n-bit integers produces a result of length n+1, where the most significant bit is the carry out bit. Therefore within the SystemVerilog description we must separate the result into an n-bit sum and a carry out bit. The code below performs these actions for both signed and unsigned addition. The curly braces concatenate a single bit and an n-bit vector to give a vector of length n+1. The complete code is shown below.

```
module adder #(parameter N = 4)
  (output logic [N-1:0] Sum, output logic Cout,
   input logic [N-1:0] A, B, input logic Cin);

  always_comb
    {Cout, Sum} = A + B + Cin;

endmodule
```

### 4.4.2 Ripple Adder

A simple model of a single-bit full adder might be:

```
module fulladder (output logic sum, cout,
                  input logic a, b, cin);

  always_comb
    begin
    sum = a ^ b ^ cin;
    cout = a & b | a & cin | b & cin;
    end

endmodule
```

This model contains two assignments, to Sum and Cout, written as Boolean expressions. We can build a multi-bit adder using several instances of this full adder. If we know how many bits will be in our adder we simply instantiate the model several times. If, however, we want to create a general N-bit adder, we need some type of iterative construct. The **generate** construct with a **for** loop allows repetition in a dataflow description. This example creates N-2 instances and, through the Ca vector, wires them up. Notice that the loop variable, i, is declared as a **genvar**.

The first and last bits of the adder do not conform to the general pattern, however. Bit 0 should have Cin as an input and bit N-1 should generate

Cout. We make special cases of the first and last elements, by instantiating them outside the generate block.

```
module ripple #(parameter N = 4)
  (output logic [N-1:0] Sum, output logic Cout,
   input logic [N-1:0] A, B, input logic Cin);

  logic [N-1:1] Ca;
  genvar i;

  fulladder f0 (Sum[0], Ca[1], A[0], B[0], Cin);

  generate for (i = 1; i < N-1; i++)
    begin : f_loop
    fulladder fi (Sum[i], Ca[i+1], A[i], B[i], Ca[i]);
    end
  endgenerate

  fulladder fN (Sum[N-1], Cout, A[N-1], B[N-1], Ca[N-1]);

endmodule
```

## 4.5 Parity Checker

The principle of parity checking was explained in Chapter 2. The IEEE symbol for a parity checker is shown in Figure 4.7. The symbol 2k indicates that the output is asserted if 2k inputs are asserted for any integer, k. Thus the output is asserted for even parity. An odd parity checker has the output inverted.

In addition to the usual programming operators, SystemVerilog has reduction operators that can be applied to all the bits of a vector. For example, the even parity bit can be generated by taking the exclusive OR of all the bits of a vector and inverting.

```
module parity #(parameter N = 4)
       (output logic even, input logic [N-1:0] a);

  always_comb
    even = ~^a;

endmodule
```

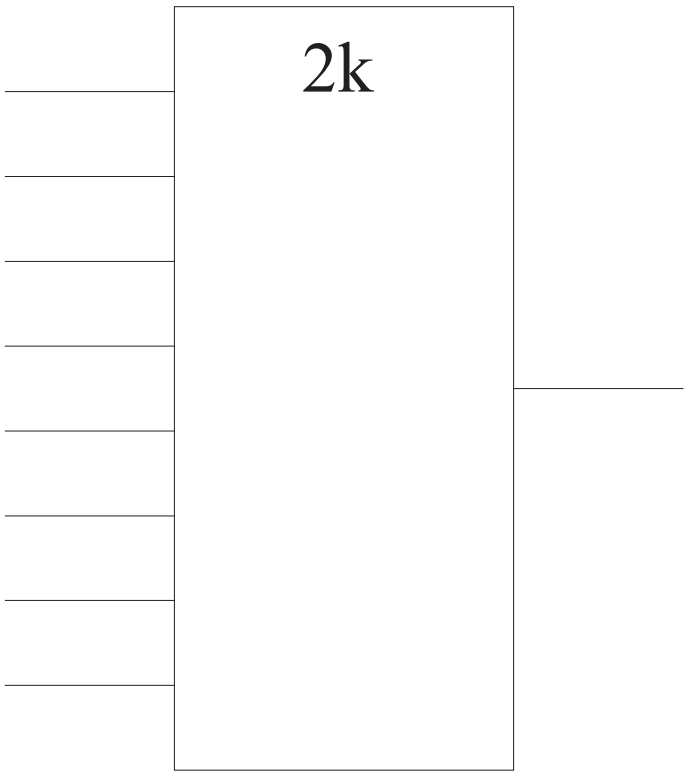



Figure 4.7: Even partity checker

## 4.6 Three state Buffers

### 4.6.1 Multi-Valued Logic

In addition to the normal Boolean logic functions, it is possible to design digital hardware using switches to disconnect a signal from a wire. For instance, we can connect the outputs of several gates together, through switches, such that only one output is connected to the common wire at a time. This same functionality could be achieved using conventional logic, but would probably require a greater number of transistors. The IEEE symbol for a three state buffer is shown in Figure 4.8. The symbol “1” shows the device is a buffer. “EN” is the symbol for an output enable and the inverted equilateral triangle indicates a three state output.

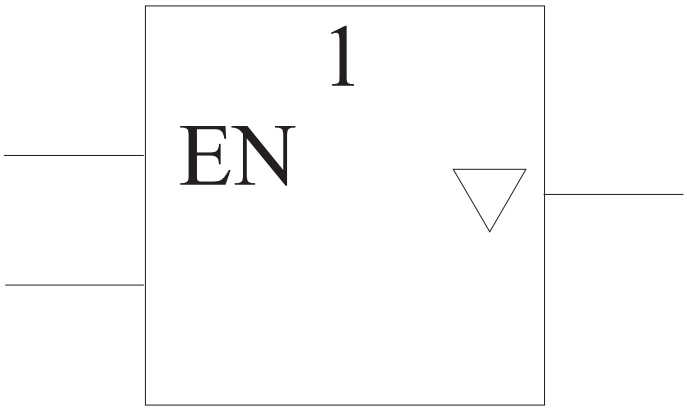



Figure 4.8: Three-state buffer

If we write a model using variables of type **logic**, we must ensure that two models do not attempt to put a value onto the same variable. The purpose of using three state buffers is to allow two or more component outputs to be connected together, provided that no more than one output generates a logic 1 or 0 and the rest of the outputs are in the high impedance state. This cannot be done with logic variables – a SystemVerilog simulator does not treat ’z’ as a special case. Resolution of conflicting logic values is done using a **wire**. Assignment of high-impedance can be done from within a procedural block, but it is easier to use a *continuous assignment*, that is outside any procedural block. Conversely, most of the examples in this chapter can be written using continuous assignments, but the procedural style is easier to use. Therefore it is recommended that all three-state elements are modelled using continuous assignments and that the continuous assignment is only used for this purpose.

A SystemVerilog model of a three-state buffer follows.

```
module threestate (output wire y,
                   input logic a, enable);

  assign y = enable ? a : 'z;

endmodule
```

It is also possible to use three state logic to build a multiplexer. A 4 to 1 multiplexer implemented in three state logic is shown below. There are four assignments to 'y'. At any time, three are 'z' and one is an input value. In order for the output value to be correctly determined, and in order not to cause a compilation error, y must be declared to be a wire.

```
module threemux4 (output wire y,
                  input logic a, b, c, d, s0, s1);

  assign y = (~s0 && ~s1) ? a : 'z;
  assign y = (s0 && ~s1) ? b : 'z;
  assign y = (~s0 && s1) ? c : 'z;
  assign y = (s0 && s1) ? d : 'z;

endmodule
```

## Summary

In this Chapter we have introduced a number of typical combinational building blocks. The IEEE standard symbols for these blocks have been described.

## Further Reading

A full description of the IEEE symbols is given in the IEEE standard and in a number of digital design textbooks. Manufacturers' data sheets may use the IEEE symbols or a less standard form.

# Exercises

4.1 SystemVerilog models can be written using continuous and procedural assignments. Explain, with examples, the meaning of continuous and procedural in this context.

4.2 Write models of a three to eight decoder using (a) Boolean operators, (b) a conditional operator and (c) a shift operator.

# Chapter 5

# SystemVerilog models of sequential logic blocks

In the previous chapter we presented several examples of combinational building blocks, at the same time introducing various aspects of SystemVerilog. In this chapter we shall repeat the exercise for sequential blocks.

## 5.1 Latches

### 5.1.1 SR latch

There is often confusion between the terms ‘latch’ and ‘flip-flop’. Here, we will use ‘latch’ to mean a level-sensitive memory device and ‘flip-flop’ to specify an edge-triggered memory element. We will discuss the design of latches and flip-flops in Chapter 12. We will simply note here that a latch is based on cross-coupled gates, as shown in Figure 5.1. Table 5.1 gives the truth table of this latch.

| $S$ | $R$ | $Q$ | $\bar{Q}$ |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | $Q$ | $\bar{Q}$ |

Table 5.1: Truth table of SR latch.

When $S$ and $R$ are both at logic 1, the latch holds onto its previous value. When both are at 0, both outputs are at 1. It is this latter behaviour that makes the SR latch unsuitable for designing larger circuits, as a latch

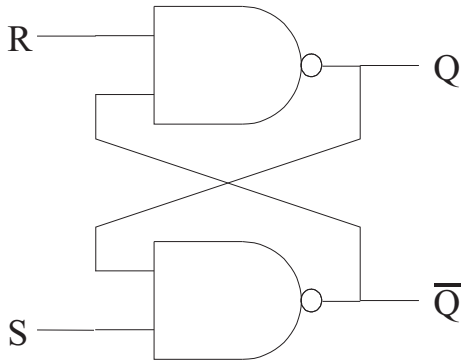



Figure 5.1: SR latch

or flip-flop would normally be expected to have different values at its two outputs, and it is difficult to ensure that both inputs will never be 0 at the same time.

The SR latch could be modelled in SystemVerilog in a number of ways. Two examples are shown below.

```
module rslatch1 (output wire q, qbar,
                 input logic r, s);

  nand n0(q, qbar, r);
  nand n1(qbar, q, s);

endmodule
```

In the first example, the latch is modelled using two NAND gates. There is nothing fundamentally wrong with this model, but it is dependent on the technology and it would be a little impractical for larger elements.

```
module rslatch2 (output logic q, qbar,
                 input logic r, s);

always @(r, s)
  unique case ({r, s})
    2'b00: {q, qbar} <= 2'b11;
    2'b01: {q, qbar} <= 2'b10;
    2'b10: {q, qbar} <= 2'b01;
    default;
  endcase

endmodule
```

In the second model, we explicitly model an element with storage. Therefore we cannot use a **always_comb** procedural block, which would imply purely combinatorial logic, without storage. We can use a general purpose **always** block, as shown. We have to list the two inputs and using a **case** statement, the truth table of the latch can be reproduced. The curly braces $\{\ \ \}$ concatenate two variables, both in the case selector and in the case branches.

In the first three branches of the **case** statement, values are assigned to q and qbar depending on the combination of inputs. Nothing is assigned in the fourth, **default** branch, so the q and qbar values are retained. In other words the values are latched. If the module is synthesised, a latch will be inferred.

Notice that we have specified that the case statement is non-overlapping (**unique**) and that there is a default (in which nothing happens).

These two examples show that omitting an assignment for one or more input conditions infers a latch. If this is done by accident in an **always_comb** block, a synthesis tool will generate a warning, *but* might interpret the code as a latch. Such warnings should always be examined carefully. Unintended latches will almost certainly cause the circuit to work incorrectly.

### 5.1.2 D latch

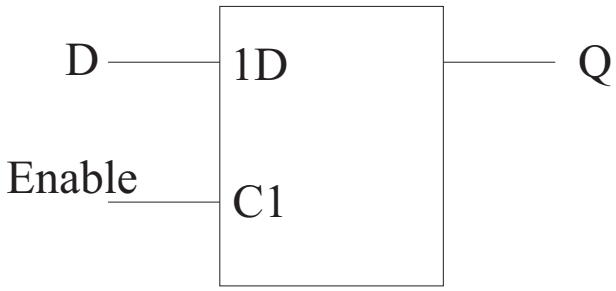



Figure 5.2: Level-sensitive D latch

Because an SR latch can have both outputs at the same value, it is seldom if ever used. More useful is the D latch, as shown in Figure 5.2. The input of a D latch is transferred to the output if an enable signal is asserted. $1D$ indicates a dependency of the D input on control signal 1 ($C1$). The $\bar{Q}$ output is not shown.

A behavioural SystemVerilog model of a D latch is

```
module dlatch (output logic q, input logic d, en);
```

```
always_latch
  if (en)
    q <= d;

endmodule
```

## 5.2 Flip-flops

### 5.2.1 Edge-triggered D flip-flop

In the previous chapter the principle of synchronous sequential design was described. The main advantage of this approach to sequential design is that all changes of state occur at a clock *edge*. The clock edge is extremely short in comparison to the clock period and to propagation delays through combinational logic. In effect, a clock edge can be considered to be instantaneous.

The IEEE symbol for a positive edge-triggered D flip-flop is shown in Figure 5.3. Again, the number 1 shows the dependency of $D$ on $C$. The triangle at the clock input denotes edge-sensitive behaviour. An inversion circle, or its absence, shows sensitivity to a negative or positive edge, respectively.

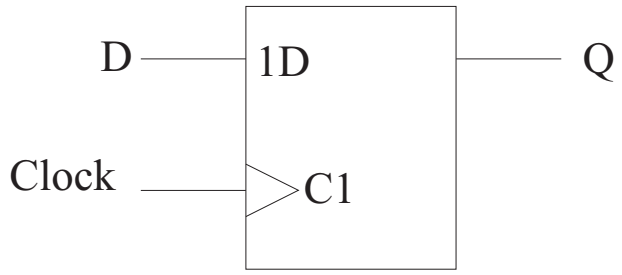



Figure 5.3: Positive edge-triggered D flip-flop

The simplest SystemVerilog model of a positive edge-triggered D flip-flop is given below.

```
module dff (output logic q, input logic d, clk);

always_ff @(posedge clk)
  q <= d;
```

**endmodule**

Similarly, a negative edge-triggered flip-flop can be modelled by detecting a transition to logic 0.

### 5.2.2 Asynchronous set and reset

When power is first applied to a flip-flop its initial state is unpredictable. In many applications this is unacceptable, so flip-flops are provided with further inputs to set (or reset) their outputs to 1 or to 0, as shown in Figure 5.4. Notice that the absence of any dependency on the clock implies asynchronous behaviour for $R$ and $S$.

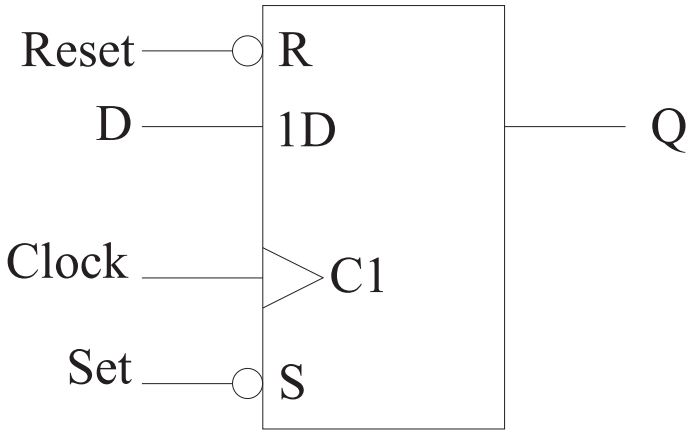



Figure 5.4: Positive edge-triggered D flip-flop with asynchronous reset and set.

These inputs should only be used to initialize a flip-flop. It is very bad practice to use these inputs to set the state of a flip-flop during normal system operation. The reason for this is that in synchronous systems, flip-flops only change state when clocked. The set and reset inputs are *asynchronous* and hence cannot be guaranteed to change an output at a particular time. This can lead to all sorts of timing problems. In general, keep all designs strictly synchronous or follow a structured asynchronous design methodology.

A SystemVerilog model of a flip-flop with an asynchronous reset must respond to changes in the clock and in the reset input.

```
module dffr (output logic q,
             input logic d, clk, n_reset);
```

```
always_ff @(posedge clk, negedge n_reset)
  if (~n_reset)
    q <= '0;
  else
    q <= d;

endmodule
```

An asynchronous set can be described in a similar way (see Exercises).

It is possible for a flip-flop to have both an asynchronous set and reset. For example:

```
module dffrs (output logic q,
              input logic d, clk, n_reset, n_set);

always_ff @(posedge clk, negedge n_reset,
            negedge n_set)
  if (~n_set)
    q <= '1;
  else if (~n_reset)
    q <= '0;
  else
    q <= d;

endmodule
```

This may not correctly describe the behaviour of a flip-flop with asynchronous inputs because asserting both the asynchronous set and reset is usually considered an illegal operation. In this model, Q is forced to 1 if Set is 0, regardless of the Reset signal. Even if this model synthesizes correctly, we would still wish to check that this condition did not occur during a simulation. A technique to do this is described later in this chapter.

### 5.2.3 Synchronous set and reset and clock enable

Flip-flops may have synchronous set and reset functions as well as, or instead of asynchronous set or reset inputs. A synchronous set or reset only takes effect at a clock edge. Thus a SystemVerilog model of such a function must include a check on the set or reset input after the clock edge has been checked. It is not necessary to include synchronous set or reset inputs in the excitation list because the process is only activated at a clock

edge. This is shown in IEEE notation in Figure 5.5. $R$ is now shown to be dependent on $C$ and is therefore synchronous.

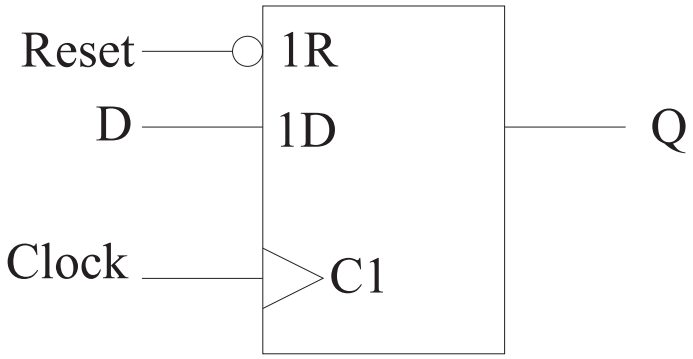



Figure 5.5: Positive edge-triggered D flip-flop with synchronous reset

```
module dffsr (output logic q,
              input logic d, clk, n_reset);

always_ff @(posedge clk)
  if (~n_reset)
    q <= '0;
  else
    q <= d;

endmodule
```

Notice that the only difference between the synchronous and asynchronous reset is whether the signal appears in the excitation list of the **always_ff** block.

Similarly, a flip-flop with a clock enable signal may be modelled with that signal checked after the edge detection. In Figure 5.6, the dependency notation shows that $C$ is dependent on $G$ and $D$ is dependent on (the edge-triggered behaviour of) $C$.

```
module dffe (output logic q,
             input logic d, clk, enable);

always_ff @(posedge clk)
  if (enable)
    q <= d;

endmodule
```

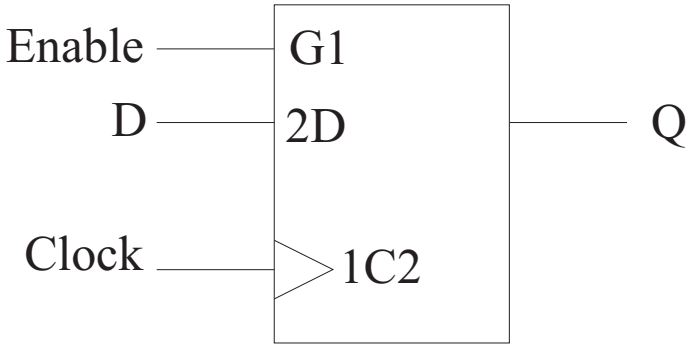



Figure 5.6: Positive edge-triggered D flip-flop with clock enable

A synthesis system is likely to interpret this as a flip-flop with a clock enable. The following model is likely to be interpreted differently, although it appears to have the same functionality.

Again, the $D$ input is latched if Enable is true and there is a clock edge. This time, however, the clock signal passes through an AND gate and hence is delayed. The $D$ input is also latched if the clock is true and there is a rising edge on the Enable signal! This is another example of design that is not truly synchronous and which is therefore liable to timing problems. This style of design should generally be avoided, although for low-power applications the ability to turn off the clock inputs to flip-flops can be useful.

## 5.3 JK and T flip-flops

A D flip-flop registers its input at a clock edge, making that value available during the next clock cycle. JK and T flip-flops change their output states at the clock edge in response to their inputs and to their present states. Truth tables for D, JK and T flip-flops are shown below.

Both the $Q$ and $\bar{Q}$ outputs are shown. Symbols for D, JK and T flip-flops with both outputs and with a reset are shown in Figure 5.7.

```
module jkffr (output logic q, qbar,
              input logic j, k, clk, n_reset);

always_ff @(posedge clk, negedge n_reset)
  if (~n_reset)
    {q, qbar} <= {1'b0, 1'b1};
  else
```

| $D$ | $Q^+$ | $\bar{Q}^+$ |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

| $J$ | $K$ | $Q^+$ | $\bar{Q}^+$ |
|---|---|---|---|
| 0 | 0 | $Q$ | $\bar{Q}$ |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | $\bar{Q}$ | $Q$ |

| $T$ | $Q^+$ | $\bar{Q}^+$ |
|---|---|---|
| 0 | $Q$ | $\bar{Q}$ |
| 1 | $\bar{Q}$ | $Q$ |

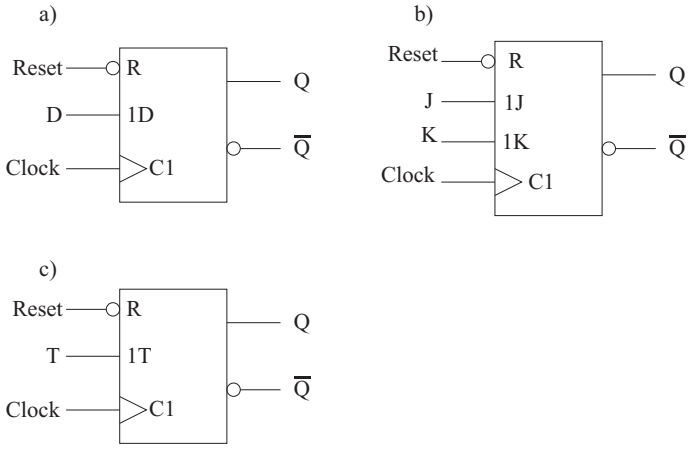



Figure 5.7: (a) D flip-flop; (b) JK flip-flop; (c) T flip-flop.

```
    case ({j, k})
      2'b11 : {q, qbar} <= {qbar, q};
      2'b10 : {q, qbar} <= {1'b1, 1'b0};
      2'b01 : {q, qbar} <= {1'b0, 1'b1};
      default:;
    endcase
```

**endmodule**

A case statement determines the internal state of the JK flip-flop. The selector of the case statement is formed by concatenating the $J$ and $K$ inputs. The **default** clause covers the '00' case and other undefined values. Nothing is done in that clause, so the internal state is retained.

```
module tffr (output logic q, qbar,
             input logic t, clk, n_reset);

always_ff @(posedge clk, negedge n_reset)
  if (~n_reset)
    {q, qbar} <= {1'b0, 1'b1};
  else
    if (t)
      {q, qbar} <= {qbar, q};
```

**endmodule**

The internal state of the T flip-flop is retained between activations of the procedural block, if the T input is not set.

## 5.4 Registers and shift registers

### 5.4.1 Multiple bit register

A D flip-flop is a one-bit register. Thus if we want a register with more than one bit, we simply need to define a set of D flip-flops using vectors:

```
module dffn #(parameter N = 8) (output logic [N-1:0]q,
    input logic [N-1:0] d, input logic clk, n_reset);

always_ff @(posedge clk, negedge n_reset)
  if (~n_reset)
```

```
    q <= '0;
  else
    q <= d;
```

**endmodule**

The IEEE symbol for a 4-bit register is shown in Figure 5.8. Note that the common signals are contained in a control block.

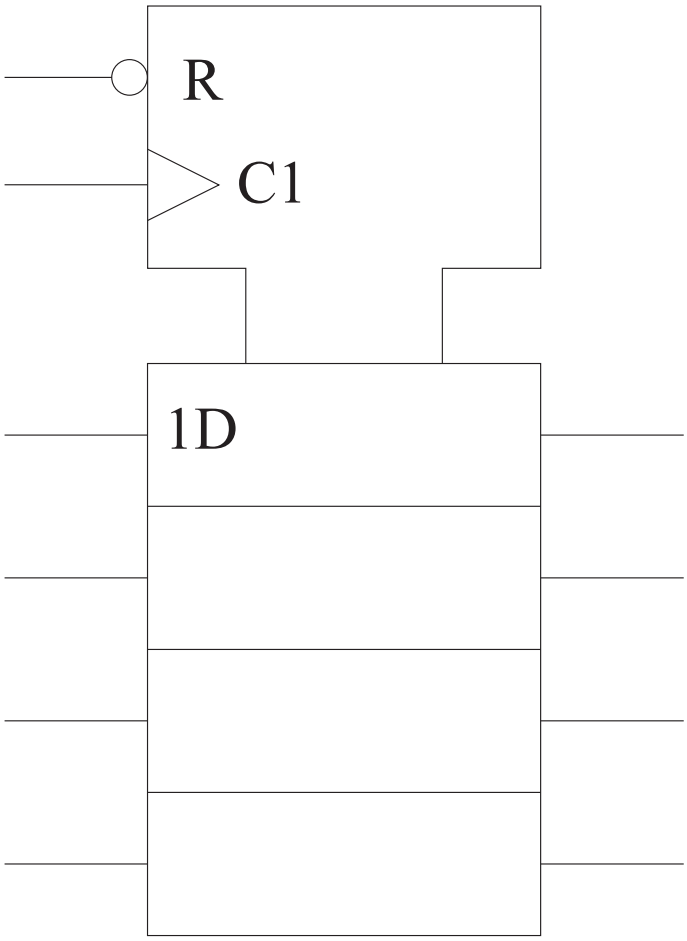



Figure 5.8: Four-bit register.

### 5.4.2 Shift registers

An extension of the above model of a register includes the ability to shift the bits of the register to the left or to the right. For example, a sequence

of bits can be converted into a word by shifting the bits into a register, and moving the bits along at each clock edge. After a sufficient number of clock edges, the bits of the word are available as a single word. This is known as a *serial-in, parallel-out* (SIPO) register.

```
module sipo #(parameter N = 8) (output logic [N-1:0] q,
              input logic a, clk);

  always_ff @(posedge clk)
    q <= {q[N-2:0], a};

endmodule
```

At each clock edge, the bits of the register are moved along by one, and the input, a, is shifted into the 0th bit. The assignment does this by assigning bits $n-2$ to 0 to bits $n-1$ to 1, respectively, and concatenating a to the end of the assignment. The old value for bit $n-1$ is lost.

A more general shift register is the universal shift register. This can shift bits to the left or to the right, and can load an entire new word in parallel. To do this, two control bits are needed. The IEEE symbol is shown in Figure 5.9.

| $S_1S_0$ | **Action** |
|---|---|
| 00 | Hold |
| 01 | Shift right |
| 10 | Shift left |
| 11 | Parallel load |

There are four control modes shown by $M\frac{0}{3}$. The clock signal is split into two for convenience. Control signal 4 is generated and in modes 1 and 2 a shift left or shift right operation, respectively, is performed. 1,4$D$ means that a D-type operation occurs in mode 1 when control signal 4 is asserted.

```
module usr #(parameter N = 8) (output logic [N-1:0]q,
  input logic [N-1:0] a, input logic [1:0] s,
  input logic lin, rin, clk, n_reset);

always_ff @(posedge clk, negedge n_reset)
  if (~n_reset)
    q <= '0;
  else
    case (s)
```

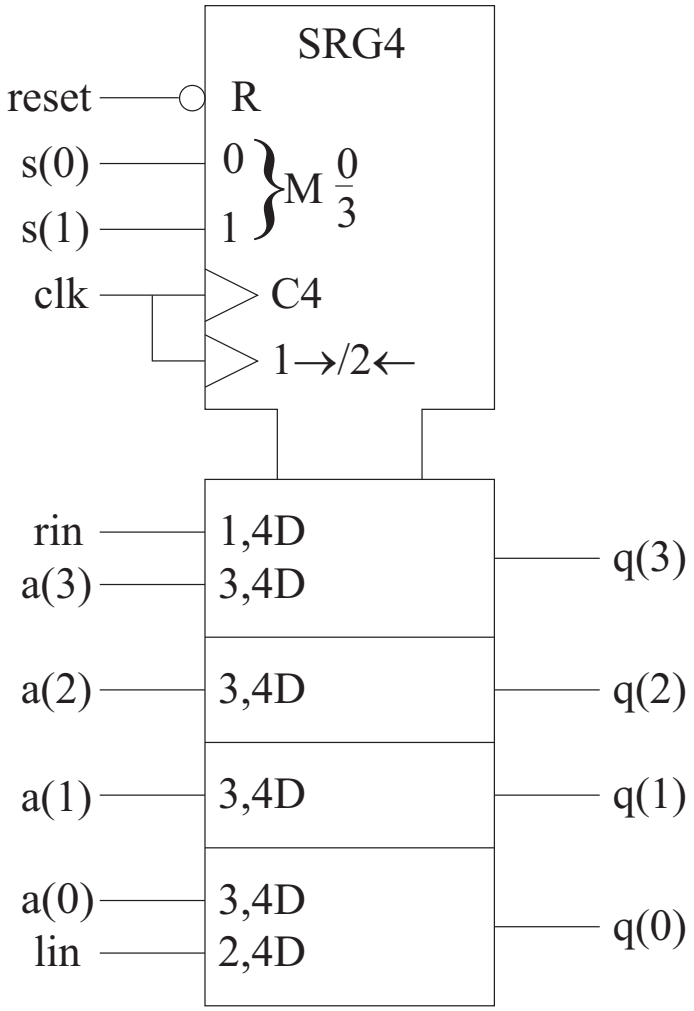



Figure 5.9: Universal shift register.

```
      2'b11: q <= a;
      2'b10: q <= {q[n-2:0], lin};
      2'b01: q <= {rin, q[n-1:1]};
      default:;
    endcase

endmodule
```

The shift operations are done by taking the lowest $(n-1)$ bits and concatenating the leftmost input (shift left) or by taking the upper $(n-1)$ bits concatenated to the rightmost input (shift right). It would be possible to use the shift operators, but in practice they are not needed.

## 5.5 Counters

Counters are used for a number of functions in digital design, e.g. counting the number of occurrences of an event; storing the address of the current instruction in a program; or generating test data. Although a counter typically starts at zero and increments monotonically to some larger value, it is also possible to use different sequences of values, which can result in simpler combinational logic.

### 5.5.1 Binary counter

A binary counter is a counter in the intuitive sense. It consists of a register of a number of D flip-flops, the content of which is the binary representation of a decimal number. At each clock edge the contents of the counter is increased by one, as shown in Figure 5.10. We can easily model this in SystemVerilog, using the '+' operator. The reset operation is shown in Figure 5.10 as setting the contents (*CT*) to 0. The weight of each stage is shown in brackets.

```
module counter #(parameter N = 8)
                (output logic [N-1:0] count,
                 input logic n_reset, clk);

always_ff @(posedge clk, negedge n_reset)
  if (~n_reset)
    count <= 0;
  else
    count <= count + 1; //don't use ++ (blocking!)
```

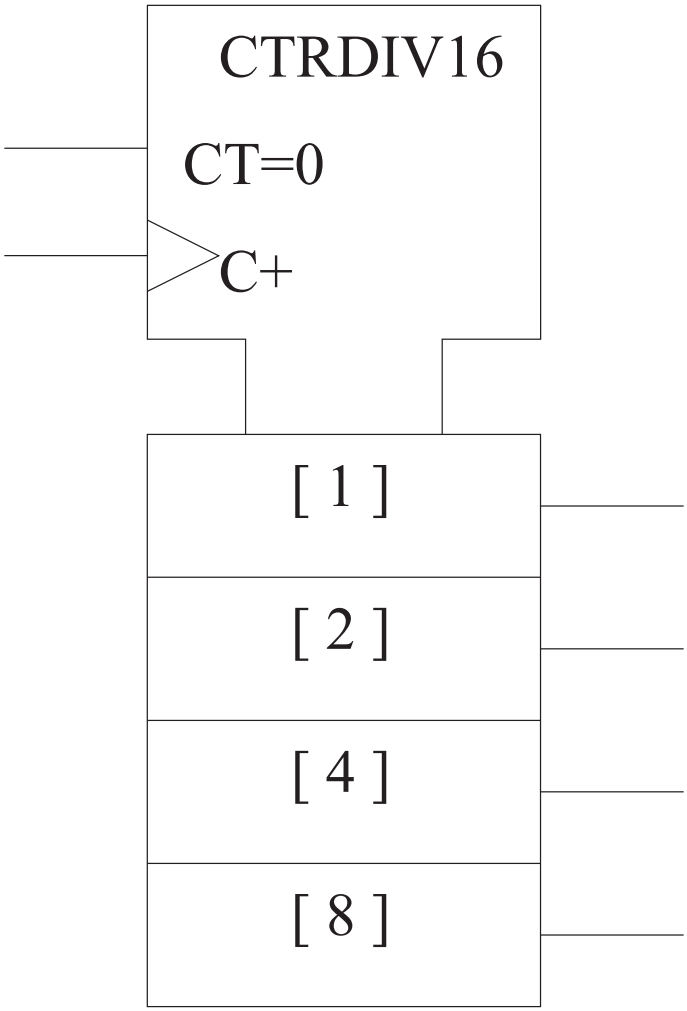



Figure 5.10: Binary counter.

```
endmodule
```

Note that the + operator does not generate a carry out. Thus when the counter has reached its maximum integer value (all 1s) the next clock edge will cause the counter to 'wrap round' and its next value will be zero (all 0s). We could modify the counter to generate a carry out, but in general counters are usually designed to detect the all-1s state and to output a signal when that state is reached. A carry out signal would be generated one clock cycle later. It is trivial to modify this counter to count down, or to count by a value other than one (possibly defined by a parameter – see the exercises at the end of this chapter).

The advantage of describing a counter in SystemVerilog is that the underlying combinational next state logic is hidden. For a counter with eight or more bits, the combinational logic can be very complex, but a synthesis system will generate that logic automatically. A simpler form of binary counter is the ripple counter. An example of a ripple counter using T flip-flops is described in SystemVerilog below, using the T flip-flop of Section 5.3.

```
module ripple_counter #(parameter N = 8)
                       (output logic [N-1:0] count,
                        input logic n_reset, clk);

  logic [N:1] Ca;
  genvar i;

  tffr t0 (count[0], Ca[1], '1, clk, n_reset);

  generate for (i = 1; i < N; i++)
    begin : t_loop
    tffr ti (count[i], Ca[i+1], '1, Ca[i], n_reset);
    end
  endgenerate

endmodule
```

Note that the $T$ input is held at a constant value in the description. When simulated using the T flip-flop model, above, this circuit behaves identically to the RTL model.

The ripple counter is, however, asynchronous. The second flip-flop is clocked from the Q output of the first flip-flop, as shown in Figure 5.11. A change in this output is delayed relative to the clock. Hence, the second

flip-flop is clocked by a signal behind the true clock. With further counter stages, the delay is increased. Further, incorrect intermediate values are generated. Provided the clock speed is sufficiently slow, a ripple counter can be used instead of a synchronous counter, but in many applications a synchronous counter is preferred.

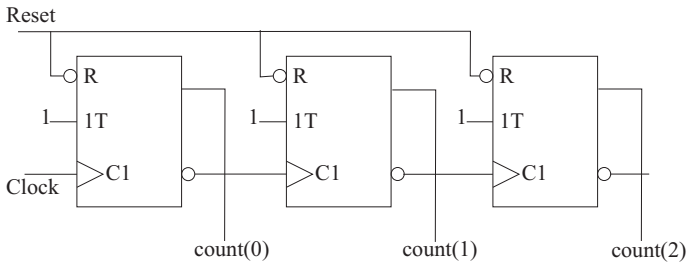



Figure 5.11: Ripple counter.

### 5.5.2 Johnson counter

A Johnson counter (also known as a Möbius counter – after a Möbius strip: a strip of paper formed into a circle with a single twist, resulting in a single surface) is built from a shift register with the least significant bit inverted and fed back to the most significant bit, as shown in Figure 5.12.

An $n$-bit binary counter has $2^n$ states. An $n$-bit Johnson counter has $2n$ states. The advantage of a Johnson counter is that it is simple to build (like a ripple counter), but is synchronous. The disadvantage is the large number of unused states that form an autonomous counter in their own right. In other words, we have the intended counter and a *parasitic* state machine coexisting in the same hardware. Normally, we should be unaware of the parasitic state machine, but if the system somehow entered one of the unused states, the subsequent behaviour might be unexpected. A SystemVerilog description of a Johnson counter is shown below.

```
module johnson #(parameter N = 8)
                (output logic [N-1:0] q,
                 input logic clk, n_reset);

always_ff @(posedge clk, negedge n_reset)
  if (~n_reset)
    q <= '0;
  else
    q <= {~q[0], q[N-1:1]};
```

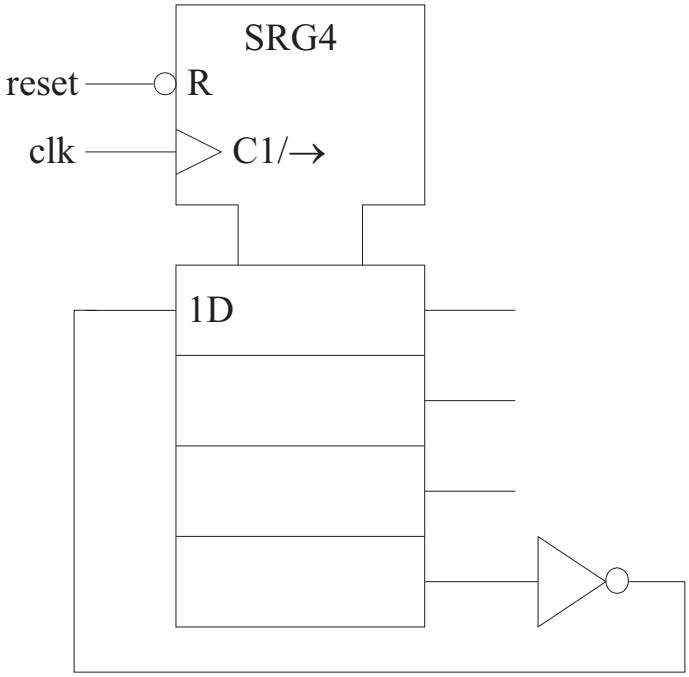



Figure 5.12: Johnson counter.

**endmodule**

The counting sequence of a 4-bit counter, together with the sequence belonging to the parasitic state machine, is shown in the table below. Whatever the size of $n$, the unused states form a single parasitic counter with $2^n - 2n$ states.

| Normal counting sequence | Parasitic counting sequence |
|---|---|
| 0000 | 0010 |
| 1000 | 1001 |
| 1100 | 0100 |
| 1110 | 1010 |
| 1111 | 1101 |
| 0111 | 0110 |
| 0011 | 1011 |
| 0001 | 0101 |

Both sequences repeat but do not intersect at any point. The parasitic set of states of a Johnson counter should never occur, but if one of the states did occur somehow, perhaps because of a power supply glitch

or because of some asynchronous input, the system can never return to its normal sequence. One solution to this is to make the counter *self-correcting*. It would be possible to detect every one of the parasitic states and to force a synchronous reset, but for an $n$-bit counter that is difficult. An easier solution is to note that the only legal state with a 0 in both the most significant and least significant bits is the all zeros state. On the other hand, three of the parasitic states have zeros in those positions. Provided that we are happy to accept that if the system does enter an illegal state it does not have to correct itself immediately, but can re-enter the normal counting sequence after 'a few' clock cycles, we can simply detect any states that have a 0 at the most and least significant bits and force the next state to be '1000' or its $n$-bit equivalent.

```
module scjohnson #(parameter N = 8)
                  (output logic [N-1:0] q,
                   input logic clk, n_reset);

always_ff @(posedge clk, negedge n_reset)
  if (~n_reset)
    q <= '0;
  else
    if (~q[N-1] & ~q[0])
      q <= {1'b1, {(N-1){1'b0}}};
    else
      q <= {~q[0], q[N-1:1]};

endmodule
```

### 5.5.3 Linear feedback shift register

Another counter that is simple in terms of next state logic is the linear feedback shift register (LFSR). This has $2^n - 1$ states in its normal counting sequence. The sequence of states appears to be random, hence the other name for the register: pseudo-random sequence generator (PRSG). The next state logic is formed by exclusive OR gates as shown in Figure 5.13.

There are a large number of possible feedback connections for each value of $n$ that give the maximal length ($2^n - 1$) sequence, but it can be shown that no more than four feedback connections (and hence three exclusive OR gates) are ever needed. The single state missing from the sequence is the all-0s state. Hence the asynchronous initialization should

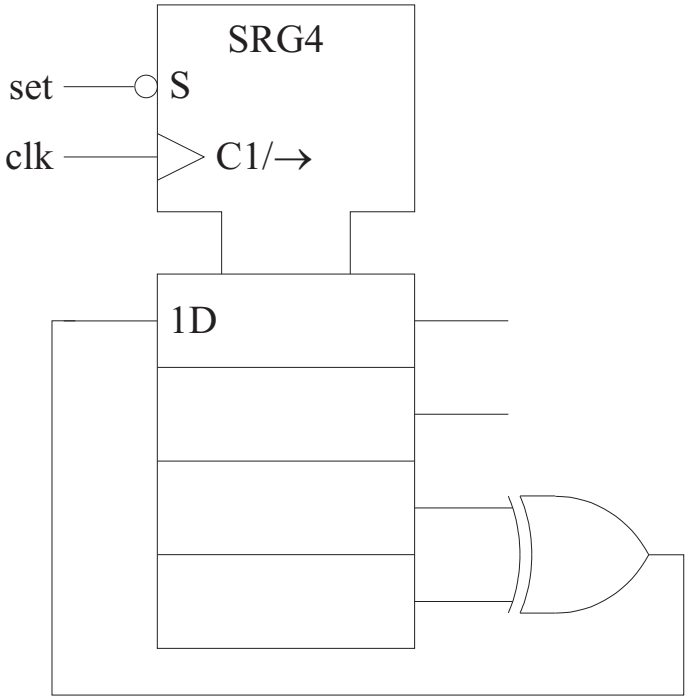



Figure 5.13: LFSR.

be a 'set'. As with the Johnson counter, the LFSR could be made self-correcting. A SystemVerilog model of an LFSR valid for certain values of $n$ is shown below.

The main advantage of using an LFSR as a counter is that nearly the full range of possible states ($2^n - 1$) can be generated using simple next state logic. Moreover, the pseudo-random sequence can be exploited for applications such as coding.

In the SystemVerilog model, the feedback connections for LFSRs with 1 to 36 stages are defined by a constant function, taps. Note that the model is only defined for the range 1 to 36 (with a default value of 8). Any attempt to use this model for a larger LFSR would result in an invalid model. The function defines up to three feedback connections – it is assumed that bit 0 is always used. The positions corresponding to the feedback connections are set to 1, using a shift operator and OR-ing theses values with the initial, all 0s value. The value returned by the function is kept in a **localparam**. The function is evaluated once, at elaboration time, and the resulting value is used to configure the model. A constant declared as a **localparam** is used only to parameterize a model and does not correspond to any hardware feature.

To construct the feedback connection for a particular size of LFSR, the stages of the LFSR referenced in the tapsreg vector are XORed together using a **for** loop.

```
module lfsr #(parameter N = 4)
             (output logic [N-1:0] q,
              input logic clock, n_set);

  logic feedback;
  int i;
  //localparam [N-1:0] tapreg = taps(N);
  logic [N-1:0] taps;

//function [N-1:0] taps (input int N);
initial
begin
  taps = '0;
  case (N)
   2: taps |= (1'b1 << 1);
   3: taps |= (1'b1 << 1);
   4: taps |= (1'b1 << 1);
   5: taps |= (1'b1 << 2);
   6: taps |= (1'b1 << 1);
   7: taps |= (1'b1 << 1);
   8: taps |= ((1'b1 << 6) | (1'b1 << 5)
              | (1'b1 << 1));
   9: taps |= (1'b1 << 4);
  10: taps |= (1'b1 << 3);
  11: taps |= (1'b1 << 2);
  12: taps |= ((1'b1 << 7) | (1'b1 << 4)
              | (1'b1 << 3));
  13: taps |= ((1'b1 << 4) | (1'b1 << 3)
              | (1'b1 << 1));
  14: taps |= ((1'b1 << 12) | (1'b1 << 11)
              | (1'b1 << 1));
  15: taps |= (1'b1 << 1);
  16: taps |= ((1'b1 << 5) | (1'b1 << 3)
              | (1'b1 << 2));
  17: taps |= ((1'b1 << 3));
  18: taps |= ((1'b1 << 7));
  19: taps |= ((1'b1 << 6) | (1'b1 << 5)
```

```
                | (1'b1 << 1));
  20: taps |= (1'b1 << 3);
  21: taps |= (1'b1 << 2);
  22: taps |= (1'b1 << 1);
  23: taps |= (1'b1 << 5);
  24: taps |= ((1'b1 << 4) | (1'b1 << 3)
                | (1'b1 << 1));
  25: taps |= (1'b1 << 3);
  26: taps |= ((1'b1 << 8) | (1'b1 << 7)
                | (1'b1 << 1));
  27: taps |= ((1'b1 << 8) | (1'b1 << 7)
                | (1'b1 << 1));
  28: taps |= (1'b1 << 3);
  29: taps |= (1'b1 << 2);
  30: taps |= ((1'b1 << 16) | (1'b1 << 15)
                | (1'b1 << 1));
  31: taps |= (1'b1 << 3);
  32: taps |= ((1'b1 << 28) | (1'b1 << 27)
                | (1'b1 << 1));
  33: taps |= (1'b1 << 13);
  34: taps |= ((1'b1 << 15) | (1'b1 << 14)
                | (1'b1 << 1));
  35: taps |= (1'b1 << 2);
  36: taps |= (1'b1 << 11);
  endcase
end
//endfunction

always_ff @(posedge clock, negedge n_set)
  if (~n_set)
    q <= '1;
  else
    q <= {feedback, q[N-1:1]};

always_comb
  begin
  feedback = q[0];
  for (i = 1; i <= N - 1; i++)
    //if (tapreg[i])
    if (taps[i])
      feedback ^= q[i];
```

```
  end

endmodule
```

## 5.6 Memory

Computer memory is often classified as ROM (read-only memory) and RAM (random access memory). These are to some extent misnomers – ROM is random access and RAM is better thought of as read and write memory. RAM can further be divided into SRAM (static RAM) and DRAM (dynamic RAM). Static RAM retains its contents while power is applied to the system. Dynamic RAM uses capacitors to store bits, which means that the capacitance charge can leak away with time. Hence DRAM needs refreshing intermittently.

### 5.6.1 ROM

The contents of a ROM chip are defined once. Hence we can use a constant array to model a ROM device in SystemVerilog. Below is the seven-segment decoder from Chapter 4 described as a ROM.

```
module sevensegrom(output logic [6:0] data,
                   input logic [3:0] address);

  logic [6:0] rom [0:15] = {7'b1110111, //0
                            7'b0010010, //1
                            7'b1011101, //2
                            7'b1011011, //3
                            7'b0111010, //4
                            7'b1101011, //5
                            7'b1101111, //6
                            7'b1010010, //7
                            7'b1111111, //8
                            7'b1111011, //9
                            7'b1101101, //E 10
                            7'b1101101, //E 11
                            7'b1101101, //E 12
                            7'b1101101, //E 13
                            7'b1101101, //E 14
                            7'b1101101}; //E 15
```

```
always_comb
  data = rom[address];
endmodule
```

Because no values can be written into the ROM, we can think of the device as combinational logic. In general, combinational logic functions can be implemented directly in ROM. Programmable forms of ROM are available (EPROM – electrically programmable ROM), but such devices require the application of a large negative voltage (–12 V) to a particular pin of the device. Such functionality is not modelled, as it does not form part of the normal operating conditions of the device.

### 5.6.2 Static RAM

A static RAM may be modelled in much the same way as a ROM. Because data may be stored in the RAM as well as read from it, the data signal is declared to be a port of type **inout** and because it can be put into a high impedance state it is delared as a **wire**. In addition, three control signals are provided. The first, CS (Chip Select) is a general control signal to enable a particular RAM chip. The address range, in this example, is 0 to 15. If we were to use, say, four identical chips to provide RAM with an address range of 0 to 63 (6 bits), the upper two address bits would be decoded such that at any one time exactly one of the RAM chips is enabled by its CS signal. Hence if the CS signal is not enabled the data output of the RAM chip should be in the high-impedance state. The other two signals are OE (Output Enable) and WE (Write Enable). Only one of these two signals should be asserted at one time. Data is either read from the RAM chip when the OE signal is asserted, or written to the chip if the WE signal is asserted. If neither signal is asserted, the output remains in the high-impedance state. All the control signals are active low.

Like in the ROM, the memory array is modelled as an array.

```
module RAM16x8(inout wire [7:0] Data,
               input logic [3:0]Address,
               input logic n_CS, n_WE, n_OE);

  logic [7:0] mem [0:15];

  assign Data = (~n_CS & ~n_OE) ? mem[Address] : 'z;

  always_latch
```

```
    if (~n_CS & ~n_WE & n_OE)
      mem[Address] <= Data;

endmodule
```

### 5.6.3 Synchronous RAM

The static RAM model is asynchronous and intended for modelling separate memory chips. Sometimes we wish to allocate part of an FPGA as RAM. In order for this to be synthesized correctly and for ease of use, it is best to make this RAM synchronous. Depending on the technology, there may be a variety of possible RAM structures, e.g. synchronous read, dual-port. Here, we will simply show how a basic synchronous RAM is modelled. This parameterisable example can be synthesized in most programmable technologies.

```
module SyncRAM #(parameter M = 4, N = 8)
  (output logic [N-1:0] Qout,
   input logic [M-1:0] Address,
   input logic [N-1:0]Data, input logic WE, Clk);

  logic [N-1:0] mem [0:(1 << M)-1];

always_comb
  Qout = mem[Address];

always_ff @(posedge Clk)
  if (~WE)
    mem[Address] <= Data;

endmodule
```

The structure of this code is almost identical to that of a flip-flop with an enable – in this case, the enable signal is the WE input. As with the SRAM example above, the Address input is interpreted as an unsigned integer to reference an array. This example could be extended to include an output enable and chip select, as above.

## 5.7 Sequential multiplier

Let us consider a multiplier for two's complement binary numbers. Multiplication, whether decimal or binary, can be broken down into a sequence of additions. A SystemVerilog statement such as

```
q = a * b;
```

where a and b are $n$-bit representations of (positive) integers, would be interpreted by a SystemVerilog synthesis tool as a combinational multiplication requiring $n^2$ full adders. If a and b are two's complement numbers, there also needs to be a sign adjustment. A combinational multiplier would take up a significant percentage of an FPGA for $n$=8 and would require many FPGAs for $n$=16.

The classic trade-off in digital design is between area and speed. In this case, we can significantly reduce the area required for a multiplier if the multiplication is performed over several clock cycles. Between additions, one of the operands of a multiplication operation has to be shifted. Therefore a multiplier can be implemented as a single $n$-bit adder and a shift register.

Two's complement numbers present a particular difficulty. It would be possible, but undesirable, to recode the operands as unsigned numbers with a sign bit. Booth's algorithm tackles the problem by treating an operand as a set of sequences of all 1s and all 0s. For example, –30 is represented as 100010. This is equal to $-2^5 + 2^2 - 2^1$. In other words, as each bit is examined in turn, from left to right, only a change from a 1 to a 0 or a 0 to a 1 is significant. Hence, in multiplying b by a, each pair of bits of a is examined, so that if $a_i = 0$ and $a_{i-1} = 1$, b shifted by $i$ places is added to the partial product. If $a_i = 1$ and $a_{i-1} = 0$, b shifted by $i$ places is subtracted from the partial product. Otherwise no operation is performed. The SystemVerilog model below implements this algorithm, but note that instead of shifting the operand to the left, the partial product is shifted to the right at each clock edge. A ready flag is asserted when the multiplication is complete.

```
module booth #(parameter AL = 8, BL = 8, QL = AL+BL)
  (output logic [QL-1:0] qout, output logic ready,
   input logic [AL-1:0]ain, input logic [BL-1:0] bin,
   input logic clk, load, n_reset);

  logic [clog2(AL):0] count;
  logic [BL-1:0] alu_out;
  logic a_1;
```

```
  function int clog2(input int n);
    begin
    clog2 = 0;
    n--;
    while (n > 0)
      begin
          clog2++;
          n >>= 1;
          end
    end
  endfunction

always_ff @(posedge clk, negedge n_reset)
  if (~n_reset)
    begin
    qout <= '0;
    a_1 <= '0;
    end
  else if (load)
    begin
    qout <= ain;
    a_1 <= '0;
    end
  else if (count > 0)
    begin
    a_1 <= qout[0];
    qout <= {alu_out[BL-1],alu_out[BL-1:0],
             qout[AL-1:1]};
    end

always_ff @(posedge clk, negedge n_reset)
  if (~n_reset)
    count <= 0;
  else if (load)
    count <= AL;
  else
    count <= count - 1;

always_comb
  case ({qout[0], a_1})
```

```
    2'b01: alu_out = qout[QL-1:AL] + bin;
    2'b10: alu_out = qout[QL-1:AL] - bin;
    default: alu_out = qout[QL-1:AL];
  endcase

always_comb
  if (~load & !count)
    ready = '1;
  else
    ready = '0;

endmodule
```

## Testbenches for sequential building blocks

## Summary

In this chapter we have discussed a number of common sequential building blocks. SystemVerilog models of these blocks have been written using processes. Most of these models are synthesizable using RTL synthesis tools. We have also considered examples of testbenches for sequential circuits.

## Further reading

As with combinational blocks, manufacturers' data sheets are a good source of information about typical SSI devices. In particular, it is worthwhile to look in some detail at the timing specifications for SRAM and DRAM devices. The multiplier is an example of how relatively complicated computer arithmetic can be performed. Hennessey and Patterson have a good description of computer arithmetic units.

## Exercises

5.1 Show how positive edge-triggered behaviour can be described in SystemVerilog.

5.2 Write a behavioural SystemVerilog model of a negative edge triggered D flip-flop with set and clear.

5.3 Write a SystemVerilog model of a negative edge-triggered T-type flip-flop.

5.4 Write a SystemVerilog model of a 10-state synchronous counter that asserts an output when the count reaches 10.

5.5 Write a SystemVerilog model of an N-bit counter with a control input "Up". When the control input is '1' the counter counts up; when it is '0' the counter counts down. The counter should not, however, wrap round. When the all '1's or all '0's states are reached the counter should stop.

5.6 Write a SystemVerilog model of an N-bit parallel to serial converter.

5.7 Write a SystemVerilog testbench for this parallel to serial converter.

5.8 What are the advantages and disadvantages of ripple counters as opposed to synchronous counters?

5.9 Design a synchronous Johnson counter that visits eight distinct states in sequence. How would this counter be modified such that any unused states lead, eventually to the normal counting sequence?

5.10 Design an LFSR which cycles through the following states:

001,010,101,011,111,110,100

Verify your design by simulation.

5.11 Explain the function of the device shown in Figure 5.14. Your answer should include a description of all the symbols.

5.12 Show, with a full circuit diagram, how the device of Figure 5.14 could be used to build a synchronous counter with 12 states. Show how a synchronous reset can be included.

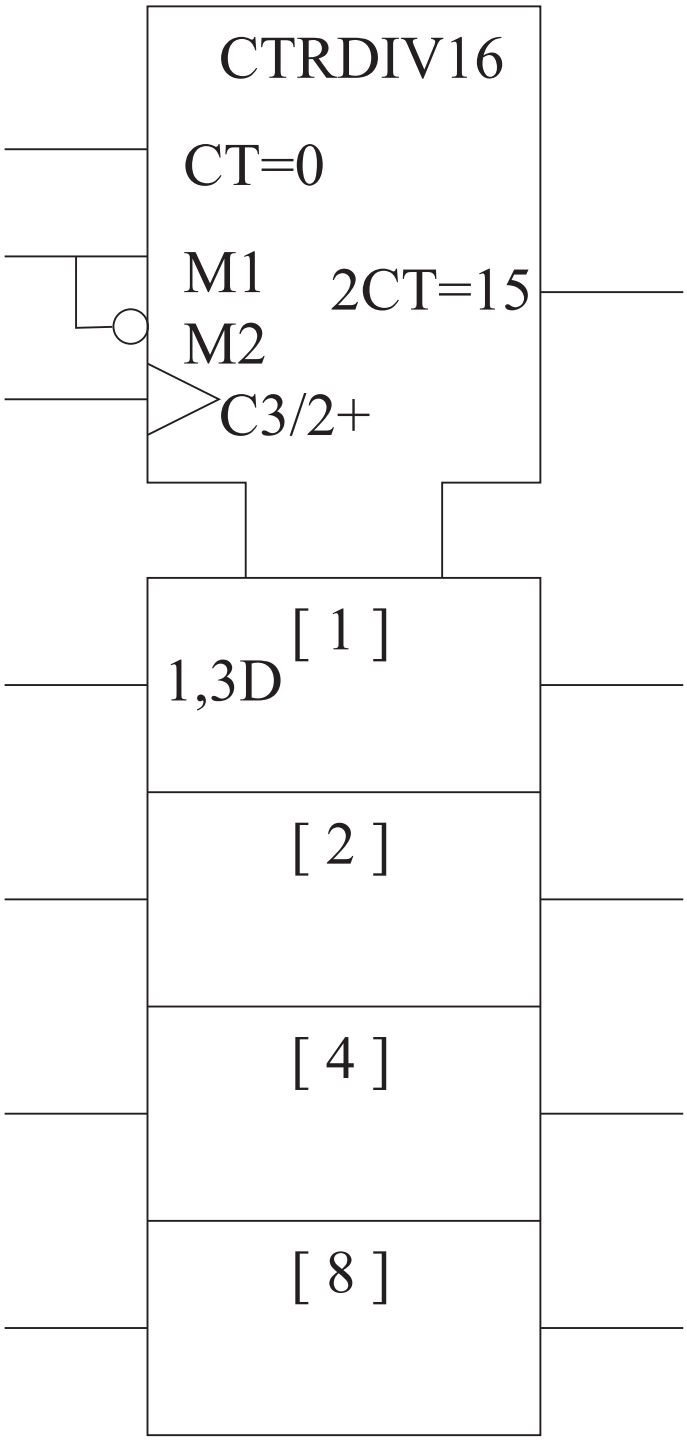



Figure 5.14: Device for Exercises 5.11 and 5.12.

# Chapter 6

# Synchronous Sequential Design

We have so far looked at combinational and sequential building blocks. Real digital systems are usually synchronous sequential systems. In this chapter we will explain how general synchronous sequential systems are designed. We will then describe how such systems may be modelled in SystemVerilog.

## 6.1 Synchronous sequential systems

Almost all large digital systems have some concept of state built into them. In other words, the outputs of a system depend on past values of its inputs as well as the present values. Past input values are either stored explicitly or cause the system to enter a particular state. Such systems are known as *sequential* systems, as opposed to *combinational* systems. A general model of a sequential system is shown in Figure 6.1. The present state of the system is held in the registers – hence the outputs of the registers give the value of the present state and the inputs to the registers will be the next state.

The present state of the system can either be updated as soon as the next state changes, in which case the system is said to be *asynchronous*, or the present state can be updated only when a clock signal changes, which is *synchronous* behaviour. In this chapter, we shall describe the design of synchronous systems. In general, synchronous design is easier than asynchronous design and so we will leave discussion of the latter topic until Chapter 12.

In this chapter we will consider the design of synchronous sequential systems. Many real systems are too complex to design in this way, thus in Chapter 7 we will show that more complex designs can be partitioned.

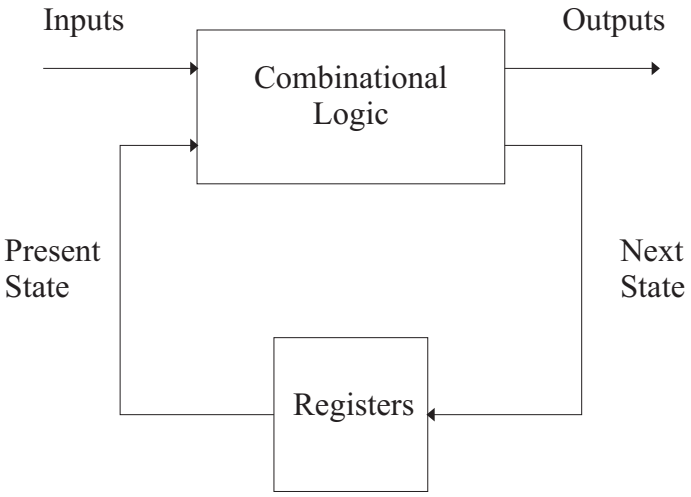



Figure 6.1: General sequential system

Nevertheless, the formal design methods described in this chapter must be applied to at least part of the design of larger systems. In the next section, we will introduce, by way of a simple example, a method of formally specifying such systems. We will then go on to describe the problems of state assignment, state minimization and the design of the next state and output logic. Throughout we will illustrate how designs can also be modelled using SystemVerilog.

## 6.2 Models of synchronous sequential systems

### 6.2.1 Moore and Mealy machines

There are two common models of synchronous sequential systems: the *Moore* machine and the *Mealy* machine. These are illustrated in Figure 6.2. Both types of system are triggered by a single clock. The next state is determined by some (combinational) function of the inputs and the present state. The difference between the two models is that in the Moore machine the outputs are solely a function of the present state, while in the Mealy machine the outputs are a function of the present state and of the inputs. Both the Moore and Mealy machines are commonly referred to as *state machines*. That is to say they have an internal state that changes.

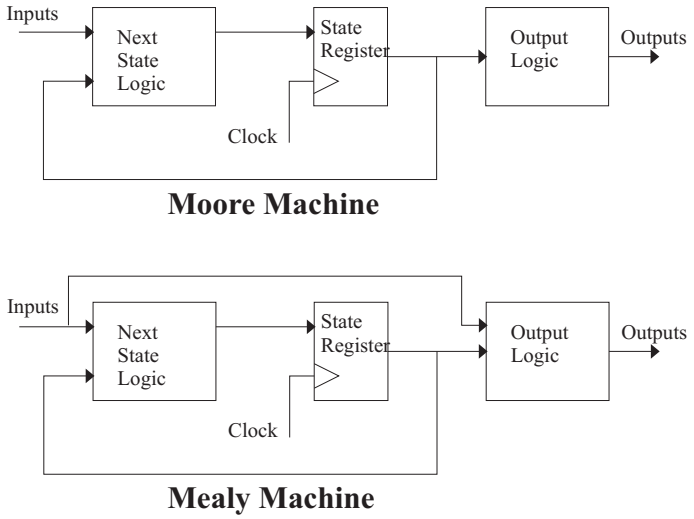



Figure 6.2: Moore and Mealy machines

### 6.2.2 State registers

As was seen in Chapter 2, combinational logic can contain hazards. The next state logic of the Moore and Mealy machines is simply a block of combinational logic with a number of inputs and a number of outputs. The existence of hazards in this next state logic could cause the system to go to an incorrect state. There are two ways to avoid such a problem: either the next state logic should include the redundant logic needed to suppress the hazard or the state machine should be designed such that a hazard is allowed to occur, but is ignored. The first solution is not ideal, as the next state logic is more complex; hence, the second approach is used. (Note that *asynchronous* systems are susceptible to hazards and the next state logic *must* prevent any hazards from occurring, which is one reason why synchronous systems are usually preferred.)

To ensure that sequential systems are able to ignore hazards, a clock is used to synchronize data. When the clock is invalid, any hazards that occur can be ignored. A simple technique, therefore, is to logically AND a clock signal with the system signals – when the clock is at logic 0, any hazards would be ignored. The system is, however, still susceptible to hazards while the clock is high. It is common, therefore, to use registers that are only sensitive to input signals while the clock is changing. The clock edge is very short compared with the period of the clock. Therefore, the data has only to be stable for the duration of the clock change, with small tolerances before and after the clock edge. These timing tolerance param-

eters are known as the setup and hold times ($t_{SETUP}, t_{HOLD}$) respectively, as shown in Figure 6.3.

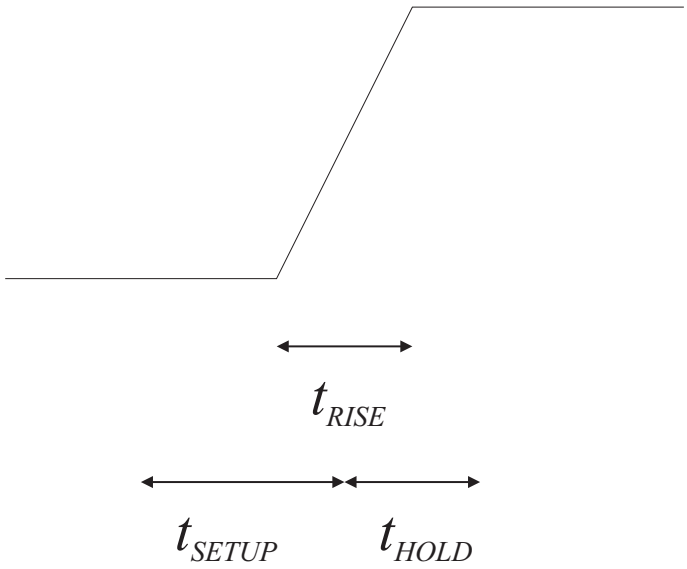


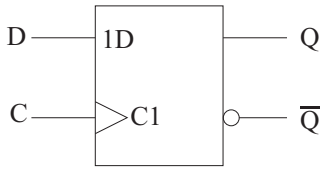

Figure 6.3: Setup and hold times

The state registers for a synchronous state machine are therefore edge-triggered elements. The symbol and truth table for a positive edge-triggered D type flip-flop are shown in Figure 6.4. The logic value at the$D$ input is stored in the flip-flop, and is available at the $Q$ output, at the rising clock edge. In the symbol, the triangle indicates edge-triggered behaviour. A negative edge-triggered flip-flop would have the clock signal inverted (using the usual circle). The notation $C1$, $1D$ shows the dependence of the $D$ input on the clock. In the truth table, the notation $Q^+$ is used to show the next state of $Q$ (i.e. after the next clock edge). An upward pointing arrow is used to show a rising edge. Flip-flops may also include asynchronous set or reset inputs, but these should only ever be used to initialize the system when it is first turned on. Asynchronous set and reset inputs should *never* be used during normal operation.

Other types of flip-flop exist and may be used to design synchronous systems, but they offer few advantages and are not common in programmable logic.

### 6.2.3 Design of a three-bit counter

In the next section, we will introduce a formal notation for synchronous sequential systems. First, however, we will consider the design of a sim-




| D | C | $Q^+$ | $\overline{Q}^+$ |
|---|---|---|---|
| 0 | ↑ | 0 | 1 |
| 1 | ↑ | 1 | 0 |
| – | 0 | Q | $\overline{Q}$ |
| – | 1 | Q | $\overline{Q}$ |

Figure 6.4: D type flip-flop

ple system that does not need a formal description. Let us design, using positive edge-triggered D flip-flops, a counter that, on rising clock edges, counts through the binary sequence from 000 to 111, at which point it returns to 000 and repeats the sequence.

The three bits will be labelled $A$, $B$ and $C$. The truth table is shown below, in which $A^+$, $B^+$ and $C^+$ are the next states of $A$, $B$ and $C$.

| $ABC$ | $A^+B^+C^+$ |
|---|---|
| 0 0 0 | 0 0 1 |
| 0 0 1 | 0 1 0 |
| 0 1 0 | 0 1 1 |
| 0 1 1 | 1 0 0 |
| 1 0 0 | 1 0 1 |
| 1 0 1 | 1 1 0 |
| 1 1 0 | 1 1 1 |
| 1 1 1 | 0 0 0 |

$A^+$ etc. are the *inputs* to the state register flip-flops; $A$ etc. are the outputs. Therefore the counter has the structure shown in Figure 6.5. The design task is thus to derive expressions for $A^+$, $B^+$ and $C^+$ in terms of $A$, $B$ and $C$. From the truth table, above, K-maps can be drawn, as shown in Figure 6.6. Hence the following expressions for the next state variables can be derived.

$$\begin{aligned} A^+ &= A.\bar{C} + A.\bar{B} + \bar{A}.B.C \\ B^+ &= B.\bar{C} + \bar{B}.C \\ C^+ &= \bar{C} \end{aligned}$$

The full circuit for the counter is shown in Figure 6.7.

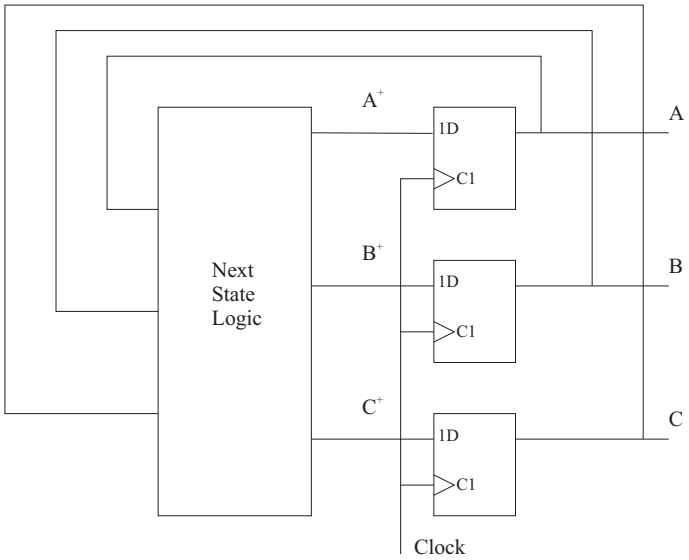



Figure 6.5: Structure of 3-bit counter

## 6.3 Algorithmic state machines

The counter designed in the last section could easily be described in terms of state changes. Most sequential systems are more complex and require a formal notation to fully describe their functionality. From this formal notation, a state table and hence Boolean expressions can be derived. There are a number of types of formal notation that may be used. We will briefly refer to one before introducing the principle technique used in this book – the *algorithmic state machine (ASM) chart*.

The form of an ASM chart is best introduced by an example. Let us design a simple controller for a set of traffic signals, as shown in Figure 6.8. This example is significantly simpler than a real traffic signal controller (and would probably be more dangerous than an uncontrolled junction!). The traffic signals have two lights each – red and green. The major road normally has a green light, while the minor road has a red light. If a car is detected on the minor road, the signals change to red for the major road and green for the minor road. When the lights change, a timer is started. Once that timer completes a 'TIMED' signal is asserted, which causes the lights to change back to their default state.

The functionality of this system can be described by the state machine diagram of Figure 6.9. This form of diagram is commonly used, but can be unclear. For some systems (e.g. that of Figure 11.19), such diagrams are sufficient. In this book, however, we will generally use ASM charts, which

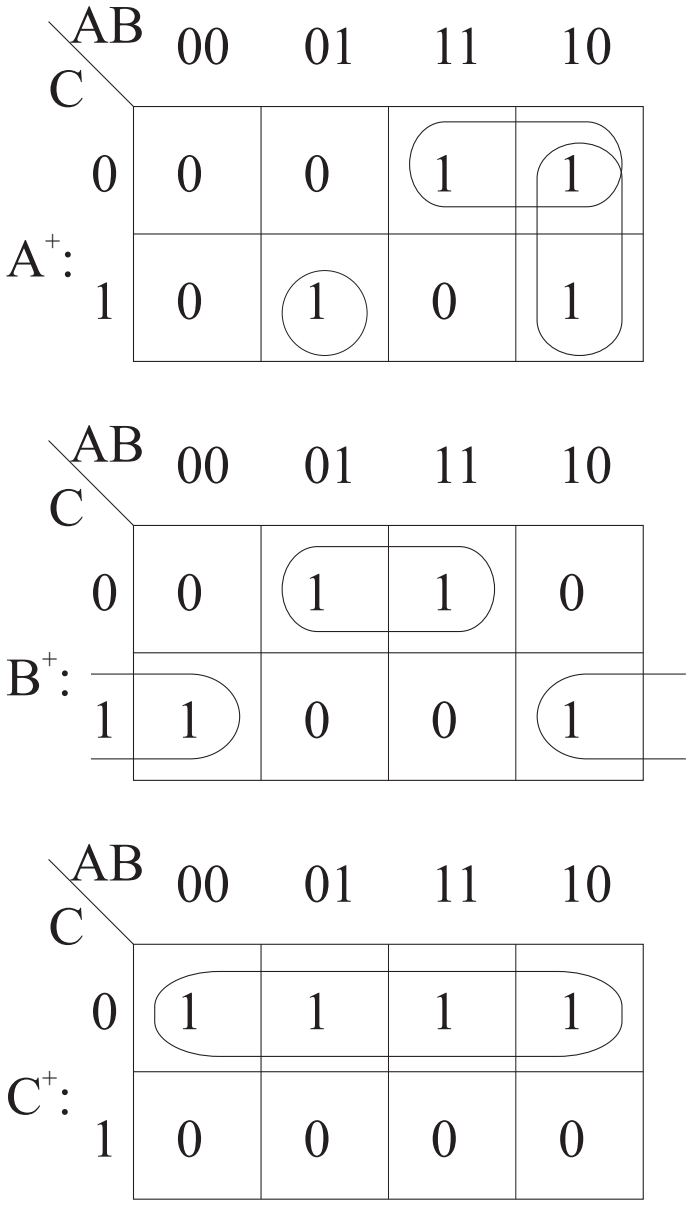



Figure 6.6: K-maps for 3-bit counter

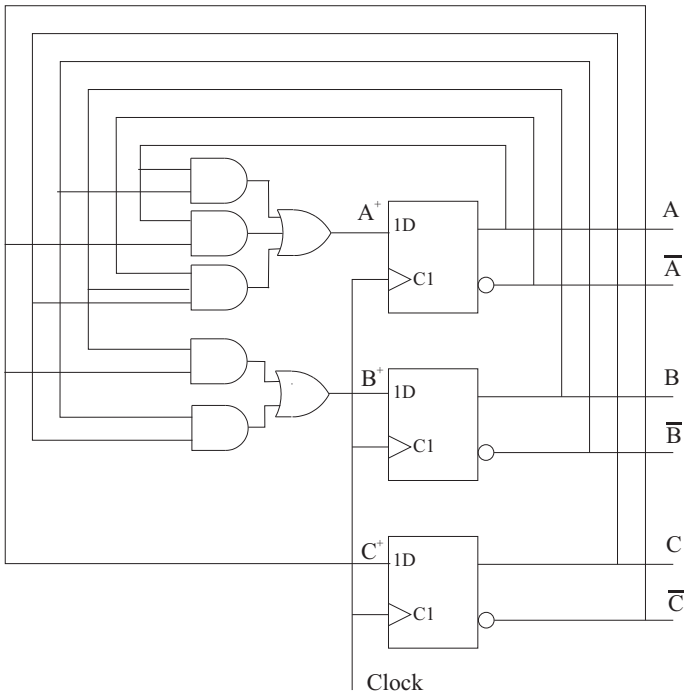



Figure 6.7: 3-bit counter circuit

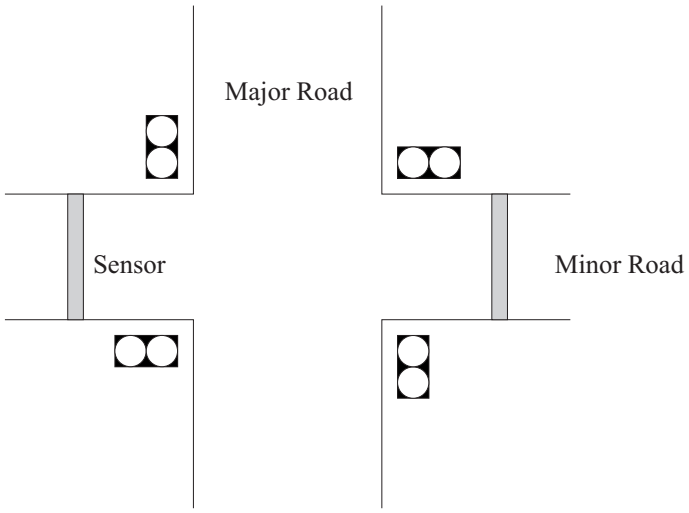



Figure 6.8: Traffic signal problem

are much less ambiguous. The ASM chart for the traffic signal controller is shown in Figure 6.10.

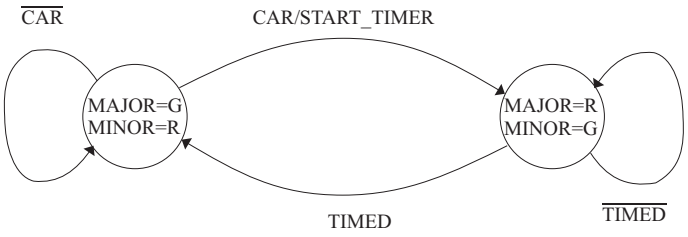



Figure 6.9: State machine of traffic signal controller

ASM charts resemble flow charts, but contain implicit timing information – the clock signal is not explicitly shown in Figure 6.10. It should be noted that ASM charts represent physical hardware. Therefore all transitions within the ASM chart must form closed paths – hardware cannot suddenly start or stop (the only exception to this might be a reset state to which the system never returns).

The basic component of an ASM chart is the state box, shown in Figure 6.11(a). The state takes exactly one clock cycle to complete. At the top left-hand corner the name of the state is shown. At the top right-hand corner the state assignment (see below) may be given. Within the state box, the output signals are listed. The signals take the values shown for the duration of the clock cycle and are reset to their default values for the next clock cycle. If a signal does not have a value assigned to it (e.g. $Y$), that signal is asserted (logic 1) during the state and is deasserted elsewhere. The notation $X \leftarrow 1$ means that the signal is assigned at the *end* of the state (i.e. during the next clock cycle) and holds its value until otherwise set elsewhere.

A decision box is shown in Figure 6.11(b). Two or more branches flow from the decision box. The decision is made from the value of one or more input signals. The decision box *must* follow and be associated with a state box. Therefore the decision is made in the same clock cycle as the other actions of the state. Hence the input signals must be valid at the start of the clock cycle.

A conditional output box is shown in Figure 6.11(c). A conditional output must follow a decision box. Therefore the output signals in the conditional output box are asserted in the same clock cycle as those in the state box to which it is attached (via one or more decision boxes). The output signals can change during that state as a result of input changes. The conditional output signals are sometimes known as Mealy outputs be-

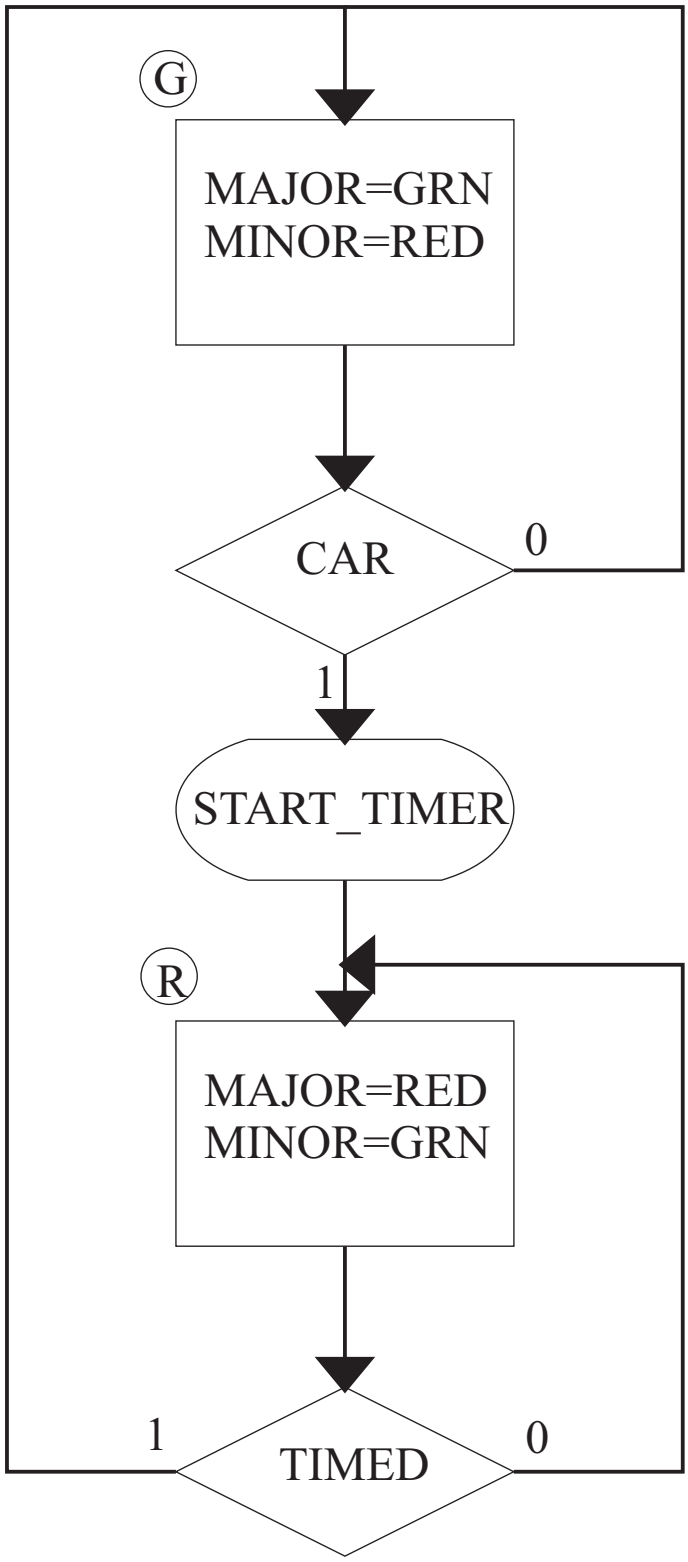



Figure 6.10: ASM chart of traffic signal controller

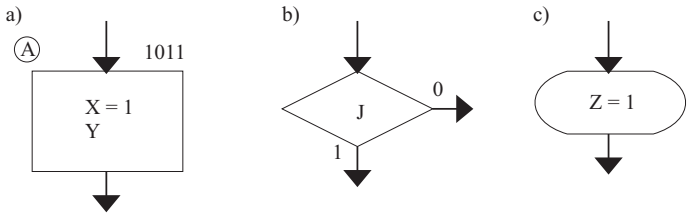



Figure 6.11: ASM chart symbols

cause they are dependent on input signals, as in a Mealy machine.

It can therefore be seen that one state, or clock cycle, consists of more than just the state box. Decision boxes and conditional output boxes also form part of the state. Figure 6.10 can be redrawn, as in Figure 6.12, where all the components of a state are enclosed within dashed lines.

The difference between state boxes and conditional output boxes is illustrated in Figure 6.13. In Figure 6.13(a), there are two states. Output $Y$ is asserted during the first state if input $C$ is true or becomes true. In Figure 6.13(b) there are three states. The difference can be seen in the timing diagrams of Figure 6.14.

## 6.4 Synthesis from ASM charts

### 6.4.1 Hardware implementation

An ASM chart is a description or specification of a synchronous sequential system. It is an abstract description in the sense that it describes *what* a system does, but not *how* it is done. Any given (non-trivial) ASM chart may be implemented in hardware in more than one way. The ASM chart can, however, be used as the starting point of the hardware synthesis process. To demonstrate this, an implementation of the traffic signal controller will first be designed. We will then use further examples to show how the state minimization and state assignment problems may be solved.

The ASM chart of Figure 6.10 may be equivalently expressed as a *state and output table*, as shown in Figure 6.15. The outputs to control the traffic signals themselves are not shown, but otherwise the state and output table contains the same information as the ASM chart. As we will see the state and output table is more compact than an ASM chart and is therefore easier to manipulate.

To implement this system in digital hardware, the abstract states, $G$ and $R$ have to be represented by Boolean variables. Here, the problem of

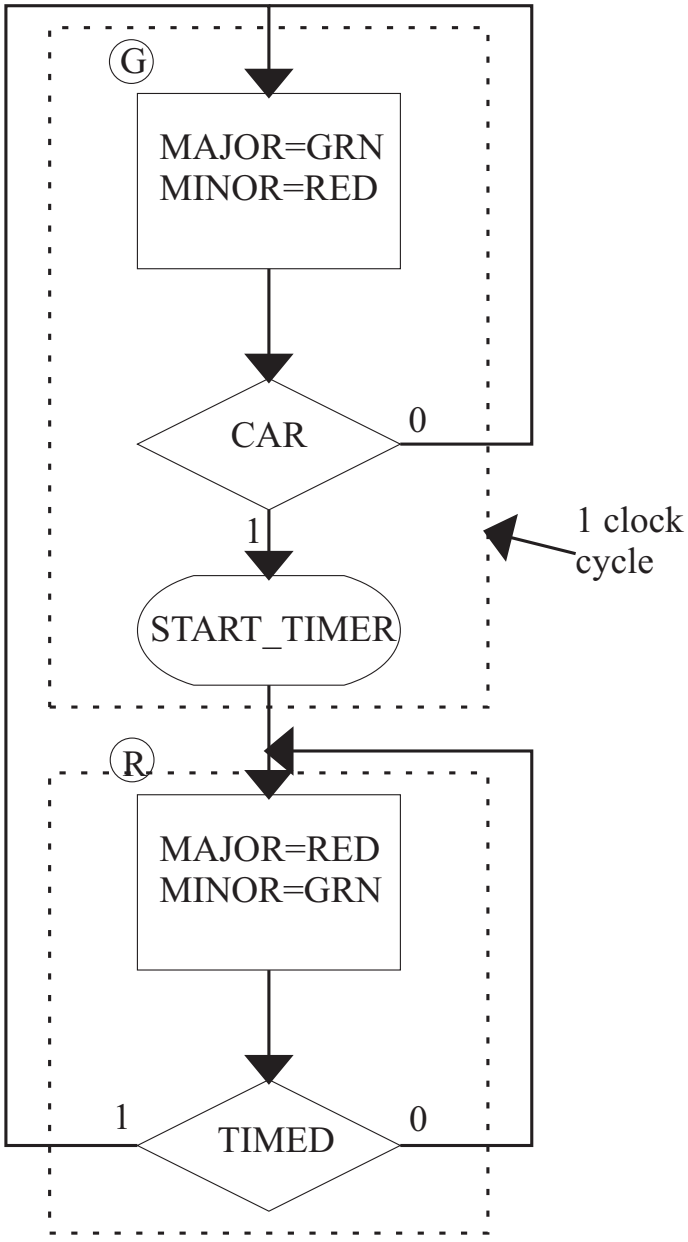



Figure 6.12: ASM chart showing clock cycles

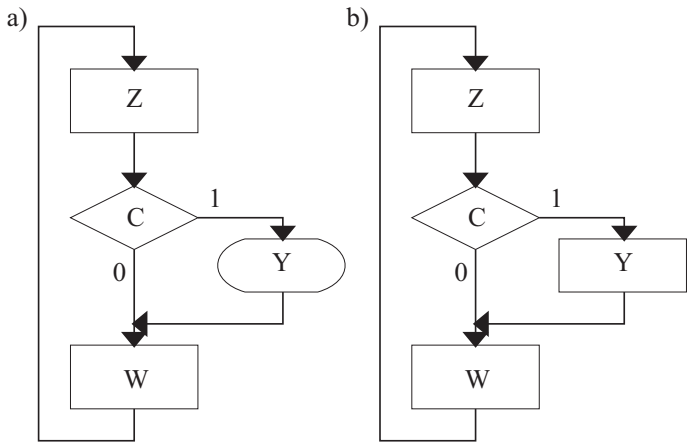



Figure 6.13: Conditional and unconditional outputs

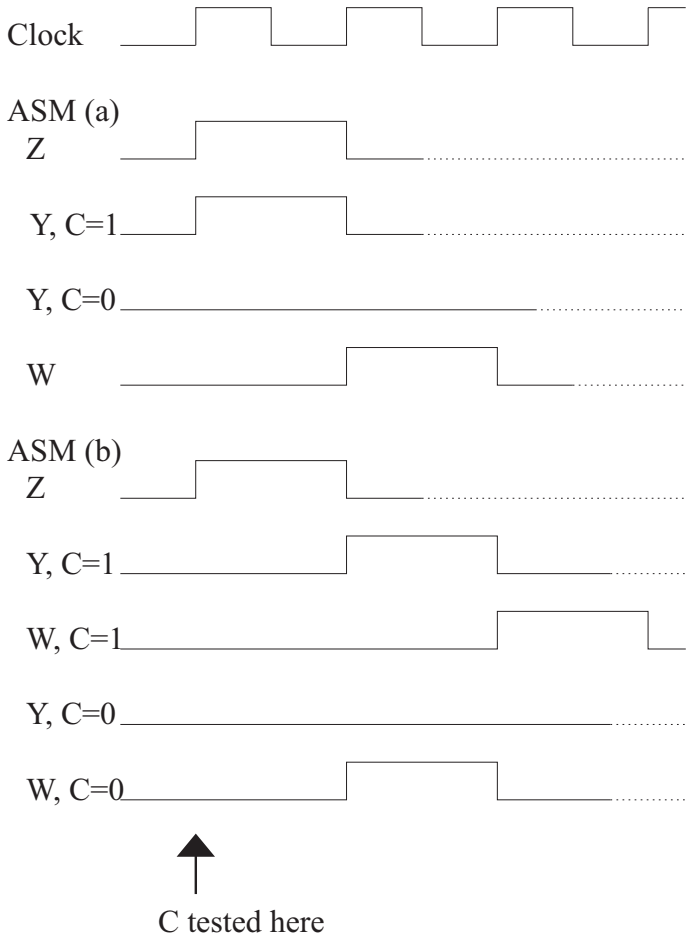



Figure 6.14: Timing diagram for Figure 6.13

| Present State | CAR, TIMED 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| G | G, 0 | G, 0 | R, 1 | R, 1 |
| R | R, 0 | G, 0 | G, 0 | R, 0 |

Next State, START_TIMER

Figure 6.15: State and output table

*state assignment* is nearly trivial. Two states can be represented by one Boolean variable. For example, when the Boolean variable, $A$, is 0 it can represent state $G$ and when it is 1, state $R$. It would be equally valid to use the opposite values. These values for $A$can be substituted into the state and output table to give the *transition and output table* shown in Figure 6.16.

| A | CAR, TIMED 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0, 0 | 0, 0 | 1, 1 | 1, 1 |
| 1 | 1, 0 | 0, 0 | 0, 0 | 1, 0 |

$A^+$, START_TIMER

Figure 6.16: Transition and output table

This transition and output table is effectively two K-maps superimposed on each other. These are explicitly shown in Figure 6.17. From these, expressions can be derived for the state variable and the output.

$$\begin{aligned} A^+ &= \bar{A}.CAR + A.\overline{TIMED} \\ START\_TIMER &= \bar{A}.CAR \end{aligned}$$

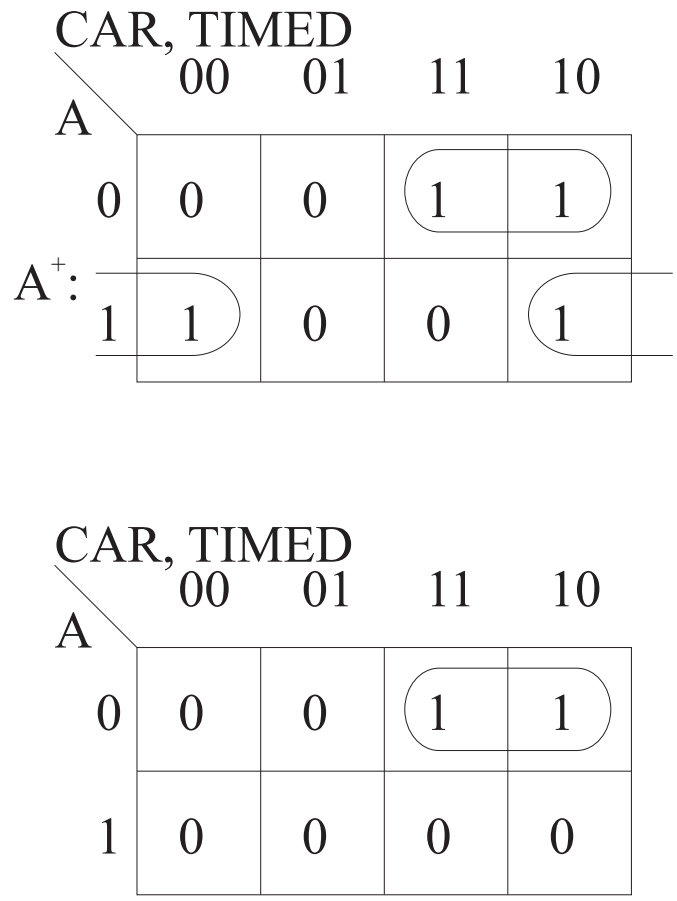



Figure 6.17: K-maps for traffic signal controller

For completeness, a hardware implementation is shown in Figure 6.18. The two flip-flop outputs can be used directly to control the traffic signals, so that when $A$ is 1 (and $\bar{A}$ is 0) the signal for the major road is green and the signal for the minor road is red. When $A$ is 0, the signals are reversed.

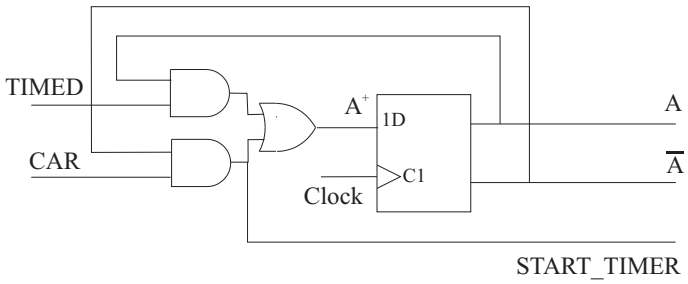



Figure 6.18: Circuit for traffic signal controller

### 6.4.2 State assignment

In the previous example there were two possible ways to assign the abstract states, $G$ and $R$, to the Boolean state variable, $A$. With more states, the number of possible state assignments increases. In general, if we want to code $s$ states using a minimal number of D flip-flops, we need $m$ Boolean variables, where $2^{m-1} < s \leq 2^m$. The number of possible assignments is given by

$$\frac{(2^m)!}{(2^m - s)!}.$$

This means, for example, that there are 24 ways to encode three states using two Boolean variables and 6720 ways to encode five states using three Boolean variables. In addition, there are possible state assignments that use more than the minimal number of Boolean variables, which may have advantages under certain circumstances. There is no known method for determining in advance which state assignment is 'best' in the sense of giving the simplest next state logic. It is obviously impractical to attempt every possible state assignment. Therefore a number of *ad hoc* guidelines can be used to perform a state assignment. Again, let us use an example to demonstrate this.

A synchronous sequential system has two inputs, $X$ and $Y$, and one output, $Z$. When the sum of the inputs is a multiple of 3, the output is true – it is false otherwise. The ASM chart is shown in Figure 6.19.

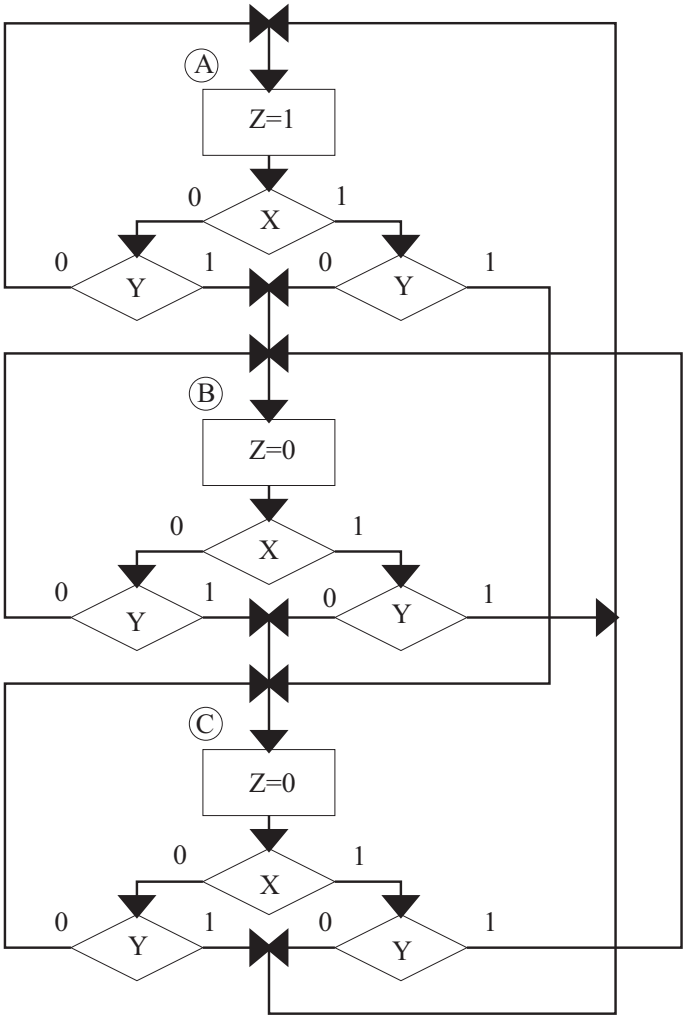



Figure 6.19: ASM chart for sequence detector

To encode the three states we need (at least) two state variables and hence two flip-flops. As noted above, there are 24 ways to encode three states; which should we use? We could arbitrarily choose any one of the possible state assignments, or we could apply one or more of the following guidelines:

- It is good practice to provide some means of initializing the state machine when power is first applied. This can be done using the asynchronous resets or sets on the system flip-flops. Therefore the first state (state $A$ in this example) can be coded as all 0s or all 1s.
- We can use the normal binary counting sequence for further states (e.g. $B$ becomes 01 and $C$ becomes 10).
- We can minimize the number of bits that change between states, e.g. by using a Gray code. (This doesn't help in this example as transitions exist from each state to every other state.)
- The states might have some particular meaning. Thus a state variable bit might be set in one state but in no others. (This can result in a non-minimal number of state variables but very simple output logic, which under some circumstances can be very desirable.)
- We can use one variable per state. For three states, we would have three state variables and hence three flip-flops. The states would be encoded as 001, 010 and 100. This is known as 'one-hot' encoding, as only one flip-flop is asserted at a time. Although this appears to be very non-optimal, there may be advantages to the one-hot (or 'one-cold') method. The next state logic may be relatively simple. In some forms of programmable logic, such as FPGAs, there is a very high ratio of flip-flops to combinational logic. A one-hot encoded system may therefore use fewer resources than a system with a minimal number of flip-flops. Furthermore, because exactly one flip-flop output is asserted at a time, it is relatively easy to detect a system malfunction in which this condition is not met. This can be very useful for safety-critical systems.

Let us therefore apply a simple state encoding to the example. The state and output table is shown in Figure 6.20(a) and the transition and output table is shown in Figure 6.20(b), where state $A$ is encoded as 00, $B$ as 01 and $C$as 11. The combination 10 is not used.

The fact that we have one or more unused combinations of state variables may cause a problem. These unused combinations are states of

a)

| | X, Y | | | |
|---|---|---|---|---|
| P | 00 | 01 | 11 | 10 |
| A | A, 1 | B, 1 | C, 1 | B, 1 |
| B | B, 0 | C, 0 | A, 0 | C, 0 |
| C | C, 0 | A, 0 | B, 0 | A, 0 |

$P^+, Z$

b)

| | X, Y | | | |
|---|---|---|---|---|
| $S_1S_0$ | 00 | 01 | 11 | 10 |
| 00 | 00, 1 | 01, 1 | 11, 1 | 01, 1 |
| 01 | 01, 0 | 11, 0 | 00, 0 | 11, 0 |
| 11 | 11, 0 | 00, 0 | 01, 0 | 00, 0 |

$S_1^+S_0^+, Z$

Figure 6.20: (a) State and output table; (b) transition and output table for sequence detector

the system. In normal operation, the system would never enter these 'unused states'. Therefore, in principle, we can treat the next state and output values as 'don't cares', as shown in Figure 6.21.

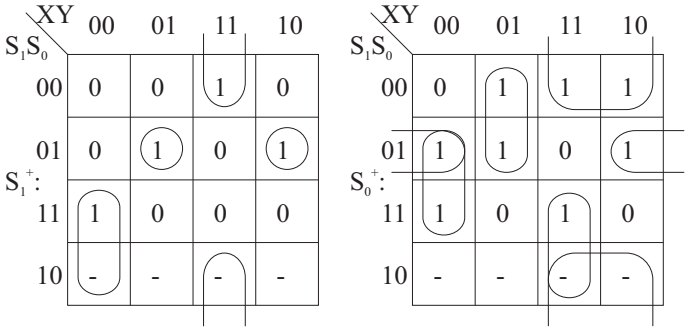



Figure 6.21: K-maps with don't cares

This gives the next state equations:

$$\begin{aligned} S_1^+ &= S_1.\bar{X}.\bar{Y} + \overline{S_1}.S_0.\bar{X}.Y + \overline{S_0}.X.Y + \overline{S_1}.S_0.X.\bar{Y} \\ S_0^+ &= S_0.\bar{X}.\bar{Y} + \overline{S_1}.\bar{X}.Y + \overline{S_0}.X + \overline{S_1}.S_0.\bar{Y} + S_1.X.Y \end{aligned}$$

The output expression can be read directly from the transition and output table:

$$Z = \overline{S_0}$$

By default, therefore, the transitions from the unused state have now been defined, as shown in Figure 6.22. Although this unused state should never be entered, it is possible that a 'non-logical' event, such as a glitch on the power supply might cause the system to enter this unused state. It can be seen from Figure 6.22 that if, for example, the inputs were both 0, the system would stay in the unused state. In the worst case, once having entered an unused state, the system might be stuck in one or more unused states. The unused states could therefore form a 'parasitic' state machine (or perhaps a 'parallel universe'!), causing the system to completely malfunction. We could, reasonably, decide that the chances of entering an unused state are so low as to be not worth worrying about. Hence we treat the transition table entries for the unused states as 'don't cares', as shown, which minimizes the next state logic. On the other hand, the system might be used in a safety-critical application. In this case, it might be important that all transitions from unused states are fully defined, so that

we can be certain to return to normal operation as soon as possible. In this case, the transitions from the unused state would not be left as 'don't cares' in the K-maps, but would be explicitly set to lead to, say, the all 0s state. Hence the 'X' entries in the K-maps of Figure 6.21 become '0's and the next state equations would be:

| | X, Y | | | |
|---|---|---|---|---|
| $S_1S_0$ | 00 | 01 | 11 | 10 |
| 00 | 00, 1 | 01, 1 | 11, 1 | 01, 1 |
| 01 | 01, 0 | 11, 0 | 00, 0 | 11, 0 |
| 11 | 11, 0 | 00, 0 | 01, 0 | 00, 0 |
| 10 | 10, 1 | 00, 1 | 11, 1 | 01, 1 |
| | $S_1^+S_0^+$, Z | | | |

Figure 6.22: Transition table implied by don't cares

$$
\begin{aligned}
S_1^+ &= S_1.S_0.\bar{X}.\bar{Y} + \overline{S_1}.S_0.\bar{X}.Y + \overline{S_1}.\overline{S_0}.X.Y + \overline{S_1}.S_0.X.\bar{Y} \\
S_0^+ &= S_0.\bar{X}.\bar{Y} + \overline{S_1}.\bar{X}.Y + \overline{S_1}.\overline{S_0}.X + \overline{S_1}.S_0.\bar{Y} + S_1.S_0.X.Y
\end{aligned}
$$

These equations are more complex than the previous set that includes the 'don't cares'; hence the next state logic would be more complex.

Therefore we have a choice: we can either assume that it is impossible to enter an unused state and minimize the next state equations by assuming the existence of 'don't cares'; or we can try to reduce the risk of becoming stuck in an unused state by explicitly defining the transitions from the unused states and hence have more complex next state logic.

### 6.4.3 State minimization

We have noted in the previous section that to encode $s$ states we need $m$ flip-flops, where $2^{m-1} < s \leq 2^m$. If we can reduce the number of states in

the system, we *might* reduce the number of flip-flops, hence making the system simpler. Such savings may not always be possible. For instance, the encoding of 15 states requires four flip-flops. If we reduced the number of states to nine, we would still need four flip-flops. So there would be no obvious saving and we would have increased the number of unused states, with the potential problems discussed in the previous section. As will be seen, state minimization is a computationally difficult task, and in many cases, it would be legitimate to decide that there would be no significant benefits and hence the task would not be worth performing.

State minimization is based on the observation that if two states have the same outputs and the same next states, given a particular sequence of inputs, it is not possible to distinguish between the two states. Hence the two states are considered to be *equivalent* and hence they may be merged, reducing the total number of states.

For example, let us design the controller for a drinks vending machine. A drink costs 40c. The machine accepts 20c and 10c coins (all other coins are rejected by the mechanics of the system). Once 40c have been inserted, the drink is dispensed. If more than 40c are inserted, all coins are returned. The machine has two lights: one to show that it is ready for the next transaction, and one to show that further coins need to be inserted. The ASM chart has been split into two parts (Figures 6.23 and 6.24) – the connections between the two parts are shown by circles with lower-case letters.

There are nine states in this state machine. Four flip-flops would therefore be required to implement it. If we could merge at least two states, we would save ourselves a flip-flop. From Figures 6.23 and 6.24 notice that states $F$, $G$ and $H$ all have transitions to state $I$ if a 20c coin is inserted and to state $B$ if a 10c coin is inserted. Otherwise all three states have transitions back to themselves. Intuitively, these three states would appear to be equivalent. Another way of looking at this is to say that states $F$, $G$ and $H$ all represent the condition where another 10c is expected to complete the sale of a drink. From the point of view of the purchaser, these states are indistinguishable.

Instead of attempting to manipulate the ASM chart, it is probably clearer to rewrite it as a state and output table (Figure 6.25). The ‘Other’ column shows the next state if no valid coin is inserted. Because there are no conditional outputs, it is possible to separate the outputs from the next state values.

The condition for two states to be considered equivalent is that their next states and outputs should be the same. States $A$, $B$ and $I$ have unique outputs and therefore cannot be equivalent to any other states.

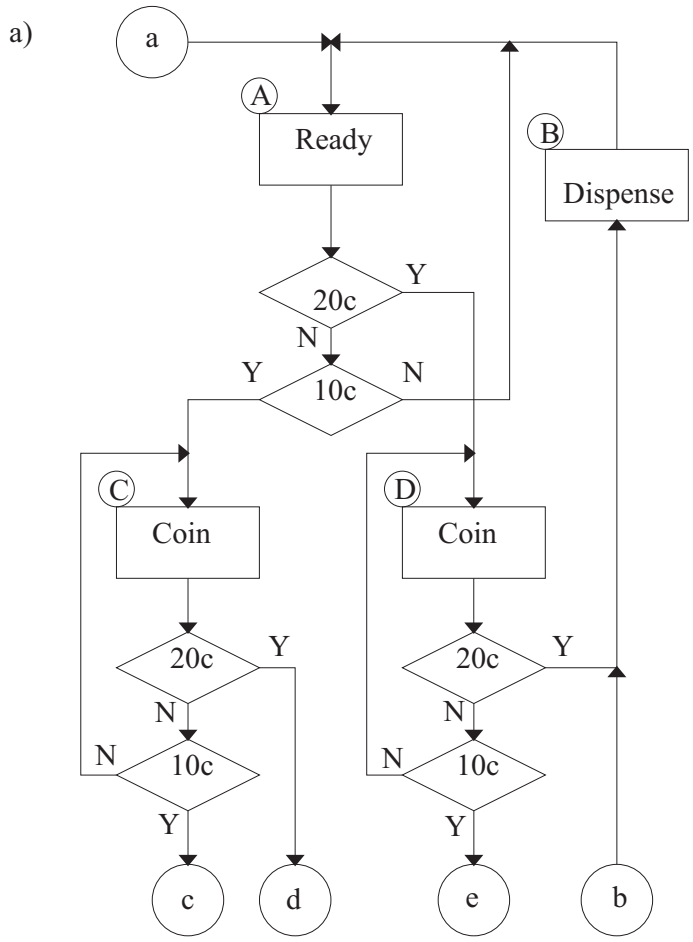



Figure 6.23: ASM chart of vending machine (Part 1)

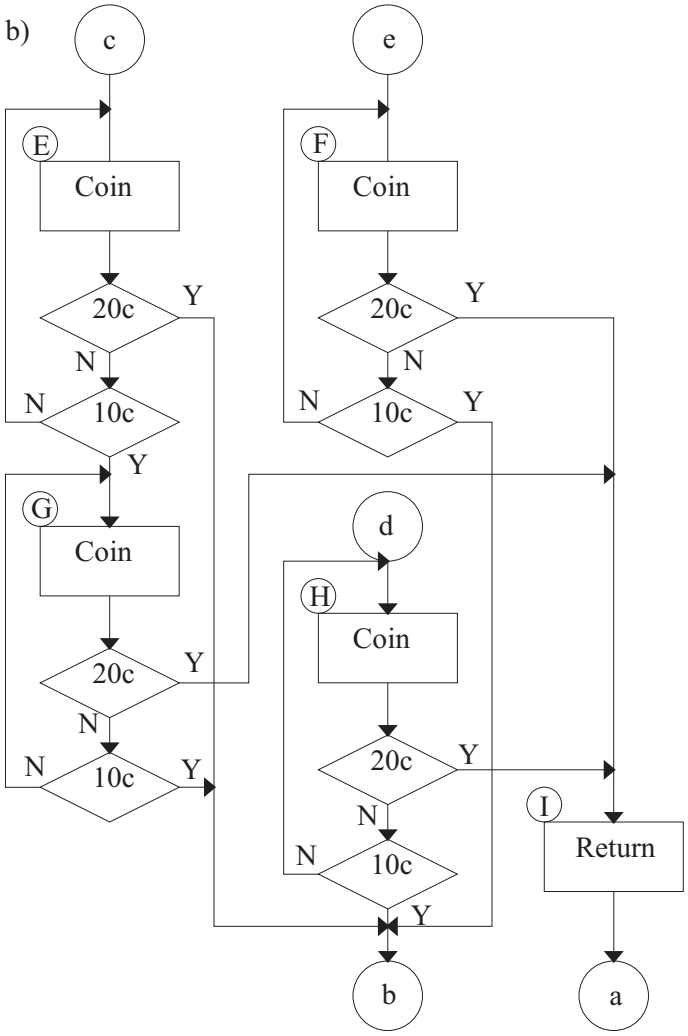



Figure 6.24: ASM chart of vending machine (Part 2)

| State | 20c | 10c | Other | Ready | Dispense | Return | Coin |
|---|---|---|---|---|---|---|---|
| A | D | C | A | 1 | 0 | 0 | 0 |
| B | A | A | A | 0 | 1 | 0 | 0 |
| C | H | E | C | 0 | 0 | 0 | 1 |
| D | B | F | D | 0 | 0 | 0 | 1 |
| E | B | G | E | 0 | 0 | 0 | 1 |
| F | I | B | F | 0 | 0 | 0 | 1 |
| G | I | B | G | 0 | 0 | 0 | 1 |
| H | I | B | H | 0 | 0 | 0 | 1 |
| I | A | A | A | 0 | 0 | 1 | 0 |
| | Next State | | | Outputs | | | |

Figure 6.25: State and output table for vending machine

States $C$ to $H$ inclusive have the same outputs. States $F$, $G$ and $H$ have the same next states, other than their default next states, which are the states themselves. In other words, states $F$, $G$ and $H$ are equivalent if states $F$, $G$ and $H$ are equivalent – which is a tautology! Therefore we can merge these three states. In other words, we will delete states $G$ and $H$, say, and replace all instances of those two states with state $F$ (Figure 6.26). Now states $D$ and $E$ are equivalent, so $E$ can be deleted and replaced by $D$ (Figure 6.27). The system has therefore been simplified from having nine states to having six. It should be remembered that the system may be implemented with nine states or with six, but it is not possible for an external observer to know which version has been built simply by observing the outputs. The two versions are therefore functionally identical.

| State | 20c | 10c | Other | Ready | Dispense | Return | Coin |
|---|---|---|---|---|---|---|---|
| A | D | C | A | 1 | 0 | 0 | 0 |
| B | A | A | A | 0 | 1 | 0 | 0 |
| C | ~~H~~ F | E | C | 0 | 0 | 0 | 1 |
| D | B | F | D | 0 | 0 | 0 | 1 |
| E | B | ~~G~~ F | E | 0 | 0 | 0 | 1 |
| F | I | B | F | 0 | 0 | 0 | 1 |
| ~~G~~ | ~~I~~ | ~~B~~ | ~~G~~ | ~~0~~ | ~~0~~ | ~~0~~ | ~~1~~ |
| ~~H~~ | ~~I~~ | ~~B~~ | ~~H~~ | ~~0~~ | ~~0~~ | ~~0~~ | ~~1~~ |
| I | A | A | A | 0 | 0 | 1 | 0 |
| | Next State | | | | Outputs | | |

Figure 6.26: State table with states $G$ and $H$ removed

| State | 20c | 10c | Other | Ready | Dispense | Return | Coin |
|---|---|---|---|---|---|---|---|
| A | D | C | A | 1 | 0 | 0 | 0 |
| B | A | A | A | 0 | 1 | 0 | 0 |
| C | ~~H~~ F | ~~E~~ D | C | 0 | 0 | 0 | 1 |
| D | B | F | D | 0 | 0 | 0 | 1 |
| ~~E~~ | ~~B~~ | ~~G~~ F | ~~E~~ | ~~0~~ | ~~0~~ | ~~0~~ | ~~1~~ |
| F | I | B | F | 0 | 0 | 0 | 1 |
| ~~G~~ | ~~I~~ | ~~B~~ | ~~G~~ | ~~0~~ | ~~0~~ | ~~0~~ | ~~1~~ |
| ~~H~~ | ~~I~~ | ~~B~~ | ~~H~~ | ~~0~~ | ~~0~~ | ~~0~~ | ~~1~~ |
| I | A | A | A | 0 | 0 | 1 | 0 |
| | Next State | | | | Outputs | | |

Figure 6.27: State table with states $E$, $G$ and $H$ removed

To conclude this example, the next state and output expressions will be written, assuming a ‘one-hot’ implementation; i.e. there is one flip-flop per

state, of which exactly one has a '1' output at any time. The next state and output expressions can be read directly from the state and output table of Figure 6.27.

$$
\begin{aligned}
A^+ &= B + I + \overline{20c}.\overline{10c}.A \\
B^+ &= D.20c + F.10c \\
C^+ &= A.10c + \overline{20c}.\overline{10c}.C \\
D^+ &= A.20c + C.10c + \overline{20c}.\overline{10c}.D \\
F^+ &= C.20c + D.10c + \overline{20c}.\overline{10c}.F \\
I^+ &= F.20c \\
Ready &= A \\
Dispense &= B \\
Return &= I \\
Coin &= C + D + F
\end{aligned}
$$

## 6.5 State machines in SystemVerilog

### 6.5.1 A first example

SystemVerilog is a very rich language, in terms of constructs. Therefore there is often more than one way to describe something. Here, we will look at two styles for modelling state machines. Both are synthesizable, both simulate. To some extent, the choice of style is a matter of taste. It should, however, be noted that some synthesis tools produce better results with one style rather than the other.

The state of the system must be held in an internal register. In SystemVerilog, the state can be represented by an *enumerated type*. The possible values of this type are the state names and the name of the variable is given after the list of values, e.g.

```
enum {s0, s1, ...} state;
```

In the following listing, there are two procedural blocks. Each has a label. The first procedural block ('SEQ') models the state machine itself. The procedural block waits until the clock input changes to '1', or the reset changes to 0. The asynchronous reset is tested first, and if it is asserted a default value is assigned to the state. Otherwise, a **case** statement is used to branch according to the current value of the state. Each branch of

the **case** statement is therefore equivalent to one of the states of Figure 6.10, together with its decision box. Within the first statement branch, the car input is tested to set the state. If the input is false, the state remains as it was (i.e. G). This is fine, as the block is declared to be **always_ff** and therefore we would expect the state to be mapped onto one or more edge-triggered flip-flops. The other state is structured in a similar way.

In the second procedural block, ('OP') the outputs are set. This is an **always_comb** block. Note that start_timer and the other outputs are given default values at the beginning. This is good practice, as it ensures that latches will not be accidentally created. Again a **case** statement is used. The structure mirrors the ASM chart. Unconditional outputs are assigned in each state; conditional assignments follow an **if** statement.

```
module traffic_1 (output logic start_timer, major_green,
                  minor_green,
                  input logic clock, n_reset, timed,
                  car);
  enum {G, R} state;

always_ff @(posedge clock, negedge n_reset)
  begin: SEQ
  if (~n_reset)
    state <= G;
  else
    case (state)
      G: if (car)
           state <= R;
      R: if (timed)
           state <= G;
    endcase
  end


always_comb
  begin: OP
  start_timer = '0;
  minor_green = '0;
  major_green = '0;
  case (state)
    G: begin
       major_green = '1;
```

```
        if (car)
          start_timer = '1;
        end
    R: minor_green = '1;
  endcase
  end

endmodule
```

Another common modelling style for state machines also uses two processes. One process is used to model the state registers, while the second process models the next state and output logic. The two processes therefore correspond to the two boxes in Figure 6.1. From Figure 6.1, it can be seen that the communication between the two processes is achieved using the present and next values of the state registers. Therefore if two SystemVerilog blocks are used, communication between them must be performed using present and next state variables.

The combinatorial block (labelled 'COM') combines the **case** statement parts of the previous version. The **case** statement now selects on present_state and next_state is updated. Note also that next_state is updated (to its existing value) even when a change of state does not occur. Failure to do this would result in latches being created.

```
module traffic_2 (output logic start_timer, major_green,
                  minor_green,
                  input logic clock, n_reset, timed,
                  car);
  enum {G, R} present_state, next_state;

always_ff @(posedge clock, negedge n_reset)
  begin: SEQ
  if (~n_reset)
    present_state <= G;
  else
    present_state <= next_state;
  end

always_comb
  begin: COM
  start_timer = '0;
  minor_green = '0;
  major_green = '0;
```

```
next_state = present_state;
case (present_state)
  G: begin
     major_green = '1;
     if (car)
       begin
       start_timer = '1;
       next_state = R;
       end
     end
  R: begin
     minor_green = 1'b1;
     if (timed)
       next_state = G;
     end
endcase
end
```

**endmodule**

It is also possible to derive a three process model: state register; next state evaluation and output assignment. There is no obvious advantage to using this model.

Some general comments apply to all styles of state machine. The inputs and outputs are of type logic.Nonblocking assignments are always used in the **always_ff** block, while blocking assignments are always used in the **always_comb** block. Never mix the two types of assignment in one block. Also note that a varaible is only ever written to by one block. To avoid accidental latches, all the outputs in an **always_comb** block are initialized at the start of the block.

### 6.5.2 A sequential parity detector

Consider the following system. Data arrives at a single input, with one new bit per clock cycle. The data is grouped into packets of 4 bits, where the fourth bit is a parity bit. (This problem could easily be scaled to have more realistically sized packets.) The system uses even parity. In other words, if there is an odd number of 1s in the first three bits, the fourth bit is a 1. If an incorrect parity bit is detected, an error signal is asserted during the fourth clock cycle.

The parity detector can be implemented as a state machine. We will

leave the design as an exercise and simply show a SystemVerilog implementation. In this example, an asynchronous reset is included to set the initial state to s0. Notice that the error signal is only set under limited conditions, making the combinational logicblock very simple.

```
module parity (output logic error,
               input logic clock, n_reset, a);

  enum {s0, s1, s2, s3, s4, s5, s6} state;

always_ff @(posedge clock, negedge n_reset)
  begin: SEQ
  if (~n_reset)
    state <= s0;
  else
    case (state)
      s0: if (~a)
            state <= s1;
          else
            state <= s2;
      s1: if (~a)
            state <= s3;
          else
            state <= s4;
      s2: if (~a)
            state <= s4;
          else
            state <= s3;
      s3: if (~a)
            state <= s5;
          else
            state <= s6;
      s4: if (~a)
            state <= s6;
          else
            state <= s5;
      s5: state <= s0;
      s6: state <= s0;
    endcase
  end
```

```
always_comb
  begin: COM
  if ((state == s5 && a) || (state == s6 && ~a))
    error = '1;
  else
    error = '0;
  end

endmodule
```

### 6.5.3 Vending machine

The following piece of SystemVerilog is a model of the (minimized) vending machine of Section 6.4.3. Two blocks are used. Note that here an asynchronous reset has been provided to initialize the system when it is first turned on.

```
module vending(output logic ready,dispense,ret,coin
               input logic clock,n_reset,twenty,ten);

  enum {A, B, C, D, F, I} present_state, next_state;

always @(posedge clock, negedge n_reset)
  begin: SEQ
  if (~n_reset)
    present_state <= A;
  else
    present_state <= next_state;
  end

always_comb
  begin: COM
  ready = '0;
  dispense = '0;
  ret = '0;
  coin = '0;
  case (present_state)
    A : begin
        ready = '1;
        if (twenty)
          next_state = D;
```

```
        else if (ten)
          next_state = C;
        else
          next_state = A;
        end
    B : begin
        dispense = '1;
        next_state = A;
        end
    C : begin
        coin = '1;
        if (twenty)
          next_state = F;
        else if (ten)
          next_state = D;
        else
          next_state = C;
        end
    D : begin
        coin = '1;
        if (twenty)
          next_state = B;
        else if (ten)
          next_state = F;
        else
          next_state = D;
        end
    F : begin
        coin = '1;
        if (twenty)
          next_state = I;
        else if (ten)
          next_state = B;
        else
          next_state = F;
        end
    I : begin
        ret = '1;
        next_state = A;
        end
endcase
```

```
  end
endmodule
```

### 6.5.4 Storing data

One (of the many) problems with the traffic light controller of Section 6.5.1 is that the minor road lights will switch to green as soon as a car is detected. This will happen even if the lights have just changed. It would be preferable if the timer were used to keep the major road lights at green for a period of time. If we did this simply by asserting the start_timer signal in both states and waiting for the timed signal to appear, as follows, an arriving car could easily be missed.

```
always_comb
  begin: COM
  start_timer = '0;
  minor_green = '0;
  major_green = '0;
  next_state = present_state;
  case (present_state)
    G: begin
       major_green = '1;
       if (car && timed)
         begin
         start_timer = '1;
         next_state = R;
         end
       end
    R: begin
       minor_green = 1'b1;
       if (timed)
         start_timer = '1;
         next_state = G;
       end
  endcase
  end
```

Therefore, the fact that a car has arrived needs to be remembered in some way. This could be done by adding further states to the state machine. Alternatively, the car arrival could be stored. It is not possible to say that one approach is better than the other. We will look at the idea of

using a state machine to control other hardware in chapter 6. Meanwhile, let us consider how a simple piece of data can be stored.

In a purely simulation model, it is possible to store the state of a variable or signal in a combinational process. This is done by assigning a value in one branch of the process. As we will see in chapter 9, when synthesized this would inevitably lead to asynchronous latches and hence timing problems. Instead, any data that is to be stored must be explicitly saved in a register, modelled as a clocked process. Storing data in this way is exactly the same as storing a state. Therefore, separate signals are needed for the present value of the car register and for the next value. We will use more meaningful names for these signals:

```
logic car_arrived, car_waiting;
```

The car_waiting signal is updated at the same time as the present_state signal.

```
always_ff @(posedge clock, negedge n_reset)
  if (~n_reset)
    begin
    present_state <= G;
    car_waiting <= '0;
    end
  else
    begin
    present_state <= next_state;
    car_waiting <= car_arrived;
    end
```

The car_arrived signal is set or reset in the following process:

```
always_comb
  begin: car_update
  if (present_state == G && car_waiting && timed)
    car_arrived = '0;
  else if (car)
    car_arrived = '1;
  else
    car_arrived = car_waiting;
  end
```

Finally, both references to car in block com at the start of this section need to be replaced by references to car_waiting. Notice that each signal is assigned in only one block. It often helps to sketch a diagram of the

system with each process represented by a box and showing all the inputs and outputs of each block. If a signal appears to be an output from two boxes, or if a signal is not an input to a block, something is not right!

## Summary

State machines can be formally described using ASM charts.

The design of a synchronous state machine from an ASM chart has a number of distinct steps: state minimization; state assignment; derivation of next state; and output logic.

A SystemVerilog model of a state machine can be written that is equivalent to an ASM chart.

This SystemVerilog model may be automatically synthesized to hardware using an RTL synthesis tool.

## Further Reading

State machine design is a core topic in digital design and therefore covered in many textbooks. Not all books use ASM notation; many use the style of Figure 6.9. The problem of state minimization is covered in detail in books such as Hill and Peterson.

## Exercises

6.1 Explain the difference between a Mealy machine and a Moore machine.

6.2 Describe the symbols used in an ASM diagram.

6.3 The following code shows part of a VHDL description of a synchronous state machine. Complete the description, by writing down the synchronization process. How would an asynchronous reset be included?

6.4 Draw the ASM chart that describes the state machine shown in Exercise 6.3.

6.5 Draw an ASM chart to describe a state machine that detects a sequence of three logic 1's occuring at the input and that asserts a logic 1 at the output during the last state of the sequence. E.g. the

sequence 001011101111 would produce an output 000000100011. Write a SystemVerilog description of the state machine.

6.6 Write a testbench to stimulate the state machine of Exercise 6.3 and verify the SystemVerilog model by simulation.

6.7 Produce next state and output logic for the state machine of Exercise 6.3 and write a SystemVerilog description of the hardware using simple gates and positive edge-triggered D flip-flops. Verify this hardware by simulation.

6.8 A state machine has two inputs, A, B, and one output, Z. If the sequence of input pairs: A=1 B=1, A=1 B=0, A=0 B =0 is detected, Z becomes 1 during the final cycle of the sequence, otherwise the output remains at 0. Write a SystemVerilog model of a state machine to implement this system.

6.9 Rewrite the model of Exercise 6.8 to use three procedural blocks: one for the registers; one for the next state logic and one for the output logic.

6.10 Rewrite the model of Exercise 6.8 to use only one process.

6.10 Design, using an ASM chart, a traffic signal controller for a crossroads. The signals change only when a car is detected in the direction with a red signal. The signals change in the (British) sequence: Red, Red & Amber, Green, Amber, Red. Note that while the signals in one direction are Green, Amber or Red & Amber, the signals in the other direction are Red (i.e. you need more than 4 states). Design an implementation that uses a minimal number of D flip-flops.

6.11 A counter is required to count people entering and leaving a room. The room has a separate entrance and exit. Sensors detect people entering and leaving. Up to seven people are allowed in the room at one time. Draw an ASM chart of a synchronous counter that counts the people in the room and that indicates when the room is empty and full. One person may enter and one person may leave during each clock cycle. The empty and full indicators should be asserted immediately the condition is true, i.e. before the next clock edge. Write a SystemVerilog model of the system.

6.12 Construct a state and output table for the state machine represented by Figure 6.28. Show that the number of states can be reduced.

Derive the next state and output logic to implement the reduced state machine using a) a minimal number of D flip-flops and b) the “one hot” D flip-flop method. What are the relative advantages of each method? How has the reduction in the number of states helped in each case?

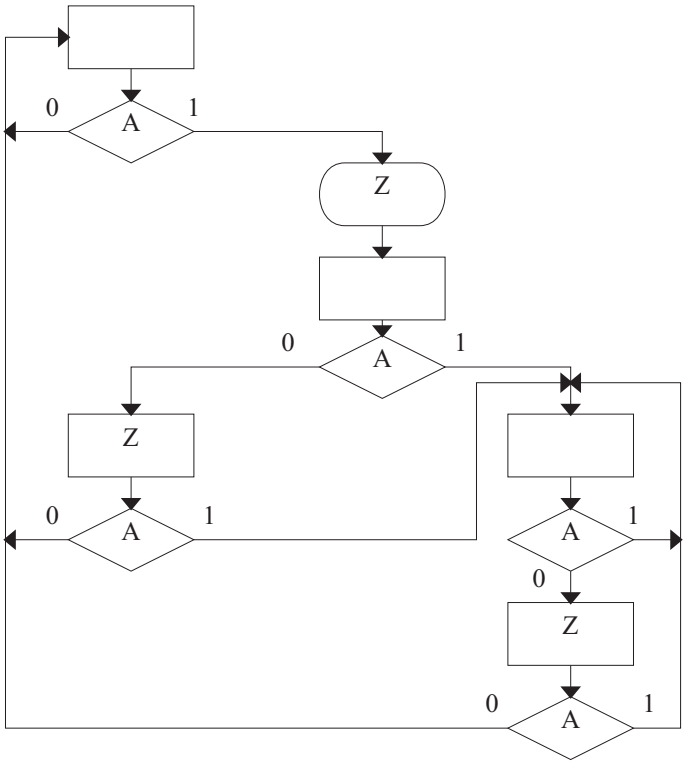



Figure 6.28: ASM Chart for Exercise 6.12

# Chapter 7

# Complex Sequential Systems

In the previous three chapters we have looked at combinational and sequential building blocks and at the design of state machines. The purpose of this chapter is to see how these various parts can be combined to build complex digital systems.

## 7.1 Linked state machines

In principle, any synchronous sequential system could be described by an ASM chart. In practice, this does not make sense. The states of a system, such as a microprocessor, include all the possible values of all the data that might be stored in the system. Therefore it is usual to partition a design in some way. In this chapter, we will show first how an ASM chart, and hence the SystemVerilog model of the state machine, can be partitioned, and second how a conceptual split may be made between the *datapath* of a system, i.e. the components that store and manipulate data, and the state machine that controls the functioning of those datapath components.

A large body of theory covers the optimal partitioning of state machines. In practice, it is usually sufficient to identify components that can easily be separated from the main design and implemented independently. For example, let us consider again the traffic signal controller.

If a car approaches the traffic signals on the minor road, a sensor is activated that causes the major road to have a red light and the minor road to have a green light for a fixed interval. Once that interval has passed, the major road has a green light again and the minor road has a red light. In Chapter 5, we simply assumed that a signal would be generated after the given interval had elapsed. Let us now assume that the clock frequency is such that the timed interval is completed in 256 clock cycles. We can

draw an ASM chart for the entire system as shown in Figure 7.1 (states 1 to 254 and the outputs are not shown, for clarity). Although this is a simple example, the functionality of the system is somewhat lost in the profusion of states that implement a simple counting function. It would be clearer to separate the traffic light controller function from the timer.

One way of doing this is shown in Figure 7.2, in which there are two ASM charts. The ASM chart on the left is the traffic light controller, in which a signal, START, is asserted as a conditional output when a car is detected. This signal acts as an input to the second state machine, allowing that state machine to move from the IDLE state into the counting sequence. When the second state machine completes the counting sequence, the signal TIMED is asserted, which acts as an input to the first state machine, allowing the latter to move from state $R$ to state $G$. The second state machine moves back into the IDLE state.

A state machine of the form of the second state machine of Figure 7.2 can be thought of as a 'hardware subroutine'. In other words, any state machine may be partitioned in this way. Unlike a software subroutine, however, a piece of hardware must exist and must be doing something, even when it is not being used. Hence, the IDLE state must be included to account for the time when the 'subroutine' is not doing a useful task.

An alternative way of implementing a subsidiary state machine is shown in Figure 7.3. This version does not correspond to the 'hardware subroutine' model, but represents a conventional counter. The use of standard components will be discussed further in the next section.

From the ASM chart of Figure 7.1 it is quite clear that the system takes 256 clock cycles to return to state $G$ after a car has been detected. The sequence of operations may be harder to follow in Figure 7.3. In state $s$255, TIMED is asserted as a conditional output. This causes the left-hand state machine to move from state $R$ to state $G$. In state $R$, ENABLE is asserted which allows the right-hand state machine to advance through its counting sequence. A timing diagram of this is shown in Figure 7.4.

At first glance this timing diagram may appear confusing. The ENABLE signal causes the TIMED signal to be asserted during the final state of the right-hand diagram. The TIMED signal causes the left-hand state machine to move from state $R$ to state $G$. According to ASM chart convention, both these signals are asserted at the beginning of a state and deasserted at the end of a state. In fact, of course, the signals are asserted some time after a clock edge and also deasserted after a clock edge. Therefore, a more realistic timing diagram is given in Figure 7.5. The changes to TIMED and ENABLE happen after the clock edges. This, of course, is necessary in order to satisfy the setup and hold times of the flip-flops in

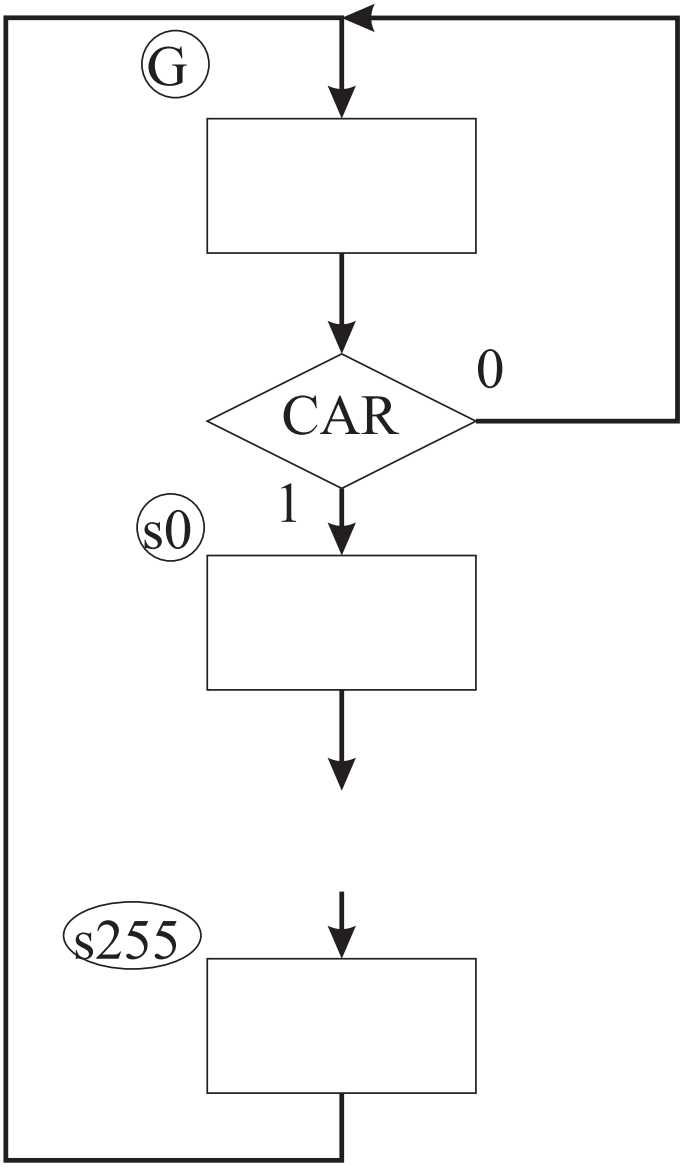



Figure 7.1: ASM chart of traffic signal controller including the timer.

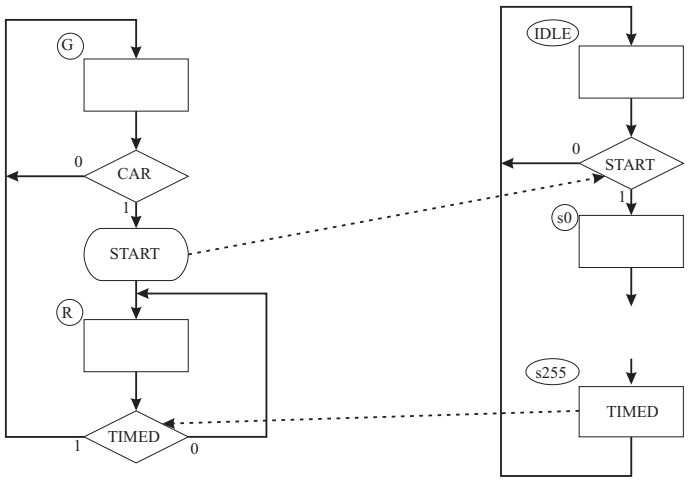



Figure 7.2: Linked ASM charts for traffic signal controller.

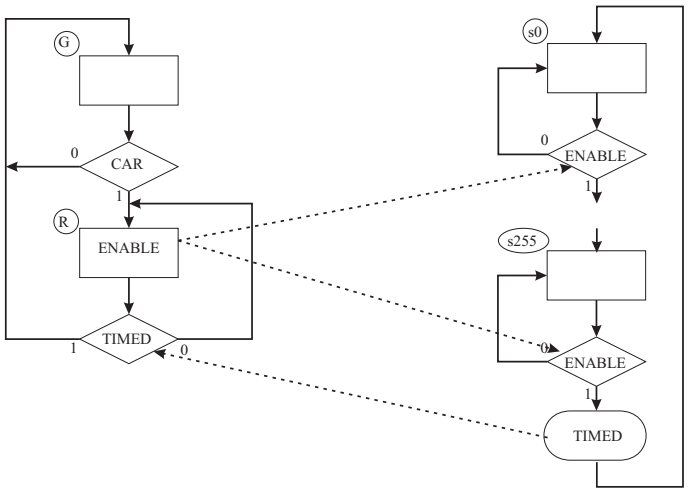



Figure 7.3: ASM chart of traffic signal controller with counter.

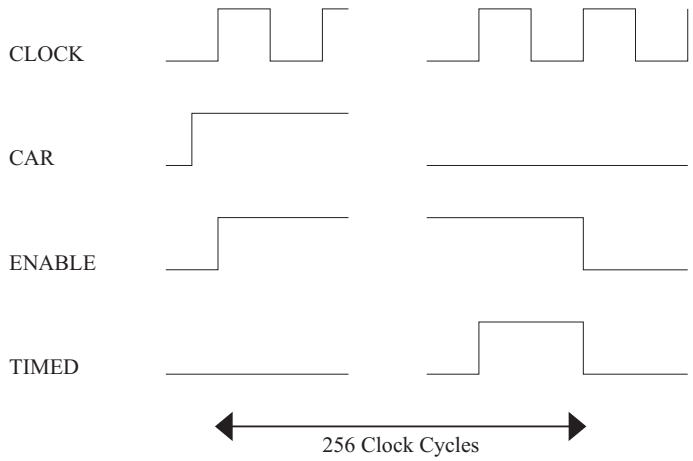



Figure 7.4: Timing diagram of linked ASM charts.

the system. The clock speed is limited by the propagation delays through the combinational logic of both state machines. In that sense, a system divided into two or more state machines behaves no differently to a system implemented as a single state machine.

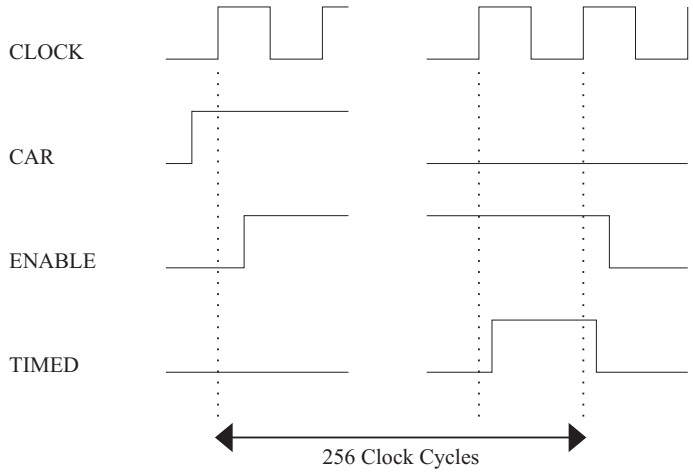



Figure 7.5: Timing diagram showing delays.

## 7.2 Datapath/controller partitioning

Although any synchronous sequential system can be designed in terms of one or more state machines, in practice this is likely to result in the

'reinvention of the wheel' on many occasions. For example, the right-hand state machine of Figure 7.3 is simply an 8-bit counter. Given this, it is obviously more effective to reuse an existing counter, either as a piece of hardware or as a SystemVerilog model. It is therefore convenient to think of a sequential system in terms of the *datapath*, i.e. those components that have been previously designed (or that can be easily adapted) and that can be reused, and the *controller*, which is a design-specific state machine. A model of a system partitioned in this way is shown in Figure 7.6.

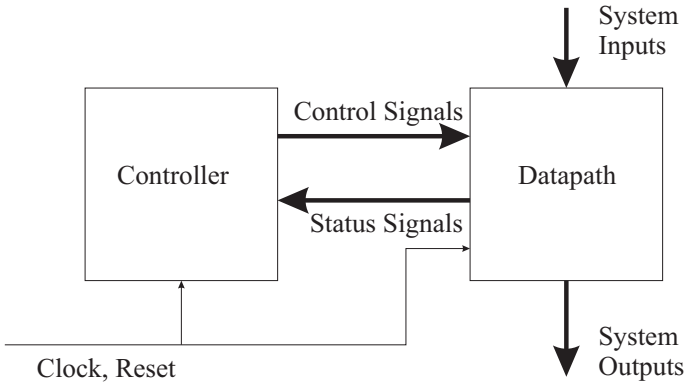



Figure 7.6: Controller/datapath partitioning.

Returning to the example of Figure 7.4, it can be seen that the left-hand state machine corresponds to a controller, while the right-hand state machine, the counter, corresponds to the datapath. The TIMED signal is a status signal, as shown in Figure 7.6, while the ENABLE signal is a control signal. We will look at a more significant example of datapath/controller partitioning in Section 7.4.

The datapath would normally contain registers. As the functionality of the system is mainly contained in the datapath, the system can be described in terms of *register transfer operations*. These register transfer operations can be described using an extension of ASM chart notation. In the simplest case a registered output can be indicated as shown in Figure 7.7(a). This notation means that $Z$ takes the value 1 *at the end* of the state indicated, and *holds that value* until it is reset. If, in this example, $Z$ is reset to 0 and it is only set to 1 in the state shown, the registered output would be implemented as a flip-flop and multiplexer, as shown in Figure 7.7(b), or simply as an enabled flip-flop as shown in Figure 7.7(c). In either implementation, the ENABLE signal is only asserted when the ASM is in the indicated state. Thus the ASM chart could equally include the ENABLE

signal, as shown in Figure 7.7(d).

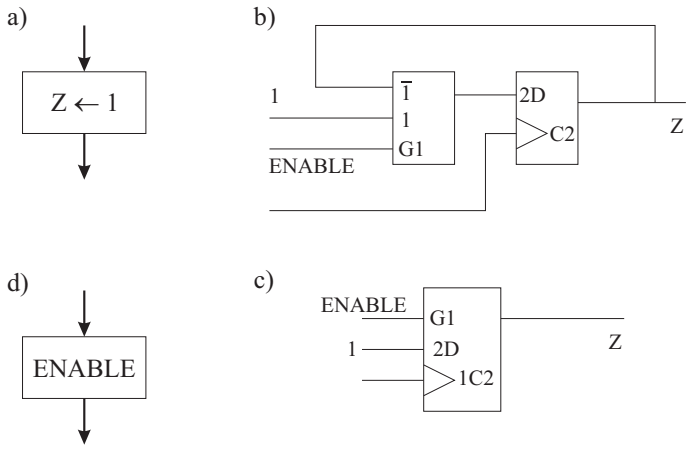



Figure 7.7: Extended ASM chart notation.

A more complex example is shown in Figure 7.8. In state 00, three registers, $B_0$, $B_1$ and $B_2$ are loaded with inputs $X_0$, $X_1$ and $X_2$, respectively. Input $A$ then determines whether a shift left, or multiply by 2, is performed ($A$=0) or a shift right, or divide by 2 ($A$=1) in the next state. If a divide by 2 is performed, the value of the least significant bit is tested, so as always to round up. From the ASM chart we can derive next state equations for the controller, either formally or by inspection:

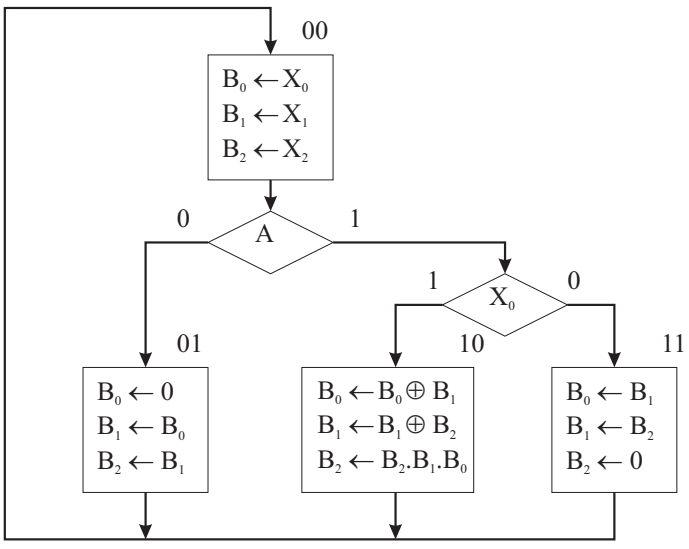



Figure 7.8: ASM chart of partitioned design.

$$S_0^+ = \bar{S}_0.\bar{S}_1.(\bar{A} + \bar{X}_0)$$

$S_1^+ = \bar{S}_0.\bar{S}_1.A$

The datapath part of the design can be implemented using registers for $B_0$, $B_1$ and $B_2$ and multiplexers, controlled by $S_0$ and $S_1$, to select the inputs to the registers, as shown in Figure 7.9. It is also possible to implement the input logic using standard gates and thus to simplify the logic slightly.

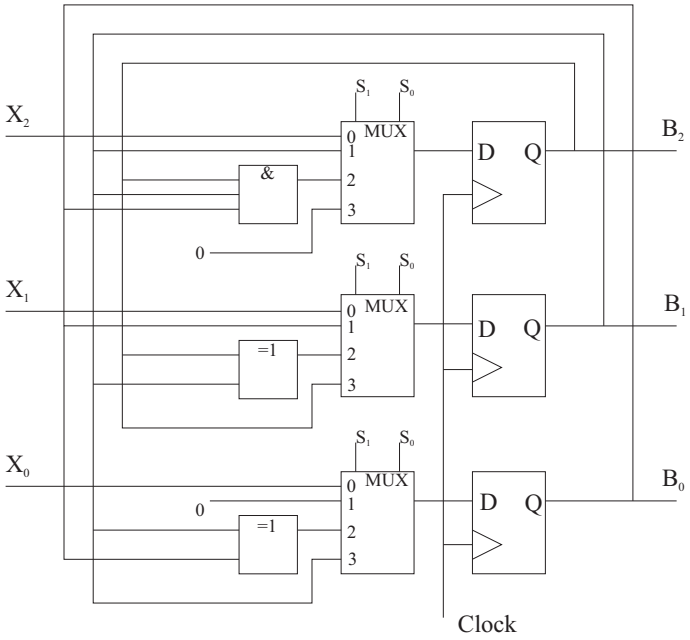



Figure 7.9: Implementation of datapath.

## 7.3 Instructions

Before looking at how a very simple microprocessor can be constructed, we will examine the interface between hardware and software. This is not a course on computer architecture – many such books exist – so the concepts presented here are deliberately simplified.

When a computer program, written in, say, C, is compiled, the complex expressions of the high-level language can be broken down into a sequence of simple assembler instructions. These assembler instructions can then be directly translated into machine code instructions.[1] These machine code instructions are sets of, say, 8, 16 or 32 bits. It is the interpretation of these bits that is of interest here.

[1] In fact, most compilers would compile directly to machine code. For the purposes of this discussion, it is easier to think in terms of assembler instructions.

Let us compile the expression:

a = b + c;

to a sequence of assembly code instructions:

LOAD b

ADD c

STORE a

The exact interpretation of these assembler instructions will be explained in the next section. If the microprocessor has eight bits, the opcode (LOAD, STORE etc.) might require three bits, while the operand (a, b, etc.) would take five bits. This allows for eight opcodes and 32 addresses (this is a *very* basic microprocessor). Hence, we might find that the instructions translate as follows.

LOAD b 00000001

ADD c 01000010

STORE a 00100011

i.e. LOAD, ADD and STORE translate to 000, 010 and 001, respectively, while a, b and c are data at addresses 00011, 00001 and 00010, respectively.

Within the microprocessor there is the datapath/controller partition described in the last section. The controller (often known as a sequencer in this context) is a state machine. In the simplest case, the bits of the opcode part of the instruction are inputs to the controller, in the same way that $A$ and $X_0$ are inputs to the controller of Figure 7.8. Alternatively, the opcode may be decoded (using a decoder implemented in ROM) to generate a larger set of inputs to the controller. The decoder pattern stored in the ROM is known as *microcode*.

The instructions shown above consist of an opcode and an address. The data to be operated upon must be subsequently obtained from the memory addresses given in the instruction. This is known as *direct* addressing. Other addressing modes are possible. Suppose we wish to compile:

a = b + 5;

This translates to:

LOAD b

ADD 5

STORE a.

How do we know that the 5 in the ADD instruction means the value ‘5’ and not the data stored at address 5? In assembler language, we would normally use a special notation, e.g. ‘ADD #5’, where the ‘#’ indicates to the assembler that the following value is to be interpreted as a value and

not as an address. This form of addressing is known as *immediate* mode addressing.

When the microprocessor executes an immediate mode instruction different parts of the datapath are used compared with those activated by a direct mode instruction. Hence the controller goes through a different sequence of states, and thus the opcodes for an immediate mode ADD and a direct mode ADD must be different. In other words, from the point of view of the microprocessor, instructions with different addressing modes are treated as totally distinct instructions and have different opcodes.

## 7.4 A simple microprocessor

Using the idea of partitioning a design into a controller and datapath, we will now show how a very basic microprocessor can be designed. We want to be able to execute simple direct mode instructions such as those described in the previous section. Let us first consider the components of the datapath that we need. In order to simplify the routing of data around the microprocessor, we will assume the existence of a single bus. More advanced designs would have two or three buses, but one bus is sufficient for our needs. For simplicity we shall assume that the bus and all the datapath components are eight bits wide, although we shall make the SystemVerilog model, in the next section, parameterizable. Because the single bus may be driven by a number of different components, each of those components will use three-state buffers to ensure that only one component is attempting to put valid data on the bus at a time. We will keep the design fully synchronous, with a single clock driving all sequential blocks. We will also include a single asynchronous reset to initialize all sequential blocks. A block diagram of the microprocessor is shown in Figure 7.10.

The program to be executed by the microprocessor will be held in memory together with any data. Memory, such as SRAM is commonly asynchronous, therefore synchronous registers will be included as buffers between the memory and the bus for both the address and data signals. These registers are the Memory Address Register (MAR) and Memory Data Register (MDR).

The Arithmetic and Logic Unit (ALU) performs the arithmetic operations (e.g. ADD). The ALU is a combinational block. The result of an arithmetic operation is held in a register, called the Accumulator (ACC). The inputs to the ALU are the bus and the ACC. The ALU may also have further outputs, or flags, to indicate that the result in the ACC has a particular

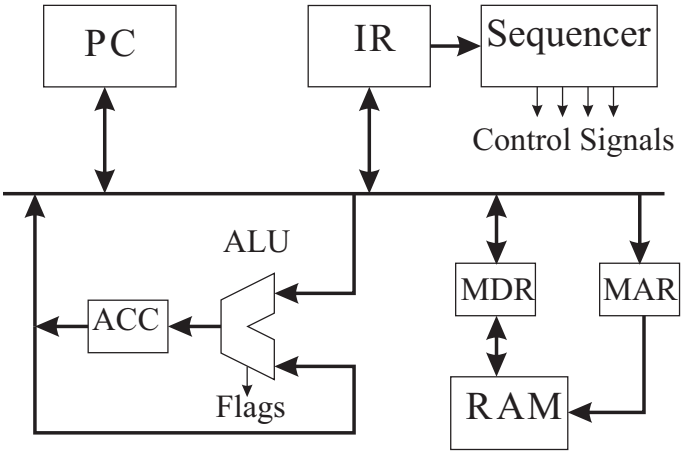



Figure 7.10: Datapath of CPU.

characteristic, such as being negative. These flags act as inputs to the sequencer.

The various instructions of a program are held sequentially in memory. Therefore the address of the next instruction to be executed needs to be stored. This is done using the Program Counter (PC), which also includes the necessary logic to automatically increment the address held in the PC. If a branch is executed, the program executes out of sequence, so it must also be possible to load a new address into the PC.

Finally, an instruction taken from the memory needs to be stored and acted upon. The Instruction Register (IR) holds the current instruction. The bits corresponding to the opcode are inputs to the sequencer, which is the state machine controlling the overall functioning of the microprocessor.

The sequencer generates a number of control signals. These determine which components can write to the bus, which registers are loaded from the bus and which ALU operations are performed. The control signals for this microprocessor are listed in Table 7.1.

Figure 7.11 shows the ASM chart of the microprocessor sequencer. Six clock cycles are required to complete each instruction. The execution cycle can be divided into two parts: the *fetch* phase and the *execute* phase. In the first state of the fetch phase, $s0$, the contents of the PC are loaded, via the bus, into MAR. At the same time the address in the PC is incremented by 1. In state $s1$, the CS and R_NW signals are both asserted to read into MDR the contents of the memory at the address given by MAR. In state $s2$, the contents of MDR are transferred to IR via the bus.

In the execute phase, the instruction, now held in IR, is interpreted and executed. In state $s3$, the address part of the instruction, the operand, is

Table 7.1: Control signals of microprocessor.

| | |
|---|---|
| ACC_bus | Drive bus with contents of ACC (enable three-state output) |
| load_ACC | Load ACC from bus |
| PC_bus | Drive bus with contents of PC |
| load_IR | Load IR from bus |
| load_MAR | Load MAR from bus |
| MDR_bus | Drive bus with contents of MDR |
| load_MDR | Load MDR from bus |
| ALU_ACC | Load ACC with result from ALU |
| INC_PC | Increment PC and save the result in PC |
| Addr_bus | Drive bus with operand part of instruction held in IR |
| CS | Chip Select. Use contents of MAR to set up memory address |
| R_NW | Read, Not Write. When false, contents of MDR are stored in memory |
| ALU_add | Perform an add operation in the ALU |
| ALU_sub | Perform a subtract operation in the ALU |

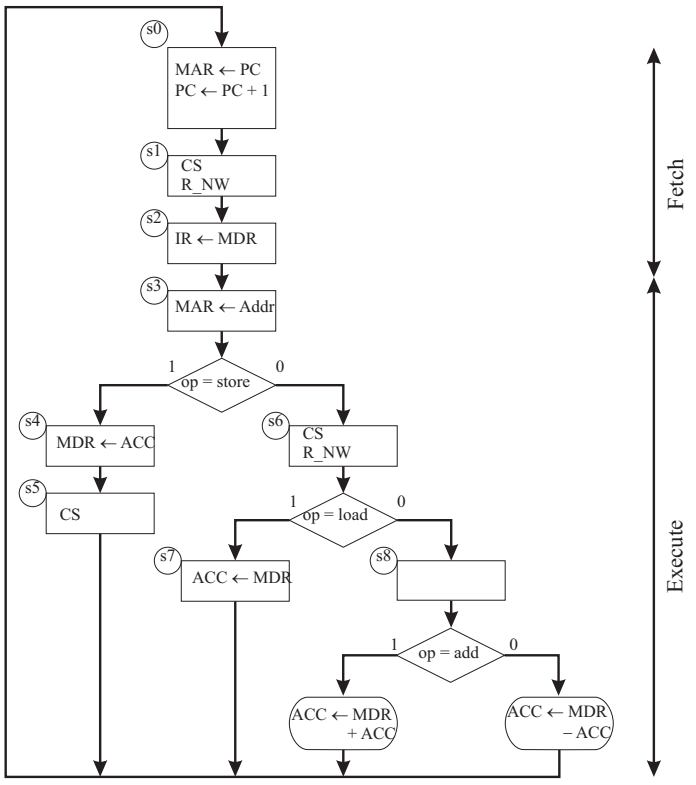



Figure 7.11: ASM chart of microprocessor.

copied back to MAR, in anticipation of using it to load or store further data. If the opcode held in IR is STORE, control passes through $s4$ and $s5$, in which the contents of ACC are transferred to MDR, then to be written into memory (at the address previously stored in MAR) when CS is asserted. If the opcode is not STORE, CS and R_NW are asserted in state $s6$, to read data from memory into MDR. If the opcode is LOAD, the contents of MDR are transferred to ACC in state $s7$, otherwise an arithmetic operation is performed by the ALU using the data in ACC and in MDR in state $s8$. The results of this operation are stored in ACC.

The ASM chart of Figure 7.11 shows register transfer operations. In Figure 7.12, the ASM chart shows instead the control signals that are asserted in each state. Either form is valid, although that of Figure 7.11 is more abstract.

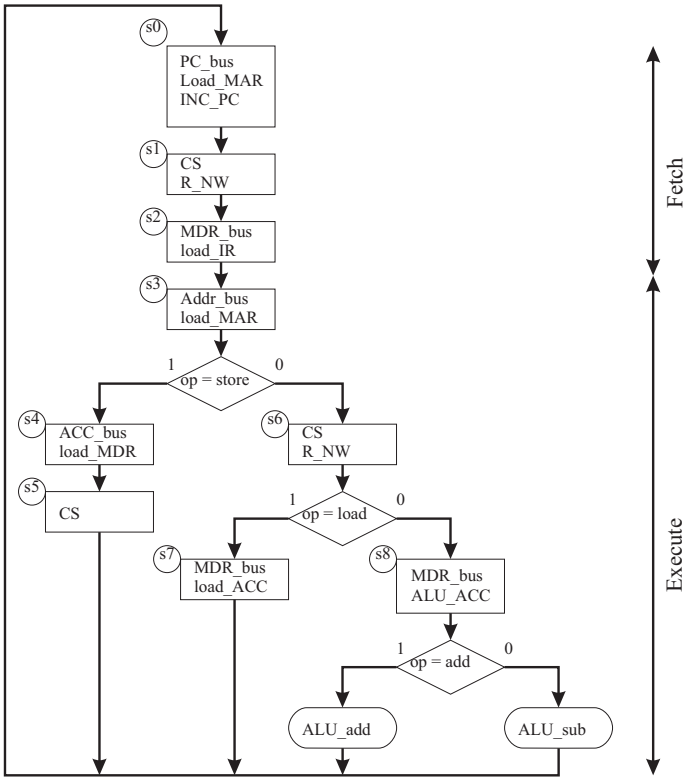



Figure 7.12: Alternative form of microprocessor ASM chart.

This processor does not include branching. Hence, it is of little use for running programs. Let us extend the microprocessor to include a branch if the result of an arithmetic operation (stored in ACC) is not zero (BNE). The ALU has a *zero flag* which is true if the result it calculates is zero and which

is an input to the sequencer. Here, we shall implement this branch instruction in a somewhat unusual manner. All the instructions in this example are direct mode instructions. To implement immediate mode instructions would require significant alteration of the ASM chart. Therefore we will implement a 'direct mode branch'. The operand of a BNE instruction is not the address to which the microprocessor will branch (if the zero flag is true), but the address at which this destination address is stored. Figure 7.13 shows how the lower right corner of the ASM chart can be modified to include this branch. An additional control signal has to be included: load_PC, to load the PC from the bus.

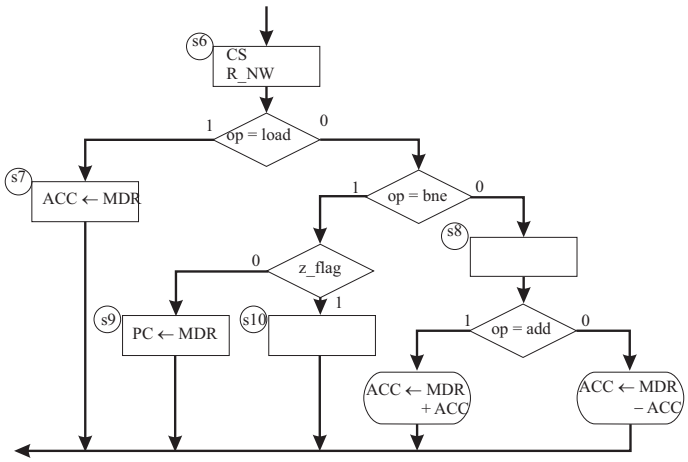



Figure 7.13: Modification of ASM chart to include branching.

## 7.5 SystemVerilog model of a simple microprocessor

The following SystemVerilog modules model the microprocessor described in the previous section. The entire model, including a basic testbench runs to around 320 lines of code. The model is synthesizable and so could be implemented on an FPGA.

The first file, cpu_defs.v, is a set of definitions. The definitions are public and may be used in any unit that uses an 'include directive. The opcodes are defined by bit patterns. The size of the bus and the number of bits in the opcode are defined by parameters. The use of this file means that the size of the CPU and the actual opcodes can be changed without altering

any other part of the model. This is important to maintain the modularity of the design.

**parameter** WORD_W = 8,
OP_W = 3;
**parameter** LOAD = 3'b000,
STORE = 3'b001,
ADD = 3'b010,
SUB = 3'b011,
BNE = 3'b100;

The controller or sequencer is described above by an ASM chart. Therefore the SystemVerilog description also take the form of a state machine. The inputs to the state machine are the clock, reset, an opcode and the zero flag from the accumulator. The outputs are the control signals of Table 7.1. Notice that the two-block model is used. Notice, too, that all the output signals are given a default value at the start of the next state and output logic block. At the end of the case statement in the combinational block, a default statement assigns X values to the state variable. This will be treated as don't care values in synthesis, and would highlight any error during simulation.

**module** sequencer (clock, reset, op, z_flag,
ACC_bus, load_ACC, PC_bus, load_PC,
load_IR, load_MAR, MDR_bus, load_MDR,
ALU_ACC, ALU_add, ALU_sub, INC_PC,
Addr_bus, CS, R_NW);
'include "cpu_defs.v"
**input** clock, reset, z_flag;
**input** [OP_W-1:0] op;
**output** ACC_bus, load_ACC, PC_bus, load_PC,
load_IR, load_MAR, MDR_bus, load_MDR,
ALU_ACC, ALU_add, ALU_sub, INC_PC,
Addr_bus, CS, R_NW;
**reg** ACC_bus, load_ACC, PC_bus, load_PC,
load_IR, load_MAR, MDR_bus, load_MDR,
ALU_ACC, ALU_add, ALU_sub, INC_PC,
Addr_bus, CS, R_NW;
**enum** {s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, s10}
Present_State, Next_State;
**always_ff** @(**posedge** clock **or posedge** reset)
**begin**: seq
**if** (reset)
Present_State ¡= s0;

```
else
Present_State ¡= Next_State;
end
always_comb
begin: com
// reset all the control signals to default
ACC_bus = 1'b0;
load_ACC = 1'b0;
PC_bus = 1'b0;
load_PC = 1'b0;
load_IR = 1'b0;
load_MAR = 1'b0;
MDR_bus = 1'b0;
load_MDR = 1'b0;
ALU_ACC = 1'b0;
ALU_add = 1'b0;
ALU_sub = 1'b0;
INC_PC = 1'b0;
Addr_bus = 1'b0;
CS = 1'b0;
R_NW = 1'b0;
case (Present_State)
s0: begin
PC_bus = 1'b1;
load_MAR = 1'b1;
INC_PC = 1'b1;
load_PC = 1'b1;
Next_State = s1;
end
s1: begin
CS = 1'b1;
R_NW = 1'b1;
Next_State = s2;
end
s2: begin
MDR_bus = 1'b1;
load_IR = 1'b1;
Next_State = s3;
end
s3: begin
Addr_bus = 1'b1;
```

```
load_MAR = 1'b1;
if (op == STORE)
Next_State = s4;
else
Next_State = s6;
end
s4: begin
ACC_bus = 1'b1;
load_MDR = 1'b1;
Next_State = s5;
end
s5: begin
CS = 1'b1;
Next_State = s0;
end
s6: begin
CS = 1'b1;
R_NW = 1'b1;
if (op == LOAD)
Next_State = s7;
else if (op == BNE)
begin
if (z_flag == 1'b0)
Next_State = s9;
else
Next_State = s10;
end
else
Next_State = s8;
end
s7: begin
MDR_bus = 1'b1;
load_ACC = 1'b1;
Next_State = s0;
end
s8: begin
MDR_bus = 1'b1;
ALU_ACC = 1'b1;
load_ACC = 1'b1;
if (op == ADD)
ALU_add = 1'b1;
```

```
else if (op == SUB)
ALU_sub = 1'b1;
Next_State = s0;
end
s9: begin
MDR_bus = 1'b1;
load_PC = 1'b1;
Next_State = s0;
end
s10: Next_State = s0;
endcase
end
endmodule
```

The datapath side of the design, as shown in Figure 7.10, has been described in four parts. Each of these parts is similar to the type of sequential building block described in Chapter 4. The system bus is described as a bidirectional port in each of the following four modules. An assignment sets a high impedance state onto the bus unless the appropriate output enable signal is set. Notice the use of the replication operator. The first module models the ALU and Accumulator.

```
module ALU (clock, reset, ACC_bus, load_ACC, ALU_ACC, ALU_add,
ALU_sub, sysbus, z_flag);
'include "cpu_defs.v"
input clock, reset, ACC_bus, load_ACC, ALU_ACC, ALU_add,
ALU_sub;
inout [WORD_W-1:0] sysbus;
output z_flag;
reg [WORD_W-1:0] acc;
assign sysbus = ACC_bus ? acc : {WORD_W{1'bZ}};
assign z_flag = acc == 0 ? 1'b1 :1'b0;
always_ff @(posedge clock or posedge reset)
begin
if (reset)
acc ¡= 0;
else
if (load_ACC)
if (ALU_ACC)
begin
if (ALU_add)
acc ¡= acc + sysbus;
else if (ALU_sub)
```

```
acc ¡= acc - sysbus;
end
else
acc ¡= sysbus;
end
endmodule
```

The program counter is similar in structure to the ALU and Accumulator. Notice the use of replication and concatenation to set the most significant bits on the system bus.

```
module PC (clock, reset, PC_bus, load_PC, INC_PC, sysbus);
‘include ”cpu_defs.v”
input clock, reset, PC_bus, load_PC, INC_PC;
inout [WORD_W-1:0] sysbus;
reg [WORD_W-OP_W-1:0] count;
assign sysbus = PC_bus ? {{OP_W{1’b0}},count} : {WORD_W{1’bZ}};

always_ff @(posedge clock or posedge reset)
begin
if (reset)
count ¡= 0;
else
if (load_PC)
if (INC_PC)
count ¡= count + 1;
else
count ¡= sysbus;
end
endmodule
```

The instruction register is basically an enabled register.

```
module IR (clock, reset, Addr_bus, load_IR, op, sysbus);
‘include ”cpu_defs.v”
input clock, reset, Addr_bus, load_IR;
inout [WORD_W-1:0] sysbus;
output [OP_W-1:0] op;
reg [WORD_W-1:0] instr_reg;
assign sysbus = Addr_bus ?
{{OP_W{1’b0}},instr_reg[WORD_W-OP_W-1:0]} :
{WORD_W{1’bZ}};
assign op = instr_reg[WORD_W-1:WORD_W-OP_W];
always_ff @(posedge clock or posedge reset)
begin
```

```
if (reset)
instr_reg ¡= 0;
else
if (load_IR)
instr_reg ¡= sysbus;
end
endmodule
```

The memory module is, again, very similar to the static RAM of the last chapter. A short program has been loaded in the RAM. In order to make the model parameterisable, the “program” has been written as an opcode followed by some 0s followed by an address. Because we know the size of the address (3 bits in each case), we can set the number of zeros by replication.

```
module RAM (clock, reset, MDR_bus, load_MDR, load_MAR, CS,
R_NW, sysbus);
‘include ”cpu_defs.v”
input clock, reset, MDR_bus, load_MDR, load_MAR, CS, R_NW;
inout [WORD_W-1:0] sysbus;
reg [WORD_W-1:0] mdr;
reg [WORD_W-OP_W-1:0] mar;
reg [WORD_W-1:0] mem [0:(1¡¡(WORD_W-OP_W))-1];
assign sysbus = MDR_bus ? mdr : {WORD_W{1’bZ}};
always_ff @(posedge clock or posedge reset)
begin
if (reset)
begin
mdr ¡= 0;
mar ¡= 0;
mem[0] ¡= {LOAD, {(WORD_W-OP_W-3){1’b0}},3’d4};
mem[1] ¡= {ADD, {(WORD_W-OP_W-3){1’b0}},3’d5};
mem[2] ¡= {STORE,{(WORD_W-OP_W-3){1’b0}},3’d6};
mem[3] ¡= {BNE, {(WORD_W-OP_W-3){1’b0}},3’d7};
mem[4] ¡= 2;
mem[5] ¡= 2;
mem[6] ¡= 0;
mem[7] ¡= 0;
end
else
if (load_MAR)
mar ¡= sysbus[WORD_W-OP_W-1:0];
else if (load_MDR)
```

```
mdr ¡= sysbus;
else if (CS)
if (R_NW)
mdr ¡= mem[mar];
else
mem[mar] ¡= mdr;
end
endmodule
```

The various parts of the microprocessor can now be pulled together by instantiating them. Here, we use the *named* style of argument passing – till now we have used the *positional* form. In the named style, arguments can appear in any order, and take the form .internal_name(external_name), where internal_name is the name used inside the module definition and external_name is the name used in the instantiating module. Named notation can be used for parameters as well as signals.

```
module CPU (clock, reset, sysbus);
‘include ”cpu_defs.v”
input clock, reset;
inout [WORD_W-1:0] sysbus;
wire ACC_bus, load_ACC, PC_bus, load_PC, load_IR, load_MAR,
MDR_bus, load_MDR, ALU_ACC, ALU_add, ALU_sub, INC_PC,
Addr_bus, CS, R_NW, z_flag;
wire [OP_W-1:0] op;
sequencer s1 (.clock(clock), .reset(reset), .op(op),
.z_flag(z_flag), .ACC_bus(ACC_bus),
.load_ACC(load_ACC), .PC_bus(PC_bus),
.load_PC(load_PC), .load_IR(load_IR),
.load_MAR(load_MAR), .MDR_bus(MDR_bus),
.load_MDR(load_MDR), .ALU_ACC(ALU_ACC),
.ALU_add(ALU_add), .ALU_sub(ALU_sub),
.INC_PC(INC_PC), .Addr_bus(Addr_bus), .CS(CS),
.R_NW(R_NW));
IR i1 (.clock(clock), .reset(reset),
.Addr_bus(Addr_bus), .load_IR(load_IR), .op(op),
.sysbus(sysbus));
PC p1 (.clock(clock), .reset(reset),
.PC_bus(PC_bus), .load_PC(load_PC),
.INC_PC(INC_PC), .sysbus(sysbus));
ALU a1 (.clock(clock), .reset(reset),
.ACC_bus(ACC_bus), .load_ACC(load_ACC),
.ALU_ACC(ALU_ACC), .ALU_add(ALU_add),
```

```
.ALU_sub(ALU_sub), .sysbus(sysbus),
.z_flag(z_flag));
RAM r1 (.clock(clock), .reset(reset),
.MDR_bus(MDR_bus), .load_MDR(load_MDR),
.load_MAR(load_MAR), .CS(CS), .R_NW(R_NW),
.sysbus(sysbus));
endmodule
```

The following piece of Verilog generates a clock and reset signal to allow the program defined in the RAM module to be executed. Obviously, this testbench would not be synthesized. Note that the timescale directive must be included here, if nowhere else.

```
‘timescale 1ns/1ns
module TestCPU;
‘include ”cpu_defs.v”
reg clock, reset;
wire [WORD_W-1:0] sysbus;
CPU c1 (.clock(clock), .reset(reset), .sysbus(sysbus));
always
begin
#10 clock = 1’b1;
#10 clock = 1’b0;
end
initial
begin
reset = 1’b0;
#1 reset = 1’b1;
#2 reset = 1’b0;
end
endmodule
```

## Summary

In this chapter we have looked at linked ASM charts and at splitting a design between a controller, which is designed using formal sequential design methods, and a datapath that consists of standard building blocks.

The example of a simple CPU has been used to illustrate this partitioning.

The SystemVerilog model can be both simulated and synthesized.

# Chapter 8

# Writing Testbenches

Writing a synthesizable model of a piece of hardware is only half (or perhaps less than half) of the design problem. It is essential to know that the model does the task for which it is intended. It would, of course, be possible to do this the hard way – by synthesizing the hardware and testing the design in the final context in whch it is to be used. This could be a very expensive and dangerous approach. Would you be happy travelling in a plane, knowing that the guidance system was still being tested?

The alternative is to verify the hardware before synthesis. In practice, this means that the hardware has to be simulated. In order to simulate a SystemVerilog model, stimuli have to be applied to the model and the responses of the model have to be analysed. For portability and to avoid having to learn a new set of language constructs, the stimuli and response analysis routines are written in SystemVerilog. It is tempting to argue that with FPGAs, it can be as fast to make changes to the hardware as it is to simulate. There is some truth to this, inasmuch as the quality of the verification cannot be truly known until the actual hardware is tested, but simulation should always be used to check any changes before synthesis is done.

We use the term *testbench* to describe a piece of SystemVerilog written to verify a synthesizable model. There are two basic features of a testbench that distinguish it from a synthesizable model. First, a testbench has no inputs or outputs; it is the entire world as far as the model is concerned. In a simulation, we can have access to every part of a model, therefore this lack of input and outputs does not restrict us in any way. Second, because a testbench is *never* synthesized, we can use the entire SystemVerilog language. This freedom to use the entire language can present its own difficulties. By sticking to an agreed subset of the SystemVerilog, it is straightforward to write portable, synthesizable hardware

models. Because of the definition of the SystemVerilog simulation cycle, it cannot be guaranteed that an arbitrary piece of SystemVerilog code will execute in exactly the same way on two simulators. Therefore it is very easy to write testbenches that behave differently, and that give different simulation results, on different simulators.

## 8.1 A First Example

Let us consider a basic two-input NAND gate. To verify that this works as expected, we can apply all four combinations of inputs. For the sake of exposition, let us also assume that the gate has a delay. Therefore we need to allow a finite amount of time to elapse between input changes, such that the output is able to stabilize.

```
‘timescale 1ns/100ps
module NAND2 #(parameter delay = 1) (output z, input x, y);
assign  #(delay) z = ∼ (x & y);
endmodule
‘timescale 1ns/100ps
module testNAND2;
wire c;
reg a, b;
NAND2 #2 n0 (.z(c), .x(a), .y(b));
initial
begin
a = 1’b0;
b = 1’b0;
#4 a = 1’b1;
#4 b = 1’b1;
#4 a = 1’b0;
end
endmodule
```

The testbench has an initial procedure containing a sequence of assignments to a and b. The initial keyword does not mean that procedure initialises signals, rather that the procedure is only executed once. Notice that the delay value is a relative delay and appears on the left side of a blocking assignment. This style is suitable for testbenches, but should not be used for modelling synthesisable hardware.

## 8.2 Clock Generation

The most important signal in any design is the clock. In the simplest case, a clock can be generated by inverting its value at a regular interval.

The default value of any signal is ‘X’. Simply inverting a signal at a regular interval will invert the ‘X’ value. Thus the following will not work, the clock would stay at ‘X’:

**assign** #10 clock = ∼clock;

Therefore the signal has to be initialised. This could be done by using an initial procedure:

**initial** clock = 1’b0;

**always** #10 clock = ∼clock;

This explicitly uses the initial procedure as an initialisation. In this case the approach will work, but in general this is a poor coding style. The clock signal is driven from two procedures. We cannot be certain in which order procedures will be executed. Some simulators will evaluate procedures in order of declaration; other simulators will evaluate all the inital procedures first. It is far better to drive each signal from exactly one procedure. An example of this is:

**initial**
**begin**
clock = 1’b0;
**forever** #10 clock = ∼clock;
**end**

This could also be done by assigning specific values to the clock.

**always**
**begin**
#10 clock = 1’b0;
#10 clock = 1’b1;
**end**

All these clock generation examples model a clock with equal high and low periods. The following example shows a clock generator in which the frequency and mark/space ratio are parameters. Notice that (a) the time precision is specified to be one tenth of the time unit and (b) the clock frequency is specified as a real number. Both of these conditions must be fulfilled for the example given to simulate correctly. If the frequency is specified as an integer, a mark period of 45% will cause a clock to be generated with a period of 9 ns, and mark and space times of 4 ns and 5 ns, respectively, because of rounding errors.

‘timescale 1ns/100ps

```
module clock_gen;
parameter ClockFreq_MHz = 100.0; // 100 MHz
parameter Mark = 45; // Mark length %
parameter ClockHigh = (Mark*10)/ClockFreq_MHz;
// Mark time in ns
parameter ClockLow = ((100 - Mark)*10)/ClockFreq_MHz;
// Space time in ns
reg clock;
initial
begin
clock = 1'b0;
forever
begin
#ClockLow clock = 1'b1;
#ClockHigh clock = 1'b0;
end
end
endmodule
```

## 8.3 Reset and other deterministic signals

After the clock, the next most important signal is probably the reset (or set). The reset signal is usually only asserted once at the beginning of a simulation, so an initial statement is used.

```
initial
begin
reset = 1'b1;
#10 reset = 1'b0;
#10 reset = 1'b1;
end
```

Other deterministic waveforms can be defined in a similar way:

```
initial
begin
control = 1'b1;
#10 control = 1'b0
#20 control = 1'b1;
#5 control = 1'b0;
#10 control = 1'b1;
end
```

## 8.4 Random timing

"Real" signals do not always occur with exact timing. There is inevitably a certain amount of uncertainty about the exact time of a transition. This deviation from exact timing is known as "jitter". In order to verify the robustness of a design, it may be desirable to model this uncertainty. SystemVerilog includes a system function – $random – that generates (32 bit) random numbers. This can be used to add a random offset to a delay. For example, the following code generates a clock signal with a mean period of 10 ns with a random offset between –4 ns and +4ns.

```
‘timescale 1ns/100ps
module randclk;
reg clk;
initial
begin
clk = 1’b0;
forever #(10+$random%5) clk = ∼clk;
end
endmodule
```

## 8.5 Synchronised signals

Suppose that a set of input changes is written in a procedure as in previous sections. It is possible that one or more of these changes coincides with a clock change. If all the signal assignments are made using blocking assignments (as shown in all the previous examples), we do not know which of the assignments would be completed first. This is a race condition that could lead to different simulated behaviour in different simulators. Race conditions are discussed elsewhere. We will note here that one way to avoid such a race is to use nonblocking assignments. All the above examples could be written using nonblocking assignments. It is also possible to combine signals into one block. It is generally better, however, to modularise code in order to allow reuse.

In exactly the same way that an RTL model can be made sensitive to the clock or to some other signal, parts of the testbench can also be made sensitive to the clock. Here we use the event control construct (@) in a context other than an always block.

```
integer count;
initial
```

```
begin
count = 0;
forever
begin
@(posedge clk);
#5 count = count + 1;
end
end
```

This example has two forms of timing control: a *delay* control (#5) and an *event* control (@(**posedge** clk)). Either or both forms can precede a statement, thus we could have written:

@(**posedge** clk) #5 count = count + 1;

It is also possible to make a statement sensitive to any edge by writing, for example, @clk. A third form of event control is the level sensitive wait statement:

**wait** (!enable) #10 count = count + 1;

If enable is at '1', the flow stops until enable becomes '0'. If enable is already '0', there is no delay[1].

It is also possible to generate named events and to control the flow. For example, one unit might have the named event trigger (defined as shown):

-¿ trigger;

In another process, there is an event control sensitive to that named event:

@trigger count = count + 1;

## 8.6 Monitoring Responses

Many simulators allow the user to interactively select signals for display. Generally it is possible to move through the hierarchy, so signals that might be invisible in the real circuit can be seen by the user in simulation. This is clearly useful for debugging. Similarly, debugging facilities that would be found in a programming language development tools, such as setting breakpoints and monitoring signals, are now built into many simulators. These facilities are unique to each simulator and are not part of the SystemVerilog language.

Some SystemVerilog simulators do not work interactively. The user must specify the signals to be displayed in advance of the simulation and

[1] If you are familiar with VHDL, be careful! The behaviour of a SystemVerilog wait statement is different to a VHDL wait until statement.

build the display functions into the testbench.

### 8.6.1 Printing output data

The simplest, simulator-independent way to monitor what is happening is to write messages to the user. For example, the following procedure is executed whenever one of the outputs from the traffic light controller of Chapter 5 changes.

**always** @(major_green **or** minor_green)
**if** (major_green)
$display("%t Major Road is Green",$time);
**else if** (minor_green)
$display("%t Minor Road is Green",$time);

The *system task*, $display writes text in a similar way to ***printf*** in C. The *system function* $time, returns the current simulation time. There are two system tasks: ***$display*** and $write for generating general textual output. The difference between them is that $display automatically includes a new line character, while $write does not. %t is an example of a format specifier. It is possible to assume default formatting. For example, it would be legal to write:

$display($time, "Major Road is Green");

Table **??** lists all the format specifiers. Note that either upper or lower case specifiers may be used (e.g. %t and %T are equivalent).

Table 8.1: Format Specifiers

| **Specifier** | **Meaning** |
|---|---|
| %h | Hexadecimal format |
| %d | Decimal format |
| %o | Octal format |
| %b | Binary format |
| %c | ASCII character format |
| %v | Net signalstrength |
| %m | Hierarchical name of current scope |
| %s | String |
| %t | Time |
| %e | Real in exponential format |
| %f | Real in decimal format |
| %g | Real in exponential or decimal format |

All specifiers appear in a string and (except for ***%m***) require a parameter following in the $display or $write call. The data is right justified unless a format specifier is included. Except for real numbers, only the value 0 may be used, which supresses leading spaces (e.g. %0o). Real numbers may be formatted as in C (e.g. %10.3f prints a number in 10 places with 3 fractional places.)

In the first string parameter, there can be a number of special characters as shown in Table 8.2.

Table 8.2: Special characters

| **Symbol** | **Meaning** |
|---|---|
| \n | New line |
| \t | Tab |
| \\ | \character |
| \" | " character |
| \xyz | Where xyz is are octal digits - the character given by that octal code |
| %% | % character |

Two other output tasks allow signals to be displayed: $monitor and $strobe. While $display and $write generate outputs at exactly the point at which they are called in the simulation cycle, $monitor outputs data continuously but $strobe only outputs data at the end of the simulation cycle. Only one $monitor process can be active at a time. Every time one of the arguments to the monitor task changes, a new set of data is displayed. On the other hand, the $strobe task will only display stable data – the state of signals at the end of the cycle, just before moving to the next simulation time.

### 8.6.2 Comparing responses

While the graphical display of simulation results can be a very useful debugging tool, with large designs and long simulation times, it can be very difficult to interpret a large amount of graphical data. Therefore, it is desirable to write testbenches that can do a certain amount of interpretation so that the information presented to the user is significantly reduced in volume.

In order to make an automatic judgement on the quality of responses, the testbench needs to know what the correct responses should be. This suggests that either responses need to be generated on the fly or need to be stored and compared dynamically.

One way to generate responses on the fly is to have two versions of the design. Thus we could compare a high-level behavioural model with an RTL model, or an RTL model with a netlist.

```
design_struct v0 (in_a, in_b, out_s);
design_rtl v1 (in_a, in_b, out_b);
always @(out_s or out_b)
if (out_s != out_b)
$display("Mismatch in behavioural and structural outputs");
```

Although simple to implement, this approach is flawed because any timing differences, however slight, will generate warning messages. In practice, it is very likely that there will be some differences between the outputs of two models at different levels of abstraction, *but* these differences will probably not be significant.

Therefore, it is preferable to compare responses only at specified *strobe* times. For example, we might wish to check responses 5 time units after a rising clock edge. This could be done as follows. Note the use of the $strobe task to ensure that the warning is generated at the end of the current simulation time.

```
always @(clock, out_s, out_b
begin
@(posedge clock);
#5 if (out_s != out_b)
$strobe("Mismatch in behavioural and structural outputs");
end
```

# Summary

Testbench writing is as important as modelling hardware. The entire SystemVerilog language can be used to write testbenches. Clock generation and reset generation can be done in separate procedural blocks. Use event and timing constructs to synchronise activities.

# Chapter 9

# SystemVerilog Simulation

In order to appreciate why certain styles of SystemVerilog code are preferred and why it is necessary to be careful when writing test benches, it is useful to have some understanding of how a SystemVerilog simulator works. It's also important to remember that RTL synthesis attempts to generate low level hardware that behaves in the same way as the original code. In other words, the interpretation of SystemVerilog structures for synthesis is based on the simulation model.

The first key point to appreciate is that the simulation moves forward in discrete time steps. The minimum time step is defined by the second parameter of the 'timescale directive. Although the minimum time step is defined, the simulator can take longer steps (that are multiples of this minimum time step). The simulator will only move to a time if something is scheduled to occur at that time.

Furthermore, at each time step, one event can cause further activity at that time. Clearly, simply adding new events to the list of event scheduled at the current time could produce a deadlock. It is convenient to think of time moving forward by an infinitesimal amount. However many of these infinitesimal steps are executed at a timestep, the time is not advanced until all events at the present time have been exhausted. The pseudo-code, below, extracted from the *Verilog* LRM (Language Reference Manual), describes the simulation cycle. Each iteration of the loop is one cycle. T is the current simulation time.

```
while (there are events) {
if (no active events) {
if (there are inactive events) {
activate all inactive events;
} else if (there are nonblocking assign update events) {
activate all nonblocking assign update events;
```

```
} else if (there are monitor events) {
activate all monitor events;
} else {
advance T to the next event time;
activate all inactive events for time T;
}
}
E = any active event;
if (E is an update event) {
update the modified object;
add evaluation events for sensitive processes to event
queue;
} else { /* shall be an evaluation event */
evaluate the process;
add update events to the event queue;
}
}
```

Only active events are processed, but a new events may be one of five types. The LRM talks of a stratified event queue, with five regions as follows:

Active events. These occur at the present time and can be processed in any order.

Inactive events. These occur at the present time and are processed after all the active events have been processed.

Nonblocking assign update events.

Monitor events.

Future events.

From the pseudo-code and this description, it can be seen that the list of active events is one of the lists, 2, 3 or 4, that has been created during some previous simulation cycle, together with any (active) events that are generated during the current cycle.

Working backwards, monitor events are created by $monitor and $strobe task. These cannot create new events, so will always be executed last at a simulation time.

Nonblocking assign update events are created by nonblocking assignments (¡=). The evaluation of the right hand side of all nonblocking assignments is *always* done before *any* nonblocking assign updates are done. This is important as it allows sequential systems to be modelled correctly.

Inactive events are those events that are due to occur at the current time but that have been explicitly delayed. In practice, this can be done with a zero delay (#0). As a general guideline, do not use zero delays!

A zero delay does not represent real hardware (nor a useful testbench construct). Therefore you are simply trying to fool the simulator. Unless you know exactly what you are doing, it will probably fool you!

Events may be processed from the active event list in any order (or to think of it another way, events can be added to the event lists in any order). This is the fundamental cause of non-determinism in SystemVerilog[1]. We can be sure of only two things:

Statements between begin and end will be executed in the order stated.

Nonblocking assignments will be performed after active events.

*Everything* else is indeterminate. Moreover, the execution of a procedural block can be interupted to allow another procedural block to be executed. The skill in writing SystemVerilog code is therefore to ensure that this indeterminism does not matter. If the code is badly written, a *race* condition is likely to result – that is a situation where the procedure writing a value and the procedure reading that value are racing each other. Depending which completes first, either the original or the updated value may be read.

## 9.1 Races

Perhaps the best way to understand races is by examples. The first example comes from the LRM.

```
assign p = q;
initial
begin
q = 1;
#1 q = 0;
$display(p);
end
```

Because the execution of the initial procedure can be interleaved with the assign statement, the value of p that is displayed could be 1 or 0. Either is “correct” and different simulators may give different results.

The second example is adapted from the last chapter. This is an example of an assignment to a variable from two different procedures.

```
initial #10 clock = 1'b0;
always #10 clock = ~clock;
```

[1]VHDL experts may be looking for *delta delays*. There is no such thing in SystemVerilog. The existence of the delta delay means that a VHDL simulation is deterministic and repeatable between simulators. The absence of delta delays in SystemVerilog means that simulations are not deterministic and not repeatable between different simulators.

The assumption (presumably) is that the initial procedure is evaluated first, causing the clock to stay at X for 10 time units and then the clock changes every 10 units. If the always procedure is evaluated first, the assignment is redundant because it is superseded by the assignment in the initial procedure. In either case, a clock waveform is generated, but the two cases are out of phase[2].

This ambiguity can be overcome by changing the assignment in the always procedure to a nonblocking assignment. By definition, the nonblocking assignment is evaluated after the blocking assignment in the initial procedure.

A third example of a race is shown below.

**always** @(**posedge** clock)
b = c;
**always** @(**posedge** clock)
a = b;

This is supposed to model two flip-flops connected in series. If the procedures are evaluated in the order written, at a rising clock edge the value of c will be copied to b. A new event will be scheduled at the current time, causing that same value to be copied to a. This is clearly not the intended behaviour. If the procedures are evaluated in the opposite order, the correct behaviour is modelled.

## 9.2 Avoiding Races

In order to achieve deterministic behaviour, there are several rules that should be followed when writing models and testbenches.

Do not assign to the same variable from two or more procedures. Not only is contention liable, but as can be seen from the second example above, multiple assignments can cause ambiguous behaviour. Part of the problem is the keyword “initial”. Procedures defined with the initial keyword are executed once; they are not initialisation blocks.

Use nonblocking assignments for modelling sequential logic. The third example, above, evaluates correctly if the two assignments are made nonblocking, irrespective of the order of evaluation. This is because nonblocking assignments are evaluated last and cannot influence each other.

Conversely, use blocking assignments to evaluate combinational logic. Assignments made in combinational logic models are supposed to take

[2]It is quite easy to demonstrate this problem in at least one simulator. Reverse the order of statements and the always procedure will be evaluated first.

immediate effect. The use of nonblocking assignments would often be confusing (and wrong).

Some experts argue that blocking and nonblocking assignments should not be mixed in the same procedure. Certainly, it is not a good idea to use both types of assignment to the same variable as some synthesis tools will object (although such constructs are syntactically valid). This restriction has generally been followed in these notes, but it is possible to mix the two types of assignment such that blocking assignments are used for all combinational logic and nonblocking assignments used for all registered outputs.

Don't use zero delays (#0). They are not necessary and will cause confusion.

## 9.3 Delay Models

SystemVerilog provides five ways to model delays:

Left-hand side (LHS) of blocking assignments:

#5 a = b;

Right-hand side (RHS) of blocking assignments:

a = #5 b;

LHS of nonblocking assignments:

#5 a ¡= b;

RHS if nonblocking assignments:

a ¡= #5 b;

LHS of continuous assignments:

**assign** #5 a = b;

None of these constructs is *needed* for RTL modelling. (a) and possibly (c) and (d) are useful for testbench writing. (d) is a transport (pure) delay that can be used to model delays in sequential logic (such as clock to output delay in a flip-flop). (e) is an inertial delay that can be used to model delays in combinational logic. To understand why some forms are useful and others not, we need to understand what precisely occurs in each case.

In form (a), the simulator waits for, e.g. 5 time units and then executes the assignment. Any changes to inputs during this wait period are ignored. Clearly this does not model real hardware, but is useful for describing a waveform in a testbench.

Form (b) causes the present value of the RHS to be scheduled for assignment at some point in the future. This is a transport or pure delay – every change, no matter how rapid is transmitted to the output. This could

be used to model, say, a transmission line. This is not particularly useful for testbench design.

The third form, (c), again causes a 5 unit wait before assigning the then present value of the right hand side. Again, any intermediate changes are ignored and so this does not model real hardware. This can be used in testbenches, but has no advantage over form (a).

As noted, (d) can be used to model delays in sequential logic. A transport delay is appropriate here, in contrast to the last form.

The final form, (e), models an inertial delay. Real gates require pulses of a minimum width to switch. The width of a pulse is (roughly) proportional to its energy. A real system needs a certain amount of energy to change state. A pulse with a width less than the delay is ignored. Therefore, this form of delay best models combinational logic.

6.2.5 Timing and logic checks

In developing digital systems, we assume certain types of behaviour such as discrete logic levels. A further assumption, discussed in Chapter 12, is that only one input to a flip-flop can change at one time. For example, the $D$ input to a flip-flop must have changed and be stable for a short period before the clock changes. Failure to observe this condition may result in an unpredictable output. In the worst case, the output of a flip-flop can exist in a *metastable* state somewhere between logic 1 and logic 0 for an *indeterminate* time. This unpredictability is not desirable. If we were verifying our designs by simulation, it would clearly be helpful if we were alerted to possible timing problems and to illegal combinations of inputs. VHDL provides the **assert** statement to generate warning messages. The **assert** statement is ignored by synthesis tools.

The form of an **assert** statement is as follows:

**assert** *condition*
**report** *message*
**severity** *level*;

The *condition* is a Boolean expression that we normally expect to be true. If the condition is false the message in the **report** part is printed. The severity level may be note, warning, error or failure. An error or failure will usually cause the simulation to halt at that point. The **report** or **severity** clauses may be omitted. It is also possible to omit the **assert** part, in which case the message in the **report** part will always be printed. **Assert** statements may be included in sequential code or in concurrent code. The difference is that a concurrent assert will only be activated when one of the signals in the condition clause changes, while the sequential assert will be evaluated whenever it is reached in a process or other sequential block.

In Section 6.2.2 it was noted that an asynchronous set and reset should

not both be at logic 0. This condition could be verified by the following assert statement

**assert** (Set ='1' **or** Reset = '1')
**report** "Set and Reset are both asserted"
**severity** WARNING;

Thus if both inputs are at 0, the message is printed. Because we are stating what we expect to be true, the logic may appear to be counter-intuitive. We could equally state the condition that we are checking for and invert it:

**assert** (**not**(Set ='0' **and** Reset = '0'))

If we wish to check that the $D$ input has stabilized before the clock input changes, we can use a form of the 'stable attribute:

**assert** (**not**(Clk ='1' and Clk'EVENT **and not** D'STABLE(3 NS)))
**report** "Setup time violation"
**severity** WARNING;

Thus, we expect that the condition that there has been a clock edge *and* $D$ has *not* been stable for 3 ns is *not* normally true.

The hold time of a flip-flop is defined as the time after a clock edge for which a data input must be stable. This can be similarly defined.

**assert** (**not**(Clk ='1' **and** D'EVENT **and not** Clk'STABLE(5 NS)))
**report** "Hold time violation"
**severity** WARNING;

The **assert** statement is *passive*, meaning that there is no signal assignment. Passive processes and statements may be included in the entity part of a declaration. The advantage to doing this is that the check applies to *all* architectures and does not have to be restated for every architecture. A model of a $D$ flip-flop with an asynchronous reset and set, a clock enable, setup time and asynchronous input checks and propagation delays is shown below.

**library** IEEE;
**use** IEEE.std_logic_1164.**all**;
**entity** D_FF **is**
**generic** (CQ_Delay, SQ_Delay, RQ_Delay: DELAY_LENGTH := 5 NS;
Setup: DELAY_LENGTH := 3 NS);
**port** (D, Clk, Set, Reset, Enable : **in** std_logic;
Q : **out** std_logic);
**begin**
**assert** (**not**(rising_edge(Clk) **and not** D'STABLE(Setup)))
**report** "Setup time violation"
**severity** WARNING;
**end entity** D_FF;

```
architecture behavioural of D_FF is
begin
p0: process (Clk, Set, Reset) is
begin
assert (not(Set ='0' and Reset = '0'))
report "Set and Reset are both asserted"
severity ERROR;
if Set = '0' then
Q ¡= '1' after SQ_Delay;
elsif Reset = '0' then
Q ¡= '0' after RQ_Delay;
elsif rising_edge(Clk) then
if (Enable = '1') then
Q ¡= D after CQ_Delay;
end if;
end if;
end process p0;
end architecture behavioural;
```

## Summary

The SystemVerilog simulation model has five distinct event queues. The ordering of events within these queues is not defined. This leads to nondeterminism in SystemVerilog simulations. It is possible to make SystemVerilog simulations deterministic by adopting well-tried design styles. These styles are also appropriate for RTL synthesis. There are five possible styles of delay modelling, but only two of these are useful for RTL modelling and one other is best suited to testbench writing.

# Chapter 10

# SystemVerilog Synthesis

Verilog was originally designed as a hardware *description* language. In other words, the language was designed to model the behaviour of existing hardware, not to specify the functionality of proposed hardware. Moreover, when the Verilog language was originally designed, there were no automatic synthesis tools in widespread use. Therefore the meaning of different SystemVerilog constructs in hardware terms was derived some years after the language was standardized. The consequence of this is that parts of SystemVerilog are not suitable for synthesis.

We should define, at this point, what we mean by the term *synthesis*. The long-standing objective of design automation tool development has been to compile high-level descriptions into hardware in much the same way that a computer software program is compiled into machine code.

Figure 10.1 shows a simplified view of the design process. After a specification has been agreed, a design can be partitioned into functional units (architectural design). Each of these functional units is then designed as a synchronous system. The design of these parts can be done by hand, as described in Chapter 5. Thus a state machine is designed by formulating an ASM chart, deriving next state and output equations and implementing these in combinational logic. At this point, the gates and registers of the design can be laid out and wired up on an integrated circuit or programmable logic device.

Figure 10.1 shows how synthesis tools can automate parts of this process. RTL (Register Transfer Level) Synthesis tools take a SystemVerilog description of a design in terms of registers, state machines and combinational logic functions and generate a netlist of gates and library cells. As we will see, the SystemVerilog models described in Chapters 4, 5, 6 and 7 are mostly suitable for RTL synthesis. Behavioural synthesis tools, on the other hand, take algorithmic SystemVerilog models and transform

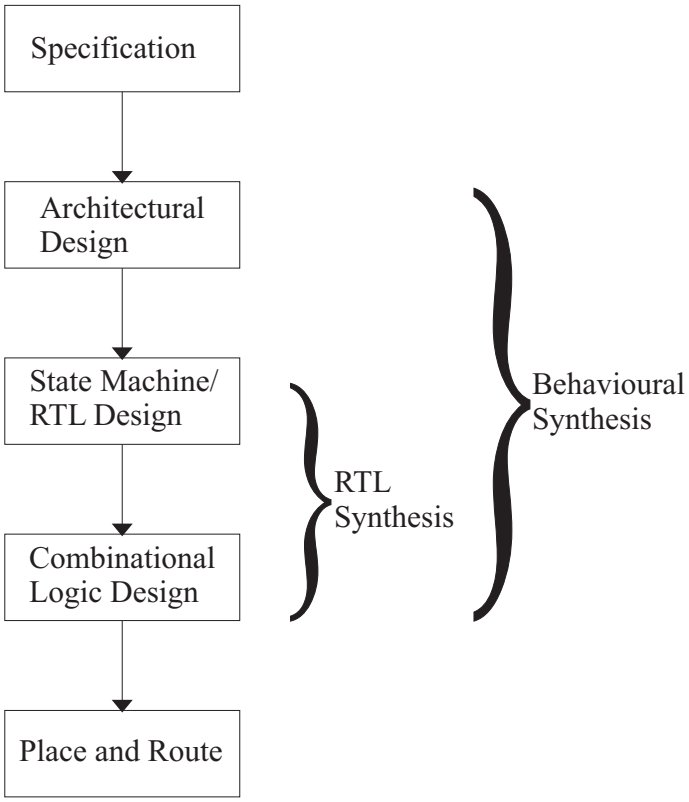



Figure 10.1: High-level design flow.

them to gates and cells. The user of a behavioural synthesis system would not have to specify clock inputs for instance, simply that a particular operation has to be completed within a certain time interval. RTL synthesis tools are gaining widespread acceptance; behavioural synthesis tools are still relatively rare. Although this chapter (and this course) is about RTL synthesis, it is likely that in a few years behavioural synthesis tools will be widely accepted, in a manner analogous to the way that high-level software programming languages such as Java are coming to replace lower-level languages such as C.

The last stage of the synthesis process, place and route, is carried out by separate specialized tools. In the case of programmable logic, the manufacturers of the programmable logic devices often supply these tools.

## 10.1 RTL synthesis

The functions carried out by an RTL synthesis tool are essentially the same as those described in Chapter 5. The starting point of the synthesis process is a model (in SystemVerilog) of the system we wish to build, described in terms of combinational and sequential building blocks and state machines. Thus we have to know all the inputs and outputs of the system, including the clock and resets. We also have to know the number of states in state machines – in general, RTL synthesis tools do not perform state minimization. From this we can write SystemVerilog models of the parts of our system. In addition, we may wish to define various *constraints*. For instance, we might prefer that a state machine be implemented using a particular form of state encoding. We almost certainly have physical constraints such as the maximum chip size and hence the maximum number of gates in the circuit and the minimum clock frequency at which the system should operate. These constraints are not part of SystemVerilog, in the sense that they do not form part of the simulation model, and are often unique to particular tools, but *may* be included in the SystemVerilog description.

The IEEE standard 1364.1-2002 defines a subset of Verilog for RTL synthesis. The purpose of this standard is to define the minimum subset that can be accepted by *any* synthesis tool. Throughout this book, we have advocated the use of and adherence to various standards. SystemVerilog is based on the 2001 enhancements to Verilog. This guide is based on the subset of the 1364.1 standard that applies to SystemVerilog.

### 10.1.1 Non-synthesizable SystemVerilog

In principle most features of SystemVerilog could be translated into hardware. In general, those parts of SystemVerilog that are not synthesizable are constructs in which exact timing is specified and structures whose size is not completely defined. Poorly written SystemVerilog may result in the synthesis of unexpected hardware structures. These will be described later.

The following SystemVerilog constructs are either ignored or rejected by RTL synthesis tools.

All delay clauses (e.g. #10). Delays are *simulation* models. A model can be synthesized to meet various *constraints*, but cannot be synthesized to meet some exact timing model. For instance, it is not possible to specify that a gate will have a delay of exactly 5 ns. It is reasonable, on the other hand, to require a synthesis tool to generate a block of combinational logic such that its total delay is less than, say, 20 ns.

File operations suggest the existence of an operating system. Hence file operations cannot be synthesized and would be rejected by a synthesis tool.

Real data types are not inherently *unsynthesizable*, but will be rejected by synthesis tools because they require at least 32 bits, and the hardware required for many operations is too large for most ASICs or FPGAs.

Initial blocks will be ignored. Hardware can't exist for a limited period of time and then disappear!

Switch-level structures (e,g. tranif1) and user-defined primitives are rejected. Signal strengths are ignored.

### 10.1.2 Inferred flip-flops and latches

It is important to appreciate that synthesis tools (like most computer software) are basically stupid. While there are reserved words in SystemVerilog to specify whether a model is combinational or sequential, inconsistent models may only generate warnings. Therefore the fundamental problem with synthesizing SystemVerilog models is to ensure that the hardware produced by the synthesis system is what you really want. One of the most likely 'errors' is the creation of additional flip-flops or latches. Therefore, in this section, we will describe how the existence of flip-flops and latches is inferred.

A flip-flop or latch is synthesized if a net or register holds its value over a period of time. In SystemVerilog a net holds its value until it is given a new value. A flip-flop or latch is created implicitly if some paths through a

procedure have assignments to a net or register while others do not. This typically happens if a **case** statement or an **if** statement is incomplete in the sense that one or more branches do not contain an assignment to a register while other branches do contain such an assignment, or if the **if** statement does not contain a final **else**.

The term ‘flip-flop’ refers here to a memory element triggered by an edge of the clock. ‘Latch’ refers to a level-sensitive device, controlled by some signal other than the clock. Thus a flip-flop would be created if the event list of a block has **posedge** or **negedge** expression, while a latch would be created if the level value of a net were used instead.

In principle, therefore, procedural blocks with various edge-triggered and level-sensitive expressions could be synthesized. In practice, synthesis tools recognize a small number of fairly simple patterns, as shown in the rest of this section. These examples can act as templates for larger examples. It should be noted that in all these examples, the net names are not significant to the synthesis tool. Thus a clock net might be called ‘Clock’ or ‘Clk1’ or, with equal validity, ‘Data’. Note, however, that good software engineering practice should be applied and *meaningful* identifiers should be used for the benefit of human readers.

### Level-sensitive latch

If we really want to create a latch we can specify it using the particular form of the always block:

**always_latch** @(Ctrl **or** A)
**if** (Ctrl)
Z ¡= A;

A general always block can also be interpreted as a latch. The following example shows the SystemVerilog that would be interpreted to specify a level-sensitive latch by an RTL synthesis tool.

**always** @(Ctrl **or** A)
**if** (Ctrl)
Z ¡= A;

The always statement has an event list containing the net (or register) Ctrl and the net, A, which is assigned to the output. Therefore the statement is executed when one of Ctrl or A changes. Z is assigned the value of A if Ctrl has just changed to a 1. While Ctrl is 1, any change in A is transmitted to the output. Otherwise, no assignment to Z is specified. Therefore it may be inferred that Z holds its value, and hence it is inferred that Z is a registered net. This inference can be avoided if the **else** clause is included:

**always** @(Ctrl **or** A)
**if** (Ctrl)
Z ¡= A;
**else**
Z ¡= 1'b0;

The value of Z is therefore Ctrl AND A. On the other hand, specifying a block as a latch when it isn't should generate a warning.

**always_latch** @(Ctrl **or** A)
**if** (Ctrl)
Z ¡= A;
**else**
Z ¡= 1'b0; // This is inconsistent

**Case** statements are interpreted in a similar manner.

**always** @(Sel, A, B)
**case** (Sel)
2'b00 : Y ¡= A;
2'b10 : Y ¡= B;
**default**;
**endcase**;

The **default** clause covers the patterns 01 and 11 (and combinations with X and Z, although they are irrelevant to synthesis). If it were omitted the **case** statement would still be syntactically correct. When Sel is one of these two misssing patterns Y is assumed to hold its value. Hence the circuit of Figure 10.2 is synthesized.

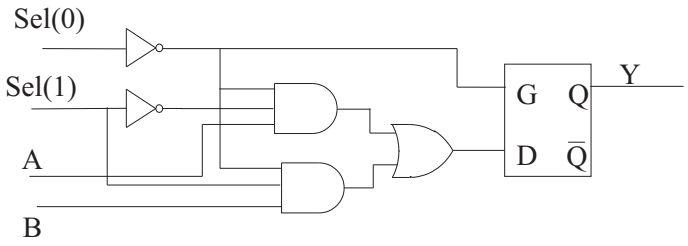



Figure 10.2: Circuit synthesized from incomplete case statement.

Note that the latch used in these examples would be taken from a library. Such elements cannot be synthesized from first principles by a synthesis tool. The continuous assignment statement

**assign** y = E ? D : y;

in which a signal appears on both the left- and right-hand sides of the net assignment, may be synthesized to the circuit of Figure 10.3. This is

apparently functionally correct, but it contains a potential hazard and is therefore a poor latch design. Such constructs should be avoided.

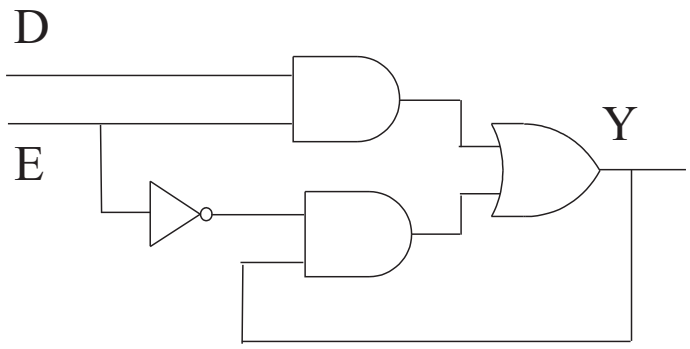



Figure 10.3: Asynchronous circuit synthesized from **when...else**.

**Edge-sensitive flip-flop**

As described in Chapter 4, edge-sensitive behaviour may be modelled by putting a **posedge** or **negedge** expression in an event list:

```
always_ff @(posedge clk)
q ¡= d;
```

or

```
always @(posedge clk)
q ¡= d;
```

The **posedge** and **negedge** statements are interpreted by a synthesis system to model edge-sensitive behaviour. Hence net assignments that can only be reached by fulfilling an edge-sensitive condition will be interpreted as assignments to registered nets. It should be remembered that the net name itself is not meaningful to the synthesis tool.

Asynchronous sets and resets are modelled using level-sensitive **if** clauses:

```
always_ff @(posedge clk or posedge reset)
if (reset)
q ¡= 1'b0;
else
q ¡= d;
```

This structure would be interpreted, correctly, as a positive-edge triggered flip-flop with an active high asynchronous reset. The reset is tested before the clock and therefore has an effect irrespective of the clock. The clock net to which the flip-flop is edge-sensitive should be tested in the

last branch of the **if** statement. Similarly, synchronous sets and resets and clock enable inputs as described in Chapter 4 will be correctly interpreted by an RTL synthesis tool.

We saw in the previous chapter that the SystemVerilog simulation model means that nonblocking assignments do not take effect until all other events have been processed at the current simulation time. Blocking assignments, without delays, on the other hand take immediate effect. The synthesized forms of non-blocking and blocking assignments should therefore be different. The following fragment of SystemVerilog synthesizes to the structure shown in Figure 10.4.

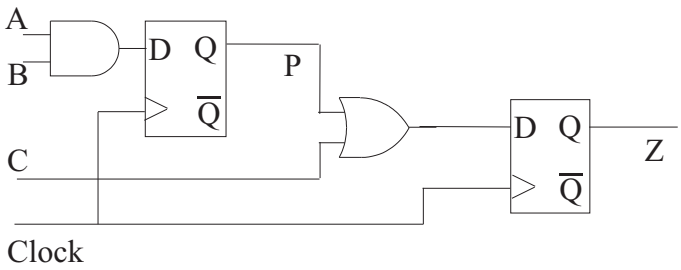



Figure 10.4: Circuit synthesized by nonblocking assignments.

**always_ff** @(**posedge** clock)
**begin**
P ¡= A & B;
Z ¡= P | C;
**end**

In the first nonblocking assignment, P is given a value. When P is referenced in the second assignment, the new value of P has not yet taken effect. Therefore the previous value of P is used. The value of P (and of Z) is not updated until the procedure resumes, at the next clock edge. Therefore P behaves exactly as if its value were stored in a flip-flop.

By contrast, a blocking assignment takes effect immediately. Therefore the following piece of code, in which P is assigned a value through a nonblocking assignment, is synthesized to the structure of Figure 10.5.

**always_ff** @(**posedge** clock)
**begin**
P = A & B;
Z ¡= P | C;
**end**

In general, use nonblocking assignments to model edge-triggered flip-flops. You can use blocking assignments to model temporary variables,

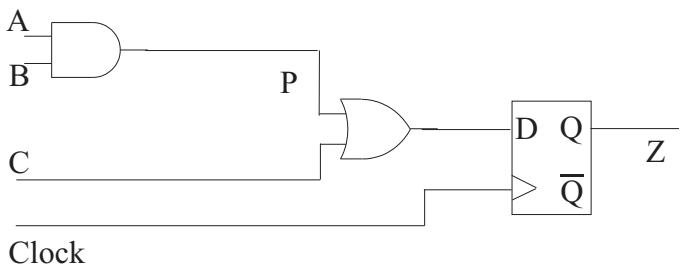



Figure 10.5: Circuit synthesized using blocking assignment.

such as P in the last example, *BUT* SystemVerilog does not allow these temporary variables to be distinguished from other registers. Therefore, to avoid ambiguity and potential race problems it is best to use only nonblocking assignments in edge-triggered procedures. Note also that according to the IEEE 1364.1 standard, the event list should only contain edge-sensitive events.

### 10.1.3 Combinational logic

In general, if a piece of hardware is not a level-sensitive or edge-sensitive sequential unit, it must be a combinational unit. Therefore, a SystemVerilog description that does not fulfil the conditions for synthesis to level-sensitive or edge-sensitive sequential elements must by default synthesize to combinational elements. Hence the problem of describing combinational hardware in SystemVerilog is to ensure that we do not accidentally cause the synthesis tool to infer the existence of registers.

To ensure that combinational logic is synthesized from a SystemVerilog procedure, we must observe three conditions. First, we must not have any edge-triggered events in the event list. Secondly, if a **reg** has a value assigned in one branch of an **if** statement or a **case** statement, that reg must have a value assigned in every branch of the statement (or it must have a value assigned before the branching statement – see below). Finally, all the nets sensed either as branching conditions or in assignments must be included in the event list of the process.

For example, the following is a model of a state machine with two states, two inputs and two outputs.

**module** Fsm (**output reg** OutA, OutB,
**input** Clock, Reset, InA, InB);
**enum** {S0, S1, S2} PresentState;
**always_ff** @(**posedge** Clock **or posedge** Reset)

```
if (Reset)
PresentState ¡= S0;
else
case (PresentState)
S0: begin
OutA ¡= 1'b1;
if (InA)
PresentState ¡= S1;
end
S1: begin
OutA ¡= InB;
OutB ¡= 1'b1;
if (InA)
PresentState ¡= S2;
end
S2: begin
OutB ¡= InA;
PresentState ¡= S0;
end
endcase
endmodule
```

Although this is an acceptable simulation model, if it were synthesized, OutA and OutB would be registered in addition to PresentState, because they have values assigned to them within an edge-triggered procedure. Thus we can divide the model into two procedures, one combinational and one sequential. We will use blocking assignments in the "combinational" procedure to ensure that the all the values are updated before they are read into registers.

```
module Fsm (output reg OutA, OutB,
input Clock, Reset, InA, InB);
enum {S0, S1, S2} PresentState, NextState;
always_ff @(posedge Clock or posedge Reset)
if (Reset)
PresentState ¡= S0;
else
PresentState ¡= NextState;
always_comb
case (PresentState)
S0: begin
OutA = 1'b1;
if (InA)
```

```
NextState = S1;
else
NextState = S0;
end
S1: begin
OutA = InB;
OutB = 1'b1;
if (InA)
NextState = S2;
else
NextState = S1;
end
S2: begin
OutB = InA;
NextState = S0;
end
endcase
endmodule
```

This will, again, simulate as a state machine giving apparently correct behaviour. When synthesized, however, OutA and OutB will be registered through asynchronous latches, because in state S0 no value is assigned to OutB and hence OutB holds onto its value. Similarly in state S2, no value is assigned to OutA. This should generate warnings.

This error can be resolved by explicitly including an assignment to both OutA and OutB in every branch of the **case** statement. Alternatively, both signals can be given default values at the start of the procedure:

```
always_comb
begin
OutA = 1'b0;
OutB = 1'b0;
case (PresentState)
S0: begin
OutA = 1b'1;
if (InA)
NextState = S1;
else
NextState = S0;
end
S1: begin
OutA = InB;
OutB = 1b'1;
```

```
if (InA)
NextState = S2;
else
NextState = S1;
end
S2: begin
OutB = InA;
NextState = S0;
end
endcase
end
```

This procedure now synthesizes to purely combinational logic, while the other procedure synthesizes to edge-triggered sequential logic.

Note, however, that it is not essential to use the **always_comb** reserved word. It is also possible to use an **always** block, with a default event list:

```
always @(*)
```

It is also possible to list those signals that should cause the block to be evaluated. For example, suppose that the block

Most synthesis tools will (or should) give a warning, however. A piece of combinational logic will be synthesized with three inputs (PresentState, InA and InB) and three outputs (NextState, OutA and OutB). Hence a change at any of the inputs could cause a change at an output. The SystemVerilog model above has only one signal in its event list (PresentState). Therefore this model and the synthesized circuit may behave differently when simulated. To avoid this, all the signals to which the combinational logic is sensitive should be included in the sensitivity list. The 'correct' interpretation of a model with an incomplete sensitivity list such as:

```
always @(a)
q = a & b;
```

is the circuit shown in Figure 10.6(a). The lower flip-flop of this circuit will always have a 0 output, so in theory this circuit can be optimized to that of Figure 10.6(b).

The complete, correct model of the example state machine is shown below.

```
module Fsm (Clock, Reset, InA, InB, OutA, OutB);
output OutA, OutB;
input Clock, Reset, InA, InB;
reg OutA, OutB;
parameter S0 = 0, S1 = 1, S2 = 2;
reg [0:2] PresentState, NextState;
always @(posedge Clock or posedge Reset)
```

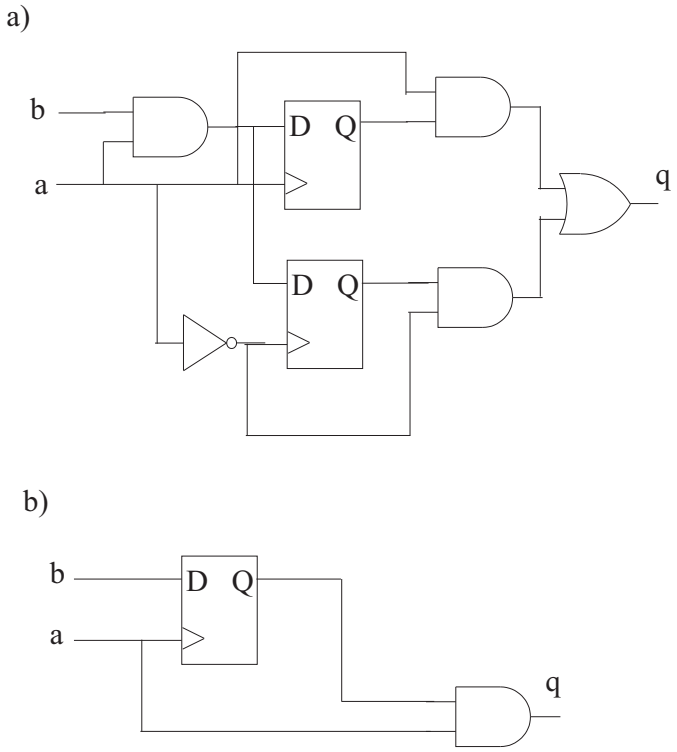



Figure 10.6: (a) Circuit synthesized from incomplete sensitivity list; (b) optimized circuit.

```
if (Reset)
PresentState ¡= S0;
else
PresentState ¡= NextState;
always @(PresentState or InA or InB)
begin
OutA = 1'b0;
OutB = 1'b0;
case (PresentState)
S0: begin
OutA = 1b'1;
if (InA)
NextState = S1;
else
NextState = S0;
end
S1: begin
OutA = InB;
OutB = 1b'1;
if (InA)
NextState = S2;
else
NextState = S1;
end
S2: begin
OutB = InA;
NextState = S0;
end
endcase
end
endmodule
```

The style of coding will also influence the final hardware. For example, nested **if ..... else** blocks, such as the priority encoder of Section 4.4.2, will tend to result in priority encoding and hence long chains of gates and large delays. On the other hand, **case** statements, such as the state machine, above, will tend to be synthesized to parallel multiplexer-type structures with smaller delays. (But see section 9.2.3, below.) Similarly, shift operations will result in simpler structures than multiplication and division operators.

### 10.1.4 Summary of RTL synthesis rules

It is easy to make mistakes and to accidentally create latches when combinational logic is intended (or worse, to deliberately create latches, when you really want a flip-flop – see section 5.5.4). Table 10.1 summarizes the rules for creating combinational and sequential logic from processes.

Table 10.1: Summary of RTL synthesis rules.

| | Event List |
|---|---|
| **Combinational Logic** | All inputs in event list (nets and registers on RHS of assignments and us |
| **Latches** | All inputs in event list (nets and registers on RHS of assignments and us |
| **Flip-flops** | Edge-sensitive clock and asynchronous set & reset only. |

There is one further rule that applies to all synthesisable logic: Do not assign a value to a net or reg in two or more procedures. The only exception to this rule is the case of three state logic, as in the bus in the microprocessor example of Chapter 7. You should be able to draw a block diagram of your design, with each procedure represented by a box. If two boxes appear to be driving the same wire, you have done something wrong. (Indeed, if you can't draw the block diagram, you have made a really serious mistake!)

## 10.2 Constraints

For any non-trivial digital function, there exist a number of alternative implementations. Ideally, a digital system should be infinitely fast, infinitesimally small, consume no power and be totally testable. In reality, of course, this ideal is impossible. Therefore, the designer has to decide what his or her objectives are. These objectives are expressed to the synthesis tool as *constraints*. Typically, a design has to fit on a particular FPGA and has to operate at a particular clock frequency. Thus two constraints of area and speed have to be specified. It is possible that these constraints will be in conflict. For example, a design may fit on a particular FPGA, but not work at the desired speed – to reach the desired speed may require more logic and hence more area, as illustrated in Figure 10.7. Assuming that CMOS logic is used and that the gate delays are identical, the circuit of Figure 10.7(a) would need 16 transistors and have a maximum delay of 4 units, while the circuit of Figure 10.7(b) requires 18 transistors and has a maximum delay of 3 units.

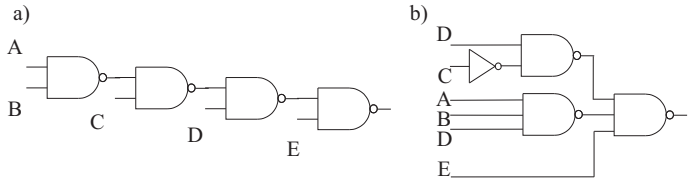



Figure 10.7: Two versions of a combinational circuit; (a) smaller, slower; (b) larger, faster.

### 10.2.1 Attributes

Synthesis constraints can be expressed in two ways: as SystemVerilog attributes in the model description or as some other format in a separate file. There is no standard between tools for either the type of constraints or the format in which they may be expressed. Thus the 1364.1 standard defines 16 attributes that can be included in the SystemVerilog description. In general, attributes are used to pass information to synthesis tools, but are ignored by simulators.

For example, 1364.1 defines one attribute for specifying the state encoding, e.g.

(* synthesis, fsm_state="onehot" *) reg [0:2] present_state;

This might instead be expressed in a separate constraints file using a format like:

define_attribute fsm_state present_state "onehot"

Other example attribute definitions could be as follows:

(* synthesis, black_box *)

(* synthesis, implementation="ripple" *)

In general the type and format of constraints are unique to particular synthesis tools; in the following sections we will discuss only the general types of constraints that can be specified.

### 10.2.2 Area and structural constraints

#### State encoding

As discussed in Chapter 5, a state machine with $s$ states can be implemented using $m$ state variables, where

$2^{m-1} < s \leq 2^m$

There are $\frac{(2^m)!}{(2^m-s)!}$ possible state assignments. There is no method for determining which of these assignments will result in minimal combinational next state logic. In addition, other non-minimal state encoding

schemes, such as one-hot exist. No RTL synthesis tools attempt to tackle the general state assignment problem. Heuristic methods may be able to choose either a binary counting sequence or one-hot encoding. Therefore one design constraint that can be specified is the state encoding method, either using the IEEE 1364.1 style or by specifying the code with a keyword, as shown above.

### Resource constraints

The use of a particular technology may constrain the type of structures that can be created. Features of different FPGA technologies will be discussed later in this chapter. Having selected a particular technology, a range of different-sized devices may exist, and very often it is desirable to select the smallest possible. Thus the specification of a particular device is a constraint on the synthesis process.

As a single ASIC or FPGA has to be connected via a printed circuit board to other devices, the functionality of each pin may have to be determined in advance of the synthesis. Therefore another constraint is the association of a signal with a particular pin.

Under some circumstances, complex logic blocks may be reused. For example, the following piece of code can be implemented with two adders or with one adder and two multiplexers.

**if** (Select)

q = a + b;

**else**

q = c + d;

A synthesis constraint can choose whether resources may be shared, either at a local level or globally. Such choices have implications for both the area and speed of the final design. The following attribute can be attached to a module:

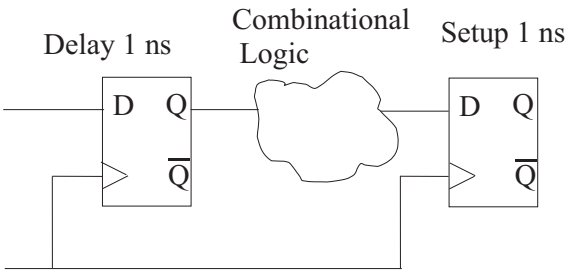
(* synthesis, op_sharing *)

Finally, it may be desirable to describe a function in SystemVerilog in order to verify the correct operation of the rest of the system, but when the system is synthesized we would rather use a predefined library component to implement that function instead of synthesizing the function from first principles. Therefore we can designate that a particular unit is a ‘black box’ that we will incorporate from a library, e.g.

(* synthesis, black_box *)

### Timing constraints

If we want a circuit to operate synchronously with a clock at a particular frequency, say 20 MHz, we know that the maximum delay through the state registers and the next state logic is the reciprocal of the clock frequency, in this case 50 ns. Therefore a constraint on the synthesis tool can be expressed as the clock frequency or as the maximum delay through the combinational logic, as shown in Figure 10.8.




Clock Frequency is 20 MHz
Clock period is 50 ns
Maximum delay through Combinational Logic is 48 ns

Figure 10.8: Basic timing constraint.

The difficulty, from the synthesis point of view, with this approach is that the delay through the combinational logic can only be estimated. The exact delay depends on how the combinational logic is laid out, and hence the delay depends on the delay through the interconnect. Therefore the synthesis is performed using an estimate of the likely delays. Having generated a netlist, the low-level place and route tool attempts to fit the design onto the ASIC or FPGA. The place and route tool can take into account the design constraint – the maximum allowed delay – and the delays through the logic that has been generated. At this stage, it may become apparent that the design objective cannot be achieved, so the design would have to be synthesized again with a tighter timing constraint to allow for the extra time in the routing. This can mean that the final goal is never reached. To speed up hardware more operations are performed concurrently, which means that the design is larger. Hence the design is harder to place and route, and hence the routing delays increase, *ad infinitum*.

More specific timing constraints can be applied to selected paths. If a design is split between two or more designers, the signal path between registers in two parts of the design may include combinational logic be-

longing to both parts of the design. If both parts of combinational logic were each synthesized without allowing for the existence of the other, the total delay between registers could be greater than one clock period. Therefore timing constraints can be placed upon paths through the input and output combinational logic in a design, as shown in Figure 10.9.

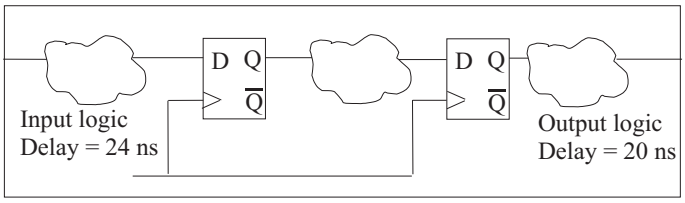



Figure 10.9: Input and output timing constraints.

### 10.2.3 full_case and parallel_case attributes

Many SystemVerilog designers attach the full_case and parallel_case attributes to case statements, without thinking. In general, this is bad practice and these directives *should not* be used. The attributes only apply to synthesis and may cause the synthesised hardware to have different functionality to that simulated in RTL. To understand why these attributes may cause problems, we need to understand what the terms “full case” and “parallel case” mean.

In SystemVerilog, each case item (the term to the left of the colon in each branch), is tested in turn against the case expression. A full case statement is one in which every combination of 0, 1, z, x in the case expression can be matched against one (or more) case item(s). This applies to casez (and casex) statements, in which there are don’t care terms. It is not required that a case statement is full, but, on the other hand, this condition can be achieved, simply by including a default item in the case statement. If a case statement is not full, and the uncovered alternatives include combinations of 0 or 1, the correct interpretation in synthesis would be to create a latch. For example, the following is a simplified version of the priority encoder from Chapter 4. If the default item were omitted, the pattern 4b’0000 (or indeed, 4b’000z or any pattern that included an x) would not be matched and the case statement would not be full.

```
    always @(a)
casez (a)
4'b1??? : y = 2'b11;
4'b01?? : y = 2'b10;
```

4’b001? : y = 2’b01;
4’b0001 : y = 2’b00;
**default** : y = 2’b00;
**endcase**

By including the full_case attribute, the designer is telling the synthesis tool to treat any unspecified combinations of inputs as don’t care conditions – in other words, to assume that a default item exists. In other words, the simulated and synthesised interpretations of the code would be different. Of course, if the default item is present, as above, the full_case attribute is redundant!

The recommendation is therefore, to omit the full_case atttribute, and to include a default item. The output values from the default should be meaningful values (not x or z), otherwise a latch might result. In the above example, setting the default result to 2b’xx would mean that no valid hardware could be produced for the default cases and hence a latch would be implied. It is usual to include the default item as the last case item.

A parallel case statement is one in which each combination of inputs is covered exactly once. It is perfectly legal to write a case statement such that an input pattern can match to two or more case items. Because the items are matched in the order they are written, this implies the existence of priority logic. The use of a case statement suggests, however, that parallel logic should be used. Adding the parallel_case attribute forces the synthesis tool to treat the case statement as if it really is parallel. Inevitably, this will result in synthesised hardware that behaves differently to what was simulated at RTL.

The example above is parallel. Including the parallel_case attribute is therefore redundant. The following example is not parallel:

**always** @(a)
**casez** (a)
4’b1??? : y = 2’b11;
4’b?1?? : y = 2’b10;
4’b??1? : y = 2’b01;
4’b???1 : y = 2’b00;
**default** : y = 2’b00;
**endcase**

In a simulation, this code would appear to function in an identical manner to the priority encoder, above. The pattern 4’b1011 would match the first pattern, and so 2’b11 would result. This pattern also matches the third and fourth items. Clearly, therefore the case statement is not parallel. By specifying the parallel_case attribute

So, do not use the parallel_case attribute. If you must have priority logic, use if statements. If you use case statements, read the messages from the synthesis tool. If the tool reports that your case statement is not parallel, *change the case statement* to make it parallel.

## 10.3 Synthesis for FPGAs

In principle, an RTL model of a piece of hardware coded in SystemVerilog can be synthesized to any target technology. In practice, the different technologies and structures of ASICs and FPGAs mean that certain constructs will be more efficiently synthesized than others and that some rewriting of SystemVerilog may be needed to attain the optimal use of a particular technology.

In this section we will compare two FPGA technologies and show how the SystemVerilog coding of a design can affect its implementation in a technology. The descriptions of the technologies are deliberately simplified.

Xilinx FPGAs are based on static RAM technology. Each FPGA consists of an array of the configurable logic blocks (CLBs) shown in Figure 1.15. Each logic block has two flip-flops and a combinational block with eight inputs. Each flip-flop has an asynchronous set and reset, but only one of these may be used at one time. Each flip-flop also has a clock input that can be positive or negative edge-sensitive, and each flip-flop has a clock enable input. In addition to the CLB shown, a number of three-state buffers exist in the array.

Actel FPGAs are based on antifuse technology. Two types of logic block exist in more or less equal numbers – a combinational block and a sequential block as shown in Figure 1.13. Each flip-flop in a sequential block has an asynchronous reset.

Both types of FPGA therefore have a relatively high ratio of flip-flops to combinational logic. Conventional logic design methods tend to assume that flip-flops are relatively expensive and combinational logic is relatively cheap, and that therefore sequential systems such as state machines should be designed with a minimal number of flip-flops. The large number of flip-flops in an FPGA and the fact that the flip-flops in a Xilinx FPGA or in an Actel sequential block cannot be used without the combinational logic reverses that philosophy and suggests that one-hot encoding is a more efficient state encoding method, particularly for small state machines.

Similarly, a single global asynchronous set or reset is the most efficient way of initializing both types of FPGA. If both set and reset are required it is necessary to use additional combinational logic, hence it is better to have, for example, an asynchronous reset and a *synchronous* set.

In both technologies, the flip-flops are edge-sensitive; therefore level-sensitive latches have to be synthesized from combinational logic. Again, this can waste flip-flops, so level-sensitive designs are best avoided. It is, however, reasonable to assume that any level-sensitive latches will exist as library elements and therefore that they will be hazard-free.

In both technologies, it may be desirable to instantiate predefined library components for certain functions. Not only is the logic defined, but the configuration of logic blocks is already known, potentially simplifying both the RTL synthesis and place and route tasks.

All the foregoing comments distinguish synthesis to FPGAs from synthesis to ASICs in general. The FPGA technologies themselves favour certain SystemVerilog coding styles. For example, the following piece of SystemVerilog shows two ways of describing a 5-to-1 multiplexer.

```
    module Mux1(a, b, c, d, e, s, y);
input a, b, c, d;
input [4:0] s;
output y;
reg y;
always @(s or a or b or c or d or e)
case (s)
5'b00001 : y = a;
5'b00010 : y = b;
5'b00100 : y = c;
5'b01000 : y = d;
default : y = e;
endcase
endmodule
    module Mux2(a, b, c, d, e, s, y);
    input a, b, c, d;
    input [4:0] s;
    output y;
    assign y = s[0] ? a : 1'bZ;
    assign y = s[1] ? b : 1'bZ;
    assign y = s[2] ? c : 1'bZ;
    assign y = s[3] ? d : 1'bZ;
    assign y = s[4] ? e : 1'bZ;
    endmodule
```

These two models have the same functionality when simulated. If version 1 were synthesized to a Xilinx FPGA, two CLBs would be needed. Version 2, on the other hand, can be implemented using the three-state buffers that exist outside the CLBs. Version 2, however, cannot be synthesized to an Actel FPGA as the technology does not support three-state logic, except at the periphery of the FPGA. Clearly, therefore the choice of architecture depends upon which technology is being used.

The two technologies have different limitations with respect to fan-outs. Antifuse technology has a fan-out limit of about 16 (one output can drive up to 16 inputs without degradation of the signal). CMOS SRAM technology has a higher fan-out limit. In practice, this means that a design that can easily be synthesized to a Xilinx FPGA cannot be synthesized to an Actel FPGA without rewriting. For example, an apparently simple structure such as the following fragment cannot be synthesized as it stands because the Enable signal is controlling 32 multiplexers.

```
reg [31:0] a, b;
always @(Enable or b)
if (Enable)
a = b;
else
a = 0;
```

Instead, the Enable signal must be split into two using buffers, and each buffered signal then controls half the bus:

```
reg [31:0] a, b;
wire En0, En1;
buf b0 (Enable, En0);
buf b1 (Enable, En1);
always @(En0 or En1 or b)
begin
if (En0)
a[15:0] = b[15:0];
else
a[15:0] = 0;
if (En1)
a[31:16] = b[31:16];
else
a[31:16] = 0;
end
endmodule
```

A good synthesis tool should recognize the fan-out limits and automatically insert buffers.

## 10.4 Verifying synthesis results

Synthesis should, by definition, produce a correct low-level implementation of a design from a more abstract description. In principle, therefore, functional verification of a design after synthesis should not be needed. For peace of mind, we might wish to check that the synthesized design really does perform the same function as the RTL description. Synthesis does, however, introduce an important extra factor to a design – timing. An RTL design is effectively cycle-based. A task takes a certain number of clock cycles to complete, but we do not really know how long each cycle takes. After synthesis, the design is realized in terms of gates or other functional blocks, and these can be modelled with delays. After placement and routing, we have further timing information in the form of wiring delays, which can be significant and which can affect the speed at which a design can operate.

It is possible, in principle, to verify a synthesized design by comparing it with the original RTL design, using techniques such as model-checking. In practice, such tools are limited to checking interfaces. Static timing analysis can give us information about delays between two points in a circuit, but cannot distinguish between realisable signal paths and false paths that are never enabled in reality. Similarly, a synthesis tool aims to meet timing constraints, but cannot distinguish between true and false paths. Therefore the only way to verify the timed behaviour of a synthesized system is to simulate it.

One approach to checking a design at two levels of abstraction is to simulate both versions at the same time and to compare the results. This is usually a bad idea for two reasons. First, the size of the system to be simulated is at least twice as large as one version in isolation, and therefore slower to execute. Second, there will, as noted, be timing differences. Therefore comparing responses may lead to false warnings.

The testbench design examples described in Chapter 7 are well-suited to simulating post-synthesis designs. In particular, the idea of checking a response by synchronizing to the clock and then waiting for the signal to stabilize is very appropriate for checking timing responses.

## Summary

SystemVerilog was conceived as a description language, but has been widely adopted as a specification language for automatic hardware synthesis. A number of tools exist for RTL synthesis, but behavioural synthe-

sis tools are appearing. Because of its origins, SystemVerilog has some features that are not synthesizable to hardware. The rules for the inference of latches and flip-flops are well defined. Synthesis constraints may be stated in terms of SystemVerilog attributes or as separate inputs to the synthesis tool. To get the most out of an FPGA may require careful writing of the SystemVerilog code.

# Chapter 11

# Testing Digital Systems

In the course of manufacture, defects may be introduced into electronic systems. Systems may also break during use. Defects may not be easy to detect. In this chapter we will discuss the importance of testing, the types of defect that can occur and how defects can be detected. We describe procedures for generating tests and how the effectiveness of tests can be assessed. We conclude with an example of fault simulation in SystemVerilog.

## 11.1 The need for testing

No manufacturing process can ever be perfect. Thus, real electronic systems may have manufacturing defects such as short circuits, missing components or damaged components. A manufacturer needs to know if a system (whether at the level of a board, an IC or a whole system) has a defect and therefore does not work in some way. While a manufacturer does not want to sell bad systems, equally he or she would not want to reject good systems. Therefore the need for testing is economic.

We also need to distinguish between the ideas of *verification* in which the design of a piece of hardware or software is checked and of *testing* in which it is assumed that the design is correct, but that there may be manufacturing faults. This chapter is about the latter concept, but the inclusion of design for test structures *may* help in verifying and debugging a design.

There are, in general, two approaches to testing. We can ask whether the system works correctly (*functional* testing) or we can ask whether the system contains a fault (*structural* testing). These two approaches might at first appear to be equivalent, but in fact the tactic we adopt can make a profound difference to how we develop tests and how long those tests take

to apply. Functional testing can imply a long and difficult task because all possible states of a system have to be checked. Structural testing is often easier, but is dependent upon the exact implementation of a system.

## 11.2 Fault Models

An electronic system might contain a large number of possible defects as a result of the manufacturing process. For example, the printed circuit board could have breaks in connections because of bad etching, stress or bad solder joints. Equally there may be short circuits resulting from the flow of solder. The components on a PCB may be at fault – so-called 'population defects' – caused by having the wrong components; wrongly inserted components or omitted components. The components themselves may fail because the operating conditions exceed the component specifications or because of electromagnetic interference (EMI) or heat.

Similar defects can occur in integrated circuits. Open circuits may arise from electromigration (movement of metal atoms in electromagnetic fields); current overstress or corrosion. Silicon or oxide defects; mask misalignment; impurities and gamma radiation can cause short circuits and incorrect transistor operation. 'Latch-up', caused by transient currents, forces the output of a CMOS gate to be stuck at a logic value. In memory circuits there may be data corruption because of alpha particles or EMI.

Clearly, to enumerate and check for every possible defect in an electronic system would be an enormous task. Therefore a distinction is made between physical *defects* and electrical *faults*. The principle of fault modelling is to reduce the number of effects to be tested by considering how defects manifest themselves. A physical defect will manifest itself as a logical fault. This fault may be static (e.g. shorts, breaks); dynamic (components out of specification, timing failures) or intermittent (environmental factors).

The relative probabilities of faults that appear during tests in manufacturing are shown in Figure 11.1. Dynamic faults may be further divided into timing faults (28%) and driver faults (21%). Timing faults and intermittent faults may be due to poor design. It is difficult to design test strategies for such faults.

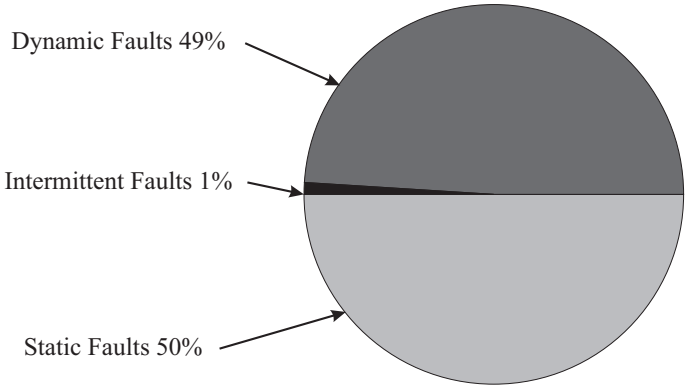



Figure 11.1: Fault Probabilities

### 11.2.1 Single-Stuck Fault Model

Static faults are usually modelled by the *stuck fault model*. Many physical defects can be modelled as a circuit node being either stuck at 1 (s-a-1), or stuck at 0 (s-a-0). Other fault models include stuck open and stuck short faults. Programmable logic and memory have other fault models.

The *Single-Stuck Fault Model* (SSFM) assumes that a fault directly affects only one node and that the node is stuck at either 0 or 1. These assumptions make test pattern generation easier, but the validity of the model is questionable. Multiple faults do occur and multiple faults can theoretically mask each other. On the other hand, the model appears to be valid most of the time. Hence, almost all test pattern generation relies on this model. Multiple faults are generally found with test patterns for single faults.

### 11.2.2 PLA Faults

PLAs consist, not of gates, but of AND and OR logic planes, connected by fuses (or anti-fuses). Thus faults are likely to consist of added or missing fuses, not stuck faults. For example, Figure 11.2 shows part of a PLA, where the output $Z$ is the logical OR of three intermediate terms, $P$, $Q$, $R$.

Each of the intermediate terms is the AND of the three inputs, $A$, $B$, $C$ or its inverse:

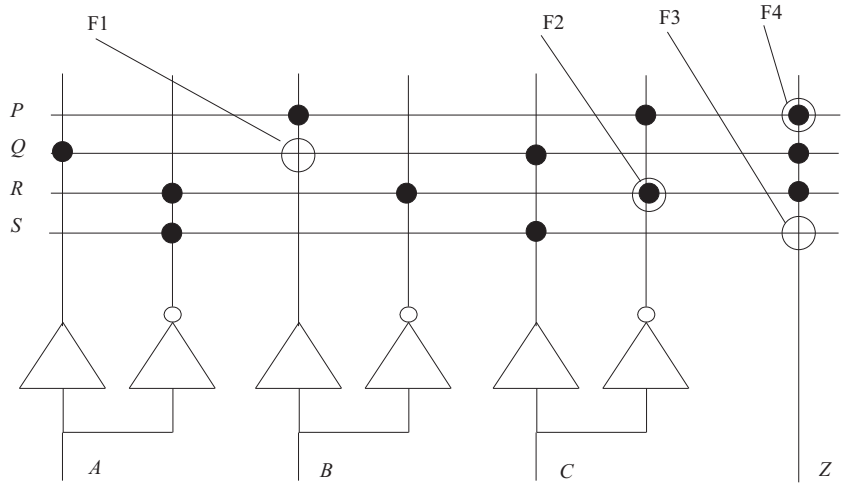



Figure 11.2: PLA Fault Models

$$
\begin{aligned}
Z &= P + Q + R \\
P &= B.\bar{C} \\
Q &= A.C \\
R &= \bar{A}.\bar{B}.\bar{C} \\
S &= \bar{A}.C
\end{aligned}
$$

- Fault F1 is an additional connection causing $Q$ to change from $A.C$ to $A.B.C$. On a Karnaugh map this represents a decrease in the number of 1s circled, therefore this can be thought of as a *shrinkage* fault.
- Fault F2 is a missing connection, causing $R$ to *grow* from $\bar{A}.\bar{B}.\bar{C}$ to $\bar{A}.\bar{B}$.
- Fault F3 causes the *appearance* term $S$ in $Z$.
- Fault F4 causes the *disappearance* of term $P$ from $Z$.

## 11.3 Fault-oriented Test Pattern Generation

Having decided that defects in a system can be modelled as electrical faults, we then need to determine whether or not any of these faults exists in a particular instance of a manufactured circuit. If the circuit were built from discrete transistors or gates, this task could, in theory, be achieved

by monitoring the state of every node of the circuit. If the system is implemented as a packaged integrated circuit, this approach is not practical. We can only observe the outputs of the system and we can only control the inputs of the system. Therefore the task of test pattern generation is that of determining a set of inputs to unambiguously indicate if an internal node is faulty. If we only consider combinational circuits for the moment, the number of possible input combinations for an $n$-input circuit is $2^n$. We could apply all $2^n$ inputs, in other words, perform an exhaustive functional test, but in general we want to find the minimum necessary number of input patterns. It is possible that, because of the circuit structure, certain faults cannot be detected. Therefore it is common to talk about the *testability* of a circuit.

Testability can be a somewhat abstract concept. One useful definition of testability breaks the problem into two parts:

- *Controllability* – can we control all the nodes to establish if there is a fault?
- *Observability* – can we observe and distinguish between the behaviour of a faulty node and that of a fault-free node?

In order to generate a minimum number of test patterns, a fault-oriented test generation strategy is adopted. In the pseudo-code below, a *test* is one set of inputs to a (combinational) circuit. The overall strategy is as follows.

- Prepare a fault list (e.g. all nodes stuck-at 0 & stuck-at 1)
- repeat
  - write a test
  - check fault cover (one test may cover ¿ 1 fault)
  - (delete covered faults from list)
- until fault cover target is reached

Test pattern generation (writing a test) may be random or optimised. This will be discussed in more detail below. One test may cover more than one fault, often faults are indistinguishable. Again this is discussed later.

If we simply want a pass/fail test, once we have found a test for a fault, we can remove faults from further consideration. If we want to diagnose a fault (for subsequent repair) we probably want to find all tests for a fault

to deduce where the fault occurs. The fault cover target may be less than 100%. For large circuits, the time taken to find all possible tests may be excessive. Moreover, the higher the cover, the greater the number of tests and hence the cost of applying the test.

### 11.3.1 Sensitive Path Algorithm

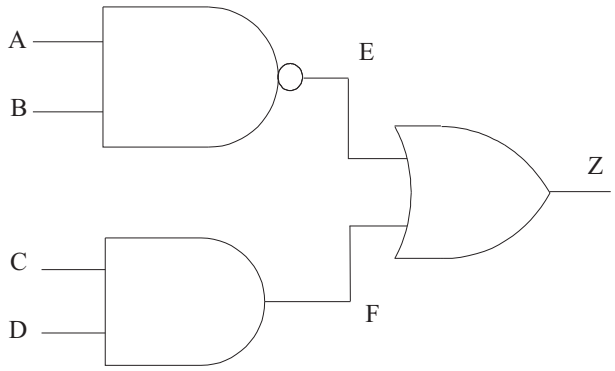



Figure 11.3: Example Circuit for Test Generation

The circuit of Figure 11.3 has 7 nodes, therefore there are 14 stuck faults:
$A$/0, $A$/1, $B$/0, $B$/1, $C$/0, $C$/1, $D$/0, $D$/1, $E$/0, $E$/1, $F$/0, $F$/1, $Z$/0, $Z$/1
where $A$/0 means '$A$ stuck-at-0', etc.

To test for $A$/0, we need to set $A$ to 1 (the fault-free condition – if $A$ were at 0, we would not be able to distinguish the faulty condition from the fault-free state). The presence or otherwise of this fault can only be detected by observing node $Z$. We now have to determine the states of the other nodes of the circuit that allow the state of $A$ to be deduced from the state of $Z$. Thus we must establish a sensitive path from $A$ to $Z$. If node $B$ is 0, $E$ is 1 irrespective of the state of $A$. Therefore, $B$ must be set to a logic 1. Similarly if $F$ is 1, $Z$ is 1, irrespective of $E$, hence $F$ must be 0. To force $F$ to 0, either $C$ or $D$ or both must be 0.

Thus, if the fault $A$/0 exists, $E$ is 1 and $Z$ is 1. If the fault does not exist, $E$ is 0, $Z$ is 0.

We can conclude from this that a test for $A$/0 is $A$=1, $B$=1, $C$=0, $D$=1, for which the fault-free output is $Z$=0. This can be expressed as 1101/0. Other tests for $A$/0 are 1110/0 and 1100/0. Therefore, there is more than one test for the fault $A$/0.

Let us now consider a test for another fault. To test for $E$/1 requires that $F$=0 to make $E$ visible at $Z$. Therefore $C$ or $D$ or both must be 0. To make

$E$=0 requires that $A = B$=1. So a test for $E$/1 is 1101/0. This is the same test as for $A$/0. So one test can cover more than one fault.

The sensitive path algorithm therefore consists of the following steps:

1 Select a fault.

2 Set up the inputs to force the node to a fixed value.

3 Set up the inputs to transmit the node value to an output.

4 Check that the input node values for steps 2 and 3 are consistent.

5 Check for coverage of other faults;

The aim is to find the minimum number of tests that cover all the possible faults, although 100% fault cover may not be possible.

Fan-out and reconvergence can cause difficulties for this algorithm. Improved algorithms (D-algorithm, PODEM) use similar techniques but overcome these drawbacks.

### 11.3.2 Undetectable Faults

Consider the function

$$Z = A.C + B.\bar{C}$$

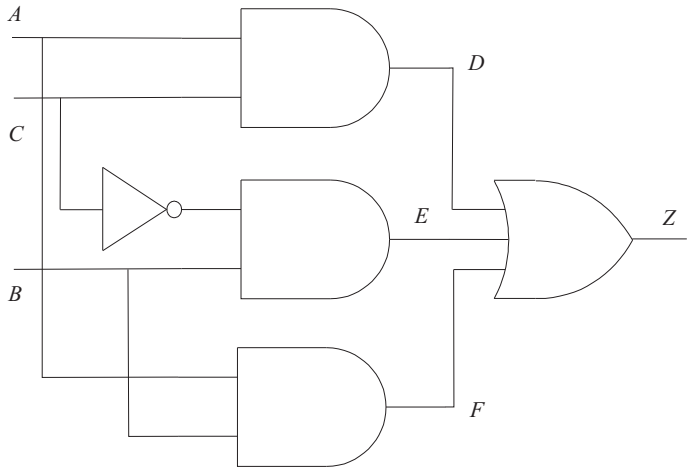



Figure 11.4: Circuit with Redundancy

To avoid hazards, the redundant term may be included, as shown in Figure 11.4:

$$Z = A.C + B.\bar{C} + A.B$$

We will now try to find a test $F$/0. This requires that $F$ be set to 1. Hence, $A = B = 1$. To transmit the value of $F$to $Z$, means that $D = E = 0$ (otherwise $Z$ would be 1, irrespective of $F$). For $E$ to be 0, $B$ must be 0 and/or $C$ must be 1. Similarly, for $D$ to be 0, A must be 0 and/or $C$ must be 0. These three conditions are inconsistent, so no test can be derived for the fault $F$/0.

There are three possible responses to this. Either it must be accepted that the circuit is not 100% testable; or the redundant gate must be removed, risking a hazard; or the circuit must be modified to provide a control input for testing purposes, to force $D$ to 0 when $A = C = 1$.

In general, untestable faults are due to redundancy. Conversely, redundancy in combinational circuits will mean that those circuits are not fully testable.

### 11.3.3 The D Algorithm

The simple sensitized path procedure does not handle reconvergent paths adequately. For example, consider the circuit of Figure 11.5. To find a test for $B$/0 requires that $B$ be set to 1. To propagate the state of $B$ to $D$ requires that $A$ is 1, and to propagate $D$ to $Z$ requires that $E$ is 0. The only way that $E$ can be at 0 is if $B$ and $C$ are both 1, but this is not the case when $B$/0. Apparently, therefore the sensitive path algorithm cannot find a test for $B$/0. In fact, 111/0 is a suitable test, because under fault-free conditions $D$, $E$ and $Z$ are all at logic 0; when $B$/0, all three nodes are at logic 1.

The D-algorithm overcomes that problem by introducing a 5-valued algebra: $\{0, 1, D, \bar{D}, X\}$. $D$ represents a node that is logic 1 under fault-free (normal) conditions and logic 0 under faulty conditions. $\bar{D}$ represents a normal 0, and a faulty 1. $X$ is an unknown value. The values of $D$ and $\bar{D}$ are used to represent the state of a node where there is a fault and also the state of any other nodes affected by the fault.

The D-algorithm works in the same way as the sensitive path algorithm, above. If step 4 fails, the algorithm backtracks. In both steps 2 and 3 it is possible that more than one combination of inputs generates the required node values. If necessary, all possible combinations of inputs are examined.

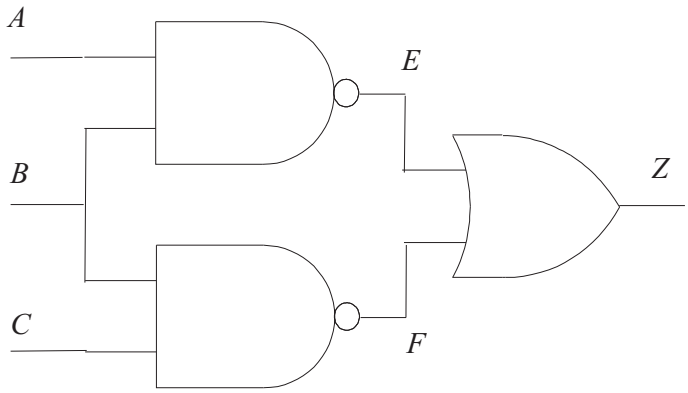



Figure 11.5: Example Circuit for D algorithm

Table 11.1 shows the inputs required to establish a fault at an internal node; to transmit that fault to an output; to generate a fixed value (to establish or propagate a fault) and finally how fault conditions can reconverge. In all cases, the inputs $A$ and $B$ are interchangeable. The table can be extended to gates with three or more inputs. The symbol '–' represents a 'don't care' input.

Table 11.1: Truth tables for the D-Algorithm

| | **AND** | | | **OR** | | | **NAND** | | | **NOR** | | | **NOT** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $A$ | $B$ | $Z$ | $A$ | $B$ | $Z$ | $A$ | $B$ | $Z$ | $A$ | $B$ | $Z$ | $A$ | $Z$ |
| Establish Fault-sensitive | 1 | 1 | $D$ | 0 | 0 | $D$ | 1 | 1 | $D$ | 0 | 0 | $D$ | 1 | $D$ |
| Condition | 0 | – | $\bar{D}$ | 1 | – | $D$ | 0 | – | $D$ | 1 | – | $\bar{D}$ | 0 | $D$ |
| Transmit Fault | $D$ | 1 | $D$ | $D$ | 0 | $D$ | $D$ | 1 | $\bar{D}$ | $D$ | 0 | $D$ | $D$ | $\bar{D}$ |
| | $\bar{D}$ | 1 | $\bar{D}$ | $\bar{D}$ | 0 | $\bar{D}$ | $\bar{D}$ | 1 | $D$ | $\bar{D}$ | 0 | $D$ | $\bar{D}$ | $D$ |
| Generate Fixed Value | 1 | 1 | 1 | 1 | – | 1 | 1 | 1 | 0 | 1 | – | 0 | 1 | 0 |
| | 0 | – | 0 | 0 | 0 | 0 | 0 | – | 1 | 0 | 0 | 1 | 0 | 1 |
| Reconvergence | $D$ | $D$ | $D$ | $D$ | $D$ | $D$ | $D$ | $D$ | $\bar{D}$ | $D$ | $D$ | $\bar{D}$ | | |
| | $\bar{D}$ | $\bar{D}$ | $\bar{D}$ | $\bar{D}$ | $\bar{D}$ | $\bar{D}$ | $\bar{D}$ | $\bar{D}$ | $D$ | $\bar{D}$ | $\bar{D}$ | $D$ | | |
| | $D$ | $\bar{D}$ | 0 | $D$ | $\bar{D}$ | 1 | $D$ | $\bar{D}$ | 1 | $D$ | $\bar{D}$ | 0 | | |

To see how the D-notation can be used, consider the circuit of Figure 11.6. To test for $A$/0, node $A$ is first given a value D, which can be propagated via node $E$ or via node $G$. To propagate the D to node $E$, node $B$ must be 1. Node $H$ then has the value $\bar{D}$. To propagate this $\bar{D}$ to $I$, requires $F$ to be 0 and to propagate the value to $Z$, means $J$ must be 1. If $F$ is 0 and $J$ is 1, $G$ must be 1, therefore nodes $A$ and $D$ must both be 1. At this point we hit an inconsistency as node $A$ has the value D. We have

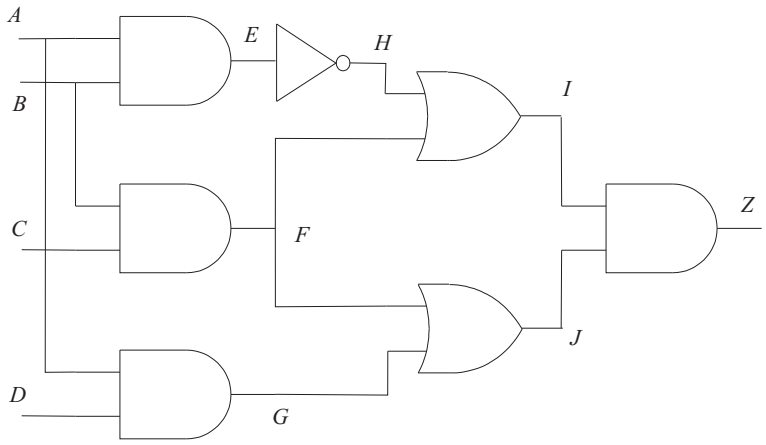



Figure 11.6: Example Circuit for D Algorithm

to return to the last decision made, which in this case was the decision to propagate the value of A through to $E$.

The alternative is to propagate the D at $A$ to $G$.Thus, $D$ must be 1; to propagate the value to $J$, $F$ must be 0 and to propagate to $Z$, $I$ must be 1. Hence, $H$ must be 1, hence $E$ must be 0. As $A$ is already assigned, $B$ must be 0. This is consistent with $F$ being 0 and $C$ may be either 1 or 0.

The D-algorithm, as presented here requires further refinement before it can be implemented as an EDA program. In particular the rules for detecting inconsistencies require more detail. Table 11.2 shows what happens when two fault-free or faulty values are propagated by different routes to the same node.

Table 11.2: Intersection rules for the D-Algorithm

| $\cap$ | **0** | **1** | $\boldsymbol{X}$ | $\boldsymbol{D}$ | $\boldsymbol{\bar{D}}$ |
|---|---|---|---|---|---|
| **0** | 0 | $\phi$ | 0 | $\psi$ | $\psi$ |
| **1** | $\phi$ | 1 | 1 | $\psi$ | $\psi$ |
| $\boldsymbol{X}$ | 0 | 1 | $X$ | $D$ | $\bar{D}$ |
| $\boldsymbol{D}$ | $\psi$ | $\psi$ | $D$ | $\mu$ | $\lambda$ |
| $\boldsymbol{\bar{D}}$ | $\psi$ | $\psi$ | $\bar{D}$ | $\lambda$ | $\mu$ |

$\phi$ inconsistent logic values

$\psi$ inconsistency between logic values and fault values

$\mu$ allowed intersection between fault values

$\lambda$ inconsistent fault values

The D-algorithm is an algorithm in the true sense of the word – if a solution exists, the D-algorithm will find it. The search for a solution can,

however, be very time-consuming. If necessary, every possible combination of node values will be examined. Subsequent test pattern generation algorithms have attempted to speed up the D-algorithm by improving the decision making within the algorithm. Examples include: 9-V which uses a 9-valued algebra and PODEM.

### 11.3.4 PODEM

The PODEM algorithm attempts to limit the decision making, and hence the time needed for a decision. Initially all the inputs are set to X (unknown). Arbitrary values are then assigned to the inputs and the implications of these assignments are propagated forwards. If either of the following propositions is true the assignment is rejected:

1 The node value of the fault under consideration has identical faulty and fault-free values.

2 There is no signal path from a net with a D or $\bar{D}$ value to a primary output.

We will use PODEM on the circuit of Figure 11.6 to develop a test for $H$/1. Initially, all nodes have an X value.

1 Set $A$=0. Fails – proposition 1 ($H$ would be 1).

2 Set $A$=1. OK.

3 Set $B$=0. Fails – proposition 1.

4 Set $B$=1. OK. $E$=D, $H = \bar{D}$.

5 Set $C$=0. OK. $F$=0, $I = \bar{D}$.

6 Set $E$=0. Fails – proposition 2 ($G$=0, $J$=0, $Z$=0).

7 Set $E$=1. OK. $G$=1, $J$=1, $Z = \bar{D}$.

Therefore a test for $H$/1 is 1101/0

### 11.3.5 Fault Collapsing

In the example of Figure 11.3, the test for $A$/0 (the input to a NAND gate) was the same as the test for $E$/1 (the output of that NAND gate). The same test can be used to detect $B$/0. These three faults are *indistinguishable* $\{A/0, B/0, E/0\}$. Similarly, a test for an input of a NAND gate being stuck at 1 will also detect if the output is stuck at 0. Two different tests are needed, however, for $A$/1 and $B$/1. Hence these faults are not indistinguishable, but an input stuck at 1 is said to *dominate* the output stuck at 0 ($A/1 \rightarrow E/0$). The set of rules for fault indistinguishability and dominance for two input ($A$, $B$), single output ($Z$) gates and the inverter are shown in Table 11.3.

Table 11.3: Fault Collapsing Rules

| Type of Gate | Indistinguishable Faults | Fault Dominance |
|---|---|---|
| **AND** | $\{A/0, B/0, Z/0\}$ | $A/1, B/1 \rightarrow Z/1$ |
| **OR** | $\{A/1, B/1, Z/1\}$ | $A/0, B/0 \rightarrow Z/0$ |
| **NAND** | $\{A/0, B/0, Z/1\}$ | $A/1, B/1 \rightarrow Z/0$ |
| **NOR** | $\{A/1, B/1, Z/0\}$ | $A/0, B/0 \rightarrow Z/1$ |
| **NOT** | $\{A/0, Z/1\}$ $\{A/1, Z/0\}$ | |

These rules can be used to reduce a fault list. These rules, however, do not apply to fan-out nodes, which must be omitted from any simplification procedure. If we apply these rules to the 14 faults of the circuit of Figure 11.3 we can see that we have two sets of equivalent faults: $\{A/0, B/0, E/1, F/1, Z/1\}$ and $\{C/0, D/0, F/0\}$ and the following fault dominances: $A/1 \rightarrow E/0$, $B/1 \rightarrow E/0$, $E/0 \rightarrow Z/0$, $F/0 \rightarrow Z/0$, $C/1 \rightarrow F/1$, $D/1 \rightarrow F/1$. As we only need to test for one fault in each equivalent set and for the dominant faults, we only need to derive tests for the following faults: $A/1$, $B/1$, $C/1$, $D/1$ and $C/0$. The fault list is cut from 14 to 5 faults, simplifying the fault generation task. Note that we have not lost any information by doing this – we cannot tell by observing node $Z$ whether a fault in the circuit is one of the five listed or a fault equivalent to or dominated by one of those faults.

## 11.4 Fault simulation

One test pattern can be used to find more than one potential fault. For example, suppose we wish to detect if node $E$ is stuck at 0 in the circuit of Figure 11.7. $E/0$ dominates $G/0$ and is equivalent to $A/0$ and $B/0$. In all these cases, $G$ will be 1 normally and 0 in the presence of one of these

faults. Hence, the input pattern $A$ = 1, $B$ = 1, $C$ = 0, $D$ = 0 can be used to detect four possible faults. As there are 7 nodes in the circuit, there are 14 possible stuck-at faults. This pattern covers 4 faults and it can be shown that of the 16 possible input patterns, 6 are sufficient to detect all the possible stuck-at faults in the circuit.

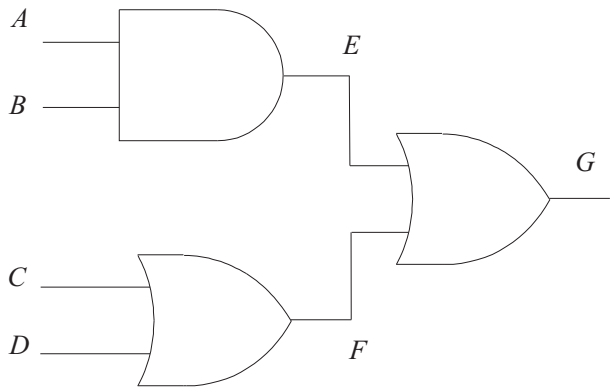



Figure 11.7: Example Circuit for Fault Simulation

It is also generally true that a fault may be covered by more than one pattern. For instance, $E$/1 can be found by attempting to force $E$ to 0. This can be achieved by setting a) $A$ = 1, $B$ = 0, b) $A$ = 0, $B$ = 1 or c) $A$ = 0, $B$ = 0; in all cases, $C$ = 0, $D$ = 0. Thus there are three possible patterns for detecting $E$/1. Note too that pattern a) also covers $B$/1 and $G$/1, b) covers $A$/1 and $G$/1, while c) covers $G$/1. To detect all the faults in the circuit we need to use both $A$ = 1, $B$ = 0, $C$ = 0, $D$ = 0 and $A$ = 0, $B$ = 1, $C$ = 0, $D$ = 0 as these are the only patterns that detect $B$/1 and $A$/1, respectively. We are, however, applying two patterns that can detect $E$/1 and $G$/1. Having found one pattern that detects these two faults, we can *drop* the faults from further consideration. In other words, in applying the second test $A$ = 0, $B$ = 1, $C$ = 0, $D$ = 0, we forget about $E$/1 and $G$/1 as we already have a pattern that detects them. We could equally decide not to drop a fault when a suitable test pattern is found, in order to try to distinguish between apparently equivalent faults.

The object of fault simulation is, therefore, to assess the fault coverage of test patterns, by determining whether the presence of a fault would cause the outputs of the circuit to differ from the fault-free outputs, given a particular input pattern.

The simplest approach to fault simulation is simply to modify the circuit to include each fault, one at a time, and to resimulate the entire circuit. As the single-stuck fault model assumes that only one fault can occur at a time

and that each node of the circuit can be stuck at 1 and at 0, this approach, known as *serial* fault simulation, will require twice as many simulation runs as there are nodes, together with one simulation for the fault-free circuit. This technique is clearly expensive in terms of computer power and time, and three main alternatives have been suggested to make fault simulation more efficient. We will first show how these three approaches can be implemented in a simulator.

### 11.4.1 Parallel fault simulation

If we use two-state logic, one bit is sufficient to represent the state of a node. Therefore one computer word can represent the state of several nodes or the state of one node under several faulty conditions. For instance, a computer with a 32-bit word length can use one word to represent the state of a node in the fault-free circuit together with the state of the node when 31 different faults are simulated. Each bit corresponds to the circuit with one fault present. The same bit is used in each word to represent the same version of the circuit. The fault-free circuit must always be simulated as it is important to know whether a faulty circuit can be distinguished from the fault-free circuit. If more faults are to be simulated than the number of bits in a word, the fault simulation must be completed in several passes, each of which includes the fault-free circuit.

Instead of simulating the circuit by passing Boolean values, words are used, so the state of each gate is evaluated for each fault modelled by a bit of the input signal words. Hence the name *parallel fault simulation.* Because words are passed instead of Boolean values, the event scheduling algorithm treats any change in a word value as an event. Thus gates may be evaluated for certain versions of the circuit even if the input values for that version remain unchanged.

The circuit of Figure 11.7 has 7 nodes, hence 14 possible stuck-at faults (Table 11.4. Thus 15 bits are needed for a parallel fault simulation. The word values of each node for the input pattern $A$ = 1, $B$ = 1, $C$ = 0, $D$ = 0 are shown below. As can be seen, this pattern, as noted earlier, normally sets $G$ to 1, but for faults $A$/0, $B$/0, $E$/0 and $G$/0, the output is 0, and therefore these faults are detected by that pattern.

There are several, obvious, disadvantages to parallel fault simulation. First the number of faults that can be simulated in parallel is limited to the number of bits in a word. If more than two states are used, in other words if a state is encoded using two or more bits, the possible number of parallel faults is further reduced. As has been noted, every version of a gate

Table 11.4: Parallel fault simulation of circuit of Figure 11.7.

| Bit | | *A* | *B* | *C* | *D* | *E* | *F* | *G* |
|---|---|---|---|---|---|---|---|---|
| 0 | – | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | $A$/0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 2 | $A$/1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| 3 | $B$/0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | $B$/1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| 5 | $C$/0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| 6 | $C$/1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 7 | $D$/0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| 8 | $D$/1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 9 | $E$/0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 10 | $E$/1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| 11 | $F$/0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| 12 | $F$/1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| 13 | $G$/0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 14 | $G$/1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |

is scheduled and re-evaluated whenever one of the versions of an input changes. This can be very inefficient as a significant number of null events are likely to be processed. Moreover, if the purpose of the fault simulation is simply to detect whether any of the given test patterns will detect any of the faults, it is desirable to drop a fault from further consideration once it has proved possible to distinguish the behaviour caused by that fault from the normal, fault-free behaviour. Faults cannot be dropped in parallel fault simulation, or perhaps more accurately, the dropping of a fault is unlikely to improve the efficiency of the simulation as the bits corresponding to that fault cannot be used for any other purpose.

### 11.4.2 Concurrent fault simulation

If only the differences between the fault-free simulation and the faulty simulations are maintained, constraints such as word size need not apply. On the other hand, the evaluation of gates would be made more complex because these lists of differences must be manipulated. Concurrent fault simulation maintains fault lists, in the form of those gates that have different inputs and outputs in the faulty circuit from the equivalent gates in the fault-free circuit. The manipulation of fault lists thus consists of evaluating input signals, in exactly the same way as is done for the fault-free circuit,

and checking to see if the output differs from the fault-free circuit.

Figure 11.8 shows the circuit with the fault lists included for the input $A$ = 1, $B$ = 1, $C$ = 0, $D$ = 0. All the stuck faults for all four inputs are listed, together with the stuck faults for the internal nodes, $E$ and $F$ and the output node, $G$. The stuck faults for $E$ and $F$ are only listed once. To distinguish the faulty versions of the circuit from the fault-free version, the gates are labelled according to their output nodes, together with a number. Gate 0 is always the fault-free version. A gate in the fault list is only passed to a gate connected to the output if the faulty value is different from the fault-free value. Thus, $E$3, $E$4, $F$1 and $F$2 appear as inputs to gates in the fault list for $G$, causing faults $G$7, $G$8, $G$9 and $G$10, respectively. As with parallel fault simulation, it can be seen that for this example, $G$1, $G$3, $G$7 and $G$8, representing $E$/0, $G$/0, $A$/0 and $B$/0, respectively, have different outputs from $G$0 and are therefore detected faults.

To see why concurrent fault simulation is more efficient than parallel fault simulation, suppose that $A$ now changes from 1 to 0. This would cause $E$0, $E$2 and $E$4 to be evaluated. $E$1 and $E$3 would not be evaluated because they both model stuck faults on $A$. Now, $E$0 is at 0, as are $E$2, $E$3 and $E$4; $E$1 is at 1. The OR gate, $F$, and its fault list would not be re-evaluated as neither $C$ nor $D$ change. As faults $E$3 and $E$4 are now the same as $E$0, the corresponding faults in $G$: $G$7 and $G$8, are removed from the fault list and a fault corresponding to $E$1, say $G$11, is now inserted. Now gate $G$ is evaluated, as $E$ has changed, and faults $G$2, $G$3, $G$5, $G$6, $G$9, $G$10 and $G$11 are evaluated.

It can be seen from Fig. 10.8 that, even with this small number of gates, the fault list for $G$ has 10 elements. In practice, the fault lists can be significantly simplified with a little pre-processing of the circuit. It has already been noted that one test can cover a number of faults, and it is possible, in many cases, to deduce that some faults are indistinguishable and that tests for certain faults will always cover certain other faults. The circuit of has 7 nodes and 14 stuck faults, but it can be shown that only tests for 5 faults: $A$/0, $C$/0, $D$/0, $A$/1 and $B$/1 are needed and that any other faults are covered by those tests. If this pre-processing is applied, faults $E$4, $F$1, $F$2, $G$1, $G$2, $G$3, $G$4, $G$5 and $G$6 can be eliminated and $G$8, $G$9 and $G$10 are in turn removed, reducing the fault list for $G$ to one element, $G$7.

Concurrent fault simulation allows efficient selective trace and event scheduling to be used, together with the full range of state and delay models. The major disadvantage is that a significant amount of list processing must be done to propagate faults through the circuit.

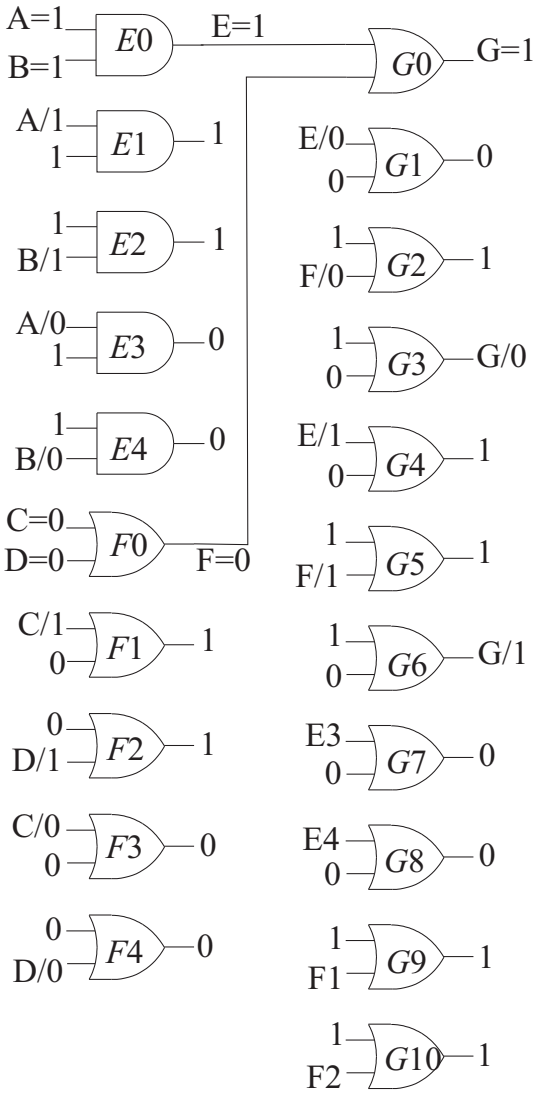



Figure 11.8: Concurrent Fault Simulation of the Circuit of Figure 11.7

## 11.5 Fault Simulation in Verilog

Fault simulators tend to be expensive pieces of software. There are also relatively few products available. Therefore, we will consider here how a Verilog design can be fault simulated using a commercial simulator. Note that we use Verilog syntax here, not that of SystemVerilog.

```
module c17 (w22gat, w23gat,
            w1gat, w2gat, w3gat, w6gat, w7gat);
  output w22gat, w23gat;
  input w1gat, w2gat, w3gat, w6gat, w7gat;
  wire w1gat, w2gat, w3gat, w6gat, w7gat, w10gat,
       w11gat, w16gat, w19gat, w22gat, w23gat;

  nand nand0 (w10gat, w1gat, w3gat);
  nand nand1 (w11gat, w3gat, w6gat);
  nand nand2 (w16gat, w2gat, w11gat);
  nand nand3 (w19gat, w11gat, w7gat);
  nand nand4 (w22gat, w10gat, w16gat);
  nand nand5 (w23gat, w16gat, w19gat);
endmodule

module top;
  wire w22gat, w23gat, w1gat, w2gat, w3gat, w6gat,
       w7gat;
  reg [0:4] pattern_memory[0:5];
  integer index, fdict, flist;

  assign {w1gat, w2gat, w3gat, w6gat, w7gat} =
         pattern_memory[index];

  c17 c17 (w22gat, w23gat, w1gat, w2gat, w3gat,
           w6gat, w7gat);

  initial
    begin
    flist = $fopen("fault_list_c17");
    fdict = $fopen("fault_dictionary_c17");
    $fs_dictionary(fdict);
    // load the test vectors
    $fs_options("one_level_only = no");
    $fs_add(c17);
```

```
    $fs_inject;
    $readmemb("c17.pat",pattern_memory);
    for (index = 0; index < 6; index = index + 1)
      begin
      $fs_strobe(w22gat, w23gat);
      $display("%d %t %b\n",index, $time,
                pattern_memory[index]);
      #10;
      end
    $display(
      "\nlist of undetected and untestable faults\n");
    $fs_list(flist | 1,"undetected","untestable",
      "list_status","list_equiv","list_faultid");
    $display("\n");
    $finish;
    end

endmodule
```

The circuit, c17, is one of a number of benchmark circuits first made public at the 1985 ISCAS conference. In fact, c17 is an example showing the structure of the netlist files (they were not originally written in Verilog). The other files are much larger – the number refers to the number of lines in the original example.

The testbench is in many respects conventional. The circuit is instantiated and a set of inputs is applied through an initial procedure. The input patterns are read from a file, c17.pat, and might have been generated using a test pattern generator incorporating the D algorithm or PODEM. The assign statement is used to apply one set of inputs at a time, from the array pattern_memory. The contents of c17.pat is:

```
01111
11110
00101
10011
01000
00000
```

The procedure calls specific to fault simulaton in the initial procedure all begin with $fs_. The remaining system procedure calls, starting with a $, are all standard. The first two calls are to $fopen. These open the files “fault_list_c17” and “fault_dictionary_c17”. The testbench is going to write to both these files, but, unlike in C, we do not specify a mode.

Then come four calls to fault simulation tasks.

$fs_dictionary(fdict);

opens the multichannel descriptor, fdict, for output from the fault simulation.

$fs_options(”one_level_only = no”);

This system task can be used to set up a number of options for fault simulation. This option (one_level_only) sets faults in the top level of the hierarchy only, by default. So setting this to ”no” allows faults to be modelled down through all levels.

$fs_add(c17);

This adds the circuit (c17) to the list of circuit models to be fault simulated. In this case, we could have omitted the argument.

$fs_inject;

This task adds all the faults in the model to the list of faults to be simulated. Here the arguments have been omitted, but a list of modules could be included, as in $fs_add.

The next system task call is to a standard task $readmemb. This copies the contents of the file, c17.pat, into the array pattern_memory. As the value of the variable, index, is incremented, one row at a time is assigned to the inputs of the circuit, using the concurrent assign statement preceding the initial block. The **for** loop is used to increment index. Within the loop, there is a call to another fault simulation system task and to the $display task. After both tasks have executed, the time is advanced by 10 units. The fault simulation system task is:

$fs_strobe(w22gat, w23gat);

As with the $strobe task, the wires listed are examined to see which faults are observable at these wires. Like the $strobe task, this is done at the end of a time step.

$fs_list(flist | 1,”undetected”,”untestable”,”list_status”,

”list_equiv”,”list_faultid”);

This lists the status of the various faults modelled in the circuit. The multichannel descriptor structure means that the results are printed to the file flist (i.e. fault_list_c17) and to the screen (channel 1). As the parameters suggest, undetected and untestable faults are listed, together with comments about each fault. For this example, all faults are testable and detected. If there were undetected faults, the file would contain records of the form:

fault outterminal sa1 top.c432.nand136.0 ’status=undetected’ ’faultid=605’ ’equiv=660’;

For this fault simulator, there are nearly 30 different fault simulation system tasks in total. Clearly, there is insufficient room to detail all of them

here. The example given provides a basic structure that can be adapted to other examples.

## Summary

The principles of digital testing have been introduced. Defects are characterized as logical faults. Test pattern generation algorithms have been described. Parallel and concurrent fault simulation algorithms have also been discussed. The use of a commercial fault simulator has been demonstrated.

## Further Reading

Abramovici, Breuer and Friedman is a very good introduction to fault modelling, test generation and fault simulation. Also recommended are the books by Wilkins and Miczo. New fault models and algorithms are still being developed, with particular emphasis on delay effects and on sequential systems. IEEE Design and Test of Computers provides a quarterly update on developments.

## Exercises

11.1 Explain the difference the difference between structural and functional testing.

11.2 What assumptions are made by the Single-Stuck Fault Model?

11.3 Write down the stuck-at-fault list for the circuit shown in Figure 11.9. Derive tests for A/1 and A/0 and determine which other faults these tests cover. Show that it is not possible to derive a test for G/0.

11.4 Suggest a test pattern to determine if nodes H and I in Figure 11.9 are bridged together. You should assume that a bridging fault may be modelled as a wired-OR; i.e. that if either wire is at logic 1, the other wire is also pulled to a logic 1.

11.5 A positive-edge-triggered D-type flip-flop is provided with an active-low asynchronous clear input, and has only its Q output available. By considering the functional behaviour of the flip-flop develop a test

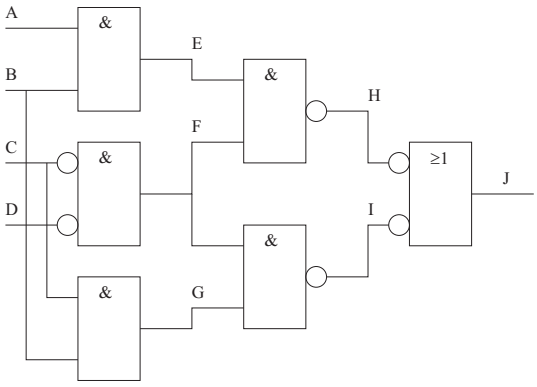



Figure 11.9: Circuit for Exercises 11.3 and 11.4.

sequence for this device for all single-stuck faults on inputs and outputs.

11.6 Describe the four types of crosspoint fault that can occur in a PLA consisting of an AND plane and an OR plane.

11.7 The AND and OR planes of a PLA can be thought of as two NAND planes. What is the minimal set of test patterns required to test an n-input NAND gate?

11.8 Write down a stuck-fault list for the circuit in Figure 11.10. How, in principle, would a test sequence for this circuit be constructed?

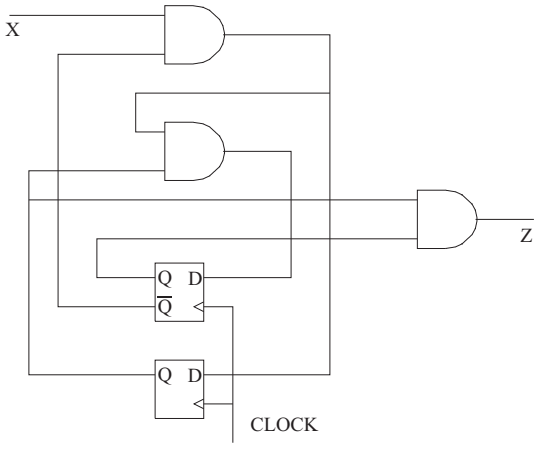



Figure 11.10: Circuit for Exercise11.8.

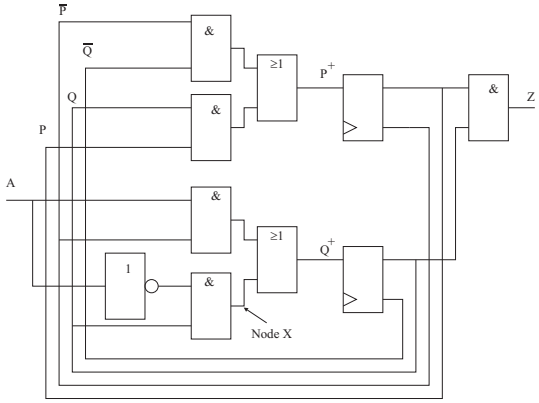



Figure 11.11: Circuit for Exercise11.9.

11.9 The circuit shown in Figure 11.11 is an implementation of a state machine with one input and one output. Derive the next state and output equations and hence show that a parasitic state machine exists, in addition to the intended state machine. Assuming that the initial state of the flip-flops is P=Q=0, suggest a sequence of input values at A that will cause the output, Z, to have the following values on successive clock cycles: 0110. Hence, show that this sequence of input values can be used to test whether node X is stuck at 0.

11.10 Explain the difference between parallel and concurrent fault simulation.

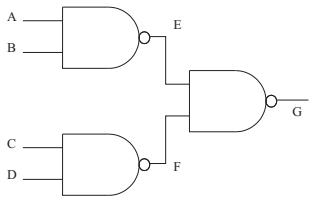



Figure 11.12: Circuit for Exercise11.11.

11.11 In the circuit of Figure 11.12, A=1, B=1, C=1, D=0. Derive the fault lists as they would be included in a concurrent fault simulator, assuming that each of the nodes can be stuck at 1 or stuck at 0. Show that the fault lists may be significantly simplified if redundant and dominated faults are removed in a preprocessing step.

# Chapter 12

# Design for Testability

As noted in the previous chapter, testability for a circuit such as that shown in Figure 12.1 can be expressed in terms of:

- Controllability – the ability to control the logic value of an internal node from a primary input.
- Observability – the ability to observe the logic value of internal node at a primary output.

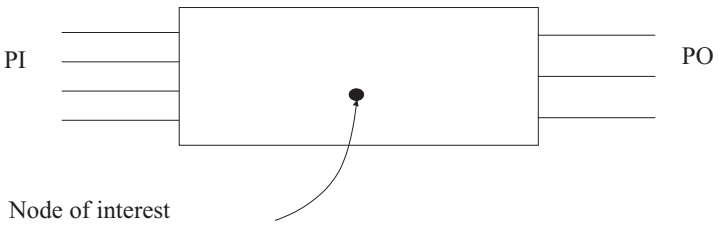



Figure 12.1: Testability of a Node.

The previous chapter discussed methods for finding test patterns for combinational circuits. The testing of sequential circuits is much more difficult because the current state of the circuit as well as its inputs and outputs must be taken into account. Although in many cases it is possible, at least in theory, to derive tests for large complex sequential circuits, in practice it is often easier to modify the design to increase its testability. In other words, extra inputs and outputs are included to increase the controllability and observability of internal nodes.

Testability can be enhanced by *ad hoc* design guidelines or by a structured design methodology. In this chapter we shall discuss general *ad hoc*

principles for increasing testability, then look at a structured design technique – the scan path. In the third section, we will see how some of the test equipment itself can be included on an integrated circuit, to provide self-test capabilities. Finally, the scan path principle can be used for internal testing, but it can also be used to test the interconnect between integrated circuits – boundary scan.

## 12.1 Ad hoc Testability Improvements

If one of the objectives of a design is to enhance the testability of that design, there are a number of styles of design that should be avoided, including:

- Redundant Logic. As seen in the previous chapter redundant combinational logic will result in the presence of potentially undetectable faults. This means that the design is not fully testable and also that time may be spent attempting to generate tests for these undetectable faults.
- Asynchronous sequential systems (and in particular unstructured asynchronous systems) are difficult to synchronize with a tester. The operation of a synchronous system can be halted with the clock. An asynchronous system is, generally, uncontrollable. If asynchronous design is absolutely necessary, confine it to independent blocks.
- Monostables are sometimes used for generating delays. They are extremely difficult to control, and again should be avoided.

On the other hand, there are a number of modifications that could be made to circuits to enhance testability. The single most important of these is the inclusion of some form of initialization. A test sequence for a sequential circuit must start from a known state. Therefore initialization must be provided for all sequential elements, as shown in Figure 12.2. Any defined state will do – not necessarily all zeros. Multiple initial states can be useful.

The cost of enhancing testability includes that of extra I/O pins (including interfaces etc); extra components (MUXs); extra wiring; the degradation of performance because of extra gates in signal paths; and in general, there are more things to go wrong. Against this must be set the benefit that the circuit will be easier to test and hence the manufacturer and consumer can be much more confident that working devices are being sold.

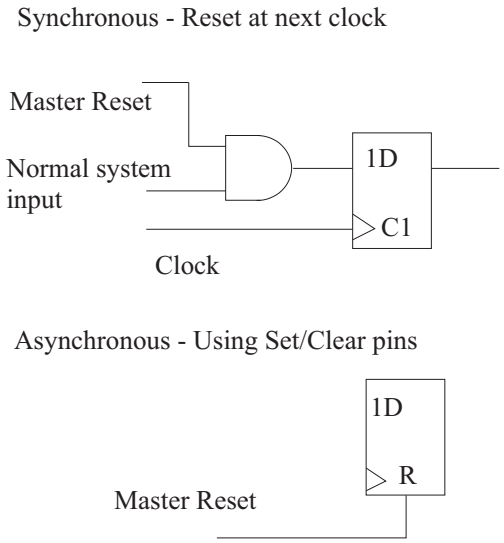



Figure 12.2: Resets add Testability.

## 12.2 Structured Design for Test

The techniques described in the previous section are all enhancements that can be made to a circuit after it has been designed. A structured design for test method should consider the testability problem from the beginning. Let us restate the problem to see how it can be tackled in a structured manner.

Testing combinational circuits is relatively easy, provided there is no redundancy in the circuit. The number of test vectors is (much) less than $2^{(no.of inputs)}$. Testing sequential circuits is difficult because such circuits have states. A test may require a long sequence of inputs to reach a particular state. Some faults may be untestable, because certain states cannot be reached. Synchronous sequential systems, however, can be thought of as combinational logic (next-state and output logic) and sequential logic (registers). Therefore, the key to structured design for test is to separate these two elements.

A synchronous sequential system does not, however, provide direct control of all inputs to the combinational logic; does not allow direct observation of all outputs from the combinational logic; and does not allow direct control or observation of the state variables

The Scan-In, Scan-Out (SISO) principle overcomes these problems by making the state variables directly accessible by connecting all the state registers as a shift register, for test purposes, as shown in Figure 12.3.

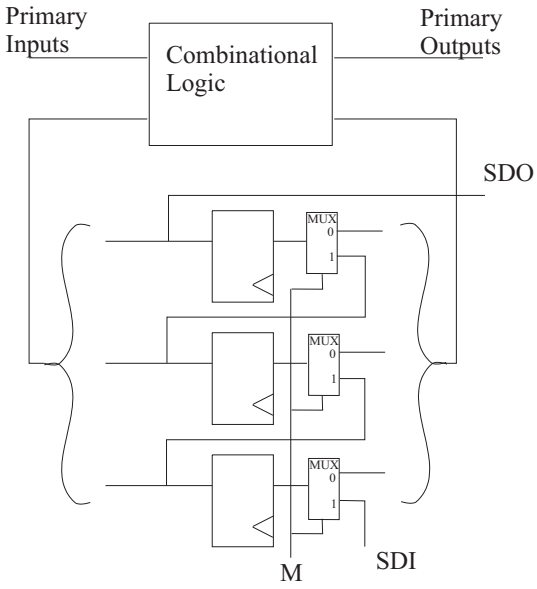



Figure 12.3: SISO Principle.

This shift register has a mode control input, $M$. In normal, operational mode, $M$ is set to 0. In scan mode, $M$ is set to 1 and the flip-flops form a shift register with the input to the shift register being the scan data in (SDI) pin and the output being the scan data out (SDO) pin.

If the combinational logic has no redundancies, a set of test patterns can be generated for it, as if it were isolated from the state registers. The test patterns and the expected responses then have to be sorted because this test data is applied through the primary inputs and through the state registers, using the scan path. Similarly, the outputs of the combinational logic are observed through the primary outputs and using the scan path.

The scan path is used to test a sequential circuit using the following procedure.

1 Set $M$ = 1 and test the flip-flops as a shift register. If a sequence of 1s and 0s is fed into SDI, we would expect the same sequence to emerge from SDO delayed by the number of clock cycles equal to the length of the shift register ($n$). A useful test sequence would be 00110... which tests all transitions and whether the flip-flops are stable.

2 Test the combinational logic

   (a) Set $M$ = 1 to set the state of the flip-flops after $n$ clock cycles by shifting a pattern in through SDI.

(b) Set $M$ = 0. Set up the primary inputs. Collect the values of the primary outputs. Apply 1 clock cycle to load the state outputs into the flip-flops.

(c) Set $M$ = 1 to shift the flip-flop contents to SDO after n-1 clock cycles.

Note that step 2(a) for the next test can be done simultaneously with step 2(c) for the present test. In other words, while the contents of the shift register are being shifted out, new data can be shifted in behind it.

The benefit of using a scan path is that it provides an easy means of making a sequential circuit testable. If there is no redundancy in the combinational logic, the circuit is fully testable. The problem of test pattern generation is reduced to generating tests only for the combinational logic. This can mean that the time to test one device can be greater than would be the case if specific sequential tests had been generated.

The costs of SISO include extra hardware: at least one extra pin for $M$; SDI and SDO can be shared with other system functions by using multiplexers. An extra multiplexer is needed for each flip-flop and extra wiring is needed for the scan path. Hence this can lead to performance degradation as the delay through the next state logic is increased. To minimize the wiring, it makes sense to decide the order of registers in the scan path *after* placement of devices on an ASIC or FPGA has been completed. The order of registers is unimportant provided it is known to the tester.

SISO has now become relatively well accepted as a design methodology. Most VLSI circuits include some form of scan path, although this is not usually documented.

A number of variations to SISO have been proposed including multiple scan paths – put flip-flops in more than one scan path to shorten the length of each path and to shorten the test time – and partial scan paths whereby some flip-flops are excluded from the scan path.

## 12.3 Built-In Self-Test

As with all testing matters, the motivation for Built-In Self-Test (BIST or BIT for Built-In Test) is economic. The inclusion of structures on an integrated circuit or board that not only enhance the testability, but also perform some of the testing simplifies the test equipment and hence reduces the cost of that equipment. BIST can also simplify test pattern generation, because the test vectors are generated internally and allows field testing

to be performed, for perhaps years after manufacture. Overall, therefore, BIST should increase user confidence.

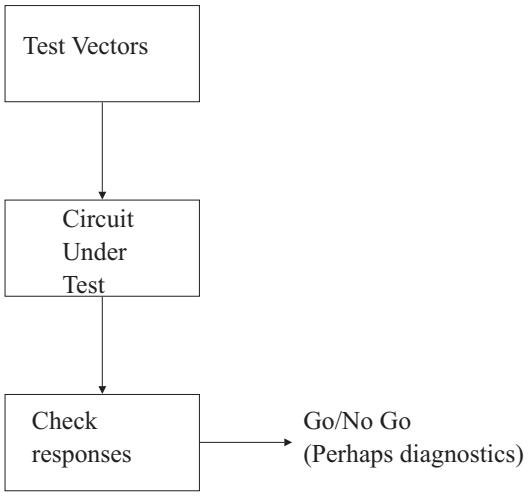



Figure 12.4: BIST Principle.

The principle of BIST is shown in Figure 12.4. The test vector generation and checking are built on the same integrated circuit as the circuit under test. Thus there are two obvious problems: how to generate the test vectors and how to check the responses. It would, in principle, be possible to store pre-generated vectors in ROM. There could, however, be a very large number of vectors. Similarly it would be possible to have a look-up table of responses.

If an exhaustive test were conducted, all possible test vectors could be generated using a binary counter. This could require a substantial amount of extra combinational logic. A simpler solution is to use a Linear Feedback Shift Register (LFSR), introduced in Chapter 6. An LFSR is a pseudo-random number generator that generates all possible states (except the all 0s state) but requires less hardware than a binary counter as shown in Figure 12.5.

A similar structure can be used instead of a look-up table to collect the responses. The Single-Input Signature Register is shown in Figure 12.6. This is an LFSR with a single data input. The register holds the residue from a modulo-2 division. In other words, it compresses the stream of input data to produce a signature that may be compared, after a certain number of cycles, with a known good signature.

Another variant is the Multiple Input Signature Register (MISR), shown in Figure 12.7. Again, this is a modified LFSR, but with more than one

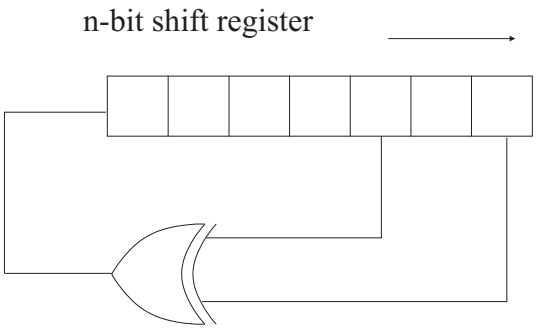



Figure 12.5: LFSR.

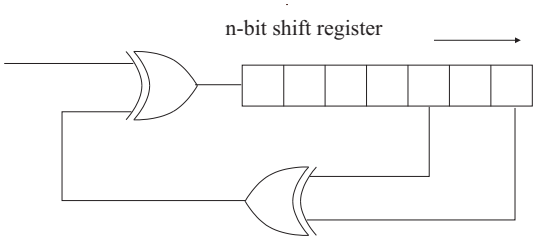



Figure 12.6: SISR.

data input. Thus, a number of output vectors can be gathered and compressed. After a number of clock cycles the signature in the register should be unique. If the circuit contains a fault, the register should contain an incorrect signature, which can easily be checked.

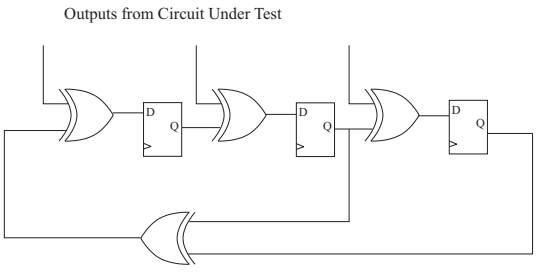



Figure 12.7: MISR.

This approach will obviously fail if the MISR is sensitive to errors. The probability that a faulty circuit generates a correct signature tends to $2^{-n}$ for an n-stage register and long test sequences

### 12.3.1 Example

For example, consider a circuit consisting of three parts: a 3 stage LFSR, a 3 stage MISR and the circuit under test, with the following functions:

$$X = A \oplus B \oplus C$$
$$Y = A.B + A.C + B.C$$
$$Z = \bar{A}.B + \bar{A}.C + B.C$$

The structure of the circuit is shown in Figure 12.8.

In order to see what the correct signature should be, we can perform a simulation. A SystemVerilog model of an LFSR was presented in Chapter 6. This model can easily be adapted to implement an MISR (see the exercises at the end of this chapter). The circuit under test can be described in SystemVerilog by the following model.

```
module cut(output logic x, y, z,
           input logic a_in, b_in, c_in);

logic a, b, c;

always_comb
  begin
  a = a_in;
  b = b_in;
  c = c_in;
  x = a ^ b ^ c;
  y = (a & b) | (a & c) | (b & c);
  z = (!a & b) | (!a & c) | (b & c);
  end

endmodule
```

The input signals a_in, b_in and c_in are not used directly because we will insert fault models into those signals later. The test bench for this circuit can therefore consist of the following code.

```
module bistex;

  logic clock, n_set;
  logic [2:0] signature, q, z;

  lfsr #(3) l0 (.*);
  misr #(3) m0 (.q(signature), .z(z), .clock(clock),
```

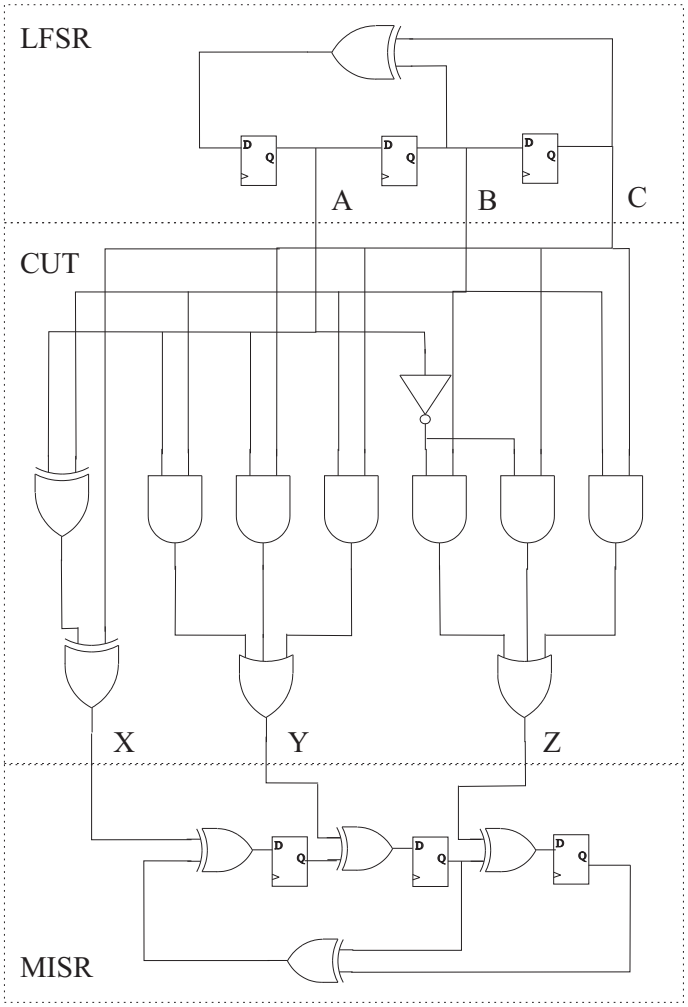



Figure 12.8: Circuit for BIST Example.

```
                .n_set(n_set));
  cut c0 (.a_in(q[2]), .b_in(q[1]), .c_in(q[0]),
          .x(z[2]), .y(z[1]), .z(z[0]));

initial
  begin
        n_set = '1;
  #5ns  n_set = '0;
  #10ns n_set = '1;
  end

always
  begin
  #10ns clock = '0;
  #10ns clock = '1;
  end

endmodule
```

Both the LFSR and MISR are initialised to the 111 state. When the SystemVerilog model is simulated, we get the following sequence of states:

| LFSR Output abc | CUT Output xyz | MISR |
|---|---|---|
| 111 | 111 | 111 |
| 011 | 011 | 100 |
| 001 | 101 | 001 |
| 100 | 100 | 001 |
| 010 | 101 | 000 |
| 101 | 010 | 101 |
| 110 | 010 | 100 |
| 111 | 111 | 000 |

The last output of the MISR, 000, is the signature of the fault-free circuit. The intermediate values of the MISR are irrelevant.

We can emulate a stuck fault at the input by changing one of the assignment statements in the CUT. To model a stuck-at 0, the line,

a = a_in;

is changed to

a = '0;

(We could, of course, perform a full fault simulation, as described in the

previous chapter.) If this perturbed circuit is simulated, the sequence of states is now:

| LFSR Output abc | CUT Output xyz | MISR |
|---|---|---|
| 111 | 011 | 111 |
| 011 | 011 | 000 |
| 001 | 101 | 011 |
| 100 | 000 | 100 |
| 010 | 101 | 010 |
| 101 | 101 | 000 |
| 110 | 101 | 101 |
| 111 | 011 | 011 |

The signature of circuit when a is stuck at 0 is therefore 011. We do not care about the sequence of intermediate states. Hence a comparison of the value in the MISR with 000 when the LFSR is at 111 would provide a pass/fail test of the circuit. In principle, we could simulate every fault in the circuit and note its signature. This information could be used for fault diagnosis. In practice, of course, we would be assuming that every defect manifests itself as a single stuck fault, so this diagnostic information would have to be used with some caution. Moreover, both the LFSR and MISR could themselves contain faults, which in turn would generate incorrect signatures.

If we run the simulation again for a stuck-at 1, the signature 000 is generated. This is an example of *aliasing* – a fault generates the same signature as the fault-free circuit. The probability of aliasing can be shown to tend to $2^{-n}$ if a maximal length sequence is used. As there are only three stages to the MISR, the probability of aliasing is $2^{-3}$ or 1/8. With larger MISRs the probability of aliasing decreases.

In this example, we have made the LFSR and the MISR the same size and used the complete sequence of inputs once. None of these restrictions is essential. We can use LFSRs of different lengths and we do not need to use all the outputs from the LFSR nor all the inputs to the MISR. We can use a shorter sequence than the complete cycle of the LFSR or we can run through the sequence more than once. In all cases, however, the sequence has to be defined when the circuit is built.

### 12.3.2 Built-In Logic Block Observation (BILBO)

The LFSR and MISR, described above, are specialist logic blocks. To include BIST in a circuit using such blocks would require additional registers to those required for normal operation. A scan path reuses the existing registers in a design for testing; in much the same way, Built-in Logic Block Observation (BILBO) registers are used both for normal operation and for BIST. A typical BILBO architecture is shown in Figure 12.9. Three control signals are required, which control the circuit as follows.

| B1 | B2 | B3 | Mode |
|---|---|---|---|
| 1 | 1 | – | Normal |
| 0 | 1 | – | Reset |
| 1 | 0 | 0 | Signature Analysis MISR |
| 1 | 0 | 1 | Test Pattern Generation LFSR |
| 0 | 0 | – | Scan |

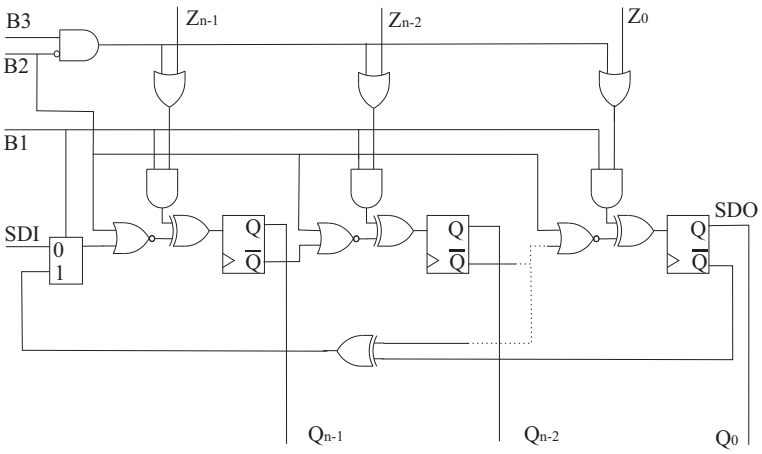



Figure 12.9: BILBO.

To understand the functionality of the circuit, it helps to redraw the functionality of the BILBO when the control signals are set to their different states. Figures 12.10, 12.11 and 12.12 show the normal mode, scan mode and LFSR/MISR modes respectively. Note that in the scan, LFSR and MISR modes, the $\bar{Q}$ output of the flip-flops is used, but inverted before being fed into the next stage. The reset mode synchronously initializes the flip-flops to 0. It was noted in Chapter 6 that an LFSR stays in the all-0s state if it ever enters that state. In LFSR/MISR modes, the BILBO inverts

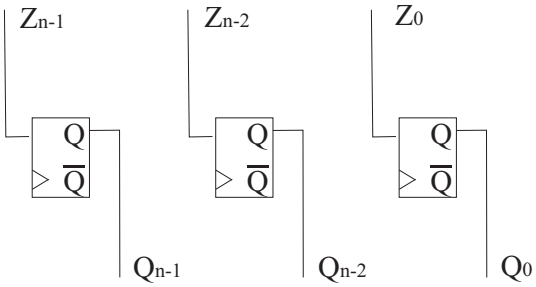



Figure 12.10: BILBO in Normal Mode.

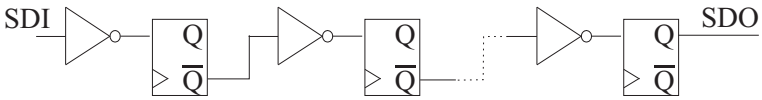



Figure 12.11: BILBO in Scan Mode.

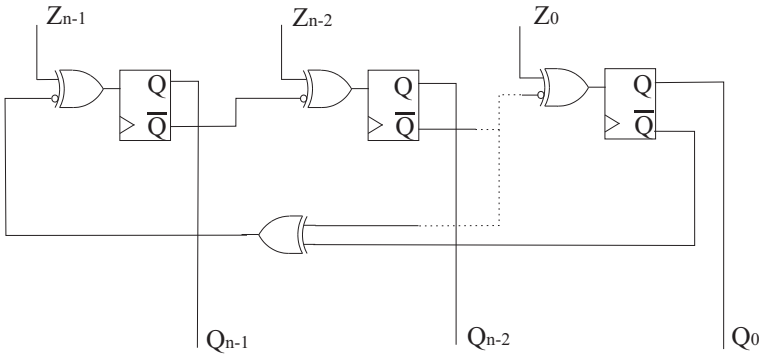



Figure 12.12: BILBO in LFSR/MISR Mode.

the feedback signal, thus making the all 0s state valid, but there still remain $2^n$-1 states in the cycle – one state is excluded from the normal sequence.

Unlike a the flip-flops in a scan-path, the flip-flops in a BILBO-oriented system must be grouped into discrete registers. (The scan mode also allows us to link all the BILBOs in a scan-path – see below.) These registers would ideally replace the normal system registers. An example of a system using BILBOs for self-test is shown in Figure 12.13. R1 and R2 are BILBOs, C1 and C2 are blocks of combinational logic. To test C1, R1 is configured as an LFSR, R2 is configured as an MISR. Similarly, to test C2, R2 is configured as an LFSR, R1 as an MISR.

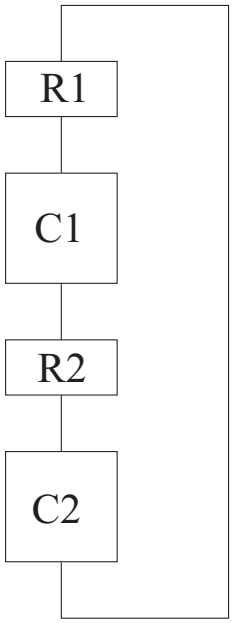



Figure 12.13: Circuit Partitioning for Self-Test.

A different arrangement is shown in Figure 12.14. R1, R2 and R3 are BILBOs; C1, C2 and C3 are combinational logic. To test C1, R2 is an LFSR, R1 is an MISR. To test C2, R1 is an LFSR, R2 is an MISR and so on.

We can therefore use BILBOs to test different structures of combinational logic, but we also need to have some confidence in the correct operation of the BILBOs themselves. Thus, how do we test the BILBOs? The first act in any test must be initialization. This can be done using the synchronous reset. Then the scan-path can be used to test the flip-flops. This implies that some form of controller is needed to generate the BILBO control signals. It is not possible to test that controller (because a further

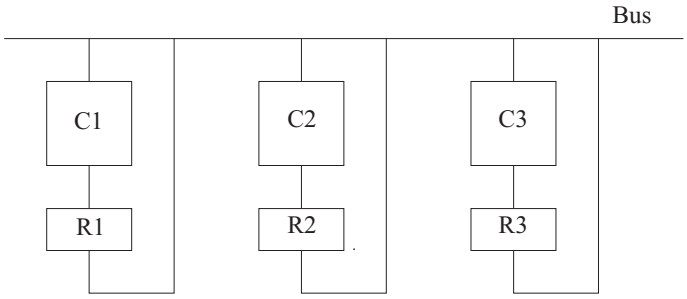



Figure 12.14: Alternate Circuit Partitioning for Self-Test.

controller would be needed, which in turn would need to be tested, *ad infinitum*). Therefore some form of reliable controller is needed to oversee the self-test regime. It makes sense therefore to adopt a "Start Small" strategy, in which part of the system is verified, before being used to test a further part of the system. If the system includes some form of microprocessor, software-based tests can be performed once the microprocessor has been checked.

Before adopting BIST in a design, the cost and effectiveness of the strategy must be considered. There is, of course, the cost of additional hardware – just over 4 gates per flip-flop for a BILBO-based design, together with the cost of a test controller and the additional assorted wiring. This means that there will be an increased manufacturing cost. The extra hardware means that the reliability of the system will be decreased – there's more to go wrong. There is also likely to be some performance degradation as the hardware between flip-flops is increased. The incorporation of BIST means the complexity of the design and hence the time taken to do the design is increased. On the other hand, using BIST means that the costs of test pattern generation disappear and that the equipment needed to test integrated circuits can be simplified. Moreover the tests can be performed every time the circuit is switched on, not merely once at the time of manufacture.

## 12.4 Boundary Scan (IEEE 1149.1)

The techniques described so far in this chapter have been oriented towards integrated circuits, in which controllability and observability may be limited. Circuits built from discrete gates on printed circuit boards (PCBs) are generally considered easier to test because it is possible to gain access to all the nodes of the circuit using a probe, as shown in Figure 12.15,

or a number of probes arranged as a “bed-of-nails”. This assumption has become invalid in recent years for thr following reasons:

- It is not possible to test mounted ICs (the pins may be connected together);
- PCBs now often have more than 20 layers of metal, so deep layers cannot be reached;
- The density of components on a PCB is increasing. Multi-chip modules (MCMs) take the chip/board concept further and have unpackaged integrated circuits mounted directly on a silicon substrate.

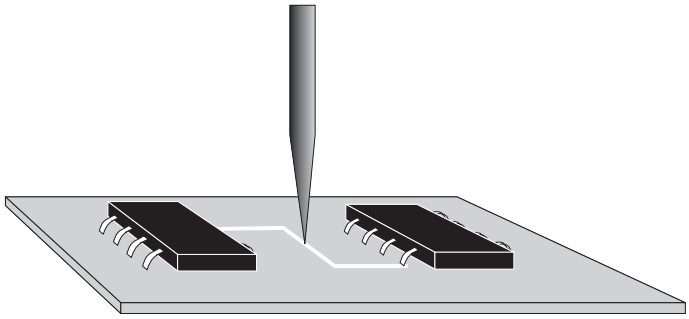


Figure 12.15: Probe Testing.

Boundary scan is a technique for testing the interconnect on PCBs and for testing ICs mounted on PCBs. As before both the ICs and the empty PCB can be tested, but boundary scan replaces the step of testing the loaded PCB with a “Bed-of-nails” tester. The bed-of-nails approach has also been criticized because of “backdriving” – in order to test a single gate its inputs would be forced to particular logic values, which also forces those logic values onto the outputs of other gates. This is not how gates are designed to work and may cause them damage.

The principle of boundary scan is to allow the *outputs* of each IC to be controlled and *inputs* to be observed. For example consider the faults shown in Figure 12.16. These faults are external to the integrated circuits and have arisen as a result of assembling (fault-free) ICs onto a PCB. Instead of using mechanical probes to access the board, the faults are sensitized electrically. The outputs of the left-hand ICs in Figure 12.16 are used to establish test patterns and the inputs of the right-hand IC are used to observe the responses. Therefore we need to control and observe the output and input pins, respectively of the integrated circuits. This can be

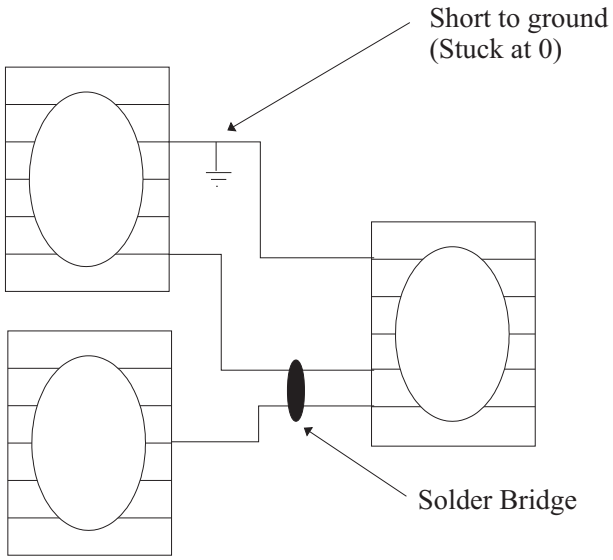



Figure 12.16: Circuit Board Faults.

done by connecting those pins, on the *boundary* of the integrated circuits, into a *scan* path, using special logic blocks at each input and output.

Figure 12.17 shows how the input and output pins of all the integrated circuits on a board are connected together in a scan path. Each IC has dedicated pins to allow the scan path to pass through it. These pins are labelled as TDI (Test Data In) and TDO (Test Data Out). In addition, control pins will be needed. The various ICs on a board may come from different manufacturers. For boundary scan to work, the ICs need to use the same protocols. Therefore an IEEE standard, 1149.1, has been defined. This standard arose from the work of the Joint Test Action Group (JTAG). The term JTAG is therefore often used in reference to the Boundary Scan architecture.

Every boundary scan compliant component has a common test architecture, shown in Figure 12.18. The elements of this architecture are as follows.

1 Test Access Port (TAP)

The TAP consists of 4 or 5 additional pins for testing. The pins are:

- TDI & TDO (Test Data In and Out). Both data and instructions are sent to ICs through the scan path. There is no way to distinguish data from instructions, or indeed to determine which particular IC a sequence of bits is intended to reach. Therefore the following pin is used to control where the data flows.

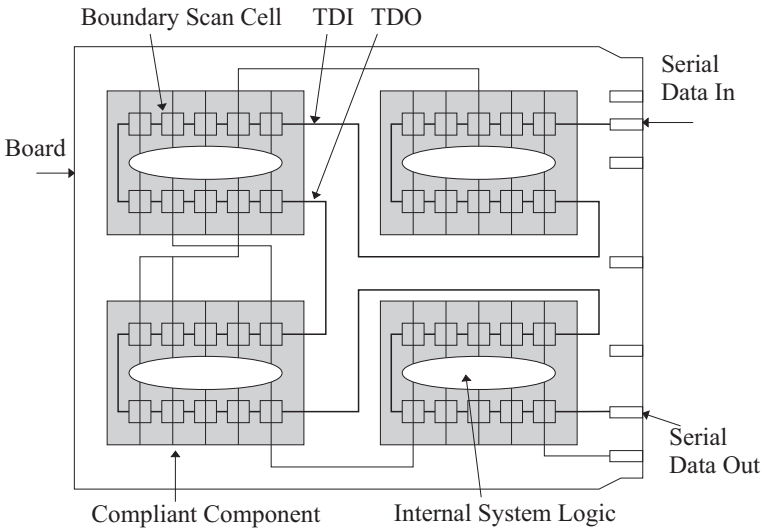



Figure 12.17: Board with Boundary Scan.

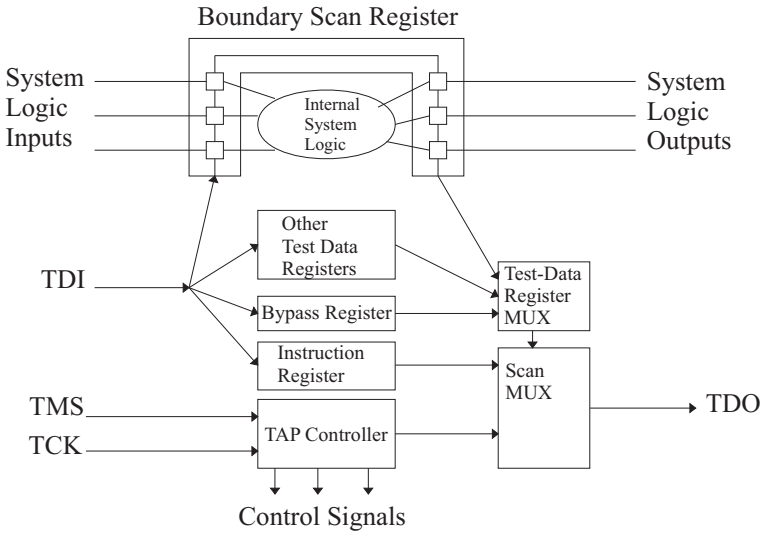



Figure 12.18: Boundary Scan Architecture.

- TMS (Test Mode Select). Together with the TCK pin, the TMS pin is used to control a state machine that determines the destination of each bit arriving through TDI.
- TCK (Test Clock)
- TRST (Test Reset) which is an optional asynchronous reset (not shown in Figure 12.18).

2 TAP Controller

This is a 16 state machine that controls the test. The inputs to the state machine are TCK and TMS. The outputs are control signals for other registers. The state chart of the TAP controller is shown in Figure 12.19. Notice that a sequence of five 1s on TMS in successive clock cycles will put the state machine into the Test-Logic-Reset state from any other state. The control signals derived from the TAP controller are used to enable other registers in a device. Thus a sequence of bits arriving at TDI can be sent to the instruction register or to a specific data register, as appropriate.

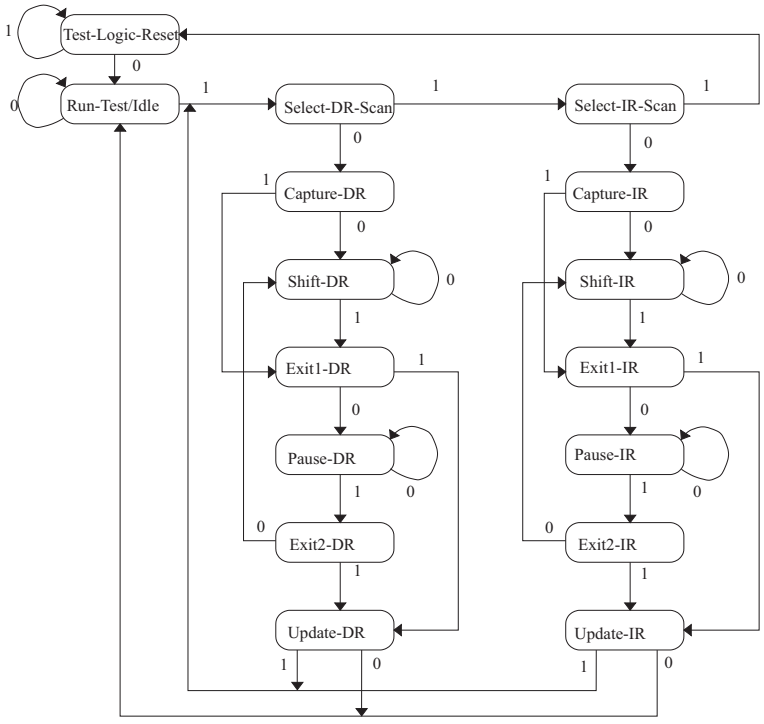



Figure 12.19: TAP Controller State Diagram.

3 Test Data Registers

A Boundary Scan compliant component must have all its inputs and outputs connected into a scan path. Special cells, described below, are used to implement the scan register. In addition, there must be a Bypass Register of 1 bit. This allows the scan path to be shortened by avoiding the boundary scan register of a component. Other registers may also be included, for example an IC might include an identification register, the contents of which could be scanned out to ensure that the correct device had been included on a PCB. Similarly the internal scan path of a device could be made accessible through the boundary scan interface. Some programmable logic manufacturers allow the boundary scan interface to be used for programming devices. Thus the configuration register is another possible data register.

4 Instruction Register

This register has at least 2 bits, depending on number of tests implemented. It defines the use of Test Data Registers. Further control signals are derived from the instruction register.

The core logic is the normal combinational and sequential logic of the device. This core logic may (should) contain a scan path and may also contain BIST structures.

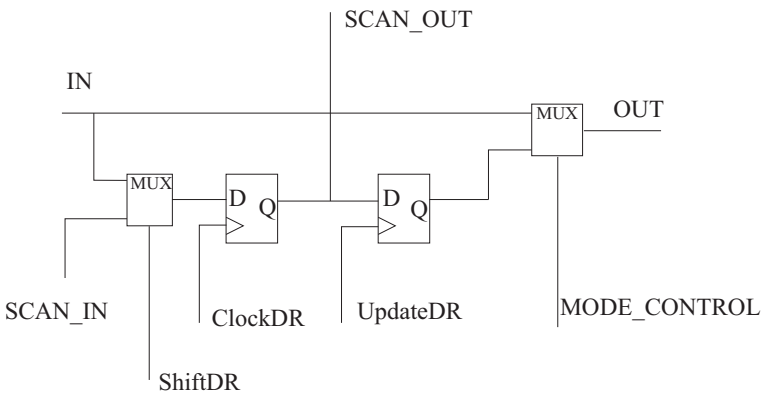



Figure 12.20: Boundary Scan Cell.

A typical Boundary Scan Cell is shown in Figure 12.20. This cell can be used for an input or an output pin. For an input pin, IN is connected to the pin, OUT is connected to the device core; for an output pin, IN comes from the core, OUT goes to the pin. Other designs of boundary scan cell are possible.

The boundary scan cell has four modes of operation.

1 Normal mode. Normal system data flows from IN to OUT.

2 Scan mode. ShiftDR selects the SCAN_IN input, ClockDR clocks the scan path. ShiftDR is derived from the similarly named state in the TAP controller of Figure 12.19. ClockDR is asserted when the TAP controller is in state Capture-DR or Shift-DR. (Hence, of course, the Boundary Scan architecture is not truly synchronous!)

3 Capture mode. ShiftDR selects the IN input, data is clocked into the scan path register with ClockDR to take a snapshot of the system.

4 Update mode. After a capture or scan, data from the left flip-flop is sent to OUT by applying 1 clock edge to UpdateDR. Again, this clock signal comes from the TAP controller when it is in state Update-DR. The TAP controller then enters the Run Test state and MODE_CONTROL is set as appropriate according to the instruction held in the instruction register (see below).

For normal input and output pins, the Boundary Scan cells are the only logic between the core and the IC pins. The only cases where logic is permitted between the boundary scan cell and an external pin are shown in Figure 12.21.

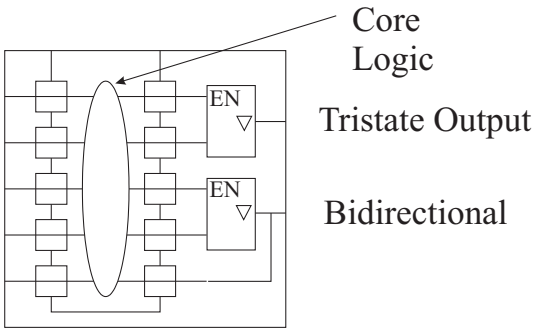



Figure 12.21: Logic Outside Boundary Scan Path.

A number of instructions may be loaded into the instruction register. These allow specific tests to be performed. During test execution, the TAP Controller is in the Run Test state. Three of these tests are mandatory; the remaining tests are optional. Some of these tests are:

- EXTEST (Mandatory). This instruction performs a test of the system, external to the core logic of particular devices. Data is sent from the output boundary scan cells of one device, through the pads and pins

of that device, along the interconnect wiring, through the pins and pads of a second device and into the input boundary scan cells of that second device. Hence a complete test of the interconnect from one IC core to another is performed.

- SAMPLE/PRELOAD (Mandatory).This instruction is executed before and after the EXTEST and INTEST instructions to set up pin outputs and to capture pin inputs.
- BYPASS (Mandatory). This instruction selects the Bypass register, to shorten the scan path.
- RUNBIST (Optional). Runs a built-in self-test on a component.
- INTEST (Optional). This instruction uses the boundary scan register to test the internal circuitry of an IC. Although such a test would normally be performed before a component is mounted on a PCB, it might be desirable to check that the process of soldering the component onto the board has not damaged it. Note that the internal logic is disconnected from the pins, so if pins have been connected together on the board, that will have no effect on the standard test.
- IDCODE, USERCODE (Optional). These instructions return the identification of the device (and the user identification for a programmable logic device). The code is put into the scan path.
- CONFIGURE (Optional). An SRAM-based FPGA needs to be configured each time power is applied. The configuration of the FPGA is held in registers. These registers can be linked to the TAP interface. This clearly saves pins as the configuration and test interfaces are shared.

The MODE_CONTROL signal of Figure 12.20 is set to select the flip-flop output when instructions EXTEST, INTEST and RUNBIST are loaded in the instruction register. Otherwise the IN input is selected.

Testing a board with boundary scan components is in many ways similar to testing a component with a scan path. First, the Boundary Scan circuitry itself must be tested for faults such as a broken scan path or a TAP failure. Then, interconnect and other tests can be performed. The Boundary Scan path allows nodes to be controlled from one point in the scan path and observed at another point. Test patterns for the interconnect (and for non-Boundary Scan compliant components) have to be derived in much the same way that tests for logic are determined. These tests and

the appropriate instructions have to be loaded into the registers of Boundary Scan components in the correct order. This process is clearly complex to set up and really has to be automated.

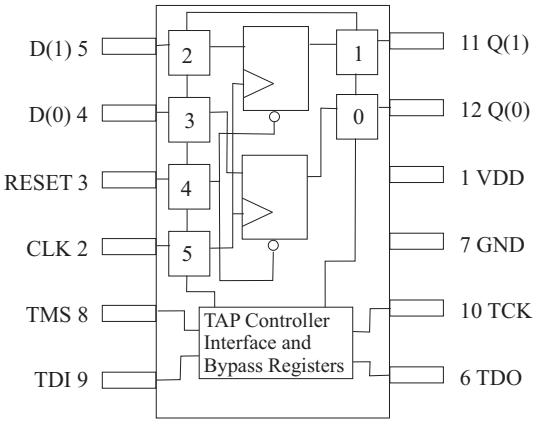



Figure 12.22: IC with Boundary Scan.

An example of how Boundary Scan might be included on an IC is shown in Figure 12.22. The basic circuit has two D type flip-flops with a clock and reset. The D, Q, clock and reset pins have boundary scan cells included as shown. A TAP controller and instruction and bypass registers are included, together with the four extra pins.

The costs of implementing Boundary Scan on an integrated circuit include the cost of a Boundary Scan Cell for each pin; the TAP Controller; the 1 bit bypass register; the instruction register and four extra pins. There will be extra wiring on PCB.

On the other hand there can be significant benefits. The fault coverage of a PCB can be close to 100%. Boundary scan is easy to implement on a PCB requiring 4 pins on an edge connector. Specialist, expensive test equipment, such as a bed-of-nails tester is not needed. Indeed, it is possible to implement a boundary scan tester using little more than a standard Personal Computer or Workstation. Tests can be performed on ICs after they have been mounted on the PCB, so field testing is easy. Because the test circuitry is independent of normal system functions it is possible to monitor the inputs and outputs of ICs in normal operation, thus providing debugging functions.

There is an increasing number of ICs with Boundary Scan compliance, e.g. Intel Pentium, Motorola 68040, Xilinx programmable logic.

## Summary

The testability of a circuit can be improved by modifying the circuit design. The simplest modifications include providing asynchronous resets to every register and avoiding redundant and other uncontrollable logic. SISO separates the sequential from the combinational logic, reducing test generation to a purely combinational circuit problem. Built-in self-test can reduce manufacturing costs by putting much of the test circuitry on the chip. Boundary scan uses the SISO principle to allow complex PCBs to be tested. These various techniques can be combined.

## Further Reading

The books by Abramovici, Breuer and Friedman, Miczo and Wilkins all describe design for test methods. Boundary scan is now incorporated into many FPGAs and the TAP interface is used to configure the internal logic. Details are on the manufacturers' websites.

## Exercises

12.1 Explain what is meant by initialization. Why is it necessary to initialize a circuit for test purposes even if it is not necessary in its system function?

12.2 What are the problems that the scan-in scan-out (SISO) method is intended to overcome? Explain the principles of the SISO method, and identify the benefits and costs involved.

12.3 An certain integrated circuit contains 50 D-type flip-flops. Assuming that all states are reachable, and that it may be clocked at 1MHz, what is the minimum time needed for an exhaustive test? If the same integrated circuit is designed with a full scan-path and if all the combinational logic may be fully tested with 200 test vectors, estimate the time now required to complete a full test.

12.4 Show that the circuit of Figure 12.23 is a suitable test generator for an n-input NAND gate. Hence, suggest a suitable BIST structure for each of the NAND planes in a PLA.

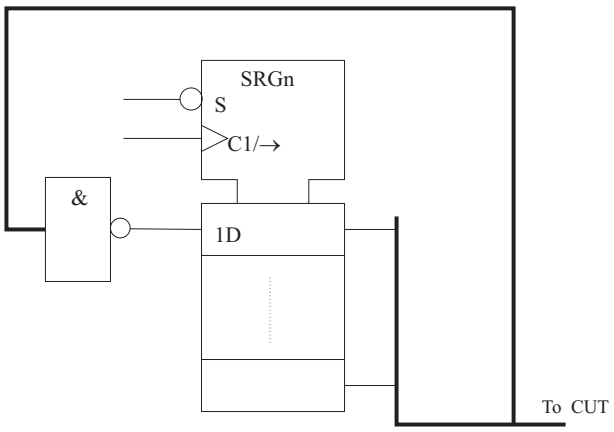



Figure 12.23: Circuit for Exercise 12.4.

12.5 Figure 12.24 shows the structure of a simple CPU (reproduced from Chapter 7). There is a single bus, 8 bits wide. “PC”, “IR”, “ACC”, “MDR” and “MAR” are 8 bit registers. “Sequencer” is a state machine with inputs from the “IR” block and from other points in the system and with outputs that control the operation of the “ALU” and that determine which register drives the bus.

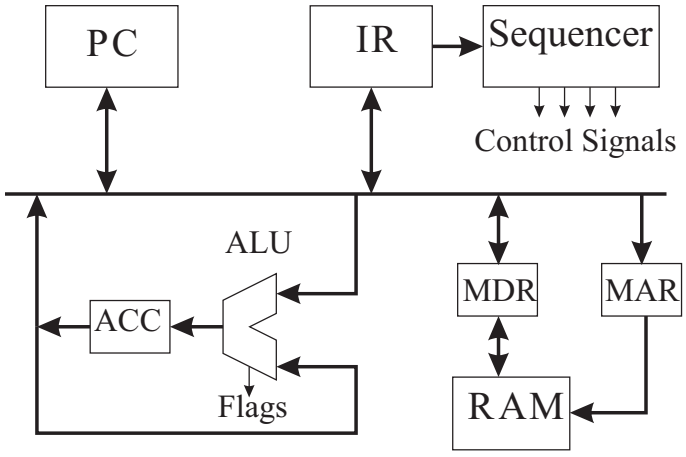



Figure 12.24: CPU Datapath for Exercise 12.5.

The CPU design is to be modified to include a self-test facility. This self-test will not require the use of any external signals or data other than the clock and will generate a simple pass/fail indication. The self-test should require as little additional hardware as possible.

(a) Describe the modifications you would make to the hardware to

allow a self-test to be performed.

(b) Describe the strategy to be used to test the system, excluding the “Sequencer”. Does testing the “ALU” present any particular difficulties?

12.6 What are the main hardware components of the IEEE 1149.1 Boundary Scan test architecture?

12.7 Figure 12.19 shows the state transition diagram of the Boundary Scan TAP controller. Assuming that the instruction for an EXTEST is 10 for a particular IC, what sequence of inputs needs to be applied to the TAP of that IC to load the pattern 1010 into the first four stages of the Boundary Scan register of the IC and to run an EXTEST? (Note that the least significant bits should be loaded first.)

12.8 If the outputs of four boundary scan register stages are connected to the inputs of four similar register stages in a second IC, show, in principle, how the test sequence from exercise 7 can be extended to capture the responses of the interconnect. What assumptions have you made about the connection of the test structures on the two ICs?

12.9 A particular integrated circuit has 2000 flip-flops and 5000 other gates. The package has 52 pins, including power, ground, clock and reset. All the buses are 16 bits wide. A new version of the circuit is to be built. Before redesigning the circuit, the manufacturer would like an estimate of the costs of including:

(a) One or more scan-paths to cover all of the flip-flops

(b) Boundary scan to IEEE 1149.1 standard

(c) Built-in self-test.

The estimates should be in terms of extra components and pins and should consider each of the three features individually, together with any savings that may be made by including 2 or more features.

12.10 Write a synthesizable SysetmVerilog model of the IEEE 119.1 TAP controller. The following outputs should be asserted:

12.11 Modify the VHDL model of the LFSR from chapter 6 to implement an n-stage MISR. Hence, write a model of an n-bit BILBO register.

| Signal | State(s) |
|---|---|
| UpdateDR | Update-DR |
| ClockDR | Capture-DR |
| | Shift-DR |
| ShiftDR | Shift-DR |
| UpdateIR | Update-IR |
| ClockIR | Capture-IR |
| | Shift-IR |
| ShiftIR | Shift-IR |

# Chapter 13

# Asynchronous Sequential Design

The sequential circuits described in Chapters 5, 6 and 7 are synchronous. A clock is used to ensure that all operations occur at the same instant. This avoids the problems of hazards, because such transient effects can be assumed to have died away before the next clock edge. Therefore irredundant logic can be used, which then makes the combinational parts of the circuits fully testable, at least in theory. The flip-flops used in synchronous design are, however, asynchronous internally. In this chapter, we will consider the design of asynchronous elements, and use a SystemVerilog simulator to illustrate the difficulties of asynchronous design.

## 13.1 Asynchronous Circuits

Throughout this book, the emphasis has been on the design of *synchronous* sequential circuits. State information or other data has been loaded into flip-flops at a clock edge. Asynchronous inputs to flip-flops have been used, but *only* for initialization. A common mistake in digital design is to use these asynchronous inputs for purposes other than initialization. This mistake is made either because of inexperience or because of a desire to simplify the logic in some way. Almost inevitably, however, circuits designed in such a manner will cause problems, by malfunctioning or because subsequent modification or transfer to a new technology will cause the assumptions made in the design to become invalid.

Synchronous sequential design is almost overwhelmingly preferred and practised because it is easier to get right than asynchronous design. Simply connecting logic to the asynchronous inputs of flip-flops is almost al-

ways wrong. Structured design techniques exist for asynchronous design and this chapter will describe the design process and its pitfalls. It should be noted, however, that we are primarily concerned with the design of circuits comprising a few gates. It is possible to design entirely asynchronous systems, but such methodologies are still the subject of research. Nevertheless, as clock speeds increase, some of the complex timing issues described here will become relevant. It is increasingly difficult to ensure that a clock edge arrives at every flip-flop in a system at *exactly* the same instance. Systems may consist of synchronous islands that communicate asynchronously. To ensure such communications are as reliable as possible, specialized interface circuits will need to be designed, using the techniques described in this chapter.

Although, as noted above, this book has been concerned with synchronous systems, reference was made to the synthesis of asynchronous elements in Chapter 9. At present, synthesis tools are intended for the design of synchronous systems, normally with a single clock. This is particularly true of synthesis tools intended for FPGA design. The SystemVerilog construct

```
assign q = c ? d : q;
```

would be synthesized to an asynchronous sequential circuit structure. Similarly, the sequential block

```
always_latch
  if (c)
    q <= d;
```

would also be synthesized to an asynchronous latch. In both cases, q explicitly holds onto its value unless c is asserted. It might be thought that the circuit structures created by a synthesis tool for the two cases would be identical. In general, this is not so. The first case is exactly the same as writing

```
assign q = (d & c) | (q & !c);
```

Hence, a synthesis tool would create an inverter, two AND gates and an OR gate (or an inverter and three NAND gates). On the other hand, a compliant synthesis tool would infer the existence of a latch from the incomplete **always_latch** statement of the second case, and use a latch from a library (while also issuing a warning message, in case the incomplete if statement were a coding error). The latch created by Boolean minimization and the library latch are not the same. Indeed, the Verilog RTL synthesis standard, IEEE 1364.1 explicitly forbids the use of concurrent

assignments of the form shown, while permitting the use of incomplete **if** and **case** statements.

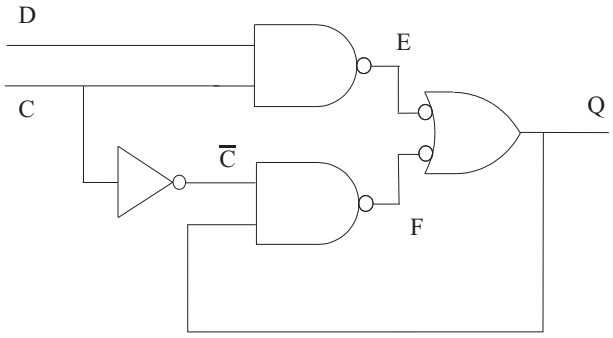



Figure 13.1: Basic D latch.

To see why, assume that the circuit has been implemented directly, as shown in Figure 13.1. This circuit should be compared with that of Figure 2.13. Indeed, the following analysis is comparable with that of section 2.4. Let us assume that each gate, including the inverter, has a delay of 1 unit of time, e.g. 1 ns. Initially, $Q$, $D$ and $C$ are at logic 1. $C$ then changes to 0. From the analysis of section 2.4, we know that this circuit contains a potential hazard. When we draw a timing diagram for this circuit, as shown in Figure 13.2, this hazard appears at $Q$. This hazard is propagated back to $F$, which causes $Q$ to change *ad infinitum*. Hence the circuit oscillates. The causality between $F$ and $Q$ is not shown in Figure 13.2, for clarity. This kind of behaviour is obviously extremely undesirable in a sequential circuit. Although the assumption of a unit delay in each gate may be unrealistic, it can easily be demonstrated, by means of a SystemVerilog simulation, that a hazard, and hence, oscillatory behaviour will occur, irrespective of the exact delays in each gate.

We should, at this point, include a very clear warning. Although we will use SystemVerilog in this chapter to model and to simulate the behaviour of asynchronous circuits, these simulations are intended to demonstrate that problems *may* exist. It is extremely difficult to accurately predict, by simulation *exactly* how a circuit will behave, particularly when illegal combinations of inputs are applied. The spurious effects result from voltage and current changes within electronic devices, not transitions between logic values.

The solution to the problem of oscillatory behaviour is, as stated in section 2.4, to include redundant logic by way of an additional gate. Thus,

$$Q^+ = D.C + Q.\bar{C} + D.Q$$

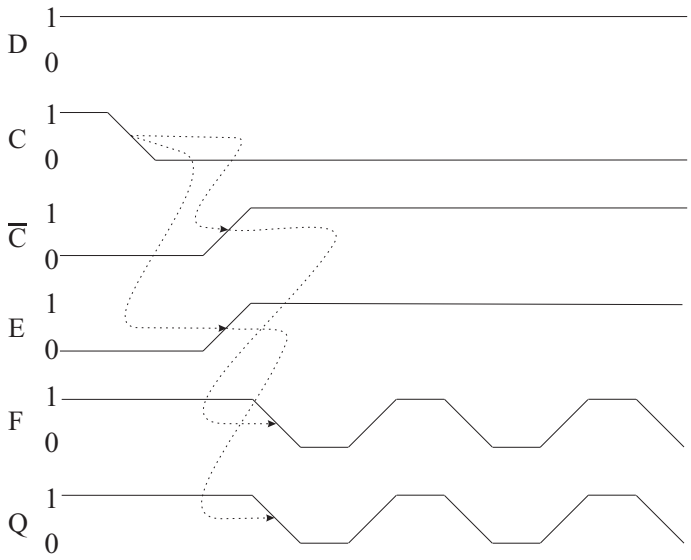



Figure 13.2: Timing diagram for circuit of Figure 13.1.

or

$$Q^+ = \overline{\overline{D.C}.\overline{Q.\bar{C}}.\overline{D.Q}}$$

where $Q^+$ represents the "next" value of Q. The redundant gate, $\overline{D.Q}$, has a 0 output while D is 1, therefore Q is held at 1.

The expression for $Q^+$ can be rearranged:

$$Q^+ = D.C + Q.\left(\bar{C} + D\right)$$

Hence the circuit of Figure 13.3 can be constructed. This would not and could not be generated by optimizing logic equations, but instead would exist in a library. It is this circuit that would be called from the library by a synthesis tool when an incomplete **if** statement was encountered.

## 13.2 Analysis of Asynchronous Circuits

### 13.2.1 Informal Analysis

The operation of the D latch of Figure 13.3 is relatively straightforward. The key is the operation of the cross-coupled NAND gates. Two NAND (or NOR) gates connected in this way form an RS latch with the truth table given below. (An RS latch built from NOR gates has a similar truth table, but with the polarities of $R$ and $S$ reversed.)

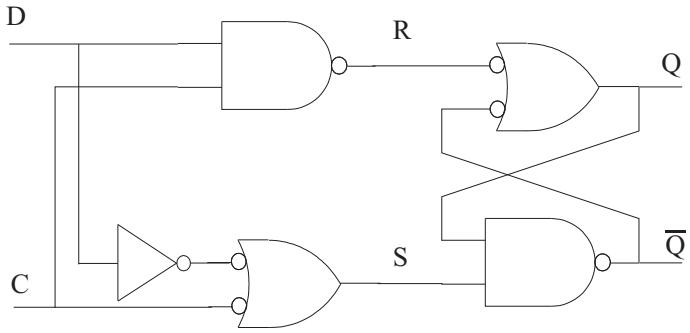



Figure 13.3: D latch with hazard removed.

| $R$ | $S$ | $Q^+$ | $\bar{Q}^+$ |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | $Q$ | $\bar{Q}$ |

The input $R = S = 0$ is normally considered illegal, because it forces the outputs to be the same, contradicting the expected behaviour of a latch.

The D latch of Figure 13.3 contains an RS latch, in which $R$ and $S$ are controlled by two further NAND gates. When $C$ is at logic 0, $R$ and $S$ are at 1, therefore the latch holds whatever value was previously written to it. When $C$ is 1, $S$ takes the value of $D$ and $R$ takes the value $\bar{D}$. From the truth table, above, we can see that $Q$ therefore takes the value of $D$. We can further note that the signal paths from $D$ to the outputs are unequal, because of the inverter. It is therefore reasonable to assume that if $D$ and $C$ were to change at the same time, the behaviour of the latch would be unpredictable.

Figure 13.4 shows the circuit of a positive edge-triggered D flip-flop. We will attempt to analyze this circuit informally, but this analysis is intended to show that a formal method is needed. Let us first deal with the "asynchronous" set and reset[1]. If $S$ is 0 and $R$ is 1, $Q$ is forced to 1 and $\bar{Q}$ is forced to 0, according to the truth table above. Similarly, if $S$ is 1 and $R$ is 0, $Q$ is forced to 0 and $\bar{Q}$ is forced to 1. Under normal synchronous operation, $S$ and $R$ are both held at 1, and therefore can be ignored in the following analysis. Note however that if both $S$ and $R$ are held at 0, both $Q$ and $\bar{Q}$ go to 1, hence this condition is usually deemed to be illegal.

[1]At this level, all the inputs are asynchronous, of course. Synchronous design works because we follow certain conventions about the use of inputs, not because particular inputs are special.

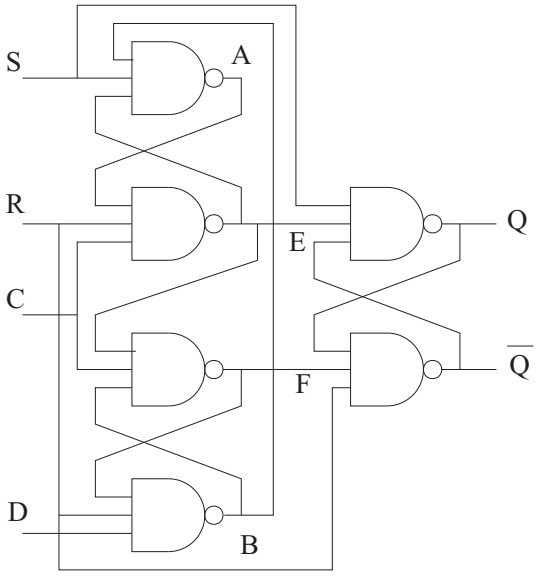



Figure 13.4: Positive edge-triggered D flip-flop.

Let us consider the effects of changes at the $D$ and $C$ inputs, while $R = S = 1$. If $C$ is at 0, then both $E$ and $F$ are at 1 and therefore $Q$ and $\bar{Q}$ are held. If $D$ is at 0, internal nodes $A$ and $B$ are at 0 and 1, respectively. If $D$ is at 1, $A$ is 1 and $B$ is 0. Therefore $D$ can change while the clock is low, causing $A$ and $B$ to change, but further changes, to $E$ and $F$, are blocked by the clock being low.

When the clock changes from 0 to 1, *either* $D$ is 0, and hence $A$ is 0 and $B$ is 1, which force $E$ to 1 and $F$ to 0 and therefore, $Q$ to 0 and $\bar{Q}$ to 1, *or* $D$ is 1, $A$ is 1, $B$ is 0 and therefore $E$ is 0, $F$ is 1, $Q$ is 1 and $\bar{Q}$ is 0. Therefore when the clock changes, it is assumed that $A$ and $B$ are stable. Hence, there is a *setup time* in which any change in $D$ must have propagated to $A$, before the clock edge.

While the clock is 1, $D$ can again change without affecting the outputs. Two conditions are possible: (a) $D$ was 0 at the clock edge, and hence $A$ is 0, $B$ is 1, $E$ is 1 and $F$ is 0. If $D$ changes to 1, there will be no change to $B$, because $F$ is 0 and hence $B$ is always 1; or (b) $D$ was 1 at the clock edge, thus $A$ is 1, $B$ is 0, $E$ is 0 and $F$ is 1. If $D$ changes to 0, $B$ changes from 0 to 1, but as $E$ is 0, this change is not propagated to $A$. Therefore again, the output is unaffected. The falling clock edge forces both $E$ and $F$ to 1 again.

It is apparent that this descriptive, intuitive form of analysis is not sufficient to adequately describe the behaviour of even relatively small asynchronous circuits. Moreover, it would be impossible to design circuits in

such a manner. It is possible to use a SystemVerilog simulator to verify the behaviour of such circuits, but we need a formal analysis technique.

### 13.2.2 Formal Analysis

Before proceeding with the formal analysis of both the D latch and the edge-triggered D flip-flop, we need to state a basic assumption. The *Fundamental Mode* restriction states that only one input to an asynchronous circuit may change at a time. The effects of an input change must have propagated through the circuit and the circuit must be stable, before another input change can occur. The need for this restriction can be seen from the two circuits already considered. If $D$ changes at almost the same time as the clock, unequal delay paths mean that internal nodes are not at expected, consistent values and unpredictable behaviour may result. In the worst case the output of a latch or flip-flop may be in an intermediate, *metastable* state, that is neither 0 nor 1. We will return to metastability later.

In order to perform a formal analysis, we have to break any feedback loops in the circuit. Of course, we don't actually change the circuit, but for the purposes of the analysis, we pretend that all the gate delays in the circuit are concentrated in one or more *virtual buffers* in the feedback loops. The gates are therefore assumed to have zero delays. The D latch is redrawn in Figure 13.5. Note that there is only one feedback loop in this circuit, although at first glance the cross-coupled NAND gate pair might appear to have two feedback loops. If the one feedback loop were really broken, the circuit would be purely combinational, which is sufficient. In Figure 13.5, the input to the virtual buffer is labelled as $Y^+$, while the output is labelled as $Y$. $Y$ is the *state variable* of the system. This is analogous to the state variable in a synchronous system. $Y^+$ is the next state. The system is *stable* when $Y^+$ is equal to $Y$. In reality, of course, $Y^+$ and $Y$ are two ends of a piece of wire and must have the same value, but, to repeat, for the purpose of analysis, we pretend that they are separated by a buffer having the aggregate delay of the system. Note that we separate the state variable from the output, although in this case, $Q$ and $Y^+$ are identical.

We can write the state and output equations for the latch as:

$$Y^+ = D.C + Y.\bar{C} + D.Y$$
$$Q = D.C + Y.\bar{C} + D.Y$$
$$\bar{Q} = \bar{D}.C + \bar{Y}$$

From this we can now write a *transition table* for the state variable, as shown in Figure 13.6.

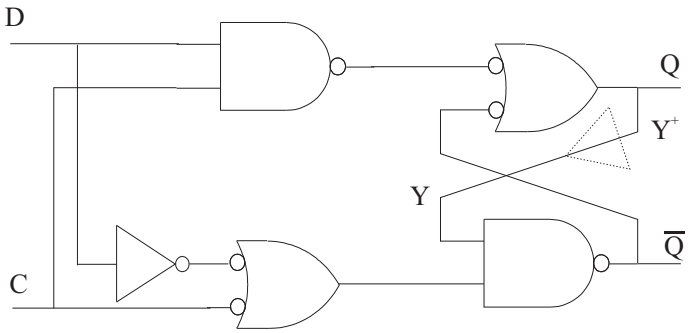



Figure 13.5: D latch with virtual buffer.

| | DC | | | |
|---|---|---|---|---|
| Y | 00 | 01 | 11 | 10 |
| 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 |

$Y^+$

Figure 13.6: Transition table for D latch.

| | DC | | | |
|---|---|---|---|---|
| S | 00 | 01 | 11 | 10 |
| K | (K),01 | (K),01 | L,11 | (K),01 |
| L | (L),10 | K,01 | (L),10 | (L),10 |

$S^+, Q\overline{Q}$

Figure 13.7: State table for D latch.

A *state table* replaces the Boolean state variables with abstract states. In the state table of Figure 13.7 the stable states are circled. A state is stable when the next state is equal to the current value. The state table can also include the outputs (*state and output table*), as shown in Figure 13.7. Notice that there is an unstable state that has both outputs the same.

Using the state and output table, we can trace the change of states when an input changes. Starting from the top left corner of the table, with the current state as $K$ and the two inputs at 0, let $D$ change to 1. From Figure 13.8, it can be seen that the state and output remain unchanged. If $C$ then changes to 1, the system moves into an unstable state. The system now has to move to the stable state at $L$, with $D$ and $C$ both equal to 1. Note that the state transition *must* be a vertical move on the state transition diagram. This is in order to comply with the fundamental mode restriction – anything other than a vertical move implies a change in an input value, which would therefore be occurring before the system was stable. It can be seen that the latch behaves as we would expect a D latch to behave. If $D$ is changed from 0 to 1, followed by $C$ changing from 0 to 1, we would expect $Q$ to change from 0 to 1, and it can be seen from Figure 13.8 that this is what happens.

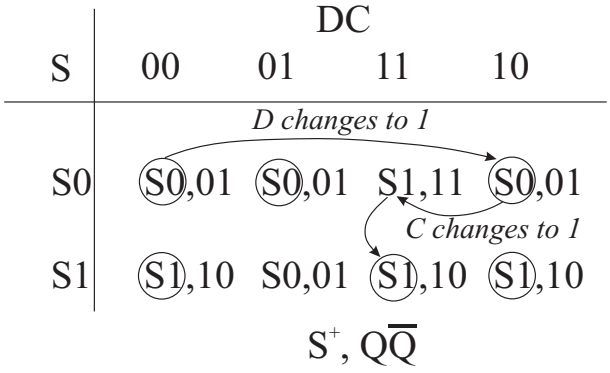



Figure 13.8: Transitions in state table.

## 13.3 Design of Asynchronous Sequential Circuits

In essence, the design procedure for asynchronous sequential circuits is the reverse of the analysis process. An abstract state table has to be derived, then a state assignment is performed, and finally state and output

equations are generated. As will be seen, however, there are a number of pitfalls along the way, making asynchronous design much harder than synchronous design. To illustrate the procedure, we will perform the design of a simple circuit, and show, both theoretically, and by simulation, the kinds of errors that can be made.

Let us design an asynchronous circuit to meet the following specification: The circuit has two inputs, *Ip* and *Enable*, and an output, *Q*. If *Enable* is high, a rising edge on *Ip* causes *Q* to go high. *Q* stays high until, *Enable* goes low. While *Enable* is low, *Q* is low.

It can be see from this specification that there are eight possible combinations of inputs and outputs, but that two combinations cannot occur: if *Enable* is low, *Q* cannot be high. This leaves six states to the system, as shown in Table 13.1.

Table 13.1: States of example asynchronous system.

| State | *Ip* | *Enable* | $Q$ |
|---|---|---|---|
| $a$ | 0 | 0 | 0 |
| $b$ | 0 | 1 | 0 |
| $c$ | 1 | 0 | 0 |
| $d$ | 1 | 1 | 0 |
| $e$ | 0 | 1 | 1 |
| $f$ | 1 | 1 | 1 |

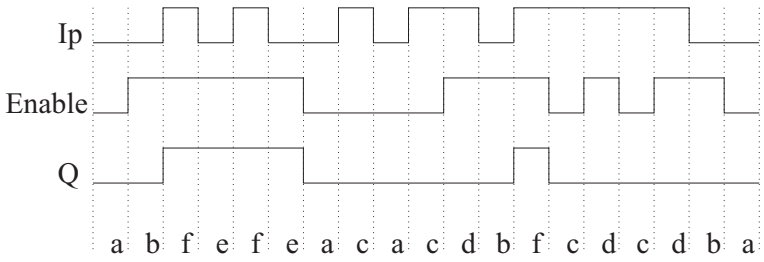



Figure 13.9: States in design example.

The first task is to work out all the possible state transitions. One way to do this is to sketch waveforms and to mark the states as shown in Figure 13.9. From this a state transition diagram can be constructed, Figure 13.10. This state diagram can also be expressed as the *Primitive Flow Table* of Figure 13.11. A primitive flow table has one state per row. Because of the fundamental mode restriction, only state transitions that are reachable from a stable state with one input change are marked. State

transitions that would require two or more simultaneous input changes are marked as “don’t cares”. The outputs are shown for the stable state and all transitions out of the state. It is also possible to assume that the outputs only apply to the stable states and that the outputs during all transitions are “don’t cares”.

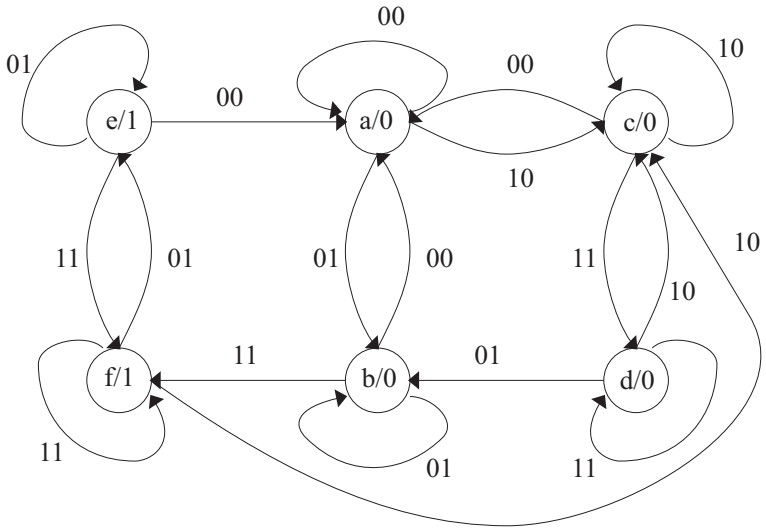



Figure 13.10: State transition diagram for design example.

There are more states in this primitive flow table than are needed. In Chapter 5, it was shown that states can be merged if they are equivalent. In this example, there are “don’t care” conditions. We now speak of states being *compatible* if their next states and outputs are the same or “don’t care”. There is an important difference between equivalence and compatibility. It can be seen that states $a$ and $b$ are compatible and that states $a$ and $c$ are compatible. States $b$ and $c$ are, however, not compatible. If $a$ and $b$ were *equivalent* and $a$ and $c$ were also equivalent, $b$ and $c$ would be equivalent by definition.

Here, states $a$ and $b$ are compatible and may be merged into state $A$, say. When compatible states are merged, “don’t cares” are replaced by defined states or outputs (if they exist). Similarly, states $c$ and $d$ may be merged into $C$ and $e$ and $f$ may be merged into $E$. The resulting state and output table is shown in Figure 13.12.

At this point, considerable care is needed in making an appropriate state assignment. We will first demonstrate how *not* to perform a state assignment. We can show, using a SystemVerilog simulation that a poor state assignment can easily result in a malfunctioning circuit. To encode three states requires two state variables, as described in Chapter 5. There

| | Ip Enable | | | | |
|---|---|---|---|---|---|
| S | 00 | 01 | 11 | 10 | Q |
| a | (a) | b | – | c | 0 |
| b | a | (b) | f | – | 0 |
| c | a | – | d | (c) | 0 |
| d | – | b | (d) | c | 0 |
| e | a | (e) | f | – | 1 |
| f | – | e | (f) | c | 1 |
| | | | $S^+$ | | |

Figure 13.11: Primitive flow table.

| | Ip Enable | | | | |
|---|---|---|---|---|---|
| S | 00 | 01 | 11 | 10 | Q |
| A | (A) | (A) | E | C | 0 |
| C | A | A | (C) | (C) | 0 |
| E | A | (E) | (E) | C | 1 |
| | | | $S^+$ | | |

Figure 13.12: State and output table.

| | Ip Enable | | | | |
|---|---|---|---|---|---|
| $Y_1Y_0$ | 00 | 01 | 11 | 10 | Q |
| 00 | (00) | (00) | 11 | 01 | 0 |
| 01 | 00 | 00 | (01) | (01) | 0 |
| 11 | 00 | (11) | (11) | 01 | 1 |
| | | | $Y_1^+Y_0^+$ | | |

Figure 13.13: Transition table.

are 24 possible state assignments. As with a synchronous system, there is no way to tell, in advance, which state assignment is "best". Therefore, let us arbitrarily assign 00 to $A$, 01 to $C$ and 11 to $E$. This gives the transition table shown in Figure 13.13. The state 10 is not used, so in deriving next state expressions, the entries corresponding to 10 are "don't cares". Hazard-free next state and output equations can be found using K-maps:

$$Y_1^+ = Y_1.Enable + Ip.Enable.\bar{Y}_0$$
$$Y_0^+ = Ip + Y_1.Enable$$
$$Q = Y_1$$

A SystemVerilog model of this circuit is as follows. The next state expressions have been given arbitrary delays. It is left as an exercise for the reader to write a suitable test bench.

If $Y_1$ and $Y_0$ are both 0 and *Ip* and *Enable* are 0 and 1, respectively, *Q* is 0. Now, let *Ip* change to 1. We would expect to move horizontally into an unstable state and then to move vertically to the stable state $Y_1Y_0$ = 11. In fact, the SystemVerilog simulation shows that the circuit goes to $Y_1Y_0$ = 01, Figure 13.14a. If the delays are reversed, however, the circuit works as expected, Figure 13.14b.

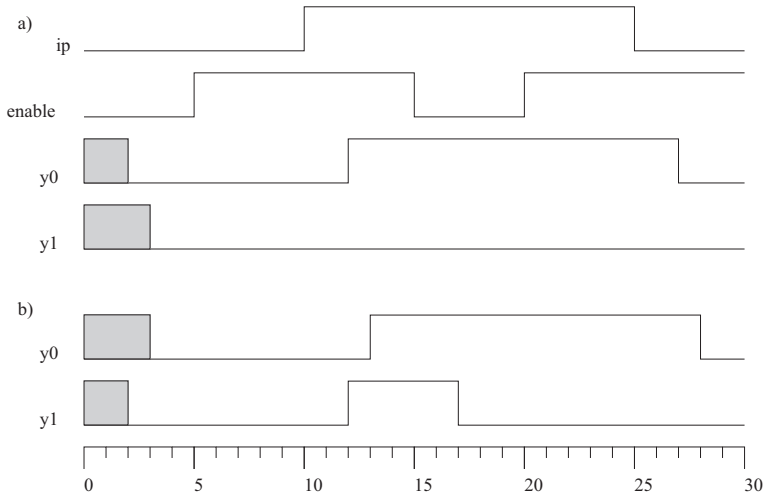



Figure 13.14: Simulation of asynchronous circuit example: (a) with race; (b) without race.

Why is the circuit sensitive to these delays? We have accounted for hazards in the Boolean minimization, so they are not the problem. Let us consider the transition table, including the unused state, with the values for the unused state as implied by the minimized equations, as shown in Figure 13.15.

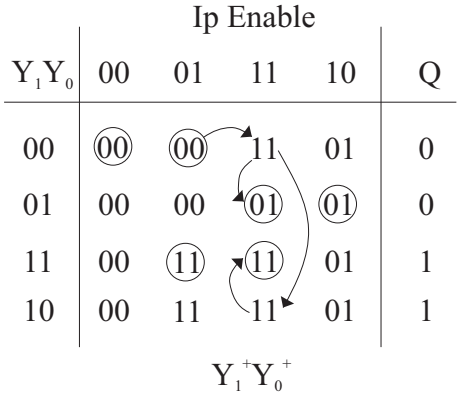



Figure 13.15: Transition table with critical race.

In the first case, $Y_1$ changes first, therefore the circuit changes to the unstable state 10, at which point $Y_0$ changes and the circuit finishes in the correct state. In the second case, $Y_0$ changes first and the circuit moves to the stable state 01, *and stays there*! In other words, the order in which the state variables change can affect the final state of the circuit. The situation in which two or more state variables change as a result of one input change is known as a *race*. If the final state depends on the exact order of the state variable changes, that is known as a *critical race*. There is a potentially even more disastrous situation. If the don't cares in the K-maps produced from the transition table of Figure 13.13 were forced to be 0 (which results in non-minimal next state expressions, but is otherwise perfectly legitimate), the next state equations become:

$$\begin{aligned} Y_1^+ &= Y_1.Y_0.Enable + Ip.Enable.\bar{Y_1}.\bar{Y_0} \\ Y_0^+ &= Ip.\bar{Y_1} + I.Y_0 + Y_1.Y_0.Enable \end{aligned} \quad .$$

When the SystemVerilog model shown below is simulated, the circuit oscillates, as shown in Figure 13.16.

Figure 13.17 shows the transition table. $Y_1$ changes to 1, before $Y_0$ can react, so the circuit moves to state 10. $Y_1$ is then forced back to 0, so the circuit oscillates between states 00 and 01. This is known as a *cycle*.

We clearly have to perform a state assignment that avoids both critical races and cycles. In this example, such an assignment is not possible with just three states. Therefore we have to introduce a fourth state. This state is unstable, but it ensures that only one state variable can change at a time. Figure 13.18 shows the modified state table, while Figure 13.19 shows a simplified state transition diagram, with the newly introduced state, $G$, and a suitable state assignment. Hence, expressions for the state variables can be derived. In this case, the state variable expressions are:

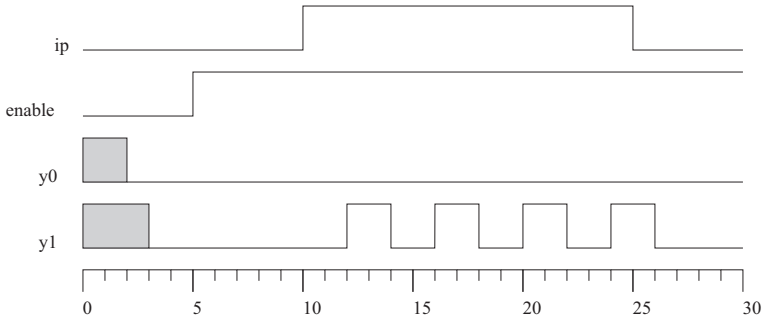



Figure 13.16: Simulation of asynchronous circuit with cycle.

| | Ip Enable | | | | |
|---|---|---|---|---|---|
| $Y_1Y_0$ | 00 | 01 | 11 | 10 | Q |
| 00 | (00) | (00) | 11 | 01 | 0 |
| 01 | 00 | 00 | (01) | (01) | 0 |
| 11 | 00 | (11) | (11) | 01 | 1 |
| 10 | 00 | 00 | 00 | 00 | 1 |
| | | | $Y_1^+Y_0^+$ | | |

Figure 13.17: Transition table with cycle.

$$Y_1^+ = Y_1.Y_0.\bar{I} + Y_1.Enable + Ip.Enable.\bar{Y}_0$$
$$Y_0^+ = Ip.\overline{Enable} + Ip.Y_0 + Y_1.Enable \quad .$$

We can simulate SystemVerilog models of this circuit with either $Y_1$ or $Y_0$ changing first, and in both cases the circuit works correctly.

| | Ip Enable | | | | |
|---|---|---|---|---|---|
| S | 00 | 01 | 11 | 10 | Q |
| A | (A) | (A) | G | C | 0 |
| C | A | A | (C) | (C) | 0 |
| E | G | (E) | (E) | C | 1 |
| G | A | – | E | – | – |
| | | | $S^+$ | | |

Figure 13.18: Modified state table.

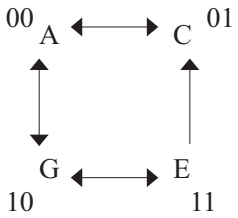



Figure 13.19: Simplified state transition diagram.

There is, however, one final potential problem. There are no possible redundant terms, in this example, so we can be sure that all potential static hazards have been eliminated. In principle, therefore, the circuit can be built as shown in Figure 13.20. If, however, as a result of the particular technology used or the particular layout adopted, the input to the top AND gate is delayed with respect to the state variables, as shown, the circuit may still malfunction. This condition can be demonstrated again with a SystemVerilog model.

The transition table of Figure 13.21 shows what happens if *Ip* changes from 1 to 0 from state 01 while *Enable* stays at 1. In theory this change should only cause transitions $1a$ and $1b$ and the final state should be 00. In practice, because of the delay in *Ip*, the circuit then follows the other

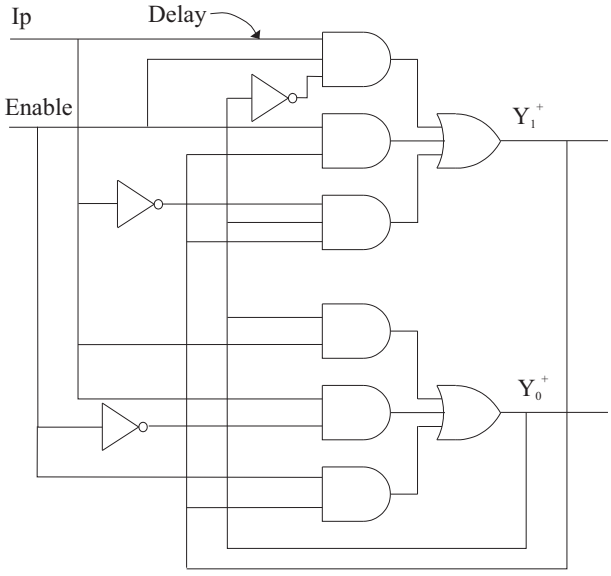



Figure 13.20: Circuit with essential hazard.

transitions shown, $2a$, $2b$, $3a$, and $3b$, to finish in state 11. This is known as an *essential hazard*, so-called because it is part of the essence of the circuit. Potential essential hazards can be identified from the transition table if a single input change results in a different final state than if the input changes three times. The only way to avoid essential hazards is to ensure that the state variables cannot be fed back round the circuit before the input transitions. This can be achieved by careful layout or possibly by deliberately introducing delays into the state variables.

| | Ip Enable | | | | |
|---|---|---|---|---|---|
| $Y_1Y_0$ | 00 | 01 | 11 | 10 | Q |
| 00 | (00) | (00) | 10 | 01 | 0 |
| 01 | 00 | 00 | (01) | (01) | 0 |
| 11 | 10 | (11) | (11) | 01 | 1 |
| 10 | 00 | – | 11 | – | – |

$Y_1^+Y_0^+$

Figure 13.21: Transition table with essential hazard.

In summary, therefore, the design of an asynchronous sequential circuit

has the following steps:

1 State the design specifications.

2 Derive a primitive flow table

3 Minimize the flow table

4 Make a race-free state assignment

5 Obtain the transition table and output map

6 Obtain hazard-free state equations

7 Check for essential hazards

## 13.4 Asynchronous state machines

In the design flow, above, the first step is to derive the design specifications. In many ways this is the hardest part of the task. Moreover, if we get that wrong, everything that follows is also, by definition wrong. By the nature of the design process, it is almost impossible to patch a mistake – the entire process has to be repeated. Therefore, it would be very desirable to ensure that the design has been specified correctly. One way to do this is to use simulation again.

The state transition diagram of Figure 13.10, is essentially the same as the state diagram of Figure 5.9 or that of Figure 11.19. One figure represents an asynchronous system and two represent a synchrnous systems. This difference is not, however, apparent from the diagrams. We advocated the use of ASM charts for the design of synchronous systems, but we could have used state diagrams. We know that an ASM chart or a state diagram has an equivalent SystemVerilog description. By the same argument, we can represent an asynchronous state machine in SystemVerilog. Instead of a set of registers synchronized to a clock, we would have a virtual buffer, in which the state variable is updated. Let us therefore write a SystemVerilog description of the state machine of Figure 13.10.

The virtual buffer has a delay of 1 ns. For this type of model to work, there must be a finite delay – a zero delay would cause the process to loop infinitely at time 0. For reasons of space, the entire state machine is not shown; the other states may be written in the same way. The don't cares have been assumed to cause the state machine to stay in the same state. As these represent violations of the fundamental mode, this is valid. It is

possible to check for fundamental mode violations by including an assert statement in the process:

This is not strictly correct as three simultaneous events would not trigger the assertion (see exercise **??**). With a suitable testbench, we can use this SystemVerilog model to reproduce Figure 13.9. Notice that the initial values of the state variables will be the leftmost entry in the state definition – a.

We can also repeat the exercise after state minimization.

Again, this can be verified by simulation. Indeed, this is one way to check that the state minimisation has been done correctly.

As a second example, consider the following. We wish to design a phase detector with two outputs: *qA* and *qB*. There are also two inputs: *inA* and *inB*. Let us assume both outputs start high. When *inA* goes high, *qA* goes low and stays low until *inB* goes high. Similarly, if *inB* goes low first, *qB* goes low until *inA* goes high. This sounds very simple! We will model the phase detector as an asynchronous state machine. It is left as an exercise for the reader to derive the SystemVerilog model, below, to implement this specification. You can further test your understanding of asynchronous design by taking this design through to gate level.

```
module phase_detector (input logic inA, inB,
                       output logic qA, qB);

enum {A, B, C, D, E, F, G, H} present_state, next_state;

always @*
  begin
  next_state = present_state;
  qA = '1;
  qB = '1;

  case (present_state)
    A: if (~inA && inB)
         next_state = E;
       else if (inA && ~inB)
         next_state = B;
    B: begin
       qA = '0;
       if (~inA && inB)
         next_state = D;
       else if (inA && inB)
```

```
            next_state = C;
          end
      C:  if (~inA && inB)
            next_state = D;
          else if (inA && ~inB)
            next_state = F;
      D:  if (~inA && ~inB)
            next_state = A;
          else if (inA && inB)
            next_state = H;
      E:  begin
          qB = '0;
          if (inA && inB)
            next_state = C;
          else if (inA && ~inB)
            next_state = F;
          end
      F:  if (~inA && ~inB)
            next_state = A;
          else if (inA && inB)
            next_state = G;
      G:  begin
          qB = '0;
          if (~inA && ~inB)
            next_state = E;
          else if (~inA && inB)
            next_state = D;
          end
      H:  begin
          qA = '0;
          if (~inA && ~inB)
            next_state = B;
          else if (inA && ~inB)
            next_state = F;
          end
    endcase
    end

    assign #1ns present_state = next_state;

endmodule
```

One final word of warning: do not try to synthesise these state machine models! In the light of the previous discussions, it should be obvious that you would generate hardware with races and hazards!

## 13.5 Setup and Hold Times and Metastability

### 13.5.1 The Fundamental Mode Restriction and Synchronous Circuits

The fundamental mode restriction requires that an input to an asynchronous circuit must not change until the circuit has become stable after a previous input change. Individual flip-flops are themselves asynchronous internally, but are used as synchronous building blocks. We do not, however, speak of the fundamental mode restriction when designing synchronous systems. Instead, we define setup and hold times.

Because of the gate delays in a circuit, the fundamental mode restriction *does not* mean that two inputs must not change at the exact same time. It means that the effect of one input change must have propagated through the circuit before the next input can change. To use the example of a D flip-flop, a change at the D input must have propagated through the flip-flop before an active clock edge may occur. Similarly, the effect of the clock edge must have propagated through the circuit before the D input can change again. These two time intervals are known as the setup and hold times, respectively.

The setup and hold times of a latch or flip-flop depend on the propagation delays of its gates. These propagation delays depend, in turn, on parametric variations. So we can never know the exact setup and hold times of a given flip-flop. Furthermore, the timing of clock edges may be subject to *jitter* – the exact period of the clock may vary slightly. Therefore there has to be a margin of tolerance in estimating the setup and hold times. It should finally be noted that some of the effects of ignoring the fundamental mode restriction, or equivalently, violating setup and hold times, are not purely digital. In particular, metastability is effectively an analogue phenomenon.

Bearing all this in mind, it is possible to get some insight into the consequences of not observing the fundamental mode restriction by using a SystemVerilog simulator. We could use trial and error to find the setup and hold times of a flip-flop; if the gate delays are specified, it is not difficult to calculate the various path lengths through the circuit. Here, however, we

will show how to generate a random pulse stream.

### 13.5.2 Random Pulse Generator

### 13.5.3 SystemVerilog Modelling of Setup and Hold Time Violations

A structural model of a level-sensitive D latch can be described in SystemVerilog using gate instances or by using a set of concurrent assignments, as shown below. Note that q and qbar are declared as ports with mode **out**, so they cannot be read. Therefore, two internal signals, y and z are used to model the RS latch. If a simulation of this latch is run, using a regular clock and a random event generator for the D input, as shown in the testbench fragment, it will be observed that the latch works correctly, unless the D input changes 2 ns or less before a falling clock edge. If this occurs, the q and qbar outputs oscillate.

Of course, two D latches can be put together to form an edge-triggered flip-flop. The clock input is inverted for the master flip-flop (introducing a delay of, say, 1 ns). Thus when the clock is low, the master flip-flop is conducting. From the previous simulation, we would expect therefore that the setup time is 2 ns, less the delay in the clock caused by the inverter, or 1 ns in total. We can verify this by simulation. Again we observe that a change in the D input 1 ns or less before the clock edge may cause the output to oscillate, depending on the state of the flip-flop and whether D is rising or falling. The six-nand gate edge-triggered D flip-flop behaves similarly. In both cases, the hold time is 0 ns.

Part of the testbench is shown below.

There has to be some doubt as to whether this modelled behaviour is exactly what would be observed in a real circuit. These SystemVerilog models assume that 0 to 1 and 1 to 0 transitions are instantaneous. Of course, in reality, such transitions are finite. Therefore, if a gate had one of its two inputs rising and the other falling simultaneously, it would be reasonable to expect that the output might switch into some state that was neither a logic 1 nor a logic 0 for a period of time. The SystemVerilog standard logic package does not include such a state; ‘X’ is generally taken to represent a state that could be one of 1 or 0.

### 13.5.4 Metastability

While the oscillations predicted by both the structural models may occur if the fundamental mode restriction is violated, another condition can occur that a SystemVerilog simulation cannot predict. All flip-flops have two stable states and a third unstable, or *metastable* state. In this metastable state both flip-flop outputs have an equal value at a voltage level between 0 and 1. A SPICE, or similar, transistor-level operating point analysis is likely to find this metastable condition. This may be likened to balancing a pencil on its point – in theory it is stable, but in practice, noise (vibrations, air movement etc.) would cause the pencil to topple. The metastable state of a flip-flop is similarly unstable, electrical or thermal noise would cause it to fall into a stable state.

Metastability is most likely to occur when external (asynchronous) signals are inputs to a synchronous system. If metastability is likely to be a problem, then care needs to be taken to minimize its effects. The threat of metastability can never be entirely eliminated, but there is no point in constructing elaborate defences if the chances of its happening are remote. Therefore the critical question is how likely is it to occur? The formula used to calculate the mean time between failures (MTBF) has been found, by experiment, to be:

$$MTBF = \frac{\exp\left(T \times t_x\right)}{f_{clk} \times f_{in} \times T_0}$$

$t_x$ is the time for which metastability must exist in order that a system failure occurs. If a metastable state occurs at the output of a flip-flop, it will cause a problem if it propagates through combinational logic and affects another flip-flip. Therefore,

$$t_x = t_{clk} - t_{pd} - t_{setup}$$

where $t_{clk}$ is the clock period, $t_{pd}$ is the propagation delay through any combinational logic and $t_{setup}$ is the setup time of the second flip-flop.

$f_{clk}$ is the clock frequency, $f_{in}$ is the frequency of the asynchronous input changes and $T$ and $T_0$ are experimentally derived constants for a particular device.

Let us put some numbers into this formula. The system is clocked at 10 MHz, therefore $t_{clk}$ is 100 ns. We will examine whether an input flip-flop with a setup time of 10 ns can go into a metastable state, therefore $t_{pd}$ is zero and, hence, $t_x$ is 90 ns. If the asynchronous input changes on average, say once every 10 clock cycles, $f_{in}$ is 1 MHz. For a relatively

slow D flip-flop (e.g. a 74LS74), $T$ is about $7\times10^8$ seconds, while $T_0$ is 0.4 seconds. Therefore

$$MTBF = \frac{\exp\left(7 \times 10^8 \times 90 \times 10^{-9}\right)}{10^7 \times 10^6 \times 0.4} = 5.7 \times 10^{12}\,\text{sec}$$

or about 200,000 years. Metastability is unlikely to be a problem in such a system. But suppose the clock frequency is doubled to 20 MHz, and hence $t_x$ becomes 40 ns. Now,

$$MTBF = \frac{\exp\left(7 \times 10^8 \times 40 \times 10^{-9}\right)}{2 \times 10^7 \times 10^6 \times 0.4} = 0.18\,\text{sec}\,.$$

So, we probably will have a problem with metastability in this system.

There are several ways to alleviate the problem. The flip-flop cited above is very slow. A faster flip-flop would have a larger $T$ and a smaller $T_0$. So, using a faster flip-flop will increase the MTBF. Another common solution is to use two flip-flops in series as shown in Figure 13.22.

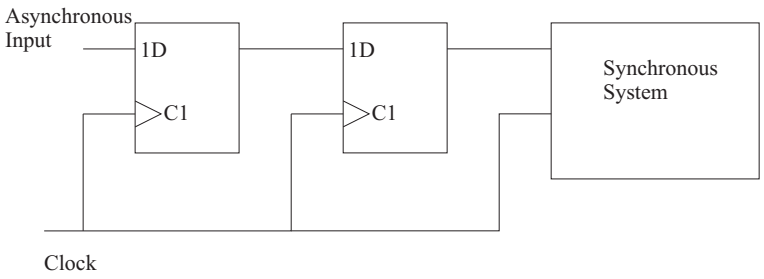



Figure 13.22: Synchronizer design.

This arrangement does not necessarily reduce the MTBF, but it does reduce the possibility that a metastable state is propagated into the synchronous system.

Although it is fairly unlikely that metastability would be observed in a student laboratory, it is apparent that with increasing clock speeds and perhaps a move towards a style of design in which there is no global clock, coping with metastability is going to be a challenge for digital designers.

# Summary

The design and analysis of asynchronous circuits is harder than for synchronous circuits. Asynchronous circuits may be formally analyzed by breaking feedback loops. The design of an asynchronous circuit starts

from a description of all the possible states of the system. A primitive flow table is constructed, which is then minimized. State assignment follows. A poor state assignment can result in race conditions or cycles. From the transition table, next state and output expressions are derived. Hazards can cause erroneous behaviour or oscillations. Essential hazards may result from uneven delays. The design of asynchronous circuits depends on observing the fundamental mode restriction. This is reflected in the specification of setup and hold times for asynchronous blocks used in synchronous design. Failure to observe these restrictions can lead to spurious behaviour and possibly metastability.

## Further Reading

Although the design of asynchronous (or level-mode, or fundamental mode) sequential circuits is covered in many textbooks, close reading reveals subtle variations in the techniques. Hill and Peterson provide a very good description. Wakerly has a very straightforward description. Unger's 1995 paper has provided perhaps the most rigorous analysis of the problems of metastability. The Amulet project has one of the most significant large asynchronous designs and the Web site (http://www.cs.man.ac.uk/amulet/index.html) has links to many sources of information about asynchronous design.

## Exercises

13.1 What is the difference between a synchronous sequential circuit and an asynchronous sequential circuit? Why is synchronous design preferred?

13.2 What assumption is made in the design of fundamental-mode sequential circuits, and why? How can essential hazards cause the fundamental mode to be violated?

13.3 The excitation equation for a D-latch may be written as

$$Q^+ = C.D + Q.\bar{C}$$

Why would a D-latch implemented directly from this transition equation be unreliable? How would the D-latch be modified to make it reliable?

13.4 Describe, briefly, the steps needed to design an asynchronous sequential circuit.

13.5 Figure 13.23 shows a master-slave edge-triggered D flip-flop. How many feedback loops are there in the circuit, and hence how many state variables?

Derive excitation and output equations and construct a transition table. Identify all races and decide if the races are critical or non-critical.

Construct a state & output table and show that the circuit behaves as a positive edge-triggered flip-flop.

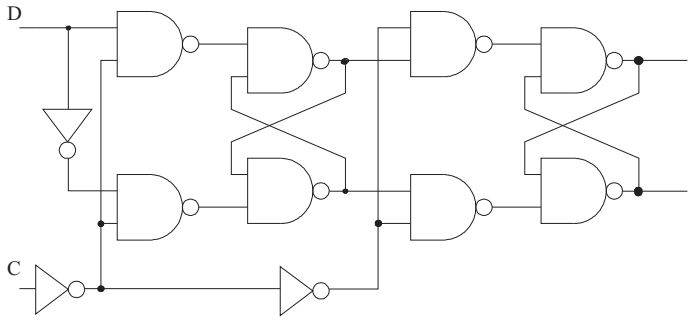



Figure 13.23: Circuit for Exercise 13.5.

13.6 Figure 13.24 shows a state diagram of an asynchronous circuit with 2 inputs, $R$ and $P$ and a single output, $Q$. The input values are shown on the arcs; the state names and the output values of the stable states are shown in the circles. Design an asynchronous circuit to implement this function.

13.7 A positive edge-triggered D flip-flop has a preset and clear input, in addition to the clock and D inputs (Figure 13.4). Write down the state equations for the flip-flop including the preset and clear inputs. Hence write a transition table.

13.8 Table 13.2 shows the transition table for an asynchronous circuit. Identify all the non-critical races, critical races and cycles (a *cycle* is a repeated series of unstable states that requires an input to change in order for a stable state to be reached).

13.9 Design a D flip-flop that triggers on both the positive and negative edges of the clock pulse.

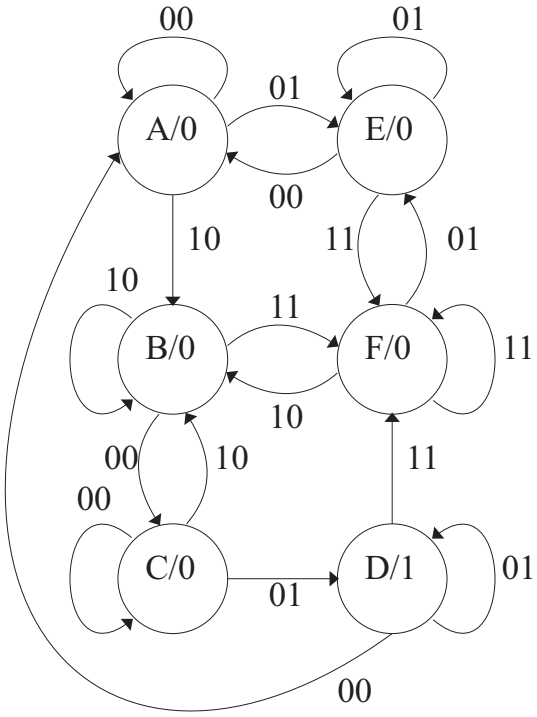



Figure 13.24: State diagram for Exercise 13.6.

Table 13.2: Transition table for Exercise 13.8

| | $AB$ | | | |
|---|---|---|---|---|
| $Y1Y2$ | 00 | 01 | 11 | 10 |
| 00 | 00 | 11 | 10 | 11 |
| 01 | 11 | 01 | 01 | 10 |
| 11 | 10 | 11 | 01 | 10 |
| 10 | 11 | 10 | 01 | 01 |
| | Y1*Y2* | | | |

13.10 An asynchronous sequential circuit has two inputs, two internal states and one output. The excitation and output functions are:

$$Y1^{+} = A.B + A.\overline{Y2} + \bar{B}.Y1$$
$$Y2^{+} = B + A.\overline{Y1}.\overline{Y2} + \bar{A}.Y1$$
$$Z = B + Y1$$

(a) Draw the logic diagram of the circuit.

(b) Derive the transistion table and output map.

(c) Obtain a flow table for the circuit.

# Chapter 14

# Interfacing with the Analogue World

In previous chapters, we have considered the world to be purely digital. Indeed, with the exception of the last chapter, we have further considered only synchronous systems. Of course the real world is asynchronous and, even worse, analogue. All digital systems must at some point interact with the real world. In this chapter, we will consider how analogue inputs are converted to digital signals and how digital signals are converted to analogue outputs. Until relatively recently, the modelling and simulation of digital and analogue circuits and systems would have been performed independently of each other. A set of analogue and mixed-signal extensions to Verilog (but not SystemVerilog) has been proposed. The language is commonly known as Verilog-AMS (Analogue and Mixed-Signal). Verilog-AMS is a complete superset of the 1995 standard for Verilog. At some point in the future, it is likely that SystemVerilog-AMS will appear, but meanwhile, simulators that support mixtures of Verilog, Verilog-AMS and SystemVerilog exist. Having looked at digital to analogue converters (DACs) and analogue to digital converters (ADCs), we will review the basics of Verilog-AMS and see how ADCs and DACs can be modelled in Verilog-AMS. There is not sufficient space to provide a complete tutorial of Verilog-AMS here. Furthermore, it should be remembered that we are only considering simulation models, designed for verifying the interaction of a digital model with the real world. Synthesis of analogue and mixed-signal designs is still a research topic. The final section of the chapter looks at some further mixed-signal circuits and their models in Verilog-AMS.

## 14.1 Digital to Analogue Converters

We will start the discussion of interface circuits with digital to analogue converters because, as we will see, one form of analogue to digital converter requires the use of a DAC. The motivation in this chapter is not to describe very possible type of converter – that would require at least an entire book – but to show one or two examples of the type of circuit that can be employed.

In moving between the analogue and digital worlds, we ideally want to preserve the maximum amount of information. This can be summarized in terms of three aspects: *resolution*, *accuracy* and *speed*. *Resolution* defines the smallest change that can be measured. For example, 8 bits can represent $2^8$ or 256 voltage levels. If we want to represent a signal that changes between 0 and 5 volts using 8 bits, the resolution is 5/256 = 19.5 mV. *Accuracy* describes how precisely a signal is represented with respect to some reference. In turn, this depends on factors such as linearity. For example, while 8 bits can represent a 5 V signal with an *average* resolution of 19.5 mV, differences (non-linearities) in the circuit might mean that some changes are really 18.5 mV, while others are 20.5 mV. These differences will add up and affect the overall accuracy. Finally, the *speed* at which data is converted between the two domains affects the design of converters. In the digital world, samples are taken at discrete points in time. The users of converters need to be aware of what happens between these sample points.

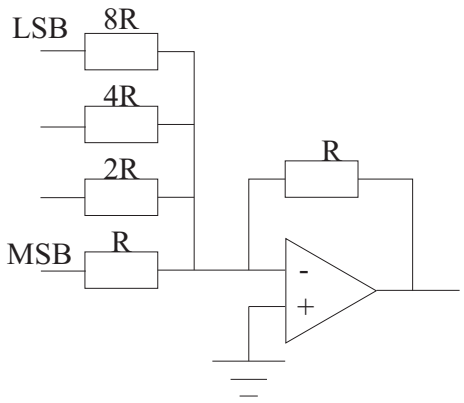



Figure 14.1: Binary-weighted ladder DAC.

The simplest type of DAC is the binary-weighted ladder circuit of Figure 14.1. The bits are added together according to their relative weights. The operational amplifier forms a classic (inverting) adder. While this circuit is

easy to understand, it is a manufacturing nightmare. The resistors have to be manufactured with very tight tolerances. Any inaccuracy in a resistor value would affect the accuracy. Note that the resistors have to be accurate with respect to the feedback resistor (R), but also with respect to each other.

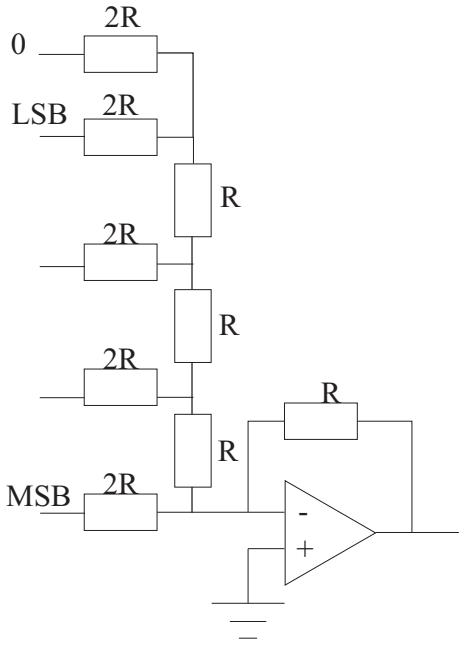



Figure 14.2: Binary-weighted R-2R ladder DAC.

A variation on the binary-weighted ladder is the R-2R ladder of Figure 14.2. To a significant extent, this overcomes the manufacturing problem as only two values of resistor need to be constructed.

For both these circuits, the speed is limited only by the response of the opamp. In practice, however, we might find that the resistors are more easily implemented as switched capacitors[1]. If this is so, the speed is limited by the clock. Notice also that the output changes in discrete steps.

## 14.2 Analogue to Digital Converters

The task of an ADC is to translate a voltage (or current) into a digital code. This is generally harder to achieve than the reverse process. Again, we

[1]In CMOS technology, it is generally easier to build accurate capacitors than accurate resistors. It is possible to emulate the behaviour of a resistor by rapidly switching a capacitor between an input and ground.

need to consider resolution, accuracy and speed, but. For example, suppose we have a signal that changes between 0V and 5V, with a maximum frequency of 10 kHz. Eight bits gives a resolution of 19.5 mV, as above. To accurately capture changes in a signal, it needs to be sampled at twice its maximum frequency. Here therefore, we need to sample at 20 kHz or greater.

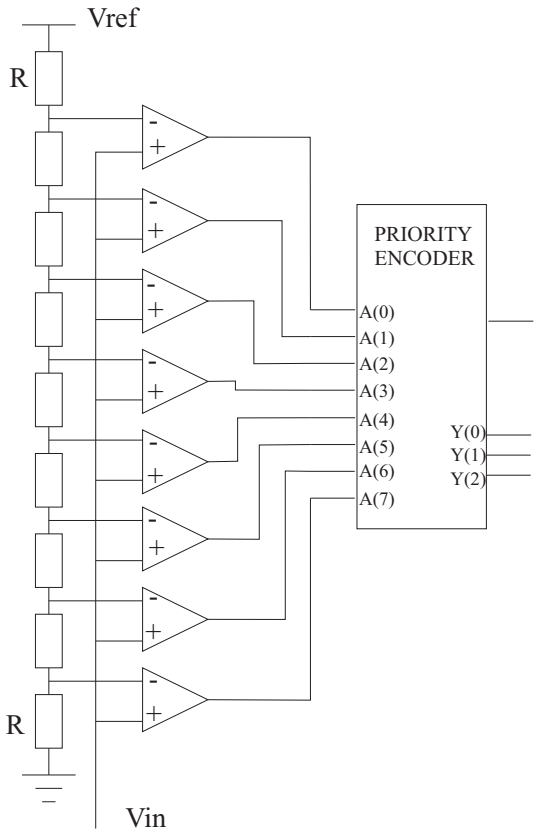



Figure 14.3: Flash ADC.

The simplest (conceptually) ADC is the flash ADC of Figure 14.3. This consists of 9 identical resistors (for 8 voltage levels) and 8 comparators. As the input voltage, Vin, increases past a level in the resistor change, the corresponding comparator output switches to 1. Therefore, we can use a priority encoder to determine which is the most significant bit, and to encode that value as a binary number. It should be immediately obvious that this circuit is impractical for large numbers of bits. We need $2^n$ identical, ideal comparators and $2^n + 1$ identical resistors to achieve n bits at the output. It is very difficult to achieve high consistency and hence high accuracy. On the other hand, this type of converter is very fast. In practice,

the cost of a flash ADC is usually too high. In return for a smaller design and better accuracy, we pay the price of slower conversion speeds.

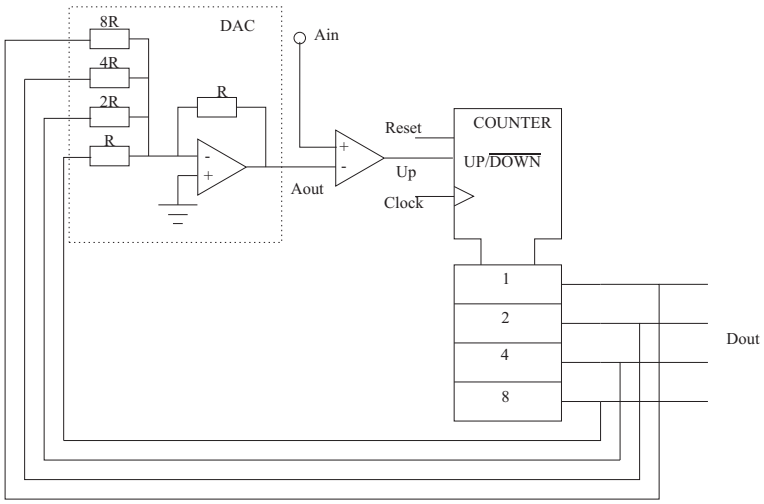



Figure 14.4: Tracking ADC.

Figure 14.4 shows a tracking ADC. This is much easier to implement than the flash ADC. It is essentially a DAC, a comparator and a counter. When the value in the counter is greater than that of the input, Ain, the counter counts down; when the counter's value is less than Ain, the counter counts up. Therefore, the counter attempts to track the input. At first glance, it might appear that a very high clock speed is needed to make this work. Suppose we wish to convert an audio signal with a maximum frequency of 20 kHz. We need to sample at twice this frequency – 40 kHz. In the worst case, the counter needs to count through its entire range, $2^4$ or 16 states, between samples. This means that the counter clock must be 16 $\times$ 40 kHz or 640 kHz. On the other hand, to achieve CD quality resolution, we would need 16 bits at the output, which implies a clock speed of nearly 3 GHz. This is clearly much less practical.

For high-speed, high-resolution applications an entirely different approach is usually taken. Delta-Sigma ADCs convert from voltage to a serial encoding. Figure 14.5 shows a simple Delta-Sigma ADC. The mark to space ratio of the output is proportional to the ratio of the input voltage to some reference, Vref, (as set by the DAC). Let us assume that the DAC output is at Vref. When Vin is less than Vref, the output of the first comparator is negative. This causes the integrator output to ramp downwards. When that output crosses zero (possibly after several clock cycles), the output of the second comparator goes negative. At the next clock edge,

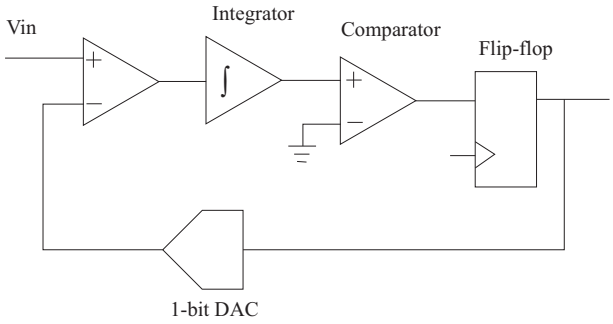



Figure 14.5: Delta Sigma ADC.

a 0 is stored in the flip-flop, causing the DAC to output zero. Now the first comparator causes the integrator to start ramping upwards. Again this might take several clock cycles. In this way, the mark to space ratio of the output is changed. This type of converter is widely used in digital audio applications. The resolution is determined by the clock freauency. As with the tracking ADC, for high resolution, a very high clock speed is needed. However, by using differential coding methods (in other words, by recording changes rather than absolute signal values), the clock speed requirement can be significantly reduced.

In the following section, we will see how some of these circuits can be modelled in Verilog-AMS. It should be borne in mind that these models simply describe the functional behaviour of converters. We have already noted that DACs and ADCs are subject to limitations in terms of accuracy, resolution and speed. Very often it is necessary to model these imperfections and to use the results of such simulations to determine the most suitable designs. As with much else in this chapter, detailed modelling of converter circuits could comprise yet another complete book.

## 14.3 Verilog-AMS

### 14.3.1 Mixed-Signal Modelling

Verilog-AMS is a mixed-signal modelling language. Therefore, we can mix ”analogue” and ”digital” constructs in the same models. Let us consider a simple comparator. We want to convert two analogue voltages into a one-bit digital signal, such that the output is a logic ’1’ when the the first input greater than the second and ’0’ otherwise.

This example simply converts a signal to one bit. We can use the

comparator as part of a flash ADC (see section 13.2 and exercise 13.??). Later, we will use the comparator again as part of a tracking ADC. We can, however, also model a flash ADC behaviourally. We simply need to convert a varying (real) quantity into a bit vector. In the example below, the model is parameterised in terms of the analogue voltage range and the number of bits. We also include a clock to sample the waveform – otherwise the model will be evaluated at every analogue time step.

In the following example, a digital to analogue converter is modelled as a voltage source and resistance in the analogue world. The voltage source can take one of three values – V1, V0 or Vx for logic 1, logic 0 or unknown, repectively. Similarly, the output resistance can take a low impedance value or a high impedance value. If we assume that metavalues such as 'U' can be mapped onto 'X', this allows us to represent all std_logic values as voltage and resistance pairs. A parameterised entity desciption for a DAC is as follows.

To convert from analogue quantities to digital signals, we write concurrent statements. To convert the other way, we need to write simultaneous statements. There is a catch, however. In discrete simulation ("standard" Verilog), signals change instantaneously. In a continuous simulation, instantaneous step changes cause problems.

Without going into great detail, an analogue or continuous solver approximates a changing quantity by taking discrete time steps. The waveform is therefore approximated by a polynomial expression. The size of these time steps is varied to minimize the error in the polynomial. A large step change makes it impossible to construct a polynomial expression across that change, so the error is considered large and the time step is reduced in an attempt to minimize the error. No matter how small the time step is made, the error will remain large and the simulation fails.

One way to avoid instantaneous changes is to force a transition to occur in a finite time. This can be done with the transition() function. The values of the voltage and resistance are held as signals within the DAC model. When the input signal changes, these signals are updated. An expression for the output voltage in terms of these signals can then be written as a simultaneous statement. Note that changes in the signals are slowed by 1 ns using the transition() function.

The obvious disadvantage of this approach is that the time to change between values has to be specified. 1 ns might easily be far too large or far too small compared with other changes in the system. It would be better to let the solver decide for itself what would constitute a suitable change. For this to happen, the solver needs to be told that there could be a problem, and this is the responsibility of the model writer. Verilog-AMS includes a

mechanism for indicating a discontinuity.

We now have the necessary parts to build the tracking ADC from section 14.2. We also need the counter from exercise 6.6. This has been written as a self-contained testbench. It would be equally valid to include the four component parts in a separate entity. Notice that we have created a netlist in exactly the same way as digital netlist, the only difference being that the analogue nodes needed for connecting components are declared as

## 14.4 Phased-Locked Loops

Although ADCs and DACs are the main interfaces between the analogue and digital worlds, another class of circuits also sits at this boundary. One of the major uses for phase-locked loops (PLLs) is for generating clocks. PLLs can be used to recover the clock from a stream of data. A PLL can also be used to "clean up" a clock that has an irregular period and to multiply a clock signal to create a higher frequency signal. All of these tasks are difficult to achieve with conventional digital circuit techniques. PLLs can be built as purely analogue circuits, as purely digital circuits or using a mixture of methods. As with ADCs and DACs, there is not enough space in a book like this to give any more than a brief introduction to PLLs. The purpose here is to show a simple example and to illustrate one way of modelling that example in Verilog-AMS. As with ADCs and DACs, the real art of modelling PLLs is to capture non-linearities and other imperfections to determine whether a particular design will work in a particular context.

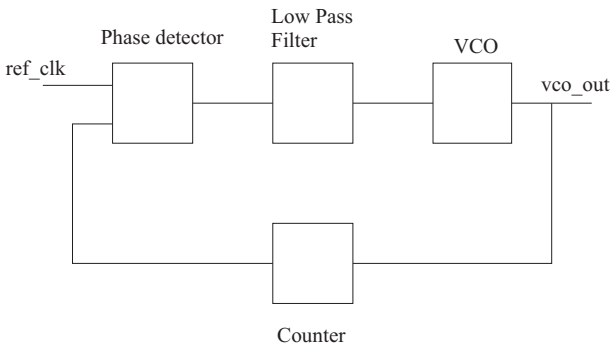



Figure 14.6: PLL structure.

Figure 14.6 shows the basic structure of a PLL. The phase detector determines the difference between the input (ref_clk) and the stabilized

output (vco_out). The phase detector could be an analogue four quadrant multiplier or a digital XOR gate or a sequential digital circuit. The output from the phase detector is a sequence of pulses. The low pass filter averages these pulses in time. This filter is crucially important to the working of the PLL. If its time constant is too small, the PLL will not settle into a regular "locked" pattern. If the time constant is too great, the PLL may not lock at all. The voltage controlled oscillator (VCO) converts the output of the filter into a oscillation whose frequency is determined by the filter output voltage. The VCO is likely to be the hardest part of the design. It can only oscillate within a relatively narrow band of frequencies. Finally, the counter is optional. By dividing the VCO output, the phase detector compares with this reduced frequency output. In other words the VCO output must be a multiple of the input frequency.

There are many books about PLL design, but perhaps the best way to understand their operation is by playing with the circuit parameters in a simulation. Therefore, we will simply present one, ideal, model of a PLL.

We start with the largest model – the phase detector. We will use the example from the last chapter. The two outputs, qa and qb, correspond to two control signals, up and down, respectively. These need to be converted to analogue voltages and filtered. We will use two instances of the one-bit DAC from the previous section. The low pass filter can be modelled using the Laplace Transform attribute in Verilog-AMS. This attribute takes two parameters, each of which is a vector of real numbers. The first vector contains the coefficients of the numerator and the second contains the coefficients of the denominator. Here we want to create a parameterized low pass filter, which in the s-domain has the transfer function:

$$\frac{1}{1+sT}.$$

Therefore the numerator has the value 1.0, and the denominator has the values 1.0 and T. Hence, this is the Verilog-AMS model. Although this is a frequency domain model, it can be interpreted in the time domain. Similarly, time domain models (such as ddt(v)) can be interpreted in the frequency domain.

```
`include "disciplines.vams"

module lpf (Ao, Ai);

inout Ao, Ai;
electrical Ao, Ai;
```

```
parameter real T = 1e-6 from [0:inf);

analog
  V(Ao) <+ laplace_nd(V(Ai), {1}, {1, T});

endmodule
```

The voltage controlled oscilllator is mixed-signal, but can be written using a Verilog process.

```
`include "disciplines.vams"
`timescale 1 s / 100 ps

module vco(Ina, Inb, vout);

inout Ina, Inb;
output vout;
electrical Ina, Inb;
reg vout;

parameter real gain = 5e5;
parameter real fnom = 2.5e5;
parameter real vc = 2.5;

real frequency;
real period;

always
  begin
  frequency = fnom + (V(Ina) - V(Inb) - vc) * gain;
  if (frequency > 0.0)
        period = 1/frequency;
  else
    period = 1/fnom;
  #(period/2) vout = 1'b1;
  #(period/2) vout = 1'b0;
  end

endmodule
```

Note that there are Verilog-AMS extensions, even within the always block. The values of the input quantities are found using the V() function

call and the wait statements take real numbers, not time units.

The counter is purely digital, although we will use it in an asynchronous way. This is a SystemVerilog model - we are assuming the simulator can accept a mixture of languages and versions.

```
module counter #(parameter N = 4)
                (output reg count,
                 input  clk);

integer cnt = 0;

always_ff @(posedge clk)
    begin
    cnt++;
    if (cnt == N)
      begin
      cnt = 0;
      count <= 1'b1;
      end
    else
      count <= 1'b0;
    end

endmodule
```

Finally, we can put all the parts together and include a suitable stimulus.

```
`include "disciplines.vams"
`timescale 1 ns / 100 ps

module pll;

electrical up, down, up_a, down_a;
wire up_d, down_d, VCO_out, VCO_div;

reg ref_clk;

initial
  begin
  ref_clk = 1'b0;
  forever
    #5000 ref_clk = ~ref_clk;
  end
```

```
phase_detector P0 (.inA(ref_clk), .inB(VCO_div), .qA(up_d), .qB(do
dac D0 (.Din(up_d), .Aout(up_a));
dac D1 (.Din(down_d), .Aout(down_A));
lpf #(50e-6) L0 (.Ai(up_a), .Ao(up));
lpf #(50e-6) L1 (.Ai(down_a), .Ao(down));
vco #(.gain(1e5), .fnom(8e5), .vc(2.5)) V0 (.Ina(down), .Inb(up),
counter #(5) C0 (.clk(VCO_out), .count(VCO_div));
```

**endmodule**

Simulation of this PLL model shows that the output frequency varies between about 450kHz and 600kHz, before settling at 500kHz after about 260$\mu$s. The clock has a frequency of 100kHz and the counter counts to 5, so the PLL behaves exactly as we would expect.

## 14.5 Verilog-AMS simulators

It could be argued that the mixed-signal models of ADCs, DACs and PLLs could be modelled entirely in standard Verilog. Indeed, there is a very limited amount of behaviour that requires an analogue solver in these models. The real power of Verilog-AMS is that it allows digital Verilog models to be simulated at the same time as analogue circuits that would traditionally have been simulated with SPICE. For several reasons, it is appropriate to bring the discussion to a close.

## Summary

At some point, digital circuits have to interface with the real, analogue world. Modelling this interface and the interaction with analogue components has always been difficult. Verilog-AMS extends Verilog to allow analogue and mixed-signal modelling. Typical converters include ladder DACs, flash ADCs, delta-sigma ADCs and PLLs. All of these components can be modelled and simulated in Verilog-AMS. There is, as yet, no way to automatically synthesize such elements from a behavioural description. Verilog-AMS simulators are still relatively new and may not support the entire language. They do, however, provide means for interfacing between SPICE models and Verilog-AMS, allowing modelling of complete systems.

## Further reading

For an explanation of analogue simulation algorithms, see Litovski and Zwolinski. Horowitz and Hill is an excellent guide to practical circuit design and includes descriptions of ADCs, DACs and PLLs. For a full description of Verilog-AMS, the language reference manual is, of course, invaluable. Manufacturers' manuals need to be read with the LRM to understand any limitations.

## Exercises

14.1 An inductor is described by the equation $v_L = L \cdot \frac{di_L}{dt}$. Write a Verilog-AMS model of an inductor, using the ddt() function.

14.2 Write an inductor model that uses the integ() function.

14.3 Write a parameterizable model of a voltage source that generates a ramp. The parameters should be: initial voltage, final voltage, delay before the ramp and rise (or fall) time.

14.4 Write a model of voltage source that generates a pulse. What parameters need to be specified? How is it made to repeat?

14.5 Write a Verilog-AMS model of the flash ADC shown in Figure 14.1.

# Bibliography


[1] Standard for systemverilog - unified hardware design, specification, and verification language. *IEC 62530:2007 (E)*, pages 1–668, 2007.

[2] M. Abramovici, M.A. Breuer, and A.D. Friedman. *Digital System Testing and Testable Design (Revised Printing)*. IEEE Press, 1990.

[3] J. Bergeron. *Writing Testbenches Using SystemVerilog*. Springer-Verlag New York Inc., rev. ed edition, 2006.

[4] S. Brown and Z. Vranesic. *Fundamentals of Digital Logic with Verilog Design*. McGraw-Hill Science/Engineering/Math, 2nd ed edition, 2007.

[5] G. de Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill International, 1994.

[6] M.D. Edwards. *Automatic Logic Synthesis Techniques for Digital Systems*. MacMillan Press Ltd., 1992.

[7] R.W. Hamming. *Coding and Information Theory*. Prentice-Hall, 1980.

[8] J.L. Hennessy and D.A. Patterson. *Computer Architecture a Quantitative Approach*. Morgan Kaufman Publishers Inc., 1990.

[9] F.J. Hill and G.R. Peterson. *Computer Aided Logical Design with Emphasis on VLSI (Fourth Edition)*. John Wiley & Sons, Inc., 1993.

[10] K. Kundert and O. Zinke. *The Designer's Guide to Verilog-AMS*. Kluwer Academic Publishers, 2004.

[11] V. Litovski and M. Zwolinski. *VLSI Circuit Simulation and Optimization*. Chapman and Hall, 1997.

[12] A.B. Maccabe. *Computer Systems : Architecture, Organization and Programming*. Richard D. Irwin, Inc., 1993.

[13] C. Maunder. *The Board Designers Guide to Testable Logic Circuits*. Addison Wesley Publishers Ltd., 1992.

[14] A. Miczo. *Digital Logic Testing and Simulation*. John Wiley and Sons, 1987.

[15] Z. Navabi. *VHDL Analysis and Modeling of Digital Systems*. McGraw-Hill, Inc., 1993.

[16] M.S. Nixon. *Introductory Digital Design : a_programmable_approach*. MacMillan Press Ltd., 1995.

[17] S. Palnitkar. *Verilog HDL: A Guide in Digital Design and Synthesis*. Prentice Hall, 2 edition, 2003.

[18] D. R. Smith and P. D. Franzon. *Verilog Styles for Synthesis of Digital Systems*. Prentice Hall, 2000.

[19] D.J. Smith. *HDL Chip Design*. Doone Publishing, 1996.

[20] C. Spear. *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*. Springer-Verlag New York Inc., 2nd ed edition, 2008.

[21] S. Sutherland, S. Davidmann, and P. Flake. *SystemVerilog for Design: A Guide to Using Systemverilog for Hardware Design and Modeling*. Springer-Verlag New York Inc., 2nd ed edition, 2006.

[22] S.H. Unger. Hazards, critical races, and metastability. *IEEE Transactions on Computers*, 44(6):754–768, 1995.

[23] S. Vijayaraghavan and M. Ramanathan. *A Practical Guide for SystemVerilog Assertions*. Springer-Verlag New York Inc., 2005.

[24] J.F. Wakerley. *Digital Design Principles and Practices (Second Edition)*. Prentice-Hall, Inc., 1994.

[25] N.H.E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design : a Systems Perspective (Second Edition)*. Addison-Wesley Publishing Company, 1992.

[26] M. Weyerer and G. Goldemund. *Testability of Electronic Circuits*. Carl Hanser Verlag, Prentice Hall International, 1992.

[27] B.R. Wilkins. *Testing Digital Circuits*. Van Nostrand Reinhold (UK), 1986.

[28] B. Wilkinson. *Digital System Design (Second Edition)*. Prentice-Hall International, 1992.

[29] W. Wolf. *Modern VLSI Design A Systems Approach*. Prentice Hall International, 1994.

[30] . *HP Boundary-Scan Tutorial and BSDL Reference Guide*. Hewlett-Packard Company, 1990.