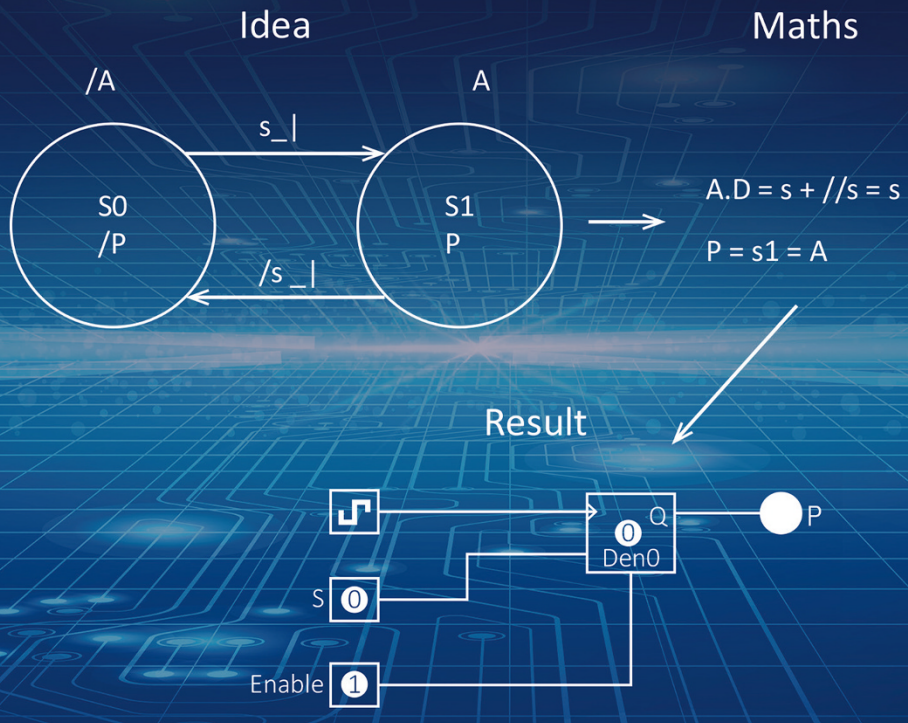
PETER D. MINNS

DIGITAL SYSTEM DESIGN USING FSMs

A PRACTICAL LEARNING APPROACH



with website

WILEY

Digital System Design using FSMs

Digital System Design using FSMs

A Practical Learning Approach

Peter D. Minns
Formerly at Northumbria University
Newcastle upon Tyne, UK

WILEY

This edition first published 2021

Registered Offices
John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, USA
John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, UK

Editorial Office
The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, UK

For details of our global editorial offices, customer services, and more information about Wiley products visit us at www.wiley.com.

Wiley also publishes its books in a variety of electronic formats and by print-on-demand. Some content that appears in standard print versions of this book may not be available in other formats.

Library of Congress Cataloging-in-Publication Data

Name: Minns, Peter D., author.
Title: Digital system design using FSMs : a practical learning approach / Peter D. Minns.
Description: Hoboken, NJ : Wiley, 2021. | Includes bibliographical references and index.
Identifiers: LCCN 2021015256 (print) | LCCN 2021015257 (ebook) | ISBN 9781119782704 (hardback) | ISBN 9781119782711 (adobe pdf) | ISBN 9781119782728 (epub)
Subjects: LCSH: Sequential machine theory. | Digital electronics.
Classification: LCC QA267.5.S4 M56 2021 (print) | LCC QA267.5.S4 (ebook) | DDC 621.381501/51135–dc23
LC record available at https://lccn.loc.gov/2021015256
LC ebook record available at https://lccn.loc.gov/2021015257

Cover Design: Wiley
Cover Image: (inset) Image by Peter Minns, (background) © Govindanmarudhai/Getty Images

Set in 10.5/13pt Times by Straive, Pondicherry, India

10 9 8 7 6 5 4 3 2 1

Contents

Preface

This book is, in large part, a development of *FSM-Based Digital Design using Verilog HDL* (Minns and Elliott 2008), a book I wrote with Ian Elliott. It is rather unusual in that it forms a linear programmed learning text in all chapters to help readers learn on their own.

The intention in this current version is to make use of programmed learning methods in which the chapters are made up of frames that must be read in a sequential manner. It is hoped that the book will help readers in their study of the material. There is also new content in Chapter 6, Appendix A5, and Appendix A6, as well as consideration of unused states in finite state machines (FSMs).

It is assumed that the reader has a good understanding of Verilog HDL; however, the interested reader will find that Chapters 6, 7, and 8 of Minns and Elliott (2008) provide a very good account of Verilog HDL. Wiley make it possible to purchase these chapters on request for a small fee.

Note that in this version of the book the reader is given help to assist them as they progress through this book.

Indeed, Chapters 3, 4, 5, and 7 as well as some of the appendices include examples of FSMs with Verilog HDL for illustrated examples. Use is also made of the Digital logic simulation program **Logisim** to help the reader become familiar with using FSMs in the development of their work. This **Logisim Simulator** is freely available throughout the world to run on Windows, OS X, and Linux Operating Systems (see Appendix A2 for details).

The chapters are organized as follows.

Chapter 1 covers the introductory ideas of what FSMs are and how to represent them using a state diagram.

Chapter 2 covers the use of external devices and how to control them with an FSM.

Chapter 3 looks at how to synthesize FSMs using *T* type flip-flops, then *D* type flip-flops.

Chapter 4 introduces asynchronous FSM design.

Chapter 5 looks at the use of the one hot method of synchronous FSM design applied to clocked FSM designs.

Chapter 6, a new chapter, looks at applying an FSM to event-driven systems, and considers one hot ideas and the one hot method.

Chapter 7 deals with Petri nets and how they can be used to synthesize electronic circuits using both sequential and parallel state machine design. This allows FSM-based systems to support both sequential and parallel structures.

There are six appendices covering the necessary aspects of FSM systems that the reader needs to understand in the support of the FSM work. These are written in a more formal manner (i.e. not using frame method or programmed learning).

Appendix A1 looks at the logic gates and Boolean algebra used in this book. This should help those readers who may not have done much work on Boolean algebra for some time.

Appendix A2 is a tutorial on how to use the simulation programs and the Verilog Hardware Descriptive Language (HDL) with the SynaptiCAD system as well as a short introduction on the use of the gate logic simulator **Logisim**.

Appendix A3 covers the use of counters and shift registers as used in a number of the tutorials in this book. The reader should find them very useful.

Appendix A4 covers the use of behavioural Verilog HDL with some examples to help the reader become familiar with its use in the design of FSM-based systems.

Appendix A5 looks at the way an FSM can be designed using digital hardware that can be programmed to produce programmable FSM systems. Tutorial examples are introduced to illustrate how this works.

Appendix A6 looks at how a rotation direction indicator can be implemented using an event-driven FSM.

This book provides enough information for the reader to learn how to design their own FSMs and simulate them using the hardware descriptive language Verilog HDL.

I hope that the content of this book is both interesting and useful, and that it helps readers to learn more about digital system design using FSMs. I have used these techniques for many years, in lectures and when working with companies, and have found them really helpful.

Peter Minns BSc(h) PhD CENG MIET (retired)

Access to Wiley Web for my Verilog HDL and Logisim files:
www.wiley.com\go\minns\digitalsystemdesign

Acknowledgements

I would like to thank all those who helped in the proofreading of this book, in particular those who have looked at specific chapters and especially Kathleen Minns for her help in the proofreading of the entire manuscript.

Also thanks to Ian Elliott for introducing the Logisim digital simulation program.

Special thanks go to my commissioning editor Sandra Grayson and managing editor Juliet Booker for their kind support and guidance in the preparation of this book as well as the Wiley publishing team.

Thanks are also extended to: Dr C. Burch – Logisim, IET Digital Library, Oxford Publishing Limited, and Sage Publishing, USA for their various permissions, and these are referred to in the relevant sections of the book.

Any errors are, of course, entirely the responsibility of the author.

About the Companion Website

This book is accompanied by a companion website

www.wiley.com/go/minns/digitalsystemdesign 

This website includes:

- An Index for Logisim Circuits
- An Index for Verilog Listings
- Guide to Supplementary Resources
- Verilog HDL
- Logisim Files

Guide to Supplementary Resources

The reader will be aware that this book is supported by working Logisim circuits and Verilog listings and these can be accessed via the Wiley website locator.

However in order to be able to access these Logisim and Verilog circuits/listings the reader needs to download two programs namely Logisim and SynaptiCAD Verilog HDL from the internet and install them on their computer (either PC or MAC) as explained in Appendix 2 of this book.

Each of the Logisim and Verilog HDL folders on the Wiley website has an individual index that shows the appropriate chapter/appendix reference locator to which they refer.

Once the appropriate software has been installed the reader will need to click onto either the .circ or .v files located in 'Logisim Circuits for Web Folder' or 'Verilog Listings for Web Folder' which they wish to access.

In both folders there is additional material and ideas that do not appear in this book but maybe of interest to the reader.

1

Introduction to Finite State Machines

This chapter (like all other chapters) is written in the form of a linear frame, programmed learning text. This is to help you learn the basic skills required to design clocked finite state machines (FMSs) so that you can develop your own designs based on traditional *T* flip-flops and *D* flip-flops. Later, other techniques will be introduced, such as 'one hot' and 'asynchronous finite state machines', but these will be developed along the same lines as the work covered in this chapter.

The text is organized into 'frames'. Each frame follows on consecutively from the previous one, but at times you may be redirected to other frames, depending upon your response to the questions you are asked. Do not cheat, but follow the frames as indicated.

1.1 SOME NOTES ON STYLE

Bold denotes **questions for you to answer** to check your understanding of the material, highlights **important points**, or indicates an **aside when further ideas are presented**.

Please read this chapter first, and attempt all the questions before moving on to the later chapters. Note that the book can be read as a textbook. The programmed aspect of the book makes it more suitable for individuals to read and learn in their own time.

Digital System Design using FSMs: A Practical Learning Approach, First Edition. Peter D. Minns.

Companion website: www.wiley.com/go/minns/digitalsystemdesign

Frame 1.1 What is a Finite State Machine?

A finite state machine (FSM) is a digital sequential circuit that can follow a number of predefined states under the control of one or more inputs. Each state is a stable entity that the machine can occupy. It can move from this state to another state under the control of an outside world input.

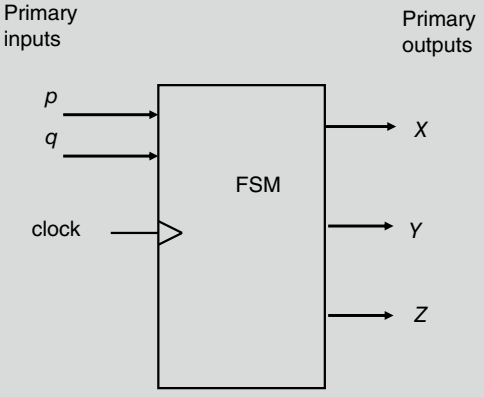


Figure 1.1 Block diagram of an FSM-based application.

In Figure 1.1, we see an FSM with three outside world inputs (*p*, *q*, and the clock) and three outside world outputs (*X*, *Y*, and *Z*). Note some FSMs have a clock input; those that don't belong to a type of FSM called 'asynchronous FSM'. However, this chapter deals with the more usual synchronous FSM, which **do** have a clock input. Only Chapter 4 and Chapter 6 will look at asynchronous FSM.

Synchronous FSM can move between states only if a clock pulse occurs.

Task: **Draw a block diagram for an FSM with five inputs (*x*, *y*, *z*, *t*, and a clock) and with two outputs (*P* and *Q*).**

When you have done this, turn to **Frame 1.2**.

Frame 1.2

The FSM with five inputs (*x*, *y*, *z*, *t*, and a clock) and two outputs (*P* and *Q)* is shown in Figure 1.2.

If you did not get this answer, go back and re-read Frame 1.1. Don't worry about using a mixture of both upper- and lower-case letters here; the only thing that matters is that the same letters are used.

Each state of the FSM needs to be identifiable. This is achieved by using a number of internal flip-flops within the FSM block. An FSM with four states would require two flip-flops since two flip-flops can store $2^2 = 4$ state numbers.

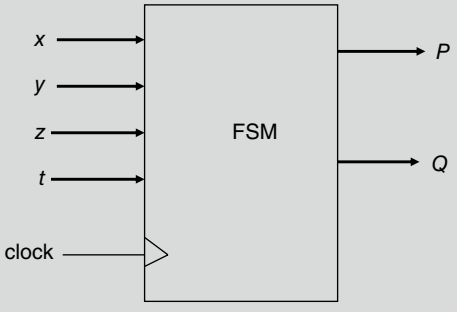


Figure 1.2 Block diagram with five inputs and two outputs.

Each state has a unique state number, and states are usually assigned numbers, such as s0 (state 0), s1, s2, and s3 (for a four-state example).

As you can see, the rule here is $2^{\text{number of flip-flops}}$.

So an FSM with 13 states would require 2^4 flip-flops (i.e. 15 states of which 13 are used in the FSM and states 14 and 15 remain unused.

How many flip-flops would be required for an FSM using 34 states?

What would the state numbers be for this FSM?

When you have answered these questions, turn to **Frame 1.3**.

Frame 1.3

The answer to the previous question is:

$$2^6 = 64, \text{which would accommodate 34 states.}$$

In general: $2^4 = 16$ states, $2^5 = 32$ states, $2^6 = 64$ states, $2^7 = 128$ states, and so on.

What would the state number be for this FSM?

s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11, s12, s13, s14, s15, s16, s17, s18, s19, s20, s21, s22, s23, s24, s25, s26, s27, s28, s29, s30, s31, s32, s33.

Answer:

The unused states would be s34 through to s63.

Note that the states run through from s0 to s*n*−1, for *n* states.

As well as containing flip-flops to uniquely define the individual states of the FSM, there is also combinational logic, which defines the outside world outputs. In addition, the outside world inputs connect to combinational logic, which supplies the flip-flops' inputs.

Please turn to **Frame 1.4**.

Frame 1.4

Figure 1.3 illustrates the internal architecture for a Mealy FSM.

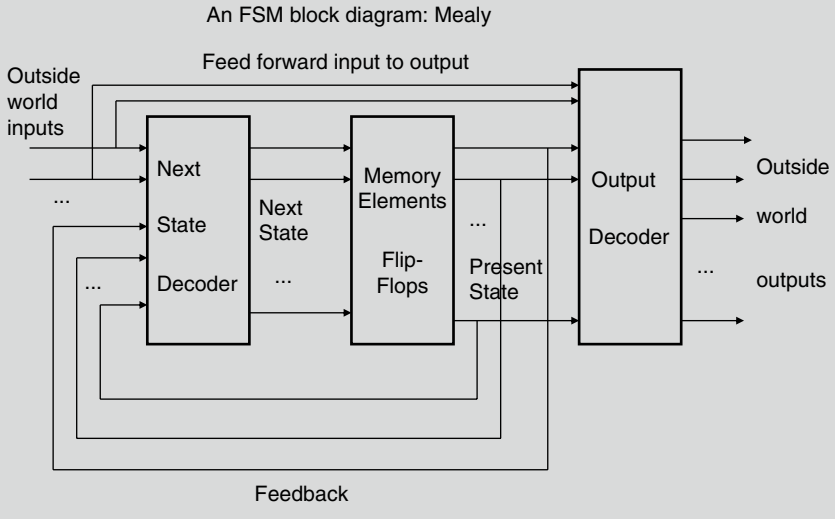


Figure 1.3 Block diagram of a Mealy state machine structure.

Note the feed forward paths between the outside world inputs and the input to the output decoder.

The figure shows that the FSM has a number of inputs that connect to the **next state decoder** (combinational) logic. The *Q* outputs of the memory element flip-flops connect to the **output decoder** logic, which in turn connects to the outside world outputs via the output decoder.

The flip-flop outputs are used as **next state** inputs to the **next state decoder**, and it is these that determine the next state that the FSM will move to. Once the FSM has moved to this **next state**, its flip-flops acquire a new **present state** as dictated by the **next state decoder**.

Note that some of the **outside world inputs** connect **directly** to the **output decoder** logic. This is the main feature of the **Mealy** type of FSM. This affects the outputs of the FSM.

Please turn to **Frame 1.5**.

Frame 1.5

Another architectural form for an FSM is the **Moore** FSM, as shown in Figure 1.4.

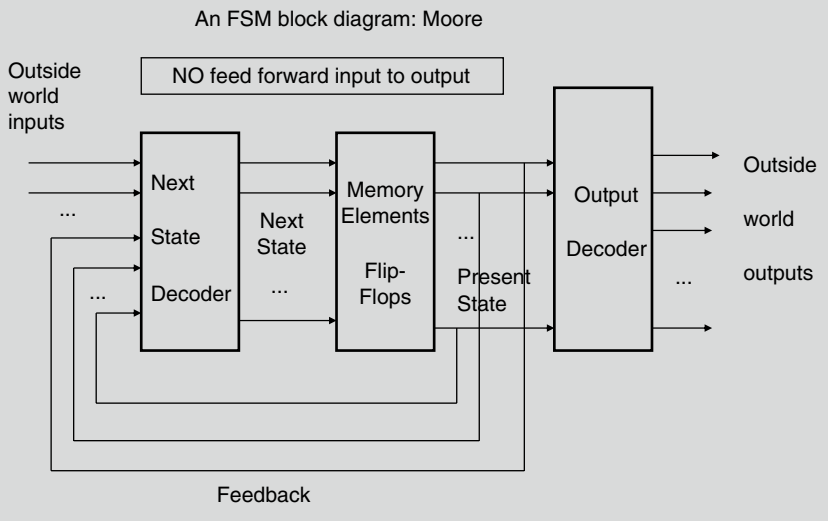


Figure 1.4 Block diagram of a Moore state machine structure.

This FSM differs from the **Mealy** FSM in that it does not have the feed forward paths.

This type of FSM is very common. Note that the **outside world outputs** are a function of the flip-flop **outputs** only (unlike the **Mealy FSM** architecture where the **outside world outputs** are a function of **flip-flop outputs** and **some outside world inputs**).

We will be using **both Moore and Mealy FSM** in our designs.

Please turn to **Frame 1.6**.

Frame 1.6

Complete the following:

- **A Moore FSM differs to that of a Mealy FSM in that it has…**
- **This means that the Moore FSM outputs depend on…**
- **Whilst the Mealy FSM outputs can depend upon…**

If you cannot complete the above sentences, go back and read Frame 1.4 and Frame 1.5.

When you have completed these questions, please go to **Frame 1.7**.

Frame 1.7

If we look at the Moore FSM architecture again and remove all of the **outside world inputs** apart from the **clock**, and we also remove the **output decoding logic**, we are left with a very familiar architecture. This is shown in Figure 1.5.

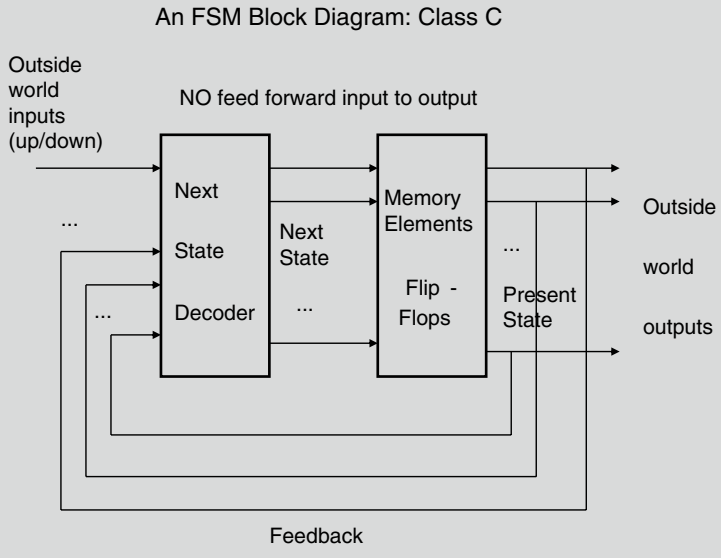


Figure 1.5 Block diagram of a Class C state machine structure.

This architecture is in fact the synchronous counter the reader may have already seen in previous studies. Note that an **up/down** counter would have the additional outside world input 'up/down', which would be used to control the direction of counting.

The flip-flop outputs in this architecture are used to connect directly to the outside world.

Please move on to **Frame 1.8**.

Frame 1.8

Historically, two types of state diagrams have evolved, one for the design of the **Mealy** FSM the other for the design of the **Moore** FSM. The two are known as 'Mealy state diagrams' and 'Moore state diagrams'.

These days we use a more general type of state diagram, which can be used to design both the **Mealy** and **Moore** type of FSM. This is the type of state diagram

we use throughout this book. **As you will learn, it allows you to build a lot of ideas into the FSM diagram.**

Figure 1.6 shows each state of the FSM and the transitions to and from that state to other states.

The states are usually drawn as **circles** (but some people like to use a **square box**).

The **transitions** between states are shown as an **arrowed line** connected between the states.

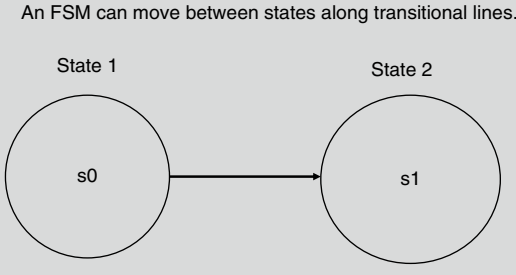


Figure 1.6 Transition between states.

In addition to the transitional line between states, there is an **input** signal name.

The right-angled lines _| represent the clock input (in this case a rising edge 0 to 1) **(Figure 1.7).**

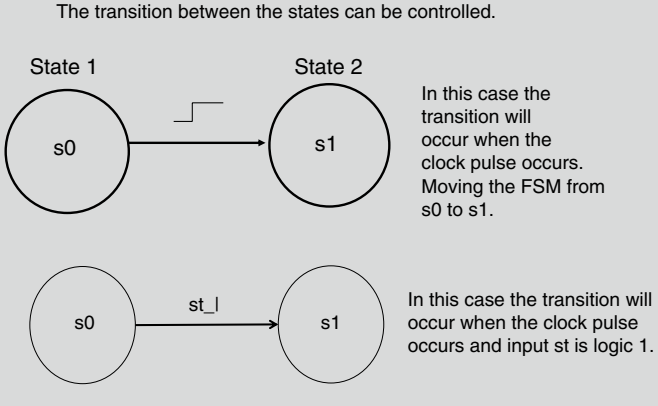


Figure 1.7 Transition with and without outside world inputs.

In Figure 1.7, the transition between states s0 and s1 will occur at the clock pulse in the upper state diagram, while in the lower state diagram it will **only** occur if the **outside world input set to 1** 'st = 1' **and** a '0 to 1' transition occurs on the clock input.

What changes would be needed to Figure 1.7 to make the transition between s0 and s1 occur when input st = 0?

Turn to **Frame 1.9** after you have attempted this question.

Frame 1.9

The answer is shown in Figure 1.8.

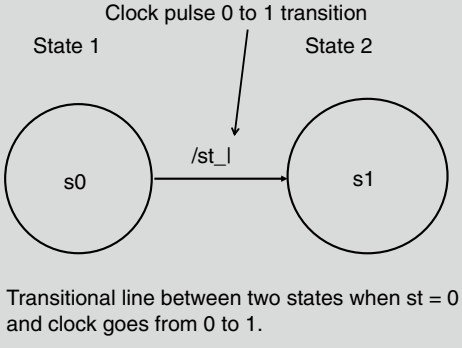


Figure 1.8 Outside world input between states.

Since in this case the outside world input 'st' must be equal to zero (denoted by the inverting bar to the left of the input st (as in /st).

That is / means **not** so /st means **not st**, i.e. when st = 0, then /st = 1.

Note that outside world inputs always lie along the transitional lines. Also, the reader could be using '·' as well as '*' for 'AND'. Also, '+' for 'OR' in Boolean equations; however, in most cases '.' will be used rather than '*'. In some cases no symbol will be used for 'AND', as in 'AB' to mean 'A·B'.

The state diagram must also show how the 'outside world outputs' are affected. This is achieved by placing the outside world outputs either:

- **inside the state circle (or square); or**
- **alongside the state circle (or square).**

Figure 1.9 shows the outside world outputs *P* and *Q* inside the state circles. In this particular case, *P* is logic 1 in state s0, and changes to logic 0 when the FSM moves to state s1. Output *Q* does not change in the above transaction, remaining at logic 0 in both states.

Draw a block diagram showing inputs and outputs for the state diagram.

Then turn to **Frame 1.10**.

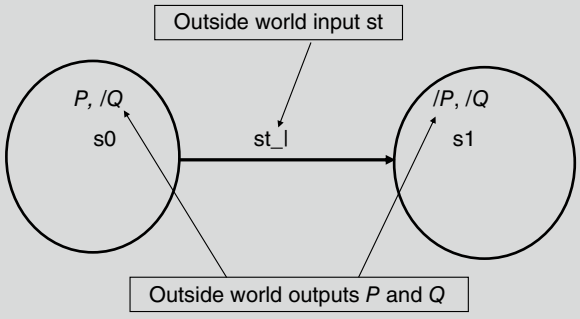


Figure 1.9 Placement of outside world outputs.

Frame 1.10

The block diagram will look like that shown in Figure 1.10.

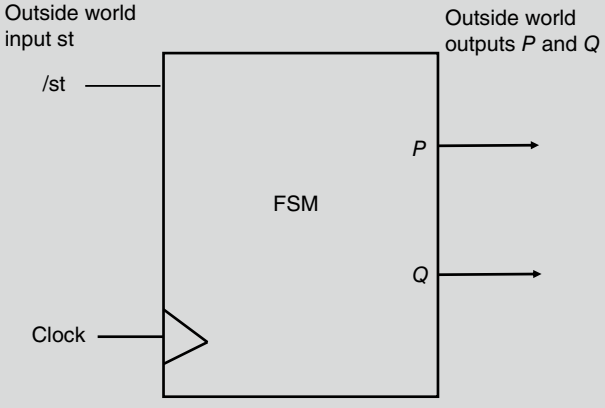


Figure 1.10 The block diagram for the state diagram shown in Figure 1.9.

Sometimes we show a negating circle to imply that the input is actually inverted (see later).

It is easily obtained from the state diagram since inputs lie along transitional lines and outputs lie inside (or alongside) the state circle. The input st would normally have a **negating circle** to show it is an **active low** input. This is common practice.

You may remember that in Frame 1.2 we said that each state had to have a unique state number and that a number of flip-flops were needed to perform this task. These flip-flops are part of the internal design of the FSM and are used to produce an internal count sequence; they are essentially acting like a synchronous counter, but one that is controlled by the outside world inputs. The internal count sequence produced by the flip-flops is used to control the outside world decoder so that outputs can be turned on and off as the FSM moves between states.

In Frames 1.4 and 1.5 we saw the architecture for the Mealy and Moore FSM. In both cases, the memory elements shown are the flip-flops discussed in the previous paragraph. We look at how the internal flip-flops are coded in a later chapter.

At this stage it is perhaps worth looking at a simple FSM design in detail. We can then bring together all the ideas discussed so far, as well as introducing a few new ones. Try answering the following questions before moving on:

1. **A Mealy FSM differs from a Moore FSM in? (See Frames 1.4 and 1.5.)**
2. **The circles in a state diagram are used to? (See Frames 1.8 and 1.9.)**
3. **Outside world inputs are shown in a state diagram where? (See Frames 1.8 and 1.9.)**
4. **Outside world outputs are shown where? (See Frame 1.9.)**
5. **The internal flip-flops in an FSM are used to do what? (See Frame 1.10.)**

Please turn to **Frame 1.11**.

Frame 1.11

Figure 1.11 shows an example of a single-pule circuit FSM.

The idea here is to develop a circuit based on the FSM that will produce a single output pulse at its output *P* whenever its input *s* is taken to logic 1. The FSM is to be clock driven so it also has an input clock. An additional output *L* is used to indicate that a *P* pulse has been produced so the user can see effect of 'fast' pulses.

The block diagram of this circuit is shown in Figure 1.11.

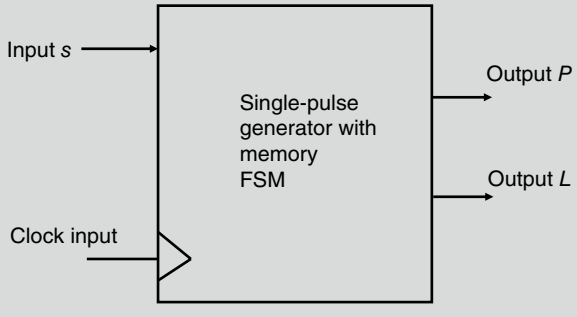


Figure 1.11 Block diagram of single pulse with memory FSM.

Figure 1.12 shows a suitable state diagram.

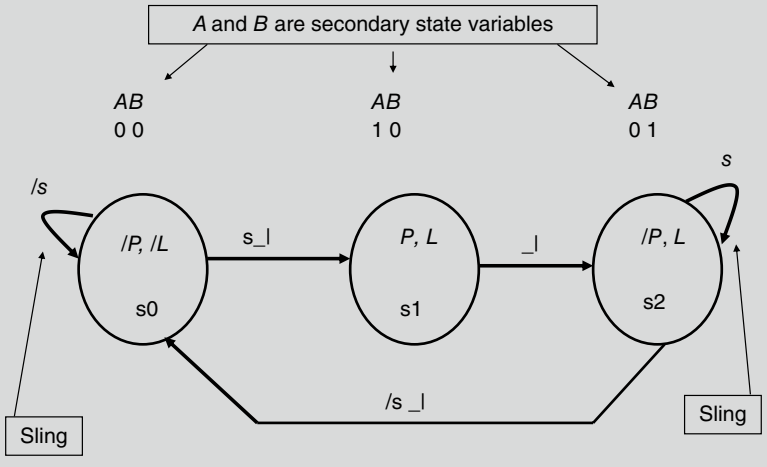


Figure 1.12 State diagram for single pulse with memory FSM.

In this state diagram the **sling** (loop */s* going to and from s0) indicates that while input *s* is logic 0 (*/s*) the FSM will remain in state s0 regardless of how many clock pulses are applied to the FSM. Only when input *s* goes to logic 1 will the FSM move from state s0 to s1, and then only when a clock pulse arrives. Once in state s1, the FSM will set its output *P* to logic 1, and on the next clock pulse the FSM will move from state s1 to s2.

The reason why the FSM will stay in state s1 for only one clock pulse is because in state s1 the transition from this state-to-state s2 occurs **on a clock pulse only**. Once the FSM arrives in state s2, it will remain there whilst input *s* = 1. As soon as input *s* goes to logic 0 (*/s*) the FSM will move back to state s0 **on the next clock pulse**.

Since the FSM remains in state s1 for only a single clock pulse, and since *P* = 1 only in state s1, the FSM will produce a single output pulse.

Note in the FSM state diagram that each state has a unique state identity: s0, s1, and s2.

Also note that each state has been allocated a unique combination of flip-flop states, for example:

- State s0 uses the flip-flop combination *A* = 0 *B* = 0, e.g. both flip-flops reset.
- State s1 uses the flip-flop combination *A* = 1 *B* = 0, e.g. flip-flop *A* is set.
- State s2 uses the flip-flop combination *A* = 0 *B* = 1, e.g. flip-flop *A* is reset, flip-flop *B* is set.

Now move on to **Frame 1.12**.

Frame 1.12

Let's continue with the one-pulse design.

The flip-flop outputs are seen to define each state. If we could see nothing more than the *A* and *B* outputs of the two flip-flops, we could tell what state the FSM was in by the output logic levels on each flip-flop.

We could also tell in which state the output *P* was to be logic 1, i.e. in state s1 where the flip- flop output logic levels are $A = 1$ and $B = 0$.

Therefore, the output $P = A/B$. (Remember, *AB* is used to indicate the logical AND operation used in Boolean algebra.)

So we now see that the flip-flops are used to provide a unique identity for each state.

We also see that, since each state can be defined in terms of the flip-flop output states, the outside world outputs can also be defined in terms of the flip-flop output states since the outside worlds output states themselves are a function of these states.

L is logic 1 in states s1 and s2 and is defined in terms of the flip-flop outputs $A/B + /AB$.

Therefore, $L = A/B + /AB = A/B + /AB$. No Boolean reduction is possible in this case.

The allocation of unique values of flip-flop outputs **to each state** is rather an arbitrary process. In theory, we can use any values so long as **each state has a unique combination**. This means that we cannot have more than one state with the flip-flop values of, say, A/B (i.e. both states cannot have the same value).

In practice it is common to assign flip-flop values so that the transition between each state involves **only one flip-flop changing state**. This is known as 'following a **unit distance pattern**': only one flip-flop changes state.

The above example does not use a unit distance pattern since there are two flip-flop changes between states s1 and s2. However, the reader will be going on to make use of the unit distance code idea.

The reader could also make the single-pulse state diagram (Figure 1.12) follow a **unit distance pattern** by adding an extra state. This **extra state** could be inserted between states s2 and s0, having the same output for *P* as state s0. In the state diagram the new state would also have the value of *L*, the same as that in state s2, since the reader does not want *L* to change until *s* goes to 0.

Try re-drawing the state diagram with this additional state and assign a unit distance pattern to the flip-flops.

When you have done this, go to **Frame 1.13**.

Frame 1.13

A completed state diagram with unit distance patterns for flip-flops is shown in Figure 1.13.

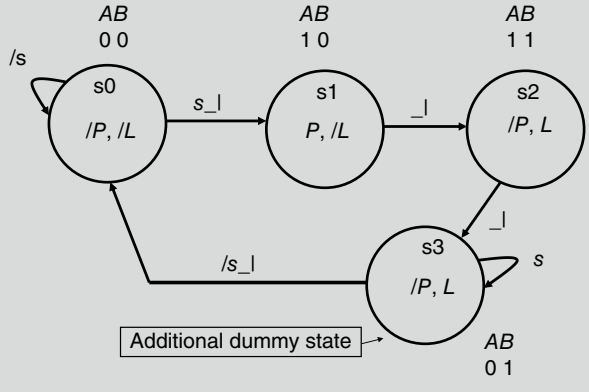


Figure 1.13 State diagram for single-pulse generator with memory and dummy state.

Note that the added state has the unique name of s3 and the unique flip-flop assignment of $A = 0$ and B = 1.

Also note that s2 uses the *A* and *B* values of $A = 1$ and $B = 1$. This provides the required unit distance coding. It also has the output $P = 0$, as it would in state s0 (the state it is going to go to when $s = 0$).

In this design the addition of the extra state has not added any more flip-flops to the design since two flip-flops can have a maximum of $2^2 = 4$ states (remember Frames 1.2 and 1.3).

The addition of this extra state is usually called a **dummy state**.

Look carefully at the state diagram in Frame 1.13 and satisfy yourself that the state diagram is doing the same thing as the one in Frame 1.11. If you cannot see this, consider reading Frames 1.11–1.13 again.

Now let us add an additional input called r to our state diagram.

Input r is to be added so that if $r = 1$ the FSM will continue to pulse output P (on and off) until r is made 0. At this point the FSM will return to state s0 but only if input $s = 0$.

Draw the block diagram for the FSM.

Draw the state diagram for this modified FSM.

Take your time and think about what you are doing.

Turn to **Frame 1.14** when you have completed this task.

Frame 1.14

The block diagram in Figure 1.14 is changing the behaviour of the state diagram.

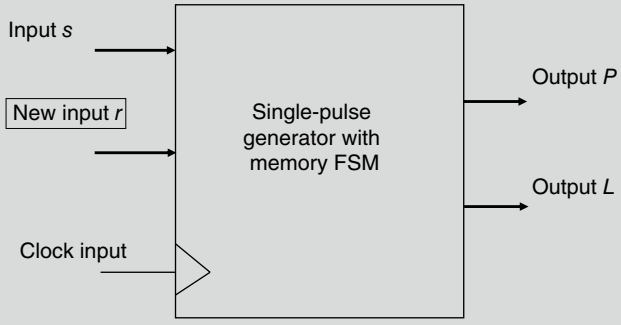


Figure 1.14 Block diagram for the FSM.

The state diagram is shown in Figure 1.15.

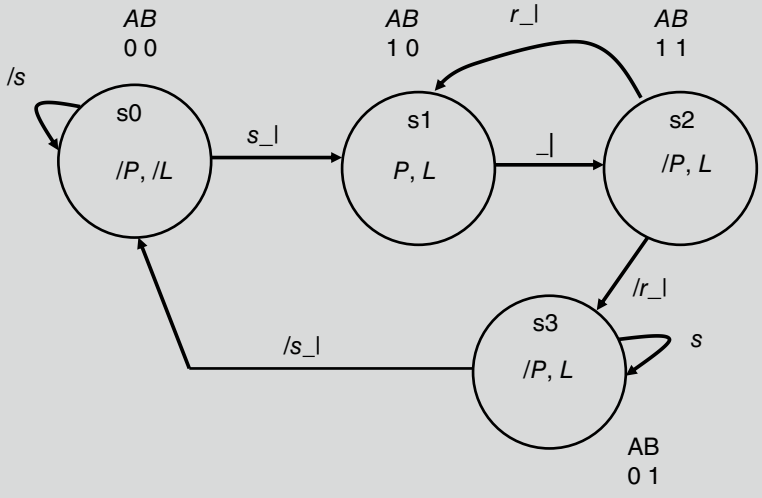


Figure 1.15 State diagram of single-pulse generator with a multipulse feature.

The new state diagram is essentially the same as that in Frame 1.13 except that now if $r = 1$ the state diagram sequence (with $s = 1$) becomes s0 → s1 → s2 → s1 → s2, and so on.

However, if $s = 1$ and $r = 0$ the sequence will be s0 → s1 → s2 → s3, which then stops until $s = 0$ then goes to s0.

From this you should understand that if we make $r = 0$ before we make $s = 1$ the state diagram will follow a single pulse on output P.

The Boolean equation for L can be active high or active low.

As active high, we look for states when the value of L is 1.

$$L = \text{s1} + \text{s2} + \text{s3} = A/B + \text{AB} + /AB = A \cdot (/\text{B} + \text{B}) + \text{B}(A + /A) = A + \text{B}.$$

The Boolean equation for P was $P = \text{s1} = A/B$ (see Frame 1.12).

The Boolean equation for $L = \text{s1} + \text{s2} + \text{s3} = A + \text{B}$ since only in states s1, s2 and s3 is the output $L = 1$.

Note that an alternative equation for L could be the **inverse** equation for L, otherwise known as 'active low'. Here we look for the states that make $L = 0$.

$$/L = /\text{s0} = /(/A/B) \text{ which is a NAND gate.}$$

In practice there is a tendency to show this active low output as:

$$L \text{ (active low)} = /(/A/B).$$

Note that to obtain the $L = 0$ the reader needs to invert s0 (/s0). This idea will be used later.

This is less complex so it may be used.

The latter equation is in terms of NOT L. This means that when in state s1, s2, or s3, L will be logic 1. Only when the FSM is NOT in any of these states will $L = 0$.

So in summary:

- Input r is used with a two-way branch from state s2.
- If $r = 0$ there is no change in the operation of the FSM (single pulse from P).
- However, if $r = 1$ the FSM will keep looping between s1 and s2 so as to keep turning the output P on and off, **in effect using input r to change the operation of the FSM.**

This shows how easily it is to change the behaviour of the FSM (in this case).

Note that this change was done in the state diagram then transferred to the actual FSM.

Please turn to Frame 1.15.

Frame 1.15

In the previous frames we have considered the flip-flop output patterns. These are often referred to as the **secondary state variables (SSVs)** (Figure 1.16).

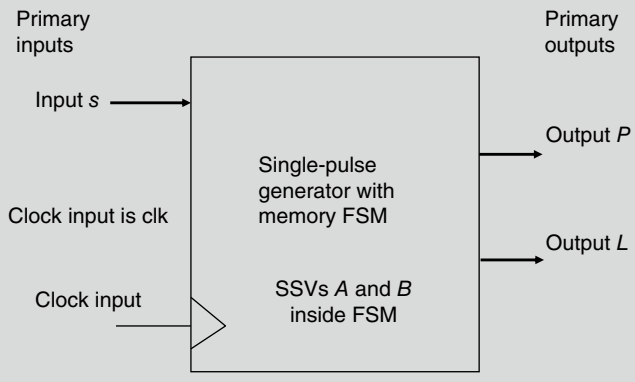


Figure 1.16 Block diagram showing secondary state variables in the FSM.

They are called 'secondary state variables' because they are (from the FSM architecture viewpoint) internal to the FSM (i.e. secondary not primary). If we consider the outside world inputs and outputs as being primary then it seems sensible to call the flip-flop outputs SSVs (and state variables because they define the states of the state machine).

Moore and Mealy state diagram

The outputs in our FSM are seen to be dependent upon the SSVs or flip-flops internal to the FSM. If you look back to Frame 1.5 you will see that Moore FSM outputs are dependent upon the flip-flop outputs only. The output decoding logic in our *P* pulse example is:

P = s1 = A/B (see Frames 1.12 and 1.13); and
L = /s0 = /($/A/B$), an active low.

That is it consists of an AND gate and a NAND gate. This means that a single *P* pulse is a Moore FSM.

How could we make our single-pulse design into a Mealy FSM?

One way would be to make the output *P* depend on the FSM being in state s1 (*A*/*B*), but we could say that the output was to be the width of a single logic 0 of the clock pulse.

How would we modify our state diagram to do this?

Try doing this, and then turn to **Frame 1.16** to find out if you got it right.

Frame 1.16

The modified state diagram is shown in Figure 1.17 (the *r* signal here has been dropped so we are back to a simple one-pulse FSM). Also, /clk is clk inverted.