Introduction to Embedded System Design Using Field Programmable Gate Arrays

Rahul Dubey

Introduction to Embedded System Design Using Field Programmable Gate Arrays

Springer

Rahul Dubey, PhD
Dhirubhai Ambani Institute of Information
and Communication Technology (DA-IICT)
Gandhinagar 382007
Gujarat
India

ISBN 978-1-84882-015-9 e-ISBN 978-1-84882-016-6

DOI 10.1007/978-1-84882-016-6

A catalogue record for this book is available from the British Library

Library of Congress Control Number: 2008939445

ChipScope™, MicroBlaze™, PicoBlaze™, ISE™, Spartan™ and the Xilinx logo, are trademarks or registered trademarks of Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124-3400, USA. http://www.xilinx.com

Cyclone®, Nios®, Quartus® and SignalTap® are registered trademarks of Altera Corporation, 101 Innovation Drive, San Jose, CA 95134, USA. http://www.altera.com

Modbus® is a registered trademark of Schneider Electric SA, 43-45, boulevard Franklin-Roosevelt, 92505 Rueil-Malmaison Cedex, France. http://www.schneider-electric.com

Fusion® is a registered trademark of Actel Corporation, 2061 Stierlin Ct., Mountain View, CA 94043, USA. http://www.actel.com

Excel® is a registered trademark of Microsoft Corporation, One Microsoft Way, Redmond, WA 98052-6399, USA. http://www.microsoft.com

MATLAB® and Simulink® are registered trademarks of The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA 01760-2098, USA. http://www.mathworks.com

Cover design: eStudio Calamar S.L., Girona, Spain

Printed on acid-free paper

9 8 7 6 5 4 3 2 1

springer.com

In memory of my father and grandmother

Preface

Overview

The realm of embedded systems is quite large and is predominantly carried out around the general purpose processor and microcontrollers. The use of field programmable gate array (FPGA) in microprocessor-based embedded systems is often for glue logic or for off-loading the processor from tasks that require fast updates. The motivation for writing this text is to present a single source of information that can be used to understand how a FPGA and the Hardware Description Language (HDL) can be used in the design of embedded digital systems.

Digital design methodology has undergone several changes over the past three decades. The use of FPGA and HDL for implementing digital logic has become widespread in the last decade. The use of FPGA in embedded systems is still in its nascent stage. The majority of the embedded applications are divided between an 8-bit microcontroller implementation and a 32-bit processor-based real time operating system (RTOS) implementation. This text provides a starting point for the design of embedded system using FPGA and HDL. To give the text a common thread of thought from the application point of view, a design example of a hypothetical industrial robot controller is taken up. Different chapters of the text provide the necessary background on FPGA and HDL along with its use in designing an industrial robot controller.

Coverage

The first FPGA, introduced in 1985, consisted of 2000 gates. Since then, gate density has grown to tens of millions of gates. With increasing density of FPGAs, varied hardware resources have become a standard feature of contemporary FPGA-based devices. The text includes simulation of digital logic using Verilog HDL, synthesis of HDL code for a given FPGA device and processor-based FPGA devices. The focus of the HDL chapter is to emphasise the synthesizable area of Verilog constructs and to provide a basis to understand the application examples that follow in subsequent chapters. A chapter is devoted to the understanding of hardware–software partitioning in a FPGA device. Proprietary 8-bit and a 32-bit soft processors are discussed along with interfacing methodology using system-on-

chip interconnections. Basic technique for serial data communication, signal conditioning, motor control and hardware prototyping is covered using FPGA and HDL.

How to Use This Book

Moore's law has kept the semiconductor business in a constant state of flux. It is very difficult to write a book that uses FPGA and continues to be relevant despite ongoing technological changes. The author has presented basic concepts and techniques for using FPGA and hence should not change quickly. Since this book covers vast areas of HDL and FPGAs, some sections are brief and sketchy. For this the author recommends that the reader supplement the contents of each chapter with additional available literature. The chapter on HDL coding and simulation should be supplemented by standard textbooks on HDL coding and simulation. The FPGA resources and synthesis topic should be supplemented by EDA tools provided by different FPGA vendors and FPGA device datasheets. The contents on FPGA embedded processors can be supplemented by application notes on interfacing processors to custom codes and datasheets of soft processors.

FPGA Device and Tools Used

For purposes of illustration and consistency, Xilinx ISE™ software and SPARTAN™3E FPGA have been used throughout the book. Though the exemplars are specific to this device, the concepts can be applied to FPGA devices available from other FPGA vendors.

Gandhinagar *Rahul Dubey*
October 2008

Acknowledgements

Many people have contributed in the process of writing this text. First and foremost, I wish to thank my PhD supervisors Professor Pramod Agarwal and Professor M.K. Vasantha at IIT Roorkee, for the training they provided during my research. Lots of impetus for being steadfast in my resolve to finish this book came from Professor S.C. Sahasrabudhe, Director DA-IICT. I wish to thank the students and faculty at DA-IICT for their help and support.

The synthesis reports attached with design examples were generated using Xilinx ISE™ software. I would like to thank Xilinx for letting me use their software tool and FPGA to demonstrate various aspects of HDL and FPGA design. I am also thankful to Doctor Parimal Patel, Xilinx, for providing valuable feedback on the text. Certain equipment, used for hardware testing of examples in the text, came through a research grant from the Department of Science and Technology of the Indian Government.

This text would not have seen the light of day without the patience and support of my family. No words can express the thoughtfulness of my wife Sulekha and daughters Aditi and Avni for enduring the extra work hours. Toward the end of the writing process, Sulekha helped out by proofreading large sections of the text. I would also wish to thank my mother and younger brother Rohit for their encouragement during the process of writing this book. My uncle Mr. Rama Shankar Dubey has been a source of strength for all these years.

I thank Mr. Oliver Jackson and Springer for giving me the opportunity to write this book. Also, I wish to thank Sorina Moosdorf and the team at le-tex for meticulously checking the final draft of the manuscript.

Contents

Abbreviations

ABEL	Advanced Boolean expression language
ADC	Analogue-to-digital converter
ANSI	American National Standards Institute
ASIC	Application specific integrated circuit
ASSP	Application specific standard product
BUFG	Global clock buffer
CAD	Computer aided design
CAN	Controller area network
CE	Clock enable
CLB	Configurable logic block
CLK	Clock signal
CMOS	Complementary metal oxide Semiconductor
CPLD	Complex programmable logic device
DAC	Digital-to- analogue converter
DCI	Digitally controlled impedance
DCM	Digital clock manager
DRAM	Dynamic random access memory
DSP	Digital signal processor
EDA	Electronic design automation
EDIF	Electronic digital interchange format
EMI	Electromagnetic interference
EPROM	Erasable programmable read only memory
FF	Flip flop
FIFO	First in first out
FIR	Finite impulse response (filter)
fMax	Frequency maximum

FPGA	Field programmable gate array
FSM	Finite state machine
GPP	General purpose processor
GPS	Global Positioning System
GPIO	General purpose I/O
GTL	Gunning transceiver logic
GTLP	Gunning transceiver logic plus
GUI	Graphical user interface
HDL	Hardware description language
HEX	Hexadecimal
HSTL	High-speed transceiver logic
I/O	Inputs and outputs
ISR	Interrupt service routine
IEEE	Institute of Electrical and Electronics engineers
ILA	Integrated logic analyzer
IOB	Input output block
IP	Intellectual property
ISA	Instruction set architecture
ISP	In system programming
JEDEC	Joint Electron Device Engineering Council
JTAG	Joint Test Advisory Group
LAN	Local area network
LC	Logic cell
LCD	Liquid crystal display
LSB	Least significant bit
LUT	Look-up table
LVCMOS	Low-voltage complementary metal oxide semiconductor
LVDS	Low-voltage differential signaling
LVPECL	Low-voltage positive emitter-coupled logic
LVTTL	Low Voltage transistor to transistor logic
MAC	Multiply and accumulate
MOSFET	Metal oxide semiconductor field effect transistors
MSB	Most significant bit
MUX	Multiplexer
NAND	Not and
NRE	Non-recurring engineering (cost)
NRZ	Non-return to zero

OE	Output enable
OTP	One time programmable
PACE	Pinout and area constraints editor
PCB	Printed circuit board
PCI	Peripheral component interconnect
PCMCIA	Personal Computer Memory Card International Association
PLC	Programmable logic controller
PI	Proportional integral
PLD	Programmable logic device
PWM	Pulse width modulation
RAM	Random access memory
RMS	Root mean square
ROM	Read only memory
SCR	Silicon controlled rectifier
SDF	Standard delay format
SOP	Sum of product
SPI	Serial peripheral interface
SRAM	Static random access memory
SRL16	Shift register LUT
SSTL	Stub series terminated transceiver logic
TTL	Transistor-transistor logic
Tpd	Time of propagation delay (though the device)
UART	Universal asynchronous receiver transmitter
UCF	User constraints file
VHDL	VHSIC high level description language
VHSIC	Very high speed integrated circuit
VREF	Voltage reference
XOR	Exclusive OR
XST	Xilinx synthesis technology

1

Introduction

Digital systems and their design have evolved greatly over the last four decades. Rising densities and speed have provided designers a huge canvas to create complex digital systems. Present-day embedded systems use single-chip microcontrollers. Contemporary microcontrollers are available with 8-, 16- and 32-bit processing capability along with a peripheral set containing ADC, timer/counter and networks (I^2C, CAN, SPI, and UART). For most applications the microcontroller-based board is adequate. For applications where there is a need to integrate custom logic for faster control and additional peripherals, the microcontroller or microprocessor board is augmented by a FPGA or an application specific standard product (ASSP) device. The focus of this chapter is to understand different digital design methodologies before embarking on a full fledged description of the use of a custom digital design based on a FPGA.

1.1 Embedded System Overview

Embedded systems are usually single function applications. Various functional constraints associated with embedded systems are low cost, single-to-fewer components, low power, provide real-time response and support of hardware-software co-existence. A general methodology used in designing an embedded system is shown in Table 1.1.

The decision on the kind of digital platform to be used takes place during the system architecture phase as each embedded application is linked with its unique operational constraints. Some of the constraints of a digital controller of embedded system hardware include (in no particular order) the following:

- Real-time update rate
- Power
- Cost
- Single chip solution
- Ease of programming
- Portability of code

- Libraries of re-usable code
- Programming tools.

Table 1.1. Embedded system design flow [1]

Design phase	Design phase details
Requirements	Functional requirements and non-functional requirements (size, weight, power consumption and cost)
User specifications	User interface details along with operations needed to satisfy user request
Architecture	Hardware components (processor, peripherals, programmable logic and ASSPs), software components (major programs and their operations)
Component design	Pre-designed components, modified components and new components
System integration (hardware and software)	Verification scheme to uncover bugs quickly

1.2 Hypothetical Robot Control System

For understanding different digital design platforms, this text uses the design of a digital controller for a robot as a case study. The robot is a hypothetical, vertically articulated robot system for an automated assembly line. The process of designing this controller will help in understanding various digital design concepts. Figure 1.1 shows the various components of an assembly line robot. Each robot consists of five electric motors that work as actuators for different joints of the robot. A programming pendant or workstation is used to program the movements of the robot along with a communications network to link this robot to other robots on the assembly line. Various sensors are interfaced to the robot control system.

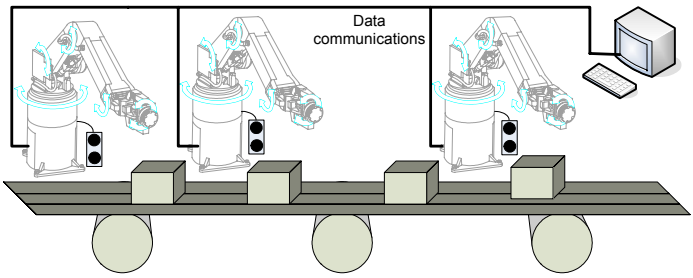


Fig. 1.1. Vertically articulated robot system used in an assembly line environment

The typical requirements of an Industrial robot controller include

- Control method for point-to-point control using servomotors
- Position detection using incremental or absolute encoder system
- Return to origin using limit switches and encoder
- Trajectory control
- Programming using a personal computer.

Table 1.2. Tasks for robot digital controller

Task	Subtask	Update time
Control of joint motors	Gate Driver, protection and current sensing	Fraction of a microsecond
	Dead time	Microseconds
	Closed-loop torque control	Tens of microseconds
	Closed-loop speed control	Hundreds of microseconds
	Position coordinate interpolation	Milliseconds
	Host communications	Tens of milliseconds
Sensor signal processing	ADC, DAC	Tens of milliseconds
Networking applications	Low-speed network	Milliseconds

Control Strategy for the Robot Controller

For implementing the robot controller on a digital system, a list of controller tasks is created in Table 1.2 along with the update time requirements. The major tasks for the robot controller for an articulated factory robot are

- Simultaneous control of five motors with details shown in Table 1.3.
- Signal processing of sensor inputs coming from robot environment — encoders, limit switches, proximity sensors, vision sensor
- Communication of robot co-ordinates to other robots in the vicinity, using CAN bus or Modbus®
- Communicating with host controller over serial port
- Computation of trajectory for robot movement.

Table 1.3. Specifications of a micro articulated robot Mitsubishi Movemaster RV-M1 [2]

Axis	Description	Encoder pulses per revolution (PPR)[1]	Gear ratio	Working range in degrees
J1	Waist	200	1:100	300°
J2	Shoulder	200	1:170	130°
J3	Elbow	200	1:110	110°
J4	Wrist pitch	96	1:180	90°
J5	Wrist roll	96	1:110	± 180°

The tasks and their update times are shown graphically in Fig. 1.2.

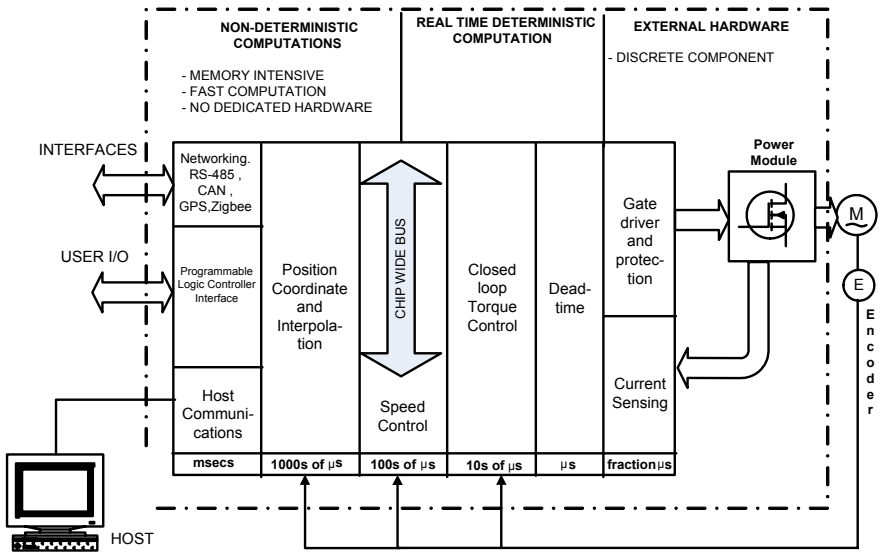


Fig. 1.2. Update times needed for various control functions of a robot control system [3]

1.3 Digital Design Platforms

Till the 1970s, electronic system designs were based on discrete analogue components such as transistors, operational amplifiers, resistors, capacitors and inductors. These circuits offered concurrent processing but had problems of parameter drift with temperature and ageing. The coming of TTL-based

[1] The encoder is used to find the position and speed of the robot joint. The working of the encoder is explained in Chap. 2.

components laid the foundation of digital design. The Intel 4004 microprocessor became the first digital platform which was configurable using software. Table 1.4 lists the major contemporary digital designs along with their relative merit.

Table 1.4. Digital design platforms

Digital design platform	Merit
Microprocessors	Reconfigurable using software. Good for computations
Microcontrollers, digital signal controllers	Combination of peripherals and CPU
Application specific standard product (ASSP)	A specialized peripheral with the ability to communicate with a host processor
Field programmable gate array (FPGA)	Ability to combine the strengths of processor, controller and ASSP

1.3.1 Microprocessor-based Design

The microprocessor has changed digital design methodology like no other digital component. It started out as a 4[2] bit programmable CPU in 1971 and still continues to be the digital controller of choice across several application areas. The microprocessor brought the concept of instruction set architecture (ISA), assembler and compiler. There are many real-time applications, with fast update rates require programming the microprocessor in its native assembly language. This is usually done when the size of available memory is a constraint. Even though most commercial microprocessors used today cater to data-centric applications, there are microprocessor cores embedded in microcontrollers for real-time control applications.

Digital control systems, like the robot application use a processor by using interrupts for real-time processing. There are interrupts for calculation of robot arm trajectory, encoder and sensor feedback, control of motors and networks. Each interrupt will occur based on the update time requirement of the given task. Figure 1.3 shows the generic nature of interrupt processing, where an interrupting device seeks CPU attention. A microprocessor-based robot controller carries out the task of arm positioning based on the flowchart shown in Fig. 1.4.

[2] The early Intel 4004 and the 8086 processor had close to 2300 and 29000 transistors. A basic 2 input NAND gate consists of 4 transistors. Effectively the early Intel processors 4004 and 8086 used only 575 and 7250 gates. This helps to put in perspective the amount of digital logic that can be accomodated in a 500,000 gate FPGA.

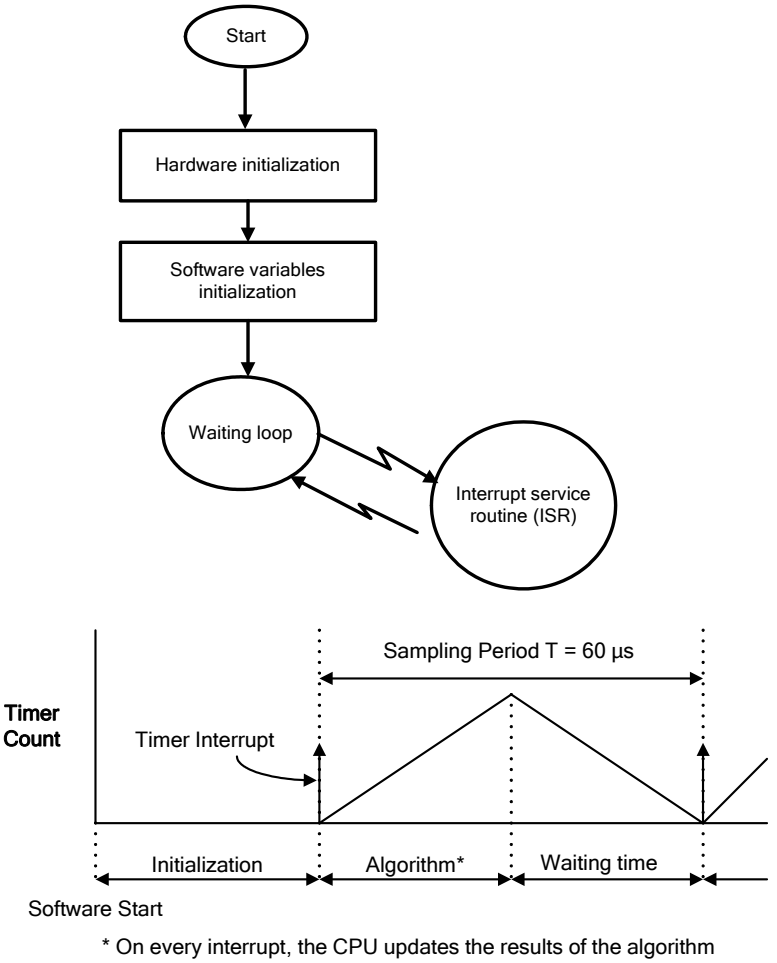


Fig. 1.3. Interrupt service routine (ISR) based processing scheme of processor-controller control scheme

Because most single core general purpose processors (GPP) are single-threaded (can process one instruction at a time), the processor use can become very high when managing multiple interrupts from different tasks of the robot controller. This can be seen from Fig. 1.5, where processor CPU use increases linearly with each motor.

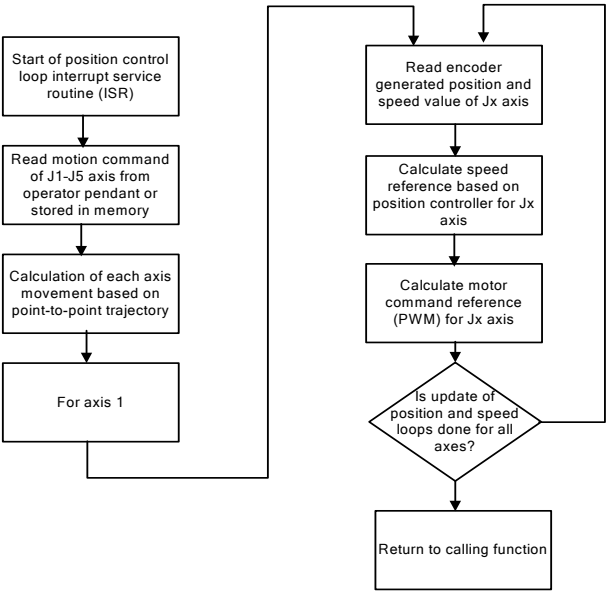


Fig. 1.4. Processor-interrupt-based flowchart needed for computing a control action [4]

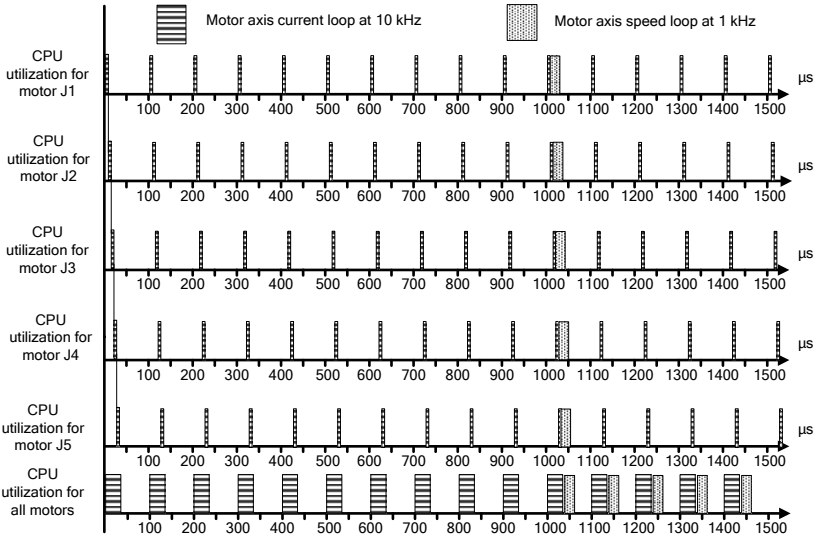


Fig. 1.5. CPU use for axis motor control for a single-threaded controller

1.3.2 Single-chip Computer/Microcontroller-based Design

The microcontroller represents the next generation of controllers for embedded systems. It allows creating systems with fewer numbers of components by

incorporating peripherals that were earlier externally interfaced with the general purpose processor. A block diagram of a typical single-chip controller, which is used as a robot motor controller, is shown in Fig. 1.6.

Like the microprocessor, tasks in a microcontroller design environment are divided as per the update rates required. For tasks requiring low update rates, coding is accomplished using a software programming language such as C. Tasks that need to have high deterministic update rates are coded using the native assembly language for a particular microcontroller. In the robot application at hand, many of the motor control routines require update rates of a few kilohertz. Traditionally, these routines are written in assembly language. It is difficult to port routines written in assembly language as they are tied to a CPU's ISA. The other constraint with a microcontroller-based system is the fixed number of available peripherals. Though microcontroller vendors offer a wide range of devices with different numbers and types of peripherals, it is not always possible to find one that matches the application requirements perfectly.

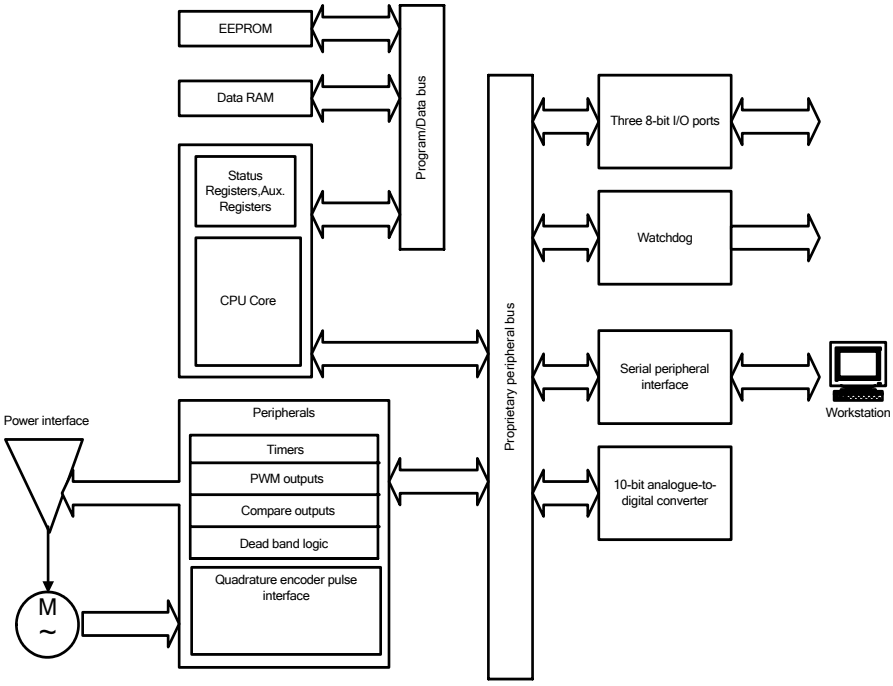
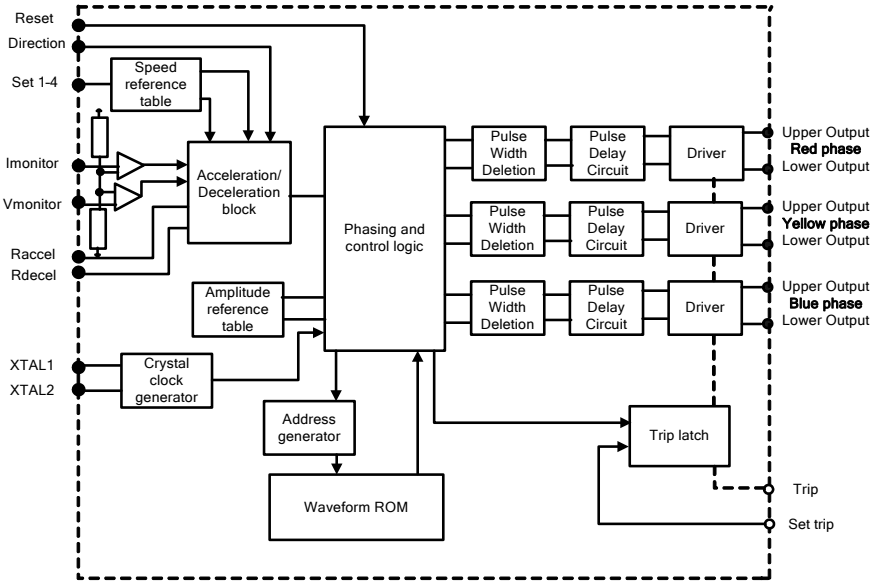


Fig. 1.6. Single-chip microcontroller environment for a motor control application

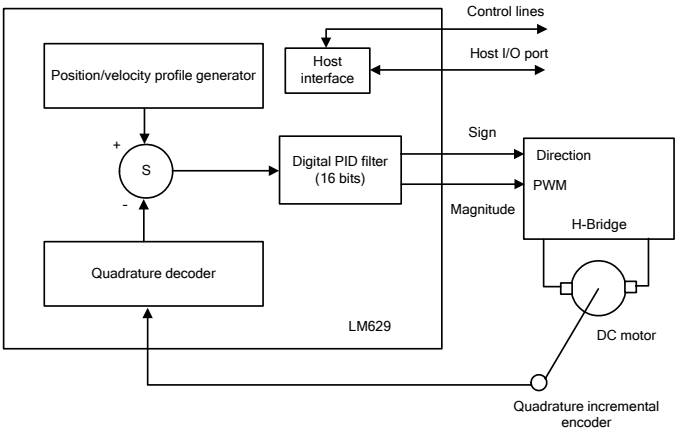
1.3.3 Application Specific Standard Products (ASSPs)

An ASSP is a configurable logic component for a specific application. The functionality of an ASSP is tweaked by specifying its control word. ASSPs are made in volumes and cater to the generic requirements of the application. Most of

the time, ASSP-based designs are used on a PCB. In the robot control application at hand, an ASSP can be used for controlling the motor for each axis of the robot. Based on the type of motor and control strategy used, a corresponding ASSP is chosen. Two examples of ASSPs for motor control include LM629 from National Semiconductor for control of a brushed DC motor and SA628 (see Fig. 1.7a and b) for three-phase motor control. Configurable ASSPs provide address, data and control bus connectivity for interfacing with the host processor.



a



b

Fig. 1.7. a ASSP chip SA628 for control of a three-phase AC Induction Motor [5]; **b** ASSP chip LM629 for control of a DC motor

1.3.4 Design Using FPGA

The present-day FPGA provides a platform that supports both processor and custom logic requirements. The microcontrollers currently have an edge over the FPGA in terms of power and cost. But FPGAs are catching up by offering portability of code across various FPGA vendors, libraries of re-usable code and availability of low-cost programming tools. Programmable devices that were traditionally low gate count devices are now in a position to support large parts of digital system logic. The digital designer today has a viable option of using only the FPGA device as the embedded system controller. The availability of high-density, low-cost FPGA devices has given digital designers lots of flexibility to design custom digital architectures using FPGA and HDLs. FPGA devices have evolved from their glue logic predecessor to a device that now contains a large variety of built-in digital components (memory, multipliers, transceivers and many more). FPGA device density has risen over the years and at the same time its cost has made it economically viable for use in several applications. Contemporary FPGAs contain thousands of look up tables (LUTs) and FFs for implementing complex digital logic.

Contemporary FPGAs offer

- Reconfigurability: Field programmable devices can be reconfigured at any time. Designers can integrate modifications or do complete personality changes.
- Software-defined design: The hardware is defined by software-like languages (HDL). Designers can develop, simulate and test a circuit fully before "running" it on a field programmable device.
- Parallelism: Circuits defined in an FPGA can be designed in a completely parallel fashion. This is similar to using multi-path analogue circuits. A user can instantiate multiple hardware implementations on the same chip without cross-module interference or computation loading. An example of FPGA-based concurrent processing is shown in Fig. 1.8.

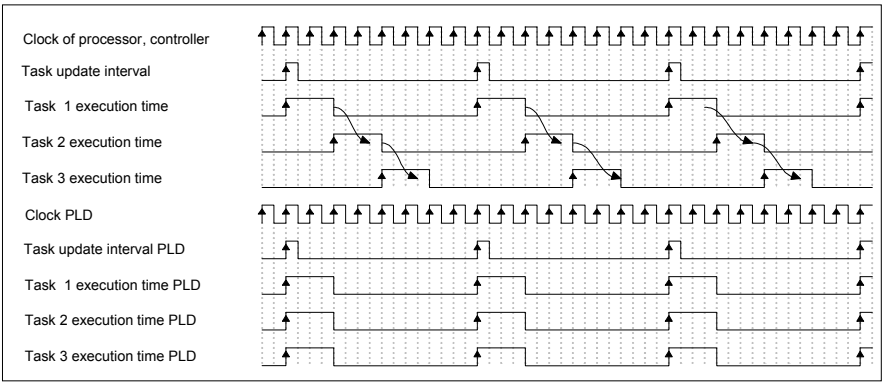


Fig. 1.8. Multi-tasking scheme using a GPP vis-à-vis a FPGA

- High speed: Because an FPGA is a hardware implementation running with fast clock rates, designers can achieve very high speeds. Coupled with parallelism, FPGA implementation can outperform processor-based systems.
- Reliability: Designers can expect true hardware reliability from FPGAs because there is no operating system or driver layer[3] that can affect system uptime.
- IP protection and re-use: Once compiled and downloaded to a FPGA, hardware implementation is difficult to reverse engineer. A tested hardware design can be re-used multiple times by instantiating.

FPGA-based systems are gaining acceptance because these systems integrate digital logic design, processors and communication interface on a single chip. The front end design flow of a FPGA is very similar to that of a custom logic design. Almost all FPGA vendors offer a suite of software tools that allows a designer to simulate, synthesize, place and route and program the FPGA. Table 1.5 shows the different design tools offered by two leading vendors. Once a designer feels comfortable in a particular design suite, it is easy to migrate to another vendor's design tools because they work in a similar fashion[4].

Table 1.5. Common design tools provided by two leading FPGA Vendors

Functionality	XILINX	ALTERA
Design synthesis, mapping, place and route	Integrated Software Environment (ISE)™	Quartus II®
FPGA embedded processor design tool	Embedded Design Kit (EDK)®	System on Programmable Chip (SoPC) builder®
Custom peripheral support	Yes	Yes
On-Chip signal logic analyzer	ChipScope™ Pro	SignalTap®
MATLAB® co-simulation and IP cores library	System Generator™	DSP Builder®

[3] Not applicable to FPGA-based processor systems.

[4] One of the strengths of HDL and associated synthesis software is to make the implementation option wider for the designer. For consistency, this book uses a contemporary Xilinx SPARTAN-3E™ 500K gate FPGA along with the Xilinx ISE™ for illustrating various examples. The author feels strongly that if the designer is able to master one vendor's specific design flow along with a given FPGA architecture, the same concepts can be applied to understand quickly and implement a digital design using FPGAs from other vendors.

From an implementation point of view, a robot controller using a FPGA device can be considered a viable alternative[5], as robots are usually low-volume application-specific systems. The FPGA allows for customization of servo-motor type for joint control, industrial communciation network, integration of custom peripherals and control algorithms.

Software-based design flows are suited for applications which are data centric and hardware design flow is suited for fast real-time applications.Table 1.6 provides a transition path for migrating from microprocessor/controller to FPGA-based design. The FPGA design process consists of design entry, which is accomplished by using either schematic or HDL. Following the design phase, digital logic is synthesized, mapped and placed on a FPGA[6].

Table 1.6. Transition path from a microcontroller-based system to a FPGA system

Existing microprocessor/microcontroller code	**Field programmable device**
Target independent 'C' Code	Embedded processor within the FPGA device
Target dependent assembly constructs for routines requiring fast update rates	Target independent HDL-based coding for routines requiring very fast update rates

1.4 Organization of the Book

The book is organized to weave together concepts, tools and techniques to help in designing FPGA-based embedded systems. This book does assume that the reader is versed in the basic concepts of embedded systems programming and interfaces. There are references at the end of each chapter where the reader can get more information on the topics covered in the chapter. This text is trying to put together many components of a system, so certain sections are not covered in detail but are used to convey the concept of system design.

The sequence of chapters is to introduce basic concepts and then build upon them. Table 1.7 details the contribution of each chapter in building up a FPGA-based digital system.

[5] The purpose of this text is to explain embedded hardware design using FPGA. It is not the intention of this text to prove that FPGA-based robot controller is the best digital platform for implementing the robot controller.

[6] The HDL design process is described in Chap. 2. The complete design flow of synthesis, mapping, place and route is described in Chap. 3.

Table 1.7. Preview of FPGA-based digital design implementation

FPGA design	**Chapter**						
	1	2	3	4	5	6	7
The case for using FPGAs	■						
Hardware description language (HDL)		■					
Synthesis of HDL design using FPGA as a target device			■				
FPGA embedded processors				■			
Serial communications and interfacing					■		
Motor control						■	
Prototyping using FPGA							■

Broadly, Chaps. 1 to 4 of the book introduce the technology and tools for implementing digital logic using a FPGA device. Chapters 5 to 7 discuss interfacing, motor control and prototyping using FPGA.

As shown in Fig. 1.9, different aspects of robot controller design are covered in chapter numbers mentioned in each component.

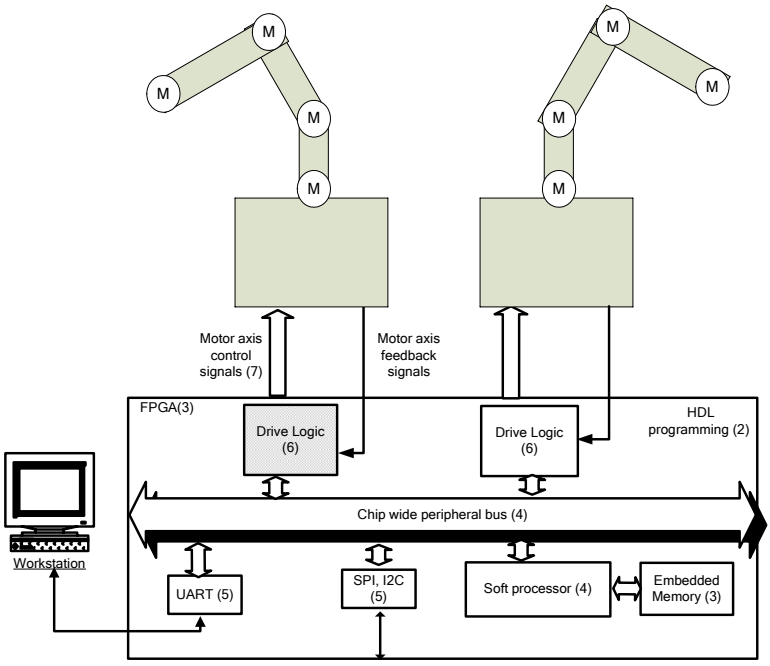


Fig. 1.9. Contribution of each chapter (shown in parentheses) for creating a robot controller

The second chapter is on simulation of digital systems using Verilog as the hardware description language (HDL). It introduces basic concepts of how a printed circuit board (PCB) containing digital components can be modelled using HDL and how it can be tested using software simulators. A simulation environment of an EDA tool is also explained.

Chapter 3 of the book introduces the architecture and resources of FPGA. Each building block of the programmable device such as embedded memory, phase-locked loops, logic blocks, multipliers and different interfacing I/O standards are explained along with their HDL based instantiation template. The chapter ends with examples of digital systems and their FPGA-based synthesis results.

FPGA-based embedded processors have made it possible to migrate from microcontroller-based embedded system design to FPGA-based embedded system design. FPGA-based designs give the designer an option to retain much of the skill set of high-level software programming. Now instead of coding in a native assembly language for a particular processor — deterministic tasks can be coded in HDL. Chapter 4 provides methodology on bringing together the software and the hardware worlds. FPGA immersed processors along with different interfacing buses connect to external standard and custom peripherals. A system-on-chip is created using this approach.

Chapter 5 discusses FPGA-based interfaces. It covers basic data communication using HDL and FPGA and protocols. The chapter also discusses asynchronous and synchronous serial data communications. The second section of the chapter discusses basic signal conditioning of the acquired signal.

The actuator is the last component of the control loop. In the robot example used in this book, the electric motor is the actuator for various joints of the robot. Chapter 6 discusses digital design and control implementation of different motors — stepper, permanent magnet DC motor, brushless DC motor, permanent magnet synchronous motor (PMSM) and permanent magnet reluctance motor.

The last chapter of the text is on prototyping the different schemes discussed using a FPGA-based board. It discusses various hardware verification and interfacing techniques, which are useful for hardware system integration.

Problems

1. Give an example of a application suited for a microcontroller and for a FPGA. Justify why one cannot replace the other.
2. What are the limitations of a FPGA-based system vis-à-vis a custom ASIC-based system.
3. How is real-time processing done on a GPP or a microcontroller based system by using interrupts?
4. What kind of power constraints are part of an articulated factory robot and that of a robotic rover shown in Fig. 1.10?
5. The robotic rover application (shown in Fig. 1.10) involves travel along terrains either by use of a remote link such as the Global Positioning System (GPS). The rover collects information about its surroundings using

sensors and relays it to a base station or operator console. A list of tasks for this rover includes

a. Power management for the rover
b. Control of six motors
c. Signal processing of sensor inputs coming from the robotic environment using a vision sensor.
d. Determining the robot position using GPS
e. Communicating with the host controller using ZigBee
f. Ability to interface with various payloads — new sensors, new actuators.

Partition the tasks as per their update time requirements and comment on the suitability of putting the task on a FPGA or a GPP.

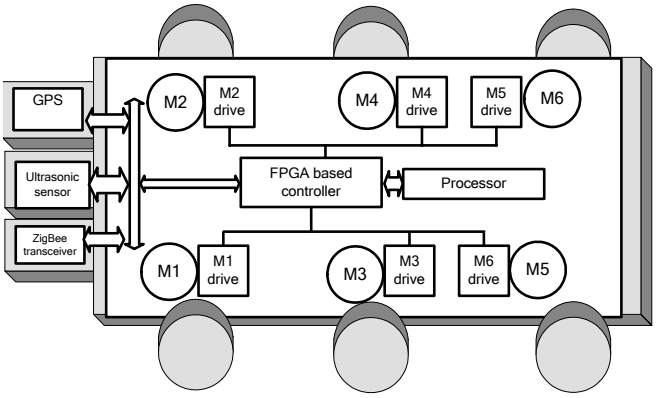


Fig. 1.10. Diagram of a robotic rover

References

1. Wolf W (2005) Computers as Components: Principles of Embedded Computer Systems Design. Morgan Kaufmann, San Francisco
2. Kung YS, Shu GS (2005) Development of a FPGA-based motion control IC for robot arm. Paper presented at IEEE Industrial conference on Industrial Technology (ICIT 2005), City University of Hong Kong, Hong Kong, December 2005
3. Goetz J, Takahashi TT (2003) A design platform optimized for inner loop motor control. Paper presented at power conversion and intelligent motion (PCIM 2003) conference. http://www.irf.com/technical-info/whitepaper/pcimeur03innerloop.pdf. Accessed 15 October 2008.
4. Kung YS, Shu GS (2005) Design and implementation of a control IC for vertical articulated robot arm using SOPC technology. Paper presented at IEEE Mechatronics ICM 2005, pp. 532–536
5. Mallinson N (1998) Plug and play single chip controllers for variable speed induction motor drives in white goods and HVAC systems. Paper presented at IEEE applied power electronics conference APEC 1998, 2:756–762

Further Reading

1. Maxfield C (2004) The design warrior's guide to FPGAs – devices, tools and flow. Newnes
2. Vahid F, Givargis T (2002) Embedded system design – A unified hardware/software introduction. John Wiley
3. Keramas JG (1999) Robot technology fundamentals. Thomson Delmar
4. Klafter RD et al (1989) Robotic engineering, an integrated approach. Prentice-Hall
5. Balch M (2003) Complete digital design, a comprehensive guide to digital electronics and computer architecture. McGraw Hill
6. Slater M (1989) Microprocessor-Based Design, A Comprehensive Guide to Hardware Design. Prentice-Hall
7. Monmasson E, Chapuis Y (2002) Contributions of FPGAs to the Control of Electrical Systems, a Review. IEEE Industrial Electronics Society Newsletter, 49(4)
8. Newman KE, Hamblen JO, Hall TS (2002) An Introductory Digital Design Course Using a Low-Cost Autonomous Robot. IEEE transactions on Education, 45(3):289–296
9. Kung YS et al (2006) FPGA-Implementation of Inverse Kinematics and Servo Controller for Robot Manipulator. Paper presented at IEEE Robotics and Biomimetics, (ROBIO 2006) at Kunming China, December 2006
10. Navabi Z (2007) Embedded Core Design with FPGAs. McGraw Hill
11. Navabi Z (2004) Digital Design and Implementation with Field Programmable Devices. Springer
12. Navabi Z (1999) Verilog Digital System Design. McGraw Hill

2

Hardware Description Language: Verilog

The technology of translating a given digital design task into digital logic has undergone many changes. The 1970s and 1980s witnessed a schematic design approach. From the mid-1990s onward, digital design has been done using hardware description language (HDL). HDLs came into existence to help the designer with the simulation of digital logic. The availability of synthesis tools that convert HDL logic to FPGA primitives has made HDL the digital design entry method of choice. Given the fact that HDLs started out primarily as a simulation language, there are many HDL constructs that cannot be synthesized to digital logic. This chapter will focus on the synthesizable subset of constructs of Verilog HDL. Describing a digital design using HDL is usually the first step toward prototyping the design using FPGA. The rest of the book will use Verilog constructs introduced in this chapter to create digital designs for interfacing, networking, signal conditioning and motor control applications. Verilog is a vast language, and it is beyond the scope of this chapter and book to dwell on all the nuances of the language.

2.1 Software and Hardware Description Languages

It helps to understand broadly how a general purpose software programming language such as C compares with the hardware description language. Both software and hardware description languages are target device independent languages. A code written in C can be compiled for execution on an Intel, Motorola or ARM microprocessor. It helps if the designer knows the processor architecture and assembly constructs. This can lead to faster and more compact programs. But for applications where memory and speed are not a constraint, the designer can get by, without knowing the details of the underlying processor architecture.

In the way software programming language shields the programmer from getting caught up in the details of an individual processor's assembly language, the

HDLs present a similar advantage. Here the digital designer writes a description for a digital circuit using HDL, without worrying about the primitives[7] of a target device. For most high-level software description languages, the execution is single –threaded because there is a single CPU core attending to the execution of logic[8]. In HDL, the designer can model and construct different concurrent paths for executing logic. This is why HDLs are said to model and aid in implementing the concurrent behaviour of circuits.

Does it mean the end of software programming languages? No, these languages continue to contribute to the design of digital and embedded systems. Chapter 4 will discuss more on the use of software programming languages when designing FPGA-based processor systems.

Basic Concept of HDLs – Verilog and VHDL

With most digital design exceeding thousands of gates, the schematic design approach has given way to more abstract descriptions of digital design. This more abstract design methodolgy is based on hardware description language. Contemporary HDL languages started out as simulation languages. Very high speed integrated circuit hardware description language (VHDL) started out as a U.S. Department of Defense initiative. It was primarily meant to integrate and correlate simulation results of digitial circuits from various defense vendors. Similarly, Verilog evolved as a tool for verifying logic in the digital domain. Both VHDL and Verilog are defined by IEEE standards. Verilog has been through revisions to cover deficiencies. Verilog is defined by the IEEE standard 1364. The IEEE 1364-1995 and IEEE 1364-2001 refer to Verilog-95 and Verilog-2001. Today with the help of EDA synthesis tools, code written in HDL can be synthesized into target specific architectures. Both HDLs can be understood by the way their design approach mirrors the use of discrete chips on a PCB.

Verilog divides its constructs into four levels of abstraction. The first level of abstraction is the switch level, where individual MOS transistor-based switches are interconnected to form gates and flip-flops. The second level of abstraction is the gate level, where one can instantiate basic gates and interconnect them to form a digital system. Both the switch level and gate level constructs are rarely used in designing high density digital logic. The third level of abstraction, the data flow provides interconnection of different combinational logic circuits using a single statement. Behavioural modeling supports the most abstract level of construct using HDL. Here the designer can code digital design in the format of a high-level software language. For Verilog, behavioural constructs resemble the C programming language constructs. Even though each abstraction layer defines different keywords, signals between different abstraction layers can be interconnected.

---

[7] Primitives are the assembly level constructs of the hardware world. Chapter 3 discusses in detail the commonly used primitives of the Xilinx field programmable gate array (FPGA)

[8] Multi-core processors can execute several threads of logic independently!

2.2 Let's Use Verilog as Our HDL!

The case for a particular HDL (either Verilog or VHDL) cannot be argued[9]. Let us say, we decided to use Verilog by tossing a coin. For the remainder of this text, we will use Verilog 2001 for design examples.

One of the ways of understanding many concepts of HDL is to view its use from the view point of a PCB. PCBs in the 1980s had lots of 74xx series chips that were interconnected using copper tracks. If you happen to have an old computer from the 1980s, you will notice discrete 74xx chips on the motherboard used for address decoding and latching data/address buses. Increased gate densities made it feasible to incorporate large quantities of combinational and sequential logic onto a single programmable chip. It is difficult to spot those 74xx series chips on the motherboard because they are now contained in a single chip.

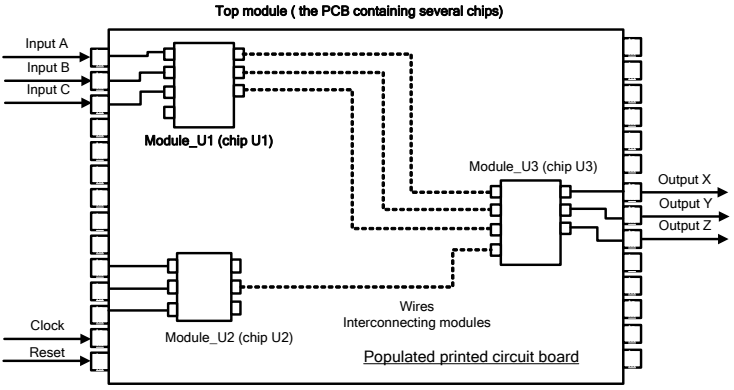


Fig. 2.1. Populated printed circuit board analogy of Verilog HDL

The fundamental concept of Verilog **module**, **port list** and **wires** can be explained by using basic PCB design terminology. As shown in Fig. 2.1, each chip on the PCB is a **module** in Verilog HDL. The **port list** denotes the number and type of I/O pins of the module. The interconnections between various chips on the PCB are denoted as **wires**. If the PCB consists of a fixed number of IC chips, then the entire PCB becomes a **top module** where the external world signals to the PCB constitute the port list of this **top module**.

2.3 Design Examples Using Verilog

A HDL is better understood through examples that illustrate facets of designs. Let us take different examples to demonstrate the use of Verilog by examples on

[9] Both Verilog and VHDL have their own devout followers. For the functionality described in this book, either of the HDLs can be used. Once one HDL is understood, it is easy to migrate to the second using the same fundamental concepts.

combinational, sequential and finite state machine (FSM) based circuits. The following section shows Verilog[10] designs based on gate, data flow and behavioural modelling.

2.3.1 Gate Level Model

The gate level constructs allow a designer to synthesize a digital circuit using basic digital gates. Gates are synthesizable constructs supported by all synthesis tools. The FPGA synthesis tool implements digital gates using a LUT.

Example 2.1. For safety, many sensors are used to protect a robot axis from self-destructing. These sensors include limit switches, proximity switches and safety interlocks. Convert the control scheme shown in Fig. 2.2, to digital logic by using Verilog HDL.

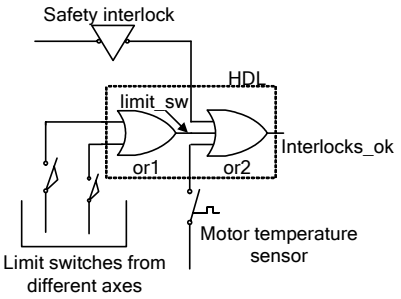


Fig. 2.2. Interlock circuit using basic gates

The port list of the example circuit consists of four inputs and one output. One bit width is the default size for each I/O; hence the port size for 1-bit I/Os is not explicitly mentioned. The code in Fig. 2.3 shows instantiation of two OR gates. To instantiate a basic gate, the output is mentioned first followed by the inputs to the gate. In the example, the wire **limit_sw** connects the output of the first OR gate (or1) to the input of the second OR gate (or2).

```
module gate_1 (input lim_sw1, lim_sw2, motor_temp, safety, output interlocks_ok);

        wire limit_sw;

        or or1 ( limit_sw , lim_sw1 , lim_sw2);
        or or2 (interlocks_ok , limit_sw , motor_temp , safety);

endmodule
```

Fig. 2.3. Verilog code for interlock circuit

[10] In all HDL design examples, Verilog keyword is **boldfaced**. There are no accompanying test-benches with the Verilog codes. The reader is encouraged to write test-benches to verify the codes presented in the examples. The Verilog examples presented in the book are for illustration only. They are neither complete nor extensively tested for use in a real system.

Example 2.2. To hold the robot joint at the desired location after the axis has positioned itself, a brake is often employed. This brake can be part of the motor controlling the joint. For the robot joint to move, the brake has to be released (usually by powering it, logic 1), when the joint has reached its pre-determined position, the brake is set (logic 0). The diagram for this interlock is shown in Fig. 2.4.

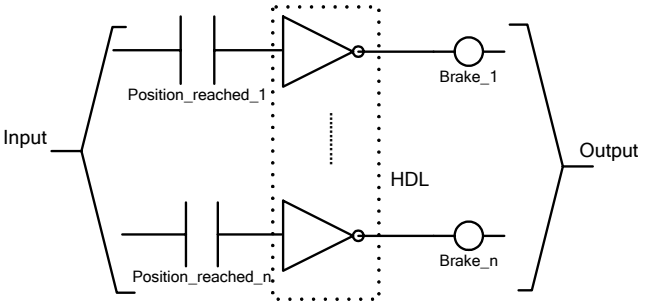


Fig. 2.4. Creating a brake interlock using digital gates

The input is a signal from a limit switch, which indicates that the desired position is reached. A NOT gate sets the brake when this position_reached contact is active. The inverter NOT gate is instantiated in a fashion similar tp the OR gate in Example 2.1 (see Fig. 2.5).

```
module brake( input axis_position, output brake);

// Gate Instantiation

not (brake, axis_position);

endmodule
```

Fig. 2.5. Verilog code for brake interlock circuit

2.3.2 Combinational Circuits Using Data Flow Modelling

The data flow method is used to model asynchronous combinational logic designed to work using the concept of transition on change. Any time an input changes, the entire logic circuit is re-evaluated. **Assign** statements in Verilog are used for modeling circuits governed by the transition of change concept. The output, which is the left-side expression of the **assign** statement changes as soon as the input, the right-side expression of the assign statement changes. The generic format for using the assign statement,

assign output = input1 ***operator*** input2;

Table 2.1 lists the operators used for data flow modeling.

Example 2.3. Create the logic for a single loop ON/OFF controller, much like the one used in refrigerators and air-conditioners. The user sets the amount of control action needed. The digital circuit computes the difference between the set point and the actual temperature and turns on a relay. Figure 2.6 shows the control scheme of the process.

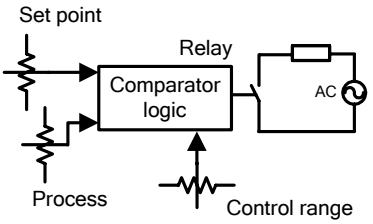


Fig. 2.6. ON/OFF controller control scheme

In this example, the inputs to the controller, set point and process values are available from an 8-bit uni-polar ADC. The Verilog code of Fig. 2.7 uses an **assign** statement to realize combination logic for computing error. A conditional operator is used to actuate the relay.

```
module on_off (input [7:0] set_point, process, input [3:0] control_range, output relay);

wire [7:0] error; // Unsigned value

assign error = set_point - process; // Set point is always more than the process value

assign relay = ( error > control_range ) ? 1'b1 : 1'b0 ;

endmodule
```

Fig. 2.7. Verilog code for ON/OFF controller

Table 2.1. Arithmetic and logic operators used for data flow design[11]

Operator Type	**Operator symbol**	**Operation performed**
Arithmetic	*	Multiply
	/	Divide
	+	Add
	-	Subtract
	%	Modulus
Logical	!	Logical negation
	&&	Logical and
	||	Logical or
Relational	>	Greater than
	<	Less than
	>=	Greater than or equal
	<=	Less than or equal
Equality	==	Equality
	!=	Inequality
Bit-wise	~	Bit-wise negation
	&	Bit-wise and
	|	Bit-wise or
	^	Bit-wise ex-or
	^~ or ~^	Bit-wise ex-nor
Shift	>>	Right shift
	<<	Left shift
Concatenation	{ }	Concatenation
Conditional	?:	Conditional

[11] The Verilog arithmetic and logic operations mentioned in this table can be converted to equivalent digital hardware, i.e. they are synthesizable. The exception is the **divide (/)** operation which is supported only for powers of 2.

2.3.3 Behavioural Logic

The description of sequential circuits using the cyclic model is also at times referred to as the behavioural model because it models the behaviour of the system when an event occurs. Sequential circuits are used when register transitions are to be modelled about a rising or falling edge of the clock. Much of the syntax of C is seen within the behavioural model of Verilog HDL. Verilog models cyclic behaviour based on either edge or level of clock or signal.

Verilog keyword for modelling cyclic processes is

```
always @ ( posedge clk) // activates on the positive edge of clock
    begin
    …..
    end

always @ ( negedge clk) // activates on the negative edge of clock.
    begin
    …..
    end
```

Shifting of digital data bits on a clock edge is a very common application in digital signal processing and data communications. In digital signal processing, data are shifted at every sample to implement a delay function designated by z^{-1} operation, and in data communications, the data word contents need to be serially shifted out or serially accepted. The rate of shifting is important for both of these applications. The clock of the shift register determines the shift rate. A shift register is a good example to demonstrate the concept of blocking and non-blocking statements in Verilog. Blocking assignment (=) statements execute in the order they are specified in a sequential block (between **begin** and **end**). Non-blocking statements (<=) allow execution of each statement without linkages to results from previous sequential statements.

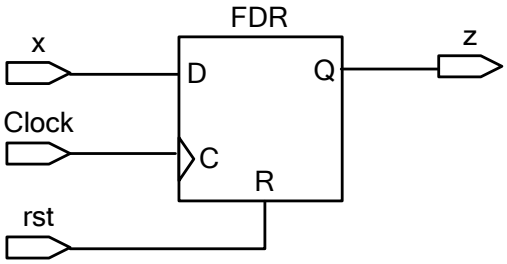
Example 2.4. Create a 4-bit shift register, where input bit stream **x** appears at the output **z** after four rising edges of the clock.

The first model of the code is shown in Fig 2.8a. This uses the blocking style of coding, which results in a single flip-flop, where the output is directly equated to the input. Because the code in Fig. 2.8a uses a blocking style of coding, the four equate statements execute in sequence. The output **z** is directly equated to input **x**. Figure 2.8b shows the synthesis results of the code, which is an instantiation of one flip-flop.

The second code shown in Fig. 2.8c is similar to that shown in Fig. 2.8a. The Verilog code of Fig. 2.8c uses non-blocking statements to assign the input **x** to output **z** in a four-stage shift register. The synthesized hardware of Fig. 2.8d shows four FFs, which the design required. The statements in non-blocking code do not execute sequentially. The right-hand side term of each statement executes concurrently at every clock cycle.

```
module shiftreg(input x, clock, rst, output reg z);

reg a,b,c;

always @ ( posedge clock)
if (rst)
        begin
            a = 0;
            b = a;
            c = b;
            z = c;
        end

endmodule
```

a



b

```
module shiftregb (input x, clock, rst, output reg z);

reg a,b,c;

always @ ( posedge clock)
        if (rst)
            begin
                    a <= 0;
                    b <= 0;
                    c <= 0;
                    z <= 0;
            end
        else
            begin
                    a <= x;
                    b <= a;
                    c <= b;
                    z <= c;
        end

endmodule
```

c

Fig. 2.8. a Verilog code for a shift register using blocking statements; **b** synthesized hardware for a shift register model using a blocking statement **c** Verilog code for a 4-bit shift register model using non-blocking statements

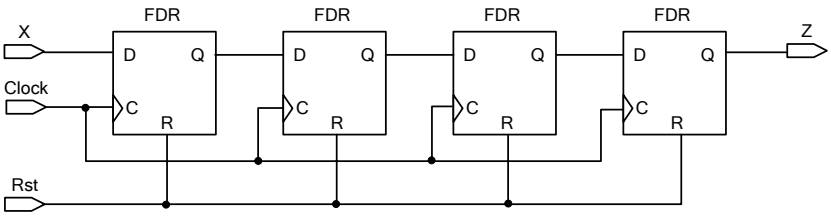


Fig. 2.8. d Synthesized hardware for a shift register model using a non-blocking statement

Example 2.5. Create a shift register to transmit bits of an input word in serial fashion. The shift register is interfaced to a FIFO, where a read (rd) command from the shift register is sent to get the new word from the FIFO (see Fig. 2.9).

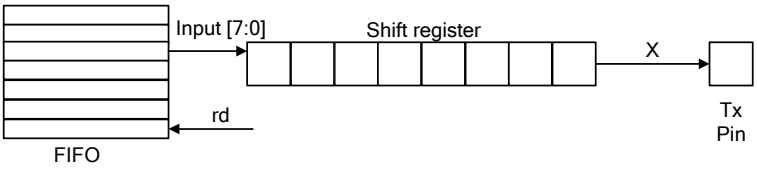


Fig. 2.9. Serial shift register connected to a FIFO

A counter is used to shift the 8-bits of the word sequentially to the output (see Fig. 2.10).

```
module shift_s( input [7:0] word, input clk, rst , output reg rd,x);

reg[2:0] count;

always @ ( posedge clk )

begin
        if (rst)
        count <= 0;

        else   if (count < 7)
                begin
                x <= word[count];
                count <= count +1;
                rd <= 1'b0;
        end

        else   if(count == 7)
                begin
                x <= word[7];
                count <=0;
                rd <= 1'b1;
                end
        end

endmodule
```

Fig. 2.10. Shift register for shifting out 8 data bits

2.3.4 Finite State Machine (FSM)

It is good design practice to breakup a given specification of digital design into discrete pre-defined states. This ensures that all possible transitions are taken into consideration and their response is pre-determined at the design stage. The design of a FSM consists of a combinational logic section that determines the next state and a sequential circuit that performs state transitions. Based on the type of circuit, a FSM in Verilog can be coded in different ways. Finite state machines can either transition synchronously or asynchronously. Because most digital systems are synchronous, state transitions take place on the edge of a common clock.

The block diagram of a finite state machine, shown in Fig. 2.11, consists of three processes with the following functionality:

- Combinational state change determining next state logic
- Sequential logic for synchronously changing states
- Combinational logic for changing output.

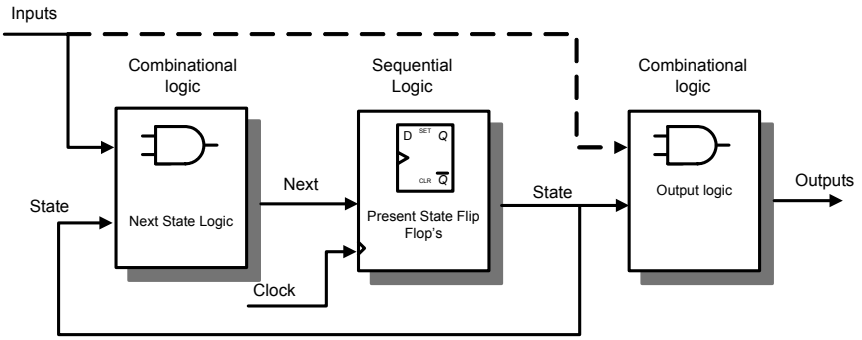


Fig. 2.11. Finite state machine model consisting of three processes

Example 2.6. Design a HDL finite state machine, which will control a car wash process. The car wash process consists of five states (see Table 2.2). The controller performs action for a particular state until the timer times out. The timer done bit controls the state transition, once the car wash process has started.

Table 2.2. States of the car wash FSM

State	Process description
S1	Wash_1
S2	Soap
S3	Scrub
S4	Wash_2
S5	Dry

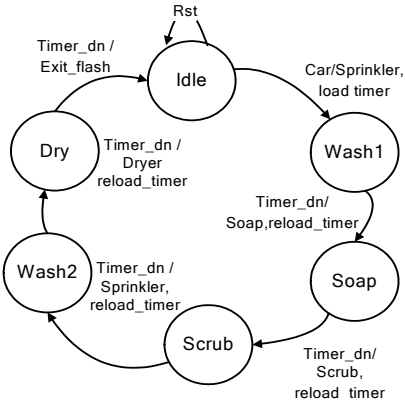


Fig. 2.12. a FSM for a car wash controller

```
module fsm (input rst, clk, car, output reg soap,sprinkler,scrub,blower);

parameter wash1 = 3'b001; parameter dispense= 3'b010; parameter scrubber = 3'b011;
parameter wash2 = 3'b100; parameter dry = 3'b101; parameter idle =3'b110;

reg [2:0] state, next_state;
reg timer_start, timer_dn;
reg [7:0] timer, timer_ld;

always @ ( state or timer_dn or car)
begin

timer_start = 0;
next_state = state;

case (state)

idle: begin
                sprinkler =0; soap =0 ; scrub =0; blower = 0;
                timer_start = 0;
                timer_ld = 8'h00;
                next_state = wash1;
                end

wash1: if ( car)

                begin
                sprinkler = 1; soap =0 ; scrub =0; blower = 0;
                timer_start = 1;
                timer_ld = 8'h04;
                next_state = dispense;
                end

dispense: if (timer_dn) begin

                        sprinkler = 0; scrub =0; soap = 1; blower = 0;
                        timer_start = 1;
                        timer_ld = 8'h01;
                        next_state = scrubber;
                        end
```

Fig. 2.12. b Finite state machine based code for a car wash controller

```
scrubber: if ( timer_dn ) begin
                                soap = 0; sprinkler = 0; blower = 0;
                                scrub = 1;
                                timer_start = 1;
                                timer_ld = 8'h02;
                                next_state = wash2;
                                end

wash2: if ( timer_dn ) begin
                                soap = 0; scrub = 0; blower = 0;
                                sprinkler = 1;
                                timer_start = 1;
                                timer_ld = 8'h05;
                                next_state = dry;
                                end

dry: if ( timer_dn ) begin
                                soap = 0; scrub = 0; sprinkler = 0; blower = 1;
                                timer_start = 1;
                                timer_ld = 8'h05;
                                next_state = idle;
                                 end

default : next_state = idle;
endcase
end

always @ (posedge clk)
begin
if (timer_start)
      timer <= 0 ;
 else
      timer <= timer + 1;
end

always @ (posedge clk )
begin
      if ( timer == timer_ld)
      timer_dn <= 1;
      else
      timer_dn <= 0;

end

always @( posedge clk )
begin : state_transitions
if ( rst ) state <= idle;
else state <= next_state;
end

endmodule
```

Fig. 2.12. b (continued)

Determining Position and Speed of the Robot Axis

To move the robot from one set of co-ordinates to another, the current position of the robot must be determined. The most commonly used device for detecting robot axis position is the rotary encoder. The rotary encoder can be either an incremental encoder shown in Fig. 2.13 or an absolute type of encoder. The quadrature encoder is a common type of incremental encoder. Here two channels (A and B) are used to sense position, velocity and direction of rotation. The two channels A and B are positioned 90° out of phase, as shown in Fig. 2.13. Using the output of these two channels, both position and direction of rotation can be determined. If A leads B, the axis coupled to the encoder is rotating in one direction and if B leads A, then the axis is rotating in the reverse direction. The resolution of the signal coming from the quadrature encoder is improved to 2x if the positive edges of A and B are counted, and it is further improved to 4x if both the active edges of the channels are counted.[12]

Many application specific standard products (ASSPs) and microcontrollers provide a built-in digital logic to decode a signal coming from a quadrature encoder. For this purpose the quadrature encoder interface (QEI) is a standard peripheral. A digital filter is used to remove noise signals from channels A and B.

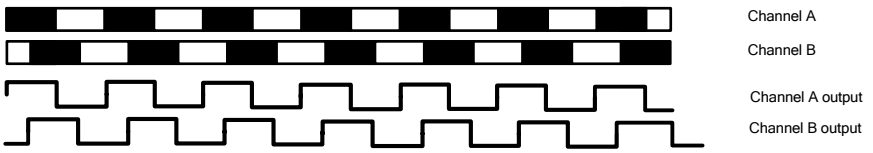


Fig. 2.13. Pulse train generated by a quadrature incremental encoder

Example 2.7. Calculate the number of pulses recorded in the position counter for the working range of Axis J1 of the robot. The encoder is mounted on the motor shaft.

Using the parameters of axis J1 (from Table 1.2),
Encoder pulses per revolution (PPR) is 200.
Gear ratio: 1: 100
Working range: 300°

The encoder pulses obtained for the specified working range are determined by the equation,

Position pulses = encoder PPR * gear ratio * (working range in degrees/360)
= 200 * 100 * (300/360)
= 16,666 pulses
= $411A_H$.

A 15-bit position register is needed to store the value of the axis movement.

[12] Increasing the resolution of the pulse counting circuit provides greater positioning capability.

Example 2.8. Calculate the speed of the motor shaft in rpm by the change in position pulses and known PPR of the encoder.

Encoder PPR: 2000
Pulses recorded in time interval of 1 ms: 10

The speed in rpm is estimated by measuring the number of pulses in a pre-defined time interval t_base (in ms). It is calculated by

$$\text{rpm} = \frac{\text{Position\_pulses} * 60 * \text{k}}{\text{PPR}}, \text{k} = \frac{1}{\text{t\_base}}.$$

$$\text{rpm} = \frac{10 * 60 * 10^3}{2000} = 300.$$

Example 2.9. Position and speed measurement of different axes in a robot application is done using a dual channel incremental encoder. These encoders use the phase difference between the signal to determine the direction and frequency of a pulse train to determine speed. Convert the block diagram (see Fig. 2.14) to digital logic using Verilog HDL.

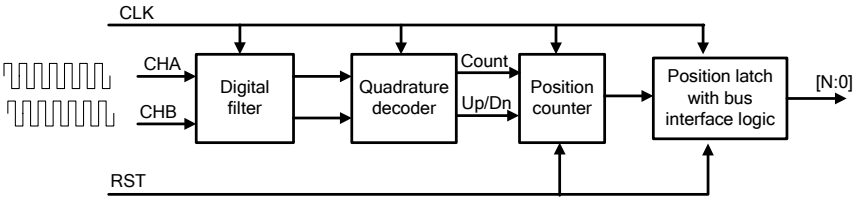


Fig. 2.14. Block diagram for finding the robot axis position using incremental encoder feedback

```
module filter (input clk, input in, input rst . output reg z);

reg a,b,c;

always @ (posedge clk)
begin
  if (rst)begin
                    a <= 0;
                    b <= 0;
                    c <= 0;
                    end
        else
        begin
                    a <= in;
                    b <= a;
                    c <= b;
        end
end

always @ (posedge clk)
        begin
        if (( a == b ) & ( b == c))
                    z = a;
        end

endmodule
```

Fig. 2.15. Verilog code for digital filter circuit

The function of the digital filter circuit is to ensure that the signal remains constant for three clock pulses before it is sent to the output. Any noise less than three clock cycles is ignored by the circuit. A three-stage shift register realised using the HDL code in Fig. 2.15 is used to check if the output of the third stage (c) matches the second (b) and the first (a) stages. HDL code for a single channel encoder based position counter is shown in Fig. 2.16. A FSM is used to check for transitions of the encoder output and the counter increments synchronously.

```
module position (input clk, in,rst, output reg [7:0] position );

reg state, next_state;

parameter one = 1'b1;
parameter zero = 1'b0;

always @ ( in or state or rst)
        begin
                if (rst)
                begin
                        position = 0;
                end
        else
                begin
                case(state)

                one : if (in)
                        begin
                                position = position +1;
                                next_state = zero;
                        end
                zero :if (~in)
                                next_state = one;
                default : next_state = zero;

                endcase
                end
        end

always @ ( posedge clk)
        begin
                state <= next_state;
        end

endmodule
```

Fig. 2.16. Position counting register for one encoder channel

For a given encoder PPR, the resolution of the position measured can be increased by counting the active edges of A and B channels of the encoder. The resolution is increased four times (theta_4x) when both active edges of channels A and B are counted (see Fig. 2.17).

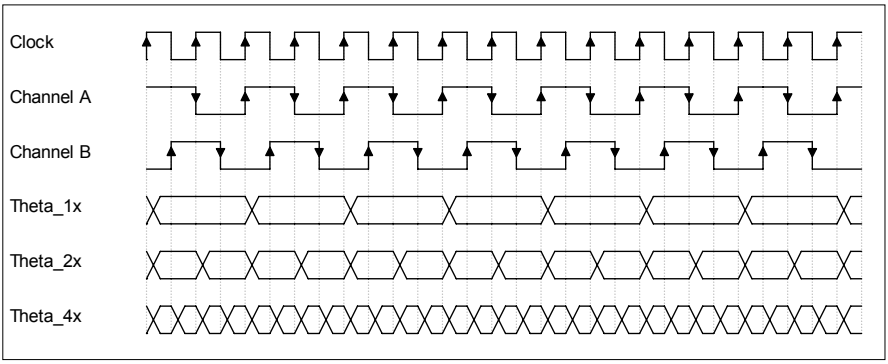


Fig. 2.17. Increasing encoder resolution by pulse edge detection

Example 2.10. The position determined by an incremental encoder has to be made part of the non-volatile memory of the robot controller. Many present-day robot systems have shifted to absolute encoders, where the present position of the axis is available on power up. A Gray coded absolute encoder is used for determining the robot axis position. In Gray code, only 1 bit changes at the time of each transition. The logic used for conversion of Gray to binary is shown in Fig. 2.19. A Verilog code using blocking assignment is used for converting 5-bit Gray to 5-bit binary.

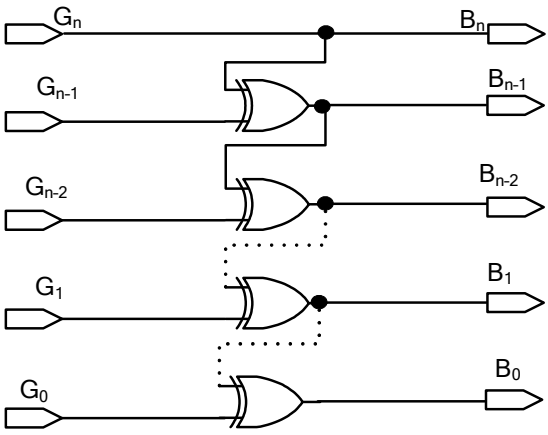


Fig. 2.18. Converting from Gray code to binary code

```
module gray(input [4:0] gray, output reg[4:0] binary);

always @(gray)
begin
        binary[4] = gray[4];
        binary[3] = gray[3] ^ binary[4];
        binary[2] = gray[2] ^ binary[3];
        binary[1] = gray[1] ^ binary[2];
        binary[0] = gray[0] ^ binary[1];
end

endmodule
```

Fig. 2.19. Verilog code for Gray to binary conversion

2.3.5 Arithmetic Using HDL

Most arithmetic operations performed using HDL are either in terms of integers or fixed-point arithmetic. Arithmetic operations using HDL-based logic provides flexibility to choose the word size of the operation based on application requirements. In arithmetic operations involving fixed-point arithmetic, it is upto the designer to interpret the position of the decimal point. Most synthesis tools[13] support the Verilog signed and unsigned data format for wire and register types.

As shown in Fig. 2.20a, the data range reduces by a factor of 2 in signed representation because 1 bit is allotted for the sign bit.

2.3.5.1 Integer Data

Real world inputs and outputs are usually in the form of integer data. Digital data from ADC or to a DAC are in the form of integers. The unsigned integer type shown in Fig. 2.20b is used for presenting timer values, whereas the signed integer type is used for representing the sine wave shown in Fig. 2.20c.

2.3.5.2 Fixed-point Data

For a fixed-point number represented as **aaa.bbbbb**, the integer part consists of three bits of data and the fractional part is represented by five bits. These 8 bits (3 integer and 5 fractional) can represent unsigned binary integer values from 0 to 7, having a fractional resolution of $1/2^b$.

Example 2.10. Choose a fixed-point representation for getting the maximum resolution for a sine wave, whose peak amplitude varies from +2 to −2 volts. The available bit range is 8 bits.

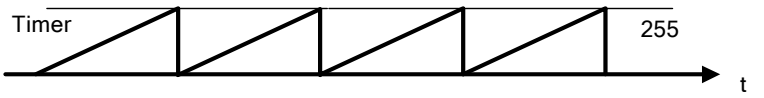
The maximum resolution is obtained by using a fixed-point data format (covering +1.xxxx to −1.xxxx) , which has one sign bit, one integer bit and the rest are fractional bits. The fixed-point notation for the above example is **aa.bbbbbb.** This provides a fractional resolution of $1/2^6$.

---

[13] The Xilinx synthesis technology (XST) provides support for signed and unsigned data types.

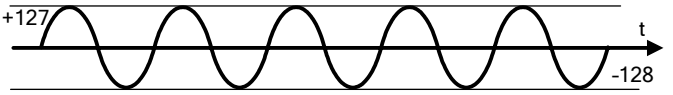
```
wire signed [ width-1 : 0] variable; // data in 2's complement format
wire unsigned [ width -1 : 0] variable ;

reg signed [ width-1 : 0] variable; // data in 2's complement format
reg unsigned [ width -1 : 0] variable ;
```

a



b

Signed integer data — sine wave with amplitude +128 to –127



c

Fig. 2.20. a Signed and unsigned representation using wire and register; **b** 8-bit unsigned representation of a Timer signal; **c** 8-bit signed representation of a sine wave

If the data are represented as a fraction, it is best to represent the number in the Q format by scaling all the numbers to a range of +–1. Q format specifies one sign bit followed by a number of bits for fractions. Once the number is in Q format it becomes easier to work with. For computations the input value is scaled to the format giving maximum resolution and then scaled back to the original format at the time of output.

2.3.5.3 Addition and Subtraction

In digital circuits, addition and subtraction operations are carried out using 2's complement arithmetic. As shown in Fig. 2.22, a common hardware circuit is used for both addition and subtraction. Though Verilog HDL synthesis supports addition and subtraction, the methodology may vary from one synthesis tool to another. The implementation uses the logic block of the target FPGA technology to implement arithmetic operations.

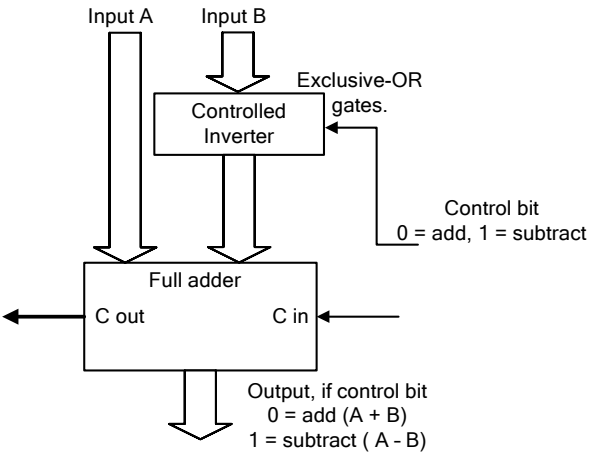


Fig. 2.21. Common hardware for performing signed addition and subtraction

When the control bit becomes 1, input B takes its 2's complement value, resulting in A – B operation.

Table 2.3. Binary fixed value addition and subtraction

Operand	Binary value	Integer	Scale factor	Result — real fixed-point value
A	00011.110	+30	$2^{-3} = 1/8$	+3.750
B	00110.011	+51	$2^{-3} = 1/8$	+6.375
A + B	01010.001	+81	$2^{-3} = 1/8$	+10.125
A – B	11101.011	–21	$2^{-3} = 1/8$	–2.625

As shown in Table 2.3, when adding or subtracting two fixed-point real numbers that have a common scale factor, the result will also have the same scale factor. When the scale factors are different, proper shifting of the binary point needs to be done so that the scale factors become equal. The Q notation is helpful because it scales all numbers to a common scale factor.

Example 2.11. Write HDL code for signed addition and subtraction using signed arithmetic Verilog constructs.

The input variables a1, b1 and c1 are converted to 2's complement representation by the **wire signed** declaration (see Fig. 2.22). This instructs the simulation and synthesis software to use 2's complement arithmetic for the addition operation.

```
module signed_arith (input [7:0] a,b, output [7:0] c);

wire signed [7:0] a1,b1,c1;

assign a1 = a;
assign b1 = b;

assign c1 = a1 + b1;

assign c = c1;

endmodule
```

Fig. 2.22. Verilog code for performing signed addition

2.3.5.4 Multiplication

Multiplication is supported by HDL simulation and synthesis software. Multiplication and division operations do not require that the binary points of the operands be aligned. The number of fractional bits in the product A * B, is equal to the sum of the number of fractional bits of A plus the fractional bits of B. This is demonstrated by the example in Table 2.4.

Table 2.4. Binary fixed value multiplication

Operand	Binary value	Integer	Scale factor	Result — real fixed-point value
A	00111.110	62	$2^{-3} = 1/8$	+7.750
B	001100.11	+51	$2^{-2} = 1/4$	+12.750
A * B	1100010.11010	+3162	$2^{-3-2} = 1/32$	+98.8125

Many FPGA chips now contain hardware multipliers. In the absence of these multipliers, signed multiplication algorithm such as the Booth multiplier can be used.

Example 2.12. Write Verilog code to multiply[14] two 8-bit unsigned and two signed numbers A and B.

The inputs (**a,b)** to codes in Fig. 2.23 and 2.24 are the same. In Fig. 2.24, the inputs are interpreted as signed numbers, because of the **wire signed** assignment.

[14] Multiplication is supported by both simulation and synthesis tools. XST supports both signed and unsigned multiplication. Details of the multiplier are given in Sect. 3.2.6

```
module multiply ( input [7:0] a , b , output [15:0] c );

assign c = a * b ;

endmodule
```

Fig. 2.23. Verilog code for unsigned multiplication

```
module signed_mult (input [7:0] a,b, output [15:0] c);

wire signed [7:0] a1,b1;
wire signed [15:0] c1;

assign a1 = a;
assign b1 = b;

assign c1 = a1 * b1;

assign c = c1;

endmodule
```

Fig. 2.24. Verilog code for signed multiplication

Division by a non-power of 2 is not supported by most synthesis tools. Division in hardware is carried out by multiplying with the reciprocal of the divisor.

While performing arithmetic operations with pre-defined word size, conditions such as overflow or loss of precision arise. It is up to the designer to correct such conditions by truncation, rounding or saturation. Table 2.5 shows examples of overflow, truncation and rounding errors arising during addition and multiplication.

Table 2.5. Rounding, truncation, overflow and saturate

Input A	Input B	Operation	True Result	Conditioning
1111 1110 (254_{10})	0000 0011 (3_{10})	Addition (8-bit result)	1 0000 0001 (257_{10})	0000 0001 (overflow - 1_{10}) 1111 1111 (saturate- 255_{10})
00111.111 (7.875_{10})	001100.11 (12.750_{10})	Multiply (9-bit result)	1100100.01101 (100.40625_{10})	1100100.01 (truncate - 100.25_{10}) 1100100.10 (rounding - 100.50_{10})

2.4 Pipelining

Pipelining is used to divide a large section of logic into small parts. It is much like the assembly line concept used in manufacturing. If a process can be broken down into small equal time sections, the speed of the process will be equal to the time taken by each section. The process of pipelining is akin to the process discussed as that of the shift register. The information from one section of the pipeline flows to the next section on the active edge of the clock controlling the pipeline. Thus non-blocking statements in Verilog are used for modelling pipeline architectures.

2.5 Module Instantiation and Port Mapping

A robot system having five control joints needs five motor controllers. These motor controllers are incorporated and interconnected using module instantiation and port mapping. If the motor of a given type of AC/DC servomotor is the same, the control scheme does not change with the rating of the joint motor. Thus, one can clone the digital control circuit. The process of duplicating or cloning digital logic is instantiation. Each instantiated piece of logic operates independently of the other in a concurrent manner. The way the motor ports — consisting of encoder feedback, speed reference and PWM output are connected is called port mapping (see Fig. 2.25). The port list for control of one motor is shown in Fig. 2.26.

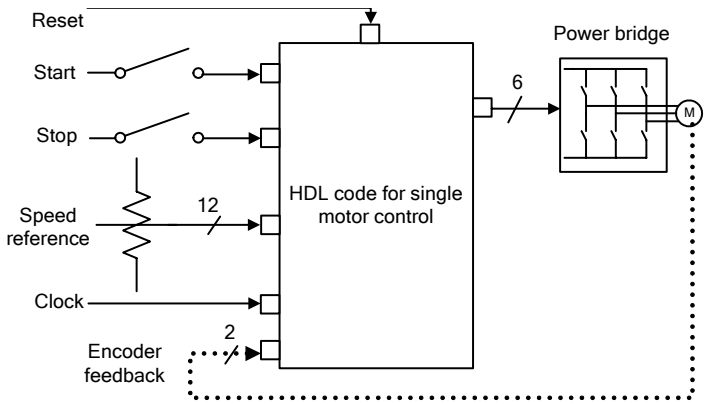


Fig. 2.25. Diagram showing HDL port map for a single motor control

```
module motor_control (input start,stop,clk,rst,
                      input [1:0] enc_fdbk,
                      input [11:0] spd_ref,
                      output [5:0] pwm_switches);

/* Code for performing control of one motor.
   Motor power circuit consists of 6 switches for control */

endmodule
```

Fig. 2.26. Verilog port list for control of one motor

A module port can be connected to external signals either by using the port names in the order contained in the port list, or they can be connected using port names. Instantiation for the port list (see Fig. 2.26) of the motor controller by name is shown in Fig. 2.27. This is a practical approach, where there are numerous ports and remembering the port list order is difficult.

```
motor_control m1 (
      .start(start),
      .stop(stop),
      .clk(clk),
      .rst(rst),
      .spd_ref(spd_ref),
      .enc_fdbk(enc_fdbk),
      .pwm_switches(pwm_switches)
      );
```

Fig. 2.27. Verilog instance for creating a motor control module

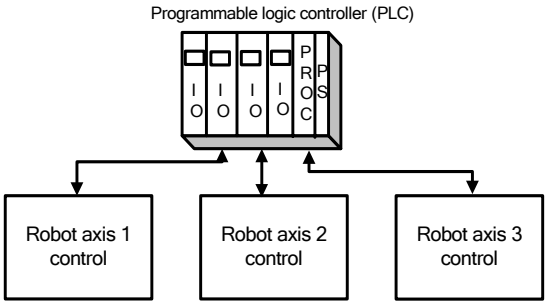
Example 2.13. Show how one motor control module created using Verilog HDL can be instantiated for simultaneous control of four motors. (see Fig. 2.28).

One advantage of using hardware-based design is the freedom of replicating a given design many times (provided there are sufficient hardware resources to support that many number of replications), without worrying about the increase in CPU utilization. The instantiation scheme shown in Fig. 2.28, shows how a HDL designed motor controller hardware similar to Fig. 2.26 is used four times for controlling four different joints of the robot assembly. Each instantiation is denoted by the module name (**motor_control**) followed by the instance name (**mx**), where x denotes the motor number. The port list of the instantiated modules consists of the port name of the HDL design followed by the signal it is connected to. In this example the four motor modules have different start, stop, speed reference, speed feedback and power bridge control signals. The Verilog top module (**robot_motors**) has control signals for different motors coming from a position controller (chapter 6 will discuss different types of controllers).

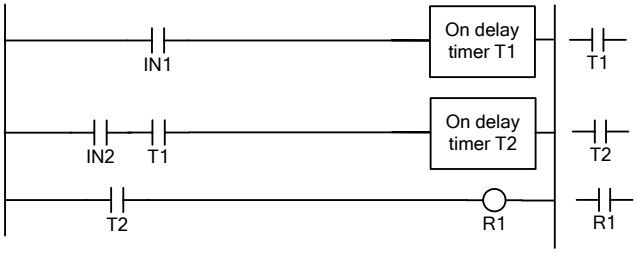
```
module robot_motors  ( input start1,start2,start3,start4,stop1,stop2,stop3,stop4,clk,rst,
                input [11:0]   spd_ref1,spd_ref2,spd_ref3,spd_ref4,
                input [1:0]    spd1,spd2,spd3,spd4,
                output [5:0]   pwm_m1,pwm_m2,pwm_m3,pwm_m4 );

    motor_control m1 (
    .start(start1),
    .stop(stop1),
    .clk(clk),
    .rst(rst),
    .spd_ref(spd_ref1),
    .spd_fdbk(spd1),
    .pwm_switches(pwm_m1)
    );

    motor_control m2 (
    .start(start2),
    .stop(stop2),
    .clk(clk),
    .rst(rst),
    .spd_ref(spd_ref2),
    .spd_fdbk(spd2),
    .pwm_switches(pwm_m2)
    );

    motor_control m3 (
    .start(start3),
    .stop(stop3),
    .clk(clk),
    .rst(rst),
    .spd_ref(spd_ref3),
    .spd_fdbk(spd3),
    .pwm_switches(pwm_m3)
    );

    motor_control m4 (
    .start(start4),
    .stop(stop4),
    .clk(clk),
    .rst(rst),
    .spd_ref(spd_ref4),
    .spd_fdbk(spd4),
    .pwm_switches(pwm_m4)
    );

endmodule
```

Fig. 2.28. Verilog top module code to control four motors

Example 2.14. A programmable logic controller (PLC) based control is shown in Fig. 2.29a. This is used to perform sequencing and interlocking of many industrial control functions. A timer, counter and normally open (NO), normally closed (NC) relays are part of the ladder logic instruction set library for programming a PLC. Convert the ladder logic shown in Fig. 2.29b to digital logic by using Verilog HDL.



a



b

Fig. 2.29. a PLC based scheme for sequencing and interlocking of robot axes; **b** A section of the ladder logic code running in the PLC

When Input 1 (IN1) turns high, an on-delay timer T1 turns on after a pre-defined time interval. Timer T2 is enabled when Input 2 is high and T1 is on. After the preset time interval of T2 is over, the contacts of relay R1 are turned on.

```
module plc_timer (input rst,clk,input [7:0] timer_set, input timer_en,output reg dn);

reg [7:0] timer;

always @ ( posedge clk)
        begin
                if (rst)
                        timer <= 0;
                else if (timer_en)
                        timer <= timer +1;
                else
                        timer <= timer;
        end

always @ (posedge clk)
        begin
                if ( timer_set == timer )
                        dn <= 1;
                else
                        dn <= 0;
        end

endmodule
```

Fig. 2.30. Verilog code for on-delay timer implementation

```
module plc_top (input clk, rst,in1,in2, input [7:0] timer1,timer2,output r1);

        wire timer1_en = in1;
        wire timer1_dn;
        wire timer2_en = timer1_dn & in2 ;


            plc_timer T1 (
                        .rst(rst),
                        .clk(clk),
                        .timer_set(timer1),
                        .timer_en(timer1_en),
                        .dn(timer1_dn)
                        );

            plc_timer T2 (
                        .rst(rst),
                        .clk(clk),
                        .timer_set(timer2),
                        .timer_en(timer2_en),
                        .dn(timer2_dn)
                        );

        assign r1 = timer2_dn ;

endmodule
```

Fig. 2.31. Verilog code to implement the ladder logic consisting of two timer elements

The implementation of ladder logic consists of two timer blocks and an interlocking circuit. Figure 2.30 shows the timer module and 2.31 shows the entire ladder logic circuit put together using HDL.

2.6 Use of Pre-designed HDL Codes

Several FPGA vendors provide a library of pre-designed HDL codes for re-use. Each pre-designed piece of digital logic is either used as per the given port map by the digital core vendor or can be connected to a standard chip-wide interface bus. Some commonly available pre-designed codes include memory interface, serial communications and arithmetic core.

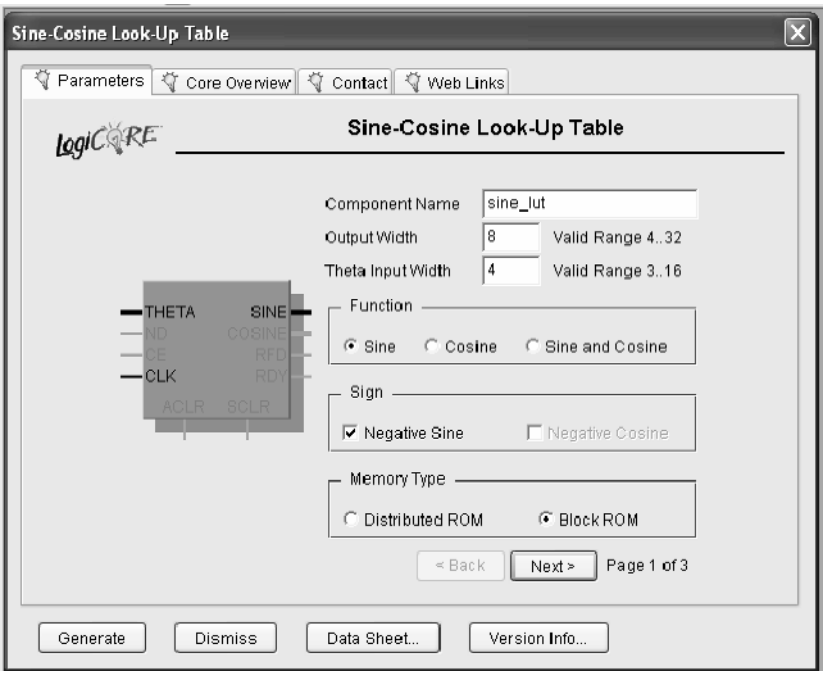


Fig. 2.32. Configuration process for a sine look-up table

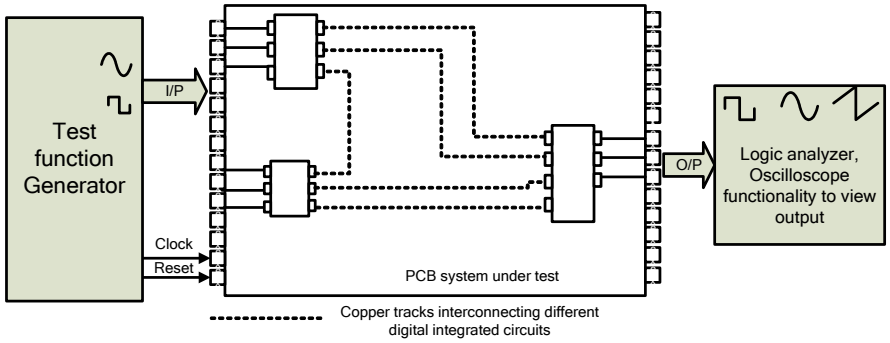
Example 2.16. Create a code that generates a sine wave using the pre-designed sine-cosine look-up table[15] shown in Fig. 2.32.

A HDL 4-bit counter module output connects with the Theta port of the instance of the pre-designed LUT (see Fig. 2.33). The sine_lut is the instance of the core that is connected to a variable count and output signal sine.

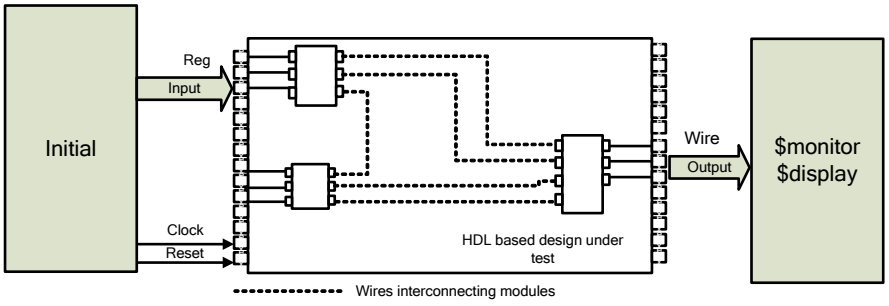
[15] Trignometric functions such as the sine function are frequently used in embedded systems. This example shows the use of a pre-designed sine LUT.

```
module sine_top (input clk,rst, output reg [7:0] sine);

reg [3:0] count;

always @(posedge clk)

begin
        if (rst)
                count <= 0;
                else
                count <= count + 1;
        end

sine_lut1 m1(
                .THETA(count),
                .CLK(clk),
                .SINE(sine));

endmodule
```

Fig. 2.33. Pre-designed sine LUT-based sine wave generator



a



b

Fig. 2.34. a Electronic test bench analogy of HDL simulation; **b** typical simulation environment in Verilog HDL

2.7 Simulating Digital Logic Using Verilog

The motivation behind HDL was to aid digital modeling and simulation. The simulation environment is akin to a test bench with provision for stimulus to the digital circuit and monitoring the circuit's output. In hardware prototyping, this is comparable to PCB-based logic surrounded by various pieces of electronic test equipment. As shown in Fig. 2.34a a PCB under test has inputs from a function generator and the output is analyzed using logic analyzers and oscilloscopes. Similarly, the digital circuit shown in Fig. 2.34b is subjected to test vectors and the output is displayed or monitored using waveforms. Test benches are written using Verilog constructs or a test waveform tool is used.

2.7.1 EDA Tool Flow for Simulation

Simulation of HDL-based digital design is supported by a variety of EDA tools. Tools from FPGA vendors give the user an option to choose the simulation environment they wish to use. Figure 2.35 shows the various simulation environment options.

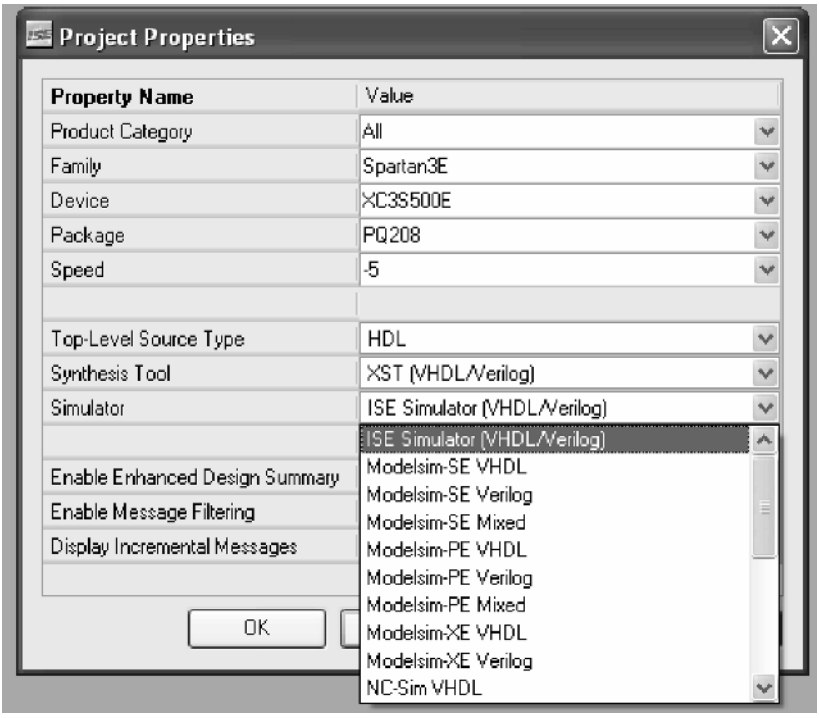


Fig. 2.35. A partial view of simulator selection using a FPGA design tool

The majority of the electronic design automation (EDA) tools for HDL simulation work along similar lines.

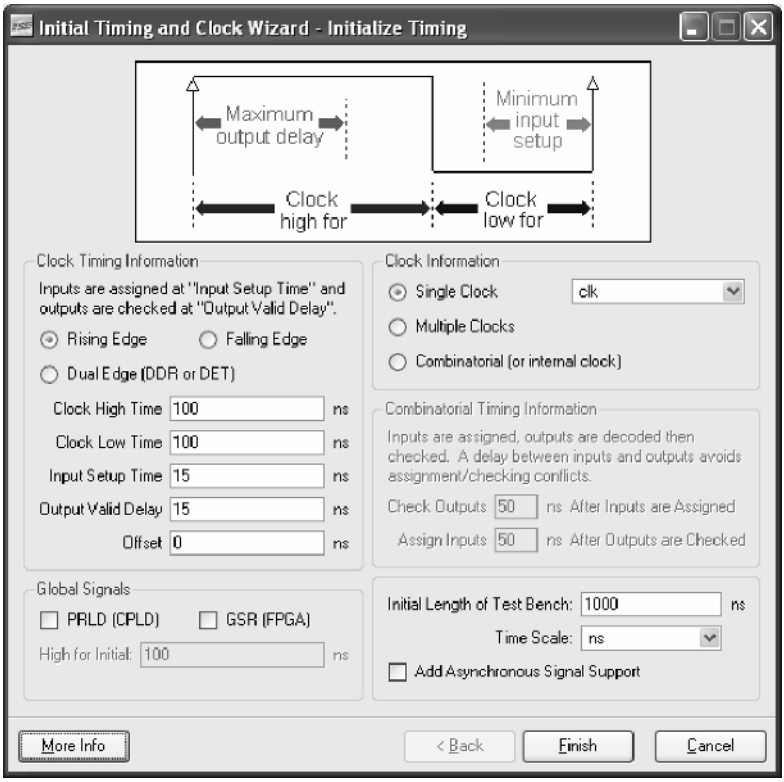


Fig. 2.36. Test bench specifications for clock period, input setup time, output valid delay and test bench length

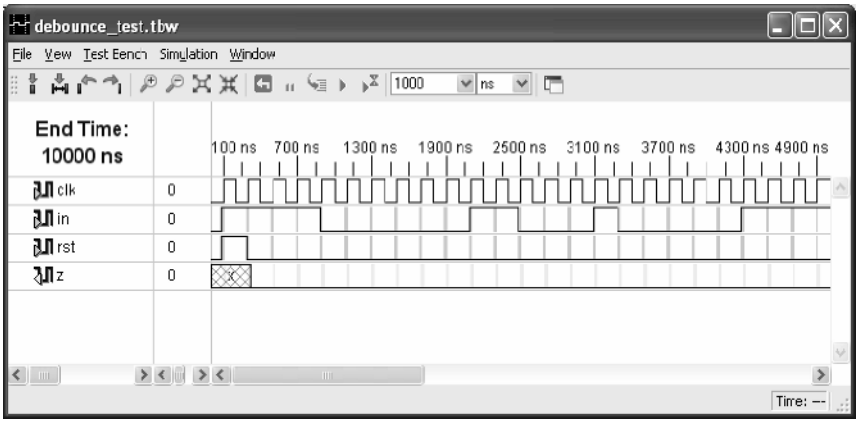
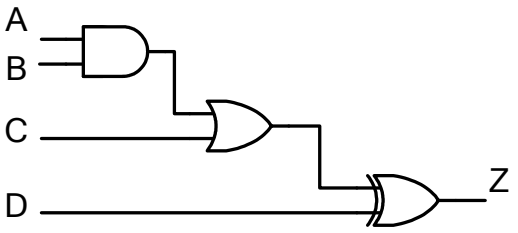


Fig. 2.37. Generating a test waveform for testing the filter circuit given in Fig. 2.15

2.7.2 Creating a Test Bench for HDL-based Digital Logic

HDL simulation tools provide different ways of specifying test bench requirements. One way of checking HDL code quickly is to draw the desired test inputs graphically, that are converted into equivalent Verilog test code (see Figs. 2.36 and 2.37).

As the design size becomes larger, it is not feasible to generate test waveforms for testing all possible combinations of digital inputs. For this purpose a text-based test waveform file can generate test vectors and also log output information from digital logic.



A	B	C	D	Z
0	0	0	0	0
0	0	0	1	1
:	:	:	:	:
1	1	1	0	1
1	1	1	1	0

Fig. 2.38. Combinational circuit and its input, output truth table

To test the combinational circuit (see Fig. 2.38) that will have 16 possible combinations of inputs, a self-checking test bench is created using a text file to read vectors (containing stimulus and expected values).

The contents of the text file are read by the testbench using the **$readmemb**, where the **b** specifies that the test inputs are in binary format. A sample test file would consist of the inputs A–D, and the expected output Z shown in the table on the right of Fig. 2.38.

2.7.3 Post Place and Route Simulation

The behavioural simulation model supports both synthesizable and non-synthesizable constructs of HDL. A partial list of Verilog HDL constructs, which cannot be realised in hardware by use of synthesis tools, is given in Table 2.6. To verify that the HDL-coded behavioural model would work in the same manner after synthesis and its placement in FPGA logic, post place and route simulation is used. The post place and route simulation uses an .SDF file to estimate the interconnect path and logic delays. Any change in the location of logic cells changes the post place and route interconnect delay. The post and route simulation provides a realistic picture of the timing, once the digital circuit is realised in hardware.

Table 2.6. Partial non-synthesizable constructs of Verilog[16]

Verilog constructs not supported or ignored by FPGA synthesis tools	
Data types	Wand, wor, triand, trior real, realtime, tri0, tri1, trireg
Continuous assignment	Drive strength, delay
Procedural assignments	Force, release, forever, fork/join, delay (#), event(@), wait, named events
Compiler directives	Timescale, uselib, resetall, celldefine, endcelldefine
Gate level primitives	Pulldown, pullup, drive strength, delay
Switch level primitives	Cmos, nmos, pmos, rcmos, rnmos, rpmos rtran, rtranif0, rtranif1, tran, tranif0, tranif1
User defined primitives (UDPs)	Both combinational and sequential

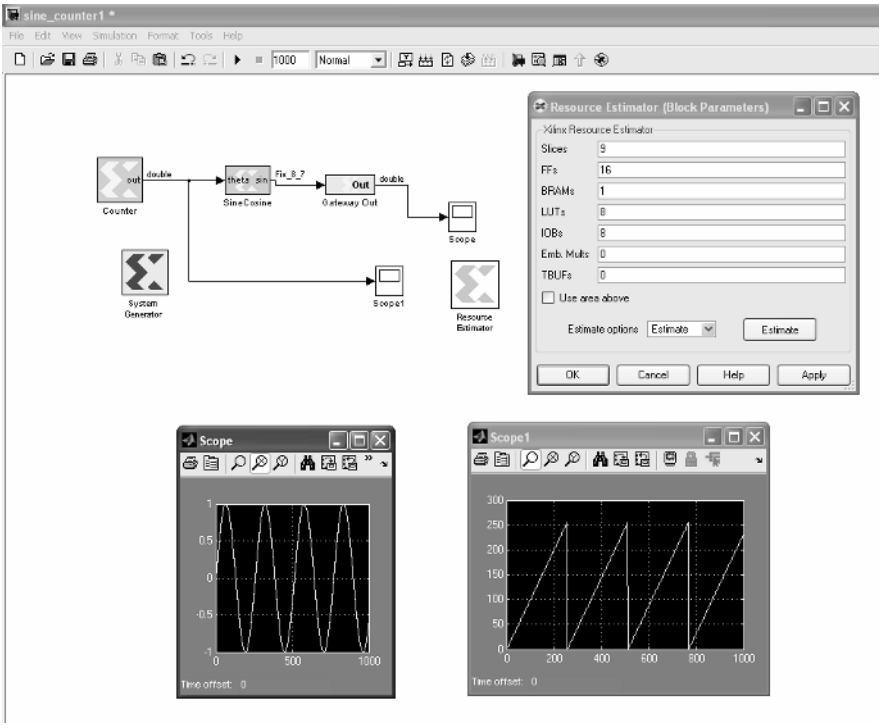


Fig. 2.39. Combinational circuit

[16] These constructs are used for simulation. Synthesis tools do not generate a FPGA-based hardware for these constructs.

2.7.4 Simulation of Algorithm Using Pre-designed Codes

Simulation of a filter or any other DSP system is important to analyze before the system is put in hardware. The MATLAB® Simulink® environment allows simulating a system using standard input/output devices such as waveform generators, oscilloscopes and FFT analyzers. Using this interface frees the designer from writing cumbersome test benches and then converting the result to viewable analogue format.

The pre-designed sine wave generator of Example 2.15 is simulated in the MATLAB® Simulink® environment. A counter-based ramp signal is used to increase the value of theta. In Fig. 2.39 *Scope* and *Scope1* show waveforms of generated sine waves and of corresponding counter values. Because the majority of pre-designed cores are synthesizable, the designer also gets an estimate of the resources used on the FPGA chip.

Problems

1. Write Verilog HDL code for an 8-bit Booth multiplier that works with and without a clock. Write a self-testing test bench which can check the correct working of the Booth multiplier.
2. Understand the working of a parallel-to-serial shift register chip 74LS165. Write a Verilog HDL code that mimics the functionality of the chip. Verify that it is working by comparing the simulated timing diagram of the HDL code and that given in the 74LS165 data sheet.
3. A CNC lathe machine uses a 5000 PPR incremental encoder for determining the speed of the cutting tool. Write a Verilog code to determine the speed of the cutting tool using the encoder input. The calculated speed is to be displayed on seven segment LEDs.

Further Reading

1. Chonnad SS, Balachander NB (2004) Verilog frequently asked questions, language, applications and extensions. Springer
2. Ciletti M (2003) Advanced digital design with the Verilog HDL. Pearson Education
3. Palnikar S (1996) Verilog HDL – a guide to digital design and synthesis. Pearson Education
4. Navabi Z (1999) Verilog digital system design. McGraw-Hill
5. Riesgo T et al (1999) Design methodologies based on hardware description languages. IEEE Transactions on Industrial Electronics, 46(1): 3–12

3

FPGA Devices

FPGA devices have grown in density from a few thousand gates in the 1980s to approximately 10 million gates in 2004. There are proven advantages in choosing a FPGA-based implementation of digital logic over a fixed custom implementation of digital logic. The advantages include cost economies when a product is produced in low volume, in-system re-programmability and a shorter design cycle from concept to silicon. Most of the contemporary FPGAs from various vendors have common on-chip resources. The purpose of this chapter is to talk about various on-chip resources or primitive devices of FPGAs and their use in synthesising digital systems.

These device hardware primitives are comparable to assembly language constructs of a general purpose processor, invoked by the compiler. Though one may rarely use programming at the primitive device level using HDLs, it is good to know the underlying hardware, used by the synthesis tool for realising digital logic[17]. The idea is to try and understand how these hardware resources can be used to implement a single-chip robot controller module that can support motor control algorithms, processor and interfacing needs.

3.1 FPGA and CPLD

Programmable hardware basically consists of two widely used programmable devices — FPGAs and CPLDs. These devices determine a hardwired circuit and the programmability of a processor-based system. With several technologies merging and overlapping, it is becoming increasingly difficult to label a programmable device as a CPLD or FPGA. CPLDs are getting features of FPGAs

[17] Many applications still use processor ISA dependent assembly language for compact code size and fast execution. Similarly, many FPGA-based soft processors are coded using device dependent primitives for a smaller silicon footprint.

and vice versa[18]. CPLD architecture consists of one or more programmable sum-of-products logic arrays connected to a clocked register. Traditionally, the CPLD is known to have lower logic cell density compared to a FPGA. Normally, CPLD devices use flash memory technology for interconnecting different logic blocks on the device.

3.2 Architecture of a FPGA

The contemporary FPGA is changing very fast. It is following Moore's law in speed and density and also incorporating lots of functionality along the way. Though FPGAs from different leading vendors such as Xilinx, Altera, Actel, and Lattice differ in some aspects, they all share some common architectural attributes. These architectural attributes can be thought of as device specific primitives, which will be discussed in detail. For illustration, a sample FPGA device, Xilinx SPARTAN-3E^{TM} is chosen, and its structure and device primitives along with the Verilog instance for each primitive are discussed.

Following are the typical features of a contemporary FPGA, shown in Fig. 3.1:

- Logic cell resources, consisting of LUT and FFs
- Hard intellectual property(IP), comprising of dedicated multipliers and embedded memory
- Clock distribution resources, digital clock manager (DCM) providing frequency synthesis and phase shift
- I/O features — number of user available I/Os and I/O standards
- Hardware immersed and software configurable processors, along with logic fabric in a single FPGA device.

Table 3.1 contains a comparison of high-volume FPGAs from three vendors. Table 3.2 details the FPGA resources for the SPARTAN-3E^{TM} FPGA used in the text.

3.2.1 FPGA Interconnect Technology

Re-programmable interconnects make the FPGA device reconfigurable. FPGA interconnect technology consists of switch boxes that route signals between various logic blocks on a FPGA. Present-day FPGAs are based on one of the following interconnect technologies:

[18] HDL design examples mentioned in the text, can also be implemented on CPLD devices. Due to lower density, contemporary CPLDs do not support primitives such as embedded memory and hardware multipliers.

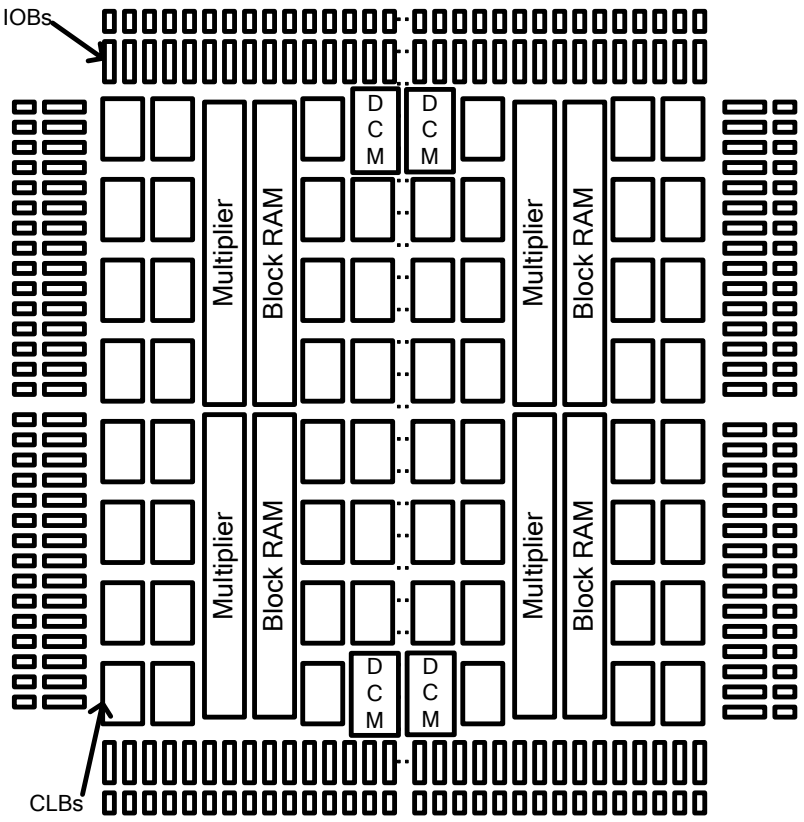


Fig. 3.1. Architecture of a SPARTAN-3E™ FPGA

Table 3.1. Architectural features of contemporary FPGA families from leading vendors

Feature	**Xilinx SPARTAN 3™**	**Altera – Cyclone III®**	**Actel Fusion®**
Combinational and sequential logic block	Logic cell (LC)	Logic element (LE)	Logic element
Embedded memory	Block RAM	RAM blocks	RAM blocks
Phase-locked loop (PLL)	Digital locked loop (DLL)	PLL	Clock conditioning circuits (CCC)
Global clock lines	Yes	Yes	Yes
Hardware multipliers	Yes	Yes	Yes
Interconnect technology	SRAM	SRAM	Flash
Integrated ADC and MOSFET driver	NA	NA	Yes

Table 3.2. Resources of the FPGA used in design examples

Device type	Logic cells	Block RAM (kbits)	Distributed RAM (kbits)	Multipliers	Clock conditioning circuits
XC3S500E-4FG320	10476	360	73	20	4

3.2.1.1 Static RAM (SRAM)
SRAM-based interconnect switches dominate the present-day FPGA market. This technology is used by two of the largest vendors of FPGAs. As the technology type suggests, SRAM-based switches are volatile and lose their configuration on power reset. Because they need to be continuously powered up to maintain their configuration, the quiescent power consumption is quite high. Usually these devices need an extra memory device for loading the configuration to the SRAM switches and registers in every power cycle.

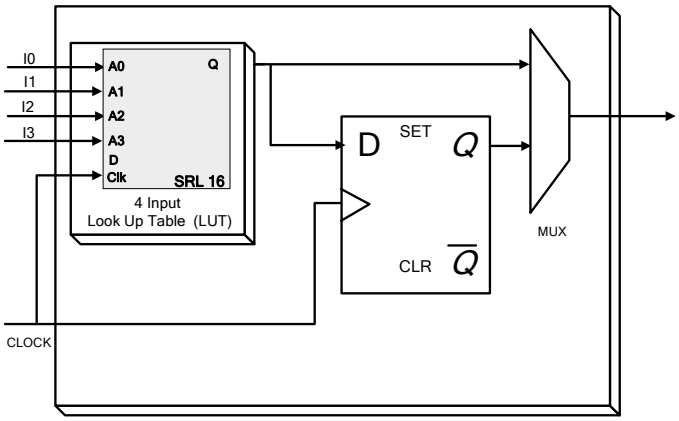
3.2.1.2 Flash
FPGAs using electrically programmable and erasable Flash-based interconnect switches, are non-volatile in their configuration. They use lesser quiescent power than the SRAM interconnect-based devices.

3.2.1.3 Anti-fuse
Anti-fuse interconnect-based devices can be programmed only once. The interconnect of an anti-fuse once programmed becomes permanent. As a result, anti-fuse FPGAs are used mainly for defense and aerospace applications.

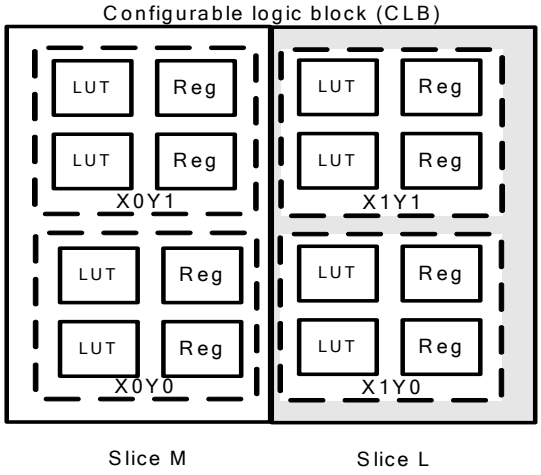
3.2.2 Logic Cell

A digital circuit can be broken down to combinational and sequential elements. Combinational logic consists of basic gates, decoders, encoders and multiplexers. Sequential logic elements consist of clock driven elements such as FFs that act as a storage element for the circuit. With this background, one would expect the logic cell of a FPGA to include elements of combinational and sequential logic for digital circuit realisation. Figure 3.2a shows a logic cell of a FPGA. It consists of a LUT followed by a FF. The LUT is used for realising various combinational circuits such as basic gates, decoder, encoder and multiplexer. The LUT RAM is initialised with the truth table of the desired logic. A four-input LUT contains 16 single bit RAM cells, followed by a multiplexer. These 16 RAM bits can also be configured for use as a shift register or memory. When configured for use as a shift register, the LUT is called a shift register LUT (SRL). Many designs can be mapped directly to the SRL, such that the resource use is optimal.

As FPGA densities are increasing, the configurable logic block (CLB) becomes the container for multiple logic cells. As shown in Fig. 3.2b, the SPARTAN-3E^{TM} CLB consists of two slices, and each slice has four logic cells.



a



b

Fig. 3.2. a Generic nature of one logic cell of a CLB; **b** diagram of the CLB of Xilinx SPARTAN-3^{TM}

Based on the structure of logic cell of the FPGA, certain basic elements are referred to by the synthesis tool. Table 3.3 shows a list of such elements.

Table 3.3. Partial list of FPGA primitives mentioned in synthesis report

	Element type	**Description**
Basic elements (BELs)	GND	Ground , logic 0
	VCC	Power, logic 1
	FDRSE	D Type flip-flop with reset, set and enable
	MUX F5	Multiplexer
	LUT(x)	(x) input look-up table
	SRL	Shift register LUT (SRL 16)
I/O buffer	IBUF	Input buffer
	OBUF	Output buffer
Clock buffers	BUFGP	Global buffer used for clock

Table 3.4. A combinational circuit to be implemented using a LUT

Input A	**Input B**	**Input C**	**Desired output E**
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

```
module lut_prim (input a, input b, input c, output e);

    LUT3 #(
        .INIT(8'hA7 ) // LUT Contents
                    ) LUT3_inst (
                                .O(e), // LUT general output
                                .I0(c), // LUT input
                                .I1(b), // LUT input
                                .I2(a) // LUT input
                );
endmodule
```

Fig. 3.3. Verilog code using the Xilinx LUT primitive for realising combinational logic

Example 3.1. Write Verilog code that uses Xilinx LUT primitive to realise the combinational logic in Table 3.4. The value of the output need not be registered.

The Xilinx LUT primitive is used for realising combinational logic. As shown in Fig. 3.3, the example uses a three input LUT for a function that has eight possible outputs. The value of the output is initialised with the (.INIT) hexadecimal value of **A7** to get the desired output E. The eight-bit value **8h'A7** is transferred to the SRAM of the LUT when the FPGA is configured.

The synthesis report shown in Fig. 3.4 shows one three input LUT used for realising the logic of Example 3.1.

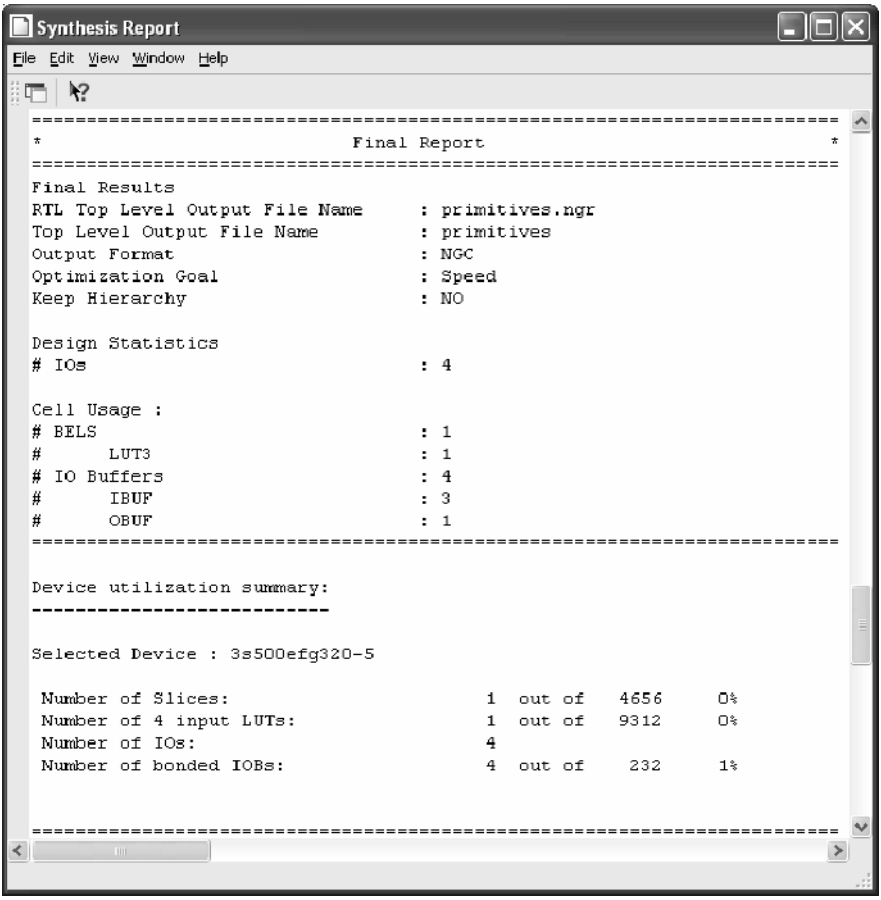


Fig. 3.4. Synthesis report of the LUT function

Example 3.2. The Verilog code (see Fig. 3.5) is used for transmitting ASCII character "A" to the serial port. Which Xilinx primitives would be used to realize this logic?

```
module uart_concate(input clk, output so);

reg [9:0] data = 10'b1010000010;

assign so = data[0];

always @ ( posedge clk ) // Baud rate clock

        begin
                data = {data[0], data[9:1]};
        end

endmodule
```

Fig. 3.5. Verilog code using the shift register logic primitive of Xilinx

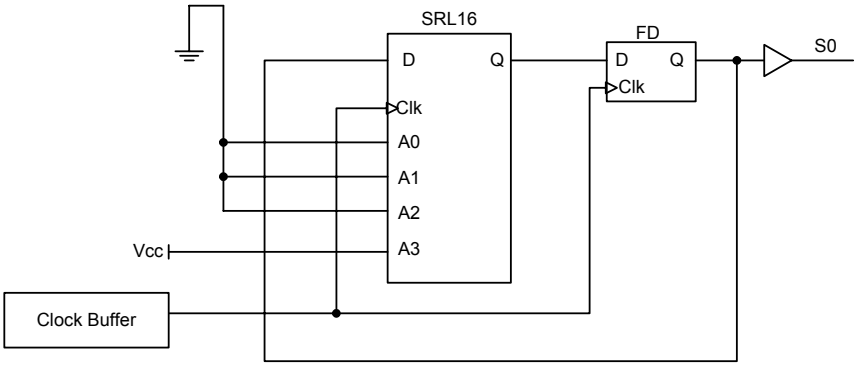


Fig. 3.6. Synthesised version of transmitter code of Fig. 3.5

The data frame for UART transmission consists of a start bit, ASCII data byte, an optional parity bit and a stop bit. In the data frame, the LSB of the ASCII value is positioned first. The data frame of 1010000010 denotes a start bit (0), ASCII character "A" (41_H) and a stop bit (1). The synthesized technology schematic shown in Fig. 3.6 consists of a SRL followed by a FF. This entire logic is contained within a single logic cell.

As pointed out in the synthesis report of Fig. 3.7 the logic delay is 3.492 ns, and the interconnect route delay is zero, because the entire logic is contained in one logic cell.

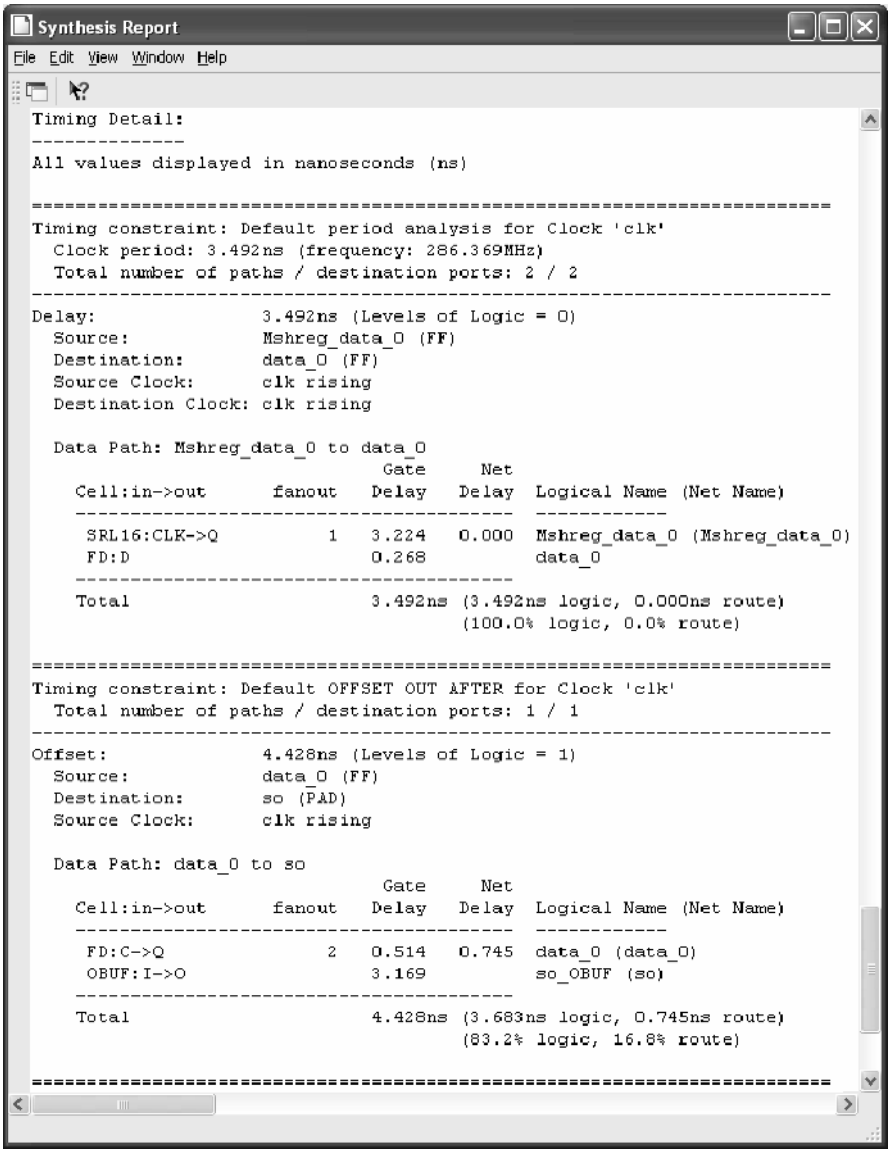


Fig. 3.7. Timing report for SRL16-based logic

3.2.3 FPGA Memory

Besides logic capability, digital systems need memory to store intermediate results of a computation or store pre-computed functions that are not realisable using hardware logic. Present-day FPGA chips contain dedicated blocks of memory or use un-used logic cells to double up as memory elements.

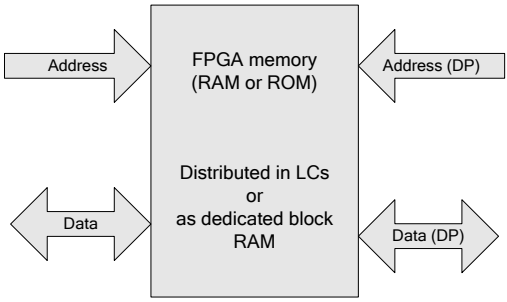


Fig. 3.8. FPGA memory block diagram. The embedded block RAM memory is configurable for either single or dual port (DP) use

3.2.3.1 Distributed Memory

The LUTs contained within the CLB can be used to store information. The LUT of a logic cell can store 16 bits. From the application point of view, the four-input LUT is designed to implement logic. The RAM bits of a LUT, used for storing memory bits, are called distributed memory.

Table 3.5. Partial table containing values of theta for one axis

Theta axis θ_1 (Time)	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
T1	0	0	0	0
T2	0	0	0	1
T3	0	0	1	0
T4	0	0	1	1
T5	0	1	0	0
T6	0	1	0	1
T7	0	1	1	0
T8	0	1	1	1
T9	1	0	0	0
T10	1	0	0	1
T11	1	0	1	0
T12	1	0	1	1
T13	1	1	0	0
T14	1	1	0	1
T15	1	1	1	0
T16	1	1	1	1

Example 3.3. Create digital logic that reads the co-ordinate locations of the robot from distributed memory. The data is stored for one axis to repeat the trajectory several times (see Table 3.5). The output from the memory location serves as a position reference.

```
module learning (input clk,rst,run, output reg [3:0] theta_ref );

reg [3:0] address;
wire A3,A2,A1,A0,D3,D2,D1,D0,O3,O2,O1,O0;



always @ ( posedge clk) // the value of clock is adjusted to real time update rate for run mode
begin
      if (rst)
                  address <= 0;

      else if ( run )
                  begin
                  address <= address + 1;
                  theta_ref <= {O3,O2,O1,O0};
                  end
end

      assign A3 = address[3];
      assign A2 = address[2];
      assign A1 = address[1];
      assign A0 = address[0];

     RAM16X4S #(
          .INIT_00(16'hAAAA), // INIT for bit 0 of RAM
          .INIT_01(16'hCCCC), // INIT for bit 1 of RAM
          .INIT_02(16'hF0F0), // INIT for bit 2 of RAM
          .INIT_03(16'hFF00) // INIT for bit 3 of RAM
     ) RAM16X4S_inst (
          .O0(O0),      // RAM data[0] output
          .O1(O1),      // RAM data[1] output
          .O2(O2),      // RAM data[2] output
          .O3(O3),      // RAM data[3] output
          .A0(A0),      // RAM address[0] input
          .A1(A1),      // RAM address[1] input
          .A2(A2),      // RAM address[2] input
          .A3(A3),      // RAM address[3] input
          .D0(D0),      // RAM data[0] input
          .D1(D1),      // RAM data[1] input
          .D2(D2),      // RAM data[2] input
          .D3(D3),      // RAM data[3] input
          .WCLK(clk), // Write clock input
          .WE(1'b0)    // Write enable input
     );
endmodule
```

Fig. 3.9. Verilog code to read from distributed RAM in run mode

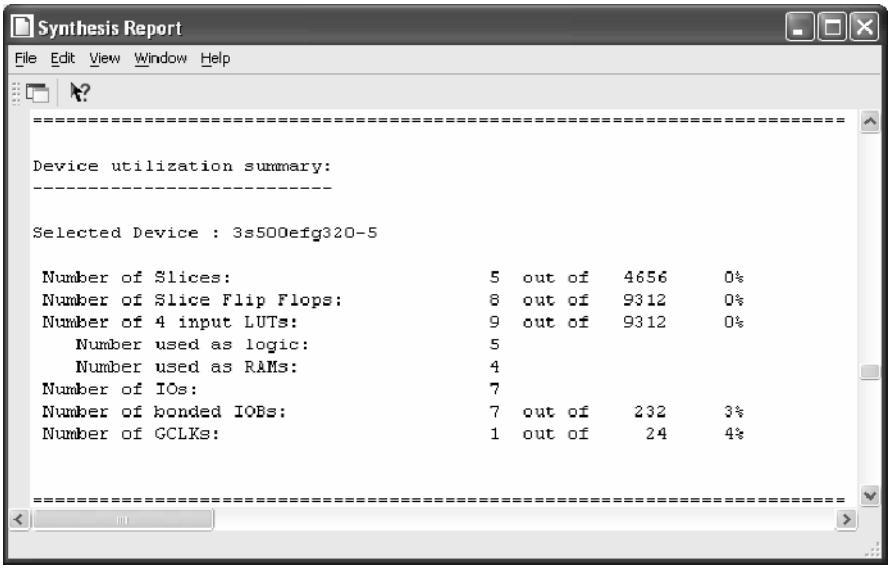


Fig. 3.10. Synthesis report of a distributed RAM , showing the use of four LUTs for 16 * 4 bit memory

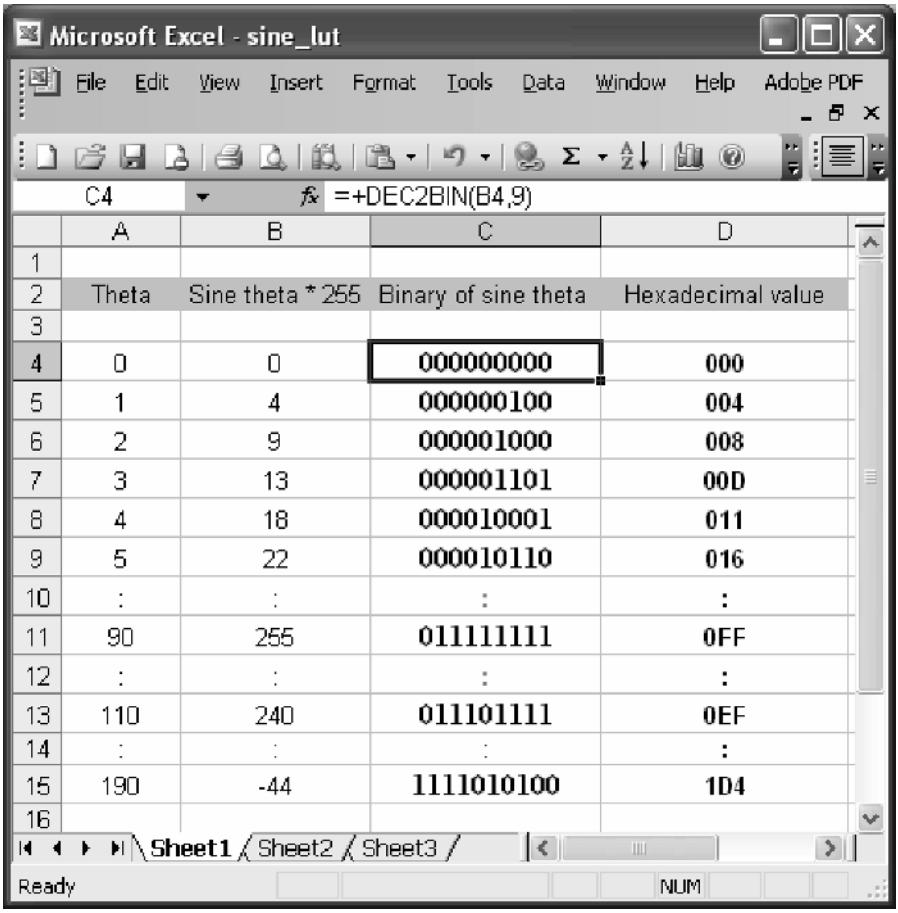
A distributed RAM is used as a ROM by disabling the write mode. When a **run** command is received, the address of the memory increases and the stored contents are sent out as **theta_ref**. The synthesis report (see Fig. 3.10) of the code shown in Fig. 3.9, indicates the use of four LUTs as a RAM that is needed to realise 16 * 4 bit memory.

3.2.3.2 Block Memory

A FPGA contains silicon dedicated exclusively for memory applications. Such memories are used for FIFOs to store constants, coefficients of filters and look-up tables. Most of the dedicated memory provided in a FPGA is dual port. Each side of the dual port memory can independently carry out read and write operations. Dual port memory provides conflict resolution during read-write operations by specifying which operation should succeed.

Example 3.4. Create a LUT-based sine function. The coefficients of the LUT are to be stored in embedded memory.

The value of sine theta is calculated using a spread sheet, such as MSExcel® and the values are exported to the memory editor using the comma-separated values (.csv) format. As shown in Fig. 3.11 column C of the spreadsheet contains 8-bit binary values of sine theta from column **A**. The **DEC2BIN** function is used to convert decimals to binary. Negative numbers are represented using the 2's-complement notation.



	A	B	C	D
1				
2	Theta	Sine theta * 255	Binary of sine theta	Hexadecimal value
3				
4	0	0	000000000	000
5	1	4	000000100	004
6	2	9	000001000	008
7	3	13	000001101	00D
8	4	18	000010001	011
9	5	22	000010110	016
10	:	:	:	:
11	90	255	011111111	0FF
12	:	:	:	:
13	110	240	011101111	0EF
14	:	:	:	:
15	190	-44	1111010100	1D4
16				

Fig. 3.11. Using Microsoft Excel® to generate coefficients of a look-up table. These coefficients are stored in the embedded memory of a FPGA

The numbers generated by the spreadsheet are made part of a coefficient file that is loaded into the memory of the FPGA. The coefficient file (.coe) format consists of radix specification (decimal, hexadecimal or binary) followed by the input variables to be embedded. In the FPGA tool flow, the initial contents for an embedded memory specified by the coefficient file, are embedded in the EDIF netlist, that is generated for implementation. For simulation purposes, a memory initialisation file (.MIF) is generated.

The coefficient file is loaded using the “Load Init File” shown in Fig. 3.12. Alternately, a memory editor shown in Fig. 3.13 is used to create values for storing in memory.

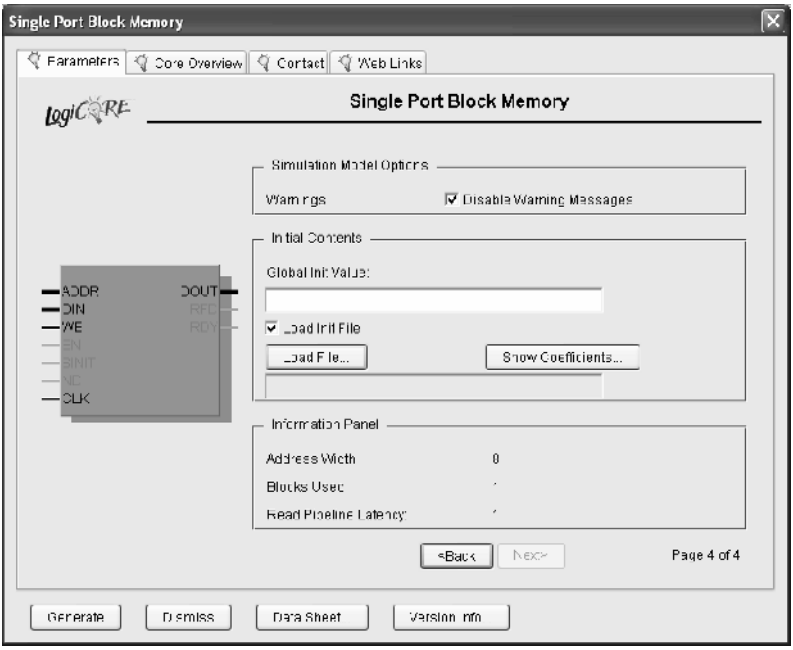


Fig. 3.12. Block memory initialisation with pre-calculated values

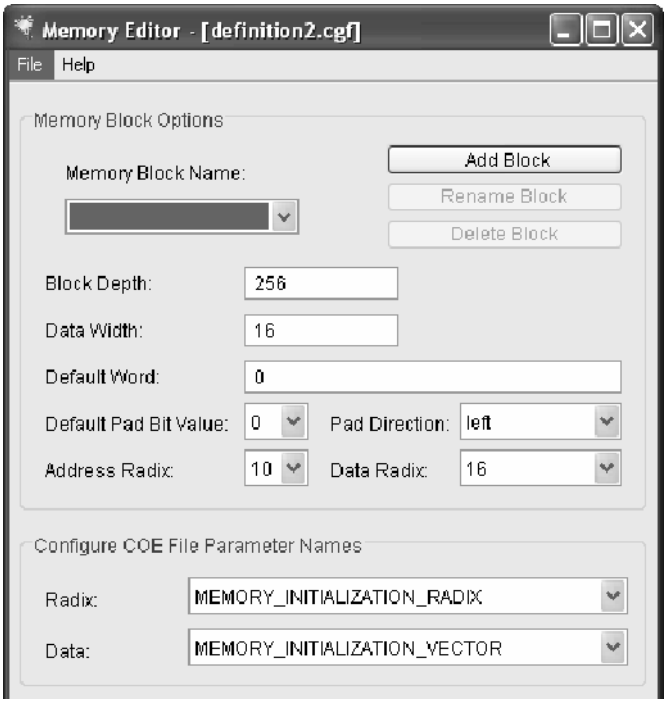


Fig. 3.13. Creation of a coefficient file

3.2.4 Clock Distribution and Scaling

Clock distribution is an important design area of digital design. In a microcontroller, clock distribution and scaling are used to provide a scaled clock to different parts of the controller logic. This scheme is shown in Fig. 3.14. The clock control unit provides the clock to the CPU core, memory and peripherals.

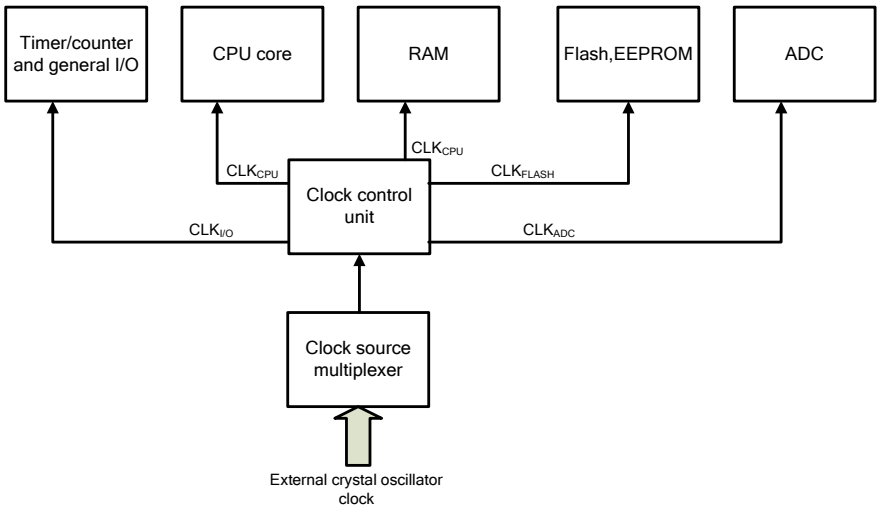


Fig. 3.14. Clock distribution circuit of a microcontroller, showing different clock domains for different blocks of logic/memory.

The clock network on a FPGA consists of a clock spine that connects all flip-flops of the FPGA. Contemporary FPGA devices can accept several global clock inputs that are routed to sequential devices on the chip. A routing diagram of the clock network is shown in Fig. 3.15.

Some sections of logic work at different clock frequencies. One way to provide an accurate clock frequency to different sections is to use an on-chip digital phase-locked loop or a digital clock manager (DCM). The use of a DCM provides a host of functions. It eliminates clock skew, conditions a clock to provide clean output with a 50% duty cycle, provides phase shift and can either multiply or divide an incoming clock frequency (see Figs. 3.16 and 3.17). In the absence of a DCM, a counter logic is used to get fractional clock frequency. This can lead to errors due to skew induced in the clock and less fan-out capability of the logic buffer.

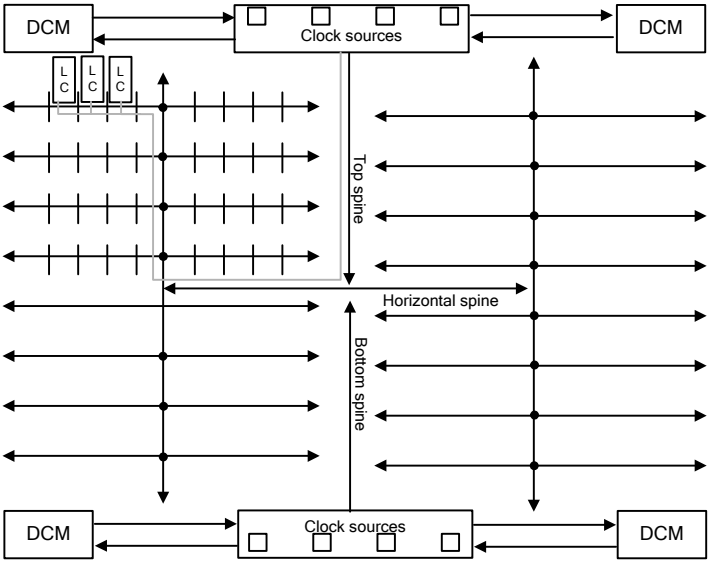


Fig. 3.15. Detail of a routing diagram for a FPGA clock network

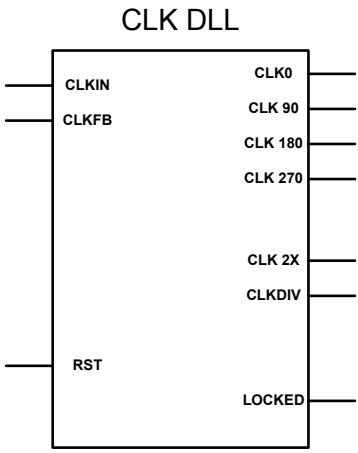


Fig. 3.16. Digital clock manager for creating phase-shifted, divided or multiplied clocks

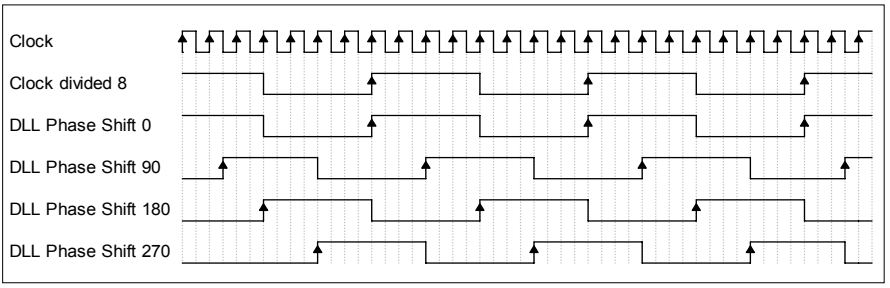


Fig. 3.17. Timing diagram to show the clock manager divide and phase shifting property

Example 3.5. Create a pre-scalar circuit for a timer using a DCM, such that the clock to the timer logic is divided by a factor of 8.

```
module timer_dll(input clk, input [2:0] scaling, input rst, output reg [7:0] timer);

wire clk_timer ;

dcm_timer u1 (
      .CLKIN_IN(clk),
      .RST_IN(rst),
      .CLKDV_OUT(clk_timer),
      .CLKIN_IBUFG_OUT(CLKIN_IBUFG_OUT),
      .CLK0_OUT(clk_0),
      .LOCKED_OUT(LOCKED_OUT),
      .STATUS_OUT(STATUS_OUT)
      );

always @ (posedge clk_timer or posedge rst)

begin
       if (rst)
       timer <= 0;
       else
       timer <= timer+1;
end


endmodule
```

Fig. 3.18. Verilog code showing the use of a DCM for generating a timer clock

The Verilog code, shown in Fig. 3.18 uses the DCM instance to provide a clock divided by a factor of 8. This divided clock is used by the timer circuit shown in Fig. 3.19.

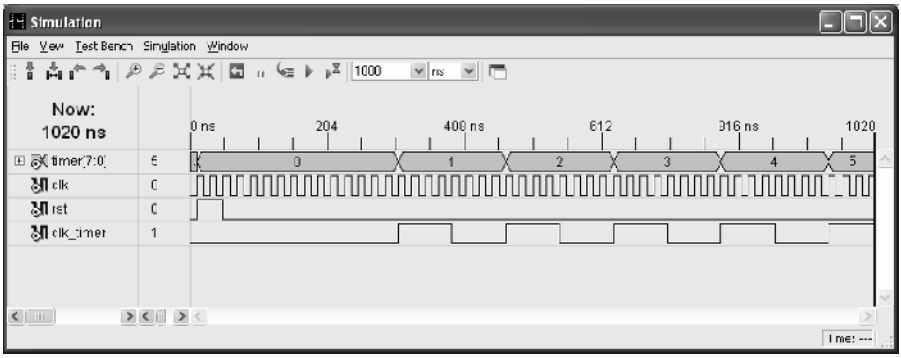


Fig. 3.19. Creation of a timer clock (clk_timer) using a digital clock manager (DCM) component to divide the system clock

3.2.5 I/O Standards

PCB-based designs using a FPGA are interfaced with a variety of other semiconductor devices. Each of these device types has a different voltage standard for interfacing. Memory chips are interfaced to the FPGA using a HSTL or SSTL I/O standard. If an off-chip processor exists, it is connected using the GTL+ interface. Many boards are designed to be compatible with the peripheral connect interface (PCI) backplane, which requires a matching PCI voltage and bus frequency I/O standard. The input and output pins of the FPGA can be configured for different I/O standards (see Fig. 3.20 and Table 3.6), based on the interface needed. Figure 3.20 is a typical PCI bus add-on card board design that uses a FPGA.

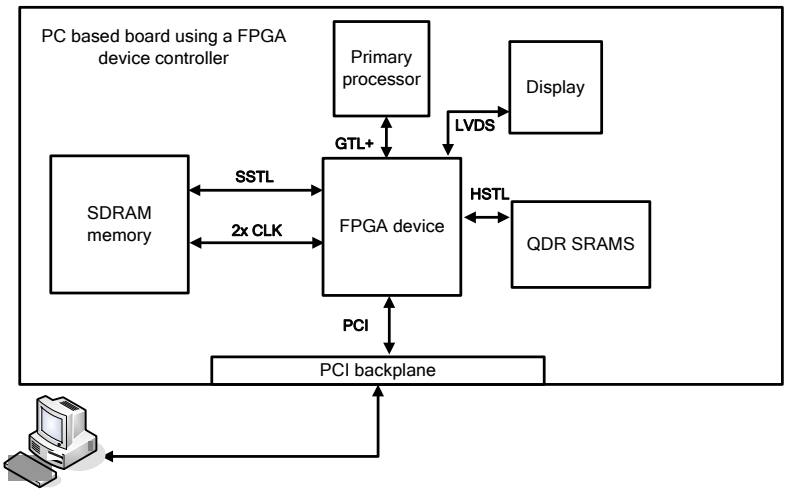


Fig. 3.20. Field programmable device supporting different digital interface standards

Table 3.6. Voltage levels supported by FPGA I/O pins

I/O Standard and description	Voltage level (volts)	Interfaced with
LVTTL, low-voltage TTL	3.3	Pushbuttons
LVCMOS33, low-voltage CMOS	3.3	ADC, DAC
PCI, peripheral component interface	3.3	PC backplane bus
SSTL2	2.5	DDR SDRAM
GTL+	0.65–1.5	Processor
HSTL	1.5	QDR SRAMS
LVDS	2.5	Display

Signals used as inputs to the FPGA connect to an input buffer (IBUF) via an external input port. The default standard for an input port is LVTTL when the buffer is not specified. An output buffer (OBUF) is used to drive outputs through an external output port. When no I/O standard is specified, the OBUF I/O standard is set to LVTTL with 12-mA drive strength (see Fig. 3.21).

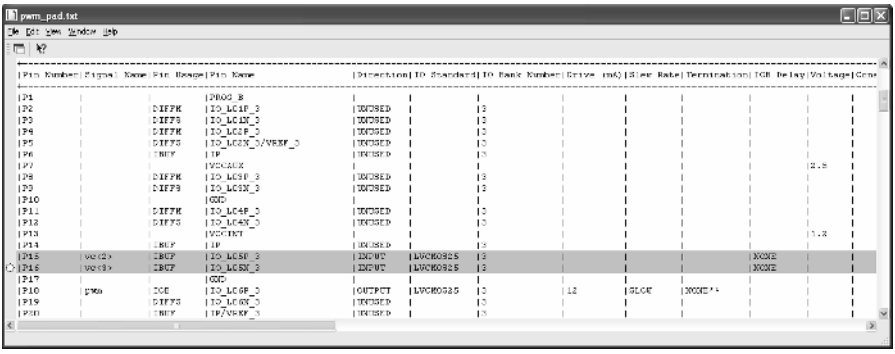


Fig. 3.21. Pin constraint report showing selection of I/O standard, drive (mA) and slew rate

3.2.6 Multipliers

Increased use of FPGA for signal processing applications has made the multiplier an important component of hardware design. Multipliers in a FPGA are implemented in a number of ways. One of the most frequently used methodologies is to let the synthesis software determine the kind of multiplier to be used. The second is to specify a multiplication algorithm, the Booth multiplier. Another way is to specifically instruct the synthesis software to use the multiplier primitive available in the particular FPGA device family. Figure 3.22 shows the block diagram of a FPGA immersed hardware multiplier that supports 18-bit multiplication.

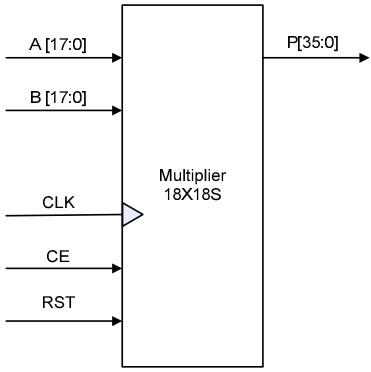


Fig. 3.22. Multiplier block diagram

Example 3.6. Create logic for multiplying two 8-bit numbers.

```
module mult(input [7:0] a, input [7:0] b, output reg [15:0] c, input clock);

always @ (posedge clock)

    c <= a * b;

endmodule
```

Fig. 3.23. Verilog code for multiplication

The code of Fig. 3.23 instantiates one of the 20 available multipliers on the target FPGA chip. The synthesis report is a quick way to ascertain the use of the hardware multiplier (see Fig. 3.24).

3.3 Floor Plan and Routing

The floor plan of the FPGA provides a graphical view of the placement of various elements mentioned in the synthesis report. Figure 3.25 shows a listing of basic elements for a 4-bit shift register. These elements can be located on the FPGA floor plan by selecting them. By default, the floor planner places elements at locations determined by the tool. A definite area of the chip can be defined for placement of logic by providing an area placement constraint. This area is defined by slice locations of the logic matrix. Intuitively, logic placed close to its I/O pins will have fewer interconnect delays.

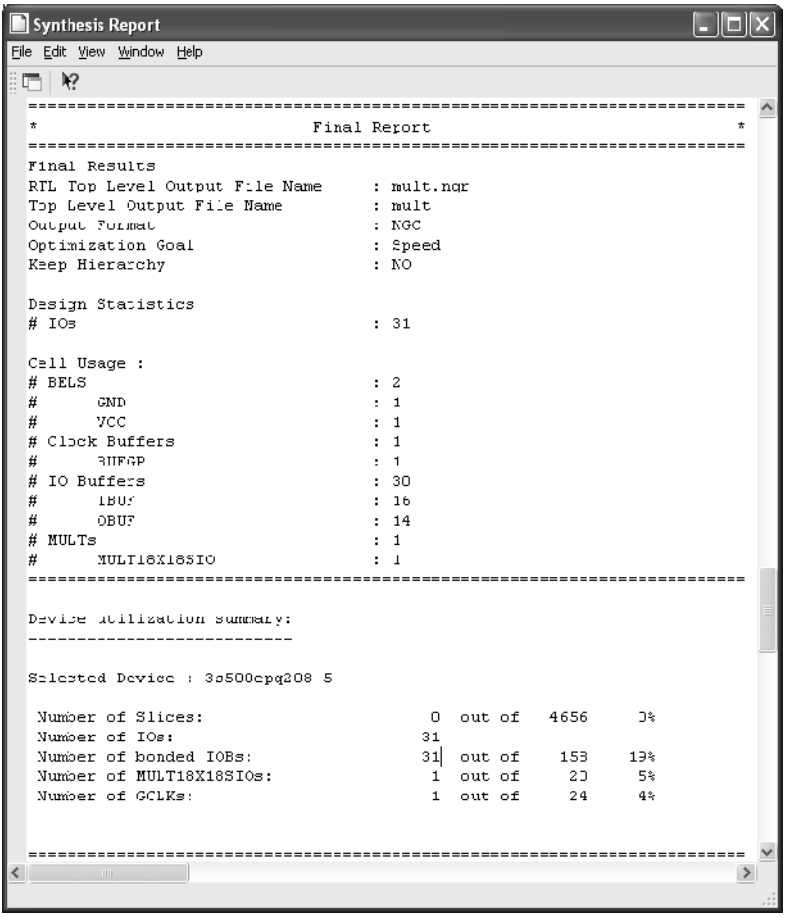


Fig. 3.24. Synthesis report showing use of the multiplier

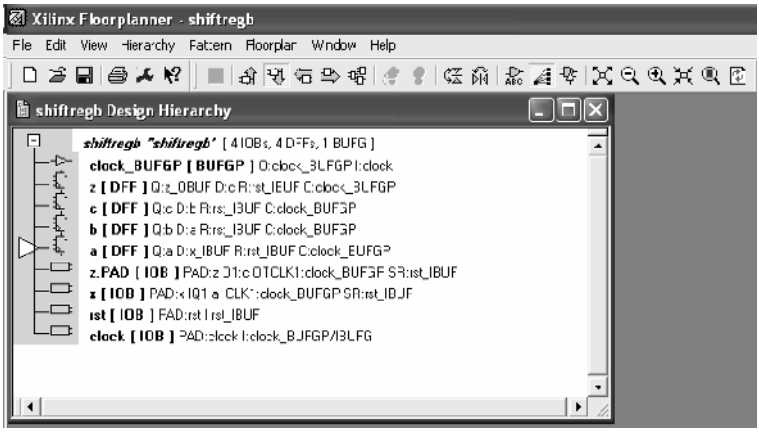


Fig. 3.25. The floor plan design hierarchy for the shift register model of Fig. 2.8c

3.4 Timing Model for a FPGA

The timing model of the FPGA aids in understanding various delays, when a digital circuit is implemented. For a design that is based entirely on combinational logic, the delay path consists of input delay, internal delay associated with combinational logic modules, routing delay and the output delay. Figure 3.26a shows various components that are used for static timing analysis on a FPGA.

The FPGA tool generates a post place and route static timing after place and route have been completed. The static timing analysis tool provides an estimate of the interconnect delay between various sections of a digital design. Figure 3.26b shows the options provided by the tool to analyze the timing delays of example 3.2. The timing between input/output pads of a design or of a defined path in the internal logic is determined by using the timing analysis tool[19]. For a real-time application, where pre-defined timing has to be met, static timing analysis is used for an estimate of the time delay in the logic.

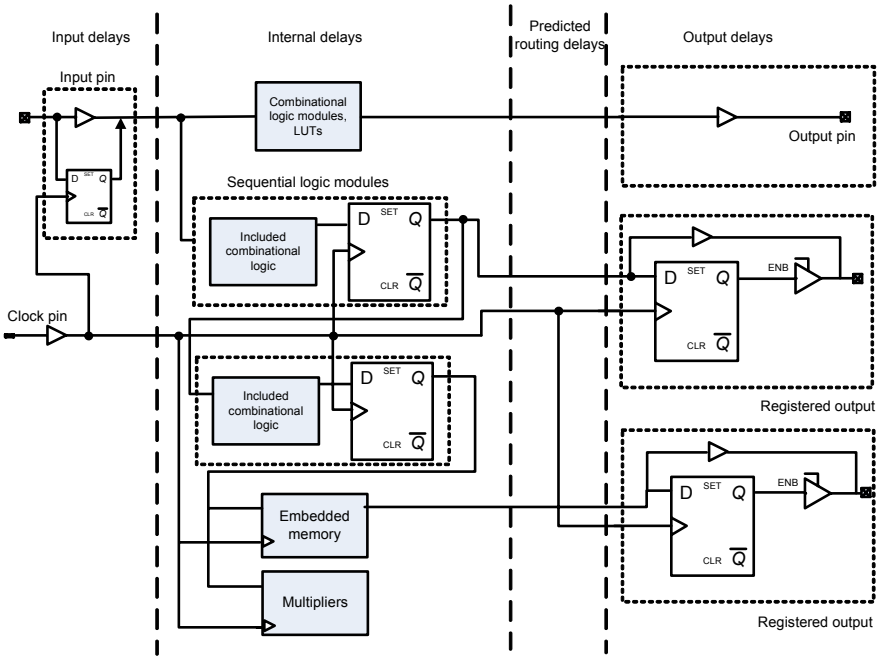


Fig. 3.26. a Timing model for a FPGA-based design

[19] FPGA vendors provide a library of simulation timing models that provide an estimate of timing details. The delay file in the standard delay format (.SDF) is used by post place and route simulation.

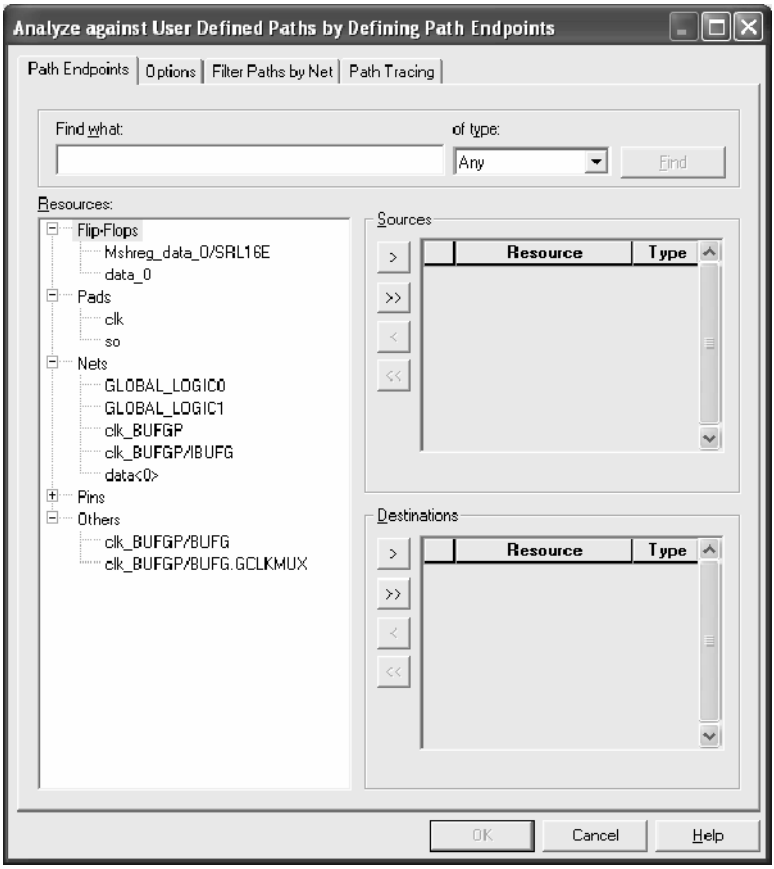


Fig. 3.26. b Timing model for SRL instantiation of Example 3.2

3.5 FPGA Power Usage

FPGA devices provide hardware reconfigurability. The technology that makes this feasible, also extracts a price for it. The price is in terms of power consumed by the FPGA. The majority of FPGAs use SRAM-based interconnect switches to offer reconfigurability. The SRAM is a volatile switch and loses its configuration when power is cycled. There is usually an external memory chip[20] containing the configuration file. This configuration is loaded in the FPGA every time on power up. To keep the interconnect inside the FPGA alive, the SRAM switches are kept powered up. This results in static power consumption. HDL coding techniques help in reducing dynamic power, and use of non-volatile interconnect technology results in static power reduction. The total power consumed in the FPGA can be broken

[20] Some SRAM interconnect-based FPGAs now have built in flash memory to store the configuration file.

down based on the components listed in Fig. 3.27. FPGA vendor provided software tools estimate the power consumption by considering the average switching rate of the components.

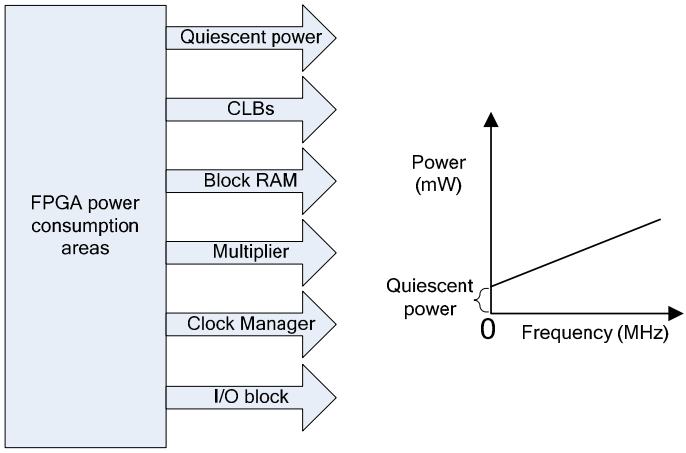


Fig. 3.27. Power consumption areas of a FPGA

Example 3.7. Design digital logic using HDL for controlling a single-phase controlled rectifier. The silicon controlled rectifier (SCR) should trigger at varying time delays α. The power and gate triggering circuits along with waveforms for control are shown in Fig. 3.28a and b.

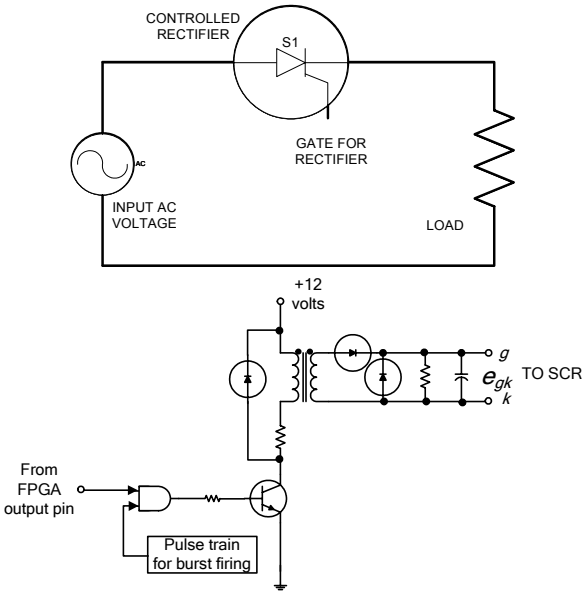


Fig. 3.28. a Single-phase controlled rectifier trigger pulse generation

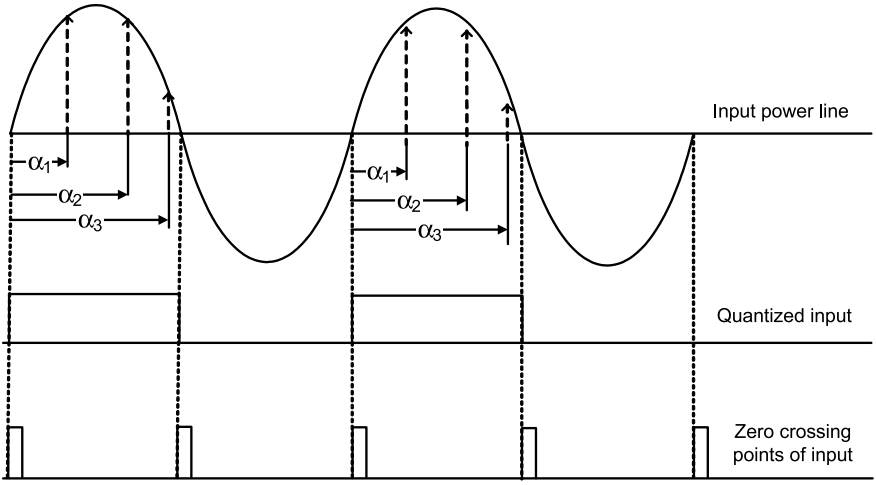


Fig. 3.28. b Single-phase controlled rectifier trigger pulse generation

The SCR triggers in the positive cycle of the input power line. A derived signal called the quantizer is used to sense the duration of the positive half cycle. The edges of the quantizer are used to generate a zero crossing point. Verilog code for zero crossing, shown in Fig. 3.29, senses the quantizer edge to start a counter that determines the width of the zero crossing pulse.

```
module zcd (input quan, input clk, input rst, output reg zcd );

wire w1;

reg [2:0] count1 ;

always @(posedge quan or posedge w1 )
begin
        if (w1)
                zcd <=0;
        else
                zcd <=1;
end

always @(posedge clk)
    begin
        if(zcd)
                count1 <=count1+1;
    else
                count1 <= 0;
    end

assign w1 = &(count1);

endmodule
```

Fig. 3.29. Verilog code to generate narrow width zero crossing signal by using a quantizer

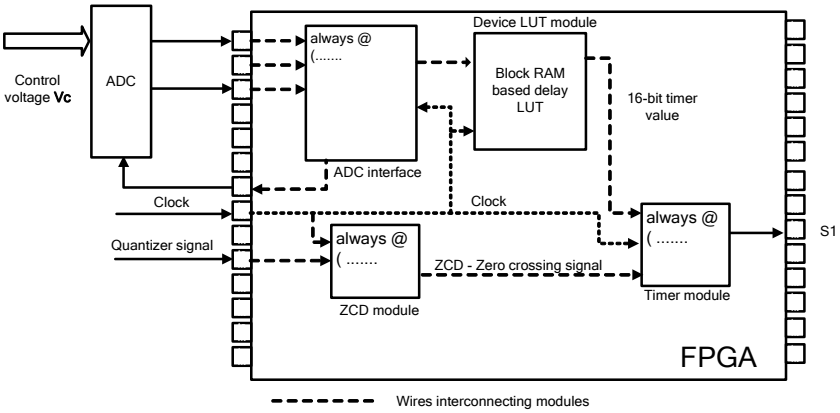


Fig. 3.30. Interconnection of different modules for controlled rectifier logic

Table 3.7. Partial look-up table (LUT) for loading counter value for a specified α angle between 0 and 60°. LUT is implemented in the block memory of the FPGA

Triggering angle α °, input frequency, 50 Hz	**Delay count @ 1.5 MHz timer clock - decimal**	**Hexadecimal value for 16-bit timer**
60	5000	1388
59	4917	1334
58	4833	12E1
57	4750	128E
56	4667	123A
55	4583	11E7
54	4500	1194
53	4417	1140
52	4333	10ED
51	4250	109A

Block Diagram of Converter Control

Digital architecture for implementing the block diagram of rectifier logic is shown in Fig. 3.30. Here the functionality is divided into discrete modules that are integrated together. The ADC interface module is used to obtain the value of the reference voltage **Vc** and subsequently the controller firing angle α. Based on the firing angle α, a timer is initialised that computes the SCR triggering delay after every zero crossing. The delay angle is computed by a 16-bit timer, partially shown in Table 3.7, which loads a pre-determined value based on system clock frequency. The firing angle delay count is stored in the block RAM of the FPGA. A port map for accessing the block RAM is shown in Fig. 3.31. The value of the firing angle is the address of the block RAM memory, and the delay count is available on the data bus of the memory.

```
firingangle UUT2 (
        .addr (m_add),
        .clk (xi_clk),
        .dout (alpha_load));
```

Fig. 3.31. Block RAM instance in top module

The timer module shown in Fig. 3.32 loads a new value based on the firing angle and starts decreasing the value after a zero crossing. A timer_dn output is generated, when the timer has reached zero, to turn on the SCR.

```
module timer (input clk, zcd, rst, input [15:0] timer_val, output timer_dn);

reg [15:0] count;

always @ ( negedge clk )
    begin
        if (rst) count = 16'b0;
    else if ( zcd )
        begin
        count [15:0] = timer_val [15:0];
        end
    else
        count = count - 1;
    end

assign timer_dn = (count == 0) ? 1'b1 : 1'b0;        // on for one clock period

endmodule
```

Fig. 3.32. Verilog timer code for determining instance of SCR turn-on

Problems

1. Complete the code shown in Example 3.7 and check its working and synthesis results.
2. Synthesize the 8-bit Booth multiplier and check the amount of combinational delay for the un-clocked circuit and the maximum frequency for a clocked circuit.
3. Create a digital function generator to generate a variable frequency sine, ramp and triangular functions. The output is sent to a 12-bit unipolar DAC.
    - Use the block RAM as LUT for the sine wave.
    - Use a free running 12-bit counter for the ramp signal.

4. There are certain errors that are not flagged at simulation time but are encountered when the design is being synthesized. One such error is the multi-sourcing error. This error results when two or more independent processes are trying to modify the contents of a particular register. In Verilog, each process is modelled using an **always** statement. The Verilog

code discussed in Example 2.5 is split in two parts. Simulate and synthesize the code given in Fig. 3.33 and determine which signal is causing the multi-sourcing error.

```
module shift_s(input [7:0] word, input clk, rst, output reg x );

reg [2:0] count;

always @ ( posedge clk )
begin
        if (rst)
        count <= 0;
        else if (count < 7)
        begin
                x <= word[count];
                count <= count +1;
        end
end

always @ (posedge clk)
begin
if (count == 7)
        begin
        x <= word[7];
        count <=0;
        end

end

endmodule
```

Fig. 3.33. An example of a multi-sourcing error

Further Reading

1. Maxfield C (2004) The design warrior's guide to FPGAs — devices, tools and flow. Newnes
2. Zeidman B (1999) Designing with FPGAs and CPLDs. Prentice-Hall
3. Xilinx (2007) SPARTAN-3 generation FPGA user guide UG 331

4

FPGA-based Embedded Processor

With rising gate densities of FPGA devices, many FPGA vendors now offer a processor that either exists in silicon as a hard IP or can be incorporated within the programmable device as a soft IP. The purpose of having a processor co-exist with conventional digital logic components is to provide flexibility of combining software and hardware based control in one chip. Many algorithms that are difficult to code in HDL and have update time requirements in milliseconds can use the processor inside the FPGA. A whole suite of tools, consisting of compilers and assemblers help the designer code in C or C++. The motivation of this chapter is to introduce the use of FPGA embedded processors and to integrate custom digitial logic with FPGA-based processors.

4.1 Hardware–Software Task Partitioning

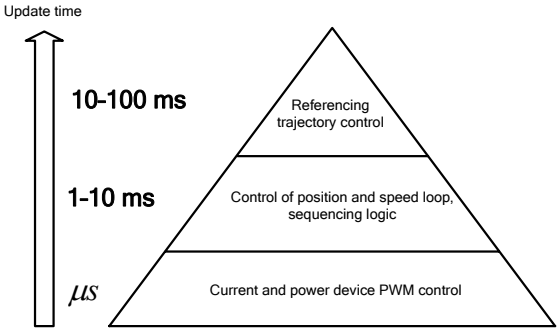


Fig 4.1. Task update rates

A designer of a digital system identifies tasks and their update time requirements. As shown in Fig. 4.1, a robot controller task pyramid consists of tasks that need microsecond or millisecond update time. In our hypothetical robot control system, the task of robot joint trajectory computation, which needs 10–100 ms update time, is assigned to a processor. The processor is driven by a timer interrupt that updates

the trajectory profile every 10–100 ms. The task of motor current and power device PWM control is part of hardware logic (designed using HDL) because it needs to update every 50 μs.

4.2 FPGA Fabric Immersed Processors

The ability to support processor logic has brought a new dimension to the use of FPGA devices. It has provided designers the freedom to partition their designs either for single-threaded software flow or to use concurrent digital logic. A quick search on the internet shows that several 8- and 32-bit proprietary processors are offered by leading FPGA vendors along with established processors. The motivation for using the time tested, established processor is to shorten the learning curve of designers and build confidence in their use. Almost all major vendors of field programmable devices provide processors, for use with their respective devices. Table 4.1 shows a partial list of vendors and the processors they offer.

Table 4.1. Partial list of contemporary FPGA-based processors

Processor name	Type/Bits	Interface bus	FPGA vendor
MicroBlaze™	Soft/32	IBM Coreconnect	Xilinx
NIOS®	Soft/32	Avalon	Altera
LatticeMico32	Soft/32	Wishbone	Lattice
CoreMP7	Soft/32	APB	Actel
ARM Cortex-M1	Soft/32	AHB	Vendor independent
LatticeMico8	Soft/8	Input/Output ports	Lattice
Core8051	Soft/8	Nil	Actel
Core8051s	Soft/8	APB	Actel
PicoBlaze™	Soft/8	Input/Output ports	Xilinx
PowerPC	Hard/32	IBM Coreconnect	Xilinx
AVR	Hard/8	Input/Output ports	Atmel

4.2.1 Soft Processors

Soft processors exist as synthesized netlists incorporated in the FPGA using logic block resources of a particular FPGA. FPGA vendors offer soft processors catering to 8-bit and 32-bit applications. The 8-bit processor occupies a small footprint on the FPGA device and it uses the FPGA embedded memory for program and data

memory storage. Figure 4.2 shows the block diagram of PicoBlaze™, an 8-bit soft controller from Xilinx. The PicoBlaze™ controller consists of an 8-bit input and an 8-bit output port. It also supports interrupt. The embedded block RAM of the FPGA serves as a location for program and data memory for the PicoBlaze™. The PicoBlaze™ assembler takes the program file and creates the coefficient (.coe) file, loaded in the embedded memory of the FPGA.

Among the 32-bit soft processors, two of the leading 32-bit proprietary soft processors are NIOS® from Altera and MicroBlaze™ from Xilinx. These processors use a portion of the FPGA resources. The remaining part of the FPGA can be used for incorporating other digital logic.

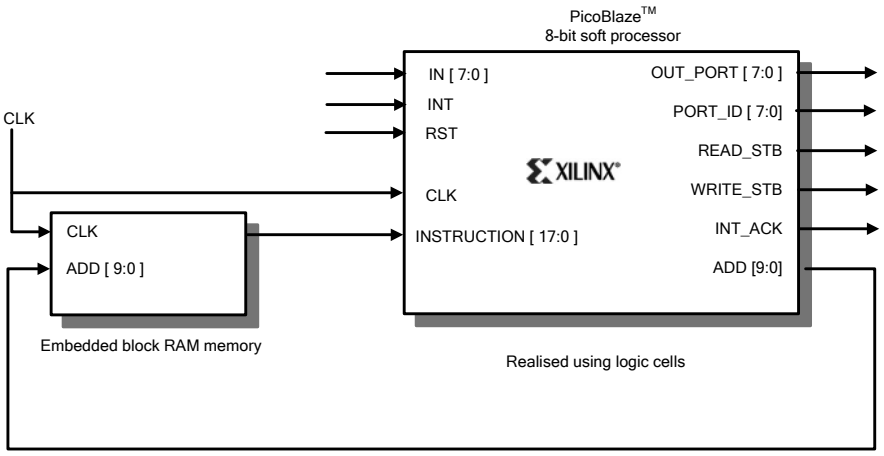


Fig. 4.2. Realising an 8-bit soft controller on a FPGA

The MicroBlaze™ 32-bit soft processor, shown in Fig. 4.3 is a reduced instruction set computer (RISC) based engine with a 32 * 32-bit LUT RAM-based register file with separate instructions for data and memory access. It supports both on-chip block RAM and external memory for program/data memory. All peripherals are implemented on the FPGA fabric and interface to the MicroBlaze™ using the on-chip peripheral bus (OPB) or processor local bus (PLB). The MicroBlaze™ processor options include instantiation of additional hardware to implement IEEE 754 single precision floating point standards. With this option included, it can support floating point addition, subtraction, multiplication, division and comparison.

Soft processors listed in Table 4.1 can be customized by adding a barrel shifter or modifying the size of the data and instruction cache. Additional processors can also be added to provide a multi-processing option. Based on the results of software profiling, certain resource intensive software algorithms can be moved to the hardware fabric as coprocessors or as custom peripherals.

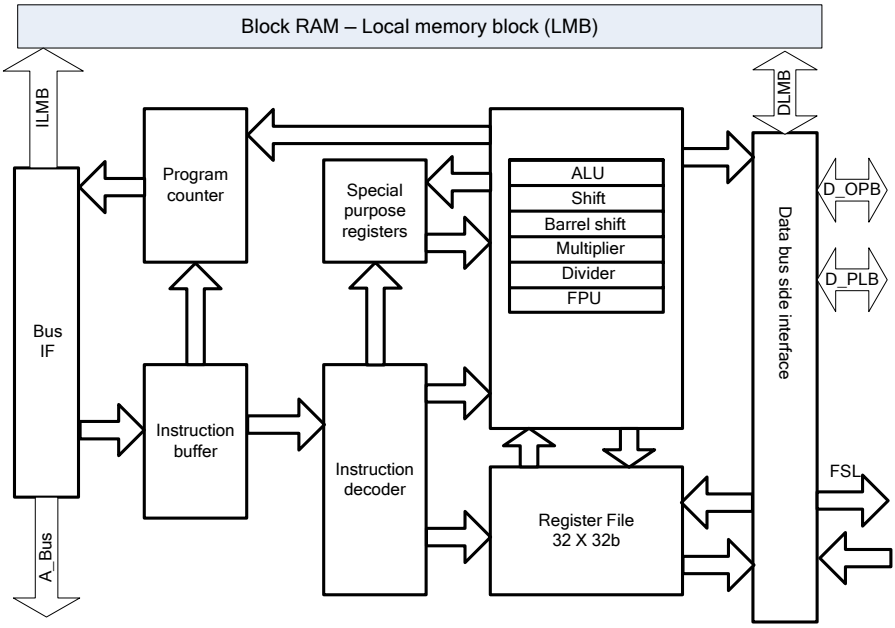


Fig. 4.3. MicroBlaze™ block diagram

4.2.2 Hard Processors

Many standard processors and microcontrollers are available as hard IP inside the FPGA. Some of these include the AVR microcontroller offered by the Atmel family and the PowerPC processor as part of Xilinx Virtex FPGAs. The design chain for programming and debugging the FPGA-based processor system is quite similar to their earlier model as stand-alone processors.

4.2.3 Tool Flow for Hardware–Software Co-design

To co-design a system, both hardware and software tasks need to be independently coded and tested. The software flow shown in Fig. 4.4 combines the source and the library files to create an executable file for the processor to use. The location of the program and data code can either be in the internal memory of the FPGA device or if the program code is large, an external memory device is used. In Fig. 4.4, the program is stored within the memory of the FPGA device. The hardware design flow takes the electronic design interchange format (EDIF) files of the soft processor, merges them with the user written custom digital code and prepares a complete system netlist after synthesis.

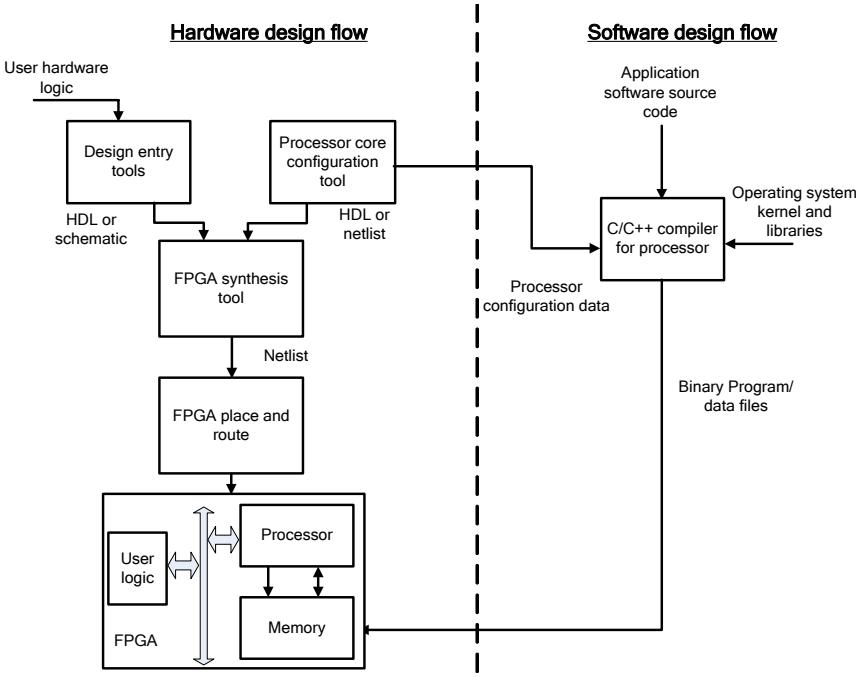


Fig 4.4. Hardware–software design flow [1]

4.3 Interfacing Memory to the Processor

The use of a processor calls for memory, where the instructions are stored. As discussed in Chap. 3, many FPGAs provide on-chip embedded memory. The size of this memory depends on the density of the FPGA device in use. For the Xilinx 500 k gate FPGA, 360 kbits of memory are available. When this onboard memory is used, a local memory bus (LMB) controller is configured to read and write to/from this memory. For small codes that are meant for the PicoBlaze^{TM} processor or assembly coded codes for the 32-bit processors, the onboard memory fulfills the requirements. When using this bus, memory accesses are much faster and are handled by the memory controller. For larger programs, the compiled code needs to be stored in an external memory chip. A memory controller is configured for accessing the external memory chip. The linker script settings shown in Fig. 4.5 need to be modified for either using an external double data rate synchronous dynamic random access memory (DDR SDRAM) or internal FPGA memory to store program code, stack or heap.

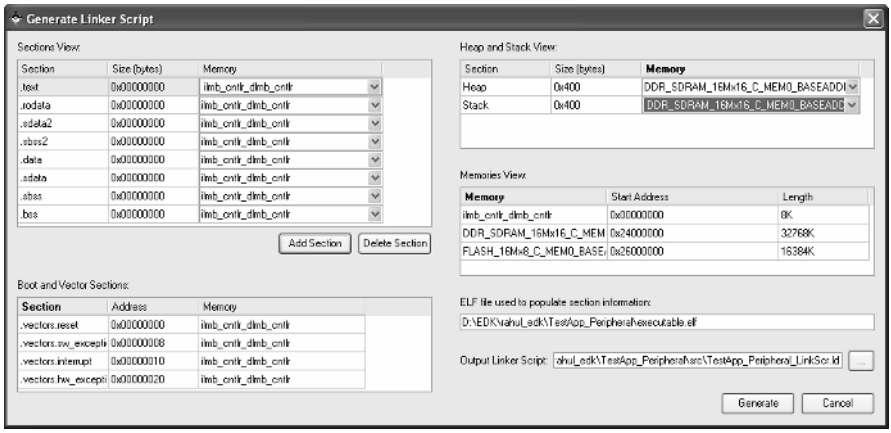


Fig. 4.5. Linker script settings for determining the type of memory to be used with a MicroBlaze™ soft processor

4.4 Interfacing Processor with Peripherals

> *"Designers can also create their own custom peripherals and integrate them into soft processor systems. For performance-critical systems that spend most CPU cycles executing a specific section of code, it is a common technique to create a custom peripheral that implements the same function in hardware. This approach offers a double performance benefit: the hardware implementation is faster than software; and the processor is free to perform other functions in parallel while the custom peripheral operates on data."*
>
> *Altera on using custom peripherals*

From the digital design point of view, the shared bus interface approach is similar to peripherals that are interfaced to a processor using tracks on a PCB. Many backplane bus standards that provide basic interfacing capabilites for peripherals have evolved over time. As shown in Fig. 4.6, signals such as address, data and control bus are used by interconnect buses. Many proprietary interface bus standards exist, with similar interfacing methodology. Tables 4.2 and 4.3 list physical backplane bus standards and system-on-chip buses used for interfacing processor and peripherals.

The hardware-software synergy available by use of a FPGA-based processor makes sense only if customized coprocessors or peripherals are part of the design. Otherwise it is much simpler and economical to go for a microcontroller or digital signal processor implementation. There are many schemes for connecting user defined custom logic to a FPGA processor. One way is to use industry standard interconnect buses. The use of standard buses improves the re-use of IP core because the bus interface logic provides the front-end connection to the IP core.

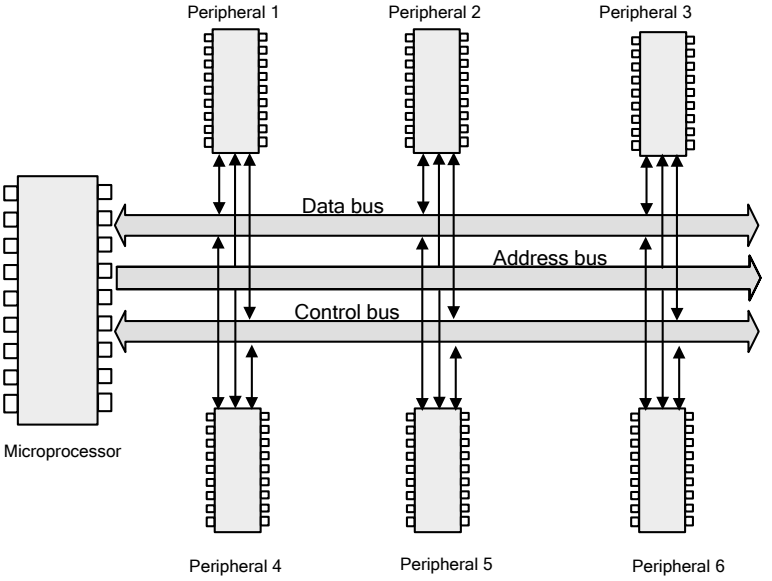


Fig. 4.6. Processor connected to different peripherals on a PCB

Table 4.2. Partial list of physical backplane bus standards

Name	**Originator**
IBM PC bus	IBM
Multibus	Intel
Multibus II	Intel
VME bus	Motorola
STD bus, IEEE 488 bus	Hewlett-Packard
Q-bus	Digital Equipment Corp.
Unibus	Digital Equipment Corp.

Table 4.3. System on Chip buses

Bus name, originator	**Interconnection type**
AMBA, ARM	Shared bus architecture
Wishbone, Silicore	Point-to-point, crossbar, data flow, shared bus
OPB CoreConnect bus, IBM	Shared bus architecture
Fast serial link, Xilinx	Point-to-point
Avalon bus, Altera	Cross bar switch

4.4.1 Types of On-chip Interfaces

On-chip processor peripheral interface topology can be divided into point-to-point, cross bar switch and shared bus interfaces.

4.4.1.1 Point-to-Point
The point-to-point interface consists of a dedicated link between the communicating devices. An example is the fast simplex link (FSL)[21]. This point-to-point link provides a unidirectional, non-arbitrated channel to perform fast communication between the MicroBlaze™ processor and custom hardware module. The MicroBlaze™ can be configured to support multiple FSL links to read and write data or control signals to a peripheral.

4.4.1.2 Cross Bar Switch
The cross bar switch consists of an interconnection fabric, configured based on the connections between various sections of the design. This is usually done at design synthesis time. The transfer rate of a crossbar switch is higher than that of a shared bus mechanism. But the crossbar switch requires more interconnection logic and routing resources than a shared bus system.

4.4.1.3 Shared Bus
Shared buses are similar to their older backplane bus counterparts. The shared bus interface defines an address for the slave or peripheral device. The address bus originating from the microprocessor carries the address of the slave device to which the microprocessor wishes to communicate. An address decoder in each slave device determines whether that particular slave is being addressed. A bank of registers within the slave device are written to or read from by the microprocessor master device. Each of the registers in the slave device has a unique address sent by the master device. A typical read and write to a particular register in the slave device is illustrated by the timing diagrams of read and write cycles.

The peripheral logic periodically updates the feedback/status registers and reads the configuration registers. If a peripheral seeks immediate attention, it generates an interrupt for the microprocessor. If there is only one interrupt generating peripheral in the system, it is connected to the interrupt port of the microprocessor. But if there are many peripherals that can possibly interrupt the processor, there is an interrupt controller peripheral that routes the interrupt signal to the microprocessor based on the pre-assigned priorities of the interrupts.

[21] The FSL link is a proprietary point-to-point link supported by Xilinx soft processor MicroBlaze™. It is used for streaming data from connected devices.

The shared bus consists of

- a unidirectional Address Bus
- a bi-directional Data Bus
- control signals
    - write/read enable
    - acknowledge
- System control signals — clock, reset.

4.4.2 Wishbone Interface

Wishbone supports three kinds of interconnections. These include point-to-point, shared bus and the crossbar switch. The interface defines the connection between the processor (master) and the wishbone slave. As Fig. 4.7 shows, the interconnections consist of address, data and handshaking signals. The handshaking mechanism is used to adjust the data transfer rate. The acknowledge [ACK_0] signal is mandatory, whereas error [ERR_o] and retry [RTY_o] are optional handshaking signals. The signals defined for the point-to-point connection are also used in the shared bus interface.

Wishbone supports single read/write, block read/write, and read-modify-write operations. All signals between master and slave are either inputs or outputs, but never bi-directional (using tri-state logic). Address and data bus widths can be changed to fit the application. Possible widths supported are 8, 16, 32 and 64 bits.

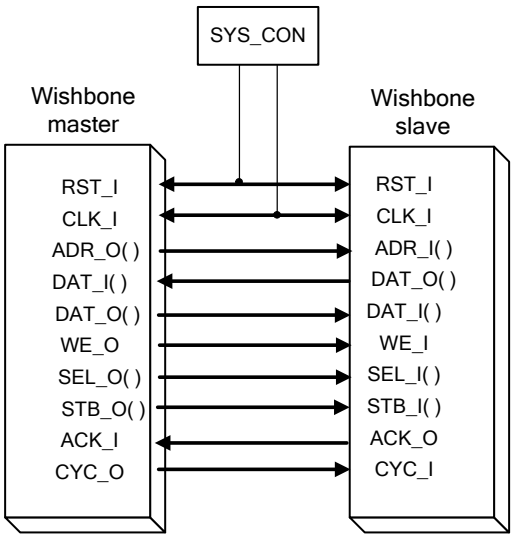


Fig. 4.7. Wishbone interface configured for a point-to-point connection [2]

4.4.3 Avalon Switch Matrix

The Avalon switch fabric is an interconnect technology used by Altera. It is generated using the system on a programmable chip (SOPC) builder tool of Altera. It provides up to a 128-bit address and data path. There is support for multiple masters, built-in address decoding, peripheral transfer support, read and write transfers and fixed and variable length transfers. The FPGA switch fabric provides for a fast configurable interconnect used to make memory mapped connections between master and slave devices. A sample configuration of an Altera NIOS® processor connected to three slaves is shown in Fig. 4.8.

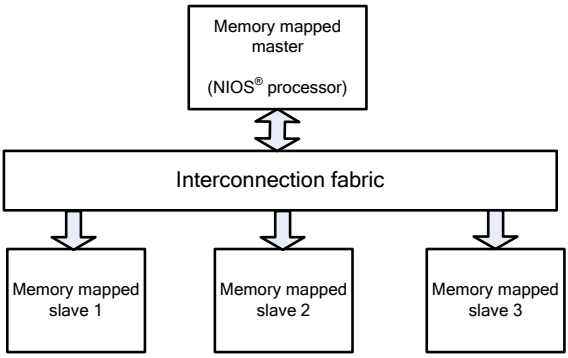


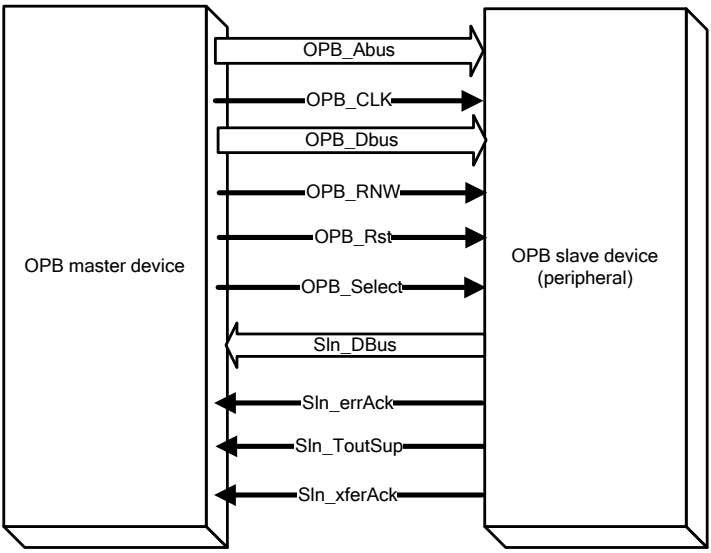
Fig. 4.8. Interconnection of Avalon Bus using interconnection fabric

4.4.4 OPB Bus Interface

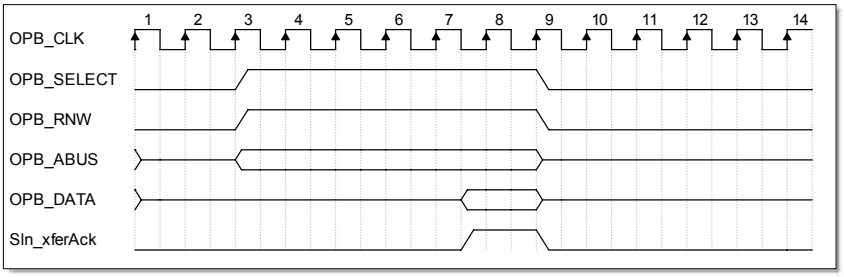
The on-chip peripheral bus (OPB) is a shared bus architecture. It is part of the CoreConnect architecture developed by IBM for integrating on-chip "cores". Although the specifications allow for a 32- or 64-bit wide address and data bus, the FPGA adaptation by Xilinx uses 32-bit as the word size. The OPB bus system uses master-slave architecture. The master which is usually part of the microprocessor can initiate a transaction by specifying a slave address. The slave responds to the requests from the master. Both the OPB and the MicroBlaze™ soft processor use big-endian form of data, where bit 0 is the most significant bit and bit 31 is the least significant. The signals exchanged between an OPB master and slaves are shown in Fig. 4.9a.

The OPB bus read cycle illustrated in Fig. 4.9b consists of the OPB_RNW signal becoming high along with the OPB_SELECT signal. The address where the OPB master (the processor in our case) wishes to read from, is sent on the OPB_ABUS and after a latency of four cycles, the data from the peripheral are available on the OPB_DATA bus.

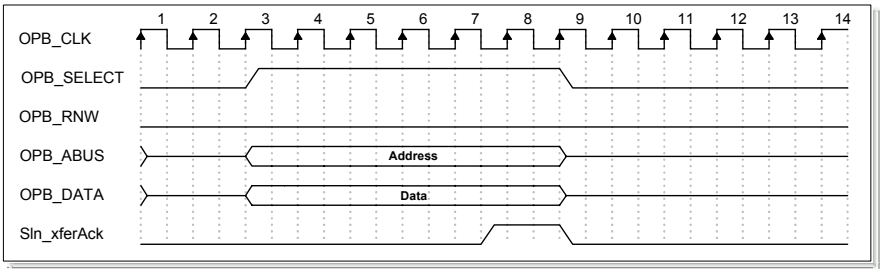
The write cycle of the OPB works in a fashion similar to the read cycle. Instead of the OPB_RNW becoming high along with OPB_SELECT, the address and data bus contents are put on their respective buses. A transfer acknowledge signal from the slave indicates the completion of the write cycle. The timing diagram for this transaction is shown in Fig. 4.9c.



a



b



c

Fig. 4.9. a OPB bus interface signals for master and slave devices; **b** OPB read cycle; **c** OPB write cycle

4.5 Design Re-use Using On-chip Bus Interface

Many times, standard digital components are re-used for different applications. Making a design compatible with an on-chip bus interface is one way to re-use a design. Different IP cores developed independently can be tied together and tested by standardizing the IP core interfaces. Many re-usable digital designs[22] available in the public domain are compatible with on-chip interfaces.

Table 4.4. List of peripherals needed for the robot controller

Name of peripheral	Description	Application
Communications	RS-232 and RS-485 networks	Communicating with PC, multi-drop networks
Motor controller	Control of speed and power module of motor	Five controllers – one for each axis
Quadrature encoder interface	Determination of position and speed of each axis	Five peripherals – one for each axis
Serial peripheral interface(SPI)	Interfacing with ADC, DAC and other sensors	Getting feedback from distance sensors, proximity sensors of robot workspace
Timer	Fixed interval timer	Providing interrupts to processor
SDRAM	Memory controller	Connecting extra memory

In the list of peripherals mentioned in Table 4.4, the motor controller and Quadrature encoder interface are not available as microcontroller peripherals.[23] These peripherals are also not included in a FPGA vendor supplied peripheral library. In such cases a custom peripheral for motor control is called for.

Motor Drive as a Peripheral

The robot motor drive logic can be made as a custom peripheral that interfaces with the system-on-chip bus (Fig. 4.10). This peripheral can be replicated or cloned multiple times on a single FPGA chip without affecting the performance of each individual drive (thanks to the independent concurrent threads that a FPGA device can support!). Chapter 6 will discuss more on how to create HDL code for robot motor drive control.

---

[22] Re-use able digital designs from www.opencores.org use the Wishbone bus standard.

[23] A microcontroller for motor control application does contain peripherals to aid in motor drive design. Along with standard GPIO it contains a 16-bit rotor speed measurement counter, three-phase PWM signal generator, 6-bit dead time generator, and interrupt generators that exchange data with the CPU over a proprietary register bus.

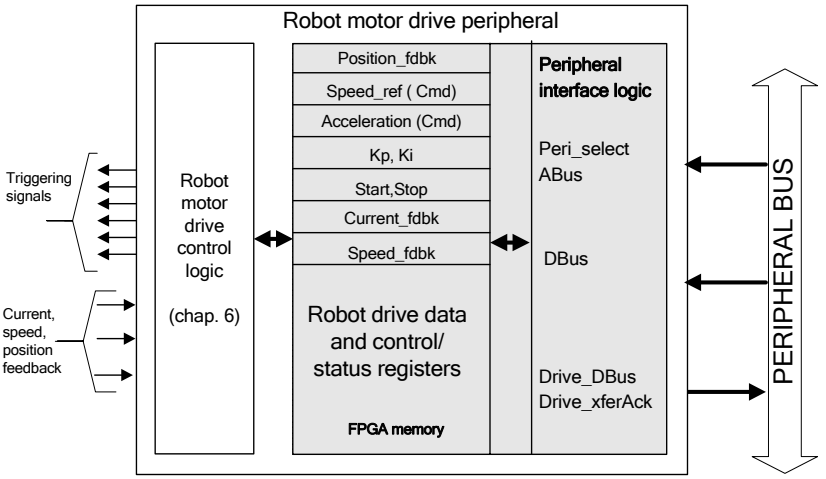


Fig. 4.10. Details for robot motor drive peripheral

Design of Custom Peripheral

A peripheral is an independent device used to off-load the processor from processes that require frequent attention. The peripheral takes input from the processor and then interrupts the processor on completion of a given task. Examples of common on-chip microcontroller peripherals include timers, UART and interrupt controllers. The availability of high-density FPGA devices, with built-in processors has made incorporation of custom peripherals feasible. A FPGA-based processor system is configurable to create a customized microcontroller. Additional user defined peripherals can be integrated with the processor. The design of a custom peripheral requires a bus interfacing logic and the custom code of the user peripheral. Based on the timing diagram of the read and write cycles of the particular bus protocol, a finite state machine is designed to interface a custom peripheral device. A typical port map of the slave device consists of the following signals:

- Bus to slave peripheral
  - Address bus
  - Data bus
  - Clock
  - Read/Write
- Slave peripheral to bus
  - Data bus
  - Transfer acknowledge
  - Time out suppression

The finite state machine would consist of idle, selected, read/write, transfer acknowledge and then again idle state. The peripheral interface logic consists of the shaded section shown in Fig. 4.11.

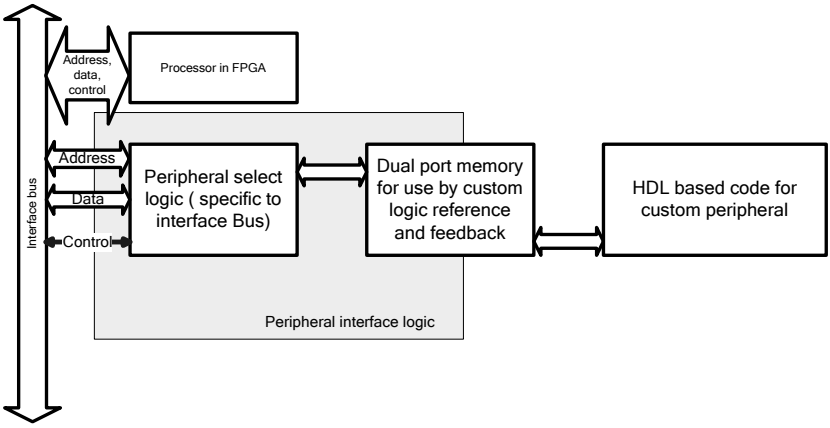


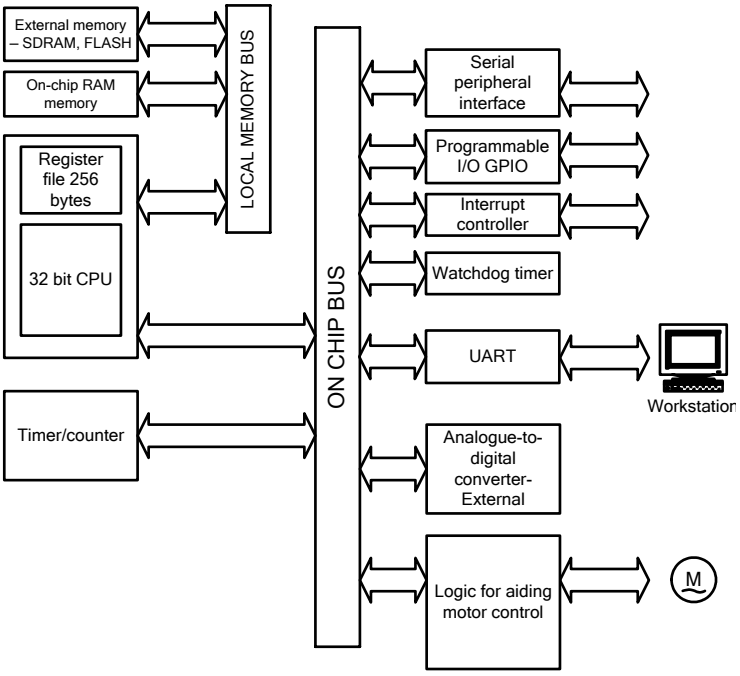
Fig. 4.11. Interface logic for connecting a peripheral to a chip interconnect bus

4.6 Creating a Customized Microcontroller

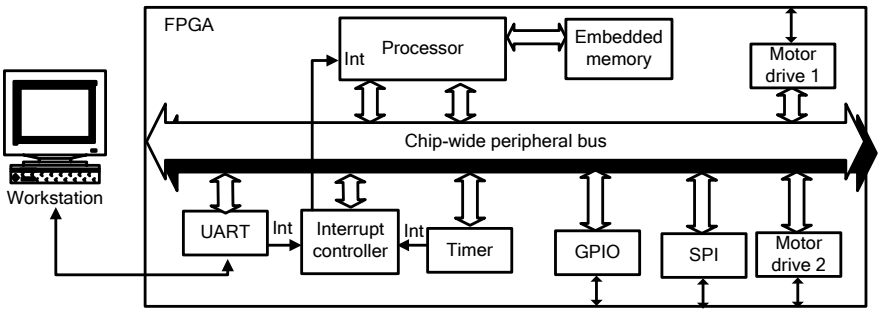
The FPGA allows the flexibility of creating a customized application-oriented microcontroller. The programming flow of these processors is similar to that used for programming a microcontroller-based system. A high-level programming interface such as C is used. An architecture for a microcontroller designed for motor control is shown in Fig. 4.12a. This architecture shows a 16-bit CPU, SPI, ADC and timer/counter interfaces. There is a proprietary internal bus within the controller that connects the processor with various peripherals.

If a similar architecture for multi-motor control were to be created using the field programmable device, the architecture would not change much. A FPGA-based hard or soft CPU is chosen, and various pre-designed peripherals offered, are put together to form a system around a given interconnect bus. If the program memory requirement is large, an external memory device is connected using a memory controller IP. The typical environment of a single-chip controller consists of general purpose input output (GPIO) devices, a communications terminal such as the UART and memory. For the sake of equivalency with the microcontroller environment, let us create a similar FPGA-based system.

Example 4.1. Create a customized microcontroller as shown in Fig. 4.12b. The microcontroller contains among other peripherals, two custom motor drive interfaces.



a



b

Fig 4.12. a Architecture of a microcontroller for control applications; **b** architecture of a FPGA-based system on chip customised for motor control application

Fig. 4.13. Snapshot of peripherals available for design of an embedded system

Figure 4.13 shows a peripheral IP library available for use in the design of an embedded system. The following peripherals are chosen from this list:

- UART: In this example the standard input and output port (STDIN and STDOUT) are configured for RS232_DCE. This ensures that program outputs using the **print** command are displayed on the serial port. An interrupt is generated when any valid character is in the receive FIFO and the interrupt stays active until the receive FIFO is empty.
- Timer: The timer provides an interrupt for real-time processing.

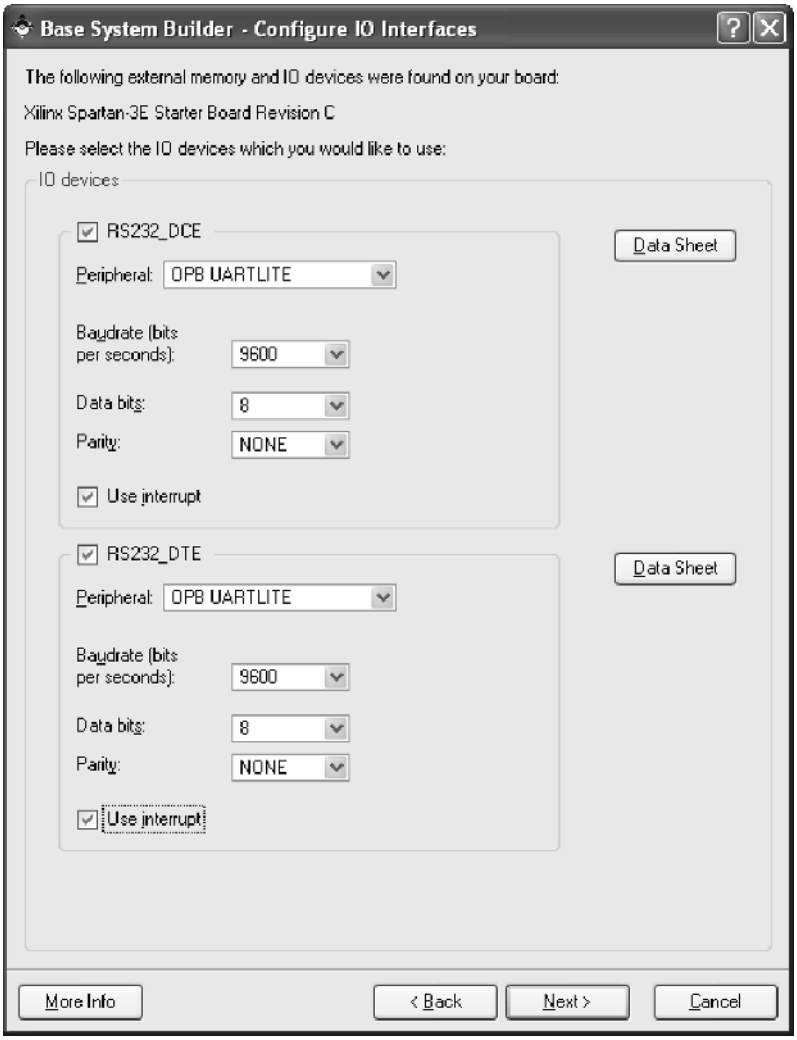


Fig. 4.14. Configuring the UART peripheral for use with the OPB bus

- SPI bus: Provides communication with off-chip ADC and DAC chips.
- Interrupt controller: For managing multi-source interrupts.

The only peripheral not available from the IP library is the motor drive peripheral. A customized peripheral discussed in Sect. 4.5 needs to be created. The configurations of the UART and the timer are shown in Figs. 4.14 and 4.15.

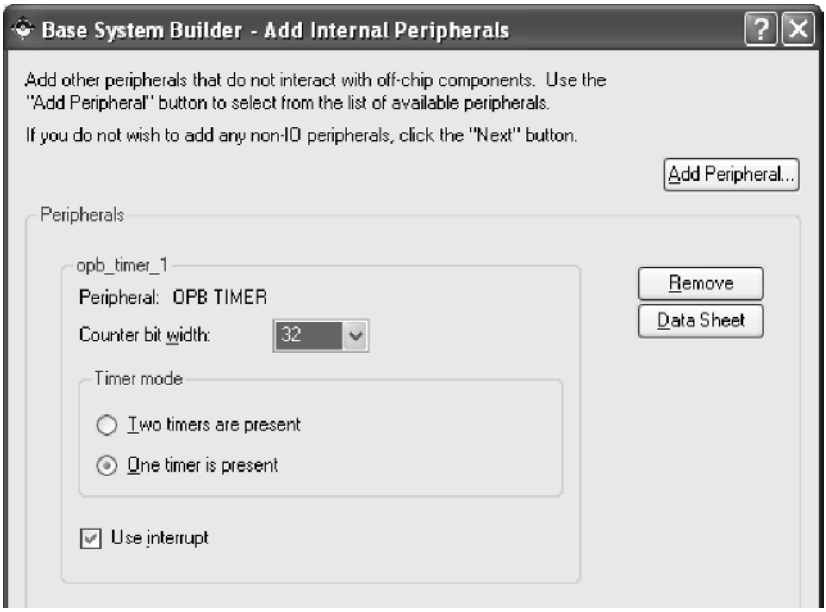


Fig. 4.15. Configuring the timer peripheral for use with the OPB bus and processor

4.7 Robot Axis Position Control

The architecture of a joint or axis position control scheme is shown in Fig. 4.16. Each joint of the manipulator is controlled by a position servo loop. The joint trajectory control algorithm uses a new joint set point, J_N. Based on trajectory/ system parameters and constraints, a position profile is generated for each joint of the manipulator.

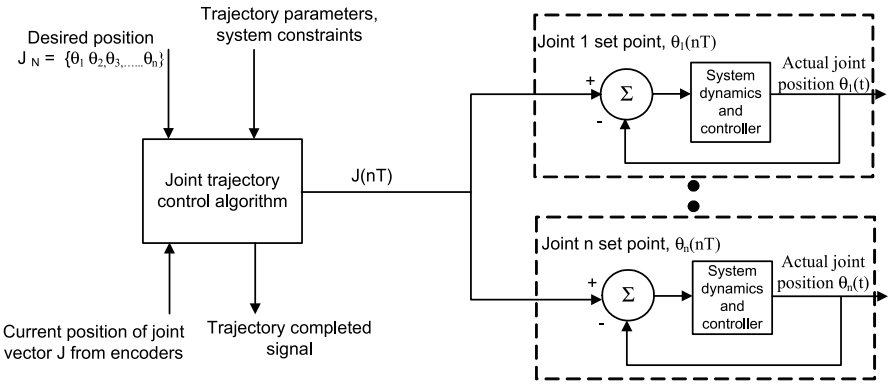


Fig. 4.16. Robot joint axis control [3]

The profile generator is usually implemented in software. The data sequence [**θn**(**nT**)] for a particular joint is output in real time and is driven by an interrupt-based update mechanism. The digital servo loop that accepts the set point for the position is discussed in detail in Chap. 6.

A robotic manipulator is modelled as a chain of links, as seen in Fig. 4.17. These links are interconnected to one another by joints. The last link has the tool or end-effector attached to it. Denavit and Hartenberg have presented a systematic procedure for assigning a co-ordinate frame to the links of a robotic manipulator. The objective of the robot controller is to position the tool in three-dimensional space. The tool is programmed to follow a planned trajectory so that it carries out operations in the workspace.

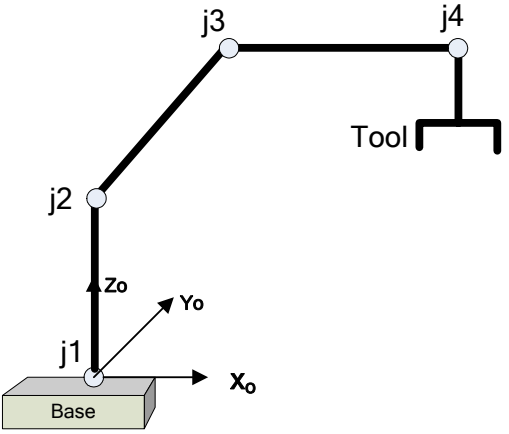


Fig. 4.17. Links and joints of a robot

Inverse kinematics for a robot is computed by knowing the desired space co-ordinate value **x**,**y**,**z** and then determining the motion angle for each robot arm axis [θ_1, θ_2, θ_3, θ_4, θ_5]. For a given robot arm of 5 degrees of freedom, an **x**,**y**,**z** co-ordinate to motor angle transform is performed using the Denavit–Hartenberg(D-H) method [4]. The D-H parameters for the five joint robot are mentioned in Tables 4.5 and 4.6. A timer driven interrupt is used for calculating a new set of motion angle commands every 100 ms (see Fig. 4.18).

Table 4.5. Robot joint and link parameters

Arm parameter	**Symbol**
Joint angle	**θ**
Joint distance	***d***
Link length	***a***
Link twist angle	α

Table 4.6. Denavit-Hartenberg parameters for a robot arm,Mitsubishi RV-M1 [5]

Joint	**θ**	***d* (mm)**	***a* (mm)**	**α**
1	θ_1	300	0	$-\pi/2$
2	θ_2	0	250	0
3	θ_3	0	160	0
4	θ_4	0	0	$-\pi/2$
5	θ_5	72	0	0

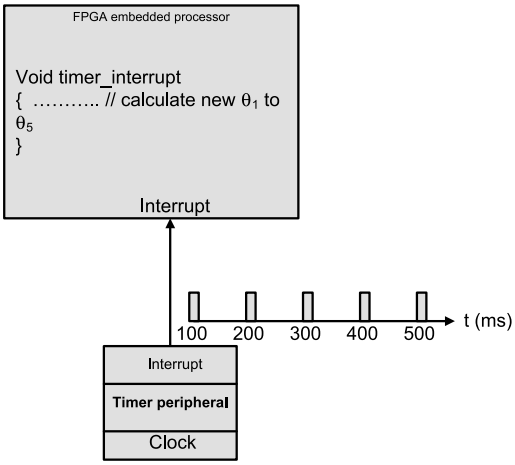


Fig. 4.18. Timer interrupt for calculating trajectory parameters every 100 ms

Problems

1. Timers and counters are omni-present components of embedded systems. They are used for setting sampling rates, measuring speed and counting external inputs. Develop HDL code for a timer (use the specifications for Intel 8254 timer) peripheral that can interface with the on-chip interconnect bus.
2. The Intel 8085 processor has multiple interrupt inputs, consisting of INT 5.5, 6.5, 7.5, NMI and TRAP. Modern processors support one interrupt line connected to an interrupt controller. Develop an HDL-based Interrupt controller (use the specifications for Intel 8259 interrupt controller) for interfacing with the Interrupt pin of a processor.

References

1. Hall TS, Hamblen JO (2004) System-on-a-programmable-chip development platforms in the classroom.IEEE Transactions on Education: 47(4): 502–507
2. WISHBONE System-on-Chip (SoC) interconnection architecture for portable IP cores. (2002) B.3. http://www.opencores.org/projects.cgi/web/wishbone/wishbone. Accessed 21 May 2008
3. Klafter RD et al (1989) Robotic engineering, an integrated approach. Prentice-Hall
4. Schilling RJ (1990) Fundamentals of robotics analysis and control. Prentice-Hall, New Jersey
5. Kung Y, Shu G (2005) Development of a FPGA-based motion control IC for robot arm. Paper presented at IEEE ICIT 2005,1397–1402

Further Reading

1. Slater M (1989) Microprocessor–based design, a comprehensive guide to hardware design. Prentice-Hall
2. Mitsubishi Industrial Micro-Robot System Manual for Model RV-M1, MovemasterEx, BFP-A5191E-B
3. Kung Y, Huang P, Chen C (2004) Development of a SOPC for PMSM Drives. Paper presented at the 47th IEEE International Midwest Symposium on Circuits and Systems 2004
4. Kung Y, Shu G (2005) Development of a FPGA-based motion control IC for robot arm. Paper presented at IEEE ICIT 2005, pp. 1397–1402
5. Navabi Z (2007) Embedded Core Design with FPGAs. McGraw Hill

5

FPGA-based Signal Interfacing and Conditioning

This chapter introduces ways to interface external world signals with a FPGA. In our robot controller scheme, this would help the robot to sense information coming from various analogue sensors and digital interfaces. A serial data communication network is discussed that helps in interconnecting multiple robots. Using this network and a protocol, command and feedback, data can be communicated from a central controller to other robots on the assembly line.

5.1 Serial Data Communication

Modern sensors have a digital front end for transmitting measured parameters. The signal acquired from the external world is formed into packets of digital data and then serially transmitted. This reduces the amount of cabling required to bring in data from various sensors of a robot's surroundings. A typical sensor along with electronics could transmit the acquired signals using one of the many standard physical layers and protocol standards. Figure 5.1 shows different interfaces for serial data communications.

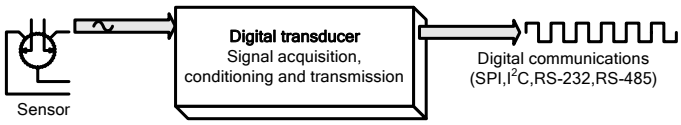


Fig. 5.1. Interfacing sensors using different physical interfaces

Though there are buses that support parallel communication between digital components, the majority of present-day digital systems use two or three-wire serial communications. As shown in Fig. 5.2, the widely used serial buses can be divided by their domain of operation into

- PCB-based communication links – synchronous
- Physically separated systems – asynchronous.

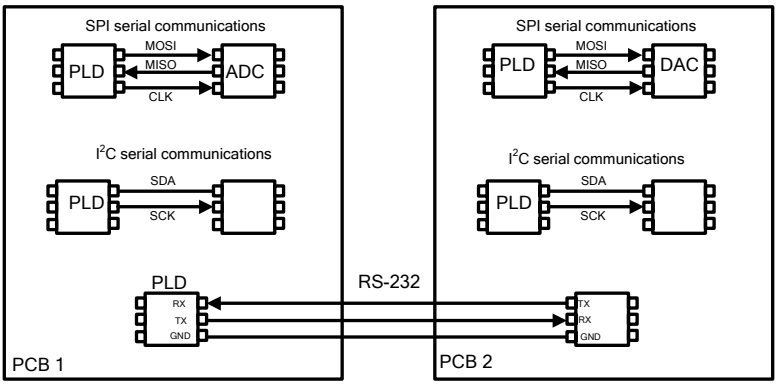


Fig. 5.2. Serial communications for inter PCB and intra PCB

Universal Asynchronous Receiver Transmitter (UART)

A UART[24] is a serial communication circuit that uses the non-return to zero (NRZ) code. As shown in Fig. 5.3, the data format of a UART consists of a high idle state, a start bit, a character frame consisting of 8 bits, an optional parity bit and one stop bit.

Stop bit (1)	Parity bit	Data bit 7	Data bit 6	Data bit 5	Data bit 4	Data bit 3	Data bit 2	Data bit 1	Data bit 0	Start bit (0)

Fig. 5.3. Data frame of a UART

A UART communication pair consists of a transmitter and a receiver exchanging data bits. Figure 5.4 shows the interconnection and internal diagram of two UARTs.

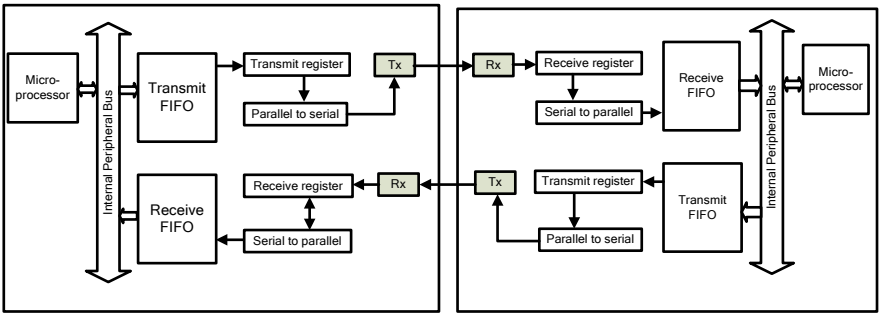


Fig. 5.4. UART block diagram of transmitter and receiver

[24] The UART is a standard peripheral available as a part of FPGA-based processor design tool. A scaled-down version of UART is part of a Xilinx embedded development kit(EDK) and is also a component of the Altera system on a programmable chip (SOPC) builder.

The FIFO is an important component of networks and signal processing applications. It acts as a buffer between the transceiver and the processor. In the transmitter the FIFO accepts the data byte to be transmitted from the internal bus, and it stores the received data byte in the receiver.

Example 5.1. A student wrote a code (Example 3.2) for repeatedly transmitting ASCII character "A" using the UART. The receiving serial communications port is configured for no parity. The received character at times correctly showed "A" but at times incorrectly showed "P". Why is this happening, and how can it be corrected?

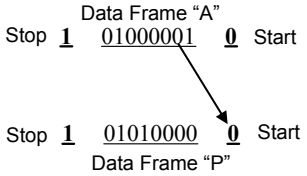


Fig. 5.5. Illustration of a data framing error in the absence of a parity bit

As shown in Fig. 5.5, the UART forms two valid frames. The first frame is the correct data for "A" and the second frame which is a shifted version of the first, is the incorrect data "P". This data framing error is due to the absence of a parity bit.

Manchester Encoding

Many of the communication interfaces are Manchester encoded rather than NRZ. Non-return to zero (NRZ) and Manchester codes are used to represent binary values "1" and "0" in digital systems. NRZ requires one level to represent a binary value, whereas Manchester code requires two levels. Manchester coding defines a positive transition for logic 1 and a negative transition for logic 0. The encoder of the Manchester takes each bit of the code to be coded and performs an EX-OR operation with the clock signal. The frame format is similar to that of a UART. The functionality of the Manchester decoder is more complex, because it involves clock recovery and centre sampling.

Example 5.2. Convert NRZ data to Manchester encoded data as shown in Fig. 5.6a.

The conversion from NRZ to Manchester encoded data is shown in Fig. 5.6b. The NRZ data is ex-ored with the baud clock.

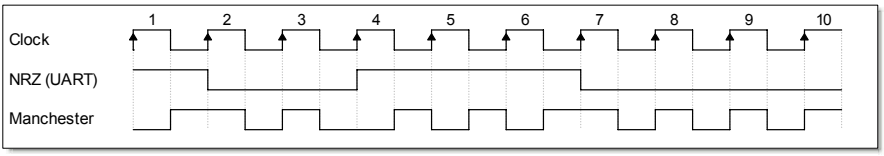


Fig. 5.6. a Timing diagram of NRZ to Manchester code

```
module ( input baud_clk, nrz, output manch )

assign manch = baud_clk ^ nrz;

endmodule
```

Fig. 5.6. b Verilog code for converting NRZ encoded data bits to Manchester code

5.2 Physical Layer for Serial Communication

The data frame created by the UART is transmitted by using different physical standards. The concept of physical layer interface and protocol (both widely used in digital data communications terminology) can be explained using the analogy of human speech communication.[25]

5.2.1 RS-232-based Point-to-Point Communication

The RS-232 standard is slowly fading away from modern electronics. Present-day personal computers classify the RS-232 as a legacy port that been superseded by the universal serial bus (USB) port. Despite this, many interfaces around the world, especially in the embedded domain, are still built around the RS-232 electrical interface. RS-232 uses single ended communications, with a common ground connecting both communicating devices. The clock signal is not exchanged between communicating devices. Figure 5.2 shows a commonly used RS-232 connection, referred to as the null-modem configuration.

5.2.2 RS-485-based Multi-point Communication

The RS-485 standard continues to be used as a multi-point communication standard. In RS-485, along with the original signal, its complement is also sent out and received by the communicating devices. Any noise introduced in the transmission path is cancelled out by cancelling the common voltage between both the complemented and original bit-stream. As shown in Fig. 5.7, RS-485 is implemented as a half-duplex, where a single twisted wire pair is used for both transmitting and receiving. A full-duplex configuration consists of separate twisted pairs for transmitting and receiving. The transmitting pin of the UART, Tx, is connected to the data in (DI) pin of each RS-485, and the Rx pin is connected to the receiver out (RO) pin.

---

[25]Human vocal cords generate frequencies in a bandwidth (~4 to 20 kHz) that the human ear comprehends. The way a human ear recognizes human sound is akin to the physical layer interface. In data communication, the physical layer interface defines the voltage level of communication and the data rate. The concept of protocol has to do with human language. Though the ear may be able to make out human sound, it may not be able to make sense of it , if the spoken language (Hindi, Japanese…) is not known. Thus, a single physical layer supports multiple protocols just the same way as humans communicate in different languages.

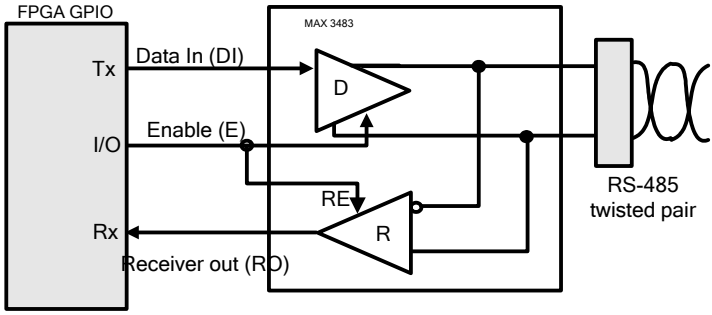


Fig. 5.7. RS-485 Differential signal generation

An example of a multi-point connection using RS-485 is shown in Fig. 5.8. The RS-485 standard defines use of 32 transceivers on the RS-485 twisted wire bus. The Modbus® protocol is a commonly used data communication protocol using the RS-485 medium. Modbus® is a widely accepted protocol for communication between devices of different vendors in industrial automation. Though the original Modbus® standard has been superseded by various related standards such as Modbus® Plus, the Modbus® protocol serves as a good example for understanding master–slave communication using RS-485. The Modbus® message structure consists of four fields. The first field consists of the device address, followed by function code, data and error check bytes. Tables 5.1 and 5.2 show the Modbus® message structure and common functions.

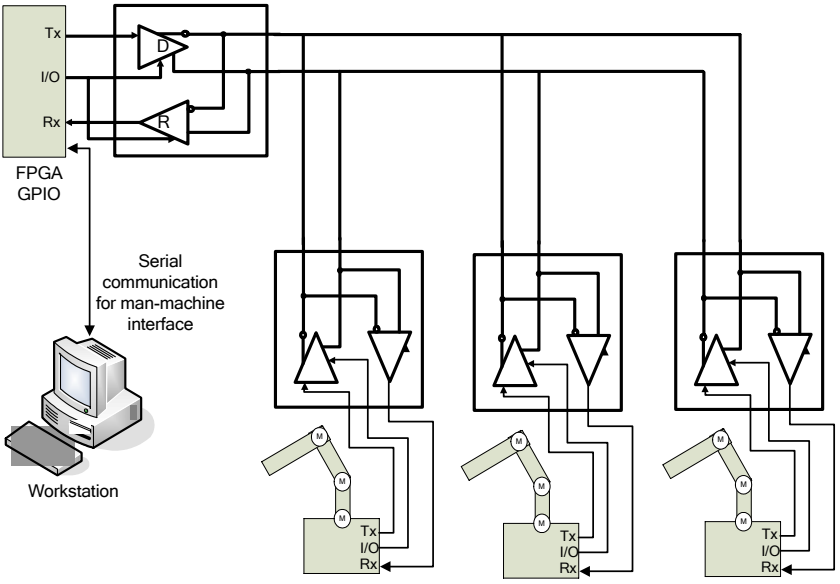


Fig. 5.8. Multi-point communication using RS-485

Table 5.1. Modbus® message structure [1]

Field	Description
Device address	Address of receiver
Function code	Code defining message type
Data	Data block with additional information
Error check	Numeric check value to check for communication errors

Table 5.2. Common Modbus® functions [1]

Code	Description
01	Read coil status
02	Read input status
03	Read holding registers
04	Read input registers
05	Force single coil
06	Preset single register
07	Read exception status
:	:
15	Force multiple coil
16	Preset multiple registers
17	Report slave ID

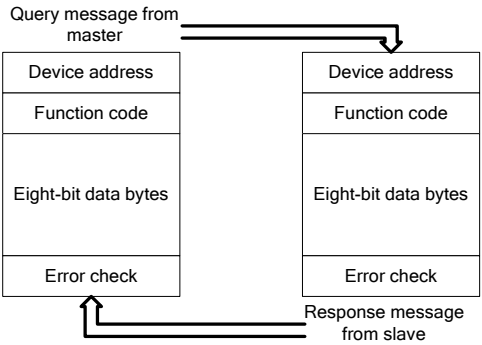


Fig. 5.9. Modbus message transaction

A query message consisting of a slave device address, function code and data is initiated by the Modbus® master. The addressed slave device responds with the requested data. A generic Modbus® message transaction between master and slave is shown in Fig. 5.9. The length of a message transaction depends on the size of the data frame component of the message. A byte-by-byte transaction detail of

a Modbus® master reading a coil status from a slave is shown in Table 5.3. A master robot controller uses read, force and preset functions to monitor and command robots using the Modbus® protocol.

Table 5.3. Sample transaction of Modbus® master and slave [1]

Modbus function 1 query			**Modbus function 1 answer**		
Byte	**Value**	**Description**	**Byte**	**Value**	**Description**
1	1..247	Slave address	1	1..247	Slave address
2	1	Function code	2	1	Function code
3	0..255	Starting address, high byte	3	0..255	Data bytes ….N
4	0..255	Starting address low byte	4 N+3	0.255	Bit pattern of coil values
5	0..255	Number of coils,	N+4		
6	0..255	high & low Bytes			
7 (..8)	LRC/ CRC	Error check		LRC CRC	Error check

5.3 Serial Peripheral Interface (SPI)

The serial peripheral interface (SPI) is a four-wire connection developed by Motorola to provide an interface between a microcontroller and peripherals. SPI is a synchronous protocol, where data transfer between the master and the slave is referenced to a common clock. The clock is generated by the master, and the slave uses it for synchronization.

Four ports are used by the master for SPI communication:

- MISO (master in slave out)
- MOSI (master out slave in)
- SCLK (serial clock)
- CS (chip select).

The ports of the slave for SPI are

- SDI (slave data in)
- SDO (slave data out)
- CLK (clock from master).

As shown in Fig. 5.10, there is a serial shift register in both master and slave. Data transfer starts with the master writing data to be transmitted to its SPI data register. The slave receives data from the MOSI on its SDI line and simultaneously sends out the contents of its SPI data register to the MISO port of the master. Like the UART, the SPI port is a standard peripheral in most microcontrollers.

A hardware instantiated SPI[26] port gives the flexibility of having the desired number of SPI ports as per the requirement of the application.

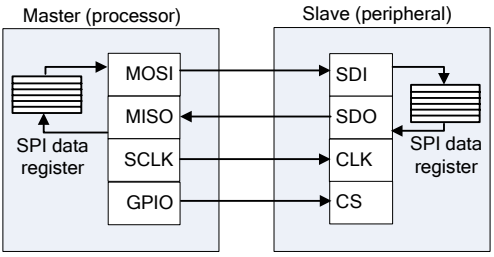


Fig. 5.10. SPI communication detail between two devices

The data of SPI is centre sampled, at a point furthest from the signal transition. The clock polarity bit (CPO) is used to set the active clock edge for sampling. Figure 5.11 shows the CPO bit set to "0", for setting the clock positive edge for data sampling. Similarly setting the CPO bit to "1" sets the clock negative edge for data sampling.

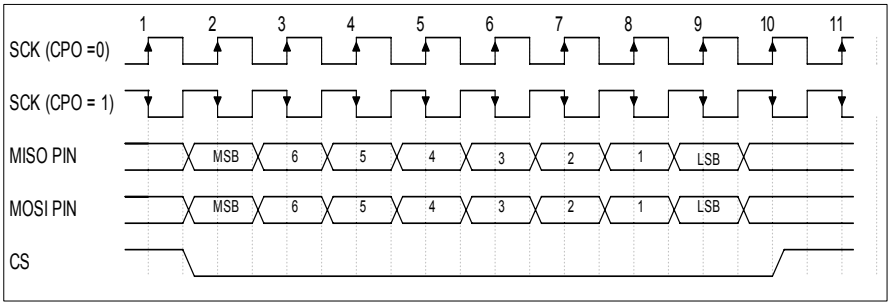


Fig. 5.11. SPI communication waveforms

Protocol Translator

Different protocols can co-exist in a robotic environment. Due to the proprietary nature of data communication protocols, one protocol needs to be converted to another for use by the control system. A protocol translator helps in converting data from one protocol to another. The protocol translator consists of a memory space where decoded data from protocols 1 and 2 are stored. Data from this memory space are converted to the required protocol format and then re-transmitted. Figure 5.12 shows the block diagram for the converter.

[26] The SPI peripheral is also provided by PLD vendors to attach with the on board soft/hard processor on the FPGA.

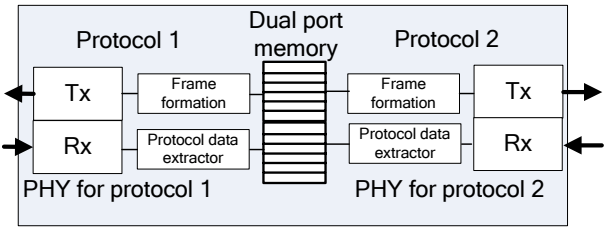


Fig. 5.12. Protocol 1 to protocol 2 translation using a FPGA

5.4 Signal Conditioning with FPGAs

Signal conditioning is needed to correctly decipher signals coming from the outside world. It helps in extracting the right frequency content from a signal or pattern from an image. Applying it to our hypothetical robot example, it is a useful tool to find robot movements based on position and image sensors. A commonly used signal conditioning block diagram is shown in Fig. 5.13.

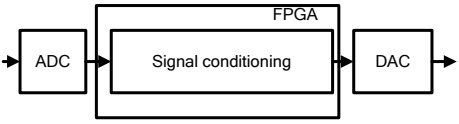


Fig. 5.13. Signal conditioning block diagram using a FPGA

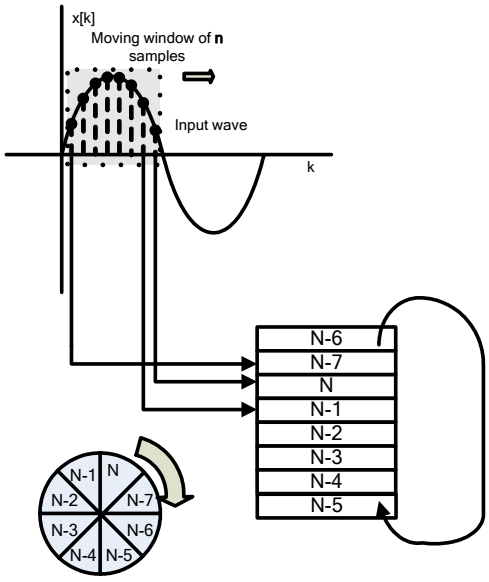


Fig. 5.14. Moving average using a circular buffer

Low-Pass Averaging Filter

A low-pass averaging filter takes the average of the last **n** samples of incoming **k** samples of data. This is a primitive way of filtering out high-frequency noise. For averaging digital data samples, the data samples are stored in buffers, as shown in Fig. 5.14. When a new sample is received, data in the entire array of input registers need not be shifted. A circular buffer is created, which is a wrapped around version of the linear buffer shown. In a circular buffer, a modulo **n** counter is used as an address pointer, which points to the location where the most recent sample is to be stored. This pointer overwrites the last sample of the moving window of **n** samples.

A FPGA-based logic to average the last four samples of incoming ADC data is shown in Fig. 5.15. The incoming data are passed through three delay blocks. The summation of the data values in the memory is used to calculate the average of the input samples. Figure 5.16 shows the Verilog code which uses a 12-bit register to take input values. A non-blocking Verilog construct is used to create a data pipeline for storing the last four data samples.

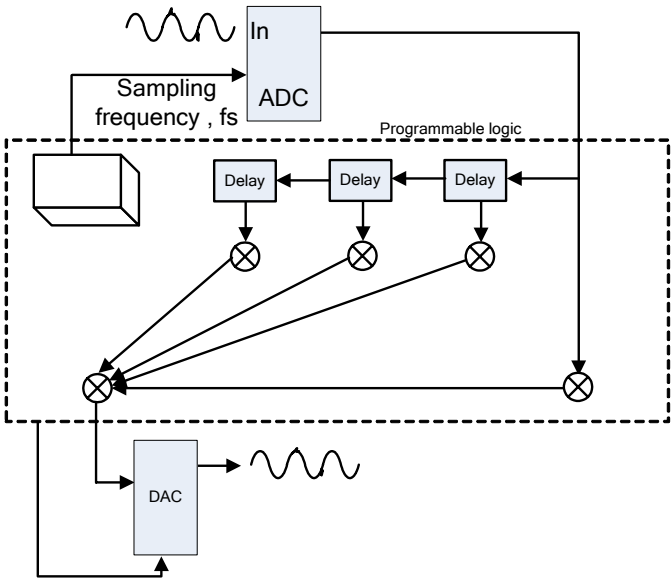


Fig. 5.15. Moving average filter using a FPGA [2]

Example 5.3. Write Verilog code for averaging the last four samples of the incoming waveform, as shown in Fig. 5.15.

```
module filter ( input clk, rst , input [11:0] adc_ip, output reg[11:0] dac_op);

reg [11:0] temp [3:0];

always @ ( posedge clk )
begin
        if (rst)
            begin
                    temp[0] <= 0;
                    temp[1] <= 0;
                    temp[2] <= 0;
                    temp[3] <= 0;
            end
        else
            begin
                    temp[0] <= adc_ip;
                    temp[1] <= temp[0];
                    temp[2] <= temp[1];
                    temp[3] <= temp[2];
              dac_op <= ((temp[3] + temp[2] + temp[1] + temp[0]) / 4); // >>2 right shift by two
            end
end

endmodule
```

Fig. 5.16. Moving average filter using a FPGA

Problems

1. The local interconnection network (LIN) is a variation of the UART. It is used for networking non-time-critical components in an automobile. The network consists of one master node connected to 2–10 slave nodes. Because there is only one master, no arbitration is needed. Develop a HDL-based logic for a LIN bus master and slave device. Implement a multi-drop LIN-based network using FPGA boards.
2. The Modbus® protocol is one of those long surviving protocols. Develop a HDL code that can read the Modbus® master initiated function 1 query frame and respond using the function 1 answer format (refer to Table 5.3 and the Modbus® reference guide).
3. One of the widely used distance measuring sensors is the GP2D02. The output of the sensor is in the form of a byte of data synchronized with the help of a Vin signal (see Fig. 5.17). Write a Verilog code that accesses the distance information from the sensor.
4. Develop a Verilog code for calculating the root mean square (rms) value of an input waveform. Use the block diagram given in Fig. 5.18. The logic should use a moving window of 20 samples. The estimated square root value can be obtained by using the equation shown in the block diagram.

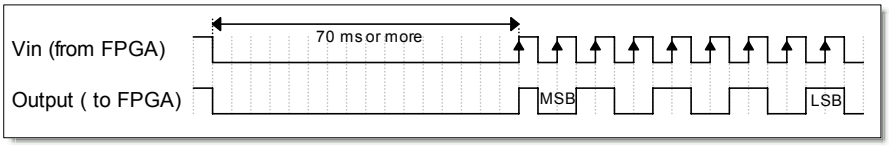


Fig. 5.17. Partial timing diagram of GP2D02 distance measuring sensor output (8-bit)

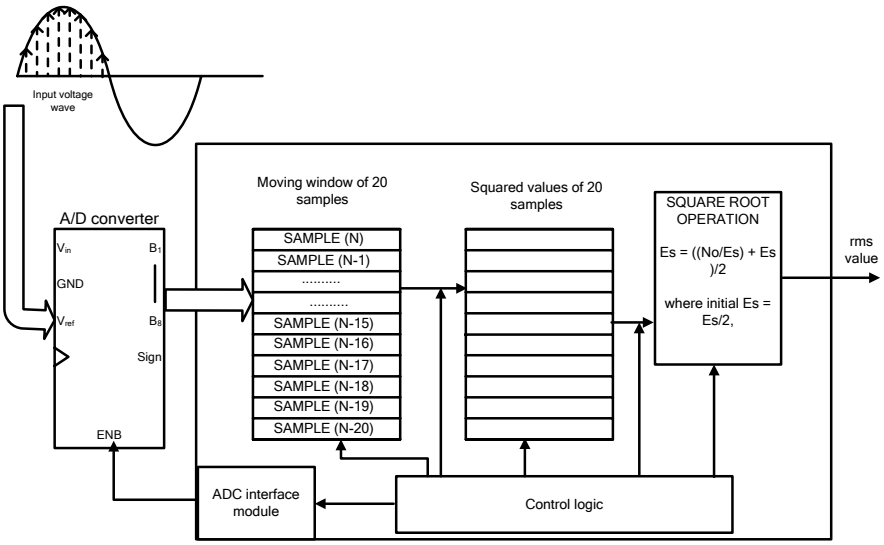


Fig. 5.18. Block diagram for calculating a rms value using 20 input samples

References

1. Modicon Modbus protocol reference guide (1996). http://www.modbustools.com/PI_MBUS_300.pdf. Accessed 23 May 2008.
2. Shuler M, Chugani M (2005) Digital signal processing, a hands-on-approach. Tata Mc-GrawHill

6

Motor Control Using FPGA

For the robot controller application chosen in this book, the electric motor is the actuator of the control scheme. There are many types of electric motors available that can be used for robot applications. The control scheme used for each motor type may be unique, but the overall control approach for motion control is similar across motor drives.

6.1 Introduction to Motor Drives

Robots make extensive use of electric motor drives as actuators. Electric motors as actuators for robot joint movement score over hydraulic and pneumatic actuators. Electric motor based control schemes are cleaner and easier to implement. Early robots used brushed DC motors as actuators. Though the control of DC motors is simple, it is not preferred due to frequent maintenance and possible hazard because of sparking of brushes. Many robot manufacturers now use AC servomotors in place of DC motors. Fast digital circuits have made implementation of complex algorithms feasible. They are needed for control of AC motors. For completeness, this chapter includes FPGA-based control of DC motors along with control techniques used for AC servomotors.

Each motor drive provides basic functionality for

- Setting of speed reference
- Control of motor direction (forward or reverse)
- Setting of acceleration/deceleration rate
- Run/Jog operating controls
- Emergency stop using dynamic or regenerative braking.

6.2 Digital Block Diagram for Robot Axis Control

Robot axis motion control consists of three control loops shown in Fig. 6.1. All three loops work together to move the robot axis to the position commanded by the

profile generator. The output of the position controller becomes the reference for the speed loop. In a similar way, the output of the speed controller is the reference for the motor current/torque loop. The job of the controller for all loops is to minimize the error between the reference and feedback values.

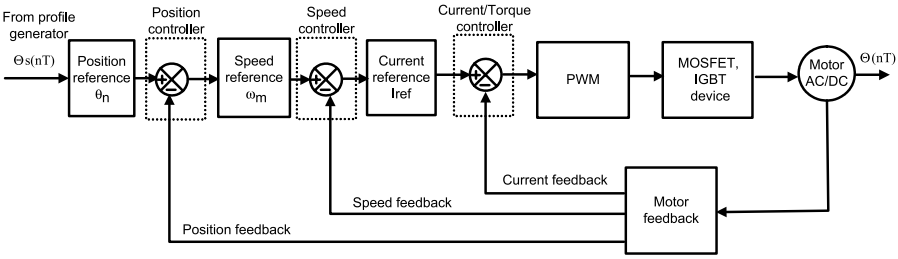


Fig. 6.1. Control loops of motor control system

6.2.1 Position Loop

The position loop is the outermost loop shown in Fig. 6.1. For the robot control system, it indicates the position of each axis of the robot. The inverse kinematics algorithm computes the desired value of rotation, $\boldsymbol{\theta_n}$ needed for each axis, and this becomes the reference to the position loop. The error of the position loop goes to a controller that generates a reference for the speed loop. A commonly used reference for the position loop is a profile generator. Because the time constant of physical movement of the robot axis is of the order of milliseconds, a software-based approach can be used for profile generation. The software code, shown in Fig. 6.2, illustrates profile generation for one axis of the robot controller. A free running timer peripheral is used to generate periodic interrupts, on which the process can run the position control algorithm.

```
On interrupt /* from timer */
{
     Θ1 = encoder_counter1 /* current position of axis one*/
     Θ1S = setpoint_register1 /* set point Θ1S axis one*/

   /*  error generating junction */
     e1 =  Θ1S  -  Θ1

/* PID controller */
     i = i_old + (e1  +  e1_old)/2; /* integrator*/
     i_old = i ;
     d =  e1  -  e1_old ;              /* derivative term */
     e1_old = e1 ;
     c = (kp *e ) + (ki * i) + (kd *d ) ;
}
```

Fig. 6.2. Control loops of a motor control system

6.2.2 Speed Loop

The job of the speed control loop is to correct the speed error by regularly sampling the speed reference and measured speed variable. The speed error is fed to a controller to generate a reference for the current loop, as shown in Fig. 6.3.

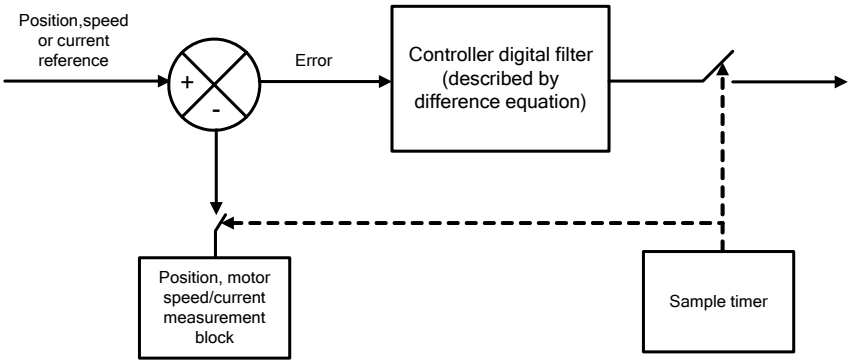


Fig. 6.3. Digital block diagram of control loop

Because the update time of a speed loop controller varies from 1–10 ms, both software and hardware approaches can be used for this update time requirement.

6.2.2.1 Software Approach

The embedded processor in the FPGA device is used to implement the control algorithm. An interrupt from the processor is used to run the proportional integral derivative (PID) software routine. Because the speed loop sampling time of a conventional servocontroller is of the order of milliseconds, processor-based architecture is suitable.

6.2.2.2 Hardware Approach

Generally, a proportional integral (PI) controller is used for motor drives. The exact equation of the PI controller transfer function is deduced using root locus or frequency domain analysis in the continuous time domain. For the digital FPGA domain, the PI controller transfer function is synthesised as a difference equation using a bilinear transform. ***Ts*** is the sampling time of the control loop.

$$s = \frac{2}{Ts}\frac{z-1}{z+1} \tag{6.1}$$

Substituting the value of "s" in terms of "z" and "Ts" results in a difference equation. The terms **u** (**k**), **u** (**k** – 1), **e** (**k**) and **e** (**k** – 1) represent the controller output and error values at present time (**k**) and a previous sample time (**k** – 1). The output of the PI controller equation is then,

$$\mathbf{u}(\mathbf{k}) = \mathbf{u}(\mathbf{k}-1) + [\mathbf{c}_1\mathbf{e}(\mathbf{k}) - \mathbf{c}_2\mathbf{e}(\mathbf{k}-1)] \tag{6.2}$$

$\mathbf{c}_1$ and $\mathbf{c}_2$ are constants, that change when sampling time **Ts** changes. The block diagram of the above PI controller difference equation reduces to a generic format as given in Eq. 6.2 and shown in Fig. 6.4. The arithmetic discussed in Chap. 2 is useful for computing Eq. 6.2. The timing diagram of the PI controller is shown in Fig. 6.5.

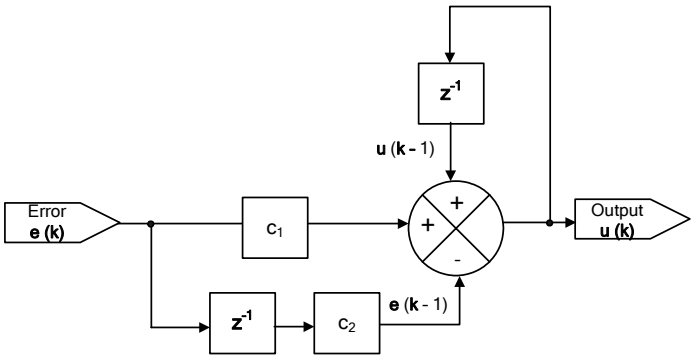


Fig. 6.4. Implementation of a PI digital controller

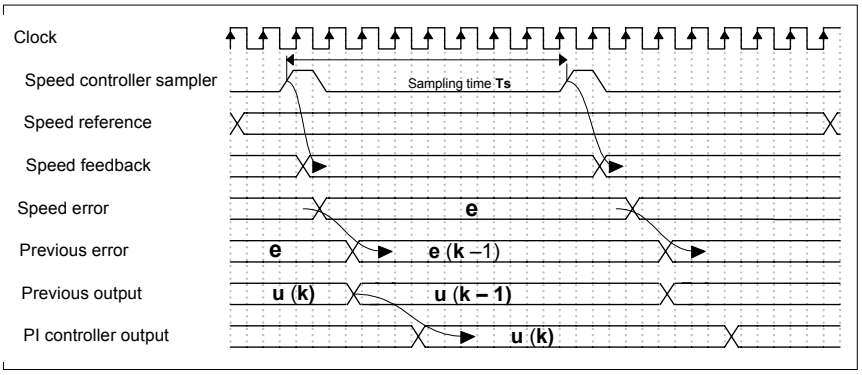


Fig. 6.5. Timing diagram of digital speed controller, with sampling time Ts

The difference equation represented in Fig. 6.4, consists of two delay elements, two coefficient multiplications and an addition/subtraction block. Because delay, multiplication and addition/subtraction are synthesisable these operations are implemented in FPGA fabric without consuming large logic resources. A state machine ensures that the output of the PI controller is activated in every sampling period and the controller is prevented from wind-up error.

6.2.3 Power Module

The output of the torque-current controller provides a set point to the power module of the firing control circuit. The sampling period of the firing circuit varies with the type of power module topology. For a single-phase rectifier circuit (see

Example 3.7), the sampling period for each firing circuit update is calculated using a zero crossing of the input waveform supplied by synchronizing transformer. For a 50-Hz input voltage to the rectifier, this time is 3.33 ms. To generate a sinusoidal PWM voltage using a three-phase bridge, the power devices of the bridge are switched at a frequency around 20 kHz.

6.3 Case Studies for Motor Control

The type of motor used for robot joint axis control varies. Simple robots use the stepper and DC motor for joint control. Contemporary industrial robots use AC servomotors such as a permanent magnet synchronous motor (PMSM) for axis control. This section describes different motors and their control techniques.

6.3.1 Stepper Motor Controller

A stepper motor is an electric machine that rotates in discrete angular increments. A cross-sectional view of the motor is shown in Fig. 6.6. The angular increment is used to calculate the number of steps needed to complete one revolution. Because stepper motors move to a commanded number of steps, many stepper motor applications do not require position sensing. This decreases the complexity of stepper motor movements. Stepper motors are used in a variety of applications such as printers, plotters, X–Y tables, image scanners, copiers, medical apparatus and other devices.

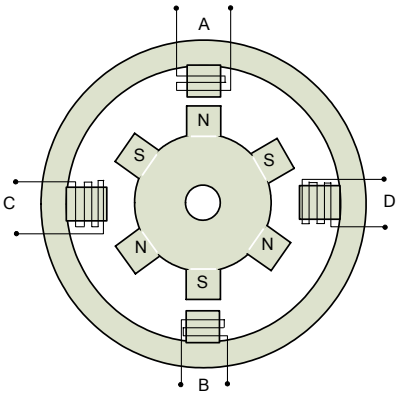


Fig. 6.6. Cross-sectional view of the stepper motor

From the digital control point of view, the stator poles of the stepper motor need to be periodically excited to cause movement of the permanent magnet rotor. The excitation table of a motor varies from options that give single step movement with one or two winding excitation. Simultaneous excitation of two windings provides greater torque than one winding. Tables 6.1 and 6.2 show half step excitation that increases the resolution of the movement.

Table 6.1. Stepper motor full step, single-phase excitation

	Winding A	**Winding B**	**Winding C**	**Winding D**	**Rotor position**
Mode 1	1	0	0	0	0
Mode 2	0	1	0	0	θ
Mode 3	0	0	1	0	2θ
Mode 4	0	0	0	1	3θ

Table 6.2. Stepper motor full step, two-phase excitation

	Winding A	**Winding B**	**Winding C**	**Winding D**	**Rotor position**
Mode 1	1	1	0	0	0
Mode 2	0	1	1	0	θ
Mode 3	0	0	1	1	2θ
Mode 4	1	0	0	1	3θ

Example 6.1. Write a Verilog HDL code that controls the speed and direction of a stepper motor working in single-phase excitation, as given in Table 6.1.

The code listed in Fig. 6.7 provides excitation to two of the four coils of the stepper motor stator. The FSM ensures that the correct sequence is followed for coil excitation. The direction of rotation is varied by changing the sequence of coil supply denoted by ***coil_supply_f*** and ***coil_supply_r***.

```
module stepper (input clk, rst, dir, output [3:0] coil_supply);

`define reset 3'd0   `define step1 3'd1   `define step2 3'd2  `define step3 3'd3
`define step4 3'd4


reg [2:0] ps, ns;  // present state (ps) and next state (ns) registers
wire clk_spd;
reg [3:0] coil_supply_f, coil_supply_r;

assign clk_spd = clk;   // Based on desired motor speed , the clk_spd is set

always @ (posedge rst or posedge clk_spd)  // state transition
     begin
          if (rst)
               ps <= `reset;
          else  ps <= ns;
     end

always @ ( ps )    // selection of next state and change of output

begin
case  (ps)
`reset : begin
                    ns <= `step1;
                    coil_supply_f <= 4'b0000;
                    coil_supply_r <= 4'b0000;
         end
`step1 : begin
                    ns <= `step2;
                    coil_supply_f <= 4'b0011;  //  4'b DCBA windings
                    coil_supply_r <= 4'b1001;
         end
`step2 : begin
                    ns <= `step3;
                    coil_supply_f <= 4'b0110;
                    coil_supply_r <= 4'b1100;
         end
`step3 : begin
                    ns <= `step4;
                    coil_supply_f <= 4'b1100;
                    coil_supply_r <= 4'b0110;
         end
`step4 : begin
                    ns <= `reset;
                    coil_supply_f <= 4'b1001;
                    coil_supply_r <= 4'b0011;
         end
default  begin
                    ns <= `reset;
                    coil_supply_f <= 4'b0000;
                    coil_supply_r <= 4'b0000;
         end
endcase
end

assign coil_supply = ( dir == 1'b1) ? coil_supply_f : coil_supply_r;

endmodule
```

Fig. 6.7. Verilog code for control of a stepper motor

6.3.2 Permanent Magnet DC Motor

The permanent magnet DC motor is one of the most commonly used motors. Its characteristic of providing a speed proportional to the applied voltage makes it very simple to control. As shown in Fig. 6.8, an **H**-bridge configuration is used to provide four-quadrant speed control to DC motors. The control scheme consists of a free running counter that generates a ramp signal. This ramp is used for setting the duty cycle of the PWM signal. As illustrated in Fig. 6.9, the counter value is compared with a control voltage (**V**c). The higher the value of **V**c, the higher the duty cycle of the PWM voltage. The voltage across the motor terminals is the average value of the duty cycle of the PWM.

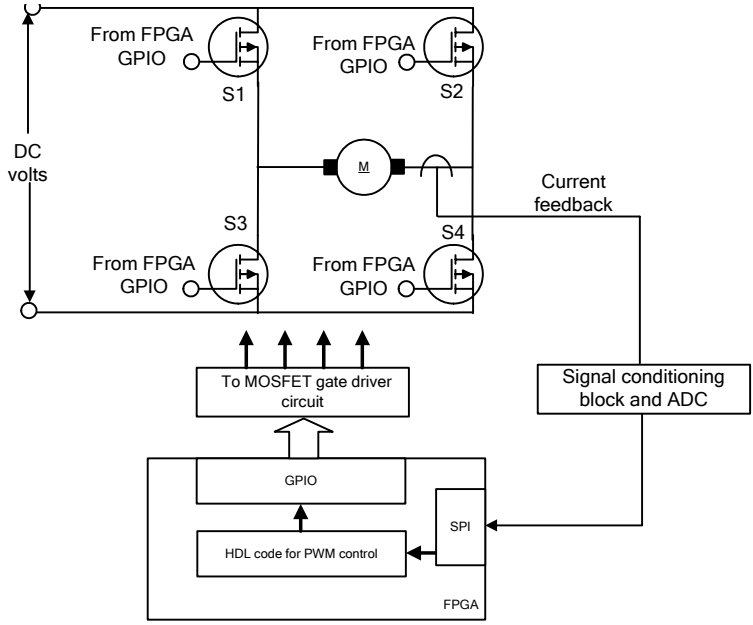


Fig. 6.8. Permanent magnet DC motor control using a field programmable device

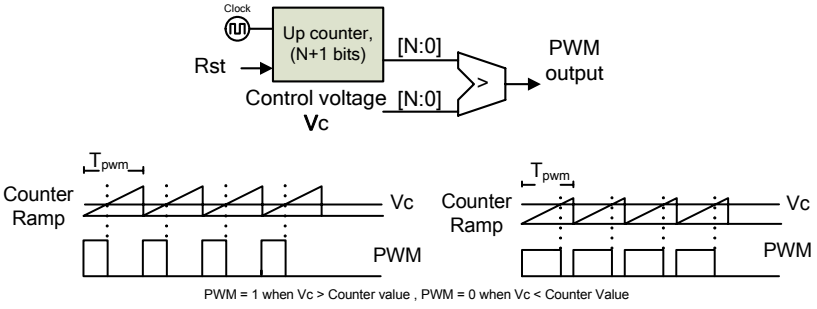


Fig. 6.9. Change in PWM duty cycle based on the value of the control voltage **V**c

```
module pwm (input wire [7:0] vc, input clk, input rst, output reg pwm);

reg [7:0] counter ;

always @ ( posedge clk)
begin
        if (rst)
                counter = 8'h00;
        else if (counter < vc )
                begin
                        pwm = 1'b1;
                        counter = counter + 1;
                end
        else
                begin
                        pwm = 1'b0;
                        counter = counter + 1;
                end

        end

endmodule
```

Fig. 6.10. Verilog code for PWM control of a PMDC motor

The Verilog code of Fig. 6.10 shows a counter circuit along with comparator logic. For values of **Vc** less than the counter value, the PWM output is set at logic 1, else it is set at logic 0. A section of the synthesis report in Fig. 6.11 shows identification of an 8-bit counter, 8-bit comparator and a 1-bit register for the PWM output signal.

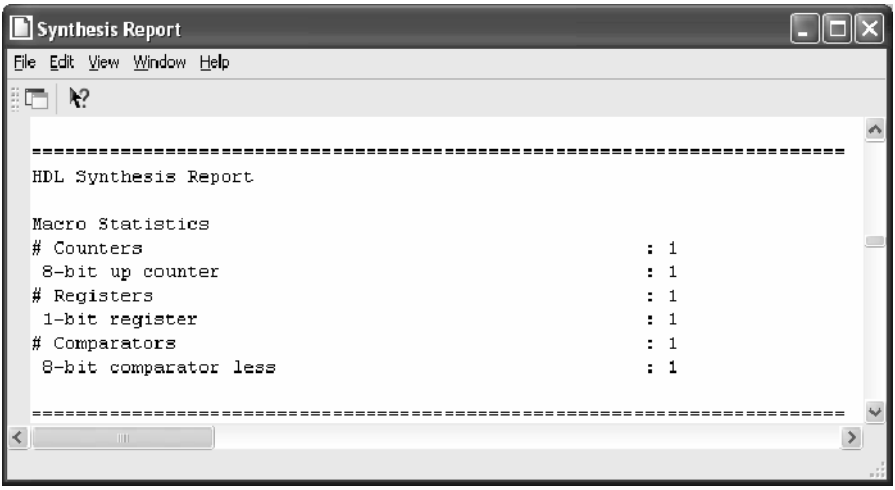


Fig. 6.11. Synthesis report of a PWM controller for a PMDC motor

Dead Time Control

The power device bridge is susceptible to shoot-through faults, when devices on the same leg turn on together. To prevent shoot-through problems, a finite delay is incorporated in the turn on and turn off of the upper and lower devices of a power bridge. Many DSPs targeted for motor applications have dedicated hardware for dead time control. A programmable dead time timer is interlocked with the *drive ok* permissive. Figures 6.12 and 6.13 show dead time delay logic and HDL implementation to provide delayed output of a device triggering signal.

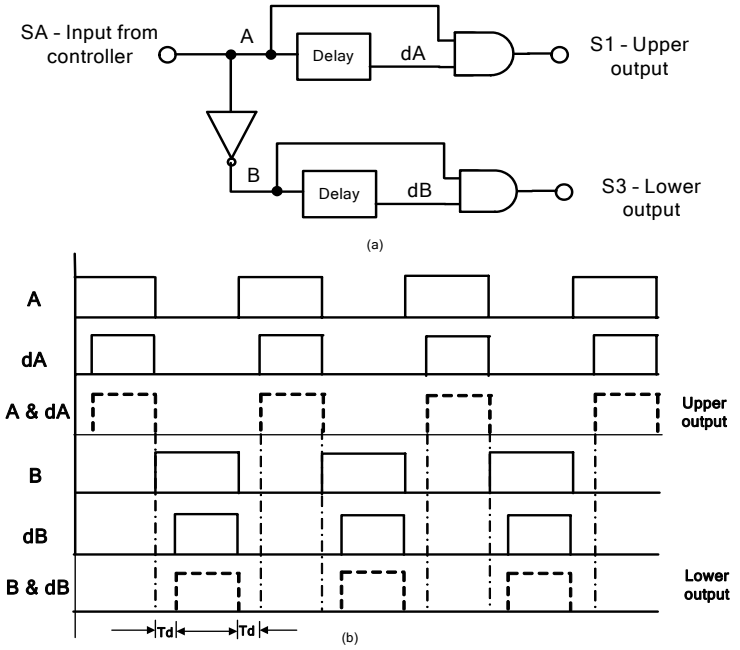


Fig. 6.12. Logic for implementing a dead band for a PMDC motor H-bridge

The HDL code shown in Fig. 6.13 consists of the logic, to control the upper device of a given leg. The input signal **A** is multiplied by a time (dead-time) delayed signal **dA** to obtain the control signal of the upper device. Dead time for the circuit can be modified by changing the constant (**6'h3C**), that is used for comparison with the **count** value.

The reader is encouraged to write a HDL code that will control the upper and lower devices of the three legs of a power bridge (since dead-time is a frequently used component in motor control, try to design using a re-usable instance based approach).

```
module deadtime (input rst ,a ,clk , output upper_sw );

reg [5:0] count;
reg da ;

always @ (posedge clk or posedge rst)
        begin
                if (rst)
                        begin
                                da <= 0 ;
                                count <= 0 ;
                        end
                else if (a)
                        begin
                                count <= count +1 ;
                                if (count == 6'h3C)  // count based on clock frequency
                                    da <= 1'b1;
                        end
                else if ( ~a)
                        begin
                        count <= 0 ;
                        da <= 1'b0;
                        end

        end

assign upper_sw = a && da;

endmodule
```

Fig. 6.13. Verilog code for setting dead time between devices on the same leg of an H-bridge

6.3.3 Brushless DC Motor

A brushless DC motor has a rotor with permanent magnets and a stator with windings. It is also referred to as a trapezoidal permanent magnet AC motor. The working is similar to a DC motor, but here the rotor position is determined by sensors, and the winding current is switched by control electronics. This eliminates the need for brushes and commutators in conventional DC motors. The removal of brushes leads to less noisy and more reliable operation. A Hall effect sensor is used to provide information to synchronize stator excitation with rotor position. The rotor magnets are used as triggers to the Hall sensor. The three Hall sensors are placed 120° apart on the stator frame.

The energized stator field leads the rotor magnet and moves ahead as soon as the rotor is about to align with it. As shown in Fig. 6.14, signals from the three Hall sensors are processed by a logic circuit to determine the position of the rotor at any time. This information is used by the driver circuit to energize appropriate motor windings by turning on transistor switches of different legs of the transistor bridge. With the help of the rotor position data, a six-step trapezoidal control is obtained by turning on/off different transistors shown in Table 6.3.

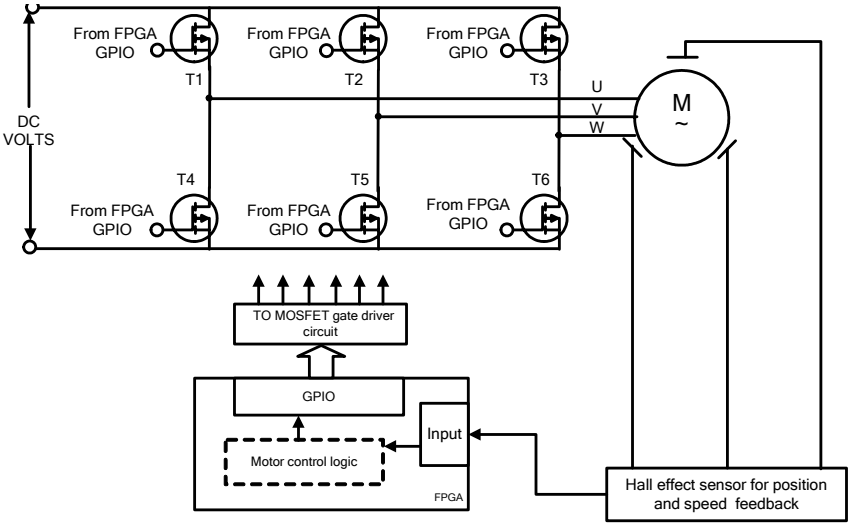


Fig. 6.14. Brushless DC motor control circuit using a FPGA

Table 6.3. Six step changes for 120° difference in Hall sensor positive edges

Hall sensor feedback			**Transistor bridge**					
S1	S2	S3	T1	T2	T3	T4	T5	T6
1	0	1	1	0	0	0	1	0
1	0	0	1	0	0	0	0	1
1	1	0	0	1	0	0	0	1
0	1	0	0	1	0	1	0	0
0	1	1	0	0	1	1	0	0
0	0	1	0	0	1	0	1	0
1	0	1	1	0	0	0	1	0

6.3.4 Permanent Magnet Rotor (PMR) Synchronous Motor

A permanent magnet rotor synchronous motor provides synchronous operation from no load to full load for applications requiring precise speed control. These motors incorporate a permanent magnet in the rotor to form salient poles for synchronous operation. PMR motors synchronize with the applied frequency with zero slip. For synchronized applications requiring variable speed, these motors are very well suited because they do not need feedback for speed regulation. A conveyor belt like the one shown in Fig. 6.15 can use open-loop synchronized drive operation. In this open-loop V/Hz control scheme, sudden changes in speed reference can cause the motor to lose synchronism.

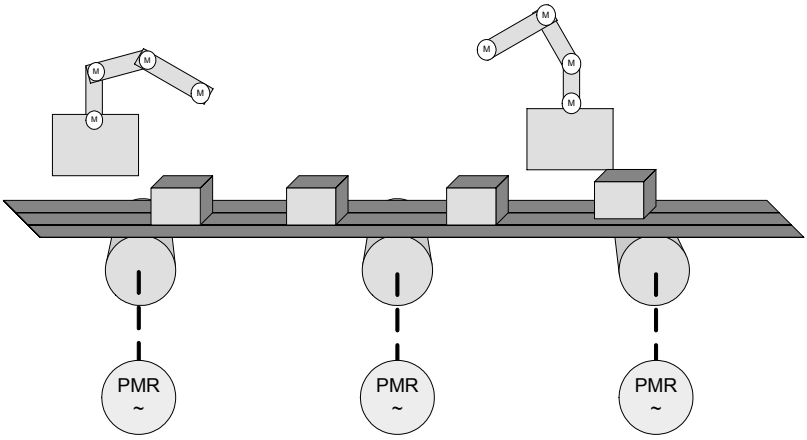


Fig. 6.15. Synchronized conveyor belt operation using PMR motors

Many algorithms are available to generate three-phase variable frequency voltage for controlling a PMR motor. The space vector pulse width modulation (SVPWM) is one algorithm. SVPWM uses base vectors formed by the eight unique switching vectors. The switching sequence for the inverter switches is mentioned in Table 6.4. Each combination is generated by six devices of the inverter circuit shown in Fig. 6.16a. One 360° revolution of the space vector operation is divided into six sectors of 60° each. For each sector, two base vectors are defined, and their intermediate values are calculated by projecting the base vectors. The switching intervals for one sampling interval **Ts** is shown in Figure 6.16b.

Table 6.4. Space vector voltage based on device switching

State	On devices	Space vector voltage
0	4 5 6	$\mathbf{V}_0$ (000)
1	1 5 6	$\mathbf{V}_1$ (100)
2	1 2 6	$\mathbf{V}_2$ (110)
3	2 4 6	$\mathbf{V}_3$ (010)
4	2 3 4	$\mathbf{V}_5$ (011)
5	3 4 5	$\mathbf{V}_5$ (001)
6	1 3 5	$\mathbf{V}_6$ (101)
7	1 2 3	$\mathbf{V}_7$ (111)

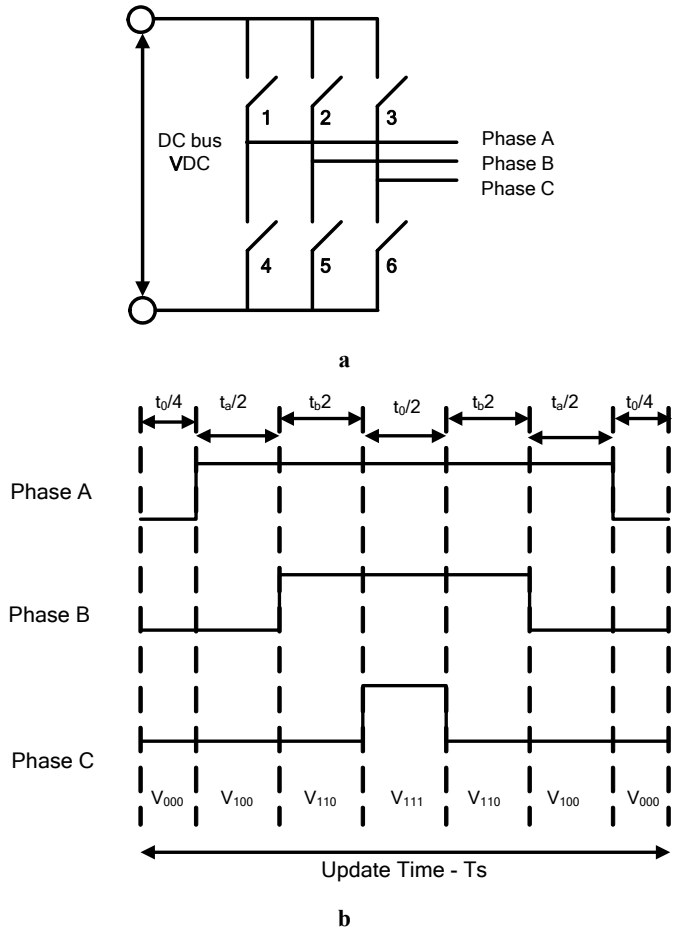


Fig. 6.16. a Concept of SVPWM generation of switching signals for one sampling period Ts;. **b** PWM counter resolution for different system clock frequencies

A reference voltage |**V**ref | is derived using the volts/hertz profile for the motor. The value of |**V**ref | is used to calculate the dwell time **T**a,**T**b and **T**0 of the base vectors. The time periods **T**a and **T**b are determined using Eqs. 6.3–6.5. VDC is the DC link voltage to the inverter bridge.

$$T_a = \frac{2 * Ts * | Vref | * \sin\theta}{V_{DC} * \sqrt{3}}. \tag{6.3}$$

$$T_b = \frac{2 * Ts * | Vref | * \sin(60 - \theta)}{V_{DC} * \sqrt{3}}. \tag{6.4}$$

$$T_0 = Ts - (Ta + Tb) \tag{6.5}$$

A complete digital block diagram for the SVPWM is shown in Fig. 6.17. It involves computation of Eqs. 6.3–6.5 using HDL.

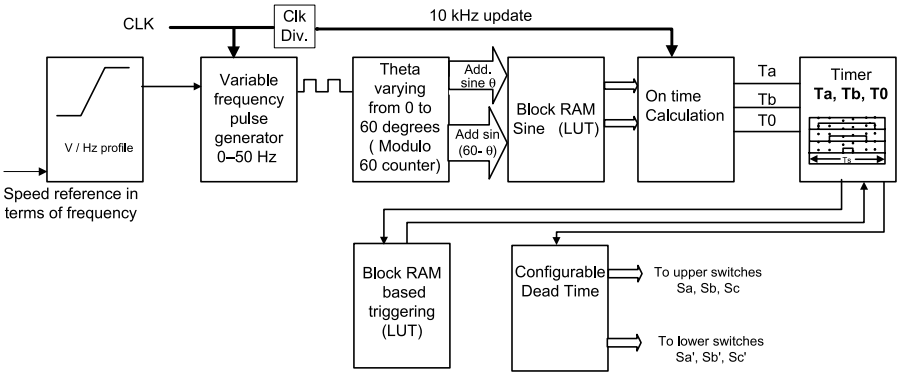


Fig. 6.17. Digital data flow diagram of SVPWM logic block. Input to block is the reference frequency, and output is the dead time compensated triggering signal for a three-phase bridge.

6.3.4.1 Sine Look-up Table

The value of sine theta varies from zero to sixty so a block RAM-based LUT is used for storing values of sin θ. The value of sin θ is multiplied by 255 to make it an 8-bit integer value, stored in the LUT. The software simulation shown in Fig. 6.18 shows the working of sin LUT with variations of theta and sector.

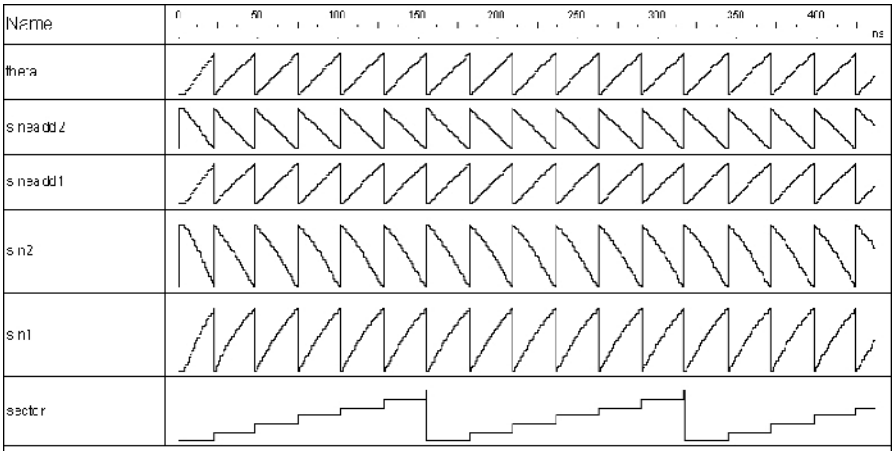


Fig. 6.18. Simulation results of SVPWM sine LUT functioning with changing theta and sectors

6.3.4.2 Variable Pulse Frequency Generator

Frequency for changing the speed of the motor is realised by using a variable pulse frequency generator. The generator varies the incremental rate of theta for different command speeds.

Example 6.3. Calculate theta frequency in hertz for a four pole motor to rotate the motor at 1500 rpm. The rate of theta change is given by Eq. 6.6.

$$\frac{d\theta}{dt} = 360 * \text{rps}\ (\text{revolutions per second}) \tag{6.6}$$

$$= 360 * \frac{1500}{60} = 9\ \text{kHz}.$$

The HDL code for the pulse generator takes a count proportional to the desired speed and generates a pulse train for increasing angle theta. Every time the internal counter equals the count to be reached, a bit is set for generating the pulse train. The entire code is listed in Fig. 6.19.

```
module pulse_gen ( input [16:0] count, clk , rst output pulse  );

        reg [16:0] count_int;
        reg temp,dn ;

        assign pulse = dn;

        always @ ( posedge clk  or posedge rst )
                begin
                        if (rst)
                                begin
                                        count_int = 0;
                                        temp = 1'b1;
                                end
                        else if ( count_int != count   )
                                begin
                                        count_int = count_int  + 1;
                                        dn = 1'b0;
                                        temp = 1'b0;
                                end
                        else if ( count_int == count  && temp == 1'b0 )
                                begin
                                        count_int = 0;
                                        dn = 1'b1;
                                        temp = 1'b1;
                                end

                end

endmodule
```

Fig. 6.19. Variable frequency pulse generator

6.3.5 Permanent Magnet Synchronous Motor (PMSM)

A PMSM motor is also referred to as a sinusoidal permanent magnet AC motor. It is widely used as a joint actuator for industrial robots. The control scheme for PMSM, shown in Fig. 6.20, consists of two control loops: (i) an inner current control loop and (ii) an outer speed control loop. The reference command speed $\boldsymbol{\omega}_m^*$ is compared with the actual speed of the drive, $\boldsymbol{\omega}_m$ and the speed error is processed through the speed controller.

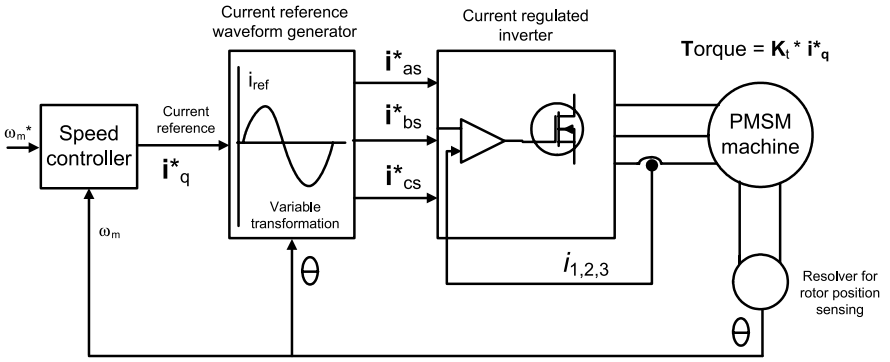


Fig. 6.20. Control scheme for a permanent magnet synchronous motor

At the kth sampling instant the motor speed error of the controller is given by

$$\omega_{error}(k) = \omega_m^*(k) - \omega_m(k) \tag{6.7}$$

and the change in speed error is given by

$$\Delta\omega_{error}(k) = \omega_{error}(k) - \omega_{error}(k-1). \tag{6.8}$$

The **q**-axis current command in terms of the torque command is,

$$i_q^*(k+1) = \frac{T_e^*(k+1)}{K_t} \tag{6.9}$$

A limiter is applied to saturate the controller output at a maximum inverter or motor (whichever is small) current rating. The output of the limiter is written as

$$i_q^*(k+1) = \begin{cases} i_{q\max}^* & \text{for } i_q^*(k+1) \geq i_{q\max}^* \\ -i_{q\max}^* & \text{for } i_q^*(k+1) \leq -i_{q\max}^* \end{cases}. \tag{6.10}$$

The output of the speed controller is the torque command for the drive, **Te***. The electrical torque of the drive is directly proportional to the **q**-axis current component of the PMSM. Dividing the torque command by the torque constant, the **q**-axis current command is obtained. The scaling of the torque component of current is motor specific and is obtained by dividing the torque command by the PMSM torque constant, $\mathbf{K_t}$. The rotor position and speed sensed with a resolver coupled to the shaft of the PMSM. The speed/position measurement block generates electrical shaft angle position Θe, which is used to get abc-axis reference currents. The electrical angle Θe is equal to the mechanical angle Θm multiplied by the motor pole pairs. The **d**-axis current component which decides the de-magnetization current component of the PMSM is kept at zero. The reference values of the **q**-axis and rotor position (angle) are used to calculate three-phase reference currents i_{as}^*, i_{bs}^* and i_{cs}^* as shown in Eq. 6.11. The PMSM motor in this discussion is assumed to have four pole pairs.

$$\begin{aligned} i_{as}^* &= i_q^* \sin 4\theta_m \\ i_{bs}^* &= i_q^* \left(\sin 4\theta_m - \frac{2\pi}{3} \right) \\ i_{cs}^* &= i_q^* \left(\sin 4\theta_m - \frac{4\pi}{3} \right). \end{aligned} \tag{6.11}$$

The current controller module compares the three-phase reference currents with the actual currents and generates switching signals for power devices of the PWM inverter. The controlled switching of the PWM inverter generates a variable frequency, variable magnitude, three-phase sinusoidal motor current to achieve the desired speed regulation.

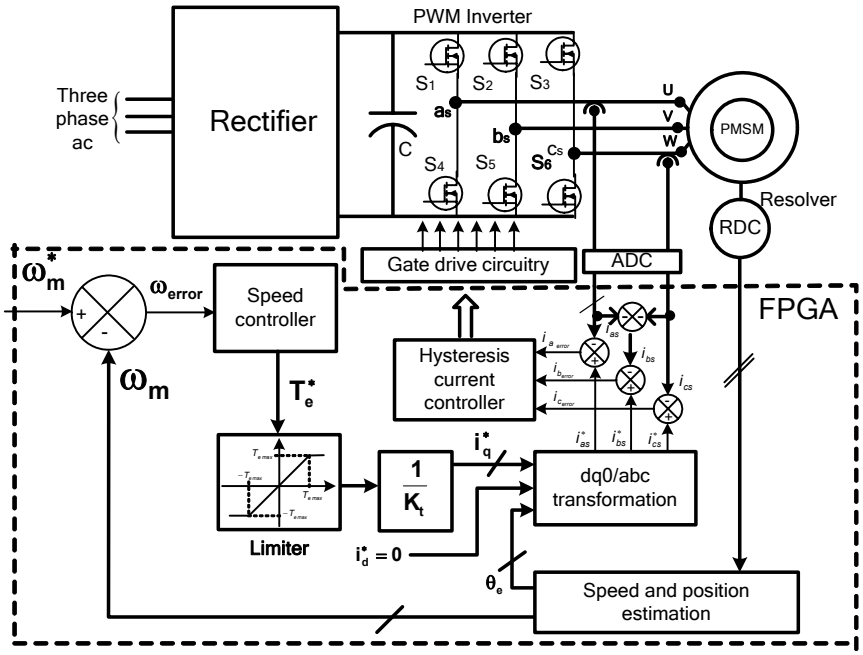


Fig. 6.21. Control scheme for a permanent magnet synchronous motor

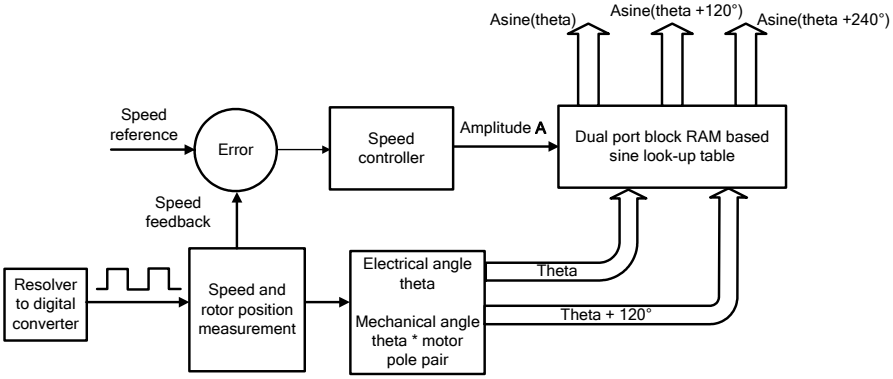


Fig. 6.22. Digital block diagram of a hysteresis reference current generator

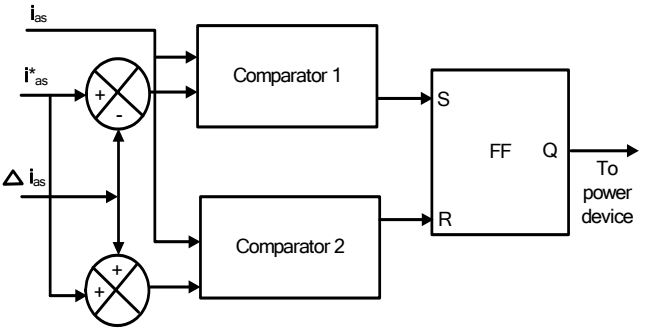


Fig. 6.23. Logic circuit for hysteresis controller

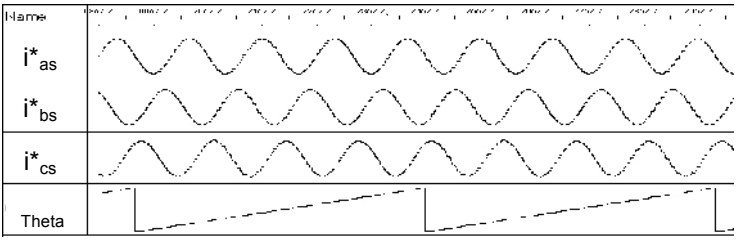


Fig. 6.24. Simulated current reference waveform

The FPGA implementation scheme for PMSM motor control is shown in Figs. 6.21 and 6.22. The digital block diagram of the code is shown in Fig. 6.22. Actual stator currents $\mathbf{i}_{as}$ and $\mathbf{i}_{cs}$ are sensed using two current sensors, and the third current i_{bs} is calculated as the negative sum of the two sensed currents. The actual currents are compared with the reference currents and current errors are sent to respective hysteresis current controllers (see Fig. 6.23). The switching pulses generated by current errors $\mathbf{i}_{aerror}$, $\mathbf{i}_{berror}$ and $\mathbf{i}_{cerror}$, are applied to devices in the inverter legs of phase a, phase b and phase c. A simulated current reference waveform using the HDL model of the control scheme is shown in Fig. 6.24.

Problems

1. Develop a FPGA-based three-phase sine wave generator, using the sinusoidal PWM shown in Fig. 6.25. An isosceles triangular carrier wave is compared with a fundamental frequency modulating wave. The points of intersection determine switching instants for the power devices.

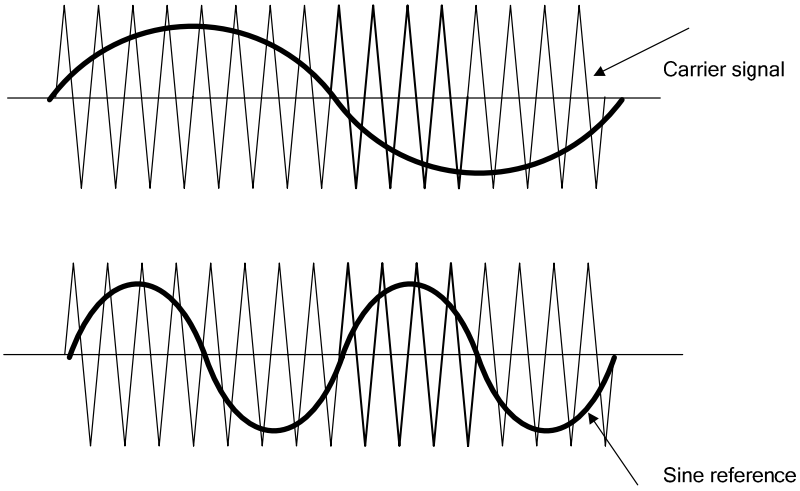
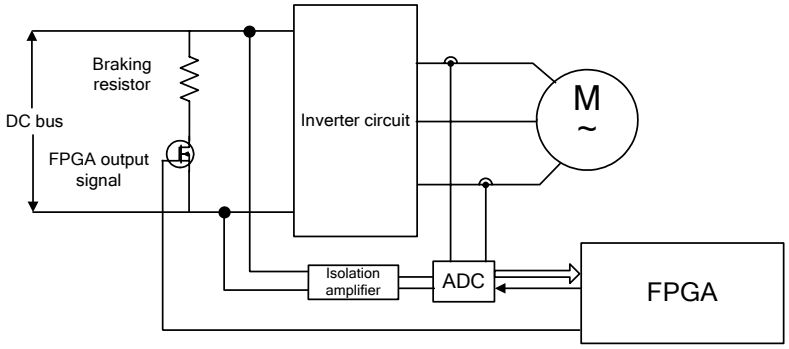
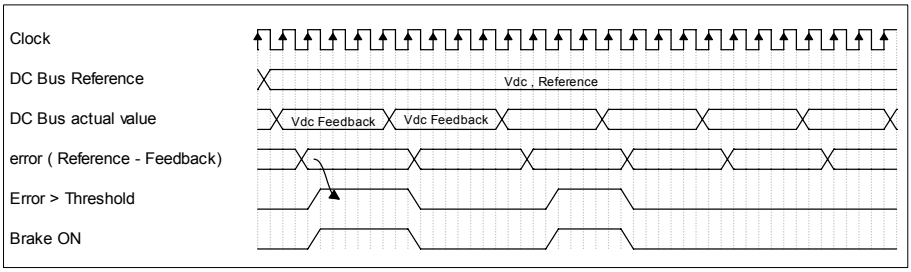


Fig. 6.25. Creating a sinusoidal PWM signal

2. Rapid deceleration of a motor drive feeds back voltage to the DC bus of the inverter circuit. A dynamic braking circuit compares the DC bus voltage against a preset reference and then turns on a switch through a resistor to dissipate the excess voltage. The power circuit and the logic diagram are shown in Fig. 6.26a and b. Write a Verilog code to sense the DC bus voltage and control the switch of the braking resistor.



a



b

Fig. 6.26. a Power circuit diagram for dynamic braking; **b** timing diagram for dynamic braking

Further Reading

1. Bose BK (1987) Introduction to microcomputer control. In: Bose BK (ed) Microcomputer Control of Power Electronics and Drives, IEEE Press, 3–22
2. Carrica D, Funes MA, Gonzalez SA (2003) Novel stepper motor controller based on FPGA hardware implementation. IEEE/ASME Transactions on Mechatronics, 8(1):120–124
3. Cirstea MN, Dinu A, Khor J, McCormick M (2002) Neural and fuzzy logic control of drives and power systems. Elsevier Science, Oxford
4. Hall TS, Hamblen JO (2004) System-on-a-programmable-chip development platforms in the classroom. IEEE Transactions on Education, 47(4):502–507
5. Hoang LH (1994) Microprocessors and digital IC's for Motion Control. Proceedings of the IEEE, 82(8):1140–1163
6. Jeon JK, Kim YK (2002) FPGA based acceleration and deceleration circuit for industrial robots and CNC machine tools. Mechatronics 12(4):635–642
7. Lygouras JN et al (1998) High-Performance Position Detection and Velocity Adaptive Measurement for Closed-Loop Position Control. IEEE transactions on instrumentation and measurement, 47(4):978–985

8. Pimentel J, Le-Huy H (2000) A VHDL library of IP cores for power drive and motion control applications. Paper presented at Canadian conference on electrical and computer Engineering, 1:184–188
9. Shireen W et al (2003) Controlling multiple motors utilizing a single DSP controller. IEEE Transactions on Power Electronics, 18(1):124–130
10. Takahashi TT, Goetz J (2004) Implementation of Complete AC Servo Control in a low cost FPGA and Subsequent ASSP Conversion. Paper presented at IEEE Applied Power Electronics Conference (APEC 2004), Anahiem California
11. Xilinx (2005) Getting started: FPGAs in motor control. Xilinx Application Note
12. Zeidman B (1999) Verilog Designer's Library. Prentice Hall 1999
13. Kung YS, Shu GS (2005) Development of a FPGA-based motion control IC for robot arm. Paper presented at IEEE ICIT 2005 conference, pp. 1397–1402
14. Kung YS, Shu GS (2005) Design and implementation of a control IC for vertical articulated robot arm using SOPC technology. Paper presented at IEEE Mechatronics ICM 2005conference, pp. 532–536
15. Kung YS, et al (2006) FPGA-Implementation of Inverse Kinematics and Servo Controller for Robot Manipulator. Paper presented at IEEE Robotics and Biomimetics, (ROBIO 2006) at Kunming China, December 2006

7

Prototyping Using FPGA

FPGA provides a platform for rapidly prototyping digital systems. High-volume digital systems are prototyped using FPGAs to avoid possible re-spins. FPGA-based prototyping boards that incorporate necessary interfaces and memory chips are used for this purpose. In this chapter we take a look at how a FPGA can be used to develop and debug the hypothetical robot controller discussed in the previous chapters.

7.1 Prototyping Using FPGAs

Prototyping a FPGA-based system involves creating a physical system that can be tested using the FPGA as a controller. Before a digital system can be used for prototyping, many functional checks need to be done. Just like other digital systems, physical prototyping in a FPGA is preceded by functional simulation and emulation. The book [1] on prototyping of digital systems using FPGA is a good reference.

Behavioural simulation tests for functional requirements. Functional requirements for a digital system are tested with the help of software test vectors. The microprocessor simulator gives insight into digital logic functional simulation. A microprocessor simulator provides data on internal registers of the processor while stepping through the code or at pre-determined break points. The simulation environment does not support the timing requirements of the digital system. Due to this, the simulator cannot provide for real delays. The speed of the simulator is tied to the microprocessor and the clock of the workstation on which the simulator is running. Emulation helps overcome the constraints of functional simulation. In the microprocessor world, the emulation environment allows access to the internal registers of the ISA. Dedicated hardware operates at the same clock frequency as that of the target chip and has equivalent ISA to test the system. This ensures creating real-time delays and offers a realistic estimate of speed. Simulation of digital design by including path delays and logic delays can be considered a component of emulation.

The last check for any digital design is to make it work in silicon. A hardware prototype tests the logic in silicon, and it also speeds up the process of verification. In most real-time systems, the clock needs to be scaled-down by a factor of 10,000. The clock requirement for stepper motor control described in Sect. 6.3.1 is of the order of kilohertz. Checking the stepper motor logic (see Fig. 6.7) with a system clock of megahertz takes lots of simulation cycles. Hardware test equipment such as a logic analyzer or oscilloscope can be used to quickly test for functionality. Table 7.1 describes simulation and physical verification on a FPGA-based system.

Table 7.1. Tools used for simulating and physically verifying FPGA-based digital designs

Simulation	Physical Verification
Behavioural simulation: Achieved using test benches and simulation software. The simulation model does not take into account element and interconnect delays of the circuit. Post Place and Route Simulation: A delay model is obtained after the design has been placed and routed. An estimate of the delay is available to determine the speed of the circuit.	• Viewing internal and external signals on an oscilloscope. The signals need to be brought out to the GPIO pins or to a DAC. • Multiple digital channels can be viewed using a logic analyzer • FPGA internal signals can be viewed using the ChipScope™ tool • A protocol analyzer for checking communication protocols

To check a digital design on a particular FPGA, an electronic ecosystem consisting of ADC, DAC, memory chips, physical interfaces and display and input devices is created on a board. This board is referred to by many vendors as the starter board[27]. For all preceding examples in this book, the designs were targeted at a Xilinx SPARTAN-3E™ FPGA. Figure 7.1 shows a board of a digital system built around a SPARTAN-3E™ FPGA.

To implement digital logic inside an FPGA, constraints are used to lock pins to particular signals within the HDL logic or to connect to chips on the prototyping board. The most frequently used constraints are shown in Table 7.2.

[27] A FPGA-based starter board consists of a FPGA surrounded by other peripheral devices. FPGA vendors provide such boards to prototype systems around the target FPGA device. The Xilinx SPARTAN-3E™ prototyping board consists of two RS232 serial ports, four DIP switches, four push buttons, 8 LEDs, a VGA port, a character LCD display, a PS/2 port, a push-button rotary encoder, a SPI analog to digital converter, a SPI digital to analog converter, a 10/100 Ethernet port, a 2-MB SPI flash, a 16-MB of parallel NOR flash and a 64-MB double data rate synchronous dynamic random access memory (DDR SDRAM).

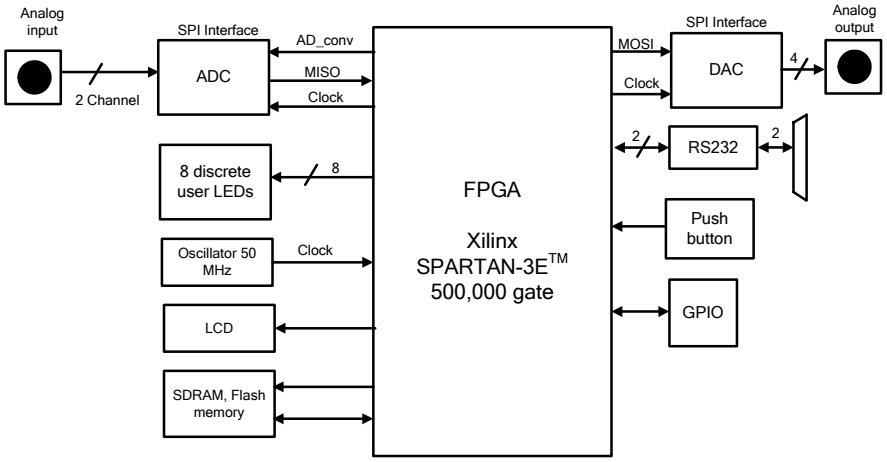


Fig. 7.1. Partial diagram of Xilinx SPARTAN-3E™ based starter kit [2]

Table 7.2. Summary of typical FPGA constraints during design implementation

Constraint	Use	Example
Pin	Lock I/O signal to FPGA pin	NET “name” LOC = "F9";
Area	Specify area on the floor-plan for design placement	AREA_GROUP “group_name” RANGE = SLICE_X6Y6: SLICE_X5Y5;
Global logic	Specify use of particular block RAM, multiplier or DCM	INST “mult_name” LOC = MULT18X18_X0Y0;

The pin locking constraint is used to connect the FPGA pin to the external device pin. A FPGA board consists of connections to switches, push buttons, a knob, LCD and to expandable connectors. The LED output pin constraint shown in Fig. 7.2 provides the option to define the slew rate (fast or slow) and the current drive capability in milliamperes.

```
NET "LED<7>" LOC = "F9"  | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LED<6>" LOC = "E9"  | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LED<5>" LOC = "D11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LED<4>" LOC = "C11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LED<3>" LOC = "F11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LED<2>" LOC = "E11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LED<1>" LOC = "E12" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LED<0>" LOC = "F12" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
```

Fig. 7.2. Pin constraints for the LEDs onboard. The DRIVE parameter specifies a current of 8 mA [2]

To place a digital design in a pre-defined area on the FPGA chip, an area constraint is used, where the row and column of the starting and ending slices are defined. The area start and end points are diagonally opposite.

```
INST design_d * LOC = SLICE_X1Y1: SLICE_X15Y5;
```

This places the logic in the area determined by slice (X1,Y1) and (X15,Y5). The constraint is implemented provided the design fits in the specified area.

A timing constraint is used to specify timing closure requirements. The constraints are defined for input pad to logic, logic to logic and logic to output pad.

7.2 Test Environment for the Robot Controller

For prototyping the robot controller, each individual block needs to be tested and then integrated together. The design is divided into functional units. In the preceding chapters, several components were discussed for the development of a robot control system. Table 7.3 lists interfaces needed for the robot controller test environment.

Table 7.3. Mapping the Robot controller to board resources for prototyping

Interface functionality needed	**Prototyping board support feature**
Start, stop switches, buttons, knob	GPIO pins
Joint motor mounted incremental encoder interface	GPIO pins
UART for serial communications interface with workstation	RS-232
External memory chip containing processor code for trajectory planning	DDR SDRAM, FLASH PROM
Display of robot co-ordinates	LCD screen
Programming of robot co-ordinates	RS-232 port
Motor current feedback	SPI-based ADC
Motor interface (control of power devices using GPIO)	GPIO
Motor interface testing using a low-pass filter	SPI-based ADC
Motor waveform testing using digital-to-analogue converter (DAC)	SPI-based DAC

The architecture of a robot controller prototyping board consists of the following components interconnected. Figure. 7.3 shows connections of the FPGA board-based robot controller to different sections of the robot.

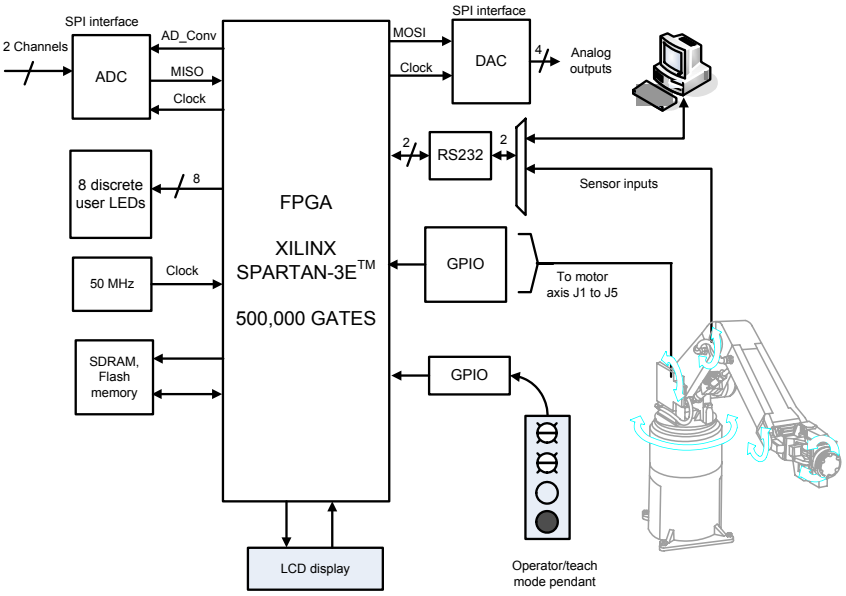


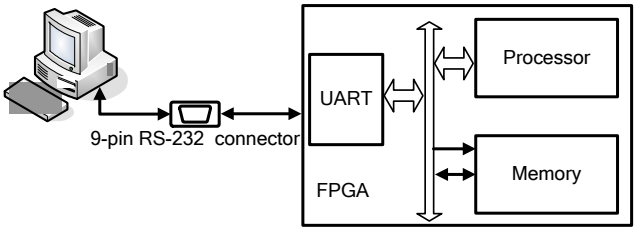
Fig. 7.3. Interfacing a FPGA-based robot controller with robot signals

7.3 FPGA Design Test Methodology

It is difficult to have a formal description of various debugging techniques used for physical verification of FPGA-based designs. Designers have their own test methodology. This section contains guidelines for testing FPGA-based hardware and software components of the robot controller. The processor-based environment discussed in Chap. 4 is tested either by using a debugger or by bringing out various parameters to a serial port. The hardware test environment can be tested using one or more tools. These include oscilloscopes, logic analyzers and FPGA-integrated logic analyzers.

7.3.1 UART for Software Testing

The UART-based serial protocol is a useful way of testing processor-based, high - level code. The **print** command of C is used to display intermediate values of software algorithms. From Chap. 4, a UART peripheral is configured for standard input and output (STDIO). To use the RS-232 functionality, a RS-232 voltage translator is connected to the FPGA I/O pin shown in Fig. 7.4a. The voltage translator takes input from the FPGA pin and converts logic 0 to +12 volts and logic 1 to −12 volts. The pin constraint for one of the RS-232 channels on the board is given in Fig. 7.4b.



a

```
NET "RS232_DTE_RXD" LOC = "U8" | IOSTANDARD = LVTTL ;
NET "RS232_DTE_TXD" LOC = "M13" | IOSTANDARD = LVTTL | DRIVE = 8 | SLEW = SLOW ;
```

b

Fig. 7.4. a Using the UART interface to debug processor code; **b** pin constraints for onboard RS-232 [2]

7.3.2 FPGA Hardware Testing Methodology

The HDL code for PWM control of the permanent magnet DC motor in Fig. 6.10, is used as an example to demonstrate the use of the internal logic analyzer. The logic analyzer allows for simultaneous viewing of multiple signals on the chip. Figure 7.5 shows the configuration of ChipScope[28] software, where control voltage **V**c, counter signals, clock and PWM are selected for viewing. When triggered, the on-chip logic analyzer stores a pre-defined number of sample values in the block RAM. The number of samples that can be recorded by the logic analyzer is limited by the amount of block RAM available on the FPGA chip.

[28] Many times it is not feasible to bring out design registers and nets for debugging. Chipscope™ from Xilinx aids in capturing values of registers and wires from the FPGA device itself.

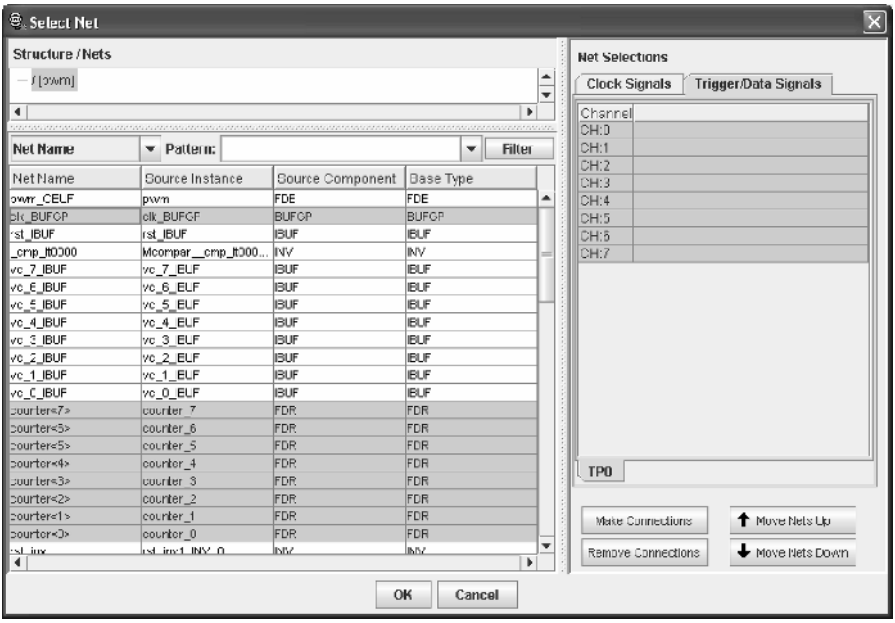
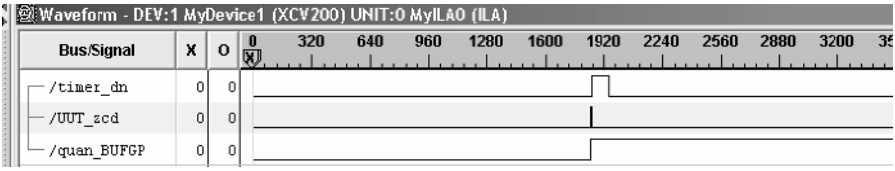
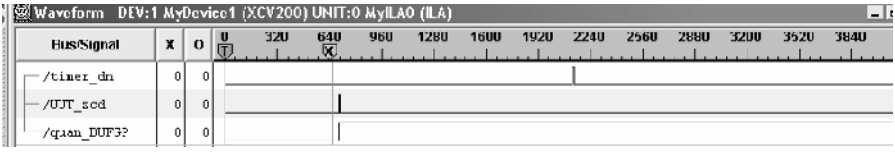


Fig 7.5. Configuring the integrated logic analyzer (ILA) channels of a PMDC motor PWM. (Refer to code of Fig. 6.10)

The use of ChipScope^{TM} is shown for testing the working of a timer circuit of a single-phase controlled rectifier. The quantizer, zero crossing and timer done bit are shown in Fig. 7.6a and b. Different timer values are used for illustration.



a



b

Fig. 7.6. a Timer loaded for zero time delay. The timer done signal appears immediately; **b** timer loaded with a small number. The timer done signal is seen to appear after the defined time interval

7.3.2.1 Viewing Real-time Signals on an Oscilloscope

The oscilloscope allows viewing real-time digital and analogue signals. The FPGA setup shown in Fig. 7.7 is used to bring out signals to the oscilloscope. The working of the dead time delay code from Fig. 6.13 is tested by bringing gate signals of upper and lower device to an oscilloscope. In a similar fashion the zero crossing code of Fig. 3.39 is tested with an oscilloscope. The oscilloscope waveforms for dead time logic and zero crossing are shown in Figs. 7.8 and 7.9.

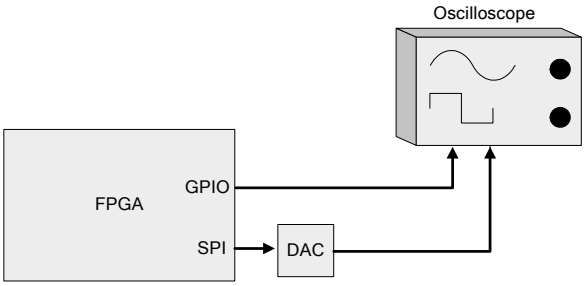


Fig. 7.7. Using the DAC to view an internal signal on an oscilloscope

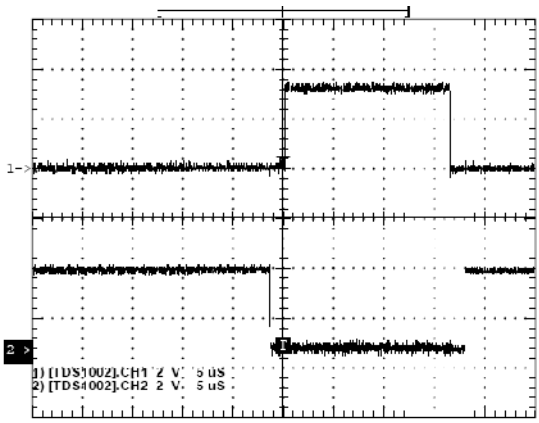


Fig. 7.8. Dead time delay of 1.5 μs between upper and lower switch of an inverter bridge. (Refer to code from Fig. 6.13).

Analogue representation of vectored digital values is done by bringing out the signal to a DAC. Table 7.4 lists the signals for connecting a DAC to a FPGA using the SPI port. Testing of time varying signals such as sine is accomplished in this manner. Figure 7.10 shows the diagram of a sine wave generator, as discussed in Sect. 6.3.4.

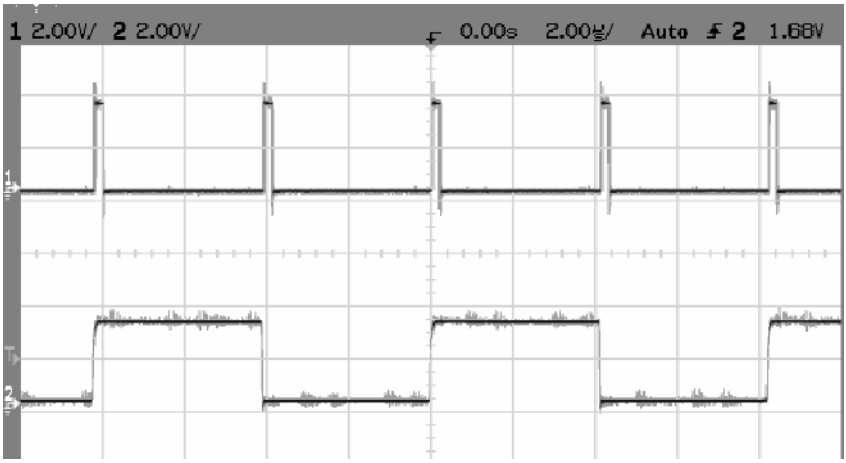


Fig. 7.9. Generation of zero crossing pulses using sample quantizer waveform. (Refer to code from Fig. 3.29).

Table 7.4. Connection details of DAC to FPGA SPI port

Signal	Direction	Description
SPI_MOSI	FPGA to DAC	Serial data (to be converted to analogue) from master to slave
DAC_CS	FPGA to DAC	Chip select
SPI_SCK	FPGA to DAC	Clock
SPI_MISO	DAC to FPGA	Serial data from slave to master

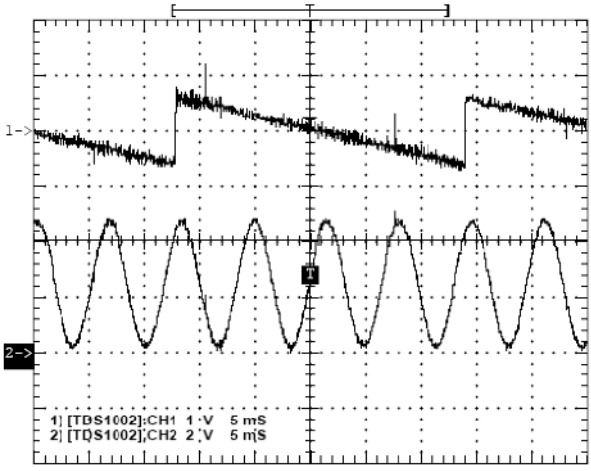


Fig. 7.10. Generation of sine wave. The top wave is the counter for angle theta and the bottom waveform is the generated sine wave. Both signals use a DAC for viewing

7.3.2.2 Quadrature Encoder Feedback

Position determination. The incremental encoder is used to determine the position of the robot axis. A variable frequency square wave function generator can be used to test the working of this module. A code similar to that given in Fig. 2.17 is used to measure transitions in the input wave and accordingly.

Determination of speed. The physical equation for determining speed divides the distance (in our case the position) covered by the time taken. The working of the HDL speed estimation code in FPGA is ascertained by dividing the value in the position register by a known time base. Figure 7.11 shows the block diagram of logic to determine speed by using position data. On-chip verification (Fig. 7.12) for determining the frequency of a 123-kHz pulse train is done using ChipScope™.

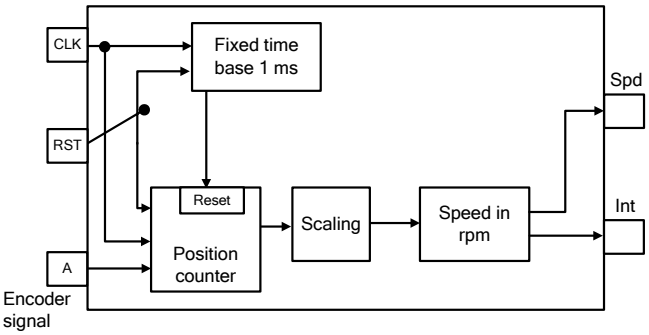


Fig. 7.11. Speed estimation logic. A fixed time base of 1 ms is used for determining speed

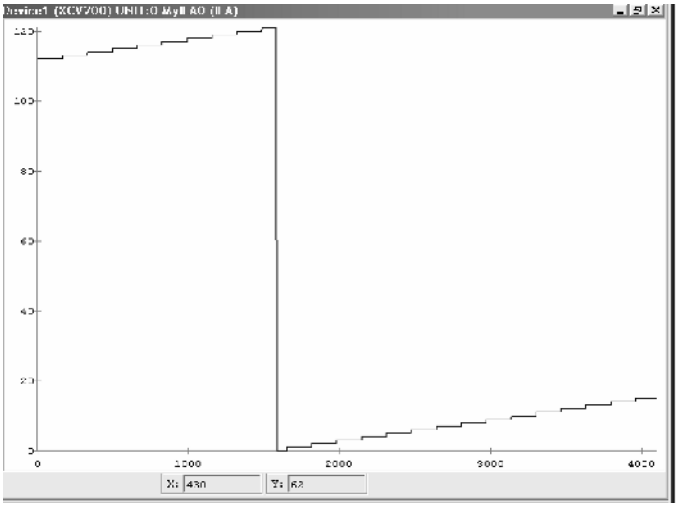


Fig. 7.12. Physical verification of speed estimation using ChipScope™. The value of the position counter is 123 in 1 ms time interval. The estimated encoder frequency is 123 kHz

7.3.2.3 Signal Conditioning

To monitor motor current, the current signal measured by a current transducer is fed to the FPGA by an ADC. The motor current is alternating between positive and negative rated values, so the corresponding voltage from the current transducer is bipolar. If a unipolar ADC is used on the FPGA board, an offset voltage equal to the maximum negative voltage is added. Figure 7.13a and b show conversion of a bipolar current input to unipolar for the correct interface to the ADC.

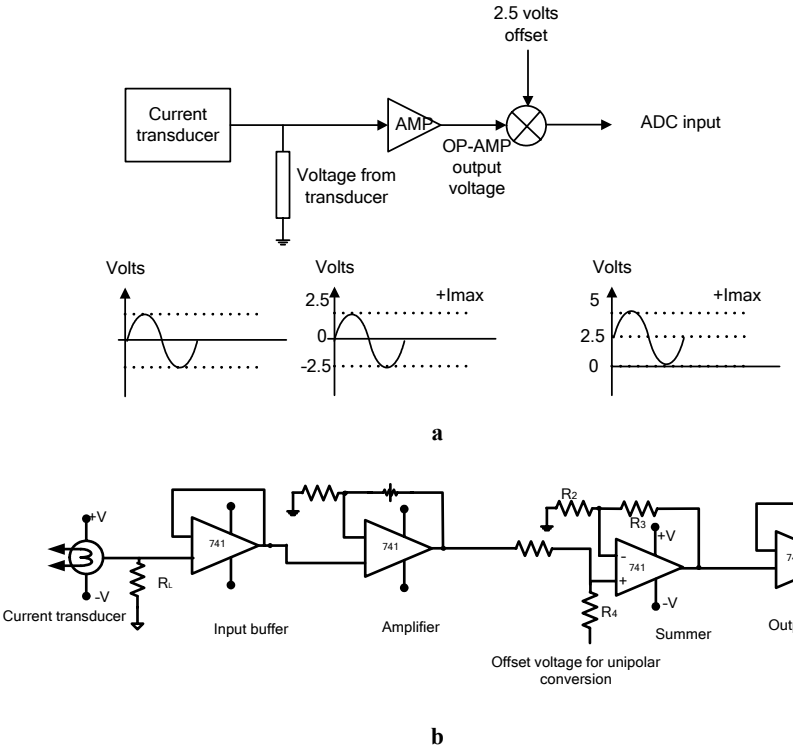


Fig. 7.13. a Interfacing a motor current signal to a unipolar ADC **b** Op-amp based circuit for getting unipolar current signal

7.3.2.4 Power Device Interface

Isolation between the FPGA-based control circuit and the power circuit is achieved by using an opto-coupler. As shown in Fig. 7.14, a logic one on the FPGA GPIO pin turns on the infrared LED of the opto-coupler through transistor T1 and resistor R1. The opto-coupler photo-transistor conducts and cuts off transistor T2. At this point, a voltage of 12 volts (determined by the 12-V Zener diode) is available for driving the MOSFET gate circuit. When the FPGA output signal goes low, the opto-coupler photo-transistor is cut off. Transistor T2 gets a base voltage through resistor R2 and conducts through resistor R3. When this happens, there is no voltage at the MOSFET gate.

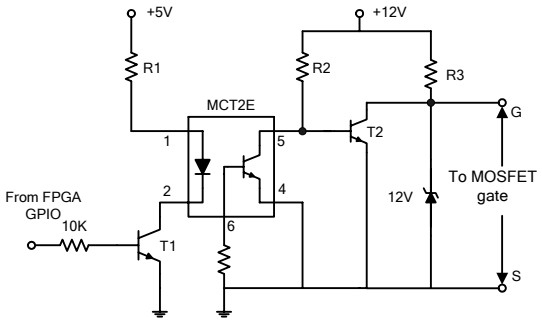


Fig. 7.14. Interfacing a FPGA GPIO and a MOSFET gate through an opto-coupler

For the three-phase bridge circuit as shown in Fig. 7.15, each of the six power devices requires a separate driver circuit.

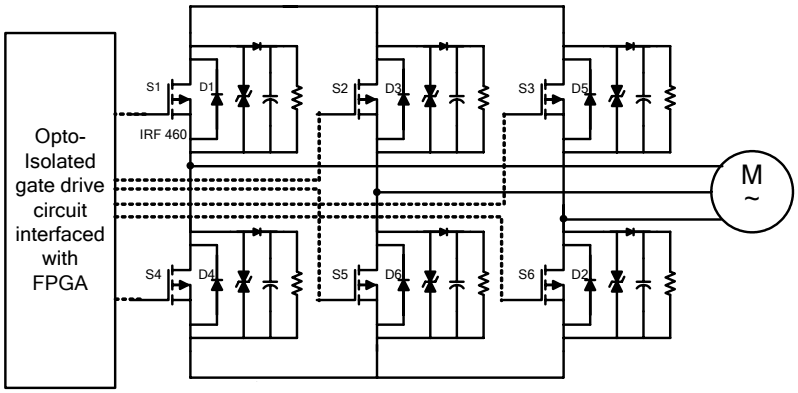
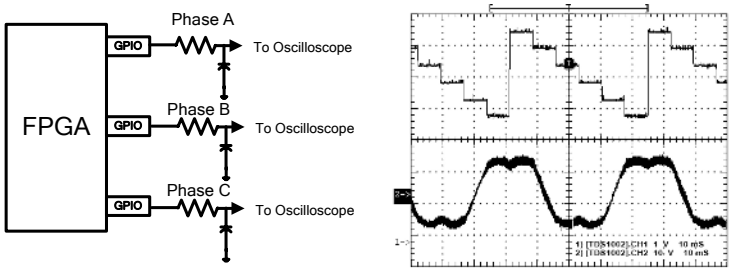


Fig. 7.15. Interfacing FPGA GPIO pins to six MOSFET devices of a three-phase bridge

A high-pass RC filter circuit shown in Fig. 7.16a is used to view SVPWM voltage.The output of the filter can be viewed on an oscilloscope (see Fig. 7.16b).



a **b**

Fig. 7.16. a A high-pass filter circuit to view a SVPWM waveform; **b** the resultant filtered waveform for one phase viewed on an oscilloscope. Details of SVPWM are discussed in Sect. 6.3.4.

Hardware-in-the-loop Testing

Hardware-in-the-loop allows testing FPGA-based signal processing algorithms. It uses o MATLAB® Simulink® environment and Xilinx System Generator™ tool to provide test vectors. These test vectors are routed through a JTAG port on the FPGA-based hardware (see Fig. 7.17). The output from the FPGA is again brought back to the same screen using the JTAG connection. This allows for cross-verification of the simulation and the physical test results for an algorithm.

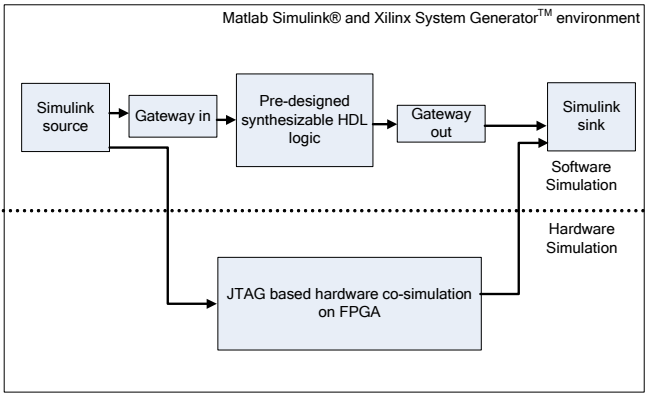


Fig 7.17. Hardware-in-the-loop test methodology [3]

Problems

1. Use the digital block diagram shown in Fig. 7.11 to estimate a range of frequencies from a function generator. Comment on the limitations of using a fixed time base.
2. Figure 7.18 is used for rotor position initialization of a permanent magnet motor. Before the motor is started, the stator windings are given rated current, so that the rotor can lock to a known position. Write logic for a current regulator to measure current using a current sensor to limit the switching of devices 1, 5 and 6 to rated value.

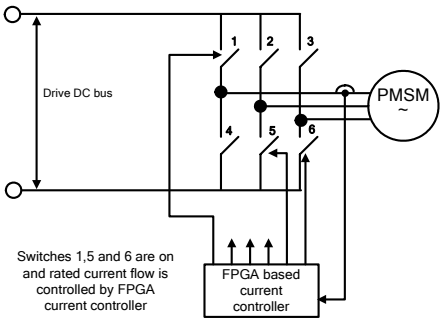


Fig. 7.18. PMSM motor rotor initialization

References

1. Hamblen JO, Hall TS, Furman MD (2006) Rapid prototyping of digital systems. Springer 2006
2. Xilinx (2006) Spartan-3E Starter Kit Board User Guide. UG230 (v1.0) March 2006
3. Xilinx (2006) System Generator for DSP performing Hardware-in-the-loop with the SPARTAN-3E Starter Kit, December 2006

Index