NASA HANDBOOK

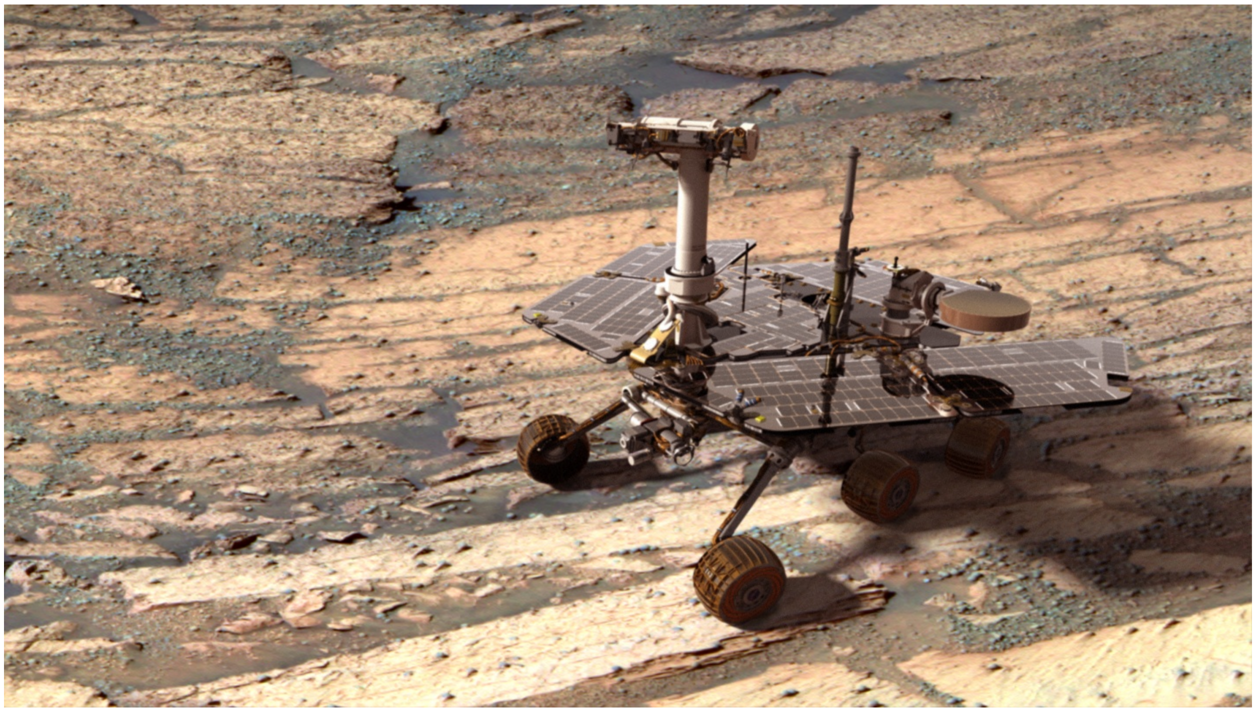
NASA-HDBK 8739.23
Baseline with Change 1

National Aeronautics and Space Administration
Washington, DC 20546

Baseline approved: 2011-02-16
Change 1 approved: 2011-03-29

NASA COMPLEX ELECTRONICS HANDBOOK FOR ASSURANCE PROFESSIONALS

**Measurement System Identification:
Metric**



Mars Exploration Rover (2003)

DOCUMENT HISTORY LOG

Status	Document Revision	Approval Date	Description
Baseline		2011-02-16	Initial Release *(JWL4)*
	Change 1	2011-03-29	Editorial correction to page 2 figure caption *(JWL4)*

This document is subject to reviews per Office of Management and Budget Circular A-119, Federal Participation in the Development and Use of Voluntary Standards (02/10/1998) and NPR 7120.4, NASA Engineering and Program/Project Management Policy.

This page intentionally left blank.

FOREWORD

This NASA Handbook (NASA-HDBK) is approved for use by NASA Headquarters and NASA Centers, including Component Facilities. This NASA-HDBK may be applied on contracts per contractual documentation as a reference or training publication.

Comments and questions concerning the contents of this publication should be referred to the National Aeronautics and Space Administration, Director, Safety and Assurance Requirements Division, Office of Safety and Mission Assurance, Washington, DC 20546.

Requests for information, corrections, or additions to this NASA-HDBK shall be submitted via "Feedback" in the NASA Technical Standards System at http://standards.nasa.gov or to National Aeronautics and Space Administration, Director, Safety and Assurance Requirements Division, Office of Safety and Mission Assurance, Washington, DC 20546.

s/ Bryan O'Connor

Bryan O'Connor
Chief, Safety and Mission Assurance

February 16, 2011

Approval Date

The Office of Safety and Mission Assurance would like to recognize Kalynnda Berens and Richard Plastow for their work in authoring this publication.

This page intentionally left blank.

TABLE OF CONTENTS

CHAPTER 1. Overview 11
1.1 Purpose 11
1.2 Scope 11
1.3 Anticipated Audience 12
1.4 Handbook Layout 12

CHAPTER 2. Reference Documents and Links 13
2.1 Reference Documents 13
2.2 Links 14

CHAPTER 3. Definitions and Acronyms 15
3.1 Definitions 15
3.2 Acronyms 20

CHAPTER 4. Complex Electronics Overview 23
4.1 Blurring the Hardware/Software Line 23
4.2 Programmable versus Designable Devices 25
4.3 Simple Programmable Logic Devices 27
4.4 Complex Programmable Logic Devices (CPLD) 28
4.5 Field Programmable Gate Array (FPGA) 30
4.6 Application Specific Integrated Circuit (ASIC) 30
4.7 System-on-Chip (SoC) 30
4.8 Concerns and Issues 31
4.9 Summary 32

CHAPTER 5. Design Process 33
5.1 Overview of the Complex Electronics Design Process 33
5.2 Requirements and Specifications 36
5.3 Design Entry 37
5.4 Abstraction 40
5.5 Hardware Description Languages 42
5.6 Programming Example 48
5.7 Synthesis 48
5.8 Implementation 53
5.9 Verification 58

CHAPTER 6. Process Assurance **64**
6.1 Process Assurance Overview 64
6.2 Identifying Complex Electronics 67
6.3 Process Assurance Activities 69

CHAPTER 7. Future Trends **79**
7.1 Changes in Complex Electronics Design and Verification 79
7.2 Into the Not so Distant Future 81
7.3 NASA Assurance Changes 83

APPENDIX A Examples **85**
A.1 CPLD 85
A.2 FPGA 89
A.3 ASIC 95
A.4 SoC 101
A.5 Reconfigurable Computing 107

APPENDIX B Coding Style Guidelines **112**
B.1 Introduction 112
B.2 Top-Down Design 112
B.3 Signals and Variables 119
B.4 Packages 122
B.5 Technology-Specific Code (Xilinx) 126
B.6 Coding for Synthesis 135

LIST OF TABLES

Table 1: Complex Electronics Examples 26
Table 2: Simple PLD Comparisons 28
Table 3: CE vs. SW Development Phases 35
Table 4: VHDL vs. Verilog 46
Table 5: FPGA vs. ASIC Comparison 57
Table 6: Requirement Verification Activities 59
Table 7: Design Entry Verification Activities 60
Table 8: Design Synthesis Verification Activities 60
Table 9: Implementation Verification Activities 61
Table 10: Testing Verification Activities 62
Table 11: Simple Complexity Guidelines 68

LIST OF FIGURES

Figure 1: How Complex Electronics Compares 23
Figure 2: Example of PAL Structure 27
Figure 3: CPLD vs. FPGA Layout 29
Figure 4: SoC Example Configuration 31
Figure 5: CE vs. SW Life Cycles 33
Figure 6: Example CE Waterfall Development 34
Figure 7: Complex Electronics Design Views 39
Figure 8: Complex Electronics Domains 41
Figure 9: Warning Buzzer Example 41
Figure 10: General HDL Development 45
Figure 11: AOI VHDL to Verilog Comparison 51
Figure 12: External Programmer 56
Figure A-1 CPLD 85
Figure A-2 FGPA 89
Figure A-3 ASIC 95
Figure A-4 ASIC Die 96
Figure A-6 Reconfigurable SoC 103
Figure A-7 ChipSat OBC 106
Figure B-1 Behavioral Code 112
Figure B-2 Structural Code 113
Figure B-3 One Line Per Signal/Named Association 114
Figure B-4 Header Template 115
Figure B-5 Process, Function, and Procedure Header 115
Figure B-6 Inline Comments 115
Figure B-7 Proper Indentation 116
Figure B-8 Confusing _in and _out suffixes 118
Figure B-9 Internal Signals Representing Output Ports 118
Figure B-10 Correct Use of Variables 121
Figure B-11 Incorrect Use of Variables 122
Figure B-12 A Constant Guiding the Generation of Logic 123
Figure B-13 Address Width Defined by a Constant 124
Figure B-14 Modular Function Use 125
Figure B-15 Use of Types and Aliases 126
Figure B-16 Three-state Implementation of 4:1 Multiplexer 129
Figure B-17 Xilinx LUT-RAM Inference 130
Figure B-18 LUT-ROM Inference 130
Figure B-19 Virtex Block RAM inference 131
Figure B-20 Clock Enable Inference 133
Figure B-21 Inference of Xilinx Shift Register LUT (SRL) 134
Figure B-22 Local Asynchronous Reset and TC & Well-Behaved Synchronous Reset & CE . 136
Figure B-23 Well Behaved Local Asynchronous Reset and TC & Well-Behaved Synchronous Reset & CE 136
Figure B-24 Priority Encoded Logic 141
Figure B-25 For-Loop Cascaded Logic Implementation 142

Figure B-26 Latch Inference 143
Figure B-27 Elimination of Inadvertent Latch Inference.. 143

NASA COMPLEX ELECTRONICS HANDBOOK FOR ASSURANCE PROFESSIONALS

CHAPTER 1. OVERVIEW

1.1 PURPOSE

Complex electronics (CE) encompasses programmable and designable complex integrated circuits. "Programmable" logic devices (PLDs) can be programmed by the user and range from simple chips to complex devices capable of being programmed on-the-fly. Some types of programmable devices this handbook will address are:

- Field Programmable Gate Array (FPGA)
- Complex Programmable Logic Device (CPLD)
- Application-Specific Integrated Circuit (ASIC)
- System-on-chip (SoC)

"Designable" logic devices are integrated circuits that can be designed but not programmed by the user. The design is submitted to a manufacturer for implementation in the device. ASICs are an example of a designable device.

Development of assurance methodologies for complex electronics is lagging behind the pace of the technology. Complex electronics are commonly used within NASA systems, sometimes in safety-critical systems. Both software assurance and quality assurance engineers need to understand what these devices are, where they are used, and how they are designed. However, the development of assurance activities for complex electronics is lagging behind the pace of the technology. This handbook provides some general suggestions that, if applied, may increase confidence in the quality of complex electronic devices.

1.2 SCOPE

This Handbook will provide an overview of complex electronics, the design process, and assurance activities. It discusses:

- Which devices are "complex electronics," and which are not.
- What each device is and examples of use on NASA projects.
- How electronics engineers design and program the devices.
- What assurance and verification activities can be used for complex electronics.
- Future trends in the design and assurance of complex electronics.

Additional assurance activities for complex electronics devices may be required in the future. While this handbook will not prepare you to perform those activities, it will provide you with a general understanding of the devices and the design and assurance activities. You will be able to "speak the language" when communicating with the hardware design engineers.

1.3 ANTICIPATED AUDIENCE

1.3.1 This handbook is primarily intended for software assurance and quality assurance engineers who do not have significant experience with complex electronics. You do not need a hardware background to understand the material in this handbook. However, being familiar with embedded systems or flight hardware may help you understand some of the concepts.

System safety personnel are encouraged to review this handbook. Modern technology, especially electronics, is changing at a rapid pace. Projects and systems you support will be using these devices in the near future, if they are not already doing so.

Software and electronic engineers are encouraged to review this handbook. An understanding of the assurance activities and concepts discussed in the handbook may be helpful to you in supporting projects and systems.

1.4 HANDBOOK LAYOUT

Chapter 1 provides the purpose, scope, and layout for the handbook.

Chapter 2 provides a list of reference documents and useful links.

Chapter 3 provides definitions and acronyms used in this handbook.

Chapter 4 gives an overview of complex electronics, describes why assurance engineers need to be aware of complex electronics and details some concerns and issues with the current state of assurance activities.

Chapter 5 describes the design process for complex electronics. A short explanation of hardware description languages, along with a simple example, is included.

Chapter 6 provides an overview of current and suggested assurance practices for complex electronics. This section also contains an overview of process assurance.

Chapter 7 discusses some future trends in design and assurance of complex electronics.

Appendix A describes each of the types of complex electronics in detail.

Appendix B contains the Hardware Description Language Coding Standard from Xilinx.

CHAPTER 2. REFERENCE DOCUMENTS AND LINKS

2.1 REFERENCE DOCUMENTS

The documents listed in this chapter provide additional information supporting this NASA-HDBK. The latest issuance of cited documents should be used unless otherwise stated in this NASA-HDBK. The applicable documents are accessible via the NASA Online Directives Information System at http://nodis3.gsfc.nasa.gov/ or directly from the Standards Developing Organizations (SDO) or other document distributors.

2.1.1 GOVERNMENT DOCUMENTS:

NASA Documents:

NPR 7150.2	NASA Software Engineering Requirements
NASA-STD 2201-91	NASA Software Configuration Management Guidebook (http://satc.gsfc.nasa.gov/GuideBooks/cmpub.html)
NASA-STD 8709.22	Safety and Mission Assurance Acronyms, Abbreviations, and Definitions
NASA-STD 8719.13	Software Safety Standard
NASA-STD 8739.8	Software Assurance Standard
NASA-GB 8719.13	NASA Software Safety Guidebook (http://www.hq.nasa.gov/office/codeq/doctree/871913.pdf)
NASA-GB A201	Software Assurance Guidebook (http://satc.gsfc.nasa.gov/assure/agb.txt)
NASA-GB A301	Software Quality Assurance Audits Guidebook (http://satc.gsfc.nasa.gov/audit/audgb.txt)

Other Government Documents:

DO-254	Design Assurance Guidance for Airborne Electronic Hardware (Federal Aviation Administration (FAA))
MIL-STD 882D	Standard Practice for System Safety (Department of Defense (DoD))

2.1.2 INTERNATIONAL CONSENSUS STANDARDS:

Institute of Electrical and Electronics Engineers (IEEE)

IEEE 830-1998	IEEE Recommended Practice for Software Requirements Specifications
IEEE 1194.1-1990	IEEE Standard Test Access Port and Boundary Scan Architecture
IEEE 1228-1994	IEEE Standard for Software Safety Plans

2.2 LINKS

NASA-related Links

URL	Description
http://nepp.nasa.gov/index.cfm	NASA Electronic Parts and Packaging Program

Other Links

URL	Description	FPGA	Reconfig. Comput.	VHDL	Verilog	Other
http://klabs.org/richcontent/Tutorial/tutorial.htm	Tutorials Information Links	x	x			x
http://www.icd.com.au/vhdl.html	Tutorials	x	x	x		
http://www.epanorama.net/links/fpga.html	Information	x		x		
http://www.verilogtutorial.info/	Tutorial				x	
http://www.asic-world.com/verilog/veritut.html	Tutorial				x	
http://www.gmvhdl.com/VHDL.html	Tutorial			x		
http://www.doulos.com/knowhow/	Tutorial Information			x	x	x
http://www.cs.ucr.edu/content/esd/labs/tutorial/	Information			x		
http://instruct1.cit.cornell.edu/courses/ee475/tutorial/VHDLTut.htm	Tutorial Links			x		
http://www.systemc.org/	Information					x
http://www.acc-eda.com/vhdlref/refguide/vhdl_examples_gallery/vhdl_examples_gallery.htm	Examples			x		
http://www.vhdl.org/	Information			x	x	x
http://www.vhdl.org/vhdlsynth/vhdlexamples/	Examples			x		
http://www.acc-eda.com/vhdlref/refguide/toclist.htm	Information			x		
http://www.mrc.uidaho.edu/fpga/index.php	Information	x				
http://www.fpga4fun.com/	Information	x				
http://equipe.nce.ufrj.br/gabriel/vhdlfpga.html	Links	x		x		
http://www.fuse-network.com/fuse/training/index.html	Training Material	x				x
http://www.radio-electronics.com/info/data/semicond/asic/asic.php	ASIC on-line book					x
http://www.netrino.com/Articles/RCPrimer/	Tutorial		x			
http://www.cotsjournalonline.com/	Journal					x
http://www.fpgajournal.com/	Journal	x				

CHAPTER 3. DEFINITIONS AND ACRONYMS

3.1 DEFINITIONS

Note: Definitions for safety and mission assuranceterms are found in NASA-STD 8709.22, Safety and Mission Assurance Acronyms, Abbreviations, and Definitions. Terms unique to this NASA-Handbook are listed below.

Adequate: When referring to fire protection or life safety, the safeguards necessary to provide facilities and their occupants with protection against all known or recognized hazards.

Antifuse: An electrical device that performs the opposite function as a fuse. Antifuses are widely used to permanently program integrated circuits (ICs) by creating an electrical connection.

Application Specific Integrated Circuit (ASIC): Integrated circuit product customized for a single application.

Architecture: The common logic structure of a family of programmable integrated circuits. The same architecture may be realized in different manufacturing processes.

Asynchronous: A signal whose data is acknowledged or acted upon immediately, irrespective of any clock signal.

Boundary scan: Boundary scan is a methodology allowing complete controllability and observability of the boundary pins of a JTAG (Joint Test Action Group)-compatible device via software control. This capability enables in-circuit testing without the need of in-circuit test equipment.

Cell Library: The collective name for the set of logic functions defined by the manufacturer of an ASIC. The designer decides which types of cells should be realized and connected together to make the device perform its desired function.

Chip: Another name for an integrated circuit.

Codec: Short for compressor/decompressor or coder/decoder, a codec is any technology for compressing and decompressing data. Codecs can be implemented in software, hardware, or a combination of both. Some popular codecs for computer video include MPEG (Moving Picture Experts Group), Indeo, and Cinepak.

Combinatorial: A digital function whose output value is directly related to the current combination of values on its inputs. Also known as combinational.

Comparator (digital): A logic function that compares two binary values and outputs the results in terms of binary signals representing less-than and/or equal-to and/or greater-than.

Complex Programmable Logic Device (CPLD): Programmable logic devices characterized by an architecture offering high speed, predictable timing, and simple software.

Configurable/Complex Logic Block (CLB): The array of multi-input and multi-output logic cells to be programmed. CLB is a configurable logic block that consists mainly of Look-up Tables (LUTs) and flip-flops.

Cores: In the semiconductor design industry, refers to predefined functions such as processors or bus interfaces that are typically licensed from the software developer. Cores can be implemented directly in silicon, either in fixed logic or programmable logic devices, and save chip designers time during product development. Synonymous with Intellectual Property.

Die: An unpackaged integrated circuit. The plural of "die" is also "die".

Digital Signal: A digital signal is a signal whose key characteristic (e.g., voltage or current) falling into discrete ranges of values. Most digital systems utilize two voltage levels (low and high values).

Digital Signal Processor (DSP): A specialized central processing unit (CPU) used for digital signal processing of signals such as sound, video, and other analog signals which have been converted to digital form. Some uses of DSP are to decode modulated signals from modems; to process sound, video, and images in various ways; and to understand data from sonar, radar, and seismological readings.

Electrically-Erasable Programmable Read-Only Memory (EEPROM): A memory device whose contents can be electrically programmed by the designer. Additionally, the contents can be electrically erased allowing the device to be reprogrammed.

Electro-Static Discharge (ESD): The term electro-static discharge refers to a charged person, or object, discharging static electricity. Although the current associated with such a static charge is low, the electric potential can be in the millions of volts and can severely damage electronic components.

Erasable Programmable Read-Only Memory (EPROM): A memory device whose contents can be electrically programmed by the designer. Additionally, the contents can be erased by exposing the die to ultraviolet light through a quartz window mounted in the top of the component's package.

Falling-Edge: A transition from a logic 1 to a logic 0. Also known as a negative edge.

Field Programmable Gate Array (FPGA): High density PLD containing small logic cells interconnected through a distributed array of programmable switches. This type of architecture produces statistically varying results in performance and functional capacity, but offers high register counts. Programmability typically is via volatile SRAM (Static Random Access Memory) or one-time-programmable antifuses.

Firmware: The combination of a hardware device and computer instructions and/or computer data that reside as read-only software on the hardware device.

First-in first-out (FIFO): Data structure or hardware buffer where items come out in the same order they came in.

Flash memory: Non-volatile storage device similar to EEPROM, but where erasing can only be done in blocks or the entire chip.

Flip-flop: A digital logic circuit that can be switched back and forth between two states.

Floorplanning: The process of identifying structures that should be placed close together on a chip, and allocating space for them.

Fuse: An electrical device that performs the same function as a fuse. Fuses are widely used to permanently program integrated circuits by opening an electrical connection.

Gate: In electronic circuitry, a pathway that may be open or closed, depending on the source of the input, the strength of a signal, or the conductivity of chemicals used in semiconductors. Logic gates are programmed to correspond to related "if-then" statements. The state of an open or closed gate is analogous to the binary state of a 0 or a 1. The application of this analogy allows computing machinery with millions of gates to respond conditionally and to perform logical functions.

Gate Array: Integrated circuit that is customized by interconnecting an array of logic elements. Customization is performed by the manufacturer and typically involves non-recurring engineering costs and several design iterations.

Glue: Generic term for any interface logic or protocol that connects two component blocks. Hardware designers call anything used to connect large VLSIs or circuit blocks "glue logic."

Hardware Description Language (HDL): A kind of language used for the conceptual design of integrated circuits. Examples are VHDL and Verilog.

Integrated Circuit (IC): A device in which components such as resistors, capacitors, diodes, and transistors are formed on the surface of a single piece of semiconductor.

In-Circuit Reconfigurable (ICR): An SRAM-based or similar component which can be dynamically reprogrammed on-the-fly while remaining resident in the system.

In-System Programmable (ISP): An EEPROM-based, flash-based, or similar component which can be reprogrammed while remaining resident on the circuit board.

JHDL: A structurally based hardware description language implemented with the Java programming language. JHDL is a method of describing (programmatically, in Java) the components and connections in a digital logic circuit. More specifically, JHDL provides object classes used to build up circuit structure.

Joint Electronic Device Engineering Council (JEDEC): A council which creates, approves, arbitrates, and oversees industry standards for electronic devices. In programmable logic, the term JEDEC refers to a textual file containing information used to program a device. The file format is a JEDEC approved standard and is commonly referred to as a JEDEC file.

Joint Test Action Group (JTAG): (or "IEEE Standard 1149.1"). A standard specifying how to control and monitor the pins of compliant devices on a printed circuit board. JTAG is a standard interface used for in-system testing and debugging.

Logic: One of the three major classes of integrated circuits in most digital electronic systems. The other two major classes are microprocessors and memory. Logic is used for data manipulation and control functions that require higher speed than a microprocessor can provide.

Logic Function: A mathematical function that performs a digital operation on digital data and returns a digital value.

Logic Gate: The physical implementation of a logic function.

Logic Synthesis: A process in which a program is used to optimize the logic used to implement a design.

Look-Up Table (LUT): An array or matrix of values that contains data that is searched. An alternative implementation of a CLB; the multiple inputs generate the complex outputs.

Macrocell: A macrocell on most modern CPLDs contains a sum-of-products combinatorial logic function and an optional flip-flop. The combinatorial logic function typically supports four to sixteen product terms with wide fan-in. Thus, a macrocell may have many inputs, but the logic function complexity is limited. On the other hand, most FPGA logic blocks have unlimited complexity, but the logic function only has four inputs.

Mapping: The process of taking the logic blocks and determining what logic gates and interconnections on the device should be used to implement those blocks.

Netlist: A list of names of symbols or parts and their connection points, which are logically connected in each net of a circuit. A file listing parameters extracted from a circuit schematic.

Noise: The miscellaneous rubbish that gets added to a signal on its journey through a circuit. Noise can be caused by capacitive or inductive coupling, or from externally generated interference.

Non-volatile: The ability of a memory element to keep its contents when power is removed from the device.

Onboard: Contained on the device or on the board.

One Time Programmable: This device can be programmed only once; its contents cannot be changed. While typically these devices are fuse or antifuse based, they can also be low-cost EPROM devices. In this case, typically used for production devices, an inexpensive package is used without a window.

Partial Reprogrammability: The ability to leave some internal logic in place and change another part of the FPGA logic.

Pinout: A diagram that indicates how wires are terminated to pins in a connector; a list that assigns device functions to specific pins.

Place and Route: Converts the results of the synthesis process to the format supported and takes the logic blocks and determines what logic gates and interconnections on the device should be used to implement those blocks.

Programmable Logic: A logic element whose function is not restricted to a particular function. It may be programmed at different points of the life cycle. At the earliest, it is programmed by the semiconductor vendor (standard cell, gate array), by the designer prior to assembly, or by the user, in circuit.

Programmable Logic Controller (PLC): A control device, usually used in industrial control applications, that employs the hardware architecture of a computer and relay ladder diagram language. Inputs to PLC's can originate from many sources including sensors and the outputs of other logic devices. Also called "programmable controller."

Reconfigurable Computing: A methodology of using programmable logic devices in a system design such that the hardware-based logic can be changed to perform various tasks. Benefits include the use of fewer components, less power, and flexibility. Also allows networked equipment in the field to be upgraded or repaired remotely.

Reprogrammable: These devices can have their configuration loaded more than once. SRAM-based devices may be reloaded without restriction. Many other forms of reprogrammable elements have restrictions on the number of write cycles, although they are high enough not to be of practical concern for most applications.

Rising-Edge: A transition from a logic 0 to a logic 1. Also known as a positive edge.

Register Transfer Level (RTL): A description of a digital electronic circuit in terms of data flow between registers which store information between clock cycles in a digital circuit. RTL description specifies what and where this information is stored and how it is passed through the circuit during its operation. Also called Register Transfer Logic.

Sensor: A transducer that detects a physical quantity and converts it into a form suitable for processing. For example, a microphone is a sensor which detects sound and converts it into a corresponding voltage or current.

Standard Cell: This device differs from the gate array since each cell may be different and optimized for each standard function. There are no standard layers to the device and each layer of the chip is a unique design.

State Machine: The actual implementation (in hardware or software) of a function that can be considered to consist of a set of states through which it sequences.

Static Random Access Memory (SRAM): A type of memory that is faster and more reliable than the more common DRAM (dynamic RAM). The term static is derived from the fact that it doesn't need to be refreshed like dynamic RAM, but it loses its memory if it is powered off.

Switch: A device for making or breaking an electric circuit or for selecting between multiple circuits.

Synchronous:

(1) A signal whose data is not acknowledged or acted upon until the next active edge of a clock signal.

(2) A system whose operation is synchronized by a clock signal.

System-on-chip (SoC): A complete product that contains all the necessary electronic circuits and parts for a system on a single integrated circuit. Also called "system-on-a-chip" or SoaC

Trace: A line or wire of conductive material – such as copper, silver, or gold – on the surface of, or sandwiched inside, printed circuit board (PCB). An individual trace is often called a run. Traces carry an electronic signal or other forms of electron flow from one point to another.

Translation: Converting the results of the synthesis process to the format supported internally by the chip vendor's place-and-route tools.

Truth Table: A convenient way to represent the operation of a digital circuit as columns of input values and their corresponding output responses.

Verilog: A Hardware Description Language for electronic design and gate-level simulation.

Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL): A Hardware Description Language for electronic design and gate-level simulation.

Via: Feed-through. A plated through-hole in a printed circuit board used to route a trace vertically in the board, that is, from one layer to another.

Volatile: A memory element that loses its contents when power is removed from the device. SRAM-based devices are volatile and require another device to store their configuration program.

3.2 ACRONYMS

A/D	Analog to Digital
ABEL	Advanced Boolean Equation Language
ADC	Analog to Digital Converter
ASIC	Application Specific Integrated Circuit
BIOS	Basic Input/Output System
CE	Complex Electronics
CEH	Complex Electronic Hardware
CLB	Configurable/Complex Logic Block
CM	Configuration Management
CMM	Capability Maturity Model
CPLD	Complex Programmable Logic Device
CUPL	Cornell University Programming Language
D/A	Digital to Analog
DSP	Digital Signal Processor

EELV	Evolved Expendable Launch Vehicle
EEPLD	Electrically Erasable Programmable Logic Device
EEPROM	Electrically Erasable Programmable Read-Only Memory
EHW	Evolvable Hardware
EPLD	Erasable Programmable Logic Device
EPROM	Erasable Programmable Read-Only Memory
FAA	Federal Aviation Administration
FIFO	First In First Out
FPGA	Field Programmable Gate Array
GAL	Generic Array Logic
GOES	Geostationary Operational Environmental Satellite
GPS	Global Positioning System
HDL	Hardware Description Language
HESSI	High Energy Solar Spectroscopic Imager
I/O	Input/Output
IC	Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
IP	Intellectual Property
ISS	International Space Station
IT	Information Technology
JEDEC	Joint Electronic Device Engineering Council
JHDL	Java Hardware Description Language
JTAG	Joint Test Action Group
LUT	Look-Up Table
MAPLD	Military-Aerospace Programmable Logic Devices (a yearly conference)
NRE	Non-Recurring Engineering
PAL	Programmable Array Logic
PCB	Printed Circuit Board

PDA	Personal Digital Assistant
PL	Programmable Logic
PLA	Programmable Logic Array
PLC	Programmable Logic Controller
PLD	Programmable Logic Device
PROM	Programmable Read-Only Memory
QA	Quality Assurance
RAM	Random Access Memory
RC	Reconfigurable Computing
RTL	Register Transfer Level
SA	Software Assurance
SBIRS-High (-Low)	Space Based Infrared System
SEI	Software Engineering Institute
SIRTF	Space Infrared Telescope Facility, renamed Spitzer Space Telescope
SoaC	System-on-a-Chip
SoC	System-on-Chip
SOHO	Solar and Heliospheric Observatory
SRAM	Static Random Access Memory
TDRS	Tracking and Data Relay Satellite
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

CHAPTER 4. COMPLEX ELECTRONICS OVERVIEW

"Complex electronics" is a term applied to various forms of programmable or designable hardware devices. The two elements of the term - complex and electronics - can be used to help distinguish what devices are, or are not, of interest.

4.1 BLURRING THE HARDWARE/SOFTWARE LINE

Programmable Logic devices are now blurring the hardware/software boundary. These devices can now be programmed to perform tasks that were previously handled in software, such as communication protocols. With increased complexity, the possibility of "software-like" bugs (incorrect logic) or unexpected interactions is more likely. It is vital to be able to assure that the systems are designed and implemented correctly, tested fully, and are reliable.

Figure 1 below shows the relationship of software, firmware, Programmable Logic Controllers (PLCs), electronics hardware, and complex electronics (the items in the red boxes). Boxes above the boundary line are software and those below the line are hardware. Complex electronics straddles the line.

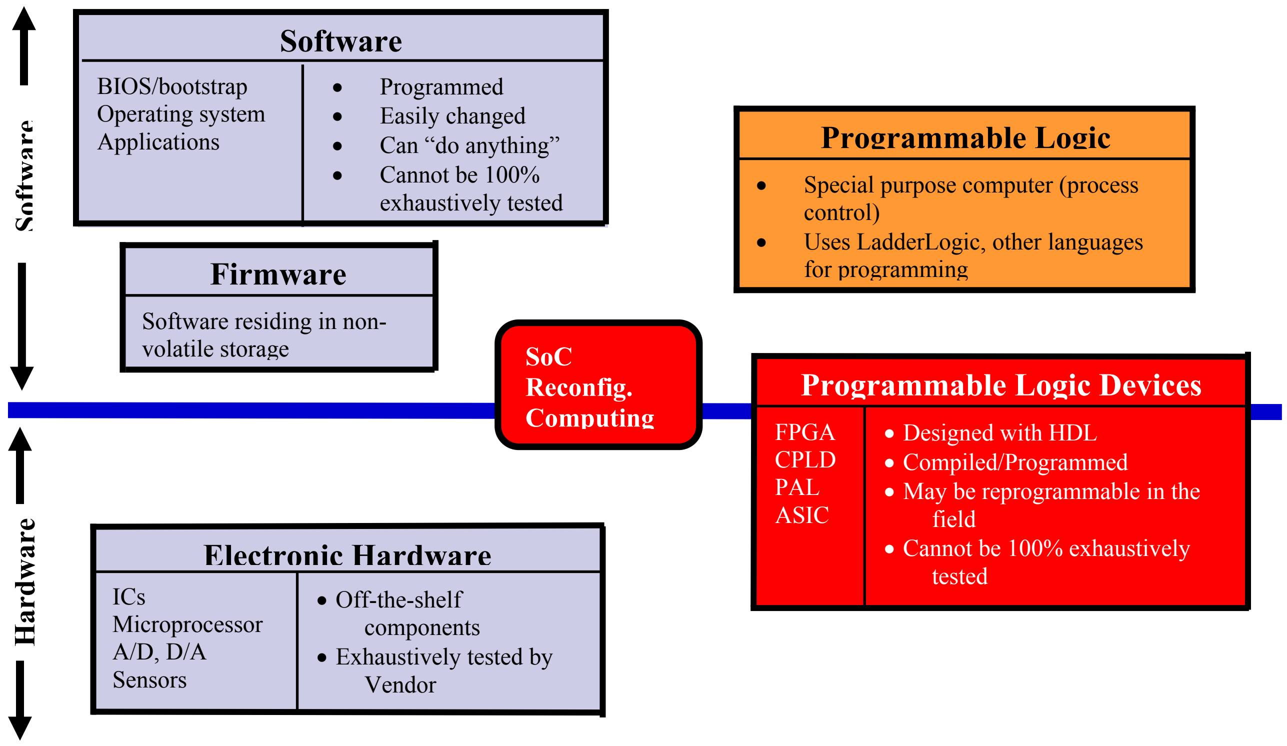

Software
Software
BIOS/bootstrap
Operating system
Applications
• Programmed
• Easily changed
• Can "do anything"
• Cannot be 100% exhaustively tested
Firmware
Software residing in non-volatile storage
Programmable Logic
• Special purpose computer (process control)
• Uses LadderLogic, other languages for programming
SoC
Reconfig.
Computing
Programmable Logic Devices
FPGA
CPLD
PAL
ASIC
• Designed with HDL
• Compiled/Programmed
• May be reprogrammable in the field
• Cannot be 100% exhaustively tested
Hardware
Electronic Hardware
ICs
Microprocessor
A/D, D/A
Sensors
• Off-the-shelf components
• Exhaustively tested by Vendor

Figure 1: How Complex Electronics Compares

The Federal Aviation Administration (FAA) has become concerned about the usage of complex electronic hardware in aviation. A study in 1995 stated, "There are no techniques and methods of design, documentation, testing, and verification identified or recognized by the Federal

Aviation Administration (FAA) for today's complex hardware designs." Since that time, the FAA has worked with other organizations to develop DO-254, "Design Assurance Guidance for Airborne Electronic Hardware," which provides guidelines on the use of process assurance for complex electronic hardware.

The pace of technological change and the new uses that people find for current technology are strong motivators for NASA to begin to define acceptable assurance practices for complex electronics. An example of an assurance challenge is adaptive or reconfigurable computing, in which computers, chips, or systems alter their functionality to adapt to changing applications and situations. Adaptive computing is usually implemented with FPGAs and allows for parallel processing. Adaptive computing is expected to be the next breakthrough in computing. Many applications of the technique for the military are being proposed, and adaptive computing is likely to be used in space systems.

4.1.1 How does Programmable Logic differ from Firmware?

Firmware has various definitions, but the most common is found in IEEE 610.12-1990: "The combination of hardware device and computer instructions and data that reside as read-only software on that device."

Complex electronics, such as FPGAs and ASICs, are not firmware because what resides in them is not a software program. Instead, software is used to define the logic structure for a hardware device, which is what these devices become once they are programmed. These devices are better thought of as hybrid hardware/software devices, or "soft hardware."

Some types of complex electronics are even harder to define, such as System-on-Chip (SoC) and FPGAs:

SoC is a complete product that contains all the necessary electronic circuits and parts for a system on a single integrated circuit. SoCs may include embedded software (i.e., firmware) as part of the device. SoC devices combine a microprocessor, input and output channels, and often an FPGA for programmability.

FPGAs are "soft hardware," except when they are used in reconfigurable or adaptable computing. In that case, they are part of a complex system that is reprogrammed on the fly. The FPGAs replace a microprocessor, and the act of reprogramming them (and the logic that determines the activities) is the software of the system. FPGAs can have from 30,000 to over one million logic gates.

4.1.2 Comparing Complex Electronics and Software

Complex electronics devices do not work in the same way as software. The main difference is that software is serial (one activity is performed after another) and hardware is parallel (multiple operations occur at the same time). It is very important to always remember that the ultimate result of a programmable logic device is hardware. Hardware programming languages, such as VHDL, can be thought of as a virtual or abstract piece of hardware.

However, similarities exist between programming languages for complex electronics (e.g., Verilog or VHDL) and software languages. VHDL, for example, is based on Ada syntax, has data types common to most higher-level languages, uses objects (e.g., constants and variables), and has sequential statements.

A software assurance engineer reviewing programmable logic "code" should not be lulled by the similarities to regular programming languages. Complex electronics and programmable logic devices are ultimately hardware, and those differences must be acknowledged.

4.2 PROGRAMMABLE VERSUS DESIGNABLE DEVICES

Programmable Logic Devices (PLDs) are hardware integrated circuits that are programmable by the user. They contain configurable logic and flip-flops, which are linked together with programmable interconnects. Memory cells control and define the function that the logic performs and how the various logic functions are interconnected. PLDs can be divided into various categories and range from simple devices to complex devices capable of being programmed on-the-fly. Devices in this category include:

- Programmable Array Logic (PAL)
- Generic Array Logic (GAL)
- Programmable Logic Array (PLA)
- Complex Programmable Logic Device (CPLD)
- Field Programmable Gate Array (FPGA)

Some integrated circuits can be designed by the user and submitted to a manufacturer for creation of multiple copies. This allows specialty circuits to be designed for a device, such as a cell phone. Once created, the devices cannot be reprogrammed by the user. ASICs and System-on-Chip (SoC) are examples of designable devices.

4.2.1 How to Identify Complex Electronics?

The electronics part of this term is fairly easy to identify. Electronics refers to the flow of charge (moving electrons) through nonmetal conductors (mainly semiconductors), as opposed to electrical, which refers to the flow of charge through metal conductors. So all the devices listed above qualify. So do off-the-shelf integrated circuits (ICs), microprocessors, logic gates, analog-to-digital converters, buffers and other components.

The "complex" adjective is used to distinguish between simple devices, such as off-the-shelf ICs and logic gates, and user-creatable devices. More information on distinguishing between simple and complex is presented later in this handbook. For now, a good rule of thumb is, if you can program or design the internal logic of the device and it has more than a few gates and connections, it is probably complex.

Does firmware fall under this category? Firmware has various definitions, but the most common is found in IEEE 610.12-1990: "The combination of hardware device and computer instructions and data that reside as read-only software on that device." In other words, it is software that is placed in a read-only device, such as an EPROM or Flash, from which the software may be read or copied. The EPROM acts solely as a storage device, much like a disk. The software may be complex and reside on electronic components, but it does not affect the internal logic or configuration of the chips. Firmware is not considered complex electronics. Table 1 gives some examples:

Table 1: Complex Electronics Examples

Item	User Interactions	Complex Electronics?	Why or Why Not?
Complex Programmable Logic Device	Define and program the internal logic elements	Yes	Electronic and complex
FIFO	Use it	No	Electronic, but not complex
Microprocessor	Execute software instructions on it to perform arithmetic and other operations.	No	The software executes pre-defined commands. It does not change the internal logic arrangement of the microprocessor.
Software	Design, develop, execute	No	Definitely complex, but not electronic
Application Specific Integrated Circuit (ASIC)	Design, use resulting chip	Yes (above a threshold)	Most ASICs are complex. It is possible to make a simple ASIC, though such devices are likely to be already available.
EEPROM (Electrically Erasable Programmable Read-Only Memory)	Program the device with data or software	No	The device itself is not complex. The software or data does not change the internal logic of the device.

4.2.2 A Bit of History

The story starts with the development of discrete logic. Each logic chip had a purpose (e.g., AND gate, OR gate, flip-flop) and could be wired together with other chips to make the desired circuit. Pinouts on the chip were fixed. Manufacturing such a system took a lot of time because each design change required that the wiring be redone. This usually meant building a new printed circuit board.

The chip makers solved the problem of time-consuming rewiring for design changes by placing an unconnected array of AND-OR gates in a single chip called a programmable logic device (PLD). The PLD contained an array of fuses that could be blown open or left closed to connect various inputs to each AND or OR gate. You could program a PLD to perform the logic functions you needed in your system. Since the PLDs could be rewired internally, there was less of a need to change the printed circuit boards which held them.

4.3 SIMPLE PROGRAMMABLE LOGIC DEVICES

There are a variety of simple PLDs. They are called simple to distinguish them from the Complex PLDs (CPLDs, discussed below), and because they are actually pretty simple devices, as modern integrated circuits go.

4.3.1 Programmable Array Logic

Programmable Array Logic (PAL) chips are a family of fuse-programmable integrated circuits originally developed by MMI (Monolithic Memories, Inc.). The word "Logic" in the name signifies that the chips allow the user to program a set of AND and OR gates (or NAND/NOR) to create the desired logic sequence. PALs consist of a programmable AND array followed by a non-programmable OR array. Inputs are fed into the AND array, which performs the desired AND functions and generates product terms, which are then fed into the OR array. In the OR array, the outputs of the various product terms are combined to produce the desired outputs.

Using a fixed number of OR gates, rather than a completely programmable set, allows the device to be fast. The high speed available in PALs makes them still popular today, despite the abundance of newer chips. Figure 2 shows the structure of the PAL.

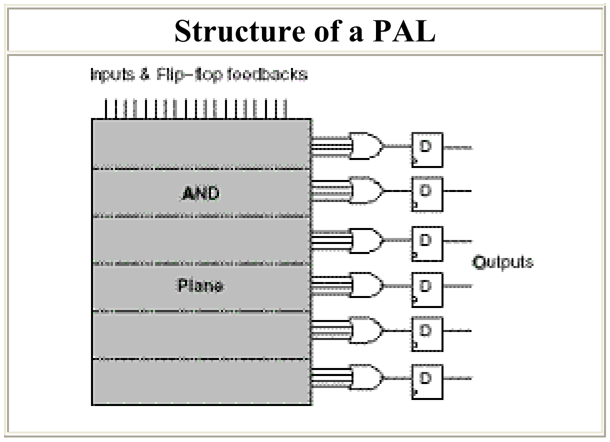


Figure 2: Example of PAL Structure

Fuse-programmable has to do with how PALs are programmed. Connections between the gates in a PAL are made using fuses that are either connected or disconnected (blown). Overvoltage (above the operational limits of the chip) is used to blow the fuses for the connections that are not desired. This operation is permanent, so once programmed, a PAL cannot be reprogrammed.

Fuse maps, which determine what fuses are, or are not, blown for a particular PAL can be generated in several ways. Languages such as PALASM or CUPL can be used, with the resulting logic design compiled into JEDEC (Joint Electronic Device Engineering Council) ASCII/hexadecimal files. Modern support software for PALs allows a direct translation from a schematic, truth table, or state table to the fuse map. Some software even accepts timing diagrams as input. Hardware description languages (HDL) can also be used to synthesize the fuse map. However the map is created, it must be provided as input to a special electronic programming device, available from either the manufacturer or a third-party, for physical programming of the chip.

4.3.2 Generic Array Logic

Generic Array Logic (GAL) was introduced by Lattice Semiconductor. A GAL consists of a reprogrammable AND array, a fixed OR array, and reprogrammable output logic. Electrically Erasable Programmable Read-Only Memory (EEPROM) is used, rather than fuses, to provide the connections. This allows the GAL to be erased and reprogrammed.

The GAL is very useful in the prototyping stage of a design, when any bugs in the logic can be corrected by reprogramming. GALs are programmed and reprogrammed using a PAL programmer, and the same types of languages or processes used for PAL chips. If speed is important (and it usually is), a PAL can be used, once the design is finalized.

4.3.3 Programmable Logic Array

Programmable Logic Array (PLA) devices differ from PALs in the OR-gates area. PALs could only be programmed in the AND-plane. With PLA chips, a set of programmable AND planes are linked to a set of programmable OR planes, which can then be conditionally complemented to produce an output. PLA devices allow far more design options than PALs, but the downside is reduced performance.

Like PALs, PLA devices are fuse-based and can be programmed only once. Tools and languages are readily available to translate a logic design into the fuse map required for PLA programming. Table 2 gives a comparison of the simple programmable devices

Table 2: Simple PLD Comparisons

	PROM	PAL	GAL	PLA
Input lines	hard-wired	programmable	programmable	programmable
Output lines	programmable	hard-wired	programmable	programmable
Versatility	low	moderate	moderate	high
Difficulty in manufacturing, programming, and testing	low	moderate	low	high
Reprogrammable?	No	No	Yes	No

4.4 COMPLEX PROGRAMMABLE LOGIC DEVICES (CPLD)

Simple PLDs can only handle up to 10 to 20 logic equations, which is not a very large logic design. Designers need to figure out how to break a larger design apart and fit the pieces into a set of PLDs. This is a time-consuming process, and means you have to interconnect the PLDs with wires. When there were only discrete logic chips, the use of wires meant that any design change will likely require a new circuit board, not just reprogramming the PLDs. To counteract this constraint, the chip makers began by building much larger programmable chips, including CPLDs and FPGAs.

A CPLD contains a set of PLD blocks whose inputs and outputs are connected together by a global interconnection matrix. So a CPLD has two levels of programmability: each PLD block can be programmed, and then the interconnections between the PLDs can be programmed. A key feature of the CPLD architecture is the arrangement of logic cells on the periphery of a central shared routing resource. CPLDs use EEPROM, SRAM, or Flash memory to hold the interconnect information.

CPLDs contain the equivalent of many PALs linked by programmable interconnections, all in one integrated circuit. CPLDs are equivalent to about 50 typical PLD devices and can replace thousands, or even hundreds of thousands, of logic gates.

Programming CPLDs depends on the chip and the application. Some CPLDs can be programmed in a PAL programmer, but that gets difficult if the chip has hundreds of pins, or is surface-mounted. Many CPLDs can be programmed over a serial line from a computer. The CPLD contains a circuit that decodes the data stream and configures the CPLD to perform its specified logic function.

A new interface for programming and testing CPLDs is Joint Test Action Group (JTAG). This interface is defined by the IEEE Standard 1149.1-1990, Test Access Port and Boundary Scan Architecture. Boundary scan is a technique for accessing and stimulating a chip or subsystem via external pins to perform internal test functions on the device. A JTAG interface is primarily used for testing integrated circuits, but it can also be used as a mechanism for debugging embedded systems. A JTAG interface is a special four-pin (data in, data out, clock, test mode select) interface added to a chip. Multiple chips on a board can have their JTAG lines daisy-chained together, so the test probe only needs to connect to a single JTAG port to have access to all chips on a circuit board. Figure 3 shows the difference between the internal layout of a CPLD vs. FPGA device.

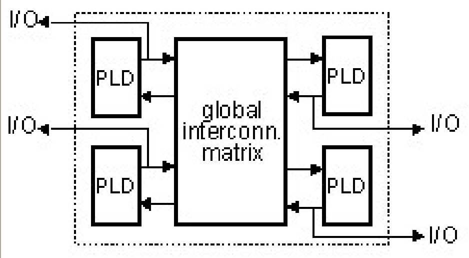
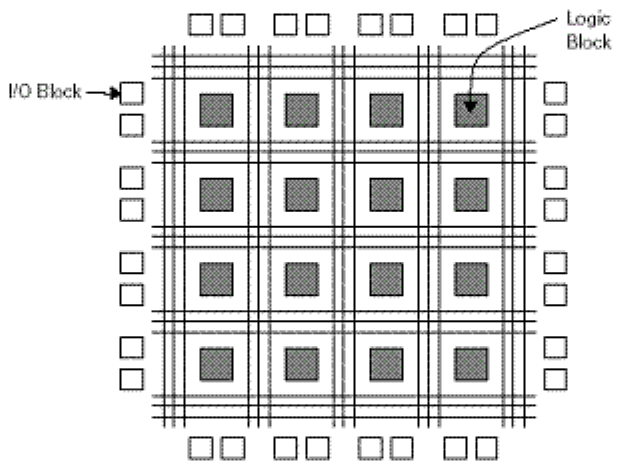
CPLD Diagram	**FPGA Diagram**
 I/O I/O PLD PLD global interconn. matrix PLD PLD I/O I/O	

Figure 3: CPLD vs. FPGA Layout

4.5 FIELD PROGRAMMABLE GATE ARRAY (FPGA)

While PALs were busy developing into GALs and CPLDs, a separate stream of development was occurring, based on gate-array technology. The resulting device is the FPGA which was first introduced in the late 1970s. “Field programmable” simply means that the device can be programmed by the user. Many field programmable devices can be programmed with the chip soldered to the circuit board, allowing true in-the-field upgrades to be possible.

FPGAs use a grid of logic gates, similar to that of an ordinary gate array. An FPGA has a collection of simple, configurable logic blocks arranged in an array with interspersed switches that can rearrange the interconnections between the logic blocks. Each logic block is individually programmed to perform a logic function (such as AND, OR, XOR, etc.) and then the switches are programmed to connect the blocks so that the complete logic functions are implemented. FPGAs vary in size from tens of thousands of logic gates to over one million.

The interconnections for the logic blocks are programmable switches. FPGAs may use EEPROM, SRAM, antifuse, or Flash technology to store the programming. In most larger FPGAs the configuration is volatile, and must be re-loaded into the device whenever power is applied or different functionality is required.

FPGAs are typically programmed in hardware description languages (HDLs) like Verilog or VHDL. These high-level languages are used because manual lower level design (such as schematic capture) becomes impractical as designs become large. HDLs also allow the FPGA design to be simulated and tested prior to implementation in the hardware.

4.6 APPLICATION SPECIFIC INTEGRATED CIRCUIT (ASIC)

ASICs are integrated circuits (ICs) designed for specific applications. Unlike standard ICs which are produced by the chip manufacturers, ASICs are designed by the end user and then produced in volume. ASICs allow a user to combine many parts and functions into a single chip, reducing cost and improving reliability.

ASICs can be large or small. They are usually produced in large quantities, and it can be very expensive to produce only a few. If you are manufacturing cell phones, it makes sense to develop an ASIC for your specific needs. If you are flying a space experiment and will need at most a few chips, it would be much more economical to use programmable logic, such as FPGA or CPLD devices.

An interesting twist is the production of ASICs that include programmable logic (FPGA, CPLD or PAL) devices as part of the chip. Another new technology that combines ASICs with programmable parts is the System-on-Chip, described below.

4.7 SYSTEM-ON-CHIP (SOC)

System-on-Chip combines all the electronics for a complete product into a single chip. SoCs include not only the brains (e.g., microprocessor) but also all required ancillary electronics, such as switches, comparators, resistors, capacitors, timing elements, and digital logic. Figure 4 gives a sample configuration for a SoC.

SoCs could include:

- Digital/analog functions
- Sensors
- I/O
- Communications
- Readymade sub-circuits (IP)
- Programmable devices
- Digital Signal Processor

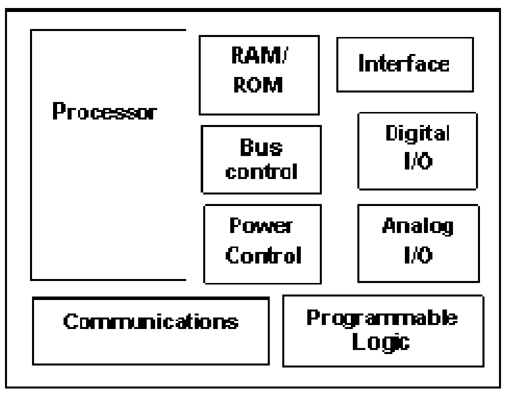


Figure 4: SoC Example Configuration

SoCs are usually ASICs, though they can be designed to include programmable logic components. SoCs can also be implemented on FPGAs. System-on-chip versions have a variety of features:

- Soft Instruction processor architectures allow a designer to customize the CPU architecture. The specific instructions supported, the peripherals available to it, and the number of registers are just some ways these devices can be tailored for your application. Some vendors provide mechanisms to add, delete, and create highly tailored instructions. Design packages for these architectures sometimes include performance tools with instant feedback on the performance, die size, and power requirements of a particular design. With the final architecture residing in silicon, these types of architectures are well suited for high volume, low cost applications which formerly would have used ASICs.
- Configurable processors are FPGA based. In these architectures, standard and customer-derived logic engines can be easily added, modified, and extended as needed. By moving discrete logic functionality to internal FPGA, the designer gets a highly flexible logic solver based around a standard processor core. With FPGA logic instead of foundry logic, the logic can be easily revised at any point in the design cycle.

4.8 CONCERNS AND ISSUES

4.8.1 Verification Issues with Complex Electronics

Verification means that you have demonstrated that the system or subsystem meets the requirements you have specified. Complex systems, especially those including software, are hard to adequately verify. Complex electronics adds additional verification concerns to the mix:

- Tool-induced design errors occur and can be difficult to detect. Tools are a vital part of complex electronics design, and the designer often does not know what errors a tool could potentially produce.
- Complex functionality cannot be completely simulated, nor the resulting chip completely tested.

- It can be difficult to detect faulty operation of complex electronics due to design or tool-induced errors, unexpected interactions, or even defects in the silicon.
- Due to extremely small ASIC geometries, certain analog and transmission line phenomena occur internal to the ASIC, generating failures that are data-sensitive. Designers and tools may not account for these effects, which can easily escape notice during test.

4.8.2 Assurance Issues with Complex Electronics

In addition to challenges with testing and verifying the designs and implementations of complex electronics, quality assurance professionals are struggling with how to adequately deal with the "software-like" aspects of these devices. Some problems and concerns are:

- ASICs and FPGAs have been used to avoid the rigors of the software approval process. This results in fundamental verification matters being bypassed.
- Complex electronic devices are designed and programmed by engineers, often without quality assurance oversight or configuration management control of the designs. In addition, the development process may not be well defined or followed.
- ASICs, FPGAs, and System-on-Chip (SoC) can contain embedded microprocessor cores with user-supplied software. They combine electronics and firmware into one chip. The presence of this firmware (i.e., software) is not always obvious to assurance personnel.
- High-level languages (e.g., C, C++) are now being used to define complex electronic designs (in whole or in part).
- Hardware quality assurance professionals may not be fully cognizant of the functions, potential problems, and issues with these devices.
- Software assurance personnel are currently not trained to understand complex electronics, and may not be able to provide effective oversight and assurance.
- Meaningful verification efforts require the person performing the verification to be knowledgeable about the complex electronic device and the tool suite used to create and implement the design.

4.9 SUMMARY

Programmable and designable electronics have grown over the years, both in number of devices and in the complexity of the devices. The devices can be roughly grouped by function and complexity.

- Simple, non-programmable logic - ICs
- Simple, programmable logic - PAL, GAL, PLA
- Complex, programmable logic - CPLD, FPGA, reconfigurable computing
- Complex, designable logic - ASIC
- Complex, designable, and/or programmable logic - SoC

To explore the complex devices in more depth, refer to the descriptions in Appendix A.

CHAPTER 5. DESIGN PROCESS

5.1 OVERVIEW OF THE COMPLEX ELECTRONICS DESIGN PROCESS

Creating complex electronics begins where all systems and subsystems begin - with defining the requirements for the device. Without good requirements, the most elegant design or implementation could fail to meet the original need. Designing and implementing complex electronics occurs within the context of the larger system, as shown in Figure 5 below.

Requirements for the complex electronics are driven by the system they are a part of and the environment where they will be used. A simple home appliance will place fewer demands (requirements) on a device than a sophisticated satellite application will. Because these devices are hardware, the process of complex electronics design involves looking at both the chip capabilities and constraints (e.g., how many gates does it have, how much power does it need) and how the design works with and against those constraints and capabilities.

5.1.1 Design Life Cycle

In typical software design, the software requirements are flowed down from the system requirements. Software development may follow a waterfall, iterative, evolutionary, spiral, or other development methodology. Regardless of the development (design) life cycle, the processes of determining the requirements, creating the design, implementing the design, and verifying the implementation are all included. Since it is easy to show graphically, this handbook will use the waterfall life cycle as a generic life cycle. Figure 5 below compares the complex electronics lifecycle to software.

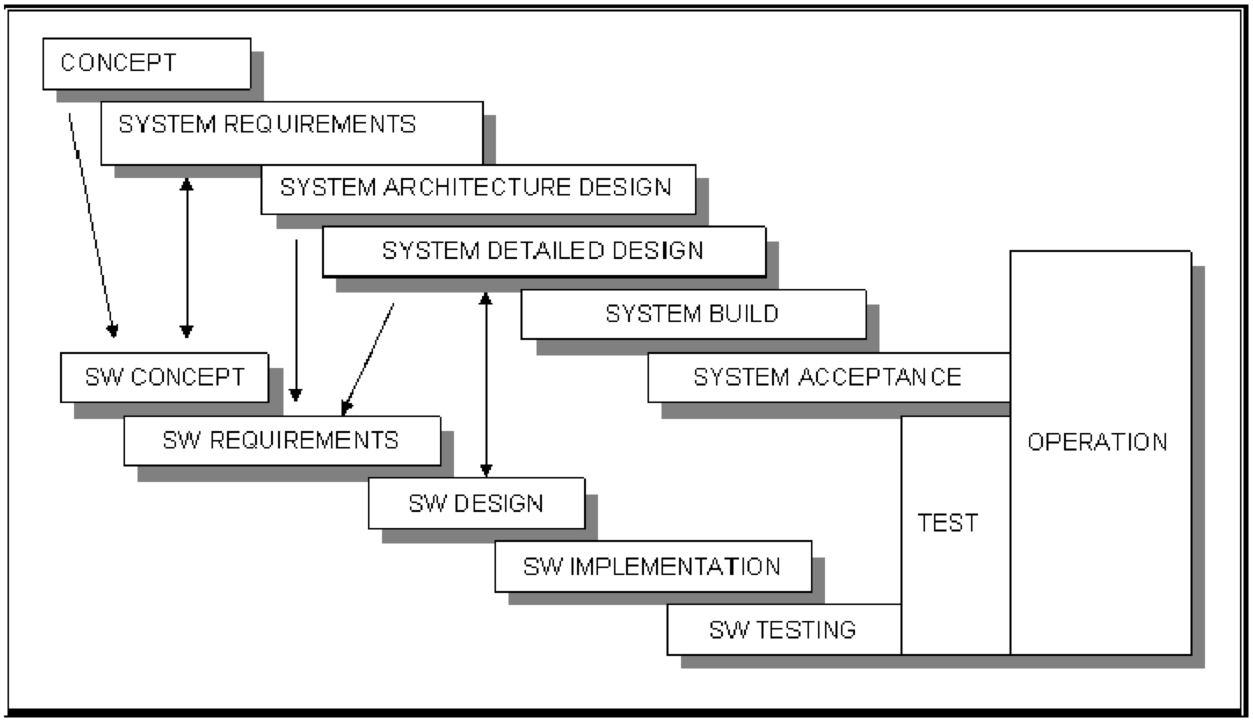


Figure 5: CE vs. SW Life Cycles

Like software, the design and development life cycle for complex electronics can follow any life cycle methodology. Some of the steps vary from those familiar to software developers. Figure 6 depicts the design process for complex electronics.

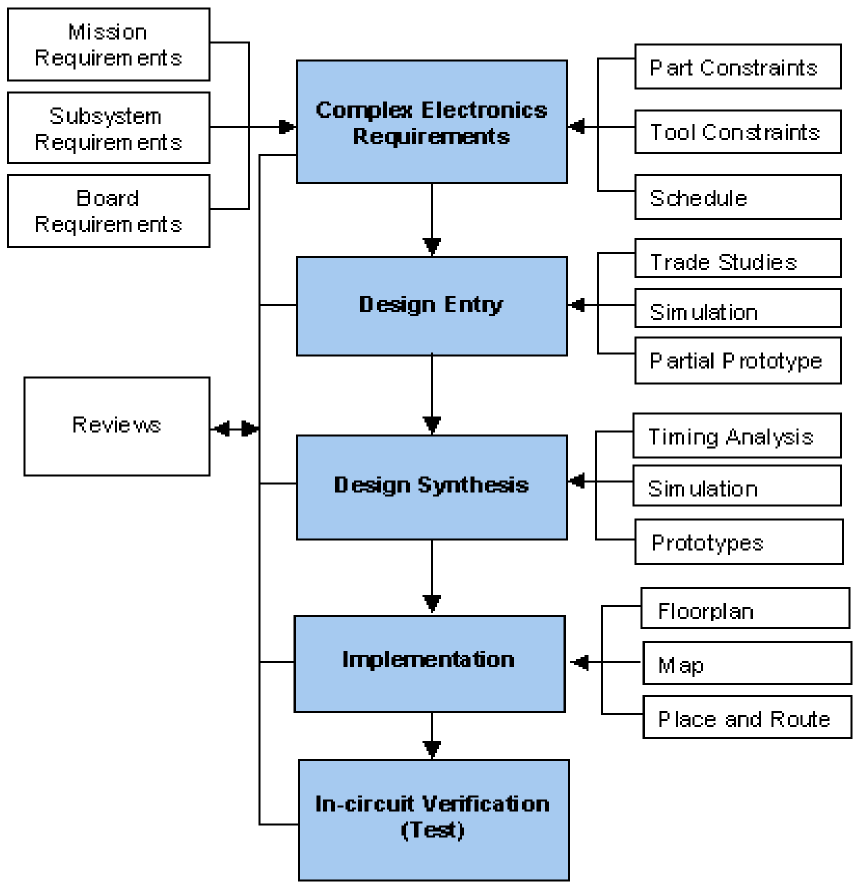


Figure 6: Example CE Waterfall Development

The basic design flow starts out very similar to software, with the decomposition of system or subsystem requirements to the particular complex electronic device. After that is completed, the engineers take the requirements and generate a design, often in a hardware description language (design entry). The design has to be "compiled" for the device (design synthesis). Synthesis is more complicated than just running a compiler. During synthesis, the design is mapped to the logic gates of the device. Simulations are used to verify that the design is correct and can meet the requirements and performance goals.

The implementation of complex electronics involves one more level in the mapping of the logic (design) to the chip. The placement of the logic blocks within the chip, and the routing between blocks, are some of the processes that occur during implementation. This process is loosely comparable to the linking step in software, where the compiled program is fixed up for the software environment in which it will operate. At the end of the implementation phase, the final step is to "burn" or program the device.

While the simulation that occurs before the design is committed to hardware can find most defects, the actual hardware device needs to be tested in the circuit. Real signals are applied, and the real output is tested. You usually cannot get the degree of testing with in-circuit verification that you can with simulation, because inputting out-of-range signals might be difficult, access to the hardware pins might not be possible, and, in real projects, someone always wants to use the hardware as soon as it is completed. However, functional testing in a variety of conditions is an important verification step. Errors in the silicon chip are possible. Errors induced by the tools are more likely. Sometimes the real world acts differently than expected and can influence how the device works.

During this process, the tasks of the assurance engineers (quality/hardware) can vary between projects with some only taking a look at the system at a high level, and then verifying that the final device matches the design and that it was programmed according to a defined process. NASA is looking at how to adequately verify the complex electronics device. More information is provided in Section 6.2. Table 3 gives a comparison of the development process for software and complex electronics.

Table 3: CE vs. SW Development Phases

Software		**Complex Electronics**	
Requirements	Software requirements flow down from system and subsystem requirements.	**Requirements**	Requirements for complex electronics flow down from system and subsystem requirements.
Design	Architectural and detailed designs are created, using UML, flow diagrams, and other tools.	**Design Entry**	The design is created primarily in a hardware description language, such as Verilog or VHDL.
		Synthesis	Synthesis is the process that takes the higher level designs and optimally translates them to a gate-level design which can be mapped to the logic blocks in a complex electronic device.
Code	The design is translated (manually or automatically) into a programming language (code), and then compiled into an executable module.	**Implementation**	Implementation is where the design meets the silicon - the mapping created by synthesis is converted into a chip layout. The final step in implementation is to put the design into the chip - either through programming (burning) or manufacturing (for ASICs).
Test	The software is tested in individual units and as part of the system. Testing may involve additional software that simulates inputs to the software under test.	**Test**	Testing occurs during the design entry, synthesis, and implementation phases, in the form of simulations. Both expected (valid) and unexpected inputs are tested. Once the device is created, it is tested as part of its subsystem (in-circuit testing).

5.2 REQUIREMENTS AND SPECIFICATIONS

The first step in the design process is to understand (and document) the functions the complex electronics device must perform and the constraints under which it operates. The act of documenting the requirements has some useful effects that actually can save you time in the long run. Benefits include:

- The design team thinks through the issues and reaches agreement. Some issues are well understood at a high level, but raise additional questions when working at the hardware level.
- Interfaces to other areas (software, other hardware) are defined and available for review by all affected parties.
- Non-engineers can understand what the chip or device is supposed to do.
- If the trade-offs and rationales are documented, as well as the requirements, future design changes will require less impact assessment.
- The requirements can be reviewed to assure that they provide measurable, testable criteria.
- Requirements traceability into the design and implementation can be performed - which is vital in mission- or safety-critical applications.

A good specification for complex electronics will contain:

- A description of how the device fits into the larger system. A block diagram is very helpful.
- A description and list of all the major functions the device will perform. A block and/or flow diagram can be used to show this information.
- A description of the device and interfaces, such as:
  - Chip physical information (size, type, number of pins, etc.)
  - I/O pin mapping and description (output drive capability, input threshold level)
- Timing estimates for:
  - Setup and hold times for input pins
  - Propagation times for output pins
  - Clock cycle time
- High-level estimates and goals
  - Gate count estimate
  - Power consumption target
- Constraints on the device
- Other requirements or criteria the device must implement
- Design-related choices (may be in a management plan)

- Tools that will be used at all stages of development
- Hardware Description Language chosen

5.2.1 Assurance Roles

What role does software or hardware quality assurance play in verifying the requirements specification for complex electronics? The reality is that many assurance engineers, regardless of their specialty, have little understanding of the complexities of these devices. Any review or evaluation will be to the level of knowledge of the assurance engineer.

- **Hardware quality assurance engineers** are the primary assurance people to deal with complex electronics. Hardware quality assurance engineers with a background in electronics will evaluate the requirements for the complex electronic device for accuracy, completeness, and compatibility with the rest of the system. When the hardware quality assurance engineer has little exposure to, or understanding of, complex electronics, the evaluation will often be very high-level.
- **System safety engineers** will be involved in the review when the devices are part of safety-critical systems or are used as controls or mitigations for hazards. As with hardware quality assurance, system safety engineers usually do not have an in-depth understanding of complex electronics.
- **Software assurance engineers** at NASA are currently only rarely involved with complex electronics. Significant education or training is required to be able to adequately review the requirements and specification for complex electronics at a detailed level. However, this handbook explains how to review specifications for complex electronics at a high-level and look for:
  - Problems with interfaces to other system elements or to the software running on the system
  - Problems, issues, or concerns regarding the functions that are implemented in the hardware
  - Additional constraints that may not be included in the specification, or incorrect constraints
  - Areas where software functions could be implemented in the complex electronics

5.3 DESIGN ENTRY

The first step in creating a design for complex electronics is to choose how you will enter (capture) your design. Early chip designs were primarily performed with schematic capture. Schematic capture (also called schematic entry) creates the electronic diagram, or schematic, of the electronic circuit. This is usually done interactively with the help of a schematic capture tool also known as schematic editor.

While schematic capture works fine for simple designs, complex electronics almost always require the use of a hardware description language (HDL). HDLs are any languages that are used for formal description of electronic circuits. These languages can describe the operation, design, and simulation tests of the circuit. HDLs can show several aspects of the design, including the temporal behavior and spatial structure. One major difference between HDLs and software languages is the aspect of timing and concurrency.

One very nice aspect of HDLs is that they can be used as "executable specification" to simulate the circuit. Simulation software can be part of the tool suite provided by the vendor or a third-party program. Simulators read the HDL "code" and model the structure and flow of the circuit through time.

The two primary description languages are VHDL and Verilog. A later section in this Handbook will discuss these two languages in greater detail. Older HDLs, such as ABEL and CUPL, are still in use, especially for simple designs. Another trend in hardware description languages is to add hardware-specific elements to software programming languages. JHDL is implemented on top of the Java language. SystemC adds hardware constructs as a C++ class library. Still, VHDL and Verilog are by far the most common hardware description languages in use.

Regardless of the method chosen to input the design (a hardware description language or schematic capture), a software tool (or tool suite) is required. Unlike most software development efforts, where tools other than editors, compilers, development environments, and version management software are rarely used, electronics designers require, and use, fairly sophisticated tools. All major complex electronics vendors offer design tools optimized for their devices at a relatively low cost. Third-party tools are common and can provide additional capability. These tools are also often quite expensive. However, because the boundaries between design entry, simulation, synthesis, and place-and-route are well defined, designers can use a variety of tools from different vendors.

A tool suite may include the following types of tools:

- HDL capture and design environment
- Configuration management
- HDL simulator
- Logic analyzer
- Logic synthesis (this is a critically important tool)
- Layout (physical synthesis)
- Design management

5.3.1 Design Views

Complex electronic devices are designed at several levels, and with several "views," or ways of looking at the device. Software shares some of these views (e.g., the behavioral/functional view and the structural view), though software is not concerned with physical layouts. Each of the various views of the device is refined at each of the levels of representation. The Y diagram below, Figure 7, shows how all these views and levels are related.

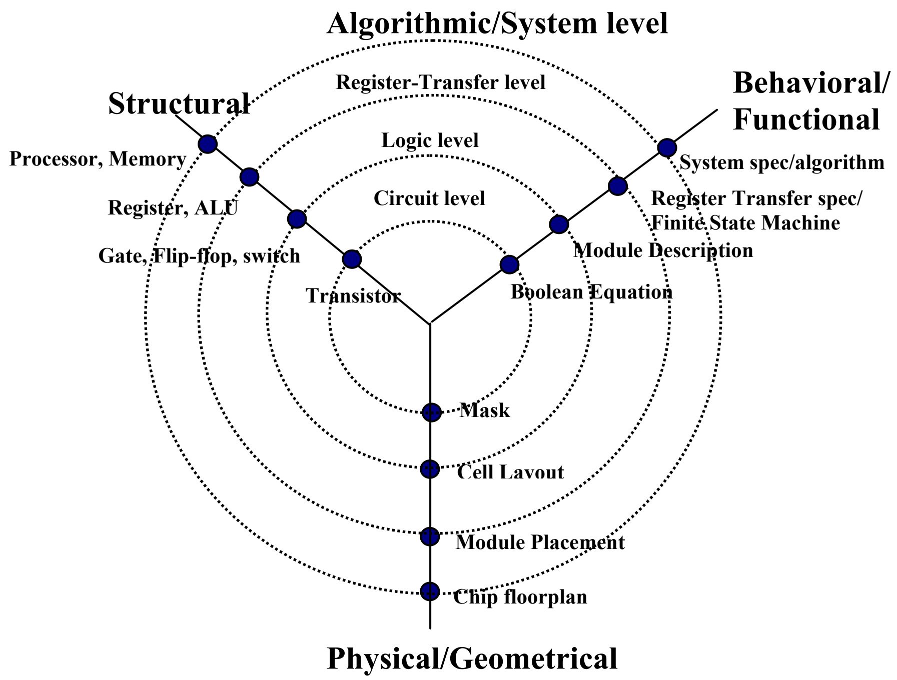


Figure 7: Complex Electronics Design Views

Modern design approaches for complex electronics focus on the behavioral/functional aspects of the devices and use sophisticated tools to create the appropriate structural and physical aspects of the design. Earlier design approaches required much more manipulation at lower levels of the device circuit. With increasing complexity of the devices, the design aspects have been advanced into a more abstract domain, and the work of converting the design into a usable circuit is left to the tools. This abstraction allows the designer and others to understand how the device functions within the context of the system.

A specification in a hardware description language consists of one or more modules. The top-level module specifies a closed system containing both test data and hardware models. Component modules normally have input and output ports. Events on the input ports cause changes on the outputs. Events can be either changes in the values of wire variables (i.e., combinational variables) or in the values register variables, or can be explicitly generated abstract events. Modules can represent pieces of hardware ranging from simple gates to complete systems (e.g., microprocessors), and they can be specified either behaviorally or structurally, or by a combination of the two.

A **behavioral specification** defines the behavior of a digital system (module) using traditional programming language constructs (e.g., IFs, assignment statements). This description of a complex electronic device divides the device (chip) into several functional blocks that are interconnected. A hardware description language is used to describe the behavior of each block. Functional blocks can be a finite state machine, a set of registers and transfer functions, or even a set of other interconnected functional blocks.

A **structural specification** expresses the behavior of a digital system (module) as a hierarchical interconnection of subordinate modules. The components at the bottom of the hierarchy are either primitives or are specified behaviorally. It is in the structural specification that individual inputs and outputs are defined.

5.3.2 Assurance Roles

At the design entry stage,

- **Hardware quality assurance engineers** with a background in electronics will, ideally, evaluate the design for the complex electronic device against the requirements. For many projects, especially when hardware quality assurance engineers have little exposure to, or understanding of, complex electronics, no evaluation will be performed.
- **System safety engineers** will review the design of the devices when they are part of safety-critical systems. Since few system safety engineers are experts in complex electronics, they will work with the designer or hardware quality assurance engineers to evaluate the design from a safety perspective.
- **Software assurance engineers** are not often involved. This handbook provides an understanding that software users and engineers can use to provide a cursory review of the design, especially the VHDL or Verilog code.

5.4 ABSTRACTION

Hardware description languages can be used to describe complex electronics at many different levels of abstraction. An abstraction is a simplified representation of something that is potentially quite complex. It is often not necessary to know the exact details of how something works, is represented or is implemented, because it can be used in its simplified form.

The levels of abstraction for a complex electronic device are:

- System or Behavioral
- Algorithm
- Register-Transfer Level (RTL)
- Gate

The highest level of abstraction is the system level, where the device is mostly a black box that interacts with its environment. Very little is known about the internals of the device, but you do know how it functions (its behavior).

A pure algorithm consists of a set of instructions that are executed in sequence to perform some task. A pure algorithm has neither a clock nor detailed delays. Some aspects of timing can be inferred from the partial ordering of operations within the algorithm. The algorithmic level of abstraction is similar to software programming (e.g.; while ready, do task A and task B, then do task C). Because of the lack of timing information, this level is not synthesizable (able to be mapped to hardware).

The Register-Transfer Level (RTL) description has an explicit clock. All operations are scheduled to occur in specific clock cycles, but there are no detailed delays below the cycle level.

A single global clock is not required but may be preferred. In addition, re-timing is a feature that allows operations to be rescheduled across clock cycles. The RTL level is the input to the synthesis tool.

The gate level of abstraction is the output from the synthesis tool. A gate level description consists of a network of gates and registers, along with technology-specific delay information for each gate. A complex electronics device can be described in one of three domains: behavioral, structural, and physical. Figure 8 shows the various domains in which complex electronics can be described.

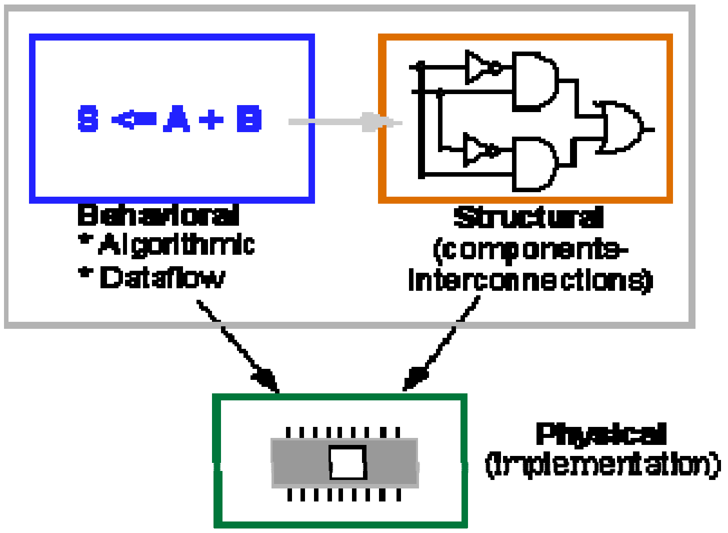


Figure 8: Complex Electronics Domains

Hardware description languages deal with the first two (behavioral and structural). The mapping from the behavioral and structural domains to the physical implementation is performed by the synthesis and place-and-route tools.

Figure 9 shows a simple circuit that warns car passengers when the door is open or the seatbelt is not used whenever the car key is inserted in the ignition lock. At the behavioral level this could be expressed as,

Warning = Ignition_on AND (Door_open OR Seatbelt_off)

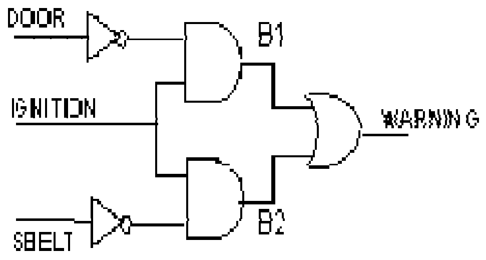


Figure 9: Warning Buzzer Example

The structural level, on the other hand, describes a system as a collection of gates and components that are interconnected to perform a desired function. A structural description could be compared to a schematic of interconnected logic gates. It is a representation that is usually closer to the physical realization of a system.

5.5 HARDWARE DESCRIPTION LANGUAGES

5.5.1 Overview of Hardware Description Languages (HDLs)

While schematic capture works well for small circuits and devices, complex designs require the ability to abstract at a higher level. Thus, hardware description languages were born. One difference between HDLs and software languages is that HDLs are essentially models of the hardware. The languages were initially created to allow simulation of the design and contain all the necessary capabilities to create test benches and simulation models. Simulation of the complex electronics is very common in the design community.

There are two major HDLs that are currently in use: Verilog and VHDL. This handbook will provide a cursory overview of these two languages. However, each of the languages is a course (or two) in its own right. Several good tutorials on the languages are provided in Section 2.2, Links.

The Verilog hardware description language was invented by Philip Moorby in 1983. The first Verilog synthesis tool was introduced in 1987. Verilog was placed in the public domain and is now specified by an IEEE standard (IEEE 1364). This language enables specification of a digital system at a range of levels of abstraction, such as switches, gates, Register-Transfer Level (RTL), and higher. In 2005, SystemVerilog was adopted as IEEE Standard 1800-2005. This update includes items such as structures, pointers, or recursive subroutines that were not present in earlier versions.

VHDL stands for VHSIC Hardware Description Language. VHSIC is an acronym for Very High Speed Integrated Circuit. VHDL is also specified by an IEEE standard (IEEE 1076). VHDL was developed over time, culminating in its initial release in 1987. In June 2006, the VHDL Technical Committee of Accellera approved Draft 3.0 of VHDL-2006. While maintaining full compatibility with older versions, this proposed standard provides numerous extensions that make writing and managing VHDL code easier. Key changes include incorporation of child standards (1164, 1076.2, 1076.3) into the main 1076 standard, an extended set of operators, more flexible syntax of 'case' and 'generate' statements, incorporation of VHPI (interface to C/C++ languages), and a subset of PSL (Property Specification Language).

5.5.2 General Hardware Description Language Concepts

As you learn about HDLs, there are a few major differences from software languages that one needs to keep in mind. First, software is inherently sequential - one instruction is executed after another. Even in multi-threaded or multi-tasking systems, no two tasks operate at the exact same moment. Hardware, however, is parallel in nature - multiple events can be happening simultaneously. Hardware description languages have ways to describe concurrency (parallel execution) and to specify timing. Second, HDLs describe hardware. While at the highest abstraction an HDL can define an algorithm similarly to a software language, at the lower levels of abstraction that algorithm is translated into gates and I/O.

Hardware description languages model two aspects of the hardware: structure and behavior. These two aspects are independent - the structure of the hardware is not dependent on the behavior, and vice versa. The interfaces (input/output signals) from the device to the outside world are part of both the structure (what the device is made of) and the behavior (what it does with the signals). In addition, because HDLs were originally designed as simulation languages, they can create test benches to exercise and test the device with simulated "real world" devices.

The first step when designing and modeling complex electronics in a hardware description language is to partition the design into natural abstract blocks, known as components. Each component is the instantiation of a design entity, which is normally modeled in a separate system file for easy management and individual compilation by simulation or synthesis tools. The total system is then modeled using a hierarchy of components, known as a design hierarchy, which consists of individual subcomponents (subdesign entities) brought together in one higher-level component (design entity). In other words, start with very simple entities (e.g., AND gate) and put them together into components (logical subdivisions within the device), which together become the model of the device.

The two main elements of the HDL description of the complex electronic device are the architecture body (the structure) and the behavioral architecture. The architecture body describes the implementation of a module's inputs and/or outputs. The electrical values of the outputs are some function of the values of the inputs. Of course, each module can be made up of sub-modules, down to the basic entities. The connections between the sub-modules (inputs/outputs) are made using signals.

The architecture body contains:

- Signal declarations, for internal interconnections
- Entity ports (also treated as signals)
- Component instances (instances of previously declared entity/architecture pairs)
- Port maps in component instances (connect signals to component ports)
- Wait statements

The behavioral architecture describes the algorithm performed by the module. While the architecture body described the inputs and outputs, the behavioral architecture describes what goes on to convert those inputs to outputs. More complex behaviors cannot be described purely as a function of inputs. In systems with feedback, the outputs are also a function of time. Fortunately, hardware description languages provide features to handle time as part of the behavior.

The behavioral architecture contains:

- process statements
- sequential statements
- signal assignment statements

- wait statements

You can describe the behavior of a module without describing its structure. You might want to do this if you have an off-the-shelf component as part of your design. You do not really care about the internal structure of the component; you just want to describe what it does. Figure 10 shows the general HDL development process.

This handbook does not provide significant detail on the two main hardware description languages (VHDL and Verilog). See the links below for some tutorials on VDHL or Verilog.

5.5.3 VHDL Tutorials

- http://www.seas.upenn.edu/~ese201/vhdl/vhdl_primer.html
- http://www.gmvhdl.com/VHDL.html

5.5.4 Verilog Tutorials

- http://www.asic-world.com/verilog
- http://ca.olin.edu/cawiki/Fall_2006/Materials?action=AttachFile&do=get&target=VerilogTutorial.pdf
- http://www.doulos.com/knowhow/verilog_designers_guide/

5.5.5 Comparison of VHDL and Verilog

You can design complex electronics in either of the main hardware description languages (VHDL and Verilog). Both provide all the capabilities you require. While the choice of language is mostly a personal preference, there are some differences between the two that may be important for specific applications. Table 4 shows how the two compare.

5.5.6 Coding Standards

Just as in writing software for embedded applications, a coding standard is important when more than one person will ever have to maintain the source code. The big danger is that when the person who wrote the original code leaves or moves on to another project, no one will understand how it works if the code ever has to change. Even the original designer is likely to forget it in several months.

One can easily write individual lines of understandable HDL code that collectively become extremely difficult to follow. A good coding standard will help alleviate this by providing guidelines for hierarchical structures and component instantiations. For instance, many books use various types of flip-flops as examples to model component instantiations (mostly because these are already understood by the readers). However, in practice, it is generally poor coding style to instantiate logic by mapping each register to various kinds of flip-flops. This can lead to longer, more obfuscating logic that does not take advantage of the ability to write in VHDL and Verilog at a higher level.

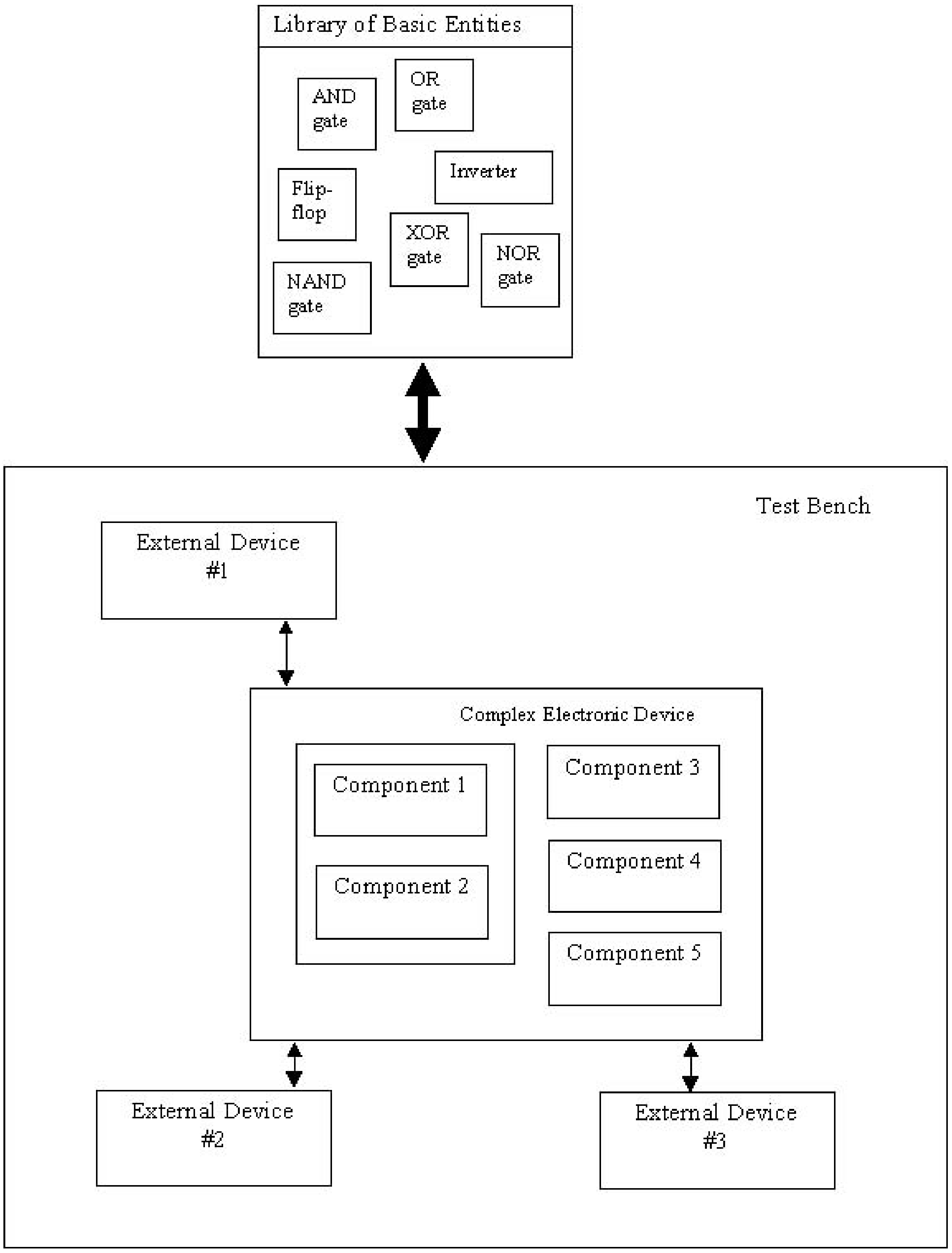


Figure 10: General HDL Development

Table 4: VHDL vs. Verilog

	VHDL	Verilog
Similarity to software programming language	Pascal and Ada	C
Level of abstraction	VHDL models well from the system level down to the RTL level, with some modeling at the gate level.	Verilog has less system modeling capabilities than VHDL, but more capabilities at the gate level.
Compilation	Allows separate compilation of multiple design-units (entity/architecture pairs) that reside in the same system file.	With Verilog, care must be taken with both the compilation order of code written in a single file and the compilation order of multiple files. Simulation results can change by simply changing the order of compilation.
Data types	VHDL has a multitude of language or user defined data types that can be used. As a result, dedicated conversion functions are needed to convert objects from one type to another.	Verilog data types are very simple, easy to use, and very much geared towards modeling hardware structure as opposed to abstract hardware modeling. All data types used in a Verilog model are defined by the Verilog language and not by the user.
Design reusability	Procedures and functions may be placed in a package so that they are available to any design-unit that wishes to use them.	There is no concept of packages in Verilog. Functions and procedures used within a model must be defined in the module.
Ease of learning	VHDL may seem less intuitive at first for two primary reasons. First, it is very strongly typed; a feature that makes it robust and powerful for the advanced user after a longer learning phase. Second, there are many ways to model the same circuit, especially those with large hierarchical structures.	Probably easiest to learn with no prior exposure or knowledge.
High level constructs	VHDL contains more constructs and features for high-level modeling than Verilog. Abstract data types can be used along with the following statements: package statements for model reuse configuration statements for configuring design structure generate statements for replicating structure generic statements for generic models that can be individually characterized	Verilog has no high-level modeling statements similar to VHDLs. Verilog allows you to parameterize models by overloading parameter constants.
Language extensions	VHDL allows architectures and subprograms to be modeled in another language by using the “foreign” attribute.	The Programming Language Interface (PLI) is an interface mechanism between Verilog models and Verilog software tools.

	VHDL	Verilog
Libraries	VHDL uses a library to store compiled entities, architectures, packages and configurations. Useful for managing multiple design projects	There is no concept of a library in Verilog. This is due to its origins as an interpretive language.
Low level constructs	Simple two input logical operators are built into the language, they are: NOT, AND, OR, NAND, NOR, XOR and XNOR. Any timing must be separately specified.	The Verilog language was originally developed with gate level modeling in mind and has very good constructs for modeling at this level and for modeling the cell primitives of ASIC and FPGA libraries.
Managing large designs	Configuration, generate, generic, and package statements all help manage large design structures.	There are no statements in Verilog that help manage large designs.
Operators	Similar operators to Verilog with the addition of a mod operator.	Similar operators to VHDL with the addition of a unary reduction operator.
Parameterizable models	A specific bit width model can be instantiated from a generic n-bit model using the generic statement.	A specific bit model can be instantiated from a generic n-bit model using overloaded parameter values.
Procedures and tasks	Allows concurrent procedure calls.	Does not allow concurrent task calls.
Readability (This is more a matter of coding style and experience than language feature)	VHDL is a concise and verbose language; its roots are based on Ada.	Verilog is more like C because its constructs are based approximately 50% on C and 50% on Ada.
Structural replication	The generate statement replicates a number of instances of the same design-unit or some subpart of a design and connects it appropriately.	There is no equivalent to the generate statement in Verilog.
Test harnesses	VHDL has generic and configuration statements that are useful in test harnesses.	Verilog does not have similar statements.
Verboseness	Because VHDL is a very strongly typed language, models must be coded precisely with defined and matching data types. Models are often more verbose, and the code often longer, than its Verilog equivalent.	Verilog allows signals representing objects of different bit-widths to be assigned to each other. The signal representing the smaller number of bits is automatically padded out to that of the larger number of bits. This has the advantage of not needing to model quite as explicitly as in VHDL, but does mean unintended modeling errors will not be identified by an analyzer.

The important elements of a HDL coding standard are:

- Consistent and defined style
- Guidelines on writing understandable code
- Commenting guidelines
- Information to capture in comments at each level
- Naming convention (for consistency)

Coding standards can be specific to a programming language or a chip family, corporate requirements, or can be more general in nature. An example coding standard, courtesy of Xilinx Corporation, can be found in Appendix B.

5.6 PROGRAMMING EXAMPLE

The below examples demonstrate, in a simple form, the programming constructs for complex electronics.

VHDL uses the concept of a "design entity," which consists of two design units. The entity declaration defines the external interface. The architecture body details the internal structure, and can define the entity's behavior, structure, or both.

Verilog uses the concept of a "module" rather than "entity." Like VHDL, the port declarations (external interface) are separate from the module body, which defines the internal behavior and/or structure. Figure 11, which includes Examples 1 and 2, shows the difference between a VHDL and Verilog design for the same circuit.

5.7 SYNTHESIS

Design synthesis is the process that takes the higher-level designs and optimally translates them to a gate-level design which can be mapped to the logic blocks in a complex electronic device. It is during synthesis that timing and area constraints can be specified by the user. Unlike software, which executes sequentially, the elements of a complex electronic chip will execute in parallel, with specific timing requirements. However, in general, synthesis is a form of compiling - translating the readable language into instructions that are implemented in the integrated circuit.

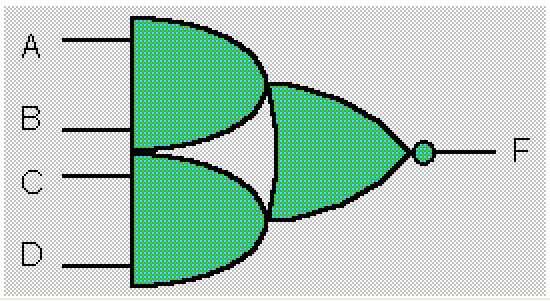
The synthesis step transforms the behavioral and structural specifications into an optimized netlist of gates. The netlist is a description of the various logic gates in the design and how they are interconnected. During synthesis, the designer can optimize parameters and constraints in the final chip. For example, a certain amount of delay may be necessary when accessing an outside element like a sensor. This delay can be included as a constraint during the synthesis process. Other constraints may be power consumption and signal timing.

Example 1: And-or-invert (AOI) gate

Description: This gate takes two sets of signals, each of which is ANDed together, ORs the resulting signals, and finally inverts the results.

Truth table (not complete):

A	B	A&B	C	D	C&D	OR	Result (F)
1	1	1	1	1	1	1	0
0	1	0	1	1	1	1	0
1	0	0	1	1	1	1	0
1	1	1	0	1	0	1	0
1	1	1	1	0	0	1	0
1	1	1	0	0	0	1	0
1	0	0	1	0	0	0	1
0	0	0	0	0	0	0	1
0	0	0	1	1	1	1	0



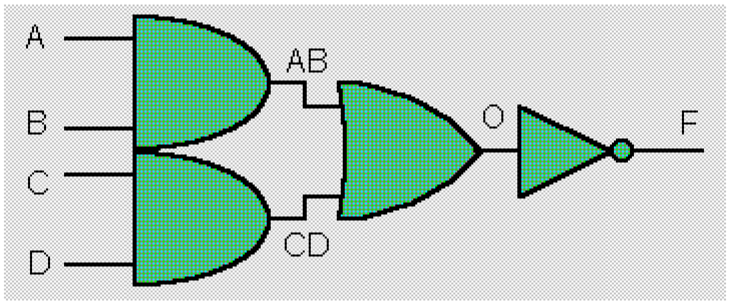
VDHL Code for AOI gate	Verilog Code for AOI gate

-- VHDL code for AND-OR-INVERT gate
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity AOI is port (A, B, C, D: in STD_LOGIC; F : out STD_LOGIC);
end AOI;
architecture V1 of AOI is
begin
  F <= not ((A and B) or (C and D));
end V1; -- end of VHDL code
	// Verilog code for AND-OR-INVERT gate
	module AOI (A, B, C, D, F);
	input A, B, C, D;
	output F;
	assign F = ~((A & B) | (C & D));
	endmodule // end of Verilog code

Explanation of VHDL code	Explanation of Verilog code
Comments begin with "--" library/use: provides the entity with access to all the names declared in the package STD_LOGIC_1164. "entity AOI is" - starts the entity description and assigns it the name AOI. "port {..}" specifies the input and output signals and the data type of each "end AOI;" terminates the entity declaration "architecture V1;" gives a label (V1) to the architecture body and connects it to the AOI entity declaration "begin" starts the architecture statement (the details). " F <= not ((A and B) or (C and D));"specifies the behavior of the signals "end V1'" terminates the architecture body.	Comments begin with "//" "module AOI (A, B, C, D, F);" - starts the module description and assigns the module a name (AOI). "input A, B, C, D; output F;" - declares which of the signals are inputs and which are outputs. "assign F = ~((A & B) | (C & D));" - logic statement defining the behavior of the signals. The concurrent assignment executes whenever one of the four ports A, B, C, or D change value. The ~, & and | symbols represent the bit-wise not, and/or operators, respectively. "endmodule" - terminates the module definition.

Example 2: AOI Gate with internal signals and timing

Description: This gate is the same as the first example, except that the internal signals (between the AND, OR, and NOT gates) are explicitly identified. Additionally, timing delays are included in this example.



VHDL code	Verilog code

library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity AOI is port (A, B, C, D: in STD_LOGIC;
        F : out STD_LOGIC);
end AOI;
architecture V2 of AOI is signal AB, CD, O: STD_LOGIC;
begin
AB <= A and B after 2 NS;
CD <= C and D after 2 NS;
O <= AB or CD after 2 NS;
F <= not O after 1 NS;
end V2;
	// Verilog code for AND-OR-INVERT gate
	module AOI (A, B, C, D, F);
	input A, B, C, D;
	output F;
	wire F; // the default
	wire AB, CD, O; // necessary assign
	AB = A & B;
	assign CD = C & D;
	assign O = AB | CD;
	assign F = ~O;
	endmodule // end of Verilog code

Explanation of VHDL code	Explanation of Verilog code
The majority of the statements are exactly the same as in the previous example. In the architecture body, the internal signal behavior is defined. Whenever a signal on the right side of the assignment (e.g., "A and B") is evaluated either A or B changes value. The signal on the left side of the assignment (e.g., "AB") is updated with the new result after a delay of 2 nanoseconds. In this example, if port A changed value, the result would propagate through the entity, to the final output, with a total delay of 5 nanoseconds.	In Verilog, a wire represents an electrical connection. A wire declaration looks like a port declaration, with a type (wire), an optional vector width, and a name or list of names. "wire AB, CD, O;" declares three wires (the internal signals). The assign statements (e.g., "assign AB = A & B;" are the same format as in the previous example. They break out the logic from one statement into several, using the internal signals (wires). These statements are independent and executed concurrently, and are not necessarily executed in the order in which they are written.

Figure 11: AOI VHDL to Verilog Comparison

Synthesis is performed almost exclusively by a software tool. Modern synthesis tools do an excellent job of optimizing complex designs, so designers do not need to manually perform that task. However, user input to the tools does have an effect on the output. For example, synthesis tools behave very differently given a common set of constraints. These timing-driven tools perform complex trade-offs to achieve the timing constraint specified, including adding extra parallel logic to paths where there is negative timing slack, or optimizing a critical path at the expense of a non-critical one. When you overconstrain a design, the tool sees many, many paths that do not meet timing and can generate lots of extra logic in a futile attempt to make all of them hit the timing goals. This can result in a much larger design with reduced overall timing performance. In a timing-driven tool the idea is to give the tool the real timing specifications, and let it work to meet that goal. Once that performance goal has been met, the tool will start optimizing for less area which translates to cost savings in your device. This can produce an even faster design because routing delays can be reduced by having less logic in non-timing-critical areas.

5.7.1 Simulation

Simulation is used in the design of complex electronics at several levels. One very nice aspect of hardware description languages is that they are "executable," and simulators that can run the code are very common. Simulators are usually part of the tool suite provided by the vendor of the complex electronic device (e.g., FPGA).

After design entry, the design is simulated at the register-transfer level (i.e., the HDL code). Simulation at this level is very fast, allowing the designer to implement many simulations to fully understand how the device will operate. Simulation can be used to help optimize the design and refine the logic, though designers need to be careful not to use it in an undisciplined code-and-fix mode. Simulation of the HDL code will look at signals and variables to check their value, trace functions and procedures, and will use breakpoints to check the status of the device at specific events. This process is very similar to using a software debugger. One caveat with simulation at this level of design is that some properties are not yet defined, such as timing and resource usage.

After design synthesis, but before physical implementation, functional simulation is used to help verify the design. The goal of functional simulation is to ensure that the logic of the design does what you want it to do, per the specification, and that it produces the correct results. This type of simulation is very important to get as many bugs out of the device as possible. If any errors are discovered, then the design entry step is revisited and necessary required changes are made, leading to a successful simulation.

After the design has been implemented, but before the device is actually programmed, a final simulation with full timing information is performed. The placement and routing process will determine any delays and other timing information, which are back-annotated to the gate-level netlist. This simulation is a much longer process, because of the timing aspects. A static timing analysis might be substituted for the timing simulation. Static timing analysis calculates the timing of combinational paths between registers and compares it against the designer's timing constraints.

5.7.2 Test Benches

Test benches for complex electronics are not made of wood or metal, but of Verilog or VHDL code. They are special programs designed to test your complex electronics design. While simulators can verify simple designs, more complex designs require a test bench to adequately verify the design.

A test bench is a HDL design you create which can load your circuit, apply stimulus to its inputs (including defining multiple clocks), and check the outputs for correctness. Because the test bench is a program you write, you have control over how your circuit is built and simulated. In addition to the above capabilities, a test bench can provide behavioral or structural models for everything on the PC board. In this way, it enables you to simulate the entire system including your complex electronics design(s) as well as external bus interfaces, external memories, etc. An engineer can design the test benches to automatically check important data conditions and to report any errors to a command window.

Comprehensive, upfront verification is critical to the success of a design project, and test benches should be created as you start to design your device. A HDL test bench/simulator can become your primary design development tool. When simulation is used right at the start of the project, you will have a much easier time with synthesis, and you will spend far less time re-running time-intensive processes, such as place-and-route tools and other synthesis-related software.

Test benches can be simple, applying a sequence of inputs to the circuit over time. They can also be quite complex, perhaps even reading test data from a disk file and writing test results to the screen and to a report file. A comprehensive test bench can, in fact, be more complex and lengthy (and take longer to develop) than the circuit being tested.

Depending on your needs (and whether timing information related to your target device technology is available), you may develop one or more test benches to:

- Verify the design functionally (with no delays).
- Check your assumptions about timing relationships (using estimates or unit delays).
- Simulate with annotated post-route timing information so you can verify that your circuit will operate in-system at speed.

A typical VHDL or Verilog test bench is composed of three main elements:

- **Stimulus Generator** - drives the unit under test with certain signal conditions (correct and incorrect values, minimum and maximum delays, fault conditions, etc.)
- **Unit Under Test** - represents the device undergoing test or verification
- **Verifier** - automatically checks and reports any errors encountered during the simulation run. Compares model responses with the expected results.

Test benches are created by human beings, often by the designer, and are subject to faults and failings like any human endeavor. If the logic of the test bench is incorrect, or if a particular stimulus is not defined, then the end result of the tests may not show an actual error. This scenario is something to keep in mind if you are called on to review verifications for a piece of complex electronics. You cannot assume that the test bench accurately and completely tested the device - especially if the device will be used in a safety-critical application.

5.7.3 Assurance Roles

At the design synthesis stage, assurance and safety engineers are usually not involved. A hardware assurance engineer might participate in or witness simulations of the device, or assess the test bench created for the simulation. The system safety engineer may review the simulation results.

Ideally, an assurance engineer should review the constraints used in the synthesis process and assess the simulations that are performed. The test bench should also be assessed to verify that it is correctly testing the device being created. All of these activities require a knowledgeable engineer who has experience with complex electronics.

A software engineer using this handbook should be able to follow along with any simulations that are performed, and be able to assess if the results match the interface specifications (e.g., if the output on a particular pin is within the valid range).

5.8 IMPLEMENTATION

Once a design has been created, simulated, and synthesized, the next step is implementation of the design in the particular complex electronic device. In software, implementation is usually translating the design into source code and compiling it. In complex electronics design, implementation is where the design meets the silicon - the higher-level design is converted into a chip layout.

The implementation process uses the tools supplied by the device (e.g., FPGA) vendor. The functions that were defined in the design have to be matched to the available blocks, gates, and other logic elements on the chip. Some basic steps in implementing a design are:

- Floorplan
- Translate
- Map
- Place and Route

The exact order of a step (or even the name a step/process is given) varies across different groups, companies, and documents. Thus, do not take the information in this handbook as the only correct way to do things. However, the concepts presented here are common across the industry and will be implemented to some extent in all programs - perhaps as part of an automated tool or under a different name. Being familiar with the concepts will help you "speak the language" when talking with a design engineer working with complex electronics.

Floorplanning is the process of identifying structures that should be placed close together, and allocating space for them. In designing complex electronics, there are multiple goals that must be met, and the goals often conflict. Finding the best balance between the various goals and requirements is something of an art. Some goals are:

- Minimize space on the chip (allows choice of less costly chips)
- Meet or exceed required performance
- Place everything close to everything else to minimize transmission time in the signal paths

Floorplanning does not have to be performed by the designer for many designs/chips. Most tool suites will perform this step as part of the automated sequence that takes the design and implements it in the chip. However, if you are creating an ASIC, need the absolute best timing possible, or are trying to cram a large design into a not-so-large chip, you will probably need to actively floorplan.

Done correctly, there are no negatives to floorplanning. However, if the floorplanning is done with insufficient regard for the architecture of the chip, then it is possible to actually do a worse job than the automated tool. It is also possible that there are constraints that are not well understood until placement is complete and routing commences.

As a general rule, data-path sections benefit most from floorplanning, and random logic, state machines and other non-structured logic can be safely left to the placer section of the place and route software. Data paths are typically the areas of your design where multiple bits are processed in parallel with each bit being modified the same way with some possible influence from adjacent bits. Example structures that make up data paths are adders, subtractors, counters, registers, and muxes.

Translation involves converting the results of the synthesis process to the format supported internally by the vendor's place-and-route tools. The incoming netlist is checked for adherence to design rules and is then optimized for the chip.

Translation may also be referred to as compilation or compiling. This process is automatic, but it takes some wading through the reports produced by the tool to verify that the translation/compile was correct. An intelligent post-processor, rather than the designer (or the quality assurance engineer), should find syntax and binding errors - otherwise you will have to do this for each design modification.

Mapping takes the logic blocks and determines what logic gates and interconnections on the device should be used to implement those blocks. During the mapping step, the functions within the device (such as counters, registers, or adders) are aligned with the logic resources of the chip. The exact process is device dependent. For example, FPGAs have look-up tables that perform logic operations. The mapping tool (part of the vendor's tool suite) collects the gates defined by the netlist into groups that will fit within the look-up tables.

Place and Route is the process of placing the logic blocks in the best spots on the chip to achieve efficient routing. Items that the place and route tool will look at include routing length (how far does a signal have to travel), track congestion (how many signals are coming into or out of an area), and path delays. While vendor-supplied tools usually perform the process automatically, the designer can specify some parameters and constraints that the final layout has to meet, including:

- the initial placement of the cells
- a position for each physical connector
- a form factor

5.8.1 Programming the Device

Once the design is successfully verified and found to meet timing and performance requirements, the final step is to actually program the device. At the completion of placement and routing, a binary programming file is created and used to configure the device. The process of programming is usually dependent on the type of memory used to store the device configuration and on the device type (e.g., FPGA or ASIC). Some of the factors in device programming are described below.

5.8.2 How Complex Electronic Devices Remember their Configuration

User-programmable complex electronic devices are a combination of a logic device and a memory device. The memory is used to store the pattern that was given to the chip during programming. The primary ways this information is stored are:

- Fuses
- Antifuses
- SRAM (static RAM)
- (E)EPROM cells (Electrically Erasable Programmable Read-Only Memory)
- Flash memory

A fuse is a special part of the programmable chip that is normally closed (connected) until an electrical current breaks that connection. Antifuses, unlike traditional fuses, are open until a voltage is applied to close (complete) the circuit path. Once closed, the connection cannot be opened. Programmable logic that uses fuses or antifuses are "program once" chips. For operations on Earth, fuses and antifuses lag behind the more reprogrammable versions in versatility and market share. In applications where ionizing radiation is a concern (such as outer space or high altitude), antifuses are usually a better choice.

SRAM, or static RAM, is a volatile type of memory. The contents of the memory are lost whenever the power is switched off. Static RAM differs from the dynamic RAM used in PCs in that memory refresh of the RAM is not required. SRAM-based programmable logic devices have to be programmed every time the chip is switched on. This is usually done automatically by another part of the system. SRAM FPGAs are susceptible to ionizing radiation, including the neutron radiation experienced at high altitudes.

An EPROM cell is a transistor that can be switched on by trapping an electric charge permanently on its gate electrode. This is done by an external programming device. The charge remains for many years and can only be removed by exposing the chip to strong ultraviolet light. EEPROM is electrically erasable PROM, which uses an electrical current rather than ultraviolet light to erase the programmed value. EPROMs have to be removed from their circuit boards to be erased and reprogrammed. EEPROMs can be erased and reprogrammed using special circuitry on the board.

Flash memory is non-volatile, which means that it retains its contents even when the power is switched off. It can be erased and reprogrammed as required. This makes it useful for programmable logic device memory. Flash-based devices combine the best of both worlds - maintaining configuration when not powered, but also allowing reprogramming when desired. Flash-based programmable devices are essentially immune to neutron radiation (generated when cosmic rays interact with the atmosphere) and are resistant to other high-energy particles.

5.8.2.1 Externally Programmed Devices

Complex electronics that use fuse, antifuse, or EPROM technology to configure the device have to be programmed in an external device, and cannot be programmed when placed on a circuit board. EEPROM-based devices may also require external programming, or may be able to be programmed in-system, depending on the specifics of the device and the circuit.

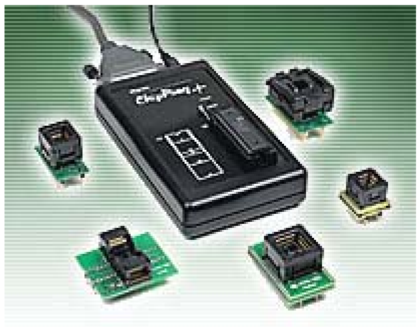


Figure 12: External Programmer

To use an external programmer, Figure 12, the chip (CPLD or FPGA, or simple programmable logic device) is placed in the appropriate socket and attached to the programming device. The programmer is attached to a computer (or may have an internal microprocessor, for stand-alone devices), which will download the binary file into the device and then apply the necessary voltages to “burn” or program the chip.

5.8.2.2 In-system Programming

Complex electronics that use SRAM, flash, or (sometimes) EEPROM can be, and usually are, programmed in-situ on the circuit board. Many boards provide a JTAG interface that can be hooked up to a personal computer for download of the device configuration.

5.8.2.3 ASICs

Application Specific Integrated Circuits are user-designable, not user-programmable complex electronics. While the basic steps in designing ASICs are the same as for other complex electronics, there are some differences, driven by the fact that ASICs are manufactured (usually in large runs), and a problem with the resulting chip is very costly.

Table 5 lists some of the main differences between creating an ASIC and a programmable complex electronic device (e.g., FPGA).

Table 5: FPGA vs. ASIC Comparison

Development Area	Differences
Vendor Selection	With FPGAs, you select which chip you will use. This is an off-the-shelf purchase, and the only question is whether the chip meets your needs and how good are the vendor-supplied tools. Since ASICs are manufactured, the vendor relationship is much more important. ASIC vendors will usually perform some of the implementation steps (such as place and route), as well as post-manufacture testing.
Careful Selection of Functionality	Because of the cost of failure with ASIC design, the selection of what functionality will be included in the ASIC is very important. While FPGA design, like software, can be changed later in the program, ASICs have a long lead time. So it is important to get everything right early in the process.
Simulation	With FPGA designs, the primary simulations are early in the design process, to verify the functionality of the design. With ASICs, simulations are mostly performed late in the design (at the gate level) to verify that all the last minute transformations and modifications do not cause an error. This difference affects what you want in a simulation environment. For ASICs, high performance (fast) simulation is essential. For FPGAs, the quality of the user environment and the speed in locating and fixing errors is more important.
Design Size	The size (in gates and/or I/O capacity) of ASICs is a somewhat continuous scale from small to very large. FPGAs are "chunky" - the size varies in vendor-defined increments within a device family. ASICs have more flexibility in size, so that a small increase will not affect the final cost too much, whereas with an FPGA you might have to go to the next higher size (and more expensive) chip. In general, FPGA designers are more concerned with getting the design to the minimum size (or to fit within the target chip) than ASIC designers.
Timing	ASICs have a relatively smooth, continuous distribution of delays as routing distances vary. With FPGAs, delays move in large, discontinuous, and relatively unpredictable steps. This means the estimated timing performance can vary by 20-30% on a net-by-net and path-by-path basis between the various design tools. With ASICs, many timing problems can often be conquered by resizing buffers, small placement and routing changes, and cell swaps. These options are not available with FPGAs. Logic synthesis in FPGA can basically only replicate logic, rebalance trees, and restructure paths to resolve timing issues.
Verification	In the ASIC world, verification is a long and time consuming process. It will take up to 70% of the total development time and resources. The reason for this is risk avoidance - you do not want a design error to slip through and cause you to waste all the time and money spent on the ASIC. This makes verification, confirmation, and re-verification every design engineer's first priority. Verification of ASICs is a lot more rigorous than FPGA verification. Besides multiple simulations at various design phases, and design reviews with your best engineers, two additional verification activities may be performed:

Development Area	Differences
	Prototyping in FPGA. Creating your ASIC design in an FPGA prototype results in discovery of bugs that may not have been identified during previous simulation. It also provides multiple platforms for software development in parallel to debugging. The FPGA prototype is available for the post-manufacture verification as an exerciser to validate the design. Formal Methods. Formal methods in the engineering world are those methods that use mathematical (formal) languages for writing specifications to prove that highest-level specifications are consistent with top-level objectives. They have an advantage over verbal prose and conventional simulators because they can be used to represent specifications that are provably consistent with objectives and higher level specifications.
Manufacturing	Programmable complex electronics use off-the-shelf chips, whereas an ASIC design is submitted to a vendor for manufacturing. The manufacturing stage incurs significant expense. The vendor assumes responsibility for fabricating, probing, and sorting wafers then assembles and packages the chip per requirements. Once the chips are created (and pass the vendor tests), the designer has to complete verification of the final product - and hope that the design was correct. Any problems found will require the ASICs to be remanufactured, at a significant cost.

5.8.2.4 SRAM-based FPGAs

Complex electronics that use SRAM will lose their memory once power is removed. Static RAM is volatile memory, thus SRAM chips need additional resources in order to function. Since the configuration is lost whenever the power is removed, the FPGA configuration has to be placed in non-volatile memory, such as an EPROM, EEPROM, or flash memory. When the FPGA is powered on, it reads the configuration from the non-volatile memory and is ready to go.

5.8.3 Assurance Roles

At the implementation stage, assurance and safety engineers are usually not involved. A hardware assurance engineer might witness the programming ("burning") of the device. Much of the implementation process is performed by automated tools, so if the tools were previously assessed, the results can be accepted without additional review. One area that the assurance disciplines can support at this time is verification that the design and implementation is appropriate for the environment where the device will operate. NASA experts in radiation or other space-related effects can be consulted if there are any questions about the device design.

A software engineer using this handbook should be able to witness the programming ("burning") of a complex electronic device and to understand the process.

5.9 VERIFICATION

As with software, verification activities do not wait until the complex electronic device is programmed and ready for test. Verification is a parallel set of activities to design and development. Various tasks are performed at each phase of the development.

This section of the handbook will answer the questions:

- What are the verification steps for complex electronics?
- How is verification for complex electronics similar to, and different from, software verification?
- Who performs the verifications?

5.9.1 Requirements

At the requirements phase, the system or subsystem level requirements are flowed down to the complex electronics. This flowdown is primarily the responsibility of the systems engineer, though the design engineer for the complex electronics should be involved to prevent requirements being imposed on the hardware that it cannot meet. Table 6 shows the verification activities done during the requirements phase.

Table 6: Requirement Verification Activities

Verification activity	Performed by
Evaluate requirements for the complex electronics	Quality assurance engineer, systems engineer
Safety assessment	System safety engineer
Requirements review (e.g., PDR)	All
Identification of applicable standards	Quality assurance engineer, safety engineer, design engineer
Formal methods	Knowledgeable practitioner

Quality assurance engineers should review the requirements for correctness, completeness, and consistency. Incomplete requirements are difficult to verify, are often interpreted differently by various people, and may not implement the functions that are desired. Finding out during testing that the device is missing important functionality, or is too slow, is something you really want to avoid.

For safety-critical or mission-critical devices, formal methods might be used as a verification tool. The requirements can be defined using a special language that allows mathematical proofs to be generated showing that the device will not violate certain properties. Formal methods can be applied at only the requirements level (to make sure you get those correct), or can be used to verify the design when it is generated. Most projects will not use formal methods.

5.9.2 Design Entry

During the design entry phase, the complex electronics functionality is defined in a hardware description language. The HDL code can be simulated in a test bench and its behavior can be observed. This is an important verification activity that is usually performed solely by the design engineer. Quality assurance engineers may review the simulation plans (if they are produced) or results, and for critical devices they may witness some of the simulation runs. Table 7 below shows the verification activities done during Design Entry.

Table 7: Design Entry Verification Activities

Verification activity	Performed by
Evaluate design (HDL) against requirements	Quality assurance engineer
Functional Simulation	Design engineer
Safety assessment	System safety engineer
Design review (e.g., CDR, peer review)	All
Static analysis of HDL code	Assurance engineer (including IV&V practitioners)

Functional simulation involves emulating the functionality of a device to determine that it is working per the specification and that it will produce correct results. This type of simulation is good at finding errors or bugs in the design. Functional simulation is also used after the design synthesis step where the gate-level design is simulated.

One or more engineers who can assess the design should review the HDL code. A good reviewer has to understand the system within which the device will operate, know the HDL language being used, and be able to compare what the device is designed to do against its requirements. This means that not just anyone can adequately review the design. Lack of knowledge or experience will hamper the review and often cause the designer to think the review is a waste of time.

For very complex or safety-critical devices, assurance engineers or Independent Verification and Validation (IV&V) practioners may be called in to review the design. One tool they can use is static analysis software for the HDL code, which can look for problems or possible errors in the code. This tool is very similar to some static analysis tools for software that look for potential logic or coding errors.

5.9.3 Design Synthesis

During design synthesis, the higher-level designs are optimally translated to a gate-level design, which can then be mapped to the logic blocks in a complex electronic device. It is during synthesis that timing and area constraints can be specified by the user. Table 8 shows the verification activities done during design synthesis.

Table 8: Design Synthesis Verification Activities

Verification activity	Performed by
Functional Simulation of gate-level circuit	Design engineer
Design review (peer review)	All
Design evaluation	Quality assurance engineer
Fault injection testing	Design engineer or quality assurance engineer

Simulation is one of the primary ways that the design synthesis process is verified. In almost all projects, the design engineer is the one who generates the test bench, defines the simulation runs, and performs the simulations. Quality assurance engineers are rarely involved, other than to perhaps verify that the simulations were performed. However, it is important to look at the design of the test bench and the simulation tests to make sure they are complete enough. This is the time to find errors or mistaken assumptions - not when you are integrating your complex electronics with other areas of the system.

Understanding how the complex electronics will operate when given invalid input is very important in verifying the devices. The real world is messy, and noisy signals or broken interfaced hardware are unfortunately common. Simulation is a great place to perform fault injection testing by inputting signals that are out of range, whose timing is not correct, that have ringing or other signal problems, or that are noisy. Encouraging this type of testing, and helping to identify the likely types of faults, is one way that quality assurance personnel can actively participate in the verification of complex electronics.

5.9.4 Implementation

Implementation is where the higher-level design is converted into a chip layout. The implementation process uses the tools supplied by the device (e.g., FPGA) vendor to match the functions that were defined in the design to the available blocks, gates, and other logic elements on the chip. Table 9 shows the verification activities done during Implementation.

Table 9: Implementation Verification Activities

Verification activity	Performed by
Timing simulation	Design engineer
Static timing analysis	Design engineer
Device programming	Witnessed by Quality Assurance engineer

Timing simulations are simply functional simulations with timing information. The timing information allows the designer to confirm that signals change in the correct timing relationship to each other. The timing information is entered in the hardware description language model file and then simulated. However, since there is a possibility of not being able to simulate all combination of inputs, a timing analysis tool can be used to evaluate a fully synchronous design.

Static timing analysis is a process that examines a synchronous design and determines its highest operating frequency. The analysis considers the path from every flip-flop in the design to every other flip-flop to which it is connected through the combinatorial logic. The analysis is usually performed by a software tool, which calculates the best case and the worst-case delays through these paths (critical-paths). Any paths that violate the set-up or hold-timing requirements of the flip-flop are flagged for later adjustment to meet the design requirements.

5.9.5 Testing

While simulation is used extensively in complex electronic design, testing the actual chip can sometimes be an eye-opening experience. Simulation involves assumptions and compromises that may not match with the real world. Testing the programmed chip - either independently or integrated onto a circuit board - is a necessary step in verifying your design. Table 10 shows the verification activities done during testing.

Table 10: Testing Verification Activities

Verification activity	Performed by
In-circuit functional and timing tests	Design engineer, may be witnessed by Quality Assurance engineers
Sub-system and system tests	All
Safety verification	All, but reviewed or witnessed by System Safety Engineer

In-circuit verification tests the functionality and timing of the design on the actual chip. Ideally, special test software running on a host computer will interface with the device under test through available test ports, such as the JTAG port. This process is similar to in-circuit emulators that run embedded software on the target processor and provide breakpoints and tracing into the actual software instructions.

The more common form of in-circuit tests is to manually run the complex electronics as part of a higher-level assembly to show that it meets all the specified requirements. This subsystem or system level test will show functionality at a black-box level, but will not provide a window into the internal functioning of the device.

If the complex electronic device is safety-critical, there will be separate safety verifications, usually at the system level.

5.9.6 What Should an Assurance Person Look for when Evaluating Complex Electronics?

So what can you do which will help improve the design process? At a minimum, you can ask questions of those producing or reviewing the design to help ensure that all of the important areas are considered. As a software assurance engineer, you may not be able to comment on the inner workings of the complex electronics, but you can certainly provide your process assurance viewpoint to the design and help make sure that defined processes are in place for configuration management, coding standards, and other areas. The following paragraphs provide some general guidance and questions to consider.

5.9.6.1 Programmatic Questions

- Is the design team experienced, or does it have at least one experienced member?
- Is there a design guideline document that defines design rules? How will the guideline help prevent a code-and-debug methodology as the design process?
- Has the team created a naming convention that provides information about objects and their timing information?
- Is there an exception handling mechanism (possibly a hardware-software cooperative arrangement) for error conditions that may be detected?

- Is the design maintained in a version control or configuration management system? Is there a formal process for changes once the design is baselined?
- Has anyone looked at what standards may be applicable (Center, NASA, other)?

5.9.6.2 Design Reviews

- Does the design meet the specification?
- Does the design pass a worst-case analysis (timing)?
- Is the design partitioned into logical components?
- Does the designer provide enough background information to understand what the device is supposed to do?
- Is there anything in the design that conflicts with other subsystem or system components?
- Do the design interfaces (input and output signals) match the interfaces as specified by the other components?
- Were the special pins on each device (e.g., mode pin on FPGA, JTAG pins, no-connect pins) verified that each is used properly?

5.9.6.3 Analyses

- Was a timing analysis performed with the following signals?
  - Pulse width of each clock, asynchronous set, clear, and load input
  - Setup and hold time for all clocked inputs
  - Recovery time for set and clear
- Did the timing analysis also consider the following?
  - Parallel clocking
  - Clock skew
  - Timing of analog circuitry
  - Minimum propagation delays
- Were the gate output drive capacities analyzed to determine that none were exceeded?
- Were the interfaces to other parts analyzed for input logic level thresholds and maximum input transition times?
- If there is a state machine, was it analyzed for:
  - Unused states and lock-up?
  - Simultaneous assertion of flip-flop sets and clears?
  - Reset conditions?
- Are resets of the correct assertion and release voltages, and is the pulse width correct?

CHAPTER 6. PROCESS ASSURANCE

6.1 PROCESS ASSURANCE OVERVIEW

According to IEEE, quality assurance is defined as "a planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements." Quality assurance (QA) can be broken down into two main areas: product assurance and process assurance.

Product assurance involves making sure that the final product meets its specifications. This is usually done thorough testing. Ideally, it also includes verifying that the requirements are correct, the design meets the requirements, and the implementation reflects the design.

Process assurance looks at the process used to create that final product. Was the development effort planned? Were the plans followed, or just put on the shelf and ignored? Does the development process meet any required standards? Are best practices used to develop the product? In process assurance, QA provides management with objective feedback regarding compliance to approved plans, procedures, standards, and analyses.

Process assurance activities are performed throughout the life cycle, including product conception, design, implementation, operation, and maintenance. Process assurance will detect, record, evaluate, approve, track, and resolve deviations from approved plans and procedures. For each life cycle phase, process assurance makes sure that planning is performed, that the plan is followed, and that the products of each phase are correct and complete. Note that verifying the quality of the requirements, design, and verifications are usually considered product assurance. This handbook includes them in process assurance because they are often overlooked when evaluating complex electronics.

For a circuit board that is assembled, product assurance would include verifying that the correct parts are on the board, assessing the quality of the soldering, and testing the board functionality. Process assurance activities would include verifying that the drawing used during the board assembly was configuration controlled and the correct revision, that proper Electrostatic Discharge (ESD) requirements were followed, and that an assembly process was defined and followed.

6.1.1 Why do Process Assurance?

While some aspects of process assurance are performed in many engineering disciplines, process assurance is the cornerstone of software assurance. In some industries, the main purpose of software quality assurance is to test the software prior to release. Within NASA, software assurance starts much earlier in the life cycle (with the requirements) and verifies the quality of all the products at each stage.

Why does software get this special treatment? Software differs from most hardware (mechanical or electrical) in several important ways:

- Software is complex and cannot be 100% tested. It is not feasible or sometimes even possible to test every possible path through the program, nor every combination of inputs.

For more than a trivial program, attempting such testing would take an astronomical amount of time.

- Software requirements are often fluid. Because software is easier to change than hardware, many defects or problems with hardware systems are overcome by changing the software.
- Software itself is fluid. It is easy to add additional functionality without sufficient thought as to the impact of that change on the entire system.

Using good practices to develop software increases the confidence in the quality of the software. Because you cannot fully test every combination of inputs and paths with software, you need a way to look at the whole development process and the test results and determine if the product is of sufficient quality. Process assurance is used to make sure those good practices are in place and that the project is following those practices.

Process assurance also looks at software throughout the life cycle and judges the quality of the process and the associated life cycle products. Software assurance engineers have a handle on the software requirements, design, and code volatility and can alert project management if too many changes are occurring. Because changes have an impact on other software or systems, software assurance engineers help identify and assess those impacts prior to the change being implemented. Process assurance is proactive in identifying and helping to correct potential problems before they become actual problems.

While some people may see process assurance as an unwanted but required activity, one of the main reasons to perform it is to embed quality throughout the life cycle. You do not want to wait until the product is finalized before you have any idea if it is a quality product or not. Process assurance provides insight into the development processes (and thus some insight into the quality of the product) long before the product is completed. This focus on problem prevention through early detection allows corrections and changes to be made to the product or process when the cost of those changes is less than it would be later in the project.

6.1.2 Process Assurance for Complex Electronics

When software cannot be fully tested, process assurance (how the product is built) is used to increase confidence in the resulting program. The same philosophy can be applied to complex programmable logic. In "Building a case for assurance from Process,"[1] the author shows how process assurance can be used in the IT security world to make a case for claims about the software quality. This is the idea behind the Software-CMM and other process improvement initiatives. If you cannot verify by testing every possible combination of inputs, decisions, etc., then knowing that you built the software according to well-defined standards gives additional confidence.

Complex electronics straddle the line between hardware and software. The design of these devices is complex enough that all combinations of inputs and timing cannot be fully tested. Complexity also increases the chance of design errors, unexpected interaction between elements

[1] "Building a case for assurance from Process," K. Ferraiolo, L. Gallagher, V. Thompson; 21st National Information Systems Security Conference

of the design, and other "software-like" errors. Because of these concerns, complex electronics cannot be completely verified using traditional approaches.

Adding process assurance to the verification of complex electronics will increase the confidence that the final device was designed to the correct requirements, the design completely implements all requirements, and the final product meets all functional and quality specifications.

The Federal Aviation Administration (FAA) is taking a similar approach to complex electronics. The document DO-254, "Design Assurance Guidance for Airborne Electronic Hardware," is basically process assurance for complex electronics. This document requires:

- Planning for all life cycle phases, including selection of design methodology, integration of hardware design processes with supporting processes, and description of process assurance policies and procedures.
- Activities performed by engineers at each life cycle phase, including requirements capture, design creation, implementation, and acceptance testing.
- Verification and validation throughout the life cycle.
- Configuration management of designs and supporting information for complex electronics.
- Process assurance activities at each life cycle phase.

6.1.3 Tools of the Process Assurance Trade

Process assurance is implemented primarily through the following activities:

- Documentation review
- Formal inspections, reviews, and walkthroughs
- Audits
- Analyses

The following paragraphs provide a quick overview of these processes. The next section of this handbook will go into more detail on which processes are appropriate for each phase of the life cycle, and what aspects of the complex electronics development they should be used for.

Documentation Review

Individual review of a document, design, or hardware description code is performed by the process assurance engineer. This type of review may or may not use a checklist (if one is available). The quality of the artifact is evaluated against best practices, and the results are fed back to the author of the artifact.

Reviews, Walkthroughs, and Formal Inspections

Formal inspection is an examination of the completed product of a particular stage of the development process (such as a design), typically employing checklists, expert inspectors, and a trained inspection moderator. The objective is to identify defects in the product. There are many techniques of doing inspections, but many follow the methods developed by Michael Fagan over 20 years ago.

Reviews are an alternative to formal inspections as a process assurance method. Informal design review methods are difficult to quantify since they are generally done at the discretion of the product author, do not follow a detailed process, and are not reported at the project level. Informal review is a valuable alternative if the more effective formal inspection is not used.

Walkthroughs are meetings in which the author of the product acts as presenter to proceed through the material in a stepwise manner. The objective is often raising and/or resolving design or implementation issues. Walkthroughs tend to be informal and lacking in close procedural control.

Audits

A process assurance audit is performed to determine the level of adherence to the project plans and procedures. Evaluation of the sufficiency or effectiveness of the procedures and plans is occasionally part of an audit, though normally the evaluation is performed when the procedures and plans are first produced. This type of audit examines a sampling of records to determine if procedures are being followed correctly.

Records can include formal products (e.g., official design document), informal development information, log files, tool output files, and even emails. Configuration management and change control records are also often examined during a process assurance audit.

Analyses

Analyses are performed when required to evaluate an aspect of the system, a project artifact, or the impact of changes. For complex electronics, the specific analyses will depend on the device, the level of criticality, safety implications, life cycle phase, and other factors. An analysis can be as simple as a documented "expert review" or as complex as a computer simulation. The method used in performing the analysis needs to be documented, as well as the results.

6.2 IDENTIFYING COMPLEX ELECTRONICS

This section explains how to recognize if a project is using complex electronics and how to determine if the programmable devices are simple versus complex.

6.2.1 Simple versus Complex

Simple electronics includes off-the-shelf integrated circuits from simple logic devices up to microprocessors. While the software that runs on microprocessors is complex, the device itself can be considered simple because it is a) well tested by the manufacturer and b) not programmed at the hardware level by the end user.

The dividing line between simple and complex electronics is not well defined, and has not been officially determined by NASA. Table 11 gives some guidelines to help make the determination.

Programmable devices used as part of a safety system or hazard control should be assumed to be complex. To be considered simple, a very strong case should be made with sufficient analysis and documentation to justify the position.

Table 11: Simple Complexity Guidelines

Simple	Off-the-shelf ICs Microprocessors PAL, GAL, PLA EPROM, EEPROM	These devices are either tested by the manufacturer or so simple that all inputs and outputs can be verified.
Gray area	CPLD	Depending on usage and size (gate count), CPLDs can be simple or complex.
Complex	FPGA System-on-Chip ASIC	These devices are too complex to be 100% tested.
Special Concerns	Distributed systems	Systems with one or more complex devices (or a complex electronic device and software) that jointly control a system or coordinate among themselves require assurance beyond the devices themselves. The interfaces and timing of communication are important to consider.
	Complex electronics as part of an off-the-shelf circuit board	Sometimes an FPGA or CPLD will be part of an off-the-shelf board. Since the design of the device is probably not available, you cannot perform any analysis or indepth verification of the device. If the device is not used for safety purposes, it can probably be considered simple.

6.2.2 How to Determine if Complex Electronics are being used in a Project

Here are some pointers to use when determining if the project includes complex electronics:

- Review project documentation. Look at the system concept, any overviews or descriptions, and system and subsystem requirements and design documents.
- Talk with the project system engineer and/or system safety engineer. The system engineer should be aware of any complex electronics. System safety engineers should be aware of any complex electronics that are part of a hazard control or otherwise safety-related.
- Talk with the project electrical engineer(s). These are the people who will develop the devices.

6.2.3 What Next?

If the project is using one or more complex electronic devices, the next step is to gather more information. Find out:

- What process is used for design of the devices?
- What tools are being used to design/develop/program the devices?
- Is configuration management (CM) used? What about change management?
- How will the devices be verified?

- How will quality assurance engineers be involved in verifying and assuring these devices?
- What is the error handling philosophy in the design? Are there ways that external signal problems (invalid voltages, missing signals, etc.) can cause problems with the device?
- Are the devices safety-related? Do they acquire or process any signals used in safety decisions (e.g., temperatures, voltages)?
- Is the function of the device mission-critical? Will failure seriously affect the ability of the system to carry out the mission?

If one or more of the devices are safety-related, share that information with the project system safety engineer. Safety-related complex electronics should be looked at by the system safety engineer in more depth.

If you see deficiencies in one or more areas (e.g., configuration management), you can research alternatives and make suggestions to the project manager or engineers on how to implement or improve the process. Configuration and change management are very important and often overlooked. You also want to guide the project away from a "program/debug/reprogram" paradigm, similar to undisciplined software development.

Be proactive. Get to know the device designers. Educate yourself on the devices, the tools used, and the design process. Do some web surfing for common errors with the devices, and make sure the designers have avoided them. Review the requirements for the device - are they clear and unambiguous? See if you can observe a simulation or two. Ask intelligent questions - ones that show that you are interested enough to have done some background work.

6.3 PROCESS ASSURANCE ACTIVITIES

Process assurance activities occur during all phases of a project life cycle. This section describes activities that are appropriate for complex electronics for each phase of the life cycle. Remember that there is currently no requirement for many of these activities, so implementing them on your project could require some negotiation. However, this information will help you apply your quality assurance expertise more thoroughly to complex electronics.

Quality assurance engineers need to possess sufficient domain knowledge to evaluate the completeness and correctness of complex electronics requirements and design. They must have the ability to determine whether the design has incorporated all requirements accurately. If you are not an electrical engineer, or do not have significant experience with complex electronics, you probably do not have that domain knowledge. For some process activities, you may wish to find an expert (either in the assurance arena or in engineering) to help you or to independently perform an analysis or evaluation. The most important aspect of assurance is evaluation by someone other than the designer, but not all evaluations have to be performed by the quality assurance engineer.

As you perform assurance activities on complex electronics, keep in mind some quality criteria. These criteria will help you judge the status of the product or process.

- Correctness. The extent to which a device fulfills its specifications.

- Efficiency. Use of resources; performance characteristics.
- Flexibility. Ease of making changes if required.
- Security. Protection of the device from unauthorized access.
- Interoperability. Effort required coupling the system to another system.
- Maintainability. Effort required locating and fixing a fault in the program within its operating environment.
- Portability. Effort required transferring a device or design (program) from one environment to another.
- Reliability. Ability not to fail, including in off-nominal environments.
- Testability. Ease of testing the device to ensure that it is error-free and meets its specification.

6.3.1 Project Conception

The initial stage of a project or system is the time when many decisions are made that will affect the project months or years down the road. While the technical decisions are driven by the results of systems engineering trade-off studies, the assurance decisions are driven by a combination of:

- Requirements and standards
  - What are the NASA, Center, and other quality assurance standards that the project must follow?
- Project management support.
  - The level of assurance is directly proportional to the amount of support that project management supplies. When quality assurance is perceived as a useful tool to help develop a functional system within the project constraints, quality assurance engineers are given adequate funds and personnel to do a thorough job. If the project manager deems quality assurance an annoyance, then the ability of the quality assurance engineer to implement an effective program is hampered.
- Effectiveness of the assurance organization
  - An assurance organization that has a track record of working with projects to develop tailored and effective assurance plans and processes will be more likely to gain project support in implementing new assurance activities. Conversely, an organization that does not have a good working relationship with projects will make it much more difficult for the assigned quality assurance engineer to persuade the project to consider any additional assurance activities for complex electronics.
- Knowledge and experience of the assurance professional
  - The assurance professional has to be proactive in implementing quality assurance activities, especially for new areas such as complex electronics. If the quality assurance engineer lacks knowledge and experience, the necessary assurance infrastructure may not be put in place.

Quality assurance is involved in project planning activities through:

- Creation of a Quality Assurance Plan that outlines the work that will be performed by the quality assurance engineer throughout the project life cycle.
- Assessment of the project plans, including the management and development plans for electronics, for completeness, correctness, and other quality attributes.
- Assurance that the project produces the required plans.

The plans a project will produce depend on the NASA and Center requirements and the project complexity and safety-criticality. The content of the plans often varies between projects, with one project combining several documents and others producing separate plans. Do not get hung up about which plan is which, but review the project plans for how they will address complex electronics. If they do not address the issues at all, try to encourage the project manager or the design engineer to at least informally document the information.

Here are some areas the project plans should address regarding complex electronics:

- What life cycle will be used to develop the complex electronics? How will the complex electronics life cycle interface with the project life cycle? In describing the life cycle, does the document discuss transition criteria between phases, and how to return to previous phases if problems are found?
- Are there standards that apply to the complex electronics? NASA currently has no defined standard.
- What is the hardware design process?
  - What activities will be performed as part of the process?
  - How will the hardware design process work with supporting processes, such as verification and assurance?
  - Is the design method for complex electronics defined and described?
  - What design environment (e.g., tools) will be used? What is the rationale for the selection?
- If deviation from established plans becomes necessary, what is the process for doing this? For example, how will changes be approved by all interested parties?
- How will the design for complex electronics and any associated data be included in the configuration management system?
  - What process is in place to review and approve any revisions to the design?
- Are the plans completed before the life cycle phase in which they will be used? Plans for configuration management should be finalized before development starts, for example.

The Quality Assurance Plan should include activities for reviewing the requirements and design, witnessing or performing testing, and other product verification steps. The plan should also include formal or informal audits to verify that the project is following the plans they defined. If a project plan is just gathering dust, it is important to look for the reason. Maybe the document is too high level. Maybe things have changed enough that the document is out of phase with project reality. Whatever the reason, try to work with the project to fix the document problems so that the plans are useful and relevant.

Risk management is an important tool that projects can use in reducing the probability or impact of risks. Complex electronics has some similarities to software, including the fluidity of the requirements, interface problems with other elements of the system, integration issues (often a result of the interface problems), and the need to create a complex program within a defined period of time. These types of issues are ideal for risk management mitigation.

6.3.2 Requirements

During the requirements development process, system requirements are allocated to various subsystems and parts, including complex electronics. These requirements need to be documented (in a separate specification for complex electronics or as part of another requirements specification document).

The process assurance activity is to review the requirements for the complex electronic devices and verify that they:

- Include all requirements that are appropriate for the complex electronics (i.e., that the allocation was complete)
- Identify any requirements that are safety-related
- Identify design constraints for the complex electronics
- Are clear, concise, and verifiable
- Are traceable to a higher level document or are noted as derived requirements

It is important that the requirements are as clear as possible, because many problems found later in system design can be traced back to ambiguous or incorrect requirements. Requirements for complex electronics should be more than just a cut-and-paste from the system requirements specification. They should be decomposed to the appropriate level of detail, and provide enough information that a designer can go off and create the device.

Activities for the verification of requirements for complex electronics must be specified in the verification plan. If a verification method cannot be determined, that indicates that the requirement is flawed and needs to be fixed.

6.3.3 Design Entry

During the design entry phase, the complex electronics functionality and structure are defined in a hardware description language (HDL). The HDL "program" is actually a model of the complex device, and can be run (simulated) and tested. This phase is when any problems with the requirements should be identified and the high-level functionality should be verified.

Prior to the start of the design, several process assurance activities should be performed:

- **Tools**. Review selected tools for applicability to the design process. Check the tool vendor web site and other sources for known tool defects or operational workarounds.
- **Design Process**. Make sure a disciplined design process is in place, and the design engineer is willing to follow it. Negotiate as necessary.

- **Configuration Managemen**t. Make sure the HDL code and other design information is configuration managed. The level of formality depends on status of design (e.g., informal version control prior to baseline, formal change control after baselining).
- **Design and Coding Standards**. Ensure that the design team is using a design and coding standard. This standard will define the basic design philosophy and specify aspects of the HDL program structure. Even if only one engineer is designing the device, a standard 1) helps ensure that the HDL program is understandable by others (and the design engineer, six months down the road) and 2) provides a way to capture and incorporate best practices in the design process.

A design and coding standard should include:

- Specific HDL coding features and methods that either should be used or should not be used.
- Design “best practices”, either as guidelines or as requirements.
- Naming conventions for modules, inputs, outputs, etc.
- Commenting rules that define what types of information to include in comments. One example would be to define a module header that includes comments on the module’s purpose and structure.
- Readability rules may be covered under naming and commenting conventions. But the standard should help guide the designer into creating HDL code that is readable by others.
- Modularization guidelines that provide information on how to decompose the high level design into individual modules.

Assessment of the HDL design can be performed in parallel with the design effort, with intermediate design elements being reviewed, if the project criticality warrants it. Otherwise, the review is normally performed after at least a fairly stable design (if not baselined) is created.

Process assurance activities post-HDL-design include:

- Ensure that the design is reviewed by someone who has enough knowledge to make an expert assessment. This can be another engineer, a quality assurance engineer, or even an outside expert. Another set of eyes will help spot problem areas of the design. This review could be part of a Formal Inspection or other peer review.
- Review the design (behavioral and structural specification in HDL) against the requirements. Are all requirements correctly and completely implemented?
- Trace the requirements into the design elements. The rigor of this tracing should be determined by the safety-criticality and mission-criticality of the device.
- Identify any derived requirements that emerge from the design process. Make sure the rationale for these requirements is captured.
- Review the design against the design and coding standard.
- Assess the design for unused functions.

- Assess the use of special pins on each device (e.g., mode pin on FPGA, JTAG pins, no-connect pins) and verify that each is used properly.
- Identify constraints (design, installation, operation) that could affect safety if not followed.
- Assess the simulations that were performed. Did they cover all the required functionality? Were all modules exercised?
- Verify that the processes defined in the project plans were followed.
- Assure that any design trade-offs done for speed, size, etc., are documented.

6.3.4 Design Synthesis

During design synthesis, the higher-level designs are translated to a gate-level design, which can then be mapped to the logic blocks in a complex electronic device. This step also optimizes the design to make the most efficient use of the target device. It is during synthesis that timing and area constraints can be specified by the user.

Process assurance activities at this phase are:

- Verify that the design process, as defined in the project plans, was followed.
- Verify that the tools specified in the previous phase are the ones that are being used.
- Verify that the configuration management system is being used as defined in the project plans.

Additional assurance activities require someone with expertise in complex electronics. They can be performed by the quality assurance engineer or by an engineer independent of the project. Additional assurance activities are:

- Evaluate the test bench that was created by the design engineer for adequate testing capability of the device design.
- Review the constraints specified by the design engineer (as input to the synthesis process) for reasonableness.
- Assess the simulations performed after design synthesis is completed. Did the addition of timing information affect the outcomes of the simulations? Did the simulations look at worst-case timing, including on incoming signals?

6.3.5 Implementation

During the implementation phase, the higher-level design is converted into a chip layout. The implementation process uses the tools supplied by the device vendor to match the functions that were defined in the design to the available blocks, gates, and other logic elements on the chip.

Automated tools perform much of the implementation process, so the assurance and safety engineers are usually not involved in any depth. Some process assurance activities at this phase are:

- Verify that the implementation process, as defined in the project plans, was followed.

- Verify that the tools specified in the project plans are the ones that are being used. Note any discrepancies and the rationale for using a different tool.
- Verify that the configuration management system is being used as defined in the project plans.
- Ensure that timing simulations or static timing analyses were performed.
- Verify that the simulations performed included out-of-range inputs, inputs that arrived in an incorrect order, and other "real world" problems that can be anticipated.
- Verify that the device is programmed according to a defined process and that the programming is witnessed by appropriate personnel (usually quality assurance).
- Verify that the interfaces to other parts were analyzed for input logic level thresholds and maximum input transition times.
- If there is a state machine, verify that it was analyzed for:
  - Unused states and lock-up
  - Simultaneous assertion of flip-flop sets and clears
  - Reset conditions

6.3.6 Testing

Once the device is programmed, it should be tested with other components. Initial testing may occur in a breadboard system, with final (acceptance) testing occurring in the real hardware system. This in-circuit verification tests the functionality and timing of the design on the actual chip.

The more common form of in-circuit tests is to manually run the complex electronics as part of a higher-level assembly to show that it meets all the specified requirements. This subsystem or system level tests will show functionality at a black-box level, but will not provide a window into the internal functioning of the device.

Process assurance activities for this phase include:

- Verify the defined processes are in place and are being followed correctly.
- Verify that the testing strategy has been documented in a plan and/or procedure, and that testing occurs according to the plan.
- Verify that the planned tests will completely verify the requirements in all reasonably expected situations. This includes verifying the functionality and performance in nominal situations and when other parts of the system have errors. How gracefully does the device handle errors it may encounter? How gracefully can it handle any internal faults?
- Verify that the planned tests will exercise all modules or other divisions in the device. Not every level of testing has to exercise all modules, but each module should be tested at some level (device, circuit board, subsystem, or system).

- Verify the planned tests exercise the device as close as possible to the functionality in the sequence and operations that the system will perform on mission with nominal and off-nominal conditions (i.e., test as you plan to operate).
- Review the test plans and procedures to identify any areas where testing is weak. You are looking for modules that are only minimally tested, requirements that are only verified under some circumstances, and other areas where additional testing may be helpful.
- Witness tests (as agreed to in the project plans) and document any anomalies and problems.
- Review the test results to verify that no unnoticed anomalies occurred. Sometimes during testing many events are occurring and an anomaly unrelated to the aspect of the particular test may be missed.

6.3.7 Operations and Maintenance

Once the system is operational, the role of process assurance is not over. While the original project assurance engineer may have moved on to another project, there should still be an assurance engineer maintaining a minimal role with the system.

Process assurance activities during operations and maintenance include:

- Review operational and maintenance procedures for inclusion of any workarounds or other information that was discovered during development and testing.
- Support any failure review boards and help assess any problems that are identified during operations.
- If the complex electronic device is to be reprogrammed, assess the impact of the changes on the device, the system, and operational procedures.

6.3.8 Metrics

During the development process it is important to know if you are developing a quality product. One way you do that is by collecting metrics during the various phases of the development cycle. A metric is defined as "a system of parameters or ways of quantitative and periodic assessment of a process that is to be measured and is usually specialized by the subject area." Metrics can be used to track trends, problems, productivity, and much more. With complex electronics, metrics must cover both the hardware and software portions of the development cycle.

There are two types of metrics used for measurement. They are called primitive and derived. Primitives are items such as time, number of problems, or lines of code, the base item we use to make a decision. A derived metric takes multiple primitives to determine a unit. A good example from the software world is errors per lines of code (errors per KLoc). The two primitives used in this measure are the number of software errors and the number of executable lines of code.

Let's look at some of the primitives available for measurement in the complex electronics development cycle, starting with the number of defects found. This measure can be broken down into when and what type of defect is found. When would specify the development phase or review. Defects could be categorized by type of defect. Examples of these are interface, requirement, and logic or data type.

6.3.9 Supporting Processes

Configuration Management (CM)

Configuration Management is, unfortunately, often not used for complex electronics design artifacts. The final design is usually saved, but the intermediate development artifacts are under the control of the designer. While formal configuration management might not be necessary until the design is finalized (baselined), some form of informal control (e.g., use of a version management system) is recommended. Being able to revert to a previous version of the design is useful when problems are discovered during development. Being able to recreate versions of the design might also be useful to help narrow down when a problem was introduced.

Once the design is baselined, formal configuration management should be applied to the design. CM includes change control. This means that a process is in place for any changes to be approved prior to the changes being implemented. Often a Configuration Control Board or an engineering board is used to review and approve (or disallow) the changes. Change control assures that:

- Changes to one part of the system do not adversely affect other parts of the system.
- The configuration of the device is always known (i.e., there are not unauthorized changes).
- Everyone who may be affected by the change has a chance to evaluate the change for impacts to their area of concern.

Reliability

Most reliability studies look at the hardware failure rates for the devices in a system. While failure of the actual device (e.g., FPGA) can be known, the failures related to design errors or unexpected interactions within the FPGA, once it is programmed, are not easy to determine. Most reliability evaluations ignore software for this very reason.

While there is currently no good way to predictively assess the reliability of a complex electronic design, the fact that there may be design errors should be considered by the reliability engineer. At a minimum, the confidence in the resulting numbers (mean-time-to-failure, system reliability) is lowered.

Maintenance and Maintainability

If the device will potentially need to be maintained (including reprogramming updates), this issue needs to be considered early in the design of the complex electronics and its supporting circuitry. Some areas to consider are:

- Will the device architecture allow for the types of enhancements that can be foreseen?
- Does the design specification provide the information that an engineer would need to understand how the product works?
- Is the HDL code readable?

- Are comments liberal and informative?
- Is the necessary physical infrastructure in place to allow reprogramming?
- Is access to the reprogramming port, if one is used, available when the system is installed?

CHAPTER 7. FUTURE TRENDS

7.1 CHANGES IN COMPLEX ELECTRONICS DESIGN AND VERIFICATION

Technology never stands still. Within the realm of complex electronics, devices such as System-on-Chip, FPGAs with embedded microprocessors, and reconfigurable computing all strain the traditional hardware-oriented design and verification approaches. Increasing complexity in designs also make it harder for the designer to conceptualize the design. Several new methods in design and verification of complex electronics will hopefully help improve verification of these devices.

7.1.1 Hardware/software Codesign and Coverification

Since complex electronics is increasingly being combined with software, codesign (and subsequently, coverification) of the hardware and software is a good idea. Hardware/software codesign is the cooperative design of hardware and software, within a single chip or within a system. One of the goals of codesign is to shorten the time-to-completion while reducing the design effort and costs of the designed products.

In hardware-software codesign, designers consider trade-off in the way hardware and software components of a system work together to exhibit a specified behavior, given a set of performance goals and technology. This trade-off between hardware and software illustrates the optimization aspect of the codesign problem. Codesign is an interdisciplinary activity, bringing concepts and ideas from different disciplines together (e.g., system-level modeling, hardware design and software design).

Current development methods for designing embedded systems and complex electronics require specification and design of the hardware and software as separate entities. A specification, often incomplete, is developed and sent to the hardware and software engineers. The hardware-software partition is decided early on in the project life cycle and is adhered to as much as is possible, because any changes in this partition may necessitate extensive redesign. Designers often strive to make everything fit in software, and off-load only some parts of the design to hardware to meet timing constraints.

The codesign process starts with specifying the system behavior at the system level. After this, the system specification is divided into a set of smaller pieces, so-called granules (e.g., basic blocks). Trade-off studies are performed to determine the most effective way to partition the functionality into hardware and software. The granules are mapped to hardware and software, resulting in sets of granules implemented on hardware (hardware parts) or software (software parts). Once the mapping is done, the implementation-independent system specification is decomposed into hardware and software specifications.

Hardware is synthesized from the given specification; the software specification is compiled for the chosen processor. The result of this co-synthesis phase is a set of complex electronics and a set of assembler programs for the processors. In a final co-simulation step, the complex electronics are simulated together with the processors executing their generated assembler programs. The results are iterated until a sufficient system implementation has been found.

The coverification problem in system-level design includes different methods to detect errors at different abstraction levels. Coverification methods include formal verification, simulation or emulation. Formal verification formally proves either the equivalence of different design representations or specific properties (e.g., the absence of dead-lock conditions of the system specification).

Simulation validates the functional correctness for a set of input stimuli. In most cases, only a small set of all combinations of input stimuli can be simulated. For this reason, simulation only ensures the correct behavior with a certain probability. Simulation can be applied during different design steps including the co-simulation step after co-synthesis.

To speed up the simulation time for simulating a partitioned hardware/software system, emulation is used. Emulation systems couple the complex electronics (either the programmable devices or, for ASICs, a programmable equivalent) with processors on a board. Therefore, emulators are the closest representation of real prototypes that is possible.

7.1.2 System Modeling

Hardware description languages (HDLs) allow you to model the system at various levels of abstraction. However, they are still fairly “low level” abstractions, representing the hardware aspects of the design. Several new modeling languages, and extensions to existing languages, allow higher-level modeling of the system.

The purpose behind higher-level modeling is to:

- Keep the design at a level of abstraction that human minds can grasp. Complex designs make it difficult for a human to understand both the device and how it interacts with its environment.
- Verify the design at a high level, and then allow tools to generate the low-level design.
- Model the complex electronics as part of a larger system that includes software and possibly biological constructs.

Researchers and industry are developing system modeling languages or language extensions for use in complex systems. There are two parts to a system design language: the ability to express ideas in a natural language and a component that can translate the functions into working architectural components. Here are two areas of language development that are being actively pursued:

- Using C or C++ to model the system. One product, SystemC, provides hardware-oriented constructs within the context of C++ as a class library implemented in standard C++. It can be used from initial concept to implementation in hardware and software. SystemC provides an interoperable modeling platform, which enables the development and exchange of very fast system-level C++ models. It also provides a stable platform for development of system-level tools.
- SystemVerilog is a new standard, enhancing Verilog so that it provides built-in support for a wide range of modern design and verification methodologies. SystemVerilog is an extension to the Verilog language, which enables the modeling and verification of systems at a high level of abstraction. It adds a significant set of language enhancements

on top of the Verilog 2001 standard, including features for high-level, abstract system modeling, test bench automation, and the integration of Verilog with the C programming language.

- MATLAB and Simulink can be used to model systems. MATLAB is a high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis, and numerical computation. Simulink is a platform for multi-domain simulation and model-based design of dynamic systems. Simulink provides an interactive graphical environment and a customizable set of block libraries that let you accurately design, simulate, implement, and test time-varying systems, including control systems, signal processing, and communications.

7.2 INTO THE NOT SO DISTANT FUTURE

What kinds of new devices and concepts are being considered? Below are a few of the new technologies being explored.

7.2.1 In-field or reconfigurable SoC

Most SoC designs use what is called a platform-based solution, where standard components like a microprocessor core make up a significant portion of the SoC. Custom devices provide further functionality. Some of those devices may be user-configurable (e.g., if a small FPGA or CPLD is part of the System-on-chip device), others may be designer-chosen only. These types of SoCs are usually implemented as ASICs.

A reconfigurable SoC provides the same kind of custom support except that the devices and peripherals are implemented using a reconfigurable matrix. The software must set up the hardware before it can be used. But from that point on, the platform-based SoC software and reconfigurable SoC software will be very similar, assuming that the microprocessor core is the same or similar and the functionality of the peripherals has the same characteristics.

With reconfigurable SoC designs, the hardware functionality can be changed simply by altering the code that performs system initialization. So, SoC could contain an analog-to-digital converter for one application, and then be reconfigured for a digital-to-analog converter, or even a totally different peripheral such as a network device, for another application. Some elements of the reconfiguration can be performed at a later time (after the basic hardware is initialized), allowing software applications to reconfigure devices. Applications that deal with multiple hardware codecs (e.g., streaming multimedia) or encryption methods, for example, could configure devices to the specific codec or encryption method being used at the time, then reconfigure for another codec or method when required for a different data stream.

7.2.2 FPGA microprocessors/systems

Some SoC devices are implemented entirely on programmable logic, in particular on FPGAs. Most reconfigurable SoCs fall into this category. However, reconfigurable SoCs use a fixed microprocessor with reconfigurable peripheral devices. What if you could change your microprocessor by just reprogramming the FPGA? What if you could customize the microprocessor for your application, then change it when that application changes? That is what the FPGA microprocessor systems offer.

FPGAs have proven themselves capable of handling a wide variety of tasks, from relatively simple control functions to more complex, algorithmic operations. While the time and cost advantages over designing custom ASIC hardware for such functions is well accepted, the advantages of using FPGAs over traditional processors and DSPs for software-oriented applications have been less clear-cut. This is due in large part to a long-standing disconnect between hardware and software development tools and disciplines.

Recent advances in software-oriented design tools for FPGAs, however, have combined with the ongoing increase in device densities to create a new environment for software developers, one in which the FPGA can be viewed as one possible target (along with traditional and non-traditional processor architectures) for a software compiler. Tools now available can help software engineers make use of FPGA platforms, as well as help these developers take advantage of the high level of algorithmic parallelism that is available when traditional processors (or processor cores) and FPGAs are combined in a single target platform.

FPGA-based computing platforms, particularly those with embedded “soft” microprocessors, have the potential to implement extreme high-performance applications without the upfront risk of creating custom fixed function hardware. Further, by using the latest generation of hardware/software co-design tools it is now possible to use multiple graphical, software-oriented design methods as part of the FPGA design process.

7.2.3 Reconfigurable computing

Someday, perhaps in the not-too-distant future, the computer at your desk may contain a typical microprocessor along with an array of reconfigurable, reprogrammable devices (FPGAs or their successors). Or, the microprocessor may be totally replaced by the FPGAs. As a user, the only thing you’ll note is that your software runs faster, allowing you to get your work done more quickly.

Typical computer systems use a single microprocessor that executes instructions sequentially. They are adaptable and configurable - you can write any kind of operating system or run any sort of application on a microprocessor. However, these systems trade speed for that adaptability.

If you have a fixed set of applications and really need more processing speed, you want an ASIC designed to meet your needs. While you can gain significant improvement in speed, you lose the ability to change the processor/ASIC uses outside of a narrow range of applications. The ASIC speed increase over general-purpose microprocessors comes from a combination of optimization for the specific purpose and the ability to perform processes in parallel.

What if you want speed and adaptability? To gain speed, you need to move from the serial processing paradigm to parallel processing. One way to do this is to use multiple processors, each performing operations in parallel. Another way is through reconfigurable computing. Both of these methods keep the adaptability component, allowing the user, through software, to run a wide variety of applications.

To have reconfigurable computing (RC), you need to have hardware that can be reconfigured to implement specific functionality. RC systems contain programmable hardware and may be combined with traditional microprocessors in order to take advantage of the strengths of each device. RC has been used in applications ranging from embedded systems to high performance computing.

Reconfigurable computing uses in-situ reconfigurable FPGAs as computing devices to accelerate operations which otherwise would be performed by software. The FPGA can be programmed with a digital circuit that implements the function to be performed, such as a fast square root operation. The processor can then access this function, as if it were in its own instruction set. When the processor needs another function, such as multiplying two numbers, the FPGA can be reprogrammed for that function.

To make this all work, the FPGA must be capable of being reconfigured quickly and allow only parts of the device to be reprogrammed. Reconfiguration has to be fast, or you quickly eat up the speed advantage you gain from moving the functions from the microprocessor to dedicated hardware. You would also lose too much time if the FPGA had to be entirely reprogrammed when you just want to change part of it. Fortunately, modern FPGAs are up to the challenge.

Reconfigurable computers already exist commercially. Early reconfigurable computers were expensive complicated monolithic FPGA arrays, but most modern commercial and research systems have evolved into relatively less expensive workstation accelerators. Research efforts in academic institutions are considering the establishment and management of parallel reconfigurable computing clusters and high-throughput networks of reconfigurable computers (NORCs). All these individual efforts are creating a new direction - reconfigurable supercomputing.

7.3 NASA ASSURANCE CHANGES

Currently, within NASA, complex electronics are treated as hardware devices. The design of complex electronics may be reviewed by quality assurance engineers, the assembly into a board or system is witnessed and/or verified by quality assurance, and the final resulting electronic device is tested. However, the complex nature of these devices requires additional assurance effort beyond that given to an off-the-shelf component. Hardware quality assurance personnel may not be fully cognizant of the functions, potential problems, and issues with these devices.

At NASA Headquarters, this assurance problem is being discussed and debated. What types of assurance activities should be applied to complex electronics? Who should be involved in the assurance of these devices? What competencies are necessary to provide adequate assurance of complex electronic devices?

The Federal Aviation Administration (FAA) faced similar concerns several years ago. They discovered that software functions were being implemented in FPGAs to avoid having to follow the FAA software assurance standard (DO-178B). The FAA struggled with the problem and finally came up with a standard for Complex Electronic Hardware (CEH) that is similar to the FAA software assurance standard. CEH includes the complex electronic devices discussed in this handbook and some additional devices. The resulting standard, DO-254, “Design Assurance Guidance for Airborne Electronic Hardware”, provides guidelines on the use of process assurance for complex electronic hardware.

NASA is reviewing the FAA approach of implementing process assurance. Software is a very complex entity that cannot be fully tested. In the software world, process assurance (evaluating how the product is built) is used to increase confidence in the resulting program. The same philosophy can be applied to complex electronics. If you cannot verify by testing every possible combination of inputs, decisions, etc., then knowing that you built the device according to well-defined standards gives additional confidence in its quality.

Process assurance will look at all life cycle stages of complex electronics development, from requirements to operations. Process assurance for complex electronics is very similar to the process part of software assurance, where we verify that the software development process was planned and the plan was followed, where requirements are reviewed and evaluated, the software design is evaluated against the requirements, code may be inspected or reviewed, and finally the resulting software is verified against the requirements. For hardware, the same types of activities are performed.

As a quality assurance engineer, you may be wondering what your role may be in the future. Since quality assurance encompasses process assurance, quality assurance engineers are well versed in the ideas and concepts. What is lacking is the knowledge to assess complex electronics. In order to effectively carry out assurance duties for complex electronic hardware, a quality assurance engineer must understand 1) the hardware itself, 2) the process and language used to design the device and 3) how and when to apply software-style assurance techniques to the device.

This handbook is one step in educating NASA software and quality assurance and system safety engineers on the design and verification of complex electronics. By itself, this handbook will not make you an expert able to perform assurance of the devices. The goal of this handbook is to present you with a broad understanding of complex electronics and the benefits and drawbacks/issues that need to be discussed and understood by the whole project team. In addition, this handbook was designed to provide you with the knowledge you need to better apply quality product and process assurance to these devices.

APPENDIX A EXAMPLES

A.1 CPLD

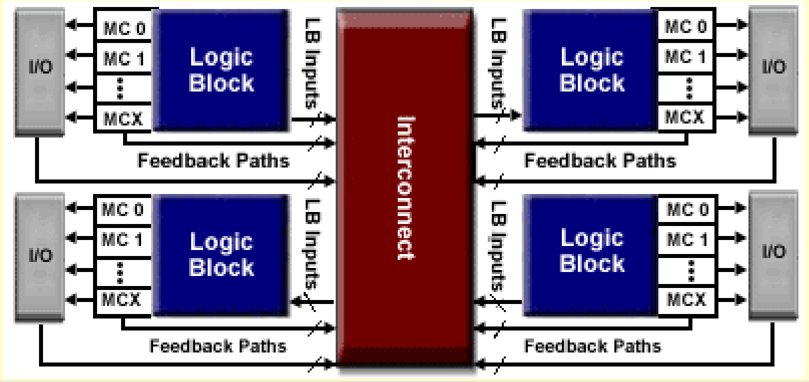
Device name:	Complex Programmable Logic Device (CPLD)
	 I/O MC 0 MC 1 MCX Logic Block LB Inputs Interconnect Feedback Paths

Figure A-1 CPLD

Description:

A CPLD contains a set of simpler Programmable Logic Device (PLD) blocks whose inputs and outputs are connected together by a global interconnection matrix. So a CPLD has two levels of programmability: each PLD block can be programmed, and then the interconnections between the PLDs can be programmed. A key feature of the CPLD architecture is the arrangement of logic cells on the periphery of a central shared routing resource. CPLDs are equivalent to about 50 typical PLD devices, and can replace thousands, or even hundreds of thousands, of logic gates.

Programming and Reprogramming

CPLDs vary in how they can be programmed or reprogrammed, depending on their underlying structure. The three basic types of CPLDs are:

- Fuse or anti-fuse. These CPLDs are programmed by passing a large current through the connections (fuses). The current “blows” the fuse to break a connection. The CPLDs are one-time programmable because you cannot rewire them internally once the fuses are blown. Programming occurs in a special device external to the circuit board the CPLD will be placed on.
- EPROM or EEPROM. In these CPLDs, the interconnections are made with transistors that are opened or closed by storing a charge on their gate electrodes using a high-voltage pulse. For EPROM-like CPLDs, you erase the CPLD and then place it in a special programmer socket and reprogram it. Reprogramming is not possible once the chip is soldered to its circuit board. EEPROM-like CPLDs may be reprogrammable on the circuit board, if special circuitry is included.
- SRAM or Flash. Static RAM (SRAM) or Flash can be used to control the transistors for each interconnection. Each memory bit controls the interconnect switches through its value. When a bit is set to ‘1,’ the switch is closed, and the logic elements are connected.

A '0' opens the switch. CPLDs built using RAM/Flash switches can be reprogrammed without removing them from the circuit board and are in-circuit reconfigurable or in-circuit programmable.

To figure out what switches to open or close to implement your logic design, there are tools available that take a logic design and output a binary file which configures the switches in a CPLD.

Applications:

CPLDs are used in a wide variety of applications from cell phones to spacecraft. They are often used as "glue logic" to connect various parts of a design, massage and process data, or to translate data from one protocol to another. CPLDs are great for:

- high speed operations
- interface controllers (bus, memory, Flash)
- interface bridging
- I/O expansion
- device configuration
- power-up sequencing
- microprocessor support logic
- glue logic
- implementing small "soft" microcontrollers (e.g., 8-bit)

They are used as support chips in most modern electronics, including:

- Cell phones
- PDAs
- Digital cameras
- Communications hardware
- GPS

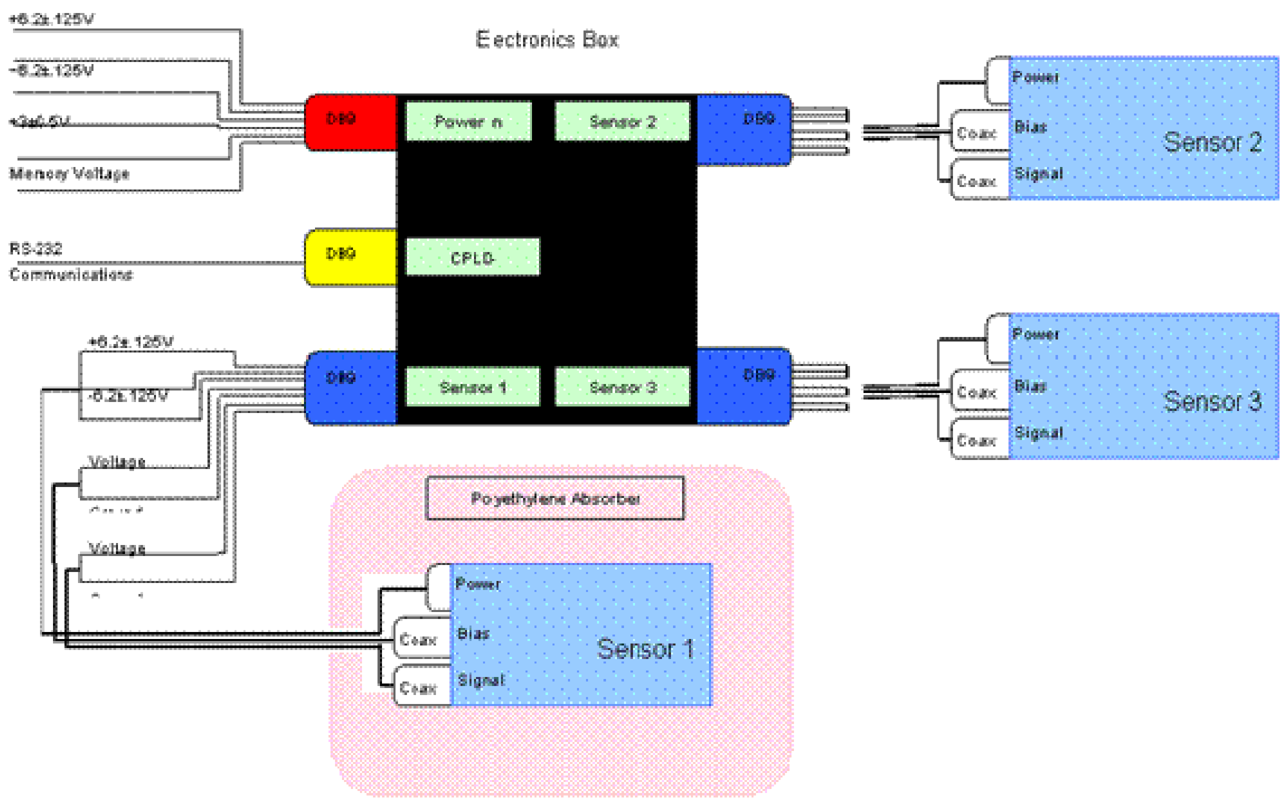
CPLDs come in a variety of density, speed, and package options. Handheld applications tend to use lower density devices, because they have less need for complex logic, require low power, and try to minimize cost per unit. When capability is more important than power usage, higher density CPLDs are a better choice.

Often a logic design could be implemented in either a CPLD or an FPGA. CPLDs are chosen when predictable timing performance is required. CPLDs have fewer routing matrices than FPGAs. Since each routing matrix adds a little delay to the signal, fewer routings translates to faster signal transit. While CPLD density is less than most FPGAs, high end CPLDs will have same density as low end FPGAs. Performance of CPLDs is usually better than FPGAs, though it depends on the vendor, size (number of cells), speed, and other factors.

Real-world Examples:

Here are some examples of CPLDs used in a variety of products.

MicroDosimeter Instrument (MIDN)



MIDN was a space payload flight tested a compact, low powered, and portable solid-state micro dosimeter. MIDN collected quantitative information on the dose and dose distribution of energy deposited in silicon cells that are tissue-sized. By inference, this data could show what the dosage would be in living tissue.

CPLDs were used in MIDN for command and data handling. This payload was part of the MidSTAR-1 (Midshipman Space Technology Applications Research) satellite which was in operation from 2007-2009.

Fluids and Combustion Facility (FCF)		
	NASA • Lewis Research Center • μg • Fluids & Combustion Facility International Space Station	

The Fluids and Combustion Facility (FCF), a permanent modular, multi-user facility to accommodate microgravity science experiments onboard the International Space Station (ISS) U.S. Laboratory Module, was first activated in 2009. FCF uses the Fluids Integration Rack (FIR) and the Combustion Integration Rack (CIR) to support research in fluid physics and combustion science. The FIR will permit a wide range of fluid investigations from microscopic imaging to particle tracking. CIR experiments look at how solid, liquid, and gaseous fuels burn in microgravity, fire prevention and suppression, pollutant and particulate formation, and combustion efficiency.

CPLDs are used within FCF to translate data from a digital camera to a high-speed fiber interface. When the data is received, two other CPLDs reformat the incoming data to what is required by a Digital Signal Processor (DSP).

A.2 FPGA

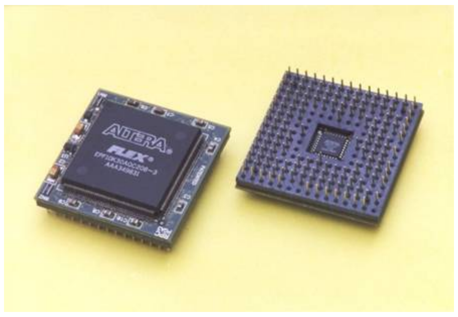
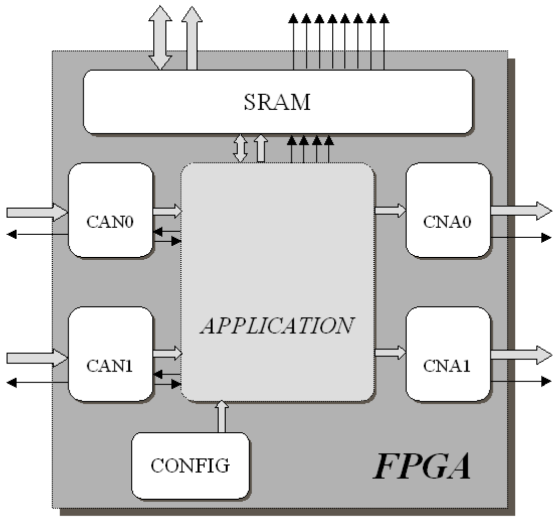
Device name:	**Field Programmable Gate Array (FPGA)**
	 SRAM CAN0 CAN1 APPLICATION CNA0 CNA1 CONFIG FPGA

Figure A-2 FGPA

Description:

A FPGA is a collection of simple, configurable logic blocks arranged in an array with interspersed switches that can rearrange the interconnections between the logic blocks. Each logic block is individually programmed to perform a logic function (such as AND, OR, XOR, etc.) and then the switches are programmed to connect the blocks so that the complete logic functions are implemented.

The interconnections for the logic blocks are programmable switches. FPGAs may use EEPROM, SRAM, antifuse, or Flash technology to store the programming. In most larger FPGAs, the configuration is volatile and must be re-loaded into the device whenever power is applied or different functionality is required.

Initially, FPGAs had only local and global routing resources (i.e., a logic block could only connect to adjacent logic blocks or to global networks). Newer FPGAs have multilevel routing hierarchies, so logic blocks can connect to different levels. Fortunately, the design software takes care of these complex issues.

Newer FPGAs are being developed that contain fixed functionality, as well as traditional programmable logic. FPGAs may contain a FIFO, arithmetic functions, memory, chip-to-chip transceivers, digital signal processor (DSP), or even an entire bus interface or microprocessor core. FPGAs with fixed functionality are cousins to the SoC devices that included programmable logic as part of their design.

SRAM FPGAs

SRAM, or static RAM, is a volatile type of memory. The contents of the memory are lost whenever the power is switched off. Static RAM differs from the dynamic RAM used in PCs in that memory refresh of the RAM is not required. SRAM-based programmable logic devices, such as FPGAs, have to be programmed every time the chip is switched on. This is usually done automatically by another part of the system.

Most SRAM-based FPGAs use a master mode, where they read the configuration information from non-volatile memory, such as a serial or parallel EPROM or flash memory. The FPGAs can also be configured via an external source in slave mode. The FPGA accepts a serial or parallel data stream that represents the configuration data. The source of the data can be a processor, computer, or an FPGA that is acting as a master. Using this technique, it is possible for several FPGAs to be programmed from a single memory. A master FPGA is wired to a daisy chain of slave FPGAs. When the master FPGA has been programmed, it will keep reading the data from the memory and pass it on to the slave devices until all of the FPGAs are configured.

Antifuse FPGAs

A fuse is a special part of the programmable chip that is normally closed (connected) until an electrical current breaks that connection. Antifuses, unlike traditional fuses, are open until a voltage is applied to close (complete) the circuit path. Once programmed closed, the connection cannot be reprogrammed to open. Programmable logic that uses fuses or antifuses are “program once” chips.

Antifuse FPGAs are best used when you do not want to have to reconfigure your chip every time power is applied (e.g., if you need a quick power-on time). They are also useful in environments where SRAM would have problems (e.g., high altitude or outer space).

Flash FPGAs

Flash memory is non-volatile, which means that it retains its contents even when the power is switched off. It can be erased and reprogrammed as required. This makes it useful for programmable logic device memory. Flash-based devices combine the best of both worlds - maintaining configuration when not powered, but also allowing reprogramming when desired. Flash-based programmable devices are essentially immune to neutron radiation (generated when cosmic rays interact with the atmosphere) and are resistant to other high-energy particles.

Software Engineers and FPGAs

What if a software engineer could create a regular software application that could run on an FPGA? Design tools for FPGAs are moving quickly in this direction. In this new environment for software developers, the FPGA can be viewed as one possible target (along with traditional and non-traditional processor architectures) for a software compiler. With currently available tools, the software engineer can make use of FPGA platforms, as well as take advantage of the high level of algorithmic parallelism that is available when traditional processors (or processor cores) and FPGAs are combined in a single target platform.

FPGA-based computing platforms, particularly those with embedded “soft” microprocessors, have the potential to implement extreme high-performance applications. With the latest generation of hardware/software codesign tools it is now possible to use multiple graphical, software-oriented design methods as part of the FPGA design process.

Radiation and FPGAs

NASA projects typically deal with environments more extreme than an office or laboratory. Spacecraft and high-altitude aircraft are bombarded with radiation. Shock and vibration, electromagnetic interference, and thermal issues are common problems when designing NASA systems.

Unfortunately, FPGAs are mostly just big RAM devices, and most of that RAM is in the configuration circuitry. An upset event in the routing can quietly alter the logical interconnections, and a problem in a lookup table (LUT) can alter the functional behavior of a design.

SRAM FPGAs are susceptible to ionizing radiation, including the neutron radiation experienced at high altitudes. SRAM FPGA designed for high-radiation environments typically use periodic read-back and verification of the configuration or frequent reconfiguration of the chip to a known good state. Because SRAM devices are vulnerable, they are used more in payload applications, where some level of failure can be tolerated and overcome, than in the more critical systems that control spacecraft flight operations.

While antifuse FPGAs lag behind the more programmable versions in size (gate density), versatility and market share, they are very useful in space applications. Radiation tolerant FPGAs use the antifuse technology, which provides immunity to radiation effects as well as low power, single-chip solutions that do not require configuration circuitry.

Flash-based FPGAs provide radiation tolerance along with reprogrammability. Like antifuse FPGAs, they are immune to upsets caused by most radiation. Like SRAM FPGAs, they can be reprogrammed in-circuit. Radiation studies of flash-based FPGAs are still ongoing.

While high-profile projects like the Mars rovers showcase the use of programmable logic in space, the majority of space-bound FPGAs are included in commercial and military satellites. FPGAs are frequently used in satellite functions such as guidance, station-keeping, and telemetry.

Applications

FPGAs had an initial niche as prototypes for ASIC. Because ASICs require a long lead time from design to implementation, and it can be very expensive to correct ASIC design errors, FPGAs were used to try out the designs. Errors detected in the design could be corrected, the FPGA reprogrammed, and testing of the design could continue. The process is not without problems, though. ASIC designs had to be created using ASIC synthesis tools, then a separate FPGA tool is used to implement the ASIC prototype in an FPGA. Switching from one synthesis tool to another requires changing code and scripts, which is time-consuming, and increases the potential for introducing errors into the prototype that do not accurately reflect the functionality of the ASIC design. FPGAs are often slower than ASICs, which prevents timing problems from being accurately diagnosed. Despite the problems, however, FPGAs are still used to prototype ASICs - because the cost of a failed ASIC can be quite expensive.

FPGAs have gained rapid acceptance and growth over the past decade because they can be applied to a very wide range of applications. Some typical applications are:

- random logic
- integrating multiple CPLDs
- device controllers
- bus controllers
- communication encoding and filtering
- small to medium sized systems with SRAM blocks

More intensive applications include:

- Digital signal processing
- Complex custom applications
- Consumer electronics
- Software radio
- Cryptographic and security devices

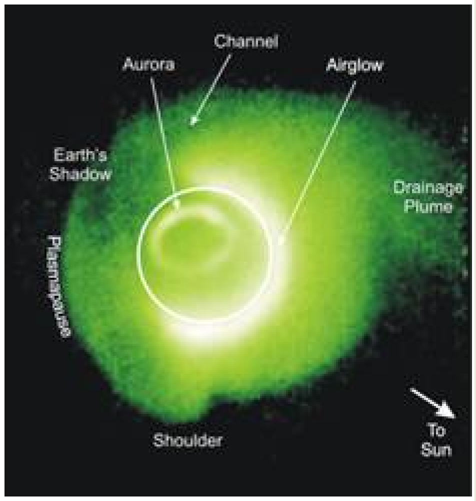
Reconfigurable or adaptive computing is a cutting-edge application for FPGAs. Instead of a traditional microprocessor that executes software, FPGAs are reprogrammed to perform the necessary calculations or operations.

<u>NASA (and other) Examples:</u>

The Mars Exploration Rovers (MERs), Spirit and Opportunity, garnered the world's attention as they rolled out onto the surface of Mars. Hidden inside the rovers and landers are FPGAs, doing their job in a harsh environment. FPGAs are used in pyrotechnics devices for landing, as well as in the arm, cameras, steering, antenna gimbals, and wheel control systems on the Mars rover missions.

Here are some other space and science projects that use FPGAs:

Cassini	
	There are FPGAs orbiting Saturn on the Cassini spacecraft, launched in 1997. Cassini has completed its primary mission and its first extended mission and is now on its second extended mission, through 2017. FPGAs are used in many instruments on Cassini, including the Visual and Infrared Mapping Spectrometer (VIMS).

Extreme Ultraviolet Imager (EUV)	
 Channel Aurora Airglow Earth's Shadow Drainage Plume Plasmapause Shoulder To Sun	FPGAs controlled parts of the EUV instrument on the IMAGE (Imager for Magnetopause-to-Aurora Global Exploration) satellite, part of NASA's MIDEX program. IMAGE was launched in 2000 on a two-year mission, but continued to provide data into 2005. FPGAs controlled the sensors and read out, formatted, and stored the data.
Optus C1	
	Radiation tolerant FPGAs have been deployed on board Optus C1, the largest hybrid commercial and defense communications satellite ever launched. The communications satellite was launched in 2003 and is still operational.

- A prototype multi-directional muon detector, operating in Sao Martinho, Brazil, was upgraded and extended, using FPGAs. The FPGAs allow a more complicated and advanced logical circuit to be designed at a reduced cost. The upgraded detector will be able to determine the incident direction of every single muon detected and record the count rates in the total 121 incident directions. The detector is part of a network used to forecast geomagnetic storms.
- NASA's Jet Propulsion Laboratory has developed a lossless image-compression algorithm that can be implemented entirely in an FPGA plus a small random-access memory chip.

Other missions that include FPGAs:

- Civilian/Scientific exploration:
  - Deep Space 1
  - Mars Pathfinder, Surveyor, Express, Climate Orbiter

   - Lunar Prospector
   - SIRTF (Space Infrared Telescope Facility, renamed the Spitzer Space Telescope)
   - TDRS (Tracking and Data Relay Satellite)
   - Hubble Space Telescope
   - GOES (Geostationary Operational Environmental Satellite)
- International Missions
   - International Space Station
   - Chandra
   - Rosetta
   - SOHO (Solar and Heliospheric Observatory)
- Commercial Satellites
   - Telstar
   - PanAm Sat
   - Intelstat IX
   - Globalstar
   - Orbview
- Military Satellites
   - Clementine
   - HESSI (High Energy Solar Spectroscopic Imager)
   - Mighty Sat
   - SBIRS-High (-Low) (Space Based Infrared System)
- Launch Vehicles
   - Ariane
   - Atlas
   - Delta
   - EELV (evolved expendable launch vehicle)
   - SeaLaunch

A.3 ASIC

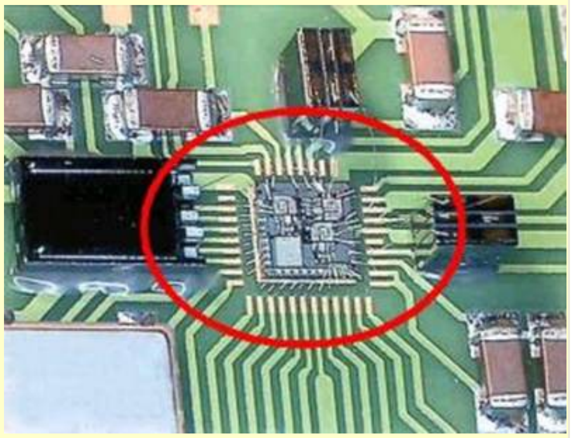
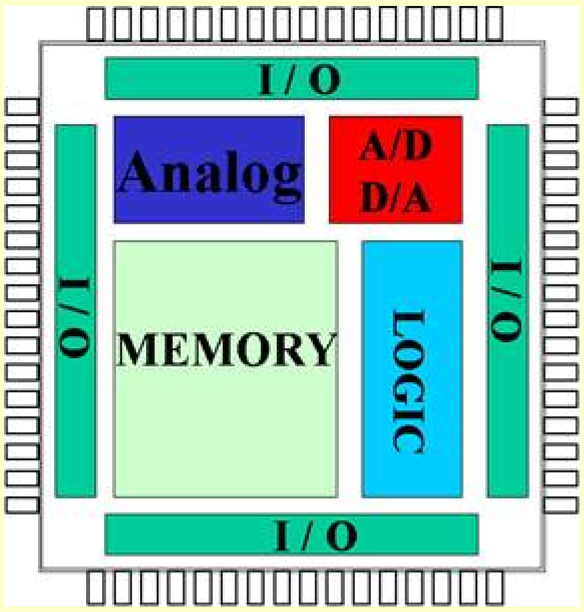
Device name:	**Application Specific Integrated Circuit (ASIC)**
	 I / O Analog A/D D/A I / O MEMORY LOGIC I / O I / O

Figure A-3 ASIC

Description:

An ASIC is an integrated circuit designed to perform a particular function by defining the interconnection of a set of basic circuit building blocks drawn from a library provided by the circuit manufacturer. They are built by connecting existing circuit building blocks in new ways. Since the building blocks already exist in a library, it is much easier to produce a new ASIC than to design a new chip from scratch.

ASICs are custom-designed integrated circuits, but they are not programmable by the user. They are manufactured (usually in large quantities) by vendors according to the design provided by the customer. If you find a problem with an ASIC after it is produced, the only option is to remanufacture (re-spin) the chip with a corrected device. To avoid costly mistakes, FPGAs are often used to check out and debug the ASIC design prior to submittal to the manufacturer.

While most integrated circuits (ICs) could be considered “application-specific,” because they have a defined purpose, off-the-shelf parts are not really ASICs. They are not designed by the user/customer to incorporate just the required functionality. Examples of ICs that are not ASICs include standard parts such as memory chips (ROMs, DRAM, and SRAM), microprocessors, and all the miscellaneous chips that are used in modern electronics (FIFOs, logic chips, drivers, clock chips, switches, etc.). Now, if a chip has been designed specifically for a talking toy, a cell phone, or a satellite, it is an ASIC. As a general rule, if you can find it in a data book, then it is probably not an ASIC.

Integrated circuits are made on a thin (a few hundred microns thick), circular silicon wafer, with each wafer holding hundreds of die. The transistors and wiring are made from many layers built on top of one another. Each successive layer has a pattern that is defined using a mask similar to a glass photographic slide. The first layers define the transistors, and the last layers define the metal wires between the transistors (the interconnections).

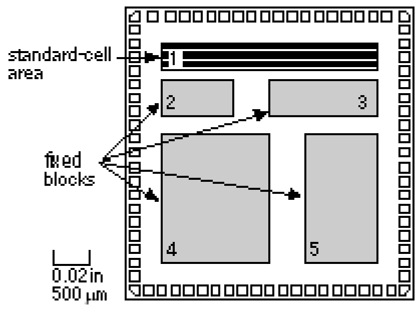
ASICs come in two basic varieties, full-custom and semi-custom, which consist of two sub-types: cell-based and gate-array. Each variety or type of ASIC has strengths and weaknesses. A microprocessor is an example of a full-custom ASIC, where each micron on the silicon is customized to give exactly what is needed. Semi-custom ASICs have pre-designed elements and customizable portions.

A full-custom ASIC allows customization of some (and possibly all) logic cells and all mask layers. Customizing all of the ASIC features in this way allows designers to include analog circuits, optimized memory cells, or mechanical structures on an IC, for example. Full-custom ASICs are the most expensive to design and manufacture. The manufacturing lead time (how long it takes to make an ASIC once the design is completed) is typically eight weeks.

Semi-custom ASICs have all of the logic cells pre-designed and some (possibly all) of the mask layers are customized. Designers use pre-designed cells from a cell library, provided by the vendor or a third party. These pre-designed units are usually referred to as IP (Intellectual Property). Semi-custom ASICs are either standard cell-based ASICs or gate-array-based ASICs.

A cell-based ASIC uses pre-designed logic cells (e.g., AND gates, OR gates, multiplexers, and flip-flops) known as standard cells. The standard-cell areas (also called flexible blocks) are built of rows of standard cells like a wall built of bricks. The standard-cell areas may be used in combination with larger pre-designed cells, such as microcontrollers, known as megacells. Megacells are also called megafunctions, full-custom blocks, system-level macros (SLMs), fixed blocks, cores, or Functional Standard Blocks (FSBs).

The ASIC designer defines only the placement of the standard cells and interconnect in a cell-based ASIC. However, the standard cells can be placed anywhere on the silicon; this means that all the mask layers are customized and are unique to a particular customer. The advantage of cell-based ASIC is that designers save time, money, and reduce risk by using a pre-designed, pre-tested, and pre-characterized standard-cell library. In addition each standard cell can be optimized individually.



If you were to look through a low-powered microscope at a cell-based ASIC die, you would see something similar to this figure. This ASIC has a single standard-cell area (a flexible block) together with four fixed blocks. The small squares around the edge of the die are bonding pads that are connected to the pins of the ASIC package.

Figure A-4 ASIC Die

In gate-array-based ASICs, the transistors are predefined on the silicon wafer. This predefined pattern of transistors on a gate array is called the base array, and the smallest element that is replicated to make the base array is called the base cell. Only the top few layers of metal, which define the interconnect between transistors, are defined by the designer using custom masks.

The designer chooses from a gate-array library of pre-designed and pre-characterized logic cells or macros. The reason for this is that the base-cell layout is the same for each logic cell, and only the interconnect (inside cells and between cells) is customized. Gate-array ASICs can be prefabricated up to a point and stored. At a later time, the final customization steps can be performed to complete the ASIC. This reduces the manufacturing time to only a few days or at most a couple of weeks.

ASIC Cell Libraries

The cell library is the key part of ASIC design. Cell libraries can be provided by the ASIC vendor, procured from a third-party library vendor, or custom-built. The first choice, using an ASIC-vendor library, requires you to use a set of design tools approved by the ASIC vendor to enter and simulate your design. An ASIC vendor library is normally a phantom library - the cells are empty boxes, or phantoms, but contain enough information for layout. After you complete layout you hand off a netlist to the ASIC vendor, who fills in the empty boxes (phantom instantiation) before manufacturing your chip.

The second and third choices require you to make a buy-or-build decision. If you complete an ASIC design using a cell library that you bought, you also own the masks (the tooling) that are used to manufacture your ASIC. This is called customer-owned tooling (COT, pronounced "see-oh-tee"). A library vendor normally develops a cell library using information about a process supplied by an ASIC foundry. An ASIC foundry (in contrast to an ASIC vendor) only provides manufacturing, with no design help. If the cell library meets the foundry specifications, it is considered to be a qualified cell library. These cell libraries are normally expensive (possibly several hundred thousand dollars), but if a library is qualified at several foundries this allows you to shop around for the most attractive terms. This means that buying an expensive library can be cheaper in the long run than the other solutions for high-volume production.

The third choice is to develop a cell library in-house. Many large computer and electronics companies make this choice. Most of the cell libraries designed today are still developed in-house despite the fact that the process of library development is complex and very expensive.

However created, each cell in an ASIC cell library must contain the following:

- A physical layout
- A behavioral model
- A Verilog/VHDL model
- A detailed timing model
- A test strategy
- A circuit schematic
- A cell icon

- A wire-load model
- A routing model

Applications:

ASICs are used widely in many types of electronics devices. Any time there are a large number of devices manufactured that require specialized operation, you will probably find an ASIC inside.

ASICs Application Examples:

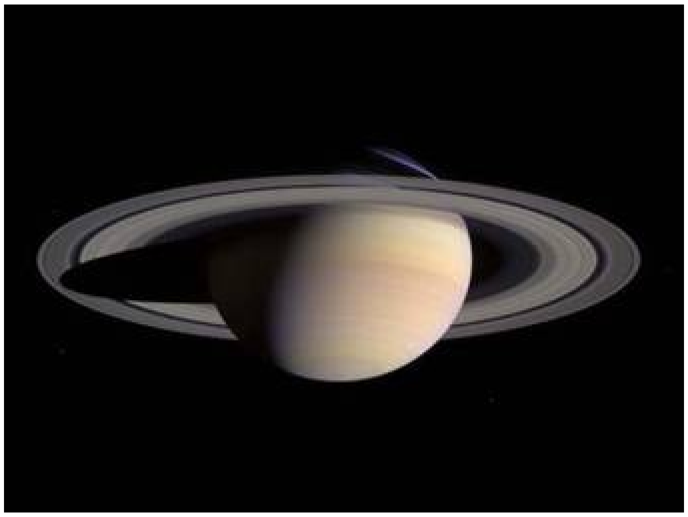
- Battery management for household appliances
- Low noise audio circuit
- Analog Integrated Circuit for industrial environment
- Sensitive photo transistors and opto-sensors
- DC-DC converters from 0.9V supply voltage
- Control circuit for cycle rear light
- 120V Linear Regulator
- Interface circuit for a bar code reader
- Control and evaluation circuit for motion detectors
- Timer electronics
- Interface and signal processing electronics for sensors (light, vibration and magnetic field)
- Control circuit for mobile phones
- Automotive control functions
- PDAs.

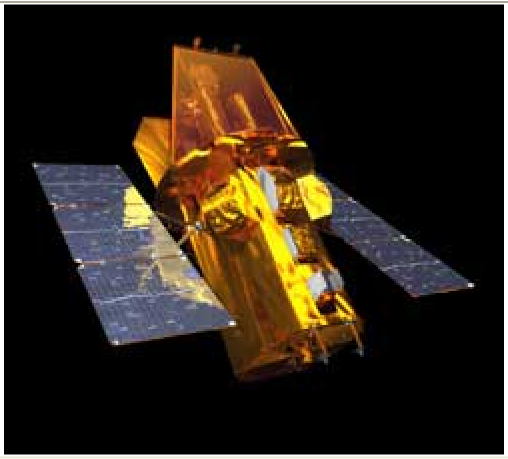
NASA Examples:

ASICs can provide several features that are especially important in spacecraft and instruments, such as:

- Customized electronics
- Smaller footprint
- Less weight
- Hard-coded (radiation resistant)

The smaller footprint on the circuit board and reduced weight are the result of including multiple functions in a single chip, rather than having to use many individual integrated circuit chips.

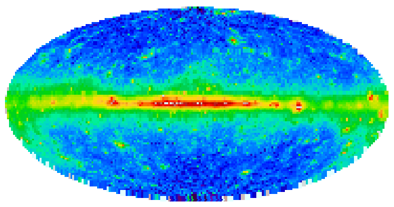
Cassini	
	
See section A.2 also. The Cassini spacecraft is a complicated system, with 22,000 wire connections and nearly nine miles of cabling. The main on-board computer uses very high-speed ICs and advanced, radiation-hardened ASICs. Each ASIC replaces one hundred or more traditional chips, allowing the development of a data system for Cassini that is ten times more efficient than earlier spacecraft designs (e.g., Galileo and Magellan), but at less than one-third the mass and volume. Mars Pathfinder and Near Earth Asteroid Rendezvous (NEAR) both used these chips directly off the Cassini production line. The Cassini program also created an advanced solid-state power switch that eliminates the rapid fluctuations (called transients) usually found in circuits utilizing conventional power switches. This power switch combined the switching attributes of the Metal-Oxide Semiconductor Field-Effect Transistor (MOS FET) with an ASIC design. This ASIC results in significantly improved component lifetime and efficiency and is widely applicable to both industrial and consumer electric and electronic products.	

Swift	
Swift is a first-of-its-kind multi-wavelength observatory dedicated to the study of gamma-	

ray burst (GRB) science and launched in 2004. Its three instruments work together to observe GRBs and afterglows in the gamma ray, X-ray, ultraviolet, and optical wavebands. The main mission objectives for Swift are to:

- Determine the origin of gamma-ray bursts
- Classify gamma-ray bursts and search for new types
- Determine how the blastwave evolves and interacts with the surroundings
- Use gamma-ray bursts to study the early universe
- Perform the first sensitive hard X-ray survey of the sky

One instrument on Swift is the Burst Alert Telescope (BAT), a large coded aperture instrument with a wide field-of-view (FOV) that provides the gamma-ray burst triggers. BAT can observe and locate hundreds of bursts per year to better than 4 arc minutes accuracy. BAT contains thousands of detector elements that are assembled into 8 x 16 arrays, each connected to 128-channel readout ASICs.

Gamma-ray Large Area Space Telescope (GLAST) [Renamed the Fermi Gamma Ray Space telescope]	

The Fermi Gamma-ray Space Telescope, launched in 2008, is an international and multi-agency space mission to study the cosmos in the energy range 10 keV - 300 GeV. Fermi has an imaging gamma-ray telescope vastly more capable than instruments flown previously, as well as a secondary instrument to augment the study of gamma-ray bursts. The main instrument, the Large Area Telescope (LAT), has superior area, angular resolution, and field of view over previous instruments. The LAT tracker subsystem was focused on compactness, minimum wiring, and redundancy. The subsystem was implemented using two ASICs.

SonoSite's TITAN™ system took part in a 10-day underwater experiment with NASA Extreme Environment Mission Operations (NEEMO) 7 Mission. Aquanauts used the laptop-sized ultrasound system to scan each other in simulated emergency situations and transmit live images to a hospital for review by radiologists. The TITAN system utilizes SonoSite's proprietary ASIC microchip technology to integrate millions of transistors onto one circuit.

NASA's Jet Propulsion Laboratory has developed a command interface ASIC and an analog interface ASIC. This chip set for remote actuation and monitoring of a collection of switches can be used to control generic loads, pyrotechnic devices, and valves in a high-radiation environment. The command interface ASIC (CIA) can be used alone or in combination with the analog interface ASIC (AIA). Designed primarily for incorporation into spacecraft control systems, they are also suitable for use in high-radiation terrestrial environments (e.g., in nuclear power plants and facilities that process radioactive materials).

A.4 SOC

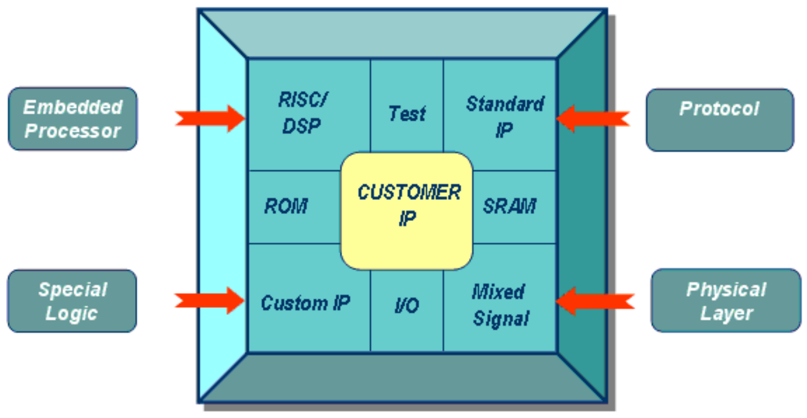
Device name:	System-on-Chip (SoC) Also known as System-on-a-chip (SoaC)
power control A/D D/A analog I/O host interface µController digital I/O interbus ROM RAM Application Controller	 Embedded Processor RISC/ DSP Test Standard IP Protocol ROM CUSTOMER IP SRAM Special Logic Custom IP I/O Mixed Signal Physical Layer

Figure A-5 SOC

<u>Description:</u>

SoC, also called “system-on-a-chip” or SoaC, is a complete product that contains all the necessary electronic circuits and parts for a "system" on a single integrated circuit. Think of it as a single-board-computer on a chip. SoCs include the hardware components and all required ancillary electronics.

- SoCs combine aspects of ASICs and field-programmable logic. SoCs can be:
- Totally ASIC, with the individual blocks specified by the designer
- ASIC for the computing unit and logic functions, with some programmable parts (e.g., CPLD)
- Implemented on programmable logic (e.g., FPGA)

SoCs can use IP designs created by others and integrated into the chip. IP blocks are pre-designed behavioral or physical descriptions of a standard component. These reusable components are usually Commercial-off-the-Shelf (COTS) products.

The benefits of SoC design include:

- Conservation of space (reduction in chip count)
- Improved performance (higher reliability)
- Lower memory requirements
- Greater design freedom (simpler logistics)

These benefits also come with some challenges including:

- Larger design space

- More expense (global on-chip communication is expensive in terms of power/propagation delay)
- Increased prototype cost
- Correctness of complete system with multiple components
- A high level of debugging methodology

Testing of the products is also a challenge due to the fact that typical testing methods have been developed for specific specialty areas, whereas the SoC requirement includes all specialties, potentially on one platform.

A SoC could include:

- Microprocessor
- Memory (e.g., SRAM, DRAM, Flash)
- Communications cores
- Digital Input/Output functions
- Analog Input/Output functions
- Bus controllers (e.g., PCI)
- DSP (Digital Signal Processor)
- Sensors
- Programmable logic (e.g., FPGA, CPLD)
- Embedded software

For example, a system-on-chip for a sound-detecting device might include an audio receiver, an analog-to-digital converter (ADC), a microprocessor, necessary memory, and the input/output logic control for a user - all on a single microchip.

Configurable System-on-Chip (CSoC)

Configurable SoCs are a new form of system-on-chip that has a configurable fabric that designers can manipulate, after chip fabrication, to achieve specific functionality. Configurability lets you change on-chip functions for a variety of reasons. These reasons include:

- change in core functionality
- compatibility with a change in a communications or other standard to which the CSoC must conform
- correcting a design error incurred during original chip development.

Post-process configurability lets you create products that can adapt to changing requirements.

Some configurable SoCs are FPGAs that combine both hard (fixed) and soft (programmable) cores. These chips are sometimes referred to as platform FPGAs. In the diagram below, the microprocessor is a hard component (fixed in the silicon), while the Digital Signal Processor (DSP) is a soft component created in the FPGA programmable infrastructure.

The reconfigurable approach offers significant advantages. It reduces design costs because changes can be made immediately to the chip during development. Chip simulation becomes less of an issue because the real hardware is available immediately. In the field, bug fixes and upgrades can be more extensive as significant portions of the hardware can be altered, not just the application code.

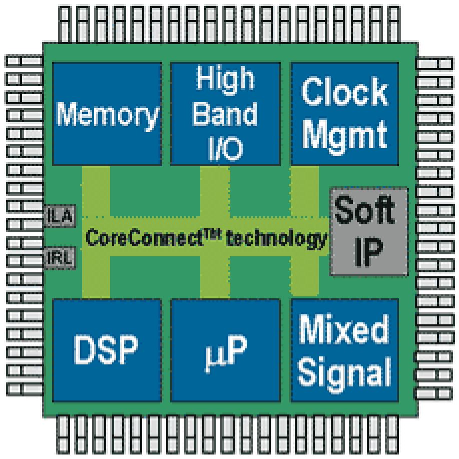


Figure A-6 Reconfigurable SoC

Cost is the main downside to using a standard reconfigurable SoC rather than creating a custom SoC. Custom designs typically have large up-front development costs, but low individual chip costs. Reconfigurable SoCs have a comparatively small up-front cost, but are usually more expensive per chip. Reconfigurable SoCs can also be used for prototyping because the core CPU and fixed peripherals are well defined. Building a custom ASIC or SoC based on a reconfigurable prototype is relatively easy.

Applications for SoC:

System-on-chip devices can be used in any application that requires a processor and peripheral components. Since the advantages of SoC are small size, integrated components, and reduced power, they are especially useful in:

- Cell/camera phones
- Medical equipment (especially portable devices)
- Portable multimedia devices
- Network-enabled devices
- PDAs
- Point-of-sale devices
- Gaming systems

In the medical world, portable equipment and implantable devices are becoming more common. Such equipment includes blood glucose monitoring systems, insulin pumps, body temperature sensors, defibrillators, neurological stimulators, pacemakers, and hearing aids. These products not only simplify the testing, monitoring, and treatment processes, but can also help to improve the quality of life for the patient by minimizing time spent in hospitals and often providing automatic, continuous treatment of chronic conditions.

To address requirements for performance, power consumption, and size, medical equipment manufacturers are incorporating as much functionality as possible into a single, complex SoC. These devices need to integrate both analog and digital capabilities and, in many cases, deliver short-range, low-data-rate wireless communications functionality. Furthermore, some applications may also require that high-voltage output stages be integrated into the same device. A variety of semiconductor technologies, IP blocks, and support tools can help to significantly simplify the implementation of SoCs for implantable and portable medical devices.

An example of a network device is the Sony Video Network Station. This device, which contains an embedded Linux operating system running on an Axis ETRAX system-on-chip processor, transmits images generated by analog video cameras to remote locations where they can be viewed using ordinary GUI-based web browsers. The device is useful in a diverse range of applications requiring remote video monitoring and control, including security monitoring, quality inspection, image distribution, access control, and market research.

NASA projects:

Like ASICs and FPGAs, SoC devices have significant benefits for NASA projects, including:

- Customizable electronics
- Smaller circuit board footprint
- Less weight
- Integrated functionality

Temperature Remote I/O (TRIO) System-on-Chip for Aerospace

The TRIO smart sensor data acquisition chip was developed by Johns Hopkins University/Applied Physics Laboratory for NASA spacecraft applications. TRIO includes a 10 bit self-corrected analog-to-digital converter, analog inputs, a front end multiplexer with selectable acquisition time, a current source, memory, serial and parallel bus, and control logic. These functions are very useful for spacecraft and subsystems health and status monitoring and control actions. The key contributions of the TRIO are feasibility of modular architectures, elimination of several miles of wire harnessing, and power savings by orders of magnitude. So far TRIO is used in many missions including Contour, Messenger, Stereo, Europa Orbiter, Mars Surveyor Program, Solar Probe, Pluto Express, and in the generic JPL X2000 spacecraft bus.

Radio Frequency (RF) components

Micro-Electro-Mechanical Systems (MEMS) integrate mechanical elements, sensors, actuators, and electronics on a common silicon substrate through microfabrication technology. Microelectronic integrated circuits can be thought of as the "brains" of a system and MEMS augments this decision-making capability with "eyes" and "arms" to allow microsystems to sense and control the environment.

NASA Glenn Research Center is developing microwave MEMS devices that integrate with miniature microwave (RF) transmission lines and components to build low loss RF distribution networks for System on a Chip (SOAC) and phase array antennas. These novel, low loss, miniature RF components will be fabricated using multilayer processing, and they will be combined with SOAC technology being developed by the University of Michigan and the JPL Center for Integrated Space Microsystems (CISM) for nano-sized science craft.

Advanced time-of-flight system-on-chip for remote sensing instruments

Accurate and/or fast time interval measurement is important in many remote sensing instruments, especially those that require detection of photon/particle events, position decoding and time-of-hit measurement. An advance time-of-flight (TOF) system-on-chip has been developed that includes the complete signal processing electronics for microchannel plate (MCP) readout. The TOF chip is capable of a time resolution of <50picoseconds. The TOF chip was used on the NASA/IMAGE spacecraft launched in 2000 and is part of many other science instruments on MESSENGER.

ChipSat

ChipSat is a long-term research program which aims to build a satellite-on-a-chip. As part of the program, an existing on-board computer (OBC) was scaled down to a SoC. The OBC chosen was developed by the Surrey Satellite Technology Limited (SSTL), a company owned by the University of Surrey in Guildford, UK. The SoC is prototyped on a single high-density programmable logic array chip using soft IP cores.

The image below shows the parts of the OBC that were mapped into the system-on-chip. An entire board was shrunk down to a single chip. The experiment showed that it is possible to implement the functionality of a small satellite OBC on a single programmable logic chip.

SCOC – A Spacecraft Controller On a Chip

The European Space Agency (ESA) is pursuing development of a system-on-chip that incorporates all the required functions for spacecraft control. This SoC is currently prototyped in an FPGA. The demonstration board is named BLADE (Development of the Board for LEON and Avionics DEmonstration). Eventually, the design will be produced in a radiation-tolerant ASIC or PROM-based FPGA.

SCOC looks to integrate multiple functions into a single chip. By integrating the functions, the external connections become on-chip interconnects. Other benefits include reduced power consumption, reduced component count (and thus lower mass), and increased performance and reliability. However, putting all the functions on a single chip reduces the accessibility to the internal functions and makes testing the complex chip more difficult.

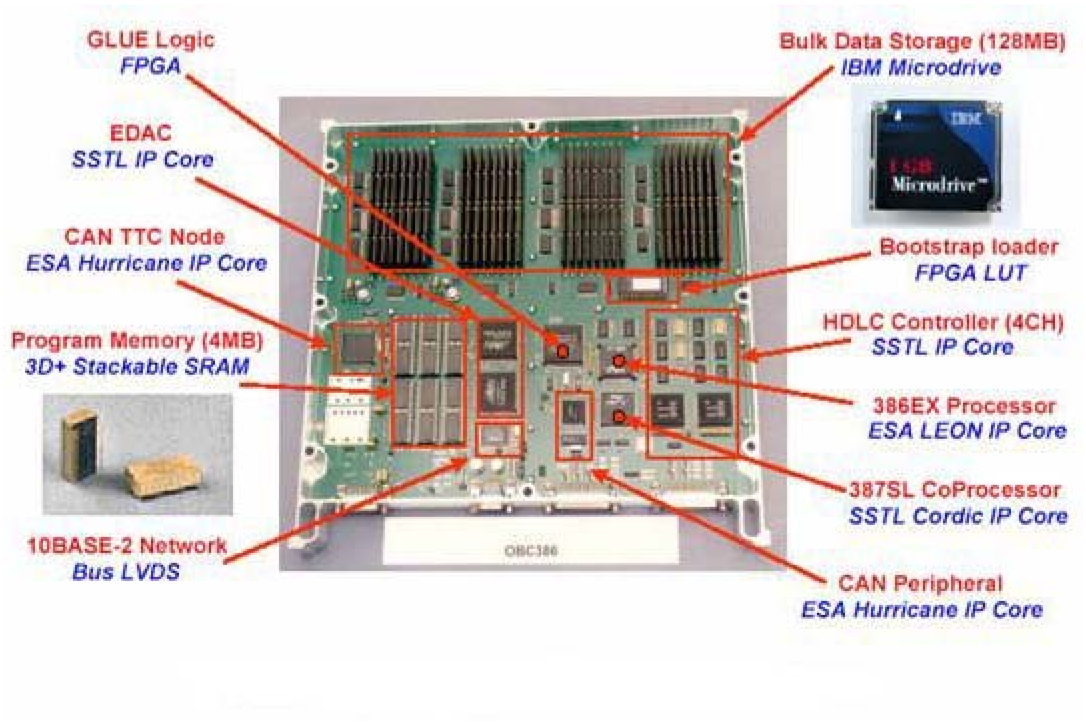


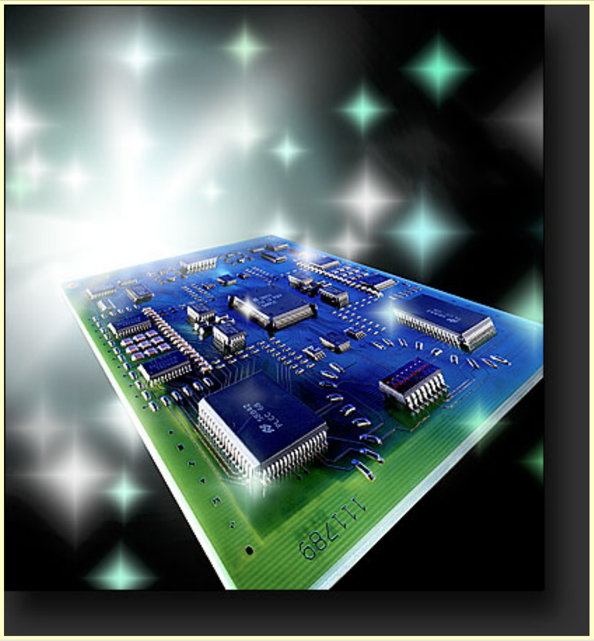
Figure A-7 ChipSat OBC

The SCOC will include the following components

- Standard processor, known to the space community (the LEON SPARC-V8)
- Flexible peripherals, which can be powered down
- Telecommand and Telemetry (TM/TC) functionality (using the CCSDS protocol)
- Housekeeping and CCSDS Time Management
- Multiple standard interfaces
- PCI parallel backbone
- Spacewire (IEEE 1355.1)
- MIL 1553 standard Bus Controller/Monitor (BC/BM) and Remote Terminal (RT)
- Dedicated data processing
- Monitoring camera interface and image compression
- GNSS navigation receiver
- Star tracker pre-processor
- Mathematical co-processor

The current BLADE development integrates the processor with standard interfaces. Additional functionality will be added in the future.

A.5 RECONFIGURABLE COMPUTING

Device name:	**Reconfigurable Computing** **AKA Adaptable Computing, Evolvable computing**
	

Description:

Can you have a computer without a microprocessor? How do you deal with situations where autonomous instruments have to adapt to changing situations? What if your device has to support multiple protocols, depending on its location or mission? How do you process signals that may come in multiple formats without advance planning?

The answer to the above questions is “reconfigurable computing.” Reconfigurable Computing represents a new idea in computing philosophy, in which some general-purpose hardware agent is configured to carry out a specific task, but can be reconfigured on-demand to carry out other specific tasks.

Traditionally, there have been two ways to implement a computation or algorithm: custom hardware or software. In some systems, this decision can be made on an individual subtask basis, placing some subtasks in custom hardware and some in software on more general-purpose processing engines.

Hardware designs offer high performance because they are:

- Customized to the problem—no extra overhead for interpretation or extra circuitry capable of solving a more general problem.
- Relatively fast, due to their highly parallel and spatial execution.

Software implementations exploit a “general-purpose” execution engine (i.e., microprocessor), which interprets a designated data stream as instructions telling the engine what operations to perform. As a result, software is:

- Flexible—task can be changed simply by changing the instruction stream

- Relatively slow—due to mostly temporal, serial execution
- Relatively inefficient–since operators can be poorly matched to computational task.

Reconfigurable computing combines the best of both implementations, allowing general-purpose software to be implemented in hardware. This class of architectures permits the computational capacity of the system to be highly customized to the instantaneous needs of an application, while also allowing the computational capacity to be reused in time at a variety of time scales.

The usual hardware agent for reconfigurable computing is a set of FPGAs. Reconfigurable computing manipulates the logic within the FPGA at run-time. The design of the hardware may change in response to the demands placed upon the system while it is running. Here, the FPGA acts as an execution engine for a variety of different hardware functions, some executing in parallel, others in serial, much as a microprocessor executes a variety of software threads.

Reconfigurable computing offers several advantages over custom hardware and general-purpose software implementations, including:

- Flexibility - the system can be changed as necessary, on the fly.
- Simpler hardware design - you do not need a fancy high-powered microprocessor, just one or more FPGAs.
- Speed - implementing algorithms in hardware results in faster execution, due to the parallel nature of hardware.

The reconfigurable computing systems built during the last years have often achieved performance several orders of magnitude higher than the traditional processor based solutions. Reconfigurable computing is now breaking into the commercial market in the areas of application-specific systems and information appliances, which include emerging areas like mobile communication, multimedia-based networks, encryption, and image processing.

What hardware is reconfigurable?

Not all FPGAs can be used in reconfigurable computing. User-configurable FPGAs can be programmed and reprogrammed by the user in a lab, or even in the field. But they cannot be dynamically reprogrammed as the system is running. Many older FPGAs read their configuration out of a serial EEPROM, and only when a chip reset signal is asserted. This means that the FPGA must be reprogrammed in its entirety and that its previous internal state cannot be captured beforehand.

In order to benefit from run-time reconfiguration, the FPGAs involved need some or all of the following features, which increase design flexibility:

- **On-the-fly reprogrammability**. Resetting the FPGA takes a lot of time and should be avoided whenever possible.
- **Partial reprogrammability**. The ability to leave most of the internal logic in place and change just one part is an important factor in reconfigurable systems. It will always be much faster to change a small piece of the logic than the entire FPGA contents.
- **Externally-visible internal state**. If you can see the internal state of the FPGA at any time, then it is also possible to capture that state and save it for later use. This allows the

internal state of the FPGA to be read and written just like memory or processor registers and makes it possible to swap hardware designs in much the same way that pages of virtual memory are swapped into and out of physical memory.

Run-time environments

How does the reconfigurable system know what to do at any given time? That job is usually handled by software. The software manages the processes of:

- Deciding which hardware objects to execute and when.
- Swapping hardware objects into and out of the reconfigurable logic.
- Performing routing between hardware objects or between hardware objects and the hardware object framework.

Having software manage the reconfigurable hardware usually means having an embedded processor or microcontroller on-board. The embedded software that runs there is called the run-time environment and is analogous to the operating system that manages the execution of multiple software threads. Like threads, hardware objects may have priorities, deadlines, and contexts. It is the job of the run-time environment to organize this information and make decisions based upon it.

Using software allows us to write our applications at a very high level of abstraction. For example, if the software needed to decompress an image, the attached FPGA could be reconfigured with the decompression algorithm and fed the data. To the main software application, this action is no different than asking an analog-to-digital converter to read a voltage and return the answer. The run-time environment software, however, is responsible for reprogramming the FPGA and executing the task.

Programming reconfigurable systems

Reconfigurable computing combines traditional software-related topics as languages, compilers, operating systems, and libraries with hardware-related topics of digital design.

Reconfigurable systems present a formidable challenge in terms of algorithm design tools. Design tools for FPGA devices, the building blocks of reconfigurable hardware, are oriented towards ASIC development environments, in which digital design engineers create large (multi-million gate), complex designs that, once created and validated, do not change. In contrast, reconfigurable supercomputers require a more software-centric development environment, in which algorithms are constantly revised and tested.

In response to the need for software-oriented tools, vendors and researchers have developed compilers for software programming languages that synthesize hardware. Compilers for several C variants, Java, and Matlab have become available in the past few years. The compiler must generate a structural hardware representation (such as VHDL-RTL) that represents the connections between units contained in a library, with direct correspondence to the operators of high-level programming languages.

Applications:

While commercial reconfigurable computing platforms are starting to become available, the majority of work has been done in a research context. There are some areas and problems that reconfigurable computing is ideal for, including:

- Real-time image analysis
- Pattern recognition
- Automatic target recognition
- Cryptography
- Computational biology
- Signal processing

One theoretical application is a smart cellular phone that supports multiple communication and data protocols, though just one a time. When the phone passes from a geographic region that is served by one protocol into a region that is served by another, the hardware is automatically reconfigured. This is reconfigurable computing at its best. Using this approach, it is possible to design systems that do more, cost less, and have shorter design and implementation cycles.

Heading into the future, evolvable hardware (EHW) is designed to adapt to changes in task requirements or changes in the environment through its ability to reconfigure its own hardware structure dynamically and autonomously. This capacity for adaptation is achieved by employing efficient search algorithms known as genetic algorithms. Evolvable hardware has great potential for the development of innovative applications, including autonomous spacecraft and exploration systems.

Here are some reasons why reconfigurable computing has valuable applications for space missions:

After launch, unmanned spacecraft electronics are generally unavailable for physical upgrade or repair. RC technology allows new hardware circuits to be uploaded via a radio link.

New circuit configurations can overcome design faults, allow improved processing algorithms to be uploaded, or change system functionality in response to changing mission requirements. Combined with artificial intelligence applications, the unmanned spacecraft may be able to select circuits on its own to correct the problems.

The same circuitry can be used with different configurations at different stages of a mission, reducing weight and power requirements.

If part of an FPGA fails, then circuitry can be reprogrammed to make use of remaining functional portions of the chips.

Use of FPGAs allows generic circuit boards to be designed, which are customized for individual applications. This helps overcome the very high NRE (non-recurring engineering) costs associated with small volume spacecraft design. Physical and environmental qualification costs can also be shared across many missions.

In-flight reconfiguration provides additional safety margins for missions with very short lead-times, or for those where mission requirements are not fully defined at launch.

NASA Examples:

NASA Langley Research Center is one NASA installation exploring reconfigurable computing applications. They have developed a reconfigurable FPGA-based research hypercomputer that is capable of performing comprehensive engineering and scientific calculations. Two approaches have been adopted to exploit Langley's Star Bridge Systems HC-38 (and 2 HAL15s) for analysis calculations:

1. Rewrite legacy code for the hypercomputer to fully exploit parallelism.
2. Use the hypercomputer to accelerate time-consuming (bottleneck) calculations.

Software was entirely rewritten from C++ or Fortran to take advantage of the parallelism inherent in the hypercomputer (approach 1). When only a small portion of a software application was computationally intensive, that portion was rewritten to the hypercomputer native language, and the rest of the code was left alone (approach 2).

FedSat, an Australian science and engineering research satellite, was launched in 2002. One payload on FedSat is the Adaptive Instrument Module (AIM), which is a reconfigurable computer optimized for spacecraft instrument use. AIM has demonstrated autonomous instrument processing that is reconfigurable and adaptive. The use of the AIM enables reconfiguration of the FPGA circuitry while the spacecraft is in flight. This flexibility reduces mission risk, especially for missions with a very tight development schedule. The AIM is designed to either directly interface with sensors or instruments or to receive data through the spacecraft data handling system. AIM conducted a series of designed experiments, including a demonstration of implementing data compression, data filtering, and communication message processing and inter-experiment data computation.

The design of the AIM specifically addresses the concerns of using SRAM-based FPGAs in the space environment. The AIM demonstrates techniques to detect and remediate radiation-induced upsets in these FPGAs and will automatically restart in the event of an upset. The design has been proven in flight. When the module suffered a memory error due to the bombardment of cosmic radiation, AIM automatically detected and then reset itself. This prevented the memory error from causing an error in the data it was processing.

The team that developed AIM at the Applied Physics Laboratory/John Hopkins University is worked with NASA's Langley Research Center to take the next step in reconfigurable, self-repairing space borne computer design. The project is called ADAPT – Adaptive Data Analysis and Processing Technology. Because it is fully reconfigurable, an ADAPT computer can serve as the front-end package for virtually any type of instrument – for example, a spacecraft might carry six scientific instruments, each served by a physically identical, but differently programmed, ADAPT computer. As the design evolves, an ADAPT computer may carry up to 20 preprogrammed operating modes for controlling its instrument.

APPENDIX B CODING STYLE GUIDELINES

Note: ***Material presented in Appendix B is based on or adapted from figures and text copyrighted by Xilinx, Inc., and used with permission.***

B.1 INTRODUCTION

This document was created to provide Xilinx users with a guideline for producing fast, reliable, and reusable HDL code.

B.2 TOP-DOWN DESIGN

HDL coding should start with a top-down design approach. Use a top-level block diagram to communicate to designers the naming required for signals and hierarchical levels. Signal naming is especially important during the debug stage. Consistent naming of signals, from top to bottom, will ensure that project manager A can easily recognize the signals written by designer B.

B.2.1 Behavioral and Structural Code

When creating synthesizable code (RTL), you should write two types of code: behavioral RTL (leaf-level logic inference, sub-blocks) and structural code (blocks) -- each exclusively in its own architecture. A simple example of behavioral RTL versus structural code is shown in Figure B-1and Figure B-2, respectively.

```
entity mux2to1 is
   port (
      a     : in  std_logic_vector(1 downto 0);
      sel   : in  std_logic;
      muxed : out std_logic);
end mux2to1;

architecture rtl of mux2to1 is
begin

   muxed <= a(1) when sel = '1' else a(0);

end rtl;
```

Figure B-1 Behavioral Code

```
entity mux4to1 is
   port (
      input : in  std_logic_vector(3 downto 0);
      sel   : in  std_logic_vector(1 downto 0);
      muxed : out std_logic);
end mux4to1;

architecture structural of mux4to1 is
   signal muxed_mid : std_logic_vector(1 downto 0);
   component mux2to1
      port (
         a     : in  std_logic_vector(1 downto 0);
         sel   : in  std_logic;
         muxed : out std_logic);
   end component;
begin

   mux2to1_1_0: mux2to1
      port map (
         a     => input(1 downto 0),
         sel   => sel(0),
         muxed => muxed_mid(0));
   mux2to1_3_2: mux2to1
      port map (
         a     => input(3 downto 2),
         sel   => sel(0),
         muxed => muxed_mid(1));
   mux2to1_final: mux2to1
      port map (
         a     => muxed_mid,
         sel   => sel(1),
         muxed => muxed);
end structure;
```

Figure B-2 Structural Code

Rules

Keep leaf-level (behavioral sub-blocks) coding separate from structural coding (blocks).

Declarations, Instantiations, and Mappings. It is important to use a consistent, universal style for such things as entity declarations, component declarations, port mappings, functions, and procedures.

B.2.2 Declarations, Instantiations, and Mappings

It is important to use a consistent, universal style for such things as entity declarations, component declarations, port mappings, functions, and procedures.

Rules

For declarations, instantiations, and mappings use one line for each signal. The exception is for relatively small components, functions, and procedures.

Always use named association.

The combination of these two rules will help eliminate common coding mistakes. Therefore, this combination will greatly enhance the ease of debugging a design at every stage of verification. A simple example is shown Figure B-3. Obeying these rules will also increase the readability, and therefore the reusability.

```
architecture structural of
mux4to1 is
   . . .
begin

     mux2to1_1_0: mux2to1
       port map (
           a     => input(1 downto
0)
```

Figure B-3 One Line Per Signal/Named Association

B.2.3 Comments

Liberal comments are mandatory to maintain reusable code. Although VHDL is sometimes considered to be self-documenting code, it requires liberal comments to clarify intent, as any VHDL user can verify.

Rules

Three primary levels of commenting:

Comments should include a header template for each entity-architecture pair and for each package- and package-body pair. See the example in Figure B-4. The purpose should include a brief description of the functionality of each lower block instantiated within it.

Use comment headers for processes, functions, and procedures, as shown Figure B-5. This should be a description of the purpose of that block of code.

Use comments internal to processes, functions, and procedures to describe what a particular statement is accomplishing. While the other two levels of commenting should always be included, this level is left to the designer to decipher what is required to convey intent. Inline comments are shown in Figure B-6.

```
---------------------------------------------------------------------------------------------------------------------------------
-- Author: John Q. Smith                                         Copyright Xilinx, 2001
--                                                               Xilinx FPGA - VirtexII
-- Begin Date: 1/10/01
-- Revision History      Date          Author                   Comments
--                       1/10/01       John Smith               Created
--                       1/14/01       John Smith               changed entity port address & data to addr & dat
---------------------------------------------------------------------------------------------------------------------------------
--  Purpose:
-- This entity/architecture pair is a block level with 4 sub-blocks.  This is the processor control interface for the
-- block level <block_level_A>.  So on, and so forth
---------------------------------------------------------------------------------------------------------------------------------
```

Figure B-4 Header Template

```
------------------------------------------------------------------------------------------------
--  demux_proc:  this process dumultiplexes the inputs and registers the
-- demultiplexed signals
------------------------------------------------------------------------------------------------
demux_proc : process(clk, reset)
begin
```

Figure B-5 Process, Function, and Procedure Header

```
------------------------------------------------------------------------------------------------
--  demux_proc:  this process dumultiplexes the inputs and registers the
-- demultiplexed signals
------------------------------------------------------------------------------------------------
demux_proc : process(clk, reset)
begin
if reset = ‘1’ then
   demux <= (others => ‘0’);
elsif rising_edge(clk) then
  -- demultiplex input onto the signal demux
   case (sel) is
   when ‘0’ =>
      demux(0) <= input;
   when ‘1’ =>
      demux(1) <= input;
   when others =>
      demux <= (others => ‘0’);
   end case;
end if;
end process;
```

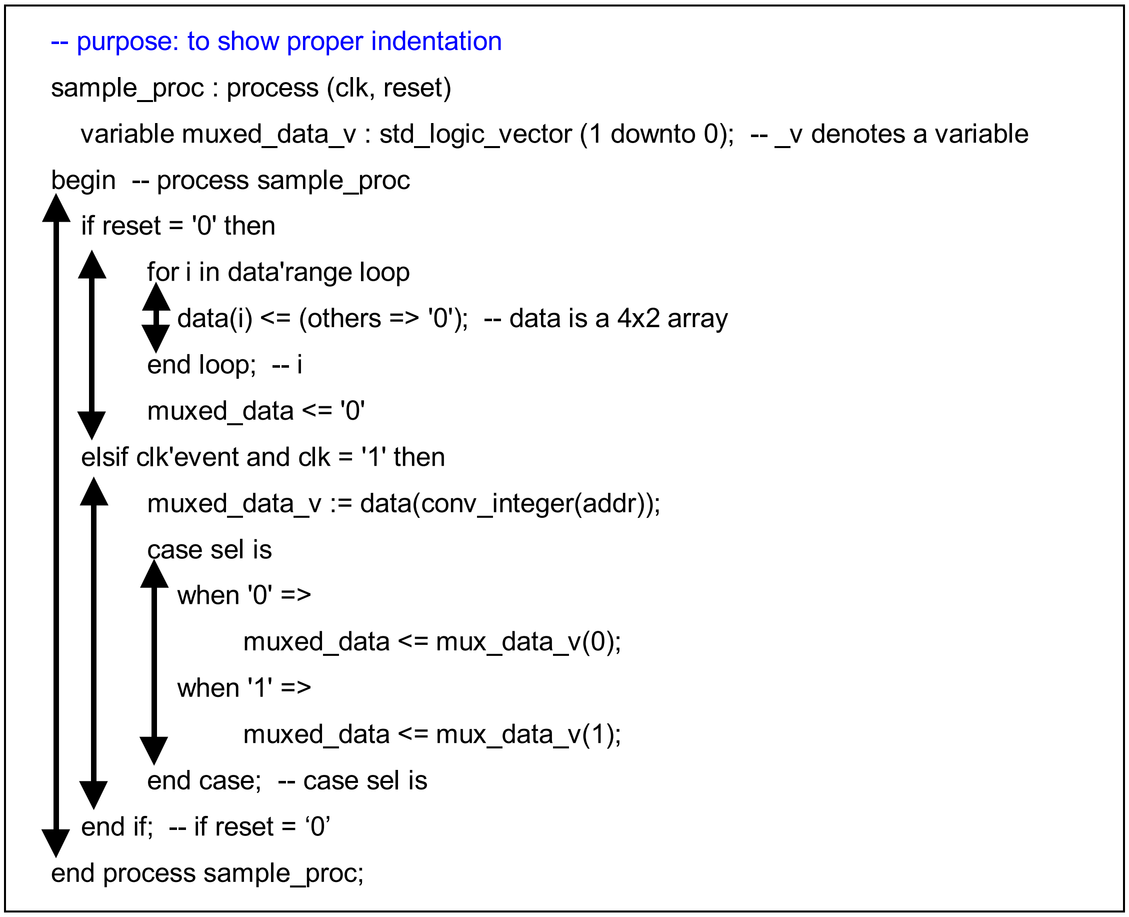
Figure B-6 Inline Comments

B.2.4 Indentation

Proper indentation ensures readability and reuse. Therefore, a consistent style is warranted. Many text editors are VHDL-aware, automatically indenting for “blocks” of code, providing consistent indentation. Emacs and CodeWright are two of the most common editors that have this capability. Figure B-7 shows an example of proper indentation. Proper indentation greatly simplifies reading the code. If it is easier to read, it is less likely that there will be coding mistakes by the designer.

Rules

Use a VHDL-aware text editor that provides a consistent indentation style.



```
-- purpose: to show proper indentation
sample_proc : process (clk, reset)
  variable muxed_data_v : std_logic_vector (1 downto 0);  -- _v denotes a variable
begin  -- process sample_proc
  if reset = '0' then
      for i in data'range loop
        data(i) <= (others => '0');  -- data is a 4x2 array
      end loop;  -- i
      muxed_data <= '0'
  elsif clk'event and clk = '1' then
      muxed_data_v := data(conv_integer(addr));
      case sel is
        when '0' =>
              muxed_data <= mux_data_v(0);
        when '1' =>
              muxed_data <= mux_data_v(1);
      end case;  -- case sel is
  end if;  -- if reset = ‘0’
end process sample_proc;
```

Figure B-7 Proper Indentation

B.2.5 Naming Conventions

Naming conventions maintain a consistent style, which facilitates design reuse. If all designers use the same conventions, designer A can easily understand and use designer B’s VHDL code.

B.2.6 Entities, Architectures, Procedures, and Functions

Rules

Use all lowercase names with underscores, for readability and name delimiting.

Each entity should have a unique name that describes that block.

Architectures do not need unique names because they are individually bound to a specific entity that has a unique name. Names for architectures should be rtl, to indicate a leaf-level sub-block, and structural, to indicate a block with no leaf-level logic – with only sub-blocks.

For entities with more than one architecture, use “rtl_xilinx” (or “structural_xilinx”) for a Xilinx-specific architecture and “rtl_asic” (or “structural_asic”) for an ASIC-specific architecture.

B.2.7 Signal Naming Conventions

For a design implemented in VHDL, an up-front specification of signal naming conventions should help you reduce the amount of non-conformity. The primary motivating factor is enhanced readability during the verification of the design. General signal naming conventions are listed below.

General Signal Naming Guidelines

Use addr for addresses. This might include sys_addr, up_addr, etc.

Use clk for clock. This might include clk_div2 (clock divided by 2), clk_x2 (clk multiplied by 2), etc.

Use reset or rst for synchronous reset.

Use areset or arst for asynchronous reset.

Use areset_l for active-low asynchronous reset.

Use rw_l for read/write (write is active low).

Rules

The following rules specify the suggested nomenclature for other widely used signals

Use <signal_name>_io for bi-directional signals.

Use a _l suffix for active low signals <signal_name>_l.

Do not use _in and _out suffixes for port signal names.

Use of in and out is very confusing in text, especially at hierarchical boundaries. Therefore, the use of _in and _out should be strictly monitored. If they must be used, be sure that _in indicates input, and, likewise, that _out is an output to the correct level of hierarchy. Figure B-8 shows an example entity and the instantiation of that entity in a higher block. Here, data_in is connected to data_out, making the code confusing.

```
entity in_out is
port (    data_in : in std_logic_vector (31 downto 0);
          data_out : out std_logic_vector(31 downto 0));
end entity in_out;


in_out_inst: in_out
port map ( data_in => ram_data_out,
             data_out => ram_data_in);
```

Figure B-8 Confusing _in and _out suffixes

Use _i to denote local signal names that are internal representations of an output port. This nomenclature is used to easily identify the internal signal that will eventually be used as an output port.

The counter in Figure B-9 provides a simple example of an output port that cannot be read. The output port count cannot be incremented because it would require count to be read. The problem is solved in the example by incrementing the local internal signal count_i. Some designers try to overcome this problem by using the port as an inout; however, not all synthesis compilers will allow this unless it is three-stated. Declaring the signal to be of type buffer is another common trap. This complicates the code because all signals to which it connects also must be of type buffer. Not all synthesis vendors support the data-type buffer. In addition, data-type buffer does not have all of the required defined functions to perform common arithmetic operations.

```
count <= count_i;
process (clk, reset)
begin
  if reset = '1' then
     count_i <= (others => '0');
  elsif rising_edge(clk) then
     count_i <= count_i + 1;
  end if;
end process;
```

Figure B-9 Internal Signals Representing Output Ports

Use _v to indicate a variable. Variables can be very useful if used correctly. The _v will serve as a reminder to the designer as to the intent and use of that signal.

Use <signal_name>_p0, <signal_name>_p1, and so forth, to represent a pipelined version of the signal <signal_name> when <signal_name> comes after the pipelining. Use <signal_name>_q0, <signal_name>_q1, and so forth, to represent a pipelined version of the <signal_name> when <signal_name> comes before the pipeline. See Figure B-19 in section 4 for an example of how to use this pipelined signal naming convention.

Append a suffix to signals that use a clock enable and will be part of a clock-enabled path (i.e., multi-cycle path). For example, if the clock enable is enabled only one-quarter of clock cycles, the clock enable should be named to represent that -- ce4. Signals that use this enable might be named <signal_name>_ce4. This will greatly aid you in your ability to specify multi-cycle constraints.

B.3 SIGNALS AND VARIABLES

Following some basic rules on the use of signals and variables can greatly reduce common coding problems.

B.3.1 Signals

The rules for using signals are not complex. The most common problem is that signals can be various data types. The problem in VHDL is "casting" from one data type to another. Unfortunately, no single function can automatically cast one signal type to another. Therefore, the use of a standard set of casting functions is important to maintain consistency between designers.

B.3.2 Casting

Rules for Casting

Use std_logic_arith, std_logic_unsigned/std_logic_signed packages.

This provides the essential conversion functions:

- conv_integer(<signal_name>): converts std_logic_vector, unsigned, and signed data types into an integer data type.
- conv_unsigned(<signal_name>, <size>): converts a std_logic_vector, integer, unsigned (change size), or signed data types into an unsigned data type.
- conv_signed(<signal_name>, <size>): converts a std_logic_vector, integer, signed (change size), or unsigned data types into a signed data type.
- conv_std_logic_vector(<signal_name>, <size>): converts an integer, signed, or unsigned data type into a std_logic_vector data type.
- ext(<signal_name>, <size>): zero extends a std_logic_vector to size <size>.
- sxt(<signal_name>, <size>): sign extends a std_logic_vector to size <size>.

All conversion functions can take for the <signal_name> data-type a std_logic_vector, unsigned, signed, std_ulogic_vector, or integer. <size> is specified as an integer value.

B.3.3 Inverted Signals

To reduce complication and to make the code easier to debug and test, it is generally recommended that you use active-high signals in hardware description languages. Generally, active-low signals make the code more complicated than necessary. If active-low signals are required at the boundaries of an FPGA, invert incoming signals at the FPGA top structural level.

Also, for outbound signals, invert them at the FPGA top structural level. Consider this a rule of thumb.

However, for FPGAs in general and Xilinx FPGAs specifically, inverters are free throughout the device. There are inverters in the IOB, and a LUT can draw in an inverter as part of its functionality, without a loss in performance.

Often, ASIC designers will use active-low signals in their code to use less power in the part. The synthesis tool will map the logic based on a vendor's libraries. Therefore, the synthesis tool will infer active-low signals internally when it sees fit. For that matter, writing code that uses active-low signals does not necessarily infer active-low signals in the ASIC. Again, the synthesis tool makes these decisions based on the vendor's libraries. Let the synthesis tool do its job.

Rule of thumb

Use only active-high signals in HDL. One exception is a signal with a dual purpose, such as a read or a write signal. In this case, a naming convention should be used to reduce complication – rw_l is an easily recognizable signal name that clearly defines that signal's role.

Where active-low signals are required, use of a _l as a suffix generally makes the intent clear. E.g., <signal_name>_l. Use of _n is generally confusing.

B.3.4 Rule for Signals

There are a few basic rules to follow when you use signals. Remember that ports are just signals with special rules that apply to them.

B.3.5 Entity Port Rules within the Bound Architecture

You can read from inputs, but you cannot assign to inputs.

You can assign to outputs, but you cannot read from outputs.

See section 1, Signal Naming Conventions, rule number four, for help in skirting this limitation.

You can both assign to and read from inouts.

B.3.6 Internal Signal Rules

Never assign to a signal in more than one process, with the exception of a three-state signal.

For a combinatorial process (no registers inferred), never assign to a signal and read from the same signal in the same process. This will eliminate infinite loops when performing behavioral simulation.

This is not true for a "clocked" process; i.e., a process that is used to register signals. A clocked process would only need to have an asynchronous set or reset signal and a clock in its sensitivity list. Therefore, this process would not execute again until there was a change on one of those signals.

In a clocked process, never assign to a signal outside of the control of the if rising_edge(clk) statement (or reset statement if an asynchronous reset exists). This is a common coding mistake.

In synthesis, it will infer a combinatorial signal. In a behavioral simulation, it will have the behavior of a signal clocked on the falling edge.

B.3.7 Filling out a Process Sensitivity List

Within a combinatorial process, all signals that are read (which can change) must be in the sensitivity list.

This will insure the correct behavioral simulation. This includes any signals that are compared in if-then-else statements and case statements. It also includes any signal on the right-hand side of an assignment operator. Remember that this is only for signals that can change. A constant cannot change; thus, it does not need to be in the sensitivity list.

Within a clocked process, only an asynchronous set or reset and the clock should be in the sensitivity list.

If others are added, the functionality of a behavioral simulation will still be correct. However, the simulation will be slower because that process will need to be evaluated or simulated whenever a signal in its sensitivity list changes.

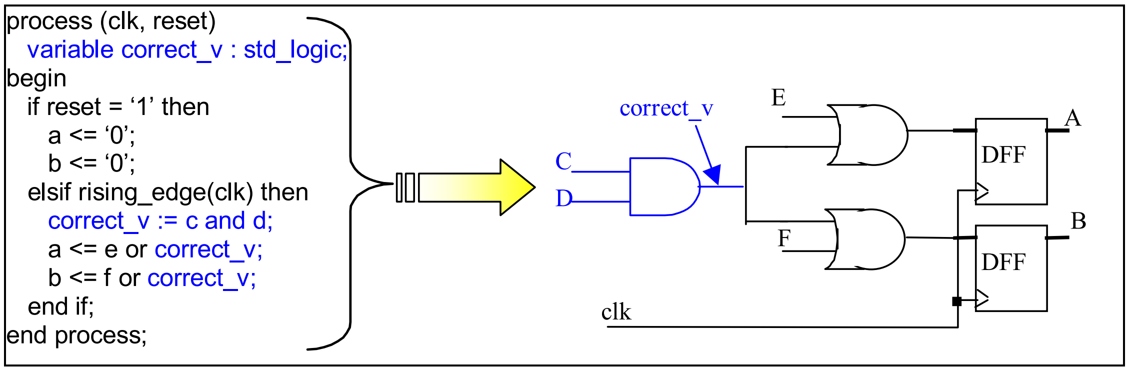
B.3.8 Rules for Variables and Variable Use

Variables are commonly not understood and are therefore not used. Variables are also commonly used and not understood. Variables can be very powerful when used correctly. This warrants an explanation of how to properly use variables.

Variables are used to carry combinatorial signals within a process. Variables are updated differently than signals in simulation and synthesis.

In simulation, variables are updated immediately, as soon as an assignment is made. This differs from signals. Signals are not updated until all processes that are scheduled to run in the current delta cycle have executed (generally referred to as suspending). Thus, a variable can be used to carry a combinatorial signal within both a clocked process and a combinatorial process. This is how synthesis tools treat variables – as intended combinatorial signals.

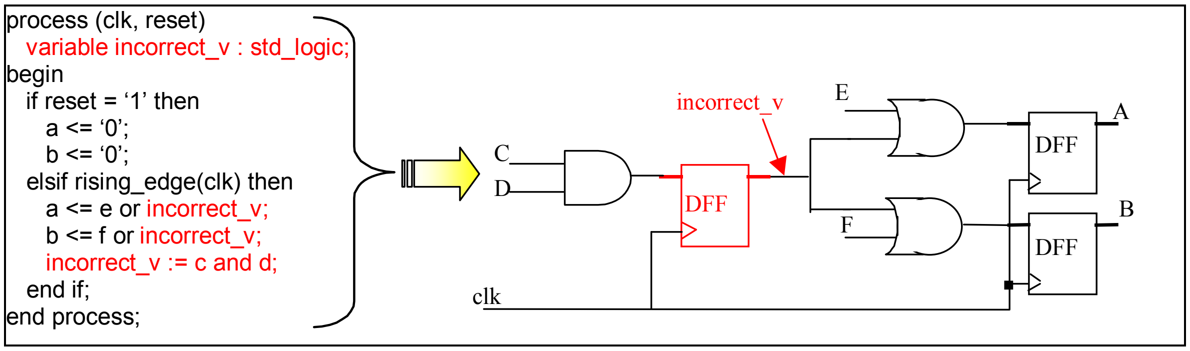
Figure B-10 shows how to use a variable correctly. In this case, the variable correct_v maintains its combinatorial intent of a simple two-input and-gate that drives an input to an or-gate for both the a and b registers.



```
process (clk, reset)
  variable correct_v : std_logic;
begin
  if reset = ‘1’ then
    a <= ‘0’;
    b <= ‘0’;
  elsif rising_edge(clk) then
    correct_v := c and d;
    a <= e or correct_v;
    b <= f or correct_v;
  end if;
end process;
```

Figure B-10 Correct Use of Variables

In Figure B-11, you read from the variable incorrect_v before you assign to it. Thus, incorrect_v uses its previous value, therefore inferring a register. Had this been a combinatorial process, a latch would have been inferred.



```
process (clk, reset)
  variable incorrect_v : std_logic;
begin
  if reset = '1' then
    a <= '0';
    b <= '0';
  elsif rising_edge(clk) then
    a <= e or incorrect_v;
    b <= f or incorrect_v;
    incorrect_v := c and d;
  end if;
end process;
```

Figure B-11 Incorrect Use of Variables

B.3.9 Rule for Variables

Always make an assignment to a variable before it is read. Otherwise, variables will infer either latches (in combinatorial processes) or registers (in clocked processes) to maintain their previous value. The primary intent of a variable is for a combinatorial signal.

B.4 PACKAGES

Packages are useful for creating modular and reusable code. There should be one or more packages used by a design team. These packages should include commonly used functions, procedures, types, subtypes, aliases, and constants. All team designers should familiarize themselves with the contents of these packages. If each designer were to create his or her own functions, procedures, types, subtypes, aliases, and constants, it could result in code that is difficult for other team members to use and read. Thus, when your team uses packages, it results in code that is more modular and more readable.

Package use can generally be broken down into the three types:

- The global package. This package is used on a company-wide basis, on each design. This package should include functions and procedures, such as reduction functions, for instance functions, and procedures that -- and, or, and xor (etc.) -- reduce individual buses. It should also include commonly used types and subtypes. This package should be created in a group setting by VHDL experts (or the most experienced in VHDL) who decide the best elements to have present in the package. This package should be used extensively and should have periodic reviews to determine what should be added to or taken away from the package. Because most divisions within a company work on the same type of projects, primarily, this package should contain the most widely and extensively used material that is common to all design teams.
- The project package. This package is used and created for a specific design project. The functions, procedures, types, subtypes, constants, and aliases are all specifically defined and created for the design at hand.

- The designer's packages. These packages are specific to a designer. Packages of this type should not be used extensively. If there is a need for something to be extensively used within the designer's package, it should be moved into the project package and possibly even the global package. Code readability and modularity is limited by the use of designer packages, as the type of function calls and types, etc. will not be readily understandable to all other designers in the group.

B.4.1 Package Contents

Constants

Used correctly, constants can ease the coding of complex and modular designs. Constants can be used in a variety of ways. They can be used to create ROMs, for modular coding, and to define what or how something should be used. For example, constants can be used in conjunction with generate statements to specify which portion of code to use (synthesize). Consider, for example, one portion of code written for an ASIC implementation and another portion written for a Xilinx implementation. The ASIC implementation should use gates to implement a multiplexer, while the Xilinx version should use three-state buffers to implement a multiplexer. Because some synthesis tools do not currently support configuration statements, a generate statement is the best solution.

Figure B-12 shows an example of how constants can be used to define the logic created. Although this is a simple example, it illustrates the possibilities. By one change to the constant ASIC, an entirely different set of circuitry is synthesized throughout the design.

```
--within a package
constant asic : boolean := True;


-- within an architecture
generate_asic :
if asic = true then
mux_proc : process (addr, sel, data)


generate_fpga :
if asic = false then
tri_state_proc : process (addr, sel, data)
```

Figure B-12 A Constant Guiding the Generation of Logic

Constants can aid modular coding. For example, you could define a constant that specifies the width of the address bus. One change to that constant in the package would make a modular change to everything in the design. See Figure B-13. Using constants to define address and data-bus widths may be better than using generics. Generics are passed from the top-down, eliminating the possibility of synthesizing bottom-up. A bottom-up synthesis is generally preferable for decreased synthesis run-times because only the modules that change need to be resynthesized.

```
--within the package pack_ase_fpga
constant addrw : integer := 18;

use work.pack_ase_fpga.all;

entity profound is
port ( addr : in std_logic_vector (addrw-1 downto 0);
```

Figure B-13 Address Width Defined by a Constant

B.4.2 Rules for Defining Constants within a Package

Define constants within a package when it can be used to improve the modularity of the code by guiding generate statements.

Define constants in a package to define sizes and widths of buses. Constants used in this manner are generally more powerful than using generics because it allows the design to be synthesized in any manner, whereas generics allow only top-down synthesis.

Functions and Procedures

By definition, functions and procedures add modularity and reuse to code. Extensive use of functions and procedures from the global and project packages is encouraged. Rather than extensively using functions and procedures from a designer's package, the designer is encouraged to add the functions and procedures at a local level (within an architecture), to maintain readability for other designers and future reuse.

When defining functions and procedures, it is beneficial to use unsized vectors to pass signals. Using unsized vectors allows a modular use of the subprogram. In addition to using unsized vectors, use signal – range attributes to define the logic.

In the function example shown below in Figure B-14, the input, named vec, is defined as a std_logic_vector. By not defining a sized vector, the actual size of the signal that is passed in will determine the implementation. The range attribute 'range specifies the size of the intended logic. This function is modular; that is, it is not limited to being used for one specific vector size. A vector of any size can be passed into this function and correctly infer any amount of logic.

```
function parity (vec : input std_logic_vector) return std_logic is
   variable temp_parity : std_logic := ‘0’;
begin
   for i in vec’range loop
      temp_parity := temp_parity xor vec(i);
   end loop;
   return temp_parity;
end function;
```

Figure B-14 Modular Function Use

B.4.3 Rules for Functions and Procedures

Extensive use of functions and procedures is encouraged. Predominately, the functions and procedures should be defined within either the global or the project packages.

Create modular functions and procedures by not specifying the width of inputs and outputs. Then use range attributes to extract the needed information about the size of an object.

Types, Subtypes, and Aliases

Types and subtypes are encouraged for readability. Types defined at the global and project level are generally required, and they help to create reusable code.

Aliases can be used to clarify the intent, or meaning, of a signal. In most cases, the intent of a signal can be clearly identified by its name. Thus, aliases should not be used extensively. While aliases can help to clarify the purpose of a signal, they also add redirection, which may reduce the readability of the code. Although aliases are not used in conjunction only with types and subtypes, it is useful for examples to be included here. In Figure B-15 there are two types defined: a record and an array. For this example, aliases can be used to clarify the use of the signal rx_packet.data (rx_data) and the intent of the signal data_addr(0) (data_when_addr0). In this example, the alias data_when_addr0 is used in place of data_array(0), this provides more meaning to the "slice" of data than data_array(0) provides. Whenever the alias data_when_addr0 is seen in the code, the intent is obvious. The use of the alias rx_data simply provides a shortened version of the signal rx_packet.data while its use and intent are maintained.

```
package alias_use is
type opcode is record
        parity : std_logic;
        address: std_logic_vector(7 downto 0);
        data : std_logic_vector(7 downto 0);
        stop_bits : std_logic_vector(2 downto 0);
end record;
type data_array_type is array (0 to 3) of std_logic_vector (31 downto 0);
end package;

architecture rtl of alias_use is
  signal addr : std_logic_vector (11 downto 0);
  signal data_array : data_array_type;
  alias data_when_addr0 : std_logic_vector(31 downto 0) is data_array(0);
signal rx_packet : opcode;
alias rx_parity is rx_packet.parity;
alias rx_addr is rx_packet.address;
alias rx_data is rx_packet.data;
alias rx_stop is rx_packet.stop_bits;

begin
  data_when_addr0 <= data when addr = x"000" else (others => '0');
  rx_data <= data_when_addr0;
. . .
```

Figure B-15 Use of Types and Aliases

Rules for Types, Subtypes, and Alias use

Types and subtypes are encouraged on a global or project basis to facilitate reusable code.

Alias use is encouraged when it clearly promotes readability without adding complex redirection.

B.5 TECHNOLOGY-SPECIFIC CODE (XILINX)

It is desirable to maintain portable, reusable code. However, this is not always possible. There are cases for each technology vendor where instantiation of blocks is required. Furthermore, writing what is intended to be generic code will not always provide the best solution for a specific technology. The tradeoffs between instantiation versus technology-specific code are discussed below.

B.5.1 Instantiation

Although instantiation of Xilinx primitives is largely unneeded and unwanted, there are some specific cases where it must be done -- and other occasions when it should be done. While some of the components that need to be instantiated for a Xilinx implementation vary, those covered here are specific for Synplify, Synopsys, Exemplar, and XST. This section will describe situations where deviation from reusable code is required.

Required Instantiation

Specific top-level (FPGA) components require instantiation, including the boundary scan component, digital delay-locked loop components (DLL) or digital clock manager (DCM), startup block, and I/O pullups and pulldowns.

Inputs and outputs, other than LVTTL, can be specified in the synthesis tool. However, it is more advantageous to specify the I/O threshold level in the Xilinx Constraints Editor. This will write a constraint into the Xilinx UCF (User Constraint File), which is fed into the Xilinx implementation tools.

To instantiate Xilinx primitives, you will need to have a correct component declaration. This information can be inferred directly from the Xilinx Libraries Guide, found in the online documentation.

B.5.2 Rules for Required Instantiations for Xilinx

Boundary Scan (BSCAN)

- Digital Clock Manager (DCM) or Delay-Locked Loop (DLL). Instantiating the DCM/DLL provides access to other elements of the DCM, as well as elimination of clock distribution delay. This includes phase shifting, 50-50 duty-cycle correction, multiplication of the clock, and division of the clock.
- IBUFG and BUFG. IBUFG is a dedicated clock buffer that drives the input of the DCM/DLL. BUFG is an internal global clock buffer that drives the internal FPGA clock and provides the feedback clock to the DCM/DLL.
- DDR registers. DDR registers are dedicated Double-Data Rate (DDR) I/O registers located in the input or output block of the FPGA.
- Startup. The startup block provides access to a Global Set or Reset line (GSR) and a Global Three-State line (GTS). The startup block is not inferred because routing a global set or reset line on the dedicated GSR resources is slower than using the abundant general routing resources.
- I/O pullups and pulldowns (pullup, pulldown).

B.5.3 Simulation of Instantiated Xilinx Primitives

Correct behavioral simulation will require certain simulation files. These can be found in the Xilinx directory structure: $Xilinx/vhdl/src/unisims. Note that unisims are similar to simprims, except that: unisims do not have component timing information enabled. Whereas, simprims have the timing information enabled but require an SDF file (from Xilinx place and route) to supply the timing information (post place and route timing simulation).

Within the unisim directory, several VHDL files need to be compiled to a unisim library. They can then be accessed by specifying the library unisim and using the use statement. For example:

- library unisim;
- use unisim.vcomponents.all;

The VHDL files must be compiled in a specific order because there are dependencies between the files. The compilation order is:

1) unisim_VCOMP.vhd
2) unisim_VPKG.vhd
3) unisim_VITAL.vhd

For post-place-and-route timing simulation, the simprim files need to be compiled into a simprim library. The VHDL files for simprims are in: $Xilinx/vhdl/src/simprims. The correct package compilation order is:

1) simprim_Vcomponents.vhd
2) simprim_Vpackage.vhd
3) simprim_VITAL.vhd

Simulation files rules

Unisims are used for behavioral and post-synthesis simulation.

Simprims are used for post place-and-route timing simulation.

B.5.4 Non-Generic Xilinx-Specific Code

This section is used to describe situations where Xilinx-specific coding may be required to get a better implementation than can be inferred from either generic code or ASIC-specific coding.

Three-State Multiplexers

Generic coding of multiplexers is likely to result in an and-or gate implementation. However, for Xilinx parts, gate implementation of multiplexers is generally not advantageous. Xilinx parts have a very fast implementation for multiplexers of 64:1 or less. For multiplexers greater than 64:1, the tradeoffs need to be considered. Multiplexers implemented with internal three-state buffers have a near consistent implementation speed for any size multiplexer.

Three-state multiplexers are implemented by assigning a value of "Z" to a signal. Synthesis further requires concurrent assignment statements. An example is shown in Figure B-16. For this example, there is a default assignment made to the signal data_tri to ‘Z’. The case statement infers the required multiplexing, and the concurrent assignment statements to the signal data infer internal three-state buffers. With those concurrent assignment statements, synthesis can only resolve the signal values by using three-states. Without the concurrent assignment statements, synthesis would implement this in gates, despite the default assignment to "Z."

```
process (r1w0, addr_integer, data_regs1)
begin  -- process
   for i in 0 to 3 loop  -- three-state the signal
          data_tri(i) <= (others => 'Z');
   end loop;  -- i
   if r1w0 = '1' then
          case addr_integer is
             when 0 to 3 =>
                    data_tri(0) <= data_regs1(0);
             when 4 to 7 =>
                    data_tri(1) <= data_regs1(1);
             when 8 to 11 =>
                    data_tri(2) <= data_regs1(2);
             when 12 to 15 =>
                    data_tri(3) <= data_regs1(3);
          end case;
   end if;
end process;
-- concurrent assignments to data
data <= data_tri(0);
data <= data_tri(1);
data <= data_tri(2);
data <= data_tri(3);
```

Figure B-16 Three-state Implementation of 4:1 Multiplexer

B.5.5 Rules for Synthesis Three-State Implementation

Use a default assignment of "Z" to the three-state signal.

Make concurrent assignments to the actual three-stated signal.

Memory

While memory can be inferred for Xilinx, it most likely cannot be inferred for the ASIC by using the same code. It is very likely that two separate implementations will be required. This section will describe the methodology used to infer Xilinx-specific memory resources. It is generally advantageous to instantiate the use of memory resources to make it easier to change for other technology implementations. While it is not always required, Xilinx's CORE Generator™ system program can generate RAM for instantiation. The CORE Generator™ system created memory must be used for dual-ported block RAMs, but it can also be used for creating other types of memory resources. The CORE Generator™ system does provide simulation files, but it is seen as a black box in synthesis; therefore, it will not provide timing information through that block.

RAM and ROM

The Xilinx LUT-RAM is implemented in the look-up tables (LUTs). Each slice has 32-bits of memory. A slice can have three basic single-port memory configurations: 16x1(2), 16x2, or 32x1. The Xilinx slices and CLBs can be cascaded for larger configurations.

LUT-RAM memory is characterized by synchronous write and asynchronous read operation. It also is not able to be reset; however, it can be loaded with initial values through a Xilinx user constraint file (UCF). Inference of Xilinx LUT-RAM resources is based on the same behavior described in the code shown in Figure B-17. Dual-port LUT-RAM can also be inferred by adding a second read address. Dual-port RAM has similar functionality with a synchronous write port and two asynchronous read ports.

```
type ram_array is array (0 to 15) of std_logic_vector (5 downto 0);
signal ram_data : ram_array;


begin
process(clk)  --synchronous write
begin
if clk'event and clk = '1' then
   if we = '1' then
      ram_data(conv_integer(addr_sp)) <= data_to_ram;
   end if;
end if;
end process;

-----------------------------------------------------------
-- for single port, use the same address as
-- is used for the write
-----------------------------------------------------------
-- asynchronous read – dual port
ram_data_dp <=ram_data(conv_integer(addr_dp));
```

Figure B-17 Xilinx LUT-RAM Inference

ROM inference is driven by constants. Example code for inferring LUT-ROM is shown in Figure B-18.

```
type rom is arrary (0 to15) of std_logic_vector (3 downto 0);

-- 16x4 ROM in Xilinx LUT's
constant rom_data : rom := (x"F", x"A", x"7", x"0", x"1", x"5",
x"C", x"D", x"9", x"4", x"8", x"2", x"6, x"3", x"B", x"E");


begin
-- ROM read
data_from_rom <= rom_data(conv_integer(addr));
```

Figure B-18 LUT-ROM Inference

Single-port block RAM inference is driven by a registered read address and a synchronous write. The example shown Figure B-19 has this characterization. In the past, block RAM has been easily inferred, simply by having the registered address and synchronous write. Synthesis tools can only infer simple block RAMs. For example, you cannot infer a dual-port RAM with a configurable aspect ratio for the data ports. For these reasons, most dual-port block RAMs should be block-RAM primitive instantiations or created with the CORE Generator™ system.

```
type ram_array is array (0 to 127) of std_logic_vector (7 downto 0);
signal ram_data : ram_array;


begin
process(clk)  --synchronous write
begin
if clk'event and clk = '1' then
   addr_q0 <= addr;  -- registered address/pipelined address
   if we = '1' then
      ram_data(conv_integer(addr)) <= data_to_ram;
   end if;
end process;

data_from_ram <= ram_data(conv_integer(addr_q0));
```

Figure B-19 Virtex Block RAM inference

B.5.6 Rules for Memory Inference

For single- or dual-port RAM implemented in LUTs, describe the behavior of a synchronous write and an asynchronous read operation.

For ROM inference in LUTs, create an array of constants.

Single-port block RAM is inferred by having a synchronous write and a registered read address (as shown in the example above, Figure B-19).

For other configurations of the Xilinx block RAM, use the CORE Generator™ system.

B.5.7 CORE Generator System

The CORE Generator™ system may be used for creating many different types of ready-made functions. One limiting factor of the CORE Generator™ system is that synthesis tools cannot extract any timing information; it is seen as a black box.

The CORE Generator™ system provides three files for a module:

Implementation file, <module_name>.ngc.

Instantiation template, <module_name>.vho

Simulation wrapper, <module_name>.vhd

For behavioral and post-synthesis simulation, the simulation wrapper file will have to be used. To simulate a CORE Generator™ module, the necessary simulation packages must be compiled. More information on using this flow and generating the necessary files can be found in the CORE Generator tool under Help →Online Documentation.

The CORE Generator™ system provides simulation models in the $Xilinx/vhdl/src/XilinxCoreLib directory. There is a strict order of analysis that must be followed, which can be found in the analyze_order file located in the specified directory. In addition, Xilinx provides a Perl script for a fast and easy analysis of different simulators. To compile the XilinxCoreLib models with ModelSim or VSS, use the following syntax at a command prompt:

```
xilinxperl.exe $Xilinx/vhdl/bin/nt/compile_mti_vhdl.pl coregen
```

```
xilinxperl.exe $Xilinx/vhdl/bin/nt/compile_vss_vhdl.pl coregen
```

Compare logic is frequently implemented poorly in FPGAs. Compare logic is inferred by the use of <, <=, >, and >= VHDL operators. For a Xilinx implementation, this logic is best implemented when described with and-or implementations. When possible, look for patterns in the data or address signals that can be used to implement a comparison with gates, rather than compare logic. If a critical path includes comparison logic, an implementation that would use and-or logic should be considered.

B.5.8 Rule for Comparator Implementation

If a critical path has comparator logic in it, then try to implement the comparison by using and-or gates.

B.5.9 Xilinx Clock Enables

Clock enables are easily inferred, either explicitly or implicitly. Clock enables are very useful for maintaining a synchronous design. They are highly preferable over the unwanted gated clock. However, not all technologies support clock enables directly. For those architectures that do not support clock enables as a direct input to the register, it will be implemented via a feedback path. This type of implementation is not a highly regarded implementation style. Not only does it add a feedback path to the register, it also uses more logic because FPGA architecture requires two extra inputs into the LUT driving the register.

The Xilinx architecture supports clock enables as a direct input to a register. This is highly advantageous for a Xilinx implementation. However, the designer must be certain that the logic required to create the clock enable does not infer large amounts of logic, making it a critical path.

In the example shown below (Figure B-20), there is an explicit inference of a clock enable and an implicit inference of clock enables. In the first section, a clock enable is via explicitly testing for a terminal count. In the second section of code, the clock enables are implied for the signals cs and state. The clock enable for cs is inferred by not making an assignment to cs in the state init. The clock enable for the signal state is inferred by not defining all possible branches for the if-then-else statement, highlighted in red. When the if-then-else condition is false, state must hold its current value. Clock enables are inferred for these conditions when they are in a clocked process. For a combinatorial process, it would infer latches.

```
process (clk)  -- Explicit inference of a clock enable
  begin  -- process
    if rising_edge(clk) then
        if tc  = '1' then
            cnt <= cnt + ‘1’;
        end if;
    end if;
  end process;

process (clk, reset)  -- Implicit inference of a clock enable
  begin  -- process
   if reset = '1' then
       state <= (others => '0');
       cs <= "00";
   elsif rising_edge(clk) then
       case (state) is
           when init => -- inference of a clock enable for signal cs
                state <= load;
           when fetch =>
                 if (a = '1' and b = '1') then  -- inference of a clock enable for signal state
                     state <= init;
                 end if;
                 cs <= "11";
            when others => null;
       end case;
   end if;
  end process;
```

Figure B-20 Clock Enable Inference

B.5.10 Rules for Clock Enable Inference

Clock enables can only be inferred in a clocked process.

Clock enables can be inferred explicitly by testing an enable signal. If the enable is true, the signal is updated. If enable is false, that signal will hold its current value.

Clock enables can be implicitly inferred two ways:

Not assigning to a signal in every branch of an if-then-else statement or case statement. Remember that latches will be inferred for this condition in a combinatorial process (see section 5, Inadvertent latch Inference).

Not defining all possible states or branches of an if-then-else or case statement.

Pipelining with SRL

In Xilinx FPGAs, there is an abundance of registers; there are two registers per slice. This is sufficient for most registered signals. However, there are times when multiple pipeline delays are required at the end of a path. When this is true, it is best to use the Xilinx SRL (Shift Register LUT). The SRL uses the LUT as a shiftable RAM to create the effect of a shift register. In Figure B-21an example of how to infer the SRL is shown. This will infer a shift register with 16 shifts (width = 4). Although this will infer registers for an ASIC, it will infer the SRL when you are targeting a Xilinx part. The behavior that is required to infer the SRL is highlighted in blue. The size could be made parameterizable by using constants to define the signal widths. It could also be made into a procedure with parameterized widths and sizes.

```
library ieee ;
use ieee.std_logic_1164.all ;

entity srltest is
  port(clk, en : in  std_logic ;
       din : in std_logic_vector(3 downto 0);
       dout : out std_logic_vector(3 downto 0)) ;
end srltest ;

architecture rtl of srltest is
   type srl_16x4_array is array (15 downto 0) of std_logic_vector(3 downto 0);
   signal sreg  : srl_16x4_array ;
begin
  dout   <= sreg(15) ;  -- read from constant location
  srl_proc : process (clk, en)
  begin
     if rising_edge(clk) then
            if (en = '1') then
               sreg <= sreg(14 downto 0) & din ; -- shift the data
                                   --  Current Value sreg (15:1)  sreg(0)
                                   --  Next Value     sreg  (14:0)   din
            end if;
     end if;
```

Figure B-21 Inference of Xilinx Shift Register LUT (SRL)

B.5.11 Rules for SRL Inference

No reset functionality may be used directly to the registers.

If a reset is required, the reset data must be supplied to the SRL until the pipeline is filled with reset data.

You may read from a constant location or from a dynamic address location. In Xilinx Virtex™-II parts, you may read from two different locations: a fixed location and a dynamically addressable location.

B.5.12 Technology-Specific Logic Generation – Generate Statements

This section has outlined ways that Xilinx-specific coding will differ from other solutions. Because many styles may exist for a similar block of code (for example a multiplexer), to get the optimal implementation, use VHDL generate statements. This is the best solution for a couple of reasons. Although configuration statements are commonly used to guide the synthesis of multiple implementation styles, some synthesis tools currently do not fully support them. Also, with generate statements, a change to a single constant will change the type of logic generated (ASIC or FPGA).

An example of using generate statements was covered in section 3, in the Figure B-12.

B.6 CODING FOR SYNTHESIS

The main synthesis issues involve coding for minimum logic level implementation (i.e., coding for speed, max frequency); inadvertent logic inference; and fast, reliable, and reusable code.

B.6.1 Synchronous Design

The number one reason that a design does not work in a Xilinx FPGA is that the design uses asynchronous techniques. To clarify, the primary concern is asynchronous techniques used to insert delays to align data, not crossing clock domains. Crossing clock domains is often unavoidable, and there are good techniques for accomplishing that task via FIFOs. There are no good techniques to implement an asynchronous design. First, and most important, the actual delay can vary based on the junction temperature. Second, for timing simulations, Xilinx provides only maximum delays. If a design works based on the maximum delays, this does not mean that it will work with actual delays. Third, Xilinx will stamp surplus –6 (faster) parts with a –5 or –4 (slower speed) speed-grade. However, if the design is done synchronously there will be no adverse effects.

B.6.2 Clocking

In a synchronous design, only one clock and one edge of the clock should be used. There are exceptions to this rule. For example, by utilizing the 50/50 duty-cycle correction of the DCM/DLL, in a Xilinx FPGA you may safely use both edges of the clock because the duty-cycle will not drift.

Do not generate internal clocks. Primarily, do not generate gated clocks because these clocks will glitch, propagating erroneous data. The other primary problems with internally generated clocks are clock-skew related problems. Internal clocks that are not placed on a global clock buffer will incur clock skew, making it unreliable. Replace these internally generated clocks with either a clock enable signal or generate divided, multiplied, phase shifted, etc. clocks with a clock generated via the DCM/DLL.

B.6.3 Rules for Clock Signals

Use one clock signal and one edge.

Do not generate internal clock signals because of glitching and clock-skew related problems.

B.6.4 Local Synchronous Sets and Resets

Local synchronous sets and resets eliminate the glitching associated with local asynchronous sets and resets. An example of such a problem is associated with the use of a binary counter that does not use the maximal binary count. For example, a four-bit binary counter has 16 possible binary counts. However, if the design calls only for 14 counts, the counter needs to be reset before it has reached its limit. An example of using local asynchronous resets is highlighted in red in Figure B-22. A well-behaved circuit is highlighted in blue, in the Figure B-23. For the binary counter that is using a local asynchronous reset, there will be glitching associated with the binary transitions, which will cause the local asynchronous reset to be generated. When this happens, the circuit will propagate erroneous data.

```
-- Asynchronous local reset and internally generated clock
process (clk, reset, cnt_reset)
begin  -- process
-- global and local async. reset
   if (reset = '1' or cnt_reset = '1') then
         tc <= '0';
         cnt <= "0000";
   elsif rising_edge(clk) then
         if cnt = "1110" then
            cnt_reset <= '1';
            tc <= '1';
         else
            cnt <= cnt + 1;
            tc <= '0';
            cnt_reset <= '0';
         end if;
   end if;
end process;
-- internally generated clock - tc
process (tc, reset)
begin  -- process
   if reset = '1' then
         data_en <= (others => '0');
   elsif rising_edge(tc) then
         data_en <= data;
   end if;
```

Figure B-22 Local Asynchronous Reset and TC & Well-Behaved Synchronous Reset & CE

```
-- Synchronous Local reset and clock enable use
process (clk, reset)
   variable tc : std_logic := '0';
begin  -- process
   if reset = '1' then  -- global asynchronous reset
         cnt <= "0000";
         data_en <= (others => '0');
   elsif rising_edge(clk) then
      if cnt = "1110" then
            cnt <= "0000";   -- local synchronous reset
           data_en <= data;  -- terminal count clock enable
      else
            cnt <= cnt + '1';
            tc := '0';
      end if;
   end if;
end process;
```

Figure B-23 Well Behaved Local Asynchronous Reset and TC & Well-Behaved Synchronous Reset & CE

B.6.5 Rule for Local Set or Reset Signals

A local reset or set signal should use a synchronous implementation.

Pipelining

Pipelining is the act of inserting registers into one path to align that data with the data in another path, such that both paths have an equal amount of latency. Pipelining may also decrease the amount of combinatorial delay between registers, thus increasing the maximum clock frequency. Pipelines are often inserted at the end of a path by using a shift register implementation. Shift registers in Xilinx's Virtex™ parts are best implemented in the LUT as an SRL, as described in section B.4. Signal naming for pipelined signals is covered in section B.1.

Registering Leaf-Level Outputs and Top-Level Inputs

A very robust technique, used in synchronous design, is registering outputs of leaf-levels (sub-blocks). This has several advantages:

- No optimization is needed across hierarchical boundaries.
- Enables the ability to preserve the hierarchy.
- Bottom-up compilation.
- Recompile only those levels that have changed.
- Enables hierarchical floorplanning.
- Increases the capability of a guided implementation.
- Forces the designer to keep like-logic together.

Similarly, registering the top-level inputs decreases the input to clock (ti2c) delays; therefore, it increases the chip-to-chip frequency.

B.6.6 Rules for the Hierarchical Registering of Signals

- Register outputs of leaf-level blocks.
- Register the inputs to the chip's top-level.

B.6.7 Clock Enables

The use of clock enables increases the routability of a Xilinx implementation and maintains synchronous design. The use of clock enables is the correct alternative to gated clocks.

Clock enables increase the routability of the design because the registers with clock enables will run at a reduced clock frequency. If the clock enable is one-half the clock rate, the clock enabled datapaths are placed-and-routed once the full clock frequency paths have been placed-and-routed. The clock enable should have a timing constraint placed on it so that the Xilinx implementation tools will recognize the difference between the normal clock frequency and the clock-enabled frequency. This will place a lower priority on routing the clock-enabled paths.

Gated clocks will introduce glitching in a design, causing incorrect data to be propagated in the data stream. Therefore, gated clocks should be avoided.

Using signals generated by sequential logic as clocks is a common error. For example, you use a counter to count through a specific number of clock cycles, producing a registered terminal count. The terminal count is then used as a clock to register data. This internal clock is routed on the general interconnect. The skew on internally generated clocks can be so detrimental that it causes errors. This may also cause race conditions if the data is resynchronized with the system clock. This error is illustrated in Figure B-23. The text highlighted in red is the implementation using the terminal count as a clock.

Instead, generate the terminal count one count previous, and use the terminal count as a clock enable for the data register. The text highlighted in blue is the well-behaved implementation using the terminal count as a clock enable. An explanation of the reset signals is covered in the next section B.5.

It may be useful to generate clock enables by using a state machine. The state machine can be encoded at run time by the synthesis tool. Thus a one-hot, gray, or Johnson encoding style could be used. It is also possible to produce precisely placed clock enables by using a linear feedback shift register (LFSR), also known as a pseudo-random bitstream generator (PRBS generator). Xilinx provides application notes on the use of LFSRs.

Clock enables for Xilinx implementations are further discussed in section 4.

B.6.8 Rules for Clock Enable

Use clock enables in place of gated clocks.

Use clock enables in place of internally generated clocks.

Finite State Machines

Coding for Finite State Machines (FSM) includes analyzing several tradeoffs.

B.6.9 Encoding Style

Enumerated types in VHDL allow the FSM to be encoded by the synthesis tool. However, the encoding style used will not be clearly defined in the code but rather in the synthesis tool. Therefore, good documentation should be provided -- stating specifically which encoding style was used. By default, most synthesis tools will use binary encoding for state machines with less than five states: one-hot for 5 to 24 states and gray for more than 24 states (or similar). Otherwise, synthesis will use one-hot encoding. One-hot encoding is the suggested implementation for Xilinx FPGAs because Xilinx FPGAs have abundant registers. Other encoding styles may also be used -- specifically gray encoding. For a gray-encoding style, only one-bit transitions on any given state transition (in most cases); therefore, less registers are used than for a one-hot implementation, and glitching is minimized. The tradeoffs for these encoding styles can easily be analyzed by changing a synthesis FSM attribute and running it through synthesis to get an estimate of the timing. The timing shown in synthesis will most likely not match the actual implemented timing; however, the timing shown between the different encoding styles will be relative, therefore providing the designer a good estimate of which encoding style to use.

Another possibility is to specifically encode the state machine. This is easily done via the use of constants. The code will clearly document the encoding style used. In general, one-hot is the suggested method of encoding for FPGAs -- specifically for Xilinx. A one-hot encoding style uses more registers, but the decoding for each state (and the outputs) is minimized, increasing performance. Other possibilities include gray, Johnson (ring-counter), user-encoded, and binary. Again, the tradeoffs can easily be analyzed by changing the encoding style and synthesizing the code.

Regardless of the encoding style used, the designer should analyze illegal states. Specifically, are all the possible states used? Often, state machines do not use all the possible states. Therefore, the designer should consider what occurs when an illegal state is encountered. Certainly, a one-hot implementation does not cover all possible states. For a one-hot implementation, many illegal states exist. Thus, if the synthesis tool must decode these states, it may become much slower. The code can also specifically report what will happen when an illegal state is encountered by using a “when others” VHDL case statement. Under the “when others” statement, the state and all outputs should be assigned to a specific value. Generally, the best solution is to return to the reset state. The designer could also choose to ignore illegal states by encoding “don’t care” values (‘X’) and allow the synthesis tool to optimize the logic for illegal states. This will result in a fast state machine, but illegal states will not be covered.

B.6.10 Rules for Encoding FSMs

For enumerated-types, encode the state machine with synthesis-specific attributes. Decide if the logic should check for illegal states.

For user-encoded state machines, the designer should analyze whether the logic should check for illegal states or not, and the designer should accordingly write the “when others” statement. If the designer is concerned with illegal states, the state machine should revert to the reset state. If the designer is not concerned with illegal states, the outputs and state should be assigned "X" in the “when others” statement.

Xilinx suggests using one-hot encoding for most state machines. If the state machine is large, the designer should consider using a gray or Johnson encoding style and accordingly analyze the tradeoffs.

B.6.11 FSM VHDL Processes

Most synthesis tools suggest coding state machines with three process statements: one for the next state decoding, one for the output decoding, and one for registering of outputs and state bits. This is not as concise as using one process statement to implement the entire state machine; however, it allows the synthesis tools the ability to better optimize the logic for both the outputs and the next-state decoding. Another style is to use two processes to implement the state machine: one for next state and output decoding and the other process for registering of outputs and state bits.

The decision to use one, two, or three process statements is entirely left up to the discretion of the designer. Moore state machines (output is dependent only on the current state) generally have limited decoding for the outputs, and the state machine can, therefore, be safely coded with either one or two process statements. Mealy state machine (outputs depend on the inputs and the current state) output decoding is generally more complex, and, therefore, the designer should use three processes. Mealy state machines are also the preferred style for FSMs because it is advantageous to register the outputs of a sub-block (as described above in section 5). Mealy state machines will have the least amount of latency with registered outputs. Mealy state machines can be used with a look-ahead scheme. Based on the current state and the inputs, the outputs can be decoded for the next state. For simple state machines where the output is not dependent on the inputs, a Moore implementation is equivalent to a look-ahead scheme. That is, the outputs can be decoded for the next state and appropriately registered to reflect the next state (rather than reflecting the current state). The purpose of this scheme is to introduce the least amount of latency when registering the outputs.

B.6.12 Rules for FSM Style

Generally, use three process statements for a state machine: one process for next-state decoding, one for output decoding, and one for the registering of state bits and outputs.

Use a Mealy look-ahead state machine with registered outputs whenever possible, or use a Moore state machine with next-state output decoding and registered outputs to incur the minimum amount of latency.

B.6.13 Logic Level Reduction

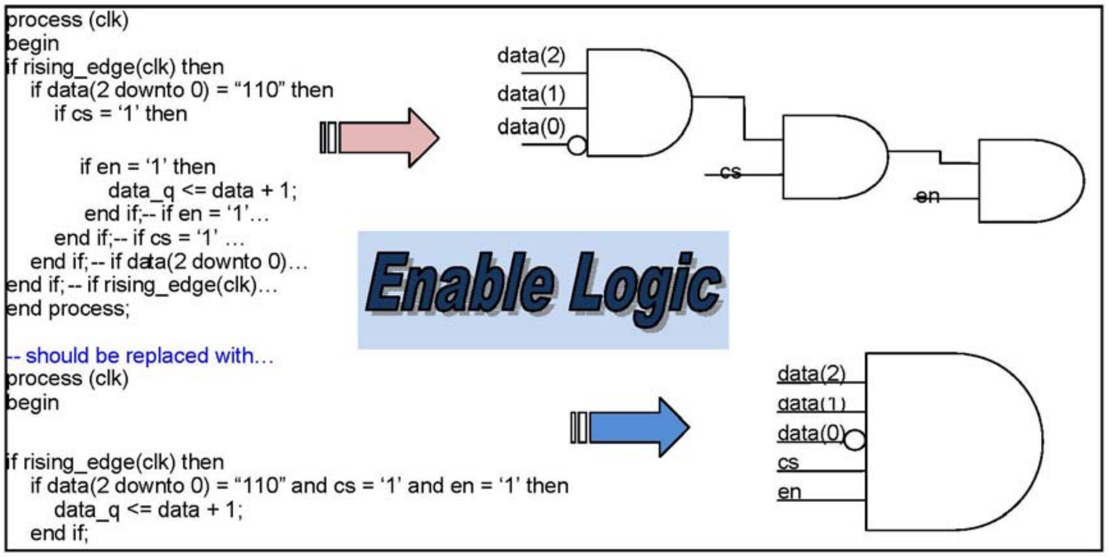
To minimize the number of cascaded logic levels, we need to follow a few simple rules of coding.

B.6.14 If-Then-Else and Case Statements

If-then-else and case statements can cause unwanted effects in a design. Specifically, nested If-then-else and case statements may cause extra levels of logic inference. This occurs because if-then-else statements generally infer priority-encoded logic. However, one level of an if-then-else will not necessarily create priority-encoded logic. For that matter, synthesis tools generally handle if-then-else or case statements very well and create parallel logic rather than priority encoded logic.

Often, a nested if statement can be combined in the original if statement and result in a reduced amount of inferred logic. A simple example is shown in Figure B-24, which shows how priority encoded logic creates cascaded logic. Nested case statements can have the same effect, as can the combination of nested case and if-then-else statements.

Priority-encoded logic can be generated for other reasons. The use of overlapping conditions in if-then-else branches causes the generation of priority-encoded logic. This condition should be avoided. There are times that priority-encoded logic must be used and may be intended. If the selector expressions in the if-then-else statement branches are not related, then priority-encoded logic will be created. Although this may be the intent, its use should be cautioned.



```
process (clk)
begin
if rising_edge(clk) then
   if data(2 downto 0) = "110" then
      if cs = '1' then

         if en = '1' then
            data_q <= data + 1;
         end if;-- if en = '1'…
      end if;-- if cs = '1' …
   end if; -- if data(2 downto 0)…
end if; -- if rising_edge(clk)…
end process;

-- should be replaced with…
process (clk)
begin

if rising_edge(clk) then
   if data(2 downto 0) = "110" and cs = '1' and en = '1' then
      data_q <= data + 1;
   end if;
```

Figure B-24 Priority Encoded Logic

B.6.15 Rules for If-Then-Else and Case Statements

Limit the use of nested if-then-else and case statements.

Avoid overlapping conditions in if-then-else statements – this condition will infer priority-encoded logic.

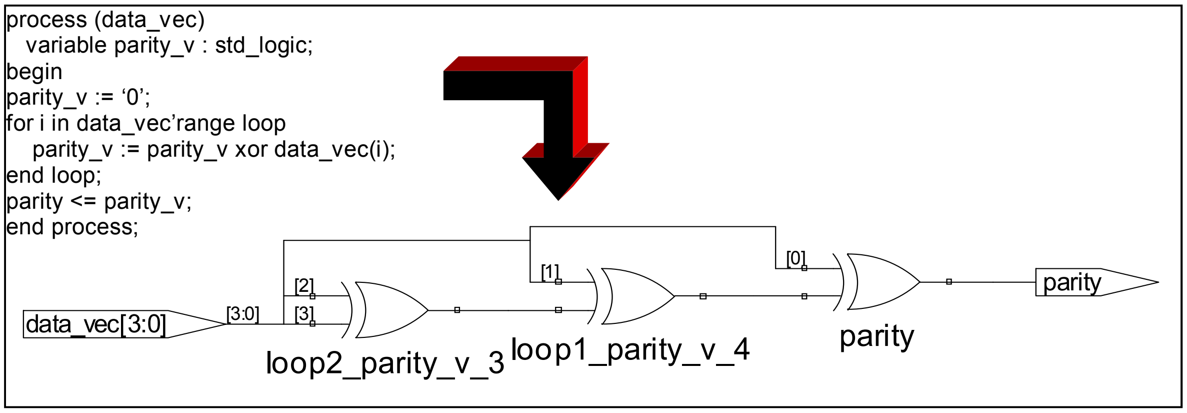
Avoid using mutually exclusive branch expressions if possible. This condition will always infer priority-encoded logic.

Instead, use mutually exclusive if-then statements for each expression (if possible).

B.6.16 For Loops

Similar to the use of if-then-else and case statements, "for loops" can create priority-encoded logic. While for loops can be a very powerful tool for creating logic, the designer should evaluate their effects.

A simple example of the adverse effect of for loops is shown in Figure B-25. Fortunately, this is a situation that most tools handle well, but in our goal of creating reusable (portable) code, this situation should be avoided.



```
process (data_vec)
  variable parity_v : std_logic;
begin
parity_v := ‘0’;
for i in data_vec’range loop
   parity_v := parity_v xor data_vec(i);
end loop;
parity <= parity_v;
end process;
```

Figure B-25 For-Loop Cascaded Logic Implementation

B.6.17 Rule for Loops

Be cautious of using for loops for creating logic. Evaluate the logic created by the synthesis tool. There will likely be another way to write the code to implement the same functionality with the logic implemented in parallel.

Inadvertent Latch Inference

Inadvertent latch inference is a common problem that is easily avoided. Latches are inferred for two primary reasons: one, not covering all possible branches in if-then-else statements and two, not assigning to each signal in each branch. This is only a problem in a combinatorial process. For a clocked process, registers with clock enables are synthesized (as covered in the Xilinx Specific Coding section, section B.4).

A latch inference example for each of these cases is shown in Figure B-26. For section one, each possible state was not covered. This is a very common mistake for one-hot encoded state machines. Section one is an example of Moore FSM output decoding. The latch inference would be eliminated by the use of a final "else" statement with an assignment in that branch to the signal "cs."

For the second section of code, the latches are inferred because each signal is not assign in each state.

In Figure B-27, the inference of latches is eliminated by covering all possible branches and assigning to each signal in each branch. The fixes are highlighted in blue. For the case implementation, the default assignment to cs before the case statement specifies a default assignment for each state. This way, each bit is changed depending on the state. This is equivalent to making a signal assignment in each branch. For the if-then-else statement, adding the else clause solves the problem.

```
-- (1) Did not specify each possible state
process (state)
begin
if state = “001” then
   cs <= “01”;
elsif state = “010” then
   cs <= “10”;
elsif state = “100” then
   cs <= “11”;
end if;
end process;

-- (2) Did not specify all possible
-- outputs for each state
process (state)
  begin
    case (state) is
      when "001" =>
        cs(0) <= '1';
      when "010" =>
        cs(1) <= '1';
      when "100" =>
        cs(1 downto 0) <= "11";
      when others =>
        cs(1 downto 0) <= "00";
    end case;
  end process;
```

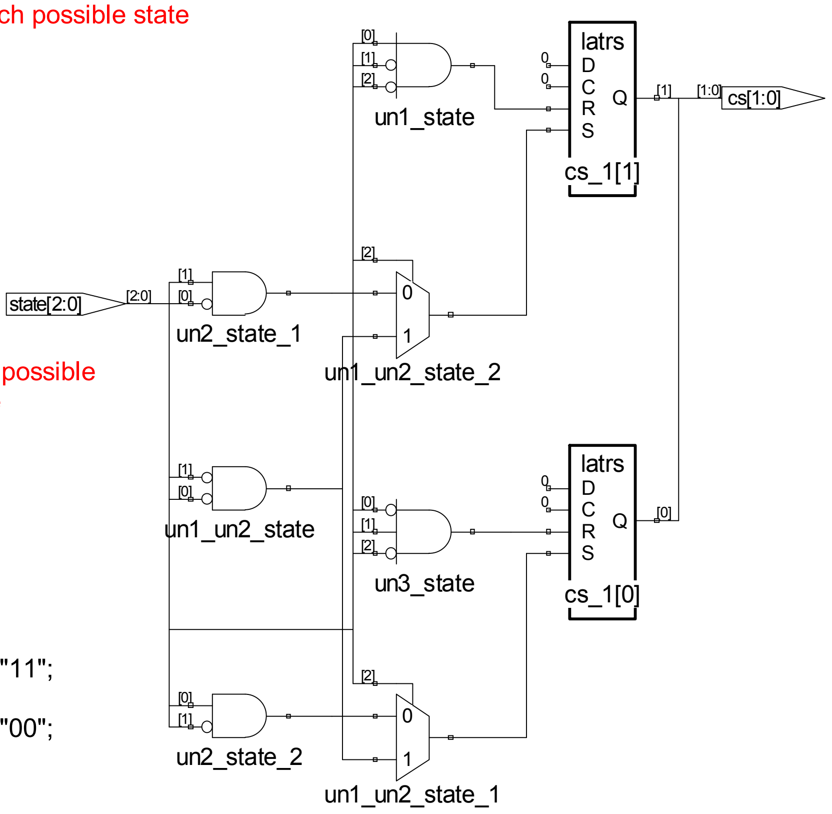


Figure B-26 Latch Inference

```
-- (1) Fixed Case implementation
process (state)
  begin
    cs <= "00";
    case (state) is
      when "001" =>
        cs(0) <= '1';
      when "010" =>
        cs(1) <= '1';
      when "100" =>
        cs <= "11";
      when others =>
        cs <= "00";
    end case;
  end process;
```

```
(2) Fixed if-then-else implementation
 process (state)
 begin
    if state = "001" then
      cs <= "01";
       elsif state = "010" then
     cs <= "10";
       elsif state = "100" then
     cs <= "11";
       else
      cs <= "00";
        end if;
   end process;
```

Figure B-27 Elimination of Inadvertent Latch Inference

B.6.18 Rules for Avoidance of Latch Inference

Cover all possible branches.

Assign to all signals at all branches.