OPTIMIZATION TECHNIQUES USED IN LOGIC CIRCUIT SYNTHESIS AND A STUDY OF THE OPTIMALITY OF SYNTHESIS RESULTS

by
Nusret Utku Altunkaya

Submitted to the Institute of Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science
in
Electrical and Electronics Engineering

Yeditepe University
2006

OPTIMIZATION TECHNIQUES USED IN LOGIC CIRCUIT SYNTHESIS AND A STUDY OF THE OPTIMALITY OF SYNTHESIS RESULTS

APPROVED BY:

Prof. Dr. Ahmet Dervişoğlu ……
(Thesis Supervisor)

Prof. Dr. Uğur Çilingiroğlu ……

Assoc. Prof. Herman Sedef ……

DATE OF APPROVAL: 21.09.2006

ACKNOWLEDGEMENTS

I'd like to thank my family, first of all, for supporting me and helping me whenever I need them, wherever I need them; and especially my parents, to whom I owe my interest in science and technology, which eventually led me to become an electronics engineer.

I am grateful to Prof. Dr. Ahmet Dervişoğlu for the inspiration and guidance he has been providing me with for the past 7 years on both academic and real life issues, with his extraordinary, good-humored personality and his respectable enthusiasm for all things technological and innovative.

Without doubt, every member of the Yeditepe EE academic staff, both full-time and part-time, past and present, has had some valuable influence on my view of the academic world, the electronics profession, and human relations. I've learned a lot here. Thank you all for the experience.

Most of the illustrations in Chapters 2 and 3 are by Clive Maxfield, who was kind enough to include the original drawings on CD accompanying his book "The Design Warrior's Guide to FPGAs" (ISBN 0750676043, 2004, Mentor Graphics Corp.), to be freely used for preparing educational material.

Lastly, I should also thank Çağlayan Bookstore in Taksim, İstanbul for their excellent service and kind attention in quickly finding and supplying the books I have asked for. Without those books, most of which I have listed in the reference section, this work would have been much shallower.

ABSTRACT

OPTIMIZATION TECHNIQUES USED IN LOGIC CIRCUIT SYNTHESIS AND A STUDY OF THE OPTIMALITY OF SYNTHESIS RESULTS

From the 1960s to present day, the rapid pace of development in integrated circuit (IC) manufacturing technology has doubled the number of transistors that can be fitted on a chip every 1.5 to 2 years. Circuit complexities have grown far beyond what is manageable with conventional pen-and-paper based design methods, and new electronic design automation (EDA) tools have emerged that can synthesize complete circuits from verbal behavioral descriptions written in a Hardware Description Language (HDL).

This thesis focuses on logic circuit synthesis and investigates the optimization techniques used by these synthesis tools. A sample circuit written in Verilog HDL has also been implemented on a Spartan-3E field programmable gate array (FPGA) with Xilinx's ISE 8.2i development software, using various different optimization settings in its XST synthesis tool. The resulting circuits have been compared in terms of device utilization and maximum operating speed, in order to find out how optimization settings affect the synthesis tool's performance.

By trying out all possible combinations of a small subset of the XST settings, several synthesis iterations were performed. The fastest implementation that could be achieved occupied 235 slices on the FPGA and had a maximum clock frequency of 186.78 MHz. In comparison, the smallest implementation occupied 214 slices, but the clock frequency was decreased to 160.88 MHz.

Synthesis results have also shown that automatic settings do not always produce optimal results. Manual adjustments and several design iterations are necessary if the automatic settings fail to produce a circuit meeting the design objectives, and a better implementation is required. Even then, a universally optimal implementation cannot be guaranteed for large designs because computational power and time restrictions prohibit an exhaustive search for the best implementation within all possible implementations.

ÖZET

LOJİK DEVRE SENTEZİNDE KULLANILAN OPTİMİZASYON YÖNTEMLERİ VE SENTEZ SONUÇLARININ OPTİMALLİĞİNİN İNCELENMESİ

1960'lardan itibaren tümdevre üretim teknolojisinde yaşanan hızlı gelişme, bir çip üzerine sığdırılabilen transistör sayısını her 1.5 ila 2 yılda bir ikiye katlamış ve devre karmaşıklıkları kalem kağıda dayalı eski tasarım yöntemleriyle altından kalkılamayacak düzeye gelmiştir. Bu sıkıntılara paralel olarak, bir donanım tanımlama dilinde yazılmış davranışsal tanımlamadan yola çıkarak lojik devre sentezi yapabilen yeni elektronik tasarım programları ortaya çıkmıştır.

Bu tez çalışması, lojik devrelerin sentezlenmesini ele almakta ve sentez programları tarafından kullanılan optimizasyon yöntemlerini incelemektedir. Ayrıca, Verilog dilinde yazılmış örnek bir devre, Spartan-3E FPGA üzerinde gerçeklenmek üzere Xilinx firmasının ISE 8.2i geliştirme ortamının parçası olan XST sentez programında, değişik optimizasyon ayarları denenmek suretiyle sentezlenmiştir. Denemeler sonucunda elde edilen devreler boyut ve maksimum çalışma hızı açılarından birbirleriyle karşılaştırılarak yazılımdaki optimizasyon ayarlarının sentez performansını nasıl etkilediği araştırılmıştır.

XST'de mevcut ayarların küçük bir kısmıyla, oluşturulabilecek olası bütün kombinezonlar denenerek sentez işlemi defalarca tekrarlanmış, sonuçta elde edilebilen en hızlı devrenin saat frekansı üst sınırı 186.78 MHz olmuş ve devre FPGA'de 235 dilimlik yer kaplamıştır. Buna karşın en küçük boyutlu devre 214 dilime sığarken, maksimum saat frekansı ise 160.88 MHz'e gerilemiştir.

Yapılan denemeler sonunda, sentez programı ayarlarını otomatik değerlerde bırakmanın her zaman optimal sonuç vermediği de görülmüştür. Tasarım hedeflerini karşılayan bir devreye otomatik ayarlarda ulaşılamazsa, tasarımcının yazılım ayarlarını bizzat kendisinin seçmesi gerekmektedir. O durumda dahi, büyük boyutlu tasarımlarda sentez sonucu elde edilecek devrenin gerçekten optimal bir devre olduğu garanti edilememekte, bilgi işlem kapasitesinin ve zamanın sınırlı oluşu yüzünden, olası bütün gerçeklemeler arasından evrensel en iyiyi arayıp bulmak imkansız hale gelmektedir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ADC	Analog to Digital Converter
ALU	Arithmetic Logic Unit
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
BGA	Ball-Grid Array
BIST	Built-in Self-Test
CAD	Computer Aided Design
CAE	Computer Aided Engineering
CLB	Configurable Logic Block
CMOS	Complementary Metal-Oxide Semiconductor
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
DAC	Design Automation Conference / Digital to Analog Converter
DCE	Data Communications Equipment
DCM	Digital Clock Manager
DDR	Double Data Rate
DIP	Dual In-Line Package
DOD	Department of Defense
DSP	Digital Signal Processing/Processor
DTE	Data Terminal Equipment
DUT	Device Under Test
EDA	Electronic Design Automation
EEPROM	Electrically Erasable Programmable Read-Only Memory
EPROM	Erasable Programmable Read-Only Memory

FBGA	Fine Ball-Grid Array
FFT	Fast Fourier Transform
FIFO	First-In First-Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
HSTL	High-Speed Transceiver Logic
I/O	Input/Output
IC	Integrated Circuit
IOB	Input/Output Block
IEEE	Institute of Electrical and Electronics Engineers
IP	Intellectual Property
JEDEC	Joint Electronic Device Engineering Council
JEIDA	Japan Electronic Industry Development Association
JTAG	Joint Test Action Group
LCD	Liquid Crystal Display
LED	Light Emitting Diode
LFSR	Linear Feedback Shift Register
LRM	Language Reference Manual
LSB	Least Significant Bit
LSI	Large-Scale Integration
LSSD	Level-Sensitive Scan Device
LUT	Look-Up Table
LVCMOS	Low Voltage CMOS
LVDS	Low Voltage Differential Signaling
LVTTL	Low Voltage TTL

MAC	Medium Access Control / Multiply and Accumulate
MSB	Most Significant Bit
MSI	Medium Scale Integration
NRE	Non-Recurring Engineering
OVI	Open Verilog International
PAL	Programmable Array Logic
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
PLA	Programmable Logic Array
PLD	Programmable Logic Device
PLI	Programming Language Interface
POS	Product of Sums
PROM	Programmable Read-Only Memory
QFP	Quad Flat Pack
RAM	Random Access Memory
ROM	Read Only Memory
RSDS	Reduced Swing Differential Signaling
RTL	Register Transfer Level
SDF	Standard Delay Format
SDRAM	Synchronous Dynamic Random Access Memory
SHA	Secure Hash Algorithm
SMA	Sub-Miniature version A (Connector)
SOC	System On A Chip
SOP	Sum of Products
SOPC	System On A Programmable Chip
SPI	Serial Peripheral Interface

SPICE	Simulation Program with Integrated Circuit Emphasis
SPLD	Simple Programmable Logic Device
SRAM	Static RAM
SSTL	Stub-Series-Terminated Logic
TTL	Transistor-to-Transistor Logic
UDL/I	Unified Design Language for Integrated Circuits
USB	Universal Serial Bus
UV	Ultra-Violet
VGA	Video Graphics Array
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VI	VHDL International
VITAL	VHDL Initiative Toward ASIC Libraries
VLSI	Very-Large-Scale Integration

1. INTRODUCTION

There has been a dramatic change in the way digital design is performed in the past 20 years. The technological advances and new innovative electronic design automation (EDA) software tools, together with the need to reduce the time to market, have all helped to fuel this dramatic change. In terms of technology, current mass-produced integrated circuits (ICs) have reached transistor channel widths as small as 65nm, while 45nm and smaller width prototypes have been demonstrated. Smaller transistors result in improved switching speed and increase the number of transistors that can be fitted per unit area. This size reduction has enabled the production of silicon chips containing over one billion transistors on a single die (Intel's "Montecito" Itanium 2 server processor, officially launched on July 18, 2006, is built using a 90nm process technology and has 1.72 billion transistors on a 596mm^2 die [1]).

The need to be able to design chips of such size, and in a timely manner, has led to innovative EDA tools being developed, with automatic synthesis tools being the major advance. Synthesis is the process of translating a high-level, implementation independent, abstract model of hardware into an optimized, technology specific, gate level implementation. These high-level models are written in a hardware description language (HDL) using algorithmic definitions comprising a sequence of operations on data being transferred between various registers. This is hence called register transfer level (RTL) modeling.

The introduction of commercial synthesis tools has enabled top down design methodologies to be adopted, starting with an abstract description of a circuit's behavior written in a hardware description language. The main benefits of adopting a top-down design methodology, and adhering to the use of these standards is that, 1) design source files are transportable between different EDA tools and, 2) the design is independent of any particular silicon vendor's manufacturing process technology. More recently, the rate of change has slowed and the introduction of standards has enabled EDA software vendors to develop integrated design tools with far less risk [2].

1.1. Thesis Outline

This thesis focuses on the synthesis of logic circuits from circuit descriptions written in a hardware description language, with emphasis on the optimization techniques performed by synthesis tools during the process.

Chapter Two gives an overview of the current technologies used for the realization of logic circuits. The target technology for a design heavily influences the way synthesis tools construct and optimize the physical circuit.

Chapter Three relates the evolution of electronic design automation software, the emergence of hardware description languages and the reasons that led to their development. This chapter also covers some of the widely known hardware description languages in use today.

Chapter Four introduces the fundamentals of the Verilog hardware description language with its code syntax, language constructs, operators and various circuit modeling techniques supported by synthesis tools.

Chapter Five examines various optimization techniques employed by synthesis tools, the effects of coding style on synthesis results, elaborates on how to write synthesis friendly code and points out some common coding mistakes that lead to synthesis of unintentional redundant logic.

Chapter Six presents a sample field programmable gate array (FPGA) implementation of a circuit developed for Xilinx's Spartan-3E Starter Kit FPGA development board. The circuit description is written in Verilog and synthesized by Xilinx's ISE 8.2i development software, with various optimization settings enabled or disabled in its XST synthesis tool. The resulting circuits are then compared in terms of device utilization and operating speed.

Chapter Seven concludes this work with a review of the results achieved and also gives some pointers on the future of electronic design software and methodologies.

2. ALTERNATIVE TECHNOLOGIES FOR IMPLEMENTATION OF LOGIC CIRCUITS

As designs grow more complex, synthesizing a high performance logic circuit that also satisfies the associated area, timing and power constraints becomes more and more difficult. Without an intimate knowledge of the target technology on which the circuits will be implemented, successful synthesis results cannot be achieved. This is why synthesis tools have become “physically aware” in recent years [3].

Below is an overview of the various technologies and devices that have been used over the years for realization of digital circuits, followed by the current state-of-the-art in logic implementation.

2.1. Standard Parts

The 74xx, 54xx and similar logic family ICs are called standard parts. They contain a small number of basic logic gates and the number code of each IC states the type of logic gates contained within the chip. For example, a 7400 chip contains four two-input NAND gates, a 7404 chip contains six inverters.

Implementing complex logic circuits with standard parts is very difficult and expensive, so they are only used in small scale, simple circuits. Standard parts have largely been replaced by programmable logic devices.

2.2. Simple Programmable Logic Devices (SPLDs)

Programmable logic devices (PLDs) have progressed through a long evolution to reach the complexity today to support an entire system on a chip (SOC). In order to distinguish the older generation of these devices, they are now generally referred to as Simple PLDs.

2.2.1. Programmable Read-Only Memories (PROMs)

These first field programmable devices were created as alternatives to expensive mask-programmed ROMs. Storing code in a ROM was an expensive process that required the ROM vendor to create a unique semiconductor mask set for each customer. Changes to the code were impossible without creating a new mask set and fabricating a new chip. The lead time for making changes to the code and getting back a chip to test was far too long. PROMs solved this problem by allowing the user, rather than the chip vendor, to store code in the device using a simple and relatively inexpensive desktop programmer. This new device was called a programmable read only memory (PROM). PROMs, like ROMs, retain their contents even after power has been turned off.

Eventually, erasable PROMs were developed which allowed users to program, erase, and reprogram the devices using an inexpensive, desktop programmer. Typically, PROMs now refer to devices that cannot be erased after being programmed. Erasable PROMS include erasable programmable read only memories (EPROMs) that are programmed by applying high-voltage electrical signals and erased by flooding the devices with ultra-violet (UV) light. Electrically erasable programmable read only memories (EEPROMs) are programmed and erased by applying high voltages to the device. Flash EPROMs are programmed and erased electrically and have sections that can be erased electrically in a short time and independently of other sections within the device.

Although PROMs were initially intended for storing code and constant data, design engineers also used them for implementing Boolean functions (BFs). Figure 2.1 shows the structure of a simple PROM.

PROMs consist of a fixed array of AND gates driving a programmable array of OR gates. They can be used for implementing m Boolean functions of n variables in sum-of-products canonical form, where n is the number of address lines the PROM has, and m the number of PROM outputs. Thus, the PROM has $2^n \times m$ bits of storage capacity [4]. The PROM is programmed as if it was a simple truth table, such that each data output bit has the corresponding value of the Boolean function.

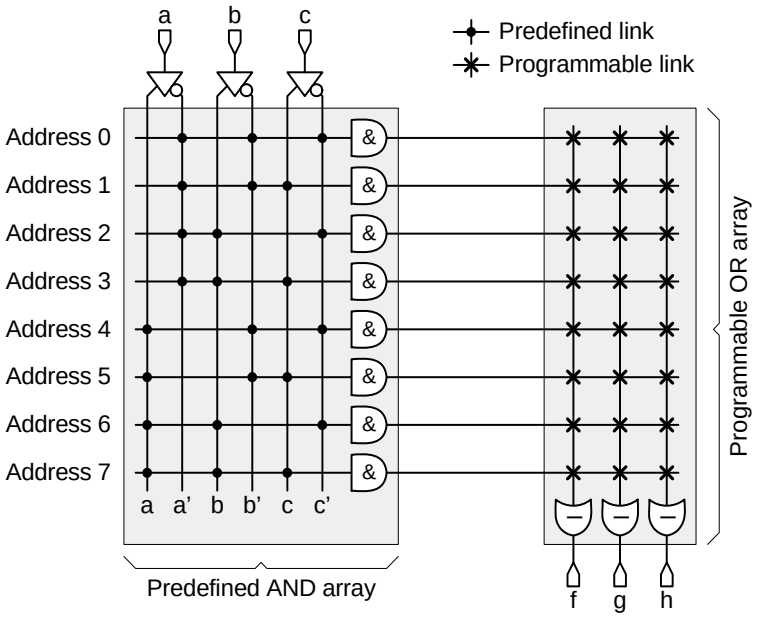


Figure 2.1. The structure of a PROM

For sequential logic, external clocked devices such as flip-flops or microprocessors were added to the design. State machine logic was programmed into a PROM, which combined inputs with bits representing the current state of the machine, to produce outputs and the next state of the machine. This allowed the creation of very complex state machines. Another benefit was that these state machines could be easily reprogrammed in order to fix bugs, test new functions, optimize existing designs, or make changes to systems that were already shipped and in the field.

The problem with PROMs is that they tend to be extremely slow (even today, access times are on the order of 40 nanoseconds or more), so they are not useful for applications where speed is an issue. Also, PROMs require a different manufacturing technology than for logic circuits, and thus, integrating PROMs onto a chip with logic circuitry involves extra masks and extra processing steps, all leading to extra costs.

2.2.2. Programmable Logic Arrays (PLAs)

Programmable logic arrays (PLAs) were a solution to the speed and input limitations of PROMs. Ron Cline from Signetics (which was later purchased by Philips and then

eventually Xilinx) came up with the idea of two programmable planes [5]. PLAs consist of a large number of inputs connected to an AND plane, where different combinations of signals can be logically ANDed together according to how the part is programmed. The outputs of the AND plane go into an OR plane, where the terms are ORed together in different combinations and finally outputs are produced (Figure 2.2). There are inverters at the inputs (and sometimes also at the outputs) so that logical NOTs can be obtained. These devices can implement a large number of Boolean functions in reduced sum-of-products (SOP) form, but, unlike a PROM, they can't implement every possible combination of their inputs within the AND plane. However, they generally have many more inputs and are much faster PROMs.

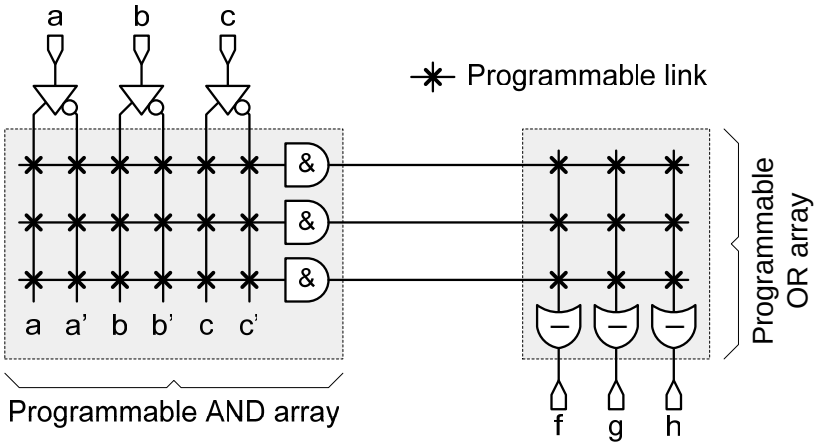


Figure 2.2. Structure of a PLA

As with PROMs, PLAs could be connected externally to flip-flops to create state machines, which are the essential building blocks for all sequential logic.

Each connection in the AND and OR planes of a PLA could be programmed to connect or disconnect. In other words, terms of Boolean functions could be created by selectively connecting wires within the AND and OR planes. Simple high level languages were developed to convert Boolean functions such as $f = (x \& y) | (!x \& !y \& z)$ into files that would program these connections within the PLA. This added a new dimension to programmable devices in that logic could now be described in readable programs at a level higher than groups of ones and zeroes.

The PLA architecture was very flexible, but at the time, wafer geometries of 10 μm made the input-to-output delays high, which made these devices relatively slow.

2.2.3. Programmable Array Logic (PAL)

The programmable array logic (PAL) is a variation of the PLA. MMI (later purchased by AMD) was enlisted as a second source for the PLA array. After several fabrication issues, it was modified to become the PAL architecture by fixing one of the programmable planes [5].

Like the PLA, it has a wide, programmable AND plane for ANDing inputs together. Programming elements at each intersection in the AND plane allow perpendicular traces to be connected or left open, creating product terms, which are multiple logical signals ANDed together. The product terms are then ORed together (Figure 2.3).

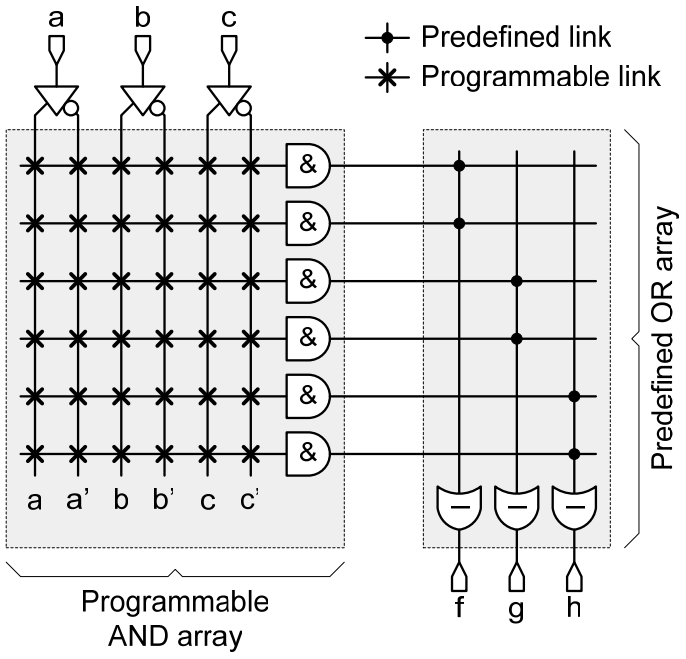


Figure 2.3. Structure of a PAL

In a PAL, unlike a PLA, the OR plane is fixed, limiting the number of terms that can be ORed together. This still allows a large number of Boolean functions to be implemented.

The reason for this can be demonstrated by DeMorgan's Law, which states that $a+b=\overline{\left(\overline{a}\cdot\overline{b}\right)}$. That means if inverters are used on the inputs and outputs, any logic function can be created with either a wide AND plane or a wide OR plane; there's no need to have both.

Including inverters reduced the need for the large OR plane, which in turn allowed the extra silicon area on the chip to be used for other basic logic devices such as multiplexers, exclusive ORs, and latches. Most importantly, clocked elements, typically flip-flops, could be included in PALs. These devices were now able to implement a large number of logic functions, including clocked sequential logic needed for state machines. This was an important development that allowed PALs to replace much of the standard logic in many designs. PALs are also extremely fast, allowing high-speed controllers be to be implemented in programmable logic.

The inclusion of extra logic devices, particularly flip-flops, greatly increased the complexity and potential uses of PALs, creating a need for new methods of programming that were flexible and readable. Thus the first hardware description languages (HDLs) were born. These simple HDLs included ABEL, CUPL, and PALASM, the precursors of Verilog and VHDL. These programming languages also allowed the use of simulation test vectors in the code. This simulation capability brought better reliability and verification of programmable devices, something that was critical when CPLDs and FPGAs were developed.

2.3. Application Specific Integrated Circuits (ASICs)

Application specific integrated circuits (ASICs) are chips that are custom designed and built to order for use in a specific application. ASICs can be implemented either as a full-custom design, or in a gate array device or a standard cell device [6].

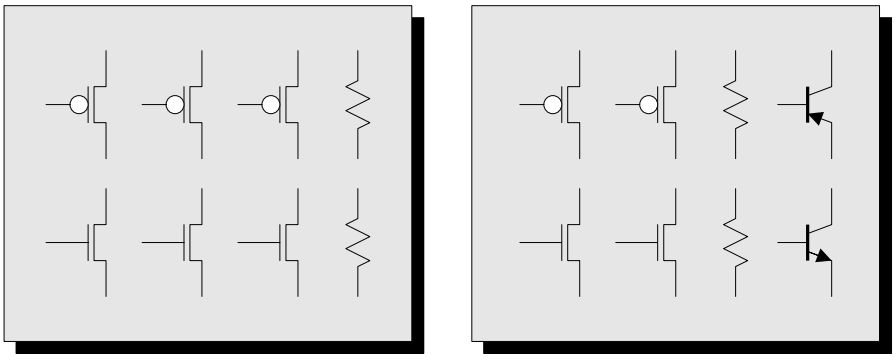
2.3.1. Full-Custom Devices

In the case of full-custom devices, design engineers have complete control over every mask layer used to fabricate the silicon chip. The ASIC vendor does not prefabricate

any components on the silicon and does not provide any libraries of predefined logic gates and functions. By using design software, the engineers can manually specify the dimensions of individual transistors and then create higher-level functions based on these elements. For example, if the engineers require a slightly faster logic gate, they can alter the dimensions of the transistors used to build that gate. The design of full-custom devices is highly complex and time-consuming, but the resulting chips contain the maximum amount of logic with minimal waste of silicon area.

2.3.2. Gate Arrays

Gate arrays are based on the idea of a basic cell, shown in Figure 2.4, consisting of a collection of unconnected transistors and resistors. Each ASIC vendor determines what it considers to be the optimum mix of components provided in its particular basic cell.



(a) Pure CMOS basic cell (b) BiCMOS basic cell

Figure 2.4. Examples of gate array basic cells

The ASIC vendor manufactures many unrouted die that contain the arrays of gates as shown in Figure 2.5. These can be used for any gate array customer. In the case of channeled gate arrays, the basic cells are typically presented as either single-column or dual-column arrays. The free areas between the arrays are known as the channels and are used for routing the connections between cells.

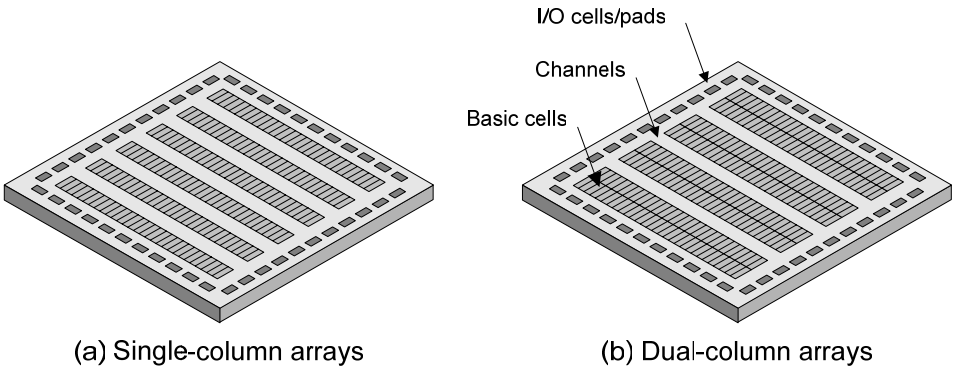


Figure 2.5. Channeled gate array architectures

By comparison, in the case of channel-less or channel-free devices, the basic cells are presented as a single large array (Figure 2.6). The surface of the device is covered in a "sea" of basic cells, and there are no dedicated channels for the interconnections. Thus, these devices are popularly referred to as sea-of-gates or sea-of-cells.

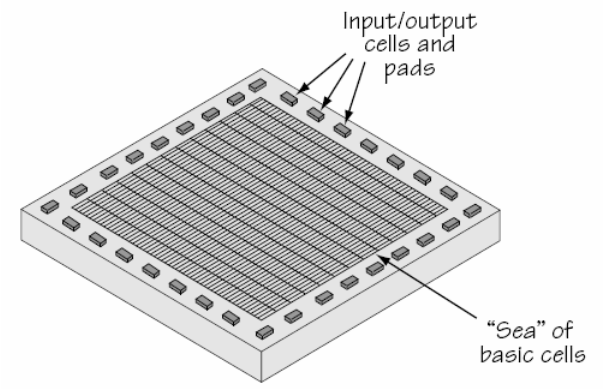


Figure 2.6. Sea-of-gates architecture

An integrated circuit consists of many layers of materials, including semiconductor material (e.g., silicon), insulators (e.g., oxides), and conductors (e.g., metal). An unrouted die is processed with all of the layers except for the final metal layers that connect the gates together. The ASIC vendor defines a set of logic functions such as primitive gates, multiplexers, and registers that can be used by the design engineers. Each of these building block functions is referred to as a cell, not to be confused with a basic cell, and the set of functions supported by the ASIC vendor is known as the cell library.

Once given a design, the layout software assigns the functions selected by the engineers to basic cells on the silicon (Figure 2.7), and figures out which transistors to connect by placing metal connections on top of the die. First, the low level functions are connected together. For example, six transistors could be connected to create a D-type flip-flop. These six transistors would be located physically very close to each other. After the low level functions have been routed, they would in turn be connected together. The software would continue this process until the entire design is complete.

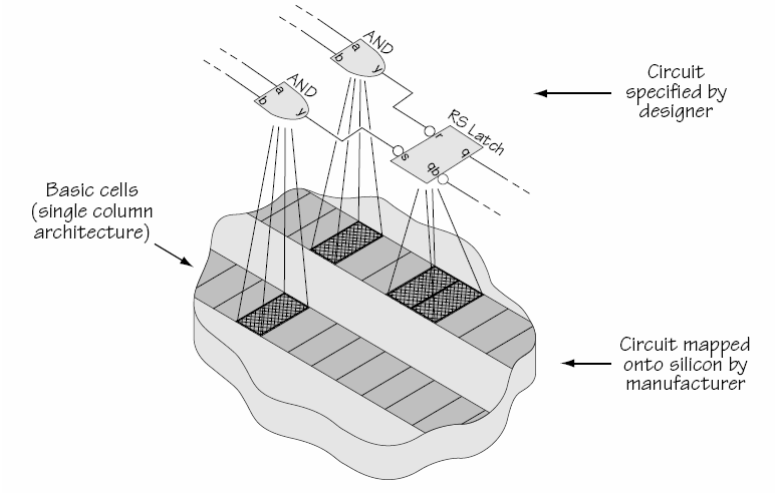


Figure 2.7. Assigning functions to basic cells

The channels in channeled devices are used for the tracks that connect the logic gates together. By comparison, in channel-less devices, the connections between logic gates have to be deposited over the top of other basic cells. In the case of early processes based on two layers of metallization, any basic cell overlaid by a track was no longer available to the user. Due to limited space in the routing channels of channeled devices, the design engineers could only actually use between 70% and 90% of the total number of available gates. Sea-of-gates architectures provided significantly more available gates than those of channeled devices because they don't contain any dedicated wiring channels. However, in practice, only about 40% of the gates were usable in devices with two layers of metallization, rising to 60% or 70% in devices with three or four metallization layers, and even higher as the number of metallization layers increases [7].

When the design is complete, the vendor simply needs to add the last metal layers to the die to create the final chip, using photo masks for each metal layer. For this reason, it is sometimes referred to as a “masked gate array” to differentiate it from a field programmable gate array.

The advantage of a gate array is that the internal circuitry is very fast; the circuit is dense, allowing lots of functionality on a die; and the cost is low for high volume production. The disadvantage is that most designs leave significant amounts of internal resources unutilized, the placement of gates is constrained, and the routing of internal tracks is less than optimal. All of these factors negatively impact the performance and power consumption of the design [8]. Also, it takes time for the ASIC vendor to manufacture and test the parts. The customer incurs a large charge up front, called a non-recurring engineering (NRE) expense, which the ASIC vendor charges to begin the entire ASIC manufacturing process. And if there's a mistake, it's a long, expensive process to fix it and manufacture new ASICs.

2.3.3. Standard Cell Devices

Standard cell devices bear many similarities to gate arrays. The ASIC vendor defines the cell library that can be used by the design engineers. The vendor also supplies hard-macro and soft-macro libraries, which include elements such as processors, communication functions, and a selection of RAM and ROM functions. Last but not least, the design engineers may decide to reuse previously designed functions or to purchase blocks of intellectual property (IP).

The design software synthesizes the circuit into a gate-level netlist, which describes the logic gates and the connections between them. However, unlike gate arrays, standard cell devices do not use the concept of a basic cell, and no components are prefabricated on the chip. Special tools are used to place each logic gate individually in the netlist and to determine the optimum way in which the gates are to be routed. The results are then used to create custom photo-masks for every layer in the device’s fabrication.

The standard cell concept allows each logic function to be created using the minimum number of transistors with no redundant components, and the related functions can be positioned close together so as to easily route any connections between them. Standard cell devices, therefore, provide a closer-to-optimal utilization of the silicon than do gate arrays.

2.4. Complex Programmable Logic Devices (CPLDs)

Complex Programmable Logic Devices (CPLDs) contain a large number of SPLD-like blocks (PAL or PLA) in a single chip, connected to each other through a programmable interconnect matrix so that, in addition to programming the individual SPLD blocks, the connections between the blocks can also be configured. This architecture made them familiar to their target market, the printed circuit (PC) board designers who were already designing SPLDs in their boards. CPLDs were used to simply combine multiple SPLDs in order to save real estate on a PC board. CPLDs use the same development tools and programmers as SPLDs, and are based on the same technologies as SPLDs, but they can realize much more complex logic and more of it.

Figure 2.8 shows the internal architecture of a generic CPLD. Although each manufacturer has a different variation, in general they are all similar in that they consist of SPLD-like function blocks, input/output blocks, and an interconnection matrix.

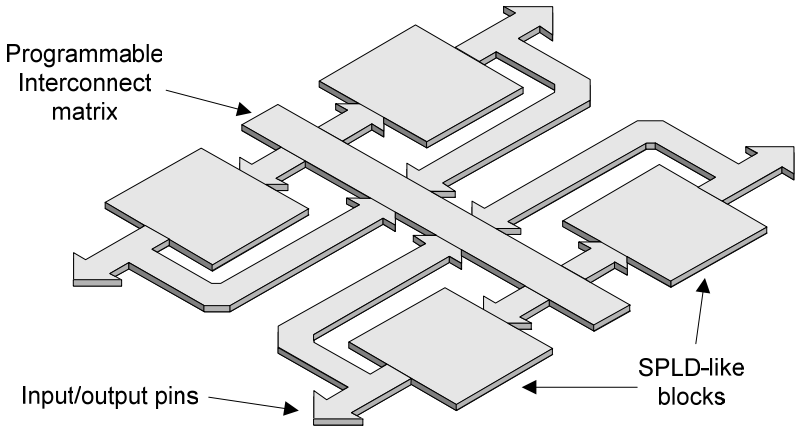


Figure 2.8. A Generic CPLD Architecture

2.5. Field Programmable Gate Arrays (FPGAs)

Around the beginning of the 1980s, it became apparent that there was a gap in the digital IC implementation technologies. At one end, there were programmable devices like SPLDs and CPLDs, which were highly configurable and had fast design and modification times, but which couldn't support large or complex functions. At the other end of the spectrum were ASICs. These could support extremely large and complex functions, but they were quite expensive and time consuming to design. Furthermore, once a design had been implemented as an ASIC it was impossible to modify the design or correct errors.

In order to address this gap, Xilinx developed a new class of ICs called a field-programmable gate array (FPGA), which was made available to the market in 1984.

The early FPGA devices were based on the concept of a configurable logic block (CLB) that comprised a 3-input lookup table (LUT), a register that could act as a flip-flop or a latch, and a multiplexer, along with a few other elements, as shown in Figure 2.9.

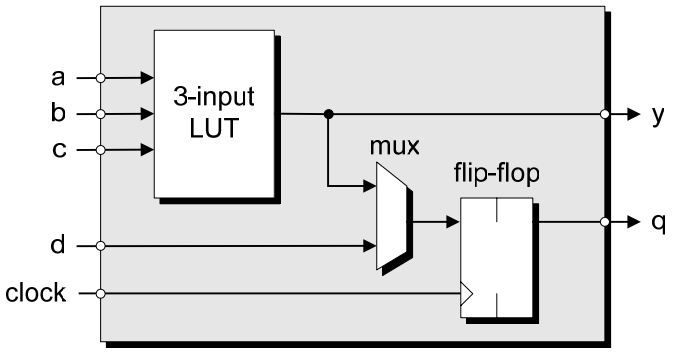


Figure 2.9. The structure of a basic FPGA configurable logic block

Each FPGA contains a large number of these programmable logic blocks. By means of appropriate Static RAM (SRAM) programming cells, every logic block in the device can be configured to perform a different function. Each register can be configured to initialize containing a logic 0 or a logic 1 and to act either as a flip-flop or a latch. As a flip-flop, the register can be configured to be triggered by a rising or falling edge clock. The multiplexer feeding the flip-flop can be configured to accept the output from the LUT or a separate input to the logic block.

The LUT can be configured to represent any 3-variable Boolean function. For example, to implement the function *y = ab + c'*, the LUT is loaded with the appropriate output values (Figure 2.10).

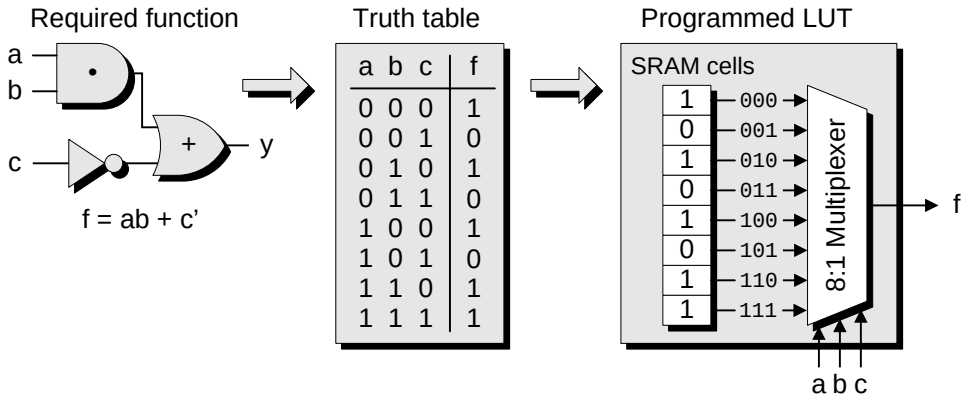


Figure 2.10 Implementation of a Boolean function in a look-up table

Although the structure varies from vendor to vendor, a typical FPGA comprises a large number of programmable logic blocks surrounded by a network of programmable interconnects, as shown in Figure 2.11.

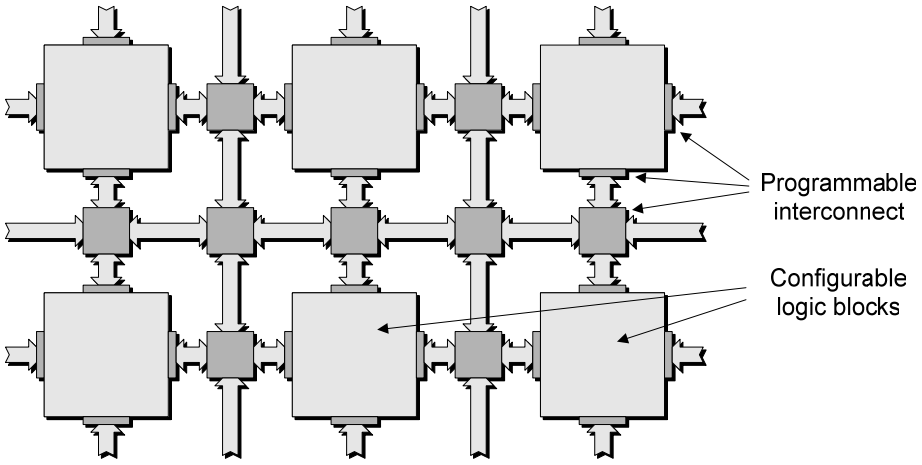


Figure 2.11. A generic FPGA architecture

In addition to the local interconnect, there are also global (high speed) interconnection paths for transporting time-critical signals across the chip without having to go through multiple switching elements. There are also clock manager circuits that

condition the incoming clock signals and then drive the clock trees throughout the device, ensuring synchronous operation of flip-flops (Figure 2.12).

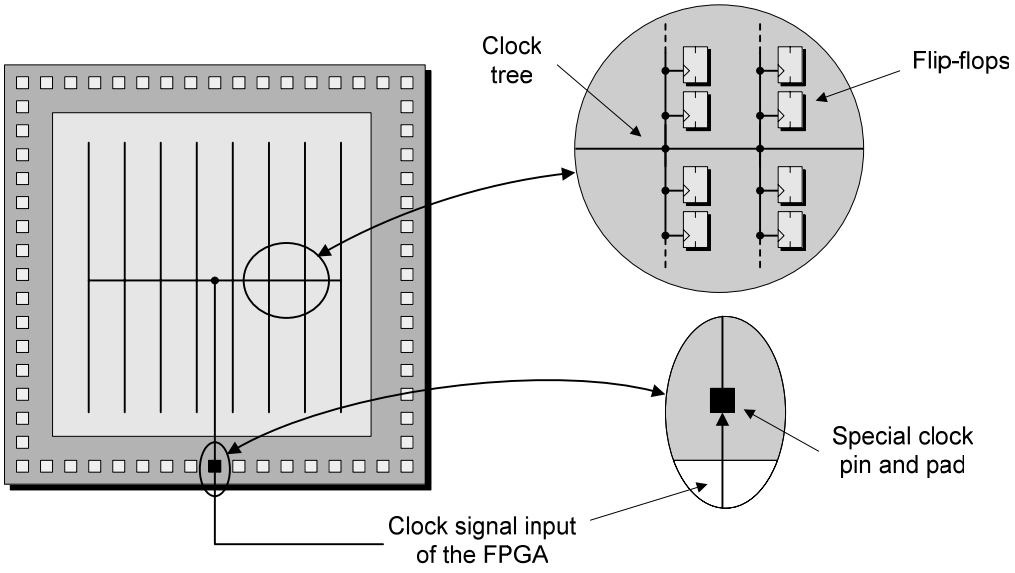


Figure 2.12. The clock tree within an FPGA

FPGAs also include input/output (I/O) blocks that drive the I/O pins on the chip. By programming the interconnect, the inputs to the device can be connected to the inputs of one or more programmable logic blocks, and the outputs from any logic block can be used to drive the inputs to any other logic block, the outputs from the device, or both.

2.5.1. Fine-Grained vs. Coarse-Grained CLBs

In theory, there are two types of CLBs, depending on the amount and type of logic that is contained within them. These two types are called “large grain” and “fine grain”.

In a large grain FPGA, the CLB contains larger functionality logic. For example, it can contain two or more flip-flops. A design that does not need many flip-flops will leave many of these flip-flops unused, poorly utilizing the logic resources in the CLBs and in the chip. A design that requires lots of combinational logic will be required to use up the LUTs in the CLBs while leaving the flip-flops untouched.

Fine grain FPGAs resemble ASIC gate arrays in that the CLBs contain only small, very basic elements such as NAND gates, NOR gates, etc. The philosophy is that small elements can be connected to make larger functions without wasting too much logic. If a flip-flop is needed, one can be constructed by connecting NAND gates. If it's not needed, then the NAND gates can be used for other features. In theory, this apparent efficiency seems to be an advantage. Also, because they more closely resemble ASICs, conversion of a design from an FPGA to an ASIC is much more straightforward.

However, in practice, one key fact renders the fine grain architecture less useful and less efficient. The routing resources are the bottleneck in any FPGA design in terms of utilization and speed [9]. It is often difficult to connect CLBs together using the limited routing resources on the chip. Also, in an FPGA, unlike an ASIC, the majority of the delay comes from routing, not logic. In an ASIC, signals are routed using metal layers, leading to RC delays that are insignificant with respect to the delay through logic gates. In an FPGA, the routing is done through programmed multiplexers in the case of SRAM-based devices and through conducting vias in the case of antifuse devices. Both of these structures add significant delay to a signal. The multiplexers have a gate delay associated with them. The conducting vias have a high resistance, causing greater RC delay. Fine grain architectures require many more routing resources, which take up space and insert a large amount of delay, which can hinder the performance of a design, in spite of the better CLB utilization. This is why all FPGA vendors currently use large grain architectures for their CLBs.

2.5.2. Embedded Devices

Many newer FPGA architectures incorporate complex devices inside the FPGA fabric. These devices range from relatively simple functions, such as address decoders or multipliers, all the way through arithmetic & logic units (ALUs), digital signal processors (DSPs), microprocessors and microcontrollers. These embedded devices are optimized and already tested, just like a standalone chip, so there's no need to design the circuit from scratch and verify its functionality. FPGAs with embedded devices offer the possibility of integrating an entire system onto a single chip, creating what is called a "system on a programmable chip" (SOPC), providing significant savings in printed circuit board area and power consumption.

One disadvantage of these devices is that designs implemented in them are tied into a single FPGA from a single FPGA vendor, losing some of the portability that engineers prefer. Each vendor has specific devices embedded into their FPGAs. In the case of embedded processors, each FPGA vendor usually licenses a specific processor core from a different processor manufacturer. This is good for the FPGA vendor because once they have a design win, that design is committed to their FPGA for some time.

By providing an embedded device that is in demand or soon will be, smaller FPGA vendors try to differentiate themselves and create a niche market for their devices, allowing them to compete with the bigger vendors.

3. EVOLUTION OF ELECTRONIC DESIGN AUTOMATION

In the early 1960s, electronic circuits were crafted by hand. Circuit diagrams, also known as schematic diagrams or just schematics, were hand-drawn using pen, paper, and stencils. These diagrams showed the symbols for the logic gates and functions that were to be used to implement the design, along with the connections between them. Similarly, the copper tracks on a circuit board were drawn using red and blue pencils to represent the top and bottom of the board [10].

Each design team usually had at least one member who was really good at performing logic minimization and optimization. Functional verification, i.e. checking that the design would work as planned insofar as its logical implementation, was typically performed by a group of engineers by manually working through the schematics. Similarly, timing verification, checking that the design met its required input-to-output and internal path delays and that no parameters associated with the timing of any of the internal registers (such as setup and hold times) were violated, was performed using pencil and paper and mechanical calculators.

Finally, a set of drawings representing the structures used to form the logic gates (or, more accurately, the transistors forming the logic gates) and the interconnections between them were drawn by hand. These drawings, which were formed from groups of simple polygons such as squares and rectangles, were subsequently used to create the photo-masks, which were themselves used to create the actual silicon chip.

3.1. The Early Days of Electronic Design Automation

Not surprisingly, the handcrafted way of designing circuits was time-consuming and prone to error. Something had to be done, and a number of companies and universities began research in a variety of different directions. In the case of functional verification, for example, special computer programs known as simulators were developed. These programs allowed students and engineers to emulate the operation of an electronic circuit without actually having to build it first. Perhaps the most famous of the early simulators was the Simulation Program with Integrated Circuit Emphasis (SPICE) [11]. SPICE was developed

by the University of California in Berkeley and was made available for widespread use around the beginning of the 1970s. SPICE was designed to simulate the behavior of analog circuits. Other programs called logic simulators were developed to simulate the behavior of digital circuits.

In order to use the logic simulator, the engineers first needed to create a textual representation of the circuit called a gate-level netlist. In those times, the engineers would typically have been using a mainframe computer, and the netlist would have been captured as a set of punched cards called a deck. As computers (along with storage devices like hard disks) became more accessible, netlists began to be stored as text files.

It was also possible to associate delays with each logic gate. These delays were typically referenced as integer multiples of some core simulation time unit.

All of the early logic simulators had internal representations of primitive gates like AND, NAND, OR, NOR, etc. These were referred to as simulation primitives. Some simulators also had internal representations of more sophisticated functions like D-type flip-flops. Alternatively, one could create a subcircuit called DFF to represent a D-type flip-flop, whose functionality was captured as a netlist of primitive AND, NAND, etc. gates. In this case, DFF would actually be seen by the simulator as a call to instantiate a copy of this subcircuit.

Next, the user would create a set of test vectors, also called stimulus, which were patterns of logic 0 and logic 1 values to be applied to the circuit's inputs. Once again, these test vectors were textual in nature, and they were typically presented in a tabular form. In today's terminology, the file of test vectors would be considered a rudimentary testbench. Once again, time values were typically specified as integer multiples of some core simulation time unit.

The engineer would then invoke the logic simulator, which would read in the gate-level netlist and construct a virtual representation of the circuit in the computer's memory. The simulator would then read in the first test vector (the first line from the stimulus file), apply those values to the appropriate virtual inputs, and propagate their effects through the circuit. This would be repeated for each of the subsequent test vectors forming

the testbench. The simulator would also use one or more control files (or online commands) to tell it which internal nodes (wires) and output pins to monitor, how long to simulate for, and so forth. The results, along with the original stimulus, would be stored in tabular form in a textual output file.

It wasn't long before engineers were working with circuits that could contain thousands of gates and internal nodes along with simulation runs that could encompass thousands of time steps. It took hours to examine output files trying to see if a circuit was working as expected, and also attempting to track down the problem if it wasn't.

3.2. Back-End Tools

As opposed to tools like logic simulators that were intended to aid the engineers who were defining the function of ICs (and circuit boards), some companies focused on creating tools that would help in the process of laying the ICs out. In this context, "layout" refers to determining where to place the logic gates (actually, the transistors forming the logic gates) on the surface of the chip and how to route the wires between them.

In the early 1970s, companies like Calma, ComputerVision, and Applicon created special computer programs that helped personnel in the drafting department capture digital representations of hand-drawn designs. In this case, a design was placed on a large-scale digitizing table, and then a mouse-like tool was used to digitize the boundaries of the shapes (polygons) used to define the transistors and the interconnect. These digital files were subsequently used to create the photo-masks, which were themselves used to create the actual silicon chip.

Over time, these early computer-aided drafting tools evolved into interactive programs called polygon editors that allowed users to draw the polygons directly onto the computer screen. Other companies like Racal-Redac, SCI-Cards, and Telesis created equivalent layout programs for printed circuit boards. Descendants of these tools eventually gained the ability to accept the same netlist used to drive the logic simulator and to perform the layout (place-and-route) tasks automatically.

3.3. Computer-Aided Design and Engineering

Tools that were used in the front-end (logical design capture and functional verification) portion of the design flow were originally gathered together under the umbrella name of computer-aided engineering (CAE). By comparison, tools like layout (place-and-route) that were used in the back-end (physical) portion of the design flow were originally gathered together under the name of computer-aided design (CAD).

For historical reasons that are largely based on the origins of the terms CAE and CAD, the term design engineer, or simply engineer, typically refers to someone who works in the front-end of the design flow; that is, someone who performs tasks like conceiving and describing (capturing) the functionality of an IC (what it does and how it does it). By comparison, the term layout designer, or simply designer, commonly refers to someone who works at the back-end of the design flow; that is, someone who performs tasks such as laying out an IC (determining the locations of the gates and the routes of the tracks connecting them together). During the 1980s, all of the CAE and CAD tools used to design electronic components and systems were gathered under the name of electronic design automation (EDA).

Toward the end of the 1970s and the beginning of the 1980s, companies like Daisy, Mentor, and Valid started providing graphical schematic capture programs that allowed engineers to create circuit (schematic) diagrams interactively. Using the mouse, an engineer could select symbols representing such entities as I/O pins and logic gates and functions from a special symbol library and place them on the screen. The engineer could then use the mouse to draw lines (wires) on the screen connecting the symbols together.

Once the circuit had been entered, the schematic capture package could be instructed to generate a corresponding gate-level netlist. This netlist could first be used to drive a logic simulator in order to verify the functionality of the design. The same netlist could then be used to drive the place-and-route software.

3.4. Hardware Description Languages

Toward the end of the 1980s, as designs grew in size and complexity, schematic based ASIC flows became a limiting factor. Visualizing, capturing, debugging, understanding, and maintaining a design at the gate level of abstraction became increasingly difficult and inefficient when dealing with 5,000 or more gates and a vast number of schematic pages.

In addition to the fact that capturing a large design at the gate level of abstraction is prone to error, it is also extremely time-consuming. Thus, some EDA vendors started to develop design tools and flows based on the use of hardware description languages (HDLs).

In a wider context, the term hardware is used to refer to any of the physical portions of an electronics system, including the ICs, printed circuit boards, cabinets, cables, and even the nuts and bolts holding the system together. In the context of an HDL, however, “hardware” refers only to the electronic portions (components and wires) of ICs and printed circuit boards. (The HDL may also be used to provide limited representations of the cables and connectors linking circuit boards together.)

In the beginning, almost all EDA tool vendors created their own HDLs to go with the tools. Some of these were analog HDLs in that they were intended to represent circuits in the analog domain, while others were focused on representing digital functionality. This text covers HDLs only in the context of designing digital ICs in the form of ASICs and FPGAs.

3.4.1. Different Levels of Abstraction

The functionality of a digital circuit can be represented at different levels of abstraction, and different HDLs support these levels of abstraction to a greater or lesser extent (Figure 3.1).

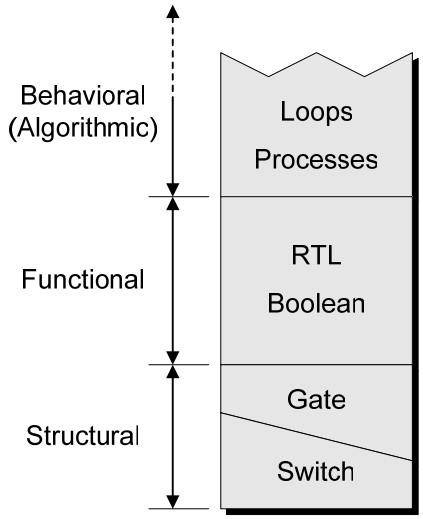


Figure 3.1. Different levels of abstraction

The lowest level of abstraction for a digital HDL would be the switch level, which refers to the ability to describe the circuit as a netlist of transistor switches. A slightly higher level of abstraction would be the gate-level, which refers to the ability to describe the circuit as a netlist of primitive logic gates and functions. Thus, the early gate-level netlist formats generated by schematic capture packages were in fact rudimentary HDLs.

Both switch-level and gate-level netlists may be classified as structural representations. It should be noted, however, that the term “structural” may also be used to refer to a hierarchical block-level netlist in which each block may have its contents specified using any of the levels of abstraction shown in Figure 3.1.

The next level of HDL sophistication is the ability to support functional representations, which covers a range of constructs. At the lower end is the capability to describe a circuit using Boolean functions. For example, after declaring a set of signals called Y, SELECT, DATA_A, and DATA_B, the functionality of a simple 2:1 multiplexer can be written using the following Boolean equation:

```
Y = (SELECT & DATA_A) | (!SELECT & DATA_B);
```

This is a generic syntax that does not favor any particular HDL and is used only for the purposes of this example. Here, the "&" character represents a logical AND, the "|" character represents an OR, and the "!" character represents a NOT.

The functional level of abstraction also encompasses register transfer level (RTL) representations. The term RTL generally refers to a design formed from a collection of registers linked by combinational logic. These registers are often controlled by a common clock signal. With two signals called CLOCK and CONTROL, along with a set of registers called REGA, REGB, REGC, and REGD, an RTL-type statement might be written similar to the following:

```
when CLOCK rises
  if CONTROL == "1" then REGA = REGB & REGC;
    else REGA = REGB | REGD;
  end if;
end when;
```

In this case, symbols like *when*, *rises*, *if*, *then*, *else*, and the like are keywords whose semantics are defined by the owners of the HDL. Once again, this is a generic syntax that does not favor any particular HDL and is used only for the purposes of this example.

The highest level of abstraction supported by traditional HDLs is known as behavioral, which refers to the ability to describe the behavior of a circuit using abstract constructs like loops and processes. This also encompasses using algorithmic elements like adders and multipliers in equations; for example:

```
Y = (DATA_A + DATA_B) * DATA_C;
```

There is also a system level of abstraction (not shown in Figure 3.1) that features constructs intended for system-level design applications.

Many of the early digital HDLs supported only structural representations in the form of switch or gate-level netlists. Others such as ABEL, CUPL, and PALASM were used to capture the required functionality for programmable logic devices. These languages

supported different levels of functional abstraction, such as Boolean functions, text-based truth tables, and text-based finite state machine (FSM) descriptions.

The next generation of HDLs, which were predominantly targeted toward logic simulation, supported more sophisticated levels of abstraction such as RTL and some behavioral constructs. It was these HDLs that formed the core of the first true HDL-based design flows.

3.4.2. An early HDL-based ASIC flow

The key feature of HDL-based ASIC design flows is their use of logic synthesis technology, which began to appear on the market around the mid-1980s. These tools could accept an RTL representation of a design along with a set of timing constraints. In this case, the timing constraints were presented in a side file containing statements along the lines of "the maximum delay from input X to output Y should be no greater than N nanoseconds".

The logic synthesis application automatically converted the RTL representation into a mixture of registers and Boolean equations, performed a variety of minimizations and optimizations (including optimizing for area and timing), and then generated a gate-level netlist that would (or at least, should) meet the original timing constraints (Figure 3.2).

There were a number of advantages to this new type of flow. First of all, the productivity of the design engineers rose dramatically because it was much easier to specify, understand, discuss, and debug the required functionality of the design at the RTL level of abstraction as opposed to working with heaps of gate-level schematics. Also, logic simulators could run designs described in RTL much more quickly than their gate-level counterparts.

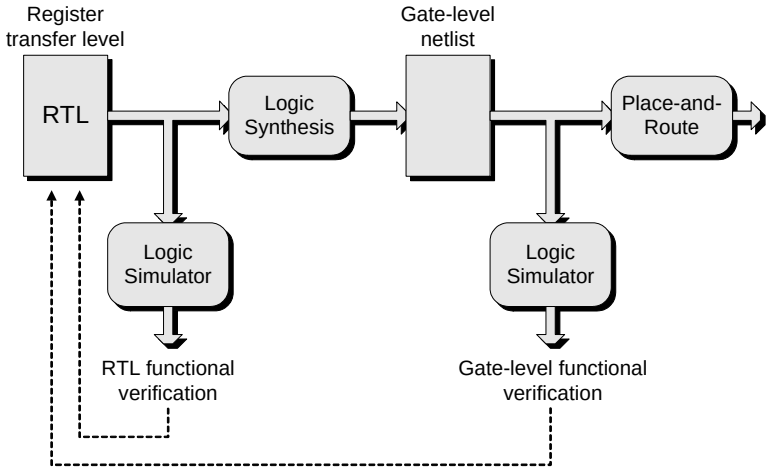


Figure 3.2. Simple HDL-based ASIC design flow

One slight glitch was that logic simulators could work with designs specified at high levels of abstraction that included behavioral constructs, but early synthesis tools could only accept functional representations up to the level of RTL. Thus, design engineers were obliged to work with a synthesizable subset of their HDL of choice.

Once the synthesis tool had generated a gate-level netlist, the flow became very similar to schematic-based ASIC flows. The gate-level netlist could be simulated to ensure its functional validity, and it could also be used to perform timing analysis based on estimated values for tracks and other circuit elements. The netlist could then be used to drive the place-and-route software, following which a more accurate timing analysis could be performed using extracted resistance and linefeed capacitance values.

3.4.3. An early HDL-based FPGA flow

It took some time for HDL-based flows to flourish within the ASIC community. Meanwhile, design engineers were still unfamiliar with the concept of FPGAs. Thus, it wasn't until the very early 1990s that HDL-based flows featuring logic synthesis technology became fully available for FPGA designs (Figure 3.3).

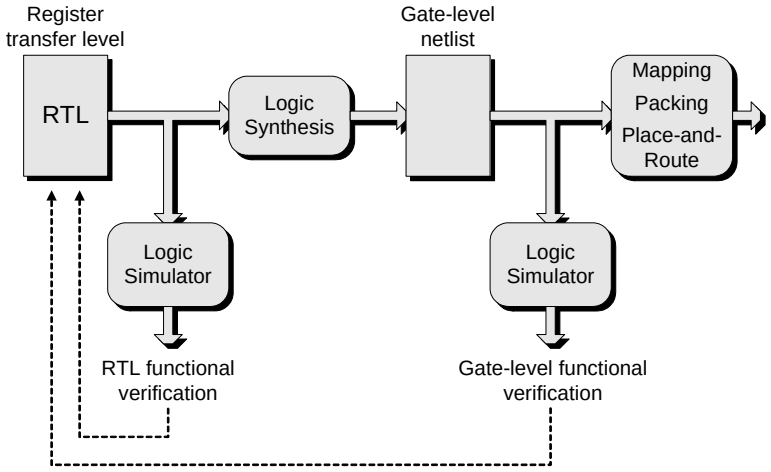


Figure 3.3. Simple HDL-based FPGA design flow

As before, once the synthesis tool had generated a gate-level netlist, it could be simulated to ensure functional validity, and it could also be used to perform timing analysis based on estimated values for tracks and other circuit elements. The netlist could then be used to drive the mapping, packing, and place-and-route software for the FPGA. Following place-and-route, a more accurate timing report could be generated using real world (physical) values.

3.4.4. Graphical Design Entry vs. HDLs

When the first HDL-based flows appeared on the scene, many assumed that graphical design entry and visualization tools, such as schematic capture systems, were going to become obsolete. Indeed, for some time, many design engineers used text editors like VI (from Visual Interface) or EMACS as their only design entry mechanism [8].

However, graphical entry techniques remain popular at a variety of levels. For example, it is extremely common to use a block-level schematic editor to capture the design as a collection of high-level blocks that are connected together. The system might then be used to automatically create a skeleton HDL framework with all of the block names and inputs and outputs declared. Alternatively, the user might create a skeleton framework in HDL, and the system might use this to create a block-level schematic automatically.

From the user's viewpoint, "pushing" down into one of these schematic blocks might automatically open an HDL editor. This could be a pure text and command based editor like VI, or it might be a more sophisticated HDL-specific editor featuring the ability to show language keywords in different colors, automatically complete statements, and so forth. Furthermore, when pushing down into a schematic block, modern design systems often give the engineer a choice between entering and viewing the contents of that block as another, lower level block-level schematic, raw HDL code, a graphical state diagram (used to represent an FSM), or a graphical flowchart. In the case of the graphical representations like state diagrams and flowcharts, these can subsequently be used to generate their RTL equivalents automatically (Figure 3.4).

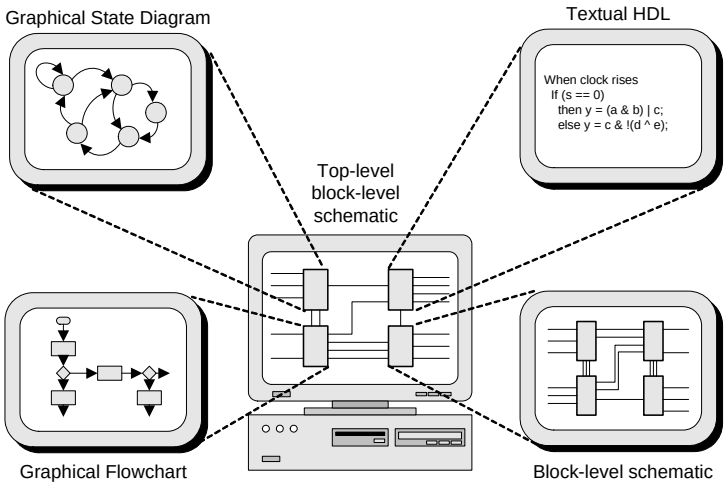


Figure 3.4. Using a mixture of different design entry methods

It is common to have a tabular file containing information relating to the device's external inputs and outputs. In this case, both the top-level block diagram and the tabular file will be directly linked to the same data and will simply provide different views of that data. Making a change in any view will update the central data and be reflected immediately in all of the other views.

3.5. Modern Hardware Description Languages

As previously noted, in the early days of digital electronics design (around the 1970s), HDL-based design tool vendors typically created their own languages to

accompany their design tools. The result was confusion. What was needed was an industry standard HDL that could be used by multiple EDA tools and vendors.

3.5.1. Verilog HDL

Sometime around the mid-1980s, Phil Moorby (one of the original members of the team that created the famous HILO logic simulator) designed a new HDL called Verilog for his company, Gateway Design Automation. In 1985 the company introduced this language to the market along with an accompanying logic simulator called Verilog-XL.

One very useful concept that accompanied Verilog and Verilog-XL was the Verilog programming language interface (PLI), which was basically an application programming interface (API). An API is a library of software functions that allow external software programs to pass data into an application and access data from that application. Thus, the Verilog PLI is an API that allows users to extend the functionality of the Verilog language and simulator.

As one simple example, assume that an engineer is designing a circuit that makes use of an existing module to perform a mathematical function such as a Fast Fourier Transform (FFT). A Verilog representation of this function might take a long time to simulate, which would be impractical if the engineer only wanted to verify the new portion of the circuit. In this case, the engineer might create a model of this function in the C programming language, which would simulate many times faster than its Verilog equivalent. This model would incorporate PLI constructs, allowing it to be linked into the simulation environment. The model could subsequently be accessed from the Verilog description of the rest of the circuit by means of a PLI call providing a bidirectional link to pass data back and forth between the main circuit (represented in Verilog) and the FFT (captured in C).

Yet one more really useful feature associated with Verilog and Verilog-XL was the ability to have timing information specified in an external text file known as a standard delay format (SDF) file. This allowed tools like post-place-and-route timing analysis

packages to generate SDF files that could be used by the simulator to provide more accurate results.

As a language, the original Verilog was reasonably strong at the structural (switch and gate) level of abstraction (especially with regard to delay modeling capability); it was very strong at the functional (Boolean equation and RTL) level of abstraction; and it supported some behavioral (algorithmic) constructs (Figure 3.5).

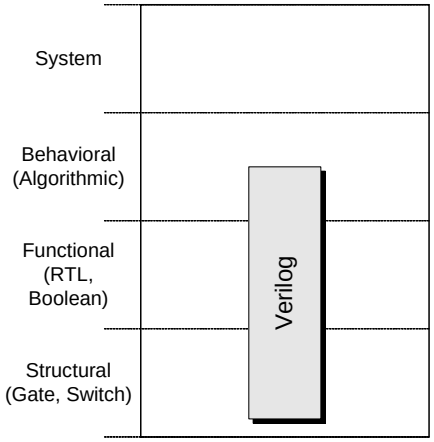


Figure 3.5. Levels of abstraction supported by Verilog

In 1989, Gateway Design Automation, along with Verilog (the HDL) and Verilog-XL (the simulator), were acquired by Cadence Design Systems. The most likely scenario at that time was for Verilog to remain as just another proprietary HDL. However, with a move that took the industry by surprise, Cadence put the Verilog HDL, Verilog PLI, and Verilog SDF specifications into the public domain in 1990.

This was a very brave move because it meant that anybody could develop a Verilog simulator, thereby becoming a potential competitor to Cadence. The reason for Cadence's decision was that the VHDL language (introduced later in this section) was starting to gain a significant user base. The benefit of placing Verilog in the public domain was that a wide variety of companies developing HDL-based tools, such as logic synthesis applications, now felt comfortable using Verilog as their language of choice.

Having a single design representation that could be used by simulation, synthesis, and other tools made engineers' job a lot easier. However, Verilog was originally conceived with simulation in mind, and applications like synthesis were something of an afterthought. This means that when creating a Verilog representation to be used for both simulation and synthesis, one is restricted to using a synthesizable subset of the language, which can be loosely defined as whatever collection of language constructs that the particular logic synthesis package understands and supports.

Verilog quickly became very popular. The problem was that different companies started to extend the language in different directions. In order to restrict this, a nonprofit body called Open Verilog International (OVI) was established in 1991. With representatives from all of the major EDA vendors of the time, OVI's mandate was to manage and standardize Verilog HDL and the Verilog PLI. The popularity of Verilog continued to rise exponentially, with the result that OVI eventually asked the IEEE to form a working committee to establish Verilog as an IEEE standard. Known as IEEE 1364, this committee was formed in 1993.

May 1995 saw the first official IEEE Verilog release, which is formally known as IEEE 1364-1995, and whose unofficial designation has come to be Verilog 95 [12]. Minor modifications were made to this standard in 2001; hence, it is often referred to as the Verilog 2001 (or Verilog 2K1) release [13]. Recently, in April 2006, Verilog has been updated once more, under the standard designation IEEE 1364-2005 [14].

3.5.2. VHDL and VITAL

In 1980, the U.S. Department of Defense (DoD) launched the very high speed integrated circuit (VHSIC) program, whose primary objective was to advance the state of the art in digital IC technology.

This program sought to address, among other things, the fact that it was difficult to reproduce ICs (and circuit boards) over the long life cycles of military equipment because the function of the parts wasn't documented in a rigorous fashion. Furthermore, different

components forming a system were often designed and verified using diverse and incompatible simulation languages and design tools.

In order to address these issues, a project to develop a new hardware description language called VHSIC HDL (or VHDL for short) was launched in 1981. One unique feature of this process was that industry was involved from a very early stage. In 1983, a team comprising Intermetrics, IBM, and Texas Instruments was awarded a contract to develop VHDL, the first official release of which occurred in 1985.

In order to encourage acceptance by the industry, the DoD subsequently donated all rights to the VHDL language definition to the IEEE in 1986. After making some modifications to address a few known problems, VHDL was released as official standard IEEE 1076 in 1987, which was also the first IEEE standard for EDA.

As a language, VHDL is very strong at the functional (Boolean equation and RTL) and behavioral (algorithmic) levels of abstraction, and it also supports some system-level design constructs. However, VHDL is a little weak when it comes to the structural (switch and gate) level of abstraction, especially with regard to its delay modeling capability (Figure 3.6).

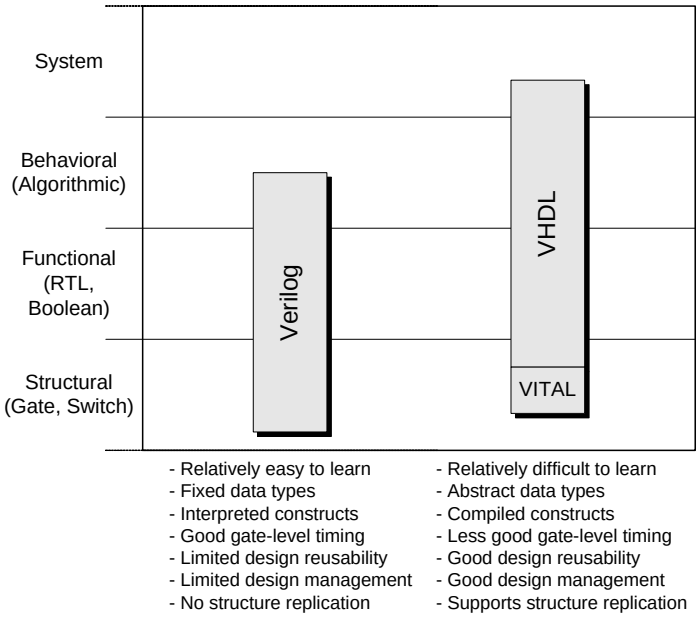


Figure 3.6. Comparison of abstraction levels (Verilog vs. VHDL)

It quickly became apparent that VHDL had insufficient timing accuracy to be used as a sign-off simulator. For this reason, the VHDL Initiative Toward ASIC Libraries (VITAL) was launched at the Design Automation Conference (DAC) in 1992. VITAL was an effort to enhance VHDL's abilities for modeling timing in ASIC and FPGA design environments. The end result encompassed both a library of ASIC/FPGA primitive functions and an associated method for back-annotating delay information into these library models, where this delay mechanism was based on the same underlying tabular format used by Verilog.

The language was further extended in a 1993 release and again in 1999 and 2002 with several child standards (1164, 1076.2, 1076.3) introduced to extend the functionality of the language. In June 2006, VHDL Technical Committee of Accellera (delegated by IEEE to work on next update of the standard) approved so called Draft 3.0 of the new VHDL specification. 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 into the main 1076 standard, extended set of operators, and more flexible syntax for 'case' and 'generate' statements. These changes should improve quality of synthesizable VHDL code, make testbenches more flexible, and allow wider use of VHDL for system-level descriptions. This new VHDL specification is targeted to become the IEEE Standard 1076-2006 [15].

3.5.3. UDL/I

As previously noted, Verilog was originally designed with simulation in mind. Similarly, VHDL was created as a design documentation and specification language that took simulation into account. As a result one can use both of these languages to describe constructs that can be simulated, but not synthesized.

In order to address these problems, the Japan Electronic Industry Development Association (JEIDA) introduced its own HDL, the Unified Design Language for Integrated Circuits (UDL/I) in 1990.

The key advantage of UDL/I was that it was designed from the ground up with both simulation and synthesis in mind. The UDL/I environment includes a simulator and a synthesis tool and is available for free (including the source code). However, by the time UDL/I arrived on the scene, Verilog and VHDL already held the high ground, and UDL/I never really managed to attract much interest outside of Japan.

3.5.4. SystemC

SystemC can be used to describe designs at the RTL level of abstraction. These descriptions can subsequently be simulated 5 to 10 times faster than their Verilog or VHDL counterparts, and synthesis tools are available to convert the SystemC RTL into gate-level netlists. In reality, SystemC is more applicable to a system-level versus an RTL design environment.

One big argument for SystemC is that it provides a more natural environment for hardware/software codesign and coverification. One big argument against it is that the majority of design engineers are very familiar with Verilog or VHDL, but they are not familiar with the object-orientated aspects of SystemC. Another consideration is that the majority of today's synthesis offerings represent hundreds of engineer years of development in translating Verilog or VHDL into gate-level netlists. By comparison, there are far fewer SystemC-based synthesis tools, and those that are available tend to be somewhat less sophisticated than their more traditional counterparts.

3.5.5. Superlog and SystemVerilog

Formed in 1997, Co-Design Automation was the developer of the Superlog language. Superlog was an amazing tool that combined the simplicity of Verilog with the power of the C programming language. It also included things like temporal logic, sophisticated design verification capabilities, a dynamic API, and the concept of assertions that are key to the formal verification strategy known as model checking (VHDL already had a simple assert construct, but the original Verilog had no such feature).

The two main problems with Superlog were (a) it was essentially another proprietary language, and (b) it was so much more sophisticated than Verilog 95 (and later Verilog 2001) that getting other EDA vendors to enhance their tools to support it would have been a major feat.

Meanwhile, as everyone in the electronics industry was wondering what the future held, the OVI group linked up with their equivalent VHDL organization called VHDL International to form a new body called Accellera. The mission of this new organization was to focus on developing new standards and formats, and to foster the adoption of new methodologies based on these standards and formats.

In the summer of 2002, Accellera released the specification for a hybrid language called SystemVerilog 3.0. The great advantage to this language was that it was an incremental enhancement to the existing Verilog, rather than the big leap represented by a full-up Superlog implementation. Actually, SystemVerilog 3.0 featured many of Superlog's language constructs donated by Co-Design. It included things like the assertion and extended synthesis capabilities that everyone wanted and, being an Accellera standard, it was well placed to quickly gain widespread adoption.

Co-Design was acquired by Synopsys in the fall of 2002. Synopsys maintained the policy of donating language constructs from Superlog to SystemVerilog, but Superlog is no longer seen as an independent language anymore. After some time, all of the mainstream EDA vendors officially endorsed SystemVerilog and augmented their tools to accept various subsets of the language, depending on their particular application areas and requirements. SystemVerilog 3.1 was released in the summer of 2003, followed by a 3.1a release (to add a few enhancements and fix some problems) around the beginning of 2004. Accellera then donated their SystemVerilog 3.1a copyright to the IEEE, and SystemVerilog has recently been established as the IEEE Standard 1800-2005 [14].

3.5.6. Mixed-Language Designs

At the beginning, it was fairly common for an entire design to be captured using a single HDL (Verilog or VHDL). As designs increased in size and complexity, however, it

became more common for different portions of the design to be created by different teams. These teams could be based in different companies or even reside in different countries, and it was not uncommon for the different groups to be using different design languages.

Another consideration was the increasing use of legacy design blocks or third-party intellectual property (IP), where the latter refers to a design team purchasing a predefined function from an external supplier.

The early 1990s saw a period known as the HDL Wars, in which the supporters of one language (Verilog or VHDL) predicted that the other language would soon become obsolete, but the years passed and both languages more or less retained their user base. The end result was that EDA vendors began to support mixed-language design environments featuring logic simulators, logic synthesis applications, and other tools that could work with designs composed from a mixture of Verilog and VHDL blocks or modules (for reference, a comparison of VHDL, Verilog, and SystemVerilog can be found in [16])

3.6. Top-Down Design Methodology

In recent years, designers have increasingly adopted top-down design methodologies even though it takes them away from logic and transistor level design to abstract programming. The introduction of industry standard hardware description languages and commercially available synthesis tools have helped establish this revolutionary design methodology. Some of the advantages of a top-down design methodology are [17]:

- increased productivity yields and shorter development cycles with more product features and reduced time to market,
- reduced Non-Recurring Engineering (NRE) costs,
- design reuse,
- increased flexibility for design changes,
- faster exploration of alternative architectures,
- faster exploration of alternative technology libraries,

- the ability to use of synthesis to rapidly sweep the design space of area and timing, and to automatically generate testable circuits,

- better and easier design auditing and verification.

In an ideal world, a true top-down system level design methodology would mean describing a complete system at an abstract level using a hardware description language and the use of automated tools, for example, design partitioners and synthesizers. This would drive the abstract level description to implementation on printed circuit boards which contain standard ICs, ASICs, FPGAs, PLDs, and full-custom ICs. This ideal has not yet been fulfilled; however, EDA tools are constantly being improved towards this vision. This means designers must constantly take on new roles and learn new skills. More time is now spent designing HDL models, considering different architectures and considering system testability and debug issues. Practically no time is spent designing at the gate level.

Technological advancements over the last years has increased the complexity of standard ICs and ASICs and resulted in the concept of “system on a chip (SOC)”. A top-down design methodology is the only practical option to design such chips.

Any ASIC or FPGA design in a hardware development project is usually on the critical path of the development schedule. Although, in rare cases for reasons of cost, a schematic entry tool may still be a viable design method for small devices such as PLDs, a top-down design approach using a hardware description language, is by far the best design philosophy to adopt, provided the budget is available for simulation and synthesis tools.

4. THE VERILOG HARDWARE DESCRIPTION LANGUAGE

In accordance with the commonly used, hierarchical, top-down design methodology, a circuit description in Verilog is specified by defining modules that correspond to the various functional blocks at different levels within the circuit.

4.1. The Verilog module

A module is the basic building block in Verilog. Circuits are described using one or more modules. Modules can represent:

- a physical block, such as an IC or ASIC cell,
- a logical block, such as the ALU portion of a processor design,
- a Device Under Test (DUT),
- a testbench,
- or a complete system.

Every module description starts with the keyword ***module***, followed by the name of the module (MUX, DFF, ALU, etc.), and a list of the module's ports. The description ends with the keyword ***endmodule***.

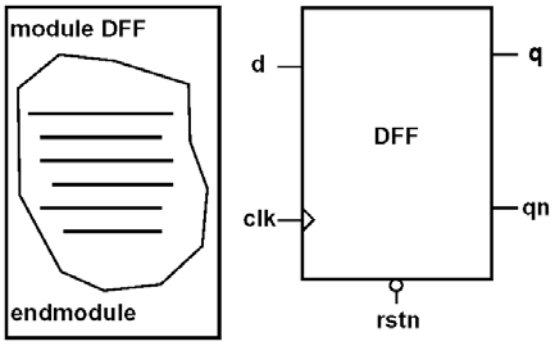


Figure 4.1. Representation of a D-type flip-flop as a module

The module description for the D-type flip-flop in Figure 4.1 is given below:

```
module DFF (q, qn, d, clk, rstn);
  output q, qn;
  input d, clk, rstn;

    D-type flip-flop definition

endmodule
```

4.2. Module Ports

Modules communicate with their environment through ports. Module ports represent device pins and board connectors. All ports are listed in parentheses after the module name. Then, within the module definition, each port is declared to be of type **input**, **output**, or **inout** (bidirectional).

Inputs and outputs having multiple-bit widths are called vectors:

```
input [0:5] x;
output [2:0] y;
```

The examples above declare an input vector **`x`** with six bits (0 to 5) and an output vector **`y`** with three bits (2 to 0). First number listed is the most significant bit of the vector. The individual bits can be accessed by specifying them within square brackets, like **`x[3]`** or **`y[1]`**.

4.3. Module Instances

Each Verilog module defines a new scope, i.e. a new level of hierarchy within the circuit description. A module may include any number of other modules and connect them together to form more complex circuits. For example, a multiplier circuit that includes a number of shift registers and adders can be modeled in Verilog by first writing the descriptions for a shift register and an adder as individual modules, and then calling these modules as many times as required from within the multiplier description. Each call to a module creates an instance of that module in the circuit.

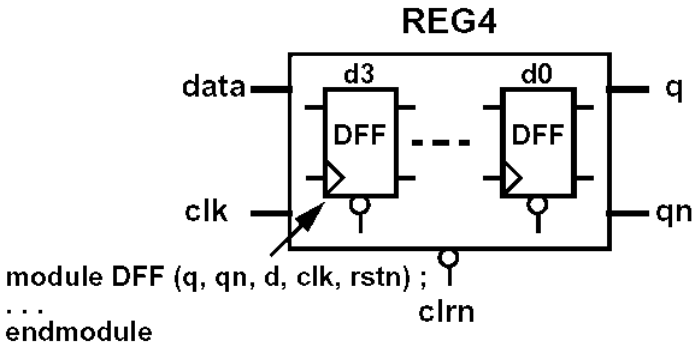


Figure 4.2. Modeling a 4-bit register comprising four D-type flip-flops

```
module REG4 (q, qn, data, clk, clrn);
  output [3:0] q, qn;
  input [3:0] data;
  input clk, clrn;

  DFF d0 (q[0], qn[0], data[0], clk, clrn);
  DFF d1 (q[1], qn[1], data[1], clk, clrn);
  DFF d2 (q[2], qn[2], data[2], clk, clrn);
  DFF d3 (q[3], qn[3], data[3], clk, clrn);
endmodule
```

The Verilog description of the REG4 module in Figure 4.2 instantiates the DFF module four times. Each DFF module has a unique instance name (d0, d1, d2, and d3). The instance name uniquely identifies the instance. Each instance is a complete, independent, and concurrently active copy of the module.

4.4. Gate Level Modeling of Circuits

Gate level modeling represents a textual description of a schematic diagram. Verilog recognizes twelve basic gates as predefined primitives. They are declared with the keywords **buf**, **not**, **and**, **nand**, **or**, **nor**, **xor**, **xnor**, **bufif1**, **bufif0**, **notif1**, **notif0**. Some of these are two-state gates and some of them are tri-state gates.

There are eight two-state gates predefined in Verilog. These are **buf**, **not**, **and**, **nand**, **or**, **nor**, **xor**, **xnor**. The declaration for a two-state gate consists of the gate type keyword,

followed by a name for the gate, and then a comma separated list of ports in parenthesis specifying in order the output and the data inputs of the gate:

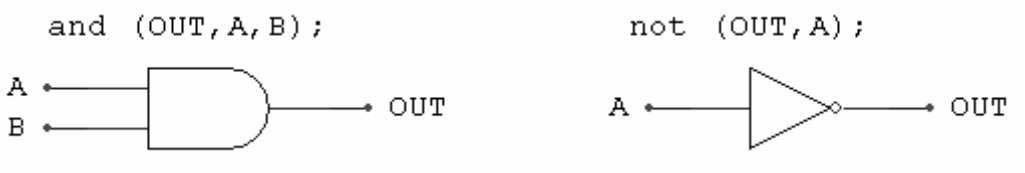


Figure 4.3. Examples of Verilog primitives

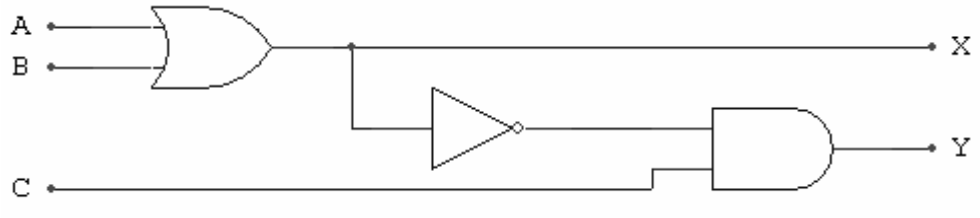


Figure 4.4. A simple combinational circuit

```
// Gate level description of the circuit in Figure 4.4
module circuit( X, Y, A, B, C );
  input A, B, C;
  output X, Y;
  wire D;

  or g1( X, A, B );
  not g2( D, X );
  and g3( Y, C, D );
endmodule
```

The module is declared as **module circuit (X, Y, A, B, C)**. Here, **(X, Y, A, B, C)** are the ports of the circuit. **A**, **B**, **C** are declared as **input**, and **X**, **Y** are declared as **output**. The internal connection between the output of the inverter and the input of the AND gate is declared as a **wire** with name D. Each gate is given a unique name (such as g1, g2, etc…). For example, the **or** gate has gate name **g1**, its output is **X**, and its inputs are **A** and **B**.

The four tri-state gates predefined in Verilog are **bufif1**, **bufif0**, **notif1**, **notif0**. The declaration for a tri-state gate consists of the gate type keyword, followed by a name for

the gate, and then a comma separated list of ports in parenthesis, specifying the output, the data input and the control input of the gate. Here are two examples of tri-state gates:

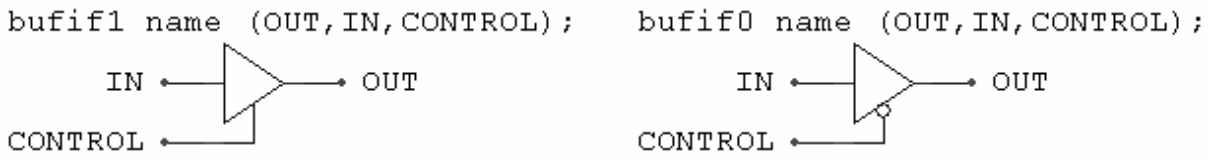


Figure 4.5. Tri-state gates in Verilog

For the first example (bufif1), if CONTROL is 1, IN is transferred to OUT; otherwise, OUT behaves as an open circuit (high impedance, denoted by Z). For the second example (bufif0), if CONTROL is 0, IN is transferred to OUT; otherwise, OUT becomes Z.

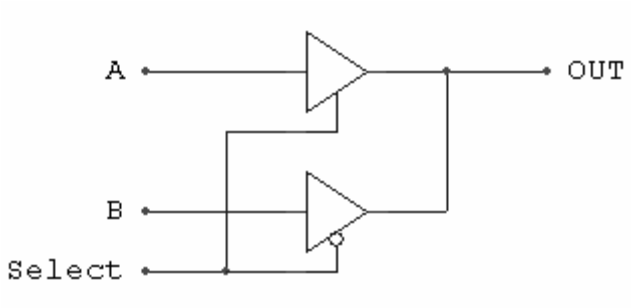


Figure 4.6. Implementing a 2-to-1 multiplexer with tri-state gates

```
// Gate level description of the
// 2-to-1 multiplexer in Figure 4.6
module mux_2to1( A, B, Select, OUT );
  input A, B, Select;
  output OUT;
  tri OUT;

  bufif1 g1(OUT, A, Select);
  bufif0 g2(OUT, B, Select);
endmodule
```

As seen in Figure 4.6, the outputs of the buffers are connected to a common node to form a single output. For such connections, the **tri** keyword must be used to indicate that the output has multiple tri-stated drivers.

4.5. Dataflow Modeling

Dataflow modeling uses some operators that act on operands to produce the desired results. Data flow modeling is used for describing the combinational circuits by their function rather than by their gate structure. It is not used for describing sequential circuits. Some of the operators used for dataflow modeling are given in Table 4.1.

Table 4.1 Logical and arithmetic operators in Verilog

&	bit-wise AND	+	addition
|	bit-wise OR	-	subtraction
~	bit-wise NOT	{}	concatenation
^	bit-wise XOR	?:	conditional assignment

It is important to distinguish between arithmetic and logic operations. For example, "+" represents arithmetic addition; for logical OR operation, the symbol "|" is used.

Dataflow modeling uses continuous assignment, declared using the keyword **assign**. A continuous assignment is a statement that assigns a value to a net which defines a gate output declared by an **output** or a **wire** statement. The keyword **assign** is followed by the target output name, an equal sign (=), and then an expression comprised of operands and operators, the result of which will be assigned to the target output.

Here is the dataflow description of a 2-to-1 line multiplexer:

```
// Dataflow description of the 2-to-1 line multiplexer
module mux_2to1( A, B, Select, OUT );
  input A, B, Select;
  output OUT;
  assign OUT = ( A & Select ) | ( B & ~Select );
endmodule
```

Next is the dataflow description of a 4-bit full adder:

```
// Dataflow description of a 4-bit full adder
module fadder4bit ( A, B, Cin, SUM, Cout );
   input [3:0] A, B;
   input Cin;
   output [3:0] SUM;
   output Cout;

   assign {Cout , SUM} = A + B + Cin;
endmodule
```

The inputs A and B are declared as 4 bit vectors. Binary addition of A, B and the carry input Cin is specified by the plus symbol (+). The 5 bit result of the addition operation is assigned to {Cout, SUM}, the concatenation of Cout and SUM. Cout receives the most significant bit of the result, and SUM receives the remaining 4 bits (the least significant bit of SUM, SUM[0], is assigned the least significant bit of the result).

4.6. Behavioral Modeling

Behavioral modeling represents digital circuits at the functional and algorithmic level. Behavioral modeling can be used for describing both sequential and combinational circuits.

Initial conditions for a circuit (or a simulation) are declared with the keyword initial. initial blocks execute only once at the beginning of a simulation, and end after all the statements have completed. The initial keyword is followed by either a single statement, or a block of statements enclosed within the keywords **begin** and **end**:

```
initial
  begin
    clock = 1'b0;
    repeat(30);
      #10 clock = ~clock;
  end
```

In the example above, clock is initially assigned a value of 0 (1'b0 represents a 1 bit constant with a value of 0) and the simulation engine is set to complement the clock 30 times every 10 time units (#10 specifies a delay of 10 time units between each repetition).

The behavior of the circuit itself is described using **always** blocks in the form

```
always @(event control expression);
  Procedural description of circuit
```

Here the procedural description can be either a single statement, or a block of multiple statements enclosed within **begin** and **end** keywords. Procedural statements define flow control and value assignment to variables within the module. The left hand side variable in an assignment statement must be a register, declared as type **reg**. Registers declared with the **reg** keyword retain their values until a new value is assigned, and thus are used for modeling flip-flops and memory devices.

The event control expression specifies the condition (or conditions) that will trigger the execution of the procedural assignment statements in the **always** block. These statements are executed on every occurrence of the condition(s) given in the event control expression. Multiple trigger conditions in an expression must be separated by the **or** keyword.

The event control expression can contain both level sensitive and edge sensitive trigger conditions. Level sensitive event controls cause execution of the **always** block each time the value of a variable in the expression changes (Figure 4.7), whereas edge sensitive event controls only respond to rising edge or falling edge transitions of the variables (Figure 4.8), specified by the keywords **posedge** or **negedge**, respectively.

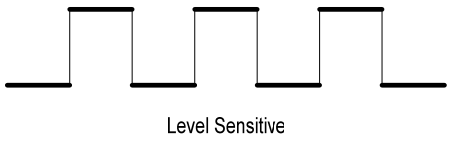


Figure 4.7. Signal levels are monitored for a level sensitive event

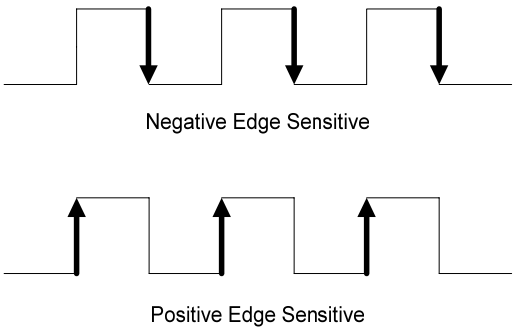


Figure 4.8. Rising and falling transitions of a signal trigger edge sensitive events

Level sensitive events are included in always blocks with only the signal names in the sensitivity list:

```
always @(A)
always @(B or C)
```

Edge sensitive events require the specific signal transition to be declared within the sensitivity list:

```
always @(negedge D)
always @(posedge E or negedge F)
```

The behavioral description of a 2:1 multiplexer would be as follows:

```
// Behavioral description of a 2-to-1 line multiplexer
module mux_2to1( A, B, Select, OUT );
  input A, B, Select;
  output OUT;
  reg OUT;

  always @ ( Select or A or B )
    if ( Select == 1 ) OUT = A;
    else OUT = B;
endmodule
```

If the event control expression involves a vector, the **case** statement can be used to decide what action to take. The definition of a **case** statement begins with the keyword

case and a control expression in parentheses, followed by a list of conditional branches, and terminated with the keyword **endcase**. The branch whose label matches the current value of the control expression gets selected and its contents are executed.

In the example below, a 4:1 multiplexer is described using the **case** statement. A change in the values of *i0*, *i1*, *i2*, *i3* or *Select* triggers the **always** block. *Select* is evaluated and compared with the labels of the conditional branches listed in the **case** statement. Here, the branch labels are all 2-bit binary numbers representing the possible values the *Select* vector can have. The matching branch is then executed.

```
// Behavioral description of a 4-to-1 line multiplexer
module mux_4to1( i0, i1, i2, i3, Select, y );

  input i0, i1, i2, i3;
  input [1:0] Select;

  output y;

  reg y;

  always @( i0 or i1 or i2 or i3 or Select )
    case ( Select )
      2'b00: y = i0;
      2'b01: y = i1;
      2'b10: y = i2;
      2'b11: y = i3;
    endcase

endmodule
```

4.6.1. Behavioral Description of Flip-Flops

Behavioral descriptions of various types of flip-flops are given below. The operation of the flip-flops are defined in edge sensitive **always** blocks by writing their state equations as Boolean functions assigned to a register variable declared as type **reg**.

D-type flip-flop:

```
module D_FF( D, Q, CLK, RST );
  input D, CLK, RST;
  output Q;
  reg Q;

  always @( negedge RST or posedge CLK )
    if (RST == 0) Q = 1'b0;
    else Q = D;
endmodule
```

T-type flip-flop:

```
module T_FF( T, Q, CLK, RST );
  input T, CLK, RST;
  output Q;
  reg Q;

  always @( negedge RST or posedge CLK )
    if (RST == 0) Q = 1'b0;
    else Q = Q ^ T;
endmodule
```

JK-type flip-flop:

```
module JK_FF( J, K, Q, CLK, RST );
  input J, K, CLK, RST;
  output Q;
  reg Q;

  always @( negedge RST or posedge CLK )
    if (RST == 0) Q = 1'b0;
    else Q = (J & ~Q) | (~K & Q);
endmodule
```

4.6.2. Modeling of Finite State Machines

Two or three **always** blocks are usually used for modeling state diagrams in Verilog, one for describing the sequential behavior of the circuit, one for describing the combinational next state logic, and one, if necessary, for describing the output logic. This approach presents a standard way of modeling any state diagram in Verilog and it is readily recognized by synthesis tools, enabling further optimizations to be performed on the circuit such as the reduction and re-encoding of states to obtain a smaller or faster circuit.

The behavioral description of the state diagram in Figure 4.9 is given below:

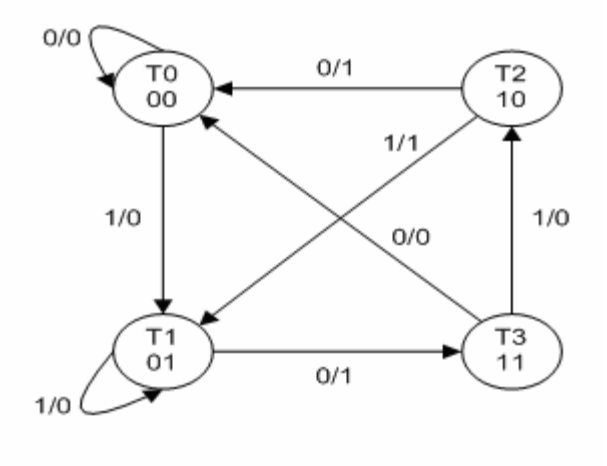


Figure 4.9. A sample state diagram

```
// Behavioral description for the state diagram in Figure 4.9

module StateMachine( X, CLK, RST, Z );

  input X, CLK, RST;

  output Z;

  reg Z;
  reg [1:0] PState, NState;

  parameter T0 = 2'b00, T1 = 2'b01, T2 = 2'b10, T3 = 2'b11;
```

```
  always @(posedge CLK or negedge RST)
    if (RST == 0) PState = T0;
    else PState = NState;

  always @(X or PState)
    case(PState)
      T0: if (X == 0) NState = T0;
          else NState = T1;
      T1: if (X == 0) NState = T3;
          else NState = T1;
      T2: if (X == 0) NState = T0;
          else NState = T1;
      T3: if (X == 0) NState = T0;
          else NState = T2;
    endcase

  always @(X or PState)
    case(PState)
      T0: Z = 1'b0;
      T1: if(X == 0) Z = 1'b1;
          else Z = 1'b0;
      T2: Z = 1'b1;
      T3: Z = 1'b0;
    endcase

endmodule
```

Here, the previous and next states of the circuit are assigned to register variables with identifiers *NState* and *PState*. The **parameter** keyword is used to assign a label to each state code. There are three **always** blocks. The first **always** block controls the synchronous state transitions in the machine and the asynchronous reset operation. The second **always** block determines the next state of the machine according to the present state and the input, and the third **always** block generates the output of the circuit.

4.7. Structural Description

In Verilog, combinational circuits are described using gate level or dataflow modeling, and sequential circuits are described using behavioral statements for the flip-flops' operation. Since all sequential circuits are made up of flip-flops and combinational logic, its structure can be described by a mix of data flow and behavioral modeling statements in separate modules, which are then combined by instantiation.

A sequential circuit is given below in Figure 4.10. This circuit will be modeled using the structural description style:

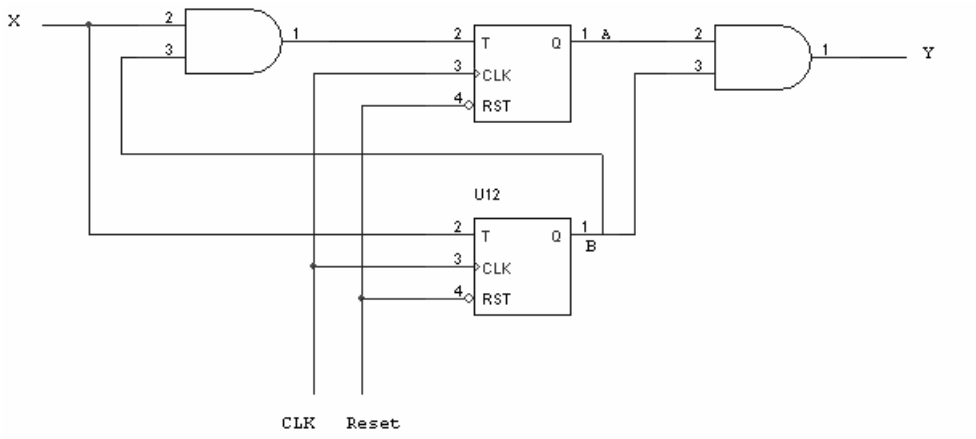


Figure 4.10. A sample sequential circuit

```
// Structural description of the
// sequential circuit in Fig. 4.10

module SeqCircuit( X, RST, CLK, Y, A, B );

  input X, RST, CLK;

  output A, B, Y;

  wire TA, TB;

  T_FF TFF_A(TA, A, CLK, RST);  // Instantiation of flip-flop A
  T_FF TFF_B(TB, B, CLK, RST);  // Instantiation of flip-flop B
```

```
  assign TA = B & X;
  assign TB = X;
  assign Y = A & B;

endmodule

// Behavioral description of T-type flip-flop

module T_FF(T, Q, CLK, RST);

  input T, Q, CLK;

  output Q;

  reg Q;

  always @(posedge CLK or negedge RST)
    if (RST == 0) Q = 1'b0;
    else Q = Q ^ T;

endmodule
```

4.8. Writing A Test Bench

A test bench is used for verifying the operation of a design by applying some stimulus and observing the response. Test benches use the **initial** and **always** statements to describe the stimulus to be applied to the circuit under test. The **initial** statement executes only once at the beginning of the simulation. The **always** statement executes repeatedly in a loop. Each operation can be delayed for a desired number of time units by using the # symbol:

```
initial
   begin
        A = 0;
        B = 0;
    #10 A = 1;
    #20 A = 0;
        B = 1;
   end
```

- At *Time = 0*, both variables A and B are set to 0
- 10 time units later at *Time = 10*, A is changed to 1
- 20 time units later at *Time = 30*, A is changed to 0 and B to 1

A test module typically has no inputs and outputs. The signals that are applied as inputs to the design module for simulation are declared in the stimulus module as local registers. The outputs of the design module that are to be displayed for testing are declared in the stimulus module as local wires.

A stimulus module has the following form:

```
module test
  - Declare local reg and wire identifiers
  - Instantiate the design module under test
  - Generate the stimulus using initial and always statements
  - Display output response
endmodule
```

Output of the simulator can be plotted as timing diagrams. It is also possible to display text messages and values of circuit variables using Verilog system tasks. System tasks are recognized by keywords that begin with the dollar sign ($):

- **$display:** Display variables or strings and move to a new line
- **$write:** Same as **$display** but does not move to a new line
- **$monitor:** Display variables whenever a specified value changes during simulation
- **$time:** Print the current simulation time
- **$finish:** Terminate the simulation

The **$display**, **$write**, and **$monitor** system task calls have the form

```
task_name (format specifier, argument list);
```

The format specifier is a string enclosed in quotes (“ ”); it contains format placeholders that start with the percent sign (%), and may also contain regular text. Each format placeholder specifies how the value of its corresponding variable in the argument list will be printed. Values can be printed in binary, octal, decimal or hexadecimal notation, identified with the placeholders %b, %o, %d, and %h respectively:

```
$monitor(“time = %d  A = %b  B = %b”, $time, A, B);
```

The Verilog description of a sample testbench is given below:

```
// A sample testbench

module test_circuit;

  reg [3:1] TA;

  wire TX, TY;

  circuit cr( TA, TX, TY );

  initial
    begin
      TA = 3'b000;
      repeat(7)
        #10 TA = TA + 1'b1;
    end

  initial
    $monitor("A = %b X = %b Y = %b Time = %0d", TA, TX, TY,
$time);

endmodule


// A simple circuit

module circuit( A, X, Y);

  input [3:1] A;
```

```
  output X, Y;

  assign X = ( A[3] & A[2] ) | A[1];
  assign Y = A[2] ^ A[1];

endmodule
```

When this testbench is run, the Verilog simulator generates the following output:

```
A=000 X=0 Y=0 Time=0
A=001 X=1 Y=1 Time=10
A=010 X=0 Y=1 Time=20
A=011 X=1 Y=0 Time=30
A=100 X=0 Y=0 Time=40
A=101 X=1 Y=1 Time=50
A=110 X=1 Y=1 Time=60
A=111 X=1 Y=0 Time=70
```

Besides textual output, nodes within the circuit can also be selected for display in a timing diagram. For the example testbench above, if signals *TA*, *TX*, *TY* are selected for graphical display, the Verilog simulator produces the following timing diagram:

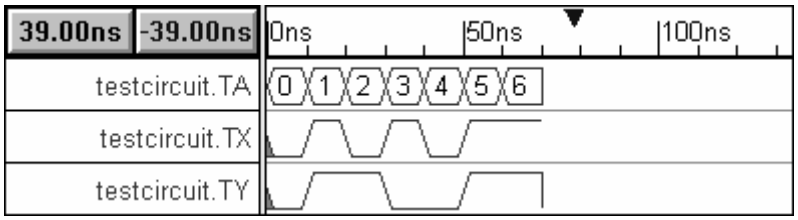


Figure 4.11. Timing diagram showing signal values in the example testbench

5. WRITING SYNTHESIZABLE VERILOG CODE

"Just as understanding Karnaugh maps is the key to manual design methods, understanding how to write synthesis-friendly HDL models is the key to automated design methods." - Michael D. Ciletti, *Advanced Digital Design with the Verilog HDL* [18]

Design entry is one of the crucial stages in the design flow, and the design engineer is responsible for coding the hardware description according to the guidelines set forth by the synthesis tool vendor in order to fully use the capabilities of the tool.

In a design, the same functionality can be expressed using different coding styles and different HDL constructs. Although any experienced engineer can easily tell that all these different pieces of code represent the same implementation, it is a challenge for the synthesis tool to recognize so many alternative coding styles and figure out the designer's intention from them. Thus, synthesis tools are programmed to work with only a limited number of commands and constructs of the HDL, which are listed together with coding guidelines in the user manuals of every synthesis tool. To take full advantage of the tools' optimization capabilities, these guidelines must be strictly followed when writing HDL code for synthesis.

5.1. Overview of The Synthesis Process

Synthesizing and optimizing a network of logic gates involves evaluating many different factors (such as simplification by exploiting don't-cares and common factors without violating fan-in and fan-out restrictions or delay constraints), and finding a balance between these to achieve the specified size or performance criteria. As designs get larger in size, the number of alternative implementations reach such enormous levels that it becomes impractical, and at some point impossible, to exhaustively search through all of them to find the optimal solution. Instead, heuristic approaches are used that yield acceptable solutions [18, 19]. To help manage this complex task, logic synthesis is divided into two phases: first, technology-independent optimizations operate on a model of the described logic that does not directly represent logic gates; then, technology-dependent optimizations improve the network's gate-level implementation [20].

Technology-independent optimizations can be grouped into three categories based on how they change the Boolean function:

- **Simplification:** Minimizing the sum-of-products expressions of the Boolean functions that make up the network by combining minterms and exploiting don't-cares.
- **Network restructuring:** Creating new nodes in the network that can be used as common factors and collapsing sections of the network into a single node in preparation for finding new common factors.
- **Delay restructuring:** Changing the factorization of a subnetwork to reduce the number of nodes through which a delay-critical signal must pass.

Technology-dependent optimizations cover the mapping, placement, and routing of the design according to the specified constraints. After the completion of the first phase, resulting Boolean functions are then mapped to the logic elements available in the target technology. These logic elements can range from simple logic gates (or LUTs in FPGAs) to pre-designed library macros for commonly used functions such as adders, multipliers, shift registers and multiplexers. Macros are either called explicitly by the designer, or inferred from the HDL code and substituted automatically by the synthesis tools. Inferred macros are listed in the synthesis reports produced by the tools, and these reports must be reviewed by the designer to ensure that macro substitutions are consistent with the design intent.

5.2. Synthesis of Combinational Logic

Synthesizable combinational logic can be described by

- a netlist of structural primitives (see Section 4.4)
- a set of continuous assignment statements (see Section 4.5)
- a level sensitive cyclic behavior (see Section 4.6)

A level-sensitive cyclic behavior will synthesize to combinational logic if it assigns a value to each output for every possible value of its inputs. This implies that the event control expression of the behavior must be sensitive to every input, and that every path of the activity flow must assign value to every output.

5.2.1. Synthesis of Priority Structures

A ***case*** statement gives higher priority to the first item that it decodes than to the last one, and an ***if*** statement gives higher priority to the first branch than to the remaining branches. A synthesis tool will determine whether the case items of a ***case*** statement are mutually exclusive. If they are mutually exclusive, the synthesis tool will treat them as though they had equal priority and will synthesize a multiplexer rather than a priority structure. Similarly, an ***if*** statement will synthesize to a multiplexer when the branching is specified by mutually exclusive conditions. When branching is not mutually exclusive, a priority structure will be created.

5.2.2. Exploiting Don't-Care Conditions

An assignment to ***x*** in a ***case*** or an ***if*** statement will be treated as a don't-care condition in synthesis. For example, a ***default*** item that assigns ***x*** values to all outputs can be included in a ***case*** statement to represent unused input combinations. The synthesized hardware will produce either a 0 or a 1, but the HDL model will produce an x in simulation. This may lead to a mismatch between simulation results and real-world operation.

5.2.3. Accidental Synthesis of Latches

Level-sensitive cyclic behaviors will synthesize to combinational logic if the description does not imply the need for storage. If storage is implied by the model, a latch will be introduced into the implementation. To avoid latches, all of the variables that are assigned value by the behavior must be assigned a value under every possible input. Failure to do so will yield a design with unwanted latches.

Verilog ***case*** and ***if*** statements that do not include all possible cases or conditions are incompletely specified and may lead to synthesis of unwanted latches. When a ***case*** or an ***if*** statement in a level-sensitive cyclic behavior does not specify an output for all of the possible inputs, the synthesis tool infers a latch, because the description implies that the output should retain its present value under the conditions that were left unspecified. Caution must be taken to ensure that ***case*** and ***if*** statements are complete.

5.2.4. Resource Sharing

When optimizing the design for size, synthesis tools must share logic resources as much as possible to minimize needless duplication of circuitry. The tool must recognize whether the physical resources required to implement complex functions can be shared. If the data flows within the function do not conflict, the resource can be shared between one or more paths. For example, the addition operators in the continuous assignment below are in mutually exclusive datapaths and can be shared in hardware.

```
assign output = select ? data_1 + data_2 : data_1 + constant;
```

This operation can be implemented by a shared adder with multiplexed input datapaths. If the synthesis tool does not automatically implement resource sharing, the Verilog description must be written so as to force sharing of the resource:

```
assign output = data_1 + (select ? data_2 : constant);
```

Failure to include the parentheses in the expression for *output* above will lead to synthesis of a circuit that uses two adders. The most efficient implementation multiplexes the datapaths and shares the adder between them, rather than multiplexing the outputs of separate adders. The important design tradeoff here is that the multiplexer will occupy significantly less area than the adder it replaces.

As a general guideline, if arithmetic functions are to be inferred from HDL operators, the operators should be grouped as much as possible within a single cyclic behavior to allow the synthesis engine to share the hardware resources that will be used to implement the function.

5.3. Synthesis of Sequential Logic with Flip-Flops

A register variable in an edge-sensitive behavior will be synthesized to a flip-flop

- if it is referenced outside the scope of the behavior,
- if it is referenced within the behavior before it is assigned value, or
- if it is assigned value in only some of the branches of the activity within the behavior.

All of these situations imply the need for memory, as the register has to retain its value under the conditions that do not assign a new value to the register. The fact that these conditions are contained in an edge-sensitive behavior dictates that the memory be realized as a flip-flop, rather than a latch.

In a level-sensitive cyclic behavior, incomplete ***case*** or ***if*** statements lead to the synthesis of latches. However, if the behavior is edge-sensitive, these types of statements will not create latches, but will synthesize logic that implements a "clock enable" mechanism, because the incomplete statements imply that the contents of the affected registers should not change under the conditions that were left unspecified, even if the clock signal makes a transition.

5.4. Synthesis of Finite State Machines

There are many ways to describe FSMs. A traditional FSM representation incorporates two or three **always** blocks, one for describing the sequential behavior of the circuit, one for describing the combinational next state logic, and one, if necessary, for describing the output logic. Sample code for a state machine was given in Section 4.6.2

5.4.1. Finite State Machine Extraction in Xilinx XST Synthesis Software

Xilinx's synthesis software XST tries to distinguish finite state machines from VHDL/Verilog code (called FSM extraction), and apply several state encoding techniques,

such as re-encoding the original state codes given in the HDL file, to get better performance or less area. FSM extraction can be enabled or disabled by the user. Note that XST can handle only synchronous state machines [21].

XST can also detect unreachable states in an FSM and lists them in the HDL synthesis report. If Safe Implementation mode is enabled, XST automatically adds logic to the FSM implementation that will let the state machine recover from an invalid state. If during its execution, a state machine gets into an invalid state, the logic added by XST will bring it back to a known state, called a recovery state. By default, XST automatically selects a reset state as the recovery state. If the FSM does not have an initialization signal, XST selects the power-up state as the recovery state. The recovery state can also be manually defined by the user.

5.4.2.Optimization of State Encodings

XST can re-encode the original state codes given in the HDL file to get better performance or less area. XST supports the following state encoding techniques:

- **Auto**

  In this mode, XST tries to select the best encoding algorithm suitable for each FSM.

- **One-Hot Encoding**

  One-hot encoding is the default encoding scheme. Its principle is to associate one code bit and also one flip-flop to each state. At a given clock cycle during operation, one and only one bit of the state variable is asserted. Only two bits toggle during a transition between two states.

  Although this approach uses a greater number of flip-flops than other forms of encoding, the decoding logic in a one-hot machine uses fewer gates because the machine has to decode only a single bit of a register rather than a vector pattern. Thus, the silicon area required by the extra flip-flops can be compensated by the area saved using simplified decoding logic. One-hot encoding is very appropriate with

most FPGA targets where a large number of flip-flops are available, and saving them does not necessarily provide a benefit. It is also a good alternative when trying to optimize speed or to reduce power dissipation.

It is quite easy to modify a one-hot design, because adding or removing a state does not affect the encoding of the other states. The design effort is reduced too, since there is no need to encode a state transition table. The state transition graph is sufficient.

- **Gray Encoding**

  Gray encoding guarantees that only one bit switches between two consecutive states. It is appropriate for controllers exhibiting long paths without branching. In addition, this coding technique minimizes hazards and glitches. The one-bit change between consecutive states will also reduce the simultaneous switching of adjacent physical signal lines in a circuit, thereby minimizing the possibility of electrical crosstalk [18]. Very good results can be obtained when implementing the state register with T-type flip-flops.

- **Compact Encoding**

  Compact encoding consists of minimizing the number of bits in the state variables and flip-flops. Compact encoding is appropriate when trying to optimize area.

- **Johnson Encoding**

  Similar to Gray, Johnson encoding shows benefits with state machines containing long paths with no branching. It uses more bits than Gray encoding.

- **Sequential Encoding**

  Sequential encoding consists of identifying long paths and applying successive radix two codes to the states on these paths. Next state equations are minimized.

- **Speed1 Encoding**

  Speed1 encoding is oriented for speed optimization. The number of bits for a state register depends on the particular FSM, but generally it is greater than the number of FSM states.

- **User**

  In this mode, XST uses original encoding, specified in the HDL file

XST log file reports the full information about the recognized FSMs during the synthesis process. If the encoding option was set to Auto for XST to determine the best encoding algorithm for FSMs, it reports the encoding it chose for each FSM.

5.5. Synthesis of Other Common Logic Elements

Depending on their level of sophistication, synthesis tools may also support recognition of other common logic elements such as registers, counters, shift registers, address decoders, adders, multipliers, etc., and even automatically replace them with pre-designed, pre-optimized, efficient macros from their device library.

5.6. Resets and Clock Enables

Synthesis tools can recognize reset and clock enable signals in a design, and utilize appropriate routing resources to distribute these signals throughout the hardware implementation. FPGAs might have dedicated clock and reset signal trees that span the whole chip, and an FPGA synthesis tool will map such control signals to these dedicated routing channels.

A global reset signal is required for ASICs to initialize the system state to a known value at power-up. However, an FPGA, during its configuration, programs all of its flip-flops individually to a fixed logic value, which can be determined within the HDL code, if desired. Thus, a design implemented in an FPGA does not require a global reset network. For the vast majority of any design, the initialization state of all flip-flops and

RAM following configuration is more comprehensive than any logical reset will ever be. There is no requirement to insert a reset for simulation because nothing will be undefined. Since FPGAs are already fully tested silicon, there is no need for inserting scan logic in a design and running test vectors, so global reset is not required as part of this process either.

Inserting a global reset will impact development time and final product costs even if these factors can not be easily quantified. With the trend towards higher speed clocks and complete systems on a chip, the reliability issues must be taken seriously. The critical parts of a system that must truly be reset should be identified and the release of those resets on start up, or during operation, must be controlled as carefully as any other signal within a synchronous circuit [22].

In an FPGA architecture, each flip-flop may have a set of dedicated control inputs to support “set”, “reset”, and “clock enable”. Set and reset inputs may work synchronously or asynchronously, and the synthesis tool uses these inputs when it can; reserving the LUT for implementing other functions. The type of dedicated set and reset control inputs (synchronous or asynchronous) is an important factor in coding a design, because using a behavior that utilizes the available type of control input is readily implemented, yielding a compact design, whereas using the opposite type requires extra LUT resources to convert the control signals to the desired type [23].

5.7. Anticipating The Results of Synthesis

The synthesis tool should not be trusted blindly. It is advisable to anticipate what the synthesis tool will produce and then examine the results against those expectations. The designer should be familiar with the synthesis tool and write Verilog descriptions that will infer the desired result. Each vendor’s software operates differently, so it is good practice to experiment with a synthesis tool to learn how it handles particular styles of coding.

6. AN FPGA IMPLEMENTATION EXAMPLE

In order to investigate the effects of optimization settings on synthesis, a sample circuit is implemented using Xilinx's Spartan-3E FPGA Starter Kit [24]. Design entry, simulation, and synthesis steps are all performed from within Xilinx's freely available ISE WebPACK 8.2i design software, which also includes the XST synthesis tool [25].

6.1. Xilinx Spartan-3E FPGA Family

The Spartan-3E family of FPGAs is specifically designed to meet the needs of high volume, cost-sensitive consumer electronic applications. Because of their exceptionally low cost, Spartan-3E FPGAs are ideally suited to a wide range of applications, including broadband access, home networking, display/projection, and digital television equipment [26]. The Spartan-3E FPGA family has five members that offer densities ranging from 100,000 to 1.6 million system gates, as shown in Table 6.1.

Table 6.1 Summary of Spartan-3E FPGA family attributes

Device	System Gates	Equiv. Logic Cells	CLB Array (One CLB = Four Slices)				Distr. RAM bits	Block RAM bits	Dedicated Multipliers	DCMs	Max. User I/Os	Max. Diff. I/O Pairs
			Rows	Col.s	Total CLBs	Total Slices						
XC3S100E	100K	2,160	22	16	240	960	15K	72K	4	2	108	40
XC3S250E	250K	5,508	34	26	612	2,448	38K	216K	12	4	172	68
XC3S500E	500K	10,476	46	34	1,164	4,656	73K	360K	20	4	232	92
XC3S1200E	1200K	19,512	60	46	2,168	8,672	136K	504K	28	8	304	124
XC3S1600E	1600K	33,192	76	58	3,688	14,752	231K	648K	36	8	376	156

6.1.1. Spartan-3E FPGA Features

Features of the Spartan-3E FPGA family are:

- Proven advanced 90-nanometer process technology
- Multi-voltage, multi-standard SelectIO™ interface pins
  - Up to 376 I/O pins or 156 differential signal pairs

- LVCMOS, LVTTL, HSTL, and SSTL single-ended signal standards
- 3.3V, 2.5V, 1.8V, 1.5V, and 1.2V signaling
- 622+ Mb/s data transfer rate per I/O
- True LVDS, RSDS, mini-LVDS, differential HSTL/SSTL differential I/O
- Enhanced Double Data Rate (DDR) support
- DDR SDRAM support up to 333 Mb/s

- Abundant, flexible logic resources
  - Densities up to 33,192 logic cells, including optional shift register or distributed RAM support
  - Efficient wide multiplexers, wide logic
  - Fast look-ahead carry logic
  - Enhanced 18 x 18 multipliers with optional pipeline
  - IEEE 1149.1/1532 JTAG programming/debug port
- Hierarchical SelectRAM™ memory architecture
  - Up to 648 Kbits of fast block RAM
  - Up to 231 Kbits of efficient distributed RAM
- Up to eight Digital Clock Managers (DCMs)
  - Clock skew elimination (delay locked loop)
  - Frequency synthesis, multiplication, division
  - High-resolution phase shifting
  - Wide frequency range (5 MHz to over 300 MHz)

- Eight global clocks plus eight additional clocks per each half of device, plus abundant low-skew routing
- Configuration interface to industry-standard PROMs
- Complete Xilinx ISE™ and WebPACK™ development system support
- MicroBlaze™ and PicoBlaze™ embedded processor cores
- Fully compliant 32-/64-bit 33 MHz PCI support
- Low-cost QFP and BGA packaging options
  - Common footprints support easy density migration
  - Pb-free packaging options

6.1.2. Spartan-3E FPGA Architectural Overview

The Spartan-3E family architecture consists of five fundamental programmable functional elements:

- **Configurable Logic Blocks (CLBs)** contain flexible Look-Up Tables (LUTs) that implement logic plus storage elements used as flip-flops or latches. CLBs perform a wide variety of logical functions as well as store data.
- **Input/Output Blocks (IOBs)** control the flow of data between the I/O pins and the internal logic of the device. Each IOB supports bidirectional data flow plus 3-state operation. These blocks support a variety of signal standards, including high-performance differential standards. Double Data-Rate (DDR) registers are also included.
- **Block RAM** provides data storage in the form of 18 Kbit dual-port blocks.
- **Multiplier Blocks** accept two 18-bit binary numbers as inputs and calculate their product.

- **Digital Clock Manager (DCM) Blocks** provide self-calibrating, fully digital solutions for distributing, delaying, multiplying, dividing, and phase-shifting clock signals.

These elements are organized as shown in Figure 6.1. A ring of IOBs surrounds a regular array of CLBs. Each device has two columns of block RAM except for the XC3S100E, which has one column. Each RAM column consists of several 18-Kbit RAM blocks. Each block RAM is associated with a dedicated multiplier. The DCMs are positioned in the center with two at the top and two at the bottom of the device. The XC3S100E has only one DCM at the top and bottom, while the XC3S1200E and XC3S1600E add two DCMs in the middle of the left and right sides.

The Spartan-3E family features a rich network of traces that interconnect all five functional elements, transmitting signals among them. Each functional element has an associated switch matrix that permits multiple connections to the routing.

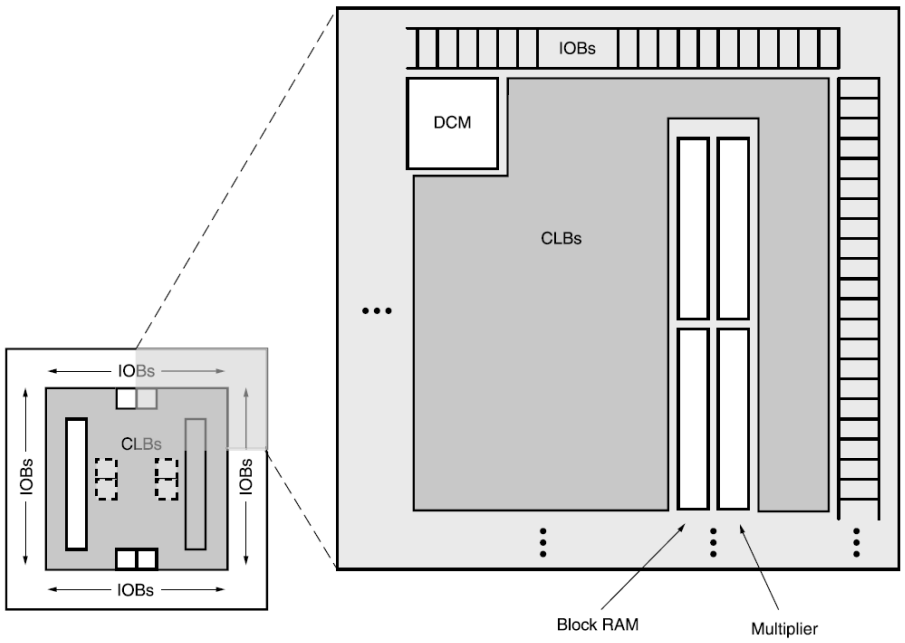


Figure 6.1 Xilinx Spartan-3E FPGA Family Architecture

Further information about the Spartan-3E FPGAs can be found in Xilinx's Spartan-3E Family Datasheet [26].

6.2. Xilinx Spartan-3E Starter Kit

Xilinx's Spartan-3E FPGA Starter Kit board features a 500,000 system gate Spartan-3E FPGA, and a rich selection of peripheral devices and connectors (Figure 6.2).

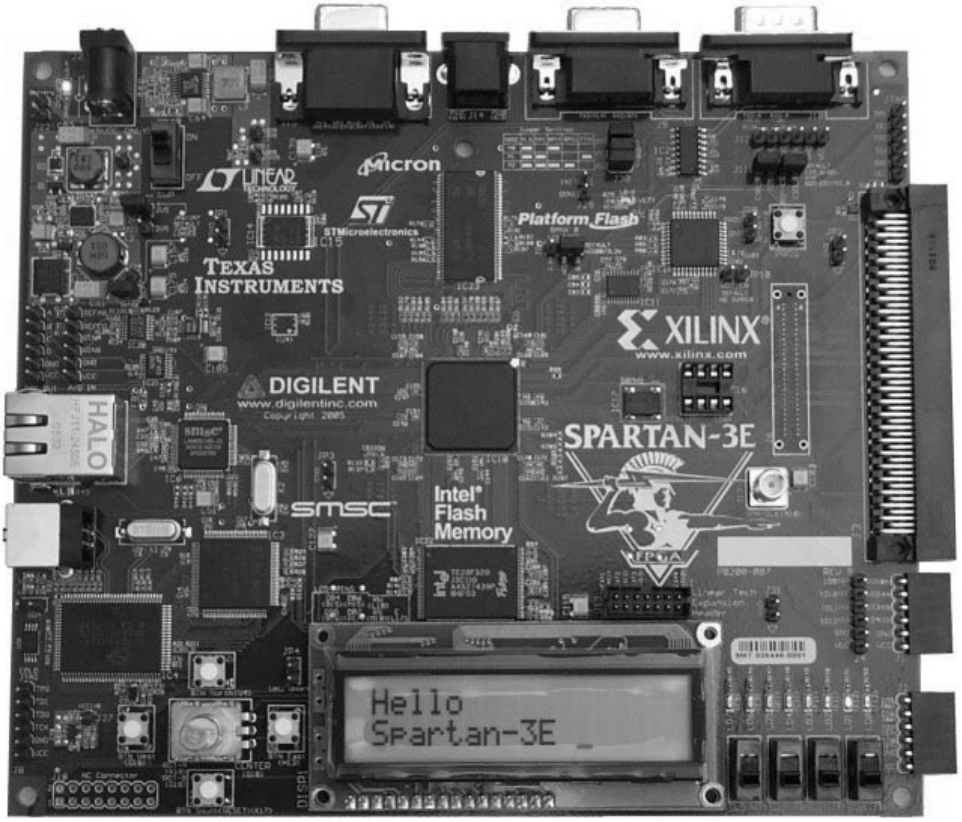


Figure 6.2 Xilinx Spartan-3E Starter Kit board

The key features and components of the Spartan-3E Starter Kit board are:

- Xilinx XC3S500E Spartan-3E FPGA
  - Up to 232 user-I/O pins
  - 320-pin Fine Ball-Grid Array (FBGA) package
  - Over 10,000 logic cells
- Xilinx 4 Mbit Platform Flash configuration PROM

- Xilinx 64-macrocell XC2C64A CoolRunner CPLD
- 64 MByte (512 Mbit) of DDR SDRAM, x16 data interface, 100+ MHz
- 16 MByte (128 Mbit) of parallel NOR Flash (Intel StrataFlash)
  - FPGA configuration storage
  - MicroBlaze code storage/shadowing
- 16 Mbits of SPI serial Flash (STMicro)
  - FPGA configuration storage
  - MicroBlaze code shadowing
- 2-line, 16-character LCD screen
- PS/2 mouse or keyboard port
- VGA display port
- 10/100 Ethernet PHY (requires Ethernet MAC in FPGA)
- Two 9-pin RS-232 ports (DTE- and DCE-style)
- On-board USB-based FPGA/CPLD download/debug interface
- 50 MHz clock oscillator
- SHA-1 1-wire serial EEPROM for bitstream copy protection
- Hirose FX2 expansion connector
- Three Digilent 6-pin expansion connectors
- Four-output, SPI-based Digital-to-Analog Converter (DAC)
- Two-input, SPI-based Analog-to-Digital Converter (ADC) with programmable-gain pre-amplifier

- ChipScope™ SoftTouch debugging port
- Rotary-encoder with push-button shaft
- Eight discrete LEDs
- Four slide switches
- Four push-button switches
- SMA clock input
- 8-pin DIP socket for auxiliary clock oscillator

The board is suitable for development of prototype systems for numerous different application areas, and several reference designs are provided on Xilinx's Spartan-3E Starter Kit web page [24].

6.3. Implementation Details

For test purposes, a small design was implemented using Xilinx's Spartan-3E Starter Kit FPGA development board. The realized circuit utilizes the analog-to-digital (A/D) and digital-to-analog (D/A) converters on the development board for input and output of various signals, with some minimal signal processing done within the FPGA.

Schematics of the A/D and D/A conversion circuitry are given in Figure 6.3 and Figure 6.4, respectively. The A/D conversion circuit comprises an LTC6912-1 programmable gain amplifier and an LTC1407A-1 14-bit analog-to-digital converter (ADC); the D/A conversion circuit uses a single LTC2624 4-channel, 12-bit digital-to-analog converter (DAC).

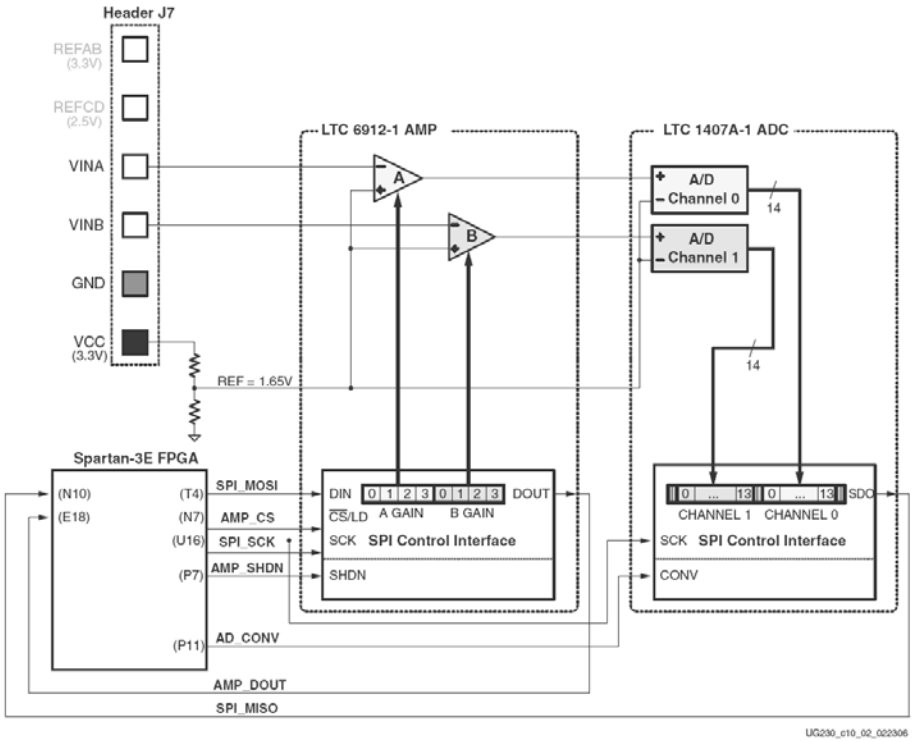


Figure 6.3 Spartan-3E Starter Kit A/D conversion circuitry

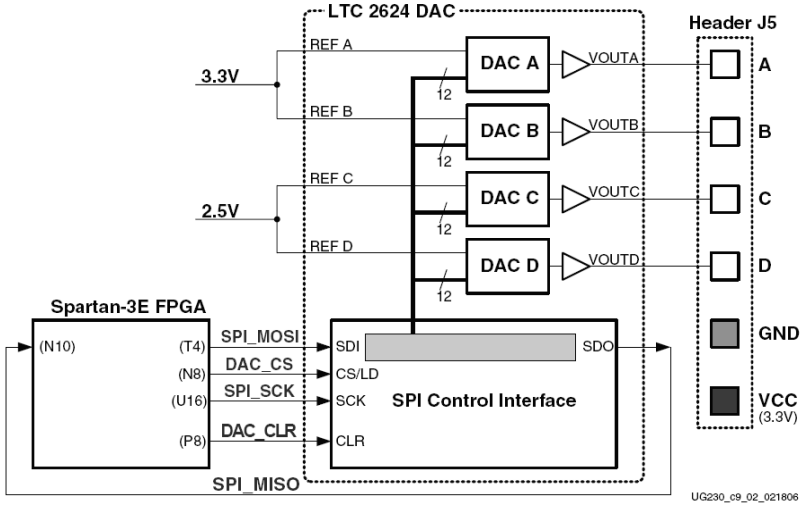


Figure 6.4 Spartan-3E Starter Kit D/A conversion circuitry

The amplifier, ADC and DAC are connected to the FPGA over a shared serial bus architecture called the Serial Peripheral Interface (SPI). Because the bus is shared between components, a bus arbiter is required that will supervise accesses to the SPI bus and allow only one component to communicate at a time. The required signal timings and

communication schemes for the amplifier, ADC, and DAC can be found in their respective datasheets [27, 28, 29].

The HDL code comprising the circuit description is divided among several Verilog files, each containing one module that controls a single component on the board. The Verilog design files are reproduced in Appendix A. This design style is in compliance with the divide-and-conquer and top-down design methodologies, which have been widely adopted by the electronics industry today as the most efficient design flow.

One of the modules is responsible for configuring the programmable gain amplifier at startup. One module drives the ADC, periodically reading the samples generated from its two analog inputs. Another module drives the DAC and sends samples to it in sync with the ADC's sampling clock. The control modules have to handle their own SPI clock generation and data transfer operations.

In between the ADC and DAC driver blocks lie two copies of a simple module that just passes the samples coming from the ADC on to the DAC without modification. This simple module practically acts as a placeholder in the design, and can be easily replaced with more complex modules that implement advanced digital signal processing (DSP) algorithms.

The remaining two channels of the DAC are connected to two copies of a simple wave generator function. The frequencies of the generated waves can be adjusted using the rotary encoder.

6.4. Synthesis Results For The Sample Circuit

Despite its small scale, the sample circuit comprises a wide variety of fundamental logic elements, such as multiplexers, adders, counters, shift registers and finite state machines. These elements are readily recognized by synthesis tools and can be subjected to additional optimizations specific to each element type. The sample circuit presents many such optimization opportunities thanks to the aforementioned variety of logic elements it contains, and thus makes a good test case for the synthesis tool.

Xilinx's XST software provides many settings that allow the user to tweak the synthesis process. Out of those, only three were chosen (namely the optimization goal, optimization effort level, and FSM encoding settings) and the sample circuit was synthesized several times with different combinations of these three settings; all the other settings were always kept at their default values. Synthesis results are given in Table 6.2 in terms of the number of slices the design occupies, and the maximum clock frequency that the synthesized circuit can run at.

Table 6.2 Synthesis results for the sample circuit

FSM Encoding	Optimized for Speed		Optimized for Area	
	Normal Effort	**High Effort**	**Normal Effort**	**High Effort**
Auto	233 slices 178.54MHz	236 slices 179.66MHz	235 slices 139.55MHz	235 slices 139.55MHz
One-Hot	225 slices 174.40MHz	226 slices 162.39MHz	214 slices 160.88MHz	214 slices 160.88MHz
Compact	232 slices 185.29MHz	235 slices 186.78MHz	217 slices 147.19MHz	217 slices 147.19MHz
Sequential	234 slices 177.84MHz	237 slices 176.34MHz	234 slices 139.90MHz	234 slices 139.90MHz
Gray	242 slices 167.73MHz	243 slices 176.99MHz	221 slices 135.10MHz	221 slices 135.10MHz
Johnson	251 slices 176.37MHz	252slices 174.03MHz	218 slices 151.15MHz	218 slices 151.15MHz
Speed1	230 slices 169.49MHz	231 slices 180.80MHz	218 slices 160.62MHz	218 slices 160.62MHz

The fastest circuit is obtained by using the Compact encoding for the FSMs in the design, which yielded a circuit with a maximum clock frequency of 186.78 MHz, occupying 235 slices out of the 4656 slices available in the XC3S500E FPGA. Using One-Hot encoding and setting the optimization goal to "Area" produced the smallest circuit with 214 occupied slices, but the maximum attainable clock frequency dropped to 160.88 MHz.

When optimizing for speed, setting the FSM encoding type to "Automatic" gives acceptable results, whereas in area optimization the obtained slice count is quite higher than the One-Hot encoded case, proving once more that the synthesis tool should not be

trusted blindly and several synthesis iterations should be performed, exploring different combinations of settings and comparing the results.

The FSM encoding type setting is a global setting that forces the encoding of all FSMs in the design to the same type. The desired encoding for an FSM can also be explicitly specified by including the encoding declaration in the Verilog source code. Using this inline encoding specification, different encodings can be assigned to different FSMs in the design, but exploring all possible combinations would take days even for this small design.

In its current state, the sample design provides easy access to the two analog inputs and four analog outputs on the Spartan-3E Starter Kit development board. These inputs and outputs can be connected to any module within the top-level block of the design. As mentioned previously, the included placeholder DSP modules and waveform generators can be replaced with any arbitrary module to perform digital signal processing or digital control tasks. The Verilog source code listings for the design are given in Appendix A, and they can be used as a reference in future Spartan-3E Starter Kit projects requiring A/D or D/A conversion functionality.

7. CONCLUSION

Due to the increasing complexity of digital systems, computer-based circuit synthesis has become indispensable for digital design today. The term “synthesis” covers a wide range of EDA activities. High-level synthesis uses an abstract behavioral or register-transfer level description and refines this into simple gates. Low-level synthesis is the term used to address the issues associated with technology mapping and in some cases the physical placement and routing of a design. Ideally, these tools would allow design entry at an abstract, functional level, regardless of what the target hardware architecture will be, and then synthesize the optimal logic circuit that implements the described system within specific area, speed and power consumption constraints. However, in practice, synthesis tools are not capable of autonomously performing all design and optimization steps necessary for realizing a digital system; they require human intervention and guidance at various stages of the process.

Although HDLs, together with synthesis technology, promise hardware independent behavioral design, writing the HDL code with the target technology in mind always provides better results. Synthesis tools cannot magically turn HDL code into hardware if the design target does not provide some key functional elements that the behavioral description calls for. The design engineer has to be aware of what logic resources are available in the target technology, and then decide on a system architecture that is suitable for implementation in that technology.

Choosing the right architecture alone does not guarantee optimal realization. In order to obtain the best results from synthesis, the HDL coding guidelines set forth by the tool vendor has to be followed to the letter. Otherwise, the synthesis tool cannot correctly recognize some design elements and properly optimize them. The resulting circuit might still be functionally identical, but it will not be the optimal implementation.

Synthesis tools present many configuration options for fine tuning their optimization algorithms. These are normally set to default values that produce acceptable results most of the time. However, an experienced engineer who is familiar with the synthesis tool and

who has knowledge of the algorithms that the tool employs, will be able to configure these settings much more appropriately according to the design at hand.

When all these conditions are met, the synthesis tool is supposed to perform at its best. Nevertheless, the tool should not be trusted blindly. It is the responsibility of the engineer to check the software's synthesis reports for issued warnings and errors, misinterpretations of the HDL code, or ineffective optimization settings. Such problems require corrections to the HDL code and/or re-adjustment of the software settings, followed by another synthesis run. Every design flow involves some trial-and-error as several architectural changes and different software settings are explored.

The only way to ensure that a synthesized circuit is the optimal one is by performing an exhaustive search within the set of all possible optimizations. This is a very compute-intensive task and the necessary processing time grows exponentially with increasing design size, making it impossible to apply this technique to current day designs; a single run of synthesis and subsequent verification with gate-level simulation can take hours, even days, for multi-million gate circuits. Heuristics and artificial intelligence is used to reduce the size of the solution set that will be searched through, but the optimal solution found within this reduced set may not be the one that is universally optimal.

With companies rushing to meet narrow, two-to-three month market windows to stay competitive, most of them cannot afford the time to experiment with synthesis tools and run hundreds of design iterations. If the synthesized circuit still does not meet the area constraint after several attempts, the designers may have to increase the die size or move to a higher capacity FPGA, as a slightly more expensive product is better than no product at all. The search for the universally optimal solution is left to the researchers at universities and synthesis software vendors.

Advances in synthesis technology might one day allow anyone with some basic programming skills to design digital systems. Today, however, the industry still needs skilled engineers with a keen intuition and a solid background in the principles of logic circuit design.

APPENDIX A. VERILOG SOURCE CODE LISTINGS FOR THE SAMPLE FPGA IMPLEMENTATION

A.1. main.v

```
module main(clk50mhz,                          // Clock
            rot_a, rot_b, rot_press,           // Rotary encoder
            btn_south, btn_west, btn_east,
            led,                               // Outputs
            spi_sdi, spi_amp_sdo, spi_amp_cs,  // Programmable amplifier ports
            spi_amp_shdn,
            spi_sdo, spi_sck, spi_adc_conv,    // ADC ports
            spi_dac_cs, spi_dac_clr,           // DAC ports
            spi_rom_cs,                        // SPI Flash chip select
            strataflash_oe, strataflash_ce,    // StrataFlash ports
            strataflash_we,
            platformflash_oe                   // Platform Flash ports
);

  input clk50mhz;
  input rot_a, rot_b, rot_press;
  input btn_south, btn_west, btn_east;
  input spi_sdo, spi_amp_sdo;

  output [7:0] led;
  output spi_sdi, spi_amp_cs, spi_amp_shdn;
  output spi_sck, spi_adc_conv;
  output spi_dac_cs, spi_dac_clr;
  output spi_rom_cs;
  output strataflash_oe, strataflash_ce, strataflash_we;
  output platformflash_oe;

  reg amp_set = 0, adc_en = 0, dac_en = 0, dsp_en = 0;
  reg [4:0] sys_state = 0;
  reg [8:0] count50k = 0;
  reg clk50khz = 0;
  reg spi_sck = 0, spi_sdi = 0;
  reg [2:0] gain = 1;

  wire rot_event, rot_left;
  wire spi_busy, amp_busy, adc_busy, dac_busy;
  wire dsp_busy, dsp_busy1, dsp_busy2, dsp_busy3, dsp_busy4;
```

```
wire adc_sck, dac_sck, dac_sdi, amp_sck, amp_sdi;
wire [15:0] adc_sample_a, adc_sample_b;
wire [15:0] dac_sample_a, dac_sample_b, dac_sample_c, dac_sample_d;
wire [3:0] gain_in, pgain_a, pgain_b;

// Disable unused components
assign spi_rom_cs = 1;
assign strataflash_oe = 1;
assign strataflash_ce = 1;
assign strataflash_we = 1;
assign platformflash_oe = 0;

// Connect board outputs and LEDs
assign led = btn_west ? adc_sample_a[15:8] : (btn_east ? adc_sample_a[7:0] \
                : {(pgain_a & pgain_b), gain_in});

assign gain_in = {1'b0, gain};
assign spi_busy = amp_busy | adc_busy | dac_busy;
assign dsp_busy = dsp_busy1 | dsp_busy2 | dsp_busy3 | dsp_busy4;

// Instantiate modules
rotenc read_rotary(clk50mhz, rot_a, rot_b, rot_event, rot_left);

amp_conf amplifier(clk50mhz, amp_set, amp_busy, gain_in, gain_in,
                   pgain_a, pgain_b, amp_sdi, amp_sck, spi_amp_cs,
                   spi_amp_sdo, spi_amp_shdn);

adc_driver read_adc(clk50mhz, adc_en, adc_busy, spi_sdo,
                    spi_adc_conv, adc_sck, adc_sample_a, adc_sample_b);

dac_driver write_dac(clk50mhz, dac_en, dac_busy, dac_sdi, spi_dac_cs,
                     spi_dac_clr, dac_sck, dac_sample_a, dac_sample_b,
                     dac_sample_c, dac_sample_d);

dsp_placeholder dsp_core1(clk50mhz, dsp_en, dsp_busy1,
                          adc_sample_a, dac_sample_a);
dsp_placeholder dsp_core2(clk50mhz, dsp_en, dsp_busy2,
                          adc_sample_b, dac_sample_b);

wavegen wg3(clk50mhz, dsp_en, dsp_busy3, 4'b0100, dac_sample_c);
wavegen wg4(clk50mhz, dsp_en, dsp_busy4, 4'b1010, dac_sample_d);

// SPI bus arbiter
always @(amp_busy or adc_busy or dac_busy or amp_sck or adc_sck or dac_sck)
  casex ({amp_busy, adc_busy, dac_busy})
```

```
      3'b100: spi_sck <= amp_sck;
      3'b010: spi_sck <= adc_sck;
      3'b001: spi_sck <= dac_sck;
      3'bxxx: spi_sck <= 0;
    endcase

  always @(amp_busy or dac_busy or amp_sdi or dac_sdi)
    casex ({amp_busy, dac_busy})
      2'b10: spi_sdi <= amp_sdi;
      2'b01: spi_sdi <= dac_sdi;
      2'bxx: spi_sdi <= 0;
    endcase

  // Process rotary encoder
  always @(posedge clk50mhz)
    if (rot_event == 1)
      if (rot_left == 1)
        gain <= gain + 1;
      else
        gain <= gain - 1;

  // Generate 50kHz sampling clock
  always@(posedge clk50mhz)
    if (count50k == 499)
      begin
        count50k <= 0;
        clk50khz <= ~clk50khz;
      end
    else
      count50k <= count50k + 1;

// System control loop
  always @(posedge clk50mhz)
    case (sys_state)
      0:
        begin
          amp_set <= 1;
          sys_state <= 1;
        end
      1:
        if (spi_busy == 1)
          begin
            amp_set <= 0;
            sys_state <= 2;
          end
```

```
2:
  if (spi_busy == 0)
    begin
      if (rot_press == 1 || btn_south == 1)
        sys_state <= 0;
      else if (clk50khz == 1)
        begin
          adc_en <= 1;
          sys_state <= 3;
        end
    end
3:
  if (spi_busy == 1)
    begin
      adc_en <= 0;
      sys_state <= 4;
    end
4:
  if (spi_busy == 0)
    begin
      dac_en <= 1;
      sys_state <= 5;
    end
5:
  if (spi_busy == 1)
    begin
      dac_en <= 0;
      sys_state <= 6;
    end
6:
  if (spi_busy == 0)
    begin
      dsp_en <= 1;
      sys_state <= 7;
    end
7:
  if (dsp_busy == 1)
    begin
      dsp_en <= 0;
      sys_state <= 8;
    end
8:
  if (dsp_busy == 0)
      sys_state <= 9;
```

```
      9:
        if (clk50khz == 0)
            sys_state <= 2;
    endcase
endmodule
```

A.2. rotary.v

```
module rotenc(clk, rot_a, rot_b, rot_event, rot_left);
  input clk;
  input rot_a, rot_b;

  output rot_event;
  output rot_left;

  reg rot_q1, previous_rot_q1, rot_q2, rot_event, rot_left;

  always@(posedge clk)
    case ({rot_b, rot_a})
      2'b00 : begin
                rot_q1 <= 0;
                rot_q2 <= rot_q2;
              end
      2'b01 : begin
                rot_q1 <= rot_q1;
                rot_q2 <= 0;
              end
      2'b10 : begin
                rot_q1 <= rot_q1;
                rot_q2 <= 1;
              end
      2'b11 : begin
                rot_q1 <= 1;
                rot_q2 <= rot_q2;
              end
      default : begin
                  rot_q1 <= rot_q1;
                  rot_q2 <= rot_q2;
                end
    endcase

  always@(posedge clk)
    begin
      previous_rot_q1 <= rot_q1;
```

```
      if (rot_q1 == 1 && previous_rot_q1 == 0)
        begin
          rot_event <= 1;
          rot_left <= rot_q2;
        end
      else
        begin
          rot_event <= 0;
          rot_left <= rot_left;
        end
    end
endmodule
```

A.3. amp_conf.v

```
module amp_conf(clk50mhz, set_gain, busy, gain_a_in, gain_b_in,
                prev_gain_a, prev_gain_b, spi_sdi, spi_sck, spi_amp_cs,
                spi_amp_sdo, spi_amp_shdn);

  input clk50mhz, set_gain, spi_amp_sdo;
  input [3:0] gain_a_in, gain_b_in;

  output busy, spi_sdi, spi_sck, spi_amp_cs, spi_amp_shdn;
  output [3:0] prev_gain_a, prev_gain_b;

  reg [4:0] state = 0;
  reg [7:0] gain = 0;
  reg [7:0] prev_gain = 0;
  reg [2:0] count5m = 0;
  reg clk5mhz = 0;
  reg spi_amp_cs = 1;
  reg spi_sck = 0;
  reg busy = 0;

  assign spi_sdi = gain[7];
  assign spi_amp_shdn = 0;
  assign {prev_gain_b, prev_gain_a} = prev_gain;

  // Generate 5MHz clock for amplifier SPI communication
  always@(posedge clk50mhz)
    if (count5m == 4)
      begin
        count5m <= 0;
        clk5mhz <= ~clk5mhz;
```

```
    end
  else
    count5m <= count5m + 1;

always @(posedge clk5mhz)
  casex (state)
    0:
      if (set_gain == 1)
        begin
          busy <= 1;
          gain <= {gain_b_in, gain_a_in};
          spi_amp_cs <= 0;
          spi_sck <= 0;
          state <= 1;
        end

    // Read previous gain and send new gain are done, don't shift once more
    // Just return SCK to zero
    16:
      begin
        spi_sck <= ~spi_sck;
        state <= 17;
      end

    17:
      begin
        busy <= 0;
        spi_amp_cs <= 1;
        state <= 0;
      end

    5'bxxxx0:
      begin
        gain <= gain << 1;
        prev_gain <= prev_gain << 1;
        spi_sck <= ~spi_sck;
        state <= state + 1;
      end

    5'bxxxx1:
      begin
        spi_sck <= ~spi_sck;
        prev_gain[0] <= spi_amp_sdo;
        state <= state + 1;
      end
```

```
    endcase
endmodule
```

A.4. adc_driver.v

```
module adc_driver(clk, trigger, busy, spi_sdo, spi_adc_conv, spi_sck,
                  sample_a, sample_b);
  input clk, trigger;
  input spi_sdo;

  output busy, spi_adc_conv, spi_sck;
  output [15:0] sample_a, sample_b;

  reg busy = 0;
  reg spi_adc_conv = 0;
  reg spi_sck = 0;
  reg [6:0] state = 0;
  reg [15:0] sample_a, sample_b;
  reg [27:0] sample_word = 0;

  always @(posedge clk)
    casex (state)
      0:
        if (trigger == 1)
          begin
            busy <= 1;
            spi_adc_conv <= 1;
            spi_sck <= 0;
            state <= 1;
          end
      1:
        begin
          spi_adc_conv <= 0;
          state <= 2;
        end
      70:
        begin
          busy <= 0;
          state <= 0;
          sample_a <= {sample_word[27], sample_word[27], sample_word[27:14]};
          sample_b <= {sample_word[13], sample_word[13], sample_word[13:0]};
        end
      2, 3, 4, 5, 34, 35, 36, 37, 66, 67, 68, 69, 7'bxxxxxx0:
        begin
```

```
          spi_sck <= ~spi_sck;
          state <= state + 1;
        end
      7'bxxxxxx1:
        begin
          spi_sck <= ~spi_sck;
          state <= state + 1;
          sample_word <= {sample_word[26:0], spi_sdo};
        end
    endcase
endmodule
```

A.5. dac_driver.v

```
module dac_driver(clk, trigger, busy, spi_sdi, spi_dac_cs, spi_dac_clr, spi_sck,
                  sample_a_in, sample_b_in, sample_c_in, sample_d_in);
  input clk;
  input trigger;
  input [15:0] sample_a_in, sample_b_in, sample_c_in, sample_d_in;

  output busy;
  output spi_sdi;
  output spi_dac_cs;
  output spi_dac_clr;
  output spi_sck;

  reg busy = 0, ch_start = 0, ch_busy = 0;
  reg spi_dac_cs = 1;
  reg spi_sck = 0;
  reg [3:0] state_dac = 0;
  reg [6:0] state_ch = 0;
  reg [1:0] num_ch = 0;
  reg [15:0] sample_ch;
  reg [23:0] dac_dataword;

  wire spi_sdi;

  assign spi_dac_clr = 1;
  assign spi_sdi = dac_dataword[23];

  always @(posedge clk)
    casex (state_dac)
      0:
        if (trigger == 1)
```

```
    begin
      busy <= 1;
      sample_ch <= sample_a_in;
      ch_start <= 1;
      state_dac <= 1;
    end
1:
  begin
    ch_start <= 0;
    state_dac <= 2;
  end
2:
  if (ch_busy == 0)
    begin
      num_ch <= 1;
      sample_ch <= sample_b_in;
      ch_start <= 1;
      state_dac <= 3;
    end
3:
  begin
    ch_start <= 0;
    state_dac <= 4;
  end
4:
  if (ch_busy == 0)
    begin
      num_ch <= 2;
      sample_ch <= sample_c_in;
      ch_start <= 1;
      state_dac <= 5;
    end
5:
  begin
    ch_start <= 0;
    state_dac <= 6;
  end
6:
  if (ch_busy == 0)
    begin
      num_ch <= 3;
      sample_ch <= sample_d_in;
      ch_start <= 1;
      state_dac <= 7;
    end
```

```
    7:
      begin
        ch_start <= 0;
        state_dac <= 8;
      end
    8:
      if (ch_busy == 0)
        begin
          num_ch <= 0;
          busy <= 0;
          state_dac <= 0;
        end
  endcase

always @(posedge clk)
  casex (state_ch)
    0:
      if (ch_start == 1)
        begin
          ch_busy <= 1;
          // Command is 0000 for channels 1,2,3 and 0010 for channel 4
          // Address is the same as the channel number
          dac_dataword <= {6'b001100, num_ch, sample_ch};
          spi_dac_cs <= 0;
          spi_sck <= 0;
          state_ch <= state_ch + 1;
        end
    48:
      begin
        ch_busy <= 0;
        spi_dac_cs <= 1;
        state_ch <= 0;
      end
    7'bxxxxxx0:
      begin
        spi_sck <= ~spi_sck;
        dac_dataword <= dac_dataword << 1;
        state_ch <= state_ch + 1;
      end

    7'bxxxxxx1:
      begin
        spi_sck <= ~spi_sck;
        state_ch <= state_ch + 1;
      end
```

```
    endcase
endmodule
```

A.6. dsp_module.v

```
module dsp_placeholder(clk, enable, busy, sample_in, sample_out);
  input clk;
  input [15:0] sample_in;
  input enable;

  output busy;
  output [15:0] sample_out;

  reg busy;
  reg [2:0] state = 0;
  reg [15:0] sample;
  reg [15:0] sample_out = 0;

  always @(posedge clk)
    case (state)
      0:
        if (enable == 1)
          begin
            busy <= 1;
            sample <= sample_in;
            state <= 1;
          end
      1:
        begin
          sample_out <= (sample + 16'b0010_0000_0000_0000) << 2;
          state <= 2;
        end
      2:
        begin
          busy <= 0;
          state <= 0;
        end
    endcase
endmodule
```

A.7. wavegen.v

```
module wavegen(clk, enable, busy, increment, sample_out);
  input clk;
  input enable;
  input [3:0] increment;

  output busy;
  output [15:0] sample_out;

  reg busy = 0;
  reg [15:0] count = 0;
  reg [1:0] state = 0;

  assign sample_out = count;

  always @(posedge clk)
    case (state)
      0:
        if (enable == 1 && busy == 0)
          begin
            busy <= 1;
            state <= 1;
          end
      1:
        begin
          count <= count + increment;
          state <= 2;
        end
      2:
        begin
          count <= count + increment;
          state <= 3;
        end
      3:
        begin
          busy <= 0;
          state <= 0;
        end
    endcase
endmodule
```

REFERENCES

1. Intel Corp.; *New Dual-Core Intel® Itanium® 2 Processor Doubles Performance*; 2006, http://www.intel.com/pressroom/archive/releases/20060718comp.htm

2. Marschner, E.; Berman, V.; *The continuing evolution of EDA standards*, IEEE Design & Test of Computers, Vol. 21, Issue 5, pp.450 – 451, Sep-Oct 2004.

3. Mauskar, A.; Improving timing closure with physical synthesis; 2004, www.eetimes.com/news/design/features/showArticle.jhtml?articleID=21400427

4. Dervişoğlu, A.; Lojik Devreler Ders Notları; 2002.

5. Parnell, K.; Mehta, N.; *Programmable Logic Design: Quick Start Handbook*; Xilinx, 2004.

6. Kang, S.; Leblebici, Y.; *CMOS Digital Integrated Circuits: Analysis and Design, 3rd Ed.*; ISBN: 0071196447, McGraw-Hill, 2003.

7. Maxfield, C.; *Bebop To The Boolean Boogie: An Unconventional Guide To Electronics Fundamentals, Components, And Processes, 2nd Edition*; ISBN: 0750675438, Elsevier, 2003.

8. Maxfield, C.; *The Design Warrior's Guide to FPGAs: Devices, Tools, and Flows*; ISBN: 0750676043, Elsevier, 2004.

9. Zeidman, B.; Designing with FPGAs & CPLDs; ISBN: 1588201128, CMP Books, 2002.

10. Maxfield, C.; *Designus Maximus Unleashed*; ISBN: 0750690895, Newnes, 1998.

11. Pescovitz, D.; *1972: The release of SPICE, still the industry standard tool for integrated circuit design*; http://www.coe.berkeley.edu/labnotes/0502/history.html

12. IEEE Standards Association; *IEEE Std 1364-1995 IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*; http://standards.ieee.org/reading/ieee/std_public/description/dasc/1364-1995_desc.html

13. IEEE Standards Association; *IEEE and Accellera Announce the Approval of Verilog-2001 as a Revised IEEE Standard*; 2001, http://standards.ieee.org/announcements/verilog2001.html

14. IEEE Standards Association; *IEEE Approves SystemVerilog And Verilog Standards For Electronic Design*; 2006, http://standards.ieee.org/announcements/pr_p1364-1800.html

15. VASG: VHDL Analysis and Standardization Group; http://www.eda.org/vhdl-200x/

16. Bailey, S.; *Comparison of VHDL, Verilog and SystemVerilog*; Model Technology Digital Simulation White Paper, 2003.

17. Smith, Douglas J.; *HDL Chip Design*; ISBN: 0965193438, Doone Publications, 1996.

18. Ciletti, M. D.; *Advanced Digital Design with the Verilog HDL*; ISBN: 0130891614; Pearson Education, Inc., 2003.

19. Brown, S.; Vranesic, Z.; Fundamentals of Digital Logic with Verilog Design; ISBN: 0072823151, McGraw-Hill, 2003.

20. Wolf, W.; *FPGA-Based System Design*; ISBN: 0131424610, Prentice Hall, 2004.

21. Xilinx, Inc.; *XST v8.2i User Guide*; 2006.

22. Chapman, K.; *Get Smart About Reset (Think Local, Not Global)*; Xilinx TechXclusives, 2001; http://www.xilinx.com/xlnx/xweb/xil_tx_home.jsp

23. Chapman, K.; *Get Your Priorities Right - Make Your Design Up To 50% Smaller*; Xilinx TechXclusives, 2004; http://www.xilinx.com/xlnx/xweb/xil_tx_home.jsp

24. Xilinx Spartan-3E Starter Kit Web Page, http://www.xilinx.com/s3estarter

25. Xilinx ISE WebPACK, http://www.xilinx.com/ise/logic_design_prod/webpack.htm

26. Xilinx Spartan-3E FPGA Family Data Sheet, 2006, Xilinx Inc., http://www.xilinx.com/bvdocs/publications/ds312.pdf

27. LTC6912 Dual Programmable Gain Amplifiers with Serial Digital Interface; www.linear.com/pc/productDetail.do?navId=H0,C1,C1154,C1009,C1121,P7596

28. LTC1407A-1 Serial 14-Bit 3Msps Simultaneous Sampling ADC with Shutdown; www.linear.com/pc/productDetail.do?navId=H0,C1,C1155,C1001,C1158,P2485

29. LTC2624 Quad 12-Bit Rail-to-Rail DAC in 16-Lead SSOP; www.linear.com/pc/productDetail.do?navId=H0,C1,C1155,C1005,C1156,P2048

REFERENCES NOT CITED

1. FPGA and Structured ASIC Journal; http://www.fpgajournal.com/

2. SOCCentral; http://www.soccentral.com/

3. EETimes Online; http://www.eetimes.com/

4. BOOM-II The Boolean Minimizer; http://service.felk.cvut.cz/vlsi/prj/BOOM/

5. The International Technology Roadmap for Semiconductors; http://www.itrs.net/

6. Synplicity, Inc.; http://www.synplicity.com/

7. Synopsys, Inc.; http://www.synopsys.com/

8. Mentor Graphics Corp.; http://www.mentor.com/

9. Taiwan Semiconductor Manufacturing Company Limited; http://www.tsmc.com/

10. United Microelectronics Corp.; http://www.umc.com/