Programmable Logic Design

Quick Start Guide

UG500 (v1.0) May 8, 2008

XILINX®

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
05/15/08	1.0	Initial Xilinx release.

About This Guide

Whether you design with discrete logic, base all of your designs on microcontrollers, or simply want to learn how to use the latest and most advanced programmable logic software, you will find this book an interesting insight into a different way to design.

Programmable logic devices were invented in the late 1970s and have since proved to be very popular, becoming one of the largest growing sectors in the semiconductor industry. Why are programmable logic devices so widely used? Besides offering designers ultimate flexibility, programmable logic devices also provide a time-to-market advantage and design integration. Plus, they are easy to design with and can be reprogrammed time and time again – even in the field – to upgrade system functionality.

This book details the history of programmable logic devices; where and how to use them; how to install the free, fully functioning design software (Xilinx WebPACK ISE software is included with this book); and then guides you through your first designs. After you have finished your first design, this book will prove useful as a reference guide or quick start handbook. There are also sections on VHDL and schematic capture design entry, as well as a data bank of useful applications examples. I hope you find this book practical, informative, and above all easy to use.

Nick Mehta

Navigating This Book

This book was written for both the professional engineer who has never designed using programmable logic devices and for the new engineer embarking on an exciting career in electronics design. To accommodate these two audiences, we offer the following navigation section, to help you decide in advance which sections would be most useful.

CHAPTER 1: INTRODUCTION

Chapter 1 is an overview of how and where PLDs are used and gives a brief history of programmable logic devices.

CHAPTER 2: XILINX SILICON SOLUTIONS

Chapter 2 describes the different silicon products offered by Xilinx. The Xilinx portfolio includes CPLD and FPGA devices.

CHAPTER 3: XILINX DESIGN SOFTWARE

Chapter 3 describes the software flow for CPLD and FPGA devices. It also introduces the Xilinx ISE WebPACK design software detailing the procedure necessary to successfully install the software.

CHAPTER 4: WEBPACK ISE DESIGN ENTRY

Chapter 4 is a step-by-step approach to your first design. The following pages are intended to demonstrate the basic PLD design entry implementation process.

CHAPTER 5: IMPLEMENTING CPLD DESIGNS

Chapter 5 discusses the synthesis and implementation process for CPLDs. The design targets a CoolRunner™-II CPLD.

CHAPTER 6: IMPLEMENTING FPGA DESIGNS

Chapter 6 discusses the synthesis and implementation process for FPGAs. The design targets a Spartan®-3E that is available on the demo board of the Spartan-3E Design Kit. The design is the same design as described in previous chapters, but targets a Spartan-3E FPGA instead.

CHAPTER 7: DESIGN REFERENCE BANK

Chapter 7 contains a useful list of design examples and applications that will give you a jump start into your future programmable logic designs. This section also offers pointers on where to locate and download code and IP cores from the Xilinx website.

Conventions

Convention	Meaning or Use	Example
`Courier font`	Messages, prompts, and program files that the system displays	`speed grade: - 100`
`Courier bold`	Literal commands that you enter in a syntactical statement	**`ngdbuild`** *design_name*
Helvetica bold	Commands that you select from a menu	**File → Open**
	Keyboard shortcuts	**Ctrl+C**
Italic font	Variables in a syntax statement for which you must supply values	**`ngdbuild`** *design_name*
	References to other manuals	See the *Development System Reference Guide* for more information.
	Emphasis in text	If a wire is drawn so that it overlaps the pin of a symbol, the two nets are *not* connected.
Square brackets [ ]	An optional entry or parameter. However, in bus specifications, such as **`bus[7:0]`**, they are required.	**`ngdbuild`** [*option_name*] *design_name*
Braces { }	A list of items from which you must choose one or more	**`lowpwr =`**{**`on`**|**`off`**}

Convention	Meaning or Use	Example
Vertical bar |	Separates items in a list of choices	**lowpwr =**{**on**|**off**}
Vertical ellipsis . . .	Repetitive material that has been omitted	IOB #1: Name = QOUT' IOB #2: Name = CLKIN' . . .
Horizontal ellipsis ...	Repetitive material that has been omitted	**allow block** *block_name loc1 loc2 ... locn;*

Chapter 1

Introduction

The History of Programmable Logic

By the late 1970s, standard logic devices were all the rage, and printed circuit boards were loaded with them. Then someone asked, "What if we gave designers the ability to implement different interconnections in a bigger device?" This would allow designers to integrate many standard logic devices into one part.

To offer the ultimate in design flexibility, Ron Cline from Signetics (which was later purchased by Philips and then eventually Xilinx) came up with the idea of two programmable planes. These two planes provided any combination of "AND" and "OR" gates, as well as sharing of AND terms across multiple ORs.

This architecture was very flexible, but at the time wafer geometries of 10 µm made the input-to-output delay (or propagation delay) high, which made the devices relatively slow. The features of the PLA were:

- Two programmable ground planes
- Any combination of ANDs/ORs
- Sharing of AND terms across multiple ORs
- Highest logic density available to user
- High fuse count; slower than PALs
- Programmable logic array

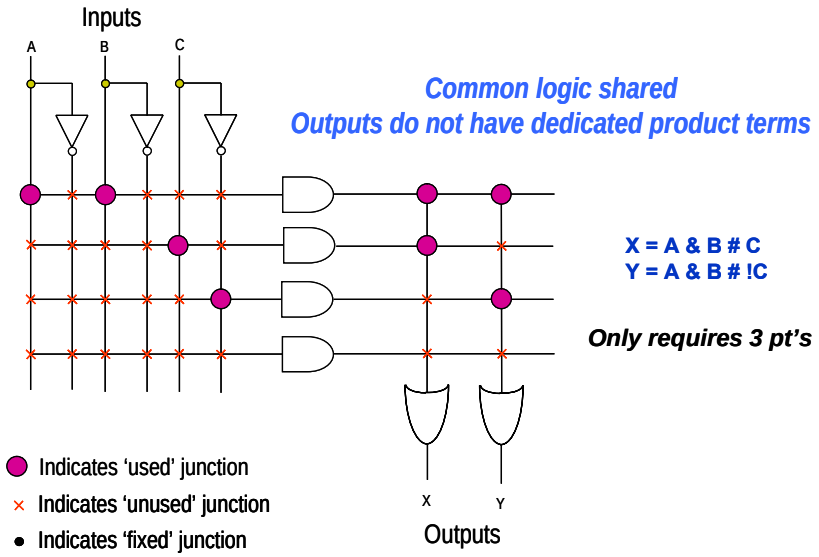


Figure 1-1: **Simple PLA**

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

This new architecture differed from that of the PLA in that one of the programmable planes was fixed – the OR array. PAL architecture also had the added benefit of faster t_{PD} and less complex software, but without the flexibility of the PLA structure.

Other architectures followed, such as the PLD. This category of devices is often called Simple PLD.

- One programmable plane: AND/Fixed OR
- Finite combination of ANDs/ORs
- Medium logic density available to user
- Lower fuse count; faster than PLAs (at the time, fabricated on a 10 µm process)
- Programmable array logic

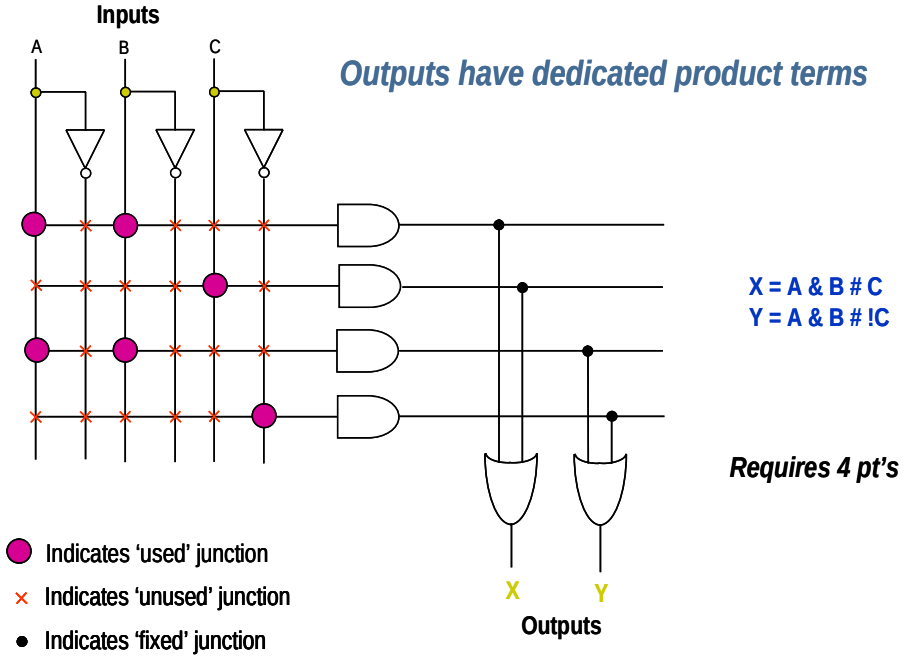


Figure 1-2: **SPLD Architectures (PAL)**

The architecture had a mesh of horizontal and vertical interconnect tracks. At each junction was a fuse. With the aid of software tools, designers could select which junctions would not be connected by "blowing" all unwanted fuses. (This was done by a device programmer, but more commonly these days is achieved with ISP).

Input pins were connected to the vertical interconnect. The horizontal tracks were connected to AND-OR gates, also called "product terms". These in turn connected to dedicated flip-flops, whose outputs were connected to output pins.

PLDs provided as much as 50 times more gates in a single package than discrete logic devices! This was a huge improvement, not to mention fewer devices needed in inventory and a higher reliability over standard logic.

PLD technology has moved on from the early days with companies such as Xilinx producing ultra-low-power CMOS devices based on flash memory technology. Flash PLDs

provide the ability to program the devices time and time again, electrically programming and erasing the device. Gone are the days of erasing for more than 20 minutes under an UV eraser.

Complex Programmable Logic Devices (CPLDs)

Complex programmable logic devices (CPLDs) extend the density of SPLDs. The concept is to have a few PLD blocks or macrocells on a single device with a general-purpose interconnect in-between. Simple logic paths can be implemented within a single block. More sophisticated logic requires multiple blocks and uses the general-purpose interconnect in-between to make these connections. CPLDs feature:

- Central global interconnect
- Simple, deterministic timing
- Easily routed
- PLD tools add only interconnect
- Wide, fast complex gating

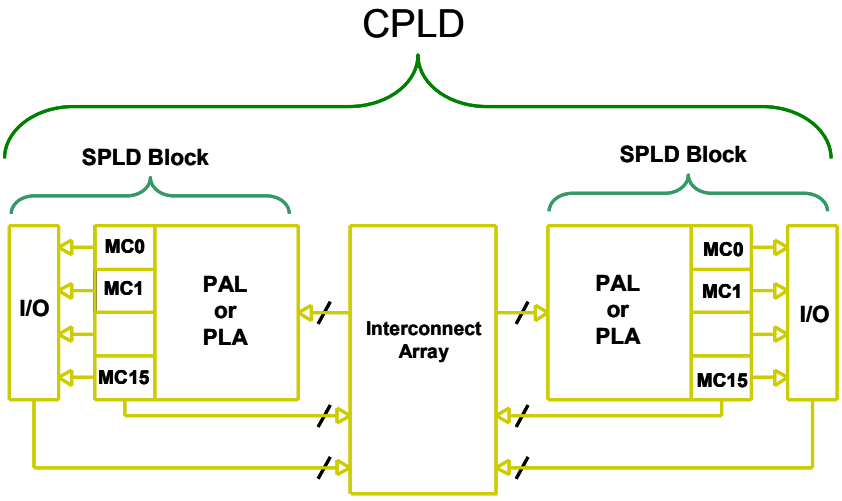


Figure 1-3: **CPLD Architecture**

CPLDs are great at handling wide and complex gating at blistering speeds – 5 nanoseconds, for example, which is equivalent to 200 MHz. The timing model for CPLDs is easy to calculate so before starting your design you can calculate your input-to-output speeds.

Why Use a CPLD?

CPLDs enable ease of design, lower development costs, more product revenue for your money, and the opportunity to speed your products to market.

- **Ease of Design:** CPLDs offer the simplest way to implement a design. Once a design has been described, by schematic and/or HDL entry, you simply use CPLD development tools to optimize, fit, and simulate the design. The development tools create a file that is used to customize (that is, program) a standard off-the-shelf CPLD with the desired functionality. This provides an instant hardware prototype and allows the debugging process to begin. If modifications are needed, you can enter

design changes into the CPLD development tool, and re-implement and test the design immediately.

- **Lower Development Costs:** CPLDs offer very low development costs. Because CPLDs are re-programmable, you can easily and very inexpensively change your designs. This allows you to optimize your designs and continue to add new features to enhance your products. CPLD development tools are relatively inexpensive (or in the case of Xilinx, free). Traditionally, designers have had to face large cost penalties such as re-work, scrap, and development time. With CPLDs, you have flexible solutions, thus avoiding many traditional design pitfalls.
- **More Product Revenue:** CPLDs offer very short development cycles, which means your products get to market quicker and begin generating revenue sooner. Because CPLDs are re-programmable, products can be easily modified using ISP over the Internet. This in turn allows you to easily introduce additional features and quickly generate new revenue. (This also results in an expanded time for revenue). Thousands of designers are already using CPLDs to get to market quicker and stay in the market longer by continuing to enhance their products even after they have been introduced into the field. CPLDs decrease TTM and extend TIM.
- **Reduced Board Area:** CPLDs offer a high level of integration (that is, a large number of system gates per area) and are available in very small form factor packages. This provides the perfect solution for designers whose products which must fit into small enclosures or who have a limited amount of circuit board space to implement the logic design. Xilinx CoolRunner® CPLDs are available in the latest chip scale packages. For example, the CP56 CPLD has a pin pitch of 0.5 mm and is a mere 6 x 6 mm in size, making it ideal for small, low-power end products. The CoolRunner-II CPLDs are also available in the QF (quad flat no-lead) packages, giving them the smallest form factor available in the industry. The QF32 is just 5 x 5 mm in size.

Figure 1-4: **Small Form Factor Packages**

- **Cost of Ownership:** Cost of Ownership can be defined as the amount it costs to maintain, fix, or warranty a product. For instance, if a design change requiring hardware rework must be made to a few prototypes, the cost might be relatively small. However, as the number of units that must be changed increases, the cost can become enormous. Because CPLDs are re-programmable, requiring no hardware rework, it costs much less to make changes to designs implemented using them. Therefore cost of ownership is dramatically reduced.

  Don't forget that the ease or difficulty of design changes can also affect opportunity costs. Engineers who spend time fixing old designs could be working on introducing new products and features ahead of the competition.

  There are also costs associated with inventory and reliability. PLDs can reduce inventory costs by replacing standard discrete logic devices. Standard logic has a predefined function. In a typical design, lots of different types have to be purchased and stocked. If the design is changed, there may be excess stock of superfluous devices. This issue can be alleviated by using PLDs. You only need to stock one device;

if your design changes, you simply reprogram. By utilizing one device instead of many, your board reliability will increase by only picking and placing one device instead of many.

- **Reliability:** Reliability can also be increased by using ultra-low-power CoolRunner CPLDs. Their lower heat dissipation and lower power operation leads to decreased FIT.

Field Programmable Gate Arrays (FPGAs)

In 1985, Xilinx introduced a completely new idea: combine the user control and time to market of PLDs with the densities and cost benefits of gate arrays. Customers liked it, and the FPGA was born. Today Xilinx is the number one FPGA vendor in the world.

An FPGA is a regular structure of logic cells (or modules) and interconnect, which is under your complete control. This means that you can design, program, and make changes to your circuit whenever you wish.

FPGAs feature:

- Channel based routing
- Post layout timing
- Tools more complex than CPLDs
- Fine grained
- Fast register pipelining

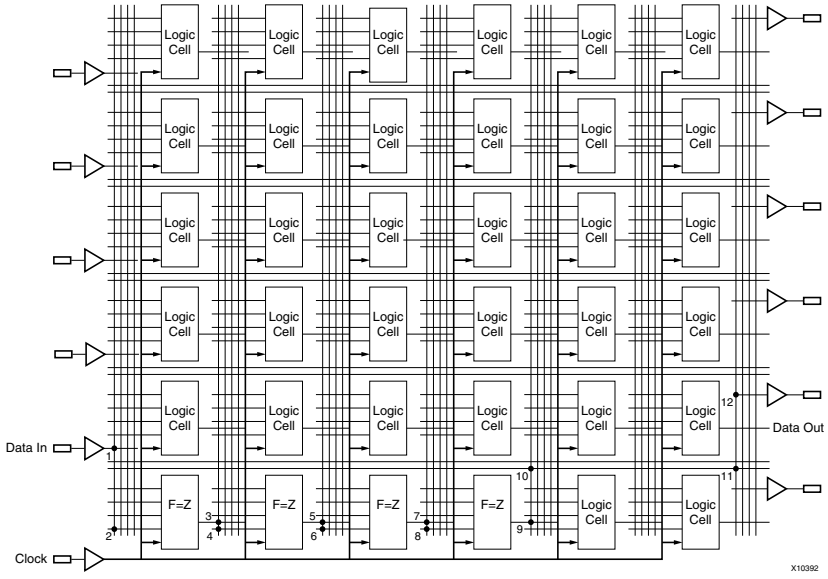


Figure 1-5: **FPGA Architecture**

With the introduction of the Spartan® series of FPGAs, Xilinx can now compete with gate arrays on all aspects – price, gate, and I/O count, as well as performance and cost.

There are two basic types of FPGAs: SRAM-based reprogrammable and OTP (One Time Programmable). These two types of FPGAs differ in the implementation of the logic cell and the mechanism used to make connections in the device.

The dominant type of FPGA is SRAM-based and can be reprogrammed as often as you choose. In fact, an SRAM FPGA is reprogrammed every time it's powered up, because the FPGA is really a fancy memory chip. That's why you need a serial PROM or system memory with every SRAM FPGA.

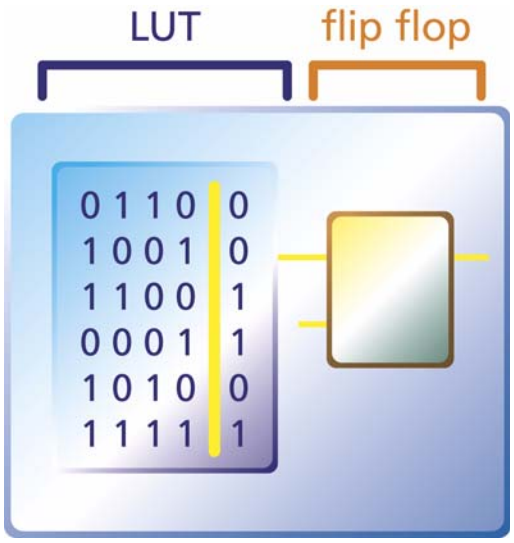


Figure 1-6: **SRAM Logic Cell**

In the SRAM logic cell, instead of conventional gates, an LUT determines the output based on the values of the inputs. (In the "SRAM logic cell" diagram above, six different combinations of the four inputs determine the values of the output.) SRAM bits are also used to make connections.

OTP FPGAs use anti-fuses (contrary to fuses, connections are made, not "blown," during programming) to make permanent connections in the chip. Thus, OTP FPGAs do not require SPROM or other means to download the program to the FPGA. However, every time you make a design change, you must throw away the chip! The OTP logic cell is very similar to PLDs, with dedicated gates and flip-flops.

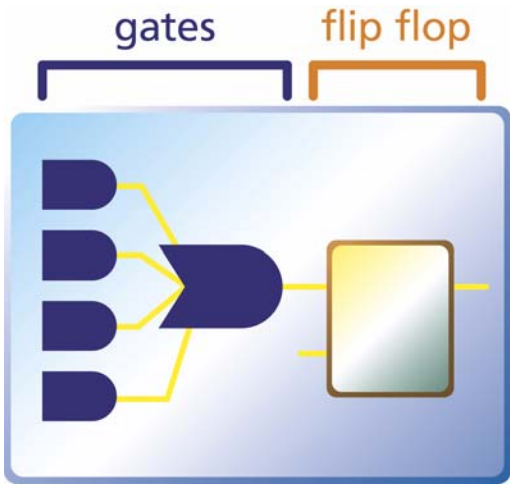


Figure 1-7: **OTP Logic Cell**

Logic Consolidation

The consolidation of 74 series standard logic into a low-cost CPLD is a very attractive proposition. Not only do you save PCB area and board layers – thus reducing your total system cost – but you only have to purchase and stock one generic part instead of 20 or more pre-defined logic devices. In production, the pick and place machine only has to place one part, therefore speeding up production. Less parts means higher quality and better FIT factor.

By using Xilinx CoolRunner devices, you can benefit from low power consumption and reduced thermal emissions. This in turn leads to the reduction of the use of heat sinks (another cost savings) and a higher reliability end product.

XILINX®

Chapter 2

Xilinx Silicon Solutions

Introduction

Xilinx programmable logic solutions help minimize risks for electronic equipment manufacturers by shortening the time required to develop products and take them to market. You can design and verify the unique circuits in Xilinx programmable devices much faster than by choosing traditional methods such as mask-programmed, fixed logic gate arrays. Moreover, because Xilinx devices are standard parts that need only to be programmed, you are not required to wait for prototypes or pay large non-recurring engineering (NRE) costs.

Customers incorporate Xilinx programmable logic into products for a wide range of markets. Those include data processing, telecommunications, networking, industrial control, instrumentation, consumer electronics, automotive, defense, and aerospace markets.

Leading-edge silicon products, state-of-the-art software solutions, and world-class technical support make up the total solution that Xilinx delivers. The software component of this solution is critical to the success of every design project. Xilinx Software Solutions provide powerful tools that make designing with programmable logic simple. Push-button design flows, integrated online help, multimedia tutorials, and high-performance automatic and auto-interactive tools help you achieve optimum results. In addition, the industry's broadest array of programmable logic technology and EDA integration options deliver unparalleled design flexibility.

Xilinx is also actively developing breakthrough technology that will enable the hardware in Xilinx-based systems to be upgraded remotely over any kind of network – including the Internet – even after the equipment has been shipped to a customer. Xilinx “Online Upgradeable Systems” would allow equipment manufacturers to remotely add new features and capabilities to installed systems, or repair problems without having to physically exchange hardware.

Xilinx Devices

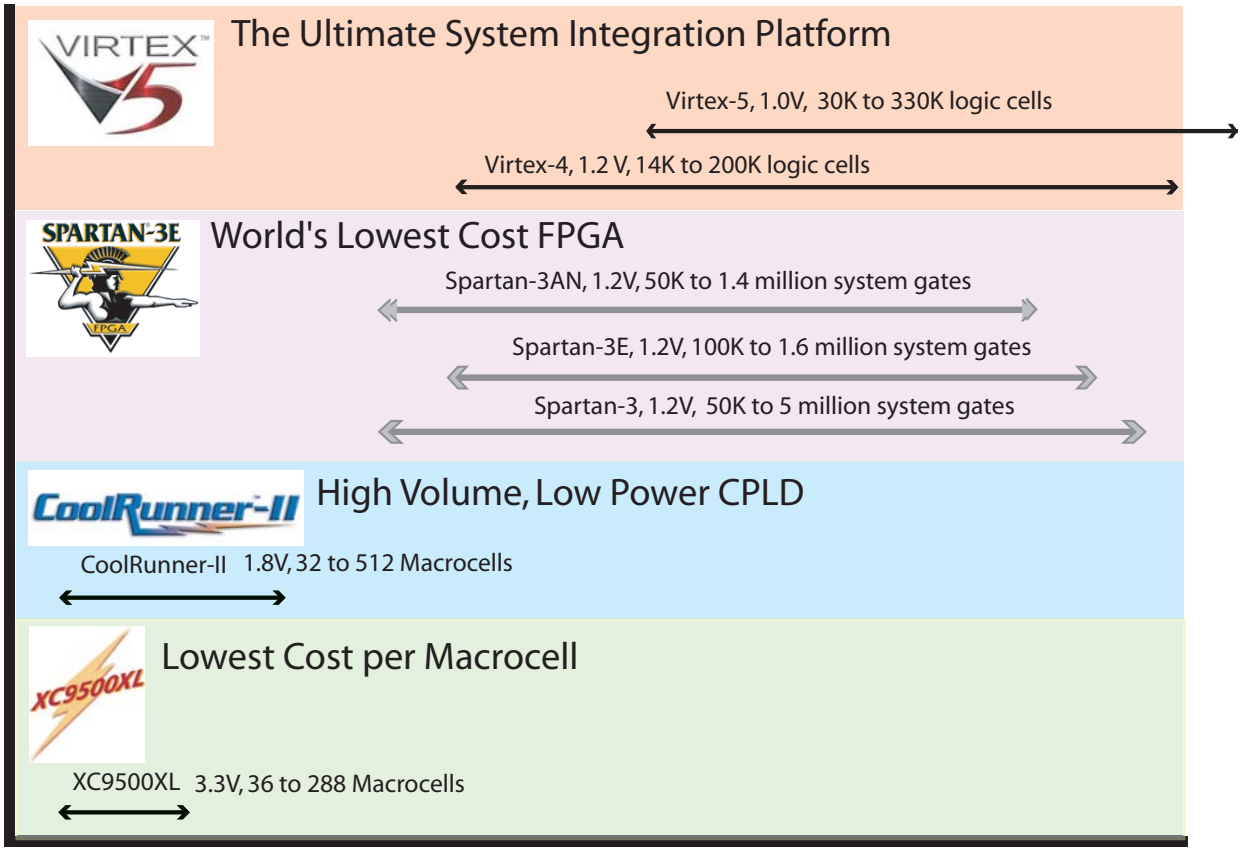


Figure 2-1: **Xilinx Devices at a Glance**

Xilinx CPLDs

Currently, Xilinx offers CPLD products in two categories: XC9500 and CoolRunner devices. This guide will focus on the two most popular families, the XC9500XL and the CoolRunner-II. To choose a CPLD that's right for you, review the product features below to identify the product family that fits your application. You should also review the selection considerations to choose the device that best meets your design criteria.

Product Features

XC9500XL Device – The XC9500XL ISP CPLD family takes complex programmable logic devices to new heights of performance, features, and flexibility. This family delivers industry-leading speeds while providing the flexibility of enhanced customer-proven pin-locking architecture, along with extensive IEEE Std.1149.1 JTAG Boundary Scan support. This CPLD family is ideal for high-speed, low-cost designs.

CoolRunner-II Device – The CoolRunnerII CPLD family offers extremely low power, making them the leaders in an all-new market segment: portable electronics. With standby current in the low micro amps and minimal operational power consumption, these parts are ideal for any application is that is especially power sensitive, such as battery-powered or portable applications. The CoolRunner-II CPLD extends usage as it offers system-level features such as LVTTL and SSTL, clocking modes, and input hysteresis.

Selection Considerations

To decide which device best meets your design criteria, take a minute to jot down your design specs (using the list below as a criteria reference). Next, go to a specific product family page to get more detailed information about the device you need.

Density – Each part gives an equivalent "gate count," or estimate of the logic density of the part.

Number of Registers – Count up the number of registers you need for your counters, state machines, registers, and latches. The number of macrocells in the device must be at least this large.

Number of I/O Pins – How many inputs and outputs does your design need?

Speed Requirements – What is the fastest combinatorial path in your design? This will determine the Tpd (in nanoseconds) of the device. What is the fastest sequential circuit in your design? This will tell you what fMax you need.

Package – What electromechanical constraints are you under? Do you need the smallest ball grid array package possible, or can you use a more ordinary QFP? Or are you prototyping and need to use a socketed device, such as a PLCC package?

Low Power – Is your end product battery- or solar-powered? Does your design require the lowest power devices possible? Do you have heat dissipation concerns?

System-Level Functions – Does your board have multi-voltage devices? Do you need to level shift between these devices? Do you need to square up clock edges? Do you need to interface to memories and microprocessors?

CoolRunner-II Low-Power CPLDs

CoolRunner -II CPLDs combine very low power with high speed, high density, and high I/O counts in a single device. The CoolRunner-II family ranges in density from 32 to 512 macrocells. CoolRunner -II CPLDs feature RealDigital technology, allowing the devices to draw virtually no power in standby mode. This makes them ideal for the fast-growing market of battery-operated portable electronic equipment, such as:

- Laptop PCs
- Telephone handsets
- Personal digital assistants
- Electronic games
- Web tablets

These CPLDs also use far less dynamic power during actual operation compared to conventional CPLDs, an important feature for high-performance, heat- sensitive equipment such as telecom switches, video conferencing systems, simulators, high-end testers, and emulators.

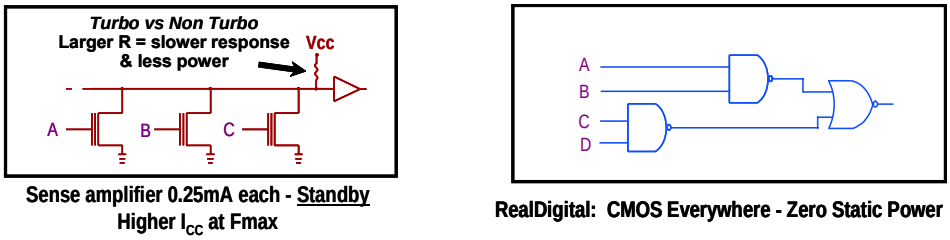


Figure 2-2: **Sense Amplifier vs. CMOS CPLDs**

The CoolRunner-II family of CPLDs is targeted for low-power applications that include portable, handheld, and power-sensitive applications. Each member of the family includes RealDigital design technology that combines low power *and* high speed. With this design technique, the family offers true pin-to-pin speeds of 5.0 ns, while simultaneously delivering power that is less than 16 µA (standby) without the need for special "power down bits" that can negatively affect device performance. By replacing conventional amplifier methods for implementing product terms (a technique that has been used in PLDs since the bipolar era) with a cascaded chain of pure CMOS gates, the dynamic power is also substantially lower than any competing CPLD. CoolRunner-II devices are the only total CMOS PLDs

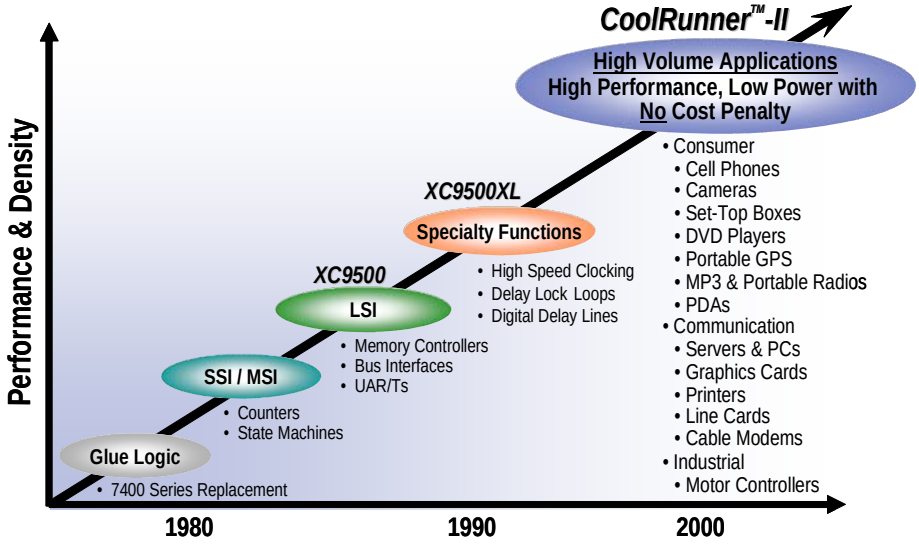


Figure 2-3: **CPLD Application Trends**

Xilinx CoolRunner-II CPLDs deliver the high speed and ease of use associated with the XC9500/XL/XV CPLD family and the extremely low power versatility of the XPLA3. This means that the exact same parts can be used for high-speed data communications, computing systems, and leading-edge portable products, with the added benefit of ISP. Low power consumption and high-speed operation are combined into a single family that is easy to use and cost effective. Xilinx-patented Fast Zero Power architecture inherently delivers extremely low power performance without the need for special design measures.

Clocking techniques and other power-saving features extend your power budget. These design features are supported from Xilinx ISE 4.1i software onwards. Figure 2-4 shows some of the advanced CoolRunner-II CPLD package offering with dimensions. All

packages are surface mount, with more than half of them ball-grid technologies. The ultra-tiny packages permit maximum functional capacity in the smallest possible area.

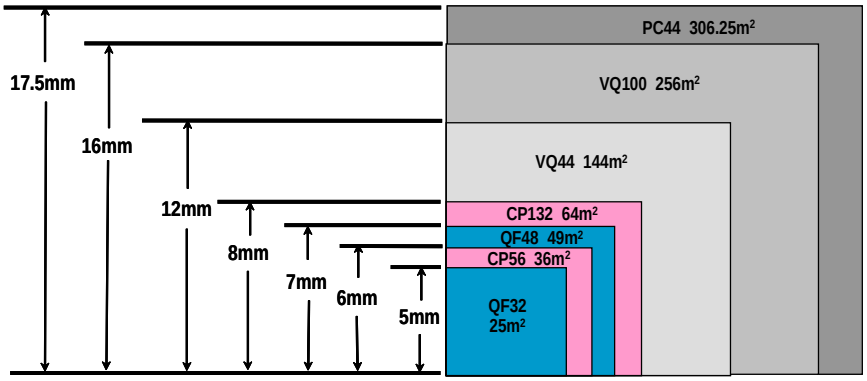


Figure 2-4: **CPLD Packages**

The CMOS technology used in CoolRunner-II CPLDs generates minimal heat, allowing the use of tiny packages during high-speed operation. At least two densities are present in each package, with three in the VQ100 (100-pin, 1.0 mm QFP) and TQ144 (144-pin, 1.4 mm QFP), and in the FT256 (256- ball, 1.0 mm-spacing FLBGA). The FT256 is particularly suited for slim-dimensioned portable products with mid to high-density logic requirements.

Table 2-1 also details the distribution of advanced features across the CoolRunner-II CPLD family. The family has uniform basic features, with advanced features included in densities where they are most useful. For example, it is unlikely that you would need four I/O banks on 32- and 64-macrocell parts, but very likely for 384- and 512-macrocell parts.

The I/O banks are groupings of I/O pins using any one of a subset of compatible voltage standards that share the same V CCIO level. The clock division capability is less efficient on small parts, but more useful and likely to be used on larger ones. DataGATE™ technology, an ability to block and latch inputs to save power, is valuable in larger parts, but brings marginal benefit to small parts.

Table 2-1: **CoolRunner-II Family Overview**

Features	XC2C32A	XC2C64A	XC2C128	XC2C256	XC2C384	XC2C512
F_{SYSTEM} (MHz)	323	263	244	256	217	179
Max User I/O	33	64	100	184	240	270
I/O Banks	2	2	2	2	4	4
LVCMOS, LVTTL (1.5,1.8,2.5,3.3)	Yes	Yes	Yes	Yes	Yes	Yes
HSTL, SSTL	-	-	Yes	Yes	Yes	Yes
DualEDGE	Yes	Yes	Yes	Yes	Yes	Yes
DataGATE, CoolCLOCK	-	-	Yes	Yes	Yes	Yes
Standby Power (µW)	28.8	30.6	34.2	37.8	41.4	45.0
Advanced Security	Yes	Yes	Yes	Yes	Yes	Yes
Packages (size)	**Maxium User I/O**					
QFG32 (5x5 mm)	21					
VQ44 (12x12 mm)	33	33				
PC44 (17.5x17.5 mm)	33	33				
QF48 (7x7 mm)		37				
CP56 (6x6 mm)	33	45				
VQ100 (16x16 mm)		64	80	80		
CP132 (8x8 mm)			100	106		
TQ144 (22x22 mm)			100	118	118	
PQ208 (30.6x30.6 mm)				173	173	173
FT256 (17x17 mm)				184	212	212
FG324 (23x23 mm)					240	270

CoolRunner-II Architecture Description

The CoolRunner-II CPLD is a highly uniform family of fast, low-power devices. The underlying architecture is a traditional CPLD architecture, combining macrocells into function blocks interconnected with a global routing matrix, the Xilinx Advanced Interconnect Matrix (AIM). The function blocks use a PLA configuration that allows all product terms to be routed and shared among any of the macrocells of the function block.

Design software can efficiently synthesize and optimize logic that is subsequently fit to the function blocks and connected with the ability to utilize a very high percentage of device resources. The software easily and automatically manages design changes, exploiting the 100% routeability of the PLA within each function block. This extremely robust building block delivers the industry's highest pin-out retention under very broad design conditions. The design software automatically manages device resources so that you can express your designs using completely generic constructs, without needing to know the architectural details. If you're more experienced, you can take advantage of these details to more thoroughly understand the software's choices and direct its results.

Figure 2-5 shows the high-level architecture whereby function blocks attach to pins and interconnect to each other within the internal interconnect matrix. Each function block contains 16 macrocells.

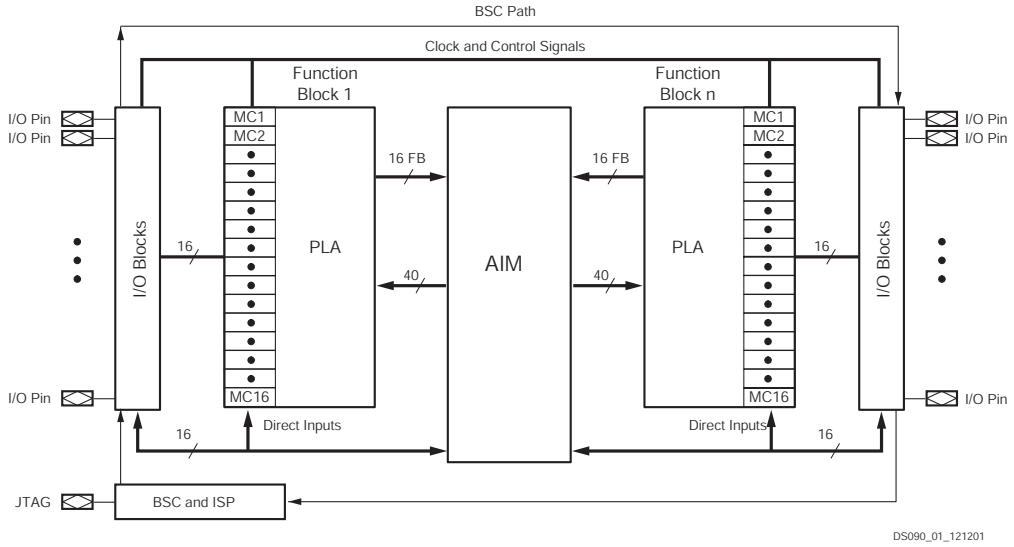


Figure 2-5: **CoolRunner-II High-Level Architecture**

CoolRunner-II Function Block

The CoolRunner-II CPLD function blocks contain 16 macrocells, with 40 entry sites for signals to arrive for logic creation and connection. The internal logic engine is a 56-product term PLA. All function blocks, regardless of the number contained in the device, are identical. At the high level, the p-terms reside in a PLA. This structure is extremely flexible and very robust when compared to fixed or cascaded p-term function blocks. Classic CPLDs typically have a few p-terms available for a high-speed path to a given macrocell. They rely on capturing unused p-terms from neighboring macrocells to expand their product term tally when needed. The result of this architecture is a variable timing model and the possibility of stranding unusable logic within the function block.

The PLA is different – and better. First, any p-term can be attached to any OR gate inside the function block macrocell(s). Second, any logic function can have as many p-terms as needed attached to it within the function block, to an upper limit of 56. Third, you can reuse product terms at multiple macrocell OR functions so that within a function block, you need only create a particular logical product once, but you can reuse it as many as 16 times within the function block. Naturally, this works well with the fitting software, which identifies product terms that can be shared.

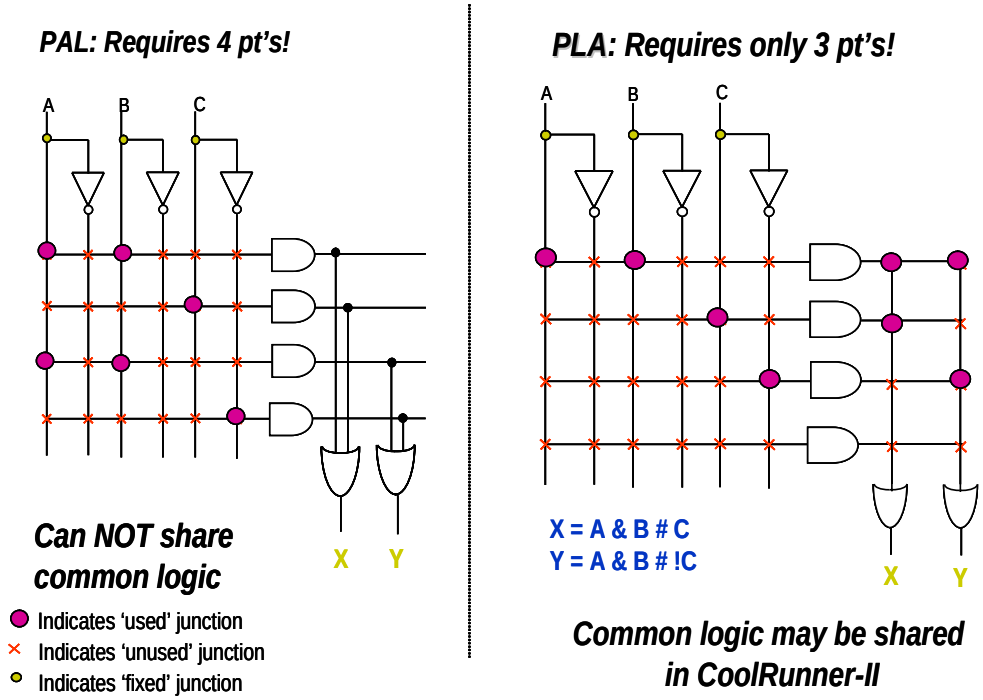


Figure 2-6: **Logic Allocation – Typical PAL vs. PLA**

The software places as many functions as it can into function blocks. There is no need to force macrocell functions to be adjacent or have any other restriction except for residing in the same function block, which is handled by the software. Functions need not share a common clock, common set/reset, or common output enable to take full advantage of the PLA. In addition, every p-term arrives with the same time delay incurred. There are no cascade time adders for putting more product terms in the function block. When the function block p-term budget is reached, a small interconnect timing penalty routes signals to another function block to continue creating logic. Xilinx design software handles all this automatically.

CoolRunner-II Macrocell

The CoolRunner-II CPLD macrocell is extremely efficient and streamlined for logic creation. You can develop SOP logic expressions comprising as many as 40 inputs and span 56 product terms within a single function block. The macrocell can further combine the SOP expression into an XOR gate with another single p-term expression. The resulting logic expression's polarity is also selectable. The logic function can be pure combinatorial or registered, with the storage element operating selectively as a D or T flip-flop, or transparent latch. Available at each macrocell are independent selections of global, function- block level, or local p-term-derived clocks, sets, resets, and output enables. Each macrocell flip-flop is configurable for either single edge or DualEDGE clocking, providing

either double data rate capability or the ability to distribute a slower clock (thereby saving power). For single-edge clocking or latching, either clock polarity may be selected per macrocell.

CoolRunner-II macrocell details are shown in Figure 2-7. Standard logic symbols are used in the in figure, except the trapezoidal multiplexers have input selection from statically programmed configuration select lines (not shown). Xilinx application note XAPP376 gives a detailed explanation of how logic is created in the CoolRunner-II CPLD family.

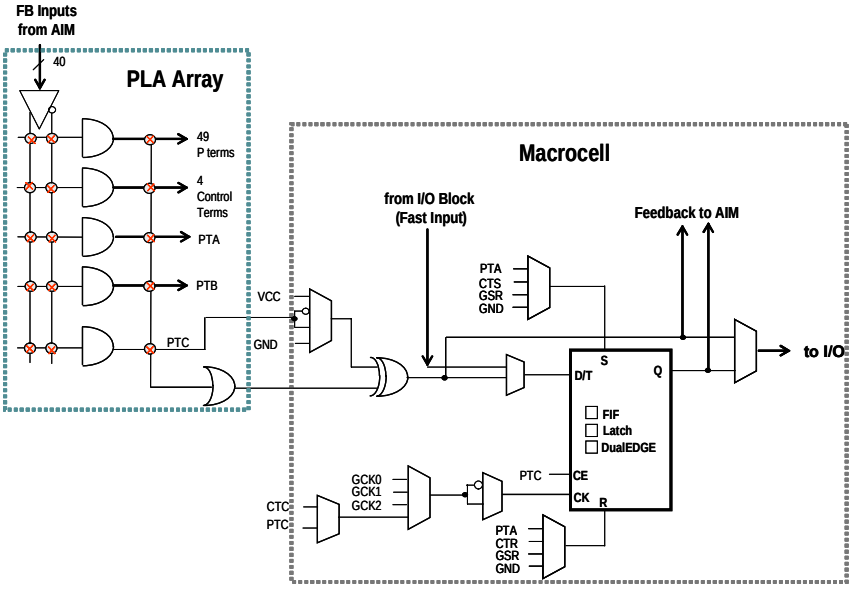


Figure 2-7: **CoolRunner-II Macrocell Architecture**

When configured as a D-type flip-flop, each macrocell has an optional clock enable signal permitting state hold while a clock runs freely. Note that control terms are available to be shared for key functions within the function block, and are generally used whenever the exact same logic function would be repeatedly created at multiple macrocells. The control term product terms are available for function block clocking (CTC), function block asynchronous set (CTS), function block asynchronous reset (CTR), and function block output enable (CTE).

You can configure any macrocell flip-flop as an input register or latch, which takes in the signal from the macrocell's I/O pin and directly drives the AIM. The macrocell combinatorial functionality is retained for use as a buried logic node if needed.

Advanced Interconnect Matrix (AIM)

AIM is a highly connected low-power rapid switch directed by the software to deliver a set of as many as 40 signals to each function block for the creation of logic. Results from all function block macrocells, as well as all pin inputs, circulate back through the AIM for additional connection available to all other function blocks, as dictated by the design software. The AIM minimizes both propagation delay and power as it makes attachments to the various function blocks.

I/O Blocks

I/O blocks are primarily transceivers. However, each I/O is either automatically compliant with standard voltage ranges or can be programmed to become compliant. In addition to voltage levels, each input can selectively arrive through Schmitt-trigger inputs. This adds a small time delay, but substantially reduces noise on that input pin. Hysteresis also allows easy generation of external clock circuits. The Schmitt-trigger path is best illustrated in Figure 2-8. Outputs can be directly driven, tri-stated, or open-drain configured. A choice of slow or fast slew rate output signal is also available.

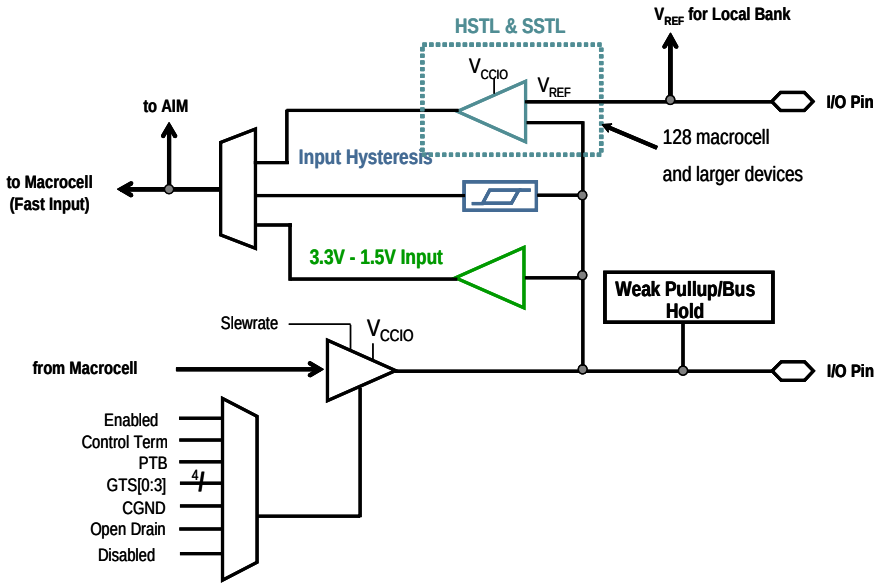


Figure 2-8: **CoolRunner-II I/O Block**

I/O Banking

CPLDs are widely used as voltage interface translators; thus, the I/O pins are grouped in large banks. The four smaller parts have two output banks. With two banks available, the outputs will switch to one of two selected output voltage levels, unless both banks are set to the same voltage. The larger parts (384 and 512 macrocell) support four output banks, split evenly. They can support groupings of one, two, three, or four separate output voltage levels. This kind of flexibility permits easy interfacing to 3.3V, 2.5V, 1.8V, and 1.5V in a single part.

DataGATE

Low power is the hallmark of CMOS technology. Other CPLD families use a sense amplifier approach to create p-terms, which always has a residual current component. This residual current can be several hundred milliamps, making these CPLDs unusable in portable systems. CoolRunner-II CPLDs use standard CMOS methods to create the CPLD architecture and deliver the corresponding low current consumption, without any special tricks.

However, sometimes you might want to reduce the system current even more by selectively disabling unused circuitry. The patented DataGATE technology permits a straightforward approach to additional power reduction. Each I/O pin has a series switch that can block the arrival of unused free- running signals that may increase power

consumption. Disabling these switches enables you to complete your design and choose which sections will participate in the DataGATE function.

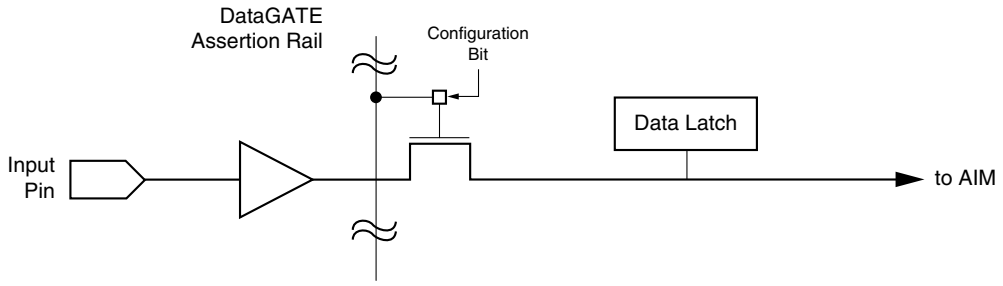


Figure 2-9: **DataGATE Function in CoolRunner-II CPLDs**

- Available on all input pins (except JTAG pins)
- Available for all I/O types
- Selectable on a per pin basis
- Data latch holds last valid pin value
- DataGATE allows additional power savings
  - ability to disable active board inputs
- DataGATE can also be used for debugging and hot plug input

The DataGATE logic function drives an assertion rail threaded through medium- and high-density CoolRunner-II CPLD parts. You can select which inputs to block under the control of the DataGATE function, effectively blocking controlled switching signals so that they do not drive internal chip capacitances. Output signals that do not switch are held by the bus hold feature. You can choose any set of input pins can be chosen to participate in the DataGATE function.

Figure 2-9 shows how DataGATE function works. One I/O pin drives the DataGATE assertion rail. It can have any desired logic function on it – something as simple as mapping an input pin to the DataGATE function or as complex as a counter or state machine output driving the DataGATE I/O pin through a macrocell. When the DataGATE rail is asserted low, any pass transistor switch attached to it is blocked. Each pin has the ability to attach to the AIM through a DataGATE pass transistor, and be blocked. A latch automatically captures the state of the pin when it becomes blocked. The DataGATE assertion rail threads throughout all possible I/Os, so each can participate if chosen. One macrocell is singled out to drive the rail, and that macrocell is exposed to the outside world (through a pin) for inspection. If the DataGATE function is not needed, this pin is an ordinary I/O.

Additional Clock Options: Division, DualEDGE, and CoolCLOCK

Division

Circuitry has been included in the CoolRunner-II CPLD architecture to divide one externally supplied global clock by standard values, with options for division by 2, 4, 6, 8, 10, 12, 14, and 16 (see Figure 2-10). This capability is supplied on the GCK2 pin. The resulting clock produced will be 50% duty cycle for all possible divisions. Note that a synchronous reset is included to guarantee that no runt clocks can get through to the global

clock nets. The signal is buffered and driven to multiple traces with minimal loading and skew.

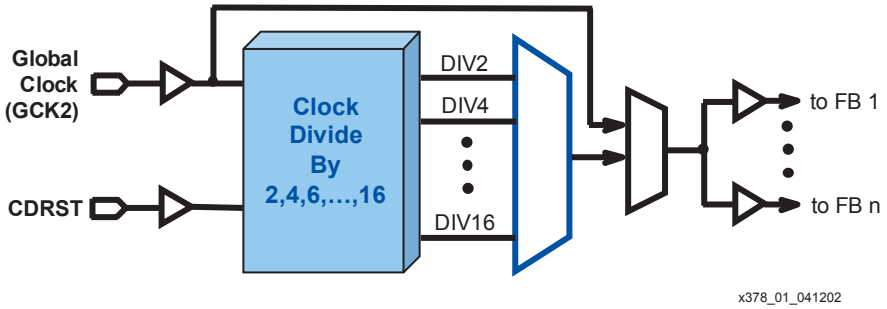


Figure 2-10: **CoolRunner-II Clock Division**

DualEDGE

Each macrocell has the ability to double its input clock switching frequency. Figure 2-7 shows the macrocell flip-flop with the DualEDGE option (doubled clock) at each macrocell. The source to double can be a control term clock, a product term clock, or one of the available global clocks. The ability to switch on both clock edges is vital for a number of synchronous memory interface applications as well as certain double data rate I/O applications.

CoolCLOCK

In addition to the DualEDGE flip-flop, you can gain additional power savings by combining the clock division circuitry with the DualEDGE circuitry. This capability is called CoolCLOCK and is designed to reduce clocking power within the CPLD. Because the clock net can be a significant power drain, you can reduce the clock power by driving the net at half frequency, and then doubling the clock rate using DualEDGE triggering at the macrocells. Figure 2-11 illustrates how CoolCLOCK is created by internal clock cascading, with the divider and DualEDGE flip-flop working together.

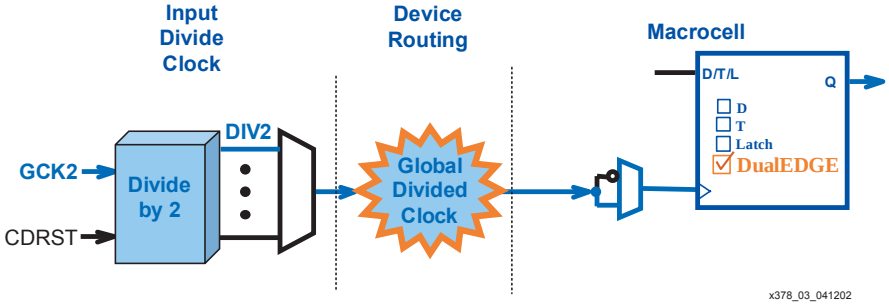


Figure 2-11: **CoolCLOCK**

Design Security

You can secure your designs during programming to prevent either accidental overwriting or pattern theft via readback. CoolRunner-II CPLDs have four independent levels of security provided on-chip, eliminating any electrical or visual detection of configuration patterns. These security bits can be reset only by erasing the entire device. Additional details are omitted intentionally.

- Four independent levels of security
  - Hidden and scattered
  - Affect different modes
  - Interconnects are buried
  - Multiple programming bits

XC9500XL CPLD Overview

The high-performance, low-cost XC9500XL family of Xilinx CPLDs are targeted for leading-edge systems that require rapid design development, longer system life, and robust field upgrade capability. The 3.3V XC9500XL family ranges in density from 36 to 288 macrocells.

These devices are In-System Programmable (ISP), which allows manufacturers to perform unlimited design iterations during the prototyping phase, extensive system in-board debugging, program and test during manufacturing, and field upgrades.

Based on advanced process technologies, the XC9500XL CPLD provides fast, guaranteed timing; superior pin locking; and a full JTAG-compliant interface. All XC9500XL devices have excellent quality and reliability characteristics with a 10,000 program/erase cycle endurance rating and 20-year data retention.

Flexible Pin-Locking Architecture

XC9500XL devices, in conjunction with our fitter software, give you the maximum in routeability and flexibility while maintaining high performance. The architecture is feature-rich, including individual product term (p-term) output enables, three global clocks, and more p-terms per output than any other CPLD. The proven ability of the architecture to adapt to design changes while maintaining pin assignments has been demonstrated in countless real-world customer designs.

Full IEEE 1149.1 JTAG Development and Debugging Support

The JTAG capability of the XC9500XL CPLD is the most comprehensive of any CPLD on the market. It features the standard support including BYPASS, SAMPLE/PRELOAD, and EXTEST. Additional Boundary Scan instructions, not found in any other CPLD, include INTEST (for device functional test), HIGHZ (for bypass), and USERCODE (for program tracking), for maximum debugging capability.

The XC9500XL family is supported by a wide variety of industry-standard third-party development and debugging tools including Corelis, JTAG Technologies, and Asset Intertech. These tools allow you to develop Boundary Scan test vectors to interactively analyze, test, and debug system failures. The family is also supported on all major ATE platforms, including Teradyne, Hewlett Packard, and Genrad.

Table 2-2: **XC9500XL Product Overview**

	XC9536XL	XC9572XL	XC95144XL	XC95288XL
Macrocells	36	72	144	288
Usable Gates	800	1,600	3,200	6,400
Registers	36	72	144	288
T_{PD} (ns)	5	5	5	6

	XC9536XL	XC9572XL	XC95144XL	XC95288XL
T_{SU} (ns)	3.7	3.7	3.7	4.0
T_{CO} (ns)	3.5	3.5	3.5	3.8
f_{SYSTEM} (MHz)	178	178	178	208

XC9500XL CPLDs also complement the higher-density Xilinx FPGAs to provide a total logic solution, within a unified development environment. The XC9500XL family is fully WebPOWERED via its free WebPACK ISE software.

Family Highlights

- Lowest cost per macrocell
- State-of-the-art pin-locking architecture
- Highest programming reliability reduces system risk
- Complements Xilinx 3.3V FPGA families
- Performance
  - 5 ns pin-to-pin speed
  - 222 MHz system frequency
- Powerful Architecture
  - Wide 54-input function blocks
  - As many as 90 product-terms per macrocell
  - Fast and routable Fast CONNECT™ II switch matrix
  - Three global clocks with local inversion
  - Individual OE per output, with local inversion

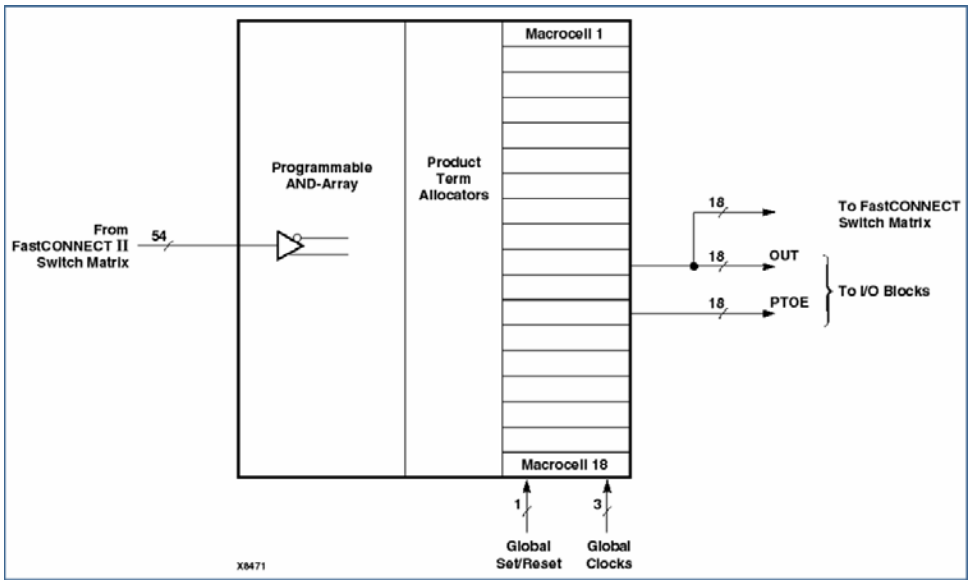


Figure 2-12: **XC9500XL Block Fan-In**

- Highest Reliability
  - Endurance rating of 10,000 cycles

- Data retention rating of 20 years
- Immune from "ISP Lock-Out" failure mode
- Allows arbitrary mixed-power sequencing and waveforms

- Advanced Technology
  - Third-generation, proven CPLD technology
  - Mainstream, scalable, high-reliability processing
  - Fast ISP and erase times
  - Outperforms All Other 3.3V CPLDs
  - Extended data retention supports longer system operating life
  - Virtually eliminates ISP failures
  - Superior pin-locking for lower design risk
  - Glitch-free I/O pins during power-up
- Full IEEE 1149.1 (JTAG) ISP and Boundary Scan testing
- Free WebPOWERED software

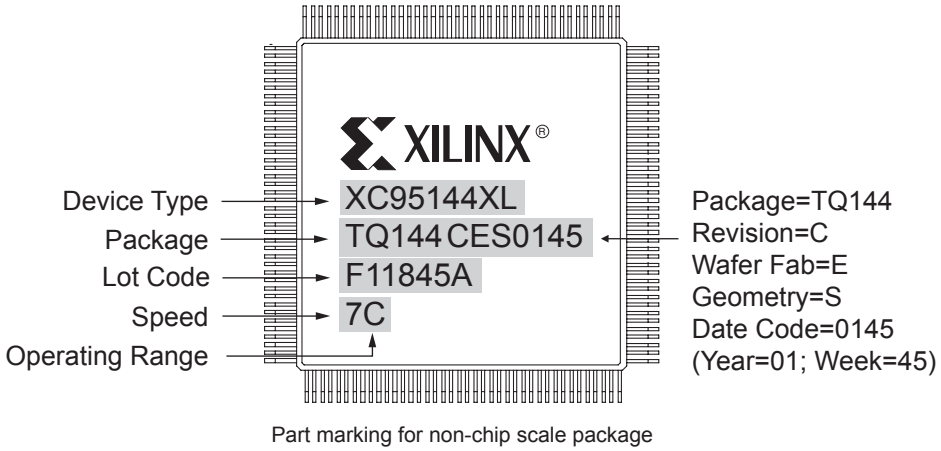


Figure 2-13: **XC9500XL Part Numbering System**

Platform FPGAs

Spartan-3/3E/3A/3AN FPGAs

Xilinx Spartan-3 FPGAs are ideal for low-cost, high-volume applications and are targeted as replacements for fixed-logic gate arrays and ASSP products such as bus interface chip sets. The Spartan-3 (1.2V, 90 nm) FPGA is not only available for a very low cost, but it integrates many architectural features associated with high-end programmable logic. This combination of low cost and features makes it an ideal replacement for ASICs (gate arrays) and many ASSP devices. For example, a Spartan-3 FPGA in a car multimedia system could

absorb many system functions, including embedded IP cores, custom system interfaces, DSP, and logic. Figure 2-14 below shows such a system.

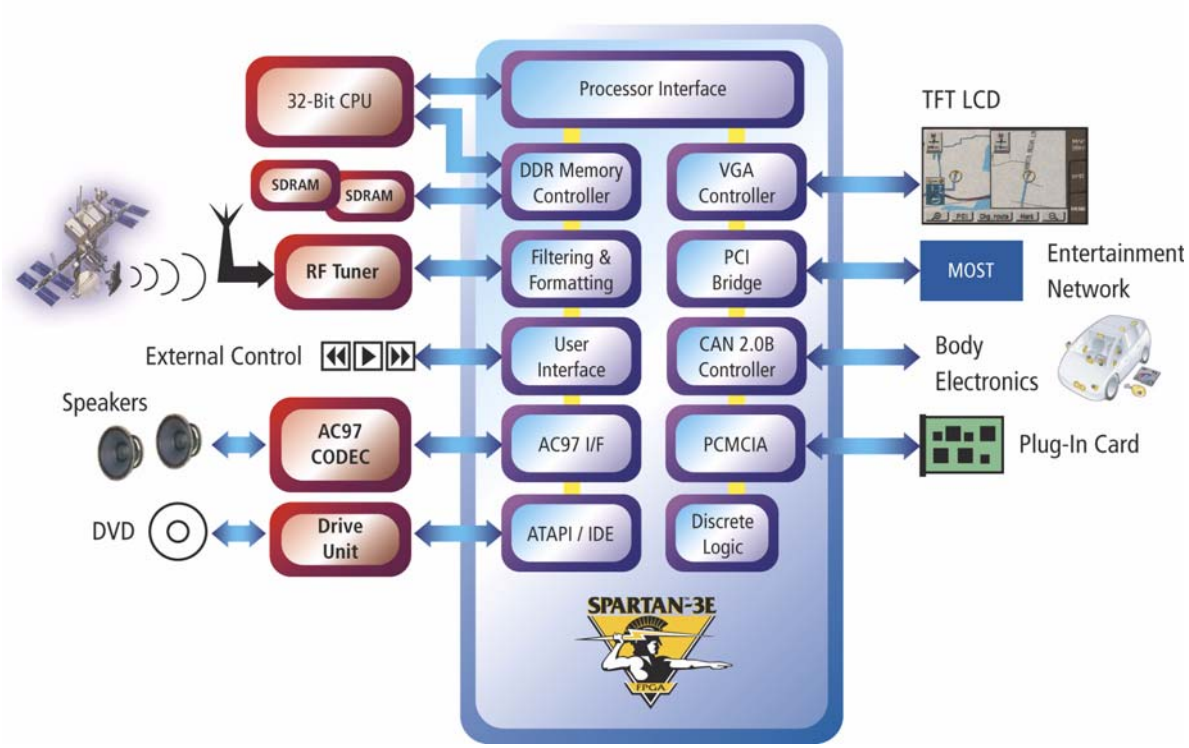


Figure 2-14: **Car Multimedia System**

In the car multimedia system shown in the above figure, the PCI bridge takes the form of a pre-verified drop in IP core, and the device-level and board-level clocking functions are implemented in the Spartan-3 on-chip DCMs. CAN core IP can connect to the body electronics modules. These cores are provided by Xilinx AllianceCORE™ partners such as Bosch, Memec Design, CAST, Inc., Xylon, and Intelliga. On-chip 18 x 18 multipliers can be used in DSP-type activities such as filtering and formatting. Other custom-designed interfaces can be implemented to off-chip processors, an IDE interface to the drive unit of a DVD player, audio, memory, and LCD. Additionally, the Spartan-3 XCITE digitally

controlled impedance technology can reduce EMI and component count by providing on-chip tunable impedances to provide line matching without the need for external resistors.

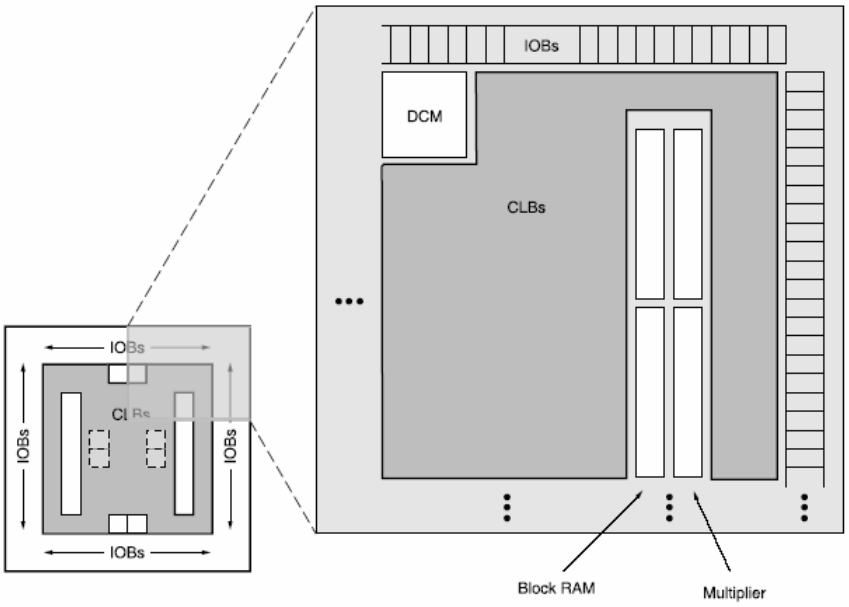


Figure 2-15: **Spartan-3 Architecture Layout**

The Spartan-3 family is based on advanced 90 nm, eight- layer metal process technology. Xilinx uses 90 nm technology to drive pricing down to under $20 for a one-million-gate FPGA (approximately 17,000 logic cells), which represents a cost savings as high as 80 percent compared to competitive offerings. A smaller die size and 300 mm wafers improve device densities and yields, thereby reducing overall production costs. This in turn leads to a more highly integrated, less expensive product that takes up less board space when designed into an end product.

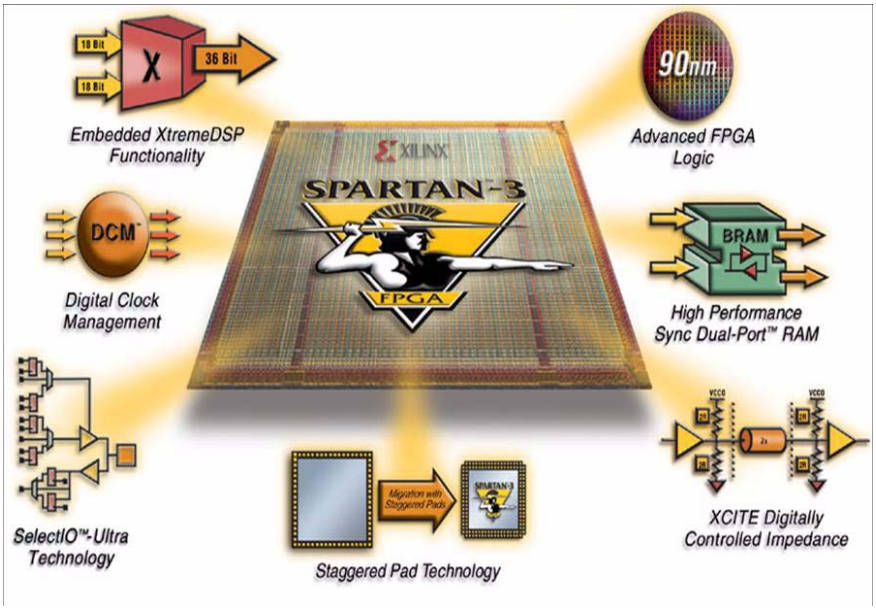


Figure 2-16: **Spartan-3 Features**

The Spartan-3 FPGA memory architecture provides the optimal granularity and efficient area utilization.

- Shift Register SRL16 Blocks
  - Each CLB LUT works as a 16-bit fast, compact shift register
  - Cascade LUTs to build longer shift registers
  - Implement pipeline registers and buffers for video or wireless

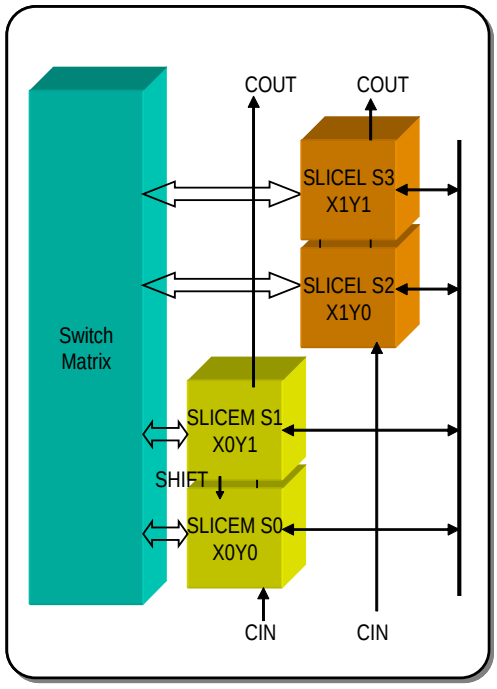


Figure 2-17: **Spartan-3 Configurable Logic Block**

- As Much as 520 Kb Distributed SelectRAM™ Memory
  - Each LUT works as a single-port or dual-port RAM/ROM
  - Cascade LUTs to build larger memories
  - Applications include flexible memory sizes, FIFOs, and buffers
- As Much as 1.87 Mb Embedded Block RAM
  - As many as 104 blocks of synchronous, cascadable 18 Kb block RAM
  - Configure each 18 Kb block as a single- or dual-port RAM
  - Supports multiple aspect ratios, data-width conversion, and parity
  - Applications include data caches, deep FIFOs, and buffers
- Memory Interfaces
  - Enable electrical interfaces such as HSTL and SSTL to connect to popular external memories
- Multipliers
  - Enable simple arithmetic and math as well as advanced DSP functions, enabling you to derive more than 330 billion MACs/s of DSP performance

- As many as 104 18 x 18 multipliers support 18-bit signed or 17-bit unsigned multiplication, which you can cascade to support wider bits
- Constant coefficient multipliers: On-chip memories and logic cells work hand-in-hand to build compact multipliers with a constant operand
- Logic Cell multipliers: Implement user-preferred algorithms such as Baugh-Wooley, Booth, Wallace tree, and others

- DCMs deliver sophisticated digital clock management that's impervious to system jitter, temperature, voltage variations, and other problems typically found with PLLs integrated into FPGAs.
  - Flexible frequency generation from 25 MHz to 325 MHz
  - 100 ps jitter
  - Integer multiplication and division parameters
  - Quadrature and precision phase shift control
  - 0, 90, 180, 270 degrees
  - Fine grain control (1/256 clock period) for clock data synchronization
  - Precise 50/50 duty cycle generation
  - Temperature compensation
- XCITE Digitally Controlled Impedance Technology – A Xilinx Innovation
  - I/O termination is required to maintain signal integrity. With hundreds of I/Os and advanced package technologies, external termination resistors are no longer viable.
  - I/O termination dynamically eliminates drive strength variation due to process, temperature, and voltage fluctuations.
- Spartan-3 XCITE DCI Technology Highlights
  - Series and parallel termination for single-ended and differential standards
  - Maximum flexibility with support of series and parallel termination on all I/O banks
  - Input, output, bidirectional, and differential I/O support
  - Wide series impedance range
  - Popular standard support, including LVDS, LVDSEXT, LVCMOS, LVTTL, SSTL, HSTL, GTL, and GTLP
  - Full- and half-impedance input buffers

Table 2-3: **XCITE DCI Technology Advantages**

Advantage	Details
• Second Generation Technology	Proven in the field and used extensively by customers
• Lower Costs	Fewer resistors, fewer PCB traces, and smaller board area, result in lower PCB costs.
• Absolute I/O Flexibility	Any termination on any I/O bank. Non- XCITE technology alternatives deliver limited functionality
• Maximum I/O Bandwidth	Less ringing and reflections maximize I/O bandwidth

• Immunity to Temperature and Voltage Changes	Temperature and voltage variations lead to significant impedance mismatches. XCITE technology dynamically adjusts on-chip impedance to such variations reducing and improving reliability
• Eliminates Stub Reflection	Improves discrete termination techniques by eliminating the distance between the package pin and resistor.
• Increases System Reliability	Fewer components on board, deliver higher reliability

Spartan-3 Features and Benefits

Table 2-4: **Spartan-3 FPGA Family Overview**

Device	XC3S50	XC3S200	XC3S400	XC3S1000	XC3S1500	XC3S2000	XC3S4000	XC3S5000
System Gates	50K	200K	400K	1000K	1500K	2000K	4000K	5000K
Logic Cells	1,728	4,320	8,064	17,280	29,952	46,080	62,208	74,480
Dedicated Multipliers	4	12	16	24	32	40	96	104
Block RAM Blocks	4	12	16	24	32	40	96	104
Block RAM Bits	72K	216K	288K	432K	576K	720K	1,728K	1,872K
Distributed RAM Bits	12K	30K	56K	120K	208K	320K	432K	520K
DCMs	2	4	4	4	4	4	4	4
I/O Standards	24	24	24	24	24	24	24	24
Max Single Ended I/O	124	173	264	391	487	565	712	784

Table 2-5: **Spartan-3 Features and Benefits**

Spartan-3 Feature	Benefit
FPGA fabric and routing, up to 5,000,000 system gates	Allows for implementation of system level function blocks, high on-chip connectivity and high-throughput
Block RAM – 18k blocks	Enables implementation of large packet buffers/FIFOs, line buffers
Distributed RAM	For implementing smaller FIFOs/Buffers, DSP coefficients
Shift register mode (SRL16)	16-bit shift register ideal for capturing high speed or burst mode data and to store data in DSP and encryption applications e.g. fast pipelining
Dedicated 18 x 18 multiplier blocks	High speed DSP processing; use of multipliers in conjunction with fabric allows for ultra-fast, parallel DSP operations
Single-ended signalling (up to 622 Mbps) – LVTTL, LVCMOS, GTL, GTL+, PCI, HSTL-I, II, III, SSTL-I,II	Connectivity to commonly used chip-to-chip, memory (SRAM, SDRAM) and chip-to- backplane signalling standards; eliminates the need for multiple translation ICs
Differential signalling (up to 622 Mbps) - LVDS, BLVDS, Ultra LVD, SRSDS and LDT	Differential signalling at low cost – bandwidth management (saving the number of pins, reduced power consumption, reduced EMI, high noise immunity

Digital clock management (DCM)	Eliminate on-chip and board level clock delay, simultaneous multiply and divide, reduction of board level clock speed and number of board level clocks, adjustable clock phase for ensuring coherency
Global routing resources	Distribution of clocks and other signals with very high fanout throughout the device
Programmable output drive	Improves signal integrity, achieving right trade off between Tco and ground bounce

Table 2-6: **Spartan-3E FPGA Family Overview**

Device	XC3S100E	XC3S250E	XC3S500E	XC3S1200E	XC3S1600E
System Gates	100K	250K	500K	1200K	1600K
Logic Cells	2,160	5,508	10,476	19,512	33,192
Dedicated Multipliers	4	12	20	28	36
Block RAM Blocks	4	12	20	28	36
Block RAM Bits	72K	216K	360K	504K	648K
Distributed RAM Bits	15K	38K	73K	136K	231K
DCMs	2	4	4	8	8
Max Differential I/O	40	68	92	124	156
Max Single Ended I/O	108	172	232	304	376
VQ100	66	66			
CP132		92	92		
TQ144	108	108			
PQ208		158	158		
FT256		172	190	190	
FG320			232	250	250
FG400				304	304
FG484					376

Spartan-3/3E System Integration

Spartan-3/3E can create substantial system savings by replacing other standard system functions.

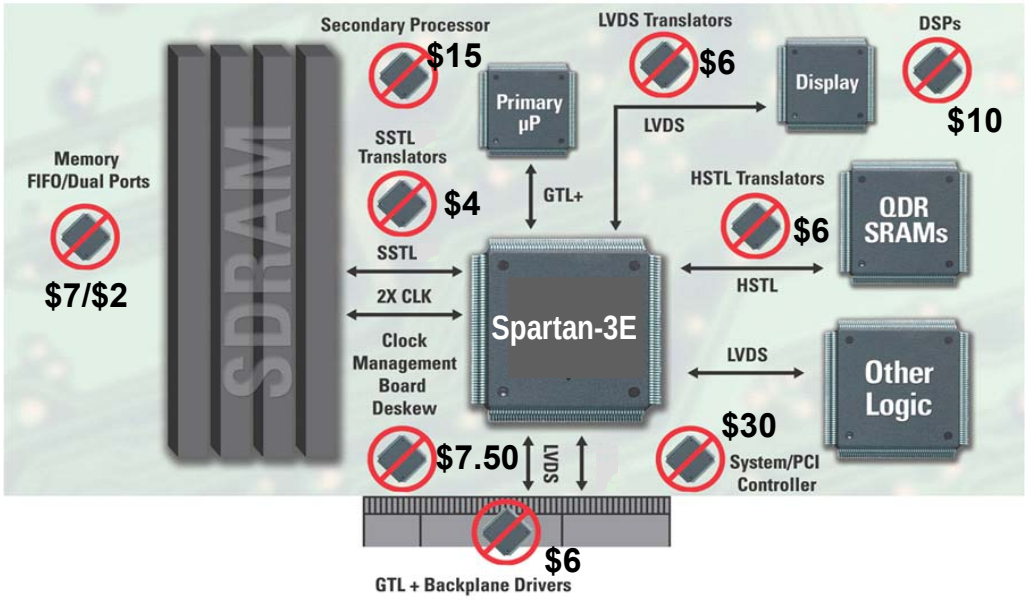


Figure 2-18: **Spartan-3/3E System Integration**

Spartan-3A/3AN

The Spartan-3A and Spartan-3AN families are pin-to-pin compatible low-cost FPGA families. The Spartan-3AN is non-volatile. Its offering is shown in Table 2-7.

Table 2-7: **Spartan-3AN Family**

	Spartan-3AN FPGA Platform				
Device	**XC3S50AN**	**XC3S200AN**	**XC3S400AN**	**XC3S700AN**	**XC3S1400AN**
System Gates	50K	200K	400K	700K	1400K
Logic Cells	1,584	4,032	8,064	13,248	25,344
Dedicated Multipliers	3	16	20	20	32
Block RAM Blocks	3	16	20	20	32
Block RAM Bits	54K	288K	360K	360K	576K
Distributed RAM Bits	11K	28K	56K	92K	176K
Flash Size Bits	1M	4M	4M	8M	16M
User Flash Bits	627K	2M	2M	5M	11M
DCMs	2	4	4	8	8
I/O Standards	26	26	26	26	26
Max Differential I/O	50	90	142	165	227
Max Single Ended I/O	108	195	311	372	502
	Package and I/O Offerings				
Device	**XC3S50AN**	**XC3S200AN**	**XC3S400AN**	**XC3S700AN**	**XC3S1400AN**
TQ144 20 x 20 mm	108				
FT256 17 x 17 mm		195			
FG400 21 x 21 mm			311		
FG484 23 x 23 mm				372	
FG676 27 x 27 mm					502

** Spartan-3AN platform is pin compatible with Spartan-3A platform supporting platform migration*

The Spartan-3A family is similar, as shown in Table 2-8.

Table 2-8: **Spartan-3A Family**

	Spartan-3A FPGA Platform				
Device	**XC3S50A**	**XC3S200A**	**XC3S400A**	**XC3S700A**	**XC3S1400A**
System Gates	50K	200K	400K	700K	1400K
Logic Cells	1,584	4,032	8,064	13,248	25,344
Dedicated Multipliers	3	16	20	20	32
Block RAM Blocks	3	16	20	20	32
Block RAM Bits	54K	288K	360K	360K	576K
Distributed RAM Bits	11K	28K	56K	92K	176K
DCMs	2	4	4	8	8
I/O Standards	26	26	26	26	26
Max Differential I/O	64	112	142	165	227
Max Single Ended I/O	144	248	311	372	502
	Package and I/O Offerings				
Device	**XC3S50A**	**XC3S200A**	**XC3S400A**	**XC3S700A**	**XC3S1400A**
TQ144 20 x 20 mm	108				
FT256 17 x 17 mm	144	195	195		
FG320 19 x 19 mm		248	251		
FG400 21 x 21 mm			311	311	
FG484 23 x 23 mm				372	375
FG676 27 x 27 mm					502

Virtex FPGAs

Virtex-4 FPGAs

With more than 100 innovations, the Virtex-4 family represents a new milestone in the evolution of FPGA technology. The idea behind the family was to offer higher performance, higher logic density, lower power, lower cost and more advanced capabilities over previous families. To offer one or two of these items is relatively easy – the challenge was to offer all at the same time. We did this through a combination of innovative process and circuit design, process development, the ASMBL architectural approach and the use of advanced embedded functions.

ASMBL Architecture

One of the most remarkable developments embodied in the new Virtex-4 FPGA family is the Advanced Silicon Modular Block (ASMBL) columnar architecture, which represents a fundamentally new way of constructing the FPGA floor plan and its interconnect to the package. First of all, ASMBL enables IO pins, clock pins and power and ground pins to be located anywhere on the silicon chip, not just along the periphery as with previous FPGA architectures. This in turn allows power and ground pins to be brought directly into the centre of the silicon die, thereby significantly reducing on-chip IR drops that can occur with the largest FPGAs running at the highest frequencies

Inside the Virtex-4

At the heart of the Virtex-4 FPGA is Xilinx' next generation 90nm triple oxide 10-layer copper CMOS process technology. With a dual oxide 90nm process, there would have been a trade off between performance and power. With triple oxide, that is not the case, both high performance and low power are achievable.

The columnar approach to building the ASMBL architecture enables Xilinx to cost-effectively develop multiple FPGA platforms, each with different combinations of feature sets. Thus, a specific platform can be optimized specifically for a certain domain of applications – such as logic, connectivity, DSP and embedded processing – to meet application requirements previously delivered only by ASICs, ASSPs and similar devices while remaining programmable at heart.

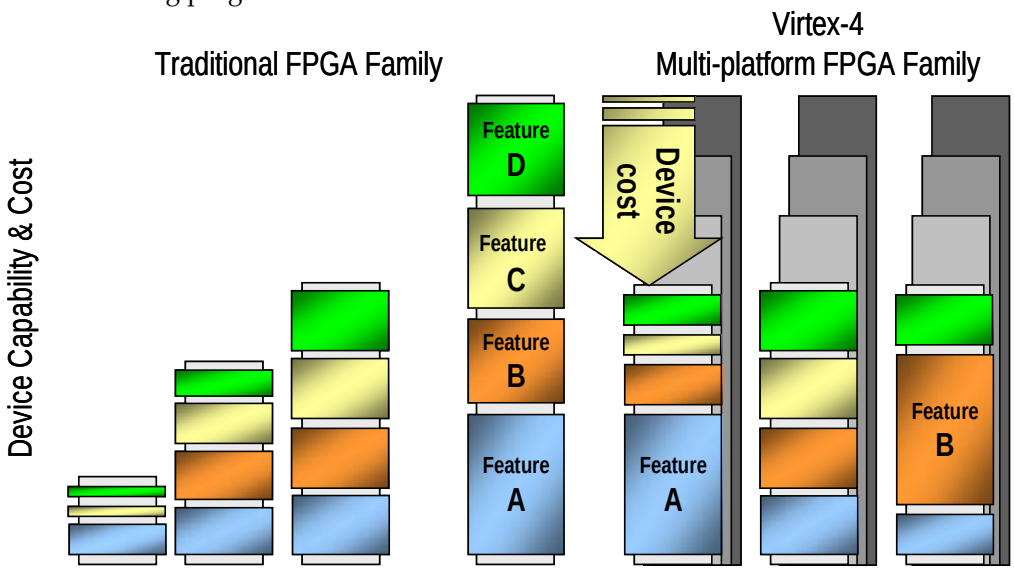


Figure 2-19: **Xilinx ASMBL Architecture**

Virtex-4 Variants

Virtex-4 is offered in three variants; Virtex-4 LX, optimized for high performance logic functions; Virtex-4 SX, optimized for high-performance signal processing; Virtex-4 FX, optimized for embedded processing and high-speed serial connectivity.

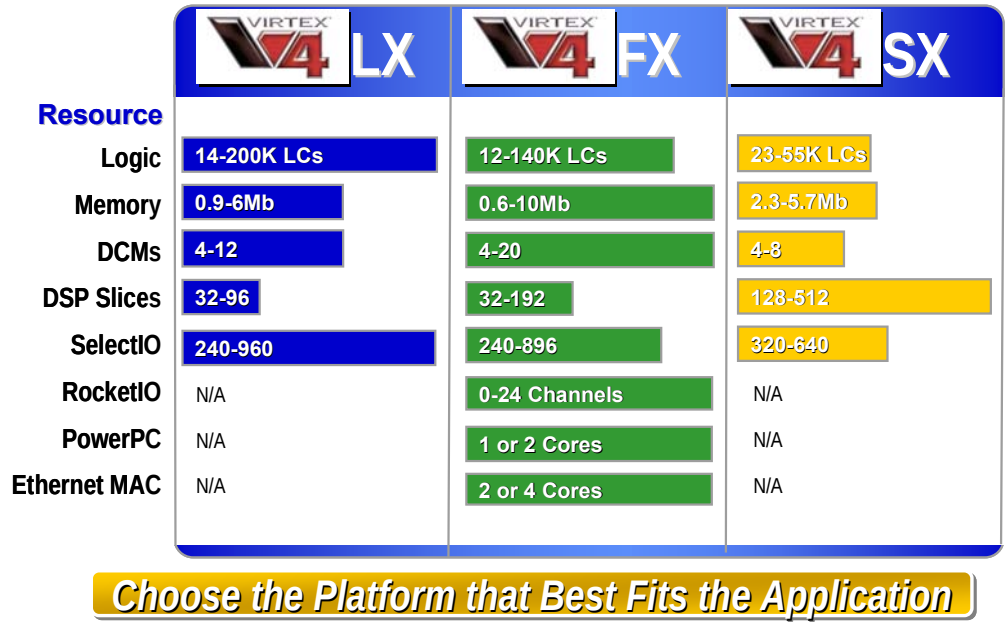


Figure 2-20: **Virtex-4 Platforms**

Virtex-4 LX

The most general-purpose family would be the Virtex-4 LX or logic optimized platform family. The LX family is similar in function to early Virtex-II devices without the embedded PowerPC processor or higher-speed serial I/Os found in the newer Virtex-II ProTM devices. All types of soft Intellectual Property (IP) cores can be implemented on this Platform, including various DSP blocks and soft processor cores such as MicroBlaze or PicoBlaze. The primary benefit is the use of highly integrated general-purpose logic elements, which makes this the most cost-effective logic platform.

The Virtex-4 LX Platform will have several family members with small–to-large-size devices, making it a suitable match for many applications. This family will have twice the logic density of any device shipping today. The cost benefits of using advanced 90nm silicon fabrication technology on 300mm wafers, together with cost-effective device packaging insures this platform's broad-based acceptance. The higher clock frequencies compared to previous generation platform FPGAs greatly expands the LX Platform's suitability for replacing ASICs.

- Highest Logic Capacity Ever (up to 200k LCs)
- Widest Capacity Range (8 LX devices ranging for 14k to 200k LCs)

Figure 2-21: **Virtex-4 LX**

Virtex-4 FX

Adding a PowerPC and high-speed serial transceivers creates the full-featured Virtex-4 FX platform. The combination of features, architecture, and fabrication process enables processor clock speeds of up to 450 MHz. Combining this capability with serial transceivers supporting any speed from 600 Mbps to 11.1Gbps provides a very capable high-performance platform FPGA that meets embedded-processing and connectivity-domain requirements.

The FX Platform incorporates advanced system features that are particularly useful in a wide-range of applications in the Telecom, Storage, and Networking space, and other system applications requiring high-performance processing and high-bandwidth I/O. These applications can be segmented into two general application domains based on the system behavior. The embedded-processing domain is dominated by control flow operations involving complex data types. The connectivity-domain involves message-based processing and is dominated by asynchronous data flow operations. Both domains are best implemented on the Virtex-4 FX full-featured platform.

- Additional Advanced System Functions
  - 10 Gbps RocketIO
  - PowerPC Cores
  - 10/100/1000 Ethernet MAC Cores
- Rich Memory Mix BRAM/FIFO
  - Up to Nearly 10 Mbits
- Six FX Devices Ranging from 12k to 140k LCs

Figure 2-22: **Virtex-4 FX**

Virtex-4 SX

Increasing the ratio of DSP and memory blocks to the number of logic elements creates the Virtex-4 SX, or signal-processing/DSP Platform family of devices. The changed ration of features creates a relatively smaller die size Platform in comparison to other Virtex-4 platforms for high-speed signal processing. This trade-off combined with the new DSP slice features packs the capability for the highest DSP performance into the most cost-effective Virtex-4 SX Platform. With significantly higher DSP bandwidth at much reduced power consumption of previous Virtex-II Pro devices, the Virtex-4 SX Platform delivers the most DSP performance per dollar compared to any other device. Each DSP Slice

implements an 18-bit x 18-bit MAC that can be clocked at 500MHz. The impact of Virtex-4 DSP specific enhancements to include new modes and capabilities, together with other parts of the optimized SX Platform architecture, enables more capable higher level DSP IP.

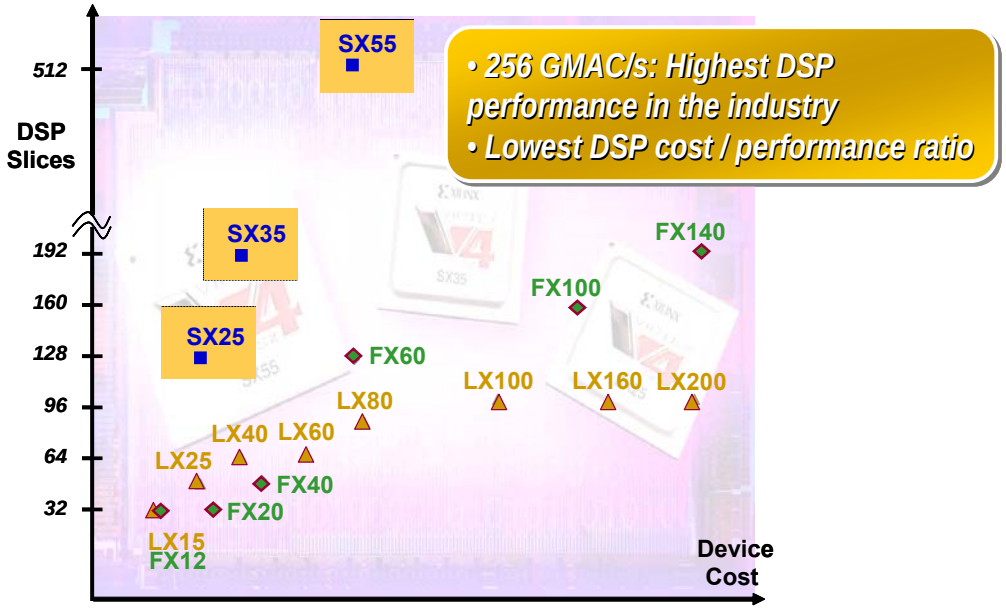


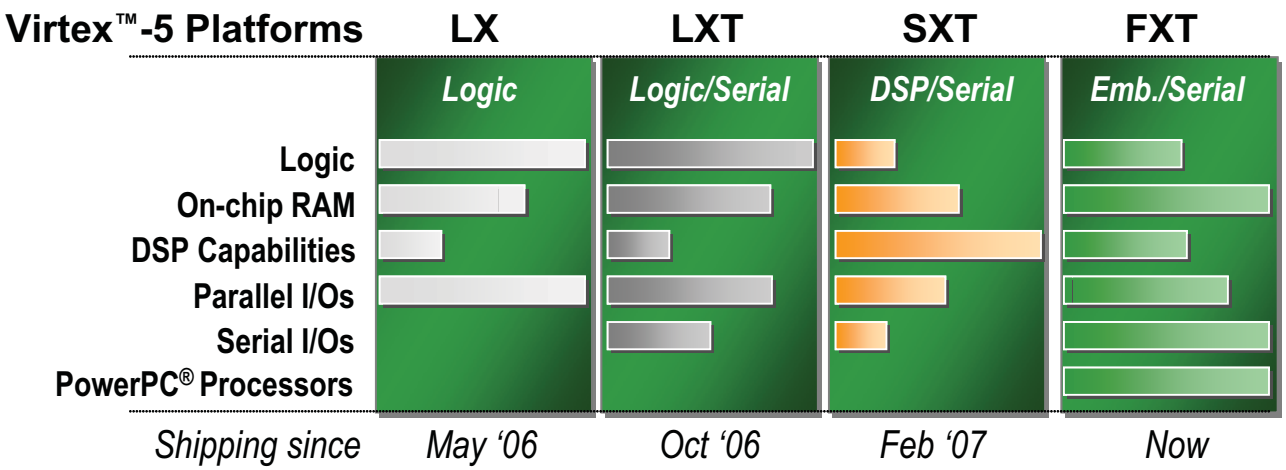
Figure 2-23: **Virtex-4 SX**

Virtex-5

The Virtex-5 family of FPGAs represents the 5th generation in the Virtex series of FPGAs. Built on 65 nm triple-oxide process technology, the proven multi-platform ASMBL architecture, and the all-new ExpressFabric technology, the Virtex-5 family offers customers the highest performance, most flexible connectivity, optimized power, lowest system cost and maximum productivity. The first family available at product launch is Virtex-5 LX, shown in Table 2-9. The complete Virtex-5 Platform family is as follows:

- Virtex-5 LX Platform - Optimized for high-performance logic
- Virtex-5 LXT Platform - Optimized for high-performance logic with low-power serial connectivity
- Virtex-5 SXT Platform - Optimized for DSP and memory-intensive applications with low-power serial connectivity
- Virtex-5 FXT Platform - Optimized for embedded processing and memory-intensive applications with highest-speed serial connectivity

Table 2-9: **Virtex-5 Family**



For a complete Virtex-5 Platform FPGA product table, see
http://www.xilinx.com/publications/prod_mktg/pn0010938-4.pdf

Military and Aerospace

Xilinx is the leading supplier of high-reliability PLDs to the aerospace and defense markets. These devices are used in a wide range of applications such as electronic warfare, missile guidance and targeting, RADAR, SONAR communications, signal processing, avionics, and satellites. The Xilinx QPro family of ceramic and plastic QML products provides you with advanced programmable logic solutions for next-generation designs. The QPro family also includes select products that are radiation hardened for use in satellite and other space applications. Our quality management system is fully compliant with all ISO9001 requirements. In 1997, Xilinx became fully qualified as a QML supplier by meeting all of the requirements for MIL Standard 38535.

Automotive and Industrial

Xilinx XA Solutions – Architecting Automotive Intelligence

In-car electronic content is increasing at a phenomenal rate. It includes such applications as navigation systems, entertainment systems, instrument clusters, advanced driver information systems, and communications devices. To address the needs of automotive electronics designers, Xilinx has created a new family of devices with an extended industrial temperature range option. This new "XA" family consists of existing Xilinx industrial grade (I) FPGAs and CPLDs, with the addition of a new extended temperature grade (Q) for selected devices. The new Q product grade (-40°C to +125°C ambient for CPLDs and junction for FPGAs) is ideal for automotive and industrial applications. The wide range of device density and package combinations enables you to deliver high-performance, cost-effective, flexible solutions that meet your application needs.

Design-In Flexibility

With Xilinx XA devices, you can design-in flexibility and get your product to market faster than ever before. Because many new standards continue to evolve (such as the LIN, MOST,

and FlexRay in-car busing standards), you need the flexibility to quickly modify your designs at any time. With our unique Internet Reconfigurable Logic (IRL) capability, you can remotely and automatically modify your designs, in the field, after your product has left the factory. By combining our latest XA PLDs with our solutions infrastructure of high-productivity software, IP cores, design services, and customer education, you can develop advanced, highly flexible products faster than ever before. For more information, visit **www.xilinx.com/automotive**.

XA Product Range

Table 2-10: **Temperature Grades**

Product Group	Temperature Grade/Range (ºC)		
	C	I	Q
FPGA	Tj = 0 to +85	Tj = -40 to +100	Tj = -40 to +125
CPLD	Ta = 0 to +70	Ta = -40 to +85	Ta = -40 to +125

Table 2-11: **Available XA Devices in Extended Temperature**

XA Device Family	Densities
Spartan-3 (3.3V)	50k gates to 1500k gates
Spartan-3E (3.3V)	100k to 1600k gates
XC9500XL (3.3V)	36 to 144 macrocells
CoolRunner-II (1.8V)	32 to 384 macrocells
Spartan-IIE (1.8V)	50k gates to 300k gates

XILINX®

Chapter 3

Xilinx Design Software

Design Tools

Programmable logic design has entered an era in which device densities are measured in the millions of gates, and system performance is measured in hundreds of megahertz. Given these system complexities, the critical success factor in the creation of a design is your productivity.

Xilinx offers complete electronic design tools that enable the implementation of designs in Xilinx PLDs. These development solutions combine powerful technology with a flexible, easy-to-use graphical interface to help you achieve the best possible designs within your project schedule – regardless of your experience level.

The availability of products such as WebPACK ISE software has made it much easier to design with programmable logic. Designs can be described easily and quickly using a description language such as ABEL, VHDL, Verilog™, or with a schematic capture package.

Schematic Capture Process

Schematic capture is the traditional method that designers have used to specify gate arrays and programmable logic devices. It is a graphical tool that allows you to specify the exact gates required and how you want them connected. There are four basic steps to using schematic capture:

1. After selecting a specific schematic capture tool and device library, begin building the circuit by loading the desired gates from the selected library. You can use any combination of gates that you need. You must choose a specific vendor and device family library at this time, but you don't yet have to know what device within that family you will ultimately use with respect to package and speed.
2. Connect the gates together using nets or wires. You have complete control and can connect the gates in any configuration required by your application.
3. Add and label the input and output buffers. These will define the I/O package pins for the device.

4. Generate a netlist.

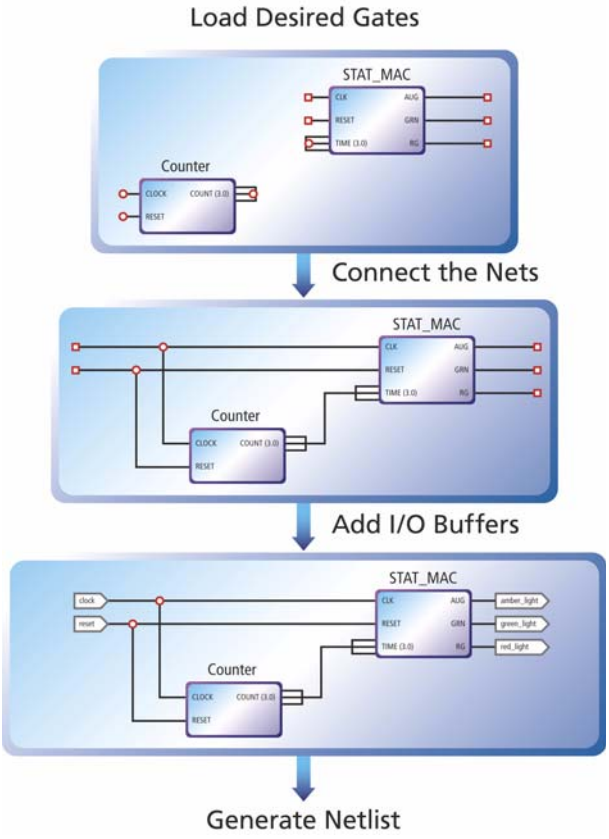


Figure 3-1: **PLD Design Flow**

A netlist is a text equivalent of the circuit. It is generated by design tools such as a schematic capture program. The netlist is a compact way for other programs to understand what gates are in the circuit, how they are connected, and the names of the I/O pins.

EDIF is the industry-wide standard for netlists; many others exist, including vendor-specific ones such as the Xilinx Netlist Format (XNF). Once you have the design netlist, you have all you need to determine what the circuit does.

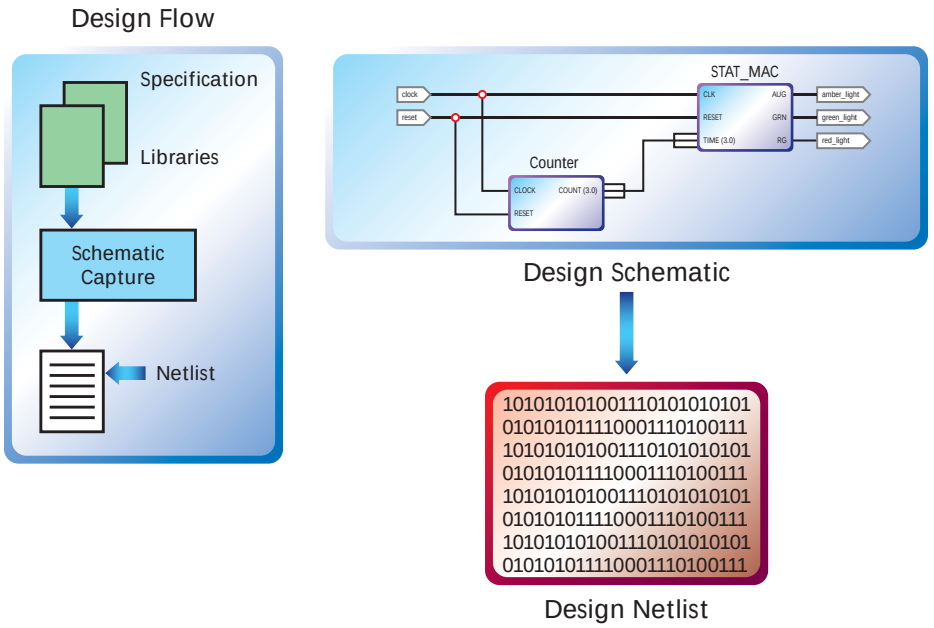


Figure 3-2: **Design Specification – Netlist**

The example on the previous pages is obviously very simplistic. Let's describe a more realistic design of 10,000 equivalent gates. The typical schematic page contains about 200 gates, contained with soft macros. Therefore, it would require 50 schematic pages to create a 10,000-gate design! Each page needs to go through all the steps mentioned previously: adding components, interconnecting the gates, adding I/Os, and generating a netlist. This is rather time-consuming, especially if you want to have a 20,000, 50,000, or even larger design.

Another inherent problem with using schematic capture is the difficulty in migrating between vendors and technologies. If you initially create your 10,000 gate design with FPGA vendor X and then want to migrate to a gate array, you would have to modify every one of those 50 pages using the gate array vendor's component library.

HDL Design Process

There has to be a better way, and, of course, there is. It is called high-level design (HLD), behavioral, or hardware description language (HDL). For our purposes, these three terms are essentially the same thing. The idea is to use a high-level language to describe the circuit in a text file rather than a graphical low-level gate description. The term *behavioral* is used because in this powerful language you describe the function or behavior of the circuit in words rather than figuring out the appropriate gates needed to create the application. There are two major flavors of HDL: VHDL and Verilog.

As an example, consider the design work needed specifying a 16 x 16 multiplier with schematic capture or an HDL file. A multiplier is a regular but complex arrangement of adders and registers that requires quite a few gates. Our example has two 16-bit inputs (A and B) and a 32-bit product output (Y = A x B) – for a total of 64 I/Os. This circuit requires approximately 6,000 equivalent gates. In the schematic implementation, the required gates

would have to be loaded, positioned on the page, and interconnected, with I/O buffers added. That would be about three days of work.

The HDL implementation, which is also 6,000 gates, requires eight lines of text and can be done in three minutes. This file contains all the information necessary to define our 16 x 16 multiplier. So, as a designer, which method would you choose? In addition to the tremendous time savings, the HDL method is completely vendor-independent. This opens up tremendous design possibilities for engineers.

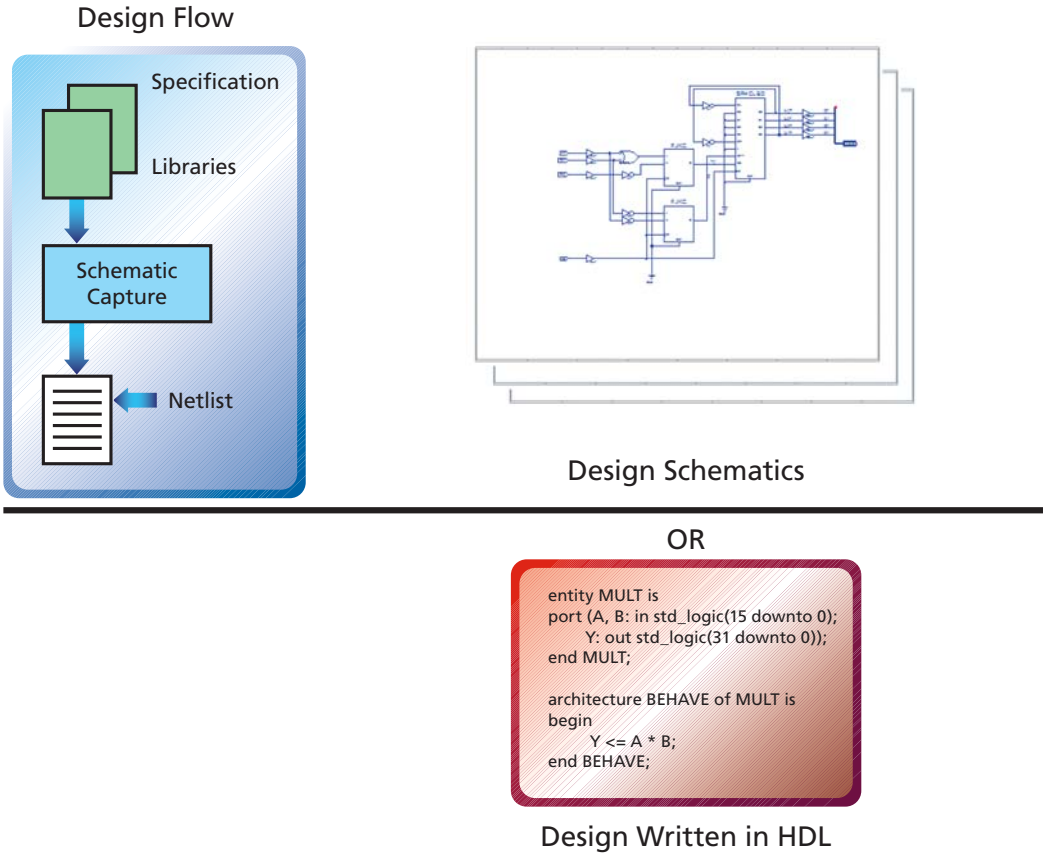


Figure 3-3: **Design Specification – Multiplier**

To create a 32 x 32 multiplier, you could simply modify the work you'd already done for the smaller multiplier. For the schematic approach, this would entail making three copies of the 30 pages, then figuring out where to edit the 90 pages so that they addressed the larger bus widths. This would probably require four hours of graphical editing. For the HDL specification, it would be a matter of changing the bus references from 15 to 31 in line 2, and 31 to 63 in line 3. This would probably require about four seconds.

HDL File Change Example

Before (16 x 16 multiplier):

```
entity MULT is
port(A,B:in std_logic_vector (15 downto 0);
       Y:out std_logic_vector (31 downto 0));
end MULT;
```

```
architecture BEHAVE of MULT is
begin
       Y <= A * B;
end BEHAVE;
```

After (32 x 32 multiplier):

```
entity MULT is
port(A,B:in std_logic_vector (31 downto 0);
       Y:out std_logic_vector (63 downto 0));
end MULT;

architecture BEHAVE of MULT is
begin
       Y <= A * B;
end BEHAVE;
```

HDL is also ideal for design re-use. You can share your "library" of parts with other designers at your company, therefore saving and avoiding duplication of effort.

HDL Synthesis

Once we have specified the design in a behavioral description we can convert it into gates using the process of synthesis. The synthesis tool does the intensive work of figuring out what gates to use, based on the high-level description file you provide (using schematic capture, you would have to do this manually.) Because the resulting netlist is vendor and device family-specific, you must use the appropriate vendor library. Most synthesis tools support a large range of gate array, FPGA, and CPLD device vendors.

In addition, you can specify optimization criteria that the synthesis tool will take into account when making the gate-level selections, also called *mapping*. Some of these options include: optimizing the complete design for the least number of gates, optimizing a certain section of the design for fastest speed, using the best gate configuration to minimize power, and using the FPGA-friendly, register-rich configuration for state machines.

You can easily experiment with different vendors, device families, and optimization constraints, thus exploring many different solutions instead of just one with the schematic approach.

To recap, the advantages of high level design and synthesis are many. It is much simpler and faster to specify your design using HDL, and much easier to make changes to the design because of the self-documenting nature of the language. You are relieved from the tedium of selecting and interconnecting at the gate level. You merely select the library and optimization criteria (e.g., speed, area) and the synthesis tool will determine the results. You can also try different design alternatives and select the best one for your application. In fact, there is no real practical alternative for designs exceeding 10,000 gates.

ISE Software

ISE advanced HDL synthesis engines produce optimized results for PLD synthesis, one of the most essential steps in your design methodology. It takes your conceptual HDL design definition and generates a logical or physical representation for the targeted silicon device. A state-of-the-art synthesis engine is required to produce highly optimized results with a fast compile and turnaround time. To meet this requirement, the synthesis engine must be tightly integrated with the physical implementation tool and proactively meet design

timing requirements by driving the placement in the physical device. In addition, cross probing between the physical design report and the HDL design code further enhances the turnaround time.

Xilinx ISE software provides seamless integration with leading synthesis engines from Mentor Graphics, Synopsys and Synplicity. The ISE product also includes Xilinx proprietary synthesis technology, or XST. With just the push of a button, you can start any leading synthesis engine within ISE. You can even use multiple synthesis engines to obtain the most optimized result of your programmable logic design.

Design Verification

Programmable logic designs are verified by using a simulator, which is a software program that confirms the functionality or timing of a circuit. The industry-standard formats used ensure that designs can be reused. If a vendors changes its libraries, only a synthesis recompile is necessary. Even if you decide to move to a different vendor and/or technology, you are just a compile away after selecting the new library. It is even design-tool independent, so you can try synthesis tools from different vendors and pick the best results. IP cores are commonly available in HDL format, since that makes them easier to modify and use with different device vendors.

After completing the design specification, you'll need to know if the circuit actually works as it's supposed to. That is the purpose of *design verification.* A simulator simulates the circuit. You'll need to provide the design information (via the netlist after schematic

capture or synthesis) and the specific input pattern, or *test vectors*, that you want checked. The simulator takes this information and determines the outputs of the circuit.

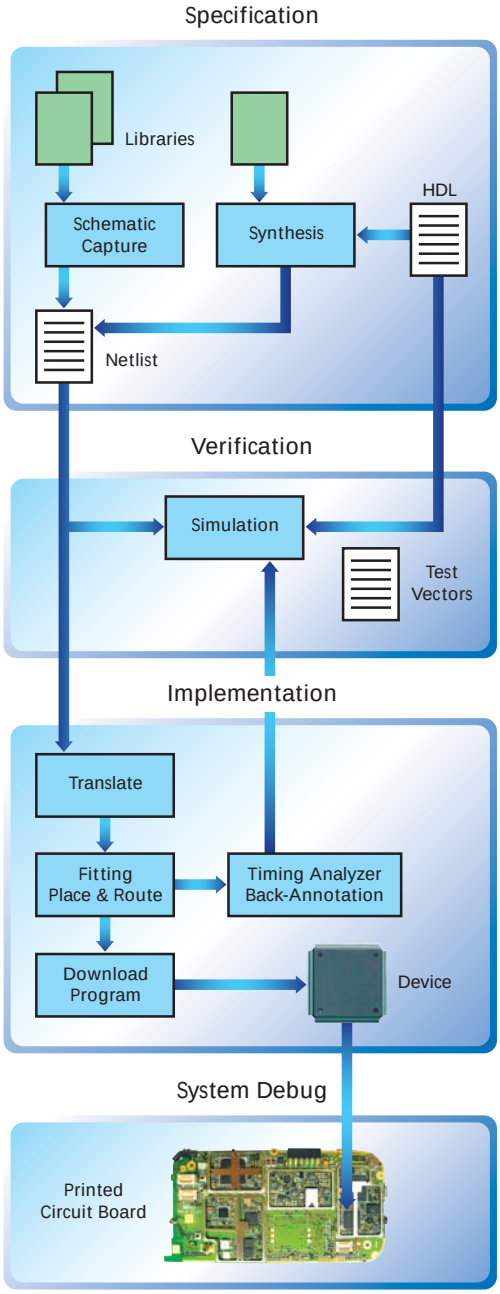


Figure 3-4: **The PLD Design Flow**

Functional Simulation

At this point in the design flow, a *functional simulation* only checks that the circuits give the right combinations of ones and zeros. You should conduct a *timing simulation* a little later in the design flow. If there are any problems, you can go back to the schematic or HDL file, make changes, regenerate the netlist, and then rerun the simulation. Designers typically

spend 50% of their development time going through this loop until the design works as required.

Using HDL offers an additional advantage when verifying the design: You can simulate directly from the HDL source file. This bypasses the time-consuming synthesis process that would normally be required for every design change iteration. Once the circuit works correctly, running the synthesis tool generates the netlist for the next step in the design flow – *device implementation.*

Device Implementation

A design netlist completely describes the design using the gates for a specific vendor/device family. Once your design is fully verified, it is time to place it on a chip, a process referred to as device implementation.

Translate comprises various programs used to import the design netlist and prepare it for layout. The programs will vary among vendors. Some of the more common programs during translate include: optimization, translation to the physical device elements, and device-specific design rule checking (for example, does the design exceed the number of clock buffers available in this device?). During this stage of the design flow, you will be asked to select the target device, package, speed grade, and any other device-specific options. The translate step usually ends with a comprehensive report of the results of all the programs executed. In addition to warnings and errors is usually a listing of device and I/O utilization, which helps you to determine if you have selected the best device.

Fitting

For CPLDs, this design step is called *fitting*, meaning to "fit" the design to the target device. In the diagram above, a section of the design is fit to the CPLD. CPLDs are a fixed architecture, so the software needs to pick the gates and interconnect paths that match the circuit. This is usually a fast process.

The biggest potential problem is if you had previously assigned the exact locations of the I/O pins, commonly referred to as *pin locking*. Most often, this occurs when using a legacy design iteration that has been committed to the printed circuit board layout. Architectures that support I/O pin locking (such as the Xilinx XC9500 and CoolRunner CPLDs) have a very big advantage. They allow you to keep the original I/O pin placements regardless of the number of design changes, utilization, or required performance. Pin locking is very important when using ISP. If you layout your PCB to accept a specific pin out, and then change the design, you can re-program confident that you pin out will stay the same.

Place and Route

For FPGAs, place and route programs are run after compile. "Place" is the process of selecting specific modules, or logic blocks, in the FPGAs where design gates will reside. "Route," as the name implies, is the physical routing of the interconnect between the logic blocks. Most vendors provide automatic place and route tools so that you don't have to worry about the intricate details of the device architecture. Some vendors offer tools that allow expert users to manually place and/or route the most critical parts of their designs to achieve better performance than with the automatic tools. Floorplanner is a type of manual tool.

Place and route programs require the longest time to complete successfully because it's a complex task to determine the location of large designs, ensure that they all get connected correctly, and meet the desired performance. These programs however, can only work well if the target architecture has sufficient routing for the design. No amount of fancy coding

can compensate for an ill-conceived architecture, especially if there are not enough routing tracks. If you were to encounter this problem, the most common solution would be to use a larger device. And you would likely remember the experience the next time you selected a vendor.

A related program is called *timing-driven place and route (TDPR).* This allows you to specify timing criteria that will be used during device layout. A *static timing analyzer* is usually part of the vendor's implementation software. It provides timing information about paths in the design. This information is very accurate and can be viewed in many different ways, such as displaying all paths in the design and ranking them from longest to shortest delay.

In addition, at this point you can use the detailed layout information after reformatting and go back to your chosen simulator with detailed timing information. This process is called *back-annotation* and has the advantage of providing the accurate timing as well as the zeros and ones operation of your design. In both cases, the timing reflects delays of the logic blocks as well as the interconnect. The final implementation step is the *download or program.*

Downloading or Programming

Download generally refers to volatile devices such as SRAM FPGAs. As the name implies, you download the device configuration information into the device memory. The bitstream that is transferred contains all the information to define the logic and interconnect of the design and is different for every design. Because SRAM devices lose their configuration when the power is turned off, the bitstream must be stored somewhere for a production solution. A common such place is a serial PROM. There is an associated piece of hardware that connects from the computer to a board containing the target device.

Program is used to program all non-volatile programmable logic devices, including serial PROMs. Programming performs the same function as download, except that the configuration information is retained after the power is removed from the device. For antifuse devices, programming can only be done once per device – hence the term one-time programmable. Programming of Xilinx CPLDs can be done in-system via JTAG or with a conventional device programmer such as Data IO. JTAG Boundary Scan – formally known as IEEE/ANSI standard 1149.1_1190 – is a set of design rules that facilitate testing, device programming, and debugging at the chip, board, and system levels.

In-system programming has an added advantage in that devices can be soldered directly to the PCB (such as TQFP surface-mount-type devices). If the design changes, the devices do not need to be removed from the board but simply re-programmed in-system.

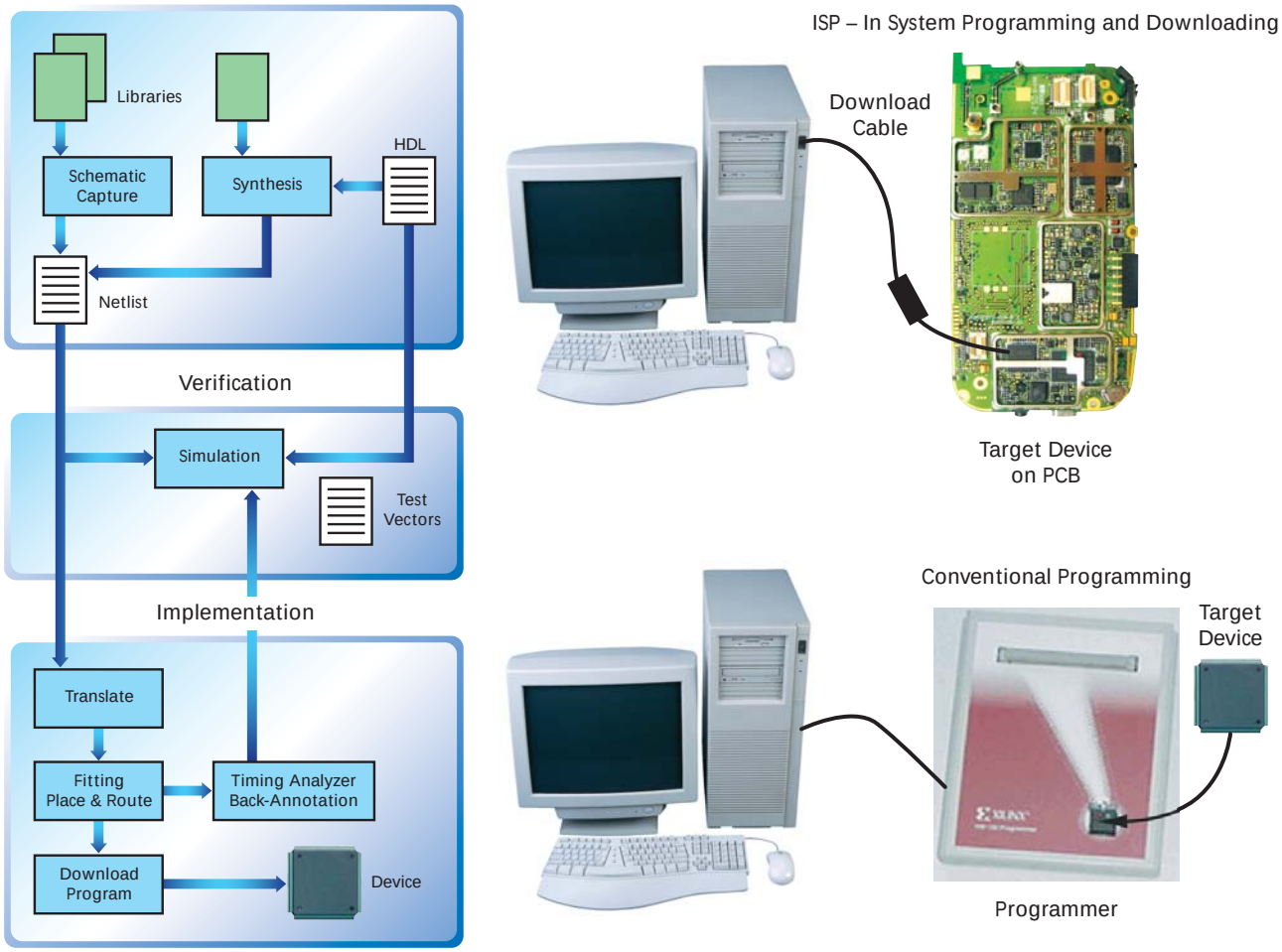


Figure 3-5: **Device Implementation – Download/Program**

System Debug

The device is now working, but you still need to verify that the device works in the actual board, a process called *system debug*. Any major problems here mean that you have made an assumption on the device specification that is incorrect, or have not considered some aspect of the signal required to/from the programmable logic device. If so, you can collect data on the problem and go back to the drawing (or behavioral) board. The "Design Tools Center" web pages cover both the Xilinx ISE tools suite as well as design tools from our software partners. It is arranged by the following topics:

Dynamic Verification

You can save time by using dynamic verification to intercept logical or HDL-based errors early in the design cycle. By exposing a design to realistic and extensive stimuli, you can find many functional problems at this stage. The following dynamic verification tools are supported:

- HDL Bencher™
- ISE Simulator
- ModelSim XE
- StateBench
- HDL Simulation Libraries

Debug Verification

Debug verification tools speed up the process of viewing, identifying, and correcting design problems at different stages of the design cycle. Debug verification includes the ability to view, "live," all internal signals and nodes within an FPGA. These tools can also assist in HDL-based designs by checking coding style for optimum performance. The following debug verification tools are supported:

- FPGA Editor Probe
- ChipScope ILA
- ChipScope Pro

Board-Level Verification

Using board-level verification tools ensures that your design performs as intended once integrated with the rest of the system. The Xilinx ISE environment supports the following board-level verification tools:

- IBIS Models
- Tau
- BLAST
- Stamp Models
- iMPACT

Advanced Design Techniques

As your FPGA requirements grow, your design problems can change. High-density design environments mean multiple teams working through distributed nodes on the same project, across the aisle or in different parts of the world. ISE software's advanced design options are targeted at making high-density designs as easy to realize as the smallest glue logic.

Floorplanner – The Xilinx high-level floorplanner is a graphic planning tool that lets you map your design onto the target chip. Floorplanning can efficiently drive your high-density design process.

Modular Design – This gives you the ability to partition a large design into individual modules. Each module can then be floorplanned, designed, implemented, and locked until the remaining modules are finished.

Partial Reconfigurability – Useful for applications requiring the loading of different designs into the same area of the device, partial reconfiguration allows you to flexibly change portions of a design without having to reset or completely reconfigure the entire device.

Internet Team Design – This allows managers to drive each team and its design module from a standard Internet browser using the corporate intranet structure.

High-Level Languages – As design densities increase, the need for a higher level of abstraction becomes more important. Xilinx is driving and supporting industry standards and their respective support tools.

Embedded SW Design Tools Center

Embedded Software Tools are for Virtex®-II Pro and Virtex-4 Platform FPGAs. The term "embedded software tools" most often applies to the tools required to create, edit, compile, link, load, and debug high-level language code, usually C or C++, for execution on a processor engine. With the Virtex-4 and Virtex-II Pro Platform FPGAs, you will be able to target design modules for either silicon hardware (FPGA gates) or as software applications, running on process engines like the embedded PowerPC™ hard core.

When it comes to embedded software development, Xilinx offers multiple levels of support. Xilinx supports the embedded processors with the Embedded Development Kit (EDK) for both low-cost and high-performance markets. More details on EDK can be found in the Design Tools Centre on the Xilinx website.

For hardware-centric engineers who want to move design modules into software running on the PowerPC core, Xilinx provides a simple and low-cost solution. Alternatively, if software-centric engineers want a feature-rich environment in which to develop more complex applications, Xilinx supplies access to specialized best-of-class tools from the embedded industry leader. This prevents you from having to embrace completely new development methodologies. You will be able to port existing legacy designs more easily to the Xilinx Platform FPGAs.

ISE WebPACK Software

The ISE WebPACK software, the only free downloadable design environment supported on Linux, is a reduced feature set version of the complete ISE tool suite. The full version of the ISE software, known as ISE Foundation, has all the tools necessary to complete a design targeting any architecture available from Xilinx. The ISE WebPACK tool set comprises all the tools necessary to complete designs targeted to Xilinx CPLDs and some Xilinx FPGAs.

This chapter explains what is available in ISE WebPACK and gives details on how to go about registering and installing the software.

Table 3-1: **WebPACK Operating System and Device Support**

Feature	ISE WebPACK
Operating Systems	Microsoft Windows XP/Vista Redhat Enterprise Linux 3 (32-bit)
Virtex Series	**Virtex:** XCV50 - XCV600 **Virtex-E:** XCV50E - XCV600E **Virtex-II:** XC2V40 - XC2V500 **Virtex-II Pro:** XC2VP2 - XC2VP7 **Virtex-4:** **LX:** XC4VLX15, XC4VLX25 **SX:** XC4VSX25 **FX:** XC4VFX12 **Virtex-5:** **LX:** XC5VLX30, XC5VLX50 **LXT:** XC5VLX30T, XC5VLX50T **FXT:** XC5VFX50T **Virtex Q:** XQV100 - XVQ600 **Virtex QR:** XQVR300 - XVQR600 **Virtex-EQ:** XQV600E
Spartan Series	**Spartan-II/IIE**: All **Spartan-3:** XC3S50 - XC3S1500 **Spartan-3E:** All **Spartan-3A/AN:** All **Spartan-3A DSP:** XC3SD1800A **XA (Xilinx Automotive) Spartan-3:** All
CoolRunner-II/A **CoolRunner XPLA3**	All
XC9500/XL/XV	All

Registration and Installation

The ISE WebPACK software is available from two sources, on CD and as a download from the internet. If this book was received as part of a CoolRunner-II or Spartan-3 Design Kit, it will have been accompanied by a copy of ISE WebPACK on CD. That CD will have a Product ID that will need to be registered to generate a registration key that will enable installation. This registration process and the downloading of the software can both be done from the ISE WebPACK main page for which visitors need to register. This page can be found by navigating as follows:

www.xilinx.com → **Products and Services** → **Design Tools** → **ISE WebPACK**

To register a CD, click on the 

button. After completion of a short survey, there will be an option to register the software. The web page asks for a Product ID.

This is usually on a sticker on the CD pack and takes the form ABC123456789. Once this has been entered, the 16 digit registration ID will be displayed on the following web page as well as emailed to the email address of the user.

To download the software, click the 

button. This will offer the option to download the complete ISE WebPACK tool suite, only the tools required for CPLD designs or only the programming tools. Xilinx recommends that, if the space is available, the entire tool suite is downloaded. Once the appropriate design tools have been downloaded, the software can be installed by simply double clicking on the self-extracting zip file.

Module Descriptions

In general, the design flow for FPGAs and CPLDs is very similar. Design Entry can be done in Schematic or HDL, such as VHDL, Verilog or, for CPLDs only, ABEL. The design can also comprise of a mixture of schematic diagrams and embedded HDL symbols. There is also a facility to create state machines in a diagrammatic form and let the software tools generate optimized code from a state diagram. All these steps will be seen in Chapter 4, ISE WebPACK Design Entry.

WebPACK ISE software incorporates the ISE Simulator Lite – Xilinx' new simulation tool. This tool will be used in the tutorial section of this book. The full version of the ISE Simulator is available in the full feature set ISE Foundation software. There is also the option to download the ModelSim Xilinx Edition (MXE) software. This is a version of Mentor Graphics' ModelSim simulator with the Xilinx libraries precompiled. For more information on the MXE software, refer to:

www.xilinx.com → **Products and Services** → **Design Tools** → **Verification**

The flow diagram below shows the similarities and differences between CPLD and FPGA software flows.

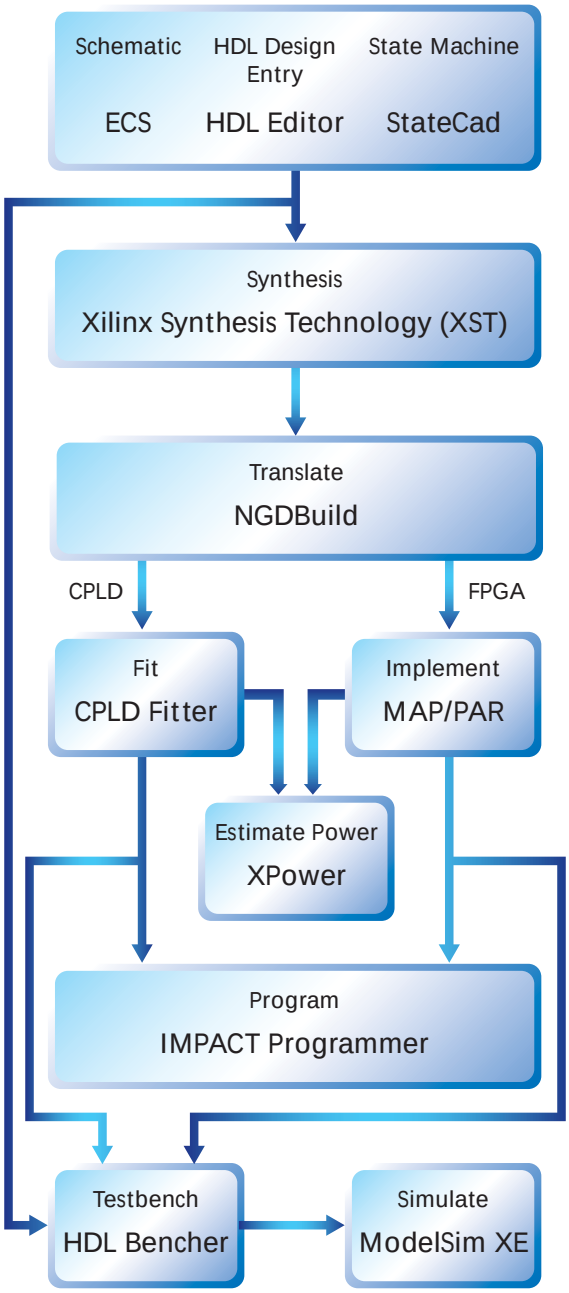


Figure 3-6: **WebPACK Software Design Flow**

When your design is complete and you are happy with the simulation results, you can then download the design to the required device.

Getting Started

Licenses

The MXE Simulator is the only tool that requires a license. MXE Simulator is licensed via the FlexLM product from Macrovision. It requires you to situate a starter license file on your hard drive, pointed to by a set **`lm_license_file`** environment setting. The license is free and you apply for it online after installation, after which you will receive a **`license.dat`** file via email.

From the Start menu, go to **Programs** → **ModelSimXE** → **Submit License Request**

Projects

When starting a project the default location of the project will be:

```
c:\Xilinx_WebPACK\bin\nt
```

You can create a unique directory on your hard drive for working on projects, for example:

```
c:\my_projects.
```

Should you need to uninstall and reinstall WebPACK ISE software due to problems on your system, we recommend that you delete the entire WebPACK ISE directory structure.

Updating Software

Between major releases of software, Service Packs are released to keep the software fully up to date. Service Packs often include new device features, updated characterization data and minor improvements to the tools. Xilinx recommends that, if a Service Pack is available for a version of the ISE WebPACK software, you install it at the earliest possible convenience.

Summary

In this chapter, we demonstrated how to access and install the ISE WebPACK software, and listed the devices it supports.

The next section will show how to embark upon a PLD design for the first time using the powerful features of WebPACK ISE software. The example design is a basic up/down counter and 7-segment display decoder. The design entry process is identical for CPLDs and FPGAs.

Chapter 4

WebPACK ISE Design Entry

This chapter describes a step-by-step approach to a simple design. The following pages are intended to demonstrate the basic PLD design entry and implementation process. In this example tutorial, you'll design a simple up/down counter and 7-segment display decoder in VHDL. Our design is initially targeted at the CoolRunner-II CPLD that is on the board in the CPLD Design Kit. If you received this manual as part of the design kit, you can download the design to the board and see it working at the end of the next chapter. If you received this manual alone, you can purchase a Design Kit from www.xilinx.com → **Online Store**. We also show how you can convert the project to target a Spartan-3E FPGA.

Design Entry

To start WebPACK ISE software, select:

Start → Programs → Xilinx ISE 10.1 → ISE → Project Navigator

To create a new project:

1. Select in Project Navigator: **File → New Project**.

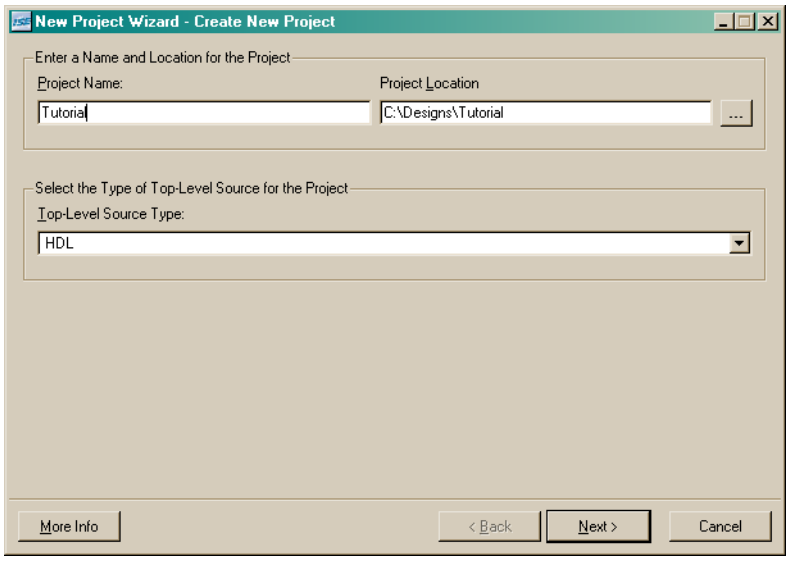


Figure 4-1: **New Project Window – Project Name**

2. Call the project "`Tutorial`" and put it in your Designs directory. For this tutorial, we will be using an HDL top level.
3. Click the **Next>** button.

4. Enter the following into the **New Project** dialog box:

   Device Family: **CoolRunner-II**

   Device: **xc2c256**

   Package: **TQ144**

   Speed Grade: **-7**

   Synthesis Tool: **XST (VHDL/Verilog)**

   Simulator:I **ISE Simulator (VHDL/Verilog)**

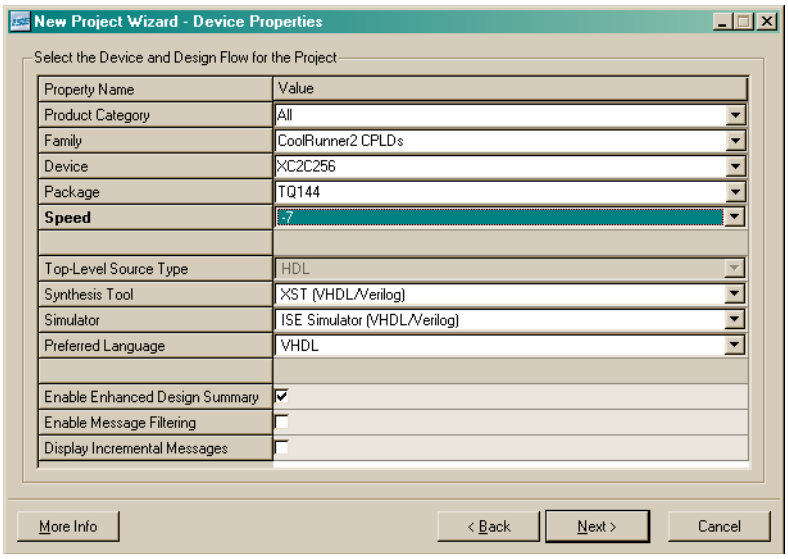


Figure 4-2: **New Project Window – Device and Design Flow**

5. Click the **Next>** button. Click **Next>** in the **New Source** wizard.
6. Click **Next>** in the **Add Existing Sources** dialog box. Click **Finish**. This will create a project.
7. Add a new source to the project by selecting **Project** → **New Source**.
8. Add a VHDL module and call it "`clock_divide`."

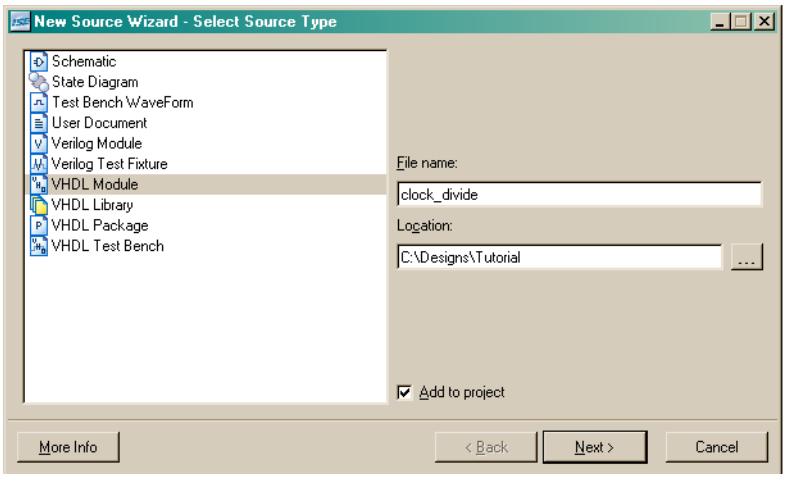


Figure 4-3: **New Source Window**

9. Click the **Next>** button.

   We will first create a clock divider module because we need to divide the 100 kHz frequency from the on board oscillator down to a frequency slow enough for the human eye to see. We will do this by using the CoolRunner-II clock divider to divide bown by 16, then building a 10-bit counter to get a final frequency of approximately 6 Hz.

10. Create a 4-Bit Counter Module

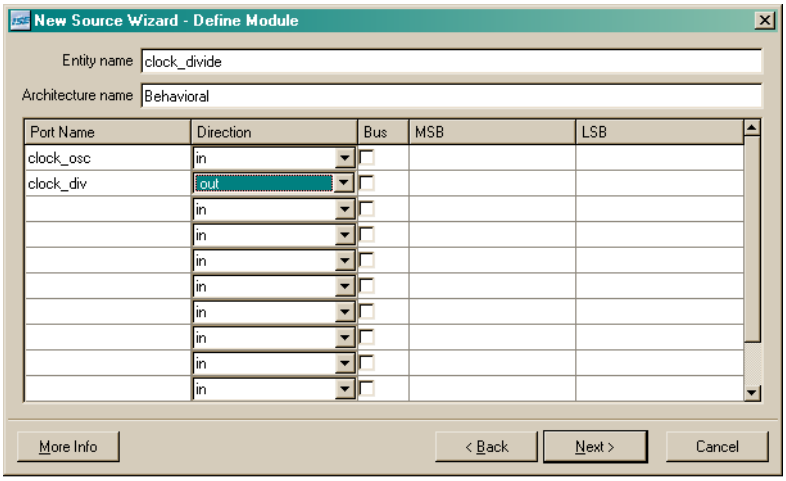


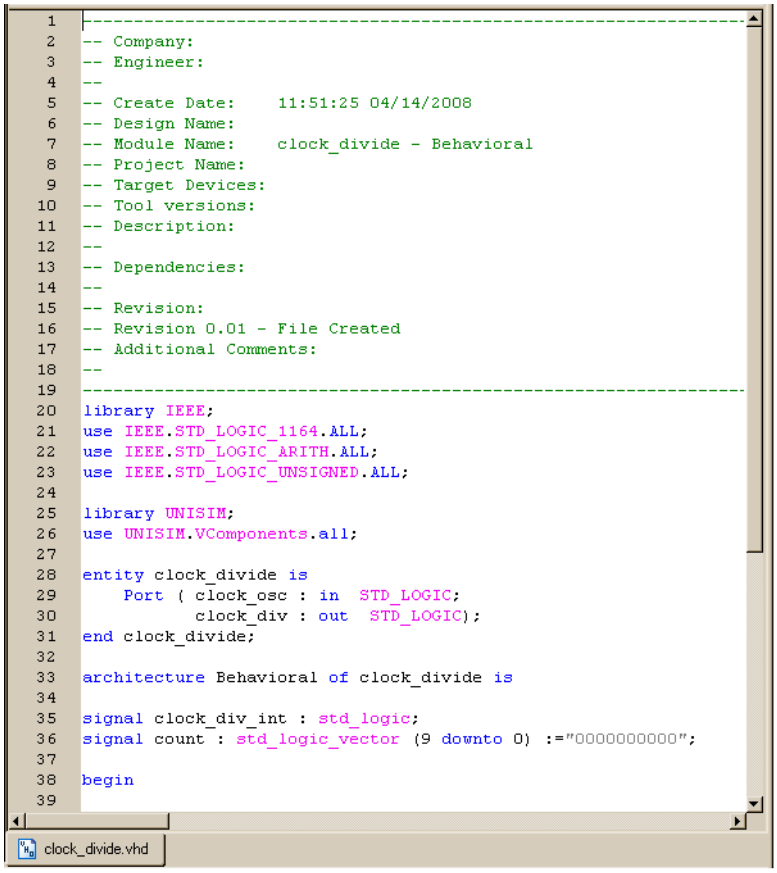
Figure 4-4: **Define VHDL Source Window**

11. Declare two ports: `clock_osc` and `clock_div`. The port `clock_osc` should be of direction "**in**", and `clk_div` should be direction "**out**".
12. Click the **`Next>`** button as many times as needed to get to the design summary window. Review the contents of the final window and click the **`Finish`** button.

This has automatically generated the entity in the counter VHDL module. Notice that a file called "clock_divide.vhd" has been added to the project in the **Sources in Project** window of the Project Navigator.

Figure 4-5: **Source in Project Window**

The source code will open automatically, but if you do not see it you can double-click on this source (highlighted in Figure 4-5) to open it in the WebPACK ISE Editor window.




```
1  ----------------------------------------------------------------------------------
2  -- Company:
3  -- Engineer:
4  --
5  -- Create Date:    11:51:25 04/14/2008
6  -- Design Name:
7  -- Module Name:    clock_divide - Behavioral
8  -- Project Name:
9  -- Target Devices:
10 -- Tool versions:
11 -- Description:
12 --
13 -- Dependencies:
14 --
15 -- Revision:
16 -- Revision 0.01 - File Created
17 -- Additional Comments:
18 --
19 ----------------------------------------------------------------------------------
20 library IEEE;
21 use IEEE.STD_LOGIC_1164.ALL;
22 use IEEE.STD_LOGIC_ARITH.ALL;
23 use IEEE.STD_LOGIC_UNSIGNED.ALL;
24
25 library UNISIM;
26 use UNISIM.VComponents.all;
27
28 entity clock_divide is
29     Port ( clock_osc : in  STD_LOGIC;
30            clock_div : out  STD_LOGIC);
31 end clock_divide;
32
33 architecture Behavioral of clock_divide is
34
35 signal clock_div_int : std_logic;
36 signal count : std_logic_vector (9 downto 0) :="0000000000";
37
38 begin
39
```

Figure 4-6: **clock_divide.vhd**

You can remove the source files from the WebPACK ISE GUI by clicking **Window** → **Float** in the ISE Menu. As the project builds, you will notice how the WebPACK ISE tool manages hierarchy and associated files in the **Sources** window.

HDL Editor

Double-clicking on any file name in the **Sources** window allows that file to be edited in the main **Text Editor**.

The Language Template

The language template is an excellent tool to assist you in creating HDL code. It has a range of popular functions such as counters, multiplexers, decoders, and shift registers. It also has templates for creating common operators (such as "IF/THEN" and "FOR" loops) often associated with software languages.

Language templates are used as a reference. They can be copied and pasted into the design, then customized for their intended purpose. Usually, you must change the bus width or signal names, and sometimes modify the functionality.

To use the language template:

1. In the HDL Editor place the cursor between `begin` and `end Behavioral.`
2. Open the language templates by clicking the button on the Project Navigator tool bar.

You can also access the language template from the **Edit** → **Language Template** menu.

3. Navigate to the Simple Counter in the Language Templates as follows:

VHDL → **Device Primitive Instantiation** → **CPLD** → **Clock Components** → **CR-II Clock Divider** → **Divide by 16** → **Simple Divider (CLK_DIV16)**

4. Highlight all the code as indicated in the comments and copy it into the `clock_divide.vhd` file at the architecture section, between the `begin` and `end` statements
5. You can close the language templates by clicking the **X** in the top right corner of the text window if you wish, but we will be using them again shortly.
6. Notice the color-coding used in the HDL Editor. The green text indicates a comment. The commented text in this template shows which libraries are required in the VHDL header. Figure 4-7 shows the CLK_DIV code that has been pasted into the clock_divide file.

```
10  -- Tool versions:
11  -- Description:
12  --
13  -- Dependencies:
14  --
15  -- Revision:
16  -- Revision 0.01 - File Created
17  -- Additional Comments:
18  --
19  ----------------------------------------------------------------------------------
20  library IEEE;
21  use IEEE.STD_LOGIC_1164.ALL;
22  use IEEE.STD_LOGIC_ARITH.ALL;
23  use IEEE.STD_LOGIC_UNSIGNED.ALL;
24
25  ---- Uncomment the following library declaration if instantiating
26  ---- any Xilinx primitives in this code.
27  --library UNISIM;
28  --use UNISIM.VComponents.all;
29
30  entity clock_divide is
31      Port ( clock_osc : in  STD_LOGIC;
32             clock_div : out  STD_LOGIC);
33  end clock_divide;
34
35  architecture Behavioral of clock_divide is
36
37  begin
38
39   CLK_DIV16_inst : CLK_DIV16
40     port map (
41        CLKDV => CLKDV,     -- Divided clock output
42        CLKIN => CLKIN      -- Clock input
43     );
44
45
46
47  end Behavioral;
48
49
```

Figure 4-7: **CLK_DIV16 Pasted Into File**

Edit the Clock Divide Module

1. At this point, you should uncomment the following two lines near the top of the `clock_divide.vhd`. To uncomment, simply delete the leading dashes (`---`), giving:

   ```
   library UNISIM;
   use UNISIM.VComponents.all;
   ```

2. Now we need to create a 10-bit counter to divide the clock down further. Open up the language templates again and navigate as follows:

   VHDL → Synthesis Constructs → Coding Examples → Counters → Binary → Up Counters → Simple Counter

3. Highlight the code and copy it into the `clock_divide.vhd` file between the `begin` and `end` statements after the clock divider code you previously pasted from the language templates. We need to make a few edits to the code to make it functional. First change all three references from "`<clock>`" to "`clock_div_int`" and then remove the <> brackets around both occurrences the word "`count.`"

4. As we have created two new signals called `count` and `clock_div_int`, we need to declare the signals. Between the architecture statement and the `begin` statement of the clock divider enter the following two signal declarations:

   ```
   signal clock_div_int : std_logic;
   signal count : std_logic_vector(9 downto 0):="0000000000";
   ```

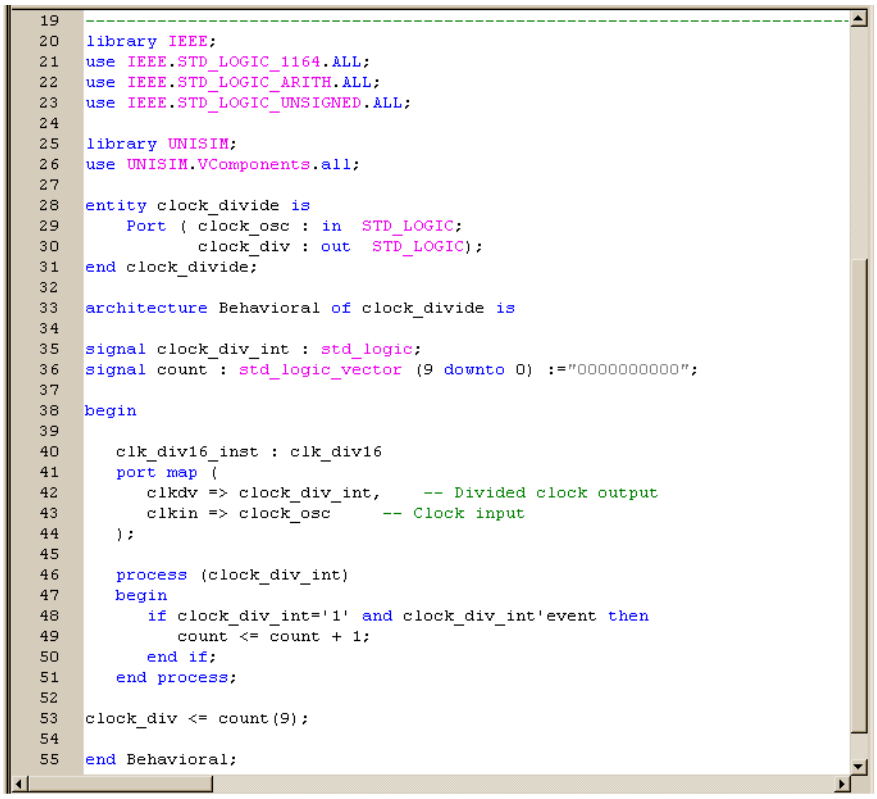
   The signal count is defined as a 10-bit signal that is given an initial value of all zeros. This is necessary for successful simulation.

5. We must now connect these two signals to the two ports of our clock divider instantiation. Edit the code so that it looks like the following:

   ```
   clk_div16_inst : clk_div16
   port map (
     clkdv => clock_div_int,
     clkin => clock_osc
   );
   ```

6. The final edit needed to make this code function as we require is to take the most significant bit of the counter and connect it up to our output port. After the "`end process;`" line and before the "`end Behavioral;`" line, enter the following logical assignment:

   ```
   clock_div <= count(9);
   ```



```
19  ----------------------------------------------------------------------------
20  library IEEE;
21  use IEEE.STD_LOGIC_1164.ALL;
22  use IEEE.STD_LOGIC_ARITH.ALL;
23  use IEEE.STD_LOGIC_UNSIGNED.ALL;
24
25  library UNISIM;
26  use UNISIM.VComponents.all;
27
28  entity clock_divide is
29      Port ( clock_osc : in  STD_LOGIC;
30             clock_div : out  STD_LOGIC);
31  end clock_divide;
32
33  architecture Behavioral of clock_divide is
34
35  signal clock_div_int : std_logic;
36  signal count : std_logic_vector (9 downto 0) :="0000000000";
37
38  begin
39
40     clk_div16_inst : clk_div16
41     port map (
42        clkdv => clock_div_int,     -- Divided clock output
43        clkin => clock_osc      -- Clock input
44     );
45
46     process (clock_div_int)
47     begin
48        if clock_div_int='1' and clock_div_int'event then
49           count <= count + 1;
50        end if;
51     end process;
52
53  clock_div <= count(9);
54
55  end Behavioral;
```

Figure 4-8: **Clock_Divide in VHDL Window**

A typical VHDL module consists of library declarations, an entity, and an architecture. The library declarations are needed to tell the compiler which packages are required. To summarize our module, the entity declares all ports associated with the design. Count (9 down to 0) means that count is a 10-bit logic vector.

This design has one input (`clock_osc`) and one output (`clock_div`). The actual functional description of the design appears after the begin statement in the architecture. The function of this design is to divide the incoming `clock_osc` signal from the oscillator by 16 using the hard clock divider, and to use the subsequent output to increment a signal "`count`" when `clock_div_int = 1` and there is an event on `clock_div_int`. This is resolved into a positive edge. The area still within the architecture – but before the begin statement – is where component declarations reside.

Save the Counter Module

Save the VHDL file by clicking the disk icon or selecting from the Project Navigator menus:

File → Save

Create an Up/Down Counter Module

We are now ready to create the second module. This module will be another counter that will step through 16 different sequences. You may need to revert back to viewing "Sources for: Synthesis/Implementation" before we commence.

1. To add another source go **Project** → **New Source** in the Project Navigator menu. Choose **VHDL Module** and call it "`counter`". This module will need five ports, four inputs named "`clock`","`reset`","`direction, and pause_design`" as well as a 4-bit output bus called "`count_out`".

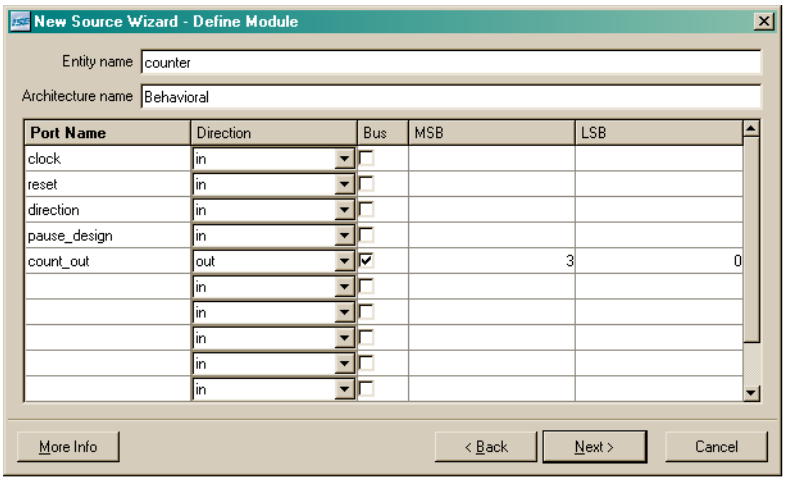


Figure 4-9: **Port Declaration Screen**

2. Place a check mark in the **Bus** check box for `count_out`. Set the **MSB** on `count_out` to 3 and the **LSB** to 0.
3. Click **Next>** and **Finish** until the VHDL file opens.
4. We need to go back to the counter section of the Language Templates and find a counter with direction control. In the Language Templates follow:

   VHDL → **Synthesis Constructs** → **Coding Examples** → **Counters** → **Binary** → **Up/Down Counters** → **/wCE**
5. Copy the code and paste it into the "`counter.vhd`" file between the `begin` and `End Behavioral` lines.
6. We now need to edit the signal names and insert a reset signal. You may find the **Edit** → **Replace** feature useful to perfom the following tasks:
   a. Change the reference to `<count_direction>` to `Direction` and remove the <> brackets around the `<clock>`and `<count>` signals.
   b. Change `<clock_enable>` to `pause_design`
   c. Change the polarity for the active condition for the `pause_design` from:

      ```
      If pause_design='1'
      ```

      to

      ```
      If pause_design='0'
      ```

   d. Add an asynchronous reset into the code by adding the following line directly after the begin of the process statement:

      ```
      if reset='0' then count <= "0000";
      ```

      The reset condition is '0' because the push button on the demonstration board is active low, making it is a '1' when not pressed, and a '0' when pressed.
7. Change the `process` statement to `process (clock , reset , pause_design)`

8. Then we need to create a 4-bit signal called `count`. Insert the following code between the architecture and begin lines:

   ```
   signal count : std_logic_vector (3 downto 0);
   ```

9. Finally, we need to connect the `Count` signal to the `count_out` port by inserting the following line after the `end process` statement and before the `end behavioral` statement:

   ```
   count_out <= count;
   ```

10. Change the `if` clock statement to start with `elsif` instead of `if`.

The code sould now look like this.

```
30  entity counter is
31      Port ( clock : in  STD_LOGIC;
32             reset : in  STD_LOGIC;
33             direction : in  STD_LOGIC;
34             pause_design : in  STD_LOGIC;
35             count_out : out  STD_LOGIC_VECTOR (3 downto 0));
36  end counter;
37
38  architecture Behavioral of counter is
39  signal count : std_logic_vector (3 downto 0);
40  begin
41
42  process (clock, reset, pause_design)
43  begin
44     if reset ='0' then count <= "0000";
45     elsif clock='1' and clock'event then
46        if pause_design='0' then
47           if direction='1' then
48              count <= count + 1;
49           else
50              count <= count - 1;
51           end if;
52        end if;
53     end if;
54  end process;
55
56  count_out <= count;
57
58  end Behavioral;
```

Figure 4-10: **Counter Code**

11. **Save** the file.

Create Conversion Module

Now we have completed two modules. The final functional model will convert a 4-bit binary number into 7-bit code to drive the seven segment display.

1. Again, start by creating a new VHDL Module source by going **Project** → **New Source**. This time call the module `display_drive`. This module only requires two ports – a 4-

bit input bus called `count_in` and a 7-bit output bus called `LED`, as shown in Figure 4-11.

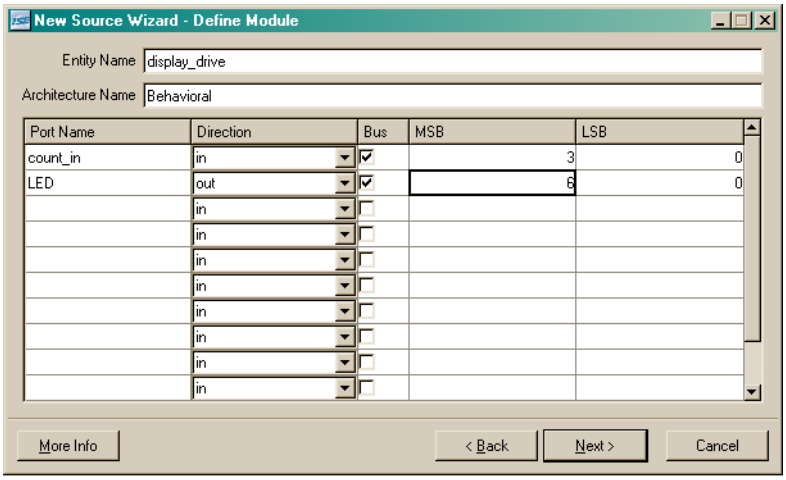


Figure 4-11: **Port Declarations**

2. Click **Next>** and **Finish**.
3. As this module is purely a logical conversion of one signal format to another, no process statement is needed. Insert the following code between the "`begin`" and `end Behavioral` lines:

```
with count_in Select
LED<= "1111001" when "0001", --1
"0100100" when "0010", --2
"0110000" when "0011", --3
"0011001" when "0100", --4
"0010010" when "0101", --5
"0000010" when "0110", --6
"1111000" when "0111", --7
"0000000" when "1000", --8
"0010000" when "1001", --9
"0001000" when "1010", --A
"0000011" when "1011", --b
"1000110" when "1100", --C
"0100001" when "1101", --d
"0000110" when "1110", --E
"0001110" when "1111", --F
"1000000" when others; --0
```

The `display_drive.vhd` files should appear as shown in Figure 4-12.

```
29
30  entity display_drive is
31      Port ( count_in : in  STD_LOGIC_VECTOR (3 downto 0);
32             LED : out  STD_LOGIC_VECTOR (6 downto 0));
33  end display_drive;
34
35  architecture Behavioral of display_drive is
36
37  begin
38
39
40  with count_in Select
41     LED<= "1111001" when "0001",   --1
42           "0100100" when "0010",   --2
43           "0110000" when "0011",   --3
44           "0011001" when "0100",   --4
45           "0010010" when "0101",   --5
46           "0000010" when "0110",   --6
47           "1111000" when "0111",   --7
48           "0000000" when "1000",   --8
49           "0010000" when "1001",   --9
50           "0001000" when "1010",   --A
51           "0000011" when "1011",   --b
52           "1000110" when "1100",   --C
53           "0100001" when "1101",   --d
54           "0000110" when "1110",   --E
55           "0001110" when "1111",   --F
56           "1000000" when others;   --0
57
58  end Behavioral;
59
```

Figure 4-12: **display_drive.vhd File**

4. **Save** this file.

Functional Simulation

You can now run a functional simulation on the `display_drive` module. With `display_drive.vhd` highlighted in the **Source** window, the **Process** window will give all the available operations for the particular module. A VHDL file can be synthesised and then implemented through to a bitstream. Normally, a design consists of several lower-level modules wired together by a top-level file. In this instance, we are going to simulate only one lower-level module to introduce the functional simulation methodology.

To simulate a VHDL file, you must first create a testbench.

1. From the **Project** menu, select **New Source** as before:

2. Select **Test Bench Waveform** as the source type and give it the name `display_drive_tb`.

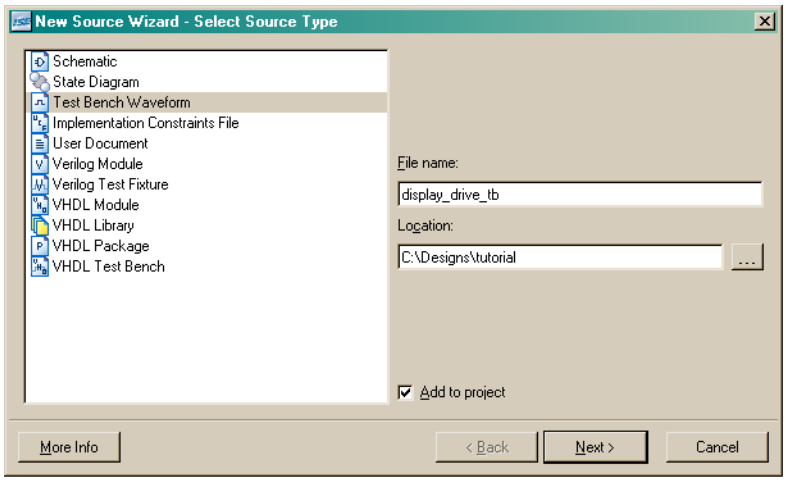


Figure 4-13: **Define Test Bench**

3. Click the **Next >** button.
4. The testbench is going to simulate the `display_drive` module, so when asked which source you want to associate the source with, select `display_drive` and click the **Next>** button.
5. Review the information and click **Finish** button.
6. The HDL Bencher tool now reads in the design. The initialize timing box sets the initial parameters. As this design is logic only and has no clock, the only parameter that needs to be set is the initial GSR length. The default setting of 100 ns is fine.
7. Click **Finish** to bring up the Waveform Editor.

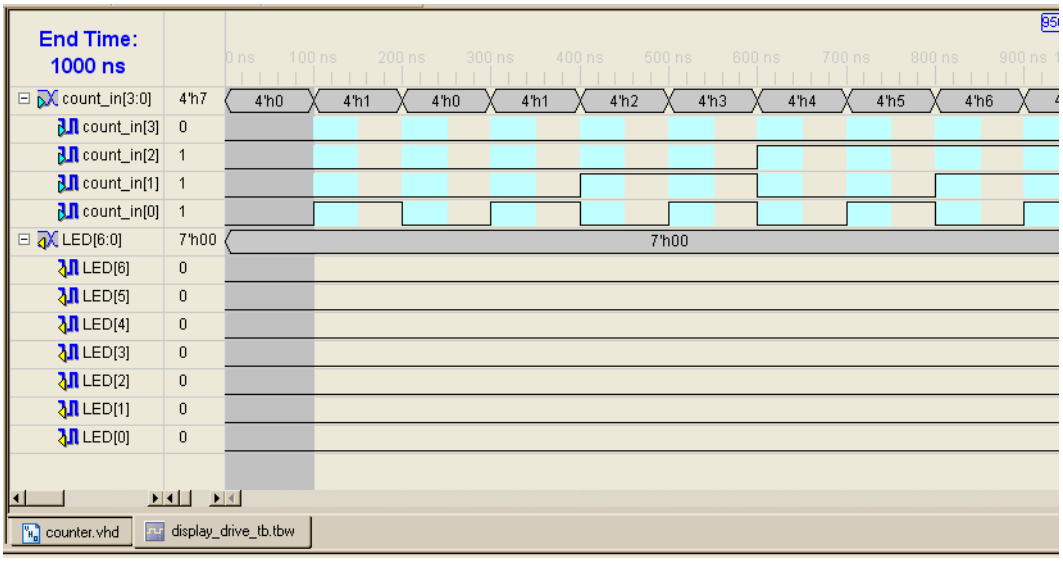


Figure 4-14: **HDL Bencher**

8. To simulate the design, you need to tell the testbench how you want the input signals to toggle. After the GSR pulse has finished, double-click in the `count_in` waveform. In the dialog box that appears, click **Pattern Wizard**. In the **Pattern Type** box, select **Count Up**. In the **Number of Cycles** box, select **8**. Select **Radix** to be **Binary**.

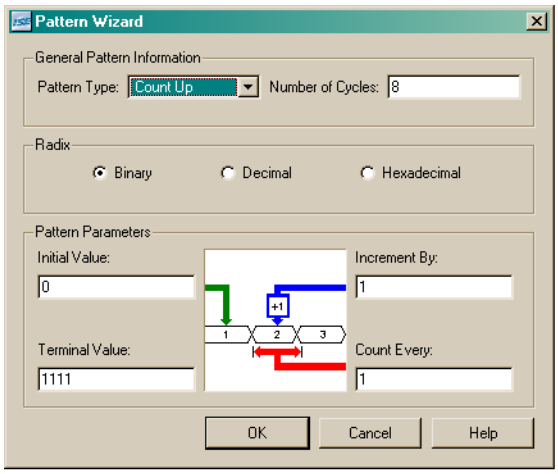


Figure 4-15: **Pattern Wizard**

9. Click **OK** to close the **Pattern Wizard** window.
10. Close this simulation interface by selecting **File** → **Close**, or by right-clicking on the bottom tab "**display_drive_tb.vhd**" and selecting **Close**.
11. Return to the implementation interface by selecting "**Source For: Implementation**" in the **Sources** window. This changes the processes available from simulation to design implementation.
12. To simulate the design, you need to change the sources you are viewing from implementation to behavioral simulation. In the box next to **Sources For:** select **Behavioral Simluation**.
13. Select `display_drive_tb`. You will notice that there are now different processes available in the **Process** window.
14. Expand **Xilinx ISE Simulator** and double-click **Simulate Behavioral Model**.

15. Select the **Simulation** tab to view the results. You may need to expand the signal names by clicking on the plus (**+**) signs.

Figure 4-16: **Simulate Behavioral Model**

The `LED[6:0]` signals now toggle as defined in the `display_drive.vhd` file.

16. Close the simulation by selecting **File** → **Close**, or by right-clicking on the bottom tab labeled "**Simulation**" and selecting **Close**. Click **Yes** when asked to confirm the closing of an active simulation.

Top-Level VHDL Designs

Now all we need to do is create a top level VHDL module to connect up our three blocks.

1. Use **Project** → **New Source** to create a new VHDL Module and give it the name `top`. As this is the top level of our design, all the ports we declare in this module will be pins on our device. In this design, we have four input ports named `clock_ext`,

reset_ext, direction_ext, and pause_design along with one output port, a 7-bit bus called LED_output. Once these are declared, your code should look like this:

```
20  library IEEE;
21  use IEEE.STD_LOGIC_1164.ALL;
22  use IEEE.STD_LOGIC_ARITH.ALL;
23  use IEEE.STD_LOGIC_UNSIGNED.ALL;
24
25  ---- Uncomment the following library declaration if instantiating
26  ---- any Xilinx primitives in this code.
27  --library UNISIM;
28  --use UNISIM.VComponents.all;
29
30  entity top is
31      Port ( clock_ext : in  STD_LOGIC;
32             reset_ext : in  STD_LOGIC;
33             direction_ext : in  STD_LOGIC;
34             pause_design : in  STD_LOGIC;
35             LED_output : out  STD_LOGIC_VECTOR (6 downto 0));
36  end top;
37
38  architecture Behavioral of top is
39
40  begin
41
42
43  end Behavioral;
44
```

Figure 4-17: **top.vhd Module**

2. Because we are going to use the three previous models as components in our top level design, we need to first declare them as components, just as we did with the clock divider module in the first module we created. Between the architecture behavioral of top is line and the begin line, insert the three component declarations as follows:

```
component clock_divide
     Port ( clock_osc : in STD_LOGIC;
            clock_div : out STD_LOGIC);
end component;

component counter
Port (      clock : in STD_LOGIC;
            reset : in STD_LOGIC;
            direction : in STD_LOGIC;
            pause_design : in STD_LOGIC;
            count_out : out STD_LOGIC_VECTOR (3 downto 0)
);

component display_drive
     Port ( count_in : in STD_LOGIC_VECTOR (3 downto 0);
     LED : out STD_LOGIC_VECTOR (6 downto 0)
     );
end component;
```

Notice that the ports of the components exactly match the ports in the three modules we created earlier.

3. Now that the components have been declared, we must use instances of them in the design. Enter the following text into the code after the begin and before the end Behavioral statements:

```
U1: clock_divide
     Port map (
          clock_osc => clock_ext,
          clock_div => clock_div_sig
```

```
     ) ;
U2: counter
     Port map (
     clock => clock_div_sig,
     reset => reset_ext,
     direction => direction_ext,
     pause_design => pause_design,
     count_out => count_in_sig
     );
U3: display_drive
     Port map (
          count_in => count_in_sig,
          LED => LED_output
     );
```

Notice that the ports of the components have all been connected to either a port in our entity declaration (`clock_ext`, `reset_ext`, `direction_ext` or `LED_output`) or an intermediate signal (`clock_div_sig` or `count_in_sig`).

4. We now need to declare these two signals that we have just used. After your last component declaration and before `begin`, enter the following two signal declarations:

```
signal count_in_sig : std_logic_vector (3 downto 0);
signal clock_div_sig : std_logic;
```

The code should be as shown in Figure 4-18.

```
30  entity top is
31      Port ( clock_ext : in  STD_LOGIC;
32             reset_ext : in  STD_LOGIC;
33             direction_ext : in  STD_LOGIC;
34             pause_design : in  STD_LOGIC;
35             LED_output : out  STD_LOGIC_VECTOR (6 downto 0));
36  end top;
37
38  architecture Behavioral of top is
39
40  component clock_divide
41  Port ( clock_osc : in STD_LOGIC;
42         clock_div : out STD_LOGIC);
43  end component;
44
45  component counter
46  Port ( clock : in STD_LOGIC;
47         reset : in STD_LOGIC;
48         direction : in STD_LOGIC;
49         pause_design : in STD_LOGIC;
50         count_out : out STD_LOGIC_VECTOR (3 downto 0) );
51
52  end component;
53  component display_drive
54  Port ( count_in : in STD_LOGIC_VECTOR (3 downto 0);
55         LED : out STD_LOGIC_VECTOR (6 downto 0) );
56  end component;
57
58  signal count_in_sig : std_logic_vector (3 downto 0);
59  signal clock_div_sig : std_logic;
60
61  begin
62
63  U1: clock_divide
64  Port map (
65  clock_osc => clock_ext,
66  clock_div => clock_div_sig ) ;
67
68  U2: counter
69  Port map (
70  clock => clock_div_sig,
71  reset => reset_ext,
72  direction => direction_ext,
73  pause_design => pause_design,
74  count_out => count_in_sig );
75
76  U3: display_drive
77  Port map (
78  count_in => count_in_sig,
79  LED => LED_output );
80
81
82  end Behavioral;
```

Figure 4-18: **Top Level Code**

5. **Save** the file. You should now see in the Project Navigator **Sources** window that the source files are ordered in a hierarchy. `Top.vhd` is the top level and the other three modules all sit underneath as shown in Figure 4-19.

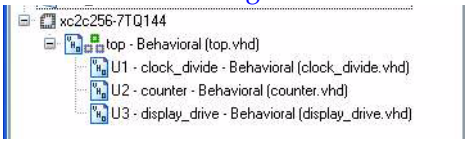


Figure 4-19: **Hierarchy in Sources Window**

Top-Level Schematic Designs

Sometimes, it's easier to visualize designs when they have a schematic top level that instantiates the individual blocks of HDL. The blocks can then be wired together in the

traditional method. For designs in the WebPACK ISE tool, the entire project can be schematic- based.

This section discusses the method of connecting VHDL modules via the ECS schematic tool. If you worked through the previous section, you will first need to remove the top level VHDL file top.vhd from the project. To do this, highlight the file in the sources for Synthesis/Implementation View, right click and select **Remove,** then click the **Yes** button in the dialog box.

This action will take you back to the stage in the flow with only the three lower level modules in the project. The **Sources** window module view should look like Figure 4-20 below.

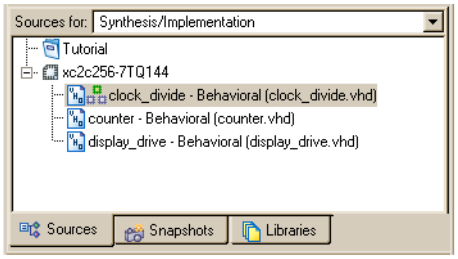


Figure 4-20: **Sources in Project Window (Synthesis/Implementation View)**

ECS Hints

The ECS schematic capture program is designed around you selecting the action you wish to perform, followed by the object on which the action will be performed. In general, most Windows applications currently operate by selecting the object and then the action to be performed on that object. Understanding this fundamental philosophy of operation makes learning ECS a much more enjoyable experience.

Creating a Top Level Schematic Design

Before we create a schematic, we need to make symbols out of the three VHDL modules. This adds a schematic representation of each module to the schematic library for this project.

1. Highlight the `clock_divide.vhd` module in the **`Sources`** window and then expand **Design Utilities** in the **Process** window. Double-click **Create Schematic Symbol**. Repeat these actions for the `counter.vhd` and `display_drive.vhd` files.

2. Now we need to make a schematic sheet. From the **Project** menu, select **New Source** → **Schematic** and give it the name `top_sch` as shown in Figure 4-21.

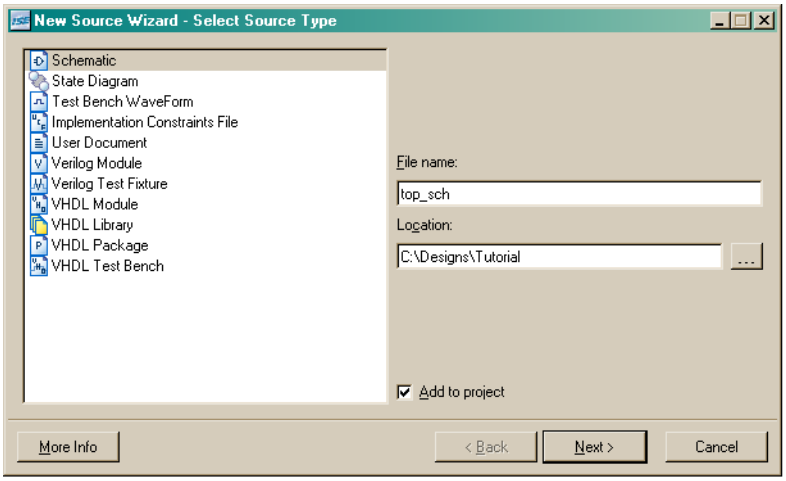


Figure 4-21: **New Source Window Showing top_sch**

3. Click the **Next>** button, then the **Finish** button. The **ECS Schematic Editor** window will now appear. There will also be an **Options** tab below the **Processes** window.
4. The symbol libraries can be found under the **Symbol** tab on the right hand tab in the **Sources** window You will see that there is a category showing the path to your project directory. If you click this category, you will see three symbols listed below. These are the three symbols that you have just created, as shown in Figure 4-22.

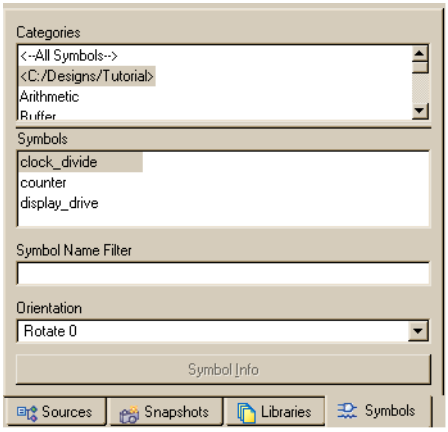


Figure 4-22: **Symbols Tab**

5. Click the `clock_divide` symbol and move your cursor over to the schematic sheet. Place the `clock_divide` symbol in the middle of the left hand side of the schematic sheet.
6. Then click the `counter` symbol and place it to the right of the previous symbol with space between them.

7. Finally, click the `display_drive` symbol and place it in the middle of the right hand side of the schematic sheet. Your sheet should now look something like this: .

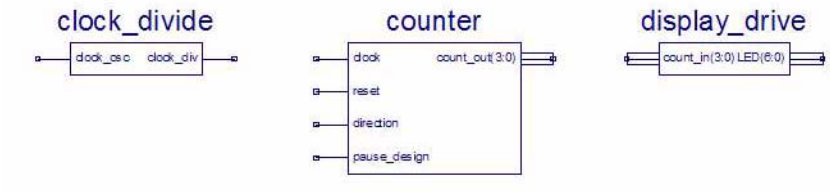


Figure 4-23: **Schematic Sheet**

Connecting the Blocks

1. Now the blocks must be connected together. Select the **Add Wire** tool from the toolbar.

Figure 4-24: **Add Wire Tool**

2. Click the `clock_div` output of the `clock_divide` module and then click the `Clock` input of the `Counter` module. Draw a wire between the two ports.
3. Repeat the process between the `Count_out(3:0)` output of the `Counter` symbol and the `count_in(3:0)` input of the `display_drive` symbol.
4. With the **Add Wire** icon still selected, we need to draw wires from the input and output ports in order that we can make them into I/O pins. First click the `clock_osc` port of the `clock_divide` symbol and draw a wire towards the left of the sheet. It does not have to be too long. Then repeat the process with the `reset`, `direction` and `pause_design` ports of the counter symbol. Then draw one wire below the other input wires for the `pause_design` signal. Finally, draw a wire from the `LED(6:0)` output of the `display_drive` symbol towards the right of the sheet.

Your schematic should now look like this:

Figure 4-25: **Connected Blocks**

Naming the Blocks

To name the ports click **Add** → **Net Name** in the menu or select the **Add Net Names** tool from the **Drawing** toolbar:

1. Select the **Add Net Names** tool from the **Drawing** toolbar.

2. The net naming options will appear in the process window. In the **Name** section, type `clock_ext` and then click the end of the net you attached to the `clock_osc port`.

Type `reset_ext` and click the end of the net attached to the reset port. Then type `direction_ext` and click the end of the net attached to the direction port. Then type `pause_design` and click the end of the net attached to the `pause_design` port.

3. Finally, type `LED_output(6:0)` and click the end of the output of the `display_drive` symbol.

I/O Markers

1. To make these named nets into I/O ports, select **Add** → **I/O Marker** or click the **Add I/O Marker** tool from the **Drawing** toolbar:

2. Then click and hold the left mouse button and draw a box round each of the five named nets. The direction of the port (input or output) will be correctly selected automatically. The finished schematic should look like this:

Figure 4-26: **Adding I/O Markers**

3. Save the design (**File** → **Save**) and exit the **Schematic Editor**. You will notice that the ISE software automatically recognizes that the schematic file is the top level file and reorganzes the design accordingly.

In the **Design Entry** utilities, you can view the VHDL code that was created from the schematic when `top_sch` was selected in the **Sources** window. The synthesis tool actually works from this file.

XILINX®

Chapter 5

Implementing CPLD Designs

Introduction

After you have successfully simulated your design, the synthesis stage converts the code-based or schematic-based design into an NGC netlist file. The netlist is a non-readable file that describes the actual circuit to be implemented at a very low level. The implementation phase uses the netlist and a constraints file to recreate the design using the available resources within the CPLD. Constraints may be physical or timing and are commonly used for setting the required frequency of the design or declaring the required pinout.

The first step is Translate, also known as NGD Build because it is building an NGD file. This step checks the design and ensures that the netlist is consistent with the chosen architecture. Translate also checks the UCF for any inconsistencies. In effect, this stage prepares the synthesized design for use within a CPLD.

The fit stage distributes the design to the resources in the CPLD and places those resources according to the constraints specified. Obviously, if the design is too big for the chosen device, the fit process will not be able to complete its job. The fitter uses the constraints that were present in the UCF file to understand timing and may sometimes decide to change the design to meet timing specifications. For example, sometimes the fitter will change the D-Type flip-flops in the design to Toggle Type or T-Type registers. It all depends on how well the design converts into product terms.

Note: Once the fitter has completed, it is good practice to re-simulate. As all the logic delays added by the macrocells, switch matrix, and flip-flops are known, the chosen simulator can use information for timing simulation.

The fitter creates a JEDEC file, which is used to program the device on the board either using a parallel cable or programming equipment.

The steps of implementation must be carried out in this order. WebPACK ISE software will automatically perform the steps required if a particular step is selected. For example, if the design has only just been functionally simulated and you decide to do a timing simulation, the software will automatically synthesize, translate, and fit the design. It will then generate the timing information before it opens the simulator and gives the timing simulation results.

The rest of this chapter demonstrates the steps required to successfully implement our counter design.

Synthesis

The XST synthesis tool will only attempt to synthesize the file highlighted in the **Sources** window. In our design, "`top.vhd`" (for VHDL designs) or "`top_sch`" (for schematic designs) instantiates three lower level blocks, "`clock_divide`", "`counter`" and

"`display_drive`." The synthesis tool recognizes all the lower level blocks used in the top-level code and synthesizes them together to create a single bitstream.

1. In the **Sources** window, ensure that `top.vhd` is highlighted.

Note: If your top level is the schematic file `top_sch`, you will need to check the syntax of the lower level VHDL files. You cannot perform a syntax check on a schematic file.

2. In the **Process** window, expand the **Synthesis** subsection by clicking on the "**+**" next to **Synthesize**.
3. You can now check your design by double-clicking on **Check Syntax**. Ensure that any errors in your code are corrected before you continue. If the syntax check is **OK**, a check mark will appear.

The design should be okay because both the HDL Bencher tool and ISE Simulator have already checked for syntax errors. (It is useful, when writing code, to periodically check your design for any mistakes using this feature.)

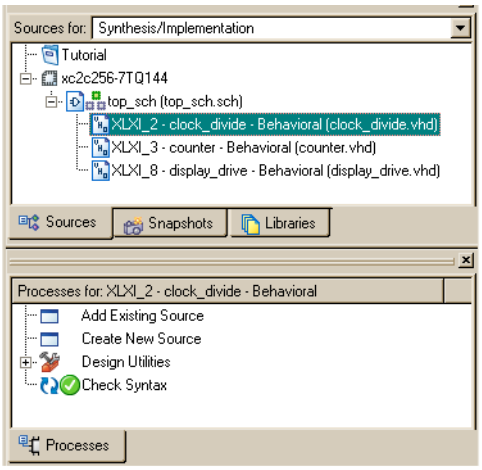


Figure 5-1: **Process Window Showing Check Syntax for clock_divide.vhd**

After you have checked all the modules, highlight the top level module, then go down to the process tree and right-click on **Synthesize** (under **Implement Design**) and select **Properties**.

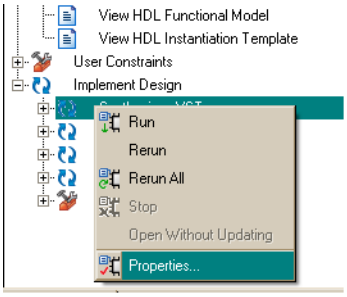


Figure 5-2: **Selecting Synthesis Properties**

A window appears allowing you to influence the way in which your design is interpreted. The **Help** feature will explain each of the options in each tab.

4. Highlight the **Xilinx Specific Options** category.

5. In the **Xilinx Specific Options** tab, ensure that the **Add IO Buffers** box is ticked. The I/O buffers will be attached to all the port names in the top-level entity of the design.

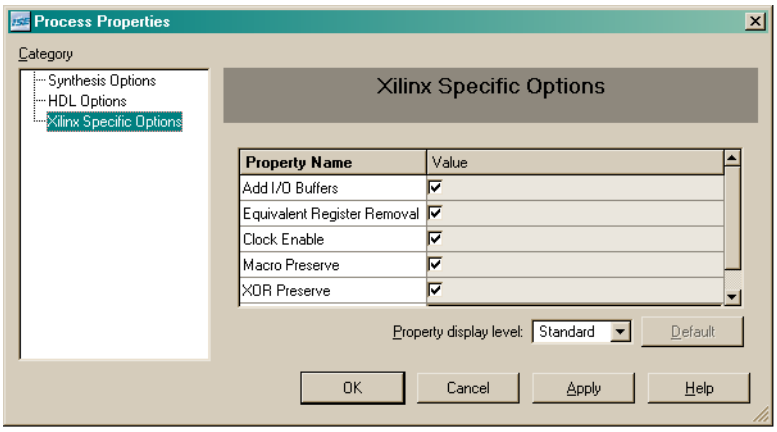


Figure 5-3: **Process Properties for Xilinx Specific Options**

Clicking on **Help** in each tab demonstrates the complex issue of synthesis and how the final result could change. The synthesis tool will never alter the function of the design, but it has a huge influence on how the design will perform in the targeted device.

6. Click **OK** in the **Process Properties** window and double-click on **Synthesize**.
7. When the synthesis is complete, a green tick will appear next to **Synthesize**. Double-click on **View Synthesis Report**. The report file (`.syr`) will appear in ISE.

Constraints Editor

To get the performance you need from a device, you must tell the implementation tools what and where performance is required. This design is particularly slow and timing constraints are unnecessary. Constraints can also be physical; pin-locking is a physical constraint.

In this example, we are going to download the design to the board from the CoolRunner-II Design Kit so the signals need to be assigned to the correct pins to be able to see the design running.

1. In the **Process** window, expand the **User Constraints** tree and double-click **Floorplan IO**. As there is no constraints file present in the design, the software will

create a constraints file and call it "`top.ucf`." It will be associated with your top level source file.

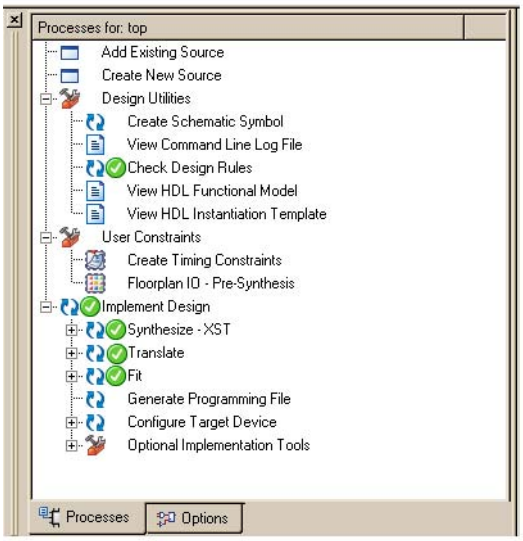


Figure 5-4: **Assign Package Pins**

Notice that the **Translate** step in the **Implement Design** section runs automatically. This is because the implementation stage must see the netlist before it can offer you the chance to constrain sections of the design. When translate has completed, the Xilinx PACE (Pinout Area and Constraints Editor) tool opens. If there are already constraints in the UCF file, these will be imported by PACE and displayed. As we have an empty UCF file, nothing exists for PACE to import. In the Design Object List, you can enter a variety of constraints on the I/O pins used in the design. PACE recognizes the five pins in the design and displays them in the list.

2. Click in the Loc area next to each signal and enter the following location constraints:

```
clock_ext        P38
direction_ext    P124
LED_output<0>    P56
LED_output<1>    P53
LED_output<2>    P60
LED_output<3>    P58
LED_output<4>    P57
LED_output<5>    P54
LED_output<6>    P61
pause_design     P39
reset_ext        P143
```

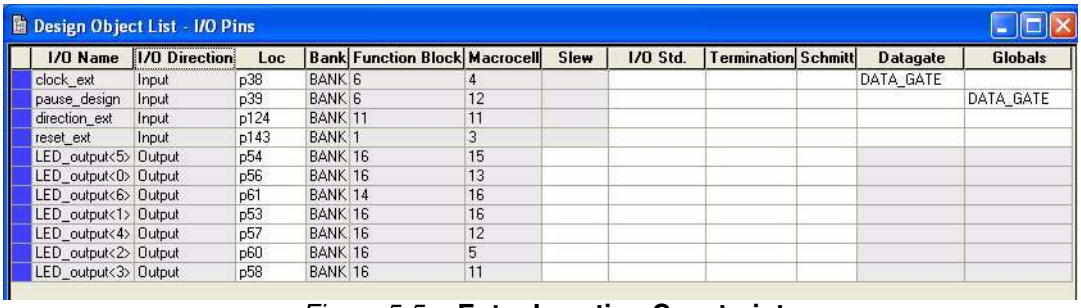
In addition, in the **Datagate** column for the signal `clock_ext`, select `DATA_GATE`. Also, for the signal `pause_design`, look under the **Globals** column and set it to `DATA_GATE`. This defines the control signal for all other inputs that have DataGATE selected. DataGATE is a feature in CoolRunner-II that enables input pins to be electrically isolated from the internal fabric of the device. The pin that turns this feature on and off is the DataGATE enable or DGE pin. On this device, the DGE pin is P39, which is mapped to slide switch

SW0. Moving SW0 will show DataGATE blocking the clock signal from the device, thereby pausing the design.

The `direction_ext` signal is also connected to one of the slide switches (in this case SW2) which is connected to P124 of the device. Therefore, changing the position of SW2 changes the direction in which the counter counts.

The LED output signals are all located to the pins of the 4 digit seven segment display.

Finally, the `reset_ext` signal is connected to BTN0 on the main board. This corresponds to P143.



I/O Name	I/O Direction	Loc	Bank	Function Block	Macrocell	Slew	I/O Std.	Termination	Schmitt	Datagate	Globals
clock_ext	Input	p38	BANK	6	4					DATA_GATE	
pause_design	Input	p39	BANK	6	12						DATA_GATE
direction_ext	Input	p124	BANK	11	11						
reset_ext	Input	p143	BANK	1	3						
LED_output<5>	Output	p54	BANK	16	15						
LED_output<0>	Output	p56	BANK	16	13						
LED_output<6>	Output	p61	BANK	14	16						
LED_output<1>	Output	p53	BANK	16	16						
LED_output<4>	Output	p57	BANK	16	12						
LED_output<2>	Output	p60	BANK	16	5						
LED_output<3>	Output	p58	BANK	16	11						

Figure 5-5: **Enter Location Constraints**

3. Save the PACE session and exit the PACE tool. It is now possible to see the constraints in the UCF file.
4. Select the `top.ucf` or `top_sch.ucf` file in the **Sources** window. The name of the UCF depends on if you are using a schematic or HDL top level file. This will display the processes available for a UCF file.
5. Now, under the **User Constraints** tab, double-click **Edit Constraints (Text)** in the **Process** window. The UCF file will open in the main window of the ISE Project Navigator.

   The constraints entered into PACE can be seen in Figure 5-6.

Note: If you do not see the UCF file in the **Sources** window, add it by using **Project** → **Add Source**.

```
1  #PACE: Start of Constraints generated by PACE
2
3  #PACE: Start of PACE I/O Pin Assignments
4  NET "clock_ext"  LOC = "p38" | DATA_GATE ;
5  NET "direction_ext"  LOC = "p124"  ;
6  NET "LED_output<0>"  LOC = "p56"  ;
7  NET "LED_output<1>"  LOC = "p53"  ;
8  NET "LED_output<2>"  LOC = "p60"  ;
9  NET "LED_output<3>"  LOC = "p58"  ;
10 NET "LED_output<4>"  LOC = "p57"  ;
11 NET "LED_output<5>"  LOC = "p54"  ;
12 NET "LED_output<6>"  LOC = "p61"  ;
13 NET "pause_design"  LOC = "p39" | BUFG = DATA_GATE ;
14 NET "reset_ext"  LOC = "p143"  ;
15
16 #PACE: Start of PACE Area Constraints
17
18 #PACE: Start of PACE Prohibit Constraints
19
20 #PACE: End of Constraints generated by PACE
21
```

Figure 5-6: **Text Constraints Imported from PACE**

CoolRunner-II architecture supports the use of non 50:50 duty cycle clocks by implementing input hysteresis. This can be selected on a pin-by-pin basis. For example, if the clock used in this design is an RC oscillator, the input hysteresis can be used to clean up the clock using the following constraint syntax:

```
NET "clock" schmitt_trigger;
```

The CoolRunner-II CPLD also supports different I/O standards. If signals had to go to a downstream device that required the signals to conform to a certain I/O standard, you could use the following constraint syntax:

```
NET "net_name" IOSTANDARD=LVTTL;
```

The permissible standards are LVTTL, LVCMOS15, LVCMOS18, LVCMOS25, LVCMOS33. On larger devices (128 macrocell and above), HSTL_I, SSTL2_I, and SSTL3_I are also included. However, you can use only one I/O standard **per bank**, so take care when assigning different I/O standards in a design.

The CoolRunner-II family has several features that are aimed at reducing power consumption in the device. One of these features is known as CoolClock. The clock signal on Global Clock Input 2 (GCK2) is divided by two as soon as it enters the device. All of the registers clocked by this clock are then automatically configured as dual-edge triggered flip-flops. The highest toggling net in the design will now be toggling at half the frequency, which will reduce the power consumption of that net without compromising the performance of the design. The CoolClock attribute can be applied by right-clicking on GCK2 in PACE or by adding the following line in the UCF:

```
NET "clock" COOL_CLK;
```

However, we will not use these features in this tutorial. For more information on the use of CoolRunner-II CPLDs, and their advanced features, visit the documentation section on *www.xilinx.com* for a number of application notes, often including free code examples.

You must re-run translate so the new constraints can be read.

6. Select the top-level file in the **Sources** window to display the implementation processes for the top-level design. This will be either `top.vhd` or `top_sch.sch`, depending on if you are using a HDL or schematic top-level file.
7. Click on the "**+**" next to **Implement Design** in the **Process** window.

   The implementation steps are now visible. An orange question mark indicates that translate is now out of date and should be re-run.
8. A right-click on **Implement Design** allows you to edit the properties for each particular step, as shown in Figure 5-7. Change the **I/O Voltage Standard** from

LVCMOS18 to **LVCMOS33**, and change the **Unused I/O Pad Termination Mode** to **Ground**.

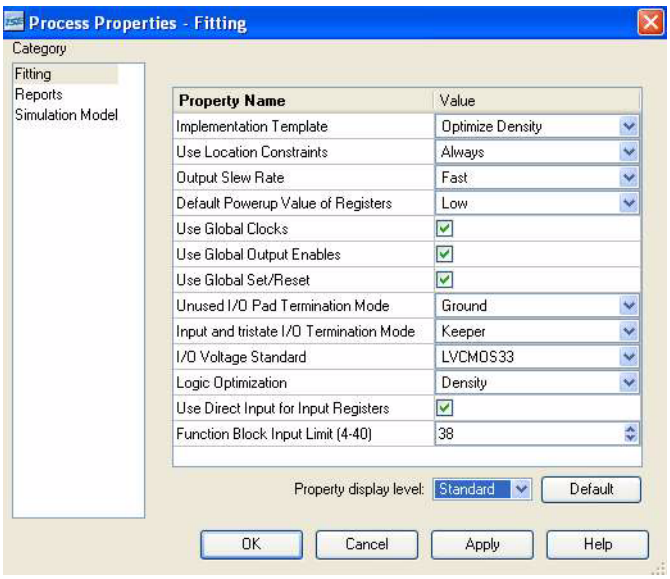


Figure 5-7: **PROCESS PROPERTIES – IMPLEMENT DESIGN**

The **Help** button will explain the operation of each field.

You can set the default I/O standard under the **Fitting** tab of the **Process Properties** window, as shown in Figure 5-7.

CPLD Reports

Two reports are available that detail the fitting results and associated timing of the design. These are:

- The Translation Report shows any errors in the design or the UCF.
- The CPLD Fitter Report can be opened in two ways, either in a standard text window within the ISE GUI or in a browser window.

1. To select which format to open, go to **Edit** → **Preferences**

2. Select **ISE General** in the **Category** box, then choose between **HTML** and **Text** under **CPLD reports**.

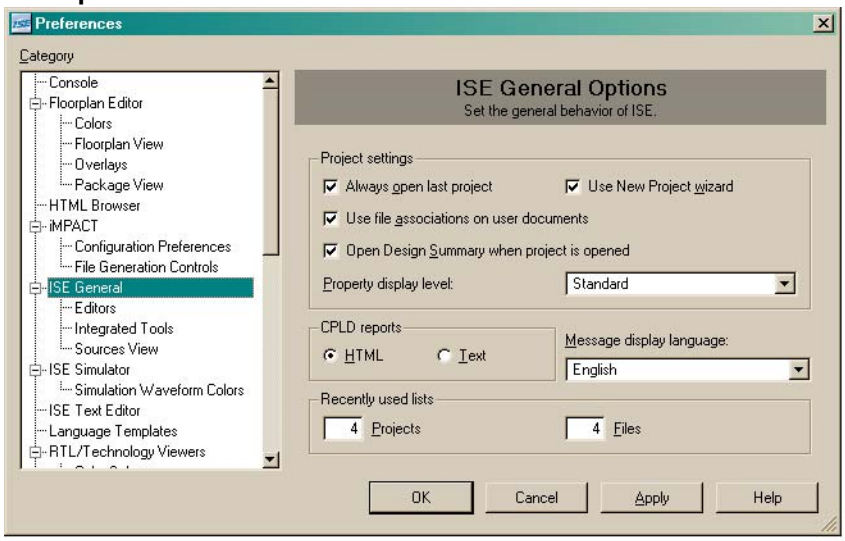


Figure 5-8: **ISE Preferences**

3. To open the **CPLD Fitter Report**, expand the **Fit** branch and double-click on the **Fitter Report** process.

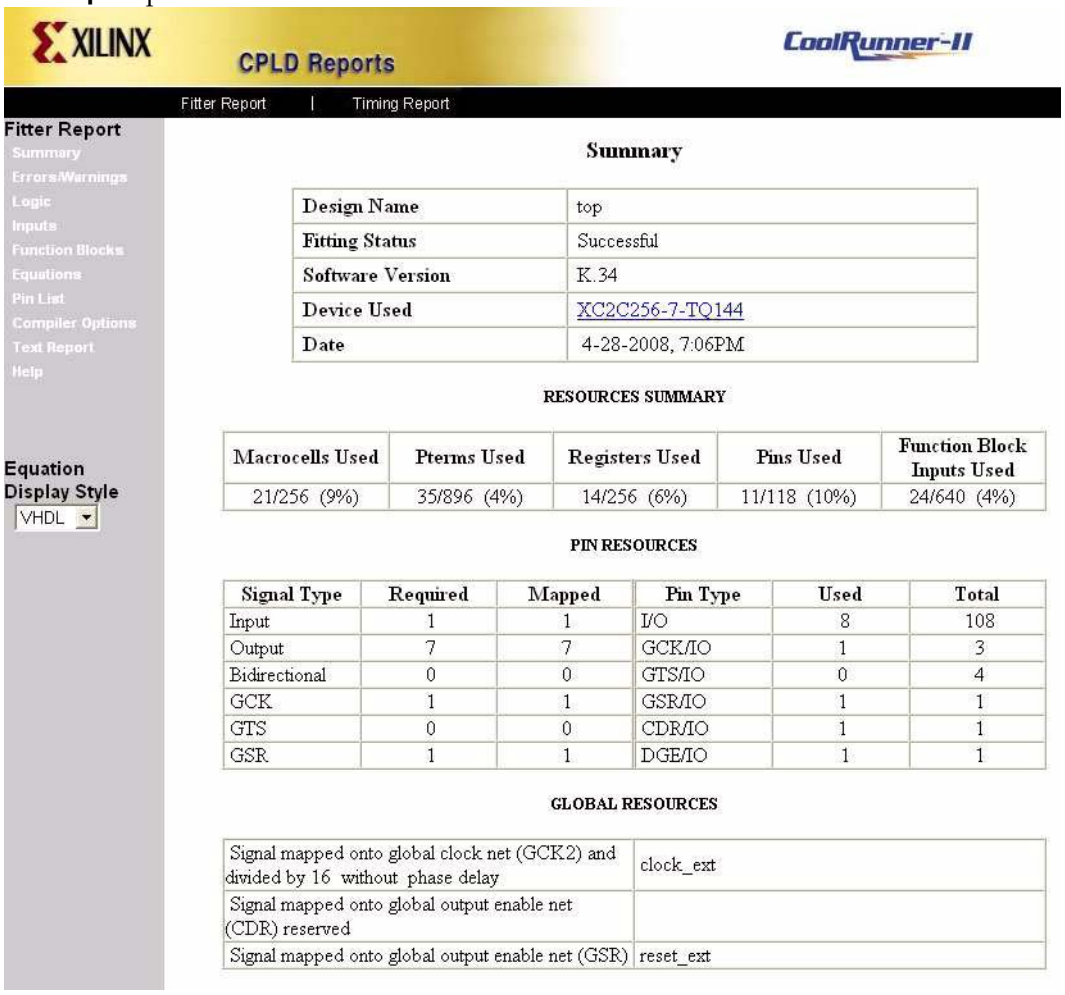
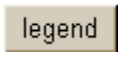


Figure 5-9: **CPLD HTML Fitter Report**

The same information is contained in both the HTML and text reports, but the HTML report has been designed to make the information more readable and easier to find. You can browse through several sections of the HTML Fitter Report by using the menu on the left-hand side of the page.

- The `Summary` section of the report gives a summary of the total resources available in the device (macrocells, I/O pins, etc.), and how much is used by the design.
- The errors and warnings generated during fitting can be seen in the `Errors/Warnings` section.
- The `Inputs` and `Logic` sections give information about signals, macrocells, and pins in the fitted design. The key to the meaning of the abbreviations is available by pressing the legend button.



- The `Function Blocks` summary looks into each function block and shows which macrocell is used to generate the signals on the external pins. By clicking on a specific

function block (e.g., FB1) in the `Function Blocks` section, all of the macrocells in that function block will be shown. Clicking on a specific macrocell will bring up a diagram of how that macrocell is configured.

The XC2C256 device has 16 function blocks, of which only two have been used for logic functions in this design. The design could be packed into a single function block, but the chosen I/O pins dictate which macrocells (and hence which function blocks) are used.

- A great feature of CPLDs is the deterministic timing, as a fixed delay exists per macrocell. The `Timing Report` is able to give the exact propagation delays and setup times and clock-to-out times. These values are displayed in the first section of the report.

  The next section lists the longest setup time, cycle time (logic delay between synchronous points as constrained by the period constraint), and clock-to-out time. The setup and clock-to-out times don't strictly affect the design's performance. These parameter limitations are dependent on the upstream and downstream devices on the board. The cycle time is the maximum period of the internal system clock.

  The next section shows all the inputs and outputs of the design and their timing relationship with the system clock. The clock to setup section details the internal nets to and from a synchronous point. The maximum delay in this section dictates the maximum system frequency.

  The last section details all the path type definitions, explaining the difference between the types mentioned previously in the report.

4. To generate a detailed timing report, right-click on **Generate Timing** in the **Process** window (under **Optional Implementation Tools**) and select **Properties** → **Timing Report Format**. Then, select **Detail** from the drop down menu.

Configuration with CoolRunner-II Utility Window

The CPLD Design Kit comes with a USB cable which can be used to configure the device. The kit also includes a software utility called "Configuration with CoolRunner-II Utility Window" that makes configuration easy. If you have a Xilinx USB cable, you can skip to the *Configuration Using iMPACT* section below, but first follow the board setup instructions.

The design does not require the LCD module, so it should be disconnected from the board.

1. Set the clock frequency to 100 kHz by removing the clock jumper on JP1.
2. Set slide switch SW1 to the left position, and SW0 to the right position.

3. First, you need to generate a configuration file for the device. Click the **Generate Programming File** process:

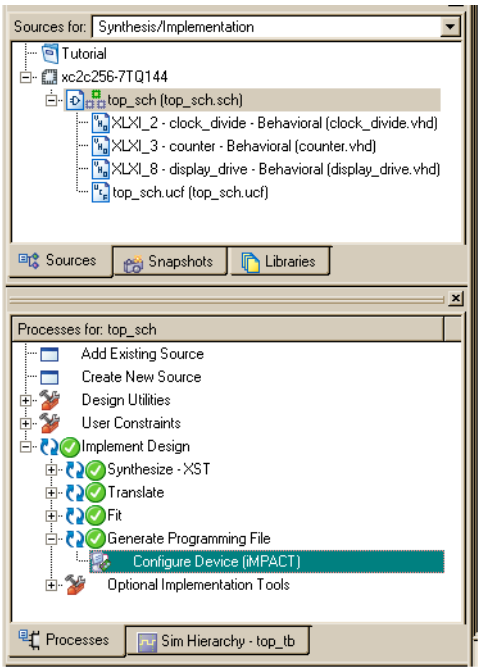


Figure 5-10: **Generate Programming File.**

If you are using the CoolRunner-II Starter Kit, and wish to use the USB cable provided, continue to the next step. If you have a Xilinx programming cable (not provided in the kit) and wish to use iMPACT for programming, jump to the section labeled "Configuration Using iMPACT".

4. Now that you have a programming file, you need to start the CoolRunner-II Utility Window. If you have not already done so, install this software from the Resource CD that came with the Starter Kit.
5. Ensure that the USB cable is plugged in to the computer and the board.

6. Start the **CoolRunner-II Utility Window** by selecting **Start** → **Programs** → **Digilent** → **Tools** → **CoolRunner-II Utility Window**.

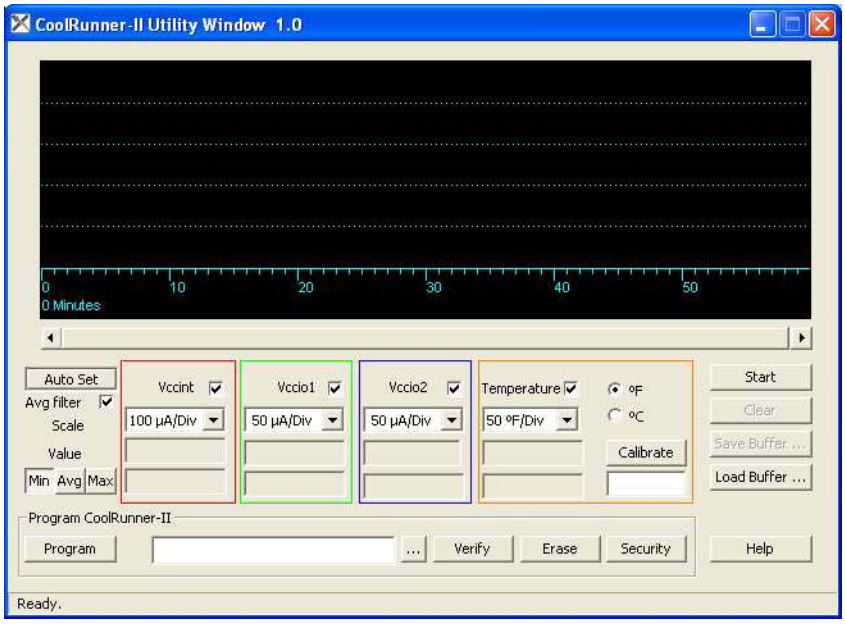


Figure 5-11: **CoolRunner-II Utility Window**

7. Click on the browse button (shown as **...** next to the program text box). Navigate to the location of the JEDEC programming file and select it. The JEDEC file will be in the directory that you have initially specified as the project directory. In this example it would be `c:\designs\tutorial`. Press the **Program** button to erase the CPLD and re-program it with the new design file.

The board will now be programmed with the design you created. It will count from 0 to F in Hex and then start again at 0. You can reset the design by pressing BTN0. You can change the direction of the count by moving slide switch SW1, and you can isolate the clock from the device, thereby pausing the design by moving SW1.

Configuration Using iMPACT

1. To configure using iMPACT, you must connect your Xilinx JTAG cable to the pins on J8. The board must be powered already either by the USB cable or by an external power source connected to JP3; it will not draw power from the Xilinx programming cable.
2. With **Sources for Synthesis and Implementation** displayed in the **Source** window, double-click on **Configure Target Device** in the **Process** window. You may have to click on the plus signs (**+**) next to **Implement Design** to see this option.
3. A dialog box labeled '**Welcome to iMPACT**' appears. Select **Configure devices using Boundary-Scan (JTAG)** and click **Finish**.
4. A dialog box labeled **Assign New Configuration File** appears. Browse to the target JEDEC file and click **Open**.
5. A dialog box labeled **Device Programming Properties** appears. Click **OK**.

6. A dialog box will appear with showing programming options. Select **Configure devices using Boundary-Scan (JTAG)** and click **Finish**. iMPACT will appear.

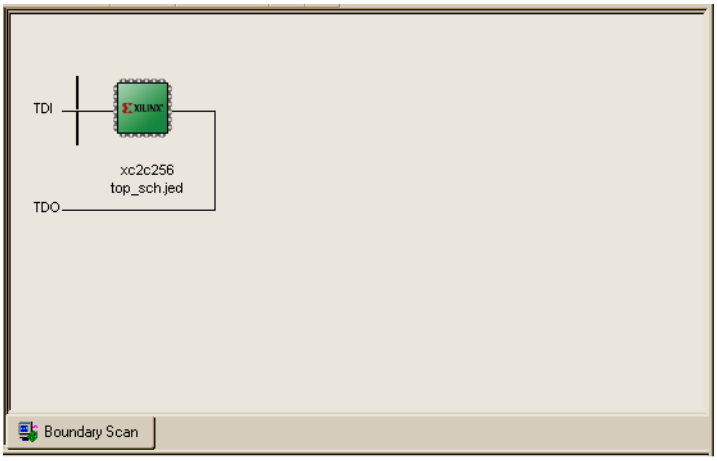


Figure 5-12: **iMPACT Programmer Main Window**

7. Right-click on the Xilinx XC2C256 icon that appears in the iMPACT window and select **Program**.

   The board will now be programmed with the design you created. It will count from 0 to F in Hex and then start again at 0. You can reset the design by pressing BTN0. You can change the direction of the count by moving slide switch SW4 and you can isolate the clock from the device, thereby pausing the design by moving SW1

Chapter 6

Implementing FPGA Designs

Introduction

After you have successfully simulated your design, the synthesis stage converts the code-based HDL or schematic-based design into an NGC netlist file. The netlist is a non-readable file that describes the actual circuit to be implemented at a very low level. The implementation phase uses the netlist and a constraints file to recreate the design using the available resources within the FPGA. Constraints may be physical or timing and are commonly used for setting the required frequency of the design or declaring the required pin-out.

The map stage distributes the design to the resources available in the FPGA. Obviously, if the design is too big for the specified device, mapping will be incomplete. The map stage also uses the UCF file to understand timing and may sometimes decide to add further logic (replication) to meet the given timing requirements. Map has the ability to "shuffle' the design around LUTs to create the best possible implementation for the design. The whole process is automatic and requires little user input.

The place and route stage works with the allocated CLBs and chooses the best location for each block. For a fast logic path, it makes sense to place relevant CLBs next to each other simply to minimize the path length. The routing resources are then allocated to each connection, again using a careful selection of the best possible routing types. For example, if you need a signal for many areas of the design, the place and route tool would use a "longline" to span the chip with minimal delay or skew.

At this point, it is good practice to re-simulate. As all of the logic delays added by the LUTs and flip-flops are now known (as well as the routing delays), the chosen simulator can use this information for timing simulation.

Finally, a program called "bitgen" takes the output of place and route and creates a programming bitstream. When developing a design, it may not be necessary to create a bit file on every implementation, as you may only need to ensure that a particular portion of your design passes timing verification. The steps of implementation must be carried out in this order:

1. Synthesize
2. Map
3. Place and Route
4. Timing Simulate
5. Program.

WebPACK ISE software will automatically perform the steps required if a particular step is selected. For example, if the design has only just been functionally simulated and you decide to do a timing simulation, the software will automatically synthesize and fit. It will

then generate the timing information before opening the simulator and giving timing simulation results.

In this chapter, we'll demonstrate the steps required to successfully implement our counter and decoder design into a Spartan-3E FPGA.

Changing the Project from CoolRunner-II to Spartan-3E

Double-click on "**xc2c256-7tq144**" in the **Sources** window, shown in Figure 6-1.

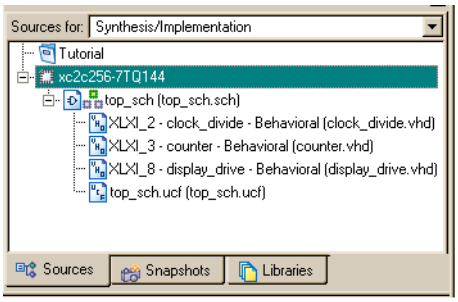


Figure 6-1: **Sources in Project Window**

The **Project Properties** dialog box will appear.

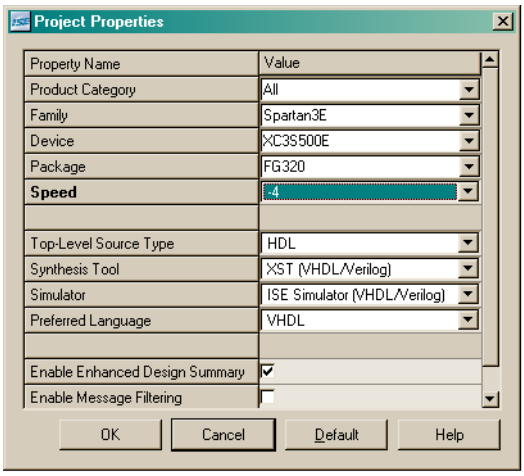


Figure 6-2: **Project Properties Dialog**

6. Enter the following characteristics as shown above:
   a. In the **Family** field, select **Spartan3E**
   b. In the **Device** field, select **xc3s500e**
   c. Change the **Package** field to **fg320**
   d. Set the **Speed** grade to **-4**
   e. Set **Top-Level Source Type** to **HDL**
   f. Set **Synthesis Tool** to X**ST (VHDL/Verilog)**
   g. Set **Simulator** to **ISE Simulator (VHDL/Verilog**)
   h. Click on **OK**.

The project, originally targeted at a CoolRunner-II CPLD, is now targeting a Xilinx Spartan-3E FPGA. The green ticks in the Process window should have disappeared and been replaced by orange question marks, indicating that the design must be re-synthesized and re-implemented.

Synthesis

The design as it stands from chapter 5 cannot be implemented directly into a Spartan-3E as it contains the CoolRunner-II hard clock divider which is present only in the CoolRunner-II family. Therefore, the `clock_divide` module must be edited to remove that block and replace its functionality with logic. The oscillator on the Spartan-3E board is 50 MHz so the clock divider must divide down more than in the CPLD design where the oscillator was 100 kHz.

1. Make the edits to the `clock_divide.vhd` module by removing the `CLK_DIV16` component and using a 23-bit counter. The new code will look like this:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
library UNISIM;
use UNISIM.VComponents.all;
entity clock_divide is
Port ( clock_osc : in STD_LOGIC;
clock_div : out STD_LOGIC);
end clock_divide;
architecture Behavioral of clock_divide is
signal count : std_logic_vector(22 downto
0):="00000000000000000000000";
begin
process (clock_osc)
begin
if clock_osc='1' and clock_osc'event then
count <= count + 1;
end if;
end process;
clock_div <= count(22);
end Behavioral;
```

2. Save and close the `clock_divide.vhd` file.
3. In the **Sources** window, ensure that "`top.vhd`" (or "`top_sch`" for schematic flows) is highlighted.
4. In the **Process** window, expand the **Synthesis** subsection by clicking on the plus sign (**+**) next to **Synthesize**. You can now check your design by double-clicking on **Check Syntax**.

Note: For schematic designs you will have to **Check Syntax** on the sub-modules.

Ensure that any errors in your code are corrected before you continue. If the syntax check is OK, a green check mark will appear (as shown in Figure 6-3).

Note: The design should be okay because both the Bencher and XSim have already checked for syntax errors. (It is useful, when writing code, to periodically check your design for any mistakes using this feature.)

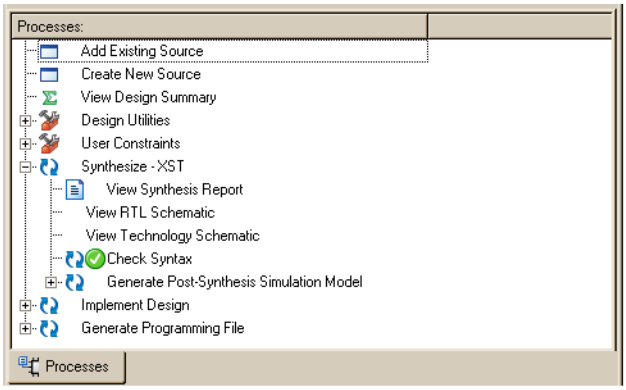


Figure 6-3: **Processes Window Showing Check Syntax Success**

The synthesis tool will never alter the function of the design, but it has a huge influence on how the design will perform in the targeted device.

Double-click **Synthesize – XST** to run the synthesis of the design. Then expand the synthesis tree in the process window and double-click **View Synthesis Report**. The first section of the report summarizes just the synthesis settings. Each entity in the design is then compiled and analyzed. The next section in the report gives synthesis details and documents how the design was interpreted.

Note: The state machine is one hot encoded, as each state name (red, amber, redamb, and green) has been assigned its own 1-bit register.

When synthesis chooses to use primitive macros it is known as “inference.” As registered outputs were selected in the state machine, three further registers were inferred.

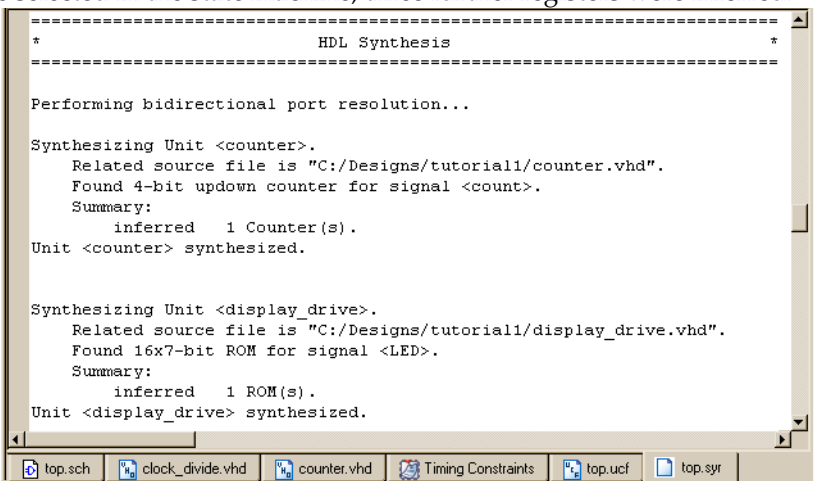


Figure 6-4: **Extract of Synthesis Report**

The `Final Report` section shows the resources used within the FPGA.

```
================================================================================
*                            Final Report                                      *
================================================================================
Final Results
RTL Top Level Output File Name     : top_sch.ngr
Top Level Output File Name         : top_sch
Output Format                      : NGC
Optimization Goal                  : Speed
Keep Hierarchy                     : NO

Design Statistics
# IOs                              : 10

Cell Usage :
# BELS                             : 81
#      GND                         : 1
#      INV                         : 2
#      LUT1                        : 22
#      LUT3                        : 1
#      LUT3_L                      : 1
#      LUT4                        : 9
#      MUXCY                       : 22
#      VCC                         : 1
#      XORCY                       : 22
# FlipFlops/Latches                : 27
#      FD                          : 23
#      FDC                         : 4
# Clock Buffers                    : 1
#      BUFGP                       : 1
# IO Buffers                       : 9
#      IBUF                        : 2
#      OBUF                        : 7
================================================================================
```

Figure 6-5: **Final Report**

The Constraints File

To get the ultimate performance from the device, you must tell the implementation tools what and where performance is required. This design is particularly slow and timing constraints are unnecessary. Constraints can also be physical; pin locking is a physical constraint. For this design, assume that the specification for clock frequency is 100 MHz and that the pin-out has been pre-determined to that of a Spartan-3 device.

There are already some constraints in the UCF from the previous project implementation. It will be necessary to delete these constraints before running implementation steps. This is because we set constraints that apply only to CPLDs, and set pin-outs for a CoolRunner-II XC2C256-TQ144.

1. Highlight "`top.ucf`" in the **Source** window. Expand the plus sign(**+**) next to **User Constraints** and double-click **Edit Constraints (Text)**.
2. Highlight all of the constraints (the entire file) and delete them. Save the UCF (**File** → **Save**).

3. Double-click on **Floorplan IO - Pre-Synthesis**. This is found under the **User Constraints** branch.

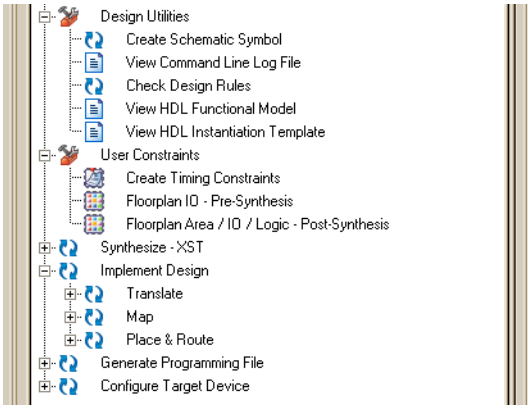
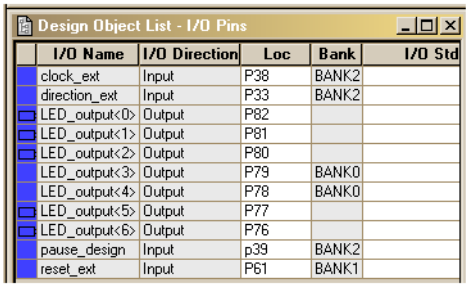


Figure 6-6: **Process Window Showing Assign Package Pins**

The PACE tool will be launched.

4. In PACE, assign all I/O pins in the **Design Object List** as shown in Figure 6-7. Simply place the cursor in the **Loc** column, click on the entry for each, and type in the values shown. If you have selected another package type, you can refer to the data sheet, or roll over the package pins in the PACE display to find a global clock pin for `clock`.



I/O Name	I/O Direction	Loc	Bank	I/O Std
clock_ext	Input	P38	BANK2	
direction_ext	Input	P33	BANK2	
LED_output<0>	Output	P82		
LED_output<1>	Output	P81		
LED_output<2>	Output	P80		
LED_output<3>	Output	P79	BANK0	
LED_output<4>	Output	P78	BANK0	
LED_output<5>	Output	P77		
LED_output<6>	Output	P76		
pause_design	Input	p39	BANK2	
reset_ext	Input	P61	BANK1	

Figure 6-7: **Design Object List**

5. Save (**File** → **Save**) and exit (**File** → **Exit**) the PACE session.

Note: You may encounter a dialog that asks you to define the bus delimiter. Select **XST Default: <>** or **XST Optional {}**.

6. Click on the plus sign "**+**" next to **Implement Design** in the **Process** window.

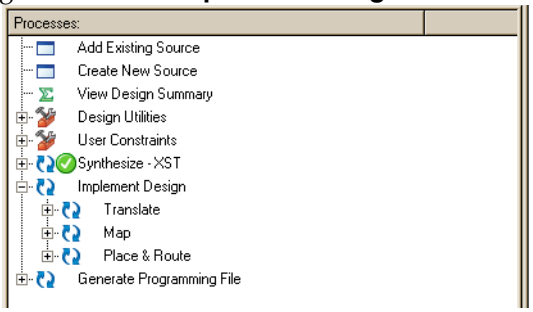


Figure 6-8: **Process Window Showing Implement Design**

7. Implement the design by double-clicking on **Implement Design** (you could run each stage separately).
8. When there is a green tick next to **Translate**, **Map**, and **Place and Route**, your design has completed the implementation stage.

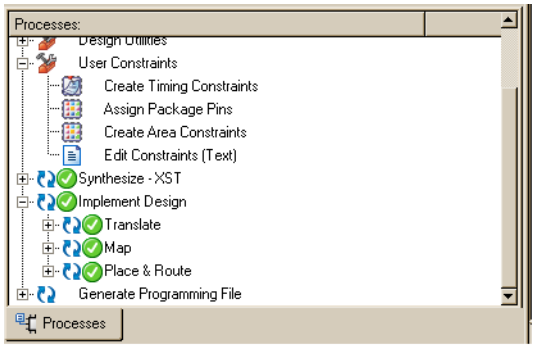


Figure 6-9: **Completed Implementation**

A green tick means that the design ran through without any warnings. A yellow exclamation point may mean that there is a warning in one of the reports. If you've followed the design procedure outlined in this example, there should be no errors or warnings.

FPGA Reports

Each stage has its own report. Clicking on the "**+**" next to each stage lists the reports available:

- The `Translate Report` shows any errors in the design or the UCF.
- The `Map Report` confirms the resources used within the device and describes trimmed and merged logic. It will also describe exactly where each portion of the design is located in the actual device. A detailed `Map Report` can be chosen in the properties for map.
- The `Post-Map Static Timing Report` shows the logic delays only (no routing) covered by the timing constraints. This design has two timing constraints, the clock period and the clock-to-out time of the three lights. If the logic-only delays don't meet timing constraints, the additional delay added by routing will only add to the problem. Without a routing delay, these designs would run at 200 MHz!
- The `Place and Route Report` gives a step-by-step progress report. The place and route tool must be aware of timing requirements. It will list the given constraints and report how comfortably the design fell within – or how much it failed – the constraints.
- The `Asynchronous Delay Report` is concerned with the worst path delays in the design – both logic and routing.
- The `Pad Report` displays the final pin-out of the design, with information regarding the drive strength and signalling standard.
- The `Guide Report` shows how well a guide file has been met (if one was specified).
- The `Post Place and Route Static Timing Report` adds the routing delays. Notice that the max frequency of the clock has dropped.

WebPACK ISE software has additional tools for complex timing analysis and floor planning, which are beyond the scope of this introductory book. To find out more about advanced software tools, visit the software documentation page at our website: http://www.xilinx.com/support/software_manuals.htm

Programming

Ensure that the Spartan-3E Design Kit board is powered and switch SWP set to the On position.

A DLC9 Platform USB cable is required to configure the device from the iMPACT Programmer. Ensure that the cable is plugged in to the computer and that the ribbon cable/flying leads are connected properly to the board.

To program a Spartan-3E:

1. Right-click on **Generate Programming File** and click on **Properties**.
2. Under the **Start-Up Options** category, ensure that the **FPGA Start-Up Clock** is set to **JTAG Clock** by selecting JTAG Clock from the drop-down menu. Click **OK**

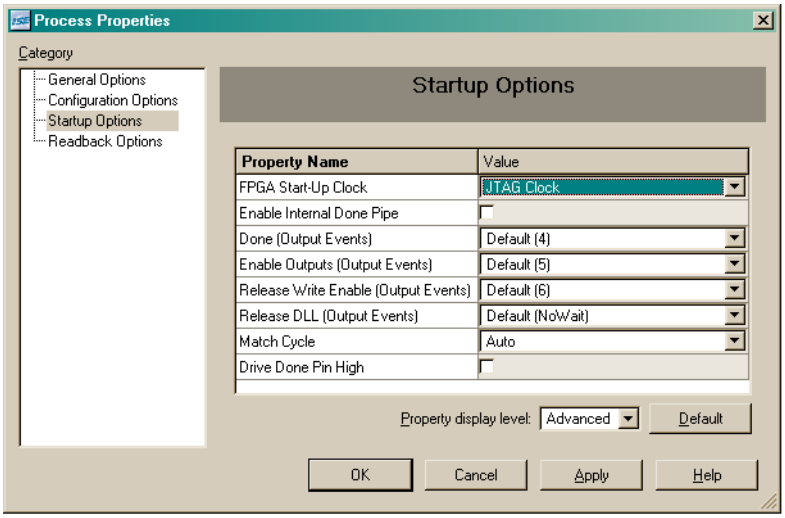


Figure 6-10: **JTAG Clock Selection**

3. Double-click on **Generate Programming File**.

   This operation creates a `.bit` file that can be used by the iMPACT programmer to configure a device.
4. Expand the **Generate Programming File** tools subsection.
5. Double-click on **Configure Device (iMPACT)**.
6. Click **Finish** on the "**Welcome to iMPACT**" screen.
7. The JTAG chain will automatically be recognized. Associate the `top.bit` file with the Spartan-3E device and put the PROM and the CPLD into bypass mode by clicking **Bypass** in the bottom right corner of the window.
8. Click on the picture of the Spartan-3E device.
9. From the Operations Menu, select **Program**. Click **OK** to program the device.

The design created is now in the Spartan-3E, and, if you have attached the seven segment displays to J1 and J2, you will see the counter counting up from 0 to F in Hex. Changing the position of switch SW0 will reset the counter back down to zero, changing the position of switch SW1 will change the direction from counting up to counting down.

Summary

This chapter has taken the VHDL or Schematic design through to a working physical device. The steps discussed were:

- Synthesis and Synthesis Report
- Timing and Physical Constraints using the Constraints Editor
- The Reports Generated throughout the Implementation flow
- Creating and Downloading a bitstream.

Chapter 7

Application Notes, Reference Designs, IP, and Services

Introduction

Our final chapter contains a useful resources that will give you a good jump start into your future programmable logic designs. Xilinx has a large inventory of free Application Notes and associated design files to facilitate the creation of complex designs. In addition, we have an extensive library of Intellectual Property, some of which must be purchased. In the pages that follow, we will outline current inventory applications.

CPLD Reference Designs

CPLD reference designs are HDL code-based designs that can help reduce the time of CPLD designs. They are all available free of charge. These reference designs take the form of IP, which can be used as is. Unlike purchased IP, these reference designs do not come with direct support. They are built around application notes and have been tested in WebPACK software. They are fully functional through the WebPACK simulator and testbench. You can find CoolRunner reference designs in the Xilinx IP Center **http://www.xilinx.com/ipcenter/** by searching on the keyword "CoolRunner."

Table 7-1 details the current reference designs. These are continually updated, so check regularly for new listings. The URL is:

http://www.origin.xilinx.com/products/silicon_solutions/cplds/resources/coolvhdlq.htm?url=/products/xaw/coolvhdlq.htm

Table 7-1: **Current Reference Designs**

Memory XAPP800: SPI Flash XAPP394: CoolRunner-II Mobile SDRAM Interface XAPP384: CoolRunner-II DDR SDRAM Interface XAPP354: NAND Interface (ABEL) XAPP354: NAND Interface (VHDL) XAPP354: NAND Interface (Verilog) XAPP906: SD Card Multiplexer XAPP944: Data Stream Switch	**PDA** XAPP357: LED Test XAPP356: XPATH Handspring Design Module (VHDL & Pocket C) XAPP355: Serial ADC Interface XAPP149: Oscilloscope XAPP147: Low Power Design XAPP146: 8 Channel Digital Volt Meter
Microcontroller XAPP432: LIN Controller Implementation XAPP432: LIN I/O Module XAPP393: 8051 Microcontroller Interface for CoolRunner -II XAPP387: PicoBlaze Microcontroller XAPP349: 8051 Microcontroller Interface for CoolRunner XPLA3	**Wireless** XAPP358: Wireless Transceiver Customer Pack XAPP345: IrDA & UART Design (VHDL) XAPP345: IrDA & UART Design (Verilog)
CoolRunner-II XAPP390: Digital Camera Design XAPP378: CoolRunner-II Example Designs XAPP381: CoolRunner-II Demo Board XAPP444: CPLD Fitting, Tips and Tricks XAPP940: Stepper Motor Controller	**Datacom** XAPP380: N x N Digital Crosspoint Switch XAPP383: SECDED XAPP372: Smart Card Reader XAPP328: MP3 for XPLA3

Table 7-1: **Current Reference Designs**

Bus Interface	Security
XAPP398: Compact Flash	XAPP374: CryptoBlaze
XAPP391: 8b/10b Encoder-Decoder for CoolRunner-II	XAPP371: Galois Field GF(2^m) Multiplier
XAPP391: 16b/20b Encoder-Decoder for CoolRunner-II	
XAPP386: CoolRunner-II Serial Peripheral Interface Master	
XAPP353: SMBus Controller (VHDL)	
XAPP348: Serial Peripheral Interface Master	
XAPP341: UART (VHDL)	
XAPP341: UART (Verilog)	
XAPP339: Manchester Encoder-Decoder (Verilog)	
XAPP339: Manchester Encoder-Decoder (VHDL)	
XAPP336: 8b/10b Encoder-Decoder for CoolRunner XPLA3	
XAPP336: 16b/20b Encoder-Decoder for CoolRunner XPLA3	
XAPP333/ XAPP385: I^2C Controller (VHDL)	
XAPP333/ XAPP385: I2C Controller (Verilog)	

CoolRunner-II Application Examples

The flexibility of the CoolRunner-II architecture means that it can be used for an infinite number of different applications. CoolRunner-II CPLDs create standard logic (gates and flip flops) efficiently. Using design software, the architecture efficiently creates just the required logic to solve the problem. Unused logic remains available for future modifications, corrections or enhancement. Xilinx design software shuts down unused circuits eliminating their participation in the power consumption budget. By using gates and flip flops to build logic, CoolRunner-II CPLDs create a rich variety of diverse protocols, modulations and interfaces. If the task can be solved with logic, CoolRunner-II

CPLDs can deliver. For example, Figure 7-1 illustrates some of the applications we have prepared for portable consumer applications, such as cell phones and PDAs.

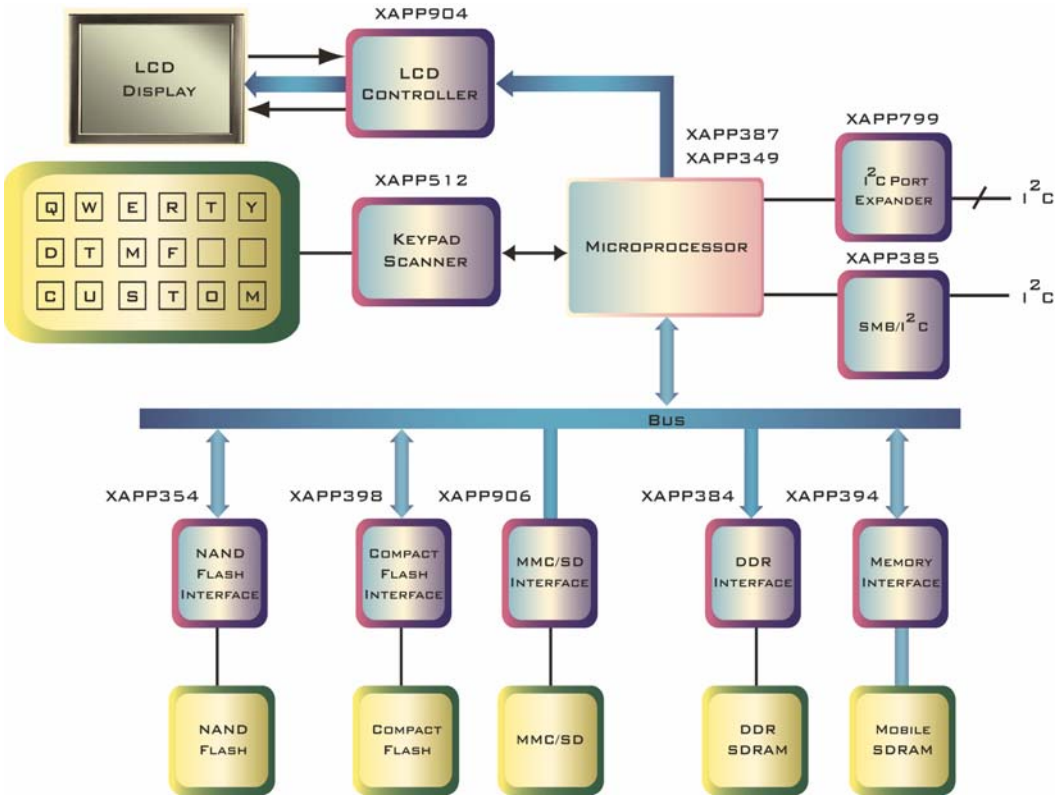


Figure 7-1: **Portable Consumer Application Notes**

For the same market we also supply application notes for using the inherent low power characteristics and features of the CoolRunner CPLD .

Get the Most out of Microcontroller-Based Designs

Microcontrollers don't make the world go round, but they most certainly help us get around in the world. You can find microcontrollers in automobiles, microwave ovens, automatic teller machines, VCRs, point-of-sale terminals, robotic devices, wireless telephones, home security systems, and satellites, to name just a few applications.

In the never-ending quest for faster, better, and cheaper products, advanced designers are now pairing CPLDs with microcontrollers to take advantage of the strengths of each. Microcontrollers are naturally good at sequential processes and computationally intensive tasks, as well as a host of non-time-critical tasks. CPLDs such as Xilinx CoolRunner devices are ideal for parallel processing, high-speed operations, and applications where lots of inputs and outputs are required.

Although faster and more powerful microcontrollers do exist, 8-bit microcontrollers own much of the market because of their low cost and low power characteristics. The typical operational speed is around 20 MHz, but some microcontroller cores divide clock frequency internally and use multiple clock cycles per instruction (operations often include fetch-and-execute instruction cycles). Thus, with a clock division of 2 – with each

instruction taking as long as three cycles – the actual speed of a 20 MHz microcontroller is divided by 6. This works out to an operational speed of only 3.33 MHz.

CoolRunner CPLDs are much, much faster than microcontrollers and can easily reach system speeds in excess of 100 MHz. Today, we are even seeing CoolRunner devices with input-to-output delays as short as 3.5 ns, which equates to impressive system speeds as fast as 285 MHz. CoolRunner CPLDs make ideal partners for microcontrollers, because they not only can perform high-speed tasks, they can perform those tasks with ultra- low power consumption.

Xilinx offers free software and low-cost hardware design tools to support CPLD integration with microcontrollers. The Xilinx CPLD design process is quite similar to that used on microcontrollers, you can quickly learn how to partition your designs across a CPLD and microcontroller to maximum advantage.

Design Partitioning

As we noted before, microcontrollers are very good at computational tasks, and CPLDs are excellent in high-speed systems, with their abundance of I/Os. Table 7-2shows how we can use a microcontroller and a CPLD in a partitioned design to achieve the greatest control over a stepper motor.

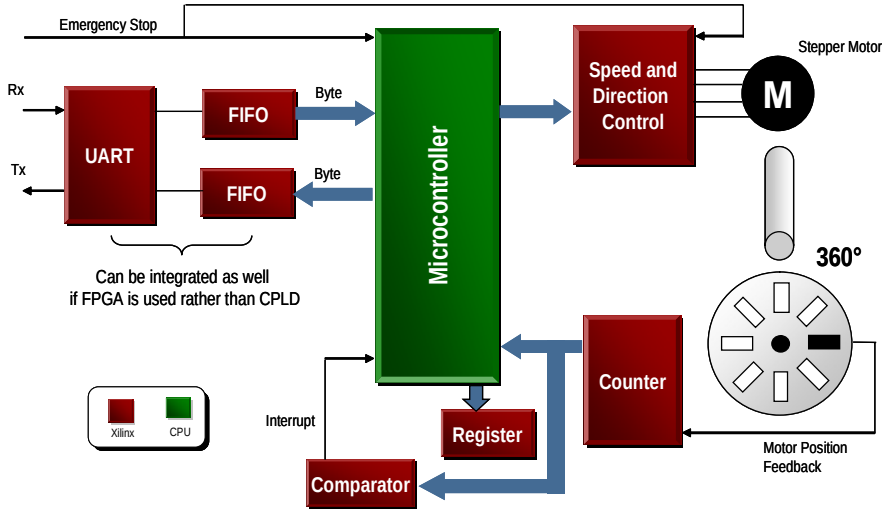


Figure 7-2: **Partitioned Design: Microcontroller and CPLD**

Meanwhile, the UART and FIFO sections of the design can be implemented in the microcontroller in the form of a microcontroller peripheral, or implemented in a larger, more granular PLD such as an FPGA – for example, a Xilinx Spartan device. Using a PLD in this design has the added benefit of gaining the ability to absorb any other discrete logic elements on the PCB or in the total design into the CPLD. Under this new configuration, we can consider the CPLD as offering hardware-based subroutines or as a mini co-processor.

The microcontroller still performs ASCII string manipulation and mathematical functions, but it now has more time to perform these operations – without interruption. The motor control is now independently stable and safe.

In low-power applications, microcontrollers are universally accepted as low-power devices and have been the automatic choice of designers. The CoolRunner family of ultra-low power CPLDs are an ideal fit in this arena and may be used to complement your low-power microcontroller to integrate designs in battery-powered, portable designs (<100 uA current consumption at standby).

Documentation and Example Code

Xilinx offer many Application notes, White Papers and User Guides to illustrate how CPLDs and FPGAs can be used. Some of these take the shape "how-to" guides that explain to the reader how to use a certain features of the architecture. Others show how to implement popular designs in CPLDs or FPGAs, and offer free code examples.

There are too many application notes to list them all here (more than 100 for CPLDs alone) but, as a sample, a few of the popular applications are listed below to illustrate the diverse range available. This list grows longer as more applications are developed. For the latest list, please visit www.xilinx.com → ***Documentation*** → ***Application Notes***.

Table 7-2: **Documentation List**

Title	Number	Family
Interfacing to DDR SDRAM with CoolRunner-II CPLDs	384	CoolRunner-II
CoolRunner-II I2C Bus Controller Implementation	385	CoolRunner-II
Compact Flash Card Interface for CoolRunner-II CPLDs	398	CoolRunner-II
CoolRunner-II Character LCD Module Interface	904	CoolRunner-II
DDR2 SDRAM Memory Interface for Spartan-3 FPGAs	454	Spartan-3
Using IP Cores in Spartan-3 FPGAs	474	Spartan-3
644MHz SDR LVDS Transmitter and Receiver	622	Virtex-II/ Spartan-3
HDTV Video Pattern Generator	682	Virtex-II/ Spartan-3
UltraController-II: Minimal footprint embedded processing engine	575	Virtex-4/ Virtex-II Pro
Virtex-4 High Speed Single Data Rate LVDS Transceiver	704	Virtex-4
Multiple Bit Error Correction	715	Virtex-4
Memory Interface Application Note Overview	802	Virtex-4/ Spartan-3

Intellectual Property (IP) Cores

IP cores are very complex pre-tested system-level functions that are used in logic designs to dramatically shorten development time. The benefits of using an IP core include:

- Faster time to market
- Simplified development process
- Minimal design risk
- Reduced software compile time
- Reduced verification time

- Predictable performance/functionality.

IP cores are similar to vendor-provided soft macros in that they simplify the design specification procedure by removing designers from gate-level details of commonly used functions. IP cores differ from soft macros in that they are generally much larger system-level functions, such as a PCI bus interface, DSP filter, or PCMCIA interface. They are extensively tested (and hence rarely free of charge) to prevent designers from having to verify the IP core functions themselves.

The Xilinx website has a comprehensive database of Xilinx LogiCORE and third-party AllianceCORE verified and tested cores. To find them, visit the Xilinx IP Center at www.xilinx.com/ipcenter. The CORE Generator tool from Xilinx delivers highly optimized cores compatible with standard design methodologies for Xilinx FPGAs. This easy-to-use tool generates flexible, high-performance cores with a high degree of predictability. You can also download future core offerings from the Xilinx website. The CORE Generator tool is provided as part of Xilinx Foundation ISE software.

End Markets

The eSP web portal is located within the "End Markets" section on the Xilinx website. It is the industry's first web portal dedicated to providing comprehensive solutions that accelerate product development. To make it as easy as possible, we provide a choice for locating material:

Aerospace/Defense	Industrial/Scientific/Medical
Automotive	Test/Measurement
Consumer	Wired Communications
Audio/Video/Broadcast	Wireless Communications

To get there, visit: http://www.xilinx.com/esp/

You can select a specific market solution or a broad-reaching technology that interests you and see what Xilinx can offer in that application area. The site was designed to decrease the time spent in the pre-design phase. This phase, which increasingly has become the designers' Achilles' heel, involves visiting seminars, learning new standards, assimilating the data, analyzing market trends, and more.

The eSP web portal saves time by proving up-to-date information about emerging standards and protocols, how and where they are used, impartial information about which one is best for your application, and pre-tested reference designs that can be purchased and used.

Xilinx Design Services

Xilinx Design Services, or XDS, combines skills and experience in system, logic, and embedded software design to provide a unique development partner. Leveraging these skill sets will help you optimize your budget, schedule, and performance requirements.

The XDS portfolio includes:

- IP Core Modification
  - Modifications or integration of existing Xilinx LogiCORE products and drivers
  - Quick access to LogiCORE source code and a team with experience and expertise with LogiCORE products

- Fixed bid/fixed price contracts

- FPGA Design From Specification
  - Turnkey FPGA design, ASIC conversions, and driver development and integration
  - Expertise in optimizing Xilinx technology to provide the best solution
  - Excellent project management to ensure on-time and correct deliveries
  - Fixed bid/fixed price contracts
- FPGA System Design
  - System architecture consulting, FPGA logic, and embedded software design
  - Broad applications experience and immense depth of experience with Xilinx embedded processing tools and products
  - Fixed bid/fixed price contracts
- Embedded Software Design
  - Complex embedded software designs with real-time constraints, driver development, and integration with hardware
  - Experience with FPGA platform design, including processors and gates
  - Expertise in hardware/software co-design techniques
  - Fixed bid/fixed price contracts

Overall, XDS offers:

- Professional project management
- System-level experience around the world
- Faster project ramp-up
- Experienced FPGA design engineers
- FPGA hardware and software experts
- Accelerated knowledge of FPGA systems
- Access to ready-made intellectual property cores.

To find out more, please visit the XDS home at:

www.xilinx.com → **Products and Services** → **Xilinx Design Services**

Lserves to make training available when you need it and for the products you need. Classes are held in centers around the world, although specific onsite instruction is also available. For more information, visit *www.support.xilinx.com* and click on "**Courses**" under "**Education**."

Design Consultants

The Xilinx XPERTS Program qualifies, develops, and supports design consultants, ensuring that they have superior design skills and the ability to work successfully with customers. XPERTS is a worldwide program that allows easy access to certified experts in Xilinx device architectures, software tools, and cores.

XPERTS partners also offer consulting in the areas of HDL synthesis and verification, customization and integration, system-level designs, and team- based design techniques. A listing of partners in the Xilinx XPERTS program is located at www.xilinx.com/ipecenter.

Technical Support

Xilinx provides 24-hour access to a set of sophisticated tools for resolving technical issues via the Web. The Xilinx search utility scans through thousands of answer records to return solutions for the given issue. Several problem-solver tools are also available for assistance in specific areas, like configuration or install. A complete suite of one-hour modules is also available at the desktop via live or recorded e-learning.

Lastly, if you have a valid service contract, you can access Xilinx engineers over the Web by opening a case against a specific issue. For technical support, visit www.support.xilinx.com.

Glossary

Glossary of Terms

ABEL – Advanced Boolean Expression Language, low-level language for design entry, from Data I/O.

AIM – Advanced Interconnect Matrix in the CoolRunner-II CPLD that provides the flexible interconnection between the PLA function blocks.

Antifuse – A small circuit element that can be irreversibly changed from being non-conducting to being conducting with ~100 Ohm. Anti-fuse-based FPGAs are thus non-volatile and can be programmed only once (see OTP).

AQL – Acceptable Quality Level. The relative number of devices, expressed in parts-per-million (ppm), that might not meet specification or be defective. Typical values are around 10 ppm.

ASIC – Application Specific Integrated Circuit, also called a gate array. Asynchronous logic that is not synchronized by a clock. Asynchronous designs can be faster than synchronous ones, but are more sensitive to parametric changes and are thus less robust.

ASSP – Application-Specific Standard Product. Type of high-integration chip or chipset ASIC that is designed for a common yet specific application.

ATM – Asynchronous Transfer Mode. A very high-speed (megahertz to gigahertz) connection-oriented bit-serial protocol for transmitting data and real-time voice and video in fixed-length packets (48-byte payload, 5-byte header).

Back Annotation – Automatically attaching timing values to the entered design format after the design has been placed and routed in an FPGA.

Behavioral Language – Top-down description from an even higher level than VHDL.

Block RAM – A block of 2k to 4k bits of RAM inside an FPGA. Dual-port and synchronous operation are desirable.

CAD – Computer Aided Design, using computers to design products.

CAE – Computer Aided Engineering, analyses designs created on a computer.

CLB – Configurable Logic Block. Xilinx-specific name for a block of logic surrounded by routing resources. A CLB contains two or four LUTs (function generators) plus two or four flip-flops.

CMOS – Complementary Metal-Oxide-Silicon. Dominant technology for logic and memory. Has replaced the older bipolar TTL technology in most applications (except very fast ones). CMOS offers lower power consumption and smaller chip sizes compared to bipolar and now meets or even beats TTL speed.

Compiler – software that converts a higher language description into a lower-level representation. For FPGAs: the complete partition, place and route process.

Configuration – The internally stored file that controls the FPGA so that it performs the desired logic function. Also, the act of loading an FPGA with that file.

Constraints – Performance requirements imposed on the design, usually in the form of max allowable delay, or required operating frequency.

CoolCLOCK – Combination of the clock divider and clock doubler functions in CoolRunner-II CPLDs to further reduce power consumption associated with high-speed clocked-in internal device networks.

CPLD – Complex Programmable Logic Device, synonymous with EPLD. PAL-derived programmable logic devices that implement logic as sum-of- products driving macrocells. CPLDs are known to have short pin-to-pin delays, and can accept wide inputs, but have relatively high power consumption and fewer flip-flops compared to FPGAs.

CUPL – Compiler Universal for Programmable Logic, CPLD development tool available from Logical Devices.

DataGATE – A function within CoolRunner-II devices to block free-running input signals, effectively blocking controlled switching signals so they do not drive internal chip capacitances to further reduce power consumption. Can be selected on all inputs.

Input Hysteresis – Input hysteresis provides designers with a tool to minimize external components, whether using the inputs to create a simple clock source or reducing the need for external buffers to sharpen a slow or noisy input signal. Function found in CoolRunner-II CPLDs (may also be referred to as Schmitt Trigger inputs in the text).

DCM – Digital Clock Manager. Provides zero-delay clock buffering, precise phase control, and precise frequency generation on Xilinx Virtex-II FPGAs.

DCI – Digitally Controlled Impedance in the Virtex-II solution dynamically eliminates drive strength variation due to process, temperature, and voltage fluctuation. DCI uses two external high-precision resistors to incorporate equivalent input and output impedance internally for hundreds of I/O pins.

Debugging – The process of finding and eliminating functional errors in software and hardware.

Density – Amount of logic in a device, often used to mean capacity. Usually measured in gates, but for FPGAs, better expressed in logic cells, each consisting of a 4-input LUT and a flip-flop.

DLL – Delay Locked Loop, A digital circuit used to perform clock management functions on- and off-chip.

DRAM – Dynamic Random Access Memory. A low-cost/read-write memory where data is stored on capacitors and must be refreshed periodically. DRAMs are usually addressed by a sequence of two addresses – row address and column address – which makes them slower and more difficult to use than SRAMs.

DSP – Digital Signal Processing. The manipulation of analog data that has been sampled and converted into a digital representation. Examples are filtering, convolution, and Fast Fourier Transform

EAB – Embedded Array Block. Altera™ name for block RAM in FLEX10K.

EDIF – Electronic Data Interchange Format. Industry-standard for specifying a logic design in text (ASCII) form.

EPLD – Erasable Programmable Logic Devices, synonymous with CPLDs. PAL-derived programmable logic devices that implement logic as sum-of- products driving macrocells. EPLDs are known to have short pin-to-pin delays, and can accept wide inputs, but have relatively high power consumption and fewer flip-flops than FPGAs.

Embedded RAM – Read-write memory stored inside a logic device. Avoids the delay and additional connections of an external RAM.

ESD – Electro-Static Discharge. High-voltage discharge can rupture the input transistor gate oxide. ESD-protection diodes divert the current to the supply leads.

5-Volt Tolerant – Characteristic of the input or I/O pin of a 3.3V device that allows this pin to be driven to 5V without any excessive input current or device breakdown. Very desirable feature.

FIFO – First-In-First-Out memory, where data is stored in the incoming sequence and is read out in the same sequence. Input and output can be asynchronous to each other. A FIFO needs no external addresses, although all modern FIFOs are implemented internally with RAMs driven by circular read and write counters.

FIT – Failure In Time. Describes the number of device failures statistically expected for a certain number of device-hours. Expressed as failures per one billion device hours. Device temperature must be specified. MTBF can be calculated from FIT.

Flash – Non-volatile programmable technology, an alternative to Electrically-Erasable Programmable Read-Only Memory (EEPROM) technology. The memory content can be erased by an electrical signal. This allows in-system programmability and eliminates the need for ultraviolet light and quartz windows in the package.

Flip-Flop – Single-bit storage cell that samples its Data input at the active (rising or falling) clock edge, and then presents the new state on its Q output after that clock edge, holding it there until after the next active clock edge.

Floorplanning – Method of manually assigning specific parts of the design to specific chip locations. Can achieve faster compilation, better utilization, and higher performance.

Footprint – The printed circuit pattern that accepts a device and connects its pins appropriately. Footprint-compatible devices can be interchanged without modifying the PC board.

FPGA – Field Programmable Gate Array. An integrated circuit that contains configurable (programmable) logic blocks and configurable (programmable) interconnect between those blocks.

Function Generator – Also called look-up-table, with N-inputs and one output. Can implement any logic function of its N-inputs. N is between 2 and 6; 4-input function generators are most popular.

GAL – Generic Array Logic. Lattice name for a variation on PALs Gate. Smallest logic element with several inputs and one output. AND gate output is high when all inputs are high. OR gate output is high when at least one input is high. A 2-input NAND gate is used as the measurement unit for gate array complexity.

Gate Array – ASIC where transistors are pre-defined, and only the interconnect pattern is customized for the individual application.

GTL – Gunning Transceiver Logic. A high-speed, low-power back-plane standard.

GUI – Graphic User Interface. A way of representing the computer output on the screen as graphics, pictures, icons, and windows. Pioneered by Xerox and the Macintosh, now universally adopted (e.g., by Windows 95).

HDL – Hardware Description Language.

Hierarchical Design – Design description in multiple layers, from the highest (overview) to the lowest (circuit details). Alternative: flat design, where everything is described at the same level of detail. Incremental design making small design changes while maintaining most of the layout and routing.

Interconnect – Metal lines and programmable switches that connect signals between logic blocks and between logic blocks and the I/O.

IOB or I/O – Input/Output block. Logic block with features specialized for interfacing with the PC board.

ISO9000 – An internationally recognized quality standard. Xilinx is certified to ISO9001 and ISO9002.

IP – Intellectual Property. In the legal sense: patents, copyrights, and trade secrets. In integrated circuits: pre-defined large functions, called cores, that help you complete large designs faster.

ISP – In-System Programmable device. A programmable logic device that can be programmed after it has been connected to (soldered into) the system PC board. Although all SRAM-based FPGAs are naturally ISP, this term is only used with certain CPLDs, to distinguish them from the older CPLDs that must be programmed in programming equipment.

JTAG – Joint Test Action Group. Older name for IEEE 1149.1 Boundary Scan, a method to test PC boards and ICs.

LogiBLOX – Formerly called X-Blox. Library of logic modules, often with user-definable parameters, like data width. Very similar to LPM.

Logic Cell – Metric for FPGA density. One logic cell is one 4-input look-up table plus one flip-flop.

LPM – Library of Parameterized Modules. Library of logic modules, often with user-definable parameters, like data width. Very similar to LogiBlox.

LUT – Look-Up Table. Also called function generator with N inputs and one output. Can implement any logic function of its N inputs. N is between 2 and 6; 4-input LUTs are most popular.

Macrocell – The logic cell in a sum-of-products CPLD or PAL/GAL.

Mapping – Process of assigning portions of the logic design to the physical chip resources (CLBs). With FPGAs, mapping is a more demanding and more important process than with gate arrays.

MTBF – Mean Time Between Failure. The statistically relevant up-time between equipment failure. See also FIT.

Netlist – Textual description of logic and interconnects. See also XNF and EDIF.

NRE – Non-Recurring Engineering charges. Startup cost for the creation of an ASIC, gate array, or HardWire. Pays for layout, masks, and test development. FPGAs and CPLDs do not require NRE.

Optimization – Design change to improve performance. See also Synthesis.

OTP – One-Time Programmable. Irreversible method of programming logic or memory. Fuses and anti-fuses are inherently OTP. EPROMs and EPROM-based CPLDs are OTP if their plastic package blocks the ultraviolet light needed to erase the stored data or configuration.

PAL – Programmable Array Logic. Oldest practical form of programmable logic, implemented a sum-of-products plus optional output flip-flops.

Partitioning – In FPGAs, the process of dividing the logic into sub-functions that can later be placed into individual CLBs. Partitioning precedes placement.

PCI – Peripheral Component Interface. Synchronous bus standard characterized by short range, light loading, low cost, and high performance. A 33 MHz PCI can support data byte transfers of up to 132 megabytes per second on 36 parallel data lines (including parity) and a common clock. There is also a new 66 MHz standard.

PCMCIA – Personal Computer Memory Card Interface Association. (Also: People Can't Memorize Computer Industry Acronyms). Physical and electrical standard for small plug-in boards for portable computers.

Pin-Locking – Rigidly defining and maintaining the functionality and timing requirements of device pins while the internal logic is still being designed or modified. Pin-locking has become important, since circuit-board fabrication times are longer than PLD design implementation times.

PIP – Programmable Interconnect Point. In Xilinx FPGAs, a point where two signal lines can be connected, as determined by the device configuration.

Placement – In FPGAs, the process of assigning specific parts of the design to specific locations (CLBs) on the chip. Usually done automatically.

PLA – Programmable Logic Array. The first and most flexible programmable logic configuration with two programmable planes providing any combination of "AND" and "OR" gates and sharing of AND terms across multiple ORs. This architecture is implemented in CoolRunner and CoolRunner-II devices.

PLD – Programmable Logic Device. Most generic name for all programmable logic: PALs, CPLDs, and FPGAs.

QML – Qualified Manufacturing Line. For example, ISO9000.

Routing – The interconnection, or the process of creating the desired interconnection, of logic cells to make them perform the desired function. Routing follows partitioning and placement.

Schematic – Graphic representation of a logic design in the form of interconnected gates, flip-flops, and larger blocks. Older and more visually intuitive alternative to the increasingly more popular equation-based or high-level language text description of a logic design.

SelectRAM – Xilinx-specific name for a small RAM (usually 16 bits), implemented in a LUT.

Simulation – Computer modeling of logic and (sometimes) timing behavior of logic driven by simulation inputs (stimuli or vectors).

SPROM – Serial Programmable Read-Only Memory. Non-volatile memory device that can store the FPGA configuration bitstream. The SPROM has a built-in address counter, receives a clock, and outputs a serial bitstream.

SRAM – Static Random Access Memory. Read-write memory with data stored in latches. Faster than DRAM and with simpler timing requirements, but smaller in size and about four times as expensive than DRAM of the same capacity.

SRL16 – Shift Register LUT, an alternative mode of operation for every function generator (LUT) that are part of every CLB in Virtex and Spartan FPGAs. This mode increases the number of flip-flops by 16. Adding flip-flops enables fast pipelining – ideal in DSP applications.

Static Timing – Detailed description of on-chip logic and interconnect delays.

Sub-Micron – The smallest feature size is usually expressed in micron (µ = millionth of a meter, or thousandth of a millimeter) The state of the art is moving from 0.35µ to 0.25µ, and may soon reach 0.18µ. The wavelength of visible light is 0.4 to 0.8µ. 1 mil = 25.4µ.

Synchronous – Circuitry that changes state only in response to a common clock, as opposed to asynchronous circuitry that responds to a multitude of derived signals. Synchronous circuits are easier to design, debug, and modify, and tolerate parameter changes and speed upgrades better than asynchronous circuits.

Synthesis – Optimization process of adapting a logic design to the logic resources available on the chip, like LUTs, longline, and dedicated carry. Synthesis precedes mapping.

SystemI/O – Technology incorporated in Virtex-II FPGAs that uses the SelectIO-Ultra blocks to provide the fastest and most flexible electrical interfaces available. Each I/O pin is individually programmable for any of the 19 single-ended I/O standards or six differential I/O standards, including LVDS, SSTL, HSTL II, and GTL+. SelectIO-Ultra technology delivers 840 Mbps LVDS performance using dedicated DDR registers.

TBUFs – Buffers with a tri-state option, where the output can be made inactive. Used for multiplexing different data sources onto a common bus. The pull- down-only option can use the bus as a wired AND function.

Timing – Relating to delays, performance, or speed.

Timing Driven – A design or layout method that takes performance requirements into consideration.

UART – Universal Asynchronous Receiver/Transmitter. An 8-bit-parallel- to-serial and serial-to-8-bit-parallel converter, combined with parity and start- detect circuitry and sometimes even FIFO buffers. Used widely in asynchronous serial communications interfaces such as modems.

USB – Universal Serial Bus. A new, low-cost, low-speed, self-clocking bit- serial bus (1.5 MHz and 12 MHz) using four wires (Vcc, ground, differential data) to daisy-chain as many as 128 devices.

VME – Older bus standard, popular with MC68000-based industrial computers.

XA – Device suffix for automotive parts.

XNF File – Xilinx proprietary description format for a logic design Alternative: EDIF.

ACRONYMS

ACRONYMS

ABEL	Advanced Boolean Expression Language
ADC	Analog-to-Digital Converter
AIM	Advanced Interconnect Matrix
ANSI	American National Standards Institute
ASIC	Application Specific Integrated Circuit
ASSP	Application Specific Standard Product
ATE	Automatic Test Equipment
BGA	Ball Grid Array
BLVDS	Backplane Low Voltage Differential Signaling
BUFG	Global Clock Buffer
CAD	Computer Aided Design
CAN	Controller Area Network
CBT	Computer Based Training
CDMA	Code Division Multiple Access
CE	Clock Enable
CLB	Configurable Logic Block
CLK	Clock Signal
CMOS	Complementary Metal Oxide Semiconductor
CPLD	Complex Programmable Logic Device
CSP	Chip Scale Packaging
DCI	Digitally Controlled Impedance
DCM	Digital Clock Manager
DCM	Digital Control Management
DES	Data Encryption Standard
DRAM	Dynamic Random Access Memory
DRC	Design Rule Checker

DSL	Digital Subscriber Line
DSP	Digital Signal Processor
DTV	Digital Television
ECS	Schematic Editor
EDA	Electronic Design Automation
EDIF	Electronic Digital Interchange Format
EMI	Electromagnetic Interference
EPROM	Erasable Programmable Read Only Memory
eSP	emerging Standards and Protocols
FAT	File Allocation Table
FIFO	First In First Out
FIR	Finite Impulse Response (Filter)
FIT	Failures in Time
FLBGA	Flip Chip Ball Grid Array
fMax	Frequency Maximum
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GPS	Global Positioning System
GTL	Gunning Transceiver Logic
GTLP	Gunning Transceiver Logic Plus
GUI	Graphical User Interface
HDL	Hardware Description Language
HDTV	High Definition Television
HEX	Hexadecimal
HSTL	High Speed Transceiver Logic
I/O	Inputs and Outputs
IBIS	I/O Buffer Information Specification
IEEE	Institute of Electrical and Electronics Engineers
ILA	Integrated Logic Analyzer
IOB	Input Output Block
IP	Intellectual Property
IRL	Internet Reconfigurable Logic
ISE	Integrated Software Environment
ISP	In System Programming
JEDEC	Joint Electron Device Engineering Council
JTAG	Joint Test Advisory Group

LAN	Local Area Network
LEC	Logic Equivalence Checker
LMG	Logic Modeling Group
LSB	Least Significant Bit
LUT	Look Up Table
LVCMOS	Low Voltage Complementary Metal Oxide Semiconductor
LVDS	Low Voltage Differential Signaling
LVDSEXT	Low Voltage Differential Signaling Extension
LVPECL	Low Voltage Positive Emitter Coupled Logic
LVTTL	Low Voltage Transistor to Transistor Logic
MAC	Multiply and Accumulate
MAN	Metropolitan Area Network
MCS	Manipulate Comment Section
MIL	Military
MOSFET	Metal Oxide Semiconductor Field Effect Transistors
MP3	MPEG Layer III Audio Coding
MPEG	Motion Picture Experts Group
MSB	Most Significant Bit
MUX	Multiplexer
NAND	Not And
NGC	Native Generic Compiler
NRE	Non-Recurring Engineering (Cost)
OE	Output Enable
OTP	One Time Programmable
PACE	Pinout and Area Constraints Editor
PAL	Programmable Array Logic
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
PCMCIA	Personal Computer Memory Card International Association
PCS	Personnel Communications System
PLA	Programmable Logic Array
PLD	Programmable Logic Device
PROM	Programmable Read Only Memory
QFP	Quad Flat Pack
QML	Qualified Manufacturers Listing
QPRO	QML Performance Reliability of Supply Off the Shelf ASIC

RAM	Random Access Memory
RC	Radio Controlled
ROM	Read Only Memory
SOP	Sum of Product
SPLD	Simple Programmable Logic Device
SRAM	Static Random Access Memory
SRL16	Shift Register LUT
SSTL	Stub Series Terminated Transceiver Logic
TIM	Time in Market
Tpd	Time of Propagation Delay (through the device)
TQFP	Thin Quad Flat Pack
TTM	Time to Market
UCF	User Constraints File
UMTS	Universal Mobile Telecommunications System
UV	Ultraviolet
VCCO	Voltage Current Controlled Oscillator
VFM	Variable Function Multiplexer
VHDL	VHISC High Level Description Language
VHSIC	Very High Speed Integrated Circuit
VREF	Voltage Reference
VSS	Visual Software Solutions
WAN	Wireless Area Network
WLAN	Wireless Local Access Network
WPU	Weak Pull Up
XCITE	Xilinx Controlled Impedance Technology
XOR	Exclusive OR
XST	Xilinx Synthesis Technology
ZIA	Zero Power Interconnect Array