DESIGN OF TIMER FOR APPLICATION IN ATM USING VHDL AND FPGA

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Bachelor of Technology

in

Electronics & Instrumentation Engineering

By

SUBHRAJIT MISHRA

And

ISHAN DHAR

Department of Electronics & Communication Engineering

National Institute of Technology

Rourkela

2007

DESIGN OF TIMER FOR APPLICATION IN ATM USING VHDL AND FPGA

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Bachelor of Technology

in

Electronics & Instrumentation Engineering

By

SUBHRAJIT MISHRA

And

ISHAN DHAR

Under the Guidance of

Prof. K.K.MAHAPATRA

Department of Electronics & Communication Engineering

National Institute of Technology

Rourkela

2007

National Institute of Technology

Rourkela

CERTIFICATE

This is to certify that the thesis entitled, "Design of Timer for application in ATM using FPGA and VHDL" submitted by Sri Subhrajit Mishra and Sri Ishan Dhar in partial fulfillments for the requirements for the award of Bachelor of Technology Degree in Electronics & Instrumentation Engineering at National Institute of Technology, Rourkela (Deemed University) is an authentic work carried out by him under my supervision and guidance.

To the best of my knowledge, the matter embodied in the thesis has not been submitted to any other University / Institute for the award of any Degree or Diploma.

Date:

Prof. K.K.Mahapatra
Dept. of Electronics & Instrumentation Engg
National Institute of Technology
Rourkela - 769008

ACKNOWLEDGEMENT

We would like to articulate our deep gratitude to our project guide Prof. K.K.Mahapatra who has always been our motivation for carrying out the project. It is our pleasure to refer Microsoft Word exclusive of which the compilation of this report would have been impossible. An assemblage of this nature could never have been attempted with out reference to and inspiration from the works of others whose details are mentioned in reference section. We acknowledge out indebtedness to all of them. Last but not the least, our sincere thanks to all of our friends who have patiently extended all sorts of help for accomplishing this undertaking.

SUBHRAJIT MISHRA

ISHAN DHAR

CONTENTS

ABSTRACT

Introduction

A watchdog timer is a computer hardware timing device that triggers a system reset if the main program, due to some fault condition, such as a hang, neglects to regularly service the watchdog (writing a “service pulse” to it, also referred to as “petting the dog”). The intention is to bring the system back from the hung state into normal operation. Such a timer has got various important applications, one of them being in ATMs (Automated Teller Machine) which we have studied and designed in our project.

Steps involved

1. **Coding using VHDL**

   The key advantage of VHDL when used for systems design is that it allows the behaviour of the required system to be described (modeled) and verified (simulated) before synthesis tools translate the design into real hardware (gates and wires).n information theory. To start coding in VHDL, one needs a simulation toolI. The simulation tool that we have used here is Xilinx ISI9.1i.First the required code for timer circuit was written in VHDL and simulated so as to obtain the required output waveforms.

2. **Burning the code on Spartan-II kit**

   After the coding was completed, VHDL model is translated into the "gates and wires" that are mapped onto a programmable logic device. The programmable logic device used here is Spartan-2

Experimental work

The above coding and burning methods were completed and the output was observed on FPGA kit. The timer code was implemented using VHDL while burning was done using Spartan-2 kit.

Chapter 1

GENERAL INTRODUCTION

Most embedded systems need to be self-reliant. It's not usually possible to wait for someone to reboot them if the software hangs. Some embedded designs, such as space probes, are simply not accessible to human operators. If their software ever hangs, such systems are permanently disabled. In other cases, the speed with which a human operator might reset the system would be too slow to meet the uptime requirements of the product. A watchdog timer is a piece of hardware that can be used to automatically detect software anomalies and reset the processor if any occur. Generally speaking, a watchdog timer is based on a counter that counts down from some initial value to zero. The embedded software selects the counter's initial value and periodically restarts it. If the counter ever reaches zero before the software restarts it, the software is presumed to be malfunctioning and the processor's reset signal is asserted. The processor (and the embedded software it’s running) will be restarted as if a human operator had cycled the power. The process of restarting the watchdog timer's counter is sometimes called "kicking the dog." The appropriate visual metaphor is that of a man being attacked by a vicious dog. If he keeps kicking the dog, it can't ever bite him. But he must keep kicking the dog at regular intervals to avoid a bite. Similarly, the software must restart the watchdog timer at a regular rate, or risk being restarted. Watchdog timers may also trigger control systems to move into a safety state, such as turning off motors, high-voltage electrical outputs, and other potentially dangerous subsystems until the fault is cleared. For example, a watchdog timer can be implemented with a x-bit counter in a system working with a clock signal of y MHz, therefore, the system will shut down if the timer is not reset in a period of seconds. Watch dog timers have got various important applications one of them being in ATMs which we have studied and designed in our project.

Chapter 2

VHDL

2.1 Introduction

VHDL is a language for describing digital electronic systems. It arose out of the United States Government's Very High Speed Integrated Circuits (VHSIC) program, initiated in 1980. In the course of this program, it became clear that there was a need for a standard language for describing the structure and function of integrated circuits (ICs). Hence the VHSIC Hardware Description Language (VHDL) was developed, and subsequently adopted as a standard by the Institute of Electrical and Electronic Engineers (IEEE) in the US.VHDL is designed to fill a number of needs in the design process. Firstly, it allows description of the structure of a design that is how it is decomposed into sub-designs, and how those sub-designs are interconnected. Secondly, it allows the specification of the function of designs using familiar programming language forms. Thirdly, as a result, it allows a design to be simulated before being manufactured, so that designers can quickly compare alternatives and test for correctness without the delay and expense of hardware prototyping. The purpose of this booklet is to give you a quick introduction to VHDL.This is done by informally describing the facilities provided by the language, and using examples to illustrate them. This booklet does not fully describe every aspect of the language. For such fine details, you should consult the *IEEE Standard VHDL Language Reference Manual*. However, be warned: the standard is like a legal document, and is very difficult to read unless you are already familiar with the language. This booklet does cover enough of the language for substantial model writing. It assumes you know how to write computer programs using a conventional programming language such as Pascal, C or Ada. The remaining chapters of this booklet describe the various aspects of VHDL in a bottom-up manner. Chapter2 describes the facilities of VHDL which most resemble normal sequential programming languages. These include data types, variables, expressions, sequential statements and subprograms. Then examines the facilities for describing the structure of a module and how it it decomposed into sub-modules. Then covers aspects of VHDL that integrate the programming language features with a discrete event timing model to allow simulation of behaviour. These facilities are combined to form a complete model of a system. Then there is a potpourri of more advanced features which you may find useful for modeling more complex systems. Throughout this booklet, the syntax of language features is presented in Backus-Naur Form (BNF). The syntax specifications are drawn from the IEEE VHDL Standard. Concrete examples are also given to illustrate the language features. In some cases, some alternatives are omitted from BNF productions where they are not directly relevant to the context. For this reason, the full syntax is included in Appendix A, and should be consulted as a reference.

One should be attentive towards the following points:
1) The purpose of VHDL.
2) The overall structure of VHDL
3) The VHDL development and execution sequence
4) The VHDL simulation cycle
5) Basic VHDL object types and declarations
6) The syntax and semantics of basic VHDL sequential statements
7) The syntax and semantics of basic VHDL concurrent statements
8) VHDL modeling techniques for the simulation and evaluation of gate-level
   digital circuits.

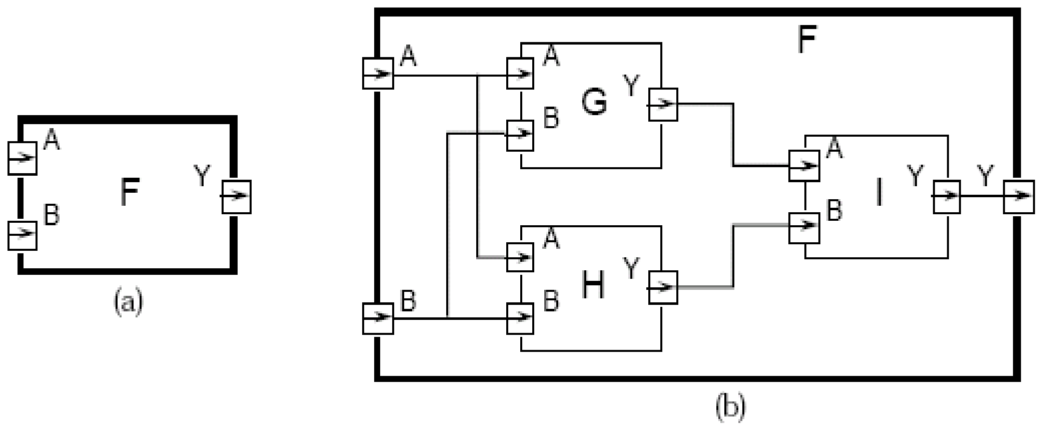


Figure 1-1. Example of a structural description.

2.2 Describing Structure

A digital electronic system can be described as a module with inputs and/or outputs. The electrical values on the outputs are some function of the values on the inputs. Figure1-1(a) shows an example of this view of a digital system. The module F has two inputs, A and B, and an output Y.Using VHDL terminology, we call the module F a design *entity*, and the inputs and outputs are called *ports*. One way of describing the function of a module is to describe how it is composed of sub-modules. Each of the sub-modules is an *instance* of some entity, and the ports of the instances are connected using *signals* Figure1-1(b) shows how the entity F might be composed of instances of entities G, H and I. This kind of description is called a *structural* description. Note that each of the entities G, H and I might also have a structural description.

2.3 Describing Behaviour

In many cases, it is not appropriate to describe a module structurally. One such case is a module which is at the bottom of the hierarchy of some other structural description. For example, if you are designing a system using IC packages bought from an IC shop, you do not need to describe the internal structure of an IC. In such cases, a description of the function performed by the module is required, without reference to its actual internal structure. Such a description is called a *functional* or *behavioral* description. To illustrate this, suppose that the function of the entity F in Figure1-1(a) is the exclusive-or function. Then a behavioural description of F could be the Boolean function

$$Y = \overline{A} . B + A . \overline{B}$$

More complex behaviours cannot be described purely as a function of inputs. In systems with feedback, the outputs are also a function of time. VHDL solves this problem by allowing description of behaviour in the form of an executable program.

2.3.1 Discrete Event Time Model

Once the structure and behaviour of a module have been specified, it is possible to simulate the module by executing its behavioural description. This is done by simulating the passage of time in discrete steps. At some simulation time, a module input may be stimulated by changing the value on an input port. The module reacts by running the code of its behavioural description and scheduling new values to be placed on the signals connected to its output ports at some later simulated time. This is called scheduling a *transaction* on that signal. If the new value is different from the previous value on the signal, an *event* occurs, and other modules with input ports connected to the signal may be activated. The simulation starts with an *initialization phase*, and then proceeds by repeating a two-stage *simulation cycle*. In the initialization phase, all signals are given initial values, the simulation time is set to zero, and each module's behaviour program is executed. This usually results in transactions being scheduled on output signals for some later time. In the first stage of a simulation cycle, the simulated time is advanced to the earliest time at which a transaction has been scheduled. All transactions scheduled for that time are executed, and this may cause events to occur on some signals. In the second stage, all modules which react to events occurring in the first stage have their behaviour program executed. These programs will usually schedule further transactions on their output signals. When all of the behaviour programs have finished executing, the simulation cycle repeats. If there are no more scheduled transactions, the whole simulation is completed. The purpose of the simulation is to gather information about the changes in system state over time. This can be done by running the simulation under the control of a *simulation monitor*. The monitor allows signals and other state information to be viewed or stored in a trace file for later analysis. It may also allow interactive stepping of the simulation process, much like an interactive program debugger.

2.3.2 A Quick Example

In this section we will look at a small example of a VHDL description of a two-bit counter to give you a feel for the language and how it is used. We start the description of an entity by specifying its external interface, which includes a description of its ports. So the counter might be defined as:

```
entity count2 is
generic (prop_delay: Time := 10 ns);
port (clock : in bit;
q1, q0 : out bit);
end count2;
```

This specifies that the entity count2 has one input and two outputs, all of which are bit values, that is, they can take on the values '0' or '1'. It also defines a generic constant called prop_delay which can be used to control the operation of the entity (in this case its propagation delay). If no value is

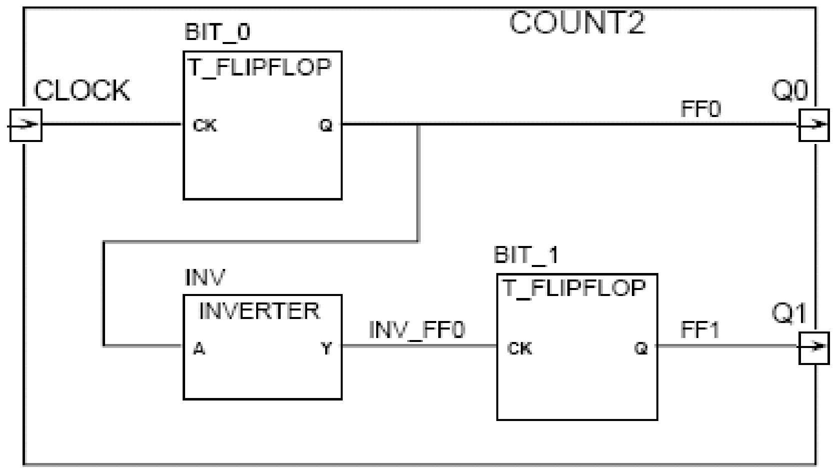


Figure1-2. Structure of count2.

explicitly given for this value when the entity is used in a design, the default value of 10ns will be used. An implementation of the entity is described in an architecture body. There may be more than one architecture body corresponding to a single entity specification, each of which describes a different view of the entity. Behavioural description of counter could be written as:

```
  architecture behaviour of count2 is
begin
 count_up: process (clock)
   variable count_value : natural := 0;
 begin
   if clock = '1' then
  count_value := (count_value + 1) mod 4;
  q0 <= bit'val(count_value mod 2) after prop_delay;
  q1 <= bit'val(count_value / 2) after prop_delay;
  end if;
 end process count_up;
end behaviour;
```

This process has a variable called count_value to store the current state of the counter. The variable is initialized to zero at the start of simulation, and retains its value between activations of the process. When the clock input changes from '0' to '1', the state variable is incremented, and transactions are scheduled on the two output ports based on the new value. The assignments use the generic constant prop_delay to determine how long after the clock change the transaction should be scheduled. When control reaches the end of the process body, the process is suspended until another change occurs on clock. The two-bit counter might also be described as a circuit composed of two T-flip-flops and an inverter, as shown in Figure1-2. This can be written in VHDL as:

```
architecture structure of count2 is
component t_flipflop
port (ck : in bit; q : out bit);
end component;
component inverter
port (a : in bit; y : out bit);
end component;
signal ff0, ff1, inv_ff0 : bit;
begin
bit_0 : t_flipflop port map (ck => clock, q => ff0);
inv : inverter port map (a => ff0, y => inv_ff0);
bit_1 : t_flipflop port map (ck => inv_ff0, q => ff1);
q0 <= ff0;
q1 <= ff1;
end structure;
```

In this architecture, two component types are declared, t_flipflop and inverter, and three internal signals are declared. Each of the components is then instantiated, and the ports of the instances are mapped onto signals and ports of the entity. For example, bit_0 is an instance of the t_flipflop component, with its ck port connected to the clock port of the count2 entity, and its q port connected to the internal signal ff0. The last two signal assignments update the entity ports whenever the values on the internal signals change.

Chapter 3

SPARTAN-II

3.1 What is FPGA?

FPGA stands for Field Programmable Gate Array. It is a semiconductor device containing programmable logic components and programmable interconnects. The programmable logic components can be programmed to duplicate the functionality of basic logic gates such as AND, OR, XOR, NOTor more complex combinational functions such as decoders or simple mathematical functions. In most FPGAs, these programmable logic components (or logic blocks, in FPGA parlance) also include memory elements, which may be simple flip flops or more complete blocks of memories.

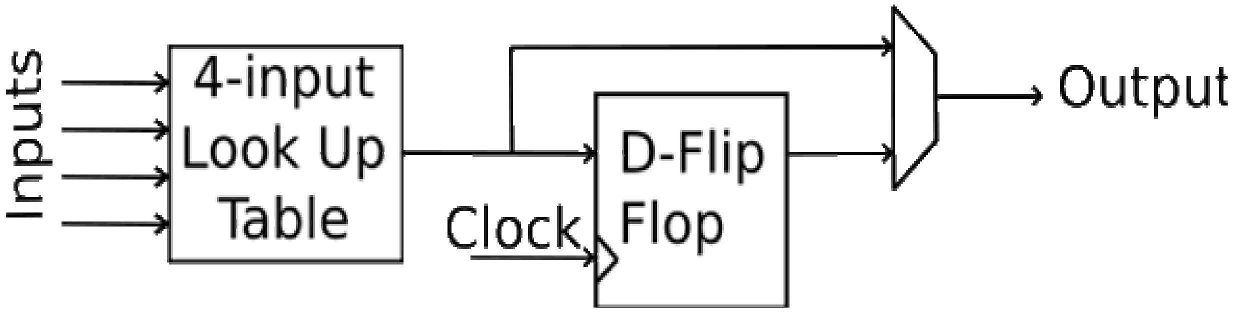
A hierarchy of programmable interconnects allows the logic blocks of an FPGA to be interconnected as needed by the system designer, somewhat like a one-chip programmablebreadboard. These logic blocks and interconnects can be programmed after the manufacturing process by the customer/designer (hence the term "field programmable", i.e. programmable in the field) so that the FPGA can perform whatever logical function is needed.

FPGAs are generally slower than their application specific integrated circuit (ASIC) counterparts, as they can't handle as complex a design, and draw more power. However, they have several advantages such as a shorter time to market, ability to re-program in the field to fix bugs, and lower non recurring engineering cost costs. Vendors can sell cheaper, less flexible versions of their FPGAs which cannot be modified after the design is committed. The development of these designs is made on regular FPGAs and then migrated into a fixed version that more resembles an ASIC. Complex programmable logic devices, or CPLDs, are another alternative.

The typical basic architecture consists of an array of configurable logic blocks (CLBs) and routing channels. Multiple I/O Pads may fit into the height of one row or the width of one column in the array. Generally, all the routing channels have the same width (number of wires).

An application circuit must be mapped into an FPGA with adequate resources.

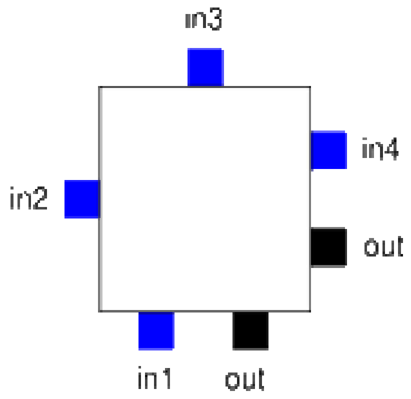
The typical FPGA logic block consists of a 4-input look-up (LUT), andflip-flop, as shown below.



Logic block

There is only one output, which can be either the registered or the unregistered LUT output. The logic block has four inputs for the LUT and a clock input. Since clock signals (and often other high-fanout signals) are normally routed via special-purpose dedicated routing networks in commercial FPGAs, they and other signals are separately managed.

For this example architecture, the locations of the FPGA logic block pins are shown below.



Logic Block Pin Locations

Each input is accessible from one side of the logic block, while the output pin can connect to routing wires in both the channel to the right and the channel below the logic block.

3.2 Introduction to Spartan-II

The Spartan-II 2.5V Field-Programmable Gate Array family gives users high performance, abundant logic resources, and a rich feature set, all at an exceptionally low price. The six-member family offers densities ranging from 15,000 to 200,000 system gates, as shown in Table 1. System performance is supported up to 200 MHz. Spartan-II devices deliver more gates, I/Os, and features per dollar than other FPGAs by combining advanced process technology with a streamlined Virtex-based architecture. Features include block RAM (to 56K bits), distributed RAM (to 75,264 bits), 16 selectable I/O standards, and four DLLs. Fast, predictable interconnect means that successive design iterations continue to meet timing requirements. The Spartan-II family is a superior alternative to mask-programmed ASICs. The FPGA avoids the initial cost, lengthy development cycles, and inherent risk of conventional ASICs. Also, FPGA programmability permits design upgrades in the field with no hardware replacement necessary (impossible with ASICs).

Features

Second generation ASIC replacement technology

- Densities as high as 5,292 logic cells with up to 200,000 system gates
- Streamlined features based on Virtex architecture
- Unlimited reprogrammability
- Very low cost
- Advanced 0.18 micron process

• System level features

- SelectRAM+™ hierarchical memory:

· 16 bits/LUT distributed RAM

· Configurable 4K bit block RAM

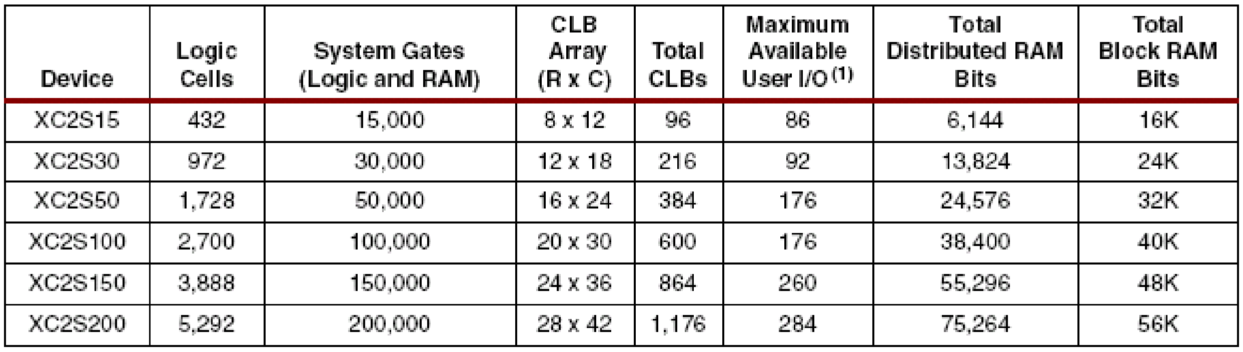
· Fast interfaces to external RAM

- Fully PCI compliant
- Low-power segmented routing architecture
- Full readback ability for verification/observability
- Dedicated carry logic for high-speed arithmetic
- Efficient multiplier support
- Cascade chain for wide-input functions
- Abundant registers/latches with enable, set, reset
- Four dedicated DLLs for advanced clock control
- Four primary low-skew global clock distribution nets
- IEEE 1149.1 compatible boundary scan logic

• Versatile I/O and packaging
- Pb-free package options
- Low-cost packages available in all densities
- Family footprint compatibility in common packages
- 16 high-performance interface standards
- Hot swap Compact PCI friendly
- Zero hold time simplifies system timing

• Fully supported by powerful Xilinx development system
- Foundation ISE Series: Fully integrated software
- Alliance Series: For use with third-party tools
- Fully automatic mapping, placement, and routing

Table 1: **Spartan-II FPGA Family Members**



Device	Logic Cells	System Gates (Logic and RAM)	CLB Array (R x C)	Total CLBs	Maximum Available User I/O (1)	Total Distributed RAM Bits	Total Block RAM Bits
XC2S15	432	15,000	8 x 12	96	86	6,144	16K
XC2S30	972	30,000	12 x 18	216	92	13,824	24K
XC2S50	1,728	50,000	16 x 24	384	176	24,576	32K
XC2S100	2,700	100,000	20 x 30	600	176	38,400	40K
XC2S150	3,888	150,000	24 x 36	864	260	55,296	48K
XC2S200	5,292	200,000	28 x 42	1,176	284	75,264	56K

3.3 General Overview

The Spartan-II family of FPGAs have a regular, flexible, programmable architecture of Configurable Logic Blocks (CLBs), surrounded by a perimeter of programmable Input/Output Blocks (IOBs). There are four Delay-Locked Loops (DLLs), one at each corner of the die. Two columns of block RAM lie on opposite sides of the die, between the CLBs and the IOB columns. These functional elements are interconnected by a powerful hierarchy of versatile routing channels Spartan-II FPGAs are customized by loading configuration data into internal static memory cells. Unlimited reprogramming cycles are possible with this approach. Stored values in these cells determine logic functions and interconnections implemented in the FPGA. Configuration data can be read from an external serial PROM (master serial mode), or written into the FPGA in slave serial, slave parallel, or Boundary Scan modes. Spartan-II FPGAs are typically used in high-volume applications where the versatility of a fast programmable solution adds benefits. Spartan-II FPGAs are ideal for shortening product development cycles while offering a cost-effective solution for high volume production. Spartan-II FPGAs achieve high-performance, low-cost operation through advanced architecture and semiconductor technology. Spartan-II devices provide system clock rates up to 200 MHz. Spartan-II FPGAs offer the most cost-effective solution while maintaining leading edge performance. In addition to the conventional benefits of high-volume programmable logic solutions, Spartan-II FPGAs also offer on-chip synchronous single-port and dual-port RAM (block and distributed form), DLL clock drivers, programmable set and reset on all flip-flops, fast carry logic, and many other features.

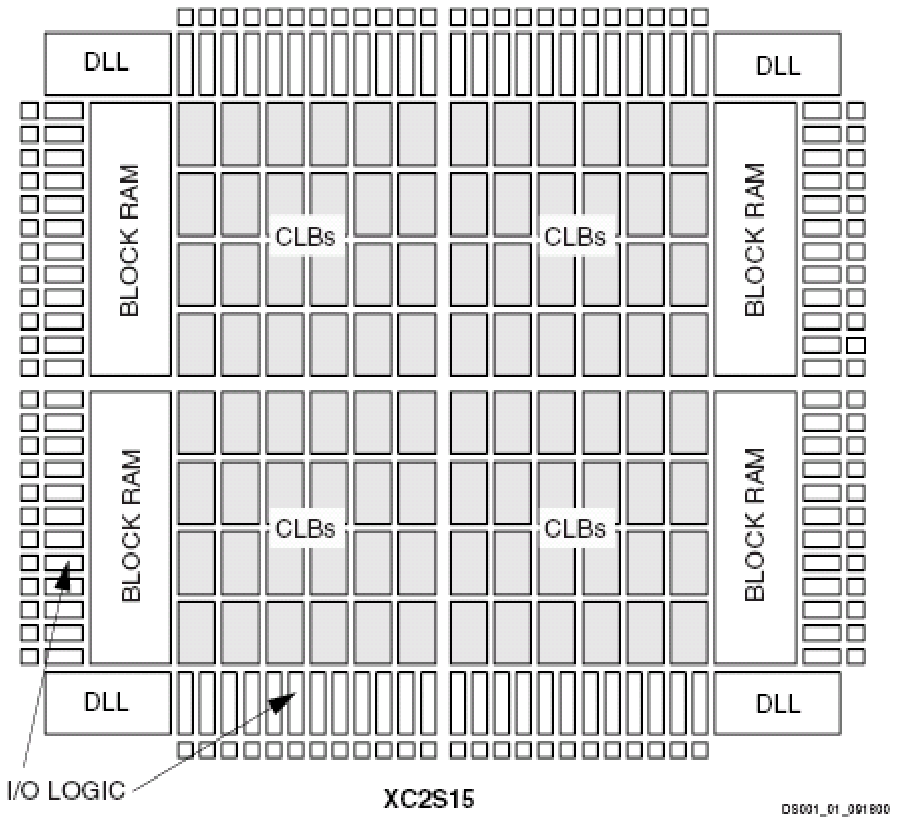
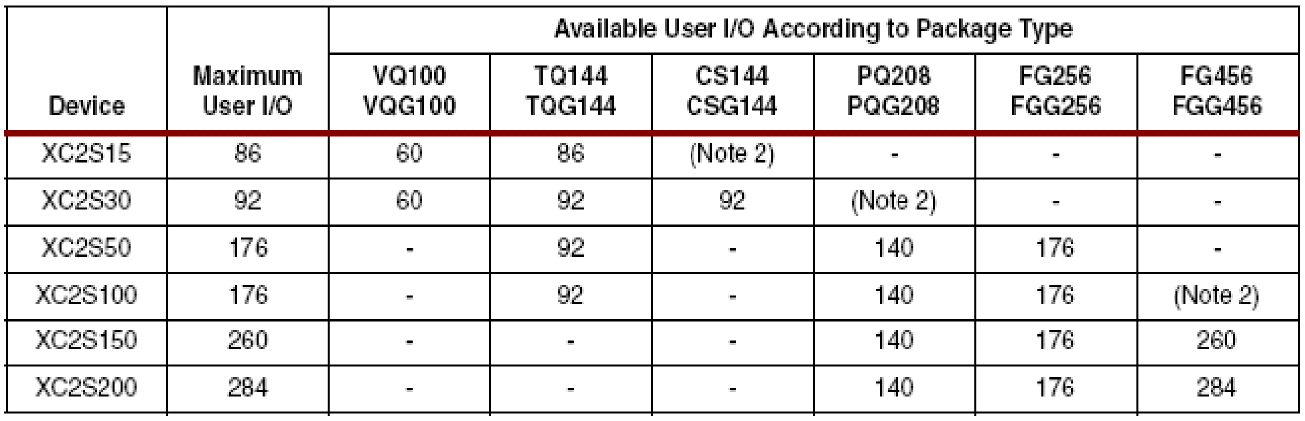


Figure 1: **Basic Spartan-II Family FPGA Block Diagram**

Spartan-II Product Avaibility

Table 2 shows the maximum user I/Os available on the device and the number of user I/Os available for each device/package combination. The four global clock pins are usable as additional user I/Os when not used as a global clock pin. These pins are not included in user I/O counts.

Table 2: **Spartan-II User I/O Chart**(1)

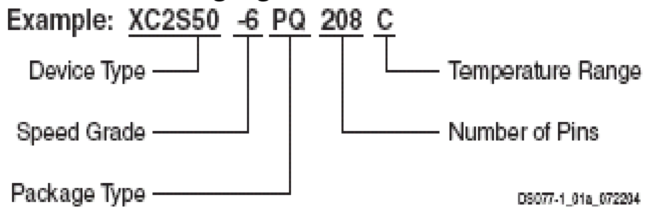


Device	Maximum User I/O	Available User I/O According to Package Type					
		VQ100 VQG100	TQ144 TQG144	CS144 CSG144	PQ208 PQG208	FG256 FGG256	FG456 FGG456
XC2S15	86	60	86	(Note 2)	-	-	-
XC2S30	92	60	92	92	(Note 2)	-	-
XC2S50	176	-	92	-	140	176	-
XC2S100	176	-	92	-	140	176	(Note 2)
XC2S150	260	-	-	-	140	176	260
XC2S200	284	-	-	-	140	176	284

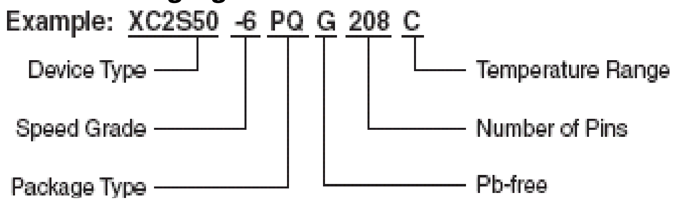
Ordering Information

Spartan-II devices are available in both standard and Pb-free packaging options for all device/package combinations. The
Pb-free packages include a special "G" character in the ordering code.

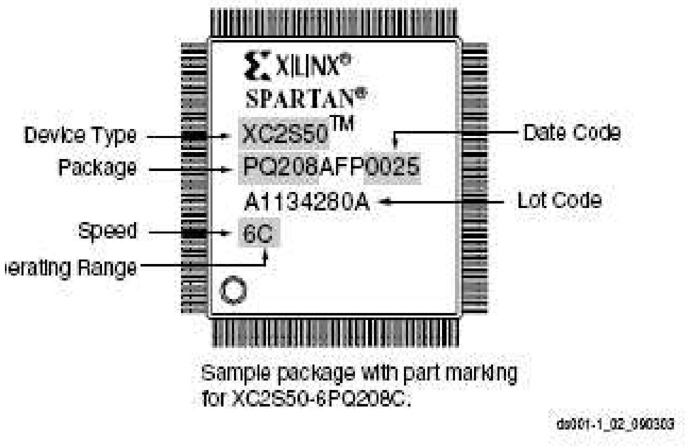
Standard Packaging



Pb-Free Packaging



Device Part Marking



3.4 Architectural Description

The Spartan-II user-programmable gate array, shown in Figure 1 is composed of five major configurable elements:

• IOBs provide the interface between the package pins and the internal logic

• CLBs provide the functional elements for constructing most logic

• Dedicated block RAM memories of 4096 bits each

• Clock DLLs for clock-distribution delay compensation and clock domain control

• Versatile multi-level interconnect structure

As can be seen in Figure 1, form the central logic structure with easy access to all support and routing structures. The IOBs are located around all the logic and memory elements for easy and quick routing of signals on and off the chip. Values stored in static memory cells control all the configurable logic elements and interconnect resources. These values load into the memory cells on power-up, and can reload if necessary to change the function of the device. Each of these elements will be discussed in detail in the following sections.

INPUT/OUTPUT BLOCK

The Spartan-II IOB, as seen in Figure 1, features inputs and outputs that support a wide variety of I/O signaling standards. These high-speed inputs and outputs are capable of supporting various state of the art memory and bus interfaces. Table 1 lists several of the standards which are supported along with the required reference, output and termination voltages needed to meet the standard.The three IOB registers function either as edge-triggered D-type flip-flops or as level-sensitive latches. Each IOB has a clock signal (CLK) shared by the three registers and independent Clock Enable(CE) for each register. In addition to the CLK and CE control signals, the three registers share a Set/Reset (SR). For each register, this signal can be independently configured as a synchronous Set, a synchronous Reset, an asynchronous Preset, or an asynchronous Clear. A feature not shown in the block diagram, but controlled by the software, is polarity control. The input and output buffers and all of the IOB control signals have independent polarity controls. Optional pull-up and pull-down resistors and an optional weak-keeper circuit are attached to each pad. Prior to configuration all outputs not involved in configuration are forced into their high-impedance state. The pull-down resistors and the weak-keeper circuits are inactive, but inputs may optionally be pulled up.

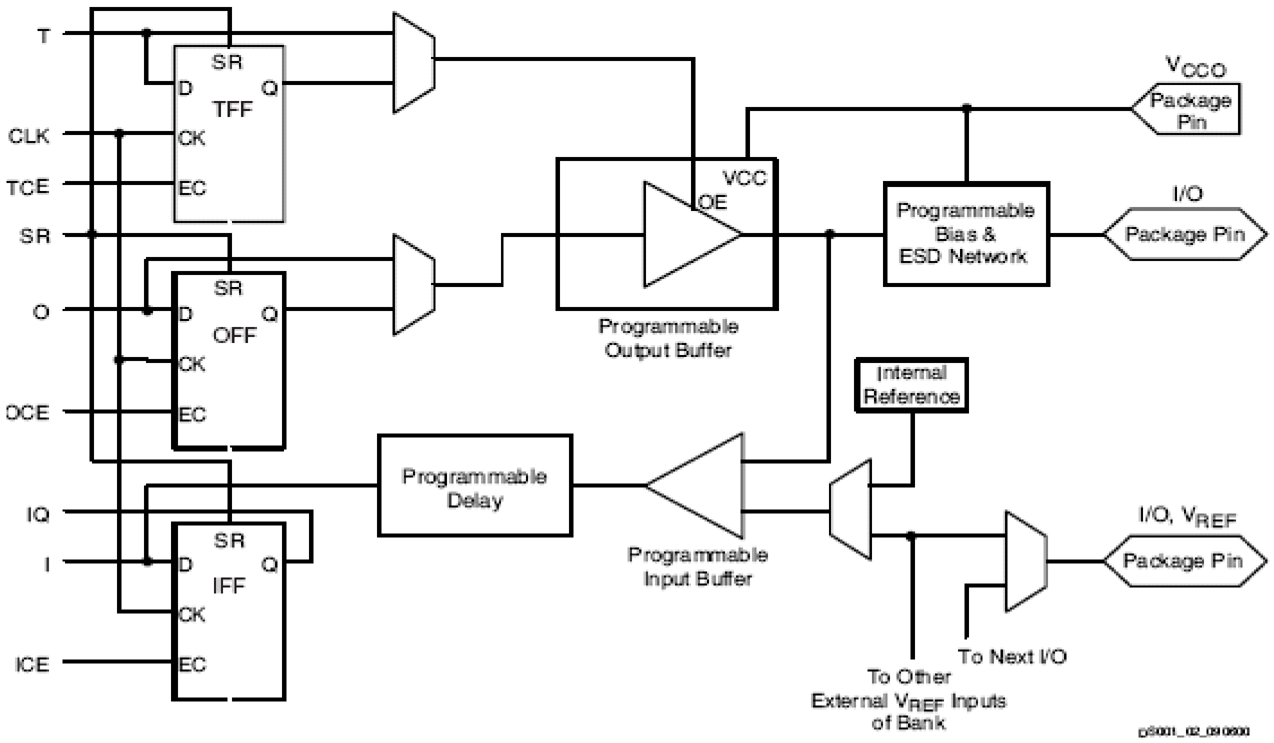


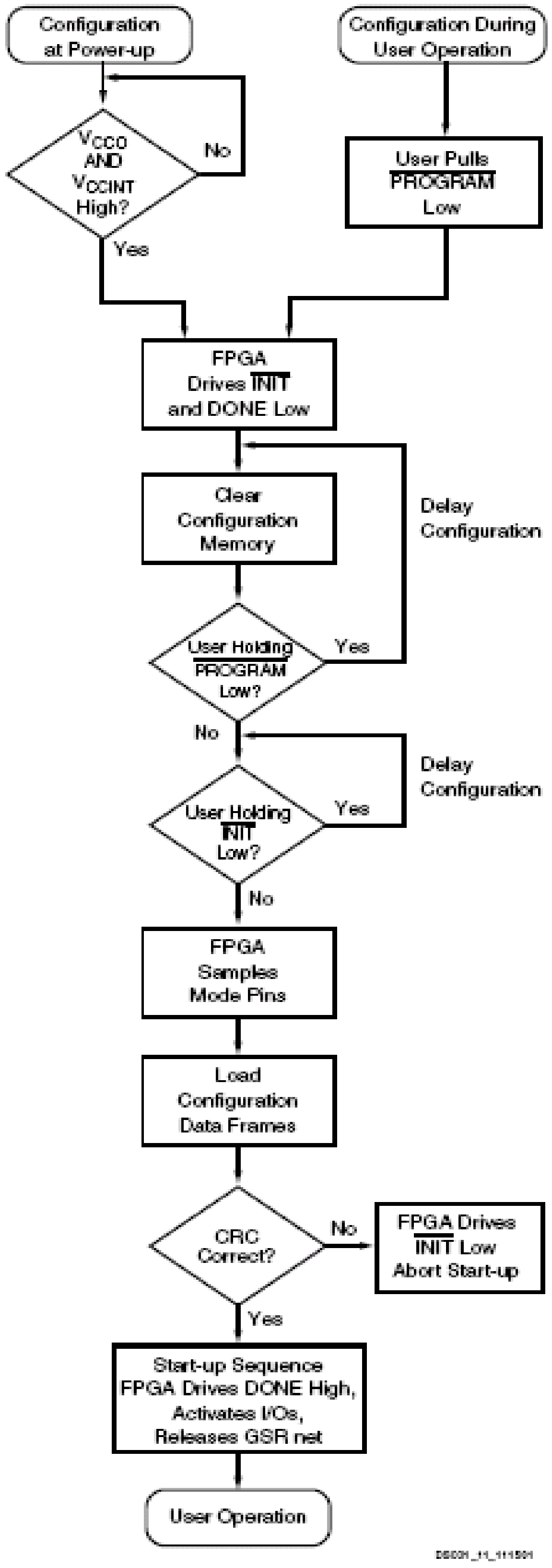
Figure 1: **Spartan-II Input/Output Block (IOB)**

35 Processes Involved

The sequence of steps necessary to configure Spartan-II devices can be divided into three different phases.

- Initiating Configuration
- Configuration memory clear
- Loading data frames
- Start-up

These can be represented in the form of a flow chart in the following manner:



Configuration at Power-up
Configuration During User Operation
V_{CCO} AND V_{CCINT} High?
No
Yes
User Pulls $\overline{\text{PROGRAM}}$ Low
FPGA Drives $\overline{\text{INIT}}$ and DONE Low
Clear Configuration Memory
Delay Configuration
User Holding $\overline{\text{PROGRAM}}$ Low?
Yes
No
Delay Configuration
User Holding $\overline{\text{INIT}}$ Low?
Yes
No
FPGA Samples Mode Pins
Load Configuration Data Frames
CRC Correct?
No
FPGA Drives $\overline{\text{INIT}}$ Low Abort Start-up
Yes
Start-up Sequence FPGA Drives DONE High, Activates I/Os, Releases GSR net
User Operation
DS031_11_111501

Initiating Configuration

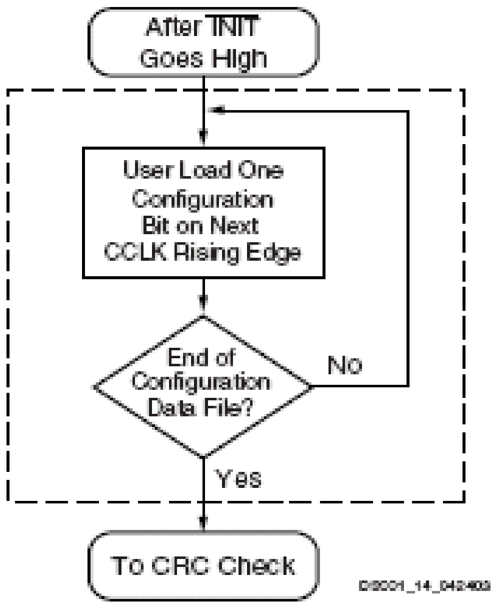
There are two different ways to initiate the configuration process:
applying power to the device or asserting the PROGRAM input. Configuration on power-up occurs automatically unless it is delayed by the user, as described in a separate section below. The waveform for configuration on power-up is shown in Figure 11, page 13. Before configuration can begin, VCCO Bank 2 must be greater than 1.0V. Furthermore, all VCCINT power pins must be connected to a 2.5Vsupply. For more information on delaying configuration, see Clearing Configuration Memory. Once in user operation, the device can be re-configured simply by pulling the PROGRAM pin Low. The device acknowledges the beginning of the configuration process by driving DONE Low, then enters the memory-clearing phase.

Clearing Configuration Mode

The device indicates that clearing the configuration memory is in progress by driving INIT Low. At this time, the user can delay configuration by holding either PROGRAM or INIT Low, which causes the device to remain in the memory clearing phase. Note that the bidirectional INIT line is driving a Low logic level during memory clearing. Thus, to avoid contention, use an open-drain driver to keep INIT Low. With no delay in force, the device indicates that the memory is completely clear by driving INIT High. The FPGA samples its mode pins on this Low-to-High transition.

Loading Configuration Data

Once INIT is High, the user can begin loading configuration data frames into the device. The details of loading the configuration data are discussed in the sections treating the configuration modes individually. The sequence of operations necessary to load configuration data using the serial modes is shown in figure.


After INIT Goes High
User Load One Configuration Bit on Next CCLK Rising Edge
End of Configuration Data File?
No
Yes
To CRC Check
DS001_14_042403

CRR Error Checking

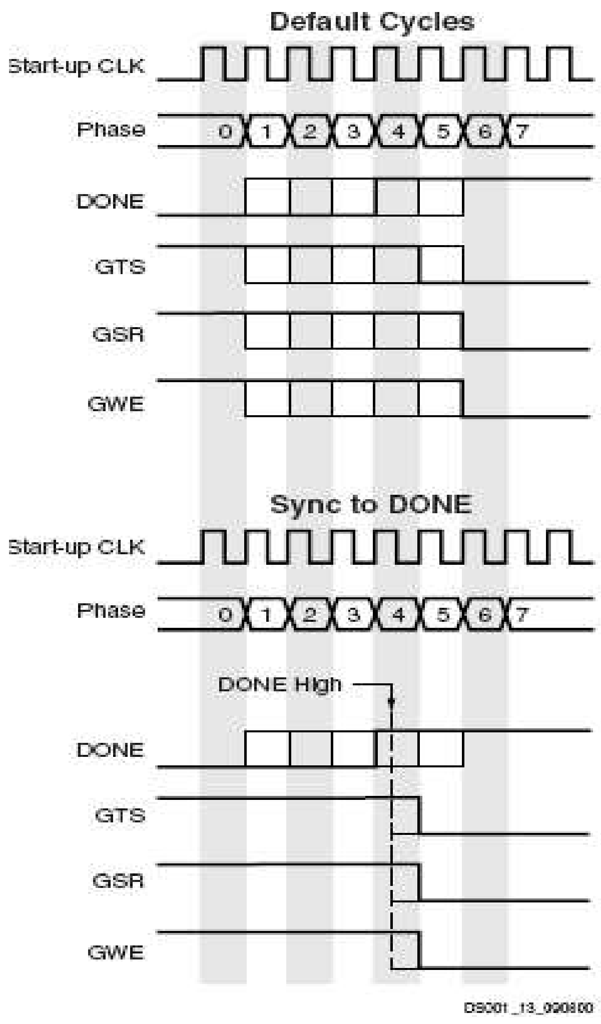
During the loading of configuration data, a CRC value embedded in the configuration file is checked against a CRC value calculated within the FPGA. If the CRC values do not match, the FPGA drives INIT Low to indicate that a frame error has occurred and configuration is aborted. To reconfigure the device, the PROGRAM pin should be asserted to reset the configuration logic. Recycling power also resets the FPGA for configuration.

Start Up

The start-up sequence oversees the transition of the FPGA from the configuration state to full user operation. A match of CRC values, indicating a successful loading of the configuration data, initiates the sequence. During start-up, the device performs four operations:

1. The assertion of DONE. The failure of DONE to go High may indicate the unsuccessful loading of configuration data.
2. The release of the Global Three State net. This activates I/Os to which signals are assigned. The remaining I/Os stay in a high-impedance state with internal weak pull-down resistors present.
3. Negates Global Set Reset (GSR). This allows all flip-flops to change state.
4. The assertion of Global Write Enable (GWE). This allows all RAMs and flip-flops to change state.

By default, these operations are synchronized to CCLK. The entire start-up sequence lasts eight cycles, called C0-C7, after which the loaded design is fully functional. The default timing for start-up is shown in the top half of figure. The four operations can be selected to switch on any CCLK cycle C1-C6 through settings in the Xilinx Development Software. Heavy lines show default settings. The bottom half of figureshows another commonly used version of the start-up timing known as Sync-to-DONE. This version makes the GTS, GSR, and GWE events conditional upon the DONE pin going High. This timing is important for a daisy chain of multiple FPGAs in serial mode, since it ensures that all FPGAs go through start-up together, after all their DONE pins have gone High. Sync-to-DONE timing is selected by setting the GTS, GSR, and GWE cycles to a value of DONE in the configuration options. This causes these signals to transition one clock cycle after DONE externally transitions high.



Start-Up Waveforms

Chapter 4

WATCHDOG TIMER AND

IT’S APPLICATION IN ATM

4.1 Introduction

Today, microcontrollers are being used in harsh environments where electrical noise and electromagnetic interference (EMI) are abundant. In environments like this, it is beneficial if the system contains resources to help ensure proper operation. In many systems, a commonly used technique for verifying proper operation is the incorporation of a watchdog timer. A watchdog timer is fundamentally a time measuring device that is used in conjunction with, or as part of, a microprocessor and is capable of causing the microprocessor to be reset. In a properly designed system, the watchdog will cause a reset when the microprocessor is not operating correctly, thereby eliminating the faulty condition. In a typical application, the watchdog timer is configured to reset the processor after a predetermined time interval. If the processor is operating correctly, it will restart the watchdog before the end of the interval. After being restarted, the watchdog will begin timing another predetermined interval. If the watchdog is not restarted by the processor before the end of the interval, a watchdog timeout occurs. This results in the processor being reset. If the system software has been designed correctly, and there has been no hardware failure, the reset will cause the system to operate properly again. Of course, the reset condition must be a safe state. For instance, it would not be wise to have the reset state of a disk drive controller enabling the write head. Many systems have been designed using an external watchdog timer. The need for this additional external component is eliminated, however, with the DS80C320. The DS80C320 contains its own, very capable internal watchdog timer. The features and the use of this watchdog timer are the subject of this application note.

Watchdog timers may be more complex, attempting to save debug information onto a persistent medium; i.e. information useful for debugging the problem that caused the fault. In this case a second, simpler, watchdog timer ensures that if the first watchdog timer does not report completion of its information saving task within a certain amount of time, the system will reset with or without the information saved. The most common use of watchdog timers is in embedded systems, where this specialized timer is often a built-in unit of a microcontroller.

Watchdog timers may also trigger control systems to move into a safety state, such as turning off motors, high-voltage electrical outputs, and other potentially dangerous subsystems until the fault is cleared.

For example, a watchdog timer can be implemented with a x-bit counter in a system working with a clock signal of y MHz, therefore, the system will shut down if the timer is not reset in a period of $\frac{2^x}{y \cdot 10^6}$ seconds.

4.2General Use of Watchdog Timer

The primary application of a watchdog timer is as a system monitor (as discussed in detail in the section below). With a watchdog timer, a system can be designed to be very good at detecting and correcting an out-of-control microprocessor. A system using a watchdog timer is particularly well suited to detecting bit errors. Momentary bit errors can be caused by such things as soft memory failures and electromagnetic discharges into memory devices and their interfaces. These can cause temporary bit polarity flipping of data into and out of the processor. When this occurs while fetching program information, the microprocessor will begin executing erroneous code. Potentially, the processor could begin executing operands instead of op-codes. When the processor begins executing this bad code, it will not properly execute the code that restarts the watchdog. After the timeout interval, the watchdog will cause a processor reset. In a properly designed system, the reset will correct the error. Regardless of how capable a watchdog timer might be, it cannot resolve all reliability issues. There are certain failures that cannot be corrected by a reset. For instance, a watchdog timer cannot prevent the corruption of data. In its basic form, the watchdog restart is dependent on proper program execution, and generally, does not depend on the values in data memory. Unless corruption of data affects program flow or some extra measures are taken, data corruption will not cause a watchdog timeout. Of course, self diagnostic software can be written in such a way as to make restarting the watchdog contingent on verification of data memory. While this approach can be very effective and is quite common, it is beyond the scope of this document to discuss in detail.

Also note that a watchdog timer cannot detect a fault instantaneously. By definition, the watchdog timer must reach the end of a predetermined time interval before it resets the processor. This fact explains why a minimum possible timeout interval should be selected. In this way, a minimum amount of time expires before an out-of-control condition is corrected.

The Watchdog as a System Supervisor

The most common use of the High-Speed Micro's watchdog timer is as a system supervisor. While it can be used in a number of different ways, some of which will be discussed in this document, system supervisor is the most common application. In system supervisor mode, the timer is restarted periodically by the processor as described above. If the processor runs out of control, the watchdog will not be restarted; it will timeout, and subsequently will cause the processor to be reset.

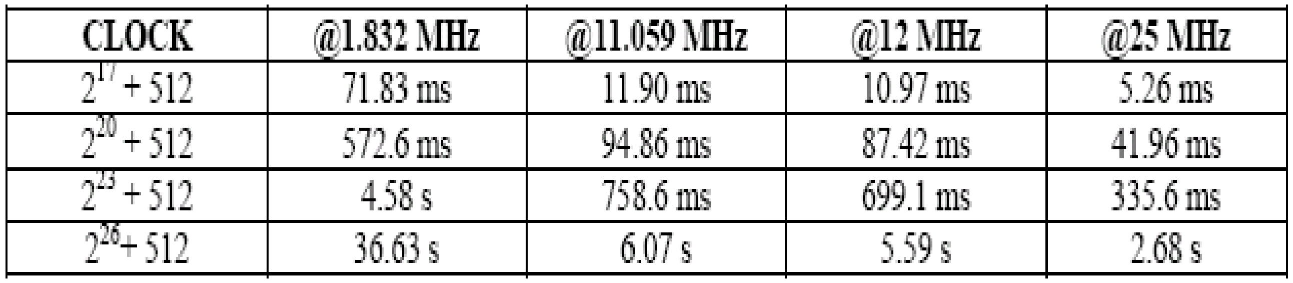
In the High-Speed Micro, the watchdog timer is driven by the main system clock that is supplied to a series of dividers. The divider output is selectable, and determines the interval between timeouts. When the timeout is reached, an interrupt flag will be set, and if enabled, a reset will occur 512 clocks later. The interrupt flag will cause an interrupt to occur if its individual enable bit is set and the global interrupt enable is set. The reset and interrupt are completely discrete functions that may be acknowledged or ignored, together or separately for various applications.

When using the watchdog timer as a system monitor, the watchdog’s reset function should be used. If the interrupt function were used, the purpose of the watchdog would be defeated. To explain, assume the system is executing errant code prior to the watchdog interrupt. The interrupt would temporarily force the system back into control by vectoring the CPU to the interrupt service routine. Restarting the watchdog and exiting by an RETI or RET would return the processor to the lost position prior to the interrupt. By using the watchdog reset function, the processor is restarted from the beginning of the program, and thereby placed into a known state.

This is not to say that the DS80C320 watchdog’s interrupt function is not useful for the system monitor application. Since the reset occurs 512 clocks after the interrupt, a short interrupt service routine can be used to store critical variables before the reset occurs. This may allow the system to return to proper operation in a state that more closely resembles the conditions before the failure. Of course, if the data is the source of the error, storing it without correction would be of no benefit. For any specific system, the approach taken is a function of the system and the level of reliability required.

As mentioned above, the watchdog timer in the DS80C320 is driven by the main system clock that is passed through a series of dividers. The divider output may be selected by the user, allowing a timeout of 217, 220, 223, or 226 clocks. If enabled, a reset of the processor will occur 512 clocks later. Table 1 shows the reset time intervals associated with different crystal frequencies.

DS80C320 WATCHDOG RESET TIME INTERVALS Table 1



CLOCK	@1.832 MHz	@11.059 MHz	@12 MHz	@25 MHz
$2^{17} + 512$	71.83 ms	11.90 ms	10.97 ms	5.26 ms
$2^{20} + 512$	572.6 ms	94.86 ms	87.42 ms	41.96 ms
$2^{23} + 512$	4.58 s	758.6 ms	699.1 ms	335.6 ms
$2^{26} + 512$	36.63 s	6.07 s	5.59 s	2.68 s

As can be seen, there is a range of timeout intervals available. The interval selected should be based on several issues. The first objective is to select an interval that represents the maximum time the processor can be allowed to run out of control. For example, a system that issues a position command to a robotic arm every 500 milliseconds ideally would not use a timeout interval greater than this. Keeping the timeout interval shorter ensures that there will be at most one bad command issued to the arm.

The other primary concern in setting the watchdog timeout interval is the ability to locate the restart commands within the system software. This can be a very complicated issue, depending on the nature of the system software. The most desirable approach is to have a single location within a single main loop of the system software that restarts the watchdog timer. The time required to pass through the main program loop will determine the required timeout interval.

The above approach assumes that the system software flow is linear enough to allow it. Some programs are too convoluted and their flow is too non-linear to allow this approach. With a program structure like that, it is difficult to locate the correct points for watchdog restarts. One possible solution to this problem is to use the DS80C320's watchdog timer itself to assist in determining the appropriate restart locations. This method uses the watchdog's interrupt capability and is described in detail in a section below.

In some systems, the software is too complex or the program flow is too variable to allow a complete and thorough analysis. It may be impossible to determine that all program paths have been covered by a watchdog restart. In this case, a different approach may be used. In this case, diagnostic software may be developed to test the system. This diagnostic software will be called at periodic intervals, perhaps using the interrupt feature of the watchdog timer. If the diagnostics pass, the watchdog is restarted. If not, the watchdog times out and the processor is reset. Of course in this case, the test must be thorough enough to be effective. The exact approach used in a given system may be any of the above or some combination of each, as appropriate for the application.

WATCHDOG RESET EXAMPLE

A short program illustrating most of the basic watchdog timer functions is shown below. This program illustrates how to initialize the watchdog timer so that when it times out, it will cause a reset. The program illustrates one of the DS80C320 watchdog timer's unique features. Software that changes the watchdog's operation must perform a timed access operation. A timed access operation is a sequence of steps that must be executed together in sequence; otherwise the access fails. The example program shows the timed access being used for restarting the watchdog and enabling its reset. As can be seen, the value 0AAh is first written to the timed access register (TA). Next, the value 055h is written to the TA register. Finally, the protected bit is modified. These instructions must be executed in the sequence shown with no interruption to gain access to the protected bit. Further details on timed access operation may be found in the High-Speed Micro User's Guide. The watchdog timer bits that are protected by the timed access procedure are the Enable Watchdog Timer Reset (EWT = WDCON.1) bit, the Watchdog Interrupt Flag (WDIF = WDCON.3) bit, and the Restart Watchdog Timer (RWT = WDCON.0) bit.

```
WD_RST.ASM Program
;;
This program demonstrates the use of the watchdog timer in
; the DS80C320. It uses the timer’s reset capability. When
; running, the program sets port 1’s pins low to indicate
; the processor is idle waiting for the watchdog to timeout. When
; the watchdog times out, the processor is reset causing the port
; pins to return high. A delay is written into the program so that
; the port pins will be high long enough to be seen if attached to
; LEDs.
;
; * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;;
Reset Vector
;
ORG 00h
SJMP START
;
; * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;;
Main program body
;
ORG 080h
;
START: ORL CKCON, #080h ; Set Watchdog timeout period 2**23
; (approximately 758 mS @ 11.059 MHz)
;;
In a real application, the next three lines would be placed
; at various locations in the program to restart the watchdog
; before it times out.
;
MOV TA, #0AAh ; Restart Watchdog timer
MOV TA, #055h ; using timed
SETB RWT ; access.
;;
MOV TA, #0AAh ; Enable Watchdog timer reset
MOV TA, #055h ; using timed
SETB EWT ; access.
;;
MOV R1, #0FFh ; Create a delay loop so the port
LOOP: MOV R2, #0FFh ; pins are high long enough after
DJNZ R2, $ ; a reset to be seen.
DJNZ R1, LOOP
;
MOV P1, #00 ; P1 = 0, Reset causes P1 = 1
;
MOV PCON, #01h ; Go to idle mode waiting for reset
SJMP $
;;
; * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;
END
```

4.3THE WATCHDOG TIMER AS A LONG INTERVAL TIMER

A slightly different application of the High-Speed Micro's watchdog timer is as a long interval timer. In this application, the interrupt is enabled using the Enable Watchdog Timer Interrupt (EWDI=EIE.4) bit and the reset is left disabled. When the timeout occurs, the Watchdog Timer will set the WDIF bit (WDCON.3), and an interrupt will occur if the global interrupt enable bit (EA=IE.7) is set. The Watchdog Interrupt Flag will indicate the source of the interrupt and must be cleared by software. As shown in the table above, intervals from 5.26 ms to 2.68 seconds are available with a 25 MHz crystal. This interval is significantly longer than any possible using the standard 16-bit timers. Another short program illustrating features of the watchdog timer is shown below. This program demonstrates how the watchdog timer and interrupts must be initialized so that a timeout causes an interrupt. A short interrupt service routine is included.

```
; * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;;;
WD_INT.ASM Program
;;
This program demonstrates the use of the watchdog timer of
APPLICATION NOTE 80
5 of 6
; the 80C320. It uses the timer's interrupt generating capability.
; For purposes of demonstration, the program toggles Port 1's pins
; each time the watchdog's Interrupt Service Routine is entered.
;;
$MODS320
;
; * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;;
Reset Vector
;
ORG 00h
SJMP START
;
; * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;;
Watchdog Interrupt Vector
;
ORG 063h
;
MOV TA, #0AAh ; Restart watchdog timer
MOV TA, #055h ; using timed
SETB RWT ; access.
;
```

```
MOV TA, #0AAh ; Clear watchdog interrupt flag
MOV TA, #055h ; using timed
CLR WDIF ; access.
;
CPL A ; Complement port 1 to show the
MOV P1, A ; interrupt routine was entered.
;
RETI ; Return from interrupt.
;;
; * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;;
Main program body
;
ORG 080h
;
START:
ORL CKCON, #040h ; Set Watchdog timeout period 2**20
; (approximately 94.8 mS @ 11.059 MHz)
;
MOV TA, #0AAh ; Restart Watchdog timer
MOV TA, #055h ; using timed
SETB RWT ; access.
;
SETB EWDI ; Enable Watchdog Interrupt and
SETB EA ; set global interrupt enable
;
Here: MOV PCON, #01 ; Go to Idle mode and wait
SJMP Here ; After interrupt, go back to idle
;;
; * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;
END
```

THE WATCHDOG TIMER AS AN AID IN LOCATING RESTART INSTRUCTIONS

As discussed above, locating the watchdog restart instructions in the system software can sometimes be difficult. The structure of the system software and the complexity of its flow determine the task's level of difficulty. In the DS80C320, the watchdog timer itself can be used to assist in this activity. The general approach for this is to allow the watchdog to cause an interrupt and, from within the service routine, determine where in the code the interrupt occurred. By placing the watchdog restart instructions prior to this point you can be assured that the watchdog will be restarted before the timeout (when the software flow follows this particular branch). This process is repeated until no more watchdog interrupts occur. If the program flow is linear and not data-dependent, the system will function as desired. The previous software example provides most of the software necessary to perform this function. As a first step, however, the desired maximum timeout interval should be determined and the code modified for this value. As always, the timeout selected is a function of the system and how long the micro an be allowed to run out of control. After modifying the software to initialize the desired watchdog timeout interval, the following instructions should be added to the Interrupt Service Routine. They will cause the processor to display the address of the instruction that would have been executed if the interrupt had not occurred. If this display mechanism is not convenient for the system implementation, the address can be converted to ASCII and output on one of the serial ports.

```
MOV R0, SP ; Get SP contents
MOV P3, @R0 ; Display high address byte
DEC R0 ; Point to low address byte
MOV P1, @R0 ; Display low address byte
SJMP $ ; Stop here
```

The instructions above move the contents of the Stack Pointer to R0, which is then used to point to the data pushed onto the stack when the interrupt was acknowledged. This address reflects the next instruction that would have been executed if the interrupt had not occurred. The high byte of the address is displayed on Port 3 pins, and the low byte of the address is displayed on Port 1 pins. If instructions to restart the watchdog timer are placed before this address, the watchdog will never reach timeout.

4.4 <u>APPLICATION OF WATCHDOG TIMER IN ATMs</u>

An automated teller machine or automatic teller machine (ATM) is a computerized telecommunications device that provides a financial institution's customers a method of financial transactions in a public space without the need for a human clerk or bank teller. On most modern ATMs, the customer identifies him or herself by inserting a plastic ATM card with a magnetic stripe or a plastic smartcard with a chip, that contains his or her card number and some security information, such as an expiration date or CVC (CVV). Security is provided by the customer entering a personal identification number (PIN).

A major issue related with ATMs is that related to its timing control.That is what happens if a person enters his card and doesn’t enter any value?????Won’t it be a test of patience for the customers standing in the long queue??? Here the watchdog timer comes to the rescue. The system waits for pre defined time period and if it exceeds that time period without any input to it,then it indicates that using some indicator. The indicator remains high as long as there is no input.And once some input comes, the timer is reset and the indicator is turned off.Normal operation can resume after that.

So, watchdog timer is indeed a very useful component and has got its application ranging over various fields. Its application in ATMs is one such field which we have studied and designed as a part of our project.

Chapter 5

EXPERIMENTATION & RESULTS

5.1 Coding

The code is written in VHDL. The simulator used is Xilinx and the FPGA used is Spartan-II,about which we have already discussed in previous chapters.The experiment can be classified into the following stages:-

- Getting acquainted with VHDL-The various knowledge required for operating VHDL like knowing about its syntax, architecture components etc. was studied and practiced by performing simpler examples.

- Learning the required circuits: The various circuits of timers,counters etc was studied from various digital books and rough diagram of the circuit was prepared.

- Writing the program code :Then the required code for the timer was written using VHDL and its errors were checked for.

- Simulating the code using xilinx:Then using xilinx simulator, the written code was simulated .The various errors and warning were taken care of.And finally the test bench waveform was obtained and studied.

- Burning the code on FPGA:Then the code was burned using the Spartan-II kit and various necessary methods were carried out till we get the required output on the FPGA board.

5.2 Source Code

```
-------------------------------------------------------------------------------
            library IEEE;

            use IEEE.STD_LOGIC_1164.ALL;

            use IEEE.STD_LOGIC_ARITH.ALL;

            use IEEE.STD_LOGIC_UNSIGNED.ALL;

            entity testclk is

            Port (

                        clkin: in STD_LOGIC; ---------the clk of FPGA

                        reset : in STD_LOGIC;-----------the reset

                        exin : in std_logic;----------the input to the system

                        msg : out std_logic;----------the output signallin lack of input

               clkout : out  STD_LOGIC );--------------divided clk

            end testclk;

            architecture Behavioral of testclk is

            signal toggle : std_logic := '0';

            signal xu : std_logic_vector(22 downto 0);

            begin
```

```
        process(clkin,reset,exin)
        begin
        if reset = '1' then
                xu <= (others => '0');
                clkout <= '1';

        elsif rising_edge(clkin) then

                        xu <= xu+1;
                        if  exin = '1' then
                        xu <= (others => '0');
                        msg <= '0';--resettin the output
                        end if;

                        if exin = '0' and xu = "11111111111111111111111" then
                        msg <= '1';--settin the interrupt
                        end if;

                        if (xu = "11111111111111111111111") then
                        toggle <= not toggle;--for toglin the clk out
                        clkout <= toggle;
                        xu <= (others => '0');
                end if;

        end if;


        end process;


        end Behavioral;
clear all;
```

6.3 Conclusion

We successfully designed a timer for ATM application and observed its output, both as test bench waveform and on Spartan-II kit.

REFERENCES

1.”Digital Logic Circuits using VHDL” by Stephen Brown..

2. "VHDL for Programmable Logic" by Kevin Skahill, Addison-Wesley

3."A Designer's Guide to VHDL Synthesis" by Douglas E. Ott and Thomas J. Wilderotter, Kluwer Academic Publishers

4. "VHDL Made Easy" by David Pellerin and Douglas Taylor.

5. "The Student's Guide to VHDL" by Peter J. Ashenden, Morgan Kauffman.

6. "VHDL Starter's Guide" by Sudhakar Yalamanchili