IMPLEMENTATION OF A RISC MICROCONTROLLER USING FPGA

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

RAŞİT GÜMÜŞ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

SEPTEMBER 2005

Approval of the Graduate School of Natural and Applied Sciences

Prof.Dr. Canan ÖZGEN
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof.Dr. İsmet ERKMEN
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Prof. Dr. Hasan GÜRAN
Supervisor

Examining Committee Members

Asst.Prof.Dr. Cüneyt BAZLAMACI (METU,EEE) _____

Prof. Dr. Hasan GÜRAN (METU,EEE) _____

Asst.Prof.Dr. İlkay ULUSOY (METU,EEE) _____

Dr. Ece SCHMIDT (METU,EEE) _____

Ekrem ARAS , Msc (MiKES A.Ş.) _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name: Raşit GÜMÜŞ

Signature :

ABSTRACT

IMPLEMENTATION OF A RISC MICROCONTROLLER USING FPGA

GÜMÜŞ, Raşit

MSc., Department of Electrical and Electronics Engineering

Supervisor: Prof. Dr. HASAN GÜRAN

June 2005, 88 pages

In this thesis a microcontroller core is developed in an FPGA. Its instruction set is compatible with the microcontroller PIC16XX series by Microchip Technology. The microcontroller employs a RISC architecture with separate busses for instructions and data. Our goal in this research is to implement and evaluate the design in the FPGA. Increasing performance and gate capacity of recent FPGA devices permits complex logic systems to be implemented on a single programmable device. Such a growing complexity demands design approaches, which can lead to designs containing millions of logic gates, memories, high-speed interfaces, and other high-performance components. In recent years, the continuous development in the area of highly integrated circuits has lead to a change in the design methods used, making it possible to economically utilize FPGAs in many designs.

A test demo board from the Digilent Inc is used to fit our testing requirements of the RISC microcontroller. The test demo board also had the capability of communicating with a personal computer (PC) so that we can load the program from PC. Based on the modern design methods the microcontroller core is developed using the Verilog hardware description language. Xilinx ISE

Foundation 6.3i software is used for its synthesis and implementation. An embedded test program code using MPLAB is also developed, and then loaded into the designed microcontroller residing in the FPGA. In order to perform a functional test of the microcontroller core a special test program downloader application is designed by using Borland C++ Builder.

First, the specification from the PIC16XX datasheet is transferred into an abstract behavioral description. Based on that, the next step is to develop a description of the microcontroller core with some minor modifications which can be synthesizable into a FPGA. Finally, the resulting gate level netlist is evaluated and tested using a demo board.

Keywords: RISC, CISC, Microcontroller, PIC, Field Programmable Gate Arrays, Xilinx, Verilog

ÖZ

FPGA KULLANARAK RISC MIKRODENETLEYİCİ GERÇEKLEŞTİRMESİ

GÜMÜŞ, RAŞİT
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Hasan GÜRAN

Haziran 2005, 88 sayfa

Bu tezde bir mikrodenetleyici çekirdeği geliştirilmiş ve gerçekleştirilmiştir. Mikrodenetleyicinin komut kümesi, Microchip firmasının PIC16 serisi mikrodenetleyicileri ile uyumludur. Bu mikrodenetleyicide RISC mimarisi kullanılmış olup, veri yolu ve komut kütüphanesi veri yolu ayrıdır. Bu araştırmadaki amacımız, mikrodenetleyicinin FPGA üzerinde tasarlanması ve gerçekleştirilmesidir. Günümüzdeki FPGA'lerin hem performans hemde lojik kapı kapasitesinin gelişmiş olması, karmaşık sistemlerin tek bir programlanabilir enntegrelerde gerçekleştirilmelerine imkan vermiştir. Bu gittikçe artan karmaşık sistemler, tasarımların milyonlarca lojik kapı, hafıza, yüksek hızlı arayüz ve diğer yüksek performanslı bileşenler içeren bir tasarım yaklaşımı istemektedir. Son yıllardaki yonga teknolojisindeki sürekli gelişmeler, tasarım metodlarının değişmesine sebeb olmuştur, bu da FPGA'lerin ekonomik olarak birçok tasarımda kullanılmalarına olanak sağlamıştır.

Tasarladığımız mikrodenetleyicinin test ihtiyaçları için Digilent firmasının bir demo kartı kullanılacaktır. Bu demo kartı bilgisayar ile haberleşebilme özelliğine sahip olduğundan, tasarladığımız gömülü yazılımı FPGA üzerine

yükleyebilmemize olanak sağlamaktadır. Mikrodenetleyici çekirdeği günümüz modern tasarım metodlarını baz alarak, Verilog donanım tanımlama dilini kullanarak geliştirilmiştir. Xilinx firmasının ISE Foundation 6.3i yazılımı sentezleme ve gerçekleştirme işlemlerinde kullanılmıştır. Ayrıca bir gömülü test yazılımı MPLAB kullanarak yazılıp, FPGA'e yüklenmiştir. Mikrodenetleyici çekirdeğinin , fonksiyonel testlerinin yapılabilmesi için, PC'den FPGA'e gömülü yazılım yüklemek için , Borland C++ Builder kullanarak , bir program yükleme yazılımı da geliştirilmiştir.

İlk once PIC16XX veri sayfalarından tasarım belirtimleri , donanım hareket betimlerine dönüştürülmüştür. Bundan sonraki adım, FPGA üzerine çok az bir değişiklikle sentezlenebilir bir mikrodenetleyici çekirdeğinin geliştirilmesi olmuştur. Son olarak kapı seviyesinde oluşturulan bağlantı listesi, demo kartı kullanılarak test edilmiştir.

Anahtar Kelimeler : RISC, CISC, Mikrodenetleyici, PIC, Saha Programlanabilir Kapı Dizisi, Xilinx, Verilog

To My Parents

ACKNOWLEDGEMENTS

I would like to thank Prof. Dr. Hasan GÜRAN for his valuable supervision and support throughout the development and improvement of this thesis. This thesis would not have been completed without his guidance.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
BUFG	Buffer Global
CAD	Computer Aided Design
CISC	Complex Instruction Set Computer
CLB	Configurable Logic Block
CPU	Central Processing Unit
DLL	Delay Locked Loop
DSP	Digital Signal Processor
EDIF	Electronic Design Interchange Format
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
I/O	Input / Output
IBUF	Input Buffer
IC	Integrated Circuit
IDE	Integrated Development Environment
IP	Intellectual Property
ISE	Integrated Software Environment
JTAG	Joint Test Action Group
LUT	Look Up Table
MCU	Microcontroller Unit
OTP	One Time Programmable
PLD	Programmable Logic Devices
PAR	Place and Route
RAM	Random Access Memory

RISC	Reduced Instruction Set Computer
ROM	Read-Only Memory
RTL	Register Transfer Level
SFR	Special Function Register
UCF	User Constrains File
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

CHAPTER 1

INTRODUCTION

The aim of this thesis is to design the complete processor core of Microchip PIC16XX and slightly modify its architecture and instruction set. The designed microcontroller will be implemented by using an FPGA.

1.1. Central Processing Unit

The central processing unit is the brain of the computer system that manages the flow of information. A central processing unit normally contains three main components: a control unit, an arithmetic and logic unit and a register collection. It is the control unit which is responsible for the control and synchronization of the actions of the processor. Thus, the control unit is the most complicated part of the system, and the one which characterizes the CPU. Figure 1.1 shows the block diagram of a basic computer system. A basic computer system must have the standard elements CPU, memory and I/O. All these elements communicate via the system bus, which is composed by the data, address buses [1].

The CPU has the ability to understand and execute instructions based on a set of binary codes, each representing a simple operation. These instructions are usually arithmetic, logic, data movement, or branch operations, and are represented by a set of binary codes called the instruction set. The memory, is used to store all the programs formed by the instruction set and all the require data. I/O interface provide an interconnection with the outside world, such as the keyboard as an input and the monitor as an output.

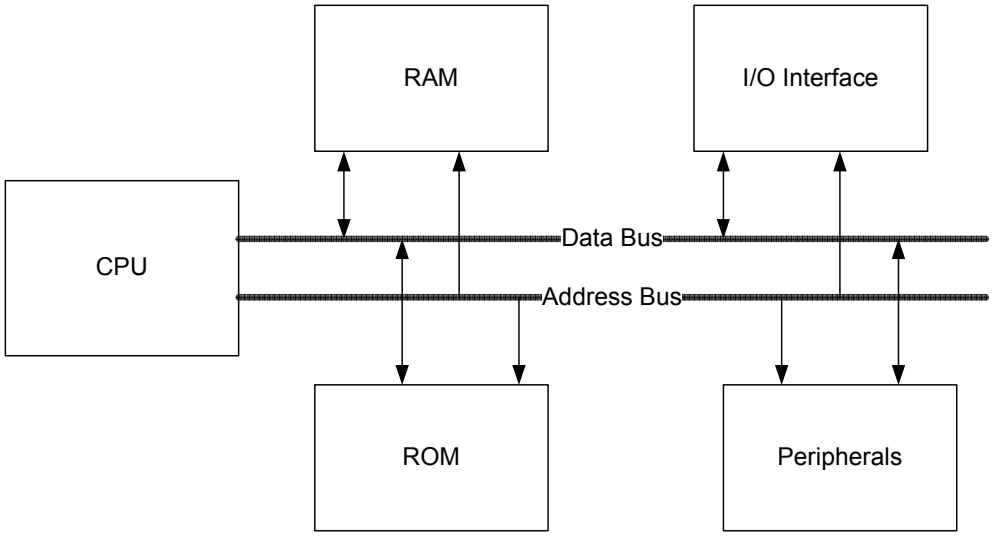


Figure 1.1. Basic Computer System Architecture

Minicomputers and mainframe computers, have CPUs consisting multiple ICs, ranging from several ICs (minicomputers) to several circuit boards of ICs (mainframes). This is necessary to achieve the high speeds and computational power of larger computers. On the other hand, the CPU of a microcomputer is contained in a single integrated circuit. They are known as a microprocessor [2].

1.2. Microcontroller

It was pointed out above that microprocessors are single-chip CPUs used in microcomputer. A microcontroller contains, in a single IC, a CPU and much of the remaining circuitry of a basic computer system. A microcontroller has the CPU, memory (RAM, ROM) and the I/O interface (parallel, serial) all within the same IC. Of course, the amount of on-chip memory does not approach that of even a modest microcomputer system [3].

Microprocessors are most commonly used as the CPU in microcomputer systems. Microcontrollers, on the other hand, are found in small, minimum-component

designs performing control-oriented activities. These designs were often implemented in the past using dozens or even hundreds of ICs. A microcontroller aids in reducing the overall component count. All that is requires is microcontroller, a small number of support components, and a control program in ROM.

There are two fundamental microcontroller architectures to access memory in the industry.

John Von Neumann's Architecture: One shared memory for instructions (program) and data with one data bus and one address bus between processor and memory. Instructions and data have to be fetched in sequential order (known as the Von Neuman Bottleneck), limiting the operation bandwidth. Its design is simpler than that of the Harvard architecture. It is mostly used to interface to external memory. Examples of processors using this type of architecture are the Motorola MC68HC11 and Intel 8051 [3].

Harvard Architecture: The Harvard architecture uses physically separate memories for their instructions and data, requiring dedicated buses for each of them. Instructions and operands can therefore be fetched simultaneously.This type of architecture speeds up execution but requires more silicon. PIC microcontrollers from Microchip Technology Inc. use this type of architecture.
Different program and data bus widths are possible, allowing program and data memory to be better optimized to the architectural requirements. E.g.: If the instruction format requires 14 bits then program bus and memory can be made 14-bit wide, while the data bus and data memory remain 8-bit wide[3].

1.3. Complex Instruction Set Computer (CISC)

In early days, computers had only a small number of instructions and used simple instruction sets, forced mainly by the need to minimize the hardware used to implement them. As digital hardware become cheaper, computer instructions tended to increase both in number and complexity. These computers also employ a variety of data types and a large number of addressing modes. A computer with a large number of instructions, are known as complex instruction set computer, abbreviated CISC [3].

Major characteristics of CISC architecture are:

- A large number of instructions – typically from 100 to 250 instructions
- Some instructions that perform specialized tasks and are used infrequently
- A large variety of addressing modes – typically from 5 to 20 different modes
- Variable-length instruction formats
- Instructions that manipulate operands in memory

1.4. Reduced Instruction Set Computer (RISC)

In the early 1980s, a number of computer designers were questioning the need for complex instruction sets used in the computer of the time. In studies of popular computer systems, almost 80% of the instructions are rarely being used. So they recommended that computers should have fewer instructions and with simple constructs. This type of computer is classified as reduced instruction set computer or RISC. The term CISC is introduced later to differentiate computers designed using the ‘old’ philosophy. The first characteristic of RISC is the uniform series of single cycle, fetch-and-execute operations for each instruction implemented on the computer system.

A single-cycle fetch can be achieved by keeping all the instructions a standard size. The standard instruction size should be equal to the number of data lines in the system bus, connecting the memory (where the program is stored) to the CPU. At any fetch cycle, a complete single instruction will be transferred to the CPU. For instance, if the basic word size is 32 bits, and the data port of the system bus (the data bus) has 32 lines, the standard instruction length should be 32-bits.

Achieving uniform (same time) execution of all instructions is much more difficult than achieving a uniform fetch. Some instructions may involve simple logical operations on a CPU register (such as clearing a register) and can be executed in a single CPU clock cycle without any problem. Other instructions may involve memory access (load from or store to memory, fetch data) or multicycle operations (multiply, divide, floating point), and may be impossible to be executed in a single cycle [3].

The characteristics of RISC architecture are summarized as follow:

- Single-cycle instruction execution
- Fixed-length, easily decoded instruction format
- Relatively few instructions
- Relatively few addressing modes
- Memory access limited to move instructions
- All operations done within the RAM and working register of the CPU

1.5. Microchip PIC16XX

The Microchip PIC family of microcontrollers was introduced in 1989 by Arizona Microchip. Microchip (as they are now known) bought General Instruments' microelectronics division as a start-up company in 1988. They re-engineered a programmable interface device that General Instruments were using as a general-

purpose reconfigurable input/output port for their microprocessor as a stand-alone microcomputer. These were called the PIC (Programmable Interface Controller) family. The second generation of this family was introduced in 1994, which are PIC16XX family. The core processor is similar within the 14-bit family members and software is identical. PIC16XX based on a RISC architecture which has 33 instructions [4].

1.6. Objectives

The main objective of this project is to design a RISC microcontroller using verilog and implement it in an FPGA. The microcontroller instruction set and the basic features are based on Microchip PIC16XX RISC microcontroller family. The objective also includes the architecture expansion of the microcontroller without changing the core structure.

1.7. Work Scope

The aim of the project is to design the complete processor core of Microchip PIC16XX and slightly modify its architecture and instruction set. The microcontroller must be able to fit into the targeted FPGA device, which is Xilinx Spartan IIE Digilent evaluation board.

CHAPTER 2

DESIGN PROCESS FLOW AND TOOLS

2.1. Design Process

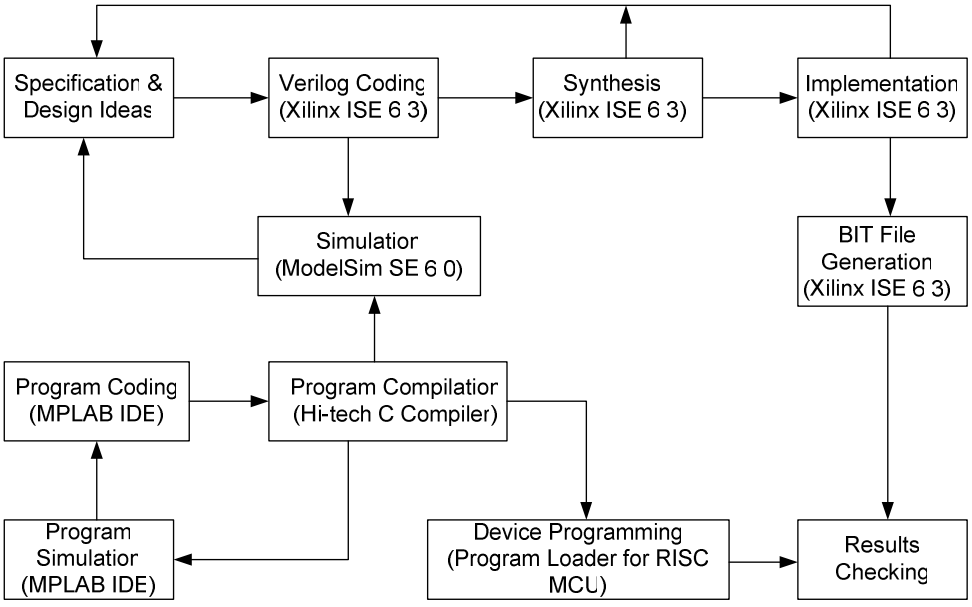


Figure 2.1. Design Process Flow

Figure 2.1 shows the design process of the project and their related CAD tools. The design process can be divided into 2 main parts – hardware design (with verilog) and hardware implementation.

Hardware design is done with the related CAD tools. The first step in the hardware design is to prepare the specification of the design (the microcontroller). The architecture and the instruction set must be understood completely. The design ideas are then described with verilog in a text editor. Then, the verilog code is synthesized with XILINX XST. If synthesized successfully, XILINX XST will generate a bit file. This file is then loaded to the FPGA on the demo board. Results are verified by the Digilent D2E board. The hardware design process is repeated until the microcontroller is complete without any errors.

Hardware implementation is performed by loading the design into the targeted FPGA device, Xilinx Spartan XC2S200-6PQ208. The hardware implementation tests the design, in real physical environment by some control applications. A microcontroller can perform thousands of control applications. For every application, different programs must be written and stored into the program ROM of the microcontroller before it can do the job. So, before the microcontroller is downloaded into the FPGA device, the application specific firmware for the microcontroller must be written.

The program is written and assembled using the HI-TECH C compiler. The MPLAB IDE is used to simulate and test the program. If no bugs are found, the binary file generated by the compiler is converted to Intel HEX format. This HEX file is downloaded to the Digilent D2E board by using a program loader application, written by using Borland C++ Builder. After loading the program, microcontroller is checked, whether it meets the design specification.

2.2. Software Tools

2.2.1. XILINX ISE

Integrated Software Environment (ISE) is the Xilinx design software suite [5]. ISE can be used by a full spectrum of designers, from the first time CPLD designer to

the experienced ASIC designer transitioning to FPGA. ISE enables designers to start the design with any of a number of different source types, including:

- HDL (VHDL, Verilog HDL, ABEL)
- Schematic design files
- EDIF
- State Machines
- IP Cores

After the design has been typed the synthesis stage converts the text based design into a Xilinx netlist file, which is a linked object 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 normally 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 first step is translate. The translate step checks the design and ensures the netlist is consistent with the chosen architecture. Translate also checks the user constraints file (UCF) for any inconsistencies. In effect, this stage prepares the synthesized design for use within an FPGA.

The Map stage distributes the design to the resources in the FPGA. Obviously, if the design is too big for the chosen device the map process will not be able to complete its job.

The Place And Route (PAR) stage works with the allocated configurable logic blocks (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 purely to minimize the path

length. The routing resources are then allocated to each connection, again using careful selection of the best possible routing types.

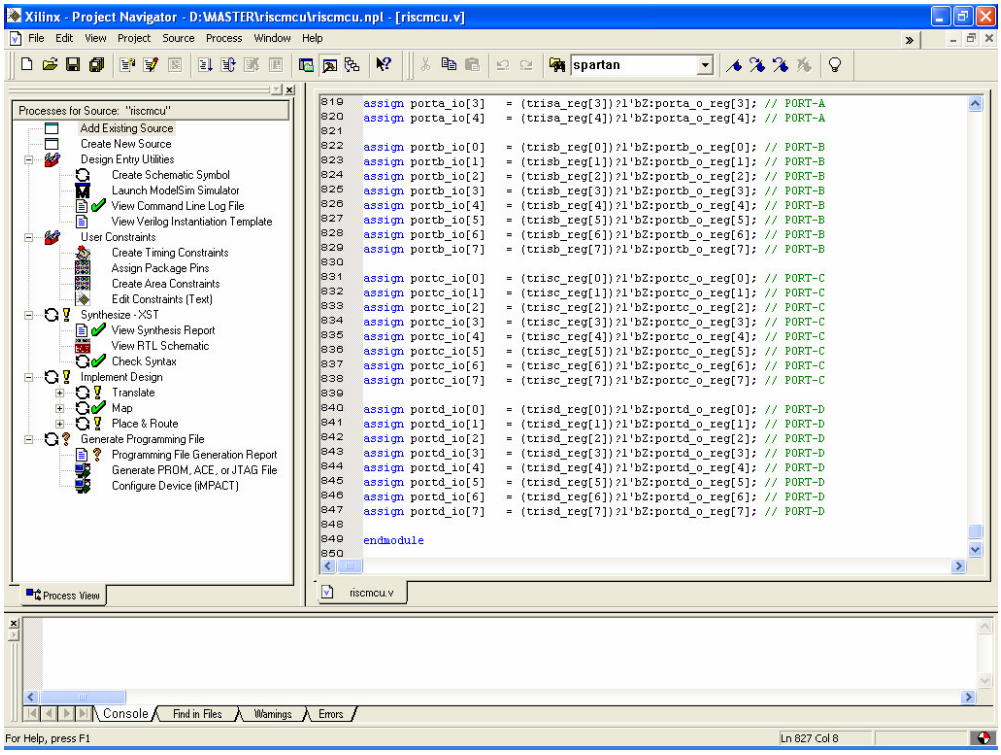


Figure 2.2. Xilinx ISE View

Finally a program called 'bitgen' takes the output of Place and Route and creates a programming bitstream. The generated bit file is ready to download the target FPGA

To implement any design on an FPGA chip, the designer should be aware of the design development tools (i.e., the CAD tools) and the target FPGA technology. An ASIC design that is efficient in terms of area and/or speed for some ASIC tools and technology is not necessarily efficient for some FPGA tools and technology. Same thing applies when considering tools and technologies from different vendors. What is efficient for Xilinx FPGAs might not be efficient for Altera FPGAs. Even this applies to different tools and technologies from the same vendor. For example, a design that is implemented using Xilinx ISE 6.1i tools

from Xilinx and efficient for the XC4000 FPGAs might not be efficient when using Xilinx ISE 7.1 tools and Spartan-II FPGAs as the target technology. So, the key is to understand how to let the tools interpret the design description efficiently and optimize it as much as possible. Also, to understand the target FPGA chip and make good use of its resources. Xilinx ISE Webpack edition can be downloaded from the web site of the Xilinx.

2.2.2. MODELSIM SE

ModelSim is a simulation and debugging tool for VHDL, Verilog, SystemC, and mixed-language designs. Modelsim SE is Mentor Graphics's UNIX, Linux, and Windows-based simulator. It utilizes the Single Kernel Simulator technology to enable VHDL, Verilog and mixed-language simulation. Its other major features include high-performance RTL and gate-level optimizations, Performance Analyzer for accelerating simulations and Waveform Compare advanced debugging feature [6]. The following diagram shows the basic steps for simulating a design in ModelSim.

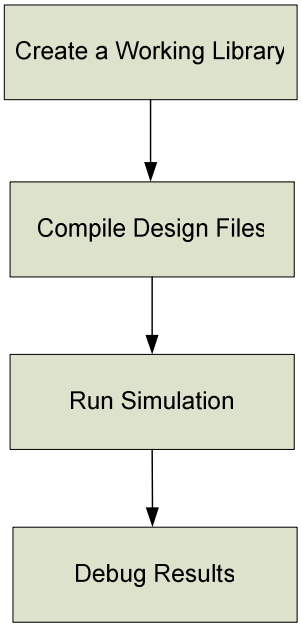


Figure 2.3. Basic Simulation Flow with using Modelsim

2.2.3. MPLAB IDE

MPLAB IDE is a free software program that runs on a PC to develop applications for Microchip microcontrollers and can be downloaded on the Microchips' website [4]. It is called an Integrated Development Environment, or IDE, because it provides a single integrated "environment" to develop code for an embedded microcontroller. MPLAB contains all the components needed to design and to deploy embedded systems applications. The MPLAB IDE allows the embedded systems design engineer to get through the development cycle without the distraction of switching among an array of tools. In MPLAB IDE all the functions are integrated, allowing the engineer to concentrate on the goal of completing the application without getting slowed down dealing with separate tools and their various, different modes of operation.

The project manager is a system that organizes the files to be edited so that they and other associated files can be sent to the language tools for assembly or compilation, and ultimately to a linker. The linker has the task of placing the object code fragments from the assembler, compiler and libraries into the proper memory areas of the embedded controller, and to make sure that the modules function with each other (or are "linked"). This entire operation from assembly and compilation through the link process is called a project "build".

The source files are text files that are written conforming to the rules of the assembler or compiler. The assembler and compiler convert them into intermediate modules machine code and placeholders for references to functions and data storage. The linker resolves these placeholders and combines all the modules into a file of executable machine code. The linker also produces a debug file which allows MPLAB IDE to relate the executing machine codes back to the source files.

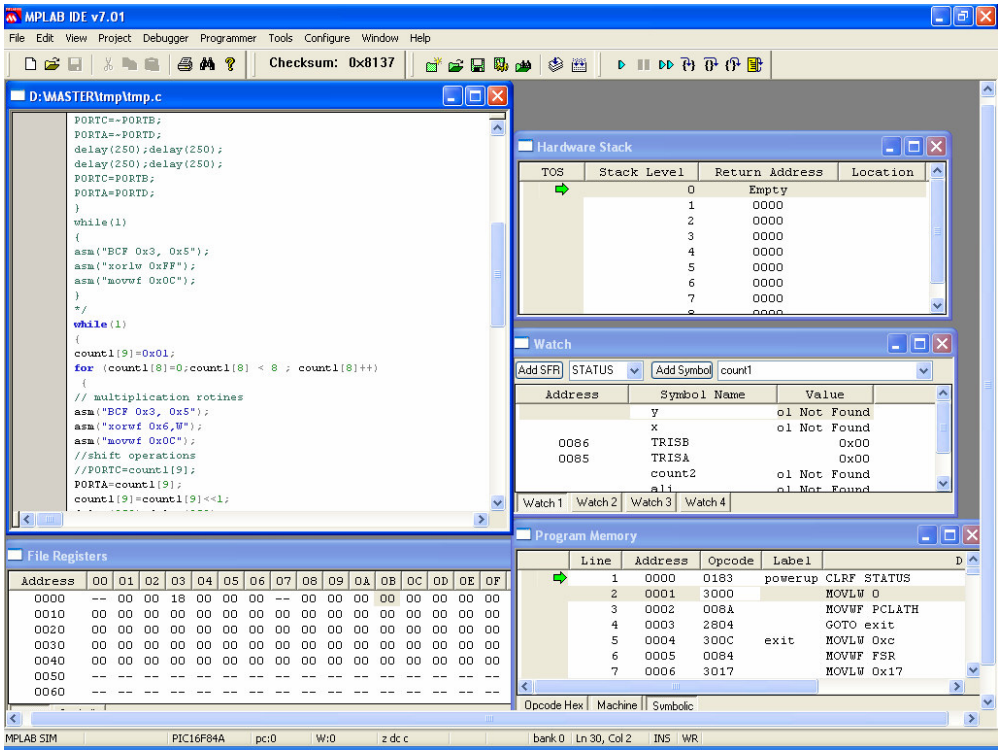


Figure 2.4. A view of the MPLAB program

The text editor recognizes the constructs in the text and uses color coding to identify various elements, such as instruction mnemonics, C language constructs, and comments. The editor supports operations commonly used in writing source code, such as finding matching braces in C, commenting and un-commenting out blocks of code, finding text in multiple files, and adding special bookmarks.

2.2.4. HI-TECH C Compiler

HI-TECH C compiler is one of the most popular high performance C compiler for the Microchip PIC 10/12/14/16/17 series of microcontrollers. HI-TECH PIC C compiler can be fully integrated with MPLAB or can be used directly from a makefile or command line [7]. The test firmware is compiled by this compiler , under the MPLAB IDE. A limited free version of this compiler is available on the website of the HI-TECH.

2.2.5. BORLAND C++ BUILDER

Borland C++ Builder is a rapid programming tool used to create computer applications for the Microsoft Windows operating systems. Borland C++ Builder is based on the C++ computer language with a lot of improvements and customized items [8].

PIC Loader program is created with using Borland C++ Builder. Its main purpose is to read the INTEL-hex format program file, and then to send the program through the RS232 serial channel of the PC to the Digilent demo board.

2.3. Hardware Tools

2.3.1. Digilent D2E Demo Board

The Digilab 2E (D2E) development board featuring the Xilinx Spartan 2E XC2S200E FPGA provides an inexpensive and expandable platform on which to design and implement digital circuits of all kinds [9]. Figure 2.5 shows the picture of the Digilent D2E demo board.

A block diagram of the Digilent demo board can be found in Figure 2.6. D2E board features include:

- A Xilinx XC2S200E FPGA;
- Dual on-board 1.5A power regulators (2.5V and 3.3V);
- A socketed 50MHz oscillator;
- An EPP-capable parallel port for JTAG based FPGA programming and user data transfers;
- A 5-wire Rs-232 serial port;
- A status LED and pushbutton for basic I/O;
- Six 100- mil spaced, right-angle DIP socket 40-pin expansion connectors.

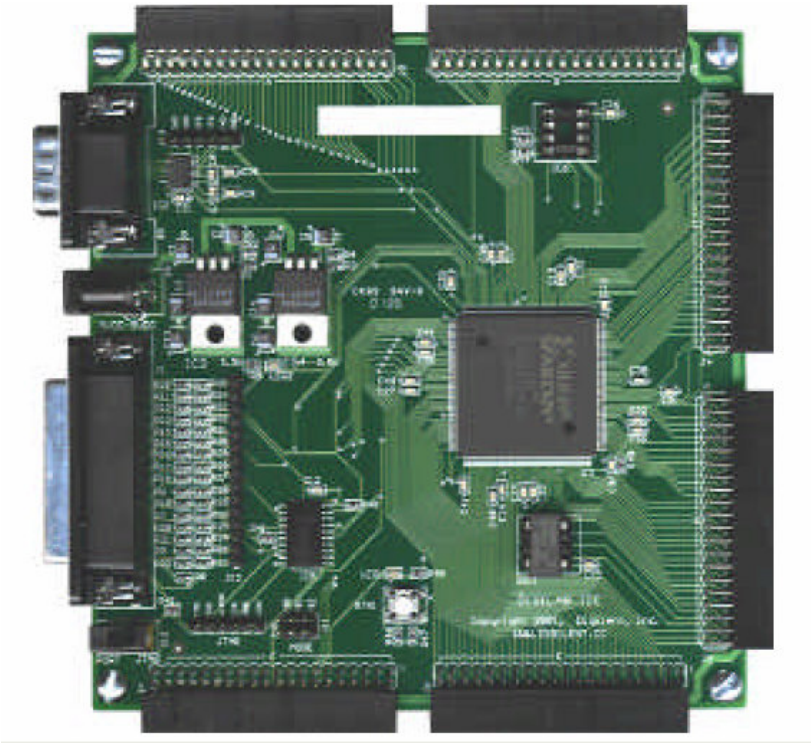


Figure 2.5. Picture of the Digilent D2E Board

The D2E board has been designed specifically to work with the Xilinx ISE CAD tools, including the free WebPack tools available from the Xilinx website.

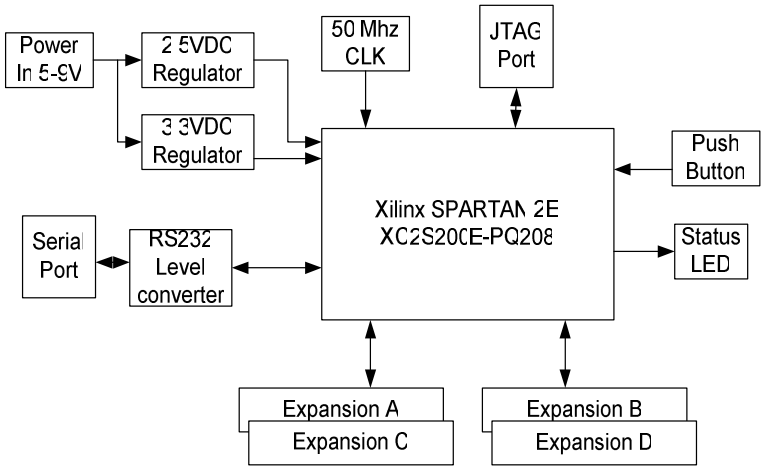


Figure 2.6. Block Diagram of the Digilent Demo Board

2.4. Hardware Description Language

Two major hardware description languages are available for the designers. These are VHDL and Verilog.

2.4.1. VHDL

VHDL is the VHSIC (Very High Speed Integrated Circuit) Hardware Description Language. It can describe the behavior and structure of electronic systems, but is particularly suited as a language to describe the structure and behavior of digital electronic hardware designs, such as ASICs and FPGAs as well as conventional digital circuits [11].

The development of VHDL was initiated in 1981 by the United States Department of Defense to address the hardware life cycle crisis. The cost of reproducing electronic hardware as technologies became obsolete was reaching crisis point, because the function of the parts was not adequately documented, and the various components making up a system were individually verified using a wide range of different and incompatible simulation languages and tools. The requirement was for a language with a wide range of descriptive capability that would work the same on any simulator and was independent of technology or design methodology. The VHDL language was first standardized in 1987 by IEEE as IEEE 1076-1987, and is commonly referred as VHDL-87. This is certainly the most important version, since most of the VHDL tools are still based on this standard. The last revision came to the VHDL in 2002 (IEEE 1076-2002). The definition of the language is non-proprietary [11].

2.4.2. Verilog

The Verilog Hardware Description Language (HDL) describes a hardware design or part of a design. Descriptions of designs in the Verilog HDL are Verilog models. The Verilog HDL is both a behavioral and structural language. Models in

the Verilog HDL can describe both the function of a design and the components and connections to the components in a design [12].

Verilog HDL is first invented by Gateway Design Automation in 1985. Gateway Design Automation grew rapidly with the success of Verilog and was finally acquired by Cadence Design Systems, San Jose, CA in 1989 [12]. Cadence Design Systems decided to open the language to the public in 1990, and thus OVI (Open Verilog International) was born. Until that time, Verilog HDL was a proprietary language, being the property of Cadence Design Systems. The Verilog HDL is an IEEE standard - number 1364. The first version of the IEEE standard for Verilog was published in 1995. A revised version was published in 2001 [13].
The basic building block of the Verilog HDL is the module. The module format facilitates top-down and bottom-up design. A module contains a model of a design or part of a design. Modules can incorporate other modules to establish a model hierarchy that describes how parts of a design are incorporated in an entire design. The constructs of the Verilog HDL, such as its declarations and statements, are enclosed in modules.

The Figure 2.7 shows the abstraction level of the Verilog. Verilog supports abstract behavioural modeling, so can be used to model the functionality of a system at a high level of abstraction. This is useful at the system analysis and partitioning stage. Verilog supports RTL (Register Transfer Level) descriptions, which are used for the detailed design of digital circuits. Synthesis tools transform RTL descriptions to gate level.Verilog supports gate and switch level descriptions, used for the verification of digital designs, including gate and switch level logic simulation, static and dynamic timing analysis, testability analysis and fault grading.

.

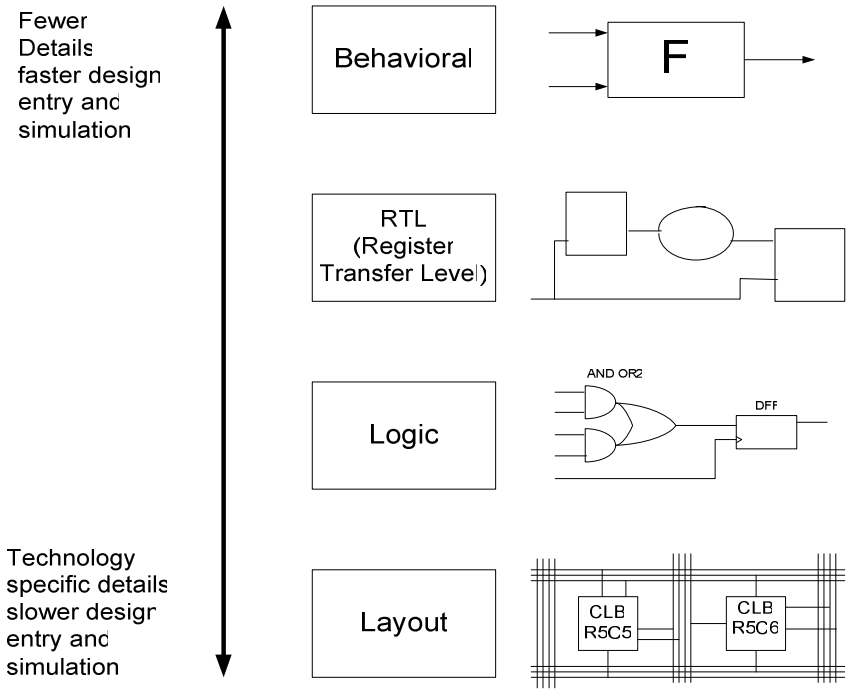


Figure 2.7. Levels of Abstraction

2.4.3. Why use Verilog HDL?

Digital systems are highly complex. At their most detailed level, they may consist of millions of elements, i. e., transistors or logic gates. Therefore, for large digital systems, gate-level design is dead. For many decades, logic schematics served as the main way of logic design, but not any more. Today, hardware complexity has grown to such a degree that a schematic with logic gates is almost useless as it shows only a web of connectivity and not the functionality of design. Since the 1970s, Computer engineers and electrical engineers have moved toward hardware description languages (HDLs). The most prominent modern HDLs in industry are Verilog and VHDL. Verilog is one of the top HDL used by over thousands of designers.

The Verilog language provides the digital designer with a means of describing a digital system at a wide range of levels of abstraction, and, at the same time,

provides access to computer-aided design tools to aid in the design process at these levels [26].

2.5. Field Programmable Gate Arrays

Field-programmable gate array (FPGA) is a step above the PLD in complexity. The difference between FPGA and PLD is very little. Both FPGA and PLD can be volatile or non-volatile. FPGA is much larger and more complex than a PLD [14]. FPGA consists of a two-dimensional array of logic blocks. Each logic block is programmable to implement any logic function. Thus, they are also called configurable logic blocks (CLBs) [15]. Switchboxes or channels contain interconnection resources that can be programmed to connect CLBs to implement more complex logic functions. Designers can use existing CAD tools to convert HDL code in order to program FPGAs. An FPGA contains 5,000 to 10,000,000 gates (or more) [16]. Since the FPGA can be reprogrammed, the turnaround time is only a few minutes. The advantages of FPGAs are lower prototyping costs and shorter production lead times, which advances the time-to-market and in turn increases profitability [17]. It can also ensure the reliability of the design on the board. The disadvantages include lower speed of operations and lower gate density, which has a larger area compared to a ASIC. Thus, a typical FPGA may be 2x-10x slower and 2x-10x more expensive than an equivalent-gate ASIC. Configurable logic blocks of the FPGA includes some fixed logic elements, such as look-up tables, multiplexers, and flip-flops. Even a simple logic inverter function uses CLB. Thus this stuation reduces the speed of the logic design. But in the ASICs, only the needed part of the functions are produced.

It has also input/output blocks to provide the interface between the chip pins and the internal signals. The signals from all blocks are connected to each other using wires, which in turn connected to each other by programmable routing switches. The CLBs have the logic resources that are necessary to implement various

combinational and sequential logic functions. Normally, a CLB has look-up tables (LUTs), multiplexers, and flip-flops.

There are two methods of programming FPGAs. The first, SRAM programming, involves static RAM bits for each programming element. Writing the bit with a zero turns off a switch, while writing with a one turns on a switch. The other method involves anti-fuses which consist of microscopic structures. A certain amount of current during programming of the device causes the two sides of the anti-fuse to connect [18].

The advantages of SRAM based FPGAs is reprogrammability, the FPGAs can be reprogrammed any number of times, even while they are in the system, just like writing to a normal SRAM. The disadvantages are that they are volatile, which means a power glitch could potentially change it. Also, SRAM based devices have large routing delays.

The advantages of Anti-fuse based FPGAs are that they are non-volatile and the delays due to routing are very small, so they tend to be faster. The disadvantages are that they require a complex fabrication process, they require an external programmer to program them, and once they are programmed, they cannot be changed.

Major FPGA manufacturers are Xilinx and Altera in the programmable logic market whose FPGAs are based on SRAM. Xilinx holds more than 50 % of the market share. Xilinx have two family of FPGAs which are SPARTAN and VIRTEX series. Virtex series FPGA is mainly focused on the very fast and complex designs, such as DSP. On contrast to Virtex series, SPARTAN FPGAs are mainly focused to low cost applications.

Spartan-IIE FPGA is made mainly of five kinds of elements: Input/Output blocks (IOBs), Configurable logic blocks (CLBs), block random-access memories (Block RAMs), Delay-locked loops (DLLs), and versatile multi-level interconnect structure [15]. A block diagram of Spartan-IIE FPGA is shown in Figure 2.8.

On the left and the right sides of the chip there are block RAMs that can be configured to realize RAMs or FIFOs as explained in [19] [24]. For each four rows of CLBs, there are two block RAMs: one on the left side and one on the right side. Each block RAM is 4 Kbits. The IOBs surround the CLBs and the block RAMs to provide the interface between the package pins and the internal signals. The versatile multi-level interconnect structure is configured to provide the necessary interconnection and routing among the various blocks as well as among the cells inside the blocks themselves. The DLLs provide multiple minimal-skew clock signals. The programming (i.e., the FPGA configuration) of all elements is done by SRAM.Which means that a Spartan-IIE needs to be reprogrammed every time the power is off.

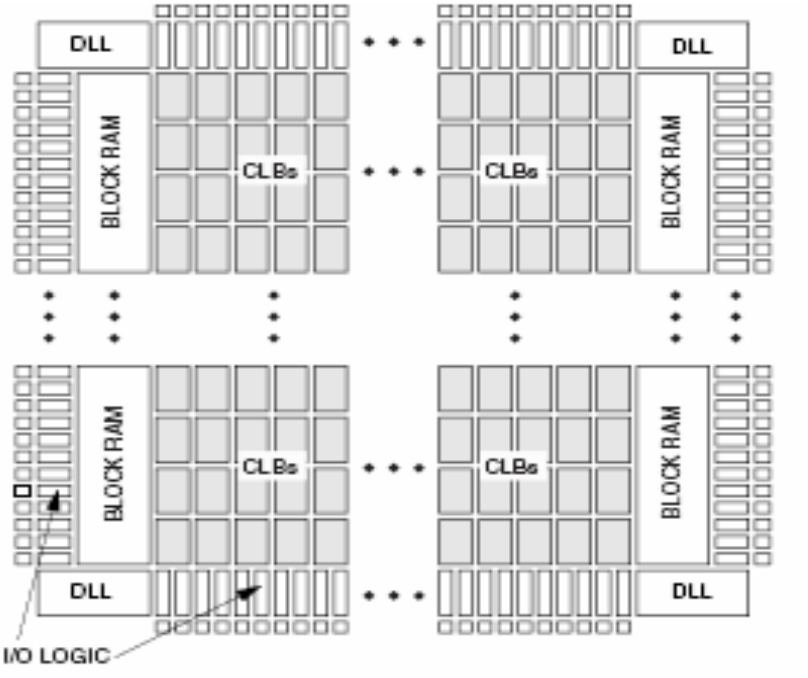


Figure 2.8. Basic Spartan-IIE Family FPGA Block Diagram

Logic of the designs are realized by using the CLBs in the FPGA. A Spartan-II FPGA contains an RxC array of CLBs.The height and width of the array depends on how big the chip is. Each CLB has two slices. Figure 2.9 shows the basic slice structure. Each slice has the following logic elements: two look-up tables (LUTs), two storage elements, one multiplexer (F5MUX), carry and control logic. Each LUT is a 16x1 RAM that can be used as a logic function generator, 16x1 synchronous RAM, or 16-bit shift register. The two LUTs can be combined to make a 32x1 or 16x2 synchronous RAM, or 16x1 dual-port synchronous RAM. The F5MUX can be used to combine the output of both LUTs. By this combination it is possible to implement a 4-to-1 multiplexer, any 5-input logic function, or some 9-input functions. Each CLB has also an F6MUX. This multiplexer combines the outputs of the two slices.

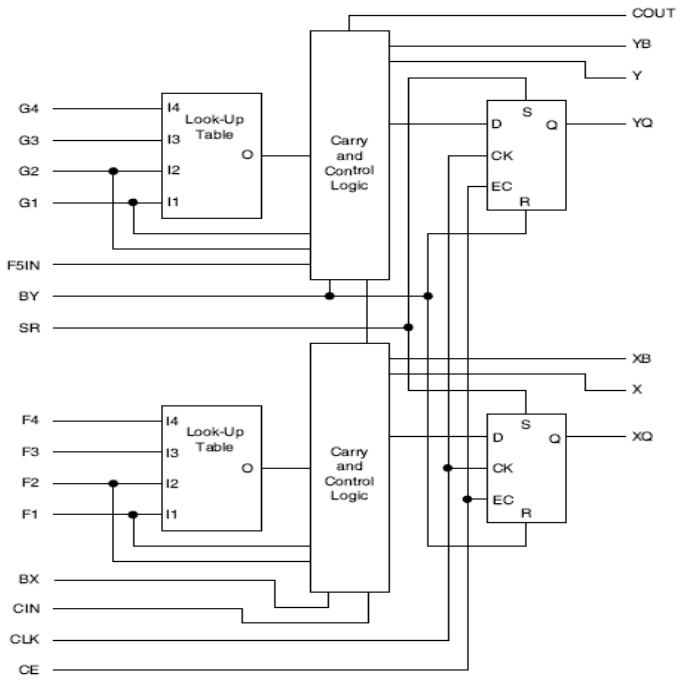


Figure 2.9. Spartan-IIE CLB Slice (two identical slices in each CLB)

This combination of two slices can implement an 8-to-1 multiplexer, any 6-input functions, or some 19-input functions. The two storage elements provide the support for implementing sequential logic functions. They can be configured to be D flip-flops or D latches. The dedicated carry logic inside each slice provides arithmetic carry chain.

To be more specific, the XC2S200 FPGA that is used in this work. It has 28x42=1176 CLBs, 146 user I/O pins, and 56 K bits of block RAM. This provides a lot of resources that should be carefully utilized. Detailed information about Spartan-İİE FPGAs can be found in [15], [20].

CHAPTER 3

BASIC FEATURES OF PIC16XX

3.1. Memory Organization

PIC16XX has two separate memory blocks, one for data and the other for program. SFR registers in RAM memory make up the data block, while FLASH or OTP memory makes up the program block.

3.1.1. Program Memory

Mid-Range PIC16XX devices have a 13-bit program counter capable of addressing an 8K x 14 program memory space. The width of the program memory bus (instruction word) is 14-bits. Since all instructions are a single word, a device with an 8K x 14 program memory has space for 8K of instructions. This makes it much easier to determine if a device has sufficient program memory for a desired application. This program memory space is divided into four pages of 2K words. To jump between the program memory pages, the high bits of the Program Counter (PC) must be modified. This is done by writing the desired value into a SFR called PCLATH (Program Counter Latch High).

3.1.2. Data Memory

Data memory is made up of the Special Function Registers (SFR) area, and the General Purpose Registers (GPR) area. The SFRs control the operation of the device, while GPRs are the general area for data storage and scratch pad operations.

The data memory is banked for both the GPR and SFR areas. The GPR area is banked to allow greater than 96 bytes of general purpose RAM to be addressed. SFRs are for the registers that control the peripheral and core functions. Banking requires the use of control bits for bank selection. These control bits are located in the STATUS Register (STATUS<7:5>). To move values from one register to another register, the value must pass through the W register. This means that for all register-to-register moves, two instruction cycles are required.

The entire data memory can be accessed either directly or indirectly. Direct addressing may require the use of the RP1:RP0 bits. Indirect addressing requires the use of the File Select Register (FSR). Indirect addressing uses the Indirect Register Pointer (IRP) bit of the STATUS register for accesses into the Bank0 / Bank1 or the Bank2 / Bank3 areas of data memory.

3.1.3. Special Function Registers

The SFRs are used by the CPU and Peripheral Modules for controlling the desired operation of the device. These registers are implemented as static RAM.

The SFRs can be classified into two sets, those associated with the “core” function and those related to the peripheral functions. Those registers related to the “core” are described in this section, while those related to the operation of the peripheral features are described in the section of that peripheral feature. Basic SFR registers can be seen by the Figure 3.1.

3.1.4. Program Counter

The program counter (PC) specifies the address of the instruction to fetch for execution. The PC is 13-bits wide. The low byte is called the PCL register. This register is readable and writable. The high byte is called the PCH register. This register contains the PC<12:8> bits and is not directly readable or writable. All updates to the PCH register go through the PCLATH register.

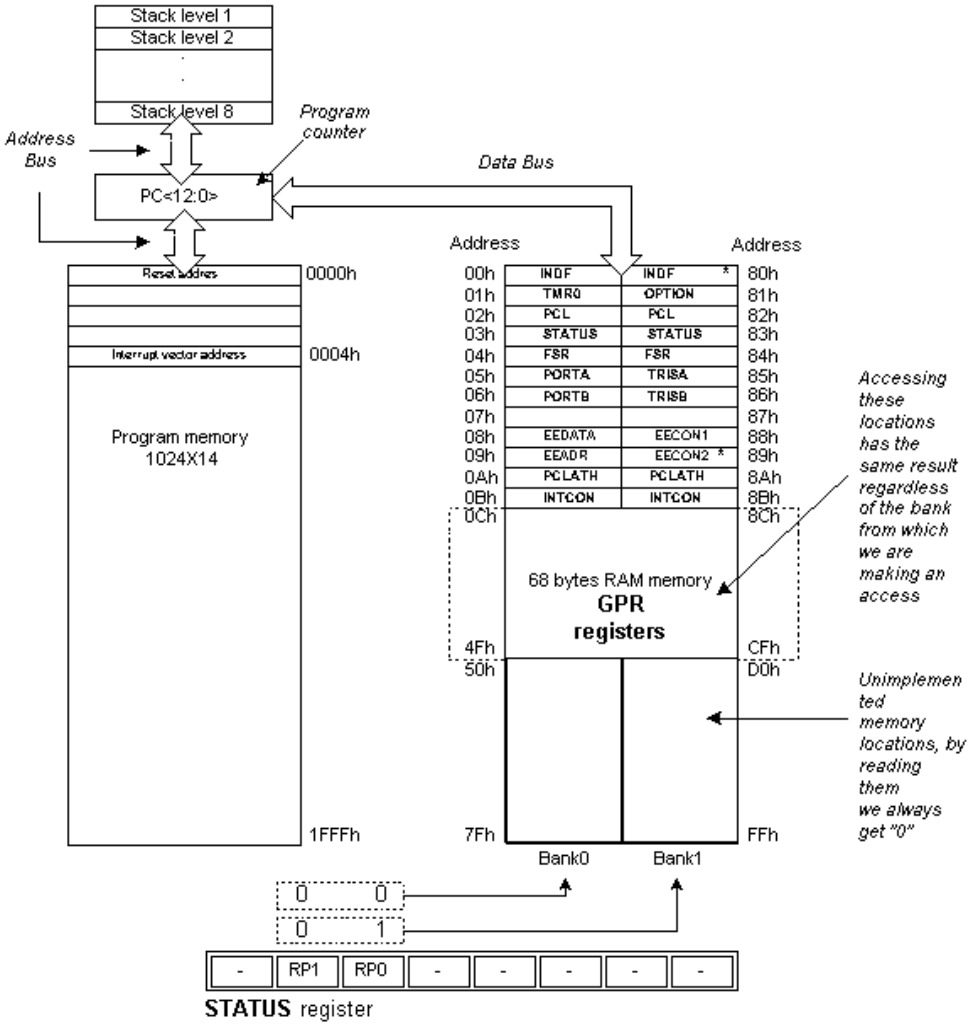


Figure 3.1 Memory Organization of PIC16F84 Microcontroller

3.1.5. Stack

The stack allows a combination of up to 8 program calls and interrupts to occur. The stack contains the return address from this branch in program execution.

Mid-Range MCU devices have an 8-level deep x 13-bit wide hardware stack. The stack space is not part of either program or data space and the stack pointer is not

readable or writable. The PC is PUSHed onto the stack when a CALL instruction is executed or an interrupt causes a branch. The stack is POPed in the event of a RETURN, RETLW or a RETFIE instruction execution. PCLATH is not modified when the stack is PUSHed or POPed. After the stack has been PUSHed eight times, the ninth push overwrites the value that was stored from the first push. The tenth push overwrites the second push (and so on)

3.2. Addressing Modes

RAM memory locations can be accessed directly or indirectly.

3.2.1. Direct Addressing Mode

Direct Addressing is done through a 9-bit address. This address is obtained by connecting 7th bit of direct address of an instruction with two bits (RP1, RP0) from STATUS register as is shown on the following picture. Any access to SFR registers is an example of direct addressing.

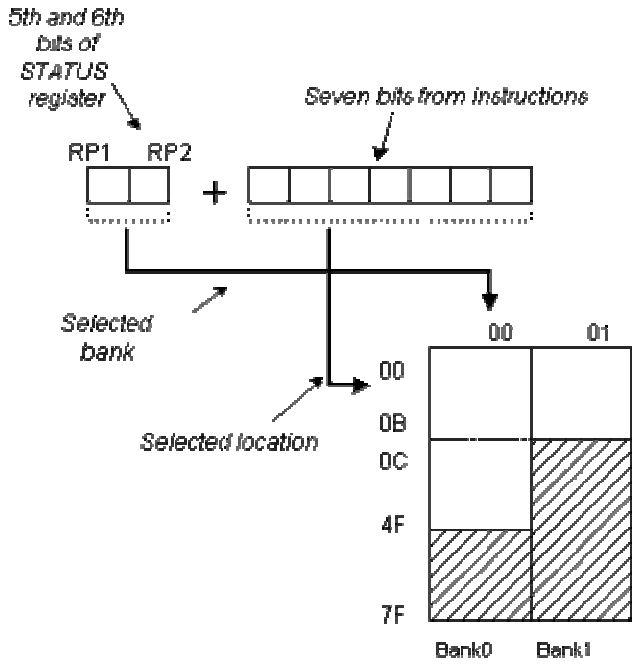


Figure 3.2 Direct Addressing Mode

3.2.2. Indirect Addressing Mode

ndirect unlike direct addressing does not take an address from an instruction but derives it from IRP bit of STATUS and FSR registers. Addressed location is accessed via INDF register which in fact holds the address indicated by a FSR. In other words, any instruction which uses INDF as its register in reality accesses data indicated by a FSR register. Let's say, for instance, that one general purpose register (GPR) at address 0Fh contains a value of 20. By writing a value of 0Fh in FSR register we will get a register indicator at address 0Fh, and by reading from INDF register, we will get a value of 20, which means that we have read from the first register its value without accessing it directly (but via FSR and INDF).

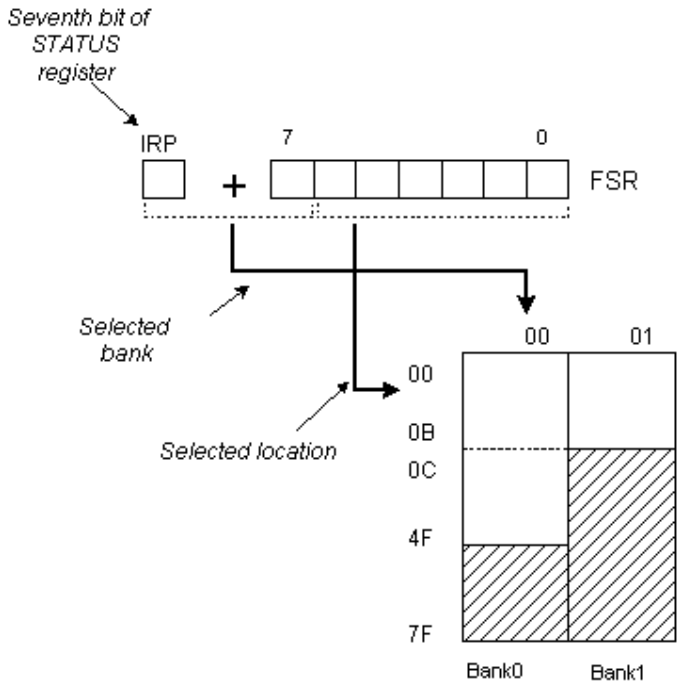


Figure 3.3 Indirect Addressing Mode

It appears that this type of addressing does not have any advantages over direct addressing, but certain needs do exist during programming which can be solved smoothly only through indirect addressing. Indirect addressing is very convenient for manipulating data arrays located in GPR registers. In this case, it is necessary

to initialize FSR register with a starting address of the array, and the rest of the data can be accessed by incrementing the FSR register.

3.3. Instruction Set Summary

The operation of the CPU is determined by the instruction it executes, referred to as machine instructions or computer instructions. The collection of different instructions that the CPU can execute is referred to as the CPU's instruction set. The instruction set defines the datapath and everything else in a processor.

Table 3.1 shows the instruction set summary of the designed microcontroller which is compatible with the PIC16XX series of the microcontroller [21].There are 35 instructions grouped into 3 basic categories:

- Byte-oriented operations
- Bit-oriented operations
- Literal and control operations

For byte-oriented instructions, 'f' represents a file register designator and 'd' represents a destination designator. The file register designator specifies which file register is to be used by the instruction. The destination designator specifies where the result of the operation is to be placed. If 'd' is zero, the result is placed in the W (Working) register. If 'd' is one, the result is placed in the file register (RAM) specified in the instruction. For bit-oriented instructions, 'b' represents a bit field designator which selects the number of the bit affected by the operation, while 'f' represents the number of the file in which the bit is located. For literal and control operations, 'k' represents an eight or eleven bit constant or literal value.

All instructions are executed in one single instruction cycle, unless a conditional test is true or the program counter is changed as a result of an instruction. In these

cases, the execution takes two instruction cycles with the second cycle executed as an NOP (NO Operation).

As mentioned earlier, instruction set of the design is based on Microchip PIC16XX instruction set. In this way, the design can use the same assembler and simulator provided by Microchip since the final design is compatible with the core of PIC16XX microcontroller.

Table 3.1 Instruction Set Summary

Mnemonics , Operands	Description	Cycles	14-Bit Instruction Word	Status Affected
			Msb Lsb	
BYTE-ORIENTED FILE REGISTER OPERATIONS				
ADDWF f,d	Add W and f	1	00 0111 dfff ffff	C,DC,Z
ANDWF f,d	AND W and f	1	00 0101 dfff ffff	Z
CLRF f	Clear f	1	00 0001 1fff ffff	Z
CLRW -	Clear W	1	00 0001 0xxx xxxx	Z
COMF f,d	Complement f	1	00 1001 dfff ffff	Z
DECF f,d	Decrement f	1	00 0011 dfff ffff	Z
DECFSZ f,d	Decrement f, Skip if Zero	1(2)	00 1011 dfff ffff	
INCF f,d	Increment f	1	00 1010 dfff ffff	Z
INCFSZ f,d	Increment f, Skip if Zero	1(2)	00 1111 dfff ffff	
IORWF f,d	Inclusive OR W with f	1	00 0100 dfff ffff	Z
MOVF f,d	Move f	1	00 1000 dfff ffff	Z
MOVWF f,d	Move W to f	1	00 0000 1fff ffff	
NOP	No Operation	1	00 0000 0xx0 0000	
RLF f,d	Rotate Left f through Carry	1	00 1101 dfff ffff	C
RRF f,d	Rotate Right f through Carry	1	00 1100 dfff ffff	C
SUBWF f,d	Subtract W from f	1	00 0010 dfff ffff	C,DC,Z
SWAPF f,d	Swap Nibbles in f	1	00 1110 1fff ffff	
XORWF f,d	Exclusive OR W with f	1	00 0110 dfff ffff	Z

Table 3.1 Instruction Set Summary (cont'd)

BIT-ORIENTED FILE REGISTER OPERATIONS				
BCF f,d	Bit Clear f	1	01 00bb bfff ffff	
BSF f,d	Bit Set f	1	01 01bb bfff ffff	
BTFSC f,d	Bit Set f , Skip if Clear	1(2)	01 10bb bfff ffff	
BTFSS f,d	Bit Set f , Skip if Set	1(2)	01 11bb bfff ffff	
LITERAL AND CONTROL OPERATIONS				
ADDLW k	Add literal and W	1	11 111x kkkk kkkk	C,DC,Z
ANDLW k	AND literal and W	1	11 1001 kkkk kkkk	Z
CALL k	Call subroutine	2	10 0kkk kkkk kkkk	
CLRWDT	Clear Watchdog Timer	1	00 0000 0110 0100	
GOTO k	Go to address	2	10 1kkk kkkk kkkk	
IORLW k	Inclusive OR literal with W	1	11 1000 kkkk kkkk	Z
MOVLW k	Move literal to W	1	11 00xx kkkk kkkk	
RETFIE	Return from Interrupt	2	00 0000 0000 1001	
RETLW k	Return with literal in W	2	11 01xx kkkk kkkk	
RETURN	Return from Subroutine	2	00 0000 0000 1000	
SLEEP	Go into Standby mode	2	00 0000 0110 0011	
SUBLW k	Subtract W from literal	1	11 110x kkkk kkkk	C,DC,Z
XORLW k	Exclusive OR literal with W	1	11 1010 kkkk kkkk	Z
MULT	*Multiply the nibbles of W*	*1*	*11 1011 xxxx xxxx*	*Z*

There is a new instruction with respect to the original PIC instructions. MULT instruction makes a 4-bit multiplication.

Detailed operation for each instruction requires further reference to the Instruction Set section in PICmicro Mid-Range MCU Family Referance Manual [21].

3.4. Instruction Formats

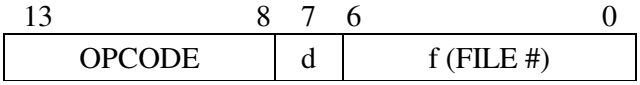
PIC microcontrollers have three general formats of instructions. As can be seen from the general format of the instructions, the opcode portion of the instruction word varies from 3-bits to 6-bits of information. Thus PIC microcontrollers have 35 instructions. Instruction Description conventions are shown is Table 3.2

Table 3.2 Instruction Description Conventions

Field	Description
f	Register file address (0x00 to 0x7F)
W	Working register (accumulator)
b	Bit address within an 8-bit file register (0 to 7)
k	Literal field, constant data or label (may be either an 8-bit or an 11-bit value)
x	Don't care (0 or 1) The assembler will generate code with x = 0
d	Destination select; d = 0: store result in W, d = 1: store result in file register f.

General format of the instructions are follows;

Byte oriented file register operations:

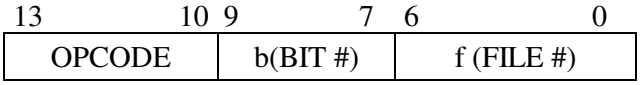


d=0 for destination W (working register)

d=1 for destination f

f= 7-bit register address

Bit oriented file register operations:

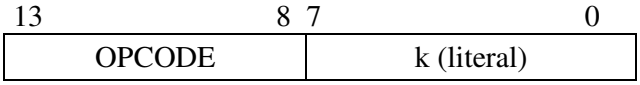


b= 3-bit bit address

f= 7-bit register address

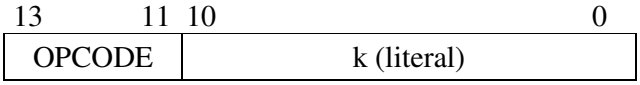
Literal and Control operations:

General:



k= 8-bit literal (immediate) value

CALL and GOTO instructions only:



k= 11-bit literal (immediate) value

CHAPTER 4

IMPLEMENTATION OF MICROCONTROLLER

4.1. Pin Description

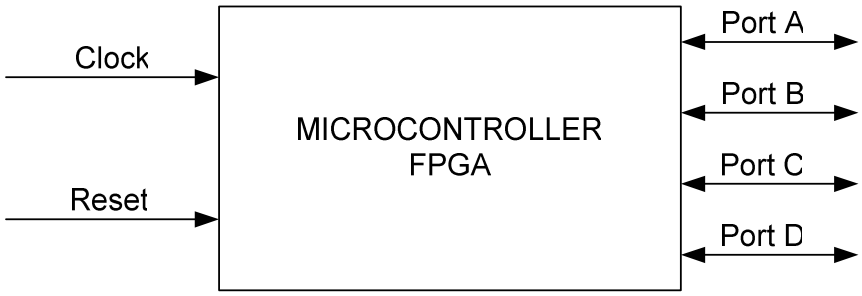


Figure 4.1. Microcontroller Pin Configuration

Figure 4.1 shows the pin configuration for the designed microcontroller. The microcontroller has 2 input pins and 4 bi-directional I/O ports. Each I/O port consists of 8 individual I/O pins except PortA. Port A has only 5 bidirectional I/O pins. So 4 I/O ports contribute to a total of 29 I/O pins. The clock signal will drive the whole microcontroller directly. Reset is active low; when asserted it resets the microcontroller to the default state even if the clock is not running. Each bit of the ports can be configured to be input or output in the software of the microcontroller. All port pins are tri-stated when the microcontroller is reset.

4.2. Architecture Overview

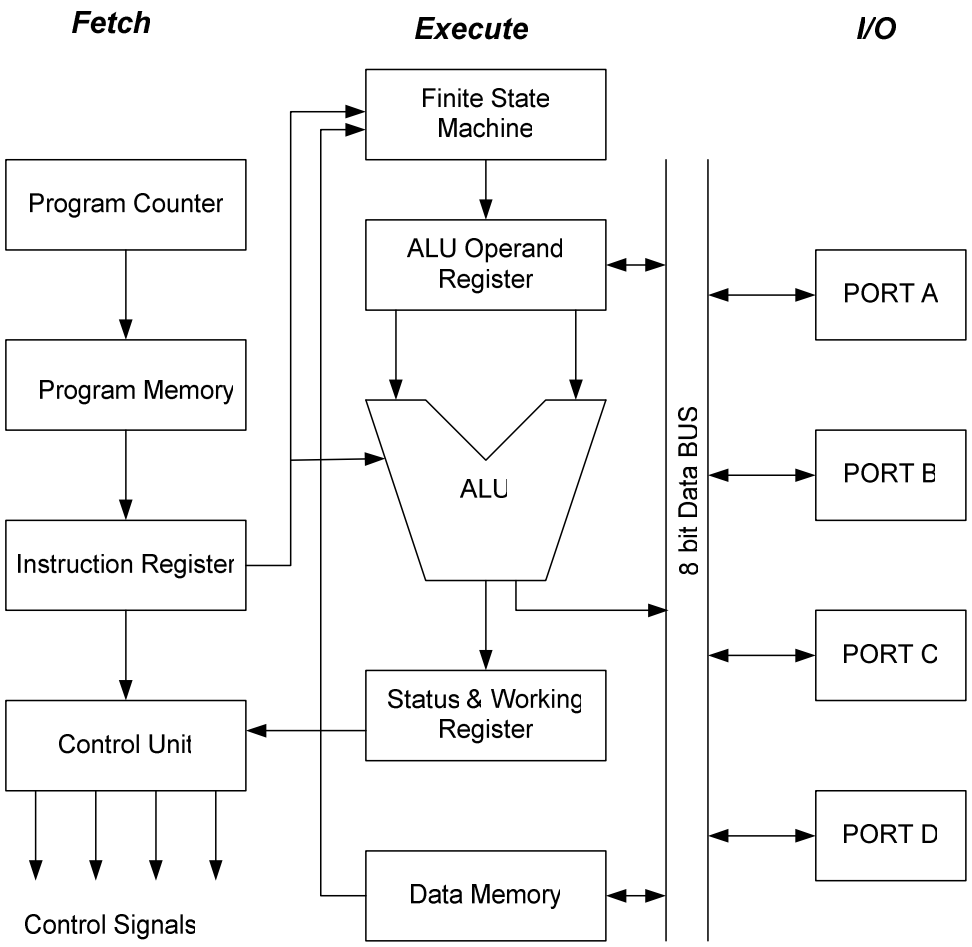


Figure 4.2. Top Level Architectural Block Diagram

Figure 4.2 shows the simplified top-level block diagram of the design, every part of this block diagram needs to be implemented in the FPGA. The microcontroller will be designed using the top down design approach. Some blocks like the I/O ports, instruction register and status register are easy to design, but modules like ALU and the finite state machine require a lot of understanding. The overall dataflow and bus structure between all the blocks must be understood before designing the block individually.

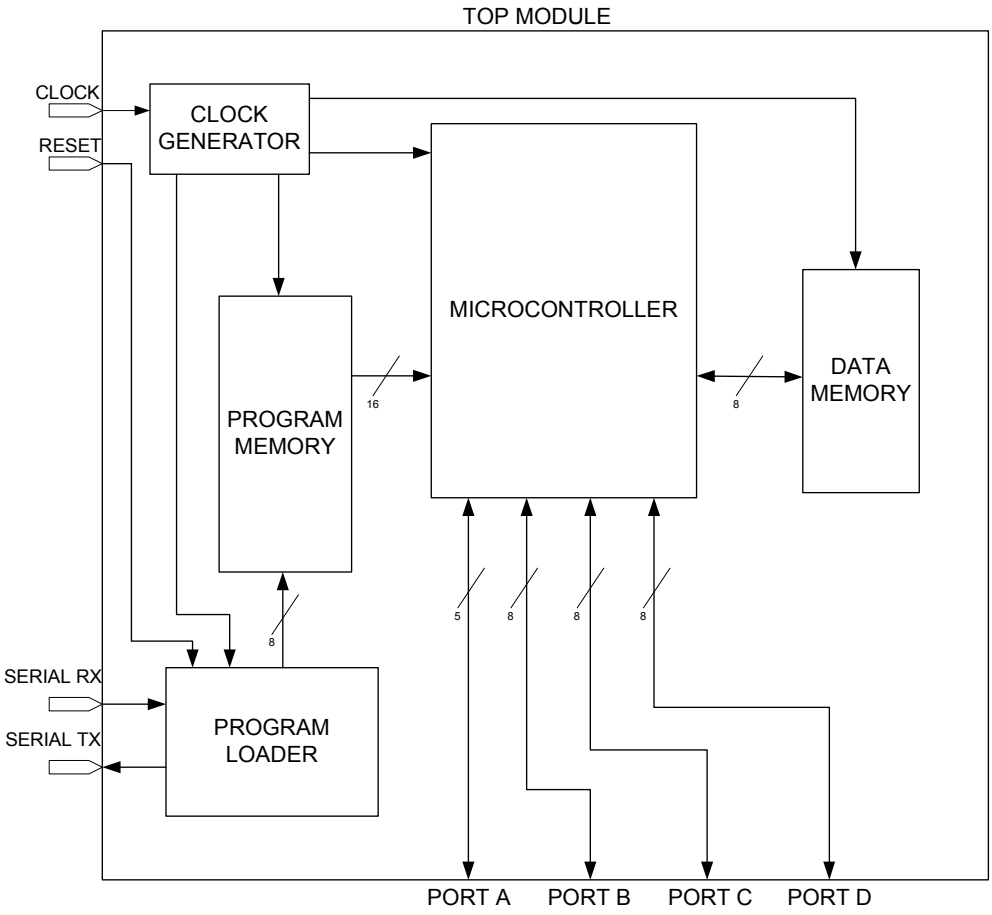


Figure 4.3. Top Module of the Microcontroller Design

The module of the microcontroller designed in the FPGA can be divided into 5 sub modules which can be seen in Figure 4.3. These sub modules are;

- Clock Generator Unit
- Program Load Unit
- Microcontroller Unit
- Program Memory
- Data Memory

File hierarchy of the top module of the microcontroller design and the files that are used in the design can be seen with the following Figure 4.4. The interconnection between the files is shown in Appendix D and Appendix E. The files that are shown in Figure 4.4 are in the CD-ROM in Appendix F.

Figure 4.4. File Hierarchy of the Microcontroller Design

4.2.1. Clock Generator Unit

The clock generator modules' main function is to produce the necessary clock rate and distribute the clock to the other modules in the FPGA. Incoming clock rate is 48 MHz, which is passed through input global clock buffer (IBUFG). IBUFG is connected to the dedicated input buffers for connecting to the clock buffer BUFG. The IBUFG input can only be driven by the global clock pins. The IBUFG output can drive CLKIN of a Delay Locked Loop (DLL), BUFG, or user logic.

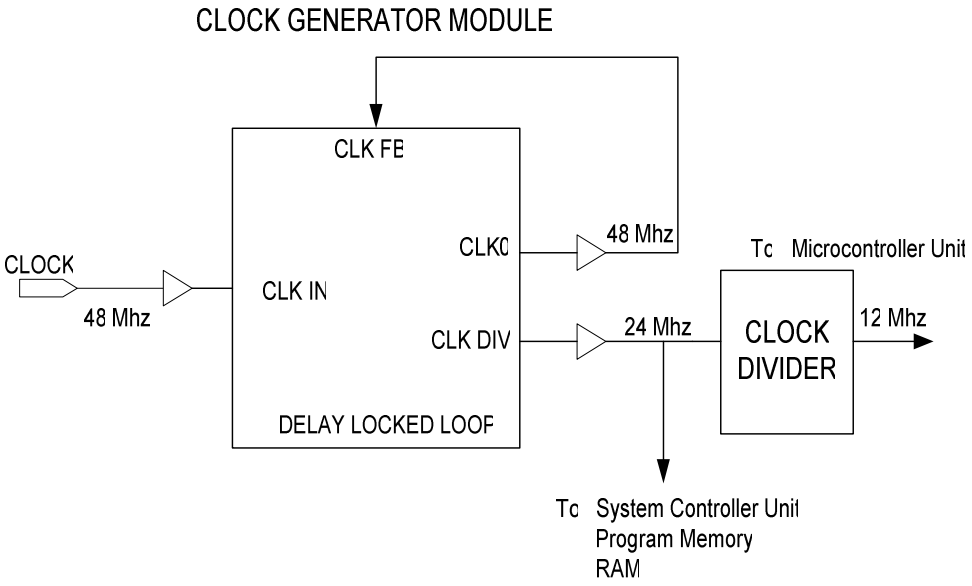


Figure 4.5. Clock Generator Unit

Associated with each global clock input buffer is a fully digital Delay-Locked Loop (DLL) that can eliminate skew between the clock input pad and internal clock-input pins throughout the device. Each DLL can drive two global clock networks. The DLL monitors the input clock and the distributed clock, and automatically adjusts a clock delay element. Additional delay is introduced such that clock edges reach internal flip-flops exactly one clock period after they arrive at the input. This closed-loop system effectively eliminates clock-distribution delay by ensuring that clock edges arrive at internal flip-flops in synchronism with clock edges arriving at the input [22], [25].

DLL synchronizes the clock signal at the feedback clock input (CLKFB) to the clock signal at the input clock (CLKIN). The frequency of the clock signal at the CLKIN input must be at least 24 MHz. The CLKIN pin must be driven by an IBUFG or a BUFG. If phase alignment is not required, CLKIN can also be driven by IBUF. On-chip synchronization is achieved by connecting the CLKFB input to a point on the global clock network driven by a BUFG, a global clock buffer. The BUFG connected to the CLKFB input of the DLL must be sourced from the CLK0

output of the same DLL. The CLKIN input should be connected to the output of an IBUFG, with the IBUFG input connected to a pad driven by the system clock. [22].

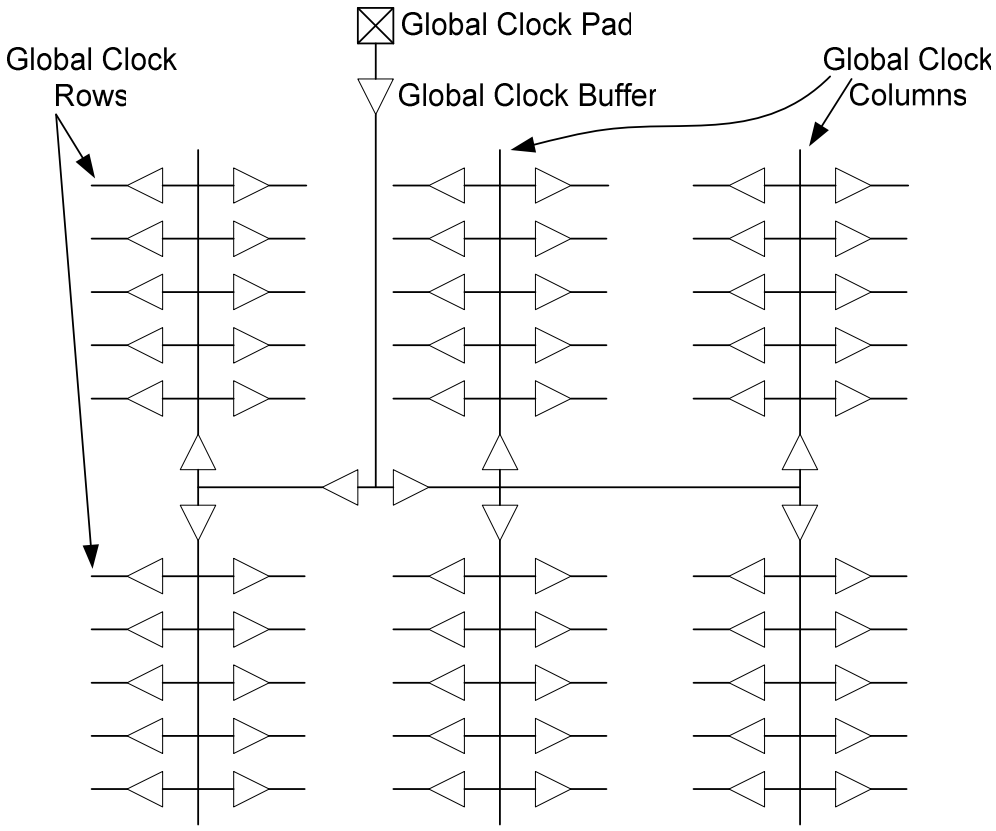


Figure 4.6. Global Clock Distribution Network Through the FPGA

In addition to eliminating clock-distribution delay, the DLL provides advanced control of multiple clock domains. DLL can divide the clock by 2. In this design CLK DIV output is used as the main clock output. At the output of the CLK DIV, the clock rate reduces at a rate of 24 MHz. This clock is used in the following modules;

- Program Load Unit
- Program Memory
- Data Memory

After obtaining the 24 MHz, 12 MHz clock is also required for the microcontroller unit.

The following simple circuit is used to generate a 12 MHz clock.

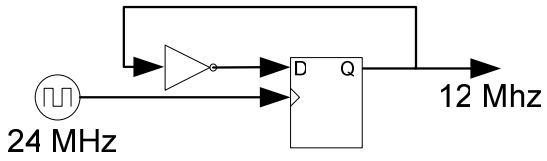


Figure 4.7. Clock Divider Circuit

Microcontroller unit requires less clock rate because of the long data path design and the worst case delays in the microcontroller unit. This block is implemented in the “clock_gen.v” file as shown in Figure 4.4..

4.2.2. Program Load Unit

Program Load unit receives the compiled program from a PC via RS232 serial port. The compiled programs are sent using a program loader designed with using Borland C++ Builder. This program takes the Intel hex format file, and sends the binary data to the microcontroller. First the communication link is established with the FPGA microcontroller. After communication link is done, program is loaded and sent through the RS232 serial port at a speed of 57600 baud.

Program load unit has 4 inputs, which are 24 MHz clock, reset input, serial rx, serial tx. Clock is received from the clock generator module. Reset input, serial receive and serial transmit I/Os are connected to the directly to the input/output pins of the FPGA.

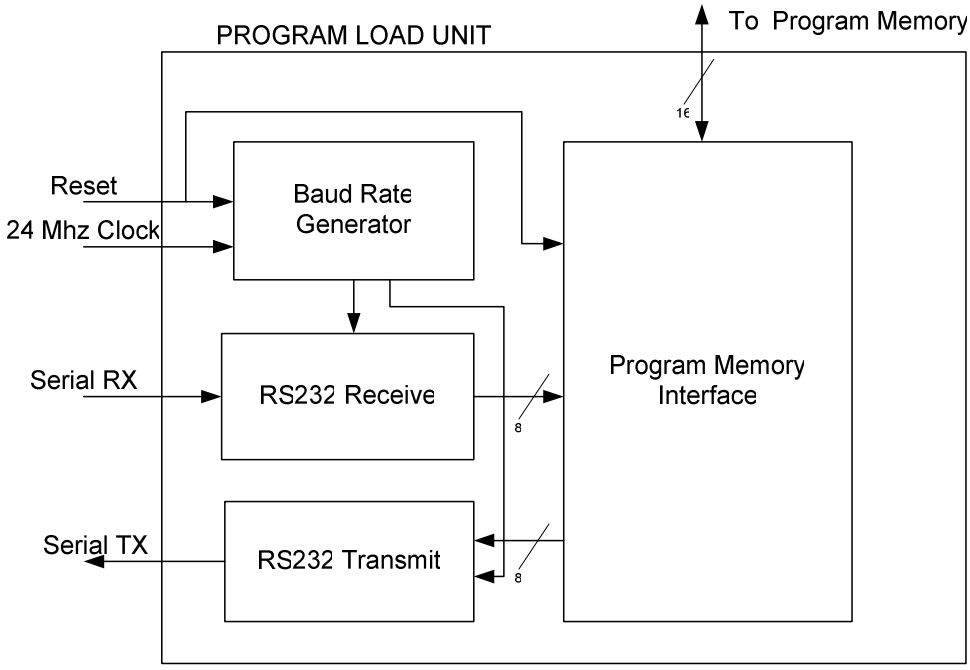


Figure 4.8. Block Diagram of the Program Load Unit

The top module for the program loader module is “rs232_loader.v”in Figure 4.4. Program load unit is mainly divided into 4 sub blocks as can be seen in Figure 4.8. These are;

- Baud Rate generator
- RS232 Receive unit
- RS232 Transmit Unit
- Program Memory Unit

4.2.2.1. Baud Rate Generator

The baud rate generator provides both the receiver and the transmitter with the baud rate clock, a bit-period clock. The input clock for this module is 24 MHz. The output clock for receive and transmit unit is 16 x Baud Rate. If the baud rate is 57600 then generated clock is 921 KHz. This module is implemented in the “baud_gen.v” file as shown in Figure 4.4.

4.2.2.2. Rs232 Receive Unit

This block takes care of receiving an RS232 input word, from the "rxd" line in a serial way. The appropriate clock is provided by the baud rate generator unit, which is 16 times the baud rate. The receive input line is sampled 16 times per bit after sensing a start bit (logic high). Mid-count value is taken as an input and passed through a shift register. Data is valid only after receiving a valid stop bit. This module is implemented in the “serial.v” file as shown in Figure 4.4.

4.2.2.3. Rs232 Transmit Unit

This module transmits the 8-bit byte using baud rate clock through the serial line “txd”. First this block generates a start bit, then serially shifts the input data and finally generates a stop bit. Since RS232 serial communication is asynchronous, bit timing requires careful attention. This module is implemented in the “serial.v” file in Figure 4.4.

4.2.2.4. Program Memory Interface Unit

This unit directly writes the received data to the appropriate location of the internal program memory. It has an 8 bit wide data bus. The detailed operation about the program memory will be discussed in section 4.2.3. This module is implemented in the top module of the program loader unit.

4.2.3. Program Memory Unit

An example view of the program memory can be seen by the Figure 4.9. It is implemented with the block RAMs, which is internally available in the FPGA. Block RAM memories are organized in FPGA as columns. Spartan IIE FPGA contains two block RAM columns, one along each vertical edge [15] [22]. Totally there are 56 kbit block RAM in the Spartan IIE FPGAs, that 16 kbit of them is used as a program memory for the designed microcontroller. Each block RAM is fully synchronous and dual-ported with independent control signals for each port.

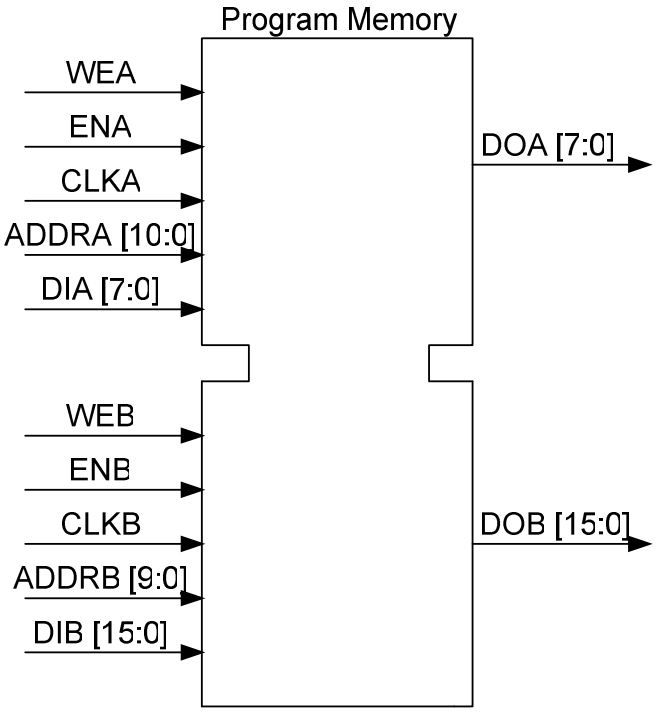


Figure 4.9. A diagram for the 16 kbit dual port Program Memory

Data bus width of each port is configurable, in our case one side of the memory's' data bus width is 8 bit wide and the address bus width is 11 bit wide which is connected to the program load unit. Since we read the data to be written to the program memory is 8bit from PC, so we need an 8 bit wide data bus for one port of the RAM. Second port of the block RAM is connected to the microcontroller, which is 16 bit data bus width. But the microcontroller uses only 14 bit of the block RAM, because the instructions are 14 bit wide.

Each port is fully synchronous with independent clock pins. All port A input pins have setup time referenced to the CLKA pin and its data output bus DOA has a clock to out time referenced to the CLKA. All port B input pins have setup time referenced to the CLKB pin and its data output bus DOB has a clock-to-out time referenced to the CLKB.

The enable ENA pin controls read, write, and reset for port A. When ENA is Low, no data is written and the outputs DOA and outputs preserve the last state. If write enable (WEA) is High, the memory contents reflect the data at DIA. When ENA is High and WEA is Low, the data stored in the RAM address (ADDRA) is read during the Low-to-High clock transition. When ENA and WEA are High, the data on the data input (DIA) is loaded into the location selected by the write address (ADDRA) during the Low-to-High clock transition and the data output (DOA) reflect the selected (addressed) location [22].

The same working operation is also applicable to port B of the dual port RAM.
The above descriptions assume active High control pins (ENA, WEA, CLKA, ENB, WEB, and CLKB).
In the design of the microcontroller, both ports are not used at the same time to prevent any contention. Program load unit only writes to the memory on port A. The microcontroller only reads the instructions from the port B of the memory. This program load unit is implemented by the coregen utility of the Xilinx ISE program[22].

4.2.4. Data Memory Unit

Implementation of the data memory unit is the same as program memory unit which is described in section 4.9. It is also a dual ported RAM which is in the FPGA. It uses 512 byte of the block RAMs of the SPARTAN FPGAs. The PORTA of the RAM is used with the implemented microcontroller unit. The PORTB is used only for debug purpose. The PORTA and PORTB have a 8 bit wide data bus. This RAM unit is implemented by the coregen utility of the Xilinx ISE program. Detailed information for the ram blocks can be found in [22].

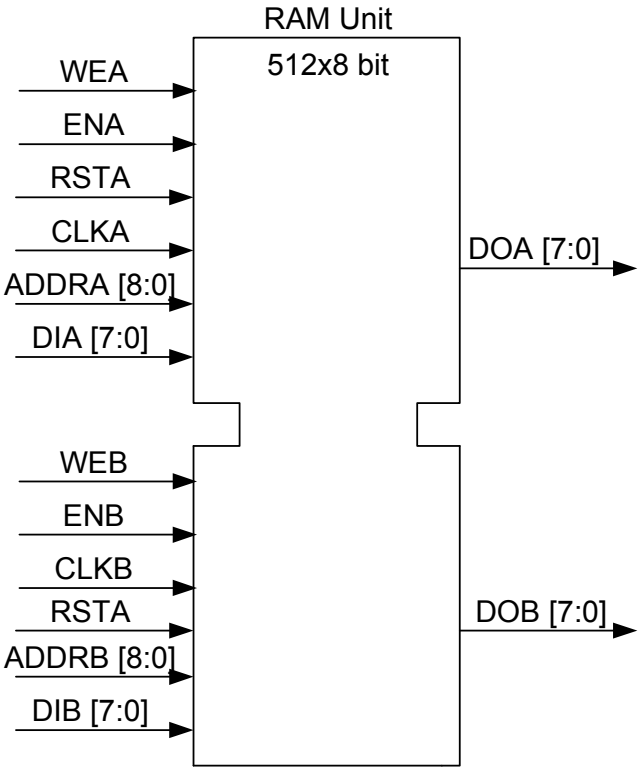


Figure 4.10. A diagram for the 512 byte RAM

4.2.5. Microcontroller Unit

This logic module implements a small RISC microcontroller, with functions and instruction set very similar to those of the mid-range family of the Microchip 16FXX chips. This module is the most complicated module among the other modules.

The input and outputs of the microcontroller, implemented in SPARTAN IIE FPGA can be seen by the following Figure 4.11.

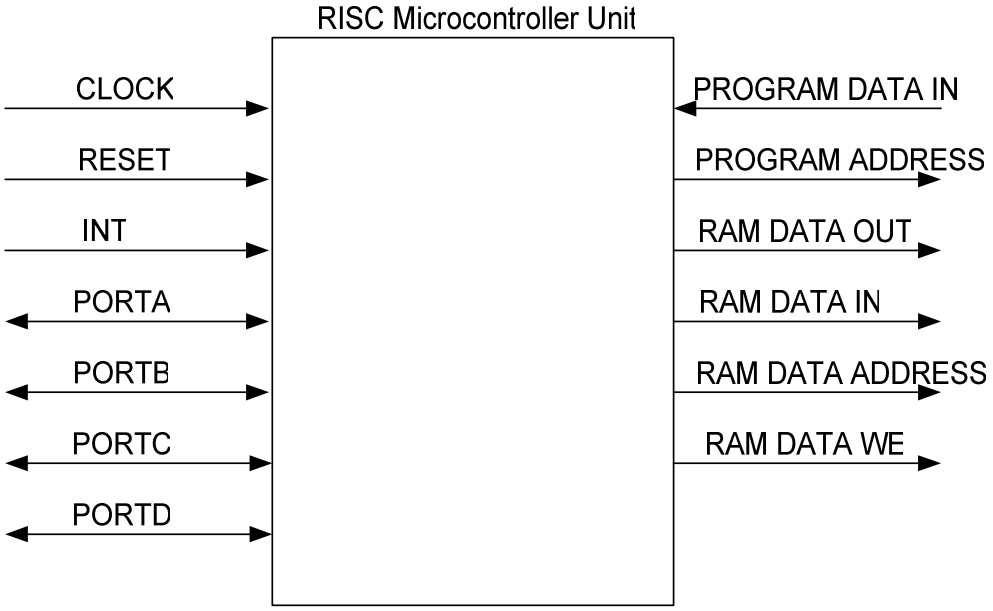


Figure 4.11. Inputs and Outputs of the Microcontroller Unit

The CLOCK input is generated by the clock generator unit which is at 12 MHz. RESET input is directly connected to the one of the pads of the FPGA and used as an asynchronous reset in the FPGA. RESET input is fed through one of the buttons located on the Digilent SPARTAN IIE Development Board. INT pin is an external interrupt input. PORTA through PORTD are connected to the various locations of the development board. Program memory and the RAM connections are made to the internal block rams of the FPGA.

The top module for the microcontroller unit is in “riscmcu.v” file as shown in Figure 4.4. Some sub modules are implemented in a separate verilog design file. The microcontroller unit has the following sub-modules and the implementation files which are listed in Table 4.1. The design of the each sub-module will be discussed one by one in this chapter.

Table 4.1. Sub-Modules inside the microcontroller

1	Instruction Fetch & Decode	instruction_decode.v
2	Calculation of RAM Access Address	calc_ram_address.v
3	Stack	fsm.v
4	Program Counter	fsm.v
5	ALU	alu.v
6	FSM Machine	fsm.v
7	Interrupts	fsm.v
8	I/O Ports	riscmcu.v

4.2.5.1. Instruction Fetch and Decode

Instructions are fetched at the end of the execution and write cycle from the program memory at state S2 of the Finite State Machine. At a reset condition, “0” is loaded to the instruction register, thus the first instruction seems to be NOP operation, it is because to formerly start-up the microcontroller.

After loading the instruction from the program memory, a combinatorial decoding operation takes place according to the instructions listed in Table 3.1. For every instruction, there is a comparator, which indicates the related instruction has arrived. For example if the received instruction is a “CALL” instruction, then a dedicated register is set to indicate the current executing instructions is a “CALL” instruction. The other blocks of the microcontroller checks the related register. So, the microcontroller has a dedicated register for every instruction that indicates the decoded instruction. This module is implemented in verilog file “inst_decode.v” as shown in Figure 4.4.

4.2.5.2. Calculation of RAM Access Address

One of the other sub units of the microcontroller is the calculation of the RAM access address. After loading the instruction from the program memory first the

instruction decoding process is completed. After then, we need to calculate the RAM access address.

First we check whether direct or indirect addressing will be used with the related instruction. There is an INDF register which is not a physical register in the microcontroller. Addressing INDF actually addresses the register whose address is contained in the FSR register. If the addressing mode is direct addressing (i.e. INDF register is used as destination), then the first 7 bit of the opcode and 5^{th} and 6^{th} bit of the status register is concatenated.

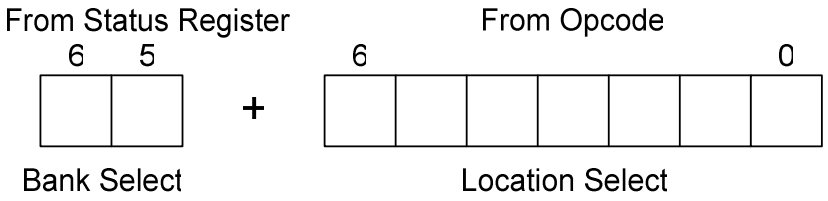


Figure 4.12. Direct Addressing Mode

If the instruction registers' first 7 bit is zero then it is behaved as indirect addressing scheme. In the indirect addressing mode, the value of the FSR register and the 7^{th} bit of the status register is concatenated and used as target address of the RAM.

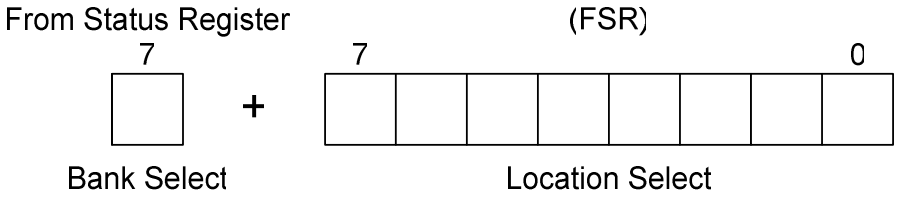


Figure 4.13. Indirect Addressing Mode

After determining the destination address of the RAM, we should classify the destination address, whether it is in the ram area or register area. Internally classification is done according the following Table 4.2.

Table 4.2. Destination RAM Access Addresses

Address	Destination	Description
0E-7F, 8E-FF	SRAM	SRAM
02,82	PCL	Program Counter Low Byte Register
03,83	STATUS	Status Register
04,84	FSR	File Select Register
05	PORTA	5-bit I/O Port
06	PORTB	8-bit I/O Port
0C	PORTC	8-bit I/O Port
0D	PORTD	8-bit I/O Port
85	TRISA	Direction register for PORTA
86	TRISB	Direction register for PORTB
8C	TRISC	Direction register for PORTC
8D	TRISD	Direction register for PORTD
0A,8A	PCLATH	PC Latch High Byte
0B,8B	INTCON	Interrupt Control Register
81	OPTION	Option register

Some of the registers have two addresses, like PCL. Both of them point the same location. It is because to put critical registers on both pages of the RAM. This behavior comes from the original configuration PIC microcontroller.

After determining the source address of the register or RAM, the data is loaded to a temporary register called as “ram_destination”. For example if the RAM access address is status register, ram_destination register is loaded with the value of the status register. In the other part of the design ram_destination register will be used for ALU operations.

Also bit-mask for logical operations (AND, OR, BTFSC,) and bit tests are constructed in this module. This module is implemented in verilog file “calc_ram_address.v” as shown in Figure 4.4.

4.2.5.3. Stack

The stack allows a combination of up to 16 program calls and interrupts to occur in the designed microcontroller. The stack contains the return address from this branch in program execution.

PIC microcontrollers have an 8 level deep x 13-bit wide hardware stack. The stack space is not part of either program or data space and the stack pointer is not readable or writable. The PC is PUSHed onto the stack when a CALL instruction is executed or an interrupt causes a branch. The stack is POPed in the event of a RETURN, RETLW or a RETFIE instruction execution. PCLATH register is not modified when the stack is PUSHed or POPed.

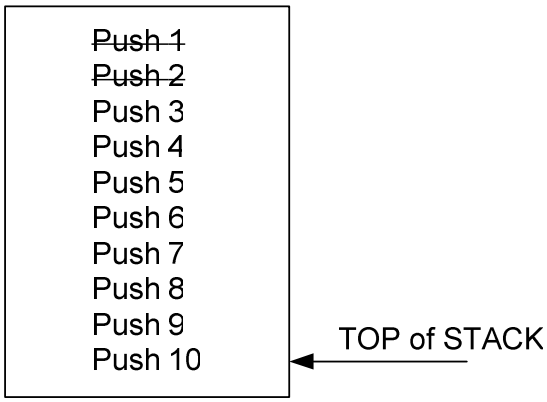


Figure 4.14. Stack Modification

In the original configuration after the stack has been PUSHed eight times, the ninth push overwrites the value that was stored from the first push as in Figure 4.14. The tenth push overwrites the second push (and so on). But the designed microcontroller with the Spartan FPGA has reconfigurable stack space in the verilog code. In the design we have chosen the stack space as 16 words. The stack is implemented in verilog file “fsm.v” as shown in Figure 4.4.

4.2.5.4. Program Counter

The program counter (PC) specifies the address of the instruction to fetch for execution. The PC is 13 bits wide. The low byte is called the PCL register. This register is readable and writable. The high byte is called the PCH register. This register contains the PC<12:8> bits and is not directly readable or writable. If the program counter (PC) is modified or a conditional test is true, the instruction requires two cycles. The second cycle is executed as a NOP. All updates to the PCH register go through the PCLATH register. The program counter is implemented in verilog file “fsm.v” as shown in Figure 4.4.

In this module, first the instruction is checked to be that it is modifying instruction. The following conditions may modify the program counter.

- CALL and GOTO instructions.
- RET, RETLW, RETFIE instructions.
- If the instruction is BTFSC, DECFSZ, INCFSZ and Arithmetic logic unit output is zero.
- If the instruction is BTFSS and Arithmetic logic unit output is one.
- If the execution destination is PCL register.

If one of the above conditions occurs, the next instruction is executed as a NOP instruction. Also if an interrupt condition occurs, the next instruction will also be executed as NOP instruction.

Serving an interrupt request will cause the PC to be loaded with the interrupt vector address (0x0004). So when serving an interrupt request, the PC is first loaded with the vector address, then the CPU execute the instruction loaded from the corresponding vector address - a jump to ISR. The PC is then loaded with the address of the ISR. And finally the CPU starts executing the ISR.

At the beginning or reset condition, both PC register and the old PC are set to zero. If an interrupt condition occurs, the current program counter is saved to the old PC register, which is to be PUSHed to the stack and later POPed by an RETFIE instruction. And the PC is set to interrupt vector address (0x004) of the microcontroller. At the normal operating condition “next PC register” is loaded to the PC register.

“Next PC register” is loaded to the PC register if there is not any reset and interrupt condition occurs. “Next PC register” is defined with the following criteria;

- If the instruction is a return (RET, RETLW, RETFIE) instruction, top of stack is loaded to the next PC register.
- If the instruction is a CALL or GOTO instruction, 3rd and 4th bit of the PCLATH register and first 11 bit of the instruction register are concatenated and loaded to the next PC register.
- If the PCL register is the data destination by the executing instruction then PCLATH register and the ALU output are concatenated and loaded to the next PC register.
- Otherwise next PC register is incremented by one.

If the sleep instruction is executing, then the PC is not allowed to be updated, since the processor will "freeze" and the instruction being fetched during the sleep instruction must be executed upon wakeup interrupt.

4.2.5.5. Arithmetic Logic Unit

The ALU executes many instructions, some directly and some indirectly. We first examine the 35 instructions that are executed directly by the ALU. These instructions are listed in Table 4.3. They are divided into 9 groups which are;

Table 4.3. ALU Group and Instructions

Group	Instruction	Flags Affected
Rotate Left	RLF	C
Rotate Right	RRF	C
Swap Nibbles	SWAP	
Complement	COMP	Z
Logical AND	ANDLW	Z
	ANDWF	Z
	BCF	
	BTFSC	
	BTFSS	
Logical OR	IORLW	Z
	IORWF	Z
	BSF	
Logical XOR	XORLW	Z
	XORWF	Z
Addition	ADDLW	C,DC,Z
	ADDWF	C,DC,Z
	SUBLW	C,DC,Z
	SUBWF	C,DC,Z
	DECF	Z
	DECFSZ	Z
	INCF	Z
	INCFSZ	Z
4-bit Multiplication	MULT	Z
Pass Through	Other Instructions	

- Rotate Left
- Rotate Right
- Swap Nibbles
- Complement

- Logical AND
- Logical OR
- Logical XOR
- Addition
- 4-bit Multiplication
- Pass Through

The ALU is implemented in verilog file “alu.v” as shown in Figure 4.4.

ALU is implemented purely by the combinatorial logic which means that ALU output is asserted immediately, according to the operand A, operand B registers and the instruction which is being executed.

Block diagram of the Arithmetic Logic Unit is as in Figure 4.15.

ALU have two operand inputs which are operand A register and Operand B register. Current executing instruction is also feed to the ALU to choose the right operand. Also the destination of the ALU output is determined by the instruction whether to write the result to the working register or to the RAM. ALU also gives output to the status register.

Depending on the instruction executed, the ALU may affect the values of the Carry (C), Digit Carry (DC), and Zero (Z) bits in the STATUS register. The C and DC bits operate as a borrow bit and a digit borrow out bit, respectively, in subtraction.

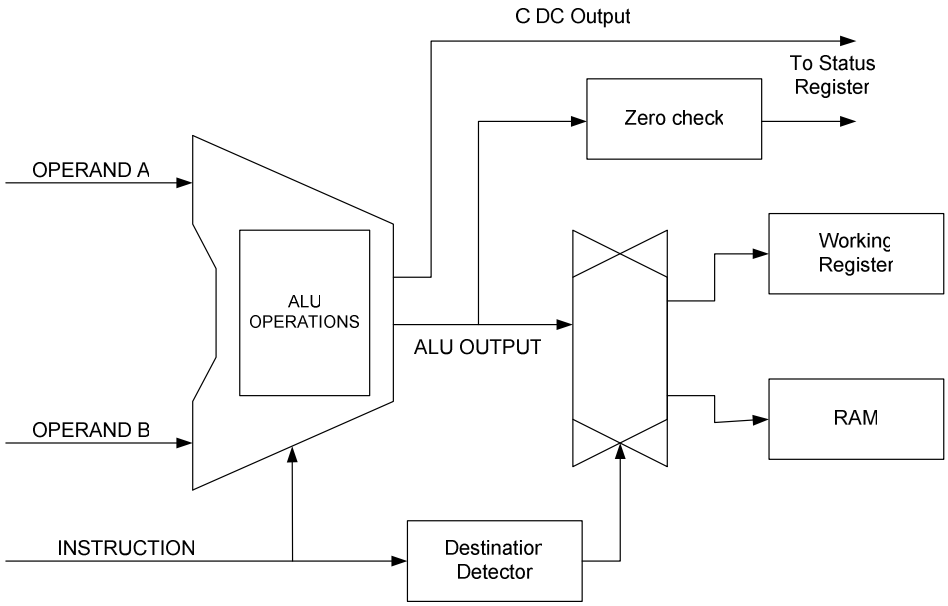


Figure 4.15. Block Diagram of the ALU

The Operand registers of the ALU are prepared within the FSM of the microcontroller. They depend on the instruction which is being executed. The detailed information of the preparation of the operand registers will be discussed later in FSM section 4.2.5.6.

The following table summarizes the destination of the ALU output register.

Table 4.4. Destination of the ALU output register

Instruction	Destination
MOVWF BCF BSF CLRF	RAM
MOVLW ADDLW SUBLW ANDLW IORLW XORLW RETLW CLRW MULT	Working Register
MOVF SWAPF ADDWF SUBWF ANDWF IORWF XORWF DECF INCF RLF RRF DECFSZ INCFSZ COMF	Determined from the “d” field of the instruction. It is 7^{th} bit of the instruction register. If it is one destination is RAM else destination is working register
OTHER INSTRUCTIONS	None.

There are nine categories of the operation of the ALU to fulfill the requirements of the instruction. These ALU operations are explained in the following sections.

4.2.5.5.1. Rotate Left Operation

If the instruction is a rotate left instruction, the content of the operand A register is rotated to the left through the carry bit of the status register.

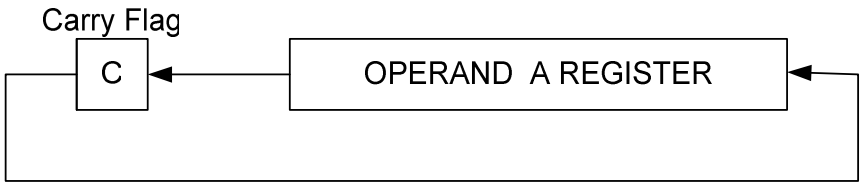


Figure 4.16. Rotate Left Operation

The result of the operation will be determined by the destination determination logic, whether to be written to the working register or RAM.

4.2.5.5.2. Rotate Right Operation

Rotate right operation is similar to the rotate left operation except the rotation direction, which is opposite .The content of the operand A register is rotated to the right through the carry bit of the status register.

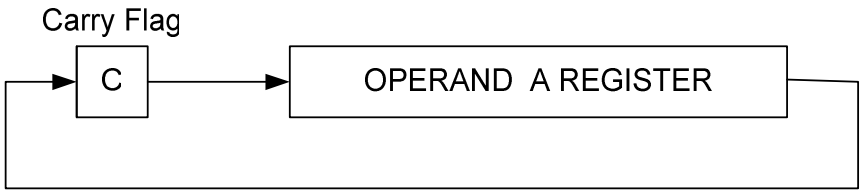


Figure 4.17. Rotate Right Operation

The result of the operation will be determined by the destination determination logic, whether to be written to the working register or RAM.

4.2.5.5.3. Swap Nibbles Operation

The upper and lower nibbles of the operand A register is swapped. For example if the content of the operand A register is “0x73”, after the swap nibbles operation the content of the ALU output register will be “0x37”.

4.2.5.5.4. Complement Operation

The ALU output register is loaded with the 1’s complement of the operand A register. The result of the operation will be assigned by the destination determination logic. As an example, if the content of the operand A register is “0x55”, after the complement operation, ALU output will be “0xAA”.

4.2.5.5.5. Logical AND Operation

With this logical AND operation 5 instructions are executed, which are ANDLW, ANDWF, BCF, BTFSC, BTFSS. The necessary inputs for the ALU, operand A and operand B registers are prepared by the finite state machine of the microcontroller. The necessary status bits are also updated according the result of the operation. ALU output destination is also resolved by the destination determination logic.

4.2.5.5.6. Logical OR Operation

IORLW, IORWF, BSF instructions are considered in this group. The operand A and operand B are ORed and result is written to the ALU output register. Zero flag of the status register is affected after the “or” operation. The ALU inputs are prepared by the finite state machine of the microcontroller.

4.2.5.5.7. Logical XOR Operation

The contents of the operand A register are XOR’ed with the contents of the operand B register. And the result is written to the destination, whether it is working register or RAM. Zero flag of the status register is also affected by this operation.

4.2.5.5.8. Addition Operation

One of the most frequently used operator is addition operator. The operator is used by the 8 instruction which are, ADDLW, ADDWF, SUBLW, SUBWF, DECF, DECFSZ, INCF, and INCFSZ. The operand A and operand B are added with each other and written to the ALU output register. INCF and INCFSZ instructions are added by one with the value of destination of the RAM. DECF and DECFSZ instructions are added by the "0xFF" value with the value of destination of the RAM. Adding a number with 0xFF means decrement by one. Subtraction operation is also carried out by this operator. The number which is going to be subtracted from a number can be implemented by addition operator. The addition of the first number and the 2's complement of the second number gives the subtraction of first number from second number.

The result of the operation affects the status flags which are carry flag, digit carry flag, and zero flag.

4.2.5.5.9. 4-bit Multiplication Operation

A 4bit multiplication operation is executed, and the result is written to the ALU output register. For example if the operand A has a value of "0x5" and operand B has a value of "0xD", the value of the ALU output register is "0x41". The ALU output register will be written to the working register or RAM, which is decided by the destination determination logic. This instruction is not available in the original PIC configuration. Multiplication operation is designed in a separate file, called "mult.v" in Figure 4.4.

4.2.5.5.10. Pass Through Operation

Some instructions do not need any operation like NOP, CALL, GOTO instructions. At this operation the operand A is reflected to the ALU output register and, none of the status flags are affected. The ALU output register is written to neither RAM nor working register, since no operation is carried out.

4.2.5.6. FSM Machine

The flow diagram of the finite state machine (FSM) can be seen in the following Figure 4.19. At the reset condition, the microcontroller starts from the S1 state. This FSM basically have 4 states which are STATE_S1, STATE_S2, STATE_SINT, and STATE_SLE. This FSM is a mealy type state machine as shown in Figure 4.18. The outputs of the FSM are decided with the current state and FSM inputs.

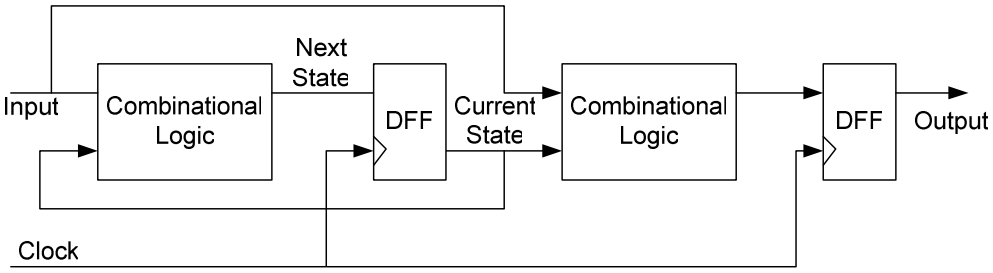


Figure 4.18. Synchronous Mealy Model State Machine

Different with the normal Mealy FSM, the synchronous Mealy FSM has their output connected to flip-flops. That is why it is called synchronous. There are two combinational logics in the state machine, one to generate the next state based on the input and current state, while the other is used to generate the outputs based also on the input and current state.

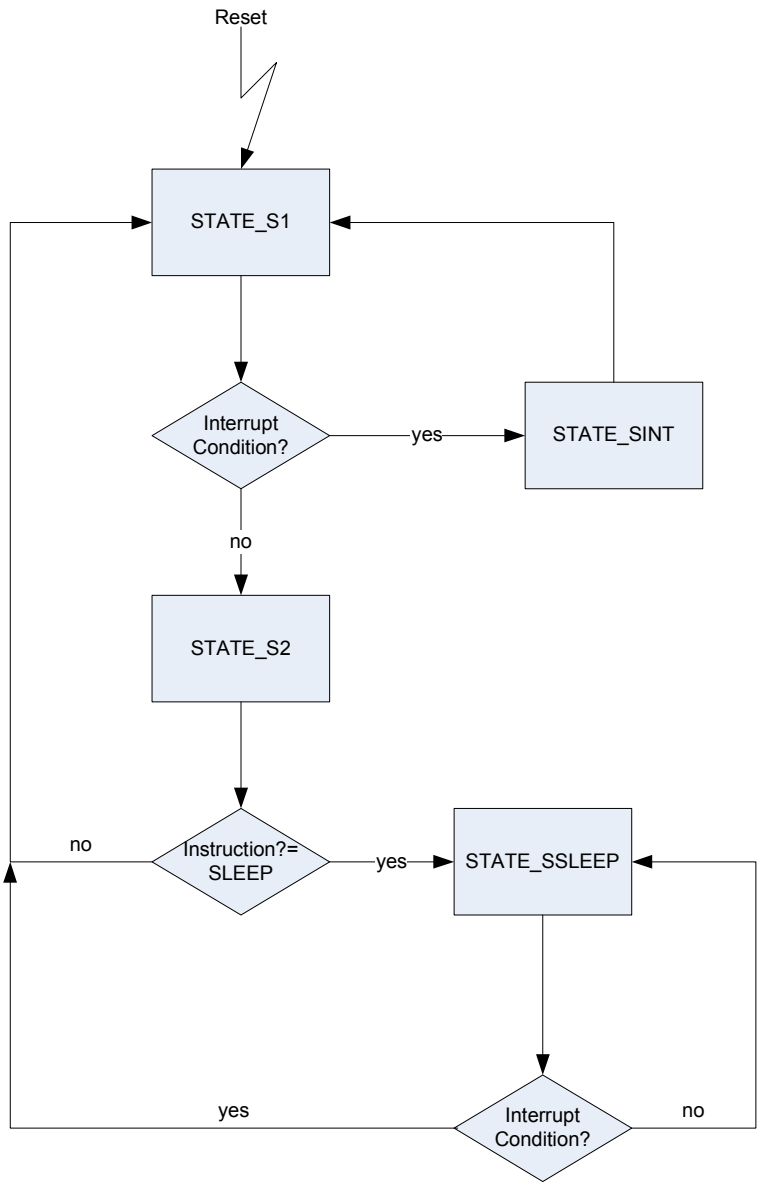


Figure 4.19 Flowchart of the Finite State Machine

The detailed description of each state will be given in the following sections one by one. The finite state machine is implemented in verilog file “fsm.v” as shown in Figure 4.4.

4.2.5.6.1. STATE S1

This state's basic purpose is to read the data from RAM or registers and then decide the value of the operand A and operand B register for the arithmetic logic unit operations. Both operand A and operand B registers are set at the same clock.

The following table summarizes the how the value of the operand A register is loaded at this state.

Table 4.5. The value of the Operand A register

Instruction	Value of the Operand A Register
MOVWF, SWAPF, ADDWF, SUBWF, ANDWF, IORWF, DECF, INCF, RLF, RRF, BCF, BSF, BTFSC, BTFSS, DECFSZ, INCFSZ, COMF , XORWF	The value of the calculated internal RAM. If the direct addressing mode is used, it is loaded with the value of the destination RAM. If indirect addressing mode is used then it is loaded with the value of RAM which is pointed with the FSR register.
MOVLW, ADDLW, SUBLW, ANDLW, IORLW, RETLW, XORLW	These instructions are immediate value instructions. Operand A register is loaded with the first 8 bit of instruction register
CLRF, CLRW	Zero is loaded to the Operand A register
MULT	First 4 bit of the working register is loaded to the Operand A register.
Other Instructions	Operand A is loaded with the value of working register

At this state operand B register is also prepared. The value of the operand B register is determined by the following Table 4.6.

Table 4.6. The value of the Operand B register

Instruction	Value of the Operand B Register
DECF, DECFSZ	A -1 is loaded to the operand B register. I.E, 0xFF is loaded to the operand B register.
INCF, INCFSZ	0x01 is loaded to the operand B register
SUBLW, SUBWF	2's complement of the value of the working register is loaded to the operand B register.
BCF	Complement of the mask node register is loaded to the operand B register. Mask node register is derived from the instruction register.
BTFSC, BTFSS, BSF	The value of the mask node register is loaded to the operand B register.
MULT	Second nibble of the working register is loaded to the Operand B register.
Other Instructions	Operand B is loaded with the value of working register

At this state if the instruction is a return instruction then pop stack operation is also performed.

4.2.5.6.2. STATE S2

This state is an execution and writing results state. The results of the ALU output register is written to the appropriate locations. And also necessary status flags are updated.

If the current executing instruction is a CALL instruction then, the current program counter is PUSHed to the stack. Stack pointer is also incremented by one. If the instruction is a RETFIE instruction, then global interrupt enable bit is also set. Carry, digit carry and zero flags of the status register are also updated according to the result of the ALU.

If the ALU output destination is working register, the result is written to the working register else the results are written to the destination of the RAM.
If the executing instruction is a SLEEP instruction then, state of the FSM goes the STATE_SLE else the next state will be STATE_S1.

4.2.5.6.3. STATE INT

When the FSM enters to this state, FSM disables the global interrupt enable bit at the INTCON register. This action is taken place to prevent a second interrupt generation. Interrupt flag of the INTCON register is also set to inform the microcontroller that an interrupt condition occurred.

Program counter is also pushed to the stack, so that pre-empted instruction can be restarted later, after the RETFIE instruction is executed. After pushing the program counter to the stack, stack pointer is also incremented by one. The next state of the FSM will be STATE_S1.

4.2.5.6.4. STATE SLE (SLEEP)

At this state microcontroller do nothing until an interrupt condition occurs. If an interrupt condition occurs then next state will be STATE_S1 else the microcontroller waits at this state infinitely. The main purpose of this state is to reduce power consumption. If no switching occurs within the FPGA then static power consumption reduces. In the original configuration of the microcontroller, this instruction also stops the oscillator of the microcontroller.

4.2.5.7. Interrupts

The original PIC microcontroller have many interrupt sources, but in this microcontroller only the PORTB interrupt is designed for the simplicity. The other interrupts can be designed in the same manner. INTCON register is used in the control and the status of the interrupts. The interrupt control register, INTCON,

records individual flag bits for core interrupt requests. It also has various individual enable bits and the global interrupt enable bit (GIE).

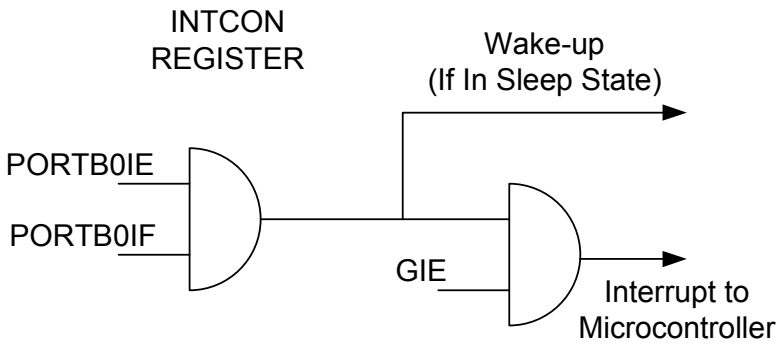


Figure 4.20. Interrupt Logic

The Global Interrupt Enable bit, GIE (INTCON<7>), enables (if set) the un-masked interrupt, or disables (if cleared) the interrupt on PORTB0. PORTB0 interrupt can be disabled through its corresponding enable bit (PORTB0IE) in the INTCON register. The GIE bit is cleared on reset. The “return from interrupt” instruction, RETFIE, exits the interrupt routine as well as sets the GIE bit, which allows any pending interrupt to execute.

When an interrupt is responded to, the GIE bit is cleared to disable any further interrupt, the return address is pushed into the stack and the PC is loaded with 0004h. Once in the interrupt service routine the source of the interrupt can be determined by polling the interrupt flag bits. The interrupt flag bit must be cleared in software before re-enabling the global interrupt to avoid recursive interrupts.

The external interrupt on the PORTB<0> pin is positive edge triggered. When a valid positive edge appears on the PORTB<0> pin, the PORTB0IF flag bit (INTCON<1>) is set. This interrupt can be enabled/disabled by setting/clearing the PORTB0IE enable bit (INTCON<4>). The PORTB0IF bit must be cleared in software in the interrupt service routine before re-enabling this interrupt. The

PORTB<0> interrupt can wake-up the processor from SLEEP, if the PORTB0IE bit was set prior to going into SLEEP.

During an interrupt, only the return PC value is saved on the stack. Typically, if the user wants to save key registers during an interrupt e.g. W register and STATUS register, this has to be implemented in software as in the original PIC microcontroller. The interrupt is implemented in verilog file “fsm.v” as shown in Figure 4.4.

4.2.5.8. Input / Output Ports

General purpose I/O ports can be considered the simplest of the peripherals. They allow us to monitor and control devices outside the microcontroller. For all ports, the PORTs pin’s direction (input or output) is controlled by the data direction register, called the TRIS register. TRIS<x> controls the direction of PORT<x>. A ‘1’ in the TRIS bit corresponds to that pin being an input, while a ‘0’ corresponds to that pin being an output.

The designed microcontroller has 4 I/O ports which are PORTA, PORTB, PORTC, and PORTD. Only the PORTA is 5 bit, the other ports are all 8 bit wide. The first bit of the PORTB has interrupt functionality, although this interrupt input can be configurable in the top module of the verilog code. At the reset state of the FSM, all PORTs direction registers are loaded with 0xFF, so the default direction of the ports is input. This part is implemented in verilog file “top.v” as shown in Figure 4.4.

4.3. Differences Between PIC16XX and The RISC Microcontroller

There are some architectural differences between the original architecture of the PIC microcontroller and the RISC microcontroller that we designed in this thesis. Normally there is not any hardware multiplication instruction in the original configuration. The multiplication instruction makes a 4-bit multiplication with the nibbles of the working register. If we extend the instruction register we may also

add some custom instructions to the microcontroller. This work showed us that it is possible to add some application specific instructions to the design.

In this design we may also extend the I/O ports of the microcontroller. The limiting factor for the number of the I/O port is the number of the I/O pads of the FPGA.

Stack size is also configurable in this configuration. In the original microcontroller, there is only 8 word size stack only, but in our case we have implemented the stack size as 16 word sizes.

In the original PIC microcontroller one instruction execution requires 4 clock periods, but the designed microcontroller requires only 2 clock periods for one instruction execution.

CHAPTER 5

SIMULATION AND TESTING OF THE MICROCONTROLLER

5.1. Test Methodology

Language based models of a circuit must be verified to assure that their functionality conforms to the specification for the design. Two methods of verification are used: logic simulation and formal verification. Logic simulation applies stimulus patterns to a circuit and monitors its simulated behavior to determine whether it is correct. Formal verification uses mathematical proofs to verify a circuit's functionality without having to apply stimulus patterns. Although the use of formal methods is increasing, due to the difficulty of fully simulating large circuits, logic simulation is still widely used. Only logic simulation will be considered in this thesis.

Today, in the era of ASICs, FPGAs and System on a Chip designs, verification consumes about 70% of the design effort. Design teams include engineers dedicated to verification. The number of the verification engineers is usually twice the number of designers [27].

5.2. Testing Environment

As a testing environment, industry standard ModelSim SE simulator is used. ModelSim SE is the high-end simulator available from Mentor Graphics. More information can be found in the ModelSim SE User's Manual[28].By creating

verilog based testbenches any type of stimulus can be provided to the designed logic system.

The term “testbench”, in verilog, usually refers to the code used to create a predetermined input sequence to a design, and then observe the design. Some external data files may also be included to the testbenches, such as program data file.

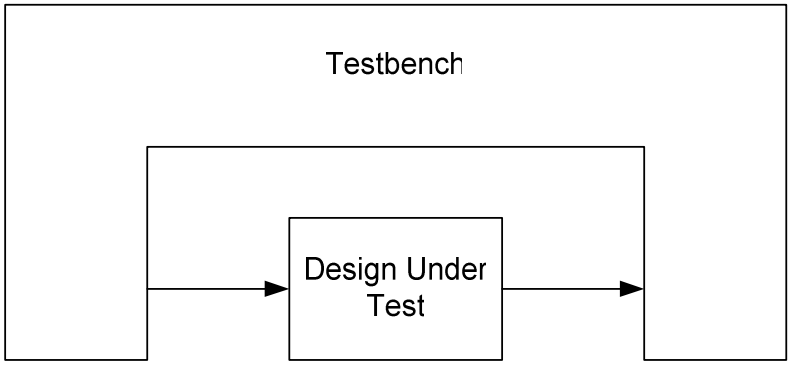


Figure 5.1 Structure of a Testbench and Design Under Test

Figure 5.1 shows how a testbench interacts with a Design Under Test (DUT). The testbench provides inputs to the design and monitors any outputs. The testbench is a completely closed system, no inputs or outputs go in or out. The testbench is effectively a model of the universe as far as the design is concerned. The verification is to determine what input patterns to supply to the design and what is the expected output of a properly working design.

In this thesis, a testbench is created to test the designed microcontroller named “riscmcu_sim.v” as shown in Figure 4.4. The testbench structure for the designed microcontroller is shown in Figure 5.2.

A virtual data memory is created first and directly interfaced to the microcontroller. It behaves as a real RAM. The contents of the memory are initially loaded as unknown. During the run-time of the simulation the contents of

the data memory can be visualized by the memory tool of the ModelSim simulator.

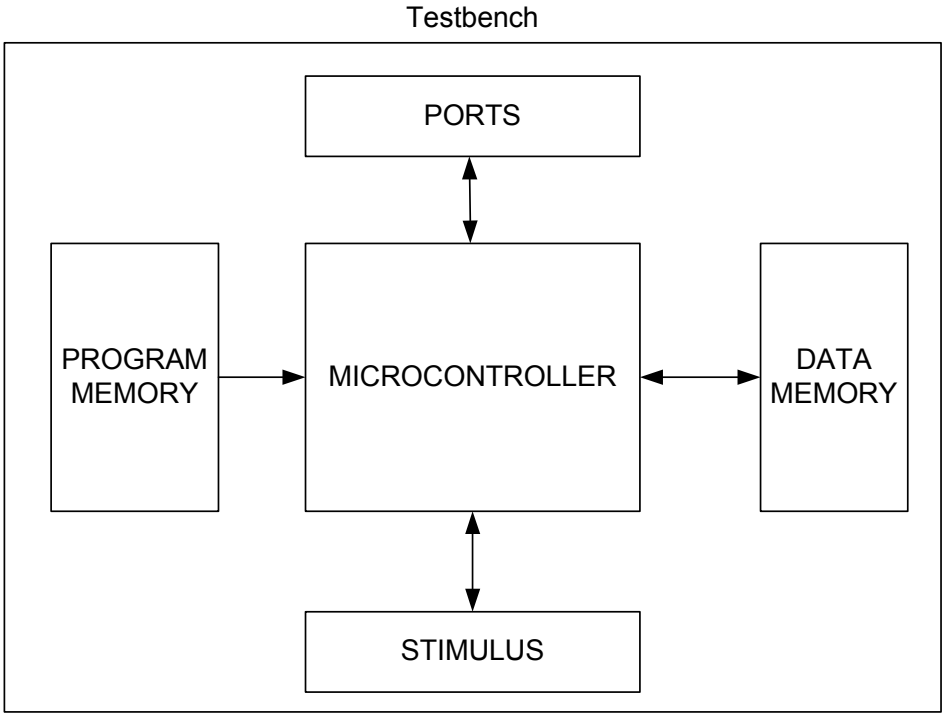


Figure 5.2 Microcontroller Testbench Structure

An asynchronous ROM file is also created in the test bench which simulates the program memory of the microcontroller as the same way with the Data Memory. The necessary clock and reset inputs are also supplied through the testbench. In the testbench input output ports are used to simulate microcontroller. Figure 5.3 shows the test flow of the microcontroller. Initial contents of the program memory are loaded with a “riscmcu.rom” file. A sample test program is written which is shown in Appendix C. This test program simply tests all instructions of the microcontroller. This file is compiled with HI-TECH C compiler and then the memory contents are exported as “riscmcu.rom”. Only the multiplication instructions machine code must be changed manually, because the multiplication instruction is a custom instruction. The testbench and all the design files are given

as an input to the ModelSim simulator. ModelSim simply checks the syntax and compiles all the verilog files. Also wave window in the ModelSim should be opened and necessary signals must be included to the wave window to see the status of the signals in the microcontroller.

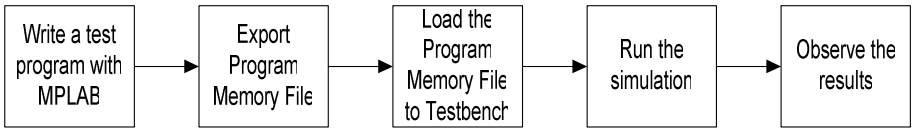


Figure 5.3 Microcontroller Test Flow

5.3. Checking the Results

In a functional verification environment, using a waveform viewer to determine the correctness of a design involves interpreting the dozens of signals on some expectations. It can be an acceptable verification method if used a few signals. But as the number of signal increases, and the number of transitions increases, and the duration of the simulation that must be checked increases, and the number of times simulation results must be checked increases, the probability of a functional error is increases.

ModelSim can compare two sets of waveforms. One is assumed to be a golden reference, while the other is verified for any difference. Golden reference must be checked manually and carefully.

An example waveforms is shown in the following figure;

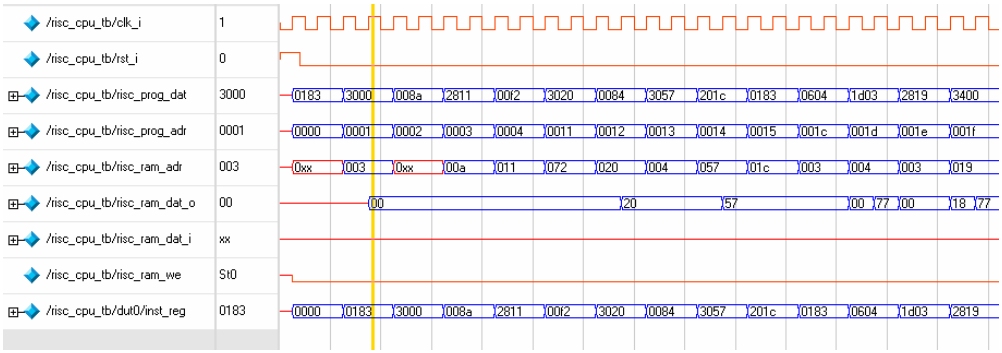


Figure 5.4 Microcontroller Simulation Startup

As shown in Figure 5.4, microcontroller successfully starts to operate, and load the instructions from the memory.

Finally, after verifying the design by the simulator, the same test program is also loaded onto the Digilent demo board, and the results are checked on the real hardware.

CHAPTER 6

CONCLUSIONS

6.1. Conclusions

In this thesis, we introduced a custom designed RISC microcontroller, whose instructions were based on industry standard Microchip PIC microcontrollers. In today's engineering applications, 8-bit microcontrollers play an important role to realize the designs. From a simple toy to complicated satellite systems 8-bit microcontrollers are widely used. This custom microcontroller is designed and implemented by using an FPGA.

The industry trend for microprocessor design is for Reduced Instruction Set Computers (RISC) designs. By implementing fewer instructions, the chip is able to dedicate some of the precious silicon real-estate for performance enhancing features. The benefits of RISC design are a smaller chip, smaller pin count, and very low power consumption. Also RISC architecture is more convenient to the C compilers.

The FPGA is an integrated circuit that contains many (64 to over 10,000) identical logic cells that can be viewed as standard components. Each logic cell can independently take on any one of a limited set of personalities. The individual cells are interconnected by a matrix of wires and programmable switches. A user's design is implemented by specifying the simple logic function for each cell and selectively closing the switches in the interconnect matrix. The array of logic cells

and interconnects form a fabric of basic building blocks for logic circuits. Xilinx Spartan™ FPGA is used in this thesis which is ideal for low-cost, high volume applications.

The design specifications are derived from the PIC microcontroller user manual. Verilog hardware description language is used to achieve the design considerations of the RISC microcontroller. Then a Verilog code is written with some modifications to the instruction set and extra ports. The designed Verilog code is synthesized with the Xilinx ISE program. A test platform is established by using a Digilent demo board. A program downloader application is written with using Borland C++ Builder on the PC to load the firmware to the microcontroller in the SPARTAN IIE FPGA, and then the results are checked whether it meets the design specification.

The microcontroller in the FPGA occupied roughly 30 % resources of the FPGA. Detailed design summary can be found in Appendix B. It means that we have free space to implement some other functions in the FPGA.

The designed microcontroller has some extra functionality with respect to the original configuration. There is not any hardware multiplication instruction in the original configuration. In this configuration a multiplication instruction is added to the instruction list. And the I/O ports are expandable, according to the used requirements.The PIC16XX microcontrollers divide the incoming clock by 4, So the pipeline of the original microcontroller is formed-up by the 4 clock cycles. But the designed microcontroller in this thesis has been implemented by 2 clock pipeline. This is an extra performance increasing factor with respect to the original configuration.

One important result of this thesis is, in the future we can implement an ASIC with the Verilog code which has already been developed in this thesis.

6.2. Future Work

In the future, I intend to improve the instruction set of the designed microcontroller core . The current microcontroller have instructions , which are 14-bits wide . If we extend the instructions to the 16-bit, we shall have new opportunity to add custom instructions the microcontroller core. By this way , we can implement very specific functions such as DSP operations in the FPGA very easily. We may improve the multiplication instruction to the 8 bit multiplication even 16-bit multiplication.

Also we can add specific peripherals to the microcontroller, such as PWM timers, and I2C bus. The utilization of the FPGA is currently 30 % , so we have much area remaining in the FPGA to implement such peripherals and instructions. But all these works must keep the compatibility to the previous microcontroller core.

REFERENCES

[1] Enoch O. Hwang, "Microprocessor Design Principles and Practices with VHDL", 2004

[2] John L. Hennessy, David A. Patterson, "Computer Architecture: A Quantitative Approach", May 2002

[3] Mark Balch, "Complete Digital Design : A Comprehensive Guide to Digital Electronics and Computer System Architecture", 2003

[4] Microchip Tech. Inc., http://www.microchip.com, Last accessed ; June 2005

[5] Xilinx Inc. , http://www.xilinx.com/products/design_resources/design_tool/, Last accessed ; June 2005

[6] Mentor Graphics Inc. , http://www.model.com/products/60/default.asp, Last accessed ; June 2005

[7] HI-TECH Software Inc., http://www.htsoft.com/products/picccompiler.php, Last accessed ; June 2005

[8] Borland Software, http://www.borland.com/us/products/cbuilder/index.html, Last accessed ; June 2005

[9] Digilent Inc., "Digilab 2E System Board Reference Manual," (http://www.digilentinc.com) , Last accessed ; Feb 2005

[10] Doulos Inc., http://www.doulos.com/knowhow/vhdl_designers_guide/, Last accessed ; June 2005

[11] IEEE Computer Society , "IEEE Standard VHDL Language Reference Manual", May 2002

[12] Doulos Inc., http://www.doulos.com/knowhow/verilog_designers_guide/, Last accessed ; April 2005

[13] IEEE Computer Society , "IEEE Standard Verilog Hardware Description Language ", September 2001.

[14] Karen Parnell, Nick Metha , “Programmable Logic Design Quick Start Handbook, Second Edition”, January 2002

[15] Xilinx Inc. , “Spartan-IIE 1.8V FPGA Family: Complete Data Sheet”, July 2004

[16] Don Bouldin, “Designing Application-Specific Integrated Circuits” http://vlsi1.engr.utk.edu/ece/bouldin_courses/551/overview_bw.pdf, Last accessed ; June 2005

[17] Bob Zeidman, **“**An Introduction to FPGA Design**,"** Embedded System Conference, November 1999.

[18] Bob Zeidmen, Introduction to CPLD and FPGA Design, The Chalkboard Network, http://www.chalknet.com , Last accessed ; June 2005

[19] Xilinx Inc, **“**High speed FIFOs in Spartan-II FPGAs**,"** Xilinx Application Note XAPP175, 1999.

[20] R. Pragasam, “Spartan FPGAs - The gate array solution," Xilinx Application Note XAPP120, August 2001.

[21] Microchip Technology Inc.,” PICmicro Mid-Range MCU Family Reference Manual”, December 1997.

[22] Xilinx Inc. , “”Xilinx Libraries Guide for ISE 6.3i”, http://toolbox.xilinx.com/docsan/xilinx7/books/docs/lib/lib.pdf , Last accessed ; December 2004.

[23] Verilog-2001 Frequently Asked Questions, http://www.sutherland-hdl.com/Verilog-2001/verilog-2001_faq.html, Last accessed; June 2005

[24] Xilinx Inc, **“**Using Block SelectRAM+ Memory in Spartan-II FPGAs**,"** Xilinx Application Note XAPP173, December 2000.

[25] Xilinx Inc, **“**High Using Delay-Locked Loops in Spartan-II FPGAs**,"** Xilinx Application Note XAPP174, January 2000.

[26] Donald E. Thomas , Philip R. Moorby “The Verilog® Hardware Description Language, Fifth Edition”,2002

[27] Janick Bergeron, “Writing Testbenches: Functional Verification of HDL Models, Second Edition”, Feb 2003

[28] ModelSim Inc., “ModelSim SE User’s Manual, Version 6.0”, Jul 2004

APPENDIX A

PROGRAM LOADER USER'S MANUAL

Program Downloader Software has been developed for PC's running Windows operating system. Program Downloader uses all the graphic and component support built in Windows which makes it user friendly software. There are no menus in the program. Because it is a simple program, all functions are done with the buttons and the tabs. The basic program structure and the basic functions of each button are as follows:

- Settings Tab; in this tab serial communication settings are established.
  - From the name pull-down drop list, Serial communication channel can be selected.
  - From the baud rate drop list, baud rate can be selected.
  - From the Communication speed drop list, communication speed can be selected between two commands.
  - Open button, opens a serial communication with the desired baud rate at the selected channel
- Commands Tab; In this tab , the compiled Intel hex file is loaded to the memory of the PC
  - File Load button loads the Intel hex file.
  - File Unload button clear the memory of PC.
  - Write Button sends the sequentially the binary data to the microcontroller to the FPGA
  - Read button reads some memory content from the FPGA

APPENDIX B

DESIGN SUMMARY AND RESULTS

The result of the mapping stage is as follows;

Design Information

Target Device : x2s200e

Target Package : pq208

Target Speed : -6

Mapper Version : spartan2e -- $Revision: 1.16.8.2 $

Mapped Date : Fri April 15 21:07:14 2005

Design Summary

Logic Utilization:

Number of Slice Flip Flops : 498 out of 4,704 10%

Number of 4 input LUTs : 1,196 out of 4,704 25%

Logic Distribution:

Number of occupied Slices: 711 out of 2,352 30%

Number of Slices containing only related logic: 711 out of 711 100%

Number of Slices containing unrelated logic: 0 out of 711 0%

Total Number 4 input LUTs : 1,255 out of 4,704 26%

Number used as logic	: 1,196
Number used as a route-thru	: 51
Number used as 16x1 RAMs	: 8
Number of bonded IOBs	: 36 out of 142 34%
IOB Flip Flops	: 30
Number of Tbufs	: 144 out of 2,464 5%
Number of Block RAMs	: 5 out of 14 35%
Number of GCLKs	: 3 out of 4 75%
Number of GCLKIOBs	: 1 out of 4 25%
Number of DLLs	: 1 out of 4 25%

Total equivalent gate count for design: 102,646

APPENDIX C

TEST CODE FOR THE MICROCONTROLLER

```
// Filename        : riscmcu_test.c
// Description     : RISC MICROCONTROLLER TEST SOFTWARE
// Author          : Rasit GUMUS
// Created On      : MPLAB 7.01 -HITECH PICC C Compiler 8.02
// Last Modified By: .
// Last Modified On: .
// Update Count    : 0
// Status          :  This test program simulates all of the features of the mcu
#include <pic.h>
void delay(unsigned char data);
void long_delay(void);
unsigned char soft_mult(unsigned char x, unsigned y);
void interrupt interrupt_service(void);

void main(void) {
unsigned char ch1,ch2;
TRISA=0x00; // PORTA is output
TRISB=0xFF; // PORTB is input
TRISC=0x00; // PORTC is output
TRISD=0xFF; // PORTD is input
INTCON=0xFE; // enable interrupt
```

```
ch1=1;ch2=2;
// The following instructions are tested with the following  software multiplication
// routines. ADDWF, CLRF, INCF, INCFSZ, MOVWF, NOP, RLF, RRF, SUBWF, XORWF, BCF, BSF
// CALL, GOTO, BTFSC, BSF, BCF, RETURN
for (ch1=0 ; ch1<16 ; ch1++)
 for (ch2=0 ; ch2<16 ; ch2++)
{
PORTC=soft_mult(ch1,ch2);
delay(250);delay(250);
}
PORTC=0xAF;
while ( (PORTD&0x0F) == 0) ;
long_delay();
// TEST FOR ANDWF
PORTC=PORTC & 0x0F ;
while ( (PORTD&0x0F) == 0) ;
long_delay();
// TEST FOR COMF
while ( (PORTD&0x0F) == 0)
        PORTC = ~PORTB;
long_delay();
// TEST FOR ADDLW
while ( (PORTD&0x0F) == 0)
        PORTC = PORTB+0x07;
long_delay();
// TEST FOR SUBLW
while ( (PORTD&0x0F) == 0)
{
asm("MOVF portb,W");
```

```
asm("SUBLW 0x10");
asm("MOVWF portc");
}
long_delay();
// TEST FOR ANDLW
while ( (PORTD&0x0F) == 0)
{
asm("MOVF portb,W");
asm("ANDLW 0xAA");
asm("MOVWF portc");
}
long_delay();
// TEST FOR XORLW
while ( (PORTD&0x0F) == 0)
{
asm("MOVF portb,W");
asm("XORLW 0xFF");
asm("MOVWF portc");
}
long_delay();
// test for multiplication instruction
while ( (PORTD&0x0F) == 0)
{
asm("NOP");
asm("NOP");
asm("NOP");
asm("BCF 0x3, 0x5");
asm("MOVF portb,W");
asm("xorlw 0xAB"); // the value at this address should be changed
asm("MOVWF portc");// from 0x3A to 0x3B to make multiplication
```

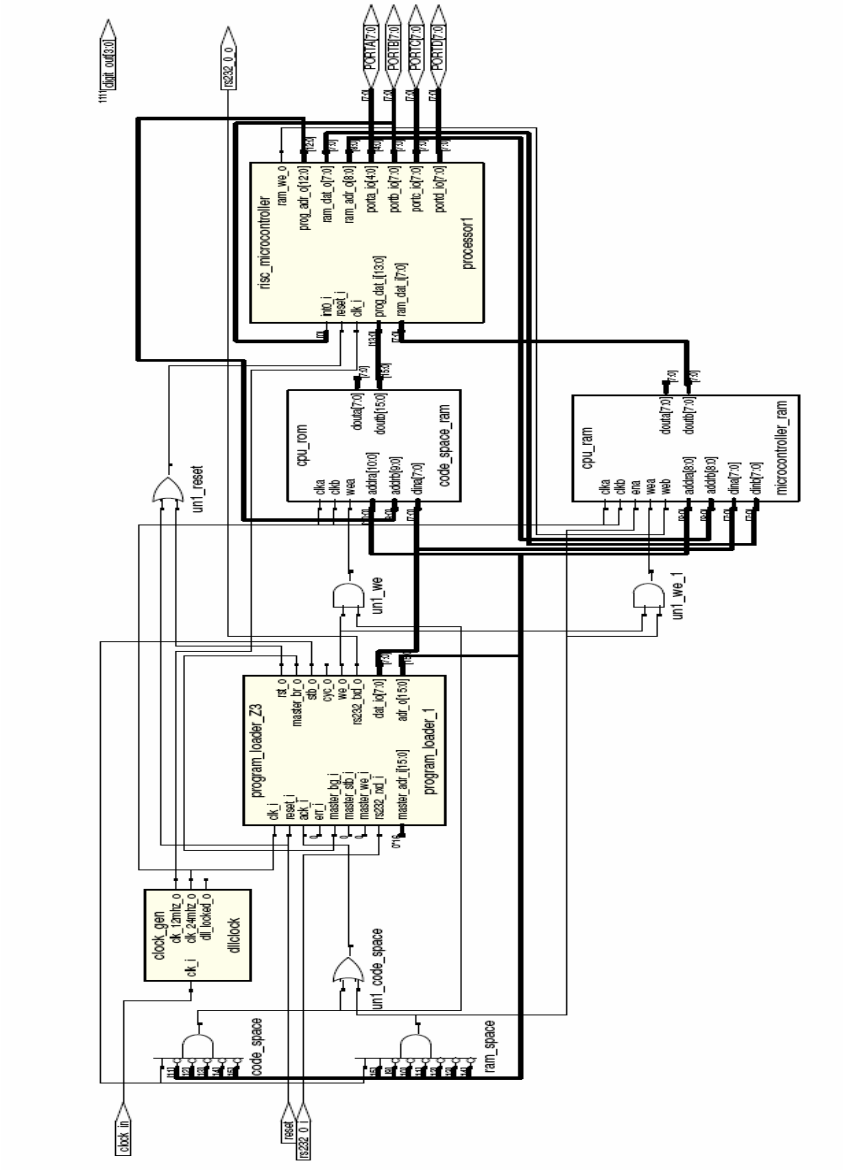
```
                    // in the produced rom file
}
long_delay();
// TEST FOR ROTATE LEFT
while ( (PORTD&0x0F) == 0)
{
ch1=0x01;
for (ch2=0;ch2 < 8 ; ch2++)
 {
//shift operations
PORTC=ch1;
ch1=ch1<<1;
long_delay();
}
}
long_delay();
// TEST FOR ROTATE RIGHT
while ( (PORTD&0x0F) == 0)
{
ch1=0x80;
for (ch2=0;ch2 < 8 ; ch2++)
 {
//shift operations
PORTC=ch1;
ch1=ch1>>1;
long_delay();
}
}
long_delay();
asm("SLEEP");
```

```
asm("NOP");

PORTC=0xF0;
while(1); // wait here forever
}
void delay(unsigned char data)
{
unsigned char i,k;
for (i=0; i<data ; i++)
  for (k=0; k<255 ; k++)
          asm("nop");
}
void long_delay(void)
{
delay(250);delay(250);
delay(250);delay(250);
delay(250);delay(250);
delay(250);delay(250);
}
unsigned char soft_mult(unsigned char x, unsigned y)
{
return x*y;
}
void interrupt interrupt_service(void)
{
PORTC=0xFF;
PORTA=~PORTA;
}
```

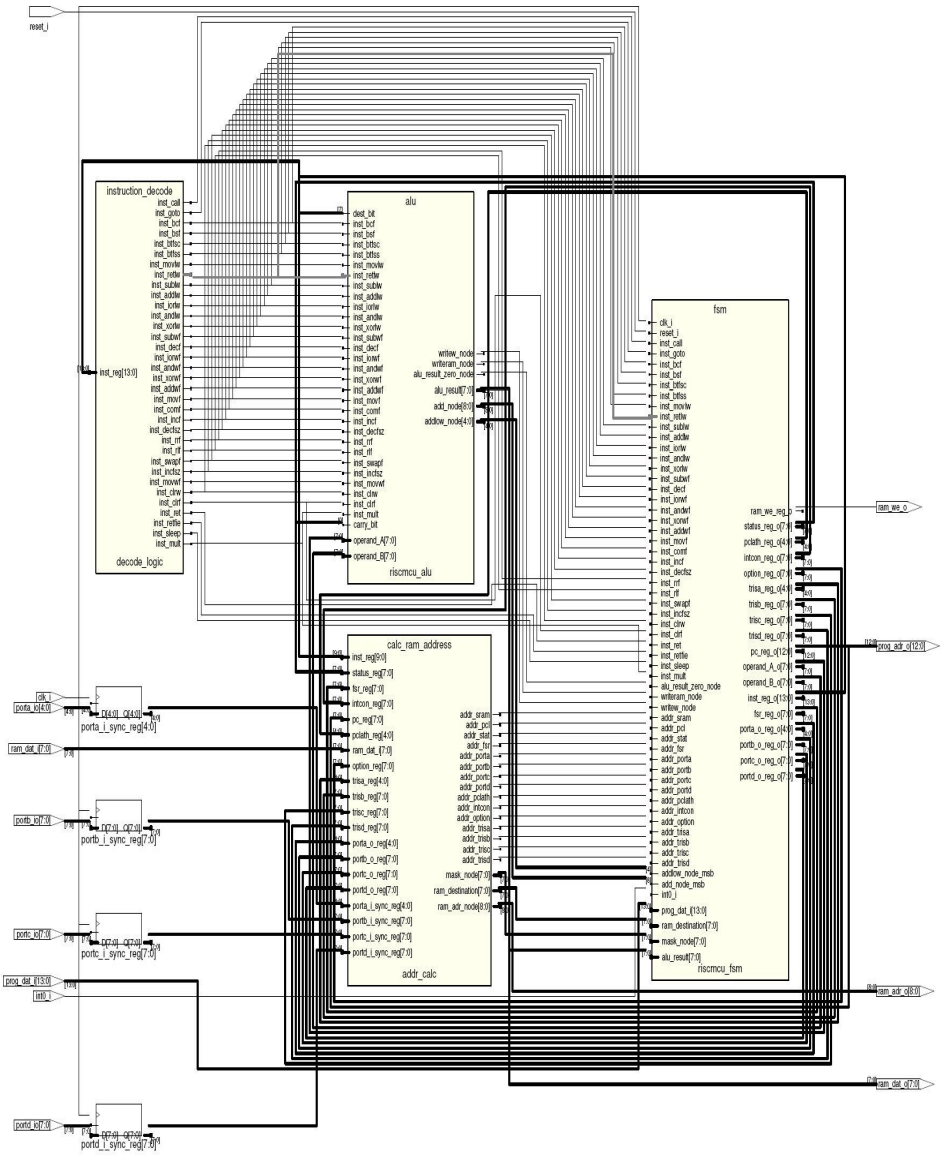
APPENDIX D

INTERCONNECTION DIAGRAM FOR THE TOP MODULE



APPENDIX E

INTERCONNECTION DIAGRAM FOR THE RISC MICROCONTROLLER MODULE



APPENDIX F

SOURCE FILES FOR THE RISC MICROCONTROLLER

The source files for the RISC microcontroller are located on the CD-ROM attached to the back cover of the thesis.