

# **IMPLEMENTATION OF 16 BIT RISC MICROCONTROLLER ON FPGA**

**A DISSERTATION**

*Submitted in partial fulfillment of the  
requirements for the award of the degree*

*of*

**MASTER OF TECHNOLOGY**

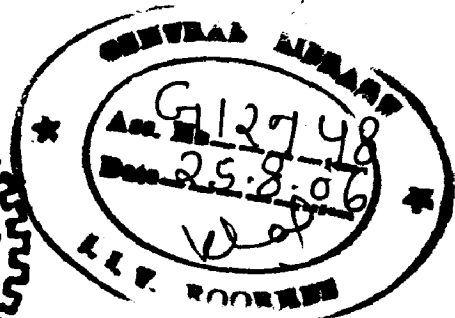
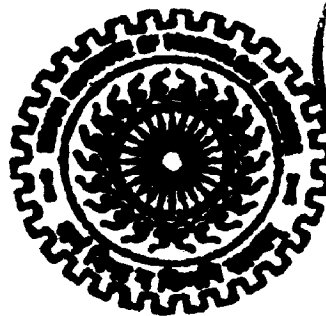
*in*

**ELECTRICAL ENGINEERING**

*(With Specialization in System Engineering & Operations Research)*

**By**

**MOHITE ARUN PANDURANG**



**DEPARTMENT OF ELECTRICAL ENGINEERING  
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE  
ROORKEE -247 667 (INDIA)  
JUNE, 2006**

## CANDIDATE'S DECLARATION

I hereby declare that the work which is being presented in this dissertation entitled, "IMPLEMENTATION OF 16 BIT RISC MICROCONTROLLER ON FPGA", submitted towards the partial fulfillment of the requirements for the award of the degree of **Master of Technology in Electrical Engineering**, with specialization in **System Engineering and Operations Research**, I. I. T. Roorkee, India, is an authentic record of my own work carried out from June 2005 to June 2006 under the supervision of **Dr. G. N. Pillai & Prof. M. K. Vasantha**, Electrical Engineering Department, Indian Institute of Technology, Roorkee, India.

The matter, embodied in this dissertation report, has not been submitted for the award of any other degree or diploma.

Dated:

Place: Roorkee



**MOHITE ARUN PANDURANG**

---

## CERTIFICATE

This is to certify that the above statement made by the candidate is correct to the best of our knowledge.



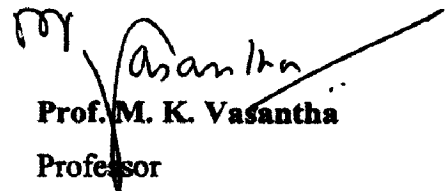
**Dr. G. N. Pillai**

Associate Professor

Electrical Department

IIT Roorkee

Roorkee-247667 (India)



**Prof. M. K. Vasantha**

Professor

Electrical Department

IIT Roorkee

Roorkee-247667 (India)

# ACKNOWLEDGEMENT

*I would like to take this opportunity to express my deepest sense of gratitude to my supervisors **Dr. G. N. Pillai** and **Prof. M. K. Vasantha**, Department of Electrical Engineering, I.I.T. Roorkee, for their invaluable support, guidance and suggestions at various stages of this Dissertation. I feel privileged to be associated under them and it was a great pleasure in learning the practical aspects of digital design and verification under their aegis and guidance. I am very much thankful to them for giving me an opportunity to work on a topic which was very much interesting and challenging for me. I remember with great emotion, the constant encouragement and help extended to me by him that went even beyond the realm of academics.*

*Special mention has to be made of **Dr. (Ms.) Indra Gupta**, in-charge Microprocessor and Computer Laboratory for providing me the computer in the lab and all other facilities including FPGA Kit required for this project as and when needed.*

*The department of Electrical Engineering of this institute provided me with all kinds of necessary facilities for carrying out my work. My sincere thanks are due to **Prof. S. P. Gupta**, Head of Electrical Engineering department and **Prof. H. O. Gupta**, Ex-head of the department, for making the opportunities available although. My sincere thanks go to all the faculty members of the department for the voluntary help, direct and indirect, extended to me during the course of the work.*

*I am also thankful to **Dr. S. Dasgupta**, Department of Electronics and telecommunication, IIT Roorkee, for his invaluable discussion and suggestions, regarding various issues of VHDL coding and FPGA implementation.*

*My sincere regards to staff and my friends at the Department who have directly and indirectly helped me in completing this Report. I am grateful to my hostel mates and my colleagues, for helpful and fruitful discussions and for the good time we spent together.*

*Last, but not least by any means, I wish to acknowledge my family members for giving me the moral strength and constant encouragement. The work could never reach its present status without their constant support and love.*

Dated:

**MOHITE ARUN PANDURANG**

Place : Roorkee

M-Tech (System Engineering & Operations Research)

Department of Electrical engineering, I. I. T. Roorkee.

# CONTENTS

<b>Title</b>	<b>Page</b>
<b>Candidate's Declaration</b>	<b>i</b>
<b>Acknowledgement</b>	<b>ii</b>
<b>Acronyms</b>	<b>vi</b>
<b>Abstract</b>	<b>vii</b>
<b>Ch 1: INTRODUCTION</b>	
1.1 Introduction	1
1.2 Motivation behind the work	2
1.3 Objective of dissertation work	5
1.4 Introduction to Atmel AVR AT90S1200	5
1.5 Overview of Microcontroller and Microprocessors	5
1.6 Comparison of Microcontrollers and Microprocessors	5
1.7 Microcontroller Performance Factors	8
1.8 CISC vs. RISC Architecture	9
1.9 Microcontroller applications	11
<b>Ch 2: MICROCONTROLLER ORGANISATION</b>	
2.1 Tradeoffs in Microcontroller Design	13
2.2 The Microcontroller Operation	13
2.3 Microcontroller organization	15
2.4 Description of the microcontroller design Steps	17
<b>Ch 3: INSTRUCTION SET ARCHITECTURE</b>	
3.1 Atmel RISC microcontroller Architecture	29
3.2 Atmel RISC microcontroller Instruction Set & Addressing Modes	31
3.3 Machine Cycle Sequence	32
3.4 Modeled Architecture	32
3.5 Instruction format	33
3.6 Instruction set summary	35
3.7 Addressing Modes	36
3.8 Brainstorming the Design, the Creative Process	36

# CONTENTS

	<b>Title</b>	<b>Page</b>
<b>Ch 4: PIPELINE UNIT DESIGN</b>		
4.1	Instruction fetch unit	41
	4.1.1 Program counter	41
	4.1.2 Instruction memory	42
	4.1.3 Branch Decide Unit	42
4.2	Instruction decode unit	42
	4.2.1 Control unit	42
	4.2.2 Register file organization	42
	4.2.3 Sign extension unit	43
4.3	Execution unit	43
	4.3.1 Branch adder unit	43
	4.3.2 Arithmetic and logical unit	44
	4.3.3 ALU control unit	45
4.4	Write back unit	45
	4.4.1 Data memory organization	45
	4.4.2 Register write unit	45
	4.4.3 Interrupts and exception handling	45
4.5	Hazards in pipeline unit	46
	4.5.1 Structural hazard	46
	4.5.2 Data hazard	46
	4.5.3 Control hazard	47
4.6	Hazard detection unit	47
4.7	Data forward unit	47
<b>Ch 5: CONTROL UNIT DESIGN</b>		
5.1	Overview of control unit	49
5.2	Instruction decode unit	50
5.3	Control unit	50
5.4	Synchronous Mealy Model Finite State Machine	51
5.5	Finite State Machine States	53

# CONTENTS

	<b>Title</b>	<b>Page</b>
5.6	External Interrupt	56
5.7	I/O Decoder	57
5.8	Branch Evaluation Unit	57
5.9	Timer	58
5.10	Implementation Problems	58
<b>Ch 6:</b>	<b>DESIGNING WITH FPGA</b>	
6.1	FPGA Architecture	59
6.2	Programming with FPGA	60
6.3	FPGA Design Environment	63
6.4	FPGA design flow for implementation	64
6.5	Verification and testing	66
<b>Ch 7:</b>	<b>Results and conclusion</b>	
7.1	Results	67
7.2	Conclusion	67
7.3	future scopes of the work	69
	<b>REFERENCES</b>	70
	<b>APPENDIX</b>	
A:	Atmel AVR microcontroller instruction set	72
B:	Complete instruction set of RISC Microcontroller	74
C:	Simulation results	75
D:	Synthesis report	80

## ABBREVIATION AND ACRONYMS

FPGA	Field Programmable Gate Array
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
RISC	Reduced Instruction Set Computer
MIPS	Million Instructions per Second
SOPC	System on Programmable Chip
VERILOG	An Industry standard HDL (IEEE std. 1364)
XST	Xilinx's Synthesis Tool
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction Processor
CLB	Configurable Logic Block ( For Xilinx FPGA)
LUT	Look Up Table
LAB	Logic Array Block
LE	Logic Element (For Altera FPGA)
CPLD	Complex Programmable logic Devices
PLD	Programmable Logic Device
DDR	Double Data Rate Interface
FCRAM	Fast Cycle RAM
PLL	Phase Locked Loops
Floor planning	process of choosing best connectivity in a design
IEEE	Institute of Electrical and Electronics Engineering
IOB	Input/Output Buffer
IOE	Input/Output Element
VGA	Video Graphic Adaptor

## ABSTRACT

The work presented in this dissertation report describes the implementation of 16-bit RISC Microcontroller in FPGA chip, using VHDL programming at Xilinx ISE 7.1i platform supported by Xilinx ISE and Aldec Active HDL simulation environment. This work explores an application area of FPGA to develop application specific integrated circuit (ASIC) as an independent System on Programmable Chip (SOPC) design. The design is targeted to make a feel of a 16-bit Microcontroller available in FPGA. Microcontroller is organized with 4 stage pipelined RISC architecture and supports a total of 81 instructions. Successful synthesis is done and design is downloaded in Xilinx Spartan II FPGA. Output along with internal states of the embedded processor can be seen on seven segment display of Xilinx Spartan II FPGA kit, available with us in microprocessor and computer lab. Synthesis reports and Place & route reports are also provided to verify the design implementation.

This work focuses on the design methodology based on tools and techniques to capture the design and develop a Hardware Prototype of it. Like any other engineering design, Microcontroller designed is tested consistently and made modifications throughout whenever any problem arose. Pipeline has been modified and remapped for the better performance. Simulation is done using Xilinx ISE and Aldec Active HDL simulation environment to perform functionality test of this code. Synthesis optimization tools were used to convert the chip design in to smaller and faster design. Lastly the synthesized design is verified and various synthesis reports are analyzed to evaluate and verify the performance of the designed chip.

Atmel AVR RISC microcontroller (AT90S1200) is chosen as prototype model for design and all the necessary features of AT90S200 is implemented successfully in Xilinx Spartan 2 FPGA. Microcontroller designed in this dissertation can process 74-86 instructions simulation result of all the instructions are been tested and validated successfully by means of hand calculations. All the important features of a microcontroller are implemented in FPGA, which now behaves as is a microcontroller core.



# CHAPTER 1: INTRODUCTION

---

## 1.1 Introduction

In the ever-changing world of technology, new ideas are born and legacy technologies are left aside to be chronicled in the history books. Although new technologies become available, sometimes it is necessary to maintain older technology when servicing electrical systems for maintenance and redesign of existing systems. The lack of money usually prevents the redesign of systems. Since some systems cannot be replaced using new technology, technicians are challenged to maintain electrical systems with parts that are not procurable by commercial buyers and or government purchasers.

Microcontrollers and microprocessors are the most used devices in electronic equipment. Modern technology demands from any engineer, a basic microcontroller or microprocessor knowledge. The basic difference between them is that microprocessors can be configured for the amount of memory and the input / output system used. The microcontroller has all the computing system (I/O system and memory) built in it. Designer's judgment determines which one should be used [1].

The emphasis of this work will be in the design of complete microcontroller with CPU, RAM, ROM and I/O system in FPGA; microcontroller and microprocessor layout, fabrication process and technology are beyond the scope of this work and will not be considered at all. Design performance parameters like speed, power dissipation, wiring, packing, and transistor sizing are also beyond the scope of this work.

The design methodologies used to develop a Behavioral and Instruction Set model for the Atmel AVR AT90S1200 RISC microcontroller and the results of testing these models will be presented. VHDL is used to create the Behavioral model and the Instruction Set model. It is the intent of this dissertation to develop a model for the Atmel AVR AT90S1200 RISC microcontroller. This will allow for an understanding of how to best replace obsolete parts with new components, especially complex parts. The replacement of obsolete or singled sourced parts by emulation of the existing chip or using remaining die at chip supply houses usually yields a cost effective. Sometimes board or system redesigns are necessary to eliminate a high percentage of system or board level obsolete components. This is often very expensive and time consuming.

The state-of-the-art of digital circuit design now provides for an efficient, CAD oriented methodology for implementing digital designs, by using VHDL. During the 60's-70's system level and microcomputer design entailed building systems out of many individual logic gates manufactured on Integrated Circuits (IC). This design technique was very costly and time consuming. As the technology moved forward from medium-scale integration (MSI) to large-scale integration (LSI), to very large-scale integration (VLSI), the need for new design tools became apparent. With the ability to incorporate many functions on one IC, engineers needed a way to quickly design a function and or circuit and test the design. The standard known as VHDL was first created in 1987, [6].

VHDL allows for hardware description in a text based language. VHDL is similar to Ada, a government standardized, portable, and object oriented software language. VHDL allows a design to model a digital system at many levels of abstraction. A description can be as simple as a 2-input logic circuit or an entire digital system. There are five different levels of modeling: Performance modeling, Behavioral modeling, Instruction Set modeling, RTL modeling, and Gate-Level modeling.

The Behavioral model developed is an abstract model of the Atmel RISC microcontroller that demonstrates a basic understanding of how instructions are fetched and executed. The model was written to give a starting point of understanding to the overall design. Hence the Behavioral model will not have any physical implications pertaining to the original microcontroller.

Once the Behavioral model has been written one can then focus on developing an Instruction Set model. The Instruction Set model developed here allows for exercising a subset of Intel MCS-51 instructions. Since Atmel RISC microcontroller is a feature-reduced version of Atmel RISC microcontroller, it uses MCS-51 instructions.

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 cope with designs containing hundreds of thousands of logic gates, memories, high-speed interfaces, and other high-performance components [26].

## **1.2 Motivation behind the work**

The motivation for this work comes after the author took the course, "online Computer Application techniques". The author realizes that microcontroller design

could be an opportunity to summarize and apply most of the electronic engineering basic and advanced courses. Basic electronic course, digital logic circuits and advanced digital design are some of the electrical engineering courses used in this work.

Another motivation for this work lies in the author's desire to learn and master the microcontroller concepts, design and operation. For many years literature has been published regarding microcontroller and digital design. Techniques, methods, and procedures have been published, but most of them are usually explained using a symbolic or algorithmic approach. Some examples of this kind of approach can be found in "Computer Organization and Architecture Principles of Structure and Function [8]. Computer Organization and Design The Hardware / Software Interface" [9], "The Intel Microprocessors 808X, Pentium and Pentium Pro" [10], "Embedded Systems and Computer Architecture"[3],

To grasp the basic concepts at the starting stage, one may feel more comfortable when they see the theoretical materialization, simulation and execution of hardware circuits, instead of large equations, diagrams, algorithms and symbols that most of the microcontroller information sources offer. The hardware implementation of every concept is what makes this work useful for beginners to learn and understand microcontroller concepts.

One of the main features of this work lies in the fact that it follows a series of steps and makes emphasis on the most important points in each and everyone of those steps. Beginners just have to follow those steps in order to design and simulate their own microcontroller. This work illustrates the design, simulation, testing, and implementation of all microcontroller circuits in each step. Through the whole process you will appreciate the complete microcontroller evolution and transformation from zero to a functional unit.

This method provides mechanisms to change some of the microcontroller parts without affecting others. It makes emphasis on modularization. Through the whole process, modules of each part are designed and can be changed individually without affecting the entire system. This allows experimentation and circuit changes to examine what is happening inside [3].

One possible application of this work is that one can transform microcontroller schematic into VHDL code and download it to an FPGA for prototype simulation. This increases understanding of microcontroller concepts and operation, with hands-on experience; one can examine how the instruction execution is and how the

microcontroller circuits work in every instruction. Also multiple versions of one microcontroller can be developed with slight changes, allowing you to observe the effect of those changes in each design and simulate each prototype on FPGA.

A weak point of this method is that it does not achieve an efficient implementation. Performance is not the main point of this work; just delivering the most important microcontroller concepts. The focus of this work is in the methodology, not in the computational capabilities and features of the microcontroller. Besides its educational approach, another important point is that this method provides a mechanism to design a microcontroller that can be simulated, as said before, on FPGA, but also can be used on real applications. In other words, slight changes can produce a different microcontroller for new applications as needed. Users do not have to buy a new microcontroller but try a different one using this method. Of course this is convenient for experimentation or academic purposes only, not for applications where performance is the critical point.

Modern microcontroller costs are relatively low, and are very useful for many applications but sometimes there are situations that are better handled with specially designed microcontrollers for specific applications. For example, a designer may want to build and control his/her own personal robot, with a specific instruction set. Designers can find in the market some inexpensive microcontrollers that suit design requirements. But those popular microcontrollers perhaps are for general use, but probably lacking features that designers would be looking for. Then, sometimes designers invest huge amounts of time and effort designing and programming assembly routine codes in order to achieve the required microcontroller performance, as to take full control of their robot [4].

Designing a microcontroller for specific needs allows designers to minimize the programming complexity and enhance designers system's performance. Designers also should keep in mind that microcontroller programming is as important as the microcontroller hardware design. Although it is not the intention of this work to discuss the microcontroller programming, this work illustrates the instruction execution of the microcontroller. This helps a lot when we are trying to understand the basic concepts of assembly programming like the addressing modes, clock cycles, and operands.

The quality of the microprogramming is what makes it possible to transform the complex circuits of the microcontroller into something useful. One of the main motivations for this work will be that inexperienced designers will not only gain an

insight of microcontroller design and operation, but also, designers will get a better understanding of the microcontroller assembly programming [2].

### **1.3 Objectives**

The main objective of this dissertation is to design a RISC microcontroller using VHDL and implement it in an FPGA. The microcontroller instruction set and features are based on Atmel AVR AT90S1200 RISC microcontroller. The microcontroller must be able to fit into the targeted FPGA device, which is Xilinx Spartan II xc2s200-5pq208. Features which cannot be implemented on an FPGA (analog comparator, pull-up resistors, etc) and which are not critical to the operation of the CPU (watchdog reset) will be ignored.

### **1.4 Introduction to Atmel AVR AT90S1200**

The AT90S1200 is a low-power CMOS 8-bit microcontroller based on the AVR RISC architecture. By executing powerful instructions in a single clock cycle, the AT90S1200 achieves throughputs approaching 1 MIPS per MHz allowing the system designer to optimize power consumption versus processing speed. The AVR core combines a rich instruction set (89 powerful instructions) with the 32 general purpose working registers. All the 32 registers are directly connected to the Arithmetic Logic Unit (ALU), allowing two independent registers to be accessed in one single instruction executed in one clock cycle. The resulting architecture is more code efficient while achieving throughputs up to ten times faster than conventional CISC microcontrollers.[]

The microcontroller also comes with 1K Bytes of in-system programmable flash as the program memory and 64 bytes of in-system programmable EEPROM. The AT90S1200 is equipped with one 8-bit timer/counter with separate prescaler, one on-chip analog comparator, a watchdog timer with on-chip oscillator and SPI for in system programming. It also features the external and internal interrupt. There are a total of 15 programmable I/O lines. The IC come in 20-pin PDIP and SOIC with 2 speed grades, 0 - 4 MHz for AT90S1200-4 and 0 - 12 MHz for AT90S1200 [25].

## **1.5 Overview of the Microprocessor and Microcontrollers**

### **1.5.1 Microprocessors**

A microprocessor, as the term has come to be known, is a general purpose digital computer's central processing unit (CPU). Microprocessor CPU contains arithmetic and

logic unit (ALU), a program counter (PC), a stack pointer (SP), some working registers, a clock timing circuit, and interrupts circuits.

To make a complete microcomputer, one must add memory; usually read only program memory (ROM) and random-access data memory (RAM), memory decoders, an oscillator, and a number of input/output (I/O) devices, such as parallel and serial data ports. In addition, special-purpose devices, such as interrupt handlers and counters, may be added to relieve the CPU from time consuming counting or timing chores. Equipping the microcomputer with mass storage devices, commonly a floppy and hard disk drives, and I/O peripherals, such as a keyboard and a CRT display, yields a small computer that can be applied to a range of general-purpose software applications [1].

The hardware design of a microprocessor CPU is arranged so that a small or very large system can be configured around the CPU as the application demands. The internal CPU architecture, as well as the resultant machine level code that operates that architecture, is comprehensive but as flexible as possible.

The prime use of a microprocessor is to read data, perform extensive calculations on that data, and store those calculations in a mass storage device or display the results for human use. The programs used by the microprocessor are stored in the mass storage device and loaded into RAM as the user directs. A few microprocessor programs are stored in ROM. The ROM-based programs are primarily small fixed programs that operate peripherals and other fixed devices that are connected to the system. The design of microprocessor is driven by the desire to make it as expandable and flexible as possible, in the expectation of commercial success in the marketplace.

### **1.5.2 Microcontrollers**

Typically microcontroller is a true computer on a chip. The design incorporates all of the features found in a microprocessor CPU (ALU, PC, SP, and registers). It also has added the other features needed to make a complete computer: ROM, RAM, parallel I/O, serial I/O, timer / counters, interrupt control and a clock circuit.

Like the microprocessor, a microcontroller is a general purpose device, but one that is meant to read data, performs limited calculations on that data, and control its environment based on those calculations. The prime use of a microcontroller is to control the operation of a machine using a fixed program that is stored in ROM and that does not change over the lifetime of the system.

The design approach of the microcontroller mirrors that of the microprocessor:

make a single design that can be used in as many applications as possible in order to sell, hopefully, as many as possible. The microprocessor design accomplishes this goal by having a very flexible and extensive repertoire of multi-byte instructions. These instructions work in a hardware configuration that enables large amounts of memory and I/O to be connected to address and data bus pins on the integrated circuit package. Much of the activity in the microprocessor has to do with moving code and data to and from external memory to the CPU. The architecture features working registers that can be programmed to take part in the memory access process, and the instruction set is aimed at expediting this activity in order to improve throughput. The pins that connect the microprocessor to external memory are unique, each having a single function. Data is handled in byte, or larger, sizes [2].

The microcontroller design uses a much more limited set of single and double byte instructions that are used to move code and data from internal memory to the ALU. Many instructions are coupled with “programmable” pins on the IC package. The microcontroller is concerned with getting data from and to its own pins; architecture and instruction set are optimized to handle data in bit and byte size.

### **1.6 Comparison of Microcontrollers and Microprocessors**

The microprocessor is an integrated circuit composed by the Control Unit, Arithmetic Logic Unit, Registers and Digital circuit support. The microprocessor uses its data bus pins, address bus pins, and control lines pins to allow connection to other circuits to configure the entire system. The main characteristic of the microprocessor is that it is an open system, which means that its configuration is variable, and can be adapted to many different applications [4].

The microcontroller is a closed system. In which all parts are fixed in the same chip. Just the lines that control the peripherals are the ones that go outside the chip. This characteristic makes microcontrollers suitable for specific applications or for general use. The microcontroller applications range is narrower than the microprocessor's range. The reason is that microcontrollers have all their computing system integrated on the same chip. This reduces the available space inside the microcontroller to include components that the microprocessor has externally like memory and I/O system. This means that a microprocessor can be used for microcontroller applications but microcontrollers cannot always be used for most microprocessor applications. Microcontrollers are preferred when the application is

defined and specific. In those situations where important system modifications are needed or applications are not specialized a microprocessor is more convenient.

The contrast between a microcontroller and a microprocessor is best exemplified by the fact that most microprocessors have many operational codes (opcodes) for moving data from external memory to the CPU; microcontrollers may have one or two. Microprocessors may have one or two types of bit handling instructions; microcontrollers will have many.

Microcontrollers are found in small, minimum component designs performing control oriented activities, such as the traffic lights. 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 required is microcontroller, a small number of support components, and a control program in ROM.

To summarize, the microprocessor is concerned with rapid movement of code and data from external addresses to the chip; the microcontroller is concerned with rapid movement of data within the chip. Microcontroller can function as a computer without the addition of any external hardware; microprocessor must have many additional parts to be operational [1].

### **1.7 Microcontroller Performance Factors**

Microcontroller performance can be defined in terms of speed, size, power, cost, design time and manufacture cost. Each depends on concepts beyond the scope of this work. The main factor that determines the microcontroller performance [5] are its architecture, design features and manufacture process. Thus the microcontroller performance depends on designer's judgment at the design stage. The architecture features determine the remaining microcontroller characteristics. The architecture depends on the microcontroller application. Different applications differ in features and data processing requirements. The Von Neumann architecture and Harvard architecture [3] are the two main architectures used in microcontroller design.

The Harvard architecture is the most popular nowadays. The Von Neumann architecture main characteristic is that it uses one main memory where data and instructions are stored. Only one system bus is used for control, data transfer, processing and addressing. Harvard architecture consists of two different and independent memories in which one contains instructions and the other one contains



data. Both have their own data bus systems for control, data transfer, processing and addressing. Both memories can be accessed simultaneously.

The Architecture and the hardware implementation features transform an idea into a circuit with specific characteristics. Computer simulation allows designers to verify that circuit works as required. When specification constraints and performance requirements are met, it is time for testing and manufacture. Design aspects defined by the architecture determines which manufacture process will be used.

## **1.8 CISC vs. RISC Architecture**

### **1.8.1 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 tend 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.

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

### **1.8.2 Reduce 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.

The first characteristic of RISC is the uniform series of single cycle fetch and execute operations for each instruction implemented on the computer system being developed [9]. 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 16 bits, and the data port of the system bus (the data bus) has 16 lines, the standard instruction length should be 16 bits.

Achieving uniform 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 multi-cycle operations (multiply, divide, floating point), and may be impossible to be executed in a single cycle.

Some of the necessary conditions to achieve a streamlined operation are:

1. Standard, fixed size of the instruction, equal to the computer word length and to the width of the data bus.
2. Standard execution time of all instructions, desirably within a single CPU cycle.

Which instructions should be selected to be on the reduced instruction list? The obvious answer is: the ones used most often. It has been established in a number of earlier studies that a relatively small percentage of instructions (10 – 20%) take up about 80% – 90% of execution time in an extended selection of benchmark programs. Among the most often executed instructions were data moves, arithmetic and logic operations. As mentioned earlier, one of the reasons preventing an instruction from being able to execute in a single cycle is the possible need to access memory to fetch operands and/or store results. The conclusion is therefore obvious we should minimize as much as possible number instructions that access memory during execution stage.

This consideration brought forward the following RISC principles:

1. Memory access, during execution stage, is done by load/store instructions only.
2. All operations, except load/store, are register-to-register, within the CPU.

Most of the CISC systems are microprogrammed; because of the flexibility that microprogramming offers the designer. Different instructions usually have microroutines of different lengths. This means that each instruction will take a number of different cycles to execute. This contradicts the principle of a uniform, streamlined handling of all instructions. An exception to this rule can be made when each instruction has a one to one correspondence with a single microinstruction. That is, each micro routine consists of a single control word, and still let the designer benefit from the advantages of microprogramming. However, contemporary CAD tools allow

the designer of hardwired control units almost as easy as micro programmed ones. This enables the single cycle rule to be enforced, while reducing transistor count.

In order to facilitate the implementation of most instruction as register-to-register operations, a sufficient amount of CPU general purpose registers has to be provided. A sufficiently large register set will permit temporary storage of intermediate results, needed as operands in subsequent operations, in the CPU register file. This, in turn, will reduce the number of memory accesses by reducing the number of load/store operations in the program, speeding up its run time. A minimal number of 32 general purpose CPU registers has been adopted, by most of the industrial RISC system designers.

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 load and store instructions
- All operations done within the registers of the CPU
- Hardwired rather than micro programmed control unit
- Relatively large (at least 32) general purpose register file

### **1.9 Microcontroller Applications**

The microcontroller is one of the most important electronic devices on which modern technology is based on. Microcontroller uses are endless; from toys to TV sets, microwaves, ovens, computers, printers, cars and so on. Digital circuits become larger and larger as more functions need to be executed. In modern digital world, most of the individual digital circuit components are sold in a single chip. Those individual chips need power and space to operate. When the circuit becomes huge, the traditional logic design approach is not the best option and microcontrollers become convenient. Microcontrollers are basically sequential machines because their operation depends on their current status and its inputs. Their power lies in the fact that the hardwire configuration allows its operation to be changed depending on programming. It is not required to use additional logic circuits if the operation is changed [4].



### 2.1 Tradeoffs in Microcontroller Design

An important question that must be answered before attempting to implement a microcontroller is, **Is it necessary to use a special purpose microcontroller?**[3] In addition to having the basic instruction set, special purpose microcontrollers usually have instructions specialized to perform specific tasks. Those microcontrollers include in their design, special hardware that is used for execution and calculation support to execute instructions in their specific applications. The application determines the microcontroller operation, and the operation is executed with specific instructions. Then, the real deal in the design process consists in making tradeoffs between designing more powerful and complex instructions that reduce the programming code, or as another alternative, the operation can be implemented in hardware to save the time-consuming programming of certain tasks and achieve faster execution.

**Should an operation be implemented in hardware or software? Is it worth?** [3] Answer to those questions depends on many factors like design requirements, available budget, technology used and so on. Hardware instructions implementation result in faster executions but increase design cost. Software implemented operations save hardware and costs but increases the instruction execution time and the programming complexity. There are not defined rules. Designers have to make their choices based on design constraints and available resources to produce the best system performance at lower cost.

### 2.2 The Microcontroller Operation

The microcontroller operation consists in four steps:

1. **Fetch process;** the fetch process consists in retrieving one instruction from memory and loads it in the Instruction Register.
2. **Decoding;** once the instruction is in the Instruction Register, the control unit receives the operational code from it. The control unit decodes the operational code to identify the instruction to be executed.
3. **Executing;** after the control unit identify the instruction, it start a series of microcontroller hardware signal activations. The control unit ensures that the

- necessary elements are on and off in each clock cycle to accomplish the instruction execution.
4. **Storing results:** after the execution of the instruction results obtained must be stored at appropriate place.

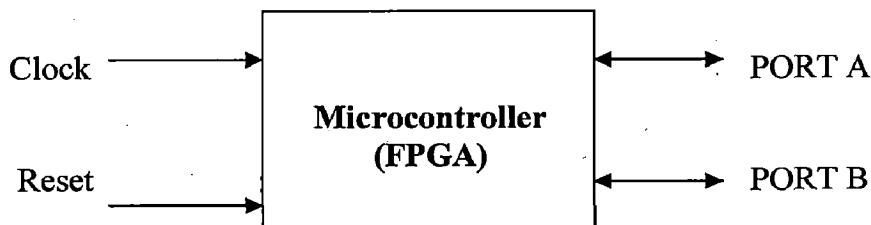


Figure 2.1: Microcontroller basic block

Basically the CPU addresses a memory location, obtains (fetches) a program instruction that is stored there, and carries out (executes) the instruction. After completing one instruction, the CPU moves on to the next one. This fetch and execute process is repeated until all of the instructions in a specific program are done. The fetch process clock cycle depend on the Instruction Register size (instruction word) and the number of bits of the data bus. For example if the IR size is 16 bits and the data path is four bits, then four clock cycles will be needed for the fetch process. The memory size will determine how many instructions can be stored in it and indeed the program size that can be stored.

The first step in the hardware design is to prepare the specification of the design. The architecture and the instruction set must be understood thoroughly. The design ideas are then described with VHDL in a text editor. Then, the VHDL code is synthesized with xilinx ISE 7.1i. If synthesized successfully, Xilinx ISE 7.1i will generate a net list files (EDF file). This file is then sent to xilinx ISE 7.1i for compilation and simulation. Results are verified by simulation.

The hardware design process is repeated until the microcontroller is complete without any errors. Hardware implementation is performed by downloading the design into the targeted FPGA device (Xilinx Spartan II xc2s200-5pq208). 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 store into the program ROM of the microcontroller before it can do the job. So, before the microcontroller is downloaded into the FPGA device, the specific program for the application must be written. This program file, together with the EDF file of the complete microcontroller is

then sent to xilinx ISE 7.1i for compilation and device programming. Once programmed into the device, the FPGA is reset to execute the application.

### 2.3 Microcontroller organization

The microcontroller has 2 input pins (reset & clock) and 2 bi-directional I/O ports. Each I/O port consists of 16 individual I/O pins, total of 32 I/O pins. The clock signal will drive the whole microcontroller directly. Reset is active high; when asserted it resets the microcontroller to the default state even if the clock is not running. Port A and Port B are all 16-bits port. Each bit can be configured to be input or output. All port pins are tri-stated when the microcontroller is reset. Pin B.7 also serves as the external interrupt source and external timer clock source.

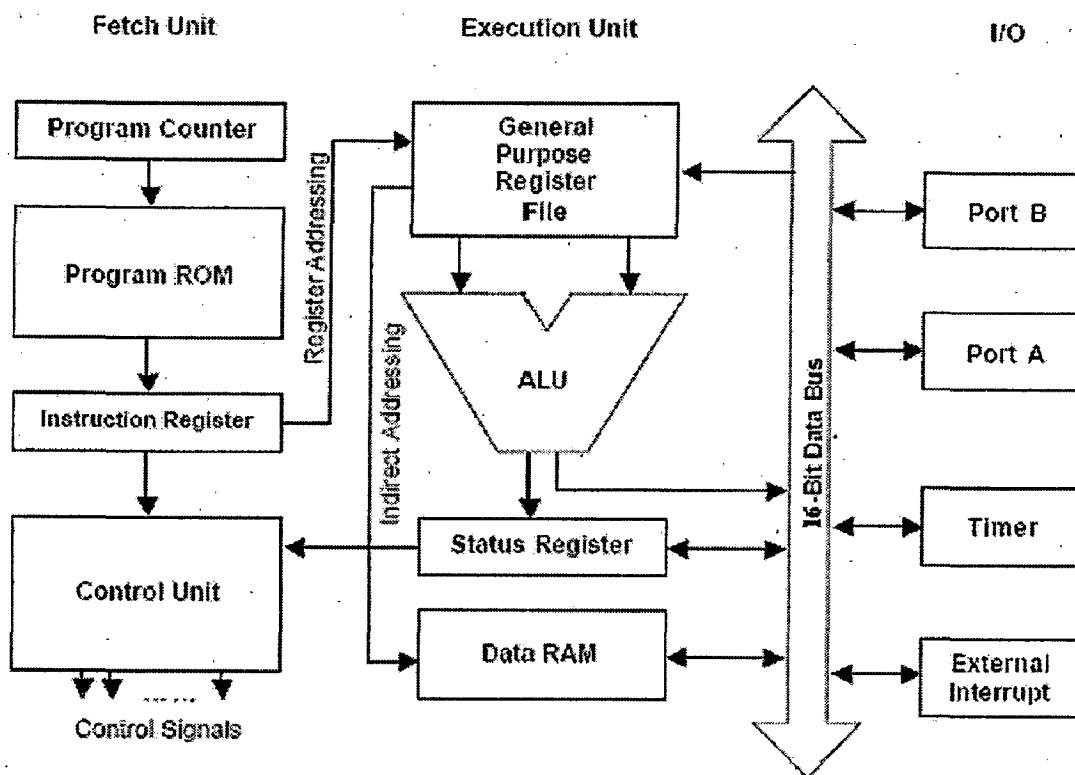


Figure 2.2: Top-level block diagram of the Microcontroller [5].

Figure 2.2 shows the top-level block diagram of the design, the bus structure has been simplified, but every block represents a module to be designed. At first glance, there are 11 modules in the top-level, with the 2 ports sharing the same module. These 11 modules are to be design separately using the top down design approach. Some modules like the instruction register and status register are easy to design, but modules like ALU and the control unit require a lot of understanding. The overall dataflow and bus structure between all the modules must be understood before designing the modules

individually. Buses provide connection between modules. There are many direct buses, such as the connection between program counter and program ROM, between program ROM and IR, between register file and ALU, etc. No control signals are required for direct buses. A common bus is a bus shared by many modules. The data bus is the only common bus in this design. The data bus provides connection between the general purpose register file, ALU, status register, SRAM and all the I/O features. Since there are so many possible data flows, control signals are required to control the correct flow direction. Only one source to the data bus is allowed at a time. If not, logic contentions will happen and the value of the data bus will be invalid. Tri-state bus is used to implement the common data bus. Only the correct source is connected to the data bus while others are in high impedance state. The impedance is so high that it can be seen as unconnected to the bus system. If the ALU is the data source, the data bus will be flooded with the result of the ALU and is available to all the connected modules. Control logic will generate an enable signal for the real destination to receive the data.

Next is a brief introduction to the whole system. The system can be divided into 5 units, the fetch unit, decode unit, execute unit, write back unit, and I/O unit. Fetch unit is in charge of fetching the next instruction, decode unit decodes instruction and generates necessary control signals, execute unit is in charge of executing the current instruction and the write back unit stores result of various operations performed by microcontroller and interrupt handling is also done. I/O unit provide a connection with the outside world. The fetch unit and execute unit form the CPU of the microcontroller.

The first module of the fetch unit is the program counter (PC). The PC contains the address of the next instruction to be executed. It points to the program ROM to locate the instruction. The instruction from the ROM is then latched into the instruction register (IR). The control unit takes the content of the IR and decodes it. It then asserts the appropriate control signals to execute the instruction. All modules are connected with direct buses.

The execute unit in charge of executing most instructions. Normally, to execute an instruction, 2 operands are output from the register file to the ALU. The ALU then perform the operation and send the result to the data bus. Contents of the data bus (result) are then stored back to the register file. The ALU also evaluate the status register flags and send them directly to the status register (SR). The whole execution process is done in a single cycle. The ALU perform many operations - include passing the contents of a general register to the data bus. SR also has a direct bus connection to



the control unit required for branch evaluation. The register file (destination and source register) is addressed directly by some bits in IR.

A RISC Microcontroller has memory access limited to only LOAD and STORE instructions. Load and store instructions can only transfer data between the RAM and the register file. A load operation sends the RAM data to the general registers through the data bus. A store operation sends the data to ALU, the ALU passes the data to the data bus and stores it into the RAM.

To implement the fetch and execute pipeline in this microcontroller, memory is implemented using the Harvard architecture. Program and data are stored in separate memories. As seen in the block diagram, program is stored in the program ROM while data are stored in the data RAM. The advantage of Harvard architecture is the ability to fetch the pre-fetched next instruction easily. A normal RAM will have an initial value of zero when powered on. In FPGA, the RAM can have initial values and thus can make it act as a ROM.

All the I/O modules contain many control registers. Data are sent to and received from it through the common data bus. The Status Register is also mapped into one of the I/O addresses. IN and OUT instructions are used to transfer data between these control registers and the general registers. The lower half of the control registers (00 h - 1F h, shaded in gray) are directly bit-accessible using the SBI and CBI (Set/Clear Bit in I/O) instructions. In this design, the lower half of control registers is all the I/O ports control registers.

## **2.4 Description of the microcontroller design Steps**

### **2.4.1 Justification for the design**

The following are some questions that could guide designers at the implementation decision stage and decide the need of a microcontroller for the application.

1. What is the application? Can the application be implemented with logic circuits? What will be the resulting circuit size? Is it affordable?
2. What could be the microcontroller implementation advantage? Microcontroller has hardwired circuits that change its operation using programming. Designers should analyze if the amount of different applications justifies the use of a microcontroller or if the use of individual operational circuits is more convenient.

3. What are the advantages or disadvantages of using a microcontroller in terms of efficiency, time, design complexity and cost? Budget and design requirements analyses are necessary to decide if a microcontroller use is convenient or not. Sometimes the use of a micro controller results in a waste of hardware resources and sometimes it is less expensive option. There are situations in which programming is avoided using logic circuit, but this choice could result in larger, expensive and more complex circuits.
4. Is a microcontroller result in the best option? How many different operations will be used? How many times one operation is executed? Is it better to use individual circuits for every operation or using a microcontroller is more efficient? Is this difference in time response needed for the application? Is the microcontroller programming complexity worth instead of using individual circuits?

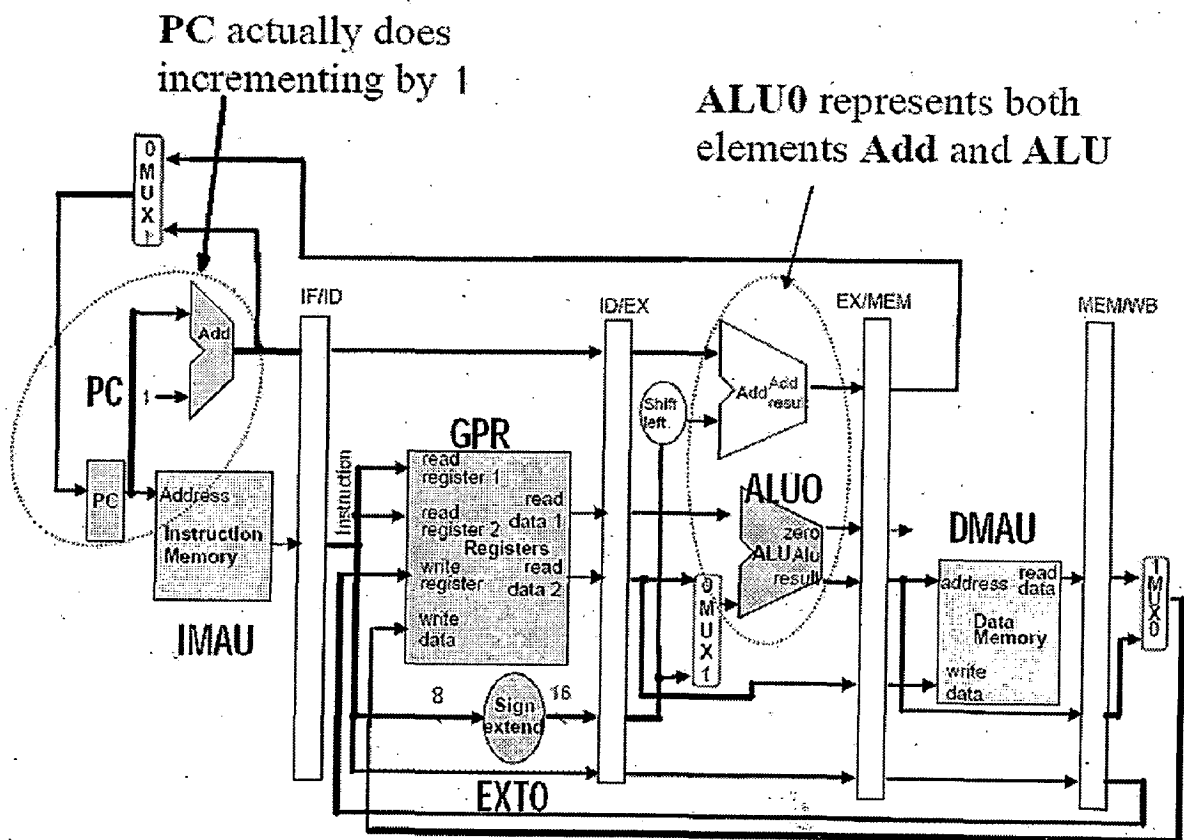


Figure 2.3: Architectural block diagram of microcontroller[9]

### 2.4.2 The Operation Definition

After a careful study of the application, the next step consists in defining the amount of different operations required for the application.

1. What are the application operation requirements? Are those operations complex or simple? How many different operations does the application have? Do designers need a new microcontroller to execute one operation or can use an existing one? Does it execute the instruction as required in terms of clock cycle, power & speed?
2. Is it more convenient to divide those operations in more simple tasks or not? Depending on the application and design requirements this could or could not be possible. Can the microcontroller with its instruction set, execute those individual and simple tasks, or a new one is needed?
3. Can those tasks be executed using more than one instruction, or is one instruction enough?

### **2.4.3 The Instruction Set Definition**

The instruction set should contain those instructions that the application requires. Tasks executed, amount of hardware used and clock cycles are very important parameters of an instruction.

- How powerful is the instruction? The term powerful means that how many tasks can be executed by single that instruction. This may result in more hardware or more clock cycles per instruction.
- How many instructions are required to perform the operation? This will be determined by the power of the instruction set. The more powerful the instruction set is, fewer instructions are needed per operation.
- What kind of instructions does every microcontroller must have? Every microcontroller must have at least; logic, arithmetic, branch and data transfer instructions.
- How many complex tasks can be executed using the simplest instruction set? The basic instruction set can be combined to execute complex tasks. E.g., multiplication operation can be executed with successive execution of the addition instruction.
- What instructions should be implemented in hardware and which ones in software and why? Instructions frequently executed must be implemented in hardware. This saves programming time and size, allowing faster instruction execution. Software instructions are used depending on the application.

#### 2.4.4 The Architecture Definition

The Computer architecture refers to the basic ideas and principles in which a computer system is based on [8]. The Architectural design steps include:

- The Instruction Set.
- The number of used bits to represent data (4, 8, 16, 32 or 64 bits).
- Instruction Format and addressing modes.
- Number of data buses.
- Instruction execution algorithm (best arrangement of hardware to process software).
- Clock cycles per instruction.
- Input / Output mechanisms.

The computer organization must be specially designed to implement a particular architectural specification. The microcontroller task is to execute each and every instruction it receives. This means that each instruction reflects the architecture in use by the microcontroller. After the selection of the desired instructions for the microcontroller, the next step consists in specifying the rest of the architecture.

##### 1. The instruction operation:

The first task must always be to specify each instruction operation. After designers identify the instruction set, they must document: the instruction's name, as well as operands and execution in symbols for each one.

##### 2. The instruction length:

The instruction length refers to the size of the group of bits processed during instruction execution. Using more than the necessary bits may result in excessive hardware use and an increase in circuit size, cost and power consumption.

##### 3. The instruction format:

The instruction format specifies the order of the instruction parameters in the instruction word. Those parameters include the operational code, registers used, and additional necessary data for the instruction execution.

Each instruction word has a group of bits that identifies its specific code. The group of bits used for this code is called the instruction operational code or opcode. This work uses 16 main instructions, so, the minimum number of bits for the opcode decoder is 4, enough to assign each instruction a specific code. There are no standard

rules for the order and meaning of the different groups of bits that compose the instruction word. That depends on designers' judgment and system architecture.

- a. Bits 15-12 stand for opcode. Those bits specifies instruction that will be executed
- b. Bits 11 to 10 represents as function code which gives function of instruction, used depending on the operation.
- c. Bits 9 to 5 labeled as Ra specify the register file address location to store the processed data or the one that has been transferred from memory.
- d. Bits 4 to 0 labeled as Rb, represent register file address location of one operand.

#### **4. The instruction format organization.**

The instruction word parameters can be organized as designers want. In this work the operational code will be at the left most side, next are the functional codes, the additional data used for the instruction execution, and finally to the right most side is the registers used during the operation.

#### **5. The Operational Code (Opcode).**

The number of instructions decides the necessary bits for the operational code. The operational code identifies each instruction with a unique code for its execution.

#### **6. Addressing modes**

The addressing modes decide the amount of registers used for data processing. The addressing modes used during the instruction execution decides if more bits have to be used to address the data or not and this affects the size of the instruction word.

#### **7. Bits used for the Register File.**

The number of registers used in the Register File determines how many address bits in the instruction word are required to address one specific location in it.

#### **8. Number of data buses.**

The number of data buses in use determines the amount of data processed per clock cycle. Using more than one data bus can save clock cycles per instructions; but increases the data path and control unit circuit complexity.

#### **9. Address Bus:**

Depending on design requirements the address bus is not necessary if the address bits can be transferred using data bus. A dual role requires additional hardware.

#### **10. I/O Handling:**

Will the I/O ports be memory mapped or handled separately? Memory mapped ports do not require special I/O instructions.

### **2.4.5 The Arithmetic Logic Unit**

In step V, the goal is to design the Arithmetic Logic Unit circuit, which is one of the most fundamental CPU components, where mathematical and logical operations are executed. Techniques used in this work for the ALU design consist in designing all its individual circuits and connecting them in parallel.

1. **ALU components:** The individual circuits that execute all the arithmetic and logical operations are joined together as one unit to compose the Arithmetic Logic Unit.
2. **Testing:** Designers must ensure that every individual circuit in the ALU correctly does every calculation; flags are added for that purpose.

All mathematical and logical calculations are executed at the same time, but only the desired calculation will be the one released to the ALU output port by means of the tristate buffer activated. Caution should be taken with significance of the input and output bits of every circuit. Mistakes can lead to miscalculations and continue through the rest of the instruction execution.

### **2.4.6 The Register File**

A register is a small high-speed memory circuit that holds binary data. Register File is a group of registers used to store data during the instruction execution. In this step, the Register File is developed. The Register File stores data retrieved from Register, memory or input port resulting from various operations in ALU. All temporary data used by the microcontroller to perform its operations is also stored in the register file. Register File design consists of three stages: register selection stage, input stage and output stage.

#### **1. Implementation alternatives**

The number of data buses in the microcontroller determines the Register File design. Sometimes more than one data bus is used to accept and release the data simultaneously in one clock cycle. Designers must decide how many data buses will be used in the microcontroller because Register File will use the same number.

#### **2. The number of registers for the application**

The number of registers is an important design parameter because it affects not only The Register File size but also the Instruction Register size because the IR has bits dedicated for the Register File address. Designers must select the number of necessary registers to hold data in each instruction clock cycle.

### **2.4.6.2 The Register File Selection Stage**

The instruction word identifies two parameters: Ra and Rb. Each of this parameter, when referring to registers, is actually addresses that identify a register from the register file. Since Ra and Rb have five bits, the register file has 32 registers. One register is selected by means of a decoder 5x 32. The control unit will activate signals to indicate which register is assigned to Ra or Rb of the instruction word.

### **2.4.6.4 Register Transfer**

The whole design contains many registers, special purpose registers (IR, PC, MAR, MBR), general purpose registers (R0, R1...R31, I/O control registers) etc. The whole system works by transferring data between these registers (register transfer). Some data are transferred without modification while some are manipulated before transfer to the next register. If the data are to be manipulated, they are manipulated by the combinational logic between these registers. How these data are transferred, how are they being manipulated before transfer, and what does different data inside the register means, will determine whether the design can work as microcontroller. The design will perform a long series of register transfer to form the functioning of a microcontroller. Registers are transferred to another through many levels of the combinational logic.

A read of the status register will bring the contents of the status register to one of the general register directly without manipulation. The combinational logic Perform AND operation between two general registers, will pass the two registers through a combinational logic (the logic unit) before writing back to one of the register. Memory (program ROM and data RAM) are treated as a kind of combinational logic. PC is passed through the program ROM to the instruction register. The instruction register will receive instruction in from program ROM pointed by PC.

So, the design process is to design all the registers along with combinational logic and the interconnection between them. This is called the data path of the system. Control signals are then used to determine how the register transfer takes place. Control signals are asserted by the control unit. The data path, along with the control unit forms complete microcontroller. It is important to know what registers exists in the system.

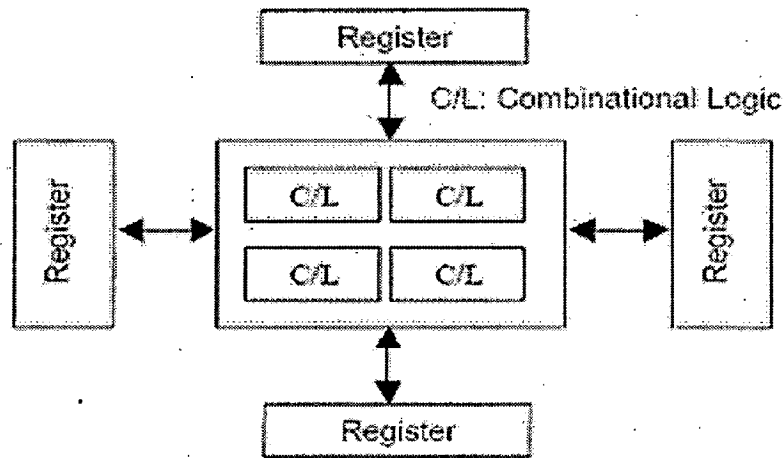


Figure 2.4: Implementation of register transfer

### 2.4.7 The Instruction Register (IR)

The Instruction Register holds the instruction word that will be executed. The IR is connected to the control unit, the Register File & data path.

#### 1. Implementation alternative:

The IR implementation consists of a register or a group of registers that holds the instruction word.

#### 2. Size:

It will be easier if the size is equal to the word size because then, the instruction word holds all the required information for the instruction execution.

The memory output is connected to the IR to load every single program instruction line. The IR does not have to be the same size of the data bus because it just transfers data and does not contain any other information about the instruction.

### 2.4.8 Data Path

The microcontroller data path is the configuration of all the circuits used for data processing. Some key points are very important in this step.

1. **Layout:** Designers must be creative and use strategic thinking to make the best circuit arrangement in order to achieve the instruction execution using the minimum amount of hardware and clock cycles.
2. **Clock cycles:** More data can be processed at the same time depending on the amount of Register File input and output ports. Another important element is the number of additional registers in data path used to hold data between clock cycles. This can make a difference in number of CPI if designers know how to use them.



### **2.4.8.2 Basic Data Path**

In order to make useful all the elements it is necessary to provide a path for communication between them to transfer data from one to another. This data path can perform the basic instructions and will be used as the basis to develop more complex instructions. As more complex instructions are added, this data path undergoes an evolution into a more complex one, adding more hardware in parallel.

To test the feasibility of basic instructions this data path can process data provided by switches. Switches can be used to store values in the Register File. Address lines of Ra and Rb are connected from the IR to Register File to access the data. Register File output port is connected to the ALU input ports to perform logic and mathematical operations. The ALU output port is connected to the Register File input port to store results.

### **2.4.8.3 Data Path with Immediate Operations**

At this point, when adding new hardware to implement new instructions, there are some details that should be taken care of, in particular:

- 1) Control signals, IR related logic and Connection to buses and other blocks
- 2) Overall issues such Signal conflict and Delays.

The data path is next modified to include other ALU operations, like, immediate addressing mode operands. The immediate values are put in bits 0 – 7 of the instruction register. The data path modification consists in making a connection between those immediate values in the IR and the ALU. But the connection cannot be done directly because the values in the Register File can cause conflict with those in the data path. To solve this problem a tristate buffer is used to isolate the data in the Register File from those in the data bus. New parts added in the data path are identified with lines.

### **2.4.9 The PC, Jump and data transfer instructions**

The instructions developed at this stage use the existing data path hardware and additional necessary circuits added in it for instruction execution.

1. Those instructions need additional circuit support because some of them make instructions are executed. It is very important to test those circuits before using them for support. Another reason for using additional hardware is that more than one task per clock cycle is executed in those instructions.

2. Draw the block diagram to show the added elements: It is convenient to show added elements to the data path to see its transformation into a more complex one.
3. Program counter: The program counter is developed at this step. This step presents the PC implementation and interconnection in the microcontroller circuit.

#### **2.4.10. The Control Unit**

The control unit is the CPU section that decodes program instructions and controls their execution. It takes control of every signal in the microcontroller, activating or deactivating those signals in each clock cycle. The signal activation and deactivation per clock cycle make possible the flow of data through all data path circuits. The circuit arrangement determines the amount of processed data in each clock cycle. Then, as more data is processed per clock cycle fewer of them are needed. The developing method used in this work requires that designers “run” by hand every single instruction and take notes of which circuit signals are activated and deactivated per clock cycle.

##### **1. Operational Code Decoder:**

This element receives one specific instruction code and release one signal that indicates the microcontroller to execute it.

##### **2. Control Unit Encoder:**

This element receives input signals from opcode decoder and from timer. The Control Unit Encoder activates the corresponding circuit signals that have to be active in the specified instruction in every clock cycle.

##### **3. Implementation Alternatives:**

The preceding explanation of the control unit operation is implemented using logic circuits for the control unit encoder and the opcode decoder. The control unit implementing this approach uses the opcode to identify the instruction location in ROM. Each line code in ROM represents each instruction clock cycle and the code in every line just controls (activates or deactivates) all the data path circuit signals.

#### **4. Control Signals Characteristics**

If the control signals are used to control the data path then the characteristics of the control signals must be understood before one can proceed further. First, a control signal will have at least a length of one clock cycle. It usually asserted at falling clock transition and deactivated a short delay after another falling clock. The data path consists of many registers and combinational logic between them, so there are basically 2 kinds of control signals. The first kind controls the combinational logic and the second kind controls the registers.

When a combinational logic encounters a control signal, it will act towards the signal immediately. The delay to get the valid result is the delay for the input to propagate through the combinational logic. The combinational logic can be functional unit such as adder and shifter, steering logic such as multiplexers and decoders or memory (program ROM and data RAM).

A register control signal requires a falling clock to operate. WR\_REG signal will only latch the data into the destination register of the register file when it encounters the falling clock. the operations is actually happened at the end of the control signal where it meet the falling clock. These kinds of control signals are the enable signals for the registers, or the increment/decrement signal for a counter.



to many embedded control applications. The AT90S1200 AVR is supported with a full suite of program and system development tools including: macro assemblers, program debugger/simulators, in-circuit emulators, and evaluation kits.

The Atmel RISC microcontroller is a rather simple machine with quite a bit of flexibility. The 4 I/O ports are perfect for communicating with many external applications. The rich interrupt structure aides well in a control environment. Fig. 3.1 is an architecture block diagram of the Atmel RISC microcontroller. More specifically, register organization is more pronounced and the bus structure is well defined. Atmel RISC microcontroller design is based on Harvard architecture; the data space and the program space are separated. One feature that is noteworthy is the memory-mapped ports. All of the external ports are memory mapped, which simplifies programming control projects. These external data port addresses reside in the memory where registers are given term Special Function Register (SFR). Careful study of architecture gives rise to several SFRs.

Although the special function registers seem to exist as individual registers within the architecture, they are part of the Atmel RISC microcontroller internal RAM structure. The internal RAM of the Atmel RISC microcontroller has four distinct spaces. The upper 128 bytes of RAM contain the SFRs. The lower half of the RAM is further divided into three segments, scratch pad, bit addressing segment, and the register bank area

Though the SFRs as being within the structure of the RAM, it is more likely and is hypothesized that most of the SFRs are physically located outside of the RAM space. Not having these registers in the RAM space would allow for easier placement and access within the datapath. For example, the SFRs for ports P0, P1, P2, and P3 are probably close to the peripheral of the chip, yet they are accessed as though they are physically in the RAM. This could not be verified since Intel keeps these secrets to themselves. Within the lower portion is a scratch pad area for general use.

A bit addressable segment has been included for control applications. The lowest portion of the RAM block contains a bank of registers. The register banks can be accessed by direct or register addressing. The Program Status Word (PSW), which is a SFR located at D0h, contains two bits that shift a pointer to specify with bank to use. Otherwise a programmer may directly request or write data to this space.

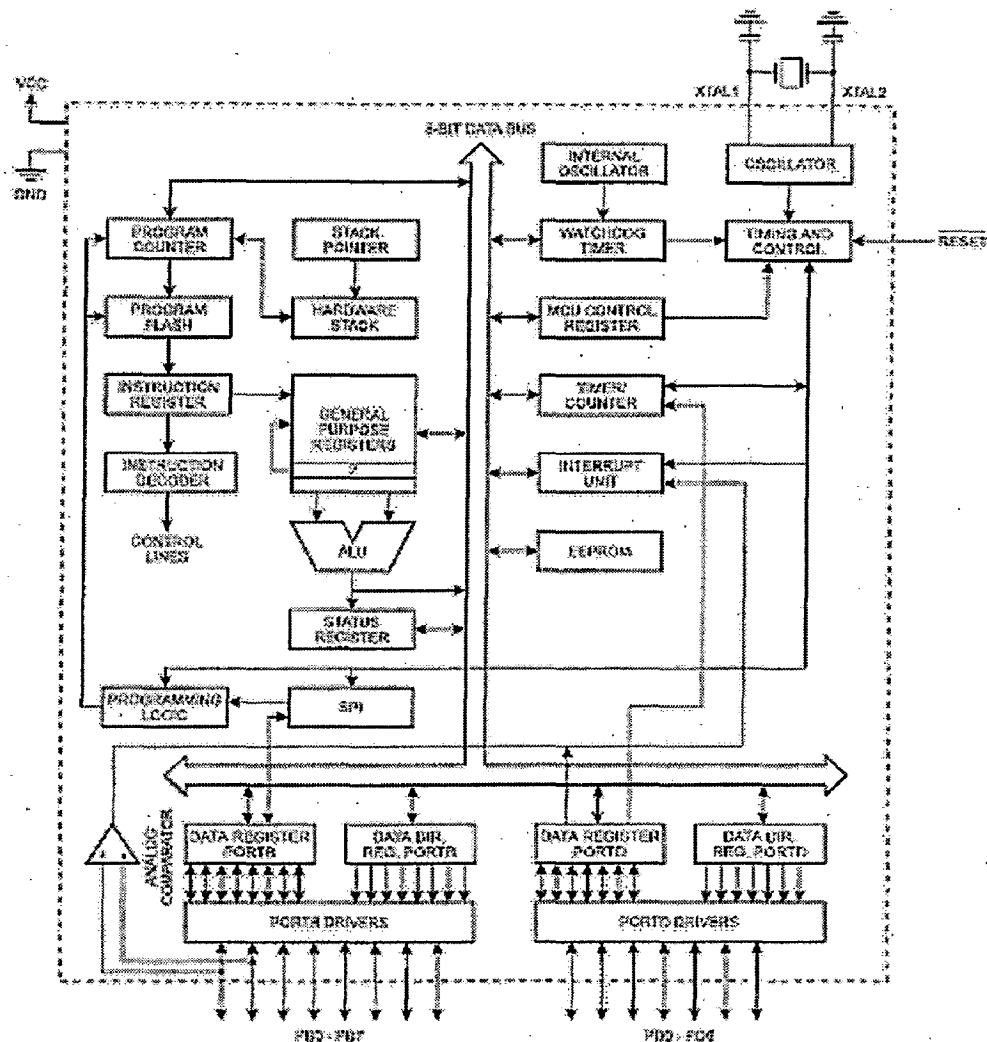


Figure 3.1: Architecture block diagram of the Atmel RISC microcontroller

### 3.2 Atmel RISC microcontroller Instruction Set and Addressing Modes

When one writes a program for a microcontroller, close attention is paid to what kind of addressing modes are available to the programmer. The Atmel RISC microcontroller program space is interfaced by four addressing modes: immediate, register, direct, and indirect.[25]

In direct addressing the operand is specified by an 8-bit address field in the instruction. Only internal data RAM and SFRs can be directly addressed. With indirect addressing both internal and external RAM can be indirectly addressed. The address register for an 16-bit address can be any register in register bank, or the stack pointer. When using register addressing, the register bank, contain registers R0 through R19. These registers can be accessed by certain instructions that include a 5-bit register specification within the opcode.

The instruction set available to the programmer of the Atmel RISC microcontroller contains 255 instructions. Dividing these instructions up into main groups, there are 4 major classifications. These groups are: arithmetic and logical operations, data transfers, program branching, and Boolean variable manipulations. The details of each instruction will not be discussed. Appendix A has been included to provide some detail about each instruction. The appendix includes the hex code, mnemonic, number of cycles, and byte count for each instruction. For this thesis the main thrust of the modeling effort was on the register addressing mode instructions. For the Behavioral model only certain basic instructions were modeled. Basically over half of the instruction set was modeled for the Instruction Set model. Certain instructions will be discussed in detail as needed.

### **3.3 Machine Cycle Sequence**

More than just study of addressing modes, instructions and general block diagrams is needed when modeling a microcontroller. If a model is to emulate its predecessor, the timing of data transactions must be accurate. The "clock" input is the base for the figure. The most critical bit of information is when Port 0 and Port 2 are read and updated. In the Intel literature, the engineers have divided a machine cycle into six major states, S1-S6. The port interaction information detailed here is essential for timing accurate models.

### **3.4 Modeled Architecture**

For the purpose of this dissertation, certain parts of Atmel RISC microcontroller architecture were modeled and others were not. Since the end result of this work is an Instruction Set model that tests the Register addressing mode using only one register bank, the architecture to support this modeling will be discussed. The Atmel RISC microcontroller instruction set contains 255 instructions. The Instruction Set VHDL model discussed can support 74 instructions. 61 of 74 instructions are register addressing mode instructions. The five remaining instructions were modeled to allow for simulation. The functions and registers listed account for 35% of the complete architecture. Items like the counter/timers and the UART were also modeled. The Instruction Set VHDL model was written such that more instructions could easily be added. The remaining functional blocks would also integrate easily.

### 3.5 Instruction format

I have implemented a total of 81 instructions; each instruction is having its particular format. There are three basic types of instructions supported by this processor. These are the Register Type, Branch Type, and the Immediate Type. The specification for each type of instructions is given below.[8]

#### 3.5.1 Register type instruction format

In this format bits 15-12 represent the opcode. Bits 11-10 represent the function code that represents the ALU function that is to be performed. A bit 9-5 represents the address of the first source register, which is also the address of the destination register. Last Bits 4-0 give the address of the second source register. If the opcode bits, 15-12, do not indicate an ALU function, then the function bits are ignored. Instruction format for this type is shown below.

Register type instructions include MOVE, ADD, ADC, SUB, SBB, AND, OR, XOR, CMP, and SHIFT.

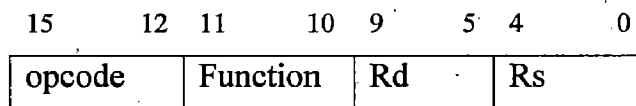


Figure 3.1: Register Type Instruction

#### 3.5.2 Immediate data type instruction format

As with the Register Type instruction, bits 15-12 represent the opcode, and bits 11-8 represent the destination source register which is also the address of the destination register. Bits 7-0 of this instruction type represent an 8-bit immediate value given in 2's complement form. When the opcode represents a unary operation, the value in this immediate field is used as the operand (instead of the value in Rd).

Immediate type instructions include MVI, ADI, SBI, ANI, ORI, XRI, and CPI. Instruction format for this type is shown below.

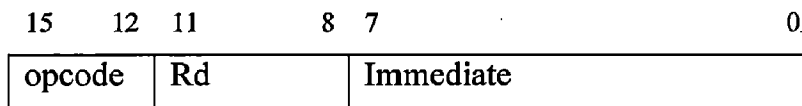


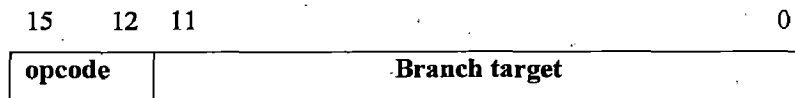
Figure 3.2: Immediate Type Instruction



### 3.5.3 Branch type instruction format

Bits 15-12 of this instruction format represent the type of branch operation to be performed. The remaining 12 bits, 11-0, represent the branch offset in 2's complement format. This number is added to the value of the PC to obtain the branch target address.

Instruction format for this type is shown below.



**Figure 3.3: Branch Type Instruction**

Branch type instructions include SJMP, LJMP, JCC, INT, IRET, CALL, and RET.

Table 3.1 below summarizes all the instructions supported by the microcontroller. A more detailed table of the instruction set along with the description for each instruction can be found in Appendix B.

As mention earlier, RISC instructions have a fix length and are easily decoded. For this microcontroller, all instructions have a fixed length of 16-bits. The instruction format is simple in order to be decoded easily.

### 3.6 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. Since the instruction set defines the data path and everything else in a processor, it is necessary to study it first.

There are 81 instructions grouped into four categories: arithmetic and logic instructions, branch instructions, data transfer instructions and the bit and bit-test instructions. As mentioned earlier, instruction set of the design is based on Atmel AVR AT90S1200 instruction set. In this way, the design can use the same assembler and simulator provided by Atmel since the final design is actually an AT90S1200 compatible microcontroller.

Table 1 INSTRUCTION SET OF RISC MICROCONTROLLER

Instruction	Description	Arguments
ADD	Addition of content of two registers	Register-Register
ADC	Addition with carry	Register-Register
ADI	Addition (immediate)	Register-Immediate value
INC	Increment content of register	Register
SUB	Subtraction of content of two registers	Register-Register
SBB	Subtraction with borrow	Register-Register
SBI	Subtraction (immediate)	Register-Immediate value
DEC	Decrement content of register	Register
CMP	Compare	Register-Register
AND	AND	Register-Register
ANI	AND (immediate)	Register-Immediate value
OR	OR	Register-Register
ORI	OR (immediate)	Register-Immediate value
NOT	NOT	Register
XOR	XOR	Register-Register
SLL	Logical shift left	Register
SRL	Logical shift right	Register
SHL	Arithmetic shift left	Register
SHR	Arithmetic shift right	Register
ROL	Rotate left	Register
ROR	Rotate right	Register
RLC	Rotate left through carry	Register
RRC	Rotate right through carry	Register
LOAD	Load word	Register-Register
STORE	Store word	Register-Register
MOVE	Move data between registers	Register-Register
MVI	Move data (immediate)	Register-Immediate value
SETB	Set bit in register	Register
CLRB	Clear bit in register	Register
SWAP	Exchange bytes in register	Register- Register
IN	Get input data from I/O port	Register- I/O
OUT	Output data to I/O port	Register- I/O
JMP	Unconditional Jump	Branch
JCC	Conditional Jump	Branch
CALL	Call the subroutine	Branch
RET	Return from the subroutine	Branch
INT	Interrupt CPU to service routine	Branch
HALT	Stop instruction execution	N/A
NOP	No operation	N/A

One of the RISC characteristics mentioned earlier is single-cycle execution for most instructions. Most instructions are single cycle except branch instructions, the LOAD/ STORE instructions. Of course, some of the instructions will have different characteristics as the original AT90S1200 instructions. They are:

1. Unconditional branch instructions now take 2 cycles.
2. Conditional branch instructions take 1 cycle if the branch is not taken and 3 cycles if the branch is taken.
3. WDR (watch-dog reset) instruction is not available since the watch-dog timer features is not included in the designed
4. SLEEP will not enter any sleep modes (there are no sleep modes in the design), it will however stop the processor and wait for an interrupt (acts as a HALT). If an interrupt occurs, the processor will 'wake up', execute the interrupt routine and resumes execution from the instruction following HALT.
5. Data RAM is included in the design although AT90S1200 does not contain any data RAM. So 4 instructions are added, which are load and store instructions with post-increment and pre-decrement.
6. General purpose registers and I/O control registers are not mapped into the data addressing space for LOAD and STORE instructions.

### **3.7 Addressing Modes**

There are 45 addressing modes in the microcontroller. Rd and Rr are devoted to the destination register and source register.

1. **Register Direct Addressing:** The operand is in Rd.
2. **Immediate addressing:** Immediate data is given in the instruction itself.
3. **I/O Direct Addressing:** First operand is one of the I/O registers. The address is contained in the instruction word. The second operand is either Rd or Rr. Used by IN and OUT instructions to read from or write to the I/O registers.
4. **Direct Memory Addressing:** Operand address is specified in the instruction itself. Used when accessing the SRAM with LOAD and STORE instructions.

### **3.8 Brainstorming the Design, the Creative Process**

A challenge for this dissertation was that there was no book to aid in breaking down the instruction set of the Atmel RISC microcontroller. The Atmel RISC microcontroller, as mentioned before, is an 8-bit microcontroller. This implies that the instruction set could have a total of 256 instructions and indeed the Atmel RISC microcontroller instruction set does have 256 instructions including one NOP, i.e. no operation instruction. The Atmel RISC microcontroller instruction set seems to have a natural pattern to how the instruction set was structured. A most crucial aspect of this

work was to determine how to decompose the instruction set and use the discovered pattern in the most efficient manner. When writing a Behavioral model, one does not want to create a 256 entry case structure. This is a little cumbersome and would not translate well into a RTL VHDL model. A RTL model would be used to synthesize the design to a specific technology.

Essentially the instruction set code had to be “cracked” as how to break the instruction set up in a logical manner which would facilitate coding the instruction set into the Behavioral and Instruction Set VHDL model. Please refer to appendix A for an “as is” version of the Atmel RISC microcontroller instruction set. This view will give the reader an idea of the decomposition challenge. It was finally decided that the instructions should be broken in the middle.

The four bit grouping consisting of the most significant bit (MSB) down to the (MSB – 3) bit of the 16 bit instruction, commonly referred to as the upper nibble, would identify what type of instruction was being decoded. The four bit grouping consisting of the next 4 bits of the 16 bit instruction would indicate what type of addressing would be needed to complete that instruction.

After the instruction set was studied, the addressing modes were analyzed. As mentioned in chapter 3, register addressing, one of four modes, would end up being the primary focus of this thesis. For behavioral modeling, six instructions were selected.

Next, as discussed in [1], it is suggested that the design process continue with the construction of a register transfer table. The next table illustrates the transaction needed to execute the above selected instructions.

Table 2 maps out the necessary register transfers to complete each instruction. In column one the different addressing modes are noted for each instruction. Column two identifies the instruction being described. The next 4 columns represent needed transactions. First register transfer necessary for any of the instructions is a fetch from the memory. The next transfer is a read from Register file/memory that contains the data that will be required for instruction execution. The Temp1 register assists by holding the data read. It was decided that the actual addition described with the register transfer notation in cycle three could be stored during the next cycle. The transfer explanation would follow a similar pattern for the other instructions.

instruction	Addressing mode	1 <sup>st</sup> cycle	2 <sup>nd</sup> cycle (decode)	3 <sup>rd</sup> cycle (execute)	4 <sup>th</sup> cycle (write back)
MVIL REG, DATA	immediate	fetch	TEMP1←SXT(DATA,16)	Result←TEMP1	REG←Result
ADI REG, DATA	immediate	fetch	TEMP1←(REG) TEMP2←SXT(DATA,16)	Result ← TEMP1+TEMP2	REG←Result
STOR MEM, REG	Memory direct	fetch	TEMP1← (REG)	Result←TEMP1	MEM←Result
LOAD REG, MEM	Memory direct	fetch	TEMP1← (MEM)	Result←TEMP1	REG←Result
MOVE REG1,REG2	Register direct	fetch	TEMP1← (REG)	Result←TEMP1	REG1←Result
ADD REG1, REG2	Register direct	fetch	TEMP1← (REG1) TEMP2← (REG2)	Result ← TEMP1+TEMP2	REG2←Result

**Table 2 Register transfer (RTL) table**

Now that states have been identified for each instruction, a state diagram can be constructed covering all the instructions. The state diagram in Fig. 5.3 was used to construct the Behavioral VHDL model. These states were added to allow for configuration of SFRs.

Following state diagram creation, a Behavioral VHDL model was developed based on following completed tasks: instruction review, addressing mode review, register transfer diagramming, & state diagramming. Behavioral VHDL model is primarily comprised of a control process with a case statement that emulates the state diagram. An arithmetic function, used to emulate the ALU of the Atmel RISC microcontroller, is called during the fetch state of the control process. The program memory was not modeled based on true architecture. The Atmel RISC microcontroller is a ROM-less microcontroller that normally communicates through ports 0 and 2 to retrieve instructions. For the Behavioral model program memory is simply designed into the same abstract structure as the process that models the states of the controller. This simplifies the model allowing for attention to be paid to the instruction decoding and execution.

The organization of the Instruction Set model is similar to that of the Behavioral model. Both models have a control process that controls the state of the model. With the Behavioral model, a single process handles instruction decoding and state control. In the Instruction Set model, these functions are separated.

The main process in the Instruction Set model is a lengthy simple looping state machine that steps through 4 cycles for every machine cycle required for instruction execution. The state machine cycle process monitors the status of the Ports, initiates control signals at the proper time according to the decoded instruction, and updates the Program Counter (PC). See appendix D for a listing of example code.

Supporting the main state machine process is a process that decodes the instructions as they are retrieved from Port 0. When the state machine enters state "ID", decoding of the incoming instruction is initiated. The IR used during decoding is "split" in to two pieces. OPN\_CODE for operation alludes to the upper nibble of the 8-bit register IR and FUN\_CODE alludes to the lower nibble of IR. A two level case statement structure is used for decoding process. The first case statement looks at the 4-bit nibble FUN\_CODE to determine what addressing mode the incoming instruction is to use. Once the addressing mode is determined, additional case statement structures within the case statement for each addressing mode type analyzes the 4-bit nibble OPN\_CODE. Once the type instruction is determined, control signals are generated according to the specific instruction.



Microcontroller architecture, designed in this dissertation work, consists of a four stage pipeline. The stages are Instruction Fetch, Instruction Decode, Execute, and Write Back. Along with four stage pipeline, Data Forward and Hazard Detection unit is designed to maintain proper data flow through the pipeline stages in case of any possible pipeline conflict. Each of the stages of the pipeline along with the data forward and Hazard Detection unit are described in detail as below.

### 4.1 Instruction fetch

Instruction fetch stage consists of the following units

1. Program Counter,
2. Program Memory, and
3. Branch Decide Unit

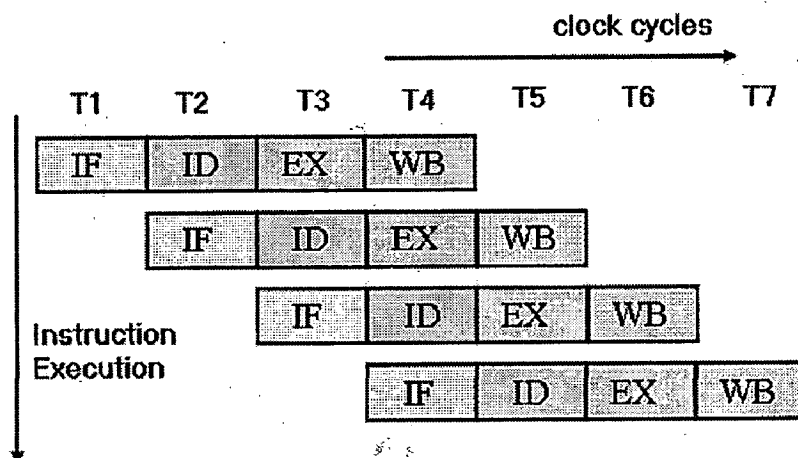


Figure 4.1: Processing in a four stage Pipelined Architecture

#### 4.1.1 Program Counter

The Program Counter contains the address of the instruction that will be fetched from the Program Memory (ROM) during the next clock cycle. Normally the PC is incremented by one during each clock cycle unless a branch instruction is executed. When a branch instruction is encountered, the new PC is calculated in execute stage is loaded directly in Program counter (Branch adder increments or decrements PC by the amount indicated by the branch offset). The PC\_write input of the Program Counter



register serves as an enable signal (output of ALU). When PC\_write signal is high, the contents of the PC are incremented during the next clock cycle, and when it is low, the contents of the PC remain unchanged.

#### **4.1.2 Program Memory**

The Program Memory contains the instructions that are executed by the processor. The input to this unit is a 16-bit address from the Program Counter and the output is a 16-bit instruction word. This module supports up to 2K words of memory.

#### **4.1.3 Branch Decide Unit**

The Branch Decide Unit is responsible for determining whether a branch is to take place or not based on the Branch signal from the Arithmetic Logic Unit (ALU). The output of this unit is a 1-bit value which is high when a branch is to take place, and otherwise it is low. This output controls a multiplexer which in turn controls whether the PC gets incremented by one or by the amount indicated by the branch offset.

### **4.2 Instruction decode Unit**

Instruction decode stage consists of the following units

1. Control Unit,
2. Register File, and
3. Sign Extension Unit.

#### **4.2.1 CONTROL UNIT**

The control unit generates all the control signals needed to control the coordination among all the components of the processor. The input to this unit is the 4-bit opcode field of the instruction word. This unit generates signals that control all the read and write operations of the Register File, and the Data Memory.

#### **4.2.2 Register file organization**

Register file is implemented using special FSM to take care of two simultaneous read and one write operation. This is a single port register file which can perform two simultaneous read and one write operation. It contains 32 16-bit general purpose registers. The registers are named R0 through R31. When the Reg\_write signal is high,

a write operation is performed to the register indicated by the write address, otherwise the value contained in the registers indicated by the read addresses are outputted.

### **4.2.3 Sign extension unit**

The input to this unit is an 8-bit immediate value provided by all the immediate type instructions. This unit sign extends the 8-bit value to a 16-bit value signed value.

## **4.3 Execute**

Execute stage consists of the following units

1. Branch Adder,
2. Arithmetic and Logic Unit(ALU), and
3. ALU Control Unit.

### **4.3.1 Branch adder**

The branch adder adds the 12-bit signed branch offset with the current value of the PC to calculate the branch target. The 12-bit offset is provided by the branch instruction. The output of this unit goes to the PC control multiplexer which updates the PC with this value only when a branch is to be taken.

### **4.3.2 Arithmetic and logic unit (ALU)**

The ALU is responsible for all arithmetic and logic operations that take place within the processor. These operations can have one operand or two, with these values coming from either the register file or from the immediate value from the instruction directly. The low power design of the ALU involves the gating the input signals to each of the separate components of the ALU. These inputs are gated using transmission gates. When a particular component of the ALU is not being used, the input to that component will be in a High Z state due to the output of the transmission gate. The operations supported by the ALU include add, subtract, compare, and, or, not, xor, logical shift, and arithmetic shift. The output of the ALU goes either to the data memory (in the case where the output is an address) or through a multiplexer back to the register file.

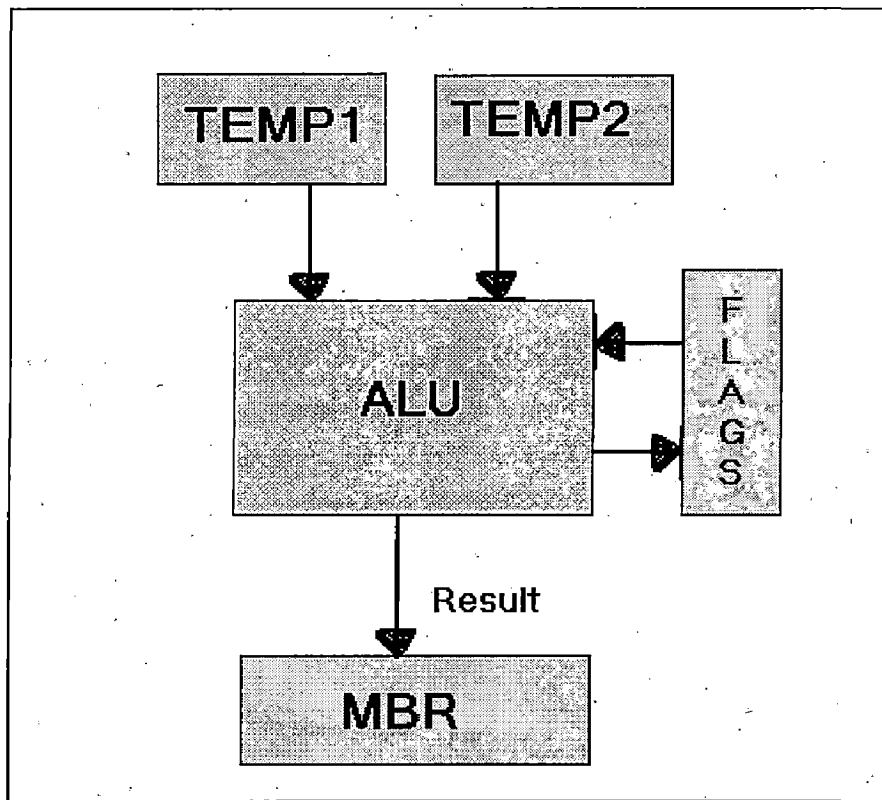


Figure 4.2: Block Diagram of ALU

#### 4.3.4 ALU control unit

This unit is responsible for providing signals to the ALU that indicates the operation that the ALU will perform. The input to this unit is the 4-bit opcode and the 4-bit function field of the instruction word. It uses these bits to decide the correct ALU operation for the current instruction cycle. This unit also provides another set of output that is used to gate the signals to the parts of the ALU that it will not be using for the current operation.

### 4.4 Write Back Unit

#### 4.4.1 Data memory

The Load and Store instructions are used to access Data memory module. This module supports up to 32 data words. When new data is to be written to the memory, the 'Mem\_write' signal is asserted. When the 'Mem\_write' signal is low, a read operation is performed for the given memory location.

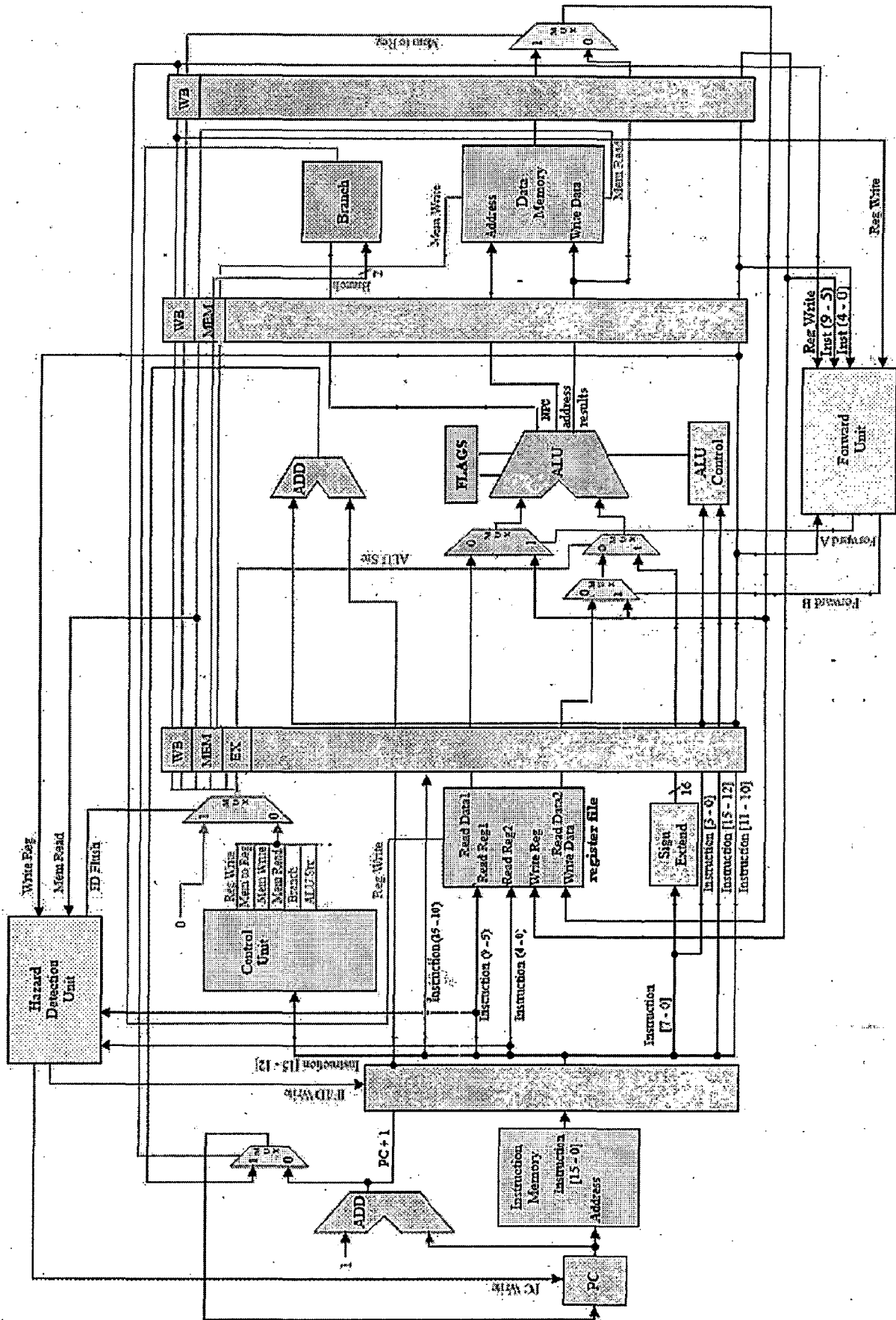


FIGURE 4.3 PIPELINED RISC ARCHITECTURE



#### **4.4.2 Register Write unit**

Results generated in execute unit should be transferred to register file in next clock cycle. This stage consists of some control circuitry that forwards the appropriate data, generated by the ALU or read from the Data Memory, to the register files to be written into the designated register.

### **4.5 Hazards in the Pipeline unit**

#### **4.5.1 Pipeline hazards:[9]**

There are situations which prevents next instruction in pipeline from executing during its designated clock cycle. Elimination of hazards often requires that some instructions in pipeline are allowed to proceed while others are delayed.

##### **1. Structural hazards:**

These hazards result from resource conflicts where hardware cannot support all possible combinations of instructions in pipeline. This occurs when some functional unit is not fully utilized in pipeline, then sequence of instruction through that unit could not proceed at rate of one pre clock cycle.

Some times structural hazards are introduced to reduce cost and latency of unit in pipeline, shorter latency comes from lack of pipeline register that introduced overhead

Solution to these hazards is stall pipeline till unit causing hazard does its work

##### **2. Data hazards**

The Data hazards arise when operands of instruction in decode stage depends on result of previous instruction which is actually being evaluated in Execute stage and results are not available as yet. Also when operands of instruction in decode stage depends on result of previous instruction whose results are in Write back stage but not yet stored in register file.

Solution to the Data hazard is data forwarding, ALU results in the execute stage is fed back to ALU. If the forwarding hardware detects that previous ALU operation has written register/memory corresponding to source for the current ALU operation, control logic selects the forwarded results as ALU inputs rather than the value supplied by instruction decode unit.

### **3. Control hazards:**

Control hazards occur when branching of program execution is required (branch instruction causes change of the PC). If instruction takes a branch then PC is changed at the end of execute cycle, this requires stall the pipeline as soon as branch is detected until instruction at new PC location in program memory reach execute stage.

Solution for Control hazards is flushing the pipeline stages and restarting program execution with new PC, 2 NOP instructions are inserted to prevent any faulty operation.

All the hazards are checked during the instruction decoding and proper action is taken by generating hazard signals to inform the execute stage, forwarding of necessary data is done in the execute stage

#### **4.5.2 Hazard detection unit**

This unit detects conditions under which data forwarding is not possible and stalls the pipeline for one or two clock cycles in order to make sure that instructions are executed with the correct data set. When it detects that a stall is necessary, it disables any write operation in the instruction decode pipeline registers, stops the PC from incrementing, and clears all the control signals generated by the control unit. By taking these steps it can delay the execution of any instruction by one clock cycle. It can do this as many times as necessary to ensure proper execution of instructions.

#### **4.5.3 Data Forward unit**

Forward unit is responsible for maintaining proper data flow to the ALU. The primary function of this unit is to compare the destination register address, of the data waiting in the Memory, and Write Back pipeline registers, to be written back to the register file, with the current data needed by the ALU, and forward the most up-to-date data to these units. By forwarding the data at the appropriate time, this unit makes sure that the pipeline works smoothly and does not stall as a result of data dependencies.

## Chapter 5: CONTROL UNIT DESIGN

---

### 5.1 Overview

We have touched the instruction set, pipeline processing and many control signals, which controls the datapath. The control unit plays the role on decoding the instruction, implements the pipeline processing and asserts the control signals for the datapath at the correct timing. This chapter covers the decoding of the instruction and the design of the finite state machine (FSM).

### 5.2 Instruction Decoder

The inputs of the control unit are the instruction machine code from instruction register, the flags value from status register, Branch request, timer interrupt request (timer IRQ) and external interrupt request (external IRQ). The machine code is decoded first before sending to the FSM, while the others inputs are connected directly to the FSM.

As discussed in chapter 3, the design process involves 74 machine codes. The instruction decoder takes the 16-bit machine code from the IR and generates 16 output signals to represents the 74 instructions. At any time, the IR can only have one instruction. So, it will not have more than one output signal active at a time. However, if the machine code received does not match any of the 74 instructions, or is actually NOP instruction, then none of the decoder output signal is active. When none of output signal is active, FSM will not assert any control signal to perform an operation, so no operation is executed in that cycle. Any undetermined instruction is executed as NOP.

### 5.3 The control unit

The control unit is essentially a sequential circuit. The control unit is the final stage for the microcontroller development in this work. The control unit takes control of signal activation of microcontroller circuits in each clock cycle.

#### The Fetch Process:

The fetch process consists in loading one memory address value in the PC, and delivering it to the memory device address port to obtain a specific programming code.



How many clock cycles were needed for fetch process can be guessed using following:

1. The data bus size.
2. The amount of memory used to store the program.
3. Program Counter size.
4. The Existing data path circuitry: It must provide necessary circuits to ensure that PC is incremented in every instruction execution, and that no signal conflict occurs.
5. The fetch process: Designers must ensure that PC is incremented in each instruction, but they must decide how the data moves between the microcontroller circuits. Designers have to make a trade off between different alternatives for fetch process and decide number of the fetch process clock cycles and their data processing route in the data path circuits.

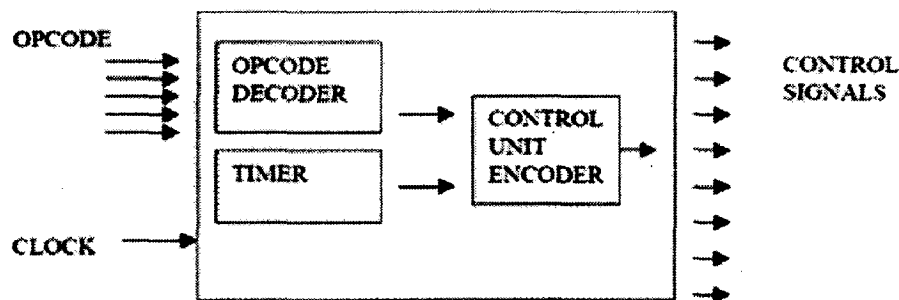


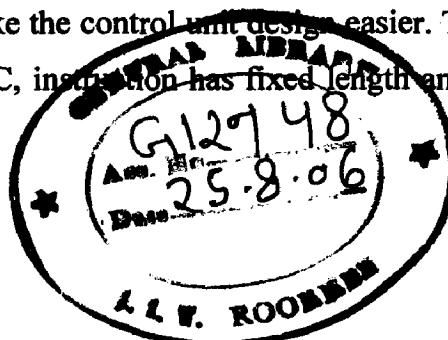
Figure 5.1: control unit implementation

The fetch process used in this work uses just one clock cycles, on rising clock edge the Control Unit activates the IR read signal to load from memory the instruction word to be executed. Also, the tristate buffer is activated to release the current PC value to the data path. On falling edge of clock, Control Unit activates the add PC signal from the ALU to increase the current PC value by one. Finally in that same clock cycle, the Program counter clock is activated to load the incremented value to the PC.

The fetch process needs one circuit that increment the PC by one. We just add one adder to the increment circuit that takes the pc\_reg data and add one to it. One of the advantages of the technique used in this work is that it allows users to add circuit elements without making significant design changes to the entire system.

#### 5.4 Synchronous Mealy Model Finite State Machine

RISC control unit should be hard-wired (logic gates) rather than micro programmed (ROM implementation). Micro programmed control unit is used by CISC because the instruction has different length and execution cycles. So micro programmed can make the control unit design easier. The disadvantage is slower speed performance. In RISC, instruction has fixed length and mostly single cycle execution.



So design using hard-wired is not that complicated and it will have the advantage of speed. The FSM in this design is hard-wired, using logic gates to generate the next state and output signals rather than a ROM. The FSM is implemented using synchronous Mealy model. Figure 5.2 shows block diagram of a synchronous Mealy model FSM.

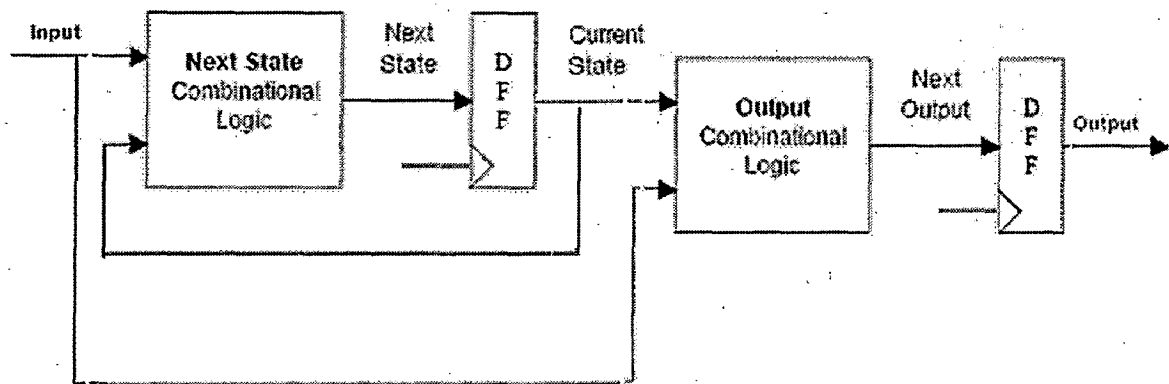


Figure 5.2: block diagram of a synchronous Mealy model FSM.

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

There are basically 2 advantages from using a synchronous Mealy FSM. For a Moore or Mealy FSM, the outputs are generated by the output combinational logic. They will be delay for the signals to pass through the combinational logic before the output is generated. This will slow down the control signals output speed. If the datapath receives control signals later, then will perform their operation later. In synchronous case, outputs are still generated by the combinational logic, but they are now gated to D flip flops.

The first advantage is, on the next clock transition, the outputs are asserted immediately. The datapath receives the control signals at the very beginning of a cycle and therefore can complete its operation faster.

The second advantage is, FSM contains only 7 states, such a small number of states are results of using synchronous Mealy implementation. Since state machine outputs are now gated to flip-flops, all single cycle instruction can share the same state. The state is unchanged but the input changed, so it can determine the next output.

## 5.5 Finite State Machine States

Figure 5.3 shows the state diagram of the finite state machine (FSM). The 7 states are FETCH, DECODE, EXECUTE, WRITE BACK, WAIT, BRANCH1, BRANCH2. The state diagram shows the state flow but does not clearly show the inputs. The inputs to the FSM are the output lines of the instruction decoder, timer IRQ, external IRQ, and branch request. Branch request is generated by the branch evaluation unit when the condition of the conditional branch instruction is fulfilled. We now assume all instructions are single cycle and there are no IRQ, or branch request. The state machine will have no state change in this case and remain at EXE state. All instructions have a fetch, decode, execute and write back cycle and are pipelined together as discussed in chapter 4. When the first instruction is fetched, its corresponding output line of the instruction decoder will become active. It happens in the fetch stage. The next state combinational logic finds that the next state is unchanged. However, the output combinational logic has prepared the control signals based on the decoder's active line. On the next clock transition, the instruction enter the execute stage and the control signals is asserted (latch into the output flip-flops). The ALU then executes the instruction.

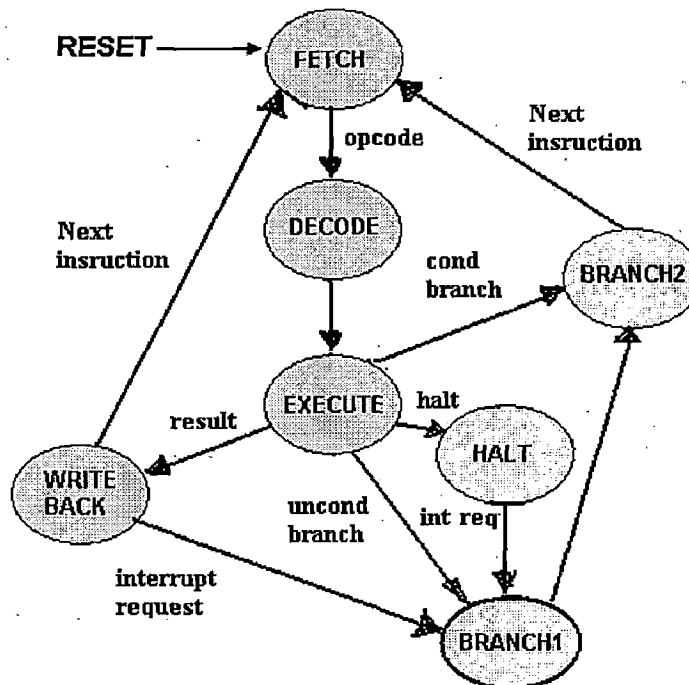


Figure 5.3: The state diagram of FSM.

On the fourth clock transition results are stored in write back cycle, Because of pipeline processing, the next instruction has been fetched at the same clock transition.

The instruction decoder decodes it and asserts another output line. Again, the output logic will prepare the correct control signals and asserts it on the next clock transition. So the FSM can perform the pipeline processing without any difficulty.

We now consider the one of the unconditional branch instruction, SJMP. When SJMP is fetched, the SJMP output line of the decoder is active. The next state logic determined that there would be a state change to BRANCH1 state on the next cycle. The output logic also prepared the control signals for SJMP, which will load the PC with the destination address. On the next clock transition, state changes to BRANCH1 and the control signals are asserted. At BRANCH1, the next state must be BRANCH2. Although the pre-fetched instruction asserts one of the decoder output lines, the output logic does not prepared any control signals for the next cycle. So this instruction is being flushed from the pipeline, as discussed in chapter 5. So on the following clock transition, state changes to BRANCH2 and at the same time, PC is loaded with the new value. The next state will be returned to EXE state. Again, no output signal is asserted based on the fetched instruction because it is flushed. On the next clock transition, the FSM enters EXE state and the destination instruction has been fetched. The decoder's destination instruction output line is active and will be executed on the next cycle. The discussion above is for the SJMP instruction.

The same concept can be applied to RCALL, RET RETI instructions as well as serving an IRQ. An IRQ (timer or external) is sent by the timer or external interrupt module in the datapath. An IRQ can only be served if the I-flag is set, else it will be ignored. To make sure all instructions are completely executed, an IRQ can be only be served in the WRITE\_BACK state. On WRITE\_BACK EXE state, the FSM first check for any IRQ (must have the I-flag set). If there is any, it will ignore the pre-fetched instruction and determines the next state to be BRANCH1. The output logic prepare control signal to load PC with the interrupt vector and to clear the I-flag. I-flag is cleared so that if there is a new IRQ occurred while serving the current one, it will not be served. After loading the interrupt vector to the PC, execution continues as normal but there will not be any IRQ served until the RETI instruction is fetched and executed. It will then set back the I-flag and allowed another IRQ to be served. All conditional branch instruction will take 3 cycles to complete. This can be count from the transitions make to complete the execution from EXE state back to EXE state. (EXE, BRANCH1, BRANCH2, EXE)

The next case to consider is the execution of conditional branch instructions. Different from conditional branch instruction, the branch may or may not be taken. They test a bit in the SR to determine whether the branch should be taken. The branch evaluation unit will do the job on testing the SR flags base on the condition specified. If the condition is fulfilled, it will immediately generate a branch request to the FSM.

When JCC is fetched, the shared instruction decoder output line become active. Different from unconditional branch instructions, there will be no state change on the next cycle. The FSM will assert the branch test signal on the next cycle to request the branch evaluation unit to perform a branch test. If the condition is not fulfilled, no branch request is generated. The pre-fetched instruction is not flushed from the pipeline and is executed. So it takes only one cycle for a conditional branch instruction if the branch is not taken.

If the condition is fulfilled, the branch evaluation unit will send back a branch request to the control unit immediately. At the same time, the control unit will also instruct the PC to load the PC with the destination address. With the branch request, the FSM will transfer to BRANCH2 state on the next clock and the pre-fetched instruction is flushed. On the next clock, the second pre-fetched instruction is also flushed but the FSM now return to EXE state. The next instruction is the destination instruction and will be executed on next cycle. So it takes 3 execution cycles if the branch is taken for conditional branch instructions. Note that the control signal to load the PC is not asserted according to clock transition. It is asserted only after the branch evaluation unit has received the branch test signal and performs the test successfully. So there is delay for the PC to receive the signal in this case.

When the FSM sees the HALT instruction, it will jump to the WAIT state. When in the WAIT state, the PC is stopped and no instruction is executed. Only when there is an IRQ (with the I-flag set), the FSM jumps to BRANCH1 state to serve the interrupt request. The process is exactly the same as serving an IRQ from the EXES. For single cycle instruction, the instruction will not need to be remembered after the control a signal is asserted because it is completed in one cycle. When enter the execute cycle, the next instruction is fetched and the current instruction is lost. However, instructions that require 2 cycles to complete must have some way to remember the instruction in order to assert the correct control signals at the second cycle. So, the FSM provides the second state to remember the instruction. Control signals are based on the state itself without considering the decoder's output line.

If the second cycle of the instructions asserts the same control signals, then the state can be shared, else it will require another one. There are 4 states each requires One cycle for executing instruction, FETCH, DECODE, EXECUTE, and WRITE BACK. The FSM jump to FETCH state, when reset pin is asserted; next state is DECODE then EXECUTE, and lastly WRITE BACK. When one of these instructions is found, the control unit will need to hold the pipeline. The EN signal send to the PC module and IR module will not be asserted for one cycle. So the PC is not incremented and the IR is still holding the pre-fetched instruction.

When the FSM sees a branch request, it will send control signals to the ALU to perform new address calculation. The ALU will send a branch request back to the FSM if the branching condition is fulfilled.

After the long discussion, we should notice when in the EXE state, it will first check to see if there are any branch request or skip request to processed (two of them will never occurred at the same time). If none, it will then check the IRQ. The IRQ must be enabled by the I-flag in order to be served. Only after then it checks the instruction decoder's output to execute an instruction.

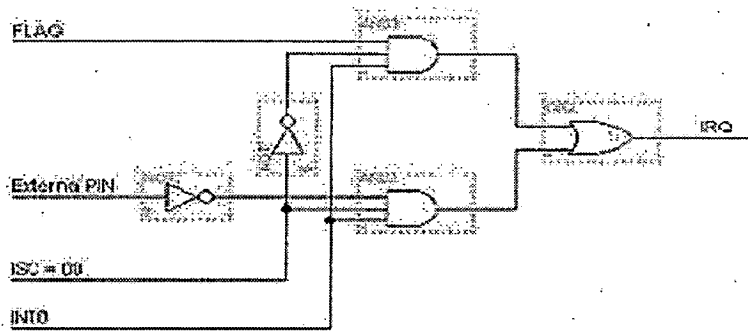
## **5.6 External Interrupt**

The external interrupt is triggered by an external pin. In this design, the external pin shares the pin with PORTs. This pin can be easily changed to share with one of the 24 I/O pins by modifying a single line in the VHDL code. The MCUCR of AT90S1200 has the bits 4 and 5 for controlling the sleep modes of the microcontroller. Since the design does not include this feature, these bits are taken away from the register. The interrupt can be triggered by the external pin on rising edge, falling edge of low level and is selected by the ISC01 and ISC00 bits (interrupt sense control 0).

The interrupt can also be triggered when the external pin is configured as output. The difference now is that the interrupt signal is provided internally from the microcontroller instead of external signal. This provides a way to generate software interrupt by the programmer.

Transitions (falling edge and rising edge) are not detected using the clock input of a flip-flop. The external pin is sampled on every system clock to detect the transitions. A low sample follows by a high sample sense a rising edge while a high sample follows by a low sample sense a falling edge. When the interrupt source is set to falling or rising edge, the external interrupt flag will be set when the require edge is

detected. The external interrupt flag are not accessible by the user. It is not placed inside any of the control register. The flag will stay until the interrupt request is served or after a reset. Figure 5.4 shows how interrupt request is generated. To generate an interrupt request to the control unit, INT0 bit (external interrupt request 0 enable) must be set. This bit is ANDed with the flag to generate interrupt request (with ISC  $\neq$  00).



**Figure 5.4 Generating external interrupt request**

Low-level interrupt are difference from edge interrupt just discussed. It does not set the external interrupt flag to generate an interrupt request. Instead, it never touches the flag. The complement of the external pin (detect low-level) is directly ANDed with the INT0 bit to generate an interrupt request. So if INT0 is set, it will generate an interrupt request as long as the pin is held low. If the interrupt is not enabled when the pin is held low, it will be forgotten when the pin goes high. If the external interrupt is set to edge triggered, the external signal must have sharp transition. If a physical switch is used to generate the interrupt, switch-bounce will occur. It will generate a second, third or more interrupt request even if interrupt request has already been served. So, it is recommended that the low-level interrupt is used, or the switch is hardware de-bounced. Figure 5.4 shows the symbol of the external interrupt module. CLR\_INTF is sent by the control unit to clear the external interrupt flag when the interrupt request is served. RD and WR signals provide reading and writing the control registers through the system data bus.

### 5.7 I/O Decoder

When either the RD\_IO or WR\_IO is asserted, the I/O decoder will decode the I/O address to know exactly which I/O register are to be read of write. Then it sends out the specific read or write control signal for that I/O. In the instruction format section in chapter 4, it is shown that there are two instruction formats for instructions that

accessed the I/O. So the bits location for the I/O address is different. The I/O decoder must be able to know which bits are to be used as the I/O address.

### 5.8 Branch Evaluation Unit

A conditional branch instruction will test one of the 8 bits in the SR. BRBC will take the branch if the specific bit is cleared while BRBS will take the branch if that bit is set. The branch evaluation unit is enabled when the BRANCH\_TEST signal is active. It will then test whether the specific bit meets the branch condition (clear/set). If it does meet the condition, a branch request is generated immediately to the control unit to generate the ADDOFFSET control signal; the next state will now be BRANCH2 state. If the condition is not fulfilled, nothing happens and CPU will execute next instruction.

### 5.9 Timer:

It is important to note that the timer clock source does not drive the TCNT0 directly. Instead, TCNT0 is driven by the system clock. The timer clock source are sampled at the rising edge of the system clock. If a low to high transition is detected (a low is sampled followed by a high), the increment signal for TCNT0 is asserted to increment it. Every transition detected will generate an increment signal pulse. If the timer clock source is the system clock, then no detection of rising edge is required the increment signal is always asserted. To assure proper sampling of the external clock source, the frequency of the external clock should be smaller than the system clock frequency, and the smaller the better.

Every time the increment signal is active, TCNT0 will be incremented by 1. If TCNT0 is \$FF before increment, it will become \$00 after increment and at the same time the timer/counter 0 overflow flag (TOV0) will be set. The timer/counter 0 interrupt overflow interrupt enable flag (TOV0) is ANDed with TOV0 to generate the timer overflow interrupt request. If the TOV0 is set (timer overflow interrupt enabled) and TOV0 is also set (timer overflow occurred), the timer will assert an interrupt request to the control unit. If the I-flag in the SR is enabled, the control unit will serve the interrupt request and clear the TOV0 flag by sending a clear TOV0 signal to the timer module.

Just like other control registers, the 4 timer registers can be read and write through the data bus. However, reserved bits are always read as zero; and the TOV0 flag can be cleared by writing a one to it. In this way, TOV0 flag can never be set by



the user. Reserved bits are not implemented with flip-flops, they are connected directly to ground and this will save a lot of flip-flops. This is why the reserved bits are always read as zero and there are no ways data can be written to them. It can easily be configured to point to any of the 24 I/O pins. CLR\_TOV0 is sent from the control unit to clear the TOV0 flag when the interrupt request is served. The 4 RD signals read the timer control registers to the data bus while the 4 WR signals write the data bus value to the corresponding register.

#### **5.10 Shift register:**

Shift register is used for serial transmission /reception. Data can be transmitted or received serially with start stop bits for serial communication, all necessary handshaking signals are generated according to requirement of program or external devices. Shift register behaves as parallel in serial out register (PISO) when transmitting and serial in parallel out in case of reception

#### **5.11 Implementation Problems**

The following is a small list of problems and important points to keep in mind at Control Unit implementation stage.

1. Due to the many existing control lines, designers must ensure that every signal that goes from the control unit is properly connected to its corresponding circuit.
2. Care should be taken at the interconnection stage because involuntary disconnections may happen.
3. More than one signal is activated per clock cycle, this means that some circuits have to wait for data because probably it is not ready for processing at the circuit signal activation moment. To solve this problem, once the control unit is connected to all circuits, designers have to run manually with the control unit clock, each and every one of the instructions to see per clock its performance.
4. Once a time delay problem has found (you will know that this problem happens because in its respective instruction clock cycle, when you run it manually, there is not data in some circuits that is supposed to be. This means that a time delay must be added to the circuit element that does not receive the data.

## 6.1 FPGA Architecture [27]

FPGA's are introduced first by the Xilinx Inc. in 1985. Since that time, the FPGA market has expanded dramatically with many different competing designs developed by companies including, Altera, AMD, Intel, Motorola, AT&T, Actel, Atmel, Cypress, Texas Instruments, Quick Logic, and Lattice semiconductor. Field Programmable Gate Arrays are a relatively new class of integrated circuit. A FPGA is of similar kind as a CPLD (complex programmable logic devices), which consists of programmable logic blocks (combinational logic blocks, programmable IO blocks and programmable interconnection matrix). The logic is broken into large number of programmable logic blocks that are individually smaller than a PLD, as shown in Fig. 6.1.

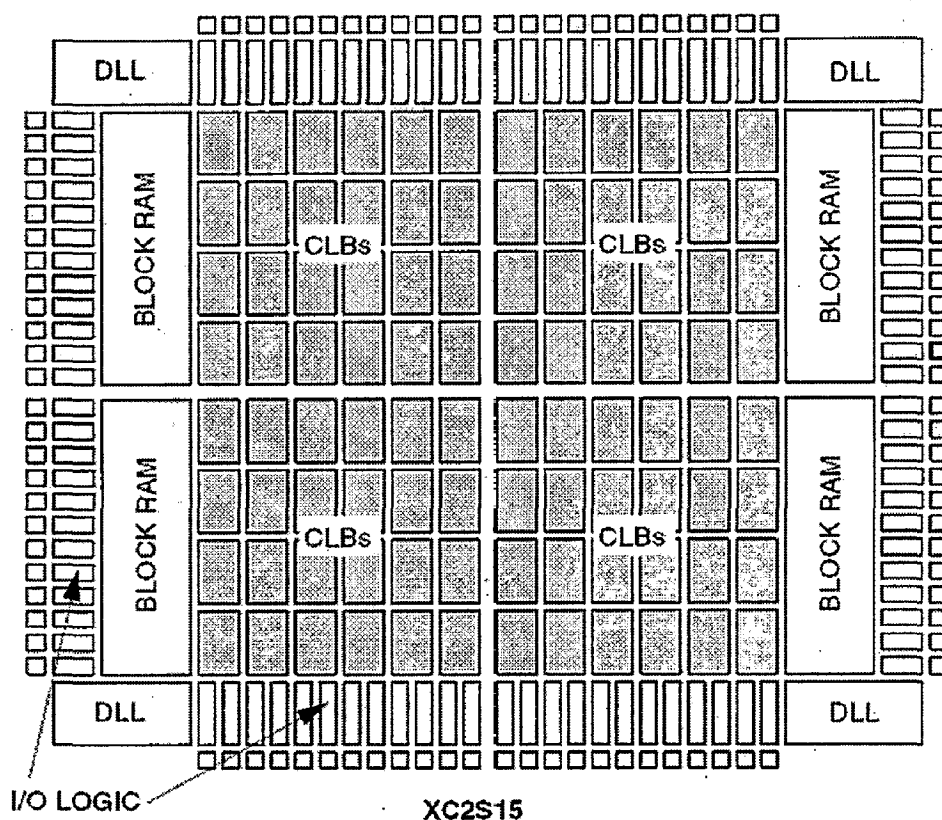


Figure 6.1: Basic Spartan II family FPGA block diagram [27]

They are distributed across the entire chip in a sea of programmable interconnections which can be configured by the user at the point of application, & the entire array is surrounded by programmable I/O blocks. User programming specifies both the logic function of each block and the connections between the blocks. An

FPGA's programmable logic block is less capable than a typical PLD, but an FPGA chip contains a lot more logic blocks than a CPLD of the same die size has PLDs.

## 6.2 Programming with FPGA

Although early PLD and FPGA designs were generated largely by hand, access to today's CPLDs requires the use of an integrated Computer Aided Design (CAD) system. Figure 6.2 illustrates the typical sequence of operations needed to go from concept to programmed chip. Both commercial CAD tool vendors and FPGA companies offer appropriate tools. For example, traditional Electronic Design Automation (EDA) vendors such as Mentor Graphics, Synopsys, Cadence, and ViewLogic all offer tools to support FPGA design. These tools are typically used for the front-end design entry and simulation operations and provide the necessary interfaces to vendor-specific back-end tools for chip placement and routing.

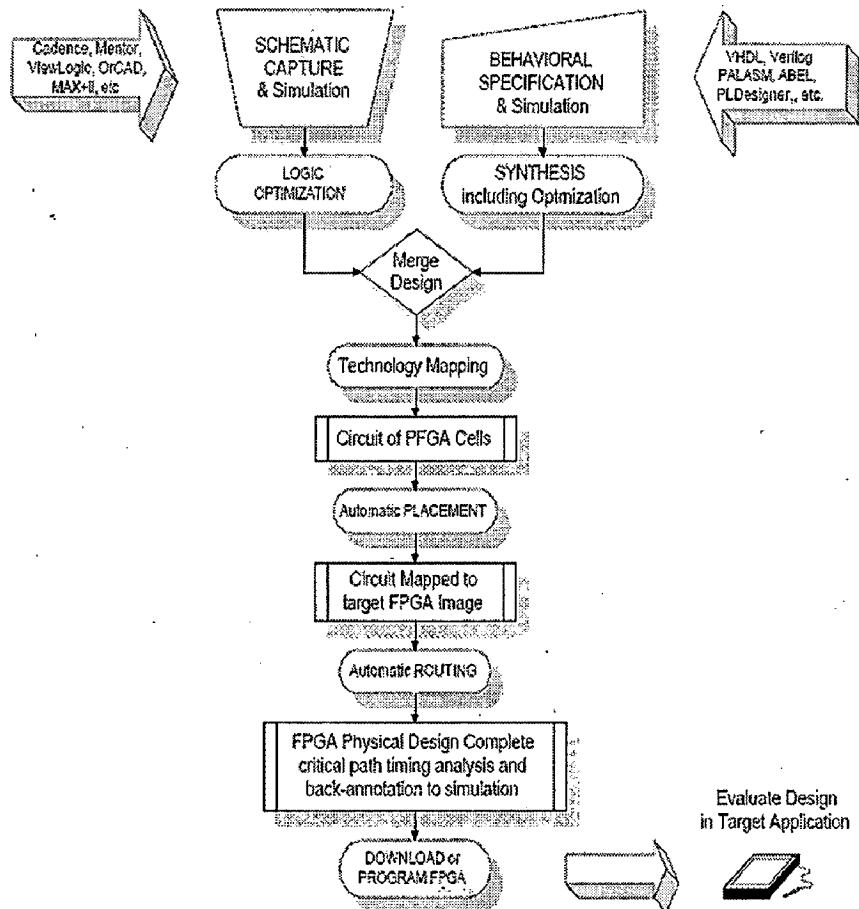
Examples of vendor specific tools are the Xilinx XST system and the Altera Quartus II software. It is worth noting that Xilinx ISE 7.1i software, which supports the entire design flow, as illustrated in Figure 6.5 on either PC or workstation platforms. The following discussion is meant to be indicative of the general operations and steps required in FPGA design. Where appropriate, examples are taken from the Xilinx and Altera CAD design flows to illustrate the generic operations.

Traditionally, a designer uses schematic capture tools for graphical entry of a logic design which has been manually generated to meet the architectural or behavioral specifications. The upper left hand arrow in Figure 6.2 identifies some of the commercial CAD tools available for FPGA schematic capture.

One of the more significant recent innovations in the EDA industry is the development of tools which allow the designer to move from the gate level to the behavioral level for design entry. A behavioral design specification is created using a Hardware Description Language (HDL), and then a synthesis tool automatically compiles the gate level schematic or netlist from the behavioral description. The upper right hand arrow in Figure 6.2 indicates some of the HDLs currently being used for FPGA behavioral modeling.

Options for behavioral description of designs include the VHSIC Hardware Description Language (VHDL), the Verilog hardware description language, timing diagrams, logic state diagrams, and PLD description languages such as ABEL. As an

example of how pervasive the behavioral design style has become, the PC-based Xilinx ISE 7.1i software provides multiple options for behavioral design entry. In addition to traditional schematic capture it will accept VHDL, text design description in the Hardware Description Language (including truth tables and Boolean expressions), and Timing Diagrams which describe the desired input and output waveforms. Whichever behavioral design entry method is chosen, the design system provides logic synthesis which automatically creates gate-level schematics.



**Figure 6.2 Typical CAD system design flow for FPGAs[26]**

No matter what method is used for initial design entry, the next step in FPGA design is to translate the entire design into a standard form which can be processed by a logic optimization tool. The goal of logic optimization is to perform minimization of the Boolean expressions and eliminate redundancy, thus minimizing the area of the final circuit. The tool may also be constrained to maximize speed at the expense of area by limiting the number of logic levels between clocked registers. This optimization process is usually merged with the logic synthesis step when behavioral design entry is

employed. Simulation is performed both before and after the logic optimization steps to verify that the design meets the original system requirements for functionality and timing. The next step is to convert the generic gate level design into one which uses the FPGA circuit building blocks of the target technology.

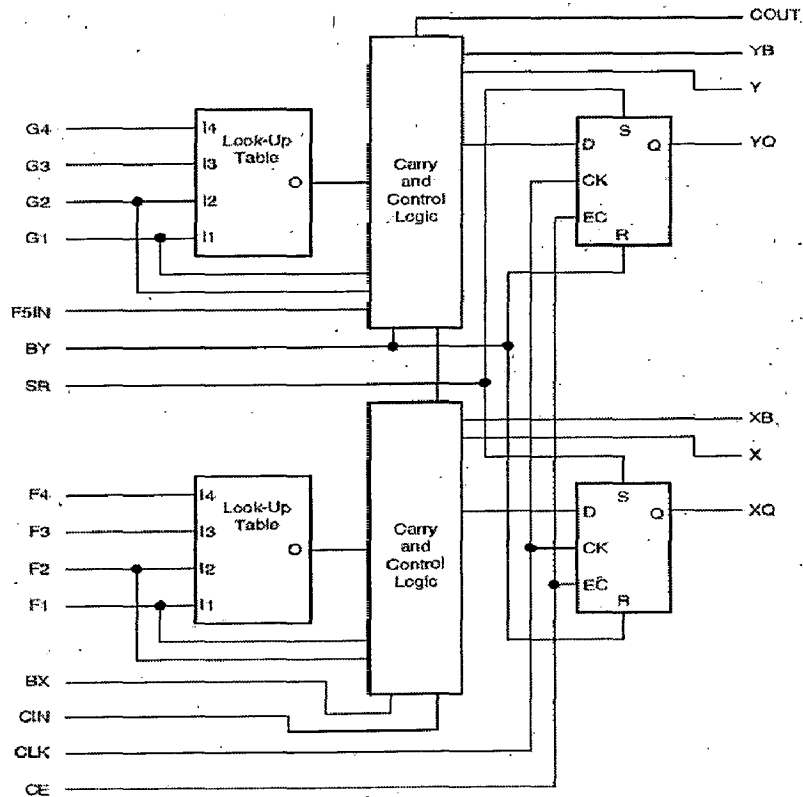


Figure 6.3: Spartan II CLB slice[27]

To provide a concrete example, the Xilinx XST design system flow will be used to illustrate the steps needed to go from logic design to programmed FPGA. In the Xilinx design flow, the native format of the logic design (Cadence, ViewLogic, OrCAD, etc.) must first be translated into the Xilinx Netlist Format (XNF) which is understood by the Xilinx tools. Next, the XNF circuit description must be mapped into Xilinx Configurable Logic Blocks (CLBs). This is the technology mapping step referred to in Figure 6.4. Xilinx calls this step "partitioning", and the XST tools also attempt to optimize the circuit during this step. For example, circuitry associated with unused logic block inputs or outputs is eliminated from the design. In addition, the partitioning program attempts to minimize either the total number of CLBs used or the number of logic stages in the critical delay path. The next step is to place and route the design on the selected chip image. The XST system allows manual and/or automatic placement and routing. In the automatic placement operation, each CLB generated during the "partitioning" step is assigned to a physical location on the chip. Xilinx uses

a Simulated Annealing algorithm which starts with a random placement, and then goes through a series of improvement passes. This program can be run multiple times with different starting random seeds in an attempt to generate a more optimal placement. Following placement, interconnections between the CLBs must be routed using the available interconnect segments and switch matrix elements. XST uses an automatic Maze Routing Algorithm to perform this operation. With the physical placement and routing completed, exact timing values can now be used to determine chip performance. The XST tools provide a critical path timing analyzer which provides delay information on longest / shortest paths through the chip.

In addition, the physical layout timing information can also be back-annotated to the schematics to get more accurate functional simulation results. The final step in the Xilinx or Altera design flow is the creation of the BIT file which contains the binary programming data needed to configure the SRAM bits of the target chip. This file is then downloaded to configure the chip for final functional & timing tests of programmed chip.

### 6.3 FPGA Design Environment

1	<b>Software Environment:</b>	
	Operating System	Windows XP-Pro sp 2
	Development Tool	VHDL
	Synthesis Tool	Xilinx ISE 7.1i
	Simulation Tool	Aldec Active HDL 6.3, Xilinx ISE 7.1i
	Implementation	FPGA (Xilinx Spartan II )

2	<b>Hardware Environments:</b>	
2.1	<b>For simulation and Synthesis:</b>	
	Processor	Pentium 4
	RAM	512 MB
	Processor Speed	2.66 GHz.
2.2	<b>For Implementation:</b>	
	FPGA Device Family	Xilinx Spartan II
	Device	XC2S200-5PQ208
	Speed Grade	5
	Output Display	Seven segment LED display
	Top-level Module Type	HDL
	Synthesis Tool	Xilinx ISE XST 7.1i (using VHDL)
	Simulator	Aldec Active HDL 6.3, Xilinx ISE 7.1i
	Generated Simulation Language	VHDL

## 6.4 FPGA design flow for implementation

Since the goal of this dissertation is to create a full custom processor design in FPGA, for this reason the implementation of Microcontroller requires FPGA design flow steps to be followed. Figure 6.4 shows a standard design flow for a FPGA design:

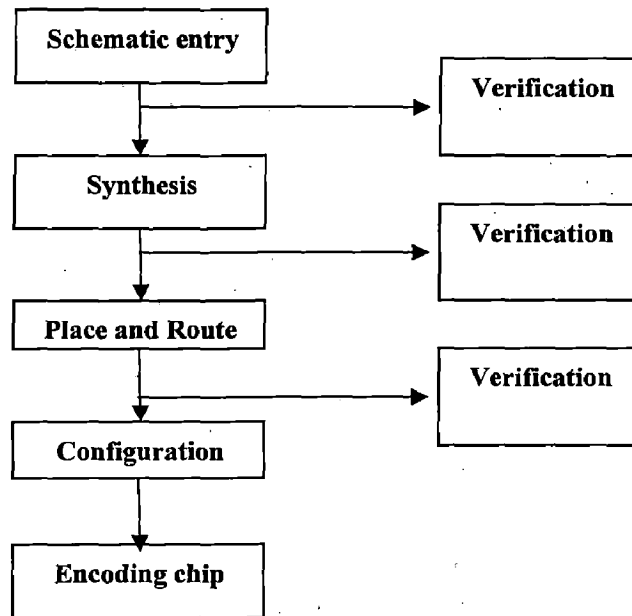


Figure 6.4: FPGA Design Flow

### 1 Schematic entry

The design is entered into a synthesis design system using a hardware description language. The language used here is VHDL.

### 2 Syntheses

A netlist is generated using the VHDL code and a logic synthesis tool using Xilinx ISE 7.1i EDA tool, synthesis report gives idea about possible shortfalls in the generated RTL model

### 3 Place and Route

The place process decides the best location of the cells in a block based on the logic and desired performance. The route process makes the connections between the cells and the blocks. Automatic place and route is done by the synthesis tool after generating netlist.

#### **4 Configuration:**

This is done by loading the configuration data into the internal memory. Synthesis tool generates a bit stream file after placing and routing, which is then downloaded in FPGA. I used JTAG cable to load my design in the FPGA.

#### **5 Verification**

At each step of the design process, I verified my architecture using software simulation using Aldec Active HDL 6.3 software package. Each instruction is simulated and its impact is studied to make any changes in design to optimize the performance.

#### **6.5 Verification and Testing**

For the testing purpose, I designed my own ROM for simulation of my program. I used binary representation of the instructions in that ROM. So it was very easy to check all the instructions just by changing the binary instruction and then comparing the simulated results with expected results.

Application program is written initially in ROM module before design is implemented in FPGA. I wrote entire program in 16 bit binary machine codes, In that program, I wrote the binary representation of all the instructions that were to be verified. The reason for using binary representation is that otherwise it would take too much time to develop a new assembler to interpret text based assembly code.

I tested designed microcontroller architecture by running many test programs that were created using the method mentioned above. The basic verification approach was to compare the simulated output results with the expected results that I computed by hand. Whenever I found a mismatch between the two, I identified the problem(s) and took care of them appropriately. I tested the functionality of all the instructions, the interactions among the instructions in the pipeline, and the correctness of the data as a result of executing those instructions.





## CHAPTER 7: RESULTS AND CONCLUSION

---

### 7.1 Results

In the recent years several embedded cores of well known microcontrollers have been implemented. ASICs encompassing these cores have proven successful and have been manufactured without complications. Compared to the standard products (genuine microcontrollers) microcontroller designed in this dissertation reach considerably higher performance; on the one hand because of higher clock rates due to newer technologies and on the other hand because of an improved internal architecture.

In this dissertation work a 16 bit RISC microcontroller modeled as the structural design using VHDL and is successfully implemented on the Xilinx Spartan II kit (xc2s200-5pq208), special provision for register file is made by using xilinx coregent to eliminate any possibility of error, a synchronous finite state machine is designed for updating registers as well as data memory, generated core for regfile and data memory works fine in simulation.

This dissertation has detailed the results of creating and simulating two VHDL models for the Atmel RISC microcontroller. These models were written to help support an effort to eradicate the component obsolescence problem faced. A simple Behavioral model was written and tested to verify proper abstract modeling. Next, an Instruction model was written for almost all of the instruction set of the Atmel RISC microcontroller. The Instruction Set model results were compared against the results from an actual Atmel RISC microcontroller accessed and simulated by using a actel active HDL 6.3 as well as Xilinx ISE 7.1i. For the modeled instructions, it was found that the Instruction Set model accurately emulates functionality of the original Atmel RISC microcontroller. The efforts of this dissertation will help support the overall goal to fabricate an 16-bit microcontroller that emulates with form, fit, and function the original Atmel RISC microcontroller.

### 7.2 Conclusion

As a conclusion, this project has been completed successfully fulfilling are the objectives and scopes specified. The author has used his extra time to optimize the speed of the design until 12 MHz. also code is optimized to minimum utilisation of

resources in FPGA The data RAM that is not specified in the scope of the project has also been included. The stack is also incorporated in design and a total of 24 I/O lines are available for I/O device programming. Since the project now occupies 92% of the FPGA device (Spartan II xc2s200-5pq208), the author recommends that the laboratory provides a larger FPGA device. Table 7.1 is the comparison chart between atmel RISC Microcontroller AT90S1200 and the RISC Microcontroller designed in the dissertation work.

Table 7.1 comparison between Atmel microcontroller and current design

<b>FEATURES</b>	<b>AT90S1200</b>	<b>RISC_MICROCONTROLLER</b>
<b>SRAM</b>	NONE	32 WORD
<b>PROGRAM ROM</b>	1KB	4KB
<b>INSTRUCTION SET</b>	89	74
<b>GP REGISTERS</b>	32	32
<b>IO PORTs</b>	2 (8 BIT EACH)	2 (16 BIT EACH)
<b>TIMMER</b>	1	2 (16 BIT)
<b>EXTERNAL INTERRUPTS</b>	1	5
<b>ADDRESSING MODE</b>	5	4
<b>SPEED</b>	4-12 MHz	12 MHz
<b>ANOLOG COMPARATOR</b>	1	NONE
<b>IMPLEMENTATION</b>	CMOS	FPGA

Developing a (VHDL) model compatible to an industry-standard component involves previously unidentified technical constraints when intended to be reused as an IP core. Implementing a building block for future integration includes issues like, thorough and exhaustive verification of both the RTL model and the gate level netlist to achieve highest quality possible; considering the trade-off between compatibility and performance improvement; In this dissertation work I have focused these issues and proposed some approaches how to tackle them.

I have tested my processor architecture by running many test programs that were created using the method mentioned above. The basic verification approach was to compare the simulated output results with the expected results that I computed by

hand. Whenever I found a mismatch between the two, I identified the problem(s) and took care of them appropriately. I tested the functionality of all the instructions, the interactions among the instructions in the pipeline, and the correctness of the data as a result of executing those instructions.

### **7.3: Recommendation on Future Works**

Items to research and complete include: full utilization of instruction set, finishing the Instruction Set model, writing an RTL model, synthesizing the RTL model, and testing all models. Some of this work has been completed, but was not documented in this thesis. RTL models and synthesis results have been made available to department.

A serious design problem is encountered by author while updating register file and also data memory, it is been found that because of some FPGA kit constraints selected RAM Block is not functioning properly, author is trying to sort out problems at various stages of design. Special care has been taken for removing any pipeline hazards and appropriate results are always been sent to ALU.

There are many more extra features available in the AVR RISC microcontroller family, such as the UART serial interface, SPI serial interface, the 16-bit timer (with output compare and input capture), etc. This works from this project should be used as a platform to implement these features in.

## REFERENCES:

---

1. Mazidi and Mazidi, "The 8051 microcontrollers and Embedded systems", Pearson education, 2006.
2. Myke Predko, Programming and customizing 8051 microcontrollers, Tata McGraw hill company, New Delhi, 2001.
3. Steve heath, "Embedded system design", second edition, EDN series for design engineers 2000.
4. Kenneth ayala, "The 8051 microcontroller Architecture, Programming and applications", second edition, 1998
5. Todd Morton, "Embedded microcontrollers", Pearson education, 2003.
6. Perry, D. L. 2004. VHDL: Programming By Example, 4th edition. New York: McGraw-Hill Companies inc.
7. Bhasker, J. 1997. A VHDL Primer. Allentown, PA: Star Galaxy Press
8. Stallings William, 1993, Computer Organization and Architecture, 3rd edition, Macmillan Publishing Company.
9. Patterson David A, Hennessy John, 1994, Computer Organization & Design, Morgan Kaufmann Publishers, Inc.
10. Brey Barry, the Intel Microprocessors 808X, 80286, 80386, 80486, Pentium & Pentium Pro, Prentice Hall, 1997.
11. RISC Pipelining, Stanford University, HTTP:  
<http://cse.stanford.edu/class/sophomore-college/projects-00/RISC/pipelining/>
12. John peatman, Design with PIC microcontroller", Pearson education, 2003.
13. Mano, M. M. 2001. Logic and Computer Design Fundamentals, 2nd ed. Upper Saddle River, NJ: Prentice Hall
14. Shashi Raj Kapoor, "advanced processor design implementation using FPGA architecture", A dissertation report M-Tech Electrical, IIT Roorkee, 2003

### Other Useful References

15. Melear, C.; "The MPC5005 RISC microcontroller", Southcon/95. Conference Record 7-9 March 1995 Page(s):79 – 83.
16. Ozaki, Nishimichi, Kakiage, Yamamoto, Sumita, Inoue, Urano, Yamashita, Maeda, Nishiyama, "A 100 Mhz Embedded RISC Microcontroller", 1994.

17. Digest of Technical Papers., 1994 Symposium on **VLSI Circuits**, JUNE 9-11, 1994 Page(s):67 – 68
18. Qing-Lan Lv; Ping Li; **A 8-bit MCU design using a four-pipeline architecture** IEEE 2002 International Conference on Communications, Circuits and Systems, Volume 2, 29 June-1 July 2002 Page(s):1462 - 1465 vol.2
19. Christian, masgonty, claude, durand “**low power 8 bit embedded coolRISC microcontroller cores**”, IEEE journal of solid state circuits vol32 no 7 july 97
20. Jan gray, “**Designing a simple FPGA optimized RISC CPU and system on chip**”
21. Ferreira, L.F.; Matos, E.L.; Menendez, L.M.; Mandado, E.; **MILES: A Microcontroller Learning System combining Hardware and Software tools** 2005. FIE '05. Proceedings 35th Annual Conference Frontiers in Education, 19-22 Oct. 2005 Page(s):F4E-7 - F4E-11
22. FPGA World “Demos on Demand” [Online Demos are available by vendors of VLSI firm, HTTP: <http://www.demosondemand.com/dod/>]
23. VHDL Tutorial: Learn by Example  
<http://www.cs.ucr.edu/content/esd/labs/tutorial/>
24. IBM RESEARCH <http://www.research.ibm.com>.
25. INTEL <http://www.intel.com>.
26. Atmel <http://www.atmel.com>.
27. Altera <http://www.altera.com>.
28. xilinx <http://www.xilinx.com>.

# APPENDIX A: Atmel AVR RISC Microcontroller data sheet

## Instruction Set Summary

Mnemonic	Operands	Description	Operation	Flags
<b>ARITHMETIC AND LOGIC INSTRUCTIONS</b>				
ADD	Rd, Rr	Add Two Registers	$Rd \leftarrow Rd + Rr$	Z,C,N,V,H
ADC	Rd, Rr	Add with Carry Two Registers	$Rd \leftarrow Rd + Rr + C$	Z,C,N,V,H
SUB	Rd, Rr	Subtract Two Registers	$Rd \leftarrow Rd - Rr$	Z,C,N,V,H
SUBI	Rd, K	Subtract Constant from Register	$Rd \leftarrow Rd - K$	Z,C,N,V,H
SBC	Rd, Rr	Subtract with Carry Two Registers	$Rd \leftarrow Rd - Rr - C$	Z,C,N,V,H
SBCI	Rd, K	Subtract with Carry Constant from Reg.	$Rd \leftarrow Rd - K - C$	Z,C,N,V,H
AND	Rd, Rr	Logical AND Registers	$Rd \leftarrow Rd \wedge Rr$	Z,N,V
ANDI	Rd, K	Logical AND Register and Constant	$Rd \leftarrow Rd \wedge K$	Z,N,V
OR	Rd, Rr	Logical OR Registers	$Rd \leftarrow Rd \vee Rr$	Z,N,V
ORI	Rd, K	Logical OR Register and Constant	$Rd \leftarrow Rd \vee K$	Z,N,V
EOR	Rd, Rr	Exclusive OR Registers	$Rd \leftarrow Rd \oplus Rr$	Z,N,V
COM	Rd	One's Complement	$Rd \leftarrow \sim Rd$	Z,C,N,V
NEG	Rd	Two's Complement	$Rd \leftarrow \$00 - Rd$	Z,C,N,V,H
SBR	Rd, K	Set Bit(s) in Register	$Rd \leftarrow Rd \vee K$	Z,N,V
CBR	Rd, K	Clear Bit(s) in Register	$Rd \leftarrow Rd \wedge (\sim K)$	Z,N,V
INC	Rd	Increment	$Rd \leftarrow Rd + 1$	Z,N,V
DEC	Rd	Decrement	$Rd \leftarrow Rd - 1$	Z,N,V
TST	Rd	Test for Zero or Minus	$Rd \leftarrow Rd \wedge Rd$	Z,N,V
CLR	Rd	Clear Register	$Rd \leftarrow Rd \oplus Rd$	Z,N,V
SER	Rd	Set Register	$Rd \leftarrow \$FF$	None
<b>BRANCH INSTRUCTIONS</b>				
RJMP	K	Relative Jump	$PC \leftarrow PC + K + 1$	None
RCALL	k	Relative Subroutine Call	$PC \leftarrow PC + k + 1$	None
RET		Subroutine Return	$PC \leftarrow STACK$	None
RETI		Interrupt Return	$PC \leftarrow STACK$	I
CPSE	Rd, Rr	Compare, Skip if Equal	if (Rd = Rr) $PC \leftarrow PC + 2$ or 3	None
CP	Rd, Rr	Compare	$Rd - Rr$	Z,N,V,C,H
CPC	Rd, Rr	Compare with Carry	$Rd - Rr - C$	Z,N,V,C,H
CPI	Rd, K	Compare Register with Immediate	$Rd - K$	Z,N,V,C,H
SBRC	Rr, b	Skip if Bit in Register Cleared	if (Rr(b) = 0) $PC \leftarrow PC + 2$ or 3	None
SBRB	Rr, b	Skip if Bit in Register is Set	if (Rr(b) = 1) $PC \leftarrow PC + 2$ or 3	None
SBIC	P, b	Skip if Bit in I/O Register Cleared	if (P(b) = 0) $PC \leftarrow PC + 2$ or 3	None
SBIS	P, b	Skip if Bit in I/O Register is Set	if (P(b) = 1) $PC \leftarrow PC + 2$ or 3	None
BRBS	s, k	Branch if Status Flag Set	if (SREG(s) = 1) then $PC \leftarrow PC + k + 1$	None
BRBC	s, k	Branch if Status Flag Cleared	if (SREG(s) = 0) then $PC \leftarrow PC + k + 1$	None
BREQ	k	Branch if Equal	if (Z = 1) then $PC \leftarrow PC + k + 1$	None
BRNE	k	Branch if Not Equal	if (Z = 0) then $PC \leftarrow PC + k + 1$	None
BRCS	k	Branch if Carry Set	if (C = 1) then $PC \leftarrow PC + k + 1$	None
BRCC	k	Branch if Carry Cleared	if (C = 0) then $PC \leftarrow PC + k + 1$	None
BRSH	k	Branch if Same or Higher	if (C = 0) then $PC \leftarrow PC + k + 1$	None
BRLO	k	Branch if Lower	if (C = 1) then $PC \leftarrow PC + k + 1$	None
BRMI	k	Branch if Minus	if (N = 1) then $PC \leftarrow PC + k + 1$	None
BRPL	k	Branch if Plus	if (N = 0) then $PC \leftarrow PC + k + 1$	None
BRGE	k	Branch if Greater or Equal, Signed	if (N $\oplus$ V = 0) then $PC \leftarrow PC + k + 1$	None
BRLT	k	Branch if Less than Zero, Signed	if (N $\oplus$ V = 1) then $PC \leftarrow PC + k + 1$	None
BRHS	k	Branch if Half-carry Flag Set	if (H = 1) then $PC \leftarrow PC + k + 1$	None
BRHC	k	Branch if Half-carry Flag Cleared	if (H = 0) then $PC \leftarrow PC + k + 1$	None
BRTS	k	Branch if T-Flag Set	if (T = 1) then $PC \leftarrow PC + k + 1$	None
BRTC	k	Branch if T-Flag Cleared	if (T = 0) then $PC \leftarrow PC + k + 1$	None
BRVS	k	Branch if Overflow Flag is Set	if (V = 1) then $PC \leftarrow PC + k + 1$	None
BRVC	k	Branch if Overflow Flag is Cleared	if (V = 0) then $PC \leftarrow PC + k + 1$	None
BRIE	k	Branch if Interrupt Enabled	if (I = 1) then $PC \leftarrow PC + k + 1$	None
BRID	k	Branch if Interrupt Disabled	if (I = 0) then $PC \leftarrow PC + k + 1$	None
<b>DATA TRANSFER INSTRUCTIONS</b>				
LD	Rd, Z	Load Register Indirect	$Rd \leftarrow (Z)$	None
ST	Z, Rr	Store Register Indirect	$(Z) \leftarrow Rr$	None
MOV	Rd, Rr	Move between Registers	$Rd \leftarrow Rr$	None
LDI	Rd, K	Load Immediate	$Rd \leftarrow K$	None
IN	Rd, P	In Port	$Rd \leftarrow P$	None
OUT	P, Rr	Out Port	$P \leftarrow Rr$	None

## Instruction Set Summary (Continued)

Mnemonic	Operands	Description	Operation	Flags
<b>BIT AND BIT-TEST INSTRUCTIONS</b>				
SBI	P, b	Set Bit in I/O Register	$IO(P,b) \leftarrow 1$	None
CBI	P, b	Clear Bit in I/O Register	$IO(P,b) \leftarrow 0$	None
LSL	Rd	Logical Shift Left	$Rd(n+1) \leftarrow Rd(n), Rd(0) \leftarrow 0$	Z,C,N,V
LSR	Rd	Logical Shift Right	$Rd(n) \leftarrow Rd(n+1), Rd(7) \leftarrow 0$	Z,C,N,V
ROL	Rd	Rotate Left through Carry	$Rd(0) \leftarrow C, Rd(n+1) \leftarrow Rd(n), C \leftarrow Rd(7)$	Z,C,N,V
ROR	Rd	Rotate Right through Carry	$Rd(7) \leftarrow C, Rd(n) \leftarrow Rd(n+1), C \leftarrow Rd(0)$	Z,C,N,V
ASR	Rd	Arithmetic Shift Right	$Rd(n) \leftarrow Rd(n+1), n = 0..6$	Z,C,N,V
SWAP	Rd	Swap Nibbles	$Rd(3..0) \leftrightarrow Rd(7..4), Rd(7..4) \leftrightarrow Rd(3..0)$	None
BSET	s	Flag Set	$SREG(s) \leftarrow 1$	SREG(s)
BCLR	s	Flag Clear	$SREG(s) \leftarrow 0$	SREG(s)
BST	Rr, b	Bit Store from Register to T	$T \leftarrow Rr(b)$	T
BLD	Rd, b	Bit Load from T to Register	$Rd(b) \leftarrow T$	None
SEC		Set Carry	$C \leftarrow 1$	C
CLC		Clear Carry	$C \leftarrow 0$	C
SEN		Set Negative Flag	$N \leftarrow 1$	N
CLN		Clear Negative Flag	$N \leftarrow 0$	N
SEZ		Set Zero Flag	$Z \leftarrow 1$	Z
CLZ		Clear Zero Flag	$Z \leftarrow 0$	Z
SEI		Global Interrupt Enable	$I \leftarrow 1$	I
CLI		Global Interrupt Disable	$I \leftarrow 0$	I
SES		Set Signed Test Flag	$S \leftarrow 1$	S
CLS		Clear Signed Test Flag	$S \leftarrow 0$	S
SEV		Set Two's Complement Overflow	$V \leftarrow 1$	V
CLV		Clear Two's Complement Overflow	$V \leftarrow 0$	V
SET		Set T in SREG	$T \leftarrow 1$	T
CLT		Clear T in SREG	$T \leftarrow 0$	T
SEH		Set Half-carry Flag in SREG	$H \leftarrow 1$	H
CLH		Clear Half-carry Flag in SREG	$H \leftarrow 0$	H
NOP		No Operation		None
SLEEP		Sleep	(see specific descr. for Sleep function)	None
WDH		Watchdog Reset	(see specific descr. for WDT/Timer)	None

Table 1. The AT90S1200 I/O Space

Address Hex	Name	Function
\$3F	SREG	Status REGISTER
\$3B	GIMSK	General Interrupt MASK register
\$39	TIMSK	Timer/Counter Interrupt MASK register
\$38	TIFR	Timer/Counter Interrupt Flag register
\$35	MCUCR	MCU general Control Register
\$33	TCCR0	Timer/Counter0 Control Register
\$32	TCNT0	Timer/Counter0 (8-bit)
\$21	WDTCR	Watchdog Timer Control Register
\$1E	EEAR	EEPROM Address Register
\$1D	EEDR	EEPROM Data Register
\$1C	EECR	EEPROM Control Register
\$18	PORTB	Data Register, Port B
\$17	DDRB	Data Direction Register, Port B
\$16	PINB	Input Pins, Port B
\$12	PORTD	Data Register, Port D
\$11	DDRD	Data Direction Register, Port D
\$10	PIND	Input Pins, Port D
\$08	ACSR	Analog Comparator Control and Status Register



## APPENDIX B: INSTRUCTION SET

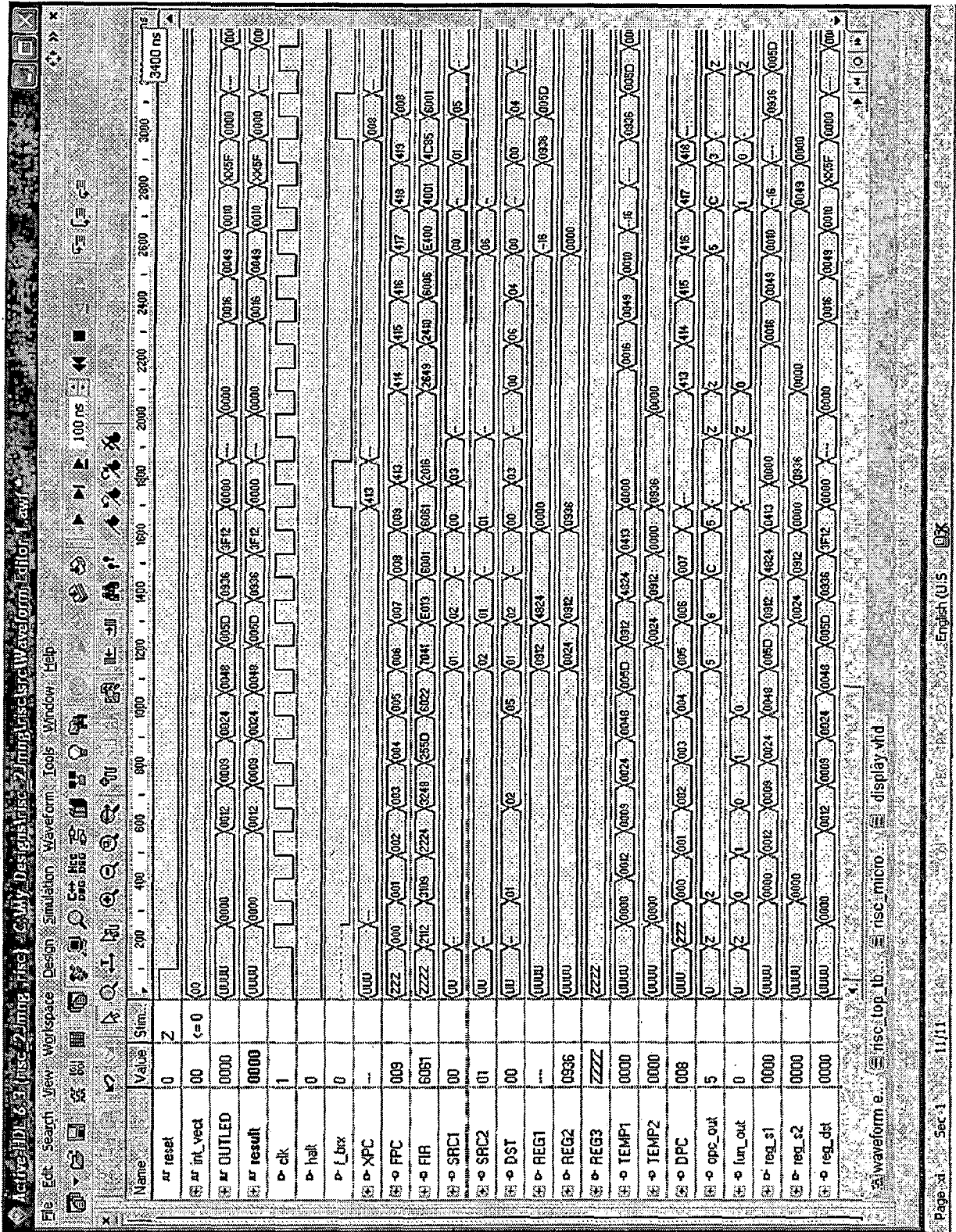
**Table: INSTRUCTION SET OF RISC MICROCONTROLLER**

Opcode	Mnemonic	Function
0000 00 xx xxxx xxxx	NOP	No operation
0000 01 xx xxxx xxxx	HALT	Halts execution
0000 10 xx xxxx xxxx	SWAP	Exchange lower byte with higher byte and vice versa of register
0000 11 xx xxxx xxxx	SWAPA	Exchange lower byte with higher byte and vice versa of accumulator
0001 000x bits(4) reg(4)	ROL A, bits	Rotate left accumulator
0001 001x bits(4) reg(4)	RLC A, bits	Rotate left accumulator through carry
0001 010x bits(4) reg(4)	ROR A, bits	Rotate right accumulator
0001 011x bits(4) reg(4)	RRC A, bits	Rotate right through carry accumulator
0001 100x bits(4) reg(4)	ROL reg, bits	Rotate left register
0001 101x bits(4) reg(4)	RLC reg, bits	Rotate left through carry register
0001 110x bits(4) reg(4)	ROR reg, bits	Rotate right register
0001 111x bits(4) reg(4)	RRC reg, bits	Rotate right through carry register
0100 reg(4) imm(8)	MVIL reg, imm8	Move data as lower byte of register
0101 reg(4) imm(8)	MVIH reg, imm8	Move data as upper byte of register
0011 reg(4) imm(8)	MVIS sp-reg, imm8	Move data as lower byte of register
0010 000x reg(4) reg(4)	MOV1 reg1, reg2	Move contents of reg2 to reg1
0010 001x reg(4) reg(4)	MOV2 sp-reg, reg2	Move contents of reg2 to sp-reg
0010 010x reg(4) reg(4)	MOV3 reg1, sp-reg	Move contents of sp-reg to reg1
0010 011x reg(4) reg(4)	MOV4 sp-reg1, sp-reg2	Move contents of sp-reg2 to sp-reg1
0010 100x reg(4) mem(4)	LOAD reg, mem	Move contents of mem to register
0010 101x mem(4) reg(4)	STORE mem, reg	Move contents of register to mem
0010 110x reg(4) mem(4)	LDA mem	Move contents of mem to ACC
0010 111x mem(4) reg(4)	STA mem	Move contents of ACC to mem
0101 00xx bits(4) reg(4)	SETB reg, bit	Set specified bit of register
0101 01xx bits(4) reg(4)	CLRB reg, bit	Clear specified bit of register
0101 10xx bits(4) reg(4)	CPLB reg, bit	complement bit of register

0101 11xx bits(4) reg(4)	GETB reg, bit	output specified bit of register
0110 000x reg(4) reg(4)	ADD reg1, reg2	add contents of reg2 to reg1
0110 001x reg(4) reg(4)	ADC reg1, reg2	add contents of reg2 to reg1, add carry to result, store result in reg1
0110 010x reg(2) imm(8)	ADI reg, imm8	add data byte to register
0110 011x reg(4) xxxxx	INC reg	Increment register by one
0110 100x reg(4) xxxxx	INCA	Increment ACC by one
0111 000x reg(4) reg(4)	SUB reg1, reg2	Subtract contents of reg2 to reg1
0111 001x reg(4) reg(4)	SBB reg1, reg2	Subtract contents of reg2 to reg1, Subtract carry from result
0111 10 reg(2) imm(8)	SBI reg, imm8	Subtract data byte from register
0111 11 reg(4) xxxxx	DEC reg	Decrement register by one
100 00 reg(3) imm(8)	ANI reg, imm8	AND data byte with register
100 01 reg(3) imm(8)	ORI reg, imm8	OR data byte with register
100 10 reg(3) imm(8)	XRI reg, imm8	XOR data byte with register
100 110 reg(2) imm(8)	CPI reg, imm8	compare data byte with register
100 111 reg(4) reg(4)	CMP reg1, reg2	compare content of reg1 with reg2
1010 00xx reg(4) reg(4)	AND reg1, reg2	AND contents of reg1 with reg2
1010 01xx reg(4) reg(4)	OR reg1, reg2	OR contents of reg1 with reg2
1010 10xx reg(4) reg(4)	XOR reg1, reg2	XOR contents of reg1 with reg2
1010 11xx reg(4) xxxxx	NOT reg	Complement content of register
1011 000x xxxx reg(4)	IN reg, port0	Input data from port0 to register
1011 001x xxxx reg(4)	IN reg, port1	Input data from port1 to register
1011 010x xxxx reg(4)	OUT port0, reg	Output content of register to port0
1011 011x xxxx reg(4)	OUT port1, reg	Output content of register to port1
1100 00 offset(10)	LJMP offset	Long Jump (within 1K)
1100 01 xx offset(8)	SJMP offset	Short Jump (up to 246)
1100 1000 offset(8)	JZ offset	Jump if Zero flag is set
1100 1001 offset(8)	JNZ offset	Jump if Zero flag is reset
1100 1010 offset(8)	JC offset	Jump if Carry flag is set
1100 1011 offset(8)	JNC offset	Jump if Carry flag is reset

1100 1100 offset(8)	JPS offset	Jump if parity flag is set
1100 1101 offset(8)	JNP offset	Jump if parity flag is reset
1100 1110 offset(8)	JOV offset	Jump if overflow flag is set
1100 1111 offset(8)	JNO offset	Jump if overflow flag is reset
1101 000 xxxx xxxxx	INT0	Interrupt on signal on pin INT0
1101 001 xxxx xxxxx	INT1	Interrupt on signal on pin INT1
1101 010 xxxx xxxxx	INT2	Interrupt on signal on pin INT2
1101 011 xxxx xxxxx	INT3	Interrupt on signal on pin INT3
1101 100 xxxx xxxxx	INT4	Interrupt on signal on pin INT4
1101 101 xxxx xxxxx	IRET	Return from Interrupt
1110 0000 imm(8)	CALL address	Call subroutine at given address
1110 0001 xxx xxxxx	RET	Return from subroutine call
1110 0100 imm(8)	CC	Call if Carry flag is set
1110 0101 xxxxxxxx	RC	Return if Carry flag is set
1110 0110 imm(8)	CNC	Call if Carry flag is reset
1110 0111 xxxxxxxx	RNC	Return if Carry flag is reset
1110 1000 imm(8)	CZ	Call if Zero flag is set
1110 1001 xxxxxxxx	RZ	Return if Zero flag is set
1110 1010 imm(8)	CNZ	Call if Zero flag is reset
1110 1011 xxxxxxxx	RNZ	Return if Zero flag is reset
1110 1100 imm(8)	COV	Call if overflow flag is set
1110 1101 xxxxxxxx	ROV	Return if overflow flag is set
1110 1110 imm(8)	CNO	Call if overflow flag is reset
1110 1111 xxxxxxxx	RNO	Return if overflow flag is reset

APPENDIX B: SIMULATION RESULTS



Simulation of program with JMP Instruction (active HDL 6.3)

Active HDL 6.3 (risc\_2\_jmp.risc) C:\My Design\hls\2\_jmp\hls\src\Waveform Editor 1.rwl

File Edit Search View Workspace Design Simulation Waveform Tools Window Help

100 ns

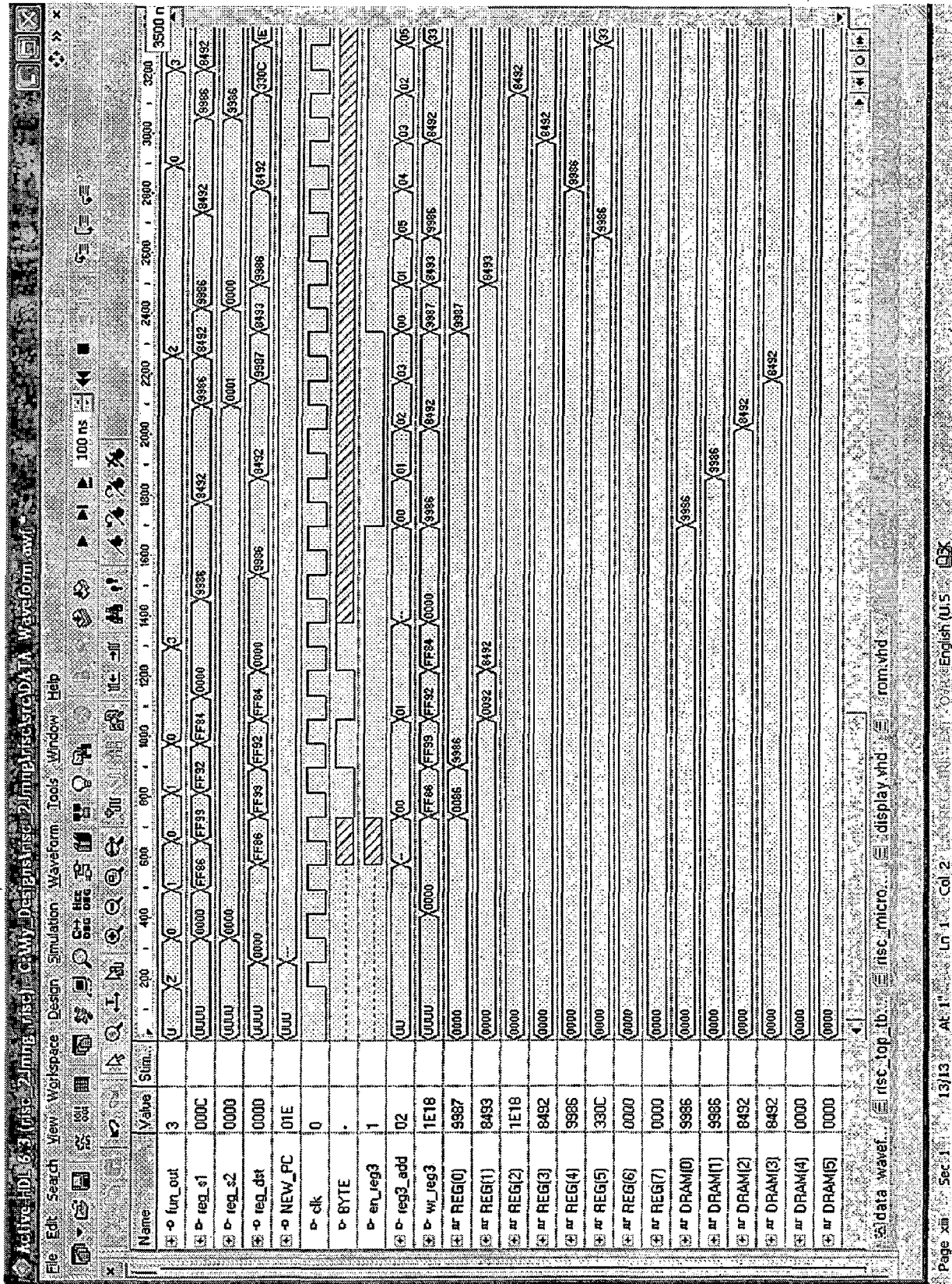
Name	Value	Sim...
nr reset	0	Z
nr int_vect	00	<= 0
nr OUTLED	0000	UUUU 0000 0012 0009 0024 0046 00FD 0838 0000 0019 0018 005F 0000 00
nr result	0000	UUUU 0000 0012 0009 0024 0048 00FD 0838 0000 0016 0049 0010 005F 0000 00
clk	1	
hclk	0	
lbrx	0	
XPC		UUU 418 009
FPC	009	ZZZ 000 001 002 003 004 005 006 007 008 009 418 415 416 417 418 418 008
FIR	6051	ZZZZ 218 309 324 3248 359D 8022 7041 6049 6001 6061 2016 2649 2410 6006 5400 4001 4058 6001
SRC1	00	UU 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
SRC2	01	UU 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
DST	00	UU 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
REG1		UUUU 0000 0012 0009 0024 0048 00FD 0838 0000 0016 0049 0010 005F 0000 00
REG2	0936	UUUU 0000 0012 0009 0024 0048 00FD 0838 0000 0016 0049 0010 005F 0000 00
REG3	ZZZZ	UUUU 0000 0012 0009 0024 0048 00FD 0838 0000 0016 0049 0010 005F 0000 00
TEMP1	0000	UUUU 0000 0012 0009 0024 0048 00FD 0838 0000 0016 0049 0010 005F 0000 00
TEMP2	0000	UUUU 0000 0012 0009 0024 0048 00FD 0838 0000 0016 0049 0010 005F 0000 00
DPC	008	UUU ZZZ 000 001 002 003 004 005 006 007 008 009 010 011 012 013 014 015 016 017 018 019 020 021 022 023 024 025 026 027 028 029 030 031 032 033 034 035 036 037 038 039 040 041 042 043 044 045 046 047 048 049 050 051 052 053 054 055 056 057 058 059 060 061 062 063 064 065 066 067 068 069 070 071 072 073 074 075 076 077 078 079 080 081 082 083 084 085 086 087 088 089 090 091 092 093 094 095 096 097 098 099 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000
reg_s1	0000	UUUU 0000 0012 0009 0024 0048 00FD 0838 0000 0016 0049 0010 005F 0000 00
reg_s2	0000	UUUU 0000 0012 0009 0024 0048 00FD 0838 0000 0016 0049 0010 005F 0000 00
reg_dst	0000	UUUU 0000 0012 0009 0024 0048 00FD 0838 0000 0016 0049 0010 005F 0000 00

Waveform e risc\_top\_ib risc\_micro display\_vhd

Page: 1 Sec: 1 11/11 English (U.S.)

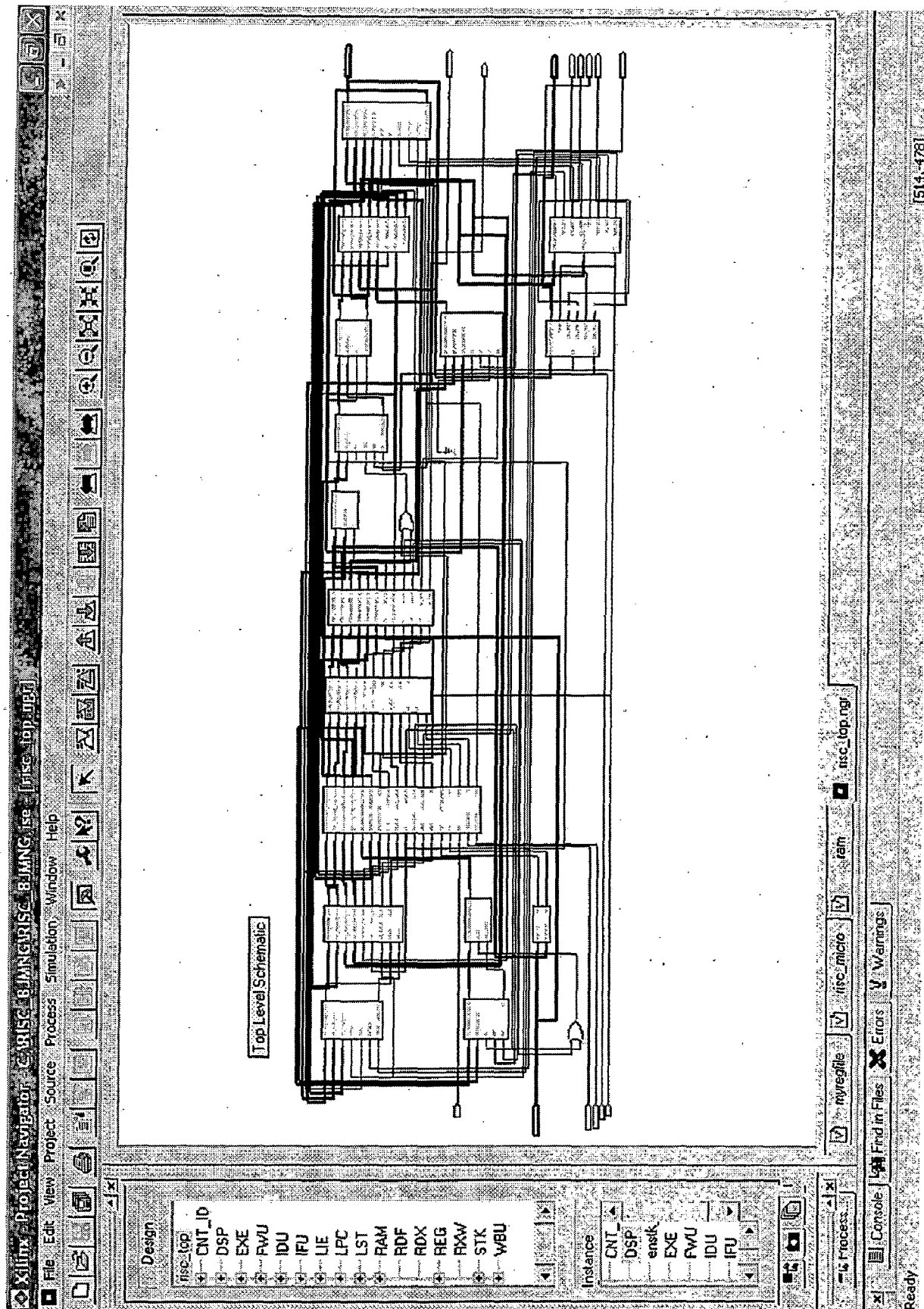
Simulation of program with CALL instruction (active HDL 6.3)





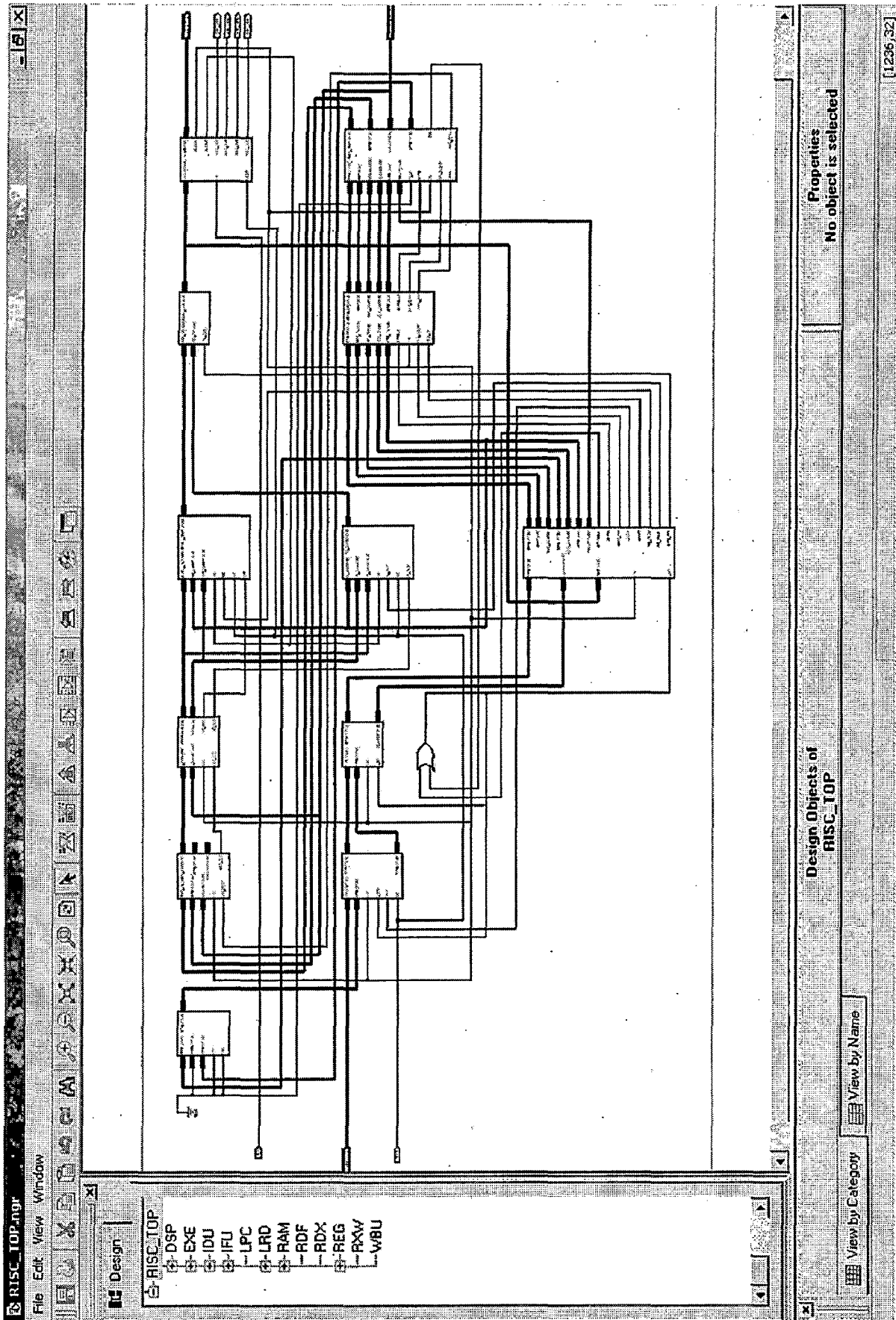
Simulation of program with LOAD, STORE instructions (continued....)

# APPENDIX C: SYNTHESIS RESULTS



Synthesized RTL description of RISC microcontroller in xilinx ise7.1i





Synthesized RTL description of RISC microcontroller in xilinx ise 8.1i

## SYNTHESIS REPORT

Release 8.1i - xst I.24 Copyright (c) 1995-2005 Xilinx, Inc. All rights reserved.

--> Reading design: RISC\_TOP.prj

---

---

### \* Synthesis Options Summary \*

---

---

#### ---- Source Parameters

Input File Name : "RISC\_TOP.prj"  
Input Format : mixed  
Ignore Synthesis Constraint File : NO

#### ---- Target Parameters

Output File Name : "RISC\_TOP"  
Output Format : NGC  
Target Device : xc2s200-5-pq208

#### ---- Source Options

Top Module Name : RISC\_TOP  
Automatic FSM Extraction : YES  
FSM Encoding Algorithm : Auto  
FSM Style : lut  
RAM Extraction : Yes  
RAM Style : Auto  
ROM Extraction : Yes  
Mux Style : Auto  
Decoder Extraction : YES  
Priority Encoder Extraction : YES  
Shift Register Extraction : YES  
Logical Shifter Extraction : YES  
XOR Collapsing : YES  
ROM Style : Auto  
Mux Extraction : YES  
Resource Sharing : YES  
Multiplier Style : lut  
Automatic Register Balancing : No

#### ---- Target Options

Add IO Buffers : YES  
Global Maximum Fanout : 100  
Add Generic Clock Buffer(BUFG): 4  
Register Duplication : YES  
Slice Packing : YES  
Pack IO Registers into IOBs : auto

Equivalent register Removal : YES

---- General Options

Optimization Goal : Speed  
Optimization Effort : 1  
Keep Hierarchy : NO  
RTL Output : Yes  
Global Optimization : AllClockNets  
Write Timing Constraints : NO  
Hierarchy Separator : /  
Bus Delimiter : <  
Case Specifier : maintain  
Slice Utilization Ratio : 100  
Slice Utilization Ratio Delta : 5

---- Other Options

lso : RISC\_TOP.lso  
Read Cores : YES  
cross\_clock\_analysis : NO  
verilog2001 : YES  
safe\_implementation : No  
Optimize Instantiated Primitives : NO  
tristate2logic : Yes  
use\_clock\_enable : Yes  
use\_sync\_set : Yes  
use\_sync\_reset : Yes

=====

\*

**HDL Synthesis**

\*

=====

Synthesizing Unit <ALU>.

Related source file is "E:/thesis/RISC\_16JNIGHT/alu.vhd".

Summary:

inferred 62 D-type flip-flop(s).  
inferred 2 Adder/Subtractor(s).  
inferred 2 Comparator(s).  
inferred 33 Multiplexer(s).  
inferred 77 Tristate(s).

Unit <ALU> synthesized.

Synthesizing Unit <forward>.

Related source file is "E:/thesis/RISC\_16JNIGHT/forward.vhd"..

Summary: inferred 16 Multiplexer(s).  
Unit <forward> synthesized.

Synthesizing Unit <hazzard>.

Related source file is "E:/thesis/RISC\_16JNIGHT/hazard.vhd".

Summary: inferred 2 Comparator(s).  
inferred 16 Tristate(s).

Unit <hazzard> synthesized.

Synthesizing Unit <CONTROL\_UNIT>.

Related source file is "E:/thesis/RISC\_16JNIGHT/control\_unit.vhd".

Summary: inferred 5 Tristate(s).

Unit <CONTROL\_UNIT> synthesized.

Synthesizing Unit <decoder>.

Related source file is "E:/thesis/RISC\_16JNIGHT/decoder.vhd".

Summary: inferred 7 Tristate(s).

Unit <decoder> synthesized.

Synthesizing Unit <rom>.

Related source file is "E:/thesis/RISC\_16JNIGHT/rom.vhd".

Found 32x16-bit ROM for signal <instn>.

Summary: inferred 1 ROM(s).

Unit <rom> synthesized.

Synthesizing Unit <PC\_REG>.

Related source file is "E:/thesis/RISC\_16JNIGHT/pc\_reg.vhd".

Summary: inferred 12 D-type flip-flop(s).

inferred 1 Adder/Subtractor(s).

Unit <PC\_REG> synthesized.

Synthesizing Unit <data\_ram>.

Related source file is "E:/thesis/RISC\_16JNIGHT/ram.vhd".

Found finite state machine <FSM\_0> for signal <PS>.

Summary: inferred 1 Finite State Machine(s).

inferred 5 Tristate(s).

Unit <data\_ram> synthesized.

Synthesizing Unit <myregfile>.

Related source file is "E:/thesis/RISC\_16JNIGHT/myregfile.vhd".

Found finite state machine <FSM\_1> for signal <PS>.

Summary: inferred 1 Finite State Machine(s).

inferred 5 Tristate(s).  
Unit <myregfile> synthesized.

Synthesizing Unit <REG\_RD>.  
Related source file is "E:/thesis/RISC\_16JNIGHT/LATCH\_RD.vhd".  
Unit <REG\_RD> synthesized.

Synthesizing Unit <LATCH\_NPC>.  
Related source file is "E:/thesis/RISC\_16JNIGHT/latch.vhd".  
Summary: inferred 12 Multiplexer(s).  
Unit <LATCH\_NPC> synthesized.

Synthesizing Unit <WRITE\_BACK>.  
Related source file is "E:/thesis/RISC\_16JNIGHT/write\_back.vhd".  
Summary: inferred 22 D-type flip-flop(s).  
Unit <WRITE\_BACK> synthesized.

Synthesizing Unit <REG\_EX\_WB>.  
Related source file is "E:/thesis/RISC\_16JNIGHT/reg\_ex\_wb.vhd".  
Summary: inferred 33 D-type flip-flop(s).  
Unit <REG\_EX\_WB> synthesized.

Synthesizing Unit <EXECUTE>.  
Related source file is "E:/thesis/RISC\_16JNIGHT/execute.vhd".  
Summary: inferred 17 D-type flip-flop(s).  
inferred 17 Tristate(s).  
Unit <EXECUTE> synthesized.

Synthesizing Unit <REG\_ID\_EX>.  
Related source file is "E:/thesis/RISC\_16JNIGHT/reg\_id\_ex.vhd".  
Summary: inferred 30 D-type flip-flop(s).  
Unit <REG\_ID\_EX> synthesized.

Synthesizing Unit <DECODE>.  
Related source file is "E:/thesis/RISC\_16JNIGHT/decode.vhd".  
Summary: inferred 46 D-type flip-flop(s).  
inferred 43 Tristate(s).  
Unit <DECODE> synthesized.

Synthesizing Unit <REG\_F\_ID>.  
Related source file is "E:/thesis/RISC\_16JNIGHT/reg\_f\_id.vhd".  
Summary: inferred 30 D-type flip-flop(s).  
inferred 28 Tristate(s).  
Unit <REG\_F\_ID> synthesized.

Synthesizing Unit <fetch>.

Related source file is "E:/thesis/RISC\_16JNIGHT/fetch.vhd".

Summary: inferred 28 D-type flip-flop(s).  
inferred 28 Tristate(s).

Unit <fetch> synthesized.

Synthesizing Unit <DISPLAY>.

Related source file is "E:/thesis/RISC\_16JNIGHT/display.vhd".

Summary:

inferred 4 ROM(s).  
inferred 2 Counter(s).  
inferred 12 Adder/Subtractor(s).  
inferred 6 Comparator(s).  
inferred 17 Multiplexer(s).

Unit <DISPLAY> synthesized.

Synthesizing Unit <RISC\_TOP>.

Related source file is "E:/thesis/RISC\_16JNIGHT/risc\_micro.vhd".

Unit <RISC\_TOP> synthesized.

### Advanced HDL Synthesis

Analyzing FSM <FSM\_1> for best encoding.

Optimizing FSM <REG/PS> on signal <PS[1:3]> with gray encoding.

Analyzing FSM <FSM\_0> for best encoding.

Optimizing FSM <RAM/PS> on signal <PS[1:3]> with gray encoding.

Reading module "mymemory.ngo" ( "mymemory.ngo" unchanged since last run )...

Loading core <mymemory> for timing and area information for instance <REG>.

Loading core <mymemory> for timing and area information for instance <REG>.

### Advanced HDL Synthesis Report

#### Macro Statistics

# FSMs	: 2
# ROMs	: 5
16x8-bit ROM	: 4
32x16-bit ROM	: 1
# Adders/Subtractors	: 15
12-bit adder	: 3
32-bit adder	: 12

```

# Counters          : 2
  31-bit up counter : 1
  32-bit up counter : 1
# Registers         : 286
  Flip-Flops       : 286
# Latches           : 26
  1-bit latch      : 21
  12-bit latch     : 1
  16-bit latch     : 3
  8-bit latch      : 1
# Comparators       : 10
  16-bit comparator equal : 1
  16-bit comparator less  : 1
  31-bit comparator greatequal : 3
  31-bit comparator less  : 2
  31-bit comparator lessequal : 1
  4-bit comparator equal  : 2
# Multiplexers      : 23
  1-bit 4-to-1 multiplexer : 17
  1-bit 8-to-1 multiplexer : 1
  12-bit 4-to-1 multiplexer : 1
  16-bit 4-to-1 multiplexer : 1
  16-bit 8-to-1 multiplexer : 1
  8-bit 4-to-1 multiplexer  : 1
  8-bit 8-to-1 multiplexer  : 1
# Xors              : 54
  1-bit xor2        : 53
  16-bit xor2       : 1

```

Mapping all equations...

Building and optimizing final netlist ...

Found area constraint ratio of 100 (+ 5) on block RISC\_TOP, actual ratio is 25.

---



---

```

*                               *
  Final Report
*                               *

```

---



---

#### Final Results

```

RTL Top Level Output File Name : RISC_TOP.ngr
Top Level Output File Name     : RISC_TOP
Output Format                   : NGC
Optimization Goal               : Speed
Keep Hierarchy                  : NO
Design Statistics
# IOs                           : 35

```

## Cell Usage :

# BELS	: 1361
# GND	: 3
# INV	: 4
# LUT1	: 46
# LUT2	: 105
# LUT2_D	: 8
# LUT2_L	: 14
# LUT3	: 122
# LUT3_D	: 9
# LUT3_L	: 62
# LUT4	: 454
# LUT4_D	: 64
# LUT4_L	: 210
# MUXCY	: 130
# MUXF5	: 62
# MUXF6	: 1
# VCC	: 1
# XORCY	: 66

# FlipFlops/Latches : 392

# FD	: 96
# FD_1	: 12
# FDC	: 19
# FDC_1	: 3
# FDE	: 25
# FDR	: 33
# FDS	: 52
# FDS_1	: 76
# LD	: 48
# LD_1	: 16
# LDC	: 2
# LDCP	: 10
# RAMS	: 2
# RAMB4_S16	: 2
# Clock Buffers	: 2
# BUFG	: 1
# BUFGP	: 1
# IO Buffers	: 34
# IBUF	: 6
# OBUF	: 28



---

---

**Device utilization summary:**

Selected Device : 2s200pq208-5

Number of Slices:	589	out of	2352	25%
Number of Slice Flip Flops:	381	out of	4704	8%
Number of 4 input LUTs:	1094	out of	4704	23%
Number of bonded IOBs:	35	out of	144	24%
IOB Flip Flops: 11				
Number of BRAMs:	2	out of	14	14%
Number of GCLKs:	2	out of	4	50%

---

---

**TIMING REPORT****Clock Information:**

Clock Signal	Clock buffer(FF name)	Load
DSP/div_clk_31	BUFG	323
DSP/div_clk_0	NONE	8
REG/DATA_EN(REG/PS_Out01:O)	NONE(*) (REG/rd_data_3)	16
hltout(IDU/CNT/_n00051:O)	NONE(*) (IFU/hlt)	1
DSP/_n0043(DSP/Mcompar__n0043_norcy_rm_5:O)	NONE(*) (DSP/LCD_1)	11
clk	BUFGP	35

(\*) These 3 clock signal(s) are generated by combinatorial logic, and XST is not able to identify which are the primary clock signals.

Please use the CLOCK\_SIGNAL constraint to specify the clock signal(s) generated by combinatorial logic.

**Timing Summary:**

Speed Grade: -5

Minimum period: 40.716ns (Maximum Frequency: 24.560MHz)

Minimum input arrival time before clock: 7.873ns

Maximum output required time after clock: 8.329ns

Maximum combinational path delay: No path found

Total memory usage is 127580 kilobytes

Number of errors : 0 ( 0 filtered)

Number of warnings : 17 ( 0 filtered)

Number of infos : 20 ( 0 filtered)