

8-Bit RISC Processor Design using Verilog HDL on FPGA

A

Dissertation

Submitted in the fulfilment of the requirements

For the award of degree

Of

MASTER OF TECHNOLOGY

In

Embedded System Design

By

JIKKU JEEMON

(Roll No. 3146510)

Under the Guidance of

Prof. A. K. GUPTA



SCHOOL OF VLSI DESIGN AND EMBEDDED SYSTEMS

NATIONAL INSTITUTE OF TECHNOLOGY

KURUKSHETRA-136119

SESSION 2014-2016

C
MT 621.38173
JEE-16





SCHOOL OF VLSI DESIGN AND EMBEDDED SYSTEMS

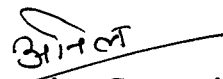
NATIONAL INSTITUTE OF TECHNOLOGY

KURUKSHETRA – 136119

CERTIFICATE

This is to certify that the dissertation entitled “***8-Bit RISC Processor Design using Verilog HDL on FPGA***” is the authentic record of work done by **Jikku Jeemon** under my guidance and supervision. This dissertation is being submitted to the ***National Institute of Technology, Kurukshetra*** towards the fulfilment of the requirements for the award of degree of ***Master of Technology in Embedded System Design***.

Date: 1-6-2016


Dissertation Supervisor:

Dr. A. K. Gupta

Professor

School of VLSI Design and Embedded Systems

National Institute of Technology, Kurukshetra

CANDIDATE'S DECLARATION

I hereby declare that the work being presented in this dissertation entitled "*8-Bit RISC Processor Design using Verilog HDL on FPGA*", submitted towards the fulfilment of the requirements for the award of degree, Master of Technology in Embedded System Design to the School of VLSI Design and Embedded Systems, National Institute of Technology, Kurukshetra, is an authentic record of my work carried out from July 2015 to June 2016, under the guidance of Prof. A. K. GUPTA, School of VLSI Design and Embedded Systems, National Institute of Technology, Kurukshetra.

I have not submitted the matter embodied in the dissertation for the award of any other degree.



Jikku Jeemon

Roll No. 3146510

School of VLSI Design and Embedded Systems

Date:

ACKNOWLEDGEMENT

I would like to express my deep gratitude and appreciation to all the people who have helped and supported me in the process of dissertation. Without their help and support, I would not have been able to reach this level of satisfaction with what I have learnt and accomplished during my Master's dissertation.

First and foremost, I would like to express my deep sense of respect and gratitude towards my supervisor, Dr. A. K. Gupta, Professor, School of VLSI Design and Embedded Systems, NIT Kurukshetra, for giving me opportunity to do my master's dissertation under his guidance. I am very thankful for his endless support, motivation, patience, encouragement and guidance during the study. His professional knowledge and faith in me were very important and gave me the strength to conclude this work.

I express my grateful thanks to Dr. R. K. Sharma, Professor in School of VLSI Design and Embedded Systems for providing 24 hour lab facility.

I would also like to thanks my friends for sharing their valuable thoughts and knowledge, which motivated me to do better.

Finally, none of this would have been possible without incredible support of my parents. They were always supporting me and encouraging me with their best wishes.



Jikku Jeemon

Roll No. 3146510

School of VLSI Design and Embedded Systems

LIST OF FIGURES

Figure 2.1 Architecture of 8-bit RISC processor [17]	6
Figure 2.2 Architecture of 8-bit RISC processor [18]	7
Figure 2.3 Computer architecture of 16-bit RISC processor [19]	8
Figure 3.1 8-bit RISC processor architecture	10
Figure 3.2 pipelining architecture.....	11
Figure 3.3 Block diagram of PCU	12
Figure 3.4 Block diagram of IM	13
Figure 3.5 Block diagram of Control unit	14
Figure 3.6 Block diagram of DM	15
Figure 3.7 Block diagram of Accumulator	16
Figure 3.8 Block diagram of Register set	18
Figure 3.9 Block Diagram of ALU unit	19
Figure 3.10 Block diagram of AND block	19
Figure 3.11 Block diagram of OR block	20
Figure 3.12 Block diagram of XOR block	20
Figure 3.13 Block diagram of ADD block	20
Figure 3.14 Block diagram of SUB block	21
Figure 3.15 Block diagram of Flag register.....	21
Figure 3.16 4-bit Flag register	22
Figure 3.17 Block diagram of I/O Module	22
Figure 3.18 INTCON register.....	23
Figure 3.19 Block diagram of Interrupt Module	23
Figure 3.20 UART serial communication protocol	24

Figure 3.21 Block diagram of Serial Module	24
Figure 4.1 Simulation of Program Counter Unit	45
Figure 4.2 Simulation of Instruction Memory	46
Figure 4.3 Simulation of Data Memory	46
Figure 4.4 Simulation of accumulator	47
Figure 4.5 Simulation of Register set	48
Figure 4.6 Simulation of ALU unit	48
Figure 4.7 Simulation of interrupt module part 1	49
Figure 4.8 Simulation of interrupt module part 2	49
Figure 4.9 Simulation of serial module	50
Figure 4.10 Simulation of Control unit	51
Figure 4.11 Simulation results of proposed processor part I	54
Figure 4.12 Simulation results of proposed processor part II	54
Figure 4.13 Simulation results of proposed processor part III	55

LIST OF TABLES

Table 3.1 Instruction set	27
Table 4.1 Twenty-instruction program for simulation	53
Table 4.2 Device utilization of the Spartan-3E Starter kit FPGA board	54

LIST OF ABBREVIATIONS

Abbreviations	Meaning
ALU	Arithmetic Logic Unit
B	Borrow flag
C	Carry flag
DM	Data Memory
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IM	Instruction Memory
IR	Instruction Register
ISA	Instruction Set Architecture
I/O	Input/Output
LPU	Low Power Unit
P	Parity flag
PC	Program Counter
PCR	Program Counter Register
PCS	Program Counter Save
PCU	Program Counter Unit
RISC	Reduced Instruction Set Computer
UART	Universal Asynchronous Receiver Transmitter
Z	Zero flag

ABSTRACT

RISC is a design technique used to reduce the amount of area required, complexity of instruction set, instruction cycle and cost during the implementation of the design. This dissertation presents an 8-bit RISC processor design using Verilog Hardware Description Language (HDL) on FPGA board. The proposed processor is designed using Harvard architecture, having separate instruction and data memory. The salient feature of proposed processor is pipelining, used for improving performance, such that on every clock cycle one instruction will be executed. Another important feature is that instruction set contains only 34 instructions, which is very simple, easy to learn and compact. The proposed processor has 8-bit ALU, Two 8-bit I/O ports, serial-in/serial-out ports, Eight 8-bit general-purpose registers, 4-bit flag register and priority based three interrupts. In RTL coding one can reduce the dynamic power by using clock-gating technique, is used for specific modules that will be clocked only when corresponding control signals are enabled. The proposed processor is physically verified on Xilinx Spartan 3E Starter Board FPGA at 25MHz clock frequency, which will work on 2.5 voltage supply.

TABLE OF CONTENTS

CERTIFICATE	I
CANDIDATE'S DECLARATION	II
ACKNOWLEDGEMENT	III
LIST OF FIGURES.....	IV
LIST OF TABLES.....	VI
LIST OF ABBREVIATIONS	VII
ABSTRACT	VIII
Chapter 1 INTRODUCTION	1
1.1 Importance of 8-bit RISC processor	2
1.2 This Work	2
Chapter 2 LITERATURE REVIEW	4
2.1 Brief History of 8-bit processor	4
2.2 Related Works	5
Chapter 3 DESIGN OF 8-BIT RISC PROCESSOR.....	9
3.1 Architecture of 8-bit RISC processor	9
3.2 Description of Functional Modules	11
3.2.1 Program Counter Unit (PCU).....	11
3.2.2 Instruction Memory (IM).....	12
3.2.3 Control Unit.....	13
3.2.4 Data Memory (DM).....	15
3.2.5 Accumulator (A).....	16
3.2.6 Register Set.....	17
3.2.8 ALU unit.....	18

3.2.8 Flag Register	21
3.2.9 I/O Module	22
3.2.10 Interrupt Module.....	22
3.2.11 Serial Module	24
3.3 List of control signals, I/O bus and Flags.....	25
3.4 Instruction Set Architecture.....	26
3.5 Source Code.....	28
Chapter 4 RESULTS AND DISCUSSIONS	45
Chapter 5 CONCLUSION.....	56
REFERENCES	57
APPENDIX A.....	59
APPENDIX B.....	64

Chapter 1

INTRODUCTION

With the rapid development of the silicon technology and fall in the cost of the integrated circuit, the usage of RISC processor is increasing extensively in every field. The architecture principle Reduced Instruction Set Computer is commonly known as RISC. RISC processors allow special load and store operations to access memory. The other operations are performed on register-to-register basis. This feature makes instruction set design more clear and simple as it allows execution of instructions at one-instruction-per-cycle rate. Simple and transparent addressing modes allow fast address computation of operands. Thus, the usage of RISC architecture reduces amount of area required, complexity of instruction set, instruction cycle and cost of the hardware ([1] and [2]). RISC processor's range of application includes signal processing, convolution application, commercial data processing, used in supercomputers such as the K computer, smart phones, tablets and real-time embedded systems. Pipelining, a typical feature in RISC processors, is an implementation technique in which multiple operations are performed at same time. It is a form of parallelism at instruction-level using a single processor, which significantly improves the performance of the processor. Pipelining increases instruction throughput but does not reduce instruction latency, which is the time to complete a single instruction from start to end [3].

The role of reconfigurable processor in embedded system design has increased greatly during the past decades. Due to the advancement in Field Programmable Gate Array (FPGA), we have reached a point where architecture of processor can be modified by programming [4]. Clock power is a vital component of overall dynamic power consumption, which should be minimized in design to reduce power consumption. One of the methods to reduce clock power is clock-gating (ANDing) ([5] and [6]), which dynamically terminates the clock signals in unused modules of the total hardware. This avoids the unnecessary power dissipation cropped up by charging and discharging of clock signal at unused gate.

Asynchronous serial communication has advantages of high reliability, simple because it does not require synchronization both communicating sides and cheap because it requires less hardware, hence is extensively used as a mode of communication between computer and peripherals. Universal Asynchronous Receiver Transmitter (UART) is a type of serial communication protocol, which is mostly used for short-distance, low speed, low-cost data exchange between computer and peripherals. Asynchronous serial communication is usually implemented by UART, which allows full-duplex communication in serial link ([7] and [8]).

1.1 Importance of 8-bit RISC processor

Modern electronic devices such as desktops, laptops, notebooks, or tablets are using 32-bit and 64-bit processors. 16-bit processors can be found in larger systems such as traffic lights, systems controlling power plants and factory controllers. 16-bit embedded processors are also used in consumer electronics including video game consoles, DVD players, digital cameras, scanners and printers. Several household appliances including microwave ovens and washing machines also uses 16-bit embedded processors.

However, 8-bit computers are used extensively as controllers for simple computational tasks. According to the divide and conquer principle, a common personal computer is divided into smaller ones (commonly 8-bit) which share information with the main computer (32 or 64-bit). An 8-bit processor generally handles the drivers for almost every component card inside a computer. 8-bit type processors are extensively used in home appliances and industrial specific systems [9]. 8-bit or 16-bit processors are better than 32-bit processors for system on a chip and microcontrollers that require extremely low power for functioning and survival.

1.2 This Work

The objective of this dissertation is to design an 8-bit RISC processor and implement it on Spartan 3E Starter kit FPGA using Verilog Hardware Description Language (HDL). The processor is designed using Harvard architecture, having separate instruction and data memory. Its most important feature is that its instruction set is very simple, contains only 34 instructions, which is easy to learn. Another important feature is pipelining, used for improving performance, such that on every clock cycle one instruction will be executed. It is planned to design an 8-bit RISC processor having the following main features.

- Harvard Architecture
 - 256 K x 16 Instruction Memory
 - 4 K x 8 Data Memory
- 8-bit system data bus
- 3 Interrupts
- Eight 8-bit General Purpose Registers
- clock frequency = 25MHz
- Two 8-bit I/O Ports
- Serial-in and serial-out ports
- 2.5V voltage supply
- Clock gating for power reduction

The dissertation is organized in five chapters. A brief outline of each chapter is described below:

Chapter 2 presents previous work done related to the proposed processor as available in literature.

Chapter 3 discusses the design and architecture of proposed processor, description of functional modules and instruction set architecture.

Chapter 4 describes simulation results. The simulation of the proposed 8-bit RISC processor is carried out in Xilinx's simulation tool ISim.

Chapter 5 concludes the work with brief summary of this dissertation.

Chapter 2

LITERATURE REVIEW

This chapter presents the background for this work. Section 2.1 of this chapter presents brief history of 8-bit processor. In Section 2.2, the related works are described.

2.1 Brief History of 8-bit processor

The evolution of the 8-bit processors is a history of the advancement of semiconductor technology from first transistors, to the quantum leap of multiple transistors on a chip, the integrated circuit (IC). 8-bit processors operate on 8-bit wide data, and normally have a 16-bit address. They require backing from subsidiary chips such as memory and I/O devices. The microcontrollers, as opposed to the CPU's, incorporate memory and I/O on the same chip.

The Intel's 8008 was the first 8-bit monolithic microprocessor to market in April 1972. It operated up to 0.8 MHz, had 3,500 transistors in a PMOS technology, with 10-micron line width. There were 48 instructions. The address space was 16 kilobytes, but direct addressing was not supported [10].

The Intel's 8080 was a great improvement over the prior 8008 chip, incorporating many features into the chip that required the use of external hardware with the 8008. The 8080 a superset of 8008, an NMOS design, with 8-bit words and a 16-bit address bus was released in 1974. It required ± 5 volts and +12 volts and had six general-purpose registers and accumulator. It used 6,000 transistors, maximum rating of 0.8 watts, had 48 instructions and operated at 2 MHz [11].

The Intel's 8085 was an advanced version of the 8080 and featured simplified hardware that needed only a single +5V supply, it included a clock-generator and bus-controller circuits on the chip and was introduced in 1976. It was binary compatible with the 8080, but required less supporting hardware, allowing simpler and less expensive microcomputer systems. The 8085 used a multiplexed data/address bus to reduce chip pin-out. This required external de-multiplexing of the 16-bit address and the 8-bit data. It had 48 basic instructions, maximum rating of 1.5 watts, operated up to 6 MHz, featured serial in/out port and supported four vectored interrupts [12].

Reduced instruction set computer (RISC) is a CPU design technique based on the perception that a simplified instruction set yields higher performance when linked with a microprocessor architecture which can perform those instructions using fewer clock cycles per instruction. The term RISC was coined by David Patterson of the Berkeley RISC project, although similar concepts had appeared before [13].

The CDC 6600 designed by Seymour Cray in 1964 used a load/store architecture with only two addressing modes (register-register, and register-immediate constant) and 74 opcodes, with the basic clock cycle/instruction issue rate being 10 times faster than the memory access time [14].

IBM 801 is the first recognized RISC system, which was started in 1975 by John Cocke and completed in 1980. The 801 was eventually produced in a single-chip form as the ROMP (Research Office products Micro Processor) in 1981. It was designed for small tasks and was used in the IBM RT-PC in 1986, which turned out to be a commercial failure. However, the 801 inspired several research projects, including new ones at IBM that would eventually lead to the IBM POWER instruction set architecture [15].

The most public RISC designs, however, were the results of university research programs run with funding from the DARPA VLSI Program. The Berkeley RISC project started in 1980 under the direction of David Patterson and Carlo H. Sequin. Berkeley RISC was based on gaining performance by pipelining and an aggressive use of a technique known as register windowing. Berkeley RISC project delivered the RISC-I processor in 1982 consisting of only 44,420 transistors (compared with averages of about 100,000 in newer CISC designs of the era). RISC-I had only 32 instructions, and yet completely outperformed any other single-chip design. They followed this up with the 40,760 transistor, 39 instruction RISC-II in 1983, which ran over three times as fast as RISC-I [16].

2.2 Related Works

The design of an 8-bit RISC processor comprises of control unit, general-purpose registers, barrel shifter, arithmetic and logical unit, universal shift register and accumulator has been reported [17]. The architecture of this 8-bit RISC processor is shown in Figure 2.1. Control unit follows instruction cycle of 3 stages fetch, decode and execute cycle. According to the instruction fetched, the control unit generate signals to

decode and execute the instruction. The architecture supports 16 instructions for arithmetic, logical, shifting and rotational operations. Instruction and data are fetched sequentially in order to reduce the latency in the machine cycle. Pipeline structure has been incorporated for fetch, decode and execute. This pipeline structure helps in enhancing the speed of operation. This processor can be used for mathematical computation in portable calculators as well as in gaming tool kit.

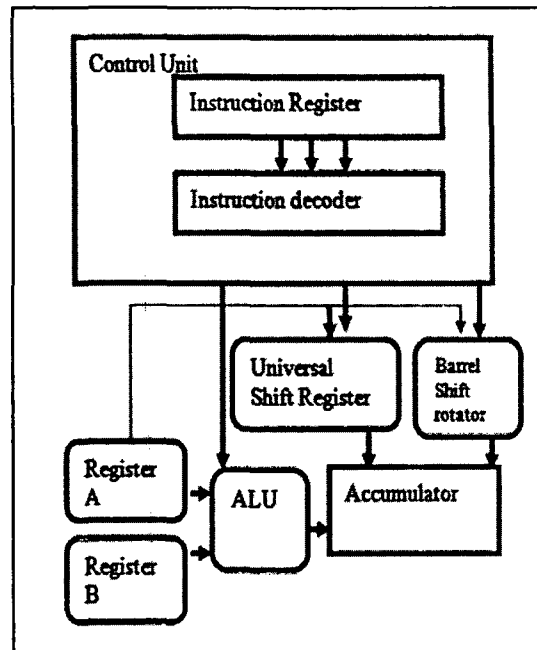


Figure 2.1 Architecture of 8-bit RISC processor [17]

Another related work [18], describes an 8-bit RISC processor that consists of arithmetic logic unit, control unit, shifter and rotator. The architecture of this processor is shown in Figure 2.2. The processor is designed with load/store (Von Neumann) architecture, one shared memory for instructions (program) and data with one data bus and one address bus between processor and memory. Instruction and data are fetched in sequential order so that the latency incurred between the machine cycles can be reduced. In this design, most instructions are of uniform length and similar structure, arithmetic operations are restricted to CPU registers and only load and store instructions access memory. Three stages of pipelining have been incorporated in the design, which increases the speed of operation. This processor can be used as a systolic core to perform mathematical computations like solving polynomial and differential equations.

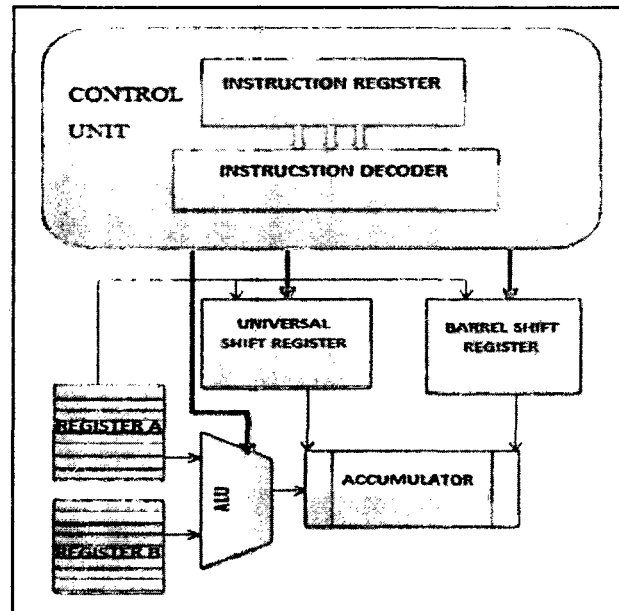


Figure 2.2 Architecture of 8-bit RISC processor [18]

The design of a 16-bit processor has been reported [19] with customised instruction set for soft-core RISC processor. Instruction Set Architecture (ISA) contains 35 basic instructions. Among all soft-core processors, RISC design is widely adopted for its single clock cycle instructions and less resource requirement compared to CISC approach. The computer architecture of 16-bit RISC processor is shown in Figure 2.3. It describes the custom simulation of a RISC soft-core processor's instruction set that is based on Microchip PIC16C5X architecture. Memory address remapping algorithm is introduced to remap the memory address to correct physical memory address due to memory banking scheme being applied. Simulation process is done on a highly customizable Java based computer architecture simulator. It provides features to insert customized instruction in an assembly language and has the ability to perform simulation at microcode level to variety of CPU architectures.

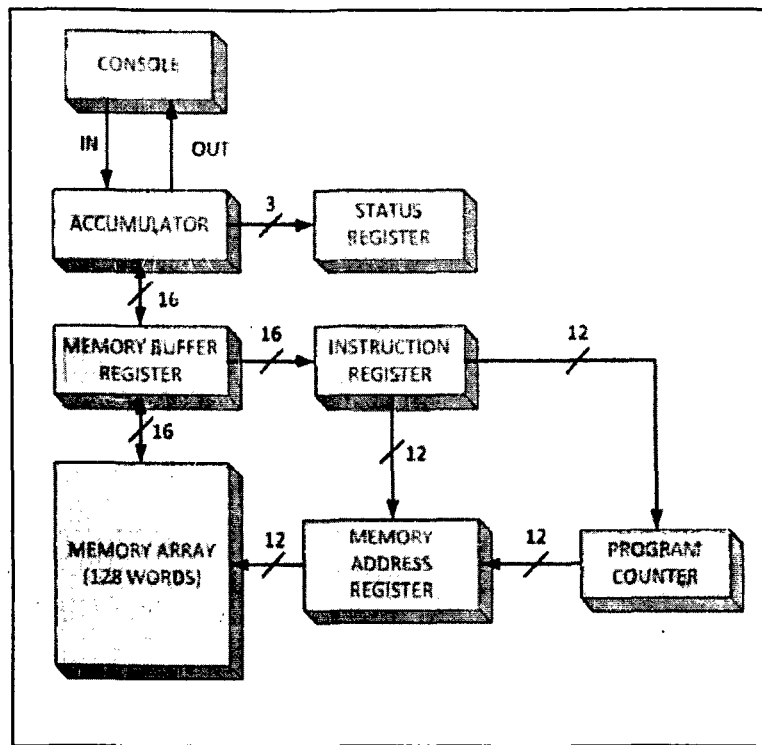


Figure 2.3 Computer architecture of 16-bit RISC processor [19]

Chapter 3

DESIGN OF 8-BIT RISC PROCESSOR

This chapter presents a design of an 8-bit RISC processor. Section 3.1 presents the architecture of proposed processor. The architecture of the proposed processor is shown in the figure 3.1. Section 3.2 presents the description of functional modules. Section 3.3 describes the instruction set architecture.

3.1 Architecture of 8-bit RISC processor

The 8-bit RISC processor is designed using Harvard architecture, having separate instruction memory and data memory. The Figure 3.1 shows the complete architecture of proposed system. The proposed processor is having 8-bit ALU, Two 8-bit I/O ports, serial-in and serial-out ports, Eight 8-bit general registers, 3 interrupts and 4-bit flag register having zero flag (Z), carry flag (C), borrow flag (B) and parity flag (P). The proposed RISC processor is running at 25MHz and on 2.5 voltage supply. The 8-bit SYSTEM DATABUS used for transferring data between different modules and 8-bit Accumulator used for arithmetic and logical operations are also integral part of the proposed processor. The Instruction Memory (IM) and Data Memory (DM) have different address and data buses for communicating between different modules. The interrupt module contains three interrupts, which are priority based. The proposed processor's most important feature is that its instruction set is very simple, contains only 34 instructions, which is easy to learn.

Another important feature is pipelining, used for improving performance and provides a way to reduce average execution time per instruction. The reduction can be observed as decreased the number of clock cycles per instruction (CPI), as falling off in the clock cycle time, or as a combinational effect. The pipelining architecture used for the proposed processor is shown in Figure 3.2. The proposed processor requires only two clock cycles for the execution of an instruction (jump instruction is an exception (TX2 also)), i.e. one fetch (TF1) and one execution cycle (TX1), which are mutually exclusive. By the pipelining technique, while executing one instruction next instruction is fetched, such that on every clock cycle one instruction will be executed. For efficient reduction of power, clock gating is used for specific modules which be clocked, only when it is required. Data memory and Register set are the modules where clock gating

is used. All loading to the registers is takes place at falling edge of clock and all control signals are generated during rising edge of clock.

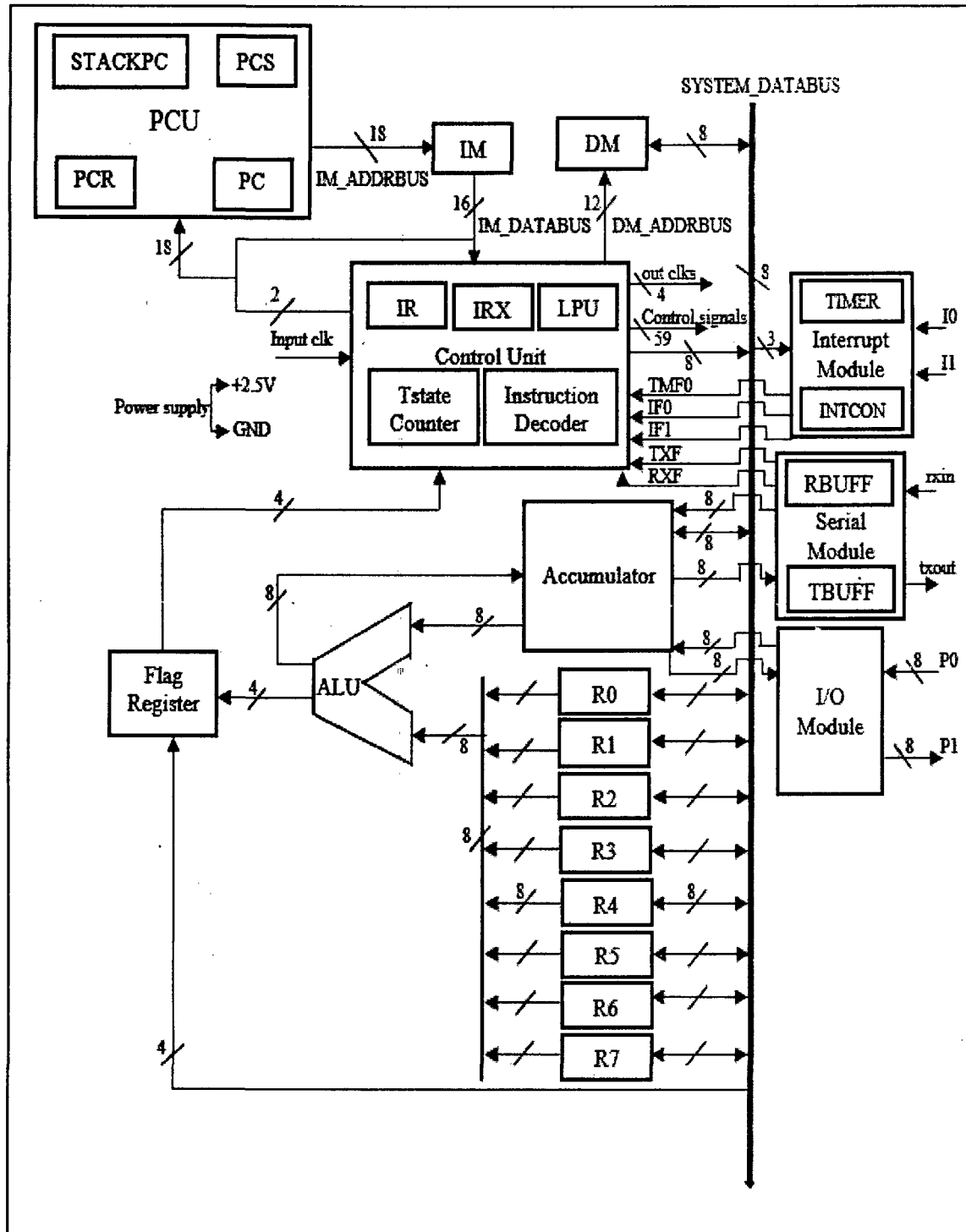


Figure 3.1 8-bit RISC processor architecture

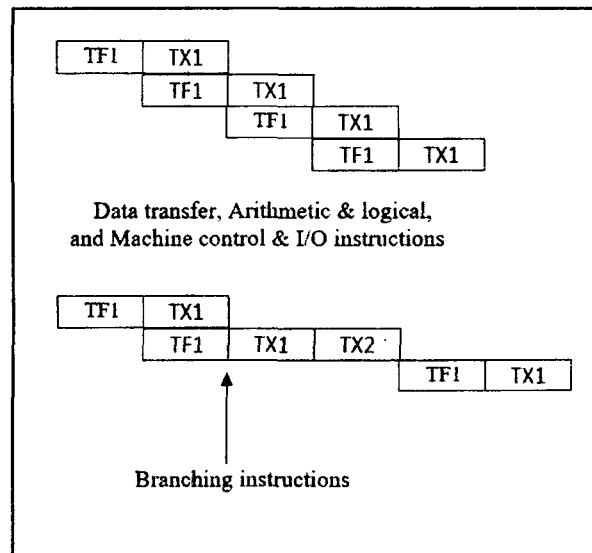


Figure 3.2 pipelining architecture

3.2 Description of Functional Modules

In this work, the RISC processor consists of blocks namely, Program Counter Unit (PCU), Instruction Memory, Data Memory, Control Unit, Register set, Arithmetic & Logical Unit (ALU), I/O module, Interrupt module and Serial Module.

3.2.1 Program Counter Unit (PCU)

Program Counter unit consists of Program Counter (PC), Program Counter Save (PCS), Program Counter Register (PCR) and STACKPC. In the proposed processor, Program Counter (PC) is 18-bit wide register that contains the address (location) of the instruction being executed at the current time. As each instruction is fetched, the program counter increases its stored value by one. In fetch cycle, an 18-bit address bus labelled as IM_ADDRBUS gets Program Counter content, when the corresponding control signal is enabled. When LPCS control signal is enabled, Program Counter Save (PCS) stores (18-bit) Program Counter content incremented by two value. PCR is used to store address (in TX1 cycle) of the next instruction to be executed, while a branching instruction is being executed. STACKPC is used to store the current PC value during the execution of interrupt service routine. All loadings to the register occur at falling edge of the clock.

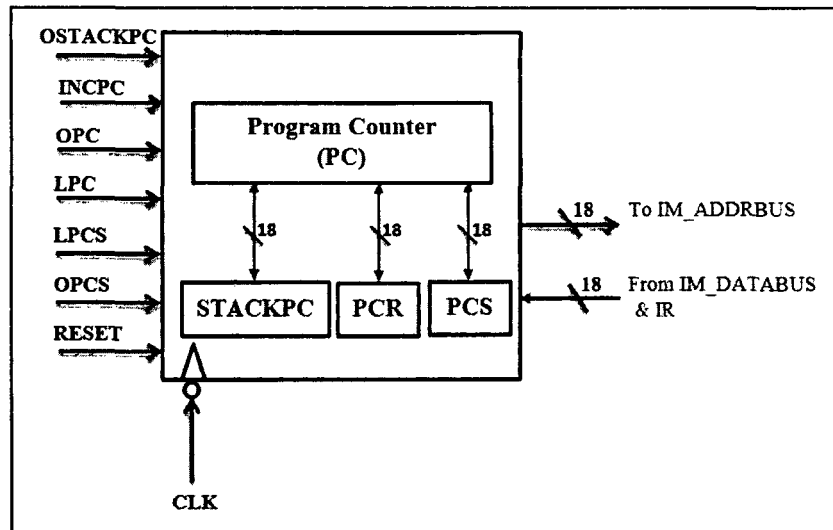


Figure 3.3 Block diagram of PCU

SIGNALS

CLK

OPC

LPC

INCPC

OPCS

LPCS

RESET

OSTACKPC

FUNCTIONS

System Clock

To Output PC data on IM_ADDRBUS

To Load PC with PCR content

To Increment PC value by one

To Load PC with PCS content

To Load PCS with PC content

To Reset PC

To Load PC with STACKPC content

3.2.2 Instruction Memory (IM)

Instruction Memory is 16-bit wide and having 262,144 address locations, so that any practical real time programs can be fitted into it. In fetch cycle, when the corresponding control signals are enabled, a 16-bit data bus labelled as IM_DATABUS gets Instruction Memory (IM) content corresponding to the valid address location provided by the IM_ADDRBUS.

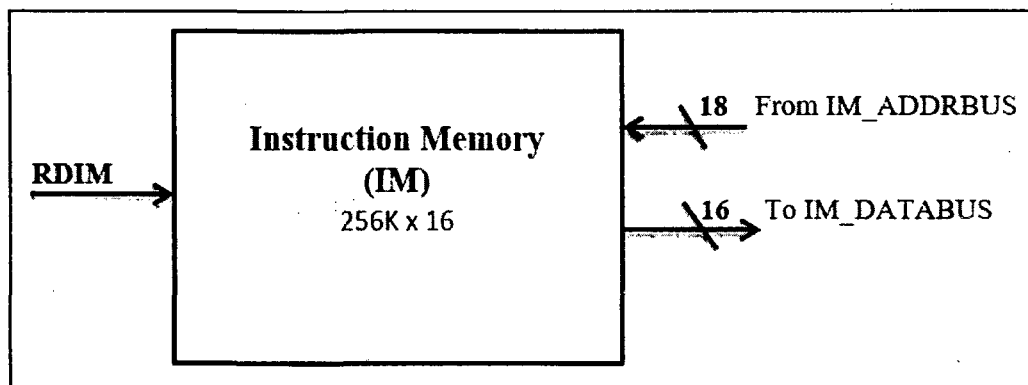


Figure 3.4 Block diagram of IM

SIGNAL

RDIM

FUNCTION

To output IM content to IM_DATABUS corresponding to the address provided by the IM_ADDRBUS

3.2.3 Control Unit

The control unit contains Instruction Register (IR), Instruction RegisterX (IRX), Tstate counter, Low Power Unit (LPU) and instruction decoder. The Instruction Register gets the instruction for decoding during fetch cycle. While executing one instruction next instruction is being fetched, therefore IR content should be stored in another register for execution of the instruction. For this purpose, Instruction Register (IR) content is moved to Instruction RegisterX (IRX) during every rising edge of clock of execution cycle. Tstate counter generates fetch and execution cycles required for the proper working of processor. The tstates TF1 (fetch cycle), TX1 (execution cycle), TX2 (execution cycle2, only for branching instruction) are generated at rising edge of clock. The pipelining feature is applied in the Tstate counter module. Interrupt priority and exceptions in instructions, like jump, are also taken in account in Tstate counter module. Instruction decoder will generate controls signals required for the modules whenever IRX is loaded with a valid instruction. As IRX is loaded at every rising edge of execution cycle, the control signals are generated at the same instance of time, because control signals are generated according to the changes in the IRX content. Clock gating for Data Memory and general purpose register set are done in Low Power Unit (LPU) of Control Unit. Control Unit receives inputs from flag register, Serial Module and Interrupt Module. The Control Unit takes input clock from source clock of FPGA and generates 59 control

signals and 4 clock signals for the proper working of all modules. The four clock signals include gated clock signals for register set and Data Memory modules, baud rate clock for Serial Module and 25 MHz output clock signal for all other modules.

The Control Unit receives inputs from Interrupt Module about the states of its three interrupts through IF0, IF1 and TMF0 flags, which are taken into account in Tstate counter module. Control Unit also receives inputs from Flag register regarding the states of its four flags, which are used for control signal generation related to branching instructions. The Control Unit receives inputs from Serial Module regarding transmission or reception of data through TXF and RXF flags, which are set whenever a transmission or reception is completed.

The Low Power Unit mainly uses clock-gating technique to reduce power dissipation. The Data Memory and Register set are the main memory elements, which consume power, so gated clock is provided to these modules. Whenever loading to a memory/general-purpose register corresponding module will be activated. Thus, power consumption in the proposed processor has been reduced by the use of clock-gating technique.

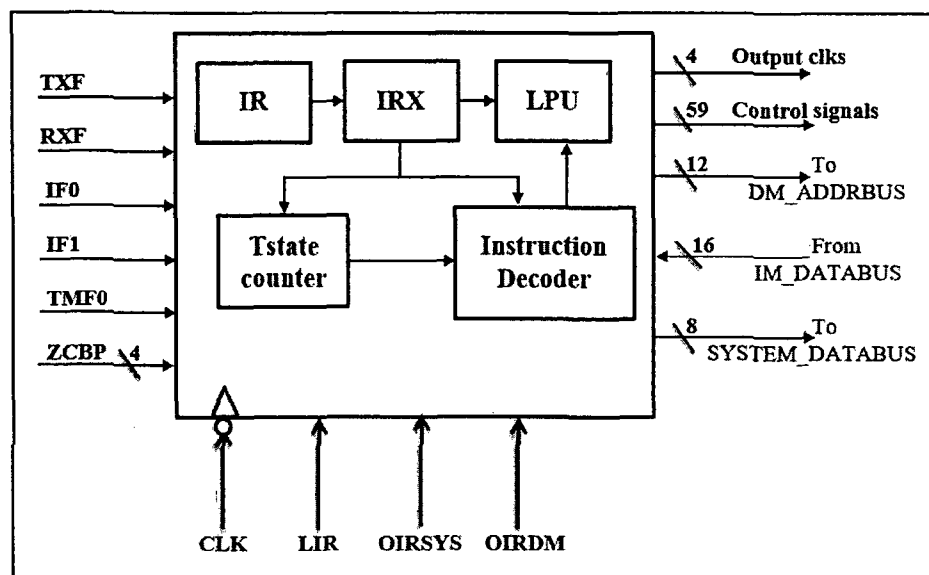


Figure 3.5 Block diagram of Control unit

All loading to memory/register takes place at falling edge of clock when the corresponding control signals are high.

SIGNALS

LIR

OIRSYS

OIRD

FUNCTIONS

To output IM_DATABUS content to IR and PC is incremented by one

To output IR content to SYSTEM_DATABUS

To output IR content to DM_ADDRBUS

3.2.4 Data Memory (DM)

Data Memory is 8-bit wide and has 4096 address locations. Data Memory gets the required address location by 12-bit address bus, DM_ADDRBUS, from control unit. Data memory provides read and write control and can be accessed by 8-bit data bus SYSTEM_DATABUS. Clock signal required for this module is provided by control unit, which is active only during loading operations for power reduction. All loading to the memory occurs at falling edge of clock, when corresponding control signals are high.

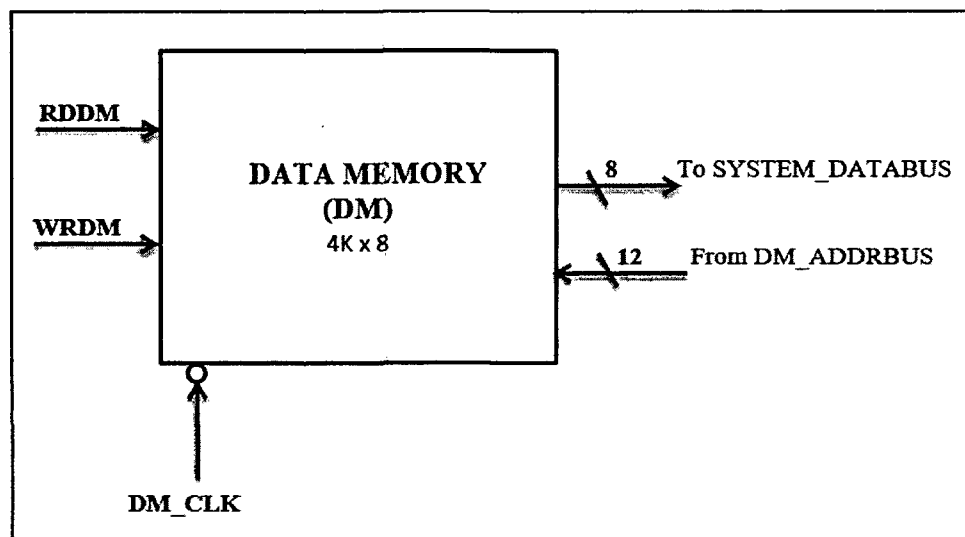


Figure 3.6 Block diagram of DM

SIGNALS

DM_CLK

RDDM

WRDM

FUNCTIONS

Data Memory Clock

To output DM content to SYSTEM_DATABUS corresponding to the address provided by the DM_ADDRBUS

To input SYSTEM_DATABUS content to DM corresponding to the address provided by the DM_ADDRBUS

3.2.5 Accumulator (A)

Accumulator is an 8-bit wide register is shown in Figure 3.7. Accumulator is connected to SYSTEM_DATABUS by a bidirectional data bus, which is used for data transfer instructions. Accumulator is also connected to ALU_DATABUS_A and ALU_RESULT data buses, which are used for arithmetic and logical instructions. Accumulator is connected to two 8-bit data buses for communicating with I/O module. The increment, decrement, rotate right, rotate left and compliment operations are performed on accumulator data. Accumulator is connected to TBUFF register in Serial Module for sending the data required for transmission through serial-out port 'txout' and is connected to RBUFF register in Serial Module for storing the data received through serial-in port 'rxin'.

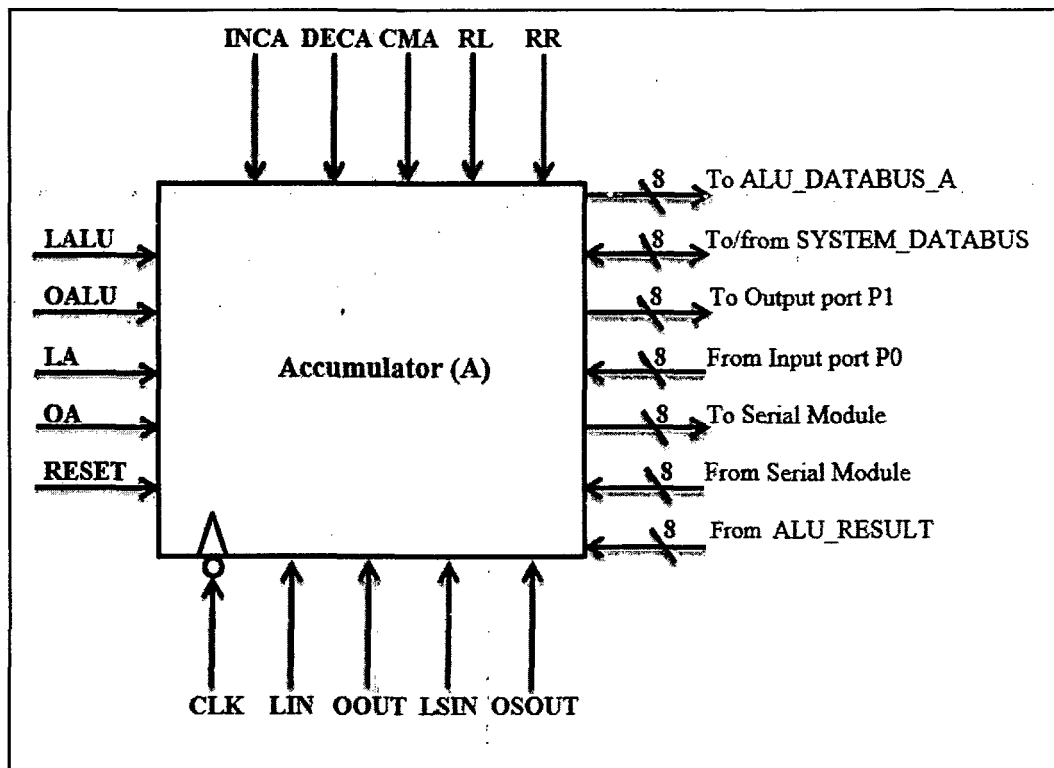


Figure 3.7 Block diagram of Accumulator

SIGNALS

CLK
RESET
LA
OA
INCA

FUNCTIONS

System clock
To Reset Accumulator
To Load Accumulator from SYSTEM_DATABUS
To Output Accumulator to SYSTEM_DATABUS
To Increment Accumulator

DECA	To Decrement Accumulator
CMA	To Complement all the bits of Accumulator
LALU	To Load Accumulator from ALU_RESULT
OALU	To Output Accumulator to ALU_DATABUS_A
RR	To Rotate the bits of Accumulator in right direction
RL	To Rotate the bits of Accumulator in left direction
LIN	To Load Accumulator from 8-bit input port P0
OOOT	To Send Accumulator content to 8-bit output port P1
LSIN	To Load Accumulator from 8-bit register RBUFF in Serial Module
OSOUT	To Send Accumulator content to 8-bit register TBUFF in Serial Module

3.2.6 Register Set

Register set contains eight 8-bit registers R0, R1, R2, R3, R4, R5, R6 and R7, which can be used for storing data that are frequently used. Register set is connected to ALU unit by ALU_DATABUS_R, which is a unidirectional data bus for performing arithmetic and logic operations. It is also connected to SYSTEM_DATABUS by a bidirectional data bus for loading and storing data. All loading to the register occur at falling edge of clock, when corresponding control signals are high. The clock input to Register set is gated-clock, which is active only during loading to any one of the registers. In the Figure 3.8, in the control signals LRX, OERX and OERALX, X is 0 – 7 implying R0, R1, R2, R3, R4, R5, R6 and R7 register respectively.

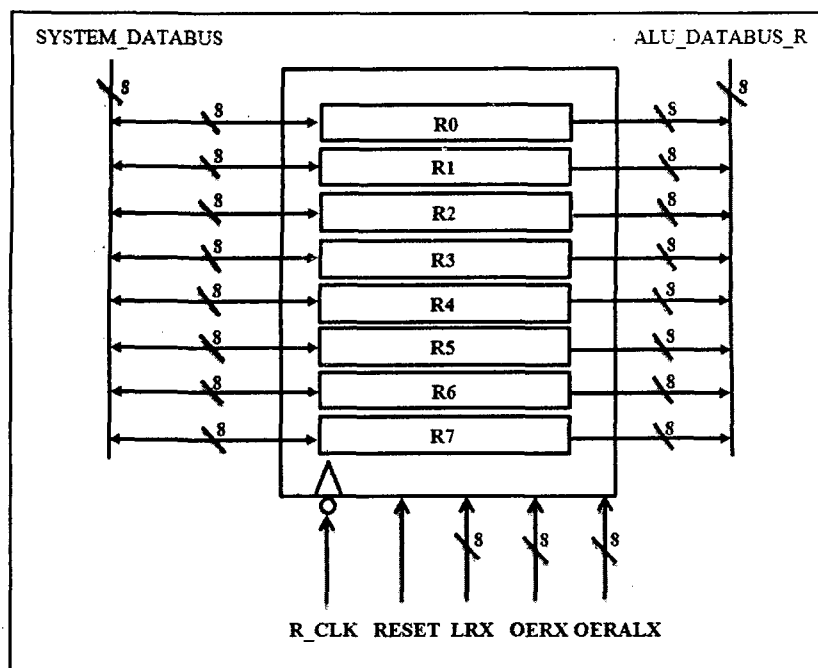


Figure 3.8 Block diagram of Register set

SIGNALS

R_CLK

LRX

OERX

OERALX

RESET

FUNCTIONS

Register set clock

To Load RX from SYSTEM_DATABUS where X= 0, 1, 2, 3, 4, 5, 6 or 7.

To Output RX to SYSTEM_DATABUS

To Output RX to ALU_DATABUS_R

To Reset all eight registers

3.2.8 ALU unit

ALU is connected to Accumulator and general-purpose registers by its 8-bit buses ALU_DATABUS_A and ALU_DATABUS_R. ALU unit consists of AND, OR, XOR, ADD, SUB operations. RX shown in figure 3.9 can be any of the register R0, R1, R2, R3, R4, R5, R6 or R7. The result of operation is stored in Accumulator by the bus labelled as ALU_RESULT. The zero flag (Z), Carry flag (C), Borrow flag (B) and parity flag (P) are updated according to the ALU operation, which are stored in 4-bit Flag Register. Parity flag is set only when resultant of ALU operation contains odd number of ones.

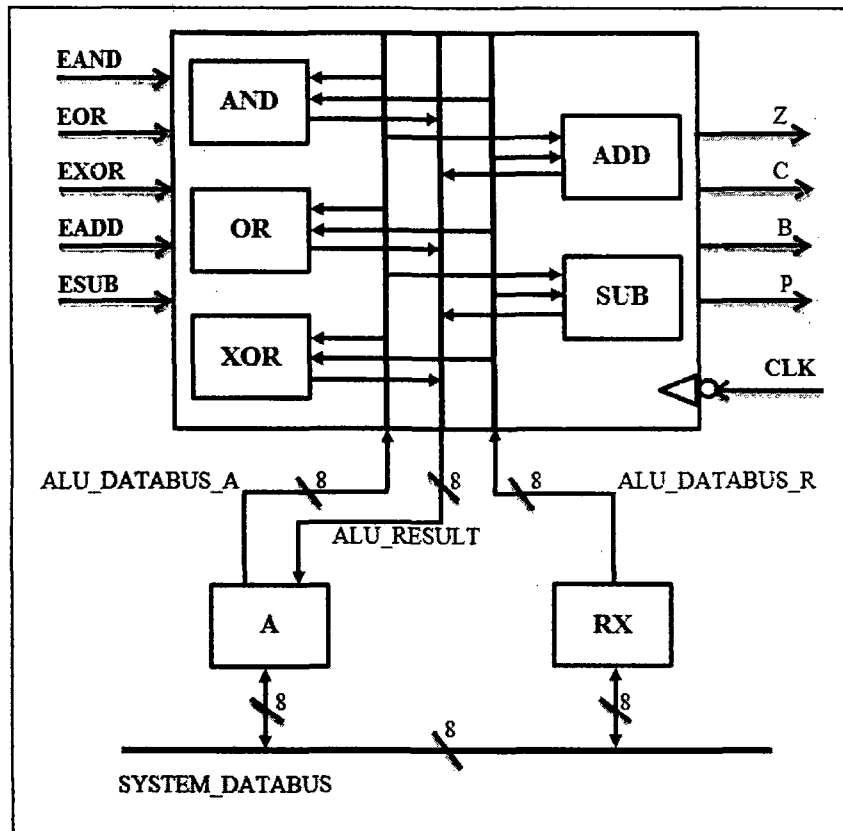
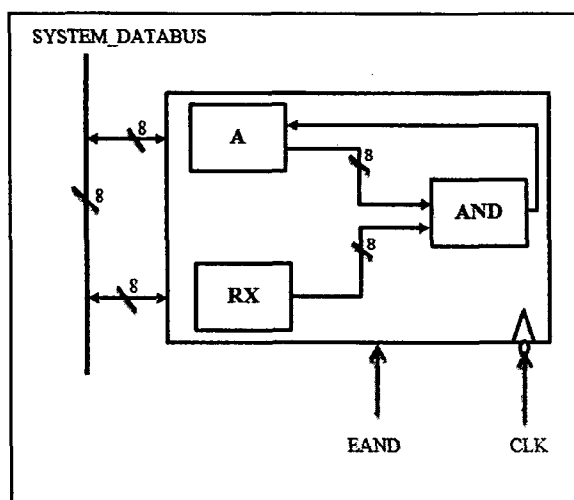


Figure 3.9 Block Diagram of ALU unit



If OALU=1 and OERALX=1, data from accumulator register and RX register are transferred to ALU data buses and if EAND=1 then AND operation will take place in AND ALU block.

Parity flag (P) and Zero flag (Z) are updated. When LALU=1, the result is transferred to accumulator during falling edge of clock.

Where X = 0, 1, 2, 3, 4, 5, 6 or 7.

Figure 3.10 Block diagram of AND block

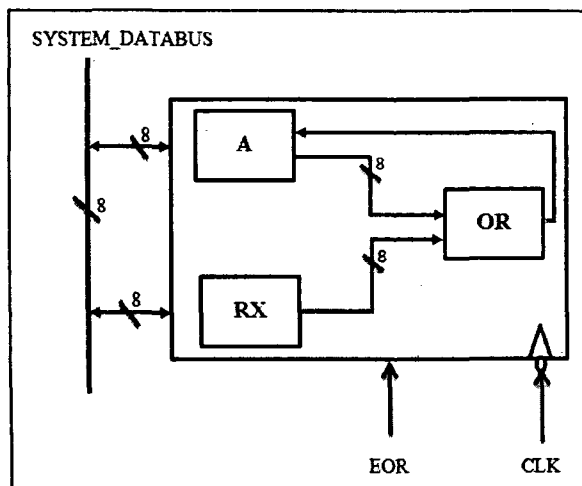


Figure 3.11 Block diagram of OR block

If OALU=1 and OERALX=1, data from accumulator register and RX register are transferred to ALU data buses and if EOR=1 then OR operation will take place in OR ALU block.

Parity flag (P) and Zero flag (Z) are updated. When LALU=1, the result is transferred to accumulator during falling edge of clock.

Where X = 0, 1, 2, 3, 4, 5, 6 or 7.

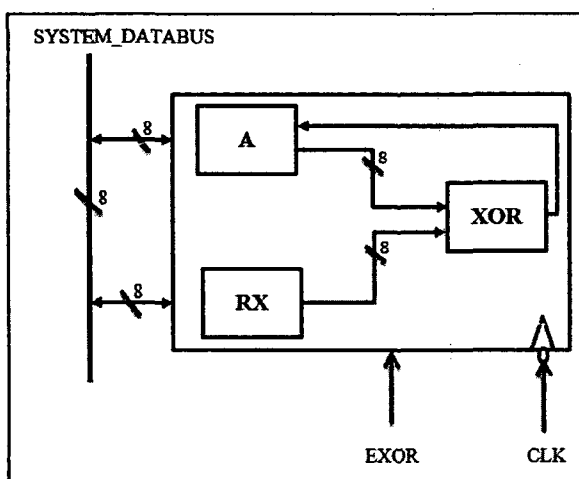


Figure 3.12 Block diagram of XOR block

If OALU=1 and OERALX=1, data from accumulator register and RX register are transferred to ALU data buses and if EXOR=1 then XOR operation will take place in XOR ALU block.

Parity flag (P) and Zero flag (Z) are updated. When LALU=1, the result is transferred to accumulator during falling edge of clock.

Where X = 0, 1, 2, 3, 4, 5, 6 or 7.

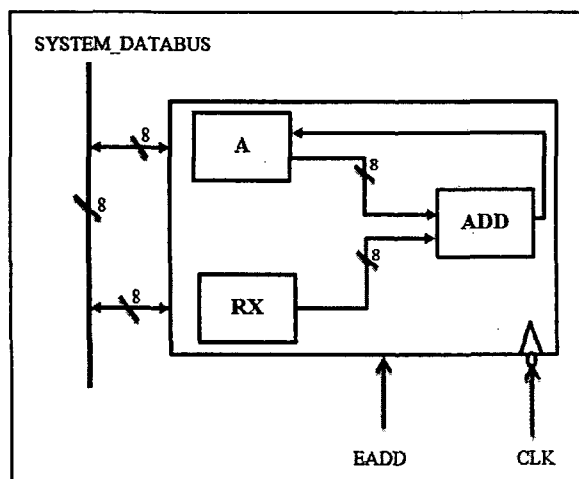
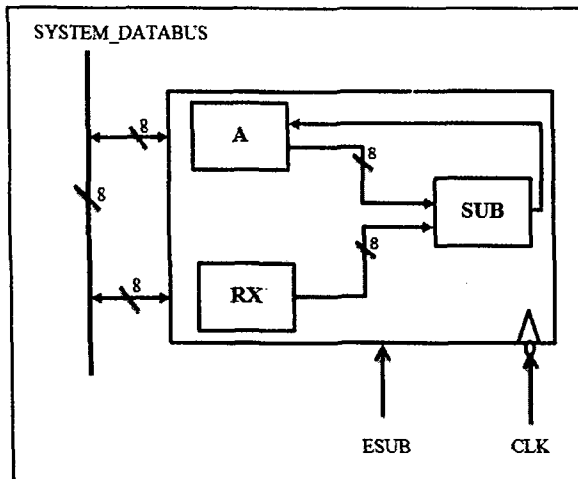


Figure 3.13 Block diagram of ADD block

If OALU=1 and OERALX=1, data from accumulator register and RX register are transferred to ALU data buses and if EADD=1 then addition operation will take place in ADD ALU block.

Carry flag (C), Parity flag (P) and Zero flag (Z) are updated. When LALU=1, the result is transferred to accumulator during falling edge of clock.

Where X = 0, 1, 2, 3, 4, 5, 6 or 7.



If OALU=1 and OERALX=1, data from accumulator register and RX register are transferred to ALU data buses and if ESUB=1 then subtraction operation will take place in SUB ALU block.

Borrow flag (B), Parity flag (P) and Zero flag (Z) are updated. When LALU=1, the result is transferred to accumulator during falling edge of clock.

Where X = 0, 1, 2, 3, 4, 5, 6 or 7.

Figure 3.14 Block diagram of SUB block

3.2.8 Flag Register

Flag register is a 4-bit special purpose register, which is used to store the status of the result of any ALU operation. Flag register consists of Zero flag (Z), Carry flag (C), Borrow flag (B) and Parity flag (P). Updating of the flags will occur corresponding to the ALU operations only. For example, when an addition operation is performed Zero, Carry and Parity flags will be updated, but Borrow flag will remain the same. All process occurs at falling edge of clock when the corresponding control signals are high. Figure 3.15 shows the block diagram of Flag register and Figure 3.16 shows the order in which flags are arranged in Flag register.

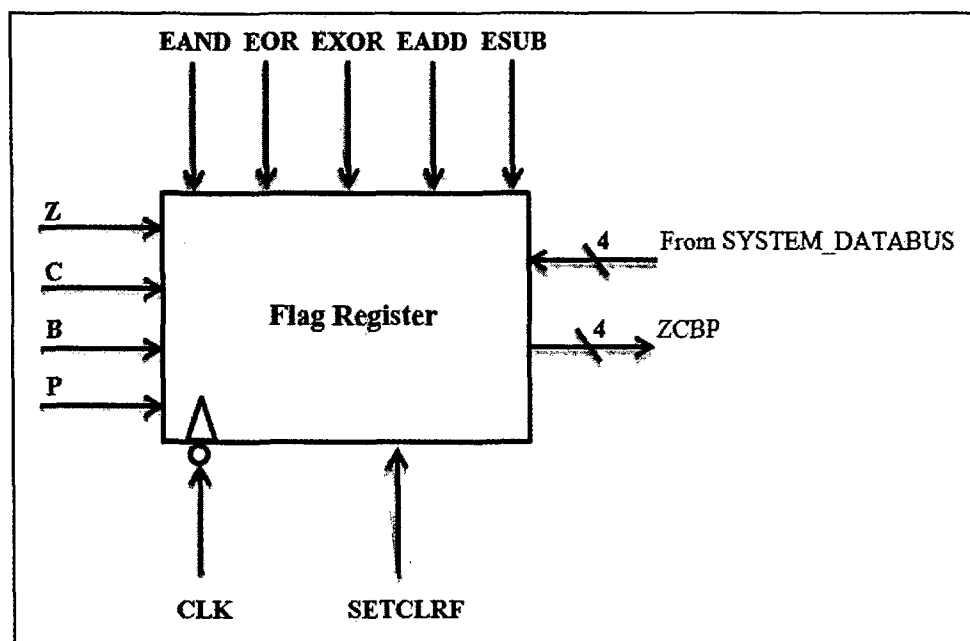


Figure 3.15 Block diagram of Flag register

Z	C	B	P
bit 3	bit 2	bit 1	bit 0

Figure 3.16 4-bit Flag register

SIGNALS

CLK

SETCLRF

EAND

EOR

EXOR

EADD

ESUB

FUNCTIONS

System clock

To Set/Clear flags bits by SYSTEM_DATABUS content

A and RX, Parity (P) and Zero (C) flags are updated

A or RX, Parity (P) and Zero (C) flags are updated

A xor RX, Parity (P) and Zero (C) flags are updated

A + RX, Carry (C), Parity (P) and Zero (C) flags are updated

A - RX, Borrow (B), Parity (P) and Zero (C) flags are updated

3.2.9 I/O Module

The I/O module has one 8-bit input port and one 8-bit output port for communicating with external environment, which can be a sensor, actuator or even another microprocessor. The input port P0 and output port P1 are directly connected to Accumulator, which will control the data flow through the ports when corresponding control signals are enabled.

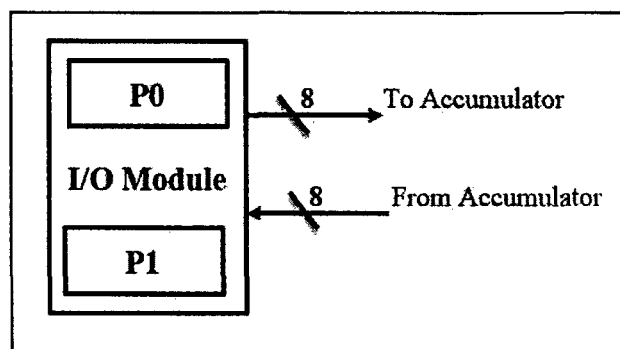


Figure 3.17 Block diagram of I/O Module

3.2.10 Interrupt Module

The interrupt module contains two external hardware interrupts I0 and I1 and one timer interrupt. The interrupts are priority based, the I0 interrupt is having highest priority, followed by I1 interrupt and timer interrupt is having lowest priority. The I1 interrupt is

maskable and I0 is not maskable. The external hardware interrupts are level triggered and must be high for two clock cycles for proper working of the proposed processor. The interrupt module is having a 3-bit register INTCON to control the operations in interrupt module. Its bit 2 is used for enabling and disabling timer interrupt, bit 1 is used for enabling and disabling external interrupts and bit 0 is used for masking I1. Whenever bit 1 of INTCON is high and I0 is high, IF0 flag will be set. However, IF1 flag will be set only if the bit 1 of INTCON is high, the bit 0 of INTCON is low and I1 is high. The Interrupt module has 10-bit register known as TIMER for counting to a predefined number, in this case it is 1023 (maximum value). When the TIMER register reaches the maximum value, timer flag TMF0 gets set. Thus in the interrupt service routine we need to clear the TMF0 flag by the instruction CLRTMRF. Whenever timer interrupt is turned on, timer flag will be raised at predefined intervals, which depends on the predefined number.

TMR	ENB	MASK
bit 2	bit 1	bit 0

Figure 3.18 INTCON register

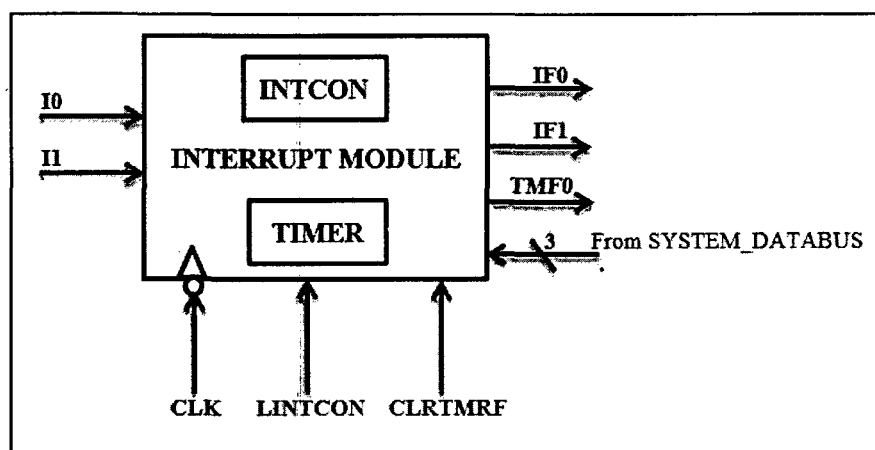


Figure 3.19 Block diagram of Interrupt Module

SIGNALS

CLK

LINTCON

CLRTMRF

FUNCTIONS

System clock

To enable/disable interrupts and mask I1 interrupt by SYSTEM_DATABUS content.

Clear timer flag TMF0

3.2.11 Serial Module

The Serial Module contains rxin as serial-in port and txout as serial-out port. The serial communication is based on UART protocol shown in Figure 3.20. The baud rate used for this Serial Module is 115200 per second. The data transmission starts with a start bit of 0, followed by the individual data bits of the word with the Least Significant Bit (LSB) being sent first and then stop bit 1. The Serial Module consists of two 8-bit registers TBUFF and RBUFF for storing data while transmission and reception. The data stored in TBUFF register is shifted out during serial data transmission and process is reversed for RBUFF register. The baud rate (clock) required for serial communication is provided by the control unit. Serial Module will provide TXF and RXF flags to Control Unit regarding transmission or reception of data, which will be set whenever a transmission or reception respectively is completed.

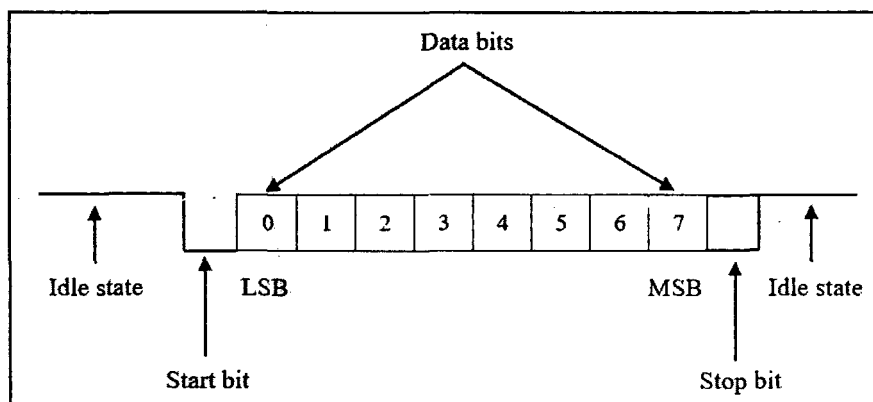


Figure 3.20 UART serial communication protocol

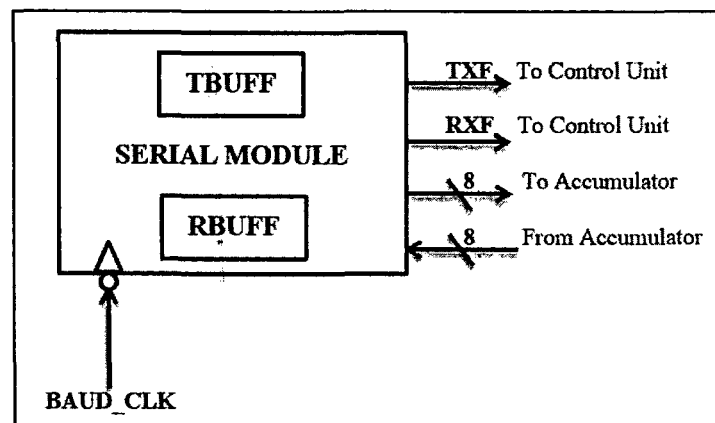


Figure 3.21 Block diagram of Serial Module

3.3 List of control signals, I/O bus and Flags

The Control Unit will generate 59 control signals and 4 output clocks for the proper working of all modules.

Program Counter Unit:

Control signals: OPC, LPC, INCPC, OPCS, LPCS, OSTACKPC and LPCR

Input bus: IM_DATABUS [15:0], IRX [9:8]

Output bus: IM_ADDRBUS [17:0]

Instruction Memory:

Control signals: RDIM

Input bus: IM_ADDRBUS [17:0]

Output bus: IM_DATABUS [15:0]

Control Unit:

Control signals: OIRSYS, LIR and OIRDM

Input bus: IM_DATABUS [15:0]

Output bus: DM_ADDRBUS [15:0], SYSTEM_DATABUS [7:0]

Flag inputs: Z, C, B, P, RXF, TXF, IF0, IF1 and TMF0

Data Memory:

Control signals: RDDM and WRDM

Input bus: DM_ADDRBUS [15:0]

Output bus: SYSTEM_DATABUS [7:0]

Accumulator:

Control signals: LA, OA, LALU, OALU, INCA, DECA, CMA, RR, RL, LIN, OOUT, LSIN, OSOUT and RESET

Input bus: P0 [7:0], ALU_RESULT [7:0]

Output bus: P1 [7:0], ALU_DATABUS_A [7:0]

Input/Output bus: SYSTEM_DATABUS [7:0]

Register set:

Control signals: LR0, LR1, LR2, LR3, LR4, LR5, LR6, LR7, OER0, OER1, OER2, OER3, OER4, OER5, OER6 and OER7

Input/Output bus: SYSTEM_DATABUS [7:0]

Output bus: ALU_DATABUS_R [7:0]

ALU Unit:

Control signals: OERAL0, OERAL1, OERAL2, OERAL3, OERAL4, OERAL5,
OERAL6, OERAL7, EAND, EXOR, EOR, EADD and ESUB

Input bus: ALU_DATABUS_A [7:0], ALU_DATABUS_R [7:0]

Output bus: ALU_RESULT [7:0]

Flag Register:

Control signals: SETCLRF

Flag output: Z, C, B and P

Interrupt Module:

Control signals: LINTCON and CLR TMRF

Flag output: IF0, IF1, TMF0

Serial Module:

Flag output: TXF and RXF

3.4 Instruction Set Architecture

The Instruction set architecture contains four type of instructions, data transfer instruction, arithmetic and logical instruction, branching instruction and machine control and I/O instruction. The instruction set architecture contains only 34 basic instructions and total number of opcodes is 83. The opcode contains 16 bits from 0 to 15, of which 15th and 14th bits decides type of instruction being performed. If the 15th and 14th bit are 00, 01, 10 or 11, type of instruction will be data transfer instruction, arithmetic and logic instruction, branching instruction or machine control and I/O instruction respectively.

The main advantage of this instruction set architecture is the use of SAV PC instruction that saves PC incremented by two value in PCS. SAV PC can be used before jump instruction so that after jumping to another location, using RES PC (loads PCS content to PC) we can come back to next instruction after jump instruction. The combined use of SAV PC and jump instruction, will act like a CALL instruction thus reducing the number of instruction required without sacrificing functionality. We can also use jump instruction alone to jump to specific location. Another special instruction is RETI that is used to restore the PC value after the execution of interrupt service routine. SETCLRF is a special instruction, which can be used to clear or set every flag in flag register.

EDINTER is another special one, which can be used to enable/disable interrupts and mask the I1 interrupt when needed.

Serial data transmission is possible by the use of SOUT TBUFF and WAIT TXF. Initially use SOUT TBUFF to send accumulator content to TBUFF. Then use WAIT TXF for waiting until data is completely transmitted serially from TBUFF register. For serial data reception, initially use WAIT RXF for waiting until data is completely received by RBUFF register and then use SIN RBUFF to store the serial data received in RBUFF register to accumulator. IN P0 is used to take 8-bit parallel input data from Port P0 that is connected to external hardware. OUT P1 is used to send 8-bit parallel data to Port P1, which is connected to external hardware. Thus by using rxin, txout, Port P0 and Port P1, the proposed processor can communicate with different types of external hardware.

Table 3.1 shows instruction set of proposed RISC processor. In the Table 3.1 Opcode column, rrr is register select code from 000 to 111, x is don't care, a is address and d is data. In the Table 3.1 Mnemonic column, RX is register representing any one of R0, R1, R2, R3, R4, R5, R6 and R7, A is accumulator, M is Data Memory address and add18 is Instruction Memory address. In Operation column, update flags means update the flags corresponding to that operation.

Table 3.1 Instruction set

Opcode	Mnemonic	Operation
Data Transfer Instructions		
00000rrrxxxxxxxxx	MOV A, RX	RX => A
00010rrrxxxxxxxxx	MOV RX, A	A => RX
0010aaaaaaaaaaaa	MOV A, M	M => A
0011aaaaaaaaaaaa	MOV M, A	A => M
00001xxxddddddd	MOVI A, 8-bit data	8-bit data => A
Arithmetic and Logical Instructions		
0100000rrrxxxxxx	AND A, RX	A <= A and RX update flags
0100001rrrxxxxxx	XOR A, RX	A <= A xor RX update flags
0100010rrrxxxxxx	OR A, RX	A <= A or RX update flags
0100011rrrxxxxxx	ADD A, RX	A <= A + RX update flags
0100100rrrxxxxxx	SUB A, RX	A <= A - RX update flags
0101000xxxxxxxxx	CMA	A <= ~A
0101001xxxxxxxxx	INC A	A <= A + 1

0101010xxxxxxx	DEC A	$A \leq A - 1$
0101011xxxxxxx	RR A	rotate accumulator right by 1 bit
0101100xxxxxxx	RL A	rotate accumulator left by 1 bit
011000xxxxxxddd	SETCLRF 4-bitdata	clear/set flags
011001xxxxxxxxxx	CLRTMRF	clear timer flag
0111xxxxxxxxxxddd	EDINTER 3-bitdata	To control interrupt's operations; enable, disable and mask
Branching Instructions		
10000aaxxxxxxxxxx aaaaaaaaaaaaaaaaaa	JUMP add18	Jump to 18-bit address add18
10001aaxxxxxxxxxx aaaaaaaaaaaaaaaaaa	JZ add18	Jump to 18-bit address add18 if Z=1
10010aaxxxxxxxxxx aaaaaaaaaaaaaaaaaa	JC add18	Jump to 18-bit address add18 if C=1
10001aaxxxxxxxxxx aaaaaaaaaaaaaaaaaa	JB add18	Jump to 18-bit address add18 if B=1
10001aaxxxxxxxxxx aaaaaaaaaaaaaaaaaa	JP add18	Jump to 18-bit address add18 if P=1
Machine Control and I/O Instructions		
110000xxxxxxxxxx	HALT	To stop operations
110001xxxxxxxxxx	RESET	To reset Accumulator, Register set and PC
110010xxxxxxxxxx	SAV PC	Save PC+2 value in PCS register
110011xxxxxxxxxx	RES PC	Restore PC with PCS content
110100xxxxxxxxxx	RETI	Return from interrupt service routine
110101xxxxxxxxxx	WAIT TXF	Wait until TXF flag gets set
110110xxxxxxxxxx	WAIT RXF	Wait until RXF flag gets set
11100xxxxxxxxxxx	IN P0	Accumulator gets Port P0 content
11101xxxxxxxxxxx	OUT P1	Accumulator sends its content to Port P1
11110xxxxxxxxxxx	SIN RBUFF	Accumulator gets RBUFF register content
11111xxxxxxxxxxx	SOUT TBUFF	Accumulator sends its content to TBU register

The more information regarding instruction set architecture is provided in appendix A and micro operations of each instruction are covered in appendix B.

3.5 Source Code

The programming language used for the design of this 8-bit RISC processor is Verilog. The source code for the proposed processor is given below.

```

/* Control Unit, Program Counter unit and Instruction Memory */

module ins_decoder(
input inclk,
input [7:0] PORT0,
output [7:0] PORT1,
input IO,I1,
output reg clk=1'b0,
inout IF0,IF1,TMF0,

```



```

inout [3:0] ZCBP,
output LR0,LR1,LR2,LR3,LR4,LR5,LR6,LR7,OER0,OER1,OER2,OER3,OER4,OER5,OER6,OER7,
output OERAL0,OERAL1,OERAL2,OERAL3,OERAL4,OERAL5,OERAL6,OERAL7,
output RESET,
output OIRDM,RDDM,WRDM,
output LA,OA,LALU,OALU,INCA,DECA,CMA,RR,RL,
output EAND,EXOR,EOR,EADD,ESUB,
output SETCLRf,CLRTMRF,
output LINTCON,
output LIN,OOOUT,
output R_CLK,DM_CLK,
output reg baudclk=1'b0,
output reg c=1'b0,
input rxin,
output LSIN,OSOUT,
output txout,
inout RXF,TXF,
inout [15:0] IM_DATABUS,
inout [17:0] PCS_DATABUS,
inout [7:0] SYSTEM_DATABUS,
output [11:0] DM_ADDRBUS,
output [17:0] IM_ADDRBUS,
output [7:0] acc);
reg [15:0] IRX;
reg [17:0] PCR;
reg [17:0] PC=18'b0;
reg [17:0] PCS;
reg [17:0] STACKPC;
reg [15:0] IM [0:262143];
reg [15:0] IR;
always@(posedge inclk)
begin
    clk=~clk;

end
reg [7:0] cb =8'b0;
always@(posedge inclk)
begin
    if(cb==217)
        begin
            baudclk=~baudclk;           //baudclock generation for serial module
            cb=0;
            c=1'b1;
        end
    else
        begin
            cb=cb+1;
            c=1'b0;
        end
end
wire OPC,LPC,INCPC,OPCS,LPCS,OSTACKPC,OIRSYS,RDIM,LPCR,LIR;
wire DT,ALU,BR,MIO;

//identifying type of instruction
assign DT=(IRX[15:14]==2'b0)?1'b1:1'b0;
assign ALU=(IRX[15:14]==2'b01)?1'b1:1'b0;
assign BR=(IRX[15:14]==2'b10)?1'b1:1'b0;
assign MIO=(IRX[15:14]==2'b11)?1'b1:1'b0;

```

```

// tstate counter instantiation
tstate_counter
ts(.clk(clk),.IFO(IFO),.IF1(IF1),.TMFO(TMFO),.TXF(TXF),.RXF(RXF),.IR8(IR[15:10]),.TF1(TF1),.TX1(TX1),.TX2(TX2));

assign RESET = (TX1&&MIO)&&(~IRX[13])&&((IRX[12:10]==3'b001)?1'b1:1'b0);
//assign STOP = (MIO)&&(~IRX[13])&&((IRX[12:10]==3'b0)?1'b1:1'b0);

// control signals of register set
assign LR0 = (DT&&TX1)&&((IRX[13:11]==3'b010)?1'b1:1'b0)&&((IRX[10:8]==3'b0)?1'b1:1'b0);
assign LR1 = (DT&&TX1)&&((IRX[13:11]==3'b010)?1'b1:1'b0)&&((IRX[10:8]==3'b001)?1'b1:1'b0);
assign LR2 = (DT&&TX1)&&((IRX[13:11]==3'b010)?1'b1:1'b0)&&((IRX[10:8]==3'b010)?1'b1:1'b0);
assign LR3 = (DT&&TX1)&&((IRX[13:11]==3'b010)?1'b1:1'b0)&&((IRX[10:8]==3'b011)?1'b1:1'b0);
assign LR4 = (DT&&TX1)&&((IRX[13:11]==3'b010)?1'b1:1'b0)&&((IRX[10:8]==3'b100)?1'b1:1'b0);
assign LR5 = (DT&&TX1)&&((IRX[13:11]==3'b010)?1'b1:1'b0)&&((IRX[10:8]==3'b101)?1'b1:1'b0);
assign LR6 = (DT&&TX1)&&((IRX[13:11]==3'b010)?1'b1:1'b0)&&((IRX[10:8]==3'b110)?1'b1:1'b0);
assign LR7 = (DT&&TX1)&&((IRX[13:11]==3'b010)?1'b1:1'b0)&&((IRX[10:8]==3'b111)?1'b1:1'b0);

assign OER0 = (DT&&TX1)&&((IRX[13:11]==3'b0)?1'b1:1'b0)&&((IRX[10:8]==3'b0)?1'b1:1'b0);
assign OER1 = (DT&&TX1)&&((IRX[13:11]==3'b0)?1'b1:1'b0)&&((IRX[10:8]==3'b001)?1'b1:1'b0);
assign OER2 = (DT&&TX1)&&((IRX[13:11]==3'b0)?1'b1:1'b0)&&((IRX[10:8]==3'b010)?1'b1:1'b0);
assign OER3 = (DT&&TX1)&&((IRX[13:11]==3'b0)?1'b1:1'b0)&&((IRX[10:8]==3'b011)?1'b1:1'b0);
assign OER4 = (DT&&TX1)&&((IRX[13:11]==3'b0)?1'b1:1'b0)&&((IRX[10:8]==3'b100)?1'b1:1'b0);
assign OER5 = (DT&&TX1)&&((IRX[13:11]==3'b0)?1'b1:1'b0)&&((IRX[10:8]==3'b101)?1'b1:1'b0);
assign OER6 = (DT&&TX1)&&((IRX[13:11]==3'b0)?1'b1:1'b0)&&((IRX[10:8]==3'b110)?1'b1:1'b0);
assign OER7 = (DT&&TX1)&&((IRX[13:11]==3'b0)?1'b1:1'b0)&&((IRX[10:8]==3'b111)?1'b1:1'b0);

assign OERAL0 = (ALU&&TX1)&&((IRX[13:12]==2'b0)?1'b1:1'b0)&&((IRX[8:6]==3'b0)?1'b1:1'b0);
assign OERAL1 = (ALU&&TX1)&&((IRX[13:12]==2'b0)?1'b1:1'b0)&&((IRX[8:6]==3'b001)?1'b1:1'b0);
assign OERAL2 = (ALU&&TX1)&&((IRX[13:12]==2'b0)?1'b1:1'b0)&&((IRX[8:6]==3'b010)?1'b1:1'b0);
assign OERAL3 = (ALU&&TX1)&&((IRX[13:12]==2'b0)?1'b1:1'b0)&&((IRX[8:6]==3'b011)?1'b1:1'b0);
assign OERAL4 = (ALU&&TX1)&&((IRX[13:12]==2'b0)?1'b1:1'b0)&&((IRX[8:6]==3'b100)?1'b1:1'b0);
assign OERAL5 = (ALU&&TX1)&&((IRX[13:12]==2'b0)?1'b1:1'b0)&&((IRX[8:6]==3'b101)?1'b1:1'b0);
assign OERAL6 = (ALU&&TX1)&&((IRX[13:12]==2'b0)?1'b1:1'b0)&&((IRX[8:6]==3'b110)?1'b1:1'b0);
assign OERAL7 = (ALU&&TX1)&&((IRX[13:12]==2'b0)?1'b1:1'b0)&&((IRX[8:6]==3'b111)?1'b1:1'b0);
assign R_CLK = (LR0|LR1|LR2|LR3|LR4|LR5|LR6|LR7) && clk; //gated clock for register set

//Control signals of Program Counter Unit
assign OPC = TF1 || (TX1&&BR);
assign LPC = (TX2 && BR)&&(((IRX[13:11]==3'b0)?1'b1:1'b0) || (((IRX[13:11]==3'b001)?1'b1:1'b0)&&ZCBP[3]) || (((IRX[13:11]==3'b010)?1'b1:1'b0)&&ZCBP[2]) || (((IRX[13:11]==3'b011)?1'b1:1'b0)&&ZCBP[1]) || (((IRX[13:11]==3'b100)?1'b1:1'b0)&&ZCBP[0]));
assign INCPC = (TX2&&BR)&&(~LPC);
assign LPCS = (TX1&&MIO)&&(~IRX[13])&&((IRX[12:10]==3'b010)?1'b1:1'b0);
assign OPCS = (TX1&&MIO)&&(~IRX[13])&&((IRX[12:10]==3'b011)?1'b1:1'b0);
assign OSTACKPC = (TX1&&MIO)&&(~IRX[13])&&((IRX[12:10]==3'b100)?1'b1:1'b0);

// Control signals of control unit
assign LIR = TF1;
assign OIRDM = (TX1&&DT)&&(IRX[13]);
assign LPCR = TX1&&BR;
//assign OIRPC = TX1&&BR;
assign OIRSYS = ((TX1&&DT)&&((IRX[13:11]==3'b001)?1'b1:1'b0) || ((TX1&&ALU)&&((IRX[13:12]==2'b10)?1'b1:1'b0) || ((IRX[13:12]==2'b11)?1'b1:1'b0)));

// Control signals of Instruction Memory

```

```

assign RDIM = TF1 || (TX1&&BR);

// Control signals of Data Memory
assign RDDM = (TX1&&DT)&&((IRX[13:12]==2'b10)?1'b1:1'b0);
assign WRDM = (TX1&&DT)&&((IRX[13:12]==2'b11)?1'b1:1'b0);
assign DM_CLK = WRDM && clk;

// Control signals of Accumulator
assign LA = (TX1&&DT)&&(~IRX[12]);
assign OA = (TX1&&DT)&&(IRX[12]);
assign OALU = (TX1&&ALU)&&((IRX[13:12]==2'b0)?1'b1:1'b0);
assign LALU = OALU;
assign CMA = (TX1&&ALU)&&((IRX[13:12]==2'b01)?1'b1:1'b0)&&((IRX[11:9]==3'b0)?1'b1:1'b0);
assign INCA = (TX1&&ALU)&&((IRX[13:12]==2'b01)?1'b1:1'b0)&&((IRX[11:9]==3'b001)?1'b1:1'b0);
assign DECA = (TX1&&ALU)&&((IRX[13:12]==2'b01)?1'b1:1'b0)&&((IRX[11:9]==3'b010)?1'b1:1'b0);
assign RR = (TX1&&ALU)&&((IRX[13:12]==2'b01)?1'b1:1'b0)&&((IRX[11:9]==3'b011)?1'b1:1'b0);
assign RL = (TX1&&ALU)&&((IRX[13:12]==2'b01)?1'b1:1'b0)&&((IRX[11:9]==3'b100)?1'b1:1'b0);
wire a = (LA|LALU|LIN|CMA|INCA|DECA|RR|RL);

// Control signals of ALU Unit
assign EAND = (TX1&&ALU)&&((IRX[13:12]==2'b0)?1'b1:1'b0)&&((IRX[11:9]==3'b0)?1'b1:1'b0);
assign EXOR = (TX1&&ALU)&&((IRX[13:12]==2'b0)?1'b1:1'b0)&&((IRX[11:9]==3'b001)?1'b1:1'b0);
assign EOR = (TX1&&ALU)&&((IRX[13:12]==2'b0)?1'b1:1'b0)&&((IRX[11:9]==3'b010)?1'b1:1'b0);
assign EADD = (TX1&&ALU)&&((IRX[13:12]==2'b0)?1'b1:1'b0)&&((IRX[11:9]==3'b011)?1'b1:1'b0);
assign ESUB = (TX1&&ALU)&&((IRX[13:12]==2'b0)?1'b1:1'b0)&&((IRX[11:9]==3'b100)?1'b1:1'b0);

// Control signals of Interrupt Module
assign LINTCON = (TX1&&ALU)&&((IRX[13:12]==2'b11)?1'b1:1'b0);
assign CLRTMRF = (TX1&&ALU)&&((IRX[13:12]==2'b10)?1'b1:1'b0)&&((IRX[11:10]==2'b01)?1'b1:1'b0);

// Control signals of Flag Register
assign SETCLRF = (TX1&&ALU)&&((IRX[13:12]==2'b10)?1'b1:1'b0)&&((IRX[11:10]==2'b0)?1'b1:1'b0);

// Control signals related I/O Module
assign LIN = (TX1&&MIO)&&((IRX[13:11]==3'b100)?1'b1:1'b0);
assign OOUT = (TX1&&MIO)&&((IRX[13:11]==3'b101)?1'b1:1'b0);

// Control signals related to Serial Module
assign LSIN = (TX1&&MIO)&&((IRX[13:11]==3'b110)?1'b1:1'b0);
assign OSOUT = (TX1&&MIO)&&((IRX[13:11]==3'b111)?1'b1:1'b0);

// I/O buses
assign SYSTEM_DATABUS = OIRSYS?IRX[7:0]:'bz;
assign DM_ADDRBUS = OIRDM?IRX[11:0]:'bz;
assign PCS_DATABUS = (LPCS|OPCS)?((LPCS?PC:18'b0)|(OPCS?PCS:18'b0)):'bz;
assign IM_ADDRBUS = OPC?PC:'bz;
assign IM_DATABUS = RDIM?IM[IM_ADDRBUS]:'bz;

//Program Counter Unit operations
always@(negedge clk)
begin
    if(LPC)
        PC[17:0]=PCR;
    else if(LPCR)
        begin
            PCR[15:0] = IM_DATABUS;

```

```

    PCR[17:16] = IRX[9:8];
    end
    else if(INCPC)
    PC=PC+1;
    else if(LIR&&LPCS)
    begin
    PCS=PCS_DATABUS+2;
    IR=IM_DATABUS;
    PC=PC+1;
    end
    else if(LIR)
    begin
    IR=IM_DATABUS;
    PC=PC+1;
    end
    else if(LPCS)
    PCS=PCS_DATABUS+2;
    else if(OPCS)
    PC=PCS_DATABUS;
    else if(OSTACKPC)
    PC=STACKPC;
    else if(RESET)
    PC=18'b0;
    else if(IF0)
    begin
    STACKPC=PC-1;
    PC=18'd15;
    end
    else if(IF1)
    begin
    STACKPC=PC-1;
    PC=18'd17;
    end
    else if(TMF0)
    begin
    STACKPC=PC-1;
    PC=18'd15;
    end
    end
end
always@(posedge clk)
begin
    IRX=IR;
end

// data memory instantiation
data_memory
a2(.clk(DM_CLK),.RDDM(RDDM),.WRDM(WRDM),.DM_ADDRBUS(DM_ADDRBUS),.SYSTEM_DATABUS(SYSTEM_DATABUS));

//alu unit instantiation
alu_unit
alusys(.clk(clk),.R_CLK(R_CLK),.EAND(EAND),.EXOR(EXOR),.EOR(EOR),.EADD(EADD),.ESUB(ESUB),.SYSTEM_DATABUS(SYSTEM_DATABUS),.acc(acc),.OALU(OALU),.LALU(LALU),.OA(OA),.LA(LA),.INCA(INCA),.DECA(DECA),.CMA(CMA),.RR(RR),.RL(RL),.PORT0(PORT0),.PORT1(PORT1),.RESET(RESET),.LIN(LIN),.OOUT(OOUT),.SETCLRF(SETCLRF),.OER0(OER0),.OER1(OER1),.OER2(OER2),.OER3(OER3),.OER4(OER4),.OER5(OER5))

```

```
,.OER6(OER6),.OER7(OER7),.LR0(LR0),.LR1(LR1),.LR2(LR2),.LR3(LR3),.LR4(LR4),.LR5(LR5),.LR6(LR6),.LR7(LR7),.OERAL0(OERAL0),.OERAL1(OERAL1),.OERAL2(OERAL2),.OERAL3(OERAL3),.OERAL4(OERAL4),.OERAL5(OERAL5),.OERAL6(OERAL6),.OERAL7(OERAL7),.flagreg(ZCBP),.baudclk(baudclk),.c(c),.rxin(rxin),.txout(txout),.LSIN(LSIN),.OSOUT(OSOUT),.RXF(RXF),.TXF(TXF));
```

```
//interrupt module instantiation
```

```
interrupt_module
```

```
inter(.clk(clk),.IO(I0),.I1(I1),.LINTCON(LINTCON),.IF0(IF0),.IF1(IF1),.TMF0(TMFO),.CLRTMRF(CLRMTMF),.SYSTEM_DATABUS(SYSTEM_DATABUS));
```

```
endmodule
```

```
/* Tstate counter */
```

```
module tstate_counter(
```

```
input clk,IF0,IF1,TMF0,TXF,RXF,
```

```
//input clock 25MHz
```

```
input [7:2] IR8,
```

```
output reg TF1=0,TX1=0,TX2=0);
```

```
reg [3:0]counter=5'b00;
```

```
reg [7:0] c = 8'b0;
```

```
always@(posedge clk)
```

```
begin
```

```
case(counter)
```

```
5'b0:begin
```

```
TF1=1'b1;
```

```
TX1=1'b0;
```

```
TX2=1'b0;
```

```
counter=5'b00001;
```

```
end
```

```
5'b00001:begin
```

```
if(IF0)
```

```
//check for interrupt flag IF0
```

```
begin
```

```
TF1=1'b0;
```

```
TX1=1'b0;
```

```
counter=5'b00011;
```

```
end
```

```
else if(IF1)
```

```
//check for interrupt flag IF1
```

```
begin
```

```
TF1=1'b0;
```

```
TX1=1'b0;
```

```
counter=5'b00011;
```

```
end
```

```
else if(TMFO)
```

```
//check for timer interrupt flag TMFO
```

```
begin
```

```
TF1=1'b0;
```

```
TX1=1'b0;
```

```
counter=5'b00011;
```

```
end
```

```
else if(IR8[7:2]==6'b110101)
```

```
//check for wait TXF instruction
```

```
begin
```

```
TF1=1'b0;
```

```
TX1=1'b0;
```

```
counter=5'b01000;
```

```
end
```

```
else if(IR8[7:2]==6'b110110)
```

```
// check for wait RXF instruction
```

```
begin
```

```

    TF1=1'b0;
    TX1=1'b0;
    counter=5'b01001;
    end
    else if(IR8[7:6]==2'b10)           //check for branching instruction
    begin
        TF1=1'b0;
        TX1=1'b1;
        counter=5'b00010;
        end
    else if(IR8[7:2]==6'b110011)      //check for RESTORE PC instruction
    begin
        TF1=1'b0;
        TX1=1'b1;
        counter=5'b0;
        end
    else if(IR8[7:2]==6'b110100)      //check for return interrupt instruction RETI
    begin
        TF1=1'b0;
        TX1=1'b1;
        counter=5'b0;
        end
    else if(IR8[7:2]==6'b110000)      // check for halt instruction
    begin
        TF1=1'b0;
        TX1=1'b0;
        end
    else
    begin
        TF1=1'b1;
        TX1=1'b1;
        counter=5'b00001;
        end
    end
5'b00010:begin
    TF1=1'b0;
    TX1=1'b0;
    TX2=1'b1;
    counter=5'b0;
    end
5'b00011:begin
    TF1=1'b1;
    TX1=1'b0;
    TX2=1'b0;
    counter=5'b00100;
    end
end
5'b00100:begin
    if(IR8[7:6]==2'b10)
    begin
        TF1=1'b0;
        TX1=1'b1;
        counter=5'b00010;
        end
    else if(IR8[7:2]==6'b110011)
    begin
        TF1=0;
        TX1=1'b1;

```

```

        counter=5'b0;
        end
        else if(IR8[7:2]==6'b110100)
        begin
            TF1=1'b0;
            TX1=1'b1;
            counter=5'b0;
        end
        else
        begin
            TF1=1'b1;
            TX1=1'b1;
            counter=5'b00101;
        end
        end
5'b00101:begin
    if(IR8[7:6]==2'b10)
    begin
        TF1=1'b0;
        TX1=1'b1;
        counter=5'b00010;
    end
    else if(IR8[7:2]==6'b110011)
    begin
        TF1=1'b0;
        TX1=1'b1;
        counter=5'b0;
    end
    else if(IR8[7:2]==6'b110100)
    begin
        TF1=1'b0;
        TX1=1'b1;
        counter=5'b0;
    end
    else
    begin
        TF1=1'b1;
        TX1=1'b1;
        counter=5'b00110;
    end
    end
    end
5'b00110:begin
    if(IR8[7:6]==2'b10)
    begin
        TF1=1'b0;
        TX1=1'b1;
        counter=5'b00010;
    end
    else if(IR8[7:2]==6'b110011)
    begin
        TF1=1'b0;
        TX1=1'b1;
        counter=5'b0;
    end
    else if(IR8[7:2]==6'b110100)
    begin
        TF1=1'b0;

```

```

        TX1=1'b1;
        counter=5'b0;
    end
    else
    begin
        TF1=1'b1;
        TX1=1'b1;
        counter=5'b00111;
    end
end
5'b00111:begin
    if(IR8[7:6]==2'b10)
    begin
        TF1=1'b0;
        TX1=1'b1;
        counter=5'b00010;
    end
    else if(IR8[7:2]==6'b110011)
    begin
        TF1=1'b0;
        TX1=1'b1;
        counter=5'b0;
    end
    else if(IR8[7:2]==6'b110100)
    begin
        TF1=1'b0;
        TX1=1'b1;
        counter=5'b0;
    end
    else
    begin
        TF1=1'b1;
        TX1=1'b1;
        counter=5'b0;
    end
end
5'b01000:begin
    if(IF0)
    begin
        TF1=1'b0;
        TX1=1'b0;
        counter=5'b00011;
    end
    else if(IF1)
    begin
        TF1=1'b0;
        TX1=1'b0;
        counter=5'b00011;
    end
    else if(TMFO)
    begin
        TF1=1'b0;
        TX1=1'b0;
        counter=5'b00011;
    end
    else if(TXF && c==217)
    begin

```



```

        TF1=1'b0;
        TX1=1'b0;
        counter=5'b0;
    end
    else
    begin
        TF1=1'b0;
        TX1=1'b0;
        counter=5'b01000;
    end
    end

5'b01001:begin
    if(IF0)
    begin
        TF1=1'b0;
        TX1=1'b0;
        counter=5'b00011;
    end
    else if(IF1)
    begin
        TF1=1'b0;
        TX1=1'b0;
        counter=5'b00011;
    end
    else if(TMFO)
    begin
        TF1=1'b0;
        TX1=1'b0;
        counter=5'b00011;
    end
    else if(RXF && c==217)
    begin
        TF1=1'b0;
        TX1=1'b0;
        counter=5'b0;
    end
    else
    begin
        TF1=1'b0;
        TX1=1'b0;
        counter=5'b01001;
    end
    end
endcase
end
always@(posedge clk)
begin
    if(c==217)
    c=0;
    else
    c=c+1;
end
end

endmodule

```

```

/* Data Memory */

module data_memory(
input clk,                                //gated clock for data memory
input RDDM,WRDM,
input [11:0] DM_ADDRBUS,
inout [7:0] SYSTEM_DATABUS);
reg [7:0] DM [0:4095];
assign SYSTEM_DATABUS = RDDM?DM[DM_ADDRBUS]:'bz;
always@(negedge clk)
if(WRDM)
DM[DM_ADDRBUS]=SYSTEM_DATABUS;
endmodule

/* ALU Unit and Flag register */

module alu_unit(
input clk,                                //input clock 25MHz
input R_CLK,                              //gated-clock for register set
input [7:0] PORT0,
output [7:0] PORT1,
input RESET,LIN,OOOUT,
input LR0,LR1,LR2,LR3,LR4,LR5,LR6,LR7,OER0,OER1,OER2,OER3,OER4,OER5,OER6,OER7,
input OERAL0,ERAL1,ERAL2,ERAL3,ERAL4,ERAL5,ERAL6,ERAL7,
input OALU,LALU,OA,LA,INCA,DECA,CMA,RR,RL,
input EAND,EXOR,EOR,EADD,ESUB,SETCLRF,

inout [7:0] SYSTEM_DATABUS,
output [7:0] ALU_DATABUS_A,
output [7:0] ALU_DATABUS_B,
output [7:0] acc,
inout [8:0] result,
output reg [3:0] flagreg=4'b0,
input baudclk,
input c,
input rxin,
input LSIN,OSOUT,
output txout,
output RXF,TFX); // Z C BF P
wire Z,C,B,P;

//accumulator instantiation
accumulator
acal(.clk(clk),.OA(OA),.LA(LA),.OALU(OALU),.LALU(LALU),.SYSTEM_DATABUS(SYSTEM_DATABUS),.ALU_DATABUS_A(ALU_DATABUS_A),.result(result),.Acc(acc),.INCA(INCA),.DECA(DECA),.CMA(CMA),.RR(RR),.RL(RL),.RESET(RESET),.LIN(LIN),.OOOUT(OOOUT),.PORT0(PORT0),.PORT1(PORT1),.baudclk(baudclk),.c(c),.rxin(rxin),.txout(txout),.LSIN(LSIN),.OSOUT(OSOUT),.RXF(RXF),.TFX(TFX));

//register set instantiation
register_set
rxal(.clk(R_CLK),.LR0(LR0),.LR1(LR1),.LR2(LR2),.LR3(LR3),.LR4(LR4),.LR5(LR5),.LR6(LR6),.LR7(LR7),.RESET(RESET),.OER0(OER0),.OER1(OER1),.OER2(OER2),.OER3(OER3),.OER4(OER4),.OER5(OER5),.OER6(OER6),.OER7(OER7),.ERAL0(ERAL0),.ERAL1(ERAL1),.ERAL2(ERAL2),.ERAL3(ERAL3),.ERAL4(ERAL4),.ERAL5(ERAL5),.ERAL6(ERAL6),.ERAL7(ERAL7),.SYSTEM_DATABUS(SYSTEM_DATABUS),.ALU_DATABUS_B(ALU_DATABUS_B));

// Flag register update

```

```

assign result = (EAND|EXOR|EOR|EADD|ESUB)?((EAND?ALU_DATABUS_A &
ALU_DATABUS_B:9'b0)|((EXOR?ALU_DATABUS_A^ALU_DATABUS_B:9'b0)|((EOR?ALU_DATABUS_A|ALU
_DATABUS_B:9'b0)|((EADD?ALU_DATABUS_A+ALU_DATABUS_B:9'b0)|((ESUB?ALU_DATABUS_A-
ALU_DATABUS_B:9'b0))):'bz;
assign C = EADD?result[8]:'bz;
assign B = ESUB?result[8]:'bz;
assign Z = ((result==9'b0)||((result==9'b100000000)))?1'b1:1'b0;
assign P = result[7]+result[6]+result[5]+result[4]+result[3]+result[2]+result[1]+result[0];

```

```
//ALU operations
```

```
always@(negedge clk)
```

```
begin
```

```
    if(EAND)
```

```
        begin
```

```
            flagreg[3]=Z;
```

```
            flagreg[0]=P;
```

```
        end
```

```
    else if(EXOR)
```

```
        begin
```

```
            flagreg[3]=Z;
```

```
            flagreg[0]=P;
```

```
        end
```

```
    else if(EOR)
```

```
        begin
```

```
            flagreg[3]=Z;
```

```
            flagreg[0]=P;
```

```
        end
```

```
    else if(EADD)
```

```
        begin
```

```
            flagreg[3]=Z;
```

```
            flagreg[2]=C;
```

```
            flagreg[0]=P;
```

```
        end
```

```
    else if(ESUB)
```

```
        begin
```

```
            flagreg[3]=Z;
```

```
            flagreg[1]=B;
```

```
            flagreg[0]=P;
```

```
        end
```

```
    else if(SETCLRF)
```

```
        begin
```

```
            flagreg=SYSTEM_DATABUS[3:0];
```

```
        end
```

```
end
```

```
endmodule
```

```
/* Accumulator, I/O Module and Serial Module*/
```

```
module accumulator(
```

```
    input clk,           //input clock 25MHz
```

```
    input RESET,
```

```
    input [7:0] PORT0,
```

```
    output reg [7:0] PORT1,
```

```
    input LIN, OOUT,
```

```
    input LA, OA, OALU, LALU, INCA, DECA, CMA, RR, RL,
```

```

inout [7:0] SYSTEM_DATABUS,
inout [8:0] result,
output [7:0] ALU_DATABUS_A,
input baudclk,           //baud clock 115200 per second
input c,
input rxin,
input LSIN,OSOUT,
output reg txout=1'b1,
output reg RXF=1'b0,TXF=1'b0,
output reg [7:0] Acc=8'b0;
reg [7:0] data;
reg [7:0] RBUFF;
reg [7:0] TBUFF;
reg [3:0] countrx=4'b1001;
reg [3:0] counttx=4'b1001;
reg txb=1'b1;

//accumulator operations
assign SYSTEM_DATABUS = OA?Acc:'bz;
assign ALU_DATABUS_A = OALU?Acc:'bz;
wire temp;
assign temp = (RR|RL)?((RR?Acc[0]:1'b0)|(RL?Acc[7]:1'b0)):'bz;

always@(negedge clk)
begin
    if(LA)
        Acc=SYSTEM_DATABUS;
    else if(LALU)
        Acc = result[7:0];
    else if(INCA)
        Acc = Acc+1;
    else if(DECA)
        Acc = Acc-1;
    else if(CMA)
        Acc = ~Acc;
    else if(RR)
        begin
            Acc = Acc>>1;
            Acc[7] = temp;
        end
    else if(RL)
        begin
            Acc = Acc<<1;
            Acc[0] = temp;
        end
    else if(RESET)
        Acc=8'b0;
    else if(LIN)
        Acc=PORT0;           //input port PORT0
    else if(OOUT)
        PORT1=Acc;           //output port PORT1
    else if(LSIN)
        Acc=RBUFF;           //RBUFF register in serial module
    else if(OSOUT)
        begin
            TBUFF=Acc;       //TBUFF register in serial module
            txb=1'b0;
        end
end

```

```

        end
    else if(TXF & c)
        txb=1'b1;
    end

end

//serial module receiver
always@(posedge baudclk)
begin
    if(rxin==0 & countrx==9)
        begin
            if(countrx)
                begin
                    RXF=1'b0;
                    RBUFF=8'b0;
                    RBUFF=RBUFF>>1;
                    RBUFF[7]=rxin;
                    countrx=countrx-1;
                end
            end
        else if(countrx<=8 & countrx>0)
            begin
                RBUFF=RBUFF>>1;
                RBUFF[7]=rxin;
                countrx=countrx-1;
            end
        else if(countrx==0)
            begin
                RXF=1'b1;
                countrx=4'b1001;
            end
        else
            begin
                RXF=1'b0;
                countrx=4'b1001;
            end
        end

end

//serial module transmitter
always@(posedge baudclk)
begin
    if ((counttx<=8)&&(counttx>=1))
        begin
            txout=data[0];
            data[7:0]=data[7:0]>>1;
            counttx=counttx-1;
            TXF=1'b0;
        end
    else if(counttx==0)
        begin
            txout=1'b1;
            counttx=4'b1001;
            TXF=1'b1;
        end
    else if(txb==0)
        begin
            txout=1'b0;
            data=TBUFF;
        end
end

```

```

        counttx=counttx-1;
        TXF=1'b0;
        end
    else
        begin
            txout=1'b1;
            counttx=4'b1001;
            TXF=1'b0;
            end
    end
endmodule

```

```

/* Register set */

module register_set(
    input clk,                //gated clock for register set
    input RESET,
    input LR0,LR1,LR2,LR3,LR4,LR5,LR6,LR7,
    input OER0,OER1,OER2,OER3,OER4,OER5,OER6,OER7,
    input OERAL0,ERAL1,ERAL2,ERAL3,ERAL4,ERAL5,ERAL6,ERAL7,
    inout [7:0] SYSTEM_DATABUS,
    output [7:0] ALU_DATABUS_B);
    reg [7:0] R0,R1,R2,R3,R4,R5,R6,R7;

    assign                SYSTEM_DATABUS                =
(OER0|OER1|OER2|OER3|OER4|OER5|OER6|OER7)?((OER0?R0:8'b0)|(OER1?R1:8'b0)|(OER2?R2:8'b0)
|(OER3?R3:8'b0)|(OER4?R4:8'b0)|(OER5?R5:8'b0)|(OER6?R6:8'b0)|(OER7?R7:8'b0)):'bz;
    assign                ALU_DATABUS_B                =
(ERAL0|ERAL1|ERAL2|ERAL3|ERAL4|ERAL5|ERAL6|ERAL7)?((ERAL0?R0:8'b0)|(ERAL1?R
1:8'b0)|(ERAL2?R2:8'b0)|(ERAL3?R3:8'b0)|(ERAL4?R4:8'b0)|(ERAL5?R5:8'b0)|(ERAL6?R6:8'b0)|
(ERAL7?R7:8'b0)):'bz;

    always@(negedge clk)
    begin
        if(LR0)
            R0=SYSTEM_DATABUS;
        else if(LR1)
            R1=SYSTEM_DATABUS;
        else if(LR2)
            R2=SYSTEM_DATABUS;
        else if(LR3)
            R3=SYSTEM_DATABUS;
        else if(LR4)
            R4=SYSTEM_DATABUS;
        else if(LR5)
            R5=SYSTEM_DATABUS;
        else if(LR6)
            R6=SYSTEM_DATABUS;
        else if(LR7)
            R7=SYSTEM_DATABUS;
        else if(RESET)
            begin
                R0=8'b0;
                R1=8'b0;
                R2=8'b0;
                R3=8'b0;
                R4=8'b0;
            end
    end
endmodule

```



```

                                begin
                                IF0=1'b0;
                                IF1=1'b0;
                                end
                                end
                                end
else
    begin
    IF0=1'b0;
    IF1=1'b0;
    end
end

//timer interrupt handling
always@(negedge clk)
begin
    if(CLRTMRF && INTCON[2])
        begin
            TMF0=1'b0;
            timer=timer+1;
        end
    else if(CLRTMRF)
        begin
            TMF0=1'b0;
        end
    else if(INTCON[2])
        begin
            if(timer==1023)
                begin
                    TMF0=1'b1;
                    timer=timer+1;
                end
            else
                begin
                    timer=timer+1;
                end
        end
end
endmodule

```


Chapter 4

RESULTS AND DISCUSSIONS

The simulation has been performed using ISim. Simulation has been performed for each module and for the whole processor. Initially simulation is performed on each module for verifying functionality of control signals corresponding to the respective module. Figure 4.1 represents simulation of Program Counter Unit. It shows that, when OPC is high, IM_ADDRBUS gets PC content. When LIR is high and at falling edge of clock, PC content is incremented. In figure 4.1, when LPCS control signal is enabled, PCS stores the value 4, which is Program Counter (2) content incremented by two value. When OPCS control signal is high and at falling edge of clock, PC gets the value 4, which is PCS content. When OSTACK control signal is high and at falling edge of clock, PC is loaded by the value 7, which is STACKPC content.

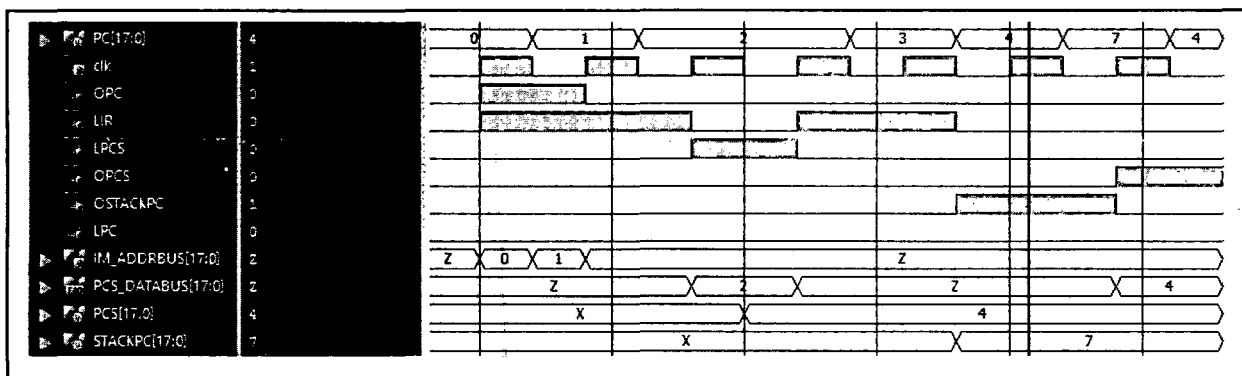


Figure 4.1 Simulation of Program Counter Unit

Figure 4.2 represents the simulation of Instruction Memory. When RDIM is high, IM_DATABUS gets the Instruction Memory content corresponding to the address provided by IM_ADDRBUS. For example, when RDIM is high, IM_ADDRBUS content is 00 0000 0000 0000 0001, IM_DATABUS gets 0000 1000 0000 0110, which is corresponding to address location 00 0000 0000 0000 0001 of Instruction Memory.

Figure 4.3 represents the simulation of Data Memory. It shows that whenever RDIM is high, SYSTEM_DATABUS gets the Data Memory content corresponding to the address provided by DM_ADDRBUS. When WRDM is high, Data Memory gets SYSTEM_DATABUS content, corresponding to address provided by DM_ADDRBUS. For example, when WRDM is high and SYSTEM_DATABUS content is 10101010,

When RR is high and at falling edge of clock, accumulator becomes 00011001, which is accumulator shifted right by one bit. When CMA is high and falling edge of clock, accumulator becomes 11100110, which is complemented value of accumulator. When LIN is high and at falling edge of clock, accumulator becomes 00110011, which is input port PORT0 content. When LALU is high and at falling edge of clock, accumulator becomes 00000011, which is result content (ALU_RESULT). When RESET is high and at falling edge of clock, accumulator becomes 00000000, which is the default reset.

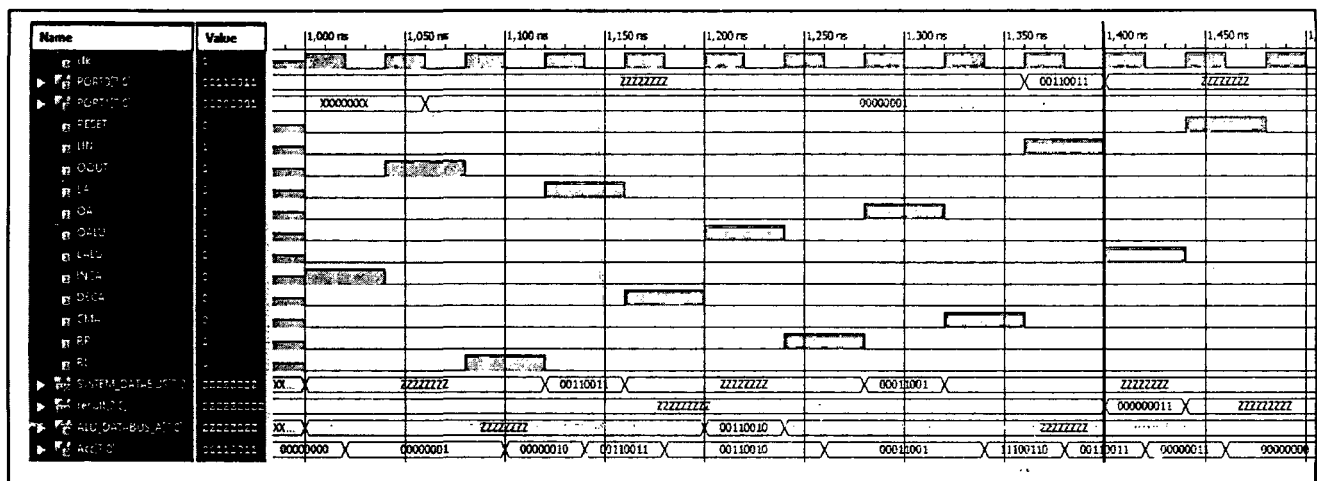


Figure 4.4 Simulation of accumulator

Figure 4.5 represents simulation of Register set. In Figure 4.5, registers R0 - R7 is initiated to values 0 - 7 respectively. When LRX is high (where X is 0 - 7) and at falling edge of clock, corresponding register is loaded by SYSTEM_DATABUS content. For example, When LR1 is high and at falling edge of clock, register R1 becomes 10101010, which is SYSTEM_DATABUS content. When OERX is high (where X is 0 - 7), SYSTEM_DATABUS gets corresponding register content. When OER4 is high, SYSTEM_DATABUS becomes 00000011, which is register R3 content. When OERALX is high (where X is 0 - 7), ALU_DATABUS_B gets corresponding register content. When OERAL7 is high, ALU_DATABUS_B becomes 00000111, which is register R7 content.

Figure 4.7 represents simulation of Interrupt module part 1. When LINTCON is high and at falling edge of clock, INTCON register becomes 110, which is SYSTEM_DATABUS content. When INTCON register's MSB bit is 1 (bit 2), timer interrupt will be enabled. When bit 1 of INTCON register is 1, both external interrupts will be enabled. When LSB bit (bit 0) of INTCON register is 0, interrupt-I1 is not masked, thus all three interrupts are enabled. In Figure 4.7, when I0 interrupt is high, IF0 flag is set. When I1 interrupt is high, IF1 flag is set. When both I0 and I1 are high, only IF0 flag is set because it is having highest priority. It shows that timer flag TMF0 is low and timer register is incrementing in each clock cycle as timer interrupt is enabled.

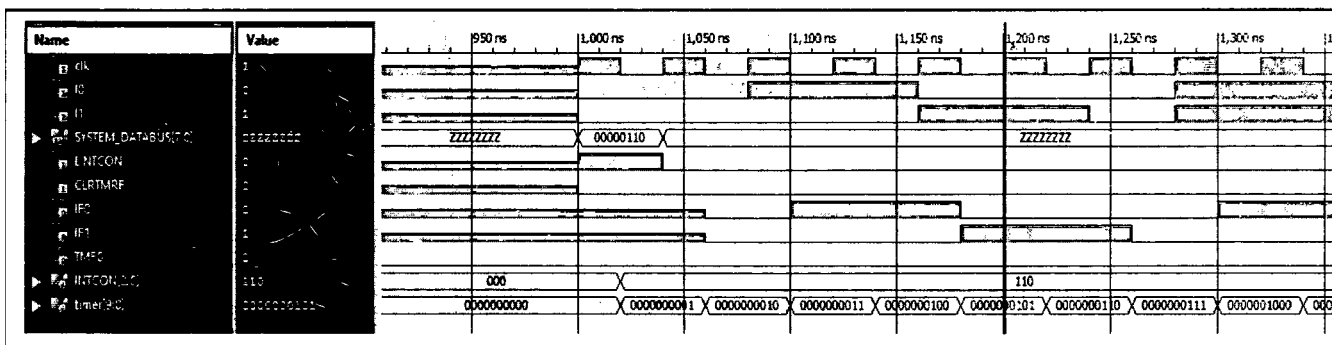


Figure 4.7 Simulation of interrupt module part 1

Figure 4.8 represents simulation of Interrupt module part 2. In Figure 4.8, when timer register reaches its maximum value 1023, timer flag TMF0 is set. When CLRTMRf is high and at falling edge of clock, TMF0 is cleared.

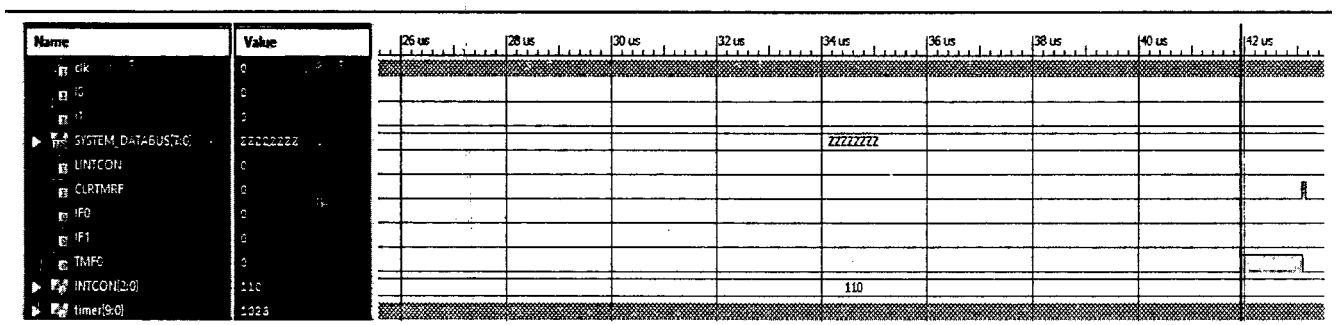


Figure 4.8 Simulation of interrupt module part 2

Figure 4.9 represents simulation of Serial Module. In Figure 4.9, rxin is input serial port and txout is output serial port. Both ports are 1 at idle state. In Figure 4.9, clk is the system clock at 25MHz and baudclk is baud rate required for serial communication, which is 115200 per second. When rxin becomes 0, by UART protocol start bit has arrived which is followed by 8-bit data. RXF flag is set when reception is completed, which will be high for one baudclk. The 8-bit received data is stored in RBUFF register.

When LSIN is high and at falling edge of clock, accumulator becomes 11011100, which is RBUFF content. When OSOUT is high and at falling edge of clock, TBUFF becomes 11011100, which is accumulator content. From next rising edge of baudclk, txout will transmit the 8-bit data on TBUFF (11011100) serially by UART protocol. After completion of transmission TXF flag is set, which will be high for one baudclk.

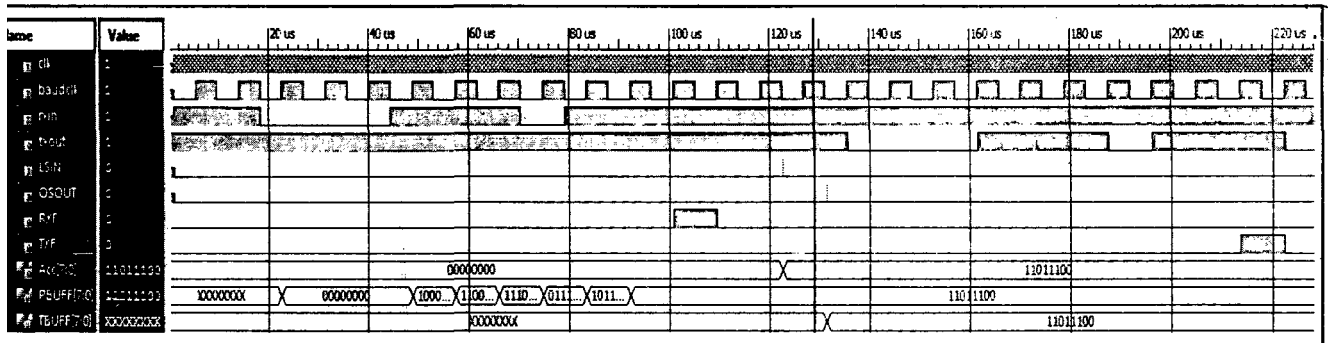


Figure 4.9 Simulation of serial module

Figure 4.10 represents the simulation of control unit. In figure 4.10, Instruction Memory IM is having data on 9 address locations (0-8). By referring to appendix A, the Opcode corresponding to each address location can be find out. Instructions are MOVI A, 00000110 (IM [0]), MOV R6, A (IM [1]), MOVI A, 00000101 (IM [2]), ADD A, R6 (IM [3]), JUMP 00 00000000000000111 (IM [4] and IM [5]), DEC A (IM [6]), INC A (IM [7]), HALT (IM [8]). By referring to appendix B, control signals corresponding to each instruction and in which tstate it is occurred, can be find out. In Figure 4.10, TF1 represents fetch cycle, TX1 represents execution cycle 1 and TX2 represents execution cycle 2. Instruction in IM [0] is having OPC, RDIM and LIR in TF1 and OIRSYS and LA in TX1 cycle. Instruction in IM [1] is having OPC, RDIM and LIR in TF1 and LR6 and OA in TX1 cycle. As control signals in fetch cycle is common for all, from next instruction onwards control signals in execution cycle (TX1 and TX2) will be considered. Instruction in IM [2] is having OALU, OERAL6, EADD and LALU in TX1 cycle. Instruction in IM [4] and IM [5] (JUMP) is having OPC, RDIM, OIRPC and LPCR in TX1 cycle and LPC and OPCR in TX2 cycle. Instruction in IM [6] is having DECA in TX1 cycle and instruction in IM [7] is having INCA in TX1 cycle. IM [8] is

next clock cycle, IR content is loaded to IRX register, which is used for decoding the instruction and generating control signals corresponding to the Opcode provided. As IR content is changed at every falling edge of clock and we need Opcode to be stable for one full clock cycle for proper execution of instruction by pipelining architecture, IR content is transferred to IRX. During execution cycle (TX1) OALU, OERAL6, EADD and LALU are high. As OALU is high, ALU_DATABUS_A gets accumulator content (00000101) and as OERAL6 is high, ALU_DATABUS_B gets register R6 content. As EADD is high, result of addition is on ALU_RESULT (00001101). When LALU is high and at falling edge of clock, accumulator (acc) becomes 00001101, which is ALU_RESULT content. In flag register ZCBP, zero flag (Z), carry flag (C) and parity flag (P) are updated. In Figure 4.10, TX1 and TF1 are high during same time for most part of execution, which shows the pipelining architecture and TX2 is high only during the execution of JUMP instruction (branching instruction), which is an exception to the pipelining architecture. In Figure 4.10, DT, ALU, BR and MIO represents type of instruction, which are data transfer, arithmetic and logical, branching and machine control and I/O instructions respectively.

The Table 4.1 presents a simple twenty-line assembly language program for simulation. In Table 4.1, IM addr represents instruction memory address and ISR is Interrupt Service Routine. The Table 4.2 shows the device utilization of the Spartan-3E Starter kit FPGA board. The Figure 4.11, 4.12 and 4.13 shows the simulation results of the proposed 8-bit RISC processor with pipeline architecture for the program in Table 4.2.

Simulation results in Figure 4.11 shows the control signals and clock signals corresponds to Register set and Data Memory. R_CLK and DM_CLK are the gated-clock inputs corresponding to Register set and Data Memory, which are activated only when instruction corresponding to loading a memory/general-purpose register is fetched.

In Figure 4.12, Accumulator is represented as acc with initial value 01010010 and flag register is represented as ZCBP. It also shows interrupt flags and relevant buses used by the processor for executing the given program. It shows that the STACKPC stores current PC value whenever an interrupt flag is raised, PCS stores address of next instruction after jump instruction (11) and PCR store address of the next instruction to be executed (19), while a branching instruction is being executed.

In Figure 4.13, DT, ALU, BR and MIO represents type of instruction, which are data transfer, arithmetic and logical, branching and machine control and I/O instructions respectively. The tstates TF1, TX1 and TX2 are also shown, which behaves in different manner, while a branching instruction is being executed or an interrupt flag is raised.

Control Unit will generate 59 control signals and 4 clock signals while simulating the program in Table 4.1, but for convenience, by Figure 4.11, 4.12 and 4.13, major part of total simulation is shown.

Table 4.1 Twenty-instruction program for simulation

IM addr	Opcode	Mnemonic	Description
0	0111000000000010	EDINTER 010	Enable interrupts I0 & I1
1	0000100000000110	MOVI A, 00000110	00000110 => A
2	0001011000000000	MOV R6, A	A => R6
3	0000100000000101	MOVI A, 00000101	00000101 => A
4	0001010100000000	MOV R5, A	A => R5
5	0000100000000011	MOVI A, 00000011	00000011 => A
6	0001001100000000	MOV R3, A	A => R3
7	0100011110000000	ADD A, R6	A+R6 => A
8	1100100000000000	SAVE PC	PC => PCS
9	1000000000000000	JUMP 00	Jump to address 19
10	0000000000010011	0000000000010011	
11	0100100011000000	SUB A, R3	A-R3 => A
12	0101100000000000	RL	Rotate A left by 1 bit
13	0011000000000001	MOV 000000000001, A	A => DM[000000000001]
14	1100000000000000	HALT	Stop operations
15	0000100000000111	MOVI A, 00000111	00000111 => A
16	1101000000000000	RETI	Return from ISR
17	0000100000000010	MOVI A, 00000010	00000010 => A
18	1101000000000000	RETI	Return from ISR
19	0100011101000000	ADD A, R5	A+R5 => A
20	1100110000000000	RES PC	PCS => PC

Table 4.2 Device utilization of the Spartan-3E Starter kit FPGA board

Logic Utilization	Used	Available	Utilization
Number of Slice Flip-Flops	291	9312	3%
Number of 4 input LUTs	3374	9312	36%
Number of occupied Slices	1821	4656	39%
Number of bonded IOBs	166	232	71%

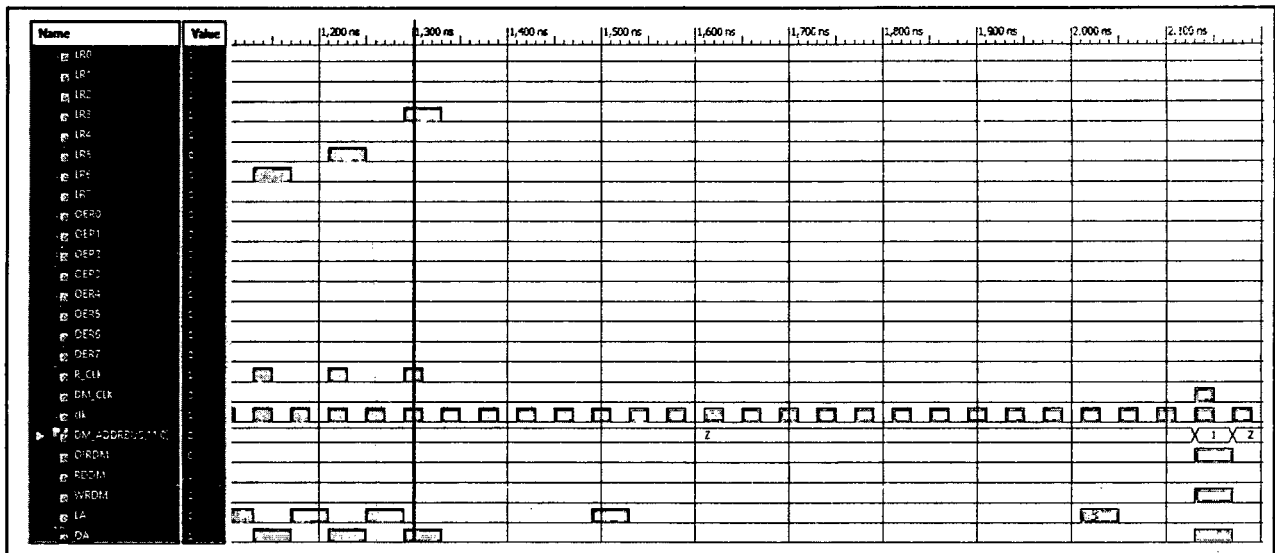


Figure 4.11 Simulation results of proposed processor part I

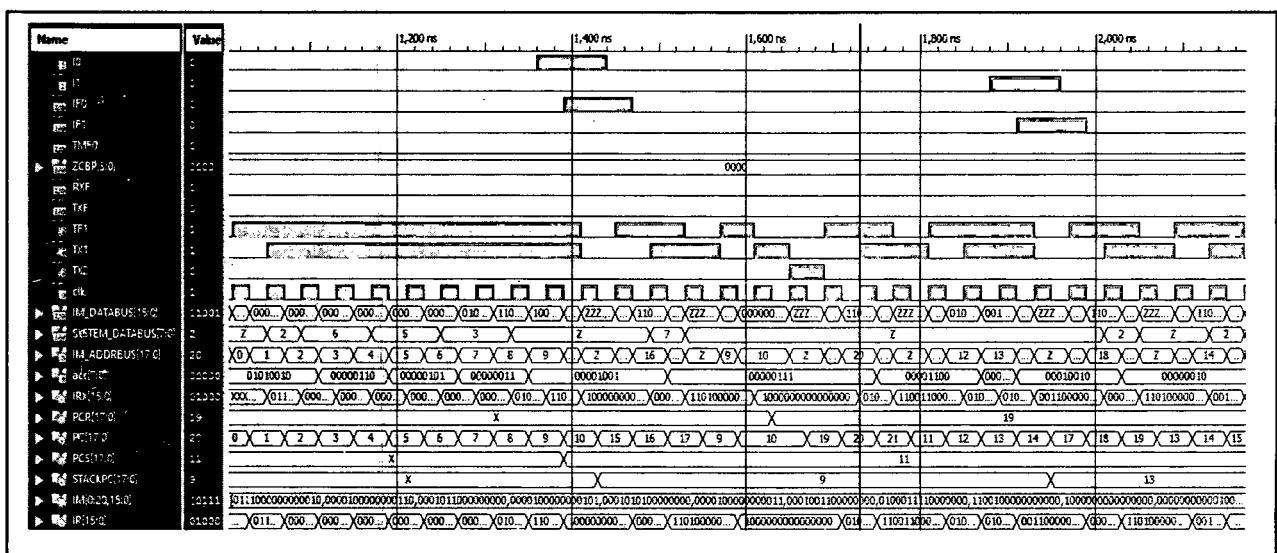
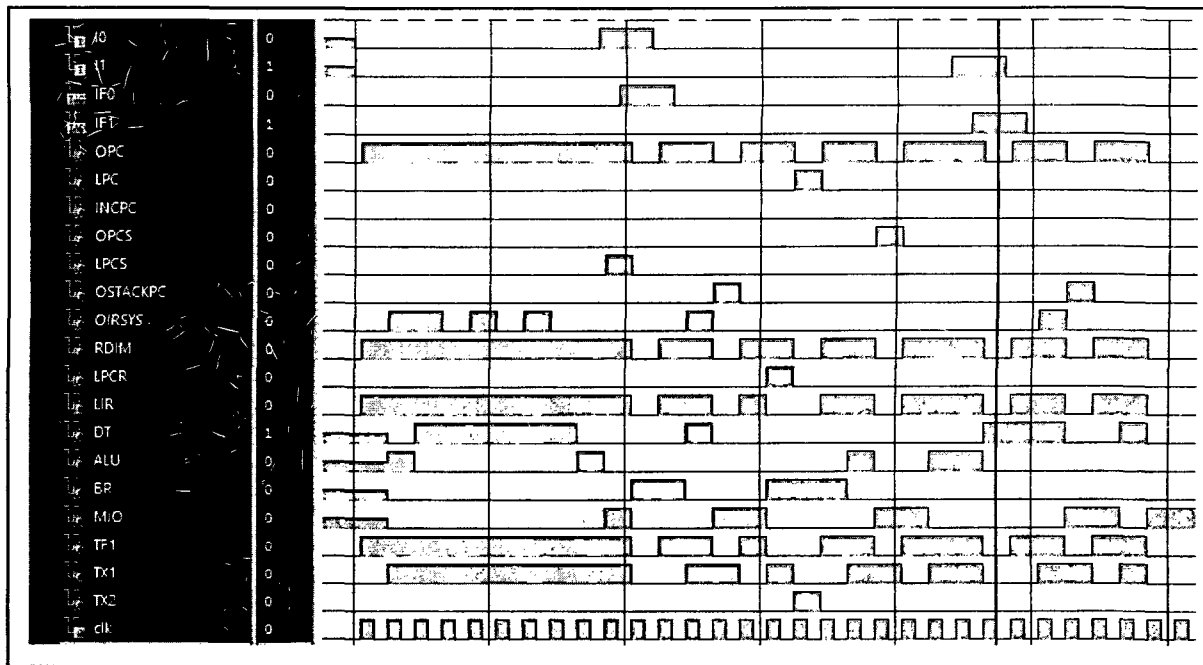


Figure 4.12 Simulation results of proposed processor part II



Chapter 5

CONCLUSION

FPGA based 8-bit RISC processor with pipelining and clock-gating technique is designed. ISim is used for performing simulation. The design is implemented on Xilinx Spartan-3E Starter kit FPGA board at 25MHz, on which all 34 instructions are verified. Pipelining technique yields better performance as the processor executes one instruction on every clock cycle. Clock gating applied to specific modules helps in reduction of power to a certain extent. Serial communication using UART protocol is successfully carried out at a baud rate of 115200. The enhanced feature of Xilinx Spartan-3E voluntarily reduces the cost per logic cell designed. RISC processor's range of application includes signal processing, convolution application, commercial data processing, used in supercomputers such as the K computer, smart phones, tablets and real-time embedded systems.

REFERENCES

- [1] Sivarama P.Dandamudi, "A Guide To RISC Processor For Programmers And Engineers", Springer.
- [2] Sivarama P.Dandamudi, "Introduction to Assembly Language Programming For Pentium and RISC Processors", Springer.
- [3] J. Hennessy and D. Patterson, Computer Architecture: A Quantitative Approach, 4th ed., 2007.
- [4] J. Ball, "Designing Soft-Core Processors for FPGAs Processor Design", in Processor Design: System-on-Chip Computing for ASICs and FPGAs, J. Nurmi, 1st Ed: Springer Netherlands, 2007, pp. 229-256.
- [5] Jagrit Kathuria, M.Ayoub khan, Arti noor, "A review of clock gating techniques", International Journal of Electronics and Communication Engineering, Vol. 1, No. 2 pp.106-114, Aug 2011.
- [6] Dr.M.Kamaraju, G.Chinavenkateswararao, "Low Power Reduced Instruction Set Architecture Using Clock Gating Technique", International Journal of VLSI design & Communication Systems (VLSICS), Vol.4, No.5, pp.35-51, October 2013.
- [7] F. Y. Yuan; C. Xue-jun, "Design and Simulation of UART Serial Communication Module Based on VHDL," Intelligent Systems and Applications (ISA), 2011 3rd International Workshop on, vol., no., pp.1,4, 28-29 May 2011
- [8] HU Hua, BAI Feng-e. Design and Simulation of UART Serial Communication Module Based on Verilog HDL [J]. J ISUANJ I YU XIANDA IHUA 2008 Vol. 8
- [9] Antonio Hernández Zavala, Oscar Camacho Nieto, Jorge A. Huerta Ruelas, Arodí R. Carvallo Domínguez, "Design of a General Purpose 8-bit RISC Processor for Computer Architecture Learning" Computación y Sistemas, Vol. 19, No. 2, 2015, pp. 371–385.
- [10] Patrick Stakem, "4- and 8-bit Microprocessors, Architecture and History", 2 June 2013.
- [11] Intel Marketing Communications, "The 8080/8085 Microprocessor Book", 2 July1980, Wiley, ISBN.

- [12] <http://www.alldatasheet.com/datasheet-pdf/pdf/122712/INTEL/8085A.html>
- [13] https://en.wikipedia.org/wiki/Reduced_instruction_set_computing
- [14] Ralph Grishman, "Assembly Language Programming for the Control Data 6000 Series", Algorithmics Press, 1974.
- [15] G Radin, "The 801 minicomputer", Proceedings of the first international symposium on Architectural support for programming languages and operating systems, 1982, pp. 39–47.
- [16] Carlo Sequin, David Patterson, "Design and Implementation of RISC I", Proceedings of the Advanced Course on VLSI Architecture, University of Bristol, July 1982.
- [17] Ramandeep Kaur, Anuj, "8 Bit RISC Processor Using Verilog HDL", International Journal of Engineering Research and Applications (IJERA), Vol. 4, Issue 3, March 2014, pp. 417-422.
- [18] R Uma, "Design and performance analysis of 8 bit RISC processor using Xilinx tools", International Journal of Engineering Research and Applications (IJERA), March-April 2012, pp. 053-058
- [19] Ahmad Jamal Salim, Sani Irwan Md Salim, Nur Raihana Samsudin, Yewguan Soo, "Customized Instruction Set Simulation for Soft-core RISC Processor," IEEE Transactions on Control and System Graduate Research Colloquium (ICSGRC 2012)", 2012, pp. 38-42.

APPENDIX A

INSTRUCTION SET ARCHITECTURE

DATA TRANSFER INSTRUCTIONS

1. MOV A, RX ; RX => A where X = 0, 1, 2, 3, 4, 5, 6, 7 or 8

Group code		Mem/Reg	S/D	Im data	Register code			Don't care							
0	0	0	0	0	0/1	0/1	0/1	x	x	x	x	x	x	x	x

2. MOV RX, A ; A => RX

Group code		Mem/Reg	S/D	Im data	Register code			Don't care							
0	0	0	1	0	0/1	0/1	0/1	x	x	x	x	x	x	x	x

3. MOV A, M ; M => A

Group code		Mem/Reg	S/D	Data Memory address											
0	0	1	0	@	@	@	@	@	@	@	@	@	@	@	@

4. MOV M, A ; A => M

Group code		Mem/Reg	S/D	Data Memory address											
0	0	1	1	@	@	@	@	@	@	@	@	@	@	@	@

5. MOVI A, Immediate data ; Immediate data => A

Group code		Mem/Reg	S/D	Im data	Don't care			Immediate data							
0	0	0	0	1	x	x	x	#	#	#	#	#	#	#	#

A = Accumulator

M = Data Memory address

RX = Register, can be any one of R0, R1, R2, R3, R4, R5, R6 or R7

Group code = 00 represents data transfer operation

Mem = Memory => presence denoted as 1

Reg = Register => presence denoted as 0

S = Source => presence denoted as 0

D = Destination => presence denoted as 1

Im data=Immediate data =>presence denoted as 1

Register code = starts from 000 to 111

x = don't care bits

@ = address bits

= data bits

ARITHMETIC AND LOGICAL INSTRUCTIONS

1. AND A, RX

; A <= A and RX update flags

Group code		ALU/ACC/CLR/INTER op		ALU code			Reg code			Don't care					
0	1	0	0	0	0	0	0/1	0/1	0/1	x	x	x	x	x	x

2. XOR A, RX

; A <= A xor RX update flags

Group code		ALU/ACC/CLR/INTER op		ALU code			Reg code			Don't care					
0	1	0	0	0	0	1	0/1	0/1	0/1	x	x	x	x	x	x

3. OR A, RX

; A <= A or RX update flags

Group code		ALU/ACC/CLR/INTER op		ALU code			Reg code			Don't care					
0	1	0	0	0	1	0	0/1	0/1	0/1	x	x	x	x	x	x

4. ADD A, RX

; A <= A + RX update flags

Group code		ALU/ACC/CLR/INTER op		ALU code			Reg code			Don't care					
0	1	0	0	0	1	1	0/1	0/1	0/1	x	x	x	x	x	x

5. SUB A, RX

; A <= A - RX update flags

Group code		ALU/ACC/CLR/INTER op		ALU code			Reg code			Don't care					
0	1	0	0	1	0	0	0/1	0/1	0/1	x	x	x	x	x	x

6. CMA

; A <= ~A

Group code		ALU/ACC/CLR/INTER op		Acc code			Don't care								
0	1	0	1	0	0	0	x	x	x	x	x	x	x	x	x

7. INC A

; A <= A+1

Group code		ALU/ACC/CLR/INTER op		Acc code			Don't care								
0	1	0	1	0	0	1	x	x	x	x	x	x	x	x	x

ALU/ACC/CLR/INTER op

ALU = ALU operation => denoted as 00

ACC = Accumulator operation => denoted as 01

CLR = Clear operation => denoted as 10

INTER = Interrupt operation => denoted as 11

Group code = 01 represents Arithmetic and logic operation

: A <= A-1

[illegible]

```

; rotate Accumulator right by 1 bit

```

Group code		ALU/ACC/CLR/ INTER op		Acc code			Don't care								
0	1	0	1	0	1	1	x	x	x	x	x	x	x	x	x

```

; rotate Accumulator left by 1 bit

```

Group code		ALU/ACC/CLR/ INTER op		Acc code			Don't care								
0	1	0	1	1	0	0	x	x	x	x	x	x	x	x	x

; clear/set flag bits

Group code		ALU/ACC/CLR/ INTER op		clear flag/timer		Don't care						4-bit data			
0	1	1	0	0	0	x	x	x	x	x	x	#	#	#	#

```

; clear TIMER flag

```

[illegible]

- **; To control interrupt's operations**

[illegible]

BRANCHING INSTRUCTIONS

; Jump to 18-bit address add18

Group code		Instruction code			MSB bits		Don't care								
1	0	0	0	0	@	@	x	x	x	x	x	x	x	x	x

[illegible]

; Jump to 18-bit address add18 if zero flag is 1

Group code		Instruction code			MSB bits		Don't care								
1	0	0	0	1	@	@	x	x	x	x	x	x	x	x	x

[illegible]

3. JC add18 ; Jump to 18-bit address add18 if carry flag is 1

Group code		Instruction code			MSB bits		Don't care								
1	0	0	1	0	@	@	x	x	x	x	x	x	x	x	x

Instruction memory address															
@	@	@	@	@	@	@	@	@	@	@	@	@	@	@	@

4. JB add18 ; Jump to 18-bit address add18 if borrow flag is 1

Group code		Instruction code			MSB bits		Don't care								
1	0	0	1	1	@	@	x	x	x	x	x	x	x	x	x

Instruction memory address															
@	@	@	@	@	@	@	@	@	@	@	@	@	@	@	@

5. JP add18 ; Jump to 18-bit address add18 if parity flag is 1

Group code		Instruction code			MSB bits		Don't care								
1	0	1	0	0	@	@	x	x	x	x	x	x	x	x	x

Instruction memory address															
@	@	@	@	@	@	@	@	@	@	@	@	@	@	@	@

MSB Bits = 2-bit MSBs of Instruction memory address

Parity flag is set when resultant of ALU operation contains odd number of ones.

MACHINE CONTROL AND IO INSTRUCTIONS

1. HALT ; To stop operations

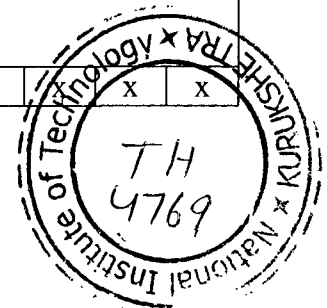
Group code		MC/IO	Instruction code			Don't care									
1	1	0	0	0	0	x	x	x	x	x	x	x	x	x	x

2. RESET ; Reset Acc, PC, RX

Group code		MC/IO	Instruction code			Don't care									
1	1	0	0	0	1	x	x	x	x	x	x	x	x	x	x

3. SAV PC ; Save PC+2 value in PCS Register

Group code		MC/IO	Instruction code			Don't care									
1	1	0	0	1	0	x	x	x	x	x	x	x	x	x	x



4. RES PC ; Restore PC with PCS content

Group code		MC/IO	Instruction code			Don't care									
1	1	0	0	1	1	x	x	x	x	x	x	x	x	x	x

5. RETI ; Restore PC with STACKPC content

Group code		MC/IO	Instruction code			Don't care									
1	1	0	1	0	0	x	x	x	x	x	x	x	x	x	x

6. WAIT TXF ; Wait until TXF flag gets set

Group code		MC/IO	Instruction code			Don't care									
1	1	0	1	0	1	x	x	x	x	x	x	x	x	x	x

7. WAIT RXF ; Wait until RXF flag gets set

Group code		MC/IO	Instruction code			Don't care									
1	1	0	1	1	0	x	x	x	x	x	x	x	x	x	x

8. IN P0 ; Accumulator gets Port P0 content

Group code		MC/IO	Instruction code			Don't care									
1	1	1	0	0		x	x	x	x	x	x	x	x	x	x

9. OUT P1 ; Accumulator sends its content to Port P1

Group code		MC/IO	Instruction code			Don't care									
1	1	1	0	1		x	x	x	x	x	x	x	x	x	x

10. SIN RBUF ; Accumulator gets RBUF content

Group code		MC/IO	Instruction code			Don't care									
1	1	1	1	0		x	x	x	x	x	x	x	x	x	x

11. SOUT TBUF ; Accumulator sends its content to TBUF

Group code		MC/IO	Instruction code			Don't care									
1	1	1	1	1		x	x	x	x	x	x	x	x	x	x

MC/IO

MC = Machine control instruction => presence denoted as 0

IO = Input/output instruction => presence denoted as 1

APPENDIX B

MICRO OPERATIONS

DATA TRANSFER INSTRUCTIONS

1. MOV A, RX

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	RX -> A
OPC RDIM LIR	OERX LA

2. MOV RX, A

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	A -> RX
OPC RDIM LIR	LRX OA

3. MOV A, M

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	IRX[11:0] -> DM DMdata -> A
OPC RDIM LIR	RDDM OIRDM LA

4. MOV M, A

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	IRX[11:0] -> DM A -> DMdata
OPC RDIM LIR	WRDM OIRDM OA

5. MOVI A, Immediate data

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	IRX[7:0] -> A
OPC RDIM LIR	OIRSYS LA

At the positive edge of every TX1 cycle of all instructions, IR content is moved to IRX for execution of instruction

ARITHMETIC AND LOGICAL INSTRUCTIONS

1. AND A, RX

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	A <- A and RX
OPC RDIM LIR	OALU OERALX EAND LALU update flags

2. XOR A, RX

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	A <- A xor RX
OPC RDIM LIR	OALU OERALX EXOR LALU update flags

3. OR A, RX

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	A <- A or RX
OPC RDIM LIR	OALU OERALX EOR LALU update flags

4. ADD A, RX

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	$A \leftarrow A + RX$
OPC RDIM LIR	OALU OERALX EADD LALU update flags

5. SUB A, RX

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	$A \leftarrow A - RX$
OPC RDIM LIR	OALU OERALX ESUB LALU update flags

6. CMA

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	$A \leftarrow \sim A$
OPC RDIM LIR	CMA

7. INC A

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	A <- A + 1
OPC RDIM LIR	INCA

8. DEC A

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	A <- A-1
OPC RDIM LIR	DECA

9. RRA

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	A[7] <- A[0] A[n] <- A[n+1] ; n=0,1,2,3,4,5,6
OPC RDIM LIR	RR

10. RLA

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	A[0] <- A[7] . A[n+1] <- A[n] ; n=0,1,2,3,4,5,6
OPC RDIM LIR	RL

11. SETCLRf 4-bit data

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	IRX[3:0] -> FLAG REGISTER
OPC RDIM LIR	OIRSYS SETCLRf

12. CLRTMRF

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	TMF0 => 0
OPC RDIM LIR	CLRTMRF

13. EDINTER 3-bit data

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	IRX[2:0] -> INTCON REGISTER
OPC RDIM LIR	OIRSYS LINTCON

BRANCHING INSTRUCTIONS

1. JUMP add18

TF1	TX1	TX2
PC -> IM IMdata -> IR PC+1 -> PC	PC -> IM IMdata -> PCR[15:0] IRX[10:9] -> PCR[17:16]	PCR -> PC
OPC RDIM LIR	OPC RDIM OIRPC LPCR	LPC OPCR

2. JZ add18

TF1	TX1	TX2
PC -> IM IMdata -> IR PC+1 -> PC	PC -> IM IMdata -> PCR[15:0] IRX[10:9] -> PCR[17:16]	PCR -> PC
OPC RDIM LIR	OPC RDIM OIRPC LPCR	OPCR check zero flag if Z=1 LPC is set else if Z=0 INCPC is set

3. JC add18

TF1	TX1	TX2
PC -> IM IMdata -> IR PC+1 -> PC	PC -> IM IMdata -> PCR[15:0] IRX[10:9] -> PCR[17:16]	PCR -> PC
OPC RDIM LIR	OPC RDIM OIRPC LPCR	OPCR check carry flag if C=1 LPC is set else if C=0 INCPC is set

4. JB add18

TF1	TX1	TX2
PC -> IM IMdata -> IR PC+1 -> PC	PC -> IM IMdata -> PCR[15:0] IRX[10:9] -> PCR[17:16]	PCR -> PC
OPC RDIM LIR	OPC RDIM OIRPC LPCR	OPCR check borrow flag if BF=1 LPC is set else if BF=0 INCPC is set

5. JP add18

TF1	TX1	TX2
PC -> IM IMdata -> IR PC+1 -> PC	PC -> IM IMdata -> PCR[15:0] IRX[10:9] -> PCR[17:16]	PCR -> PC
OPC RDIM LIR	OPC RDIM OIRPC LPCR	OPCR check parity flag if P=1 LPC is set else if P=0 INCPC is set

MACHINE CONTROL AND IO INSTRUCTIONS

1. HLT

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	Stops operations. An interrupt is necessary to exit from the halt state.
OPC RDIM LIR	-

2. RESET

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	RESET PC, A, RX
OPC RDIM LIR	RESET

3. SAV PC

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	PC+2 -> PCS
OPC RDIM LIR	LPCS

4. RES PC

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	PCS -> PC
OPC RDIM LIR	OPCS

5. RETI

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	STACKPC -> PC
OPC RDIM LIR	OSTACKPC

6. WAIT TXF

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	Waits until TXF flag gets set
OPC RDIM LIR	-

7. WAIT RXF

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	Waits until RXF flag gets set
OPC RDIM LIR	-

8. IN P0

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	P0 -> A
OPC RDIM LIR	LIN

9. OUT P1

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	A -> P1
OPC RDIM LIR	OOUT

10. SIN RBUFF

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	RBUFF -> A
OPC RDIM LIR	LSIN

11. SOUT TBUFF

TF1	TX1
PC -> IM IMdata -> IR PC+1 -> PC	A -> TBUFF
OPC RDIM LIR	LSIN

