DESIGN METHODOLOGY FOR MODELING A MICROCONTROLLER

A Thesis Presented to

The Faculty of the

Fritz J. and Dolores H. Russ
College of Engineering and Technology

Ohio University

In Partial Fulfillment

of the Requirement for the Degree

Master of Science

by

Phillip D. Southard

June, 2000

THIS THESIS ENTITLED

"DESIGN METHODOLOGY FOR MODELING A MICROCONTROLLER"

by Phillip D. Southard

has been approved

for the School of Electrical Engineering and Computer Science

and the Russ College of Engineering and Technology

Janusz Starzyk, Professor
School of Electrical Engineering and Computer Science

Warren K. Wray, Dean
Fritz J. and Dolores H. Russ
Russ College of Engineering and Technology

ACKNOWLEDGMENTS

I would like to sincerely thank my father Dave, my mother Kris, my brothers Erik and Andrew, my sisters Angela and Malissa, and my friends for their support during my research. I would also like to thank Dr. Starzyk for his guidance and support. I also thank the Sarnoff Corporation and the U.S. government for sponsoring my research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SYMBOLS AND ABBREVIATIONS

ALE	Address Latch Enable
ASIC	Application-Specific Integrated Circuit
CAD	Computer Aided Drafting
CMOS	Complimentary Metal Oxide Semiconductor
DARPA	Defense Advanced Research Projects Agency
FPGA	Field Programmable Gate Array
IEEE	Institute of Electrical and Electronics Engineers
IC	Integrated Circuit
IS	Instruction Set
LSB	Least Significant Bit
MSB	Most Significant Bit
PC	Program Counter
PSENn	Program Store Enable active low
RAM	Random Access Memory
ROM	Read Only Memory
RASSP	Rapid-Prototyping Application Specific Signal Processors
RTL	Register Transfer Level
RTN	Register Transfer Notation
SFR	Special Function Register
UUT	Unit Under Test

VHDL VHSIC hardware description language

VHSICVery High Speed Integrated Circuit

1.0 INTRODUCTION

In the ever-changing world of technology, new ideas are born and legacy technologies are left aside to be chronicled in the history books. Although new technologies become available, sometimes it is necessary to maintain older technology when servicing electrical systems. The U.S. military and other military units around the world fight an ongoing battle against technology life cycles. Program managers of radar systems or fire control systems may or may not redesign a unit due to the advent of new technology. The lack of money usually prevents the redesign of systems. Since some systems cannot be replaced using new technology, technicians are challenged to maintain electrical systems with parts that are not procurable by commercial buyers and or government purchasers.

The work described within this thesis presents an engineering solution that stems from the problem of parts obsolescence as it pertains to military systems. Sarnoff Corporation of Princeton NJ, has developed BiCMOS gate arrays that makes it possible to emulate older technology with newer and faster technology, providing a pin for pin replacement part. For this thesis, the Intel 8031 microcontroller was chosen as a device that would be modeled with VHDL so that a military grade device would be available if need be in the future to help maintain electrical systems used by the U.S. military.

The design methodologies used to develop a Behavioral and Instruction Set (IS) model for the Intel 8031 and the results of testing these models will be presented. Very High Speed Integrated Circuits Hardware Description Language (VHDL) was used to

create the Behavioral model and the IS model for the Intel 8031 microcontroller. VHDL is a text based language that can capture the functionality of a digital device as simple as a NAND gate to an advanced digital communications system. The nature of this research project does not allow for disclosure of intellectual property, hence the actual VHDL text files will not be provided. Certain aspects of the VHDL code developed will be explained as needed. To verify the Behavioral and IS model of the Intel 8031, it was decided that a Hardware Modeler from Synopsys would be used.

This thesis was sponsored in part by the Sarnoff Corporation and the U.S. government. It is the intent of this research to develop a model for the Intel 8031 microcontroller. This will allow for an understanding of how to best replace obsolete parts with new components, especially complex parts. The replacement of obsolete or singled sourced parts by emulation of the existing chip or using remaining die at chip supply houses usually yields a cost effective solution for the government and the tax payer. Sometimes board or system redesigns are necessary to eliminate a high percentage of system or board level obsolete components. This is often very expensive and time consuming.

This thesis has been organized as follows. In chapter 2, a brief description of VHDL and the associated levels of modeling are given. In chapter 3, an architecture overview of the Intel 8031 is provided. Next, in chapter 4, an approach to modeling microcontrollers and similar devices is described. The methods of testing and verifying the models developed for this thesis are given in chapter 5. Chapters 6 and 7 discuss the

simulation results of the Behavior and the IS model respectively. Finally, conclusions and recommendations for continuance of this research are discussed in chapter 8.

2.0 VHDL Modeling

The state-of-the-art of digital circuit design now provides for an efficient, CAD oriented methodology for implementing digital designs, by using VHDL. In the not so distant past, design engineers were forced to develop logic functions using Karnaugh maps putting together their designs logic gate by logic gate.

During the 60's and the 70's system level and microcomputer design entailed building systems out of many individual logic gates manufactured on Integrated Circuits (IC). This design technique was very costly and time consuming. As the technology moved forward from medium-scale integration (MSI) to large-scale integration (LSI), to very large-scale integration (VLSI), the need for new design tools became apparent. With the ability to incorporate many functions on one IC, engineers needed a way to quickly design a function and or circuit and test the design. The standard known as VHDL, was first created in 1987, [2]. VHDL allows for hardware description in a text-based language. VHDL is similar to Ada, a government standardized, portable, and object oriented software language. VHDL allows a design to model a digital system at many levels of abstraction. A description can be as simple as a 2-input logic circuit or an entire digital system.

As devised by the DARPA/TRI-Service Rapid-prototyping of Application Specific Signal Processors (RASSP) and some aerospace contractors, there exists taxonomy for VHDL models [6]. There are five different levels of modeling:

Performance modeling, Behavioral modeling, Instruction Set modeling, Register Transfer Level (RTL) modeling, and Gate-Level modeling.

Performance modeling is used to specify the general make up of a complete system. Items that might be specified in performance modeling could include frames per second for a video processor or number of instructions per second for a processor.

Behavioral modeling adds one level of detail. A Behavioral model will include a break down of the system into subcomponents and functional blocks. There is little detail to how the functions will be physically implemented.

The next level of detail is included in the Instruction Set model. This level of detail provides a model that can be used to simulate the instruction set of the microcontroller, microprocessor, or DSP chip in question. This thesis focuses on the instruction set model level.

RTL modeling is a level of detail that specifies exactly how data will transfer from register to register. RTL is the most useful level of modeling to a designer who is going to actually implement his or her design in an Application-Specific Integrated Circuit (ASIC) or a Field Programmable Gate Array (FGPA). There are many synthesis tools that will "synthesize" or translate the RTL model into a netlist of primitive technology cells for placement and routing. There has been a great deal of focus in the area of synthesis since this is where the designer can save many hours of work. Instead of drawing thousands upon thousands of gates in a schematic, the VHDL savvy designer can describe a design in a text-based language that can be automatically transferred to a schematic in a matter of seconds.

The final level of detail is the Gate-Level model. After synthesis is complete, the result is a netlist of primitive gates like NAND gates and NOR gates. This netlist can be simulated with timing information provided by the vendor of the target silicon. This simulation is called post synthesis pre place and route simulation.

As mentioned above, this thesis focuses on an IS model for the Intel 8031 microcontroller. Prior to developing a detailed IS model, a designer needs to understand the two levels above an IS model, the Performance model and the Behavioral model. In this case, the circuit of discussion has been thoroughly described in the datasheets available from Intel. This documentation takes the place of the Performance model. That leaves the Behavioral model to develop first.

The Behavioral model developed is an abstract model of the Intel 8031 that demonstrates a basic understanding of how instructions are fetched and executed. The model was written to give a starting point of understanding to the overall design. Hence the Behavioral model will not have any physical implications pertaining to the original microcontroller.

Once the Behavioral model has been written one can then focus on developing an IS model. The IS model developed for this thesis allows for exercising a subset of Intel MCS-51 instructions. Since the Intel 8031 is a feature-reduced version of the Intel 8051, the Intel 8031 uses the MCS-51 instructions.

3.0 Intel 8031 Architecture

It is not the intent of this chapter to provide a thorough analysis of the Intel 8031 architecture. Fig. 1 provides a general overview of the Intel 8031. A sufficient portion of the architecture will be described to give the reader an understanding of the Intel 8031 as it relates to the research presented.

Intel 8031 core features [5]:

- 8 bit CPU optimized for Control Applications
- Extensive Boolean processing (single-bit logic) capabilities
- 64K Program Memory address space
- 64K Date Memory address space
- 128 bytes of on-chip Data RAM
- 32 bi-directional and individually addressable I/O lines
- Two 16-bit timer/counters
- Full duplex UART
- 6-source/5-vector interrupt structure with two priority levels
- On-chip clock oscillator

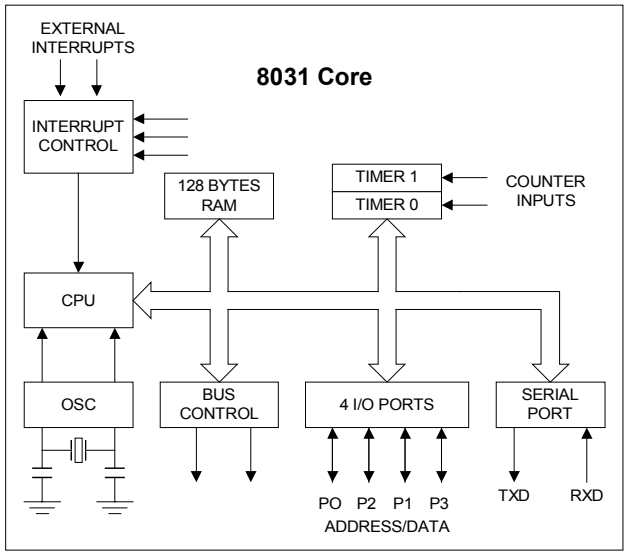


Fig. 1. Block Diagram, Intel 8031 Core

The Intel 8031 is a rather simple machine with quite a bit of flexibility. The 4 I/O ports are perfect for communicating with many external applications. The rich interrupt structure aides well in a control environment. Fig. 2 is an architecture block diagram of the Intel 8031. Even though the previous diagram shows some architecture detail, more detail is needed to get a better understanding of the true architecture. Fig. 2 shows a higher level of register detail for the Intel 8031. More specifically, register organization is more pronounced and the bus structure is well defined.

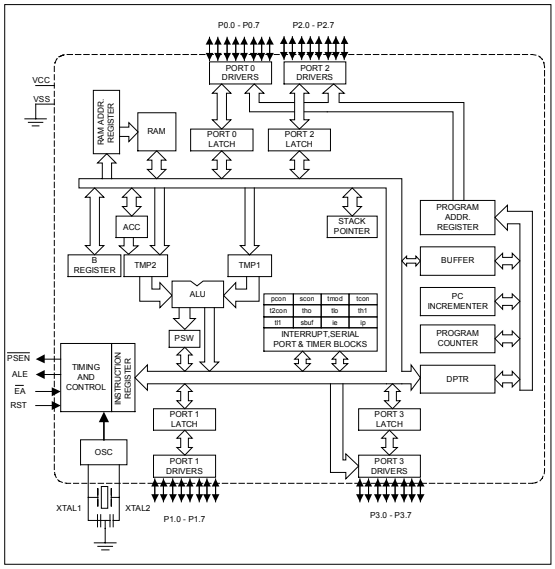


Fig. 2. Intel 8031 Detailed Architecture

Fig. 2 shows that the Intel 8031 design is based on the Harvard architecture; the data space and the program space are separated. Fig. 2 also shows a clear separation between the Data Path and the Control Unit. This will make modeling the Intel 8031 much easier. One feature that is noteworthy is the memory-mapped ports. All of the external ports are memory mapped, which simplifies programming control projects. These external data port addresses reside in the memory where registers are given the term

Special Function Register (SFR). Careful study of the architecture gives rise to several SFRs.

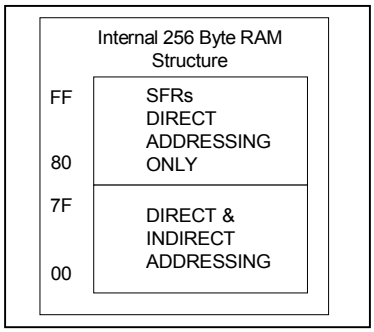
Table 1. Special Function Registers

SYMBOL	NAME	ADDRESS	RESET VALUE
ACC*	Accumulator	0E0h	00000000
B*	B Register	0F0h	00000000
PSW*	Program Status Word	0D0h	00000000
SP	Stack Pointer	81h	00000111
DPTR	Data Pointer 2 Bytes		
DPL	Low Byte	82h	00000000
DPH	High Byte	83h	00000000
P0*	Port 0	80h	11111111
P1*	Port 1	90h	11111111
P2*	Port 2	0A0h	11111111
P3*	Port 3	0B0h	11111111
IP*	Interrupt Priority Control	0B8h	xxx00000
IE*	Interrupt Enable Control	0A8h	0xx00000
TMOD	Timer/Counter Mode Control	089h	00000000
TCON*	Timer/Counter Control	088h	00000000
TH0	Timer/Counter 0 High Byte	08Ch	00000000
TL0	Timer/Counter 0 Low Byte	08Ah	00000000
TH1	Timer/Counter 1 High Byte	08Dh	00000000
TL1	Timer/Counter 1 Low Byte	08Bh	00000000
SCON*	Serial Control	098h	00000000
SBUF	Serial Data Buffer	099h	Indeterminate
PCON	Power Control	087h	0xxx0000
* = Bit Addressable			

Table 1 contains register descriptions for the SFRs and their respective addresses. Although the special function registers seem to exist as individual registers within the architecture, they are part of the Intel 8031 internal Random Access Memory (RAM) structure. The next two figures to be introduced will help to describe the internal 256-byte RAM configuration. The internal RAM of the Intel 8031 has four distinct spaces. The upper 128 bytes of RAM contain the SFRs. The lower half of the RAM is further

divided into three segments, scratch pad, bit addressing segment, and the register bank area. Fig.3 shows the initial division of the RAM block. It also gives some details as to which addressing modes may be used with the internal RAM. The addressing modes will be discussed in the next section.

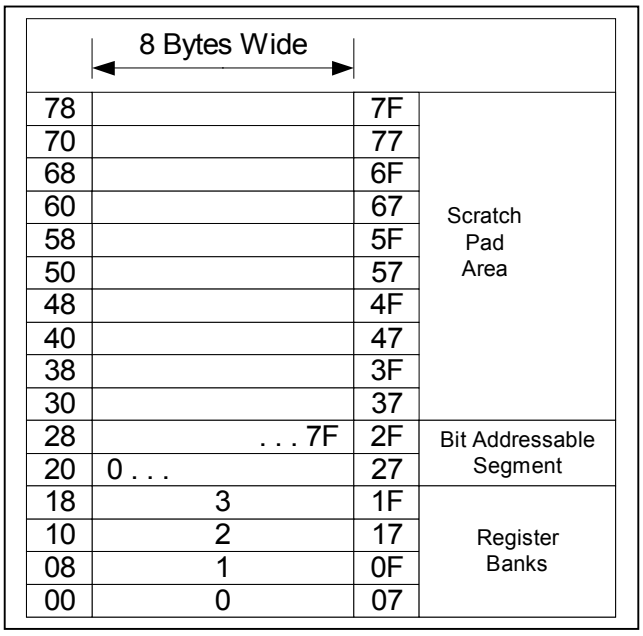
Fig. 3. 256 Byte Ram Block



Though Fig. 3 depicts the SFRs as being within the structure of the RAM, it is more likely and is hypothesized that most of the SFRs are physically located outside of the RAM space. Not having these registers in the RAM space would allow for easier placement and access within the datapath. For example, the SFRs for ports P0, P1, P2, and P3 are probably close to the peripheral of the chip, yet they are accessed as though they are physically in the RAM. This could not be verified since Intel keeps these secrets to themselves. Fig. 4 on the next page provides more detail for the lower portion of the RAM block. Within this lower portion we see a scratch pad area for general use. A bit addressable segment has been included for control applications. The lowest portion of the RAM block contains a bank of registers. The register banks can be accessed by direct or register addressing. The Program Status Word (PSW), which is a

SFR located at D0h, contains two bits that shift a pointer to specify with bank to use. Otherwise a programmer may directly request or write data to this space.

Fig. 4. Lower Half of RAM Block



3.1 Intel 8031 Addressing Modes and Instruction Set Notes

When one writes a program for a microcontroller, close attention is paid to what kind of addressing modes are available to the programmer. The Intel 8031 program space is interfaced by four addressing modes: immediate, register, direct, and indirect.

In direct addressing the operand is specified by an 8-bit address field in the instruction. Only internal data RAM and SFRs can be directly addressed. With indirect addressing both internal and external RAM can be indirectly addressed. The address register for an 8-bit address can be R0 or R1 of the selected register bank, or the stack pointer. The address register for 16-bit addresses can only be the 16-bit "data pointer" register, DPTR. When using register addressing, the register banks, contain registers R0 through R7. These registers can be accessed by certain instructions that include a 3-bit register specification within the opcode. One of four banks is selected at execution time by the two bank select bits in the PSW mentioned earlier [5].

The instruction set available to the programmer of the Intel 8031 contains 255 instructions. Dividing these instructions up into main groups, there are 4 major classifications. These groups are: arithmetic and logical operations, data transfers, program branching, and Boolean variable manipulations. The details of each instruction will not be discussed. Appendix A has been included to provide some detail about each instruction. The appendix includes the hex code, mnemonic, number of cycles, and byte count for each instruction.

For this thesis the main thrust of the modeling effort was on the register addressing mode instructions, primarily with register bank R0 operations. For the Behavioral model only certain basic instructions where modeled. Basically over half of the instruction set was modeled for the IS model. Certain instructions will be discussed in detail as needed.

3.2 Machine Cycle Sequence

More than just study of addressing modes, instructions and general block diagrams is needed when modeling a microcontroller. If a model is to emulate its predecessor, the timing of data transactions must be accurate. Fig. 5 depicts the waveforms for critical signals during a memory fetch and their critical relative timing.

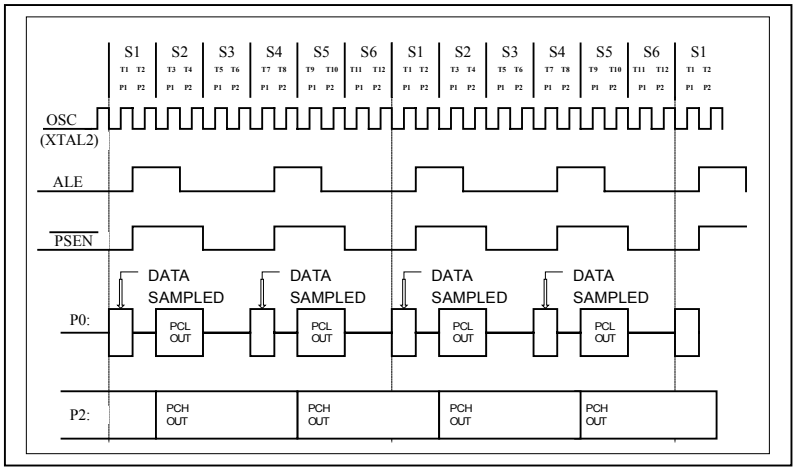


Fig. 5. External Program Memory Fetches

The "clock" i.e. the XTAL input is the bases for the figure. Address Latch Enable (ALE) and Program Store Enable active low (PSENn) are also included. Since the Intel 8031 does not have an internal program space, external memory is the only program source. The most critical bit of information is when Port 0 and Port 2 are read and updated. In the Intel literature, the engineers have divided a machine cycle into six major states, S1-S6. Furthermore these states are divided into phases P1 and P2. The T states shown are added as a result of this work. It will be discussed in chapter 7 why these T states were added to the diagram. The port interaction information detailed here

is essential for timing accurate models. Fig. 6 not only contains the state ALE, and PSEN waveforms as in Fig. 5, but also includes four examples of how different instructions have different state timing.

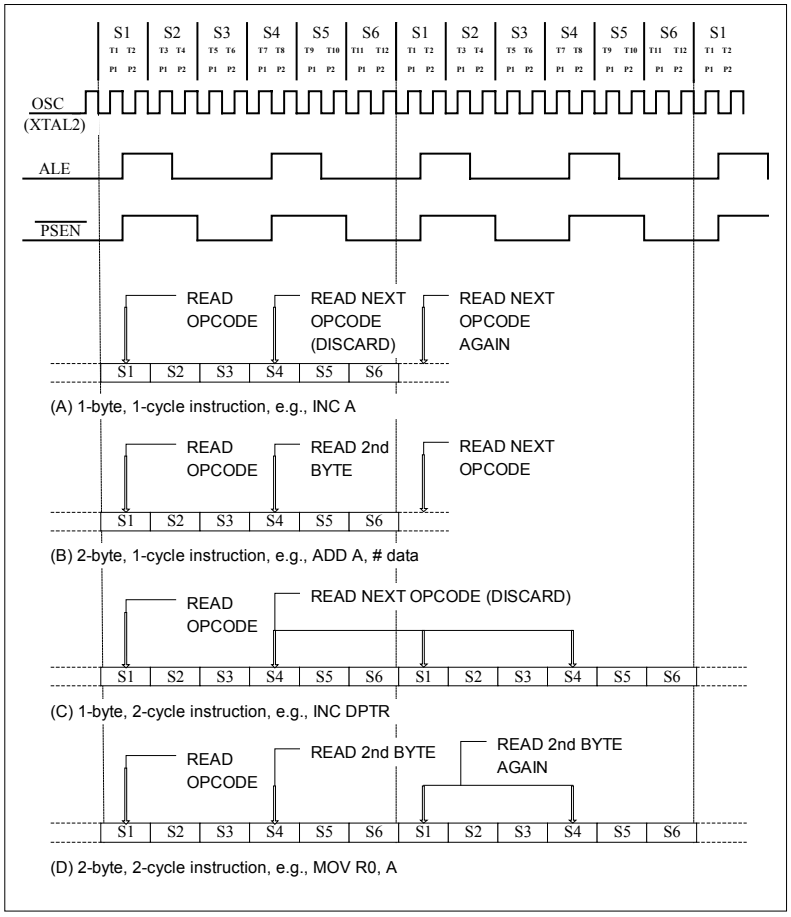


Fig. 6. Instruction Examples with State Timing

In Fig.6, example A, register addressing mode instruction INC A states' timing is illustrated. For INC A, it is obvious that this instruction is a single-machine cycle instruction or a twelve T states instruction. For example B, an immediate addressing mode instruction, ADD A, # data is a single machine cycle instruction. This instruction

is a two-byte instruction. INC DPTR, a register address mode instruction, is used for example C. There is only one critical port interaction, but two machine cycles or 24 T states or 24 cycles of the system clock, are needed to execute the instruction. Finally for example D, MOV R0, A, is shown to be a 2-byte, 2-cycle instruction.

As one can see from the previous architecture figures, the Intel 8031 is a microcontroller without a Read Only Memory (ROM). This would explain the need to fetch instructions from off-chip memory. In order to use the Intel 8031, a minimum system must be created to enable the controller to have basic functionality. Fig. 7 illustrates that minimum system.

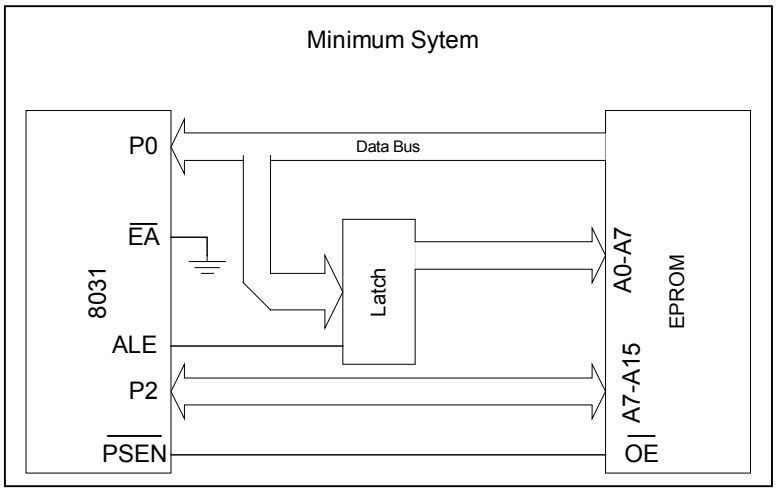


Fig. 7. Intel 8031 Minimum System

3.3 Modeled Architecture

For the purpose of this thesis, certain parts of the Intel 8031 architecture were modeled and others were not. Since the end result of this work is an IS model that tests the Register addressing mode using R0, the architecture to support this modeling will be discussed. The Intel 8031 instruction set contains 255 instructions. The IS VHDL model discussed in chapter 7 can support 133 instructions. 128 of the 133 instructions are register addressing mode instructions. The five remaining instructions were modeled to allow for simulation. The following is a summary of the instructions that were modeled.

Table 2. Modeled Register 0 Instructions

ADD A,R0
ADDC A,R0
ORL A,R0
ANL A,R0
XRL A,R0
INC R0
DEC R0
MOV R0,#DATA
MOV DATA ADDR, R0
SUBB A,R0
MOV R0,DATA ADDR
CJNE R0, #DATA, CODE ADDR
XCH A,R0
DJNZ R0, CODE ADDR
MOV A,R0
MOV R0,A

As mentioned previously, five additional instructions where modeled. These instructions are critical for basic functionality. These five instructions are described in the following table.

Table 3. Essential Modeled Instructions

LJMP CODE ADDR
MOV A, # DATA
MOV DATA ADDR, A
CLR C
SETB C

The first three instructions in table 3 allow for basic data exchange between memory and the Intel 8031 core. The last two instructions aided in determining if certain arithmetic instructions were performing correctly.

The architecture modeled in the IS model to support these instructions include the following registers and functional blocks as displayed in Fig. 2.

Table 4. IS Model Instructions

Port 0, latch and driver
Port 1, latch and driver
Port 2, latch and driver
RAM
RAM ADDR REGISTER
ALU
TMP1
TMP2
PSW
ACC
TIMING AND CONTROL
INSTRUCTION REGISTER
PROGRAM ADDR REGISTER
PC INCREMENTER
PROGRAM COUNTER

The functions and registers listed account for 35% of the complete architecture. Items like the counter/timers and the UART were not modeled. VHDL models for a UART were found to exist. Models of the counter/timer were viewed as not necessary to display proof of concept of modeling the core of the Intel 8031 microcontroller. The IS VHDL model was written such that more instructions could easily be added. The remaining functional blocks would also integrate easily.

4.0 Brainstorming the Design, The Creative Process

When trying to design a microcontroller, it is difficult to know where to start. Literature research turned up a couple of books that would lead to a design process that would aid in defining the goals for this thesis and provide a systematic approach for modeling the Intel 8031. In the book titled Digital Systems Design Using VHDL, Charles H. Roth, Jr. suggested a design process. The book was written to educate readers on the use VHDL and how to write VHDL models for systems larger than an MSI IC. The book had a chapter about designing a microcontroller. The author suggested that, together, he and the student would study the MC68HC05 and write behavioral code to model a portion of this chip. The author broke the design into subsystems. The main focus though was on the basic functions of the microcontroller. He recommends to the student to familiarize himself with the instruction set of the target microcontroller or thoroughly plan out a new design's instruction set.

A challenge for this research was that there was no book to aid in breaking down the instruction set of the Intel 8031. The Intel 8031, as mentioned before, is an 8-bit microcontroller. This implies that the instruction set could have a total of 256 instructions and indeed the Intel 8031 instruction set does have 256 instructions including one NOP, i.e. no operation instruction. The Intel 8031 instruction set seemed to have a natural pattern to how the instruction set was structured. A most crucial aspect of this work was to determine how to decompose the instruction set and use the discovered pattern in the most efficient manner.

When writing a Behavioral model, one does not want to create a 256 entry case structure. This is a little cumbersome and would not translate well into a RTL VHDL model. A RTL model would be used to synthesize the design to a specific technology. Essentially the instruction set code had to be "cracked" as how to break the instruction set up in a logical manner which would facilitate coding the instruction set into the Behavioral and IS VHDL model. Please refer to appendix A for an "as is" version of the Intel 8031 instruction set. This view will give the reader an idea of the decomposition challenge. It was finally decided that the instructions should be broken in the middle. The four bit grouping consisting of the most significant bit (MSB) down to the (MSB – 3) bit of the 8 bit instruction, commonly referred to as the upper nibble, would identify what type of instruction was being decoded. The four bit grouping consisting of the least significant bit (LSB) up to the (LSB + 3) bit of the 8 bit instruction, commonly referred to as the lower nibble, would indicate what type of addressing would be needed to complete that instruction.

For instance, if one looks at register addressing mode addition instructions, hex codes 28h through 2Fh, one can see that the upper nibble does not change (see table 5). It was helpful to take the instruction hex codes and write then in binary format. If this is done, one will see that for register addition the last three bits of the instruction indicate which register should be used to complete the instruction. It was this pattern that led to completing the decomposition of the instructions.

Table 5. Register Addressing Mode Addition Instruction Set Subset

Hex Code	Number of Bytes	Mnemonic	Operands
28	1	ADD	A,R0
29	1	ADD	A,R1
2A	1	ADD	A,R2
2B	1	ADD	A,R3
2C	1	ADD	A,R4
2D	1	ADD	A,R5
2E	1	ADD	A,R6
2F	1	ADD	A,R7

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

```
LJMP CODE ADDR
MOV A, #DATA
MOV DIRECT, A
ADD A, #DATA
ADD A, REGISTER
```

The LJMP instruction was selected due to its importance in beginning any program for the Intel 8031. Mentioned in chapter 3, the Intel 8031 has an interrupt structure. The addresses for the interrupt vectors are located at the beginning of the program space. For this reason, a LJMP is needed to perform an unconditional branch to a safe address in memory to start a program. The two move instructions are critical in setting up control registers like the PSW. The ADD instructions were selected to test out ALU functionality.

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

Table 6. Register Transfer Table

Addressing Mode	Instruction	1st Cycle	2nd Cycle	3 rd Cycle	4th Cycle
Immediate	Add A, #data	{fetch}	{addr1} Tmp1 ←mem(PC) PC ←PC + 1	(A ←A + Tmp1)	
Direct	Add A, Direct	{fetch}	{addr1} Rar ←mem(PC) PC ←PC + 1	{data} Tmp1 ←Ram(Rar)	(A ←A + Tmp1)
Direct	MOV Direct, A	{fetch}	{addr1} Rar ←mem(PC) PC ←PC + 1	{RamWrite} Ram(Rar) ←A	
Register	Add A, Rn	{fetch}	{addr1} Rar ←mem(PC) PC ←PC + 1	{data} Tmp1 ←Ram(Rar)	(A ←A + Tmp1)
	LJMP	{fetch}	{addr1} MarH ←mem(PC) PC ←PC + 1	{addr2} PCL ←mem(PC) PCH ←MarH	

Table 6 maps out the necessary register transfers to complete each instruction. In column one the different addressing modes are noted for each instruction. Column two identifies the instruction being described. The next four columns represent needed transactions. To explain this table, the ADD A, Direct instruction will be reviewed.

The first register transfer necessary for any of the instructions is a fetch from the memory. The next transfer for the ADD A, Direct instruction is a read from memory of the RAM address that contains the data that will be added to the accumulator. During this “address 1” state the program counter also gets attention, it is updated. The Tmp1 register assists by holding the data retrieved from the RAM in cycle three. To complete the instruction, the value of the accumulator is added with the value of Tmp1 and stored

in the accumulator. It was decided that the actual addition described with the register transfer notation in column four or cycle four could be executed during the next fetch cycle. This explains why there is no ALU state enclosed in {} in cycle 4 for the ADD A, Direct instruction. The transfer explanation would follow a similar pattern for the other four instructions.

Now that states have been identified for each instruction, a state diagram can be constructed covering all five instructions. The state diagram in Fig. 8 was used to construct the Behavioral VHDL model. The reset state and states cycle 8, cycle 9, and cycle 10 are not related to the states in the transfer table in table 6. These states were added to allow for configuration of SFRs.

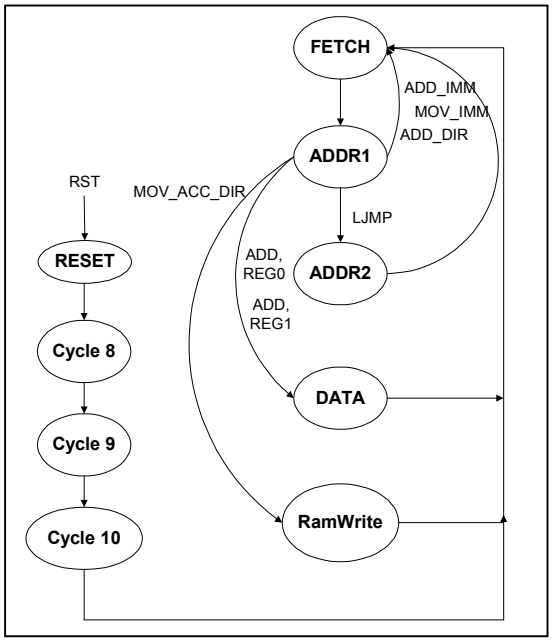


Fig. 8. Simple State Diagram

Instruction ADD A, Direct was determined to need three states to execute, fetch, addr1, and data. Reviewing the state diagram one can see that the ADD A, Direct instruction is similar to the other four instructions.

Following state diagram creation, a Behavioral VHDL model was developed based on the following completed tasks: instruction review, addressing mode review, register transfer diagramming, and state diagramming. The Behavioral VHDL model is primarily comprised of a control process with a case statement that emulates the state diagram. An arithmetic function, used to emulate the ALU of the Intel 8031, is called during the fetch state of the control process. The program memory was not modeled based on true architecture. The Intel 8031 is a ROMless microcontroller that normally communicates through ports 0 and 2 to retrieve instructions. For the Behavioral model the program memory is simply designed into the same abstract structure as the process that models the states of the controller. This simplifies the model allowing for attention to be paid to the instruction decoding and execution.

The Behavioral model simulation results pertaining to the five instructions discussed will be charted and explained in chapter 6. Before the results are shown, the test method used will be discussed. The next chapter will provide information as to the testing technique used to simulate the Behavioral and IS VHDL model. Before moving onto chapter 5, a broad brush explanation of the IS model will be provided.

The organization of the IS model is similar to that of the Behavioral model. Both models have a control process that controls the state of the model. Both models also have the same ALU_OP procedure structure to perform arithmetic operations. With the

Behavioral model, a single process handles instruction decoding and state control. In the IS model, these functions are separated. The main process, a state machine like process, in the IS model is a lengthy simple looping state machine that steps through 12 cycles for every machine cycle required for instruction execution. The state machine cycle process monitors the status of the Ports, initiates control signals at the proper time according to the decoded instruction, and updates the Program Counter (PC). See appendix D for a listing of example code. The sample code of appendix D also shows the signal declarations and aliases used to support the IS model.

Supporting the main state machine process is a process that decodes the instructions as they are retrieved from Port 0 (see appendix E). When the state machine enters state M1T2 or Machine cycle 1 T cycle 2, decoding of the incoming instruction is initiated (see line 443 appendix D). The Instruction Register (IR) used during decoding is "split" in to two pieces. OP for operation alludes to the upper nibble of the 8-bit register IR and MODE alludes to the lower nibble of IR. A two level case statement structure is used for the decoding process. The first case statement looks at the 4-bit nibble MODE to determine what addressing mode the incoming instruction is to use (see line 94 appendix E). Once the addressing mode is determined, additional case statement structures within the case statement for each addressing mode type analyzes the 4-bit nibble OP (see line 205 appendix E). Once the type instruction is determined, control signal bits are set or cleared for the specific instruction.

As mentioned, a procedure called ALU_OP is used to execute arithmetic instructions (see appendix F). The VHDL code in appendix F just contains the code to complete the Register Addressing mode ADD instruction.

5.0 Testing and Verification

Testing is the most feared task of all engineers. Most IC design engineers love to design and hate to bother with thoroughly testing their new creations. Personal feelings aside, testing should not be taken lightly. Testing is a critical, time consuming, and an expensive necessity of product development. Testing is just as important as the design. Without an accurate and validated design, an engineers work is not complete and has a little value to his or her company.

The IC technologies of today have sub-micron feature sizes. Designs can consist of millions of gates. VHDL allows an engineer to easily design tens of thousands of gates per day. There can be many instructions to consider and many items in a data path to test and verify. For these reasons, a sound test method must be used. This is where a VHDL test bench, a VHDL model using available packages included in the VHDL standard library STD, will help to minimize the testing challenge.

VHDL inherently contains constructs to aid in the creation of a test bench. A test bench, developed in VHDL, is VHDL code written to evaluate a digital design. A test bench is a VHDL model itself describing an instantiation of an architecture of the design to be tested and the signals and functions or procedures to be used. A test bench can contain processes such as a clock generation process or procedure to read test vectors from a text file that are to be clocked against the design or the Unit Under Test (UUT). The preferred test bench will be a VHDL behavioral model that can be simply executed with a VHDL simulator. The test bench should initialize all necessary signals, apply test

vectors, collect the results from the simulation for storage or review, and determine which test vectors did not pass, i.e. self-checking.

During development of a VHDL model, having a test bench already prepared that can be used by the designer will help to reach the design goal quickly. In some cases, an engineer other than the design engineer may write a test bench. Having a test bench that can be executed quickly and repeatedly to test a design is a desired method for VHDL model development. A test bench also creates in itself a form of documentation. If the creator of the test bench comments the VHDL code well, other engineers will be able to understand how the UUT is being tested. In a complete VHDL design flow using synthesis, test benches are a must. The test bench is written once and covers all of the different levels of abstraction. The same test bench that is used for the behavioral model is used to test the RTL and also the Post synthesis and Post layout netlists. The UUT configuration statement is simply changed to point to the desired model.

For this research a test bench was written for the IS model and not for the behavioral model. During the development of the VHDL models, the value a VHDL test benches was not fully appreciated until work began on the IS model. The Behavioral model is so simple that the VHDL simulator used allowed for simple testing. In addition, the Behavioral model was so abstract, that the interface necessary for a test bench did not exist. During IS model creation, a more detailed interface was created to allow for testing with a VHDL test bench.

Fig. 9 details the virtual testing environment that was created with a test bench for the IS model. This virtual environment allows for repetitive and convenient test of the UUT.

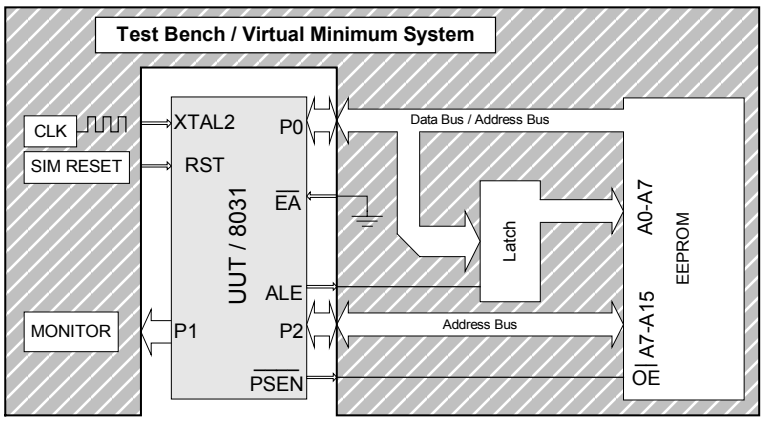


Fig. 9. Test Bench Environment

The area with a white hatched pattern represents the virtual test environment described with the VHDL test bench. The area surrounding the solid light gray block with various arrows represents the interface between the test bench and the UUT. The UUT in this instance is an IS VHDL model of the Intel 8031. Instead of having the test instructions internal to the Intel 8031 model as with the Behavioral model, this configuration allows for different sets of instructions to be loaded into the simulated EEPROM. The test bench allows for easy control of stimulus like the reset and clock signals. The monitor can display simulation results as a waveform. The results can also be written to disk for record.

The advances of VHDL and hardware development have given creative engineers tools to further testing and verification methods. Synopsis makes a product called a

Hardware Modeler [3]. This device can be used to verify a VHDL design against real hardware. An actual IC is soldered to a circuit board in a module. This module can then be fastened to the front of an interface box. The interface box allows for communication between the actual IC and the hosting server. This server allows a VHDL simulator tool to interact with the actual IC. Once all the hardware is installed, configured and powered, an engineer can simulate his VHDL system design using the access to an IC that he or she may have not yet modeled, i.e. Pentium processor.

For this research, the modeling unit was used in a slightly different manner. The modeler was used to verify a model of a specific IC, the Intel 8031 microcontroller. A set of hex codes was used in conjunction with a test bench to capture the output of the physical Intel 8031. The "golden code" was used later in conjunction with a test bench to check the results of the IS model.

The VHDL test bench referred to in the previous paragraphs contains the following structure. The VHDL test bench contains 6 processes, 1 procedure, and the standard configuration statements needed to refer to the correct UUT. The one and only procedure was designed to read a text file that contains the desired test code in a hexadecimal format. The 6 processes are as follows. The CLK process is a rather short process that simply generates a reoccurring square wave signal with 50% duty cycle. Two processes were needed to help model the virtual test environment. One of those processes modeled a latch and the other process provides a model of a ROM. The latch model was written to provide a functional representation of a 74LS373 Octal D-type transparent latch. The ROM model was written for the NMC26C64 64K bit CMOS

EPROM. The Reset process created a virtual reset of the UUT. Together the CLK, Reset, Latch, and EPROM process models created a virtual minimum system that allowed for exercising different sets of hex code against the IS model.

The last two processes, the "golden" hardware modeler results process and the check results process completed the self-checking test bench. The "golden" Hardware Modeler results process read a test results file to be used for comparison with the IS model results. Test results were generated by exercising test code, later used for IS model testing, against the actual Intel 8031. Any port changes were recorded. At this point the results were considered "golden" i.e. actual and correct part results. The check results process uses a simple *if then* statement to compare the actual IS model results against the "golden" results that were read from the test results file. If a discrepancy was found, a report message was generated for display in the simulator using VHDL's *assert* statement [4]. The two resulting sets of signals were displayed in the simulator. These results will be graphically displayed in the next chapter.

6.0 Behavioral Model and Simulation Results

The design process discussed so far has laid the groundwork necessary for creating a Behavioral model for the Intel 8031. A VHDL model was written based on the transfer table and the state machine graph discussed in chapter 4. After completing the Behavioral model, verification of the model was needed. Fig. 10 shows the program that was used to test the Behavioral model. Fig. 10 displays a program designed to perform data transfers and two arithmetic operations that will now be reviewed along with the results of the simulation.

```
Program
LJMP  0AH
MOV  A, #08h
MOV PSW, A
MOV A, #04h
MOV 09h, A
MOV 08h, A
ADD  A, R1
ADD  A, R0
```

Fig. 10. Test Code

The boldface entries in Fig. 10 correspond to the simulation results in Fig. 11 on the next page. The ST signal represents the state of the controller as discussed in chapter 4. The first instruction is LJMP 0Ah. The LJMP instruction, as detailed in table 6 in chapter 4, should cause the state machine to cycle through three states, fetch, addr1, and addr2. These expected results as well as the program counter counting from 00h to 01h

to 02h and then to 0Ah are shown below. The next instruction performs a data transfer of immediate data to the accumulator.

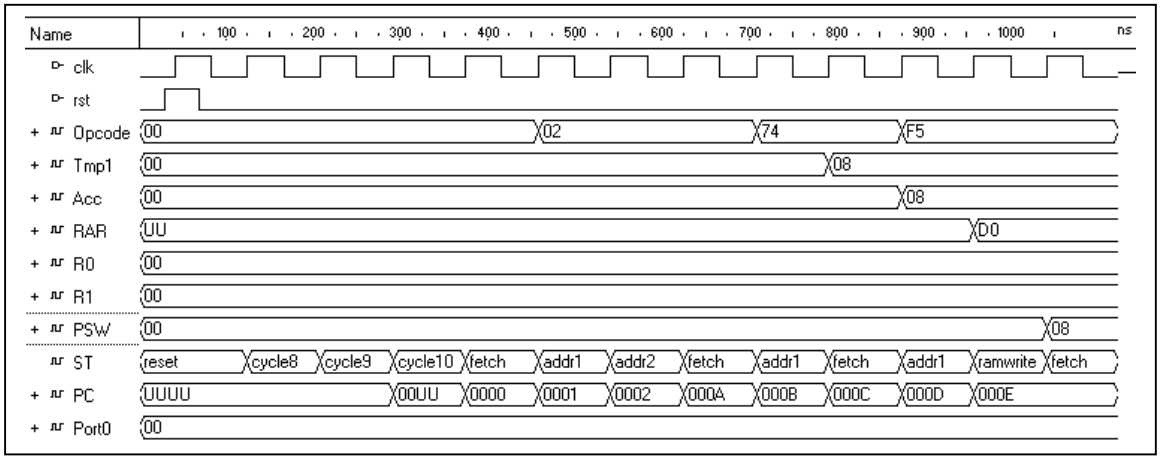


Fig. 11. Behavioral Simulation Chart 1

The MOV instruction, MOV A, #08h just needs two cycles for execution. It can be seen that while two cycles are used; the completion of the instruction is carried out during the next fetch. The MOV A, #08h instruction result is shown by seeing that the accumulator is updated with 08h. The Program Status Word (PSW) is updated by the next instruction, MOV PSW, A. This MOV instruction requires an additional cycle for a ramwrite as shown on the ST signal. At this point three instructions have been executed. The overall results being that the test program was initiated and that the PSW register now has a value of 08h. The value of 08h for the PSW selects the first bank of registers in the RAM for register addressing instructions. This was completed to give the necessary direction to the controller for register addressing instructions like the two ADD instructions to be discussed later.

Another signal to review is the Port 0 signal. This address/data port is never updated. Study of the architecture diagram in chapter 3 indicates that data should be transferred on this port. For this Behavioral model, the program memory model was included in the CPU. Hence the instruction fetches and the return of data from an external ROM was not visible at the port level. This issue was later addressed with the IS model. Chapter 7 will provide discussion of the IS model that will demonstrate the necessary port activity for Port 0.

The next three instructions, shown in Fig. 12 in bold type, directs the Intel 8031 to transfer an immediate value of 04h to two registers within the RAM. Looking at the Opcode signal in Fig. 13 on the next page one can see instruction 74h being read. The Tmp1 signal shows what value is going to move to the accumulator.

```
Program
LJMP  0AH
MOV  A, #08h
MOV PSW, A
MOV A, #04h
MOV 09h, A
MOV 08h, A
ADD  A, R1
ADD  A, R0
```

Fig. 12. Test Code

The MOV instruction is completed during the next fetch cycle with the accumulator being updated with a value of 04h. The next two instructions obtain the same results except for different registers. Register R1 or 09h gets a value of 04h as does register R0 or 08h.

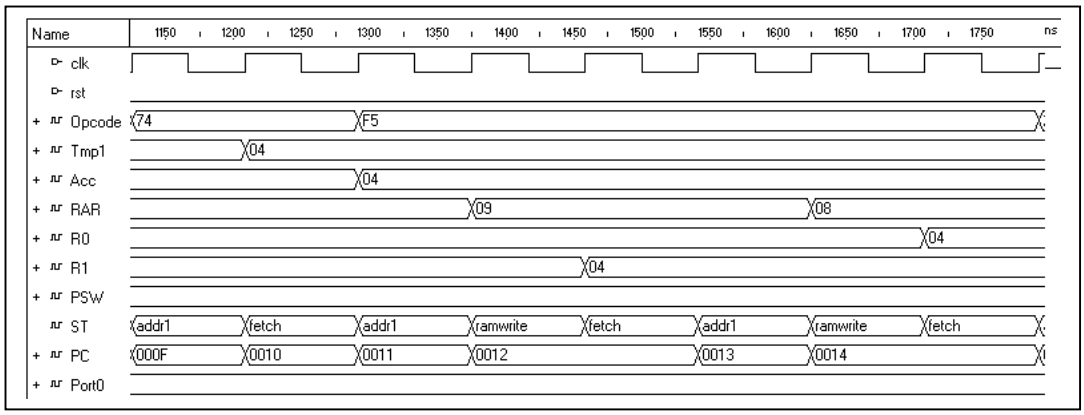


Fig. 13. Behavioral Simulation Chart 2

The final two instructions, in bold type in Fig.14, to be reviewed are two arithmetic operations. Both instructions perform an addition operation with the register and the accumulator with the results placed in the accumulator.

Program
LJMP 0AH
MOV A, #08h
MOV PSW, A
MOV A, #04h
MOV 09h, A
MOV 08h, A
ADD A, R1
ADD A, R0

Fig. 14. Test Code

Fig. 15 on the next page demonstrates instruction ADD A, R1 which will take the current value of 04h in Register R1 and add it to the accumulator which has 04h as it value. The result of this instruction is seen in the signal Acc with a value of 08h. ADD A, R0 will take the current value of 04h in Register R0 and add it to the accumulator that

has 08h as it contents. The result of this instruction is seen in signal Acc. Acc changes to 0Ch.

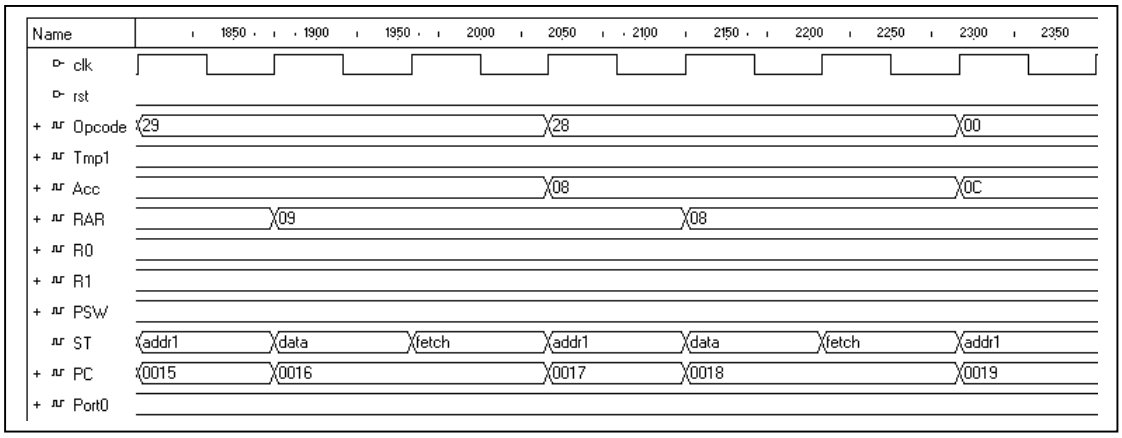


Fig. 15. Behavioral Simulation Chart 3

In summary, it has been demonstrated that the VHDL Behavioral model that produced the above simulation charts correctly executes the instructions that were chosen to be modeled.

7.0 IS Model and Simulation Results

The simulation results of chapter 6 have shown that the mini-instruction set Behavioral model of the Intel 8031 behaved correctly, as far as functionality is concerned. When reviewing the simulations with respect to timing, the waveforms are no where close to meeting the timing diagrams as specified by the Intel 8031 databook [5]. In translating the Behavioral model to an IS model much attention was paid to the timing characteristics of the microcontroller. The IS model needs to match the relative timing as seen in chapter 3.

Table 7. T Cycles and RTN for ADD,Rn

	ALE	$\overline{\text{PSEN}}$	P0Lat	RTN
T1	0	0	1	
T2	1	1	0	PC ←PA; PCinc ←PC
T3	1	1	0	P0,P2 ←PA; PC + 1
T4	0	1	0	PC ←PCinc
T5	0	0	0	P0 ←High Z
T6	0	0	0	
T7	0	0	1	
T8	1	1	0	IR←P0Lat
T9	1	1	0	RamAddReg ←(Rn); TMP2 ←ACC
T10	0	1	0	TMP1 ←Ram(Rn)
T11	0	0	0	ALUout ←TMP1 + TMP2
T12	0	0	0	ACC ←ALUout

Table 7 shows a T cycle table along with the critical control signals and their associated register transfers. Table 7 shown is for any ADD, Rn instruction. A table similar to this one was constructed for other single machine cycle instructions and for dual machine cycle instructions. As directed by the Intel data book, a single machine

cycle is broken into twelve evenly time-spaced steps. A blank table was constructed with twelve lines. Details were added to the table as extracted from the Intel 8031datasheet timing diagrams. In particular, the Address Latch Enable (ALE) and Program Store Enable active low (PSENn) signals that are critical to external memory fetches were included. If the timing of the ALE and PSENn signals are not correct then the IS model is worthless, i.e. the microcontroller will not properly interface with the external memory leaving the microcontroller "brain dead". In table 7 the ALE and PSENn columns have a 1 or 0 depending on Intel specified relative timing. See lines 454-489 of appendix D to see an example of how ALE and PSENn are toggled. The P0Latch column shows an entry of 1 were the data should be sampled from the external memory bus.

After building all of the RTN tables for several instructions, the IS model was constructed. Instead of having a cycle or state for the action needed to complete each instruction as with the Behavioral model, the IS model has a state machine with 24 states. If an instruction is termed a single machine cycle instruction then that instruction, through decoding, directs the state machine to use 12 cycles. If an instruction is termed a two-machine cycle instruction then that instruction will cause the state machine to cycle through 24 states. This will become apparent to the reader during the discussion of the simulation results. In the IS VHDL model each state of the CPU state machine process was named with much detail. This allowed for easier debugging (see lines 96-110 appendix D). As a consequence, the state names will not be visible in the simulations shown in the next section.

7.1 IS Model Simulation Results with Behavioral Code

Before explaining the results of the IS model, the timing of the instruction interactions should be discussed. In Fig. 16 below, one will notice that with the IS VHDL model the ST signal is comprised of many states. See lines 96-110 of appendix D to view the state names used. There is also activity on Port 0 and Port 2 that was not present in the Behavioral model. This port activity indicates that the IS model is interacting with the test bench and retrieving instructions from the virtual ROM. Also included on the simulation figures for the IS model are the signals ALE and PSENn. The activity on these signals also indicate that the IS model is communicating with the virtual external ROM.

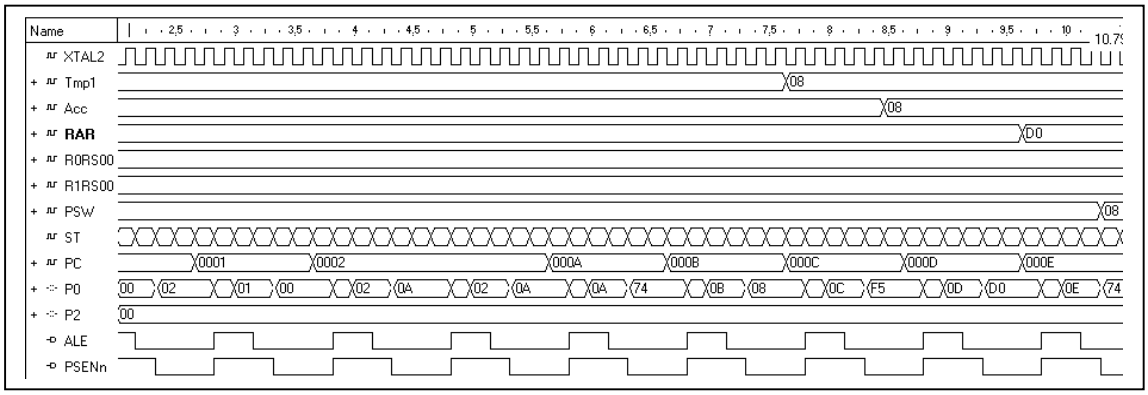


Fig. 16. IS Model Simulation Chart 1

The simulation results shown in Fig. 16 are similar to those for the Behavioral model and also cover the same first three instructions, a LJMP and two MOV's. Again we see that the end result of processing the LJMP instruction is that the PC is incremented to 0Ah. The first instruction, MOV A, #08h, completes as before with the value of 08h being transferred while the next instruction is being fetched. The third

instruction, MOV PSW, A, completes with the PSW register updated to a value of 08h as instructed.

The results for the next three instructions, entries in bold type in Fig. 17, are displayed in Fig. 18. The MOV A, #04h instruction executes with the same results as in the Behavioral model; the accumulator is updated with 08h. The next two MOV instructions are identical to those used before except for the specified destination registers. Both registers R1 and R0 of register bank 0 receive the value of 04h.

```
Program
LJMP  0AH
MOV  A, #08h
MOV PSW, A
MOV A, #04h
MOV 09h, A
MOV 08h, A
ADD  A, R1
ADD  A, R0
```

Fig. 17. Test Code

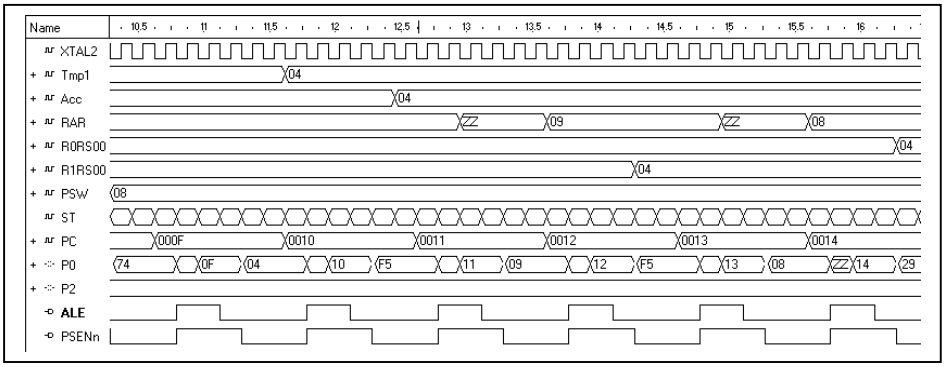


Fig. 18. IS Model Simulation Chart 2

The last two instructions of the test of code, as indicated by the bold type in Fig. 19, are arithmetic instructions. In Fig. 20, registers R1 and R0 of register bank zero are updated with the results from the ALU. The value stored in R1, 04h, is added to the accumulator and the result of the addition stored in the accumulator. The ADD A, R0 instruction operates the same way except targeting register R0. As a result the final value of the accumulator should be 0Ch and is. This result compares with Behavioral model result.

```
Program
LJMP  0AH
MOV  A, #08h
MOV PSW, A
MOV A, #04h
MOV 08h, A
MOV 09h, A
ADD  A, R1
ADD  A, R0
```

Fig. 19. Test Code

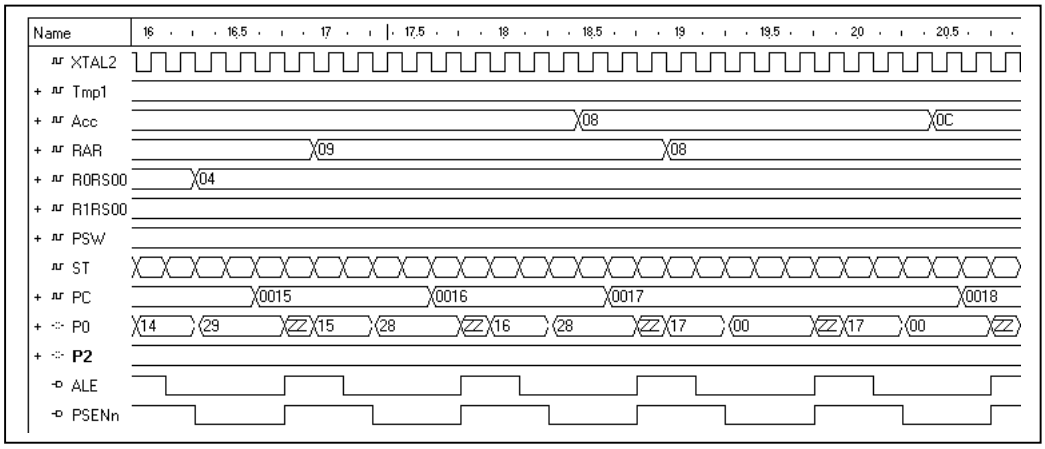


Fig. 20. IS Model Simulation Chart 3

7.2 IS Model Simulation Results with R0 Test Code

In the previous section, simulation results were shown for a simple eight-instruction program. A more exhaustive test of the register addressing instructions was completed. A program with 54 instructions was loaded by the test bench and re-simulated. A complete listing of the program used for IS model testing can be found in appendix B. For this test sequence, the results obtained from the hardware modeler, referred to as the "golden code" for the same 54 instructions, were used for comparison. A couple of the instructions for this new test sequence will be discussed in this section. A complete set of results for the IS model tests can be found in appendix C.

Fig. 21 shows again the beginning of the test program. This new program, like the last program, begins with a LJMP instruction. In Fig. 21 there are a couple new signals not present in all previous simulation charts.

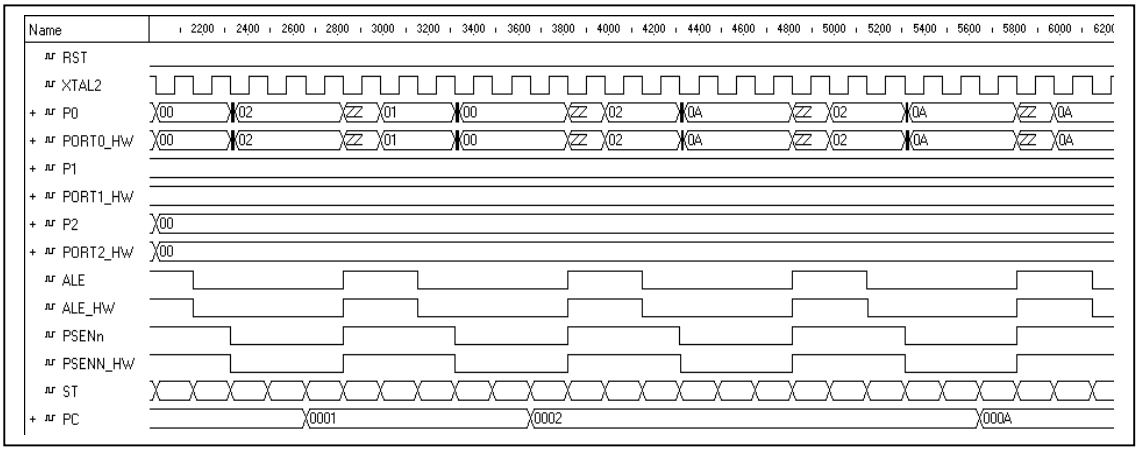


Fig. 21. IS Model Simulation Chart 4

The signals with the suffix _HW are signals that were played back from a text file as the simulation was running. The text file was generated from running the hardware modeler with the same program except the results were recorded from an actual Intel 8031 microcontroller. The signal results recorded from the real Intel 8031 are termed "golden vectors". This process of playing the "golden vectors" against the IS model was decided to be the verification method. While the hardware modeler does not allow access to the internal registers of the actual Intel 8031, results of all port activities were recorded. Since the internal register activity can not be viewed, instructions were chosen to exercise the ALU and the like and the results were observed on Port1. So if an addition instruction was executed the next instruction would be a port write instruction to see if the IS model would output the same result as the Intel 8031 chip as represented by the signals with the _HW extension.

In Fig. 22, again we see a simulation chart of an arithmetic instruction execution. This time a port write instruction was used to view the result of the arithmetic instruction. The result of this port write can be compared with the result from the hardware modeler. We see that on Port 0, instruction 08h i.e. increment R0, is being read form external memory. Since the events of the Intel 8031 internal registers could not be recorded, the register R0 signal was not included. The next instruction will let us see if an increment had really occurred. The previous value of register R0 was 01h. After the completion of the port write instruction on Port 1, we see that indeed the result is 02h and that matches the "golden code" recorded from the hardware model simulation.

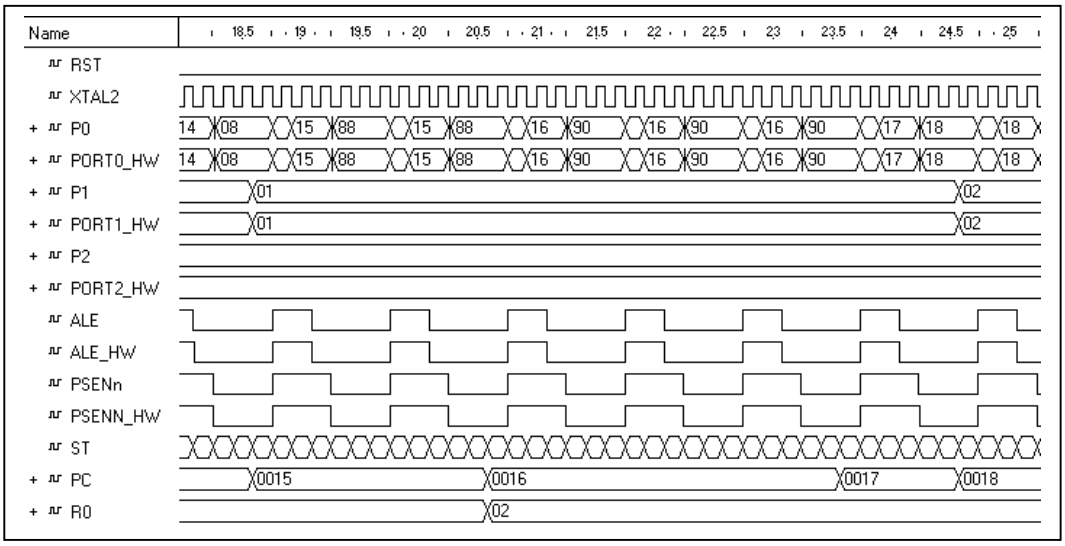


Fig. 22. IS Model Simulation Chart 5

As previously mentioned, the program for this test sequence contains 54 instructions. The remaining simulation results are similar to what has been presented here. See appendix C for a complete set of simulation charts. In summary, all tested instructions matched the results from the hardware modeler. The results obtained here greatly support the statement; the IS model developed in this research is an accurate model of the Intel 8031 for the instructions modeled.

8.0 Summary and Future Work

This thesis has detailed the results of creating and simulating 2 VHDL models for the Intel 8031. These models were written to help support an effort to eradicate the component obsolescence problem faced by the U.S. government. A simple Behavioral model was written and tested to verify proper abstract modeling. Next, an Instruction Set (IS) model was written for one half of the instruction set of the Intel 8031. The IS model results were compared against the results from an actual Intel 8031 accessed and simulated by using a Hardware Modeler made by Synopsys. For the modeled instructions, it was found that the IS model accurately emulates the functionality of the original Intel 8031. The efforts of this research will help support the overall goal to fabricate an 8-bit microcontroller that emulates with form, fit, and function the original Intel 8031 microcontroller.

Continuation of this research will be carried out at Ohio University. Items to research and complete include: finishing the IS model, writing an RTL model, synthesizing the RTL model, and testing all models. Some of this work has been completed, but was not documented in this thesis. RTL models and synthesis results have been made available to Dr. Starzyk.

References

[1] C. H. Roth, Jr., Digital Systems Design Using VHDL, PWS Publishing Company, Boston, Copyright1998 pp. 387-411.

[2] J. Bhasker, A VHDL Primer, Prentice-Hall, Inc. , New Jersey, Copyright 1995 pp 2-3.

[3] Synopsys, Logic Modeling Development Manual, Synopsys, Inc. Copyright 1994. pp. 1-2.

[4] IEEE Standard VHDL Language Reference Manual, Std 1076-1993, IEEE, New York, 1993.

[5] Intel, Embedded Microcontrollers and Processors Vol. I, Intel Corporation, 1993 pp. 5.3-5.17, 6.1 – 6.75, 7.1 – 7.34.

[6] D. Barker, "Why Are People Always Talking About VHDL" Air Force Research Laboratory Article, Wright-Patterson Air Force Base, Ohio, 1999. pp. 2-5.

Appendix A - Intel 8031 INSTRUCTION SET

Instruction Opcodes in Hexadecimal Order

Hex Code	Number of Bytes	Mnemonic	Operands
00	1	NOP	
01	2	AJMP	code addr
02	3	LJMP	code addr
03	1	RR	A
04	1	INC	A
05	2	INC	data addr
06	1	INC	@R0
07	1	INC	@R1
08	1	INC	R0
09	1	INC	R1
0A	1	INC	R2
0B	1	INC	R3
0C	1	INC	R4
0D	1	INC	R5
0E	1	INC	R6
0F	1	INC	R7
10	3	JBC	bit addr, code addr
11	2	ACALL	code addr
12	3	LCALL	code addr
13	1	RRC	A
14	1	DEC	A
15	2	DEC	data addr
16	1	DEC	@R0
17	1	DEC	@R1
18	1	DEC	R0
19	1	DEC	R1
1A	1	DEC	R2
1B	1	DEC	R3
1C	1	DEC	R4
1D	1	DEC	R5
1E	1	DEC	R6
1F	1	DEC	R7

Appendix A - Intel 8031 INSTRUCTION SET

Instruction Opcodes in Hexadecimal Order (Continued)

Hex Code	Number of Bytes	Mnemonic	Operands
20	3	JB	bit addr, code addr
21	2	AJMP	code addr
22	1	RET	
23	1	RL	A
24	2	ADD	A, # data
25	2	ADD	A, data addr
26	1	ADD	A, @ R0
27	1	ADD	A, @ R1
28	1	ADD	A,R0
29	1	ADD	A,R1
2A	1	ADD	A,R2
2B	1	ADD	A,R3
2C	1	ADD	A,R4
2D	1	ADD	A,R5
2E	1	ADD	A,R6
2F	1	ADD	A,R7
30	3	JNB	bit addr, code addr
31	2	ACALL	code addr
32	1	RETI	
33	1	RLC	A
34	2	ADDC	A, # data
35	2	ADDC	A, data addr
36	1	ADDC	A, @ R0
37	1	ADDC	A, @ R1
38	1	ADDC	A,R0
39	1	ADDC	A,R1
3A	1	ADDC	A,R2
3B	1	ADDC	A,R3
3C	1	ADDC	A,R4
3D	1	ADDC	A,R5
3E	1	ADDC	A,R6
3F	1	ADDC	A,R7

Appendix A - Intel 8031 INSTRUCTION SET

Instruction Opcodes in Hexadecimal Order (Continued)

Hex Code	Number of Bytes	Mnemonic	Operands
40	2	JC	code addr
41	2	AJMP	code addr
42	2	ORL	data addr, A
43	3	ORL	data addr, # data
44	2	ORL	A, # data
45	2	ORL	A, data addr
46	1	ORL	A, @ R0
47	1	ORL	A, @ R1
48	1	ORL	A,R0
49	1	ORL	A,R1
4A	1	ORL	A,R2
4B	1	ORL	A,R3
4C	1	ORL	A,R4
4D	1	ORL	A,R5
4E	1	ORL	A,R6
4F	1	ORL	A,R7
50	2	JNC	code addr
51	2	ACALL	code addr
52	2	ANL	data addr, A
53	3	ANL	data addr, # data
54	2	ANL	A, # data
55	2	ANL	A, data addr
56	1	ANL	A, @ R0
57	1	ANL	A, @ R1
58	1	ANL	A,R0
59	1	ANL	A,R1
5A	1	ANL	A,R2
5B	1	ANL	A,R3
5C	1	ANL	A,R4
5D	1	ANL	A,R5
5E	1	ANL	A,R6
5F	1	ANL	A,R7

Appendix A - Intel 8031 INSTRUCTION SET

Instruction Opcodes in Hexadecimal Order (Continued)

Hex Code	Number of Bytes	Mnemonic	Operands
60	2	JZ	code addr
61	2	AJMP	code addr
62	2	XRL	data addr, A
63	3	XRL	data addr, # data
64	2	XRL	A, # data
65	2	XRL	A, data addr
66	1	XRL	A, @ R0
67	1	XRL	A, @ R1
68	1	XRL	A,R0
69	1	XRL	A,R1
6A	1	XRL	A,R2
6B	1	XRL	A,R3
6C	1	XRL	A,R6
6D	1	XRL	A,R5
6E	1	XRL	A,R6
6F	1	XRL	A,R7
70	2	JNZ	code addr
71	2	ACALL	code addr
72	2	ORL	C,bit addr
73	1	JMP	@A + DPTR
74	2	MOV	A, # data
75	3	MOV	data addr, # data
76	2	MOV	@ R0, # data
77	2	MOV	@ R1, # data
78	2	MOV	R0, # data
79	2	MOV	R1, # data
7A	2	MOV	R2, # data
7B	2	MOV	R3, # data
7C	2	MOV	R4, # data
7D	2	MOV	R5, # data
7E	2	MOV	R6, # data
7F	2	MOV	R7, # data

Appendix A - Intel 8031 INSTRUCTION SET

Instruction Opcodes in Hexadecimal Order (Continued)

Hex Code	Number of Bytes	Mnemonic	Operands
80	2	SJMP	code addr
81	2	AJMP	code addr
82	2	ANL	C,bit addr
83	1	MOVC	A,@A + PC
84	1	DIV	AB
85	3	MOV	data addr, data addr
86	2	MOV	data addr, @ R0
87	2	MOV	data addr, @ R1
88	2	MOV	data addr,R0
89	2	MOV	data addr,R1
8A	2	MOV	data addr,R2
8B	2	MOV	data addr,R3
8C	2	MOV	data addr,R8
8D	2	MOV	data addr,R5
8E	2	MOV	data addr,R8
8F	2	MOV	data addr,R7
90	3	MOV	code addr
91	2	ACALL	code addr
92	2	MOV	C,bit addr
93	1	MOVC	@A + DPTR
94	2	SUBB	A, # data
95	2	SUBB	data addr, # data
96	1	SUBB	@ R0, # data
97	1	SUBB	@ R1, # data
98	1	SUBB	R0, # data
99	1	SUBB	R1, # data
9A	1	SUBB	R2, # data
9B	1	SUBB	R3, # data
9C	1	SUBB	R4, # data
9D	1	SUBB	R5, # data
9E	1	SUBB	R8, # data
9F	1	SUBB	R7, # data

Appendix A - Intel 8031 INSTRUCTION SET

Instruction Opcodes in Hexadecimal Order (Continued)

Hex Code	Number of Bytes	Mnemonic	Operands
A0	2	ORL	C,/bit addr
A1	2	AJMP	code addr
A2	2	MOV	C,bit addr
A3	1	INC	DPTR
A4	1	MUL	AB
A5		reserved	
A6	2	MOV	@ R0, data addr
A7	2	MOV	@ R1, data addr
A8	2	MOV	R0, data addr
A9	2	MOV	R1, data addr
AA	2	MOV	R2, data addr
AB	2	MOV	R3, data addr
AC	2	MOV	R4, data addr
AD	2	MOV	R5, data addr
AE	2	MOV	R6, data addr
AF	2	MOV	R7, data addr
B0	2	ANL	C,/bit addr
B1	2	ACALL	code addr
B2	2	CPL	bit addr
B3	1	CPL	C
B4	3	CJNE	A, # data, code addr
B5	3	CJNE	A, data addr, code addr
B6	3	CJNE	@ R0, # data, code addr
B7	3	CJNE	@ R1, # data, code addr
B8	3	CJNE	R0, # data, code addr
B9	3	CJNE	R1, # data, code addr
BA	3	CJNE	R2, # data, code addr
BB	3	CJNE	R3, # data, code addr
BC	3	CJNE	R4, # data, code addr
BD	3	CJNE	R5, # data, code addr
BE	3	CJNE	R8, # data, code addr
BF	3	CJNE	R7, # data, code addr

Appendix A - Intel 8031 INSTRUCTION SET

Instruction Opcodes in Hexadecimal Order (Continued)

Hex Code	Number of Bytes	Mnemonic	Operands
C0	2	PUSH	data addr
C1	2	AJMP	code addr
C2	2	CLR	bit addr
C3	1	CLR	C
C4	1	SWAP	A
C5	2	XCH	A, data addr
C6	1	XCH	A,@ R0
C7	1	XCH	A,@ R1
C8	1	XCH	A,R0
C9	1	XCH	A,R1
CA	1	XCH	A,R2
CB	1	XCH	A,R3
CC	1	XCH	A,R4
CD	1	XCH	A,R5
CE	1	XCH	A,R6
CF	1	XCH	A,R7
D0	2	POP	data addr
D1	2	ACALL	code addr
D2	2	SETB	bit addr
D3	1	SETB	C
D4	1	DA	A
D5	3	DJNZ	data addr, code addr
D6	1	XCHD	A,@ R0
D7	1	XCHD	A,@ R1
D8	2	DJNZ	R0, code addr
D9	2	DJNZ	R1, code addr
DA	2	DJNZ	R2, code addr
DB	2	DJNZ	R3, code addr
DC	2	DJNZ	R4, code addr
DD	2	DJNZ	R5, code addr
DE	2	DJNZ	R6, code addr
DF	2	DJNZ	R7, code addr

Appendix A - Intel 8031 INSTRUCTION SET

Instruction Opcodes in Hexadecimal Order (Continued)

Hex Code	Number of Bytes	Mnemonic	Operands
E0	1	MOVX	A, @ DPTR
E1	2	AJMP	code addr
E2	1	MOVX	A, @ R0
E3	1	MOVX	A, @ R1
E4	1	CLR	A
E5	2	MOV	A, data addr
E6	1	MOV	A,@ R0
E7	1	MOV	A,@ R1
E8	1	MOV	A,R0
E9	1	MOV	A,R1
EA	1	MOV	A,R2
EB	1	MOV	A,R3
EC	1	MOV	A,R4
ED	1	MOV	A,R5
EE	1	MOV	A,R6
EF	1	MOV	A,R7
F0	1	MOVX	code addr
F1	2	ACALL	code addr
F2	1	MOVX	C,bit addr
F3	1	MOVX	@A + DPTR
F4	1	CPL	A, # data
F5	2	MOV	data addr, A
F6	1	MOV	@ R0, A
F7	1	MOV	@ R1, A
F8	1	MOV	R0,A
F9	1	MOV	R1,A
FA	1	MOV	R2,A
FB	1	MOV	R3,A
FC	1	MOV	R4,A
FD	1	MOV	R5,A
FE	1	MOV	R6,A
FF	1	MOV	R7,A

Appendix B - Instruction Set Model Test Code

This following test program was used to test the Register 0 Register Addressing Mode. Results from this test code are available in Appendix C.

Reg Status	RTN	Mnemonic	Operands	Op Code
A = 01h	A ←#01h	MOV	A, #01h	74 01
R0 = 1d	R0 ← A	MOV	08, A	F5 08
P1 = 1d	P1 ←R0	MOV	90, R0	88 90
R0 = 2d		INC	R0	08
P1 = 2d	P1 ←R0	MOV	90, R0	88 90
R0 = 1d		DEC	R0	18
P1 = 1d	P1 ←R0	MOV	90, R0	88 90
A = 2d		ADD	A, R0	28
P1 = 2d	P1 ←A	MOV	90, A	F5 90
C = 1	C ←1	SETB	C	D3
A = 4d	A ←A + C + R0	ADDC	A, R0	38
P1 = 4d	P1 ←A	MOV	90, A	F5 90
	A ← 5d	ORL	A, R0	48
P1 = 5d	P1 ←A	MOV	90, A	F5 90
	A ← 1d	ANL	A, R0	58
P1 = 1d	P1 ←A	MOV	90, A	F5 90
	A ← 00h	XRL	A, R0	68
P1 = 00h	P1 ←A	MOV	90, A	F5 90
	R0 ←20h	MOV	R0, #20h	78 20
P1 = 20h	P1 ←R0	MOV	90, R0	88 90

Appendix B - Instruction Set Model Test Code

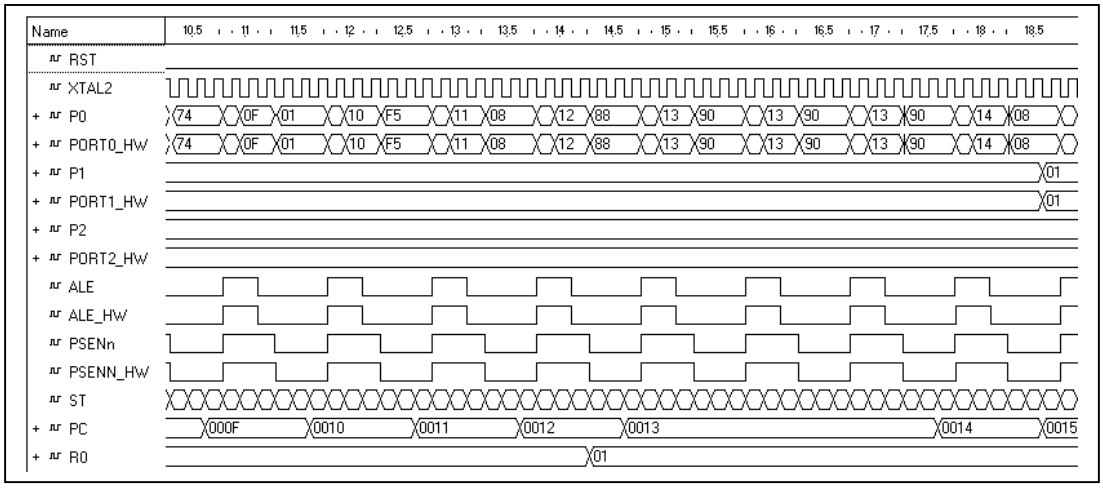
Reg Status	RTN	Mnemonic	Operands	Op Code
	A ← R0	MOV	A, R0	E8
P1 = 20h	P1 ←A	MOV	90, A	F5 90
	R0 ←0Bh	MOV	R0, #0Bh	78 0B
P1 = 0Bh	P1 ←R0	MOV	90, R0	88 90
	C ←0	CLR	C	C3
	C ←1	SETB	C	D3
	A ←A - C - R0	SUBB	A, R0	98
P1 = 14h	P1 ←A	MOV	90, A	F5 90
	R0 ←#22h	MOV	R0, #22h	78 22
B3 = 22h	B3 ←R0	MOV	B3, R0	88 B3
	R0 ←#48h	MOV	R0, #48h	78 48
	R0 ←B3	MOV	R0, B3	A8 B3
P1 = 22h	P1 ←R0	MOV	90, R0	88 90
R0 = 01h	R0 ←#01h	MOV	R0, #01h	78 01
		DJNZ	R0, 10h	D8 10
R0 = 02h	R0 ←#02h	MOV	R0, #02h	78 02
		DJNZ	R0, 10h	D8 04
				00 00
				00 00
R0 = #48h	R0 ←#48h	MOV	R0, #48h	78 48
		CJNE	R0, #48h	B8 48 04
		CJNE	R0, #50h	B8 50 04
				00 00
				00 00
R0 = 03h	R0 ←#03h	MOV	R0, #03h	78 03
A = 0Ch	A ←0Ch	MOV	A, 0Ch	74 0C
		XCH	A, R0	C8
P1 = 03h	P1 ←A	MOV	90, A	F5 90
P1 = 0Ch	P1 ←R0	MOV	90, R0	88 90

Appendix B - Instruction Set Model Test Code

Reg Status	RTN	Mnemonic	Operands	Op Code
R0 = #05h	R0 ←#05h	MOV	R0, #05h	78 05
A = #09h	A ←#09h	MOV	A, #09h	74 09
		MOV	A, R0	E8
P1 = 05h	P1 ←A	MOV	90, A	F5 90
R0 = #02h	R0 ←#02h	MOV	R0, #02h	78 02
A = #04h	A ←#04h	MOV	A, #04h	74 04
		MOV	R0, A	F8
P1 = 04h	P1 ←R0	MOV	90, R0	88 90

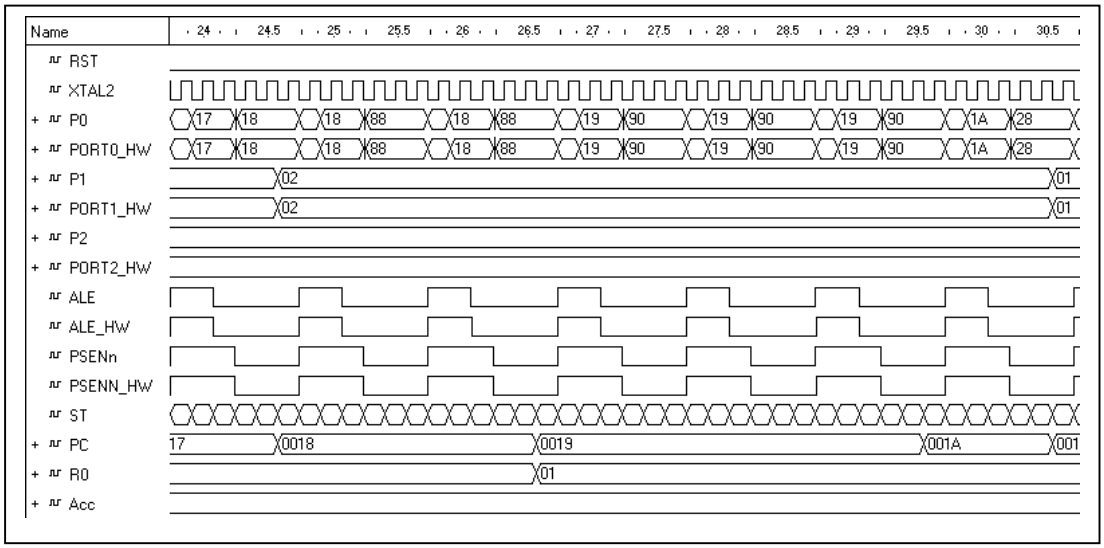
Appendix C - Instruction Set Model Simulation Results with Test Code

Reg Status	RTN	Mnemonic	Operands	Op Code
A = 01h	A ←#01h	MOV	A, #01h	74 01
R0 = 1d	R0 ← A	MOV	08, A	F5 08
P1 = 1d	P1 ←R0	MOV	90, R0	88 90



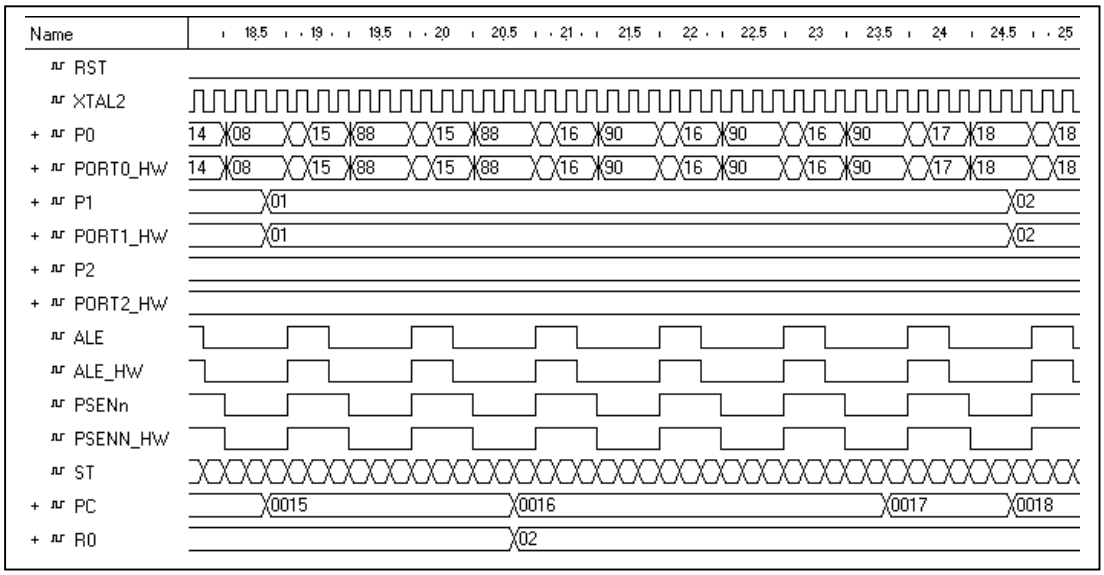
Appendix C - Instruction Set Model Simulation Results with Test Code

Reg Status	RTN	Mnemonic	Operands	Op Code
R0 = 1d		DEC	R0	18
P1 = 1d	P1 ←R0	MOV	90, R0	88 90



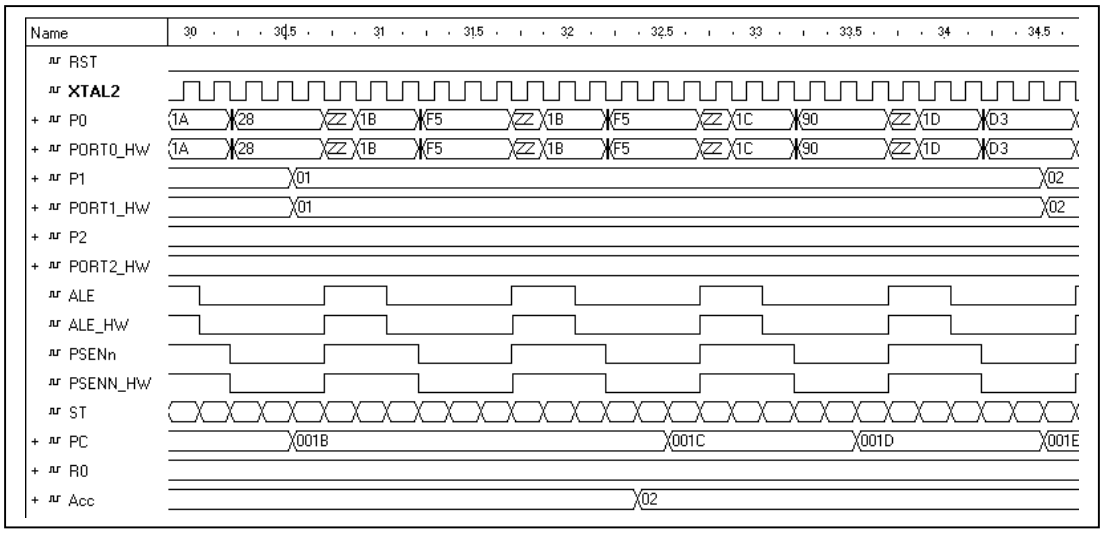
Appendix C - Instruction Set Model Simulation Results with Test Code

Reg Status	RTN	Mnemonic	Operands	Op Code
R0 = 2d		INC	R0	08
P1 = 2d	P1 ←R0	MOV	90, R0	88 90



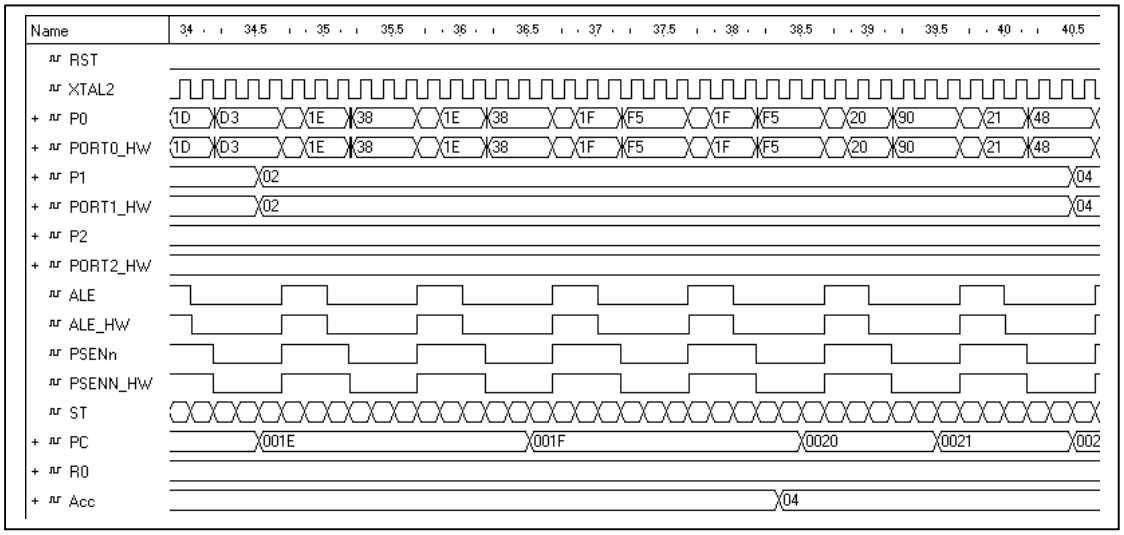
Appendix C - Instruction Set Model Simulation Results with Test Code

Reg Status	RTN	Mnemonic	Operands	Op Code
A = 2d		ADD	A, R0	28
P1 = 2d	P1 ←A	MOV	90, A	F5 90



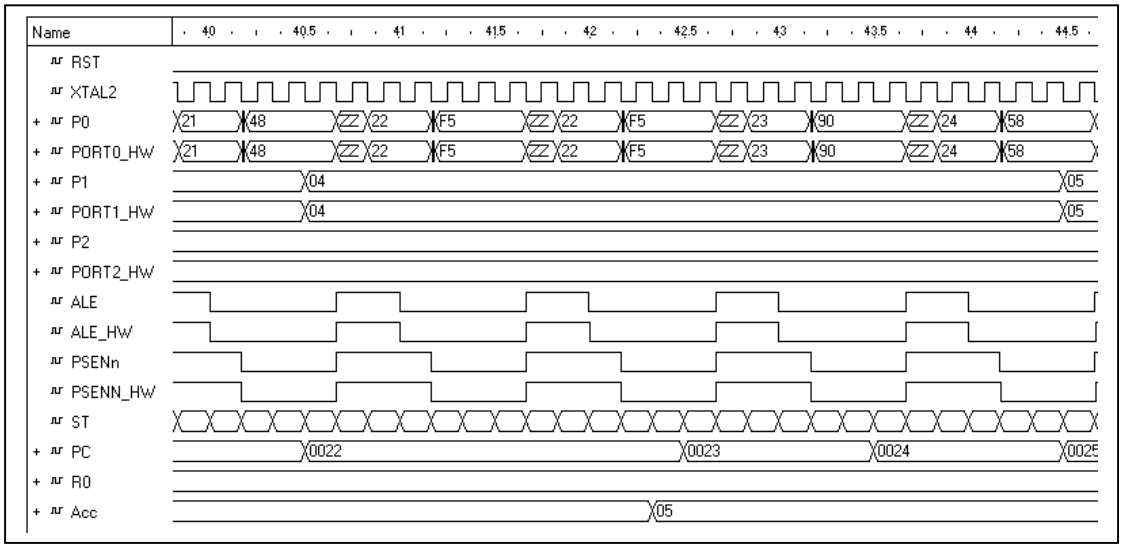
Appendix C - Instruction Set Model Simulation Results with Test Code

Reg Status	RTN	Mnemonic	Operands	Op Code
C = 1	C ←1	SETB	C	D3
A = 4d	A ←A + C + R0	ADDC	A, R0	38
P1 = 4d	P1 ←A	MOV	90, A	F5 90



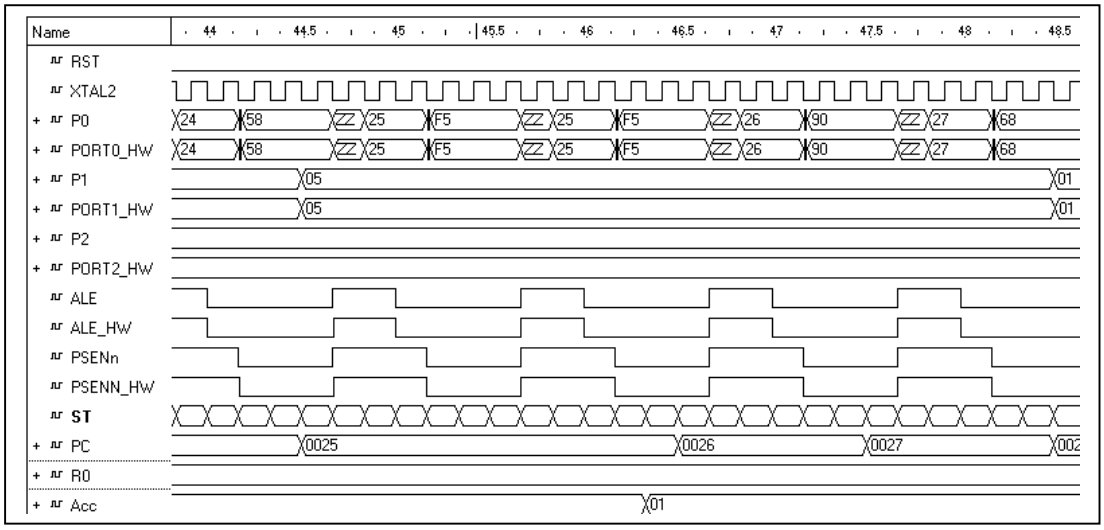
Appendix C - Instruction Set Model Simulation Results with Test Code

Reg Status	RTN	Mnemonic	Operands	Op Code
	A ← 5d	ORL	A, R0	48
P1 = 5d	P1 ←A	MOV	90, A	F5 90



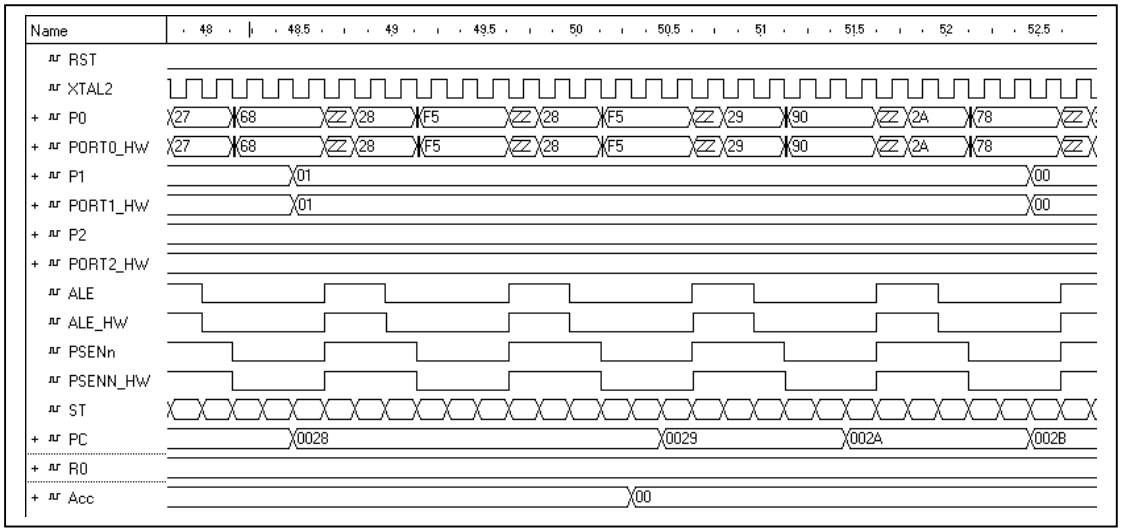
Appendix C - Instruction Set Model Simulation Results with Test Code

Reg Status	RTN	Mnemonic	Operands	Op Code
	A ← 1d	ANL	A, R0	58
P1 = 1d	P1 ←A	MOV	90, A	F5 90



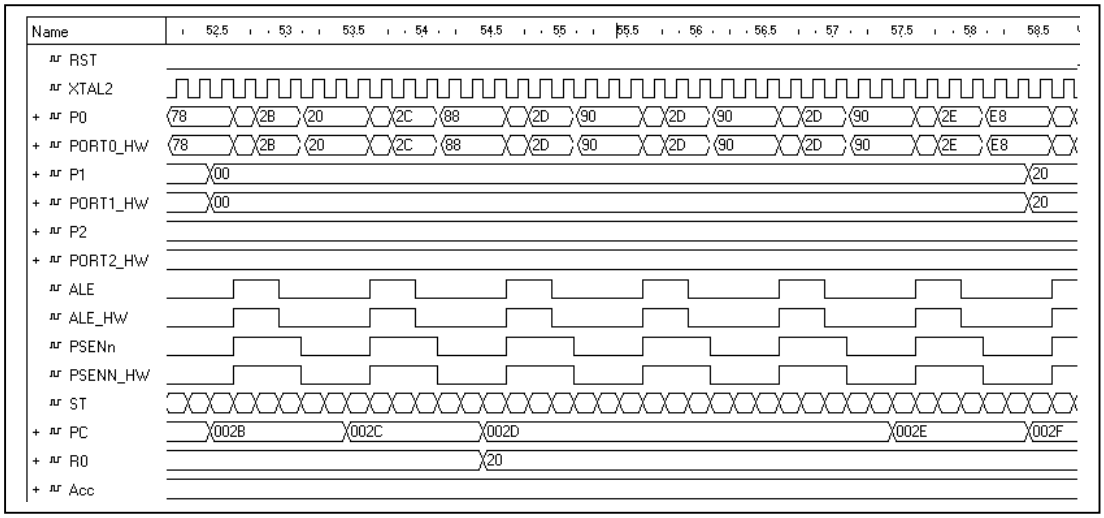
Appendix C - Instruction Set Model Simulation Results with Test Code

Reg Status	RTN	Mnemonic	Operands	Op Code
	A ← 00h	XRL	A, R0	68
P1 = 00h	P1 ←A	MOV	90, A	F5 90



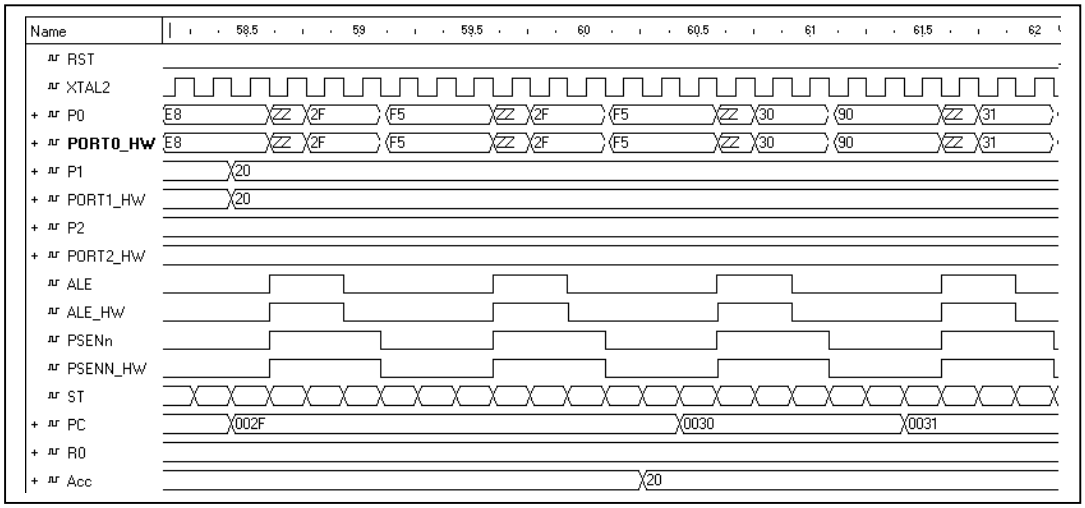
Appendix C - Instruction Set Model Simulation Results with Test Code

Reg Status	RTN	Mnemonic	Operands	Op Code
	R0 ←20h	MOV	R0, #20h	78 20
P1 = 20h	P1 ←R0	MOV	90, R0	88 90



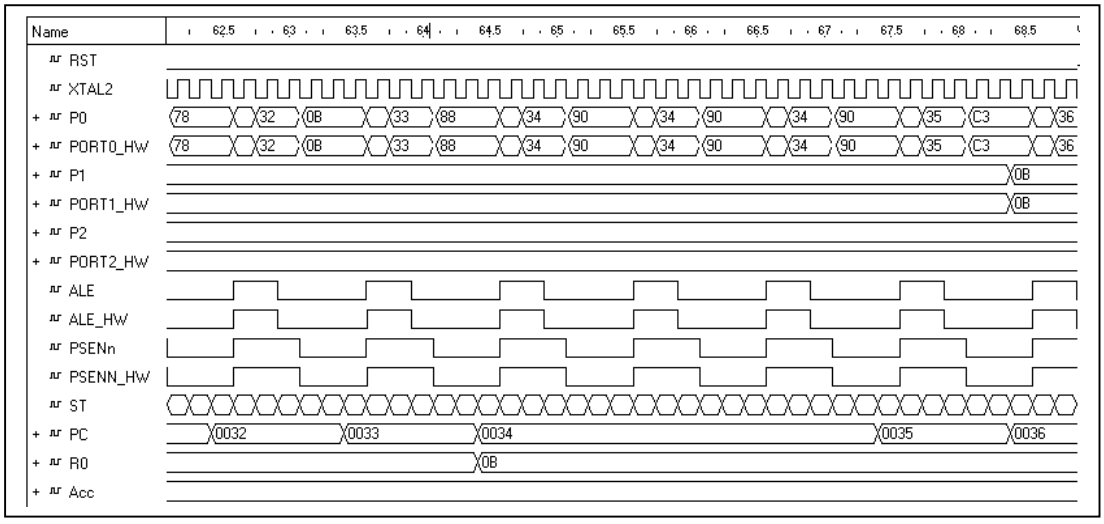
Appendix C - Instruction Set Model Simulation Results with Test Code

Reg Status	RTN	Mnemonic	Operands	Op Code
	A ← R0	MOV	A, R0	E8
P1 = 20h	P1 ←A	MOV	90, A	F5 90



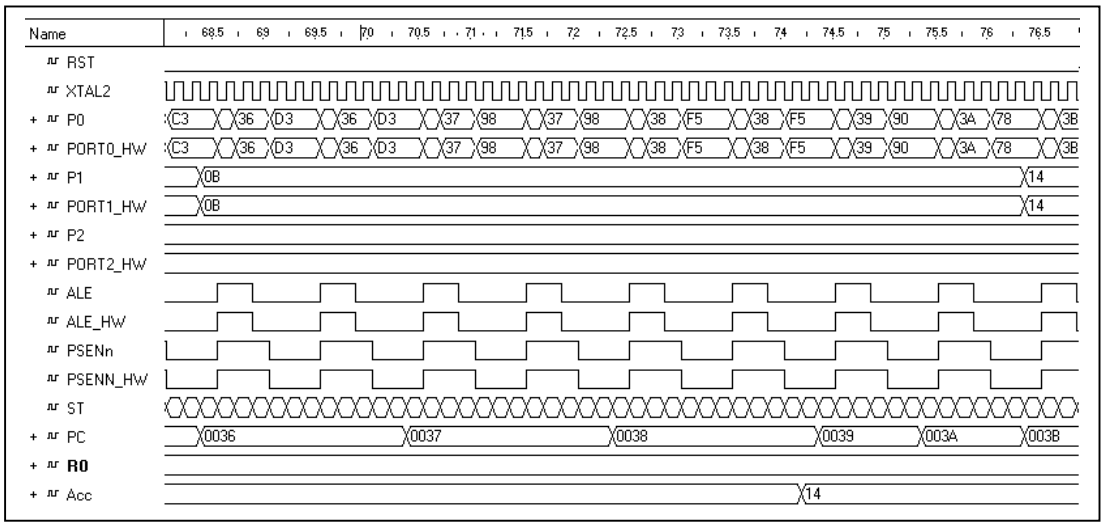
Appendix C - Instruction Set Model Simulation Results with Test Code

Reg Status	RTN	Mnemonic	Operands	Op Code
	R0 ←0Bh	MOV	R0, #0Bh	78 0B
P1 = 0Bh	P1 ←R0	MOV	90, R0	88 90



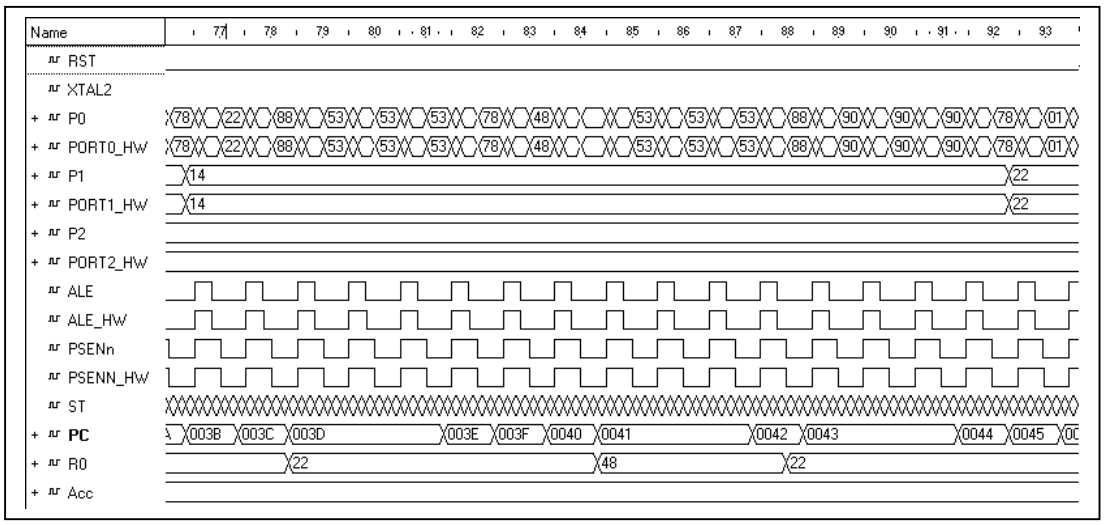
Appendix C - Instruction Set Model Simulation Results with Test Code

Reg Status	RTN	Mnemonic	Operands	Op Code
	C ←0	CLR	C	C3
	C ←1	SETB	C	D3
	A ←A - C - R0	SUBB	A, R0	98
P1 = 14h	P1 ←A	MOV	90, A	F5 90



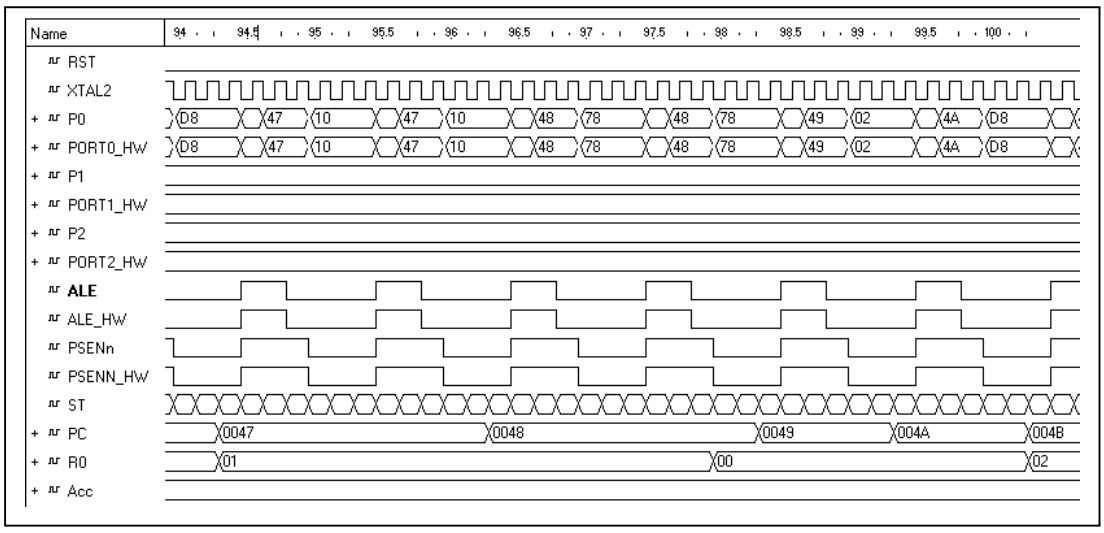
Appendix C - Instruction Set Model Simulation Results with Test Code

Reg Status	RTN	Mnemonic	Operands	Op Code
	R0 ←#22h	MOV	R0, #22h	78 22
B3 = 22h	B3 ←R0	MOV	B3, R0	88 B3
	R0 ←#48h	MOV	R0, #48h	78 48
	R0 ←B3	MOV	R0, B3	A8 B3
P1 = 22h	P1 ←R0	MOV	90, R0	88 90



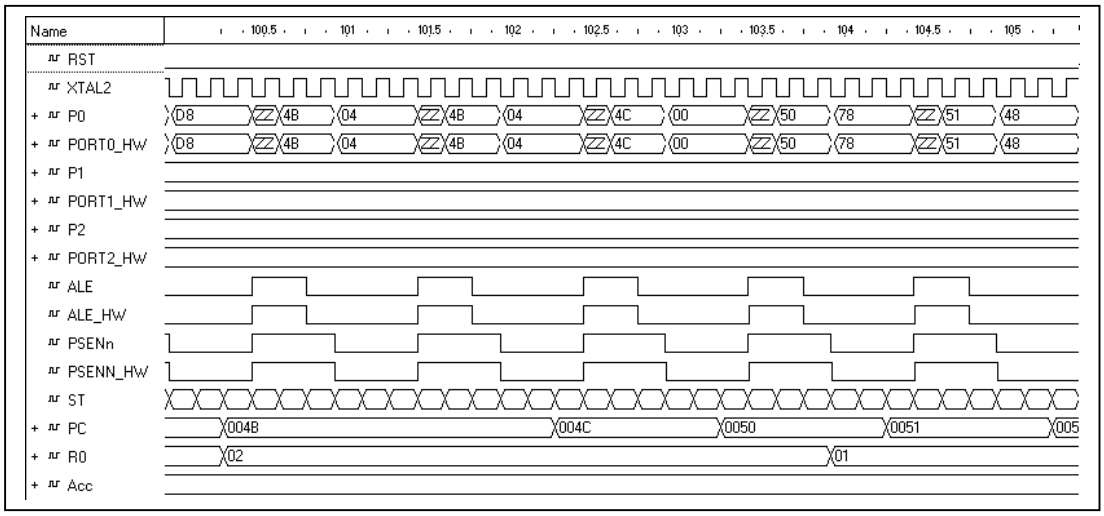
Appendix C - Instruction Set Model Simulation Results with Test Code

Reg Status	RTN	Mnemonic	Operands	Op Code
R0 = 01h	R0 ←#01h	MOV	R0, #01h	78 01
		DJNZ	R0, 10h	D8 10
R0 = 02h	R0 ←#02h	MOV	R0, #02h	78 02
		DJNZ	R0, 10h	D8 04



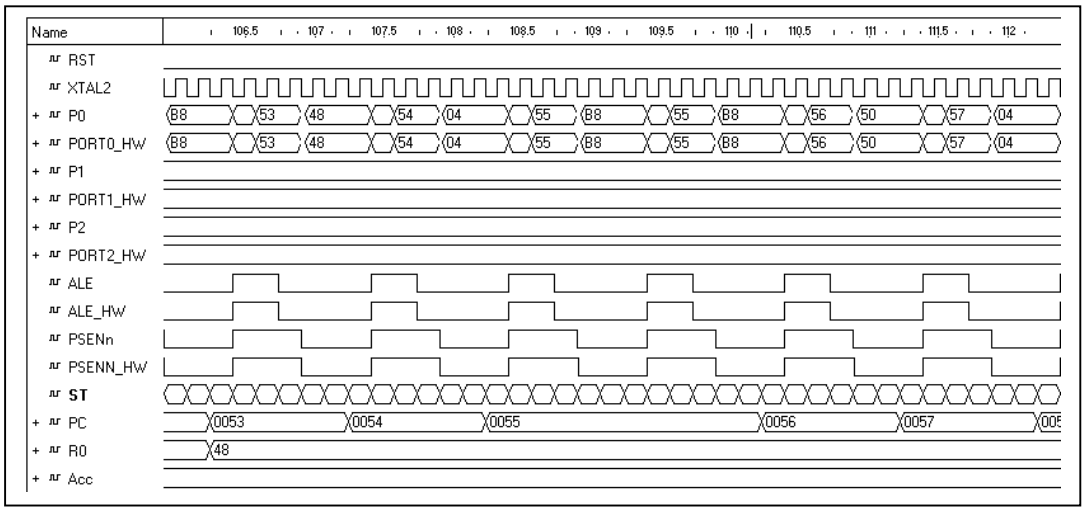
Appendix C - Instruction Set Model Simulation Results with Test Code

Reg Status	RTN	Mnemonic	Operands	Op Code
		DJNZ	R0, 10h	D8 04 00 00 00 00
R0 = #48h	R0 ←#48h	MOV	R0, #48h	78 48



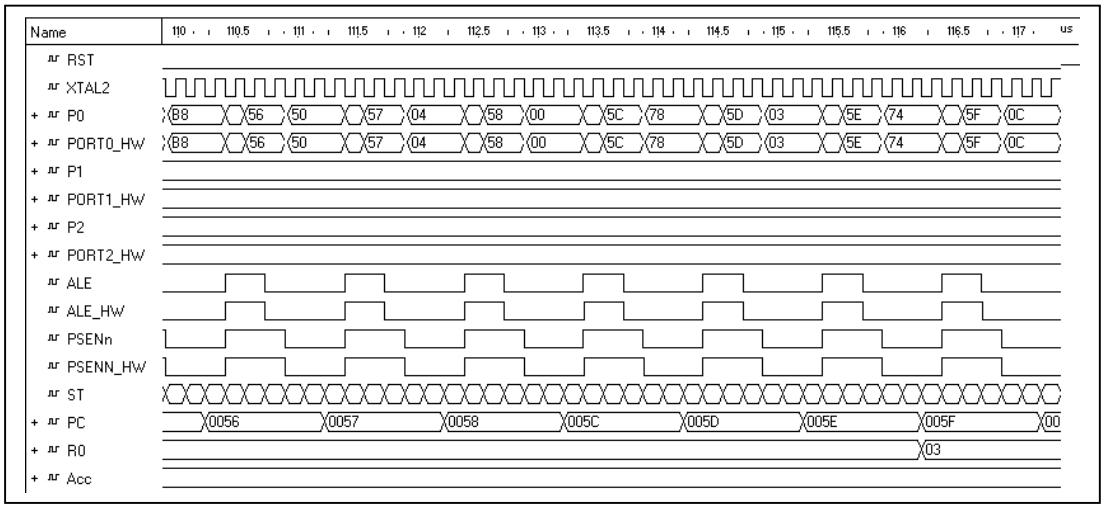
Appendix C - Instruction Set Model Simulation Results with Test Code

Reg Status	RTN	Mnemonic	Operands	Op Code
R0 = #48h	R0 ←#48h	MOV	R0, #48h	78 48
		CJNE	R0, #48h	B8 48 04
		CJNE	R0, #50h	B8 50 04



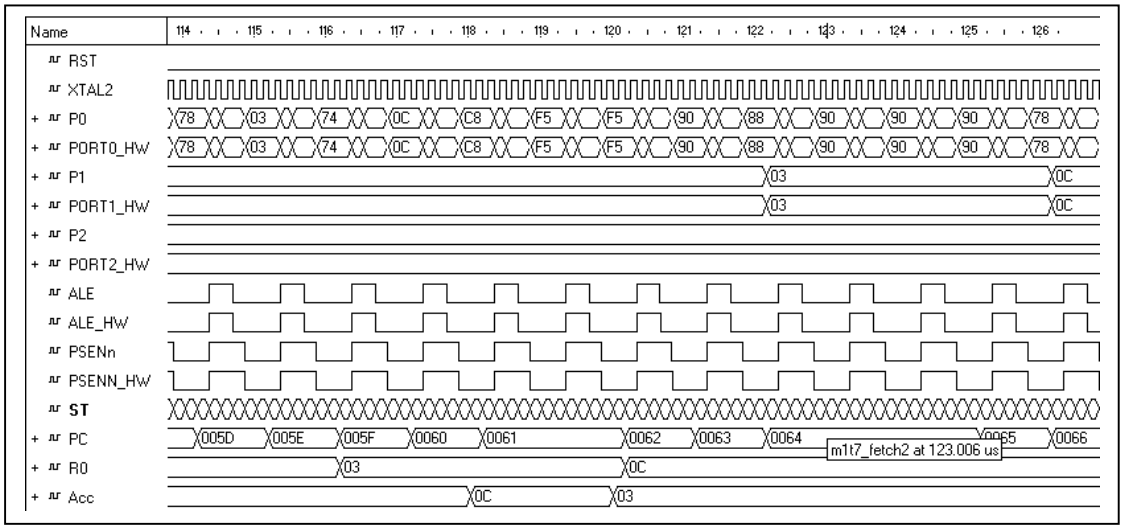
Appendix C - Instruction Set Model Simulation Results with Test Code

Reg Status	RTN	Mnemonic	Operands	Op Code
		CJNE	R0, #50h	B8 50 04 00 00 00 00
R0 = 03h	R0 ←#03h	MOV	R0, #03h	78 03



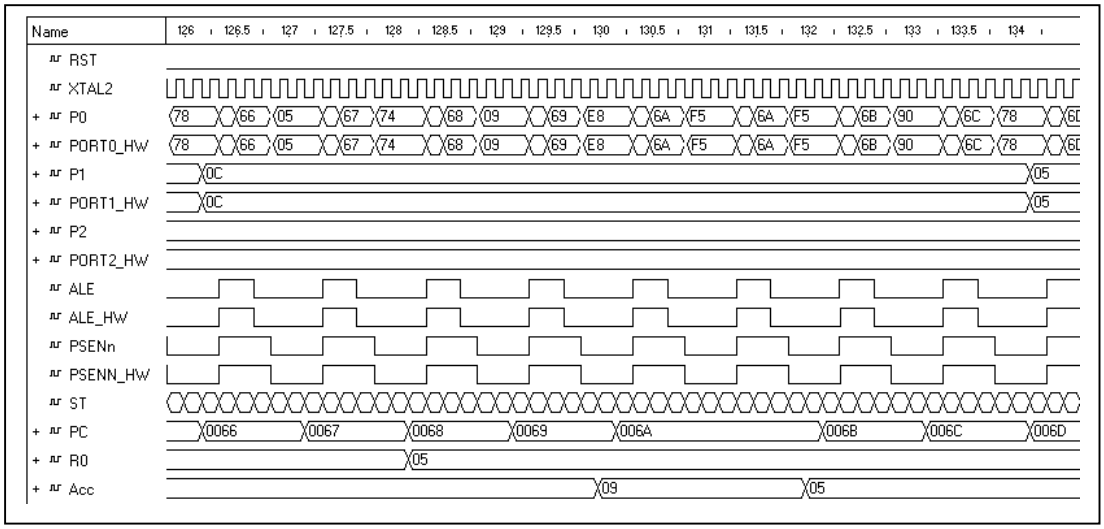
Appendix C - Instruction Set Model Simulation Results with Test Code

Reg Status	RTN	Mnemonic	Operands	Op Code
R0 = 03h	R0 ←#03h	MOV	R0, #03h	78 03
A = 0Ch	A ←0Ch	MOV	A, 0Ch	74 0C
		XCH	A, R0	C8
P1 = 03h	P1 ←A	MOV	90, A	F5 90
P1 = 0Ch	P1 ←R0	MOV	90, R0	88 90



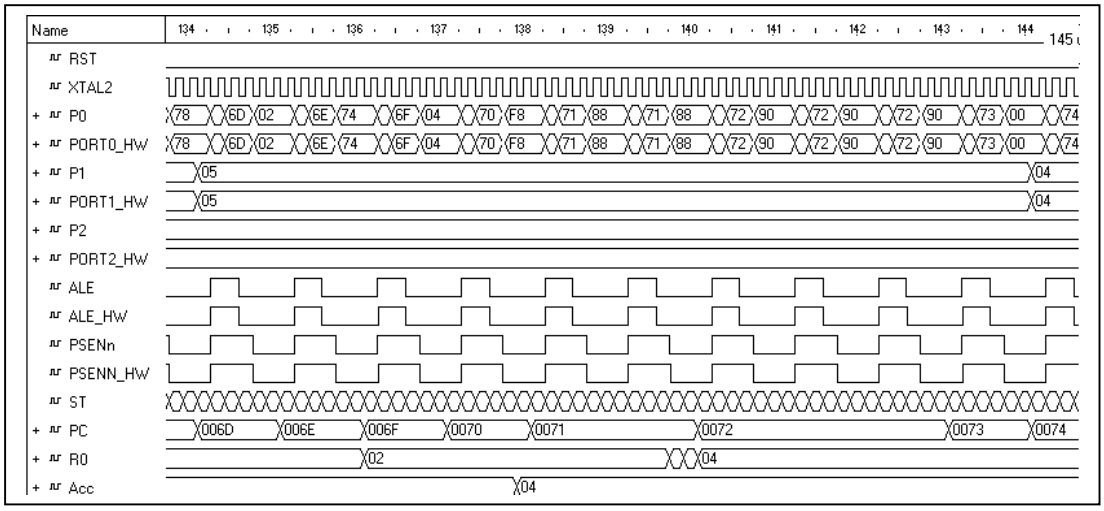
Appendix C - Instruction Set Model Simulation Results with Test Code

Reg Status	RTN	Mnemonic	Operands	Op Code
R0 = #05h	R0 ←#05h	MOV	R0, #05h	78 05
A = #09h	A ←#09h	MOV	A, #09h	74 09
		MOV	A, R0	E8
P1 = 05h	P1 ←A	MOV	90, A	F5 90



Appendix C - Instruction Set Model Simulation Results with Test Code

Reg Status	RTN	Mnemonic	Operands	Op Code
R0 = #02h	R0 ←#02h	MOV	R0, #02h	78 02
A = #04h	A ←#04h	MOV	A, #04h	74 04
		MOV	R0, A	F8
P1 = 04h	P1 ←R0	MOV	90, R0	88 90



Appendix D - Machine Cycle VHDL Code Example

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3   use IEEE.std_logic_unsigned.all;
4   --use IEEE.numeric_bit.all;
5   --use std.textio.all;
6
7   entity cpu8031 is
8       port (
9           P1:       inout std_logic_vector(7 downto 0);
10          RST:      in    std_logic;
11          P3:       inout std_logic_vector(7 downto 0);
12          XTAL2:    in    std_logic;
13          XTAL1:    in    std_logic;
14          P0:       inout std_logic_vector(7 downto 0);
15          EAn:      in    std_logic;
16          ALE:      out   std_logic;
17          PSENn:    out   std_logic;
18          P2:       inout std_logic_vector(7 downto 0)
19         );
20  end cpu8031;
21
22  architecture behavior of cpu8031 is
23
24        type IntRAMtype is array (0 to 255) of
25              std_logic_vector(7 downto 0);
26
27        signal IntMem: IntRAMtype:= (others=>(others=> '0'));
28
29        -- Upper Internal Memory
30        alias Acc:   std_logic_vector(7 downto 0) is IntMem(224);
31        alias B:     std_logic_vector(7 downto 0) is IntMem(240);
32
33        alias TL0:   std_logic_vector(7 downto 0) is IntMem(138);
34        alias TL1:   std_logic_vector(7 downto 0) is IntMem(139);
35        alias TH0:   std_logic_vector(7 downto 0) is IntMem(140);
36        alias TH1:   std_logic_vector(7 downto 0) is IntMem(141);
37        alias R0:   std_logic_vector(7 downto 0) is IntMem(8);
38        alias R1:   std_logic_vector(7 downto 0) is IntMem(9);
39
40
41        alias IP:   std_logic_vector(7 downto 0) is IntMem(219);
42        alias IE:   std_logic_vector(7 downto 0) is IntMem(168);
43
44        alias SCON: std_logic_vector(7 downto 0) is IntMem(152);
45        alias SBUF: std_logic_vector(7 downto 0) is IntMem(153);
46        alias PCON: std_logic_vector(7 downto 0) is IntMem(135);
47        alias TMOD: std_logic_vector(7 downto 0) is IntMem(137);
48        alias TCON: std_logic_vector(7 downto 0) is IntMem(136);
49        alias SP:   std_logic_vector(7 downto 0) is IntMem(129);
50        -- Stack Pointer
51
52
53
```

```
54          alias DPH:  std_logic_vector(7 downto 0) is IntMem(131);
55          -- Data Pointer High byte
56
57          alias DPL:  std_logic_vector(7 downto 0) is IntMem(130);
58          -- Data Pointer Low byte
59
60          alias PSW:  std_logic_vector(7 downto 0) is IntMem(208);
61          -- Program Status Word
62
63          alias CY:  std_logic is PSW(7);     -- Carry Flag
64          alias AC:  std_logic is PSW(6);     -- Auxiliary Carry Flag
65          alias FO:  std_logic is PSW(5);     -- Flag 0
66          alias RS1: std_logic is PSW(4);
67          -- Register Bank selector bit 1
68
69          alias RS0: std_logic is PSW(3);
70          -- Register Bank selector bit 0
71
72          alias OV:  std_logic is PSW(2);     -- Overflow Flag
73          alias UD:  std_logic is PSW(1);     -- User definable flag
74          alias P:   std_logic is PSW(0);
75          -- Parity Flag, Parity of Accumulator
76
77          alias Port0_Ram_Reg: std_logic_vector(7 downto 0) is
78          IntMem(128);
79
80          alias Port1_Ram_Reg: std_logic_vector(7 downto 0) is
81          IntMem(144);
82
83          alias Port2_Ram_Reg: std_logic_vector(7 downto 0) is
84          IntMem(160);
85
86          alias Port3_Ram_Reg: std_logic_vector(7 downto 0) is
87          IntMem(176);
88
89
90          alias R0RS00: std_logic_vector(7 downto 0) is IntMem(08);
91          alias R1RS00: std_logic_vector(7 downto 0) is IntMem(09);
92
93
94
95
96    type state_type is (reset, fetch, addr1, addr2, data,
97    cycle8,cycle9,cycle10,
98
99                                        M1T1_fetch1,
100                                       M1T2_AleDrvHigh1,
101                                       M1T3_PcOut1,
102                                       M1T4_AleDrvLow1,
103                                       M1T5_PsenDrvLow1,
104                                       M1T6_Null1,
105                                       M1T7_fetch2,
106                                       M1T8_AleDrvHigh2,
107                                       M1T9_PcOut2,
108                                       M1T10_AleDrvLow2,
109                                       M1T11_PsenDrvLow2,
110                                       M1T12_Null2);
```

```
111         signal ST, nST: state_type;
112
113
114         signal IR : std_logic_vector(7 downto 0) := (others=> '0');
115         alias OP:   std_logic_vector(3 downto 0) is IR(7 downto 4);
116         alias mode: std_logic_vector(3 downto 0) is IR(3 downto 0);
117
118         signal RAR: std_logic_vector(7 downto 0) := (others =>
119         'Z'); -- Ram Address Register
120
121         signal RARTemp: std_logic_vector(7 downto 0) := (others =>
122         'Z'); -- Ram Address Register
123
124         signal PC, PAR: std_logic_vector(15 downto 0);
125
126         alias PCH : std_logic_vector(7 downto 0) is
127         PC(15 downto 8);
128         -- Program Counter High
129
130         alias PCL : std_logic_vector(7 downto 0) is PC(7 downto 0);
131         -- Program Counter Low
132
133         alias PARH : std_logic_vector(7 downto 0) is
134         PAR(15 downto 8);
135         -- Program Address Register High
136
137         alias PARL : std_logic_vector(7 downto 0) is
138         PAR(7 downto 0);
139         -- Program Address Register Low
140
141         signal PCLtemp : std_logic_vector(7 downto 0);
142         signal PCHtemp : std_logic_vector(7 downto 0);
143
144         signal AluOutput : std_logic_vector(7 downto 0)
145         := "00000000";
146
147         constant zero: std_logic_vector(7 downto 0) := "00000000";
148
149         subtype ot is std_logic_vector(3 downto 0);
150
151         -- upper 4 bits of opcode
152
153               constant ADD: ot:= "0010";
154               constant ADDC: ot:= "0011";
155               constant INC: ot:= "0000";
156               constant DEC: ot:= "0001";
157               constant ANL: ot:= "0101";
158               constant ORL: ot:= "0100";
159               constant XRL: ot:= "0110";
160               constant XCH: ot:= "1100";
161               constant MOVR2A: ot:= "1110";
162               constant MOVA2R: ot:= "1111";
163               constant MOVIMM2R: ot:= "0111";
164               constant MOVR2DIR: ot:= "1000";
165               constant MOVDIR2R: ot:= "1010";
166
```

```
167         -- Register Addressing Operands
168
169               constant  INC_REG: ot:= "0000";
170               constant  DEC_REG: ot:= "0001";
171               constant  ADD_REG: ot:= "0010";
172               constant ADDC_REG: ot:= "0011";
173               constant  ORL_REG: ot:= "0100";
174               constant  ANL_REG: ot:= "0101";
175               constant  XRL_REG: ot:= "0110";
176               constant SUBB_REG: ot:= "1001";
177               constant XCH_REG: ot:= "1100";
178               constant DJNZ_REG: ot:= "1101";
179               constant CJNE_REG: ot:= "1011";
180               constant MOV_REG_A2R: ot:= "1110";
181               constant MOV_REG_R2A: ot:= "1111";
182               constant MOV_REG_IMM: ot:= "0111";
183               constant MOV_DIR_REG: ot:= "1000";
184               constant MOV_REG_DIR: ot:= "1010";
185
186         -- Immediate Addressing Operands
187
188               constant INC_IMM: ot := "0000"; -- Hex Code 04
189               constant DEC_IMM: ot := "0001"; -- Hex Code 12
190               constant ADD_IMM: ot := "0010"; -- Hex Code 24
191               constant MOV_IMM: ot := "0111"; -- Hex Code 74
192
193         -- Direct Addressing Operands;
194
195               constant ADD_DIR: ot := "0010";
196               constant MOV_DIR: ot := "1110";
197               constant MOV_DIR_ACC: ot:= "1111";
198               constant LJMP: ot := "0000";
199               constant SETB: ot := "1101";
200               -- Upper nibble for SETB opcode D3 HEX
201               constant CLRB: ot := "1100";
202         -- Upper nibble for CLR C opcode C3 HEX
203
204         -- lower 4 bits of opcode
205
206         constant IMM: ot := "0100"; -- Immediate Addressing
207
208         constant DIR: ot := "0101"; -- Direct Addressing
209
210         constant MM2: ot := "0010"; -- Mix and Match Set 2
211         constant MM3: ot := "0011"; -- Mix and Match Set 3
212
213         constant IND0: ot := "0110"; -- Indirect Addressing R0
214         constant IND1: ot := "0111"; -- Indirect Addressing R1
215
216         constant REG0: ot := "1000"; -- Register Addressing R0
217         constant REG1: ot := "1001"; -- Register Addressing R1
218         constant REG2: ot := "1010"; -- Register Addressing R2
219         constant REG3: ot := "1011"; -- Register Addressing R3
220         constant REG4: ot := "1100"; -- Register Addressing R4
221         constant REG5: ot := "1101"; -- Register Addressing R5
222         constant REG6: ot := "1110"; -- Register Addressing R6
223         constant REG7: ot := "1111"; -- Register Addressing R7
224
```

```
225         signal Opcode,Tmp1,Tmp2: std_logic_vector(7 downto 0);
226
227
228         type decode_type is array (0 to 15) of
229         bit_vector(15 downto 0);
230
231         signal opd : bit_vector(15 downto 0);
232
233         constant decode: decode_type :=
234         (X"0001", X"0002", X"0004", X"0008",X"0010", X"0020",
235         X"0040", X"0080",
236         X"0100", X"0200", X"0400", X"0800",X"1000", X"2000",
237         X"4000", X"8000");
238
239         --  Data Registers and Control Signals
240
241         -- Data Registers
242
243         signal shiftout, op1, op2: std_logic_vector(7 downto 0):=
244         "00000000";
245
246         signal alu9: std_logic_vector(8 downto 0);
247
248         signal Cin : std_logic;
249
250         signal Read2ndByte: bit := '0';
251         signal Read3rdByte: bit := '0';
252         signal Read4thByte: bit := '0';
253
254         signal SecondByteDest: integer range 0 to 5 := 0;
255               --    0 = Discard
256               --    1 = Tmp1
257               --    2 = PCHtemp
258               --    3 = RAR
259               --    4 = PCLtemp -- for DJNZ code
260               --    5 = Tmp2
261
262         signal ThirdByteDest: integer range 0 to 2 := 0;
263               --    0 = Discard
264               --    1 = Tmp1
265               --    2 = PCLtemp
266
267         signal FourthByteDest: integer range 0 to 3 := 0;
268               --    0 = Discard
269               --    1 = Tmp1
270               --    2 = PCLtemp
271               --    3 = RAR
272         signal MCycleLength: integer range 1 to 4 := 1;
273               -- 1 = 1 Machine Cycle
274               -- 2 = 2 Machine Cycles
275               -- 4 = 4 Machine Cycles
276
277         signal CycNum : integer := 0;
278
279         signal ChangeRAR : std_logic := '0';
280         -- this Rar Change sig is checked in M1T3
281
282
```

```
283         signal ChangeRAr2 :std_logic := '0';
284         -- this Rar Change sig is checked in M2T3
285
286         signal incPCxNO : std_logic := '0';
287         signal incPCx2 :std_logic := '0';
288         signal incPCx3 :std_logic := '0';
289         signal incPCx4 :std_logic := '0';
290         signal PCtemps2PC :std_logic := '0';
291
292         signal M1T1incPCxNO : std_logic := '0';
293         signal M1T1incPCx2 :std_logic := '0';
294         signal M1T1incPCx3 :std_logic := '0';
295         signal M1T1incPCx4 :std_logic := '0';
296         signal M1T1PCtemps2PC :std_logic := '0';
297
298         signal M1T7incPCxNO : std_logic := '0';
299         signal M1T7incPCx2 :std_logic := '0';
300         signal M1T7incPCx3 :std_logic := '0';
301         signal M1T7incPCx4 :std_logic := '0';
302         signal M1T7PCtemps2PC :std_logic := '0';
303
304         signal M2T1incPCxNO : std_logic := '0';
305         signal M2T1incPCx2 :std_logic := '0';
306         signal M2T1incPCx3 :std_logic := '0';
307         signal M2T1incPCx4 :std_logic := '0';
308         signal M2T1PCtemps2PC :std_logic := '0';
309
310         signal M2T7incPCxNO : std_logic := '0';
311         signal M2T7incPCx2 :std_logic := '0';
312         signal M2T7incPCx3 :std_logic := '0';
313         signal M2T7incPCx4 :std_logic := '0';
314         signal M2T7PCtemps2PC :std_logic := '0';
315
316         signal M2T7addPCxRel :std_logic := '0';
317
318         signal AccRamWrite : std_logic := '0';
319         signal RamReadTmp1 : std_logic := '0';
320         signal RamWrite : std_logic := '0';
321         signal RamWrite2 : std_logic := '0';
322         -- This Ram Write signal will be for 2 Cycle
323         -- instructions that need to write to the RAM
324
325         signal Updating_Port1_Driver : std_logic := '0';
326         signal Port1_Driver_NeedsUpdate : std_logic := '0';
327         signal SetCarry : std_logic := '0';
328         signal ClearCarry : std_logic := '0';
329
330         signal CurrentAluOutputNotZero : std_logic := '0';
331         signal UseRelOffset: std_logic := '0';
332
333         constant hi_Z8 : std_logic_vector(7 downto 0) :=
334         (others => 'Z');
335         constant hi_Z16 : std_logic_vector(15 downto 0) :=
336         (others => 'Z');
337
338
339
340
```

```
341   ------------------------------------------------------
342   -- Begin Finite State Machine, i.e. CPU_Cycle Process
343   ------------------------------------------------------
344
345   CPU_Cycles: process
346
347         variable reg_mem : Boolean;
348
349         variable CurrentInstLength: integer := 0;
350
351   begin
352
353   wait until XTAL2 = '0' and XTAL2'event;
354
355
356   if ((mode = IMM) or (mode = DIR) or (mode = REG0) or (mode =
357   REG1) or (mode = IND0)
358         or (mode = IND1))  then
359
360         reg_mem:= true;
361
362   else
363
364         reg_mem:= false;
365
366   end if;
367
368
369   if(RST = '1' ) then ST <= reset;
370
371   else
372
373   case ST is
374
375   -----------
376   -- RESET --
377   -----------
378
379         when reset =>
380
381               ST <= cycle8;
382
383   ------------------------------
384   --
385   ----  Machine Cycle 1
386   --
387   ------------------------------
388
389         when M1T1_fetch1 =>  -- Machine Cycle 1, T Cycle 1
390
391               if RamWrite = '1' then
392                     report "Writing to Ram";
393                     IntMem(CONV_INTEGER(RAR)) <= AluOutput;
394               end if;
395
396
397               if Port1_Driver_NeedsUpdate = '1' then
398
```

```
399                     P1 <= Port1_Ram_Reg;
400               end if;
401
402   -----------------
403   --    Special increments for the PC
404   -----------------
405
406               if M1T1incPCx2 = '1' then
407
408                     PC <= PC + 2;     -- Increment PC by 2
409                     report "Incrementing the PC by 2";
410
411               elsif M1T1incPCx3 = '1' then
412
413                     PC <= PC + 3;     -- Increment PC by 3
414                     report "Incrementing the PC by 3";
415
416               elsif M1T1incPCx4 = '1' then
417
418                     PC <= PC + 4;     -- Increment PC by 4
419                     report "Incrementing the PC by 4";
420
421               elsif M1T1PCtemps2PC = '1' then
422
423
424                     PCH <= PCHtemp;
425                     PCL <= PCLtemp;
426                     report "Setting PC to Coded Value";
427
428               else
429
430                     PC <= PC + 1;     -- Increment PC by 1
431                     report "incremening PC by 1";
432               end if;
433
434   ----------------
435   --End Special PC Incrementing
436   ----------------
437
438               IR <= P0;   -- Send Instruction to Instruction
439   Register
440
441               ST <= M1T2_AleDrvHigh1;
442
443         when M1T2_AleDrvHigh1 =>  -- Machine Cycle 1, T Cycle 2
444
445               report "Current Stat is M1T2 / ";
446
447               if Updating_Port1_Driver <= '1' then
448                     Updating_Port1_Driver <= '0';
449               end if;
450
451               PAR <= PC;
452               -- Move PC Value to Program Address Register
453
454               ALE   <= '1';
455               -- Set Address Latch Enable High
456
```

```
457               PSENn <= '1';
458               -- Set Program Store Enable High
459
460               ST <= M1T3_PcOut1;
461
462
463         when M1T3_PcOut1 =>   -- Machine Cycle 1, T Cycle 3
464
465               if ChangeRAR = '1' then
466                     RAR <= RARTemp;
467               end if;
468
469               P0 <= PARL;
470               -- Move the Lower byte of PAR to Port 0
471
472               P2 <= PARH;
473               -- Move the Higher byte of PAR to Port 2
474
475               ST <= M1T4_AleDrvLow1;
476
477         when  M1T4_AleDrvLow1 =>
478               -- Machine Cycle 1, T Cycle 4
479
480               ALE <= '0';
481               -- Set Address Latch Enable Low
482
483               ST <= M1T5_PsenDrvLow1;
484
485
486         when M1T5_PsenDrvLow1 =>  -- Machine Cycle 1, T Cycle 5
487
488               P0 <= hi_Z8;      -- Drive Port 0 to Z state
489               PSENn <= '0';
490
491               ST <= M1T6_Null1;
492
493         when M1T6_Null1 =>      -- Machine Cycle 1, T Cycle 6
494
495               if SetCarry = '1' then
496                     CY <= '1';
497               end if;
498
499               if ClearCarry = '1' then
500                     CY <= '0';
501               end if;
502
503               ST <= M1T7_fetch2;
504
505         when M1T7_fetch2 =>  -- Machine Cycle 1, T Cycle 7
506
507               If SecondByteDest = 0 then
508
509               report"Discard Current Byte";
510
511               elsif SecondByteDest = 1 then
512
513                     Tmp1 <= P0;
514
```

```
515                   report " Second Byte Destination is Tmp1";
516
517
518             elsif SecondByteDest = 2 then
519
520                   PCHtemp <= P0;
521
522                   report "Second Byte Destination is PCH temp";
523
524
525             elsif SecondByteDest = 3 then
526
527                   RAR <= P0;
528
529                   report "Second Byte Destination is RAR";
530
531
532             elsif SecondByteDest = 4 then
533
534                   report "Second Byte Destination is PCL temp";
535
536                   PCLtemp <= P0;
537
538             elsif SecondByteDest = 5 then
539
540                   report "Secondbyte Destination is Tmp2 ";
541
542                   Tmp2 <= P0;
543
544             end if;
545
546       -----------------
547       --    Special increments for the PC
548       -----------------
549
550             if M1T7incPCxNO = '1' then
551                   report "Not incrementing PC";
552
553             elsif M1T7incPCx2 = '1' then
554
555                   PC <= PC + 2;     -- Increment PC by 2
556                   report "Incrementing the PC by 2";
557
558             elsif M1T7incPCx3 = '1' then
559
560                   PC <= PC + 3;     -- Increment PC by 3
561                   report "Incrementing the PC by 3";
562
563             elsif M1T7incPCx4 = '1' then
564
565                   PC <= PC + 4;     -- Increment PC by 4
566                   report "Incrementing the PC by 4";
567
568             elsif M1T7PCtemps2PC = '1' then
569
570
571                   PCH <= PCHtemp;
572                   PCL <= PCLtemp;
```

```
573                     report "Setting PC to Coded Value";
574
575               else
576
577                     PC <= PC + 1;     -- Increment PC by 1
578                     report "Incrementing PC by 1";
579
580               end if;
581
582   ------------------
583   ----End Special PC Incrementing
584   ------------------
585
586               ST <= M1T8_AleDrvHigh2;
587
588         when M1T8_AleDrvHigh2 =>  -- Machine Cycle 1, T Cycle 8
589
590
591               PAR   <= PC;
592               -- Move Program Counter to Program Address Register
593
594               ALE   <= '1';
595               -- Set Address Latch Enable High
596
597               PSENn <= '1';
598
599               ST <= M1T9_PcOut2;
600
601         when M1T9_PcOut2 =>     -- Machine Cycle 1, T Cycle 9
602
603               P0 <= PARL;
604               -- Move the Lower byte of PAR to Port 0
605
606               P2 <= PARH;
607
608               ST <= M1T10_AleDrvLow2;
609
610         when M1T10_AleDrvLow2 =>  -- Machine Cycle 1, T Cycle 10
611
612               ALE <= '0';
613               ST <= M1T11_PsenDrvLow2;
614
615         when M1T11_PsenDrvLow2 =>  -- Machine Cycle 1, T Cycle 11
616
617               P0 <= hi_Z8;      -- Drive Port 0 to Z state
618               ST <= M1T12_Null2;
619               PSENn <= '0';
620
621               if RamReadTmp1 = '1' then
622                     report "Attempting RamReadTmp1";
623                     Tmp1 <= IntMem(CONV_INTEGER(RAR));
624               end if;
625
626               if AccRamWrite = '1' then
627                     report "Attemping AccRamWrite";
628                     IntMem(CONV_INTEGER(RAR)) <= Acc;
629               end if;
630
```

```
631         when M1T12_Null2 =>  -- Machine Cycle 1, T Cycle 12
632
633               if reg_mem then
634
635               report "Attemping ALU_OP Call";
636
637               ALU_OP(Tmp1,Tmp2,Acc,AluOutput,CY,UseRelOffset);
638
639               end if;
640
641               IR <= "00000000";  -- Reset IR
642
643               if MCycleLength = 2 then
644               -- if the current opcode is a 2 cycle
645               -- instruction then go into M2
646
647                     ST <= M2T1_fetch1;
648                     report" transitioning to M2T1_fetch";
649               else
650
651                     ST <= M1T1_fetch1;
652                     report "transitioning to M1T1_fetch";
653               end if;
654
655   -------------------------------------------
656   -- End Machine Cycle 1
657   -------------------------------------------
658
```

Appendix E - Decoder Process VHDL Example Code

```
1   ------------------------------------------------------
2   --
3   -- Opcode Decode Process
4   --
5   ------------------------------------------------------
6
7   OpcodeDecode: process(ST)
8
9   begin
10
11  case ST is
12        when M1T1_fetch1 =>
13
14              MCycleLength <= 1; -- Reset the Current Machine Cycle
15              AccRamWrite <= '0'; -- Reset AccRamWrite flag
16              incPCxNO <= '0';
17
18        when M1T3_PcOut1 =>
19
20              M1T1incPCxNO    <= '0';
21              M1T1incPCx2     <= '0';
22              M1T1incPCx3     <= '0';
23              M1T1incPCx4     <= '0';
24              M1T1PCtemps2PC  <= '0';
25
26        when M1T5_PsenDrvLow1 =>  -- Machine Cycle 1, T Cycle 5
27
28              ChangeRAR <= '0';
29
30        when M1T9_Pcout2 =>
31
32              M1T7incPCxNO    <= '0';
33              M1T7incPCx2     <= '0';
34              M1T7incPCx3     <= '0';
35              M1T7incPCx4     <= '0';
36              M1T7PCtemps2PC  <= '0';
37
38        when M1T12_Null2 =>
39
40              SecondByteDest <= 0;
41              Read2ndByte <= '0';
42              RamReadTmp1 <= '0';
43
44              -- Reset Ram Read  to Temp 1 register
45
46              SetCarry <= '0';
47              -- Reset Set the Carry Bit Signal
48
49              ClearCarry <= '0';
50              -- Clear Clear the Carry Bit Signal
51
52        when M2T2_AleDrvHigh1 =>
53
```

```
54                SecondByteDest <= 0;
55                incPCxNO <= '0';
56
57          when M2T3_PcOut1 =>
58
59                M2T1incPCxNO    <= '0';
60                M2T1incPCx2     <= '0';
61                M2T1incPCx3     <= '0';
62                M2T1incPCx4     <= '0';
63                M2T1PCtemps2PC  <= '0';
64
65          when M2T5_PsenDrvLow1 =>
66
67                ChangeRAR2 <= '0';
68
69          when M2T9_PcOut2 =>
70
71                M2T7incPCxNO    <= '0';
72                M2T7incPCx2     <= '0';
73                M2T7incPCx3     <= '0';
74                M2T7incPCx4     <= '0';
75                M2T7PCtemps2PC  <= '0';
76                M2T7addPCxRel   <= '0';
77
78          when M2T12_Null2 =>
79
80                PCtemps2PC <= '0';
81                -- Turns off the Special PC Incrementation
82
83                ThirdByteDest <= 0;
84                FourthByteDest <= 0;
85                RamWrite <= '0'
86                RamWrite2 <= '0';
87
88    ----------------------------
89    -- Decoding Instructions ---
90    ----------------------------
91
92          when M1T2_AleDrvHigh1 =>
93
94                case MODE is
95
96                      when IMM =>
97
98                            case OP is
99
100                               when ADD_IMM | MOV_IMM =>
101
102                                     report "/\***/\ Decoding an
103                                     Immediate ADD or MOV /\***/\
104                                     ";
105
106                                     -- Instruction type, 2 Byte
107                                     Read2ndByte <= '1';
108
109                                     -- Read Next Byte
110
111                                     SecondByteDest <= 1;
```

```
112
113                                     -- Move Data Byte 2
114                                     --Tmp1
115
116                                     MCycleLength <= 1;
117
118                               when others => null;
119
120                         end case; -- End Case OP
121
122                   when MM2=>
123
124                         case OP is
125
126                               when LJMP =>
127
128                                     report "/\***/\ Processing
129                                     Long Jump Instruction /\***/\
130                                     ";
131
132                                     Read2ndByte <= '1';
133                                     -- 1 is True
134
135                                     Read3rdByte <= '1';
136                                     Read4thByte <= '0';
137
138                                     SecondByteDest <= 2;
139                                     -- 2 = PCHtemp
140
141                                     ThirdByteDest <= 2;
142                                     -- 2 = PCLtemp
143
144                                     M2T7PCtemps2PC  <= '1' ;
145                                     -- This Causes the PC to be
146                                     -- set to a coded value
147
148                                     M2T1incPCxNO <= '1';
149
150                                     MCycleLength <= 2;
151
152
153                         when others => null;
154
155                         end case;
156         -- End case OP--
157
158                   when MM3 =>
159
160                         case OP is
161
162                               when SETB =>
163
164                                     report "/\***/\ Processing
165                                     Set Carry Instruction /\***/\
166                                     ";
167
168                                     Read2ndByte <= '0';
169                                     -- 1 is True
```

```
170
171                                     SecondByteDest <= 0;
172                                     -- 2 = PCHtemp
173
174                                     MCycleLength <= 1;
175
176                                     SetCarry <= '1';
177
178                                     M1T7incPCxNO <= '1';
179
180                               when CLRB =>
181
182
183                                     report "/\***/\ Processing
184                                     Clear Carry Instruction
185                                     /\***/\ ";
186
187                                     Read2ndByte <= '0';
188                                     -- 1 is True
189
190                                     SecondByteDest <= 0;
191                                     -- 2 = PCHtemp
192
193                                     MCycleLength <= 1;
194
195                                     ClearCarry <= '1';
196
197                                     M1T7incPCxNO <= '1';
198
199                               when others => null;
200
201                         end case;
202
203                   when REG0 =>  -- REGISTER 0  <----------
204
205                         case OP is
206
207                               when ADD_REG  =>
208
209                                     report"/\***/\ Decoding
210                                     Register 0 Addition /\***/\
211                                     ";
212                                           Read2ndByte <= '0';
213                                     -- Don't Read Next Byte
214
215                                     SecondByteDest <= 0;
216                                     -- Discard Byte
217
218                                     M1T7incPCxNO <= '1';
219
220                                     RARTemp <= "000"& RS1 & RS0 &
221                                     IR(2 downto 0);
222
223                                     ChangeRAR <= '1';
224                                     RamReadTmp1 <= '1';
225                                     MCycleLength <= 1;
226
227                   when others => null;
```

```
228
229                 end case;
230
231     when others => null;
232
233     end case;
234
235     end process OpcodeDecode;
236
237     -------------------------------------------------------
238     --
239     -- End Opcode Decode Process
240     --
241     -----------------------------------------------------
```

Appendix F - ALU_OP Procedure Example Code

```
1   -----------------
2   --
3   -- ALU_OP Procedure
4   --
5   -----------------
6
7   procedure ALU_OP(Tmp1 : in std_logic_vector(7 downto 0);
8                    Tmp2 : in std_logic_vector(7 downto 0);
9                    signal A: inout std_logic_vector(7 downto 0);
10                   signal RegResult: inout std_logic_vector(7
11  downto 0);
12                   signal CY: inout std_logic;
13                   signal UseRelOffset: out std_logic) is
14
15                               -- A is Accumulator
16                               -- RegResult is ALU Output reg
17                               -- Cy is Carry bit
18
19  variable res: std_logic_vector(8 downto 0); -- result of ALU
20  operation
21  variable temp: std_logic_vector(8 downto 0);
22  variable updateCY : Boolean := True;  -- update CY flag by
23  default
24
25  begin
26        UseRelOffset <= '0';
27
28        case OP is
29
30        -- Covers Immediate Add and Register ADD
31              when ADD => res := ('0'&A) + ('0'&Tmp1);
32                                 A <= res(7 downto 0);   --
33              when others => updateCY := FALSE;
34        end case;
35
36  end ALU_O
```

1