A VERILOG 8051 SOFT CORE FOR FPGA APPLICATIONS

Sakina Rangoonwala

Thesis Prepared for the Degree of

MASTER OF SCIENCE

UNIVERSITY OF NORTH TEXAS

August 2009

APPROVED:

Elias Kougianos, Major Professor
Saraju P. Mohanty, Co-Major Professor
Robert G. Hayes, Committee Member
Dave Clark, Exe Consulting, Industrial Representative
Nourredine Boubekri, Chair of the Department of Engineering Technology
Costas Tsatsoulis, Dean of the College of Engineering
Michael Monticino, Dean of the Robert B. Toulouse School of Graduate Studies

Rangoonwala, Sakina. A Verilog 8051 Soft Core for FPGA Applications. Master of Science (Engineering Systems), August 2009, 78 pp., 7 tables, 42 figures, references, 23 titles.

The objective of this thesis was to develop an 8051 microcontroller soft core in the Verilog hardware description language (HDL). Each functional unit of the 8051 microcontroller was developed as a separate module, and tested for functionality using the open-source VHDL Dalton model as benchmark. These modules were then integrated to operate as concurrent processes in the 8051 soft core. The Verilog 8051 soft core was then synthesized in Quartus® II simulation and synthesis environment (Altera Corp., San Jose, CA, www.altera.com) and yielded the expected behavioral response to test programs written in 8051 assembler residing in the v8051 ROM. The design can operate at speeds up to 41 MHz and used only 16% of the FPGA fabric, thus allowing complex systems to be designed on a single chip. Further research and development can be performed on v8051 to enhance performance and functionality.

Copyright 2009

by

Sakina Rangoonwala

TABLE OF CONTENTS

LIST OF TABLES v

LIST OF FIGURES vi

LIST OF ABBREVIATIONS ix

CHAPTER 1. INTRODUCTION... 1

CHAPTER 2. THE 8051 IP SOFT CORE APPLICATIONS.................................. 5

CHAPTER 3. 8051 VERILOG SOFT CORE 7

3.1. Features of the 8051 Soft Core 7

3.2. Overview 8

CHAPTER 4. COMPONENTS OF THE 8051 SOFT CORE 11

4.1. Controller.. 11

4.1.1. Detailed Design.. 12

4.1.2. Simulation and Testing.. 27

4.1.3. Verification 29

4.2. ROM.. 30

4.2.1. Detailed Design.. 30

4.2.2. Simulation & Testing 31

4.2.3. Verification 32

4.3. RAM 33

4.3.1. Detailed Design 33

4.3.2. Simulation and Testing 38

4.3.3. Verification 43

4.4. External RAM 44

4.5. Decoder 44

4.5.1. Detailed Design 44

4.5.2. Simulation & Testing 45

4.5.3. Verification 46

4.6. ALU 47

4.6.1. Detailed Design 47

4.6.2. Simulation and Testing 55

4.6.3. Verification 57

CHAPTER 5. 8051 MODEL INTEGRATION 59

5.1. Detailed Design 59

5.2. Simulation and Testing 60

CHAPTER 6. CONCLUSIONS 65

APPENDIX 67

ASSEMBLER LISTING OF TEST PROGRAM T_BCD_R2 68

REFERENCES 75

LIST OF TABLES

Table 4.1-1: Encoding Table CPU_STATES 12

Table 4.1-2: Encoding Table EXE_STATES 13

Table 4.5-1: Decoder Test - Instructions & Expected Results 45

Table 4.6-1: ALU Operations 48

Table 4.6-2: ALU - Test Instructions and Expected Results 56

Table 4.6-3: Mismatched Output Signals (Dalton & Verilog ALU Modules) and Their Impact on Result 57

Table 5.2-1: Expected Result for Test Program t_bcd_r2.................................. 63

LIST OF FIGURES

Figure 3.2-1: Block Diagram of the v8051 Soft Core 9

Figure 4.1-1: Controller Module 11

Figure 4.1-2: State Diagram CPU_STATES 12

Figure 4.1-3: State Diagram EXE_STATES 13

Figure 4.1-4: Overview of Controller Program Flow... 15

Figure 4.1-5: Flow Diagram for Controller Reset Cycle CS_0 16

Figure 4.1-6 : Flow Diagram for Instruction Fetch Cycle CS_2 (ES_0 to ES_2). 19

Figure 4.1-7: Flow Diagram for Instruction Fetch Cycle CS_2 (ES_3 to ES_4).. 20

Figure 4.1-8: Flow Diagram for Instruction Fetch Cycle CS_2 (ES_5 to ES_7).. 21

Figure 4.1-9: Flow Diagram for Instruction Execute Cycle CS_3 for Arithmetic ADD A, #data 25

Figure 4.1-10: Flow Diagram of Instruction Execution Cycle CS_3 for Shift Left RLC A 26

Figure 4.1-11: Instruction Fetch & Decode During CS_2.................................... 28

Figure 4.1-12: Execute Cycle CS_3 for Instruction RLC..................................... 28

Figure 4.2-1: ROM Contents of Test Program Test1_r1.txt - 8051 Instructions.. 32

Figure 4.2-2: Simulation Waveform for ROM Module (Verilog Model)................ 32

Figure 4.2-3: Simulation Waveform for ROM Module (Dalton Model)................. 33

Figure 4.3-1: Flow Diagram for RAM Module 35
Figure 4.3-2: Flow Diagram for RAM - Bit Manipulation 36
Figure 4.3-3: Flow Diagram for RAM - Byte Read /Write 37
Figure 4.3-4: Reset Asserted 38
Figure 4.3-5: Byte Read / Write to RAM - Functional Simulation 40
Figure 4.3-6: Byte Read / Write to RAM - Timing Simulation 40
Figure 4.3-7: Worst-Case Delays in Timing Simulation 40
Figure 4.3-8: Byte Read / Write to RAM / SFRs 41
Figure 4.3-9: Bit Manipulation on RAM Locations 20H to 2FH 42
Figure 4.3-10: Bit Read from SFRs – A-Register, B-Register, Port1, & PSW 43
Figure 4.5-1: Simulation of 8051 Instruction Decoding (Verilog Model) 46
Figure 4.5-2: Worst–Case Delay in Timing Simulation of Decoder Functions 46
Figure 4.5-3: Comparison Report of Simulation of 8051 Instruction Decoding (Dalton Model with Verilog model) 47
Figure 4.6-1: Flow Diagram for ALU Response to ADD or SUB Instruction 49
Figure 4.6-2: Flow Diagram for ALU Response to MUL or DIV Instruction 50
Figure 4.6-3: Flow Diagram for ALU response to DIV instruction (cont'd) 51
Figure 4.6-4: Flow Diagram for ALU response to DA instruction 52
Figure 4.6-5: Flow Diagram for ALU Response to Logical Instructions 53
Figure 4.6-6: Flow Diagram for ALU Response to Shift Instructions 54
Figure 4.6-7: Simulation of 8051 ALU Operation for Op-Codes 0 to 9h 55

Figure 4.6-8: Simulation of 8051 ALU Operation for Op-Codes Ah to Fh 56
Figure 4.6-9: Comparison Report of Simulation of 8051 ALU Operation (Dalton Model with Verilog Model) for Op-Codes 0 to 9h........................... 58
Figure 4.6-10: Comparison Report of Simulation of 8051 ALU Operation (Dalton Model with Verilog Model) for Op-Codes Ah to Fh 58
Figure 5.2-1: 'Test_Led' Test Program in 8051 Assembly Code. 60
Figure 5.2-2: Waveform Showing Input/ Output Ports with 8051 Soft Core Running 'Test_Led' 61
Figure 5.2-3: Simulation Results for Soft Core Running Test Program t_bcd_r2 64

LIST OF ABBREVIATIONS

ac	Auxiliary carry flag
cy	Carry flag
DPTR	Data pointer register
IP	Intellectual property
FPGA	Field programmable gate array
FSM	Finite state machine
HDL	Hardware description language
ov	Overload flag
PC	Program counter register
PSW	Program status word register
RAM	Random access memory
ROM	Read only memory
SFR	Special function register
SP	Stack pointer register

CHAPTER 1.

INTRODUCTION

An IP (intellectual property) core is a block of logic or data that is used in configuring a field programmable gate array (FPGA) or application-specific integrated circuit (ASIC) towards a final product [1]. The growing progression on the use of IP cores in the electronic design automation (EDA) industry can be attributed to repeated use of previously designed components. Design reuse shortens time to develop and hence the time-to-market for a new product. An IP core should be entirely portable for it to be used as a building block with multiple vendor technologies or design methodologies.

There are three types of IP cores: hard cores, firm cores, or soft cores [1]. Hard cores are physical implementations of the IP design logic and interconnections. These are best for plug-and-play applications, and are less portable and flexible than the other two types of cores. Like the hard cores, firm (sometimes called semi-hard) cores also hard wired but have some flexibility and are configurable to various applications. The most flexible of the three, soft cores exist either as a netlist (a list of the logic gates and associated interconnections making up an integrated circuit) or hardware description language (HDL) code. The hard core must be designed using a particular technology, while a soft core has the advantage of being synthesizable in any technology as long as the

specific design tools and technology libraries are available. Furthermore, the designer can optimize the soft core for a particular use, for example by removing unused functions, thus reducing size and power [1].

Embedded controllers are special purpose computers programmed to perform one or a few dedicated functions. The common features that characterize embedded controllers are [2]:

- Programmed to perform dedicated task. The specific program is stored in ROM (read-only memory) and therefore does not change.
- Low-power devices, operating at a fraction of the power of a desktop. A desktop computer is plugged into a wall socket and might consume 500 watts of electricity while a battery-operated microcontroller might consume only 50 milli-watts.
- An embedded controller has a dedicated input device and may have a small light emitting diode (LED) or liquid crystal display (LCD) for output. A microcontroller also takes input from the device it is controlling and controls the device by sending signals to different components in the device.
- Small and low cost. The components are chosen to minimize size and to be as inexpensive as possible.
- It is made rugged in some way, to cater for the application.

- Highly reliable. Embedded systems often reside in machines that are expected to run continuously for years without error and in some cases automatically recover, if an error occurs.

The portable feature of IP cores makes it easier to import the technology to a new system, and to build a new product pivoting on the advantages of intellectual property. Another important advantage of using IP cores is the reduction of engineering costs for the new systems. Design reuse, ease of reconfiguration and customization are the attributes that make use of IP cores an attractive methodology to build systems on a chip [3].

Every modern device has embedded controllers. The range of applications of embedded systems vary from simple portable devices such as mobile phones and MP3 players, to large stationary installations like traffic lights, industrial process controllers, or the systems controlling satellites. Complexity varies from low, with a single microcontroller chip, to very high with multiple units, peripherals and networks mounted inside a large chassis or enclosure.

Embedded systems span all aspects of modern life. Some examples [2] of their use are listed below:

Embedded controllers are extensively used in Telecommunications systems, from telephone switches for the network to mobile phones at the end-user. Dedicated routers and network bridges are used in Computer networking to route data. Examples in consumer electronics are MP3 players, personal digital assistants (PDAs), digital cameras, mobile phones, videogame consoles, DVD

players, GPS receivers, printers and scanners. Household appliances have embedded controllers to enhance features, flexibility and efficiency in devices such as microwave ovens, washing machines and dishwashers. Advanced HVAC systems provide more accurate and efficient temperature control, programmable to change by time of day and season, using networked thermostats. Home automation uses wired- and wireless-networking that can be used to control lights, climate, security, and audio/visual output signals, all of which use embedded devices for sensing and controlling.

Transportation systems: Airplanes use advanced avionics such as inertial guidance systems and GPS receivers with embedded controllers also functioning to meet the safety requirements. Various electric motors — brushless DC motors, induction motors and DC motors — are using electric/electronic motor controllers.

Automobiles, electric vehicles and hybrid vehicles are increasingly using embedded systems to maximize efficiency and reduce pollution. Other automotive safety systems such as anti-lock braking system (ABS), electronic stability controls (ESC/ESP), and automatic four-wheel drive use large numbers of embedded microcontrollers.

Medical equipment is progressively becoming sophisticated with more embedded systems for vital signs monitoring, electronic thermometers, stethoscopes, sound amplifiers, and various visualization techniques (PET, SPECT, CT, MRI, MPR) for non-invasive internal inspections.

CHAPTER 2.

THE 8051 IP SOFT CORE APPLICATIONS

The 8051 microcontroller core is suitable for building mixed-signal systems on a chip. It also provides an easily-programmed alternative to hard-coded control logic in many existing applications. It is suited for low-power and small FPGA-based systems.

The soft core on FPGA for system–on–chip provides a platform for changes to embedded design. Additional functions can be added, the chip can be upgraded or simple modifications implemented by reprogramming. Such examples are seen in the following research applications:

1. Auto peripheral detection, hardware dynamic partial self-reconfiguration and software dynamic driver loading implemented on one single FPGA [4].
2. An embedded implementation of the "Player" client mobile robot using the NIOS® II soft core (Alter Corp., San Jose, CA, www.altera.com) [5].
3. The integration of 8-bit 8051 processor core with a general purpose 16-bit fixed-point digital signal processor (DSP) core using the Virtex™-II FPGA (Xilinx Inc., San Jose, CA, www.xilinx.com) and RAM/ ROM memories to generate a new processor family called the digital signal controller (DSC) [6].

4. Embedded On-Chip Debugging support module integrated onto 8051 microcontroller for in-circuit emulation of embedded systems with debugging mechanisms such as single step, breakpoint setting and detection, and internal resource monitoring and modification [7].

CHAPTER 3.

8051 VERILOG SOFT CORE

3.1. Features of the 8051 Soft Core

The objective is to develop an 8051 soft core, in the Verilog HDL, capable of being used as an embedded controller, and customizable for specific applications.

The key features of the Verilog soft core are as follows:

- Executes the 8051 basic instruction set.
- Addresses up to 256 bytes of random access memory configured as follows to emulate the 8051 RAM.

  128 bytes of data memory (00h – 7Fh) classified as:

    - 00h – 1Fh: 4 banks of 8 registers (R0 to R7) each.
    - locations 20h -2Fh bit addressable registers
    - 30h – 7Fh general purpose registers

  21 defined Special function registers assigned within address space 80h to FFh.

- Supports 4KB of program memory
- Addresses up to 64KB external memory
- Four 8-bit I/O ports with following characteristics:

Each port is bit-addressable for both input and output directions. It can be configured as single bit or 8-bit ports. Thus 32 I/O lines are available for connecting the 8051 to peripheral devices.

- Interrupts are not implemented. Provisions exist for adding interrupt/s handling.
- Data/ address lines are not tri-stated.

The programmable parts of the soft core can be configured to achieve any desired function or application. That is, by writing software in 8051 assembly code, loading it to ROM and setting the parameters.

3.2. Overview

The functional units of a basic 8051 microcontroller, as shown in the Block diagram in

Figure 3.2-1, were each analyzed, implemented and tested individually. The units were then integrated to synthesize the v8051 soft core.

The Verilog 8051 soft core was developed with the Dalton model as the reference soft core. Dalton model is a working, freely available VHDL IP soft core [8].

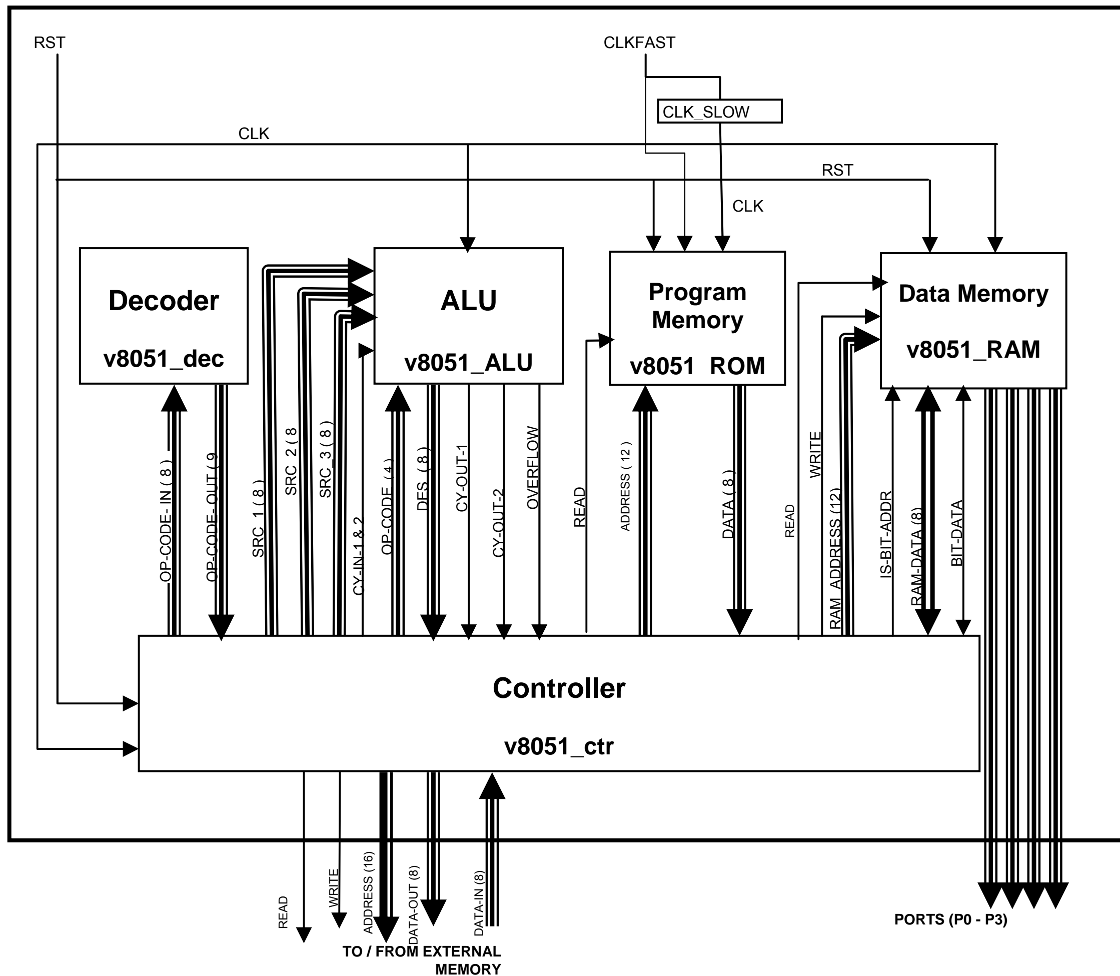


Figure 3.2-1: Block Diagram of the v8051 Soft Core

The Verilog 8051 soft core was developed following the method of ipProcess Workflow [9]. It is represented by five major workflows of IP design, as follows:

Statement of Requirements- These are stated in section 3.1 above.

Analysis and Design- Design overview is shown in Figure 3.2-1. Detailed design flow for each module is explained in the following chapter.

Implementation- Design is implemented in Verilog HDL.

Functional Verification- These are also detailed in chapter 4. The Dalton model was used as benchmark to verify functionality.

FPGA Prototyping- the FPGA on DE2 Board from ALTERA® was then programmed using the tools for prototyping in Quartus® II design software (Altera Corp., San Jose, CA, www.altera.com).

The design, development and testing of the soft core was done using Quartus II version 8.0 Web edition software. The only other tool required was an 8051 simulator, for testing the 8051 assembler code, and an assembler for the Soft core ROM. The freely available EdSim51™ 8051 simulator (James Roger, 8051 simulator for teachers and students, IT Sligo, Ireland, www.edsim52.com) was used [10].

CHAPTER 4.

COMPONENTS OF THE 8051 SOFT CORE

4.1. Controller

The controller, as the name implies, controls the sequence of all activities in the 8051. It steers the data to the proper destination, according to the instruction being executed. It also monitors the stage of the instruction processing and determines the value of the control signals. The controller module manages the data path for the 8051 instruction set, which consists of 49 one-byte long, 45 two-byte long and 17 three-byte long instructions [12]. Figure 4.1-1 shows the input and output signals for the controller.

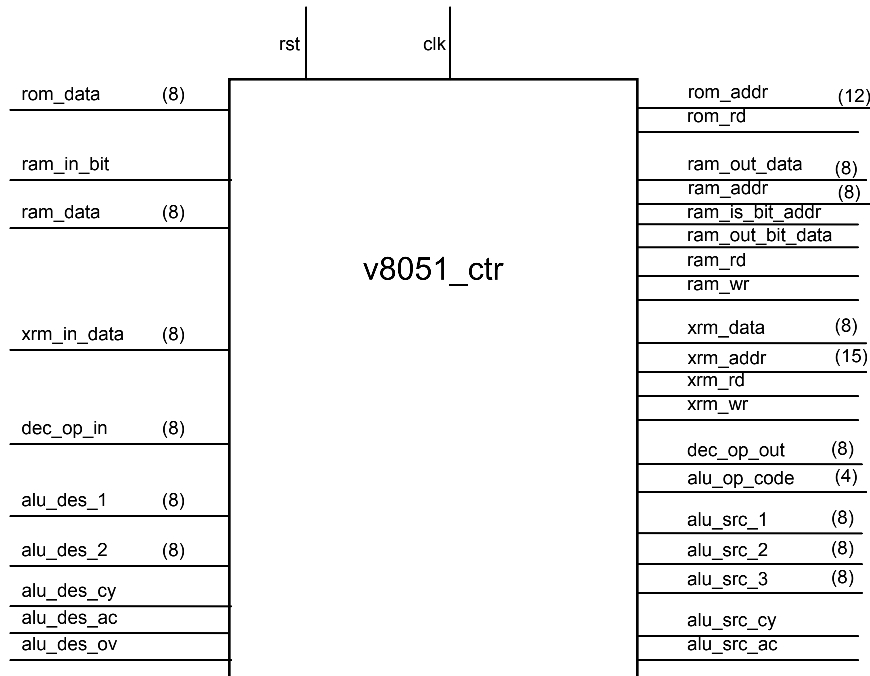


Figure 4.1-1: Controller Module

4.1.1. Detailed Design

The controller module is implemented as a FSM [13]. The functions of the v8051 controller are implemented in four control phases, defined as "CPU_STATES". The functions of the controller and corresponding CPU_STATES are: Reset (CS_0), instruction fetch & decode (CS_2), and instruction execute (CS_3). CS_1 is reserved for future development of interrupt handling. Figure 4.1-2 and Figure 4.1-3 are the graphical, and Table 4.1-1 and Table 4.1-2 the tabular representations of the FSM for control states generated by the Quartus® II design software (Altera Corp., San Jose, CA, www.altera.com) upon compiling the controller module.

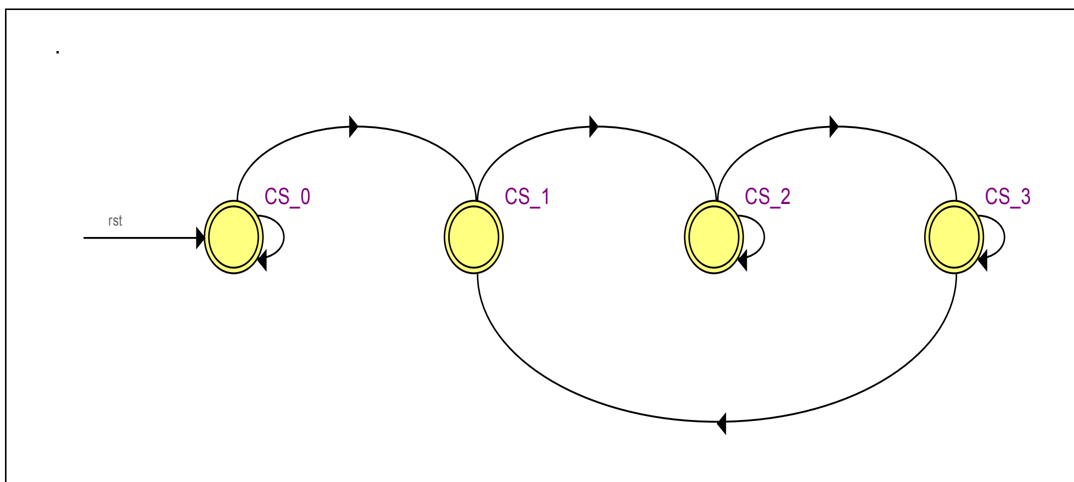


Figure 4.1-2: State Diagram CPU_STATES

Table 4.1-1: Encoding Table CPU_STATES

Name	CS_0	CS_3	CS_2	CS_1
CS_0	0	0	0	0
CS_1	1	0	0	1
CS_2	1	0	1	0
CS_3	1	1	0	0

In each CPU_STATE the control of the data path is executed in a sequence of eight EXE_STATES. For timing control each EXE_STATE is synchronized with, and equal to, one internal clock period. Both CPU_STATES and the EXE_STATES are one-hot coded. The state diagram and the encoding table for execution of the control states are represented in Figure 4.1-3 and Table 4.1-2 respectively. These diagrams are software generated upon compiling the controller module.

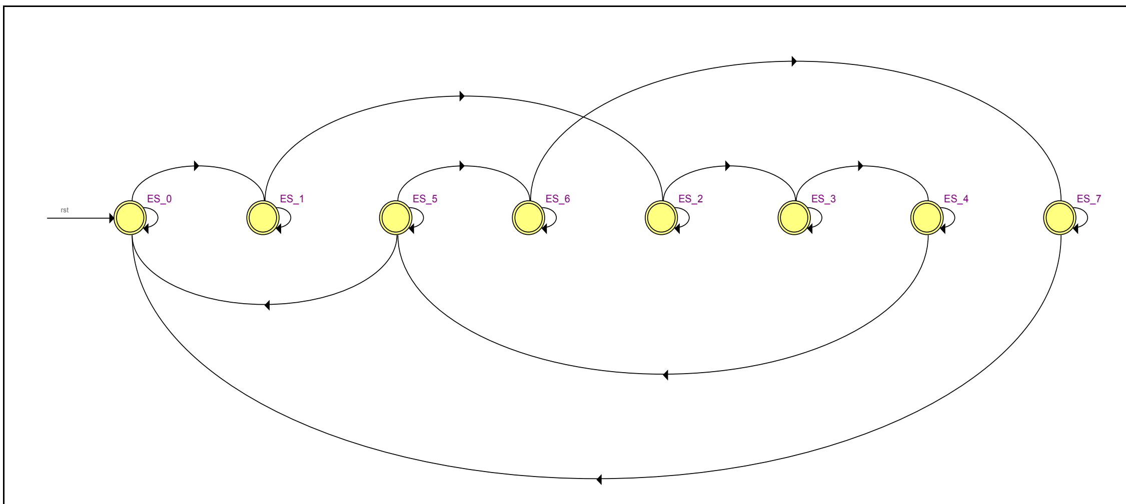


Figure 4.1-3: State Diagram EXE_STATES

Table 4.1-2: Encoding Table EXE_STATES

Names	ES_0	ES_7	ES_6	ES_5	ES_4	ES_3	ES_2	ES_1
ES_0	0	0	0	0	0	0	0	0
ES_1	1	0	0	0	0	0	0	1
ES_2	1	0	0	0	0	0	1	0
ES_3	1	0	0	0	0	1	0	0
ES_4	1	0	0	0	1	0	0	0
ES_5	1	0	0	1	0	0	0	0
ES_6	1	0	1	0	0	0	0	0
ES_7	1	1	0	0	0	0	0	0

The controller maintains uniform timings for its CPU States. CS_0 is executed in 6 clock cycles that is in EXE_STATES ES_0 to ES_5. The CS_2 and CS_3 are each completed in ES_0 to ES_7 for all instructions types.

Figure 4.1-4 is an overview of program flow of the controller module.

The control or CPU states and the steps taken to execute each instruction are as follows:

CPU_STATES:

- CS_0: Controller reset cycle. This state is entered after reset is asserted.

  The controller is initialized in a sequence of the following EXE_STATES:

  ES_0: Initializes output Port_0 to FFh. Changes EXE_STATE to ES_1.

  ES_1: Initializes output Port_1 to FFh. Changes EXE_STATE to ES_2.

  ES_2: Initializes output Port_2 to FFh. Changes EXE_STATE to ES_3.

  ES_3: Initializes output Port_3 to FFh. Changes EXE_STATE to ES_4.

  ES_4: Initializes Stack-pointer register to 07h. Changes EXE_STATE to ES_5.

  ES_5: Changes EXE_STATE to ES_0 and CPU_STATE to CS_1.

  The operation is shown in Figure 4.1-5.

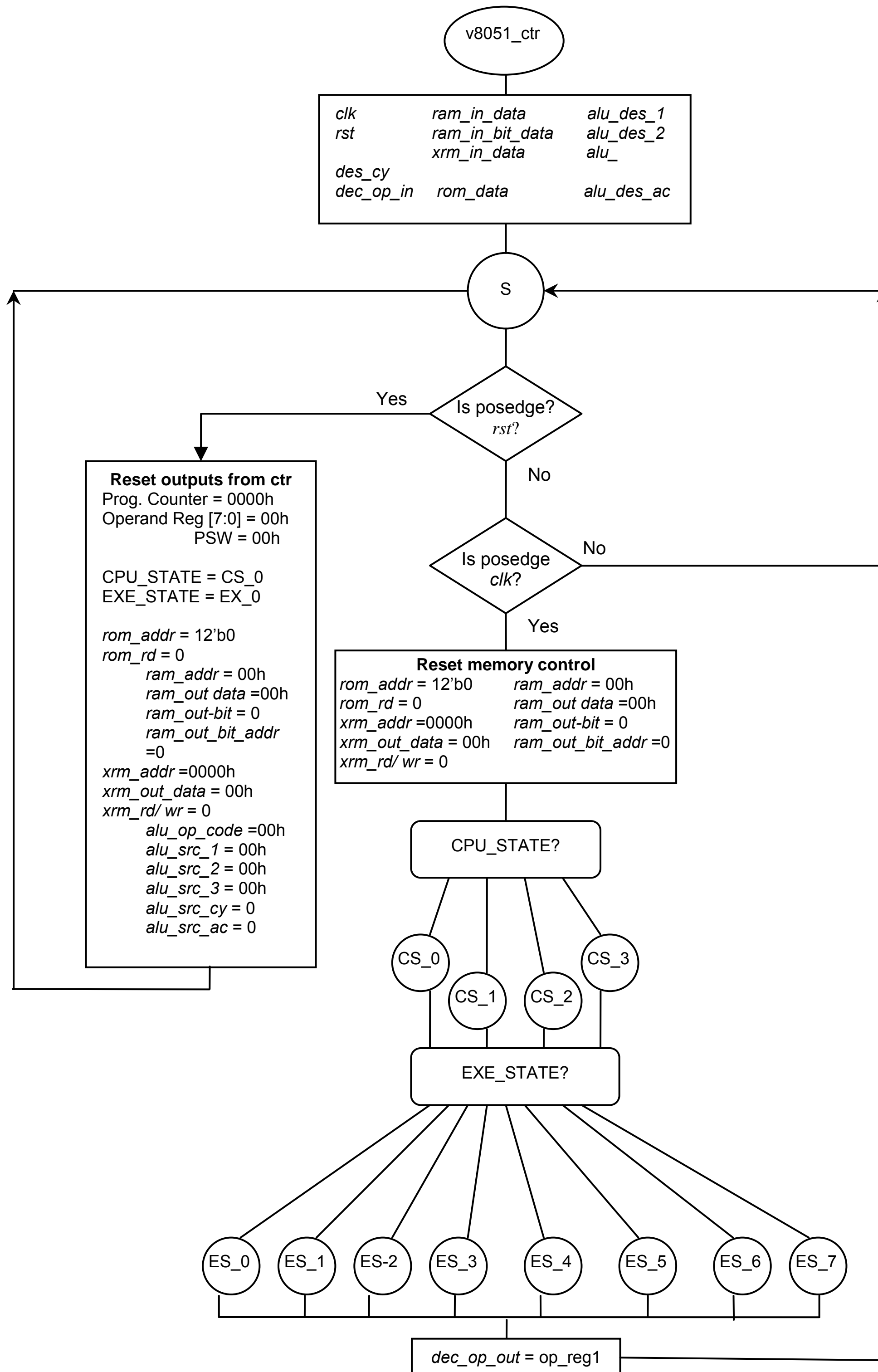


Figure 4.1-4: Overview of Controller Program Flow

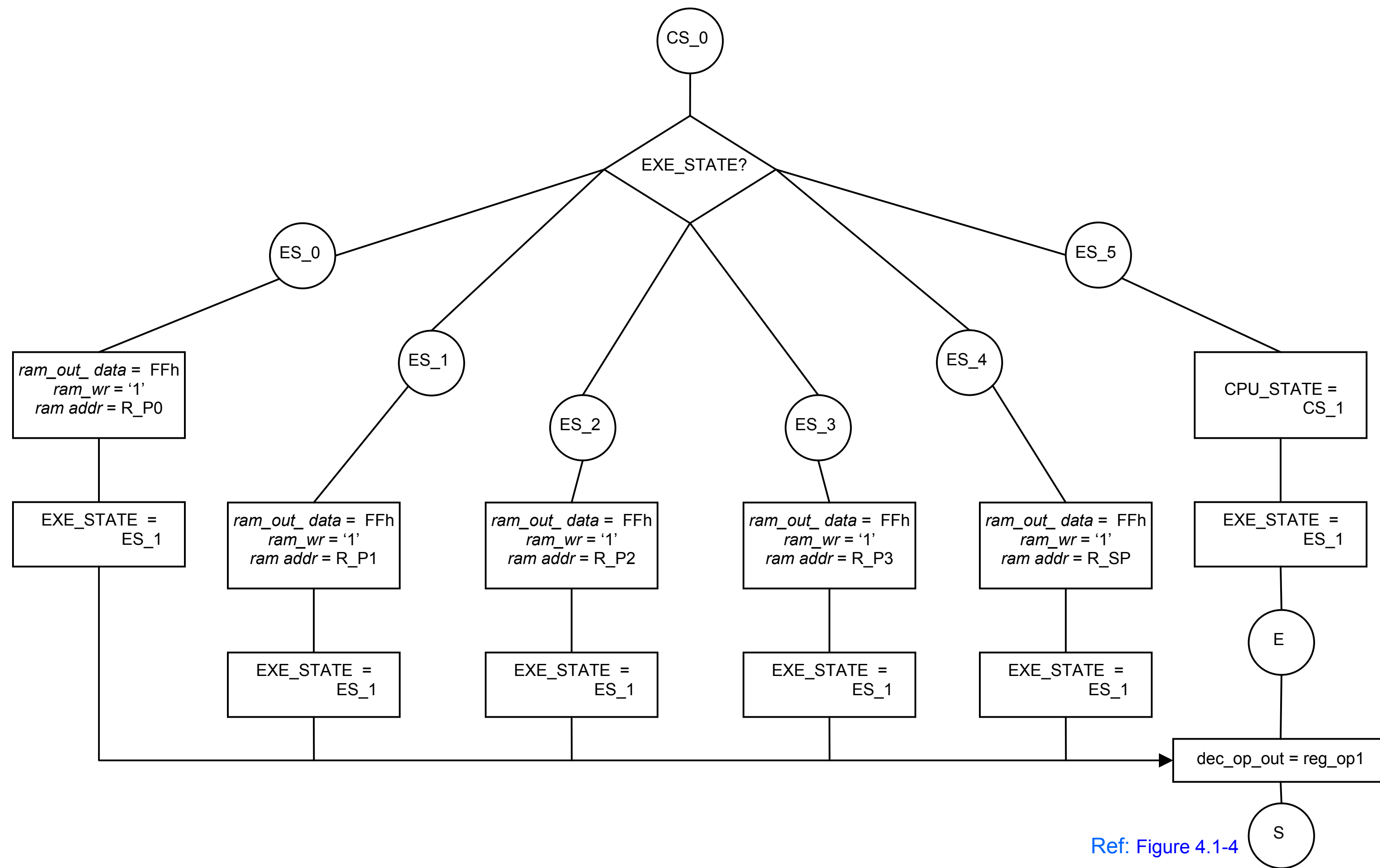


Figure 4.1-5: Flow Diagram for Controller Reset Cycle CS_0

- CS_1: This is reserved for future development of interrupt handling. The controller module currently just advances to CS_2.
- CS_2: Instruction-fetch cycle. Instructions and operands are fetched and decoded sequentially in the following EXE_STATES:

ES_0: Reads a ROM instruction from the address pointed by the PC. Increments address by enabling ALU to perform a PCUADD (program counter unsigned Add). It advances EXE_STATE to ES_1.

ES_1: Reads the PSW register. Changes EXE_STATE to ES_2.

ES_2: Loads *rom_data* to operand 1 register. Reads value of A-register (accumulator) from RAM. Changes EXE_STATE to ES_3.

ES_3: Fetches operand(s) from ROM. Loads PSW. If decoder op code *dec_op_in bit* [7] is set, signifying the requirement of 2nd operand in the instruction, it increments *rom_addr*. The v8051_ctr manages this by enabling the PCUADD in the ALU and using the result from the ALU as the ROM address in the next state. It then changes EXE_STATE to ES_4.

ES_4: Fetches operand(s) from ROM. Loads internal register sfr-acc with *ram_in_data*. If bit [8] is set, signifying the requirement of a 3rd operand in the instruction, it increments *rom_addr*. Changes EXE_STATE to ES_5.

ES_5: Loads operand 2 register. Loads PC with the result of last PCUADD performed by ALU. Changes EXE_STATE to ES_6.

ES_6: Loads operand3 register. Changes EXE_STATE to ES_7.

ES_7: Reset all outputs to ALU. Changes EXE_STATE to ES_0 and CPU_STATE to CS_3. Fetch and decode are completed.

In this cycle the controller module reads 3 consecutive bytes from ROM and stores them in internal registers. It interacts with the decoder to get the decoded op-code, and with the ALU to update the PC, corresponding to the instruction from ROM. It passes the data from its internal registers to the ALU again corresponding to the instruction, during the execute cycle. Refer to Figure 4.1-9 and Figure 4.1-10.

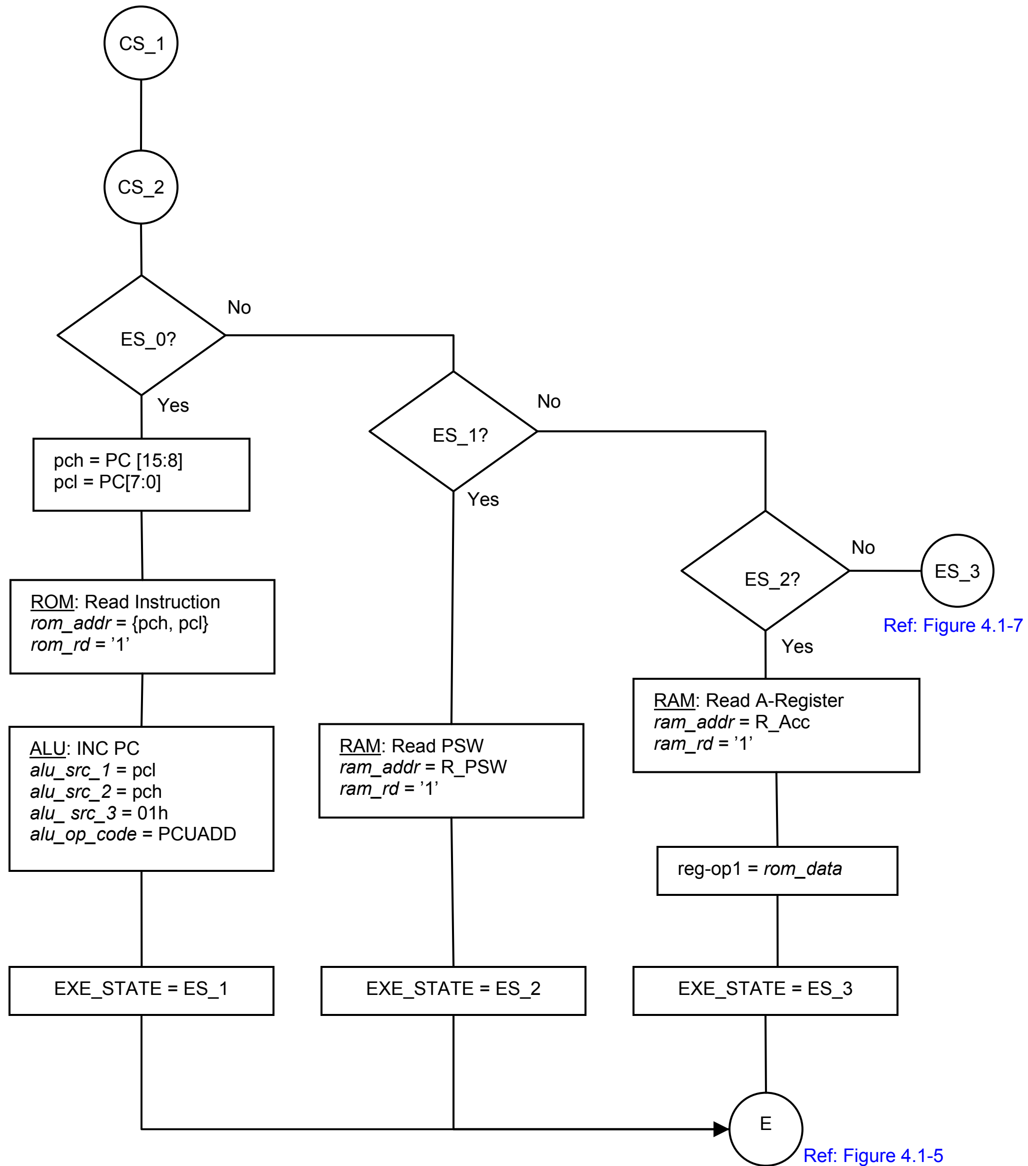


Figure 4.1-6 : Flow Diagram for Instruction Fetch Cycle CS_2 (ES_0 to ES_2)

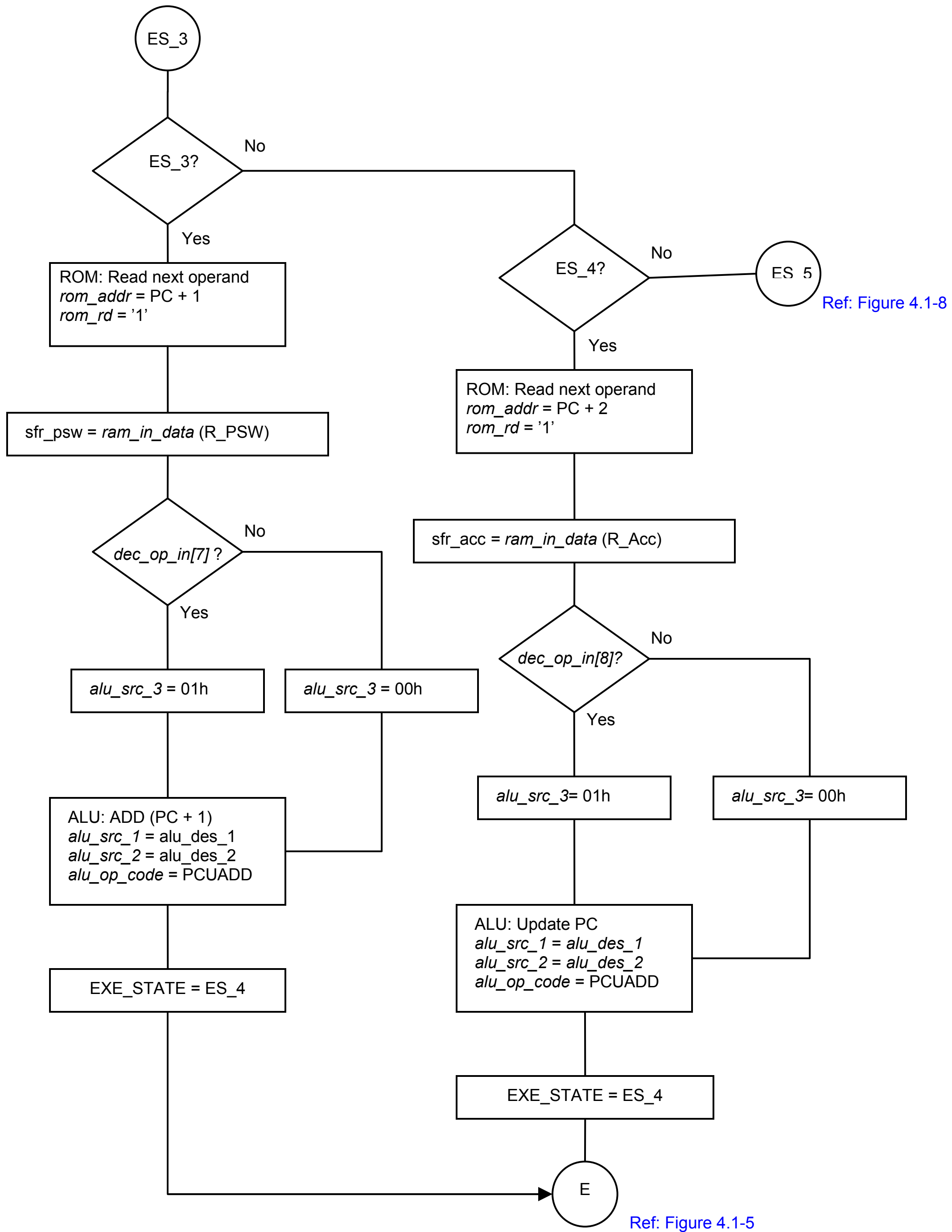


Figure 4.1-7: Flow Diagram for Instruction Fetch Cycle CS_2 (ES_3 to ES_4)

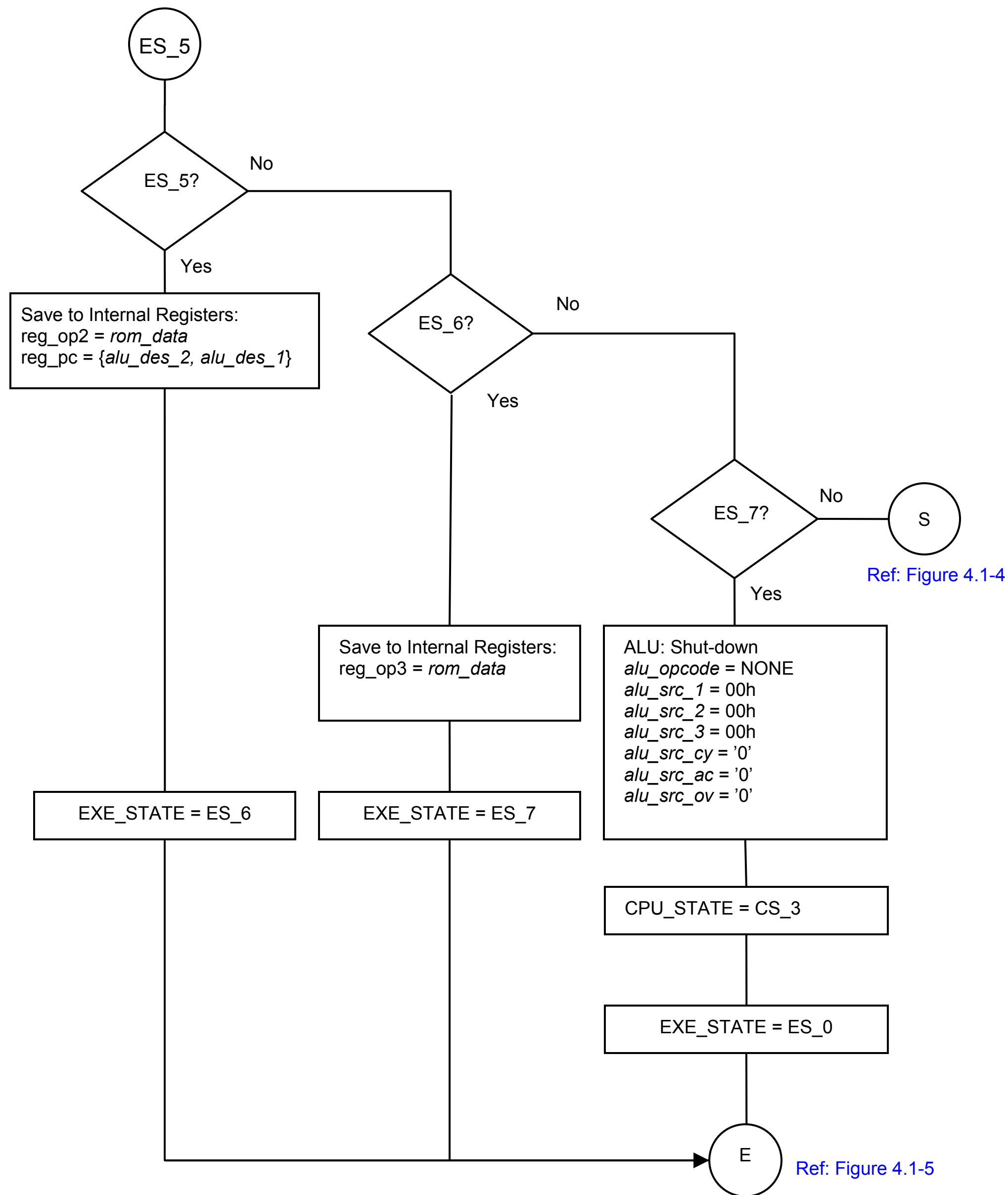


Figure 4.1-8: Flow Diagram for Instruction Fetch Cycle CS_2 (ES_5 to ES_7)

- CS_3: Instruction-execute cycle. The controller scans *dec_op_in [6:0]* and corresponding to the decoded op-code, processes the instruction in a sequence of EXE_STATEs. As an example, three instructions are explained:

I. ACALL addr11 (ACALL):

This is a 2 byte instruction. The controller module, during its CS_2 (instructions fetch/ decode cycle) has this data in reg-op1 & reg_op2, for execution cycle. The operations during the sequence of EXE_STATES are:

ES_0: Enable RAM read and read contents of SP. Change EXE_STATE to ES_1.

ES_1: Change EXE_STATE to ES_2.

ES_2: Enable ALU datapath to increment the contents of the SP register. Change EXE_STATE to ES_3.

ES_3: Enable datapath to RAM. Get contents of low byte of PC. Write contents of PC (low byte) to memory address pointed by the SP. Enable ALU data path to increment contents of SP. Change EXE_STATE to ES_4.

ES_4: Enable datapath to RAM. Get contents of high byte of PC. Write contents of PC (high byte) to memory address pointed by the SP. Change EXE_STATE to ES_5.

ES_5: Enable RAM datapath to update SFR SP. Change EXE_STATE to ES_6

ES_6: Update contents of PC to go to new address. Change EXE_STATE to ES_7.

ES_7: Reset all outputs to ALU. Change EXE_STATE to ES_0 and CPU_STATE to CS_1 to enable fetch of next instruction. Instruction execution completed.

II. ADD A, #data (ADD_4)

This is a 2 byte instruction, with op-code in first byte and data in second byte. The operations during the sequence of EXE_STATES are:

ES_0: Enable ALU datapath to perform an addition of contents of reg_acc and reg_op2. Change EXE_STATE to ES_1.

ES_1: Enable RAM datapath to result of addition obtained from ALU to SFR A-register and to update the internal status flags. Change EXE_STATE to ES_3.

ES_2: enable RAM datapath to update the SFR PSW. Change EXE_STATE to ES_3

ES_3: Change EXE_STATE to ES_4.

ES_4: Change EXE_STATE to ES_5

ES_5: Change EXE_STATE to ES_6.

ES_6: Change EXE_STATE to ES_7.

ES_7: Reset all outputs to ALU. Change EXE_STATE to ES_0 and CPU_STATE to CS_1 to enable fetch of next instruction. Instruction execution completed.

III. RLC A:

ES_0: Enable ALU datapath with A-register contents in source operand1 and the contents of carry flag. Change EXE_STATE to ES_1.

ES_1: Enable RAM datapath to update SFR A-register and the controller internal *sfr_psw* respectively, with result of rotate operation in ALU module. Change EXE_STATE to ES_2

ES_2: Enable RAM datapath to update SFR PSW. Change EXE_STATE to ES_3.

ES_3: Change EXE_STATE to ES_4.

ES_4: Change EXE_STATE to ES_5.

ES_5: Change EXE_STATE to ES_6.

ES_6: Change EXE_STATE to ES_7.

ES_7: Reset all outputs to ALU. Change EXE_STATE to ES_0 and CPU_STATE to CS_1 to enable fetch of next instruction. Instruction execution completed.

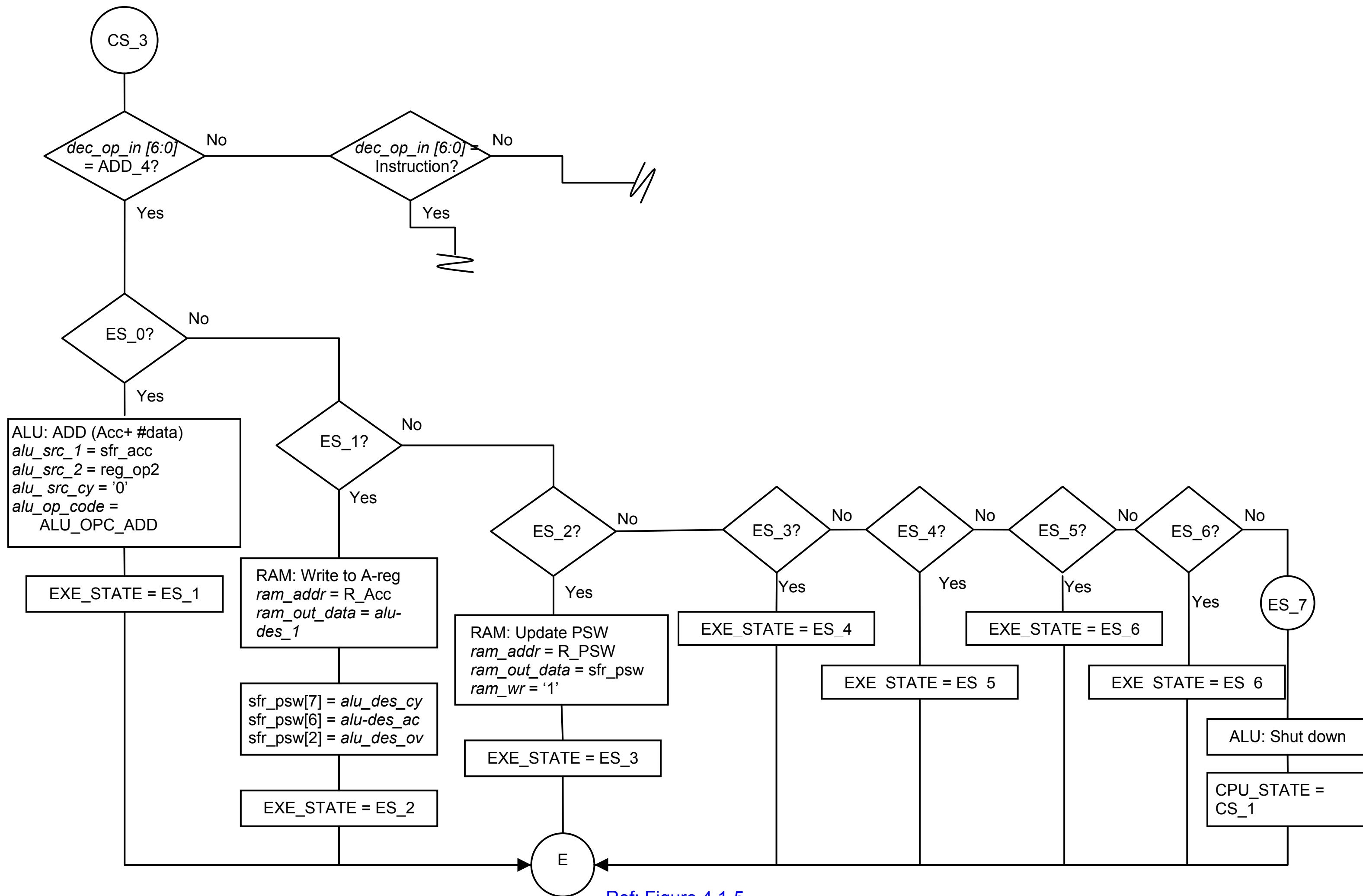


Figure 4.1-9: Flow Diagram for Instruction Execute Cycle CS_3 for Arithmetic ADD A, #data

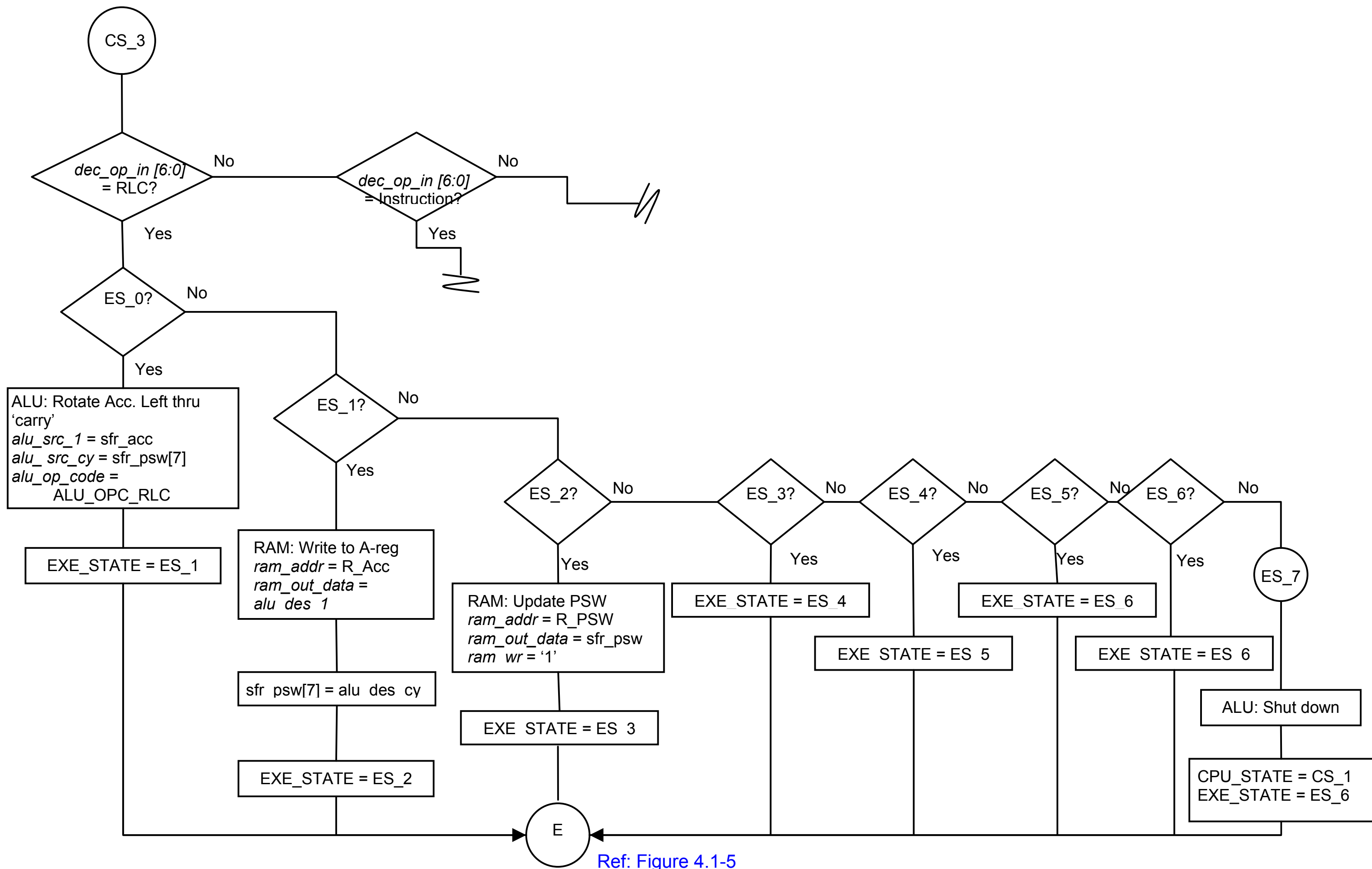


Figure 4.1-10: Flow Diagram of Instruction Execution Cycle CS_3 for Shift Left RLC A

4.1.2. Simulation and Testing

The controller FSM diagram shows that the CPU_STATES and EXE_STATES are sequencing as designed. The function of the controller module is to synchronize the control signals to / from other components of the 8051. Therefore the controller was tested after integrating all the modules. A test program, Test_r1, was loaded in the ROM, a Synchronizing Clock of 12Mhz was started and the inter-module signals, during Controller CPU_STATEs & EXE_STATEs were monitored. CS_0 state is entered when reset is asserted. The controller resets the SFR in RAM, namely ports (p0 to p1) to FFh and the SP to 07h, by sequencing thru the EXE_STATES ES_0 thru ES_5.

Figure 4.1-11 shows CS_2 the instructions fetch and decode cycles. The controller reads the ROM contents at the address pointed by the PC, interfaces with the ALU to perform a PCUADD (Program Counter Unsigned ADD) and increments its local pc-register; sends the instruction read from ROM to decoder; the decoder returns a 9-bit decoded op-code; the controller fetches the next location from ROM, and if bit-7 of decoder op-code is set, sends a source operand of value 1 to perform a PCUADD by the ALU, and updates its local pc-register; then fetches ROM contents as pointed by pc-register, and if bit-8 of the decoder op-code is set sends a source operand value 1 else 0 to the ALU for PCUADD. The controller having collected the data updates the PC and operands for the instruction execution cycle. To account for the register latching delays the

controller loads its registers with data from ROM or RAM modules at the third EXE_STATE with respect to the corresponding ‘read’ control signal.

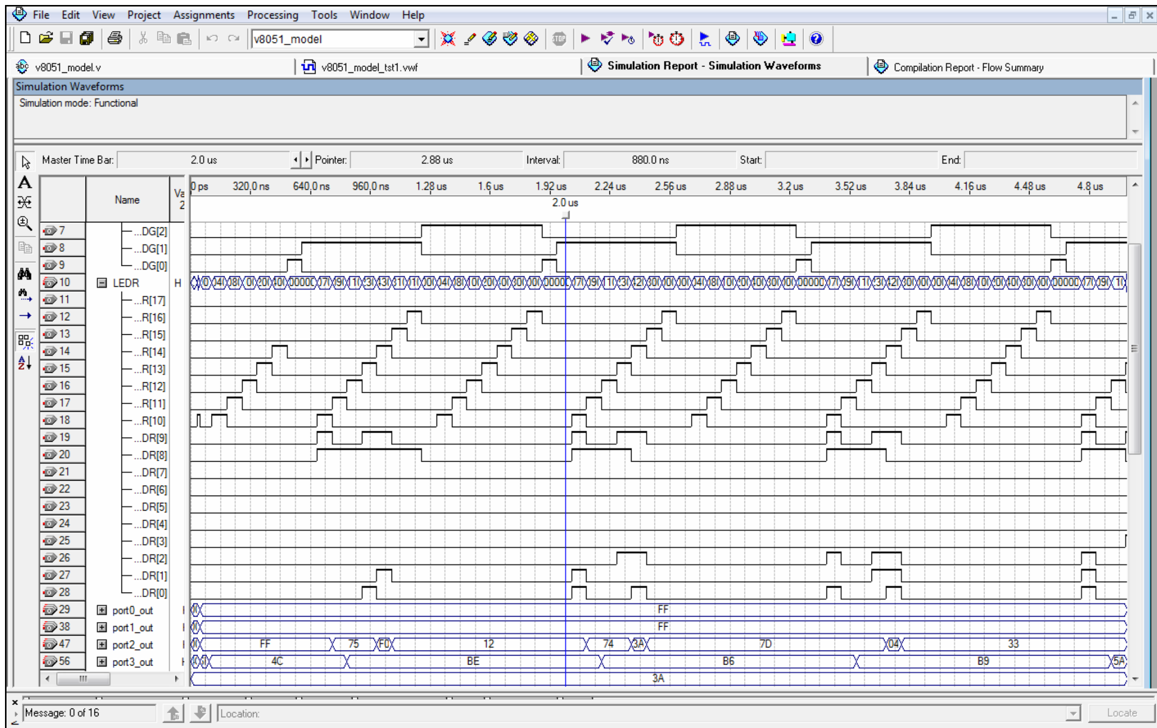


Figure 4.1-11: Instruction Fetch & Decode During CS_2

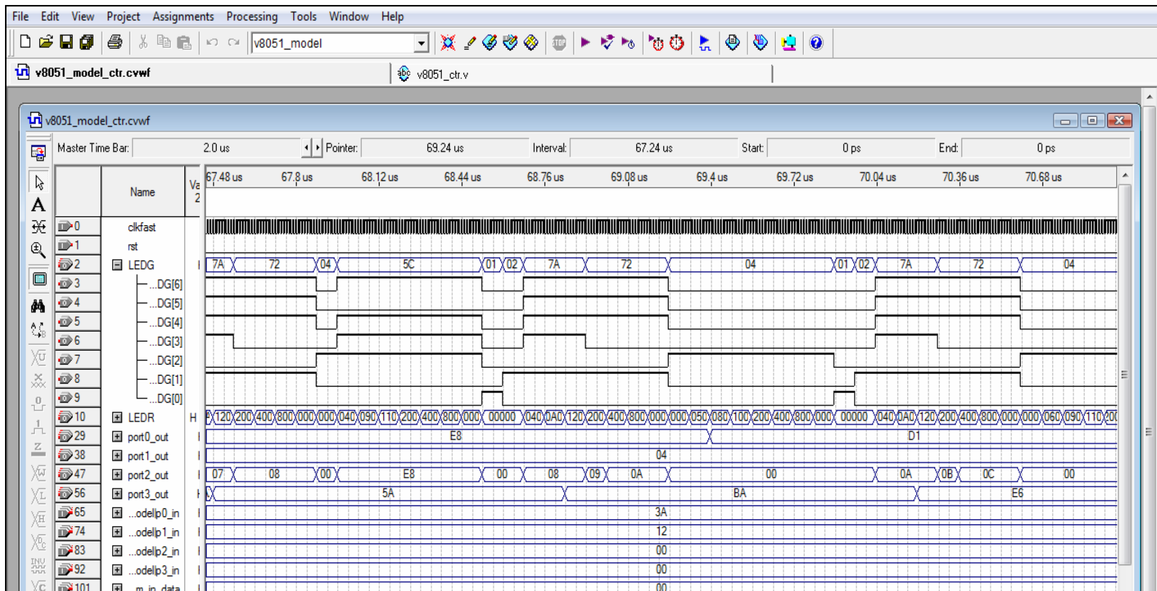


Figure 4.1-12: Execute Cycle CS_3 for Instruction RLC

Figure 4.1-12 shows details of the instruction execution cycle for the 8051 instructions RLC A (rotate A-register left through carry) and the next instruction MOV P0, A (move contents of A register to output port p0). For execution of RLC in this example, at ES_0 the controller enables the ALU by sending control variable *alu_opcode* (‘B’ corresponding to RLC) and the A-register contents as source operand to perform the rotate operation. During ES-1 the controller loads the data lines to RAM with the result from the ALU and enables ‘RAM write’ to write the new value to the SFR A-Register. The carry-bit from ALU is also stored in its internal register. During ES_2 it sends the data to update the PSW, also an SFR on RAM. The controller sequences thru the next states ES_3 to ES_6 until it reaches ES_7 when the execution is completed by resetting all variables to ALU, and jumping to CS_1 and ES_0 to fetch next instruction. In this example the next instruction is MOV P0, A. The waveforms of

Figure 4.1-12 show that in the fetch cycle the decoder output corresponds to the MOV instruction. During the execution cycle of this instruction there is no ALU operation. During ES_0, the contents of A-register are written to the port address, also an SFR in RAM. The rotated data value is seen on port0-out on the simulation waveform.

4.1.3. Verification

The results of the simulation waveforms verify that the controller module is operational.

4.2. ROM

The ROM is the dedicated 4KB of program memory space on the 8051 microcontroller. It contains the binary code with the instructions to be executed. The ROM module takes as input the 12 bits of the program counter PC, and gives as output the 8- bit data of instruction op-code and operands.

4.2.1. Detailed Design

Altera recommends the use of a ‘Parameterized ROM mega-function’ to implement all ROM functions[1]. This mega-function allows for a synchronous read only memory with clock for strobing in the address. The Quartus II ‘Mega-function Plug-in Manager’ tool was used to configure the LPM_ROM mega-function [14] named “v8051_rom_mem” for the following parameters to implement the 8051 ROM:

Memory size 4096 words.

Word width 8-bits

Address width 12- bits

Initialize file – A hexadecimal (Intel Format) file of the program code to be stored in the ROM memory.

Test programs/applications were written in 8051 assembly language. The assembled test runs were verified on the 8051 simulator [10]. The program code

[1] QUARTUS II Help on LPM_ROM

in Intel hex format was stored in the module directory, was then specified as data file for loading the ROM at power-up/ reset.

The memory address is enabled at the positive clock edge, requires a hold- time of 3 clock pulses for the memory content to become available on the data lines.

The ROM module therefore requires two clock inputs, a fast clock to read the ROM and a slower one to synchronize the *rom_data* with the controller timings. The ROM_module instantiates the ROM mega-function v8051_rom_mem to read the instructions at valid *rom_addr* from controller and to latch it to the out data register in the *rom_rd* signal interval.

4.2.2. Simulation & Testing

To test the program code in ROM, this module was simulated with the following input conditions, and the data lines were observed:

Clock set at 12MHz. Read asserted at 6 MHz frequency. Address lines incremented by 1 starting from 00H. The simulation was carried for a period of 5 μsec.

Result: Refer to listing of the text file 'Test1_r1' below. The data lines (ref Figure 4.2-2) read the values of the first column below which is program code.

```
75      //MOV_12
F0
12
74      // MOV_4
3A
7D      //MOV_7
04
```

```
33      //RLC
F5      //MOV_8
80
C5      //XCH_2
F0
13      //RRC
F5      //MOV_8
90
C5      //XCH_2
F0
DD      //DJNZ_1
F4
80      //SJMP
EB
00      //NOP
00      //NOP
```

Figure 4.2-1: ROM Contents of Test Program Test1_r1.txt - 8051 Instructions

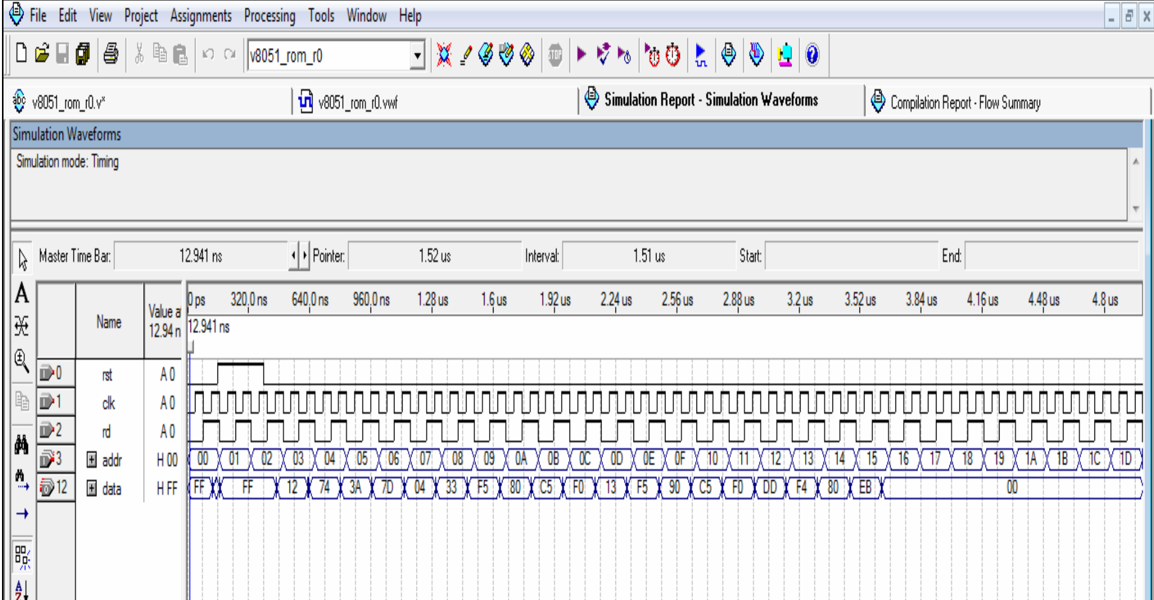


Figure 4.2-2: Simulation Waveform for ROM Module (Verilog Model)

4.2.3. Verification

The same simulation conditions were applied to the ROM module of Dalton model. The two waveforms are similar.

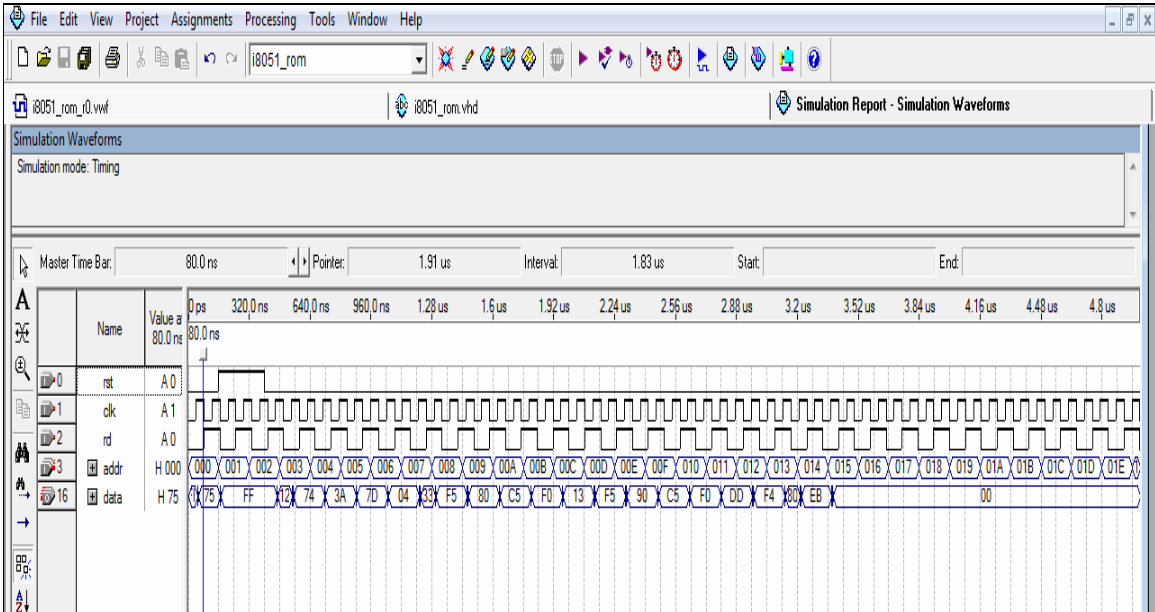


Figure 4.2-3: Simulation Waveform for ROM Module (Dalton Model)

4.3. RAM

The 8051 has an internal Data Memory (internal RAM) of 256 bytes. The internal RAM is divided into 2 blocks: the first 128 byte block is the general purpose RAM, and a second part starting from address 80H, is the Special Function Register (SFR) area [15].

The RAM module defines an array of 128 bytes in the address range 00h-7Fh and 21 Special Function Registers (SFR) within the address space 80H-FFh.

4.3.1. Detailed Design

The detailed design of internal RAM (Data memory) of the 8051 IP soft core is shown in Figure 4.3-1: Flow Diagram for RAM. The RAM module generates a 128 byte array of write (parallel load) and read (output enable) memory (registers). Locations 20H to 2FH are bit addressable. It also has 21

bytes allocated to function as Special Function Registers (SFRs) of the 8051 microcontroller. Eleven of these SFRs are bit-addressable [15].

Verilog does not support two-dimensional array in which any cell can be addressed. A word in Verilog memory can be addressed directly. A cell (bit) in a word is addressed by first loading the word onto a buffer and then addressing the bit of the word [17], [18].

The Verilog code for bit read from bit addressable internal memory (20h-2Fh) was accomplished as follows:

```
begin
add = {4'b0010, addr [6:3]};          // address hi_nibble= 2h, lo_nibble =addr [6:3]
memword = iram[add];                  // get contents of addressed memory
out_bit_data = memword [cbit];        // read bit value
end
```

The v8051_ram module performs the following tasks:

Clears the memory when reset signal reset is asserted.

Performs a synchronous read or write from / to the addressed memory location.

If bit data is requested, reads or writes to the addressed bit number of the addressed memory location.

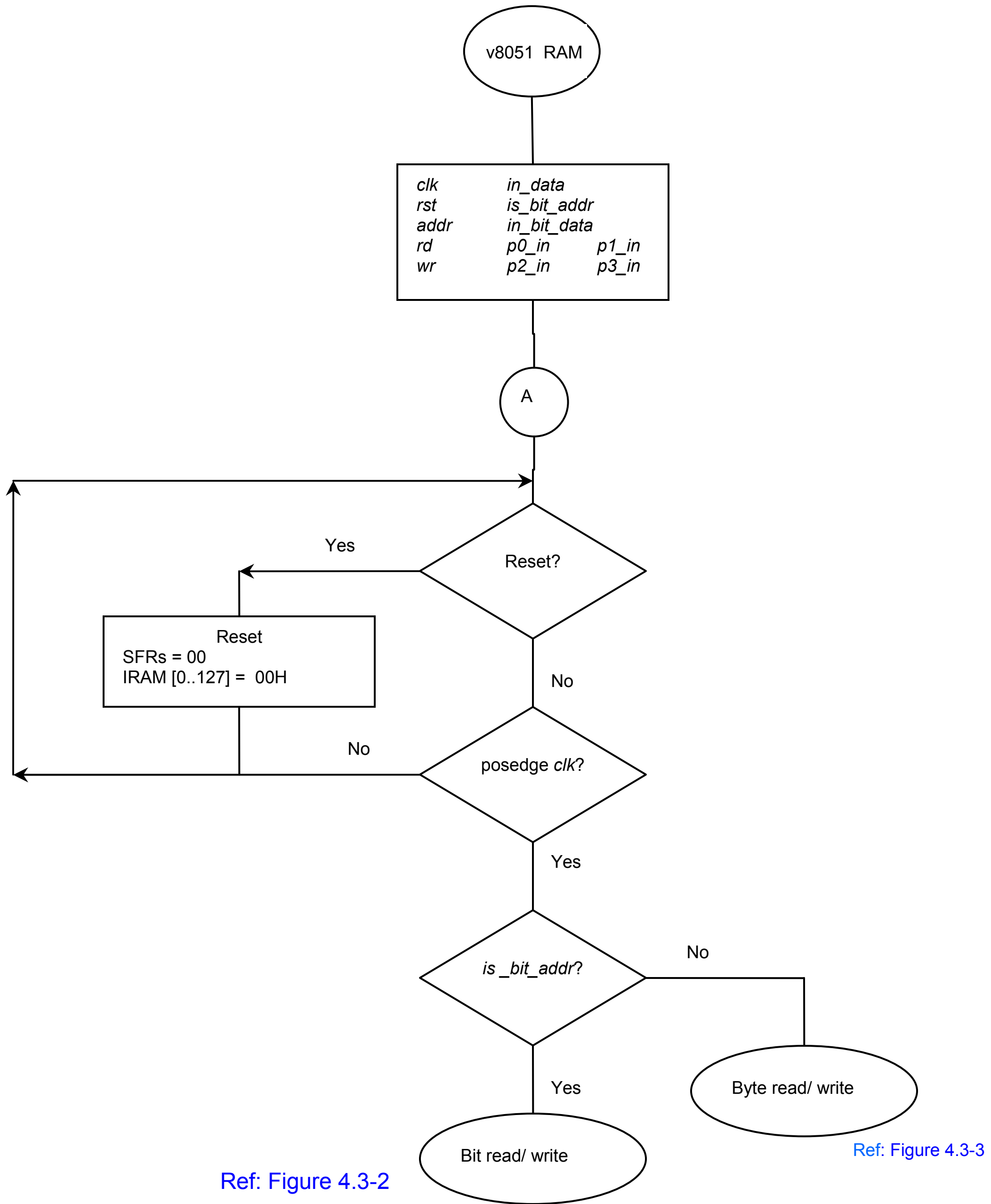


Figure 4.3-1: Flow Diagram for RAM Module

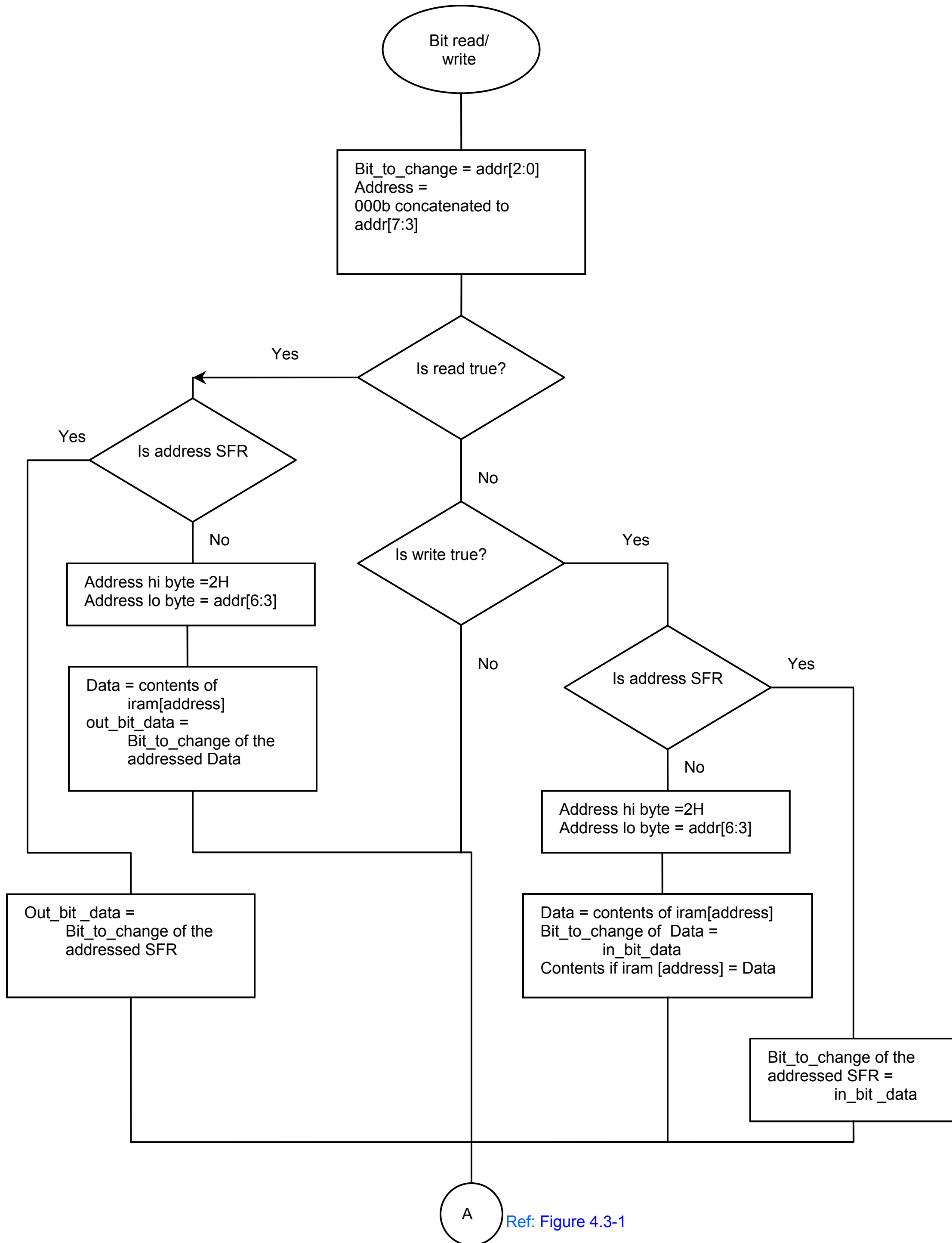


Figure 4.3-2: Flow Diagram for RAM - Bit Manipulation

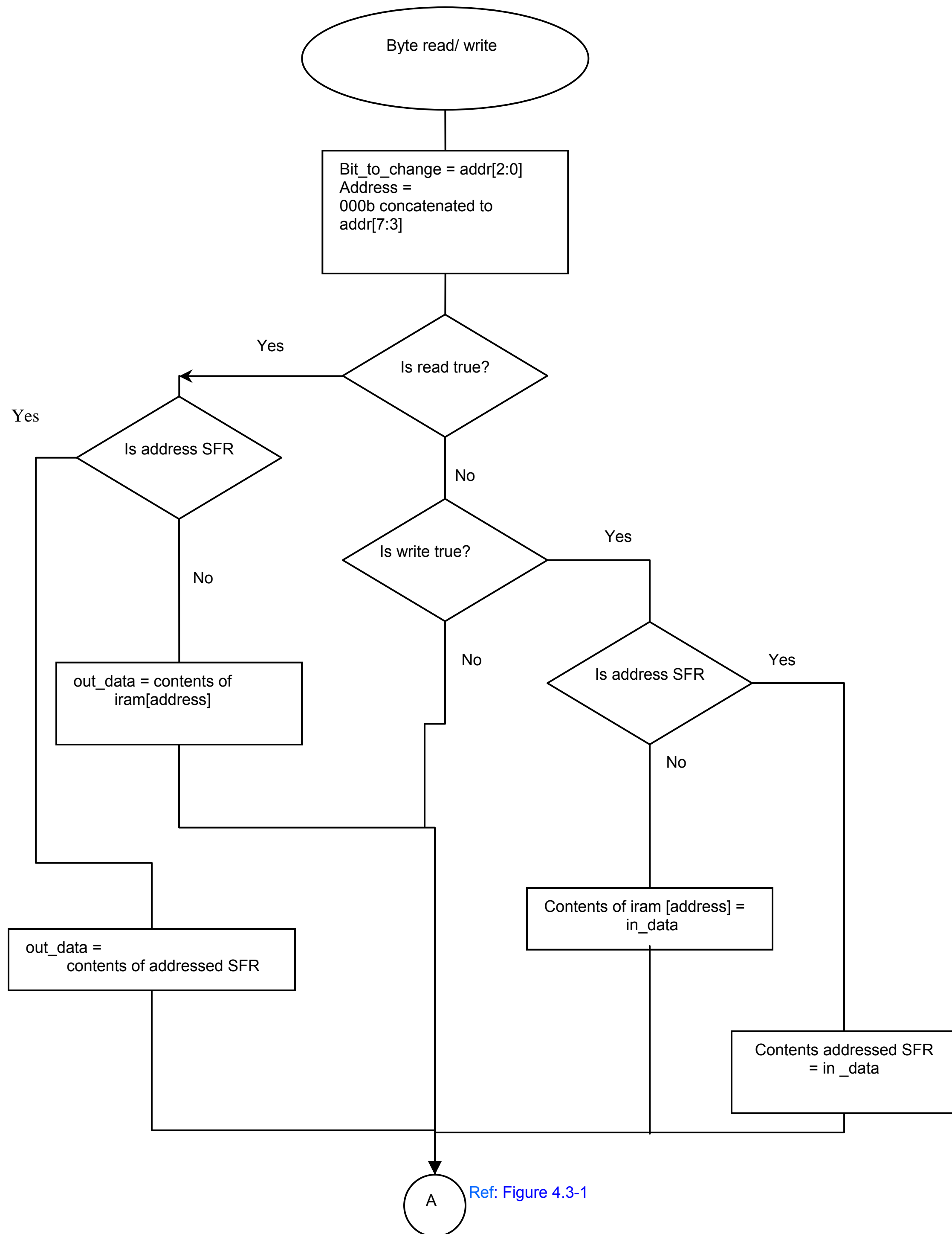


Figure 4.3-3: Flow Diagram for RAM - Byte Read /Write

4.3.2. Simulation and Testing

The input signals to the RAM module were defined in the vector waveform file. Clock frequency was set to 12 MHZ (standard 8051 clock specification)

The RAM module was tested for the following simulated functions/ conditions:

1) Reset.

   a) Test: While clock was continuously applied reset was asserted. Memory locations were then read to confirm response of the module.

   b) Observation: Refer to the waveforms in Figure 4.3-4**.**

   While reset is asserted *out_data* is seen to be held at last read memory value, indicating read and write operations were inhibited.

   For subsequent 'read' pulses, *out_data* changed to contents of addressed RAM or SFR, at the positive edge of clock.

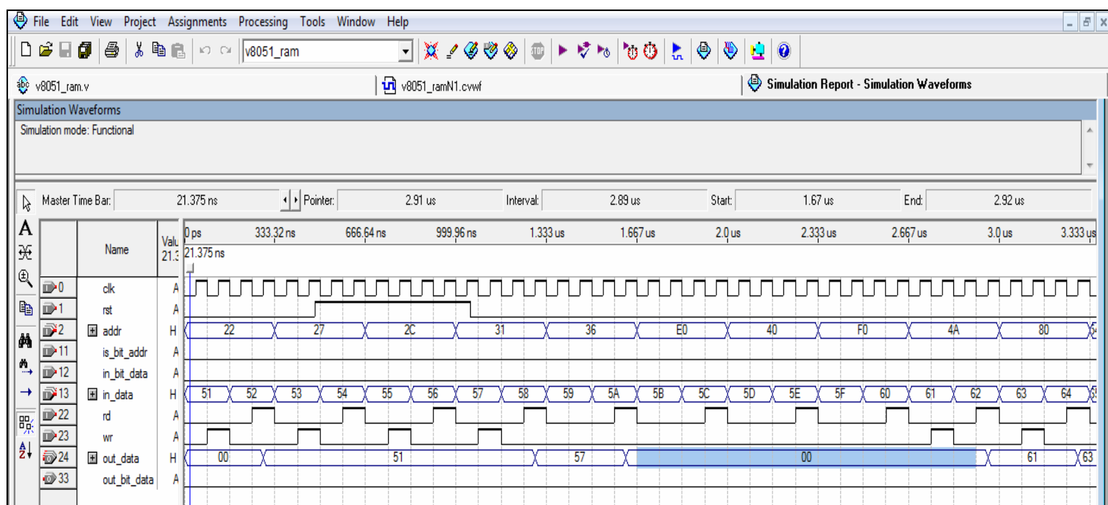


Figure 4.3-4: Reset Asserted

c) Result: During reset memory is initialized and all other operations are inhibited. The contents of RAM and Registers are reset to 00H

2) Byte Read /write

For byte read /write tests, reset was asserted for approx 10 cycles, *is_bit_address* and *in_bit_data* (the bit addressing signals) were forced 'lo' (disabled), and address, in_data, read and write signals were varied to monitor the RAM module functions.

a) Test:

i) RAM

Address incremented from 00H to 0DH, each address is held for 4 clock cycles.

In-data lines count from A2H, incrementing by 01h and hold the data for 2 clock cycles.

For each address write and then read was asserted.

ii) SFRs

Addresses E0H (A-register), F0H (B-register), 80H (Port 0) and 81H (SP) sequentially asserted on address lines, and each address was held for 4 clock cycles. Write and then read signals pulsed for each addressed location.

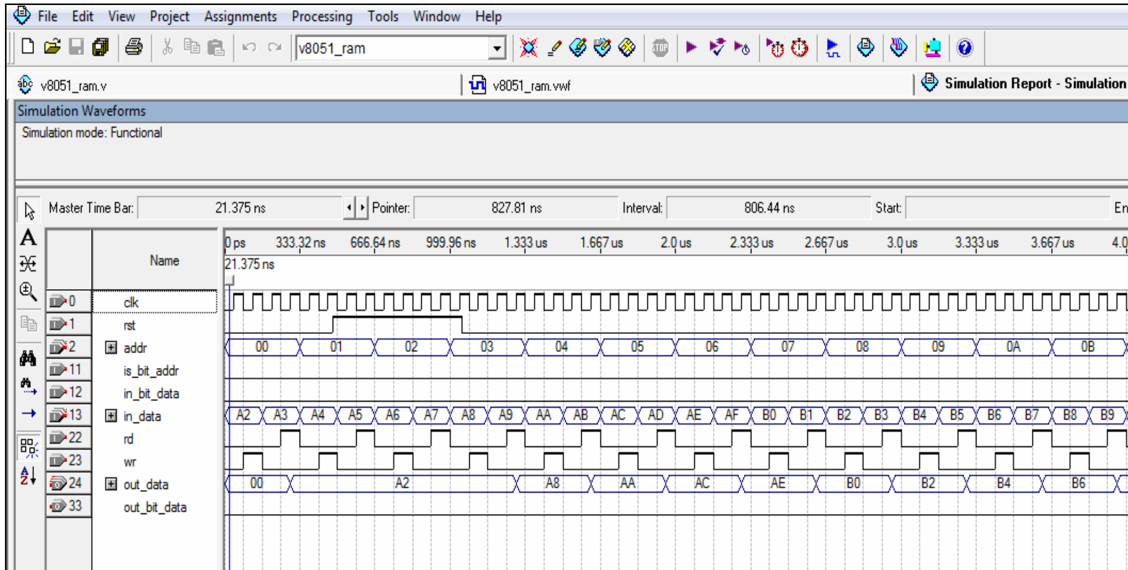


Figure 4.3-5: Byte Read / Write to RAM - Functional Simulation

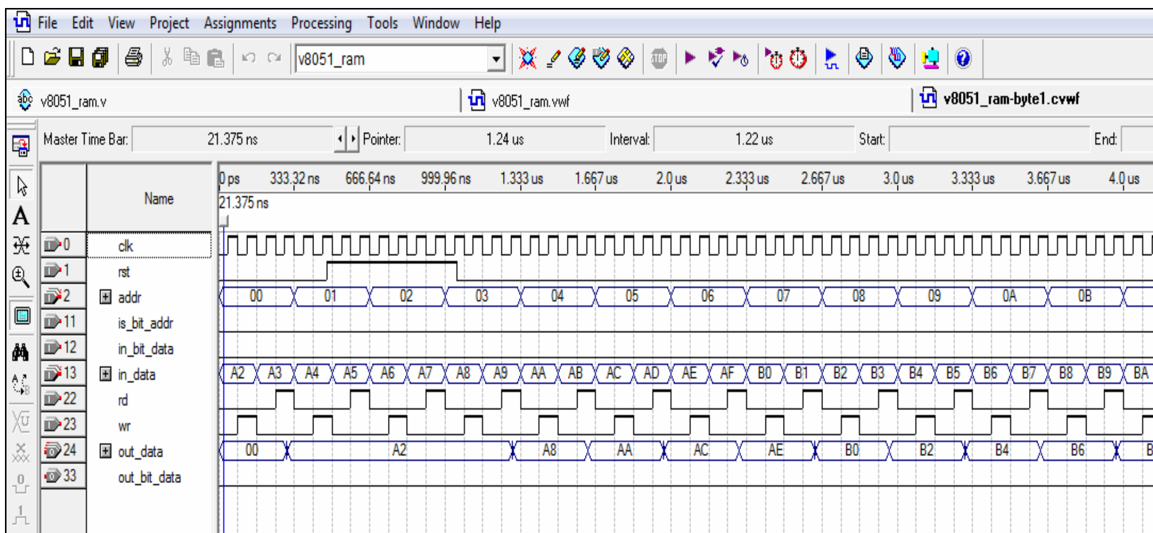


Figure 4.3-6: Byte Read / Write to RAM - Timing Simulation

Timing Analyzer Summary

	Type	Slack	Required Time	Actual Time	From	To	From Clock	To Clock	Failed Paths
1	Worst-case tsu	N/A	None	13.642 ns	addr[5]	out_data[1]~reg0	--	clk	0
2	Worst-case tco	N/A	None	7.915 ns	out_data[5]~reg0	out_data[5]	clk	--	0
3	Worst-case th	N/A	None	-0.432 ns	addr[6]	iram[34][5]	--	clk	0
4	Clock Setup: 'clk'	N/A	None	145.69 MHz (period = 6.864 ns)	sfr_p1[4]	out_bit_data~reg0	clk	clk	0
5	Total number of failed paths								0

Figure 4.3-7: Worst-Case Delays in Timing Simulation

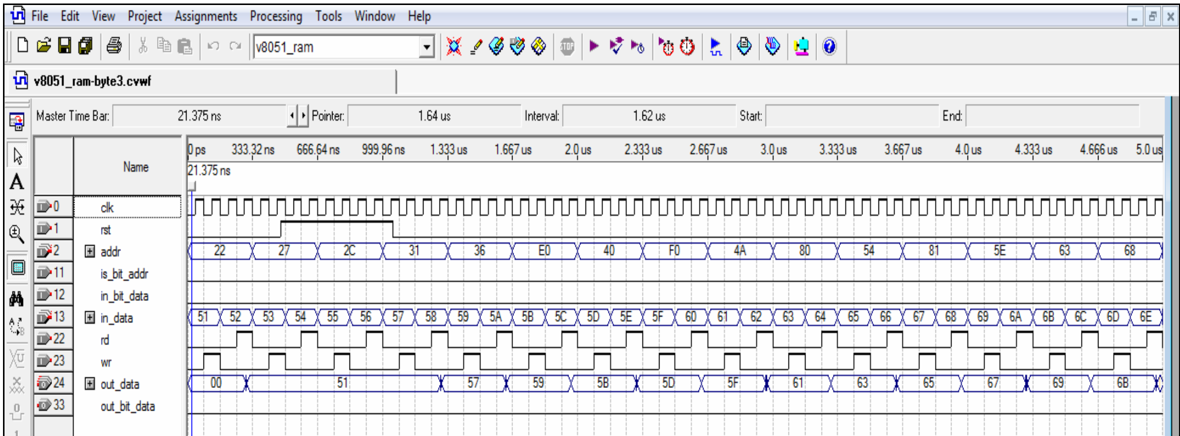


Figure 4.3-8: Byte Read / Write to RAM / SFRs

b) Observations:

i) The waveforms in Figure 4.3-5, Figure 4.3-6 and Figure 4.3-8, show *out-data* holds the contents of addressed memory location and switches to corresponding *in-data* at the consecutive read pulse synchronously with positive edge of clock pulse.

ii) No switching on *out_bit_data* line

iii) The timing simulation waveforms indicate the same functionality. Maximum achievable frequency during simulation is over 145MHz, refer to Figure 4.3-7.

3) Bit Manipulation

For bit read /write tests, a short reset was asserted, then input signals simulated as follows:

a) Test

i) RAM bit manipulation

Address incremented from 20H to 2FH, each address is held for 2 clock cycles. *in-data* lines count from A2H, incrementing by 11h and hold the data for 4 clock cycles. *is_bit_addr* asserted in parallel with write and then with read signals. *in_data* line switched for serial data C3H, 67H.

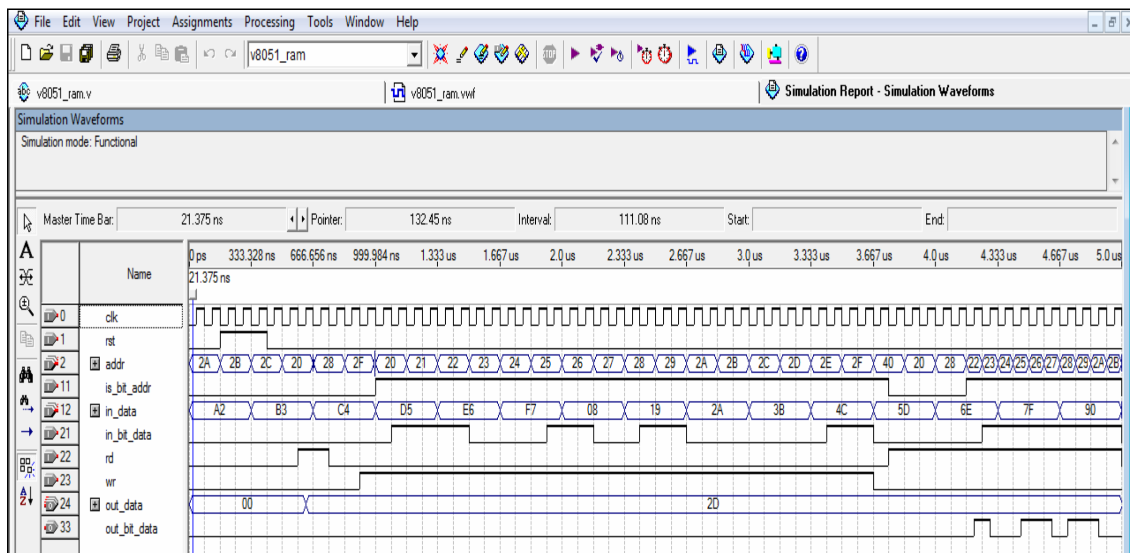


Figure 4.3-9: Bit Manipulation on RAM Locations 20H to 2FH

ii) SFR bit manipulation

Address 90H (Port 1), D0H (PSW), E0H (A-register), and F0H (B-register), asserted on address lines, and each address was held for 4 clock cycles. *in-data* lines switched to E6H, 2AH, F7H, and F7H each held for 4 clocks, write asserted.

Then, *rd* and *is_bit_addr* asserted to read the above registers.

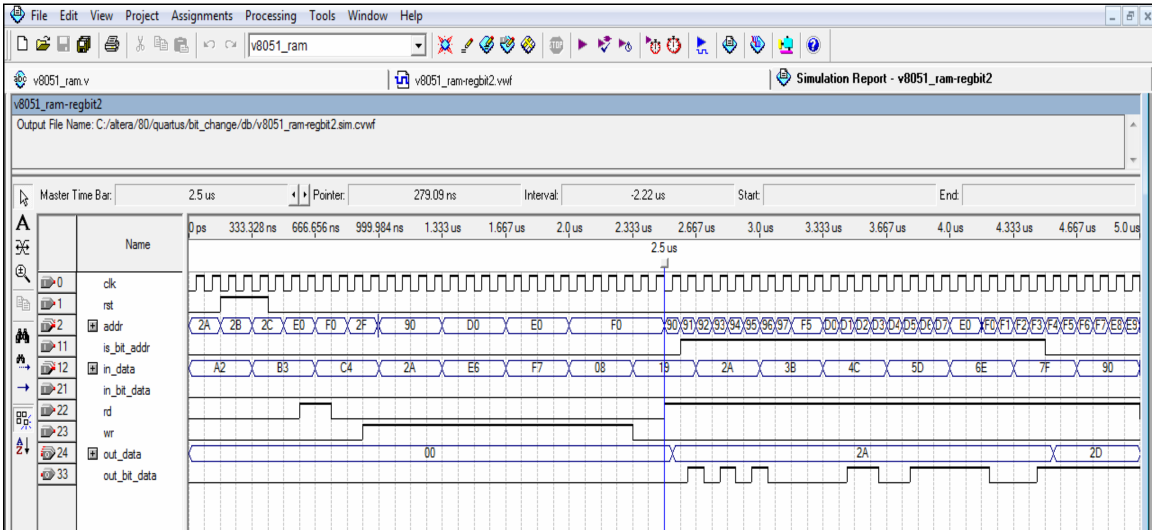


Figure 4.3-10: Bit Read from SFRs – A-Register, B-Register, Port1, & PSW

b) Observation

Referring to Figure 4.3-9 and Figure 4.3-10

i) Waveforms show *out-data* holds contents of addressed memory location and switches to corresponding *in-data* at the consecutive read, only if *is_bit_addr* is not asserted.

ii) Switching on *out_bit_data* line occurs when *is_bit_addr* and read is true.

iii) Data as written to the addressed bit of the respective location during write was correctly read back.

4.3.3. Verification

The simulation waveforms indicate that the module performs the functions of the internal memory (128 locations + registers) of the standard 8051. Correct addressing as well as bit and byte read / write, as per design, was observed.

4.4. External RAM

The External RAM is just like the RAM module but has 16 address lines and can have a size of up to 64KB. This memory is external to the 8051 module. The controller can access the external memory for read or write of 8-bit data in response to a MOVX instruction from ROM.

4.5. Decoder

The function of this module is to convert the op-code of the 8051 instruction, as read by the controller from program memory, to a pointer for the controller to implement the corresponding execution cycle for that instruction. The decoder also generates corresponding signals if the instruction requires additional operands.

4.5.1. Detailed Design

The 8-bit op-code input from the controller is converted by checking against a look-up table defined in the ‘opcodelookup.txt’ file and stored in the project directory. The output of the decoder is a 9-bit signal composed of 7-bit op-code pointer, bit 8 set if instruction requires 2nd operand, and bit 9 set if instruction requires a 3rd operand.

4.5.2. Simulation & Testing

Functional simulation was carried out on the Decoder module by varying the input op-code signal at 10 ns for a period of 260ns. 2 types of addressing each for arithmetic and logical instructions were tested. Table 4.5-1 shows input conditions and expected results. Figure 4.5-1 shows the simulation waveforms correspond to the expected results.

Table 4.5-1: Decoder Test - Instructions & Expected Results

Type	Instruction	Opcode_in			Opcode _out [bits]		
		hex	binary	parameter	[8]	[7]	[6:0]
Immediate to acc.	ADD A, #xx	24	00100100	ADD-4	0	1	0000100 (04H)
Indirect RAM to acc.	ADD A, @Ri	26	0010011x	ADD-3	0	0	0000011 (03H)
Register from Acc. w/borrow	SUBB A, Rn		10011xxx	SUBB-1	0	0	1100000 (60H)
Direct from Acc. w/borrow	SUBB Alder	95	10010101	SUBB-2	0	1	1100001 (61H)
Move A= (A+DPTR)	MOVC A,@A+DPTR	93	10010011	MOVC-1	0	0	1001010 (45H)
No operation	NOP	00	00000000	NOP	0	0	1001100 (4CH)
Shift right	RR A	03	00000011	RR	0	0	1011011 (5BH)
Shift left w/carry	RLC A	33	00110011	RLC	0	0	1011010 (5AH)
Logical acc to direct RAM	ANL dir, A	52	01010010	ANL-5	0	1	0001110 (0EH)
Logical immediate to direct RAM	ANL dir, #xx	53	01010011	ANL-6	1	1	0001111 (0FH)
Logical direct RAM to Acc.	XRL A, dir	65	01100101	XRL-2	0	1	1101010 (6AH)
Logical register to Acc.	XRL A, Rn		01101xxx	XRL-1	0	0	1101001 (69H)

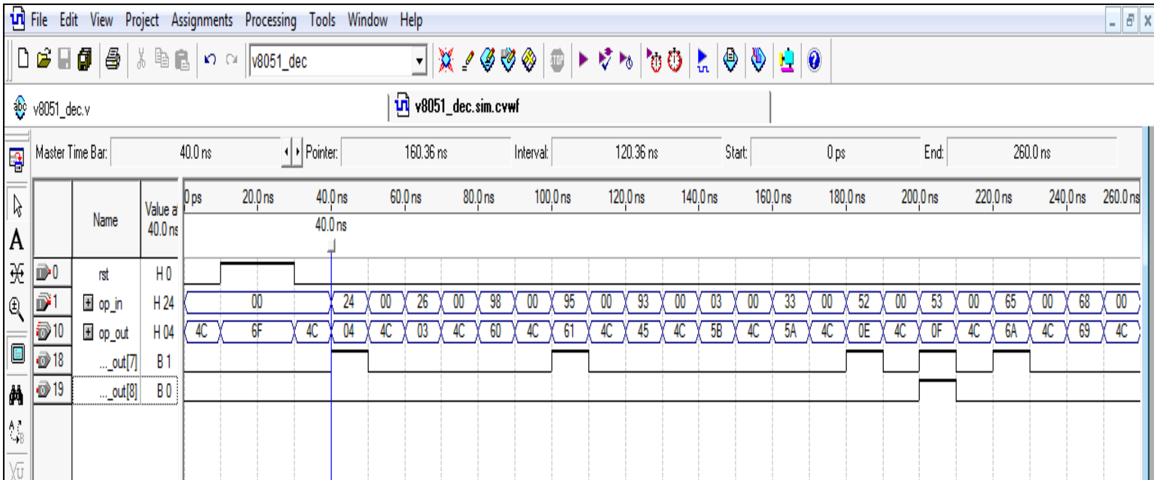
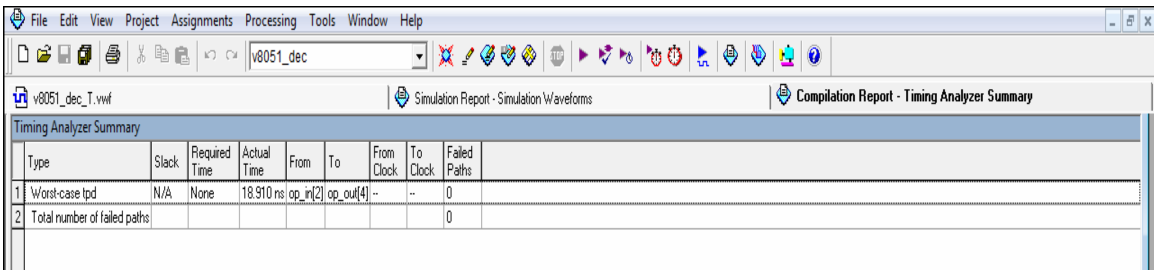


Figure 4.5-1: Simulation of 8051 Instruction Decoding (Verilog Model)

Timing simulation for the same input conditions was carried out. Maximum input to output delay was approx. 18ns. Refer to Figure 4.5-2.



Timing Analyzer Summary

	Type	Slack	Required Time	Actual Time	From	To	From Clock	To Clock	Failed Paths
1	Worst-case tpd	N/A	None	18.910 ns	op_in[2]	op_out[4]	--	--	0
2	Total number of failed paths								0

Figure 4.5-2: Worst–Case Delay in Timing Simulation of Decoder Functions

4.5.3. Verification

The same tests were applied to the decoder module of the Dalton model. Quartus II “Compare waveform file” command was used to compare the simulation waveforms obtained from the Dalton module with that of the Verilog module. View of both sets of waveforms is shown in Figure 4.5-3. The waveforms from the simulation of both the modules are overlapped, indicating an exact match.

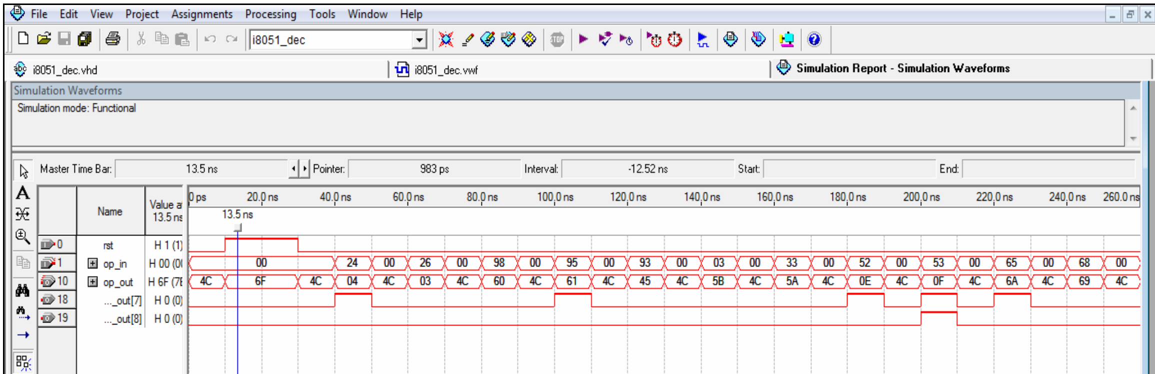


Figure 4.5-3: Comparison Report of Simulation of 8051 Instruction Decoding (Dalton Model with Verilog model)

4.6. ALU

The Arithmetic Logic Unit, as the name suggests, performs the arithmetic and logical operations on the instructions being executed. The Verilog module performs 16 types of functions to implement the 111 logical & arithmetic instructions of the 8051 microcontroller instruction set. The ALU module receives from the controller, three 8-bit source operands, status of carry flags, and the op-code for instruction type.

4.6.1. Detailed Design

The operation of the ALU implements the following functions selected by the 4-bit op-code received from the controller:

- NOP - 0000: no operation is performed, all registers retain their values.
- Arithmetic Instructions

  ADD - 0001: The ALU module executes this instruction by performing a 4-bit full-adder operation on two 8-bit numbers *src_1* and *src_2*. Result is an

8-bit number *des_1* which is sent back to the controller. The carry flag is set if there is a carry from bit 7. The overflow flag is set if there is a carry from either bit 7 or bit 6 but not from both. The auxiliary-carry flag is set if there is a carry from bit 3 (lower nibble).

Table 4.6-1: ALU Operations

Op-code	Operation	Description	Flags affected
0000	ALU_OPC_NONE	No operation	
0001	ALU_OPC_ADD	*src_1* + *src_2*	c, ac, ov
0010	ALU_OPC_SUB	*src_1* – *src_2*	c, ac, ov
0011	ALU_OPC_MUL	*src_1* * *src_2*	ov
0100	ALU_OPC_DIV	*src_1* / *src_2*	ov
0101	ALU_OPC_DA	*src_1*: any nibble > 9, Adjust to decimal equivalent	c, ac
0110	ALU_OPC_NOT	Compliment *src_1*	none
0111	ALU_OPC_AND	Bitwise *src_1* AND *src_2*	none
1000	ALU_OPC_XOR	Bitwise *src_1* XOR *src_2*	none
1001	ALU_OPC_OR	Bitwise *src_1* OR *src_2*	none
1010	ALU_OPC_RL	Rotate *src_1* left: *[0] → [7]*	none
1011	ALU_OPC_RLC	Rotate *src_1* left thru carry: *[0] → c*	c →[7]
1100	ALU_OPC_RR	Rotate *src_1* right: *[0] ← [7]*	none
1101	ALU_OPC_RRC	Rotate *src_1* left thru carry: *[0] ← c*	c ←[7]
1110	ALU_OPC_PCSADD	signed {*src_2*,*src_1*} + signed (*src_3*)	none
1111	ALU_OPC_PCUADD	{*src_2*,*src_1*} + *src_3*	none

SUB - 0010: Subtraction is also addition. Only 2's complement of the 2nd operand and carry-in bit is used. Hence execution is similar to ADD. In coding the Verilog HDL arithmetic operator '-' is used to execute the subtraction.

MUL (Multiply)- 0011: The built-in Verilog HDL operator '*' is used to multiply the two 8-bit numbers and store the result in the 16 bit word

formed by concatenating 8-bit outputs *des_2* and *des_1*. If the result is greater than 8-bits in value, the overflow flag is set. This instruction clears the carry flag.

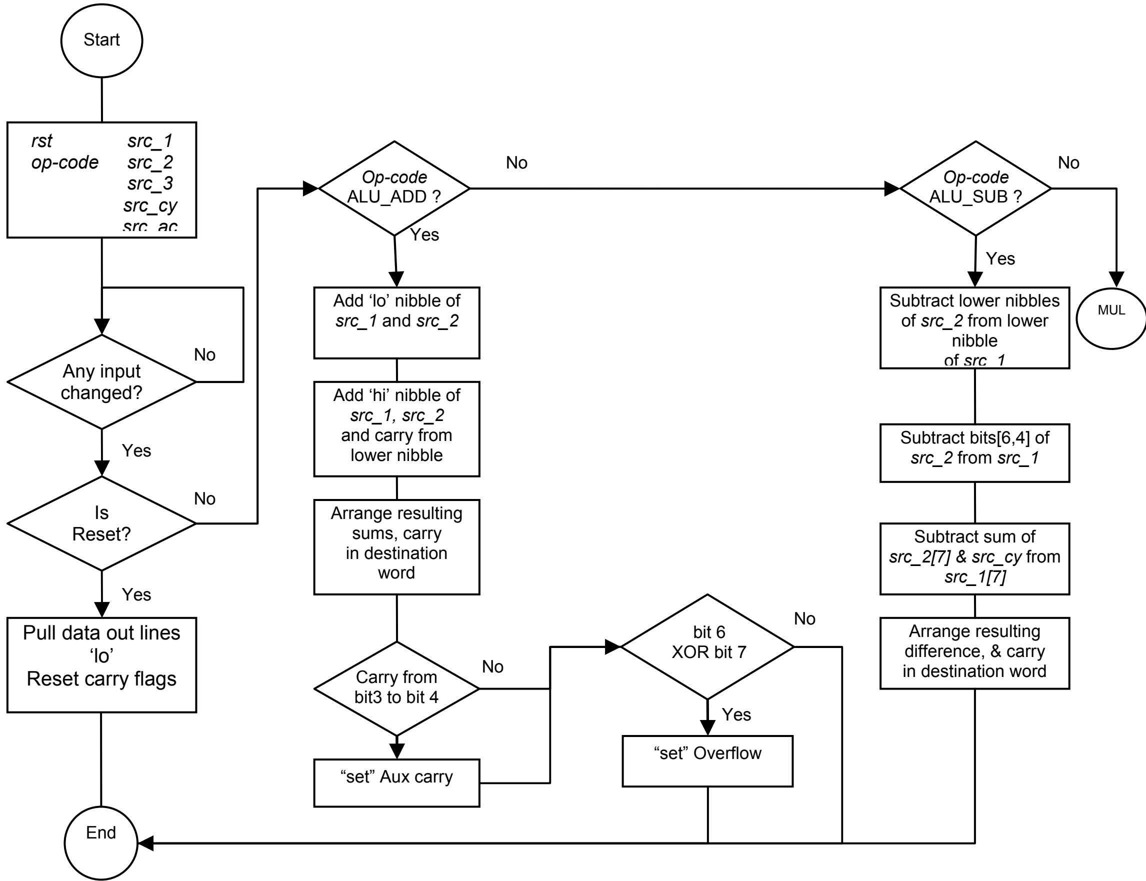


Figure 4.6-1: Flow Diagram for ALU Response to ADD or SUB Instruction

DIV (Divide) - 0100: The algorithm for division given in reference [19] is used when the divisor is less than the dividend. The division is achieved by initializing the remainder to zero, and repeating the following steps n times, where n is the number of bits -1.

- Shift dividend left one bit into remainder.

- Subtract divisor from remainder, placing the answer back to remainder.
- If most significant bit of remainder is 1, set quotient Q0 to 0, and add divisor back to remainder (restore remainder).
- Otherwise, set Q0 to 1.

The flow diagram in Figure 4.6-2 and 4.6-3 shows the value of quotient and remainder as well as the status of the overflow flag in all cases of the value of divisor with respect to the dividend.

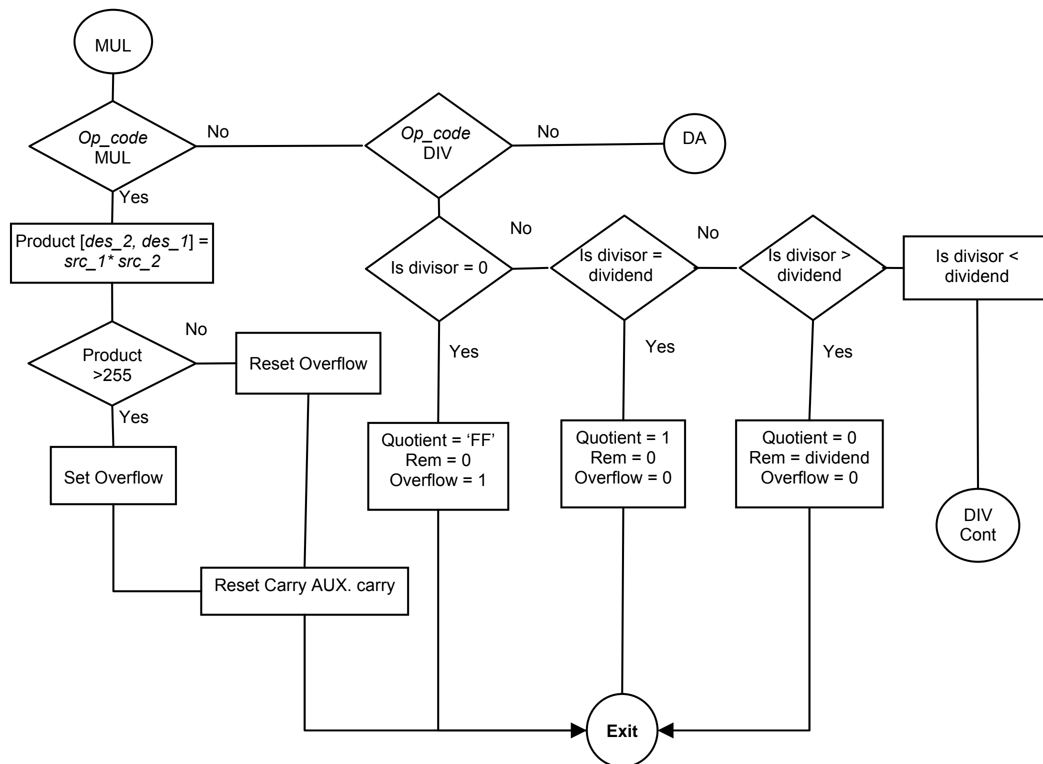


Figure 4.6-2: Flow Diagram for ALU Response to MUL or DIV Instruction

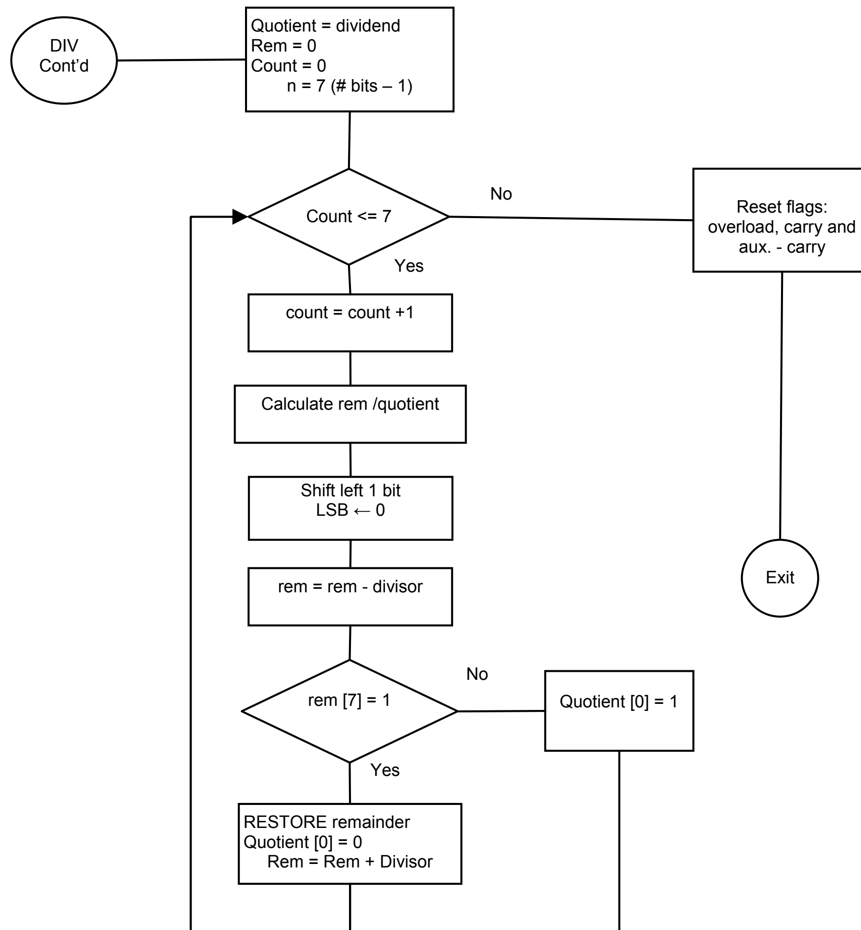


Figure 4.6-3: Flow Diagram for ALU response to DIV instruction (cont'd)

DA (Decimal Adjust) - 0101: Converts an 8-bit number to its BCD equivalent. This is accomplished by adding 06h if the lower nibble is either greater than 9 or there is a carry from bit 3 to bit 4 (auxiliary flag is set). Similarly if the higher nibble is greater than 9 or the carry flag is set, the number is adjusted by adding 60h to the number.

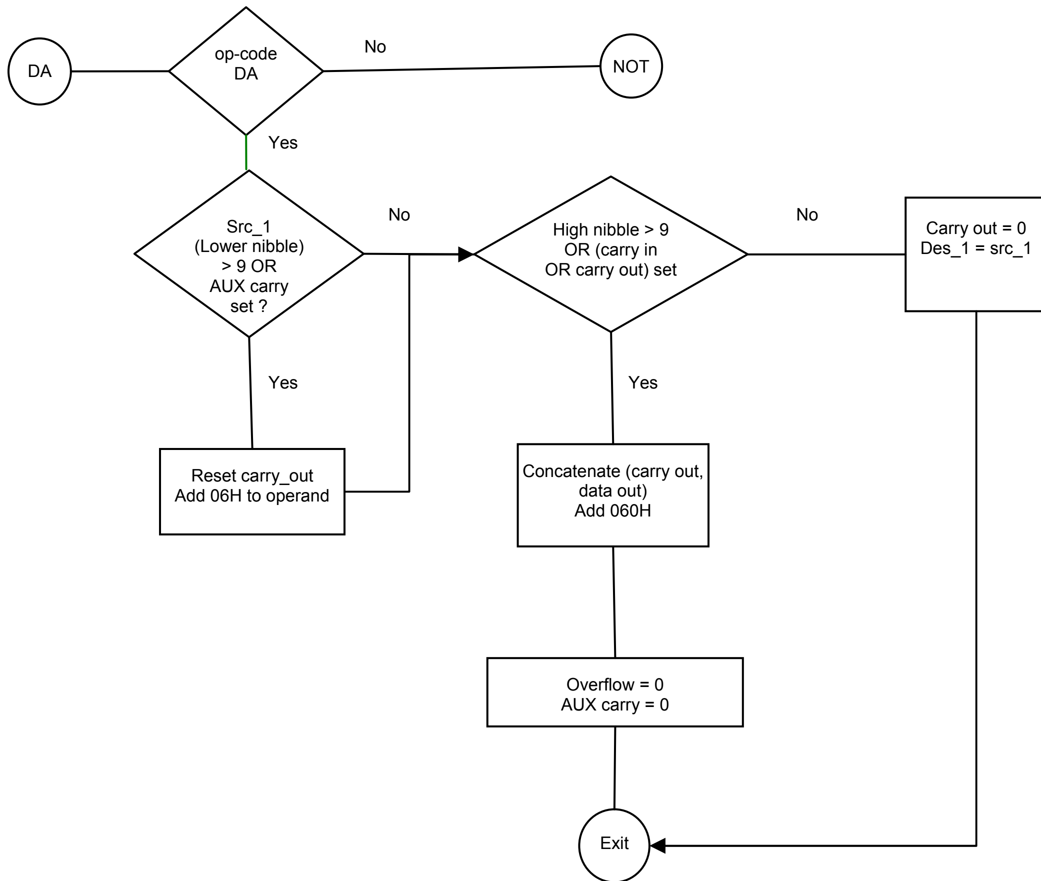


Figure 4.6-4: Flow Diagram for ALU response to DA instruction

PCSADD – 1110: This function of the ALU is called upon by the controller for implementing the Program Control Instructions with relative addressing. The high and lo bytes of the PC are transferred to the ALU, as src_2 and src_1 respectively. These are concatenated to form the 16 bit PC value, and src_3 containing the relative address (signed value) is added to it. The resulting value is passed back to the controller to update the PC.

PCUADD – 1111: This is used for updating the PC during sequential instruction fetching. Unsigned add is performed on the 16 bit PC value to increment the PC as each operand of the instruction is fetched.

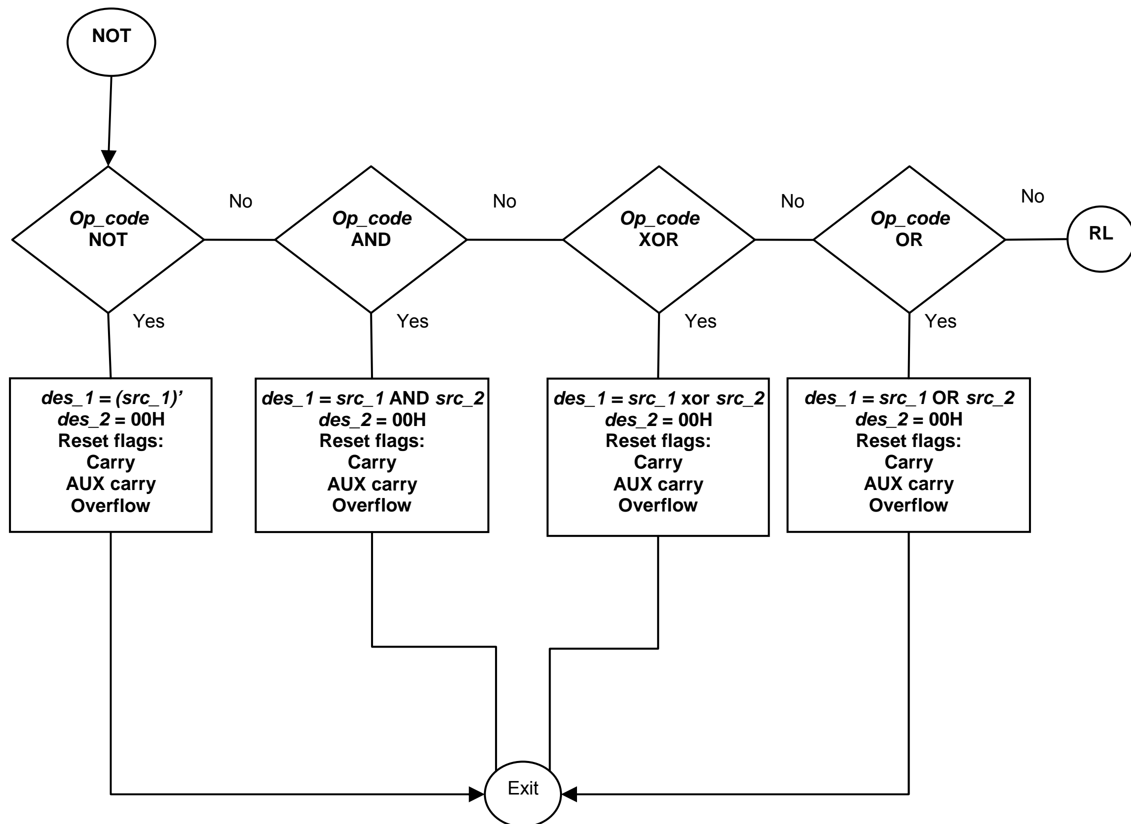


Figure 4.6-5: Flow Diagram for ALU Response to Logical Instructions

- Logical: The module utilizes the Verilog bitwise logical operators to implement the following operations on two 8-bit numbers.

  NOT - 0110

  AND - 0111

  XOR - 1000

  OR – 1001

The result is an 8-bit number. The carry, auxiliary carry and overflow flags are reset.

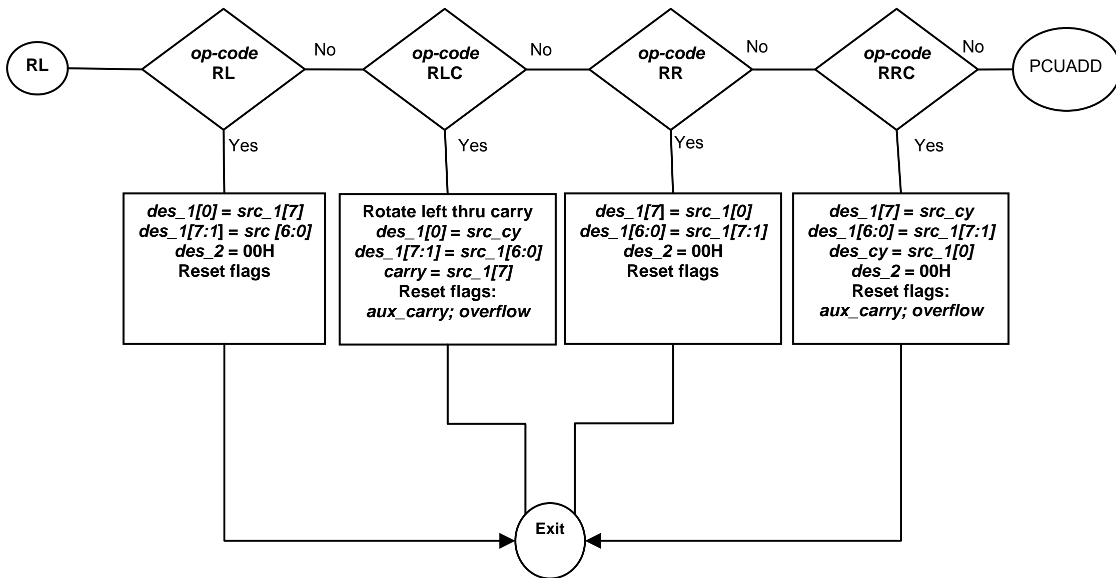


Figure 4.6-6: Flow Diagram for ALU Response to Shift Instructions

- Shift: These are single byte rotate operations. The auxiliary carry and overflow flags are reset.

  RL - 1010: Each bit of is shifted left, the MSB is shifted to LSB. Carry flag is reset.

  RLC – 1011: Each bit of is shifted left, the MSB is shifted to carry flag and the carry flag to LSB.

  RR - 1100: Each bit of is shifted right, the LSB is shifted to MSB. Carry flag is reset.

  RRC - 1101: Each bit of is shifted right, the LSB is shifted to the carry flag, and the carry flag to MSB.

In each instruction implementation, the ALU resets the unaffected flags and /or the 2nd output operand, as per specification of the respective instruction.

4.6.2. Simulation and Testing

Functional simulation for a 1us period, and sampling size of 10ns was carried out on the ALU module to test all 16 types of instructions implemented. The input signals and conditions simulated were as follows:

Input op-code signal: Incremented from 0 to 15 to test all values in this 4-bit signal. Each value held for a period of 50 ns.

Reset: Asserted for a period of 20 ns.

Source operands src_1 & src_2: Arbitrary data (refer to Table 4.6-2) imposed for different periods during the simulation cycle.

Source operand *src_3*: Data held at 00h.

Flags *src_ac*, and *src_cy*: status of the flags at input reset to 'low'.

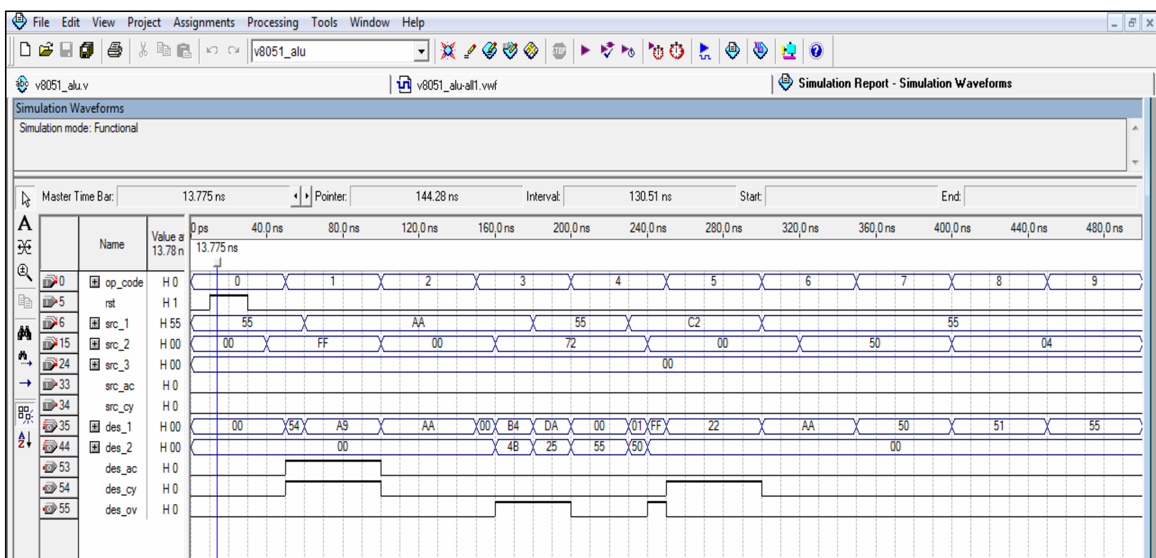


Figure 4.6-7: Simulation of 8051 ALU Operation for Op-Codes 0 to 9h

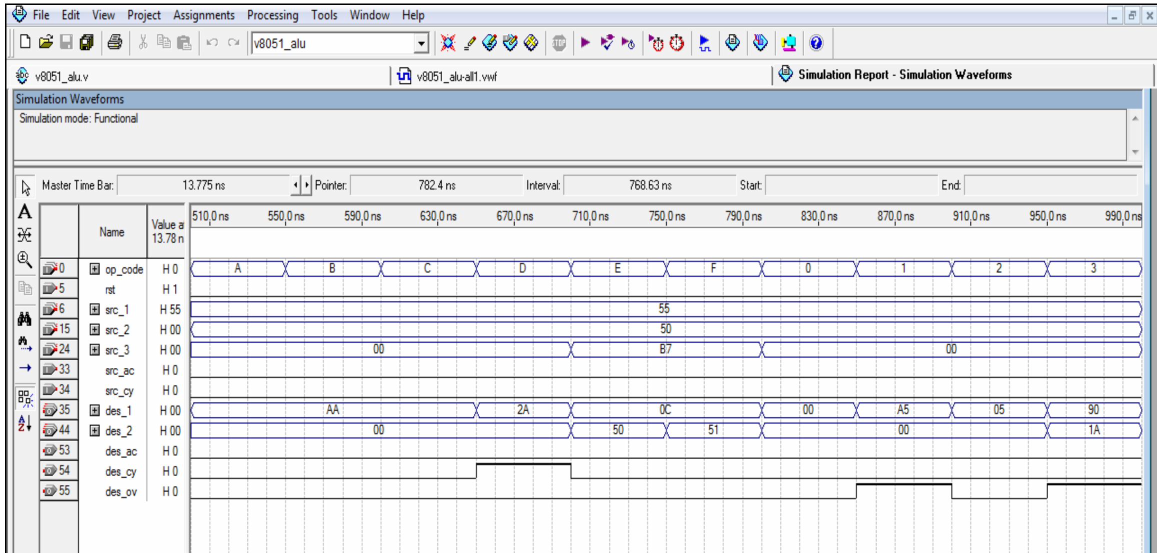


Figure 4.6-8: Simulation of 8051 ALU Operation for Op-Codes Ah to Fh

Table 4.6-2: ALU - Test Instructions and Expected Results

op-code		*src_1*	*src_2*	*src_3*	*src_ac*	*src-cy*	*src-ov*	*des_1*	*des_2*	*des_ac*	*des_cy*	*des_ov*
ADD	1	55	FF	00	0	0	0	54	00	1	1	0
		AA	FF	00	0	0	0	A9	00	1	1	0
		55	50	00	0	0	0	A5	00	1	1	0
SUB	2	AA	00	00	0	0	0	AA	00	0	0	0
		55	50	00	0	0	0	05	00	0	0	0
MUL	3	AA	00	00	0	0	0	00	00	0	0	0
		AA	72	00	0	0	0	B4	4B	0	0	1
		55	72	00	0	0	0	DA	25	0	0	1
DIV	4	55	72	00	0	0	0	00	55	0	0	0
		C2	72	00	0	0	0	01	50	0	0	0
		C2	00	00	0	0	0	FF	00	0	0	1
DA	5	C2	00	00	0	0	0	22	00	0	1	0
NOT	6	55	50	00	0	0	0	AA	00	0	0	0
AND	7	55	50	00	0	0	0	50	00	0	0	0
XOR	8	55	50	00	0	0	0	51	00	0	0	0
OR	9	55	50	00	0	0	0	55	00	0	0	0
RL	A	55	50	00	0	0	0	AA	00	0	0	0
RLC	B	55	50	00	0	0	0	AA	00	0	0	0
RR	C	55	50	00	0	0	0	AA	00	0	0	0
RRC	D	55	50	00	0	0	0	2A	00	0	0	0
PCSADD	E	55	50	B7	0	0	0	0C	50	0	0	0
PCUADD	F	55	50	B7	0	0	0	OC	51	0	0	0

4.6.3. Verification

The same simulation tests (i8051_alu-all1.vwf) were applied to the ALU module of Dalton model. The Quartus II v8.0 “Compare waveform file” command was used to compare simulation waveforms obtained from the Dalton module with those of the Verilog module. View of both set of waveforms is shown in Figure 4.6-9 and Figure 4.6-10. The waveforms from the simulation of both modules are overlapped (in red). Mismatched output signal durations are shown in black. Signal names and the instruction being executed in duration of the mismatch are shown in Table 4.6-3. As per specifications [12], these signals are not affected by the instruction; hence the mismatch does not affect the results. The Dalton model assigns the value ‘--------‘ (undefined) to the unaffected 8-bit data word, and ‘-‘ to the unaffected data bit. The Verilog model resets them to 00h and 0b respectively.

Table 4.6-3: Mismatched Output Signals (Dalton & Verilog ALU Modules) and their Impact on Result

Duration (of instruction)	Signals not matched	Impact on ALU operation
ADD/ SUB	*des_2*	Don’t care
MUL / DIV	*des_cy, des_ac*	Don’t care
Logical (NOT / AND / NOR/ OR)	*des_2, des_ac, des_cy, des_ov*	Don’t care
Shift (RL, RLC, RR, RRC)	*des_2, des_ac, des_cy, des_ov*	Don’t care
PCSADD/ PCUADD	*des_ac, des_cy, des_ov*	Don’t care

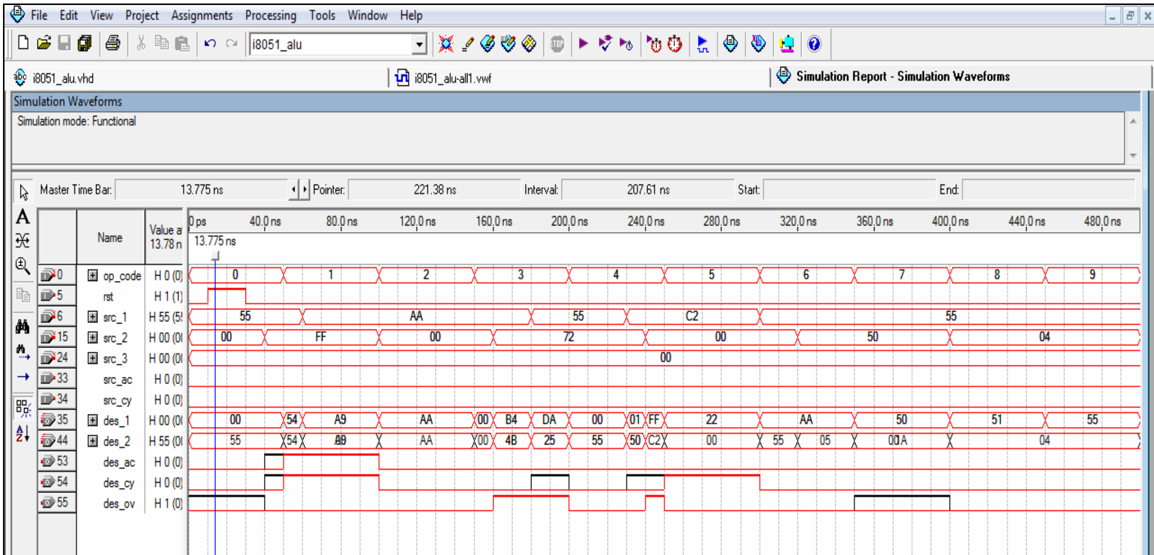


Figure 4.6-9: Comparison Report of Simulation of 8051 ALU Operation (Dalton Model with Verilog Model) for Op-Codes 0 to 9h

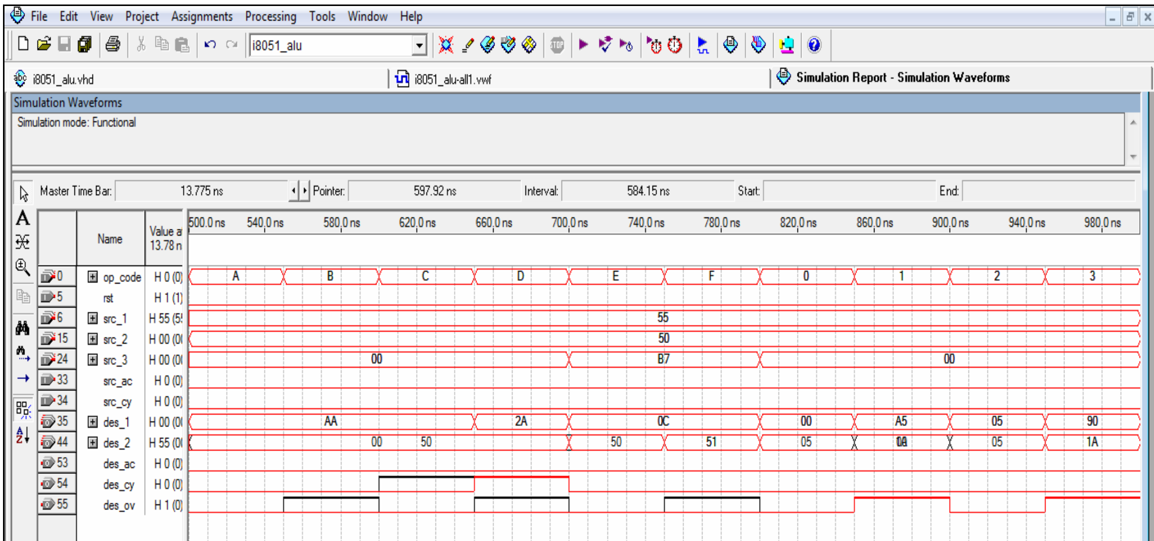


Figure 4.6-10: Comparison Report of Simulation of 8051 ALU Operation (Dalton Model with Verilog Model) for Op-Codes Ah to Fh

Hence, the simulation verifies that the results of ALU operation from both the Dalton and Verilog modules are the same.

CHAPTER 5.

8051 MODEL INTEGRATION

5.1. Detailed Design

To achieve the 8051 soft core the above designed and verified functional units had to be interfaced. Integration was therefore achieved by instantiation of the component modules, namely the arithmetic logic unit (ALU), controller (CTR), decoder (DEC), data memory (RAM) and program memory (ROM), with correct mapping of the port names in a top level encapsulating module v8051_model.

The v8051_model is driven by external clock *clkfast*. A slower clock *clk* which synchronizes all internal operations is derived by dividing *clkfast* by three. A 3-bit lpm_counter is used generate the internal clock. The external clock is used to meet the address hold- time requirements for the ROM module.

The v8051_model interacts with the external device/s through the ports p0 thru p3. These ports are addressed as part of the special registers at addresses 80h, 90h, A0h and B0h respectively of the 8051 microcontroller. The bi-directional feature of the microcontroller's I/O ports is achieved by p0_in thru p1_in serving as input data lines for write to, and p0_out thru p3_out serving as output data lines for read from the respective registers. The port addresses are bit- addressable, hence the soft core can be programmed to configure the

interface of the device, serial or parallel, to which these lines are connected as per application requirements.

5.2. Simulation and Testing

The functionality of the integrated system that is the 8051 soft core was tested by loading the ROM with Intel hex format files for test programs written in 8051 assembly code.

Test_led: This program residing in the ROM loads A-register and B-register with two arbitrary values as read from input ports 'p0_in' and 'p1_in' respectively. Register R5 assigned to count-down, is preset to value of 04h. It then rotates left the A-register through carry and the result from A-register is sent to output port0. The contents of A & B registers are exchanged. The new value is rotated right through carry and the result is sent to output port1. The A & B registers are again exchanged. R5 is decremented and the process of rotating and sending the values of the registers is repeated until R5 becomes zero. Then the process restarts with initial values. The assembly listing is shown in Figure 5.2-1

```
0000: INIT:     MOV A, P0        ; move ACC, dir
0002:           MOV B, P1        ; move dir, dir
0005:           MOV R5, #04H     ; move register, #immediate
0007: LOOP:     RLC A            ; shift left w/carry
0008:           MOV P0, A        ; move dir, ACC
000A:           XCH A, B         ; transfer
000C:           RRC A            ; shift right w/carry
000D:           MOV P1, A        ; move dir, ACC
000F:           XCH A, B         ;
0011:           DJNZ R5, LOOP    ; conditional jump
0013:           JMP INIT         ; short jump
```

Figure 5.2-1: 'Test_Led' Test Program in 8051 Assembly Code.

To first verify the controller function, the CPU_STATE and EXE_STATE were temporarily declared output ports and connected to monitoring points LEDG [2:0] and LEDR [6:0] of the DE2 FPGA development board, respectively. Each functional unit's outputs were monitored in several runs to verify the functionality of the controller and its data path, with respect to the designed FSM states.

Figure 5.2-2: Waveform Showing Input/ Output Ports with 8051 Soft Core Running 'Test_Led'

After all errors were removed the 8051 soft core was simulated with a test waveform file for an external clock of 50 MHz, and the output ports were monitored. The resulting 'port0_out' and 'port1_out' values verified that the soft core was running the ROM program as desired. Seen in Figure 5.2-2 is the A-register initial value of 12H when rotated left thru carry 4 times repeat output

values of 74h, E8h, D1h, & A2h, while Port1 repeats B-register value at 09h, 04h, 82h, & C1h.

Test_bcd: Another 8051 assembler program t_bcd_r2 was written to read input port0 if start (port2 bit 7) is set. The byte is read as a 2 digit hexadecimal value and converted to its binary coded decimal (BCD) equivalent. The 7 segment code for each of the digits is then output to 'port2_out' (hundred), 'port1_out' (tens), and 'port0_out' (units) so that it could be used to connect to a seven segment led display. The program then calls a delay subroutine. It then decreases the decimal value by one, sends the 7-segment code to the output ports, and again goes to the delay subroutine. It repeats until the value becomes zero. It then restarts, waiting for the next start signal to read the in port again.

This program loads the data memory with 7-segment code, and the bit position value for hex to decimal conversion. The program code fetches these values from data memory when required.

This program code is 188 bytes long, uses memory registers R0 thru R7, and registers A & B (SFRs) for addressing and data manipulation. 8051 instructions tested can be seen in the Program listing attached in the appendix.

The initialize file for v8051_rom_mem was defined as t_bcd_r1.hex. The ROM for the 8051 soft core was thus loaded with the test program t_bcd_r1.hex. Simulation was carried out with *clkfast* = 10ns, input port 'p0_in' value of 18H, and the start signals were inserted randomly. 2ms simulations of the I/O signals were generated in approx. 30sec. The waveforms in

Figure 5.2-3 show that the expected result of Table 5.2-1. 18H is converted to decimal 24. The 7-segment code for the decimal digits, as the number decreases is sent to the output ports.

This test program verifies execution of the following types of instructions:

Arithmetic- ADD, INC, DEC, DA

Logical – ANL, CLR, SWAP

Data Transfer – MOV (8/15 different addressing modes), PUSH, POP

Boolean – CLR C

Program control – DJNZ, CJNE, JNC, JNB, SJMP, LCALL, RET

Table 5.2-1: Expected Result for Test Program t_bcd_r2

p0_in	Decimal	port2_out (digit3)	port1_out (digit2)	port0_out (digit1)
18h	024	40h	24h	1Bh
	023	40h	24h	30h
	022	40h	24h	24h
	021	40h	24h	79h
	020	40h	24h	40h
	019	40h	79h	10h
	018	40h	79h	00h
	017	40h	79h	78h
	016	40h	79h	02h
	015	40h	79h	12h
	014	40h	79h	1Bh
	013	40h	79h	30h
	012	40h	79h	24h
	011	40h	79h	79h
	010	40h	79h	40h
	009	40h	40h	10h
	008	40h	40h	00h
	007	40h	40h	78h
	006	40h	40h	02h
	005	40h	40h	12h
	004	40h	40h	1Bh
	003	40h	40h	30h
	002	40h	40h	24h
	001	40h	40h	79h
	000	40h	40h	40h

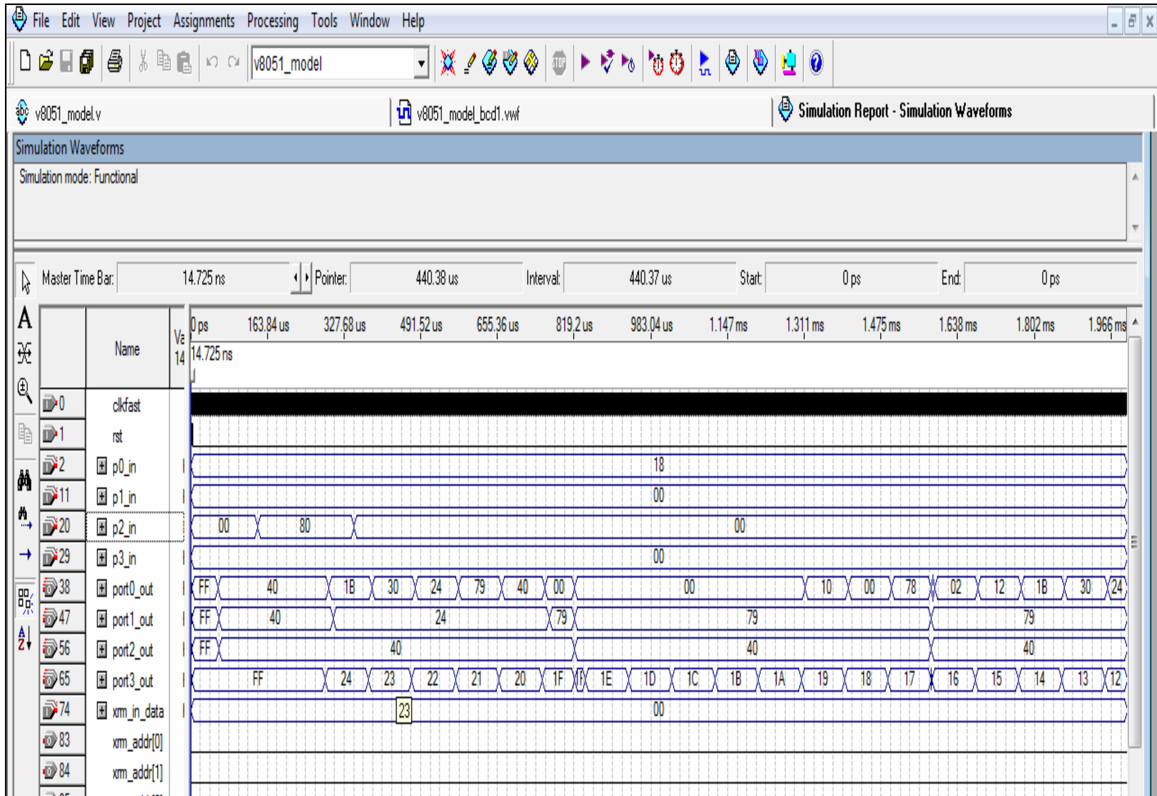


Figure 5.2-3: Simulation Results for Soft Core Running Test Program t_bcd_r2

The results of the test programs verify expected results on the output ports. Based on the results it is verified that the integration of the modules and hence the 8051 soft core is functional.

CHAPTER 6.

CONCLUSIONS

The design of the soft core has been verified by successful execution of two different test programs loaded in the ROM. 22 different instructions out of the set of one hundred and eleven 8051-instructions [20] were executed in the 188-byte long program code. The soft core is functionally operational as the controller can steer and synchronize the functions of each of the component modules, and also communicate with its external ports.

The compilation report shows input Clock *clkfast* has internal maximum frequency of approx. 40 MHz between the source register "v8051_ctr:U_CTR|alu_src_2 {3}" and the destination register "v8051_ctr:U_CTR|alu_src_2 {1}" (period= 25.033 ns). Longest register to register delay is 24.738 ns. The soft core utilizes 16% of the total logic elements, 4% of dedicated registers and 7% of total memory bits from the resources of the target Cyclone™ II - EP2C35F672C6 FPGA (Altera Corp., San Jose, CA, www.altera.com). The memory block corresponds to 4KB, the specification for a base 8051 ROM. This being 7% of available resources, memory for the soft core can be expanded. “The soft core implementation of a microcontroller saves space when there are unused resources, as these unused resources are found

to be unreachable or never used for the synthesizer and they are ripped out by the optimization tool" [21].

The 8051 soft core is just a basic model concluded as functional based on behavioral simulation. It requires further verification of the whole instruction set. Also optimizations on lower RTL level and physical structure are needed. The netlist can then be loaded on an FPGA to measure performance in real-time environment.

Future projects could develop this soft core by implementing:

Optimization of instruction execution times based on instruction type. It is designed for uniform execution cycle time for all instructions. Performance increase (at the architectural level) can be accomplished by eliminating the idle EXE_states [22].

Increasing instruction throughput of the 8051 soft core by pipelining the fetch-decode and execute cycles [23].

Interrupts. Provision exists in the Controller module, where CPU_STATE CS_1 is reserved for handling interrupts.

Internal Timers of 8051. The counter/ timer registers exist as special function registers.

Enhanced instruction set. The op-code width (6-bit) has provision for 8 new special instructions.

APPENDIX

ASSEMBLER LISTING OF TEST PROGRAM T_BCD_R2

```
; Read P0= XXF, Convert to decimal Result =nnn,
; Display on HEX0, HEX1, and HEX2 respectively,
; Decrease: result =result-1 until result = 0.
; Fetch new P0 & continue
     ; Generate Lookup table for 7segment display
               MOV R1, #050H
               MOV @R1,#040H         ; '0'
               INC R1
               MOV @R1,#079H         ;'1'
               INC R1
               MOV @R1,#024H         ;'2'
               INC R1
               MOV @R1,#030H         ;'3'
               INC R1
               MOV @R1,#01BH         ;'4'
               INC R1
               MOV @R, #012H         ;'5'
               INC R1
               MOV @R1, #02H         ;'6'
```

```
        INC R1
        MOV @R1, #078H      ;'7'
        INC R1
        MOV @R1, #00H       ;'8'
        INC R1
        MOV @R1, #010H      ;'9'
; Generate lookup table for decimal equivalent bit position
        MOV R1, #70H
        MOV @R1, #01H       ; 2^0 =1
        INC R1
        MOV @R1, #02H       ; 2^1 =2
        INC R1
        MOV @R1, #04H       ; 2^2 =4
        INC R1
        MOV @R1, #08H       ; 2^3 =8
        INC R1
        MOV @R1, #016H      ; 2^4 =16
        INC R1
        MOV @R1, #032H      ; 2^5 =32
        INC R1
        MOV @R1, #064H      ; 2^6 =64
        INC R1
```

```
            MOV @R1, #028H          ; 2^7 =128
            INC R1
            MOV @R1, #01H
INIT:       MOV R0, #050H
            MOV A,@R0               ; Display '000'
            MOV P0, A               ; hex0
            MOV P1, A               ; hex1
            MOV P2, A               ; hex2
; Initialize result
            CLR A
            MOV R1, #7AH            ; Address for storing BIN value
            MOV @R1, A              ; Reset to 00H
            INC R1
            MOV @R1, A              ; RESET carry-over values
            DEC R1                  ; R1 = Address result
            MOV R0, #070H           ; Start of Dec LUT
CHECK:      JNB P2.7, CHECK
READ:       MOV R6, P0              ; Read P0_IN
            MOV B, R6               ; Save I/P IN B
            MOV R3, B               ; Temporary register
            MOV A, @R1
            MOV R5, A
```

```
            MOV R4, #07H          ; Exp. counter
            MOV R2, #08H          ; Bit Position
            MOV 05AH, #00H        ; Reset Bin result
LOOP1:      CLR C
            MOV A, R3
            RLC A                 ; C = A[7]
            MOV R3, A             ; Save remaining Bits
            CLR A
            JNC NXTDIG            ; Jump IF Bit A[7] =0
; Lookup decimal equivalent. Add, Decimal Adjust, Update sum
DECIML:     MOV R1, #07BH         ; Address for result carry
            MOV A, @R1            ; previous carry value
            CJNE R4, #07H, CONT
            INC A                 ; for n=8, BIN=128, 3rd dig=1
            MOV @R1, A            ; Store 3rd Dec. Dig
            MOV R1, #7AH          ; Address for storing BIN
            MOV R7, A
CONT:       MOV R0, #70H          ; Start of decimal LUT
            MOV A, R4             ; locate decimal equivalent
            ADD A, R0             ; start address of LUT
            MOV R0, A             ; Address of decimal value
            MOV A, @R0            ; decimal value
```

```
            CLR C
            ADD A, R5           ; Update Sum
            DA A
            MOV R5, A           ; Store new Sum
DECML1:     JNC NXTDIG
            MOV R1, #07BH       ; Address for result carry
            MOV A, @R1          ; previous carry value
            INC A               ; increment 3rd dig.
            MOV @R1, A
            MOV R7, A
NXTDIG:     DEC R4              ; Exp Counter
            DJNZ R2, LOOP1      ; Bit Counter
            INC R7              ; Counter for display
            DEC R1              ; Address of result
            MOV A, R5
            MOV @R1, A          ; Store result
DISPLAY:     NOP
DHEX0:      MOV R0, #050H       ; Start of BCD LUT
            MOV A, R5           ; bin value
            MOV P3, A           ; Decimal Tens/Units to P3
UNITS:      ANL A, #0FH         ; 1st dig
            CJNE A, #0FH, OKAY
```

```
            MOV A, R5           ; Decimal > 9
            ANL A, #0F9H        ; Units = 9
            MOV R5, A           ; Update
            SJMP UNITS
OKAY:       ADD A, R0
            MOV R1, A
            MOV A, @R1          ; fetch 7-seg
            MOV P0, A           ; P0_out
            MOV A, R5
            SWAP A
DHEX1:      ANL A, #0FH         ; 2nd dig
            ADD A, R0
            MOV R1, A
            MOV A, @R1          ; fetch 7-seg
            MOV P1, A           ; P1_out
HEX2:       MOV A, R7
            DEC A               ; VALUE = 3rd dig
            ADD A, R0
            MOV R1, A
            MOV A, @R1          ; Fetch 7-seg code
            MOV P2, A           ; P2_out
DELAY:      MOV R0, #10
```

```
            LCALL DEL
LESS:       DJNZ R5, DISPLAY
            MOV R5, #99H
            DJNZ R7, DISPLAY
            SJMP READ
DEL:        PUSH 0E0H              ; Delay Subroutine
DEL1:       MOV R1, #02H
DEL2:       DJNZ R1, DEL2
            DJNZ R0, DEL1
            POP 0E0H
            RET
```

REFERENCES

[1] *Information Technology Encyclopedia and Learning Center*, http://whatis.techtarget.com/ definition/0,,sid9_gci759036,00.html, accessed Sep. 2008.

[2] "Embedded Controller Applications", http://en.wikipedia.org/wiki/Embedded_system, accessed Sep. 2008

[3] M. Bashiri, S.G.M. Miremadi and M. Fazeli, "A checkpointing technique for rollback error recovery in embedded systems," in *ICM '06 International Conference on Microelectronics,* 16-19 Dec. 2006, pp 174 – 177. Digital Object Identifier: 10.1109/ICM.2006.373295

[4] Lu Yi and N. Bergmann, 2005. "Dynamic loading of peripherals on reconfigurable system-on-chip," in *Proceedings of IEEE International Conference on Field-Programmable Technology,* 11-14 Dec. 2005, pp 279 – 280. Digital Object Identifier: 10.1109/FPT.2005.1568560

[5] D.F. Wolf, J.A. Holanda, V. Bonato, R. Peron, and E. Marques, "An FPGA-based mobile robot controller." in *Proceedings of 3rd Southern Conference on Programmable Logic*, 28-26 Feb. 2007, pp 119 – 124. Digital Object Identifier: 10.1109/SPL.2007.371734

[6] A.J. Salim, "Integration of 8051 with DSP in Xilinx FPGA," in *Proceedings of IEEE International Conference on Semiconductor Electronics*, 29 Oct. –

1 Dec. 2006, pp 562-566.

Digital Object Identifier: 10.1109/SMELEC.2006.380694

[7] Yue-li Hu, and Ke-xin Zhang, “Design of on-chip debug module based on MCU,” in *International Symposium on High Density packaging and Microsystem Integration*, 26-28 Jun. 2007, pp 1 – 4. Digital Object Identifier: 10.1109/HDP.2007.4283633

[8] Dalton Project, University of California, Riverside, CA, “Synthesizable VHDL model of 8051,” http://www.cs.ucr.edu/~dalton/i8051/i8051syn0 accessed Jul. 2008

[9] Mar´ılia Lima, Andr´e Alves Aziz, Diogo Lira, Patr´ıcia Schwambach,Vitor and Edna Barros, “ipPROCESS: using a process to teach IP-core development,” in *Proceedings of IEEE International Conference on Micro electronics Systems Education*, 12-14 Jun. 2005, pp 27 – 28. Digital Object Identifier 10.1109/MSE.2005.38

[10] “EdSim51- 8051 Simulator for Teachers and Students”, o*n-line*. http://www.edsim51.com/ accessed 20 Apr. 2009.

[11] Frank Vahid, and Tony Givargis, *Embedded System Design: A Unified Hardware/ Software Introduction,* Hoboken: John Wiley and Sons, 2002.

[12] Zdravko Karakehayov, Knud Smed Christensen, and Ole Winther, *Embedded Systems Design with 8051 Microcontrollers – Hardware and Software,* New York: Marcel Dekker, 1999.

[13] Clifford E. Cummings, “The fundamentals of efficient synthesizable finite state machine design using NC-Verilog and build gates,” *Proceedings of International Cadence Usergroup Conference,* 16-18 Sep. 2002, http://www.sunburst-design.com/papers/CummingsICU2002_FSMFundamentals.pdf accessed Mar. 2009.

[14] Altera Corporation, “ROM Functions—Inferring ALTSYNCRAM and LPM_ROM Megafunctions from HDL Code,” *Quartus II Handbook Design and Synthesis* Ver. 9.0, Vol. 1, 2009.

[15] Milan Verle, *8051 Microcontroller Architecture: In Architecture and Programming of 8051 Microcontrollers*. Belgrade: mikroElektronika, http://www,mikroe.com/en/books/8051book/ch2/ accessed Apr. 3, 2008.

[16] Zorian Yervant, Erik Jan Marinissen and Sujit Dey, “Testing embedded-core based system chips,” *Computer,* vol. 32, no. 6, pp 52-60, June 1999.

[17] Michael D. Ciletti, *Advanced Digital Design with VERILOG HDL,* 1st ed., Upper Saddle River: Prentice- Hall, 2003.

[18] Deepak Jain, 2004. “Analysis and VHDL modeling of 8051-microcontroller using determinant-functional object modeling (D-FOM) approach,” posted 10 Jul. 2004, http://www.codeproject.com/KB/architecture/DFOM-MCU.aspx., accessed 12 Jul. 2008.

[19] Malek Miroslaw, “ALU (3) - division algorithms,” *Lecture 12:Summer semester* 2002, Humbolt-Universitat Zu Berlin, www.informatic.hu_berlin.de/rok/ca, accessed16 Dec. 2008.

[20] Data-sheet, MC5-51: 8-bit Control Oriented Microcomputers, Intel Corp., http://www.ic-on-line.cn/iol.8051AH/pdfview /186937.html, accessed Dec. 2008.

[21] Daniel Francisco Gómez Prado, “Embedded Microcontrollers and FPGAs Soft-cores,” translated automatically from *ELECTRÓNICA UNMSM Journals* no. 18, Amherst: UNMSM, Dec. 2006. http://sisbib.unmsm.edu.pe/BibVirtualData/publicaciones/electronica/n18_2006/a02.pdf, accessed Apr. 2009.

[22] M. Schutti, M. Pfaff and R. Hagelauer, “VHDL design of embedded processor cores: the industry-standard microcontroller 8051 and 68HC11,” in *Eleventh Annual IEEE International ASIC Conference Proceedings,* 13-16 Sep. 1998, pp 265 – 269. Digital Object Identifier 10.1109/ASIC.1998.722990

[23] Chang-Jiu Chen, Wei-Min Cheng, Ruei-Fu Tsai, Hung-Yue Tsai and Tuan-Chieh Wang, “A pipelined asynchronous 8051 soft-core implemented with Balsa,” in *IEEE Asia Pacific Conference on Circuits and Systems Proceedings,* 30 Nov. - 3 Dec. 2008, pp 976 – 979. Digital Object Identifier: 10.1109/APCCAS.2008.4746187