IMPLEMENTATION OF AN 8-BIT MICROCONTROLLER WITH SYSTEM C

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

BY

LOKMAN KESEN

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

NOVEMBER 2004

Approval of the Graduate School of Natural and Applied Sciences

Prof. Dr. Canan Özgen
Director

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

Prof. Dr. İsmet Erkmen
Head of Department

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

Prof. Dr. Murat Aşkar
Supervisor

Examining Committee Members

Prof. Dr. Tayfun Akın	(METU, EE)	
Prof. Dr. Murat Aşkar	(METU, EE)	
Yrd. Doç. Cüneyt Bazlamaçcı	(METU, EE)	
Dr. Ece Güran	(METU, EE)	
M.Sc. Ali Yazıcı	(ASELSAN)	

PLAGIARISM

Name, Last name	: Lokman Kesen
Signature	:

ABSTRACT

IMPLEMENTATION OF AN 8-BIT MICROCONTROLLER WITH SYSTEM C

Kesen, Lokman

M.Sc., Department of Electrical and Electronics Engineering

Supervisor: Prof. Dr. Murat Aşkar

November 2004, 122 pages

In this thesis, an 8-bit microcontroller, 8051 core, is implemented using SystemC programming language. SystemC is a new generation co-design language which is capable of both programming software and describing hardware parts of a complete system. The benefit of this design environment appears while developing a System-on-Chip (SoC), that is a system consisting both custom hardware parts and embedded software parts. SystemC is not a completely new language, but based on C++ with some additional class libraries and extensions to handle hardware related concepts such as signals, multi-valued logic, clock and delay elements. 8051 is an 8 bit microcontroller which is widely used in industry for many years. The 8051 core is still being used as the main controller in today's highly complex chips, such as communication and bus controllers. During the development cycles of a System-on-Chip, instead of using separate design environments for hardware and software parts, the usage of a unified co-design environment provides a better design and simulation methodology which also decreases the number of iterations at hardware software integration. In this work, an 8-bit 8051 microcontroller core and external memory modules are developed using SystemC that can be re-used in future designs to achieve more complex System-on-Chip's. During the development of the 8051 core, simulation results are analyzed at each step to verify the design from the very beginning of the work, which makes the design processes more structured and controlled and faster as a result.

Keywords: SystemC, 8051, System-on-Chip, Microcontroller, Hardware-Software Co-design

ÖZ

8-BİT MİKRO DENETLEYECİNİN SYSTEM C İLE GERÇEKLEŞTİRİLMESİ

Kesen, Lokman

Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü

Tez yöneticisi: Prof. Dr. Murat Aşkar

Kasım 2004, 122 sayfa

Bu tezde SystemC programlama dili kullanılarak 8051 8-bit mikro denetleyici çekirdeğinin tasarımı gerçekleştirilmiştir. SystemC, bütün bir sistemin hem donanımı tanımlamaya hem yazılımını programlamaya yetkin yeni nesil bir tümleşik tasarım dilidir. Bu tasarım ortamının faydaları, özel donanım modülleri ve tümleşik yazılımlardan oluşan “Tek Yongada Sistem”lerin (SoC, System-on-Chip) geliştirilmesinde ortaya çıkmaktadır. SystemC tamamen yeni bir dil değildir, aksine, C++ programlama dilini temel alır ve çok seviyeli mantık devreleri, saat sinyalleri ve gecikme öğeleri gibi donanıma ilişkin konuları desteklemek üzere bir takım nesne kütüphaneleri ve eklentiler içermektedir. 8051 mikro denetleyicisi 8 bit tabanlıdır ve uzun yıllardır sanayide yaygın ölçüde kullanılmaktadır. 8051 çekirdeği, veri yolu denetleyicileri ve iletişim denetleyicileri gibi günümüzün karmaşık yongalarında halen temel denetleyici olarak kullanılmaktadır. Tek Yongada Sistem’lerin geliştirilme sürecinde, donanım ve yazılım modülleri için ayrı tasarım ortamları kullanmak yerine, tümleşik bir tasarım ortamı kullanmak daha iyi bir tasarım ortamı sağladığı gibi donanım ve yazılım bütünleme adımlarının sayısında da önemli kazançlar sağlamaktadır. Bu çalışmada, 8 bit mikro denetleyici olan 8051 çekirdeği ve çevresel bellek elemanları, ileride daha karmaşık Yonga-Sistem'lerin tasarımında yeniden kullanılabilecek şekilde SystemC kullanılarak geliştirilmiştir. 8051 çekirdeğinin geliştirme sürecinde, tasarımı en temelinden itibaren her adımda doğrulamak üzere simülasyon sonuçları incelenmiş, böylece süreç daha denetimli, yapısal ve sonuç olarak hızlı olmuştur.

Anahtar Kelimeler: SystemC, 8051, Mikro Denetleyici, Donanım-Yazılım Bütünleşik Tasarımı

To My Family

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to my advisor, Prof. Dr. Murat Aşkar for his guidance, encouragement and support in every stage of this research.

I am also grateful to my colleagues for their encouragement and support.

I would like to express my deep gratitude to all who have encouraged and helped me at the different stages of this work.

And finally I am grateful to my wife for everything.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

INTRODUCTION

Integrated circuits (ICs) constitute the heart of most of the electronic devices available today. Typical systems use many ICs to respond to the increasing demand for high performance and low cost. Microprocessors and Digital Signal Processors (DSPs), which are two most common types of software programmable ICs, are used for implementing the more complex tasks such as user interfaces and signal processing functions

As an IC replicates the functionality of a circuit consisting of at least hundreds of thousands of transistors, designing the chip is not an easy task. Design tools are required at many levels of the design process to simulate the circuit, verify the constraints, and create the final layout and post layout simulations. Such software tools are commonly called Electronic Design Automation (EDA) tools.

Usually, the functionality of a chip is implemented in hardware. In such a scenario, the chip can be used only for the intended application and is termed an Application Specific Integrated Circuit (ASIC). Such chips cannot be re-programmed to perform for another function. Most of the proprietary chips fall in this category. Field Programmable Gate Arrays (FPGAs) can be re-configured, and working scenarios can be altered. On the other hand, Microprocessors (μP), Microcontrollers (μC) and Digital Signal Processors (DSP) can be software programmable either at manufacture time or at run time. Today, a typical circuit consists of one or more processors, some discrete components, and may have DSPs and s ASICs on a circuit board. Even though this methodology is sufficient for reasonably complex systems, a change in design methodology is required, as systems get more and more complex.

Another design method is to incorporate software as well as IP (Intellectual Proprietary) cores, which result in the term System-on-Chip (SoC). In other words, a SoC consists of both hardware and software modules to achieve the functionality of the system. A typical example of a SoC design is shown in Figure 1.1. A SoC is usually much more complex than an ASIC and is the current trend in chip design.

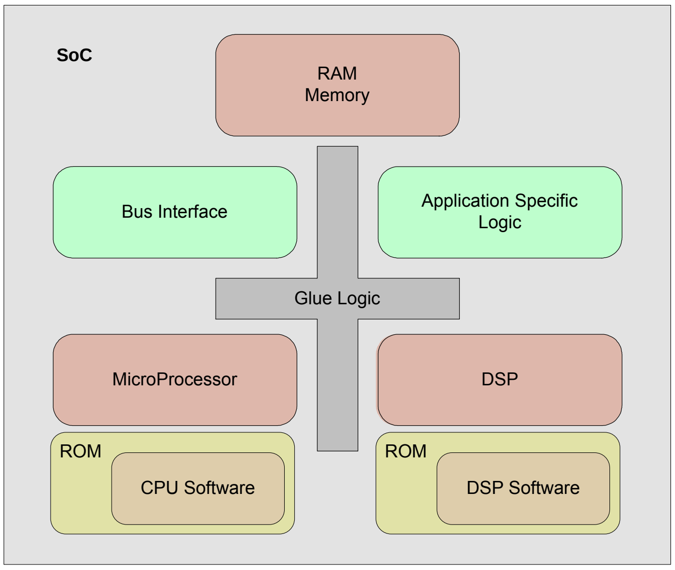


Figure 0.1 A Typical SoC (System-On-Chip).

In SoC design, a given specification must be broken down to two main categories; hardware and software. This problem is called the hardware-software partitioning problem and is an important issue that can affect the performance of the overall system.

The goal is to partition the components of the system to either hardware or software so that the resulting system provides optimum performance. Usually, the partitioning is done based on the evaluation of a cost function whose exact form depends on the design constraints [1] As an example, execution time, data memory, frequency of activation, etc. for a unit can be included in its cost function. This process is not standardized and a typical analysis can yield multiple partitions, which makes the design process an iterative one where each partition is selected and evaluated one by one until the design specifications are met at the lowest cost. Also, it is difficult to automate this process completely, most of the time; the user interaction is required. This problem is an active research area, and success has been achieved in some cases. Another closely related problem issue is testing these heterogeneous components. Although it is possible to test hardware and software independently, a complete test, which includes both, will be preferred more. The reason is simple, as the final product will be a compound of software and hardware, it is better to integrate them as early as possible. So, a co-simulation environment will be beneficiary for simultaneous hardware-software validation.

Hardware Description Languages (HDLs) are an important set of EDA tools. These are high level programming languages that are designed to model the behavior of hardware. Two of the most popular HDLs are the Very High Speed Integrated Circuit Hardware Description Language (VHDL) and Verilog. HDLs have semantics that model typical hardware features like concurrency, delay, clock, ports and signals. Once a model is coded in an HDL, it can be simulated to verify the intended functionality and then synthesized to hardware. This approach is suited for creating chips that contain only hardware parts, it is inappropriate for SoC designs which contain both hardware and embedded software. This is because; current HDL design and simulation environments do not support hardware-software co-design and co-simulation. In other words, although HDLs are very powerful in creating hardware chips they do not have sufficient structures for systems with embedded software.

Software programming languages such as C and C++ are used for embedded software inside the SoC chip, which is also called firmware. These languages are originally designed for sequential programming of microprocessors and not suitable for describing the hardware behaviors. The reason is, they are having the lack of concurrent structures, clock signals and delays. Opposite to the case in HDLs, software programming languages are very efficient in sequential programming but they have short comings for the design of concurrent hardware structures.

Because of the limitations of both Hardware Description Languages and Software Programming Languages for designing SoCs, System Level Modeling Languages (SLMLs) are used for this purpose. One of the recent developments in this field is the new modeling language named SystemC [2] [3]. This language is based on C++ and has specific constructs to model hardware-related concepts. Details of SystemC are covered in the next chapters.

Typical systems can be represented at higher levels of abstraction than the ones handled by HDLs. Languages such as C and C++ are commonly used to express highly abstract systems. Because of higher simulation speed, functional verification is always preferred at the highest level of abstraction possible[8]. Verification of C/C++ models is much faster than those written in HDLs.

Once the chip is designed and verified in one of HDLs, the next step is translating the written code to the real hardware components. There are tools available in the market to perform this task and they are generally termed as synthesis tools. Synthesis tools use predefined library components and map algorithmic code to component instantiations. To enable this translation, the algorithmic code must be written in a hardly restricted format specified by the tool vendor.

A new development in the synthesis market is the introduction of SystemC compiler tools such as the Synopsys CoCentric (CCSC), Cadence NC Simulator, Mentor's Seamless C-Bridge [4][8][15][16]. This tool reads code written in SystemC and translates the design to hardware, using library components. The combination of SystemC and CCSC would be a very powerful methodology for designing complex chips in a short time window.

The goal of this thesis is to study the effectiveness of SystemC in modeling real life systems. While it is easy to test SystemC on simple designs, real-life designs are often complicated, and the performance of the language in modeling these designs decides whether it is acceptable by the industry or not. For demonstrating the capabilities of SystemC, design of an 8 bit general purpose microprocessor, 8051 core is used as a case study. The reason for selecting 8051 core is the popularity of the chip as well as its highly verified structure.

Chapter 1 is the introduction to this thesis, and provides a summary of the background of this thesis. The task description and the thesis organization are also provided. Chapter 2 describes the SystemC language and the synthesis process. The features of the language are explained. Alternatives to SystemC are considered, and the reasons for selecting SystemC for this work are summarized. A description of the 8051 Microcontroller core is given in chapter 3. The design structure of 8051 microcontroller core, which is used as a case study in this thesis, is explained in Chapter 4. The implementation of the 8051 core with SystemC language is also explained in this chapter. Chapter 5 explains how the verification issues handled in the design and compares the signal traces of the designed 8051 core to the original one. And finally chapter 6 states the conclusions drawn from the work done in this thesis and suggests possible directions for future research.

THE SYSTEMC DESIGN ENVIRONMENT

SystemC is a modeling language based on C++. It is a set of class libraries in C++ that allow users to model hardware related concepts like concurrency, timing etc. In other words, SystemC is an extension of C++ with classes to add the desired functions, using the object-oriented programming methods. Since this language is basically C++, any ANSI C++ compiler can be used to run SystemC models. Because of these features, the language offers the following advantages;

- *Executable Specification:* A model written in SystemC can be compiled and made be executable. This feature implies that the executable specification can be shared among the members of a team without the need to share the source code or any associated script file.
- *Faster simulation*: Since SystemC depends on the underlying C++ framework for simulation, the simulation speed is high compared to either VHDL or Verilog. The speedup could be higher with the use of commercial tools.
- *Higher abstraction levels*: Compared to hardware description languages, C++ has the ability to model highly abstract concepts in an elegant fashion. This feature is inherent in SystemC.
- *Implementation independence*: A model specified in a hardware description language is usually targeted for a hardware implementation. In contrast, a SystemC model does not specify a particular implementation. It can be implemented either in hardware or in software, using a general-purpose processor or a DSP. This feature is very useful in hardware-software co-design, as explained before.

2.1 Open SystemC Initiative Organization (OSCI)

Another important feature of SystemC is that it is provided as an open source distribution, where users can download the source code of the SystemC language. Users can submit their bug fixes or add their own features to the language. This ensures that the language evolves rapidly through the collective effort of the design community. SystemC is entirely based on C++, and the source code for the SystemC reference simulator can be freely downloaded from SystemC official web page under an Open Community Licensing agreement.

There are other languages that are similar to SystemC in that they allow modeling hardware at a high level of abstraction.

SpecC is a system specification description language based on C. It is developed as a language extension to ANSI C, with special constructs to model hardware. A model can be described at a high level of abstraction in SpecC. It can then be simulated using the SpecC reference compiler (SCRC). SCRC consists of an open source pre-processor that converts the SpecC specific constructs to equivalent ANSI C code, which is then compiled and simulated. Then, it is possible to get executable specifications of models. The main difference of SpecC from SystemC is that the modeling domain of SpecC covers higher levels of abstraction than that done using SystemC.

Cynlib is another C++ based language for hardware modeling. The focus of this language is hardware design, and not system level specification.

Superlog is a system level description language based on Verilog and C. It is a superset of Verilog with C language-like constructs. Code written in Superlog cannot be compiled to executable format, and requires the availability of specific simulator tools.

Among other System Level Modeling Languages, SystemC is the best supported one. There are many companies having specific SystemC products in design, simulation, verification and language conversion fields. For the whole design cycle for a real life project, these tools are also needed as well as basic design language specification. These are the fundamental reasons why SystemC has been chosen as the main issue of this thesis.

2.2 Modeling with SystemC

The features of SystemC allow designer to model the system at a higher level of abstraction. The modeling process is an iterative one, with data and control refinement occurring at each step. Data refinement involves improvements in the way data types are modeled, and control refinement refers to the evolution of the control protocols in the model. The refinements can be evaluated by performing simulation at each design step. Once the refined model is finalized, it can be partitioned and synthesized.

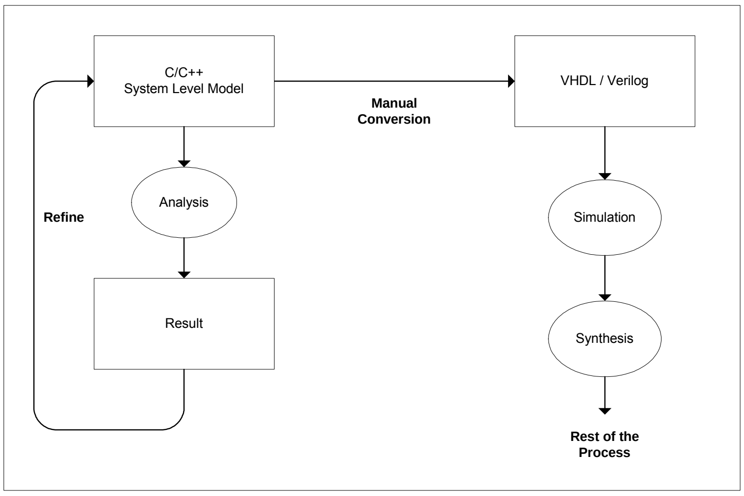


Figure 0.1 Conventional Design Method.

In conventional design method, which is shown in Figure 2.1, the first step of the project is the validation of the main idea in the project. This process is also called as “proof of concept”. In most cases, the designer prepares the models for the modules used in the system. The best way is to use a programming language like C/C++ or Pascal to simulate the behavior of the modules. Another alternative is to prepare the prototype of the module under development. In both cases, after having the results from the analysis step, the designer makes necessary refinements to the model. When this process results with satisfied outputs, designer manually converts the proven model to the design environment programming language i.e. Verilog or VHDL. Then simulation and synthesis steps are activated to have the final design.

On the other hand, in proposed design method, validation of the new concept and design process are combined in a single task. The SystemC design flow is shown in Figure 2.2. In this method the designer starts preparing a SystemC model of the system under development. Indeed this is very similar to the one in conventional design method. Moreover the design has modules, ports and signals from the very beginning of the process, which will clarify the design activities.

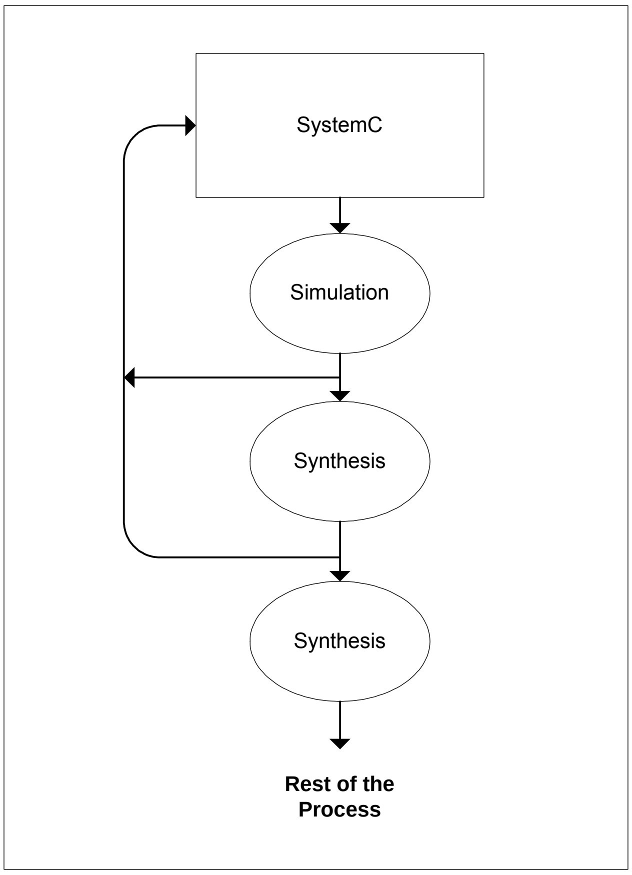


Figure 0.2 Design Method with SystemC.

As SystemC is providing a modular design concept, development of big systems in multi team environments is a well defined task. Each team can grab one ore more modules then start the design process in their own design environment. After teams have mature SystemC models they can combine their models to have the final design.

2.3 Hardware Software Co-Design and Co-Simulation

SystemC is a modeling platform consisting of a set of C++ class libraries, plus a simulation kernel that supports hardware modeling concepts at the system level, behavioral level and register transfer level. The C/C++ programming languages are widely used by systems architects and software engineers in their domains, but these languages lack semantics to adequately describe hardware modeling concepts. SystemC offers a solution that uses C++ extensions to add hardware modeling constructs.

In classical design methodologies, once a system has been partitioned and handed off to the hardware and software teams, software engineers re-describe the system-level architecture before proceeding with their work. Moreover, the hardware engineers re-write the high-level C/C++ description into an entirely different language like VHDL or Verilog. As a result, there occurs a distance between systems and software engineers and hardware engineers. It means there is a possibility of introducing errors and inconsistencies into the design, as C/C++ code is manually re-written. SystemC is a solution that cures this split by providing a refinement methodology from the original C/C++ functional and architectural descriptions to enable hardware/software co-design.

SystemC offers the ability to describe both hardware and software in the same high-level language, providing well-defined C-based constructs in a familiar and consistent programming environment. In addition, SystemC is open, available to everyone and it provides the ability to take advantage of a wide range of EDA tools that are being developed around it. SystemC provides a robust software environment for hardware/software co-design, but its greatest strength is the fact that it is being widely adopted by a large and growing group of system houses, semiconductor companies, IP providers, embedded systems and EDA tool vendors. In addition, the underlying source code is open and available through the Open SystemC Community Licensing model. Unlike proprietary design languages that require designers to depend on one company for tools and support, the open community licensing approach ensures that the SystemC modeling platform will evolve quickly and that many companies will provide a wide range of tools, libraries and services based on this standard.

2.4 Language Features

As SystemC is an open source design language, the documentation about the language, the basic simulation kernel and design methodologies mostly used with SystemC, are also open to public and may be found on the organization's web page [6],[7]. The important features of SystemC that are useful for modeling are briefly discussed here.

2.4.1 Modules, Ports and Signals

The basic block of a SystemC program is a *module*. A module is similar to the concept of entity in VHDL and module in Verilog. It is an abstract representation of a functional unit, without specifying any implementation details. Each module has a set of *ports* through which it interacts with the outside world. Ports can be input ports, output ports, or input/output (I/O) ports. If a module reads data, it must have at least one input or I/O port, and if it writes data, it should have one or more output or I/O ports. Individual modules communicate with one another through *signals* that connect the ports of the modules.

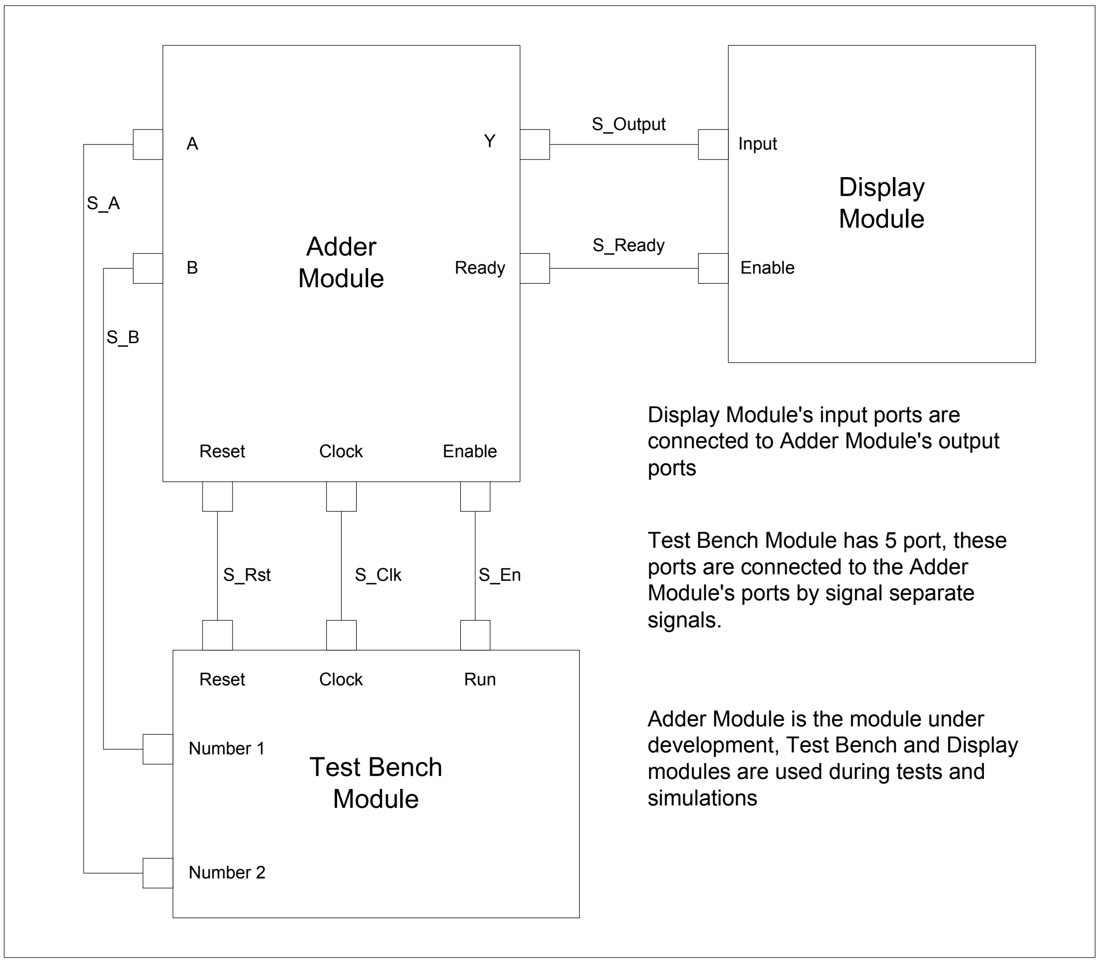

S_Output
Display Module
Input
Enable
S_Ready
Adder Module
A
Y
B
Ready
S_A
S_B
Reset
Clock
Enable
S_Rst
S_Clk
S_En
Reset
Clock
Run
Test Bench Module
Number 1
Number 2
Display Module's input ports are connected to Adder Module's output ports
Test Bench Module has 5 port, these ports are connected to the Adder Module's ports by signal separate signals.
Adder Module is the module under development, Test Bench and Display modules are used during tests and simulations

Figure 0.3 Modules, Ports, Signals and Their Relations.

Thus, signals are similar to the wires that interconnect different hardware units on a circuit board, and the ports correspond to the pins of these units. A simple SystemC design is given in Figure 2.3.

2.4.2 Processes

The code that implements the algorithm of a module is encapsulated in one or more *processes*. There are three types of processes in SystemC. This classification is based on how the underlying SystemC simulation kernel calls and executes the processes.

- *Method Process (SC_METHOD)* – Each method process has a *sensitivity list* that lists the signals that can activate this process. Whenever there is a change in one of the signals in this list, the process is executed. Once the process starts execution, it cannot be suspended and it runs until it returns.
- *Thread Process (SC_THREAD)* – This process is similar to SC_METHOD in that it also has a sensitivity list to control its activation. It can be suspended and reactivated by the user by adding relevant language constructs in the code. When the process is suspended, it waits until one of the signals in its sensitivity list changes. It then resumes execution from the point where it was suspended.
- *Clocked Thread Process (SC_CTHREAD)* – This process is a special case of SC_THREAD where the sensitivity list has only one signal, and it is activated when a specific edge (low to high or high to low) of that signal occurs. This activation scheme allows the modeling of synchronous designs in a simple manner.

2.4.3 Data Types and Constructs

SystemC is not a completely new programming language but C++ with additional classes, so all C++ data types are supported. For modeling hardware, additional data types are available. These include types for representing bits, bit vectors, 4-valued logic, variable precision integers, etc. In addition to these data types, the language also provides constructs that enable representing the hardware behavior. There are wait() statements that suspend execution, write() and read() functions to send and receive data from ports, and so on. More details of the language are discussed in [6], and a complete list of features can be found in the SystemC user’s guide [7].

The language features can be illustrated with an example. Typically, the code is split to two files – a header file that describes the ports and the processes, and a C++ file that provides the implementation.

The example in Figure 2.4, represents a counter module. The code is split to two files counter.h and counter.cpp. The header file defines a module named counter. The module has an SC_METHOD process named *action* and it is declared to be sensitive to the positive edge of the signal *clock*. Every time the clock signal makes a low to high transition, the process is invoked. The implementation of the process *action* is given in the file counter.cpp.

```
//************
//  counter.h
//************

SC_MODULE(counter) {

  sc_in<bool>  reset;
  sc_in<bool>  enable;
  sc_out<int>  result;
  sc_in_clk    CLK;


  SC_CTOR(counter)
     {
      SC_CTHREAD(entry, CLK.pos());
    }

  void entry();
};
```

```
//*************
// counter.cpp
//*************

#include <systemc.h>
#include "counter.h"

int  counter_value;

void counter::entry() {


  // main functionality
  while(1) {
    wait();
      if(reset.read() == 1 ){
        printf("Counter :\n" );
        counter_value = 0 ;
      }else{
        if(enable.read() == 1){
          counter_value += 1 ;
          printf("Counter : value
            %d\n", counter_value);
        }
      }
      result.write((int)counter_value);
  }
}
```

Figure 0.4 Sample SystemC Model of an Integer Counter.

2.5 SystemC Synthesis

In general, synthesis tool reads the behavioral description of a model and translates that to an equivalent netlist. The tool has a set of hardware building blocks with well-defined parameters. The parameters include timing, area, power dissipation information, etc. When the tool parses the behavioral code, it maps the statements to the appropriate hardware components. The conceptual diagram for synthesis operation is given in Figure 2.5.

As shown in Figure 2.6, the code shown on the left side is mapped to the hardware circuit on the right side. In general, the tool generates hardware based on the coding style used in the model. The same model can be written in different ways without affecting the simulation results, but the synthesized circuit may be different in each case. This is because the tool is basically a piece of software that performs the hardware mapping based on a pre-defined algorithm. Hence care has to be taken to follow the correct coding style that is suited for a given scenario.

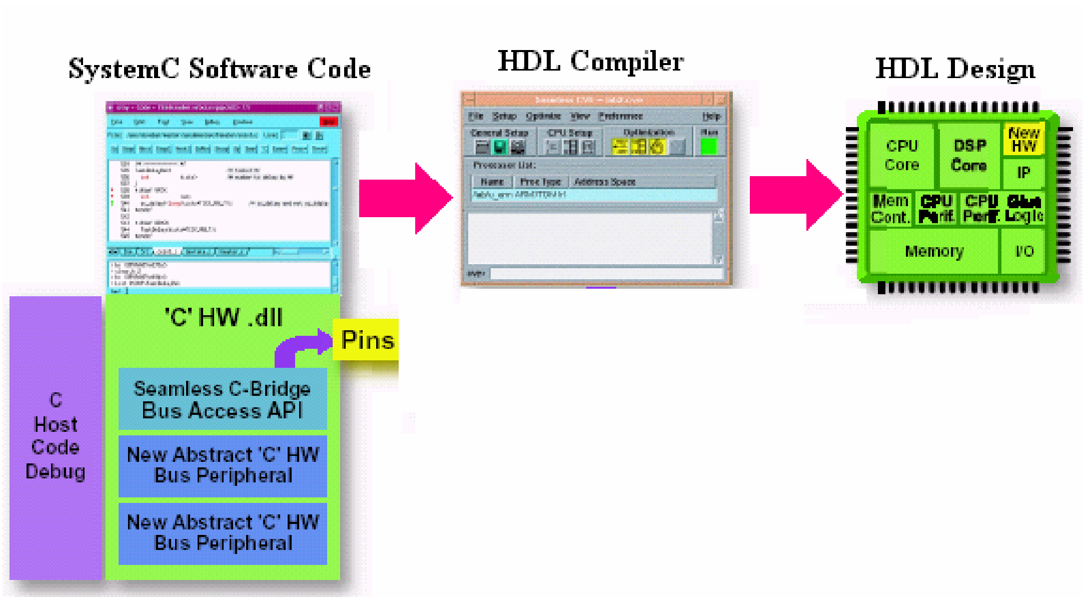
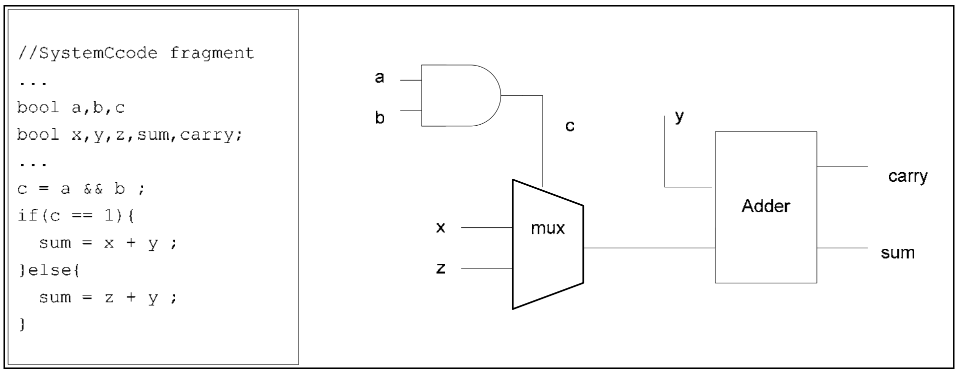


Figure 0.5 SystemC Synthesis to HDL.



```
//SystemCcode fragment
...
bool a,b,c
bool x,y,z,sum,carry;
...
c = a && b ;
if(c == 1){
  sum = x + y ;
}else{
  sum = z + y ;
}
```

Figure 0.6 Behavioral Synthesis Example.

2.5.1 Synchronous Sequential Systems

While the above task of mapping software code to hardware library components is ideal for simple combinational circuits, additional considerations have to be taken to synthesize synchronous sequential circuits. These circuits have at least one clock signal that controls the operation of the circuit. Typically, each clock cycle causes specific sub-units of the circuit to execute. Hence the hardware implementation of such a model consists of;

- Hardware components to perform the actions at each clocked state
- Control unit to control the operation of the state machine, to ensure that the components in step 1 are activated at the appropriate clock cycles.

While the first point can be addressed by mapping the relevant lines of code to hardware, the second issue is more complicated, and is termed *scheduling*. A given set of hardware units can be scheduled in different ways to perform trade-off analysis.

2.5.2 Synthesis Tool Operation

In general, a synthesis tool performs at least the two tasks mentioned in the previous section. The general operation flow for a synthesis tool is given in Figure 2.7.

The synthesis tool first reads the model and performs a syntax check to ensure that only synthesizable constructs are used in the model. If this check fails, the user is notified, and it is the task of the user to correct the code and run the tool again. After this, the tool maps the code to hardware components. The next step is scheduling the design to synthesize the control unit for the system. The result of this step is the netlist of the synthesized design. This is an equivalent hardware representation of the system modeled at the behavioral level.

2.5.3 CoCentric SystemC Compiler

Synthesis from SystemC is a relatively new area, and hence the availability of synthesis tools that support SystemC is limited. One of the SystemC synthesis tool suites is the CoCentric [4] SystemC Compiler and the design compiler (DC). CCSC reads a model written in SystemC and allocates hardware, and DC performs the scheduling operation. In addition, there is another tool, BCView that provides a graphical user environment for analysis after the scheduling step. Using this feature, the user can visualize the hardware allocation and utilization for each clock cycle, in addition to getting further information about the model. Some types of scheduling errors can be analyzed using this tool.

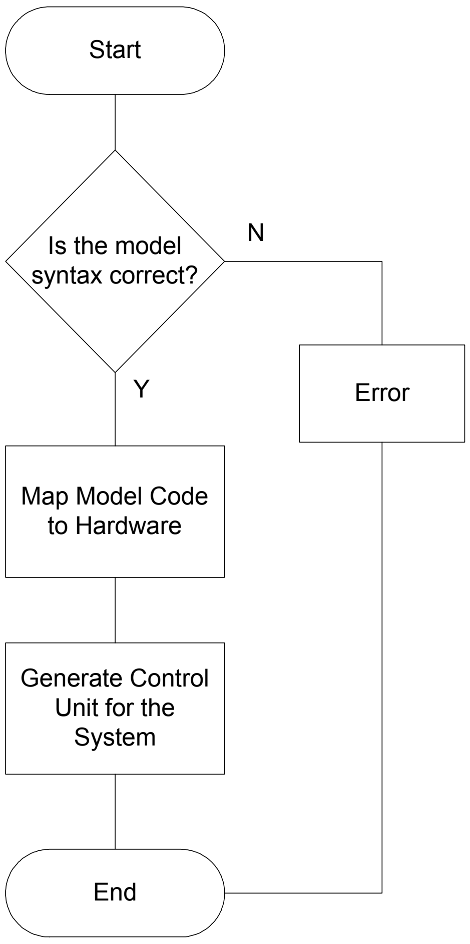

Start
Is the model syntax correct?
N
Y
Error
Map Model Code to Hardware
Generate Control Unit for the System
End

Figure 0.7 Flowchart Showing the Synthesis Tool Operation.

2.5.4 Cadence NC SystemC Simulator

Among the other tools that support SystemC, The one from Cadence is important NC SystemC Simulator [8]. It reads a specification written in SystemC and generates synthesizable HDL code for the hardware part. So, NC can be considered a SystemC synthesis tool. It also has provisions for hardware-software co-design and interface synthesis.

2.6 Design Constrains for Synthesizable SystemC Code

By basic definition terms, synthesizing is the mapping operation from the SystemC domain to the Hardware Description Language (HDL) domain. During this operation certain predefined rules and constraints are obeyed. There are different synthesizers from different companies in the market today, and each of them has their specific rule sets [10]. The most general constraints are briefly explained here

2.6.1 Modules

The basic building block in SystemC is the module. A SystemC module is a container in which processes and other modules are instantiated. A typical module can have

- Single or multiple RTL processes to specify combinational or sequential logic
- Multiple RTL modules to specify hierarchy
- One or more member functions that are called from within an instantiated process or module

It is allowed to declare member functions in a module that are not processes. This type of member function is not registered as a process in the module’s constructor. It can be called from a process. Member functions can contain any synthesizable

C++ or SystemC statement allowed in a SC_METHOD process. A member function that is not a process can return any synthesizable data type.

In the module implementation file, the functionalities of SC_METHOD process and member functions are defined.

In the module implementation description, the programmer can read from or write to a port or signal by using the read and write methods or by assignment. In order to read or write a port, as a recommended coding practice, the programmer is advised to use the read() and write() methods. The assignment operator should be used for variables.

It is possible to read or write all bits of a port or signal. It is not allowed to read or write the individual bits, regardless of the type. To do a bit-select on a port or signal, the port value should be read into a temporary variable and a bit-selection may be done on the temporary variable.

When a value is assigned to a signal or port, the value on the right side is not transferred to the left side until the process ends. This means the signal value as seen by other processes is not updated immediately, but it is deferred.

When a value is assigned to a variable, the value on the right side is immediately transferred to the left side of the assignment statement within the process.

A hierarchical module can be created with multiple instantiated modules. To create such a hierarchical module;

1. Create data members in the top-level module that are pointers to the instantiated modules.
2. Allocate the instantiated modules inside the constructor of the top-level module, giving each instance a unique name.
3. Bind the ports of the instantiated modules to the ports or signals of the top-level module. Use either binding by position or binding by name coding style

2.6.2 Processes

SystemC provides processes to describe the parallel behavior of hardware systems. This means processes execute concurrently rather than sequentially like C++ functions. The code within a process, however, executes sequentially.

Defining a process is similar to defining a C++ function. A process is declared as a member function of a module class and registered as a process in the module's constructor. Registering a process means that it is recognized as a SystemC process rather than as an ordinary member function. Programmer can register multiple different processes, but it is an error to register more than one instance of the same process. To create multiple instances of the same process, enclose the process in a module and instantiate the module multiple times.

It is possible to define a sensitivity list that identifies which input ports and signals trigger execution of the code within a process. It is also possible to define level-sensitive inputs to specify combinational logic or edge-sensitive inputs to specify sequential logic.

A process can read from and write to ports, internal signals, and internal variables. Processes use signals to communicate with each other. One process can cause another process to execute by assigning a new value to a signal that interconnects them. It is not advised to use data variables for communication between processes, because the processes execute in random order and it can cause non-determinism (order dependencies) during simulation.

SystemC provides three process types; SC_METHOD, SC_CTHREAD, and SC_THREAD, that execute whenever their sensitive inputs change. For simulation, any of the process types can be used. For RTL synthesis, only the SC_METHOD process can be used. The SC_METHOD process is sensitive to either changes in signal values (level-sensitive) or to particular transitions (edges) of the signal (edge-sensitive) and executes when one of its sensitive inputs changes.

2.6.3 Ports

Each module has any number of input, outputs, and inout ports which determine the direction of data into or out of the module. A port is a data member of SC_MODULE. Any number of sc_in, sc_out, and sc_inout ports can be declared. To read from an output port, declare it as an sc_inout rather than an sc_out port.

Ports connect to signals and have a data type associated with them. For synthesis, programmer should declare each port as one of the synthesizable data types.

2.6.4 Signals

Modules use ports to communicate with other modules. In hierarchical modules, signals are used to communicate between the ports of instantiated modules. Internal signals are used for peer-to-peer communication between processes within the same module,

A signal's bit-width is determined by its corresponding data type. Data type can be specified as any of the synthesizable SystemC or C++ data types. Signals and the ports they connect must have the same data types.

Inside a module, data member variables of any synthesizable SystemC or C++ type can be declared. These variables can be used for internal storage in the module. It is not advised to use data variables for peer-to-peer communication in a module. This can cause pre- and post-synthesis simulation mismatches and non-determinism (order dependency) in the design.

SystemC processes are declared in the module body and registered as processes inside the constructor of the module. Programmer must declare a process with a return type of void and no arguments. To register a function as an SC_METHOD process, the SC_METHOD macro that is defined in the SystemC class library, is used. The SC_METHOD macro takes one argument, the name of the process.

2.6.5. Sensitivity List

An SC_METHOD process reacts to a set of signals called its sensitivity list. Designer can use the sensitive(), sensitive_pos(), or sensitive_neg() functions or the sensitive, sensitive_pos, or sensitive_neg streams in the sensitivity declaration list.

For combinational logic, define a sensitivity list that includes all input ports, inout ports, and signals used as inputs to the process. It is possible to use the sensitive method to define the level-sensitive inputs. Programmer may specify any number of sensitive inputs for the stream-type declaration, and specify only one sensitive input for the function-type declaration. The sensitive function can be made multiple times with different inputs.

To eliminate the risk of pre- and post-synthesis simulation mismatches, the programmer should include all the inputs to the combinational logic process in the sensitivity list of the method process.

For sequential logic, sensitivity list should be defined for the input ports and signals that trigger the process. One of the sensitive_pos, sensitive_neg, or both the sensitive_pos and sensitive_neg methods should be used to define the edge-sensitive inputs that trigger the process. Ports and the edge-sensitive inputs should be declared as type sc_in<bool>. Any number of sc_in<bool> inputs may be declared. The sensitivity list may be defined by using either the function or the stream syntax.

Note that, It is not allowed to specify both edge-sensitive and level-sensitive inputs in the same process for synthesis. It is not possible to declare an sc_logic type for the clock or other edge-sensitive inputs. Only sc_in<bool> data type can be declared.

2.6.6. Converting to a Synthesizable Subset

To prepare for synthesis, all non-synthesizable code should be converted into synthesizable code. This is required only for functionality that is to be synthesized. Although any SystemC class or C++ construct can be used for simulation and other stages of the design process, only a subset of the language can be used for synthesis. To comment out code that is needed only for simulation, #ifdef and #endif precompiler commands can be used by the programmer. The full list of the non-synthesizable C++ and SystemC constructs is given in Appendix-B,

Table 0.1 Synthesizable Data Types.

SystemC type	**Description**
sc_bit	A single-bit true or false value. Supported but not recommended. Use the bool data type.
sc_bv<*n*>	Arbitrary-length bit vector. Use sc_uint<*n*> when possible.
sc_logic	A single-bit 0, 1, X, or Z.
sc_lv<*n*>	Arbitrary-length logic vector.
sc_int<*n*>	Fixed-precision integers restricted in size up to 64 bits and 64 bits of precision during operations.
sc_uint<*n*>	Fixed-precision integers restricted in size up to 64 bits and 64 bits of precision during unsigned operations.
sc_bigint<*n*>	Arbitrary-precision integers recommended for sizes over 64 bits and unlimited precision.
sc_biguint<*n*>	Arbitrary-precision integers recommended for sizes over 64 bits and unlimited precision, unsigned.
bool	A single-bit true or false value.
int	A signed integer, typically 32 or 64 bits, depending on the platform.
unsigned int	An unsigned integer, typically 32 or 64 bits, depending on the platform.
long	A signed integer, typically 32 bits or longer, depending on the platform.
unsigned long	An unsigned integer, typically 32 bits, depending on the platform.
char	8 bits, signed character, platform-dependent.
unsigned char	8 bits, unsigned character, platform-dependent.
short	A signed short integer, typically 16 bits, depending on the platform.
unsigned short	An unsigned short integer, typically 16 bits, depending on the platform.
struct	A user-defined aggregate of synthesizable data types.
enum	A user-defined enumerated data type.

SystemC supports most of the regular data types used in C++. SystemC provides a set of limited-precision and arbitrary-precision data types that allows the designer to create integers, bit vectors, and logic vectors of any length. SystemC also supports all common operations on these data types. The supported set of data types are given in Table 2.1.

2.7. A Design Example

In order to demonstrate design flow with SystemC, a small example is given here. For this purpose, a 4 bit counter is selected because of its simple operation. With the counter module designed, a test bench is also designed with SystemC in order to test the module under development. The design of the test bench is given in Figure 2.8.

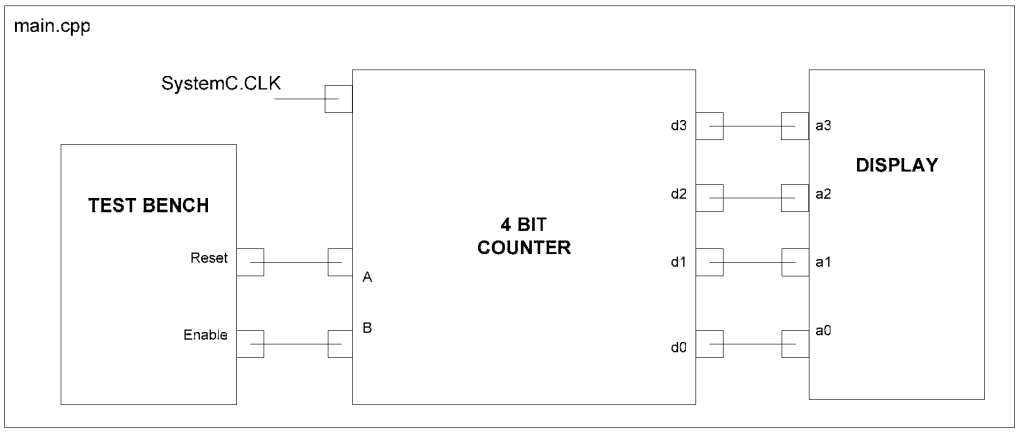


Figure 0.8 Test Bench for Counter Example.

Counter module simply counts up when Enable, Reset and CLK inputs are driven with proper signals as their names imply. Indeed these are the basic requirements for the project under development.

```
//-----------
// counter.h
//-----------

SC_MODULE(counter) {

  sc_in<bool>  reset;
  sc_in<bool>  enable;
  sc_out<bool> d3;
  sc_out<bool> d2;
  sc_out<bool> d1;
  sc_out<bool> d0;
  sc_in_clk    CLK;


  SC_CTOR(counter)
  {
    SC_CTHREAD(entry, CLK.pos());
  }

  void entry();
};
```

```
//-------------
// counter.cpp
//-------------

#include <systemc.h>
#include "counter.h"
sc_uint<4>  counter_value;

void counter::entry() {

  // main functionality
  while(1){
    wait();
      if(reset.read() == 1 ){
        counter_value = 0 ;
      }else{
        if(enable.read() == 1){
          counter_value +=  1;
        }
      }
      d3 = counter_value[3];
      d2 = counter_value[2];
      d1 = counter_value[1];
      d0 = counter_value[0];
  }// while
}
```

Figure 0.9 Counter Source Files.

For the test bench operation stimulus.h and stimulus.cpp files are created to have the reset and enable functionalities. Source files are shown in Figure 2.10, Reset and Enable output ports are driven depending on an internal counter. This stimulus module is only used for testing the counter module, so not too much effort is spent on it.

Starting with the counter module; to define the input and output ports and register the necessary methods to handle the counting function counter.h and counter.cpp files are introduced as shown in Figure 2.9. In the header file the input ports Reset and Enable are declared as well as output ports d3 to d0. In the source file the functionality is implemented. At each clock cycle, which is controlled by the **`wait()`** statement, internal counter value is incremented by one, and the output ports are updated.

```
//------------
// stimulus.h
//------------

SC_MODULE(stimulus) {

  sc_out<bool> reset;
  sc_out<bool> enable;
  sc_in<bool>  CLK;

  SC_CTOR(stimulus)
  {
    SC_CTHREAD(entry, CLK.pos());
  }
  void entry();
};
```

```
//-------------
// stimulus.cpp
//-------------

#include <systemc.h>
#include "stimulus.h"

int cycle = 0;

void stimulus::entry() {

  while(1){
    wait();
    cycle++;
    //sending some reset values
    if (cycle<2) {
      reset.write(true);
      enable.write(false);
    }else {
      reset.write(false);
      enable.write(true);
    }
 }//while
}
```

Figure 0.10 Source Files for Stimulus.

Similar to Stimulus module, there is one Display module for testing purposes. This module takes the counter output and display them to the user with printf() functions. Because Counter module is the module under design it is not advisable to use C++ library functions. Instead these type of informing functions are used in test bench modules to indicate the test results to the user. Source files are shown in Figure2.11.

```
//------------
// display.h
//------------

SC_MODULE(display) {

  sc_in<bool>  a3;
  sc_in<bool>  a2;
  sc_in<bool>  a1;
  sc_in<bool>  a0;
  sc_in<bool>  CLK;


  SC_CTOR(display)
  {
      SC_METHOD(entry);
      sensitive_neg(CLK);
  }

  void entry();
};
```

```
//-------------
// display.cpp
//-------------

#include <systemc.h>
#include "stimulus.h"

int display_cycle = 0;
sc_uint<4> tmp ;

void display::entry(){
  tmp[3] = a3.read();
  tmp[2] = a2.read();
  tmp[1] = a1.read();
  tmp[0] = a0.read();
  display_cycle++;
  printf("Display : time : %d
         counter : %d \n",
        (int)sc_simulation_time(),
        (unsigned int)tmp );
}
```

Figure 0.11 Source Files for Display Module.

Finally all these modules are instantiated in main.cpp file and the ports are connected to each other by dedicated signals. Main file is shown in Figure 2.12. In the main.cpp file, one copy of each module is created then their connections are made with their method calls.

```
//------------
// main.cpp
//------------
#include <systemc.h>
#include "counter.h"
#include "display.h"
#include "stimulus.h"

int sc_main (int argc , char *argv[]) {
  sc_clock         clock;
  sc_signal<bool>  reset;
  sc_signal<bool>  enable;
  sc_signal<bool>  s3;
  sc_signal<bool>  s2;
  sc_signal<bool>  s1;
  sc_signal<bool>  s0;

  stimulus stimulus1("stimulus_block");
           stimulus1.reset(reset);
           stimulus1.enable(enable);
           stimulus1.CLK(clock.signal());

  counter counter1( "counter_body");
           counter1.d3(s3);
           counter1.d2(s2);
           counter1.d1(s1);
           counter1.d0(s0);
           counter1.enable(enable);
           counter1.reset(reset);
           counter1.CLK(clock);

. .  .
```

```
// main.cpp
// continued

  display  display1 ( "display");
           display1.a3(s3);
           display1.a2(s2);
           display1.a1(s1);
           display1.a0(s0);
           display1.CLK(clock);

         // tracing:
         // trace file creation
         sc_trace_file *tf =
         sc_create_vcd_trace_file("counter");
         // External Signals
         sc_trace(tf, clock.signal(), "clock");
         sc_trace(tf, reset, "reset");
         sc_trace(tf, enable, "enable");
         sc_trace(tf, s3, "s3");
         sc_trace(tf, s2, "s2");
         sc_trace(tf, s1, "s1");
         sc_trace(tf, s0, "s0");
         sc_start(200);
         sc_close_vcd_trace_file(tf);

//  sc_start(clock, -1);
  return 0;
}
```

Figure 0.12 Main Source File.

SystemC open source library comes with the free SystemC simulator, this means when this project is compiled and linked, then the output executable is already a standalone simulator for the system under design. In order to have signals traced, they should be explicitly indicated to the SystemC simulator kernel.

When the executable is run it produces a trace file called counter.vcd, which is a standard trace file and viewed by any trace file viewer. During the thesis work, an open source free vcd viewer is used. The trace for the signals is shown in Figure 2.13.

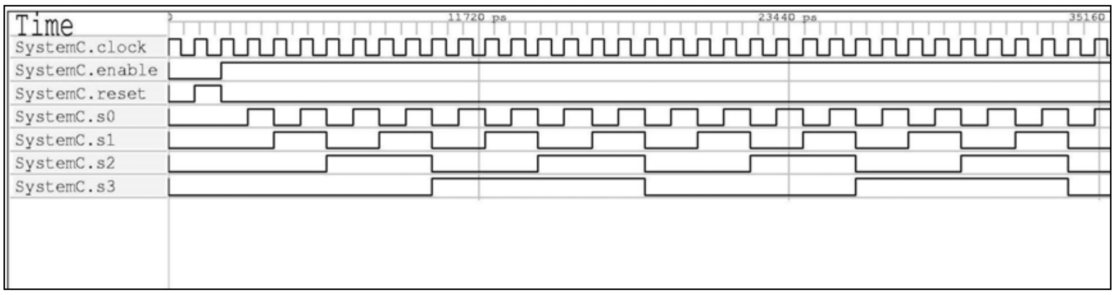


Figure 0.13 Signal Trace of Counter Example.

THE 8051 CORE

Despite it’s relatively old age, the 8051 is one of the most popular microcontrollers in use today. Many derivative microcontrollers have since been developed that are based on--and compatible with--the 8051. Thus, the ability to program an 8051 is an important skill for anyone who plans to develop products that will take advantage of microcontrollers.

The original 8051 core is an accumulator-based design with 255 instructions. Each basic instruction cycle takes 12 clocks. The CPU has four banks of eight 8-bit registers in on-chip ram for context switching; these registers reside within the 8051's lower 128 bytes of ram along with a bit-operation area and scratchpad ram. The architecture of the 8051 processor core is given in Figure 3.1.

The popularity of 8051 microprocessor core is one of the reasons for choosing it as the case study object for this thesis. For many years there occurred 8051 variants employing the original core and some peripheral elements such as, Analog Digital Converters, Specific Communication Buses, Extended RAM and ROM etc. A list for known variants is given in Appendix-D. Although 8051 core is being there for many years and it may be thought that the popularity is because of its legacy codes and backward compatibility, this is not the truth. Even in today’s high end technologies 8051 core is selected because of its effective structure. Most of the Smart Cards employ 8051 core inside to handle secure communication and transaction [19]. Also most of the SIM Cards used in GSM mobile phones uses 8051 core inside. Philips’ new Contactless Smart Card project, which is known as MIFARE and heavily used in public transportation area, uses 8051 cores in card reader/writer modules [20].

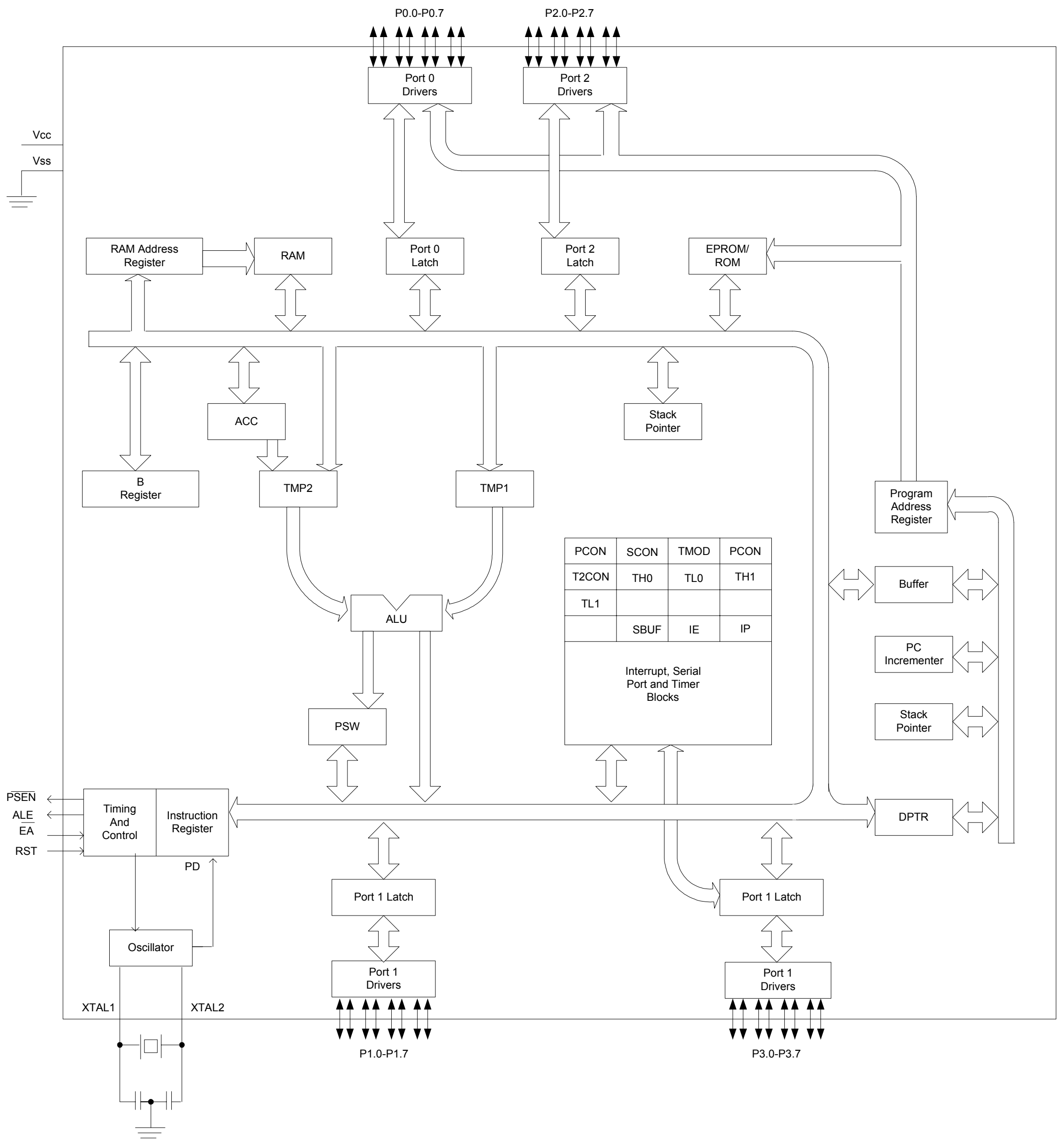


Figure 0.1 The Original 8051 Core.

The other reason is, although there are many C++ simulators, VHDL and Verilog models of 8051 core, a SystemC model is not available yet [17].[18]. One of those HDL models for 8051, which also encourages our work, was studied in a thesis given to Middle East Technical University [9] The real power of SystemC appears while designing a system composing of one microprocessor and some customized digital peripheral circuits. Having a 8051 model in SystemC will provide a fundamental design platform for those type of projects in the future, so this is also encouraging our work.

3.1 Types of Memory

The 8051 has three very general types of memory. To effectively program the 8051 it is necessary to have a basic understanding of these memory types. They are:

- *On-Chip Memory* refers to any internal memory. It can be code, RAM, or any other memory that physically exists on the microcontroller itself.
- *External Code Memory* is program memory that resides off-chip. This is often in the form of an external EPROM.
- *External RAM is* RAM memory that resides off-chip. This is often in the form of standard static RAM or flash RAM

3.1.1 Code Memory

Code memory is the memory that holds the actual 8051 programs that is to be run. This memory is limited to 64K and comes in many shapes and sizes: Code memory may be found *on-chip*, either burned into the microcontroller as ROM or EPROM. Code may also be stored completely *off-chip* in an external ROM or, more commonly, an external EPROM. Flash RAM is also another popular method of storing a program. Various combinations of these memory types may also be used--that is to say, it is possible to have 4K of code memory *on-chip* and 64k of code memory *off-chip* in an EPROM.

When the program is stored on-chip the 64K maximum is often reduced to 4k, 8k, or 16k. This varies depending on the version of the chip that is being used. Each version offers specific capabilities and one of the distinguishing factors from chip to chip is how much ROM/EPROM space the chip has.

However, code memory is most commonly implemented as off-chip EPROM. This is especially true in low-cost development systems and in systems developed by students.

3.1.2 External RAM

As an obvious opposite of *Internal RAM*, the 8051 also supports what is called *External RAM*. As the name suggests, External RAM is any random access memory which is found *off-chip*. Since the memory is off-chip it is not as flexible in terms of accessing, and is also slower. For example, to increment an Internal RAM location by 1 requires only 1 instruction and 1 instruction cycle. To increment a 1-byte value stored in External RAM requires 4 instructions and 7 instruction cycles. In this case, external memory is 7 times slower. What External RAM loses in speed and flexibility it gains in quantity. While Internal RAM is limited to 128 bytes, the 8051 supports External RAM up to 64K.

3.1.3 On-Chip Memory

As mentioned at the beginning of this chapter, the 8051 includes a certain amount of on-chip memory. On-chip memory is really one of two types: Internal RAM and Special Function Register (SFR) memory. The layout of the 8051's internal memory is presented in the following memory map:

As is illustrated in this map, the 8051 has a bank of 128 bytes of *Internal RAM*. This Internal RAM is found *on-chip* on the 8051 so it is the fastest RAM available, and it is also the most flexible in terms of reading, writing, and modifying it’s contents. Internal RAM is volatile, so when the 8051 is reset this memory is cleared.

The 128 bytes of internal ram is subdivided as shown on the memory map. The first 8 bytes (00h - 07h) are "register bank 0". By manipulating certain SFRs, a program may choose to use register banks 1, 2, or 3. These alternative register banks are located in internal RAM in addresses 08h through 1Fh. Bit Memory is also a part of internal RAM, from addresses 20h through 2Fh.

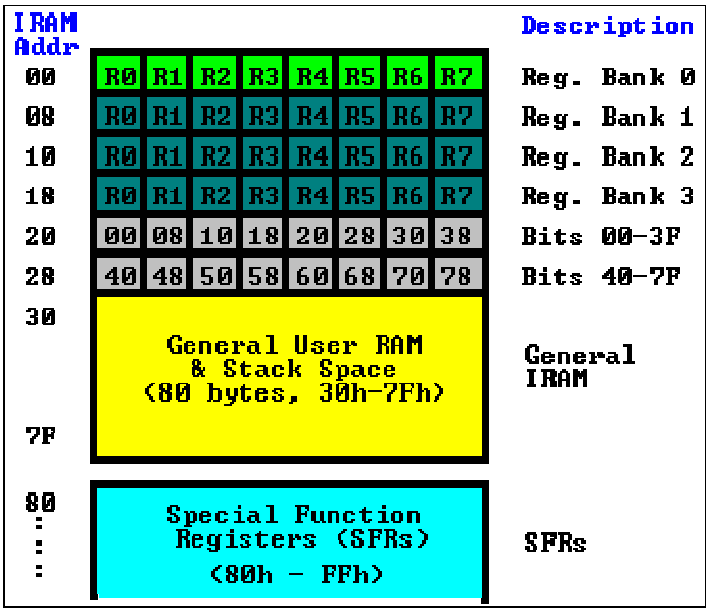


Figure 0.2 Memory Map of 8051 Core

The 80 bytes remaining of Internal RAM, from addresses 30h through 7Fh, may be used by user variables that need to be accessed frequently or at high-speed. This area is also utilized by the microcontroller as a storage area for the operating *stack*. This fact severely limits the 8051’s stack since, as illustrated in the memory map, the area reserved for the stack is only 80 bytes--and usually it is less since this 80 bytes has to be shared between the stack and user variables.

Register Banks

The 8051 uses 8 "R" registers which are used in many of its instructions. These "R" registers are numbered from 0 through 7 (R0, R1, R2, R3, R4, R5, R6, and R7). These registers are generally used to assist in manipulating values and moving data from one memory location to another. For example, to add the value of R4 to the Accumulator, following instruction should be executed:

```
ADD A,R4
```

If the Accumulator (A) contained the value 6 and R4 contained the value 3, the Accumulator would contain the value 9 after this instruction was executed. However, as the memory map shows, the "R" Register R4 is really part of Internal RAM. Specifically, R4 is address 04h. This can be see in the bright green section of the memory map. Thus the above instruction accomplishes the same thing as the following operation:

```
ADD A,04h
```

This instruction adds the value found in Internal RAM address 04h to the value of the Accumulator, leaving the result in the Accumulator. Since R4 is really Internal RAM 04h, the above instruction effectively accomplished the same thing.

As the memory map shows, the 8051 has four distinct register banks. When the 8051 is first booted up, register bank 0 (addresses 00h through 07h) is used by default. However, your program may instruct the 8051 to use one of the alternate register banks; i.e., register banks 1, 2, or 3. In this case, R4 will no longer be the same as Internal RAM address 04h. For example, if your program instructs the 8051 to use register bank 3, "R" register R4 will now be synonomous with Internal RAM address 1Ch.

The concept of register banks adds a great level of flexibility to the 8051, especially when dealing with interrupts. However, note that the register banks really reside in the first 32 bytes of Internal RAM.

Bit Memory

The 8051, a communications-oriented microcontroller, gives the user the ability to access a number of *bit variables*. These variables may be either 1 or 0.

There are 128 bit variables available to the user, numbered 00h through 7Fh. The user may make use of these variables with commands such as SETB and CLR. For example, to set bit number 24 (hex) to 1 you would execute the instruction:

```
        SETB 24h
```

It is important to note that Bit Memory is really a part of Internal RAM. In fact, the 128 bit variables occupy the 16 bytes of Internal RAM from 20h through 2Fh. Thus, if you write the value FFh to Internal RAM address 20h you've effectively set bits 00h through 07h. That is to say that:

```
        MOV 20h,#0FFh
```

is equivalent to:

```
        SETB 00h
        SETB 01h
        SETB 02h
        SETB 03h
        SETB 04h
        SETB 05h
        SETB 06h
        SETB 07h
```

As illustrated above, bit memory isn't really a new type of memory. It's really just a subset of Internal RAM. But since the 8051 provides special instructions to access these 16 bytes of memory on a bit by bit basis it is useful to think of it as a separate type of memory. However, always keep in mind that it is just a subset of Internal RAM--and that operations performed on Internal RAM can change the values of the bit variables.

Bit variables 00h through 7Fh are for user-defined functions in their programs. However, bit variables 80h and above are actually used to access certain SFRs on a bit-by-bit basis. For example, if output lines P0.0 through P0.7 are all clear (0) and user want to turn on the P0.0 output line then execute:

```
        MOV P0,#01h
```

or:

```
        SETB 80h
```

Both these instructions accomplish the same thing. However, using the SETB command will turn on the P0.0 line without effecting the status of any of the other P0 output lines. The MOV command effectively turns off all the other output lines which, in some cases, may not be acceptable.

Special Function Register (SFR) Memory

Special Function Registers (SFRs) are areas of memory that control specific functionality of the 8051 processor. For example, four SFRs permit access to the 8051’s 32 input/output lines. Another SFR allows a program to read or write to the 8051’s serial port. Other SFRs allow the user to set the serial baud rate, control and access timers, and configure the 8051’s interrupt system.

When programming, SFRs have the illusion of being Internal Memory. For example, if you want to write the value "1" to Internal RAM location 50 hex you would execute the instruction:

```
        MOV 50h,#01h
```

Similarly, to write the value "1" to the 8051’s serial port, write this value to the SBUF SFR, which has an SFR address of 99 Hex. Thus, to write the value "1" to the serial port you would execute the instruction:

```
        MOV 99h,#01h
```

It appears that the SFR is part of Internal Memory. This is not the case. When using this method of memory access (it’s called direct address), any instruction that has an address of 00h through 7Fh refers to an Internal RAM memory address; any instruction with an address of 80h through FFh refers to an SFR control register.

The Accumulator

The Accumulator, as it’s name suggests, is used as a general register to accumulate the results of a large number of instructions. It can hold an 8-bit (1-byte) value and is the most versatile register the 8051 has due to the shear number of instructions that make use of the accumulator. More than half of the 8051’s 255 instructions manipulate or use the accumulator in some way.

The "R" Registers

The "R" registers are a set of eight registers that are named R0, R1, etc. up to and including R7.These registers are used as auxillary registers in many operations.

The "B" Register

The "B" register is very similar to the Accumulator in the sense that it may hold an 8-bit (1-byte) value. The "B" register is only used by two 8051 instructions: MUL AB and DIV AB. Thus, to quickly and easily multiply or divide A by another number, store the other number in "B" and make use of these two instructions. Aside from the MUL and DIV instructions, the "B" register is often used as another temporary storage register much like a ninth "R" register.

The Data Pointer (DPTR)

The Data Pointer (DPTR) is the 8051's only user-accessable 16-bit (2-byte) register. The Accumulator, "R" registers, and "B" register are all 1-byte values. DPTR, as the name suggests, is used to point to data. It is used by a number of commands which allow the 8051 to access external memory. When 8051 accesses external memory it will access it at the address indicated by DPTR.

While DPTR is most often used to point to data in external memory, many programmers often take advantage of the fact that it's the only true 16-bit register available. It is often used to store 2-byte values which have nothing to do with memory locations.

The Program Counter (PC)

The Program Counter (PC) is a 2-byte address which tells the 8051 where the next instruction to execute is found in memory. When the 8051 is initialized PC always starts at 0000h and is incremented each time an instruction is executed. It is important to note that PC isn’t always incremented by one. Since some instructions require 2 or 3 bytes the PC will be incremented by 2 or 3 in these cases.

The Program Counter is special in that there is no way to directly modify it’s value. That is to say, it is not possible to do something like PC=2430h. On the other hand, if user executes LJMP 2430h you’ve effectively accomplished the same thing.

It is also interesting to note that while you may change the value of PC (by executing a jump instruction, etc.) there is no way to read the value of PC. That is to say, there is no way to ask the 8051 "What address are you about to execute?" As it turns out, this is not completely true:

The Stack Pointer (SP)

The Stack Pointer, like all registers except DPTR and PC, may hold an 8-bit (1-byte) value. The Stack Pointer is used to indicate where the next value to be removed from the stack should be taken from. When a value pushed onto the stack, the 8051 first increments the value of SP and then stores the value at the resulting memory location.

When a value popped off the stack, the 8051 returns the value from the memory location indicated by SP, and then decrements the value of SP.

This order of operation is important. When the 8051 is initialized SP will be initialized to 07h. If you immediately push a value onto the stack, the value will be stored in Internal RAM address 08h. This makes sense taking into account what was mentioned two paragraphs above: First the 8051 will increment the value of SP (from 07h to 08h) and then will store the pushed value at that memory address (08h).

SP is modified directly by the 8051 by six instructions: PUSH, POP, ACALL, LCALL, RET, and RETI. It is also used intrinsically whenever an interrupt is triggered

3.2 Addressing Modes of 8051

As is the case with all microcomputers from the PDP-8 onwards, the 8052 utilizes several memory addressing modes. An "addressing mode" refers to how the programmer accesses (.addresses.) a given memory location or data value. In summary, the addressing modes are listed below with an example of each:

• Immediate Addressing	`MOV A,#20h`
• Direct Addressing	`MOV A,30h`
• Indirect Addressing	`MOV A,@R0`
• External Direct	`MOVX A,@DPTR`
• External Indirect	`MOVX A,@R0`
• Code Indirect	`MOVC A,@A+DPTR`

Each of these addressing modes provides important flexibility to the programmer.

3.2.1 Immediate Addressing

Immediate addressing is so-named because the value to be stored in memory immediately follows the opcode in memory. That is to say, the instruction itself dictates what value will be stored in memory. For example:

```
MOV A,#20h
```

This instruction uses Immediate Addressing because the Accumulator (A) will be loaded with the value that immediately follows; in this case 20h (hexadecimal).

Immediate Addressing is very fast since the value to be loaded is included in the instruction. However, since the value to be loaded is fixed at compile-time it is not very flexible. It is used to load the same, known value every time the instruction executes.

3.2.2 Direct Addressing

Direct addressing is so-named because the value to be stored in memory is obtained by directly retrieving it from another memory location. For example:

```
MOV A,30h
```

This instruction will read the data out of Internal RAM address 30h (hexadecimal) and store it in the Accumulator (A). Direct addressing is generally fast since, although the value to be loaded is not included in the instruction, it is quickly accessible since it is stored in the 8051's Internal RAM. It is also much more flexible than Immediate Addressing since the value to be loaded is whatever is found at the given address--which may change.

Also, it is important to note that when using direct addressing any instruction that refers to an address between 00h and 7Fh is referring to Internal RAM. Any instruction that refers to an address between 80h and FFh is referring to the SFR control registers that control the 8051 itself

3.2.3 Indirect Addressing

Indirect addressing is a very powerful addressing mode that in many cases provides an exceptional level of flexibility. Indirect addressing appears as follows:

```
MOV A,@R0
```

This instruction causes the 8051 to analyze the value of the R0 register. The 8051 will then load the Accumulator (A) with the value from Internal RAM that is found at the address indicated by R0. As an example, suppose R0 holds the value 40h and Internal RAM address 40h holds the value 67h. When the above instruction is executed the 8051 will check the value of R0. Since R0 holds 40h the 8051 will get the value out of Internal RAM address 40h (which holds 67h) and store it in the Accumulator.So, the Accumulator ends up holding 67h.

Indirect addressing always refers to Internal RAM; it never refers to an SFR. A simple example, SFR 99h can be used to write a value to the serial port. Thus one may think that the following would be a valid solution to write the value .1. to the serial port:

```
MOV R0,#99h       ; Load the SBUF address to R0
MOV @R0,#05h      ; Send data to Serial Port , WRONG!
```

On an 8051 these two instructions would produce an undefined result since the 8051 only has 128 bytes of Internal RAM. This is not valid. Since indirect addressing always refers to Internal RAM these two instructions would write the value 01h to Internal RAM address 99h on an 8052 processor which has 256 bytes of Internal RAM.

3.2.4 External Direct

With this addressing mode, external memory is accessed using DPTR register. There are only two commands that use External Direct addressing mode:

```
MOVX A,@DPTR
MOVX @DPTR,A
```

Both commands utilize DPTR. In these instructions, DPTR must first be loaded with the address of external memory that is to be read or written. Once DPTR holds the correct external memory address, the first command will move the contents of that external memory address into the Accumulator. For example, to read the contents of external RAM address 1516h, the following instructions are executed:

```
MOV  DPTR,#1516h        ; Adjust DPRT
MOVX A,@DPTR            ; Move the external data to Acc
```

In order to to write the contents of the Accumulator to external RAM address 1516, the following instructions are executed:

```
MOV  DPTR,#1516h        ; Adjust DPRT
MOVX @DPTR,A            ; Move the Acc value external RAM
```

3.2.5 External Indirect

External memory can also be accessed using a form of indirect addressing that is usually used in projects that have a small amount of external RAM. An example of this addressing mode is:

```
MOVX @R0,A
```

Once again, the value of R0 is first read and the value of the Accumulator is written to that address in External RAM. Since the value of @R0 can only be 00h through FFh the project would effectively be limited to 256 bytes of External RAM.

3.2.6 Code Indirect

Two additional 8052 instructions allow the developer to access the program code itself. This is useful for accessing data tables, strings, etc. The two instructions are:

```
MOVC A,@A+DPTR
MOVC A,@A+PC
```

As an example, to access the data stored in code memory at address 1200h, following instructions are executed:

```
MOV  DPTR,#1200h  ; Adjust DPTR
CLR  A            ; Clear The Acc to point first element
```

```
MOVC A,@A+DPTR
```

The `MOVC A,@A+DPTR` instruction moves the value contained in the code memory address that is pointed to by adding DPTR to the Accumulator.

3.3. Program Flow Instructions

When an 8051 is first initialized the PC SFR is reset to 0000h. The 8051 then begins to execute instructions sequentially in memory unless a program instruction causes the PC to be otherwise altered. There are various instructions that can modify the value of the PC; specifically, conditional branching instructions, direct jumps and calls, and "returns" from subroutines. Additionally, interrupts, when enabled, can cause the program flow to deviate from its otherwise sequential scheme.

3.3.1 Conditional Branching

The 8051 contains a suite of instructions which, as a group, are referred to as "conditional branching" instructions. These instructions cause program execution to follow a non-sequential path if a certain condition is true. Take, for example, the JB instruction. This instruction means "Jump if Bit Set." An example of the JB instruction might be:

Conditional branching is the fundamental building block of program logic since all "decisions" are accomplished by using conditional branching. Conditional branching can be thought of as the "IF...THEN" structure in 8051 assembly language. An important note worth mentioning about conditional branching is that the program may only branch to instructions located within 128 bytes prior to or 127 bytes after the address that follows the conditional branch instruction.

3.3.2 Direct Jumps

While conditional branching is extremely important, it is often necessary to make a direct branch to a given memory location without basing it on a given logical decision. In this the program continue at a given memory address without considering any conditions. This is accomplished in the 8051 using "Direct Jump and Call" instructions.

The LJMP instruction means "Long Jump" When 8051 executes this instruction the PC is loaded with the address of new address and program execution continues sequentially from there. The obvious difference between the Direct Jump and Call instructions and the conditional branching is that with Direct Jumps and Calls program flow always changes. With conditional branching program flow only changes if a certain condition is true. It is worth mentioning that, aside from LJMP, there are two other instructions that cause a direct jump to occur: the SJMP and AJMP commands. Functionally, these two commands perform the exact same function as the LJMP command--that is to say, they always cause program flow to continue at the address indicated by the command. However, these instructions differ from LJMP in that they are not capable of jumping to any address. They both have limitations as to the .range. of the jumps;

1. The SJMP command, like the conditional branching instructions, can only jump to an address within +/- 128 bytes of the SJMP command.
2. The AJMP command can only jump to an address that is in the same 2k block of memory as the AJMP command.

3.3.3 Direct Calls

Another instruction is the LCALL instruction which makes the long call operations. When the 8051 executes an LCALL instruction it immediately pushes the current Program Counter onto the stack and then continues executing code at the address indicated by the LCALL instruction. Similar in format to the AJMP instruction that was described in the previous section, the ACALL instruction provides a way to perform the equivalent of an "LCALL" with a two-byte instruction instead of three as long as the target routine is within the same 2k block of memory.

3.3.4 Returns from Routines

Another structure that can cause program flow to change is the "Return from Subroutine" instruction, known as RET in 8051 Assembly Language. The RET instruction, when executed, returns to the address following the instruction that called the given subroutine. More accurately, it returns to the address that is stored on the stack. The RET command is direct in the sense that it always changes program flow without basing it on a condition, but is variable in the sense that where program flow continues can be different each time the RET instruction is executed depending on from where the subroutine was called originally.

3.3.5 Interrupts

An interrupt is a special feature that allows the 8051 to break from its normal program flow to execute an immediate task, providing the illusion of "multi-tasking." The word "interrupt" can often be substituted with the word "event." An interrupt is triggered whenever a corresponding event occurs. When the event occurs, the 8051
temporarily puts "on hold" the normal execution of the main program and executes a special section of code referred to as the "Interrupt Service Routine" (ISR). The ISR performs whatever special functions are required to handle the event and then returns control to the 8051 at which point program execution continues as if it had never been interrupted.

3.4. 8051 Timers

The 8051 core is equipped with two timers, which may be controlled, set, read, and configured individually. The timers have three general functions:, keeping time and/or calculating the amount of time between events, counting the events themselves, or generating baud rates for the serial port. Timers always count up. It does not matter whether the timer is being used as a timer, a counter, or a baud rate generator: A timer is always incremented by the microcontroller.

Table 0.1 Timer Registers.

SFR	**Description**	**SFR Address**	**Bit Addressable**
TH0	Timer 0 High Byte	8Ch	No
TL0	Timer 0 Low Byte	8Ah	No
TH1	Timer 1 High Byte	8Dh	No
TL1	Timer 1 Low Byte	8Bh	No
TCON	Timer Control	88h	Yes
TMOD	Timer Mode	89h	No

Table 0.2 Timer Mode Registers.

Bit	**Name**	**Function**	**Timer**
7	GATE1	When this bit is set the timer will only run when INT1 (P3.3) is high. When this bit is clear the timer will run regardless of the state of INT1.	1
6	C/T1	When this bit is set the timer will count events on T1(P3.5). When this bit is clear the timer will be incremented every machine cycle.	1
5	T1M1	Timer mode bit for Timer1	1
4	T1M0	Timer mode bit for Timer1	1
3	GATE0	When this bit is set the timer will only run when INT0 (P3.2) is high. When this bit is clear the timer will run regardless of the state of INT0.	0
2	C/T0	When this bit is set the timer will count events on T0 (P3.4). When this bit is clear the timer will be incremented every machine cycle.	0
1	T0M1	Timer mode bit for Timer 0	0
0	T0M0	Timer mode bit for Timer 0	0

The two timers in 8051 core share two SFRs (TMOD and TCON) which control the timers, and each timer also has two SFRs dedicated solely to maintaining the value of the timer itself (TH0/TL0 and TH1/TL1). The SFRs used to control and manipulate the timers are presented in Table 3.1. If a timer contains the value 65,535 and is subsequently incremented, it will reset or overflow back to 0.

Timer Mode register, TMOD (0x0089), is used to control the mode of operation of both timers. Each bit of the SFR gives the microcontroller specific information concerning how to run a timer. The high four bits relate to Timer 1 whereas the low four bits perform the exact same functions for Timer 0. These functions are given in Table 3.2.

As noted in timer mode registers table four bits are used to specify a mode of operation of the two timers, these are explained in Table 3.3.

Table 0.3 Timer Modes.

M1	**M0**	**Timer Mode**	**Description**
0	0	0	13-bit Timer
0	1	1	16-bit Timer
1	0	2	8-bit auto-reload
1	1	3	Split timer mode

13-bit Timer Mode (mode 0):

This mode is kept around in 8051 core to maintain compatibility with its predecessor, the 8048. In this mode, THx will count from 0 to 31. When THx is incremented from 31, it will "reset" to 0. Effectively, only 13 bits of the two timer bytes are being used: bits 0-4 of THx and bits 0-7 of TLx. This means, the timer can only contain 8192 values. If a 13 bit timer is set to 0, it will overflow back to zero 8192 instruction cycles later.

16-bit Time Mode (mode 1)

This is a very commonly used mode and It functions just like 13-bit mode except that all 16 bits are used. TLx is incremented from 0 to 255. When TLx is incremented from 255, it resets to 0 and causes THx to be incremented by 1. Since this is a full 16-bit timer, the timer may contain up to 65536 distinct values.

8-bit Time Mode (mode 2)

This is 8-bit auto-reload mode. When a timer is in this mode, THx holds the "reload value" and TLx is the timer itself. TLx starts counting up. When TLx reaches 255 and is subsequently incremented instead of resetting to 0 it will be reset to the value stored in THx.

Split Timer Mode (mode 3)

Timer mode-3 is a split-timer mode. When Timer 0 is placed in mode 3, it essentially becomes two separate 8-bit timers. That is to say, Timer 0 is TL0 and Timer 1 is TH0. Both timers count from 0 to 255 and overflow back to 0. All the bits that are related to Timer 1 will now be tied to TH0 and all the bits related to Timer 0 will be tied to TL0. While Timer 0 is in split mode, the real Timer 1 (i.e. TH1 and TL1) can be put into modes 0, 1 or 2 normally--however, you may not start or stop the real timer 1 since the bits that do that are now linked to TH0. The real timer 1, in this case, will be incremented every machine cycle no matter what. The only benefit of using split timer mode is, while having two separate timers; additionally a baud rate generator is also available.

Finally, there is one more SFR that controls the two timers and provides valuable information about them. The TCON SFR has the following structure:

Note that, only 4 bits of TCON are defined instead of 8, this is because the other 4 bits of the TCON do not have anything to do with timers, they are related with interrupts

Table 0.4 Timer Control Register, TCON, 88h.

Bit	Name	Address	Function	Timer
7	TF1	8Fh	Timer 1 Overflow. This bit is set by the microcontroller when Timer 1 overflows.	1
6	TR1	8Eh	Timer 1 Run. When this bit is set Timer 1 is turned on. When this bit is clear Timer 1 is off.	1
5	TF0	8Dh	Timer 0 Overflow. This bit is set by the microcontroller when Timer 0 overflows.	0
4	TR0	8Ch	Timer 0 Run. When this bit is set Timer 0 is turned on. When this bit is clear Timer 0 is off.	0

3.5 Serial Ports

One of the 8051's many powerful features is its integrated UART, otherwise known as a serial port. The fact that the 8051 has an integrated serial port means that you may very easily read and write values to the serial port. If it were not for the integrated serial port, writing a byte to a serial line would be a rather tedious process requiring turning on and off one of the I/O lines in rapid succession to properly "clock out" each individual bit, including start bits, stop bits, and parity bits. However, the designer does not have to do this. Instead, he/she need to configure the serial port's operation mode and baud rate. Once configured, writing to an SFR will write a value to the serial port or, similarly reading the same SFR will read a value from the serial port. The 8051 will generate an interrupt when it finishes sending the character or received a character from the other party.

Setting the Serial Port Mode

The Serial Control Register, SCON, controls the serial port operation and definitions are given in Table 3.5.

Serial Port mode bits define in which mode it will work, this is given in Table 3.6.

Table 0.5 Serial Control Register, SCON, 99h.

Bit	Name	Address	Function
7	SM0	9Fh	Serial port mode bit 0
6	SM1	9Eh	Serial port mode bit 1.
5	SM2	9Dh	Mutliprocessor Communications Enable
4	REN	9Ch	Receiver Enable. This bit must be set in order to receive characters.
3	TB8	9Bh	Transmit bit 8. The 9th bit to transmit in mode 2 and 3.
2	RB8	9Ah	Receive bit 8. The 9th bit received in mode 2 and 3.
1	TI	99h	Transmit Flag. Set when a byte has been completely transmitted.
0	RI	98h	Receive Flag. Set when a byte has been completely received.

Table 0.6 Serial Mode Definitions.

SM0	SM1	Serial Mode	Description	Baud Rate
0	0	0	8-bit Shift Register	Oscillator / 12
0	1	1	8-bit UART	Set by Timer 1
1	0	2	9-bit UART	Oscillator / 32
1	1	3	9-bit UART	Set by Timer 1

3.6. Interrupts

As the name implies, an interrupt is some event that interrupts normal program execution. As stated earlier, program flow is always sequential, being altered only by those instructions that expressly cause program flow to deviate in some way. However, interrupts give us a mechanism to "put on hold" the normal program flow, execute a subroutine, and then resume normal program flow as it had been never left. This subroutine, called an interrupt handler or interrupt service routine (ISR), is only executed when a certain event (interrupt) occurs. The event may be one of the timers "overflowing," receiving a character via the serial port, transmitting a character via the serial port, or one of two "external events." The 8051 may be configured so that when any of these events occur the main program is temporarily suspended and control passed to a special section of code, which presumably would execute some function related to the event that occurred. Once complete, control would be returned to the original program. The main program never even knows it was interrupted. The ability to interrupt normal program execution when certain events occur makes it much easier and much more efficient to handle certain conditions. If there were no interrupts then the programmer would have to manually

check in main program whether the timers had overflowed, whether serial port had received another character or if some external event had occurred. Besides making the main program ugly and hard to read, such a situation would make the program inefficient.

There are several events that can trigger 8051 interrupts;

1. Timer 0 Overflow.
2. Timer 1 Overflow.
3. Reception/Transmission of Serial Character.
4. External Event 0.
5. External Event 1.

Obviously it is needed to be able to distinguish between various interrupts and be able to execute different code depending on what interrupt is triggered. This is accomplished by jumping to a fixed address when a given interrupt occurs, this is explained in detail in next chapter.

By default, at power-up, all interrupts are disabled. This means that even if, for example, the TF0 bit is set, the 8051 will not execute the interrupt. The software program must specifically tell the 8051 that it wishes to enable interrupts and specifically which interrupts it wishes to enable. It is possible to enable and disable interrupts by modifying the IE SFR (A8h):

The 8051 automatically evaluates whether an interrupt should occur after every instruction. When checking for interrupt conditions, it checks them in the following order:

1. External 0 Interrupt
2. Timer 0 Interrupt
3. External 1 Interrupt
4. Timer 1 Interrupt
5. Serial Interrupt

This means that if a Serial Interrupt occurs at exactly the same instant that an External 0 Interrupt occurs, the External 0 Interrupt will be executed first and the Serial Interrupt will be executed once the External 0 Interrupt has completed.

The 8051 offers two levels of interrupt priority: high and low. By using interrupt priorities the programmer may assign higher priority to certain interrupt conditions

When an interrupt is triggered, the following actions are taken automatically by the microcontroller:

1. The current Program Counter is saved on the stack, low-byte first, high-byte second.
2. Interrupts of the same and lower priority are blocked.
3. In the case of Timer and External interrupts, the corresponding interrupt flag is cleared.
4. Program execution transfers to the corresponding interrupt handler vector address.
5. The Interrupt Handler routine, written by the developer, is executed.

If the interrupt being handled is a Timer or External interrupt, the microcontroller automatically clears the interrupt flag before passing control to your interrupt handler routine. This means it is not necessary that you clear the bit in your code.

An interrupt ends when your program executes the Return from Interrupt, RETI instruction. When the RETI instruction is executed the following actions are taken by the microcontroller:

1. Two bytes are popped off the stack into the Program Counter to restore normal program execution.
2. Interrupt status is restored to its pre-interrupt status.

Serial Interrupts are slightly different than the rest of the interrupts. This is due to the fact that there are two interrupt flags: RI and TI. If either flag is set, a serial interrupt is triggered. The RI bit is set when a byte is received by the serial port and the TI bit is set when a byte has been sent. This means that when serial interrupt is executed it may have been triggered because of either of RI and TI or both flags set. So the interrupt service routine must check the status of these flags to determine what action is appropriate. Also, the RI and TI flags should be cleared with in the interrupt service routine since they are not automatically cleared by 8051.

DESIGN OF 8051 CORE WITH SYSTEMC

As SystemC is a modular programming language, in the design of 8051 core, a modular methodology is selected. The core processor is divided into modules and those modules are divided into sub modules where needed. The top level design is shown in Figure 4.1.

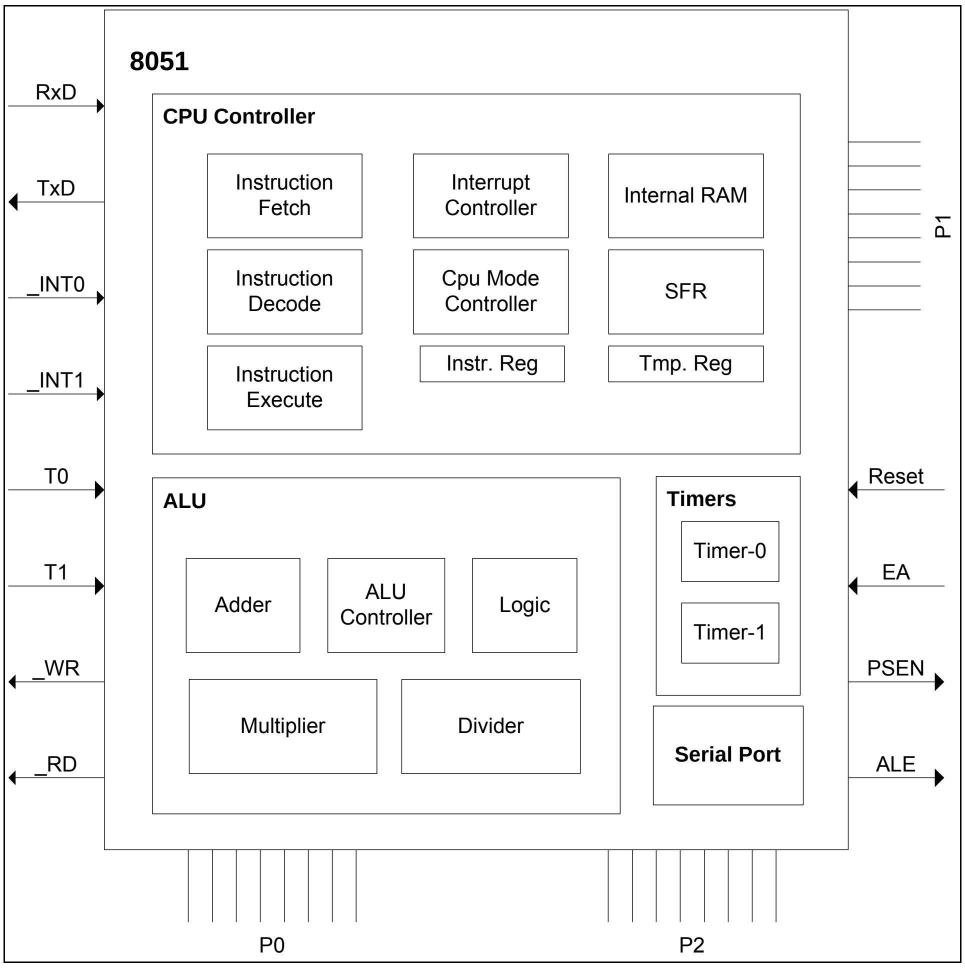


Figure 0.1 The Design of 8051 Core.

The partitioning of the problem into sub modules is another problem indeed. In general the final partition map of a certain design may differ from designer to designer. In this thesis, the top level 8051 core is divided into four main modules, namely, *CPU Executer* where the main state machine of the processor is implemented, *ALU* in which arithmetic and logical operations are done as well as shift operations, *Serial Port* and *Timer* modules as their names indicates their roles.

4.1 CPU Executer

CPU Executer is the primary module of 8051 processor core. Most important activities in this module are; execution of instructions, interrupt operations, accessing to the internal RAM blocks and I/O ports.

4.1.1 Main State Machine

In order to execute an instruction, first of all it should be fetched from the code memory with necessary operands if available. This property is modeled in a main state machine, which is driven directly by the oscillator clock. In early cycles of the state machine pre-fetch and fetch operations are applied for the current PC (Program Counter). In the next cycle, or CPU state in other words, instruction is read from the code memory. Current instruction is copied to the internal instruction register and then decoded.

If any operands are needed for the execution of the current instruction, they are read in the trailing cycles and copied to the temporary internal registers. After having the enough number of operands, execution takes places. Most of the time an execution includes an access to the ALU (Arithmetic Logic Unit) at correct CPU cycles. Then the result is copied to the destination, which is known as either a part of instruction itself or an operand.

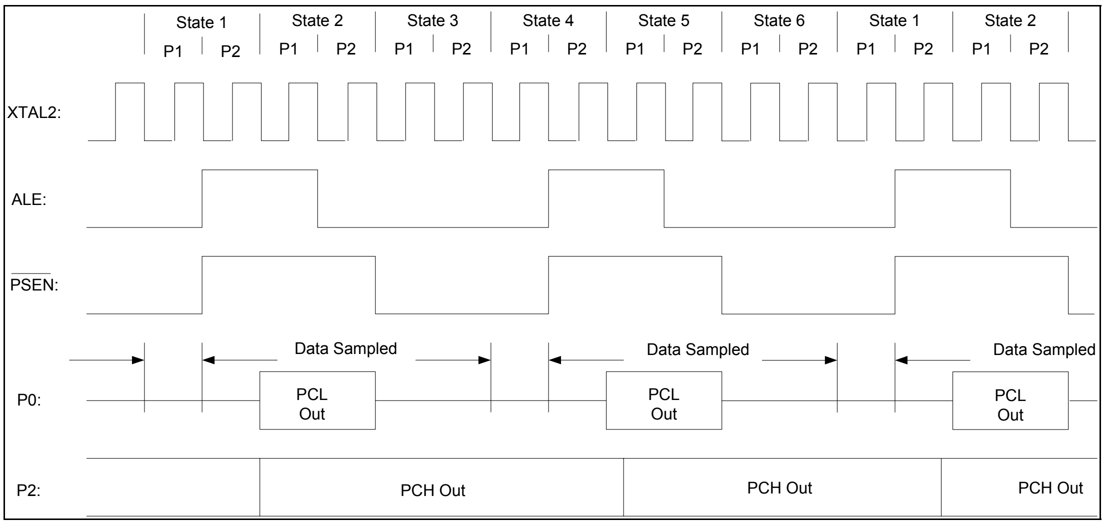


Figure 0.2 Cpu Signals During Code Read.

In 8051 CPU Core, one machine cycle consists of 12 oscillator cycles. During a typical machine cycle ALE and PSEN pulses twice, Port2 outputs high byte of the address twice, Port0 outputs low byte of the address twice and reads the program code from external memory twice [11].

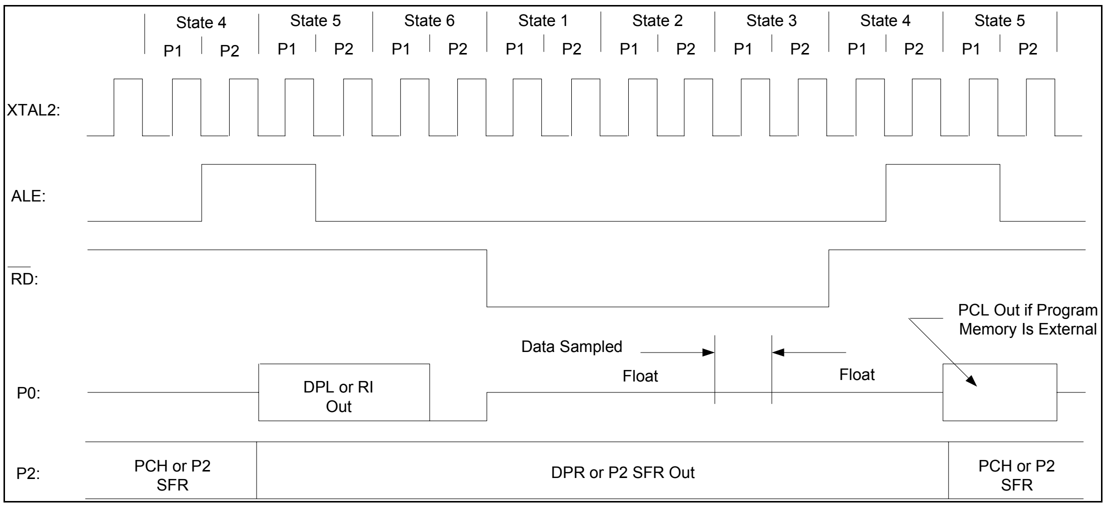


Figure 0.3 Cpu Signals During External Read.

In some instructions the second read byte is ignored and in some instructions one more byte may be read in the following machine cycle that will certainly occupy one more set of 12 oscillator cycles. Some instructions will last in one machine cycle and some in two. There are definitely two exceptions; multiplication and division instructions, they will last in four machine cycles that is 48 oscillator cycles.

The timing diagram of reading code bytes from external program memory is shown in Figure 4.2. In fact there is an exception for this timing diagram in `MOVX` instructions. While executing external data transfer instructions, instead of pulsing PSEN, external RD and WR signals are used to access external data memory. This variant of timing diagram is shown in Figure 4.3.

As, 8051 core has 48 oscillator cycles in the longest instruction, it makes sense to have a fundamental state machine having 48 basic states which is shown in the pseudo code below;

```
switch(cpu_state){

case 0 :    Fetch Instruction

case 1 :    Decode

case 2 :    Execute-1  &  Prefetch-1

case 3 :    Execute-2  &  Prefetch-2

case 4 :    Execute-3  &  Prefetch-3

case 5 :    Prefetch-4

case 6 :    Fetch Operand if necessary

case 7 :    Execute-1

case 8 :    Execute-2  &  Prefetch-1

case 9 :    Execute-3  &  Prefetch-2

case 10 :   Execute-4  &  Prefetch-3

case 11 :   Prefetch-4
            if (one cycle instruction) cpu_state = 0

case 12 :   Fetch Operand-2 if necessary
case 13 :   Decode

case 14 :   Execute-1  &  Prefetch-1

case 15 :   Execute-2  &  Prefetch-2

case 16 :   Execute-3  &  Prefetch-3

case 17 :   Execute-4  &  Prefetch-4

case 18 :   Fetch (Discard)

case 19 :   Wait

case 20 :   Prefetch-1

case 21 :   Prefetch-2

case 22 :   Prefetch-3

case 23 :   Prefetch-4
            if (two cycle instruction) cpu_state = 0

case 24 :   Wait
  .
  .
  .
case 43 :   Wait

case 44 :   Prefetch-1

case 45 :   Prefetch-2

case 46 :   Prefetch-3

case 47 :   Prefetch-4
            cpu_state = 0

}
```

One byte one cycle instructions, such as `INC A` are executed just after fetching the instruction from the code memory. In cases of theses types of instructions, execution at steps between 7 and 9 are not operated as it has already been carried out. The operand fetched at step 6 is also discarded for this case.

Two byte one cycle instructions such as `ADD A,#data` do not execute between steps 2 and 4, but simply runs after reading the necessary operand. Similarly three byte two cycle instructions like `MOV direct,#data` fetch both operand-1 and operand-2 and then execute. The four cycle operations such as `MUL A,B` and `DIV A,B` discards all read operands in between.

4.1.2 CPU Operation Mode

8051 CPU has six distinct operational modes, each having their own behavior;

- ***Reset Mode***: In this mode CPU is set to its preset values. Program Counter is set to 0x0000 where RESET interrupt vector lies. Internal cpu_state is set to 44 in order to have necessary prefetch cycles before fetching the instruction at state zero.

- ***Normal Mode****:* In this mode, main state machine runs step by step, instructions and operands are fetched and executed.

- ***Interrupt Start Mode****:* When an interrupt input pin changes state and an interrupt should be generated, this mode is executed. The reason for jumping interrupt start mode instead of jumping to interrupt service mode directly is to let CPU enough time to complete the current instruction. After having completed the current instruction the Program Counter is loaded with the vector of the pending interrupt and program branches to that location. The only exception is RETI instruction, which will postpone interrupt service routine for one instruction time.

- ***Interrupt Service Mode:*** Indeed, this mode is same as normal mode except for interrupts are disabled.

- ***Idle Mode:*** As the name implies CPU is idle in this mode. Main state machine is not running but interrupts, serial ports and timers are active. It may be possible to get out from idle mode in one of interrupt routines or by reset. Although CPU is not active in this state, all internal RAM and SFR contents are kept.

- ***PowerDown Mode:*** CPU is in power down mode, where all activities are stopped and only lasts with a CPU reset.

4.1.3 Internal RAM and SFR

Internal RAM and SFR are implemented as a memory array consisting of 256 registers each 8 bit wide. 8051 CPU has eight general purpose registers named as R0-R7. Moreover, for a flexible usage they are organized as four distinct set of registers. Two bits of PSW status register shows which set of general purpose registers are active. The first 32 bytes of the RAM are used for this register bank.

Table 0.1 Special Function Registers.

Name	Address	Definition	Name	Address	Definition
P0	0x80	Port-0 Register	P1	0x90	Port 1 Register
SP	0x81	Stack Pointer	SCON	0x98	Serial Port Control Register
DPL	0x82	Data Pointer Low part	SBUF	0x99	Serial Port Buffer
DPH	0x83	Data Pointer High part	P2	0xA0	Port 2 Register
PCON	0x87	Power Control Register	IE	0xA8	Interrupt Enable
TCON	0x88	Timer Control Register	P3	0xB0	Port 3 Register
TMOD	0x89	Timer Mode Register	IP	0xB8	Interrupt Priority Register
TL0	0x8A	Timer-0 Low part	PSW	0xD0	Processor Status Word
TL1	0x8B	Timer-1 Low part	ACC	0xE0	Accumulator
TH0	0x8C	Timer-0 High part	B	0xF0	B Register
TH1	0x8D	Timer-1 High part			

Next 16 registers are a type of special memory that can be addressable either bit by bit or as byte by byte. The memory region from 0x30 to 0x7F is used as a general purpose RAM area. System stack is also placed within this memory.

Special Function Registers (SFR) are placed at addresses 0x80 to 0xFF as shown in Table 4.1.

4.1.4 Interrupt Controller

To many system designers, interrupts are the most valuable part of a processor because almost every designed system needs external stimulation. In 8051 processor there are five interrupt resources that can be set either high or low priority. It is possible to enable or disable each of the interrupt resources as well as disabling or enabling all of the interrupts at a time. This is done by setting or resetting the relevant bits in the IE register which is shown in Table 4.2.

Table 0.2 Interrupt Enable Register.

BIT	SYMBOL	BIT ADDRESS	DESCRIPTION
IE.7	EA	0xAF	Global Enable/disable
IE.6	-	0xAE	Undefined
IE.5	-	0xAD	Reserved
IE.4	ES	0xAC	Enable Serial Port interrupt
IE.3	ET1	0xAB	Enable Timer-1
IE.2	EX1	0xAA	Enable External Interrupt-1
IE.1	ET0	0xA9	Enable Timer-0
IE.0	EX0	0xA8	Enable External Interrupt-0

Two of the interrupts may have external connections; namely External Interrupt 0 and 1. These interrupt pins may be selected either level sensitive or edge sensitive by setting the relevant bits in TCON registers.

Two other interrupt resources are on chip timers. When the timers are activated they generate interrupts at the states where the timer mode indicates. These interrupts are also enabled/disabled and prioritized by setting the relevant bits in TCON and IE registers The other interrupt resource is also internal and pointed to the Serial Port of the 8051 chip. All these interrupts are prioritized by IP register as shown in Table 4.3.

Table 0.3 Interrupt Priority Register.

BIT	SYMBOL	BIT ADDRESS	DESCRIPTION
IP.7	-	-	Undefined
IP.6	-	-	Undefined
IP.5	-	0xBD	Reserved
IP.4	PS	0xBC	Priority for Serial Port interrupt
IP.3	PT1	0xBB	Priority Timer-1
IP.2	PX1	0xBA	Priority External Interrupt-1
IP.1	PT0	0xB9	Priority Timer-0
IP.0	PX0	0xB8	Priority External Interrupt-0

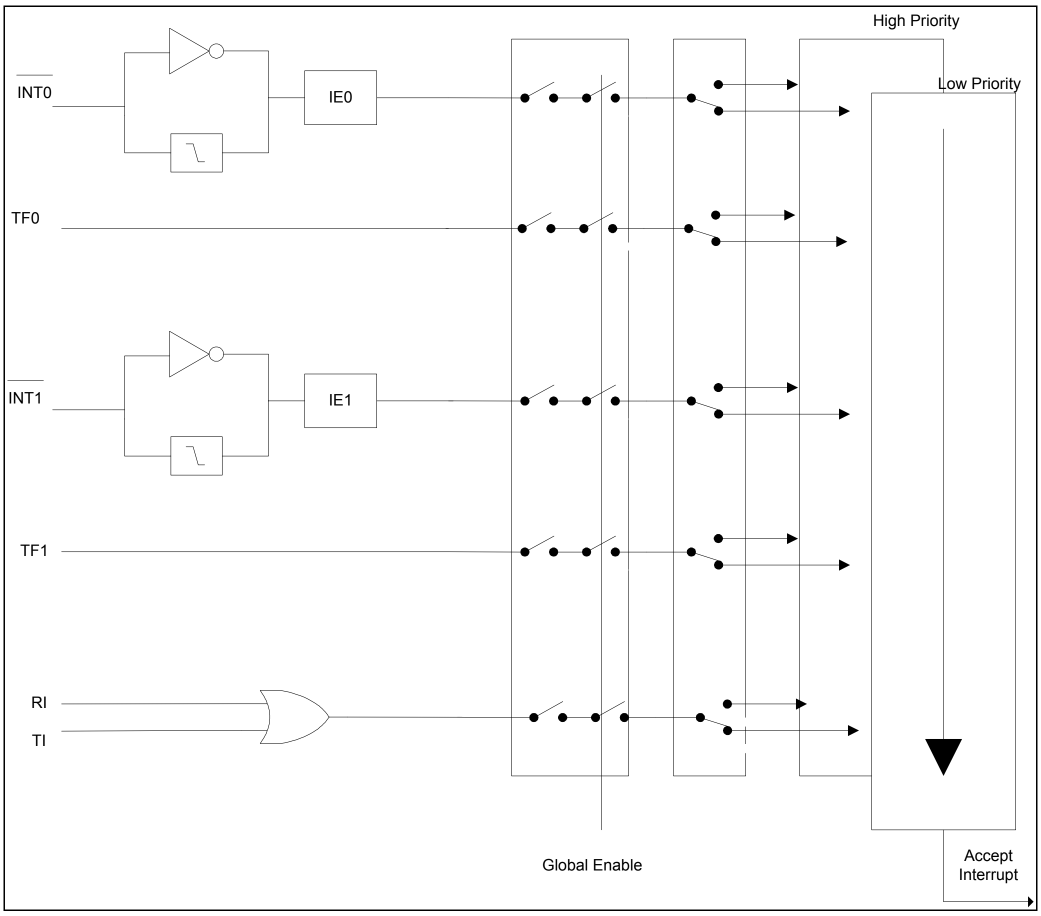


Figure 0.4 Interrupt Priority Scan.

At every clock cycle interrupt pins are scanned as shown in Figure 4.4, any change that may trigger an interrupt event initiates Interrupt Start Mode, which will watch the suitable state to start interrupt mode. The basic rules to have an interrupt are; Global Interrupt Enable should be enabled as well as particular Interrupt Enable, then the edge or level event should occur and the interrupt has appropriate interrupt priority.

When an interrupt event occurs, the current Program Counter will be saved on the stack and updated with the respective vector address of the interrupt. At this time interrupt service routine is started with fetching the instruction where interrupt vector points. This vector table is shown in Table 4.4.

Table 0.4 Interrupt Vector Table.

INTERRUPT	FLAG	VECTOR ADDRESS
System Reset	RST	0x0000
External 0	IE0	0x0003
Timer 0	TF0	0x000B
External 1	IE1	0x0013
Timer 1	TF1	0x001B
Serial Port	RI or TI	0x0023

4.1.5 I/O Ports

In 8051 chips, there four general purposes I/O ports, which can be used for special purposes as well as address and data output/input buses. In most of the designs P0 and P2 are used for address and data bus. 8051 processor can address up to 65536 bytes of external memory by using 16 bit address bus and 8 bit wide data bus. P2 is dedicated to high byte of the 16 bit address; P0 is shared between data and low byte of address. This mechanism is used with Address Latch Enable pin which will indicate the low byte address is outputted at P0. Most of the time there is an external circuit to handle this latching operation if external RAM or ROM is being used.

Sharing of Port-0 between data and address is coded in the main state machine functions of the 8051 core. As explained in 4.1.1 Main State Machine section, ALE, PSEN and P0 outputs are driven such that external circuit captures the necessary low byte address and declares it when needed.

Port1 is a general purposes eight bit wide bidirectional port. Port3 is also a bidirectional general purpose port. Moreover it has some alternate functions. Read, write, ALE , PSEN, reset, EA and serial RX TX signals are outputted by this port

4.2 ALU Module

Arithmetic Logic Unit (ALU) is designed as a separate module with in the 8051core. It has the standard signal and command interface so that it can be interchanged by any standard ALU unit in the future, although there is no technical requirement for that. This is shown in Figure 4.5.

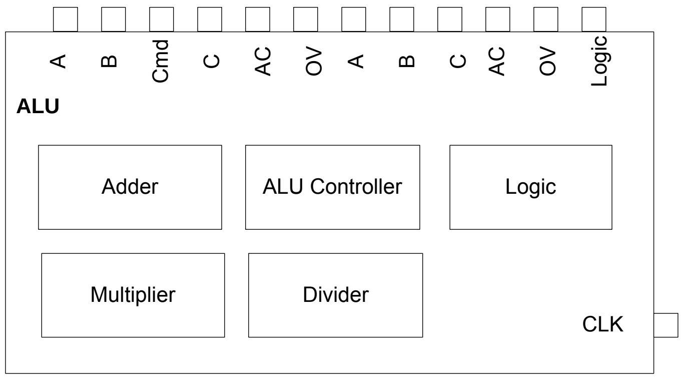


Figure 0.5 Design of ALU (Arithmetic Logic Unit).

CPU Executer commands ALU with the standard command set given in Table 4.5.

Table 0.5 Standard ALU Commands.

Code	Command	Description
0x00	Alu_INC_DEC_EQZ	Increment 8-bit operand-0 by 1 Decrement 8-bit operand-1 by 1 Logic is 1 if operand-0 is equal to 0 otherwise 0
0x01	Alu_ADD	Twos complement addition of two 8 bit operands
0x02	Alu_ADDC	Twos complement addition with carry of two 8 bit operands
0x03	Alu_SUBB	Twos complement subtraction with borrow of two 8 bit operands
0x04	Alu_MUL_1	Start of 8 bit multiplication of two operands
0x05	Alu_MUL_2	Enable of 8 bit multiplication
0x06	Alu_DIV_1	Start of 8 bit division of two operands
0x07	Alu_DIV_2	Enable of 8 bit division
0x08	Alu_ANL_ORL	Bitwise logical AND operation of two operands Bitwise logical OR operation of two operands
0x09	Alu_XRL_SWAP_NE	Bitwise logical XOR operation of two operands Swap nibbles of second operand Logic is 1 if two operands not equal else 0
0x0A	Alu_ROL_ROR_NEQZ	Rotate the 8-bit operand to left by one bit. Rotate the 8-bit operand to right by one bit. Locig is 0 if the operand is equal to 0, else0.
0x0B	Alu_ROLC	Rotate the 8-bit operand to left by one bit through carry.
0x0C	Alu_RORC	Rotate the 8-bit operand to right by one bit through carry.
0x0D	Alu_XCHD	Exchange low-order nibbles of two 8-bit operands.
0x0E	Alu_CLR_CPL	Clears the 8-bit operand as 0. Logical complement of each bit in 8-bit operand.
0x0F	Alu_DA	Decimal adjust operation for addition.
0x10	Alu_INC16_BNS	Increment 16-Bit operand by 1. Logic is if the specified bit in the operand is zero, else 0.
0x11	Alu_CLRC	Clear carry flag.
0x12	Alu_CLRB_CS	Clear the specified bit in the 8-bit operand. Logic is 1 if carry is 1, else 0.
0x13	Alu_SETC	Sets the carry flag.
0x14	Alu_SETB_CNS	Sets the specified bit in the 8-bit operand. Logic is 0 if carry is 0, else 1.
0x15	Alu_CPLC	Complement the carrr flag.
0x16	Alu_CPLB_MOVBC_BS	Complement the specified bit in the 8-bit operand. Store the carry flag into the specified bit in the 8-bit operand. Logic is the specified bit in the operand is 1, else 0.
0x17	Alu_ANDCB	Logical 'AND' of the carry with the specified bit.
0x18	Alu_ANDCNB	Logical 'AND' of the carry with the complement of the specified bit.
0x19	Alu_ORCB	Locical 'OR' of the carry with the specified bit.
0x1A	Alu_ORCNB	Locical 'OR' of the carry with the complement of the specified bit.
0x1B	Alu_MOVCB	Move the specified bit to the carry
0x1C	Alu_BNSC	Set logic as 0, clear the bit if the bit is 0 else set logic to 0
0x1D	Alu_PASS	Pass the operands without change
0x1E	Alu_LT_GTE	Set carry if operand_1 is greater than operand_2 else clear carry
0x1F	Alu_NOP	No Operation for ALU

4.2.1 Addition and Subtraction

Addition and subtraction operations are handled with in the Arithmetic Logic Unit. For both operations the two 8 bit operands of ALU are used, carry input is also used where needed. For addition and subtraction operations no special circuit used but instead regular plus and minus signs are used in order to let the synthesizer generate necessary addition and logic.

4.2.2 Logic and Shift Operations

Similar to addition operations, logic and shift operations are also handled by SystemC, so no special circuit is used. During the simulation, SystemC compiler handles these operations and at RTL conversion, the synthesizer generates necessary logic for these operations.

4.2.3 Multiplication

For 8 bit multiplication the regular successive addition and shifting algorithm is used [12]. The main idea is same with the multiplication with paper and pencil. The multiplication is handled at eight steps since there are 8 bits to multiply. Before starting the operations, a temporary 16 bit register representing the sum is reset to zero. Then the most significant bit of the multiplier is selected, if this bit is zero no operation is done, if it is one, the sum is added by the multiplicand. After then the sum is shifted to left by one. Then the second most significant bit of the multiplier is selected and tested, if it is one, the sum is incremented by the multiplicand. After then the sum is shifted to left by one. This goes down to the least significant bit of multiplier. At the end, the 16 bit temporary register holds the multiplication of the two unsigned 8 bit integers.

4.2.4 Division

Division is based on the following formula;

$D = A \times B + R$

The problem can be viewed as finding B and R from given D and A. From this point of view, the problem is reduced to the inverse of multiplication and can be thought as eight successive subtractions.

At the beginning a 16 bit temporary register holds the 8 bit dividend, the division is set to zero and the divisor is left untouched. At first step, 16 bit dividend is shifted to left by one and high byte is compared to divisor, if it is equal or greater than divisor, the most significant bit of division is set to one, and high byte of dividend is decremented by divisor. Then at the second step 16 bit dividend is shifted left once more and the high byte is compared to divisor, if it is greater than or equal to divisor, then the second most significant bit of division is set to one and the high byte of dividend is decremented by divisor. This continues down to the least significant bit of division. At that time, division holds the result, the high byte of dividend holds the remainder.

4.3 Serial Port Controller

Serial Port of the 8051 CPU is designed as a separate module in order to make the design modular. This serial port is hundred percent compatible with the original one for compatibility reasons, although some serial modes are not likely to be used in today’s new designs. The benefit of using SystemC as a design language shows up here, in future use of this processor core, those unused modes of the serial port can easily be taken from the design.

There are four fundamental functionalities in serial port module. Two of them are receive shift register and transmit shift register functions. The other one is the clock generation, there are two different possibilities as baud rate generators, one is to get the clock from the CPU oscillator the other is to get the ticks from the internal timer. The last function is arranging the serial operation mode.

8051's serial port is configured via the Serial Control Register, SCON at 99. The transmit data or receive data is taken or given by using the SBUF register. This register is indeed two separate registers although they are seen as a single combined register.

As given in Table 3.6, the four operation modes are;

- Shift register at fixed rate oscillator frequency / 12
- 8 bit UART at variable rate
- 9 bit UART at fixed rate
- 9 bit UART at variable rate selected by Timer

In mode-0, the Shift Register mode, there is only one possible baud rate which is set by oscillator frequency divided by 12. Receiver of this mode is handled by a state machine composing of 18 internal states. The state machine starts when REN bit is set in SCON register, after that the receiver data pin of the 8051 core is sampled at half clock cycle time and the register is shifted by one. The transmitter state machine having 18 internal states will start after SBUF is written, then at each cycle one more of the serialized bits is sent via the transmit data pin of serial port of 8051 core. This will continue until all 8 bits are sent. In this mode, the baud rate is fixed by oscillator /12.

In either 8-bit or 9-bit UART modes, a more complicated scheme is followed. First of all, before sending the data, a start bi is asserted in the transmit line, after one clock cycle the serialized bits are started to be sent one by one. After completing the 8 bits, one stop bit should be asserted to the transmit line. All these operations are controlled by a 12-state internal state machine. At each state, the suitable clock cycle is waited and the serialized bit is prepared to send.

For UART receive operations, the line is always polled by the serial port receive module. If any start condition is caught, which is a high to low edge, the receiver gets ready to start. Before starting to receive line is double checked by false start bit logic. If this test is also passed, then receive operation is started, one bit is sampled in the middle of the successive each cycles.

The fixed rate may be either the oscillator frequency /64 or oscillator frequency / 32 which is set by SMOD bit in PCON register. Similarly the variable rate may be divided either by 32 or by 16 by setting or resetting the SMOD bit .

4.4 Timer Operations

In 8051 core there are two timers operate independently. The two timers are designed in one single Timers module to have it a handy module for 8051 core. As in the case for Serial Port module, Timer modules also have some modes which are not suitable for new designs but implemented with in the module for compatibility reasons. In future, these un needed functionalities of timers can be drawn back without affecting the operation of 8051 core.

Operation of timers are explained in *3.4. 8051 Timers* section, The module designed here is hundred percent compatible with the original 8051 timer. This functionality is achieved by incrementing the relevant timer registers upon the clock events. The incremented register is tested whether it makes an overflow or not concerning the current operation mode. If overflow occurs the flags are set and internal interrupt is generated.

VERIFICATION OF 8051 DESIGN

In order to verify the designed 8051 core, a test bench is used. This is shown in Figure5.1. In real life designs, the 8051 CPU is not used alone, conversely it is used by some peripheral components such as a ROM to have the software code, a RAM for user data area, an address latch logic for handling multiplexed data and address buses. In real life systems some asynchronous events are happing, and these are tied to interrupt pins of the CPU to process them.

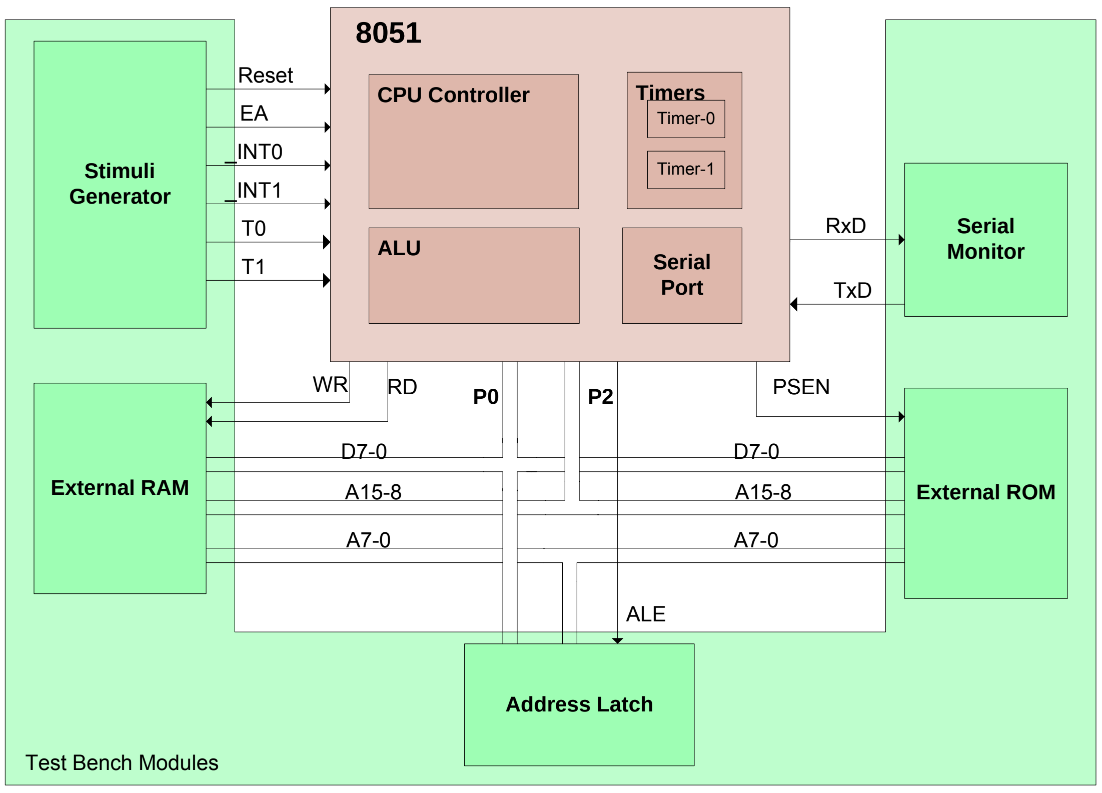


Figure 0.1 Test Bench for 8051 Core.

The test bench for the 8051 core is designed such that almost all real life components are available and all possible scenarios can be run. For this purpose a ROM and a RAM modules are designed in SystemC. To convert the multiplexed data and address buses to separate busses an Address Latch module is also created. To generate the interrupts and reset inputs for the processor to run in a more complicated environment, a stimulator module is used.

Since, 8051 core is the module under design, it is not a good idea to have printf or any other debugging statements inside that module, instead a serial monitor module is created. This module has two missions, one is to test the serial port operations of the 8051 core, the other is informing the user about the test results.

All these modules are the hardware part of the design indeed. The processor should have some software to be loaded and run. This is supplied by a test program which has almost all possible 8051 assembly instructions called at least once. It also has some interrupt service routines included to test the interrupt hardware. This software program is developed using Keil 8051 development environment outside of SystemC environment. This is the original code development tool that 8051 programmers use. The test code is written in assembly, compiled and linked with the tool mentioned above, and a hex file is achieved. When this hex file is loaded to an EPROM or ROM in a real life system, the 8051 processor grabs it and runs it. Here it works the same, that hex file is shown to the ROM module in SystemC, at the beginning this module reads the hex file parses it, and locates the content to an internal memory. During the run time each time 8051 core try to read external ROM, this module gives the defined byte. All these operations happen just as in real hardware happens, that means all signals are hundred percent compatible with the real hardware signals as shown in the coming sections.

During the development of the 8051 core with SystemC, the test bench mentioned above, is used with a test software to run on the microcontroller to span all possible instructions. The aim of this software is to verify the design by analyzing both the signals traces and register contents as each type of instruction is executed. After the completion of the design of 8051 core, a standard 16-bit CRC software is loaded to the microcontroller in order to verify the execution of a program as a whole.

5.1 Verification of Generated Signals

In order 8051 processor to read external code memory, it should generate a set of signals with correct transactions. These are, ALE, PSEN, RD and WR signals which are used for, address latch enable, program memory select, read and write functions respectively. With SystemC, it is possible to trace selected signals, so that the designer can evaluate the design. To do so, a set of function calls are made at the beginning of the code, as shown below;

```
sc_trace(tf, clk.signal(), "clock");
sc_trace(tf, s_p0, "p_0");
sc_trace(tf, s_p2, "p_2");
sc_trace(tf, s_rx, "read");
sc_trace(tf, s_wx, "write");
sc_trace(tf, s_ea, "ea");
sc_trace(tf, s_ale, "ale");
sc_trace(tf, s_psen, "psen");
sc_trace(tf, s_rxd, "serial_rx");
sc_trace(tf, s_txd, "serial_tx");
sc_trace(tf, s_int0, "Ext_Int_0");
sc_trace(tf, s_int1, "Ext_Int_1");
```

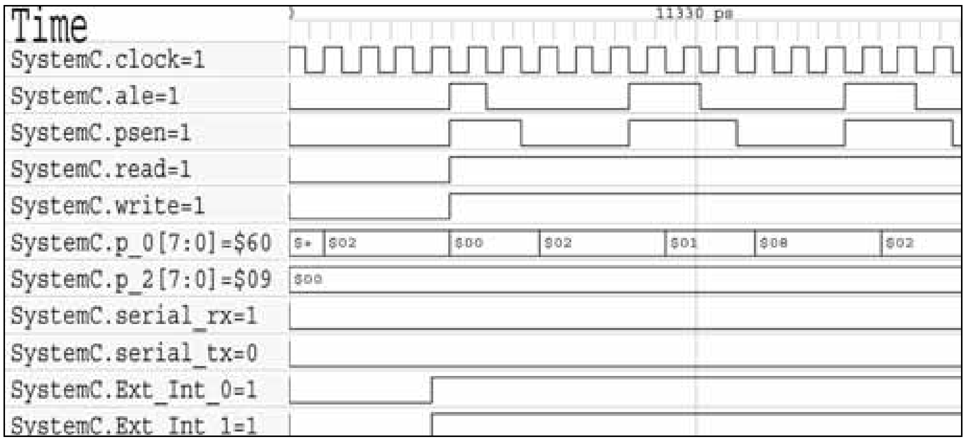


Figure 0.2 Basic CPU Signals.

As SystemC is an open source programming language, the SystemC library is both open source and free to use. When the library once added to the design, after the project is compiled and linked as a regular C++ project, the output executable works as the fundamental simulator for SystemC language. When this executable runs, it will generate a standard trace file and this file can be analyzed with both commercial and open source free tools. The basic CPU signals are traced to verify the CPU operation as shown in Figure 5.2.

8051 core is using multiplexed data and address buses to reach external memories. There are only two 8 bit wide ports to handle both 16 bit wide address and 8 bit wide data. The first step is declaring the address; CPU outputs both low and high bytes of the address then changes Address Latch Enable (ALE) from high to low to tell the peripheral circuit that low part of the address is available at Port-0. When ALE signal makes a negative edge the output address latch circuit stores the low part of the address. Then CPU lowers the _PSEN signal in order to select the program memory, at this time Port-2 is holding the high byte of the address, latch circuit is outputting the recently stored low byte of the address and Port-0 is the data byte put by the external program memory. Data byte is read from Port-0 to either internal instruction or operand register depending on the internal CPU cycle. Reading of the external code bytes as shown in Figure 5.3, continues just like this as far as no MOVC or MOVX instructions are executed.

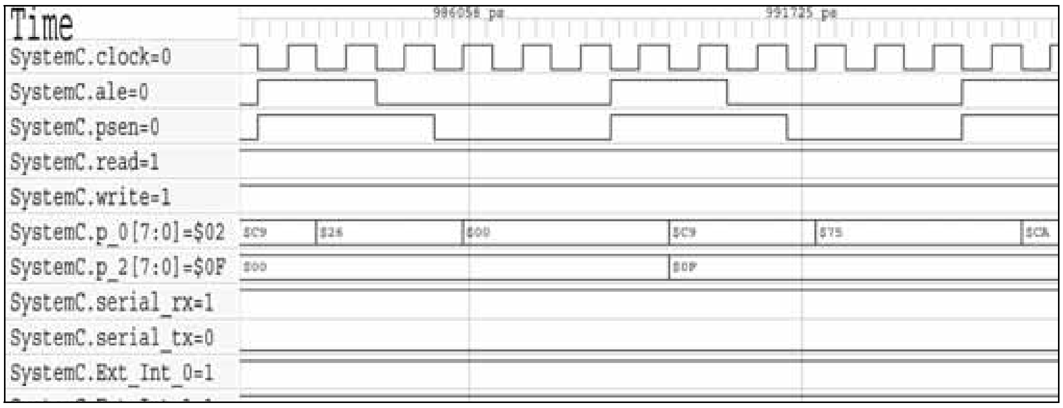


Figure 0.3 External Code Read.

As explained above, external code read is the very fundamental operation of 8051 processor, the verification of these signals is done by comparing the SystemC simulation output signals and the original 80C51 processor's code read signals which is shown in Figure 5.4, copied from Philips 80C51 data sheet[14].

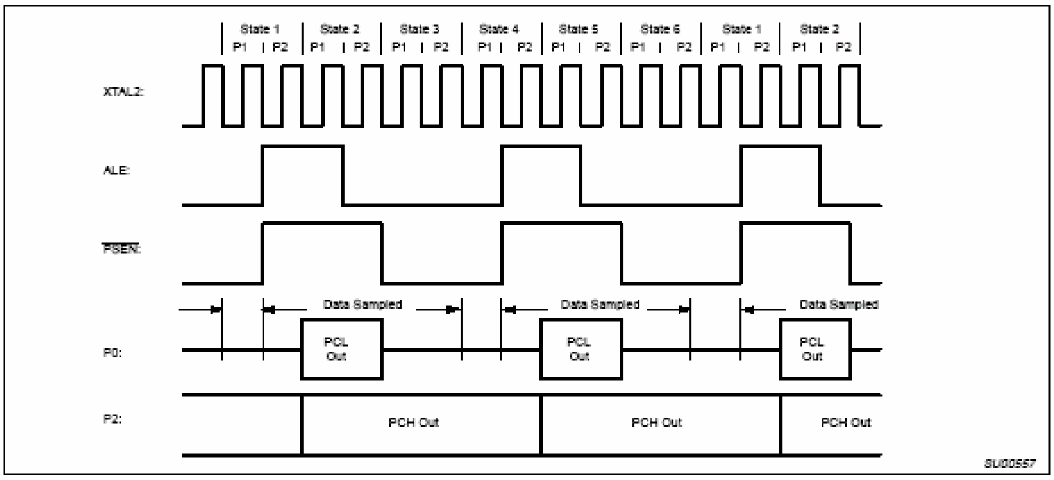


Figure 0.4 8051's External Code Read.

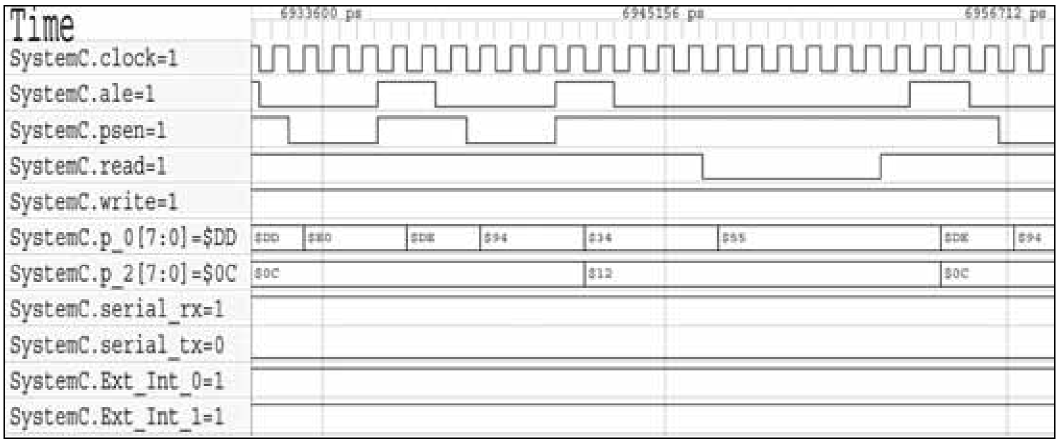


Figure 0.5 External Data Memory Read.

8051 CPU can distinguish between the program memory and data memory chips both are externally connected. In order to read the program memory, which is typically a ROM, 8051 uses Program Select (_PSEN) control signal, to read or write to external data memory, that is a static RAM, 8051 uses Read (_RD) and

Write (_WR) signals in combination with the _PSEN signal. There is only one type of external data read and write instruction in 8051 assembly language, the MOVX command. It either reads a data byte from the external RAM or writes a data byte. Both cases are shown in Figure 5.5 and Figure 5.7.

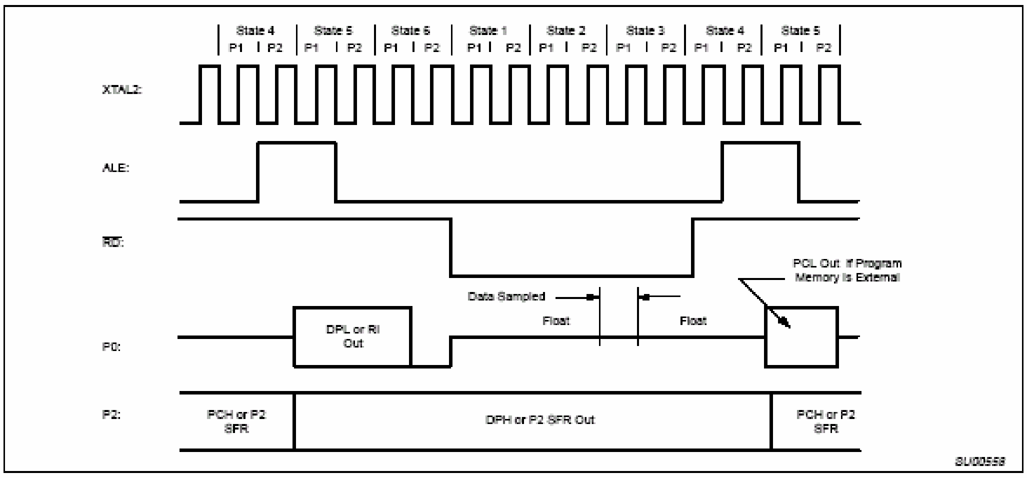


Figure 0.6 8051's External Data Memory Read.

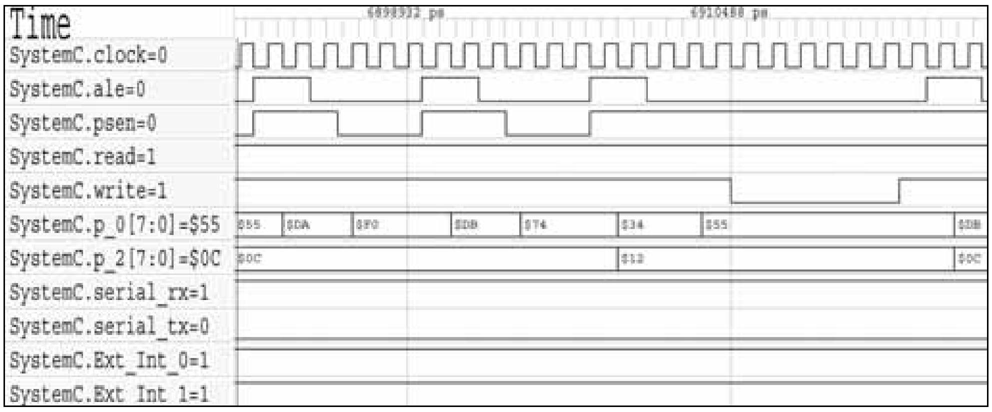


Figure 0.7 External Data Memory Write.

Our 8051 processors external read and write signal diagrams are compared with the original 80C51's read and write signal diagrams which are given in Figure 5.6 and Figure 5.8, which are copied from Philips' 8051 data sheet. As easily seen, all the read, write and port signals are compatible to the original ones.

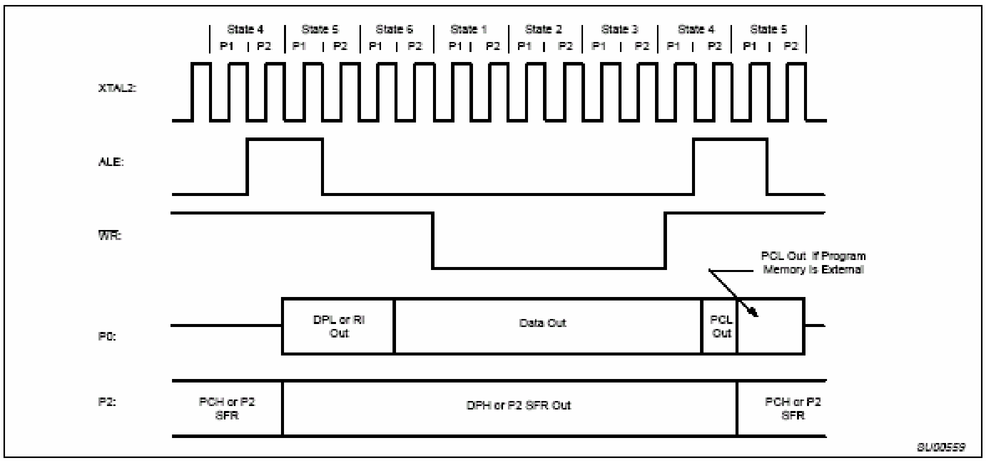


Figure 0.8 8051's External Data Memory Write.

5.2 Verification of ALU

8051's ALU is designed as a separate module and connected to the core with a set internal data buses. The 8051's Executer Module calls ALU module where an arithmetic or logical instruction is executed. This interface has two 8 bit wide operand, two 8 bit wide result, one 5 bit wide ALU command busses, and carry, auxiliary carry, overflow, new carry, new auxiliary carry and new overflow control and data signals. Executer module puts the necessary operands to the operand data paths and the command to the command port with the supplementary carry signals. In the next clock cycle, if ALU command is different from the ALU_NOP instruction, then ALU module executes the command and stores the result to the result ports with carry and overflow signals. This flow is shown in Figure 5.9.

In the figure above, ALU_INC_DEC_EQZ operation is executed. Most of the ALU commands define more than one command. As an example, this command defines both increment and decrement instructions and also a comparison to zero instruction. The operand given in alu_a is incremented by one and is put to result_a port, similarly it is decremented by one and is put to result_b port, moreover a logic output generated depending on the comparison of operand_a and zero.

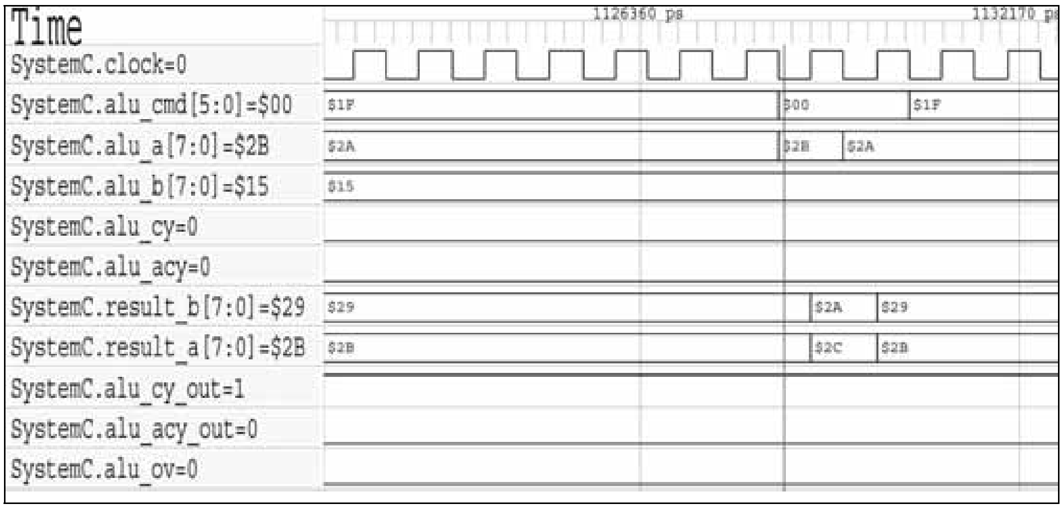


Figure 0.9 Basic ALU Signals.

From the Executer's point of view; it puts 2Bh unsigned integer value to the operand port, ALU_INC_DEC_EQZ, (00h) command to the command port, and gets the 2Ch from one of the result operands and 2Ah from the other. At this point Executer module knows which result is the valid one since it has called the ALU to do either an increment instruction or a decrement instruction.

Multiplication operation is somewhat more complex than other arithmetic operations. A multiplication operation is indeed eight successive addition and shift operations, so it is not possible to do it in a single clock cycle. Unlike other operations, for multiplication, Executer module puts the operands and MUL_1 command to the ALU module to start the operation. After a predefined time period, Executer module asks for the result with an other command MUL_2. At this time it reads the results from the dedicated result ports. Signal diagram of multiplication is shown in Figure 5.10.

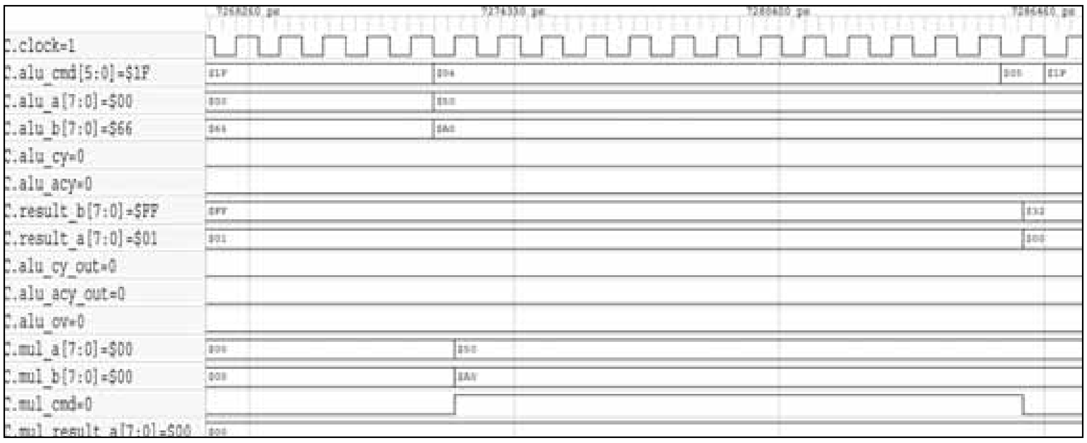


Figure 0.10 ALU Multiplication.

As shown in Multiplication signal diagrams, Executer puts, 50h and A0h unsigned integer values to the operand ports and ALU_MUL_1 (04h) command to the command port. Beginning from the next clock cycle ALU module calculates this operation. After 12 clock cycles have passed, Executer module asks for the result by putting ALU_MUL_2 (05h) command. The ALU module stores the result to the result ports as 32h and 00h as shown in the figure. This is the correct result of the multiplication of 50h and A0h.

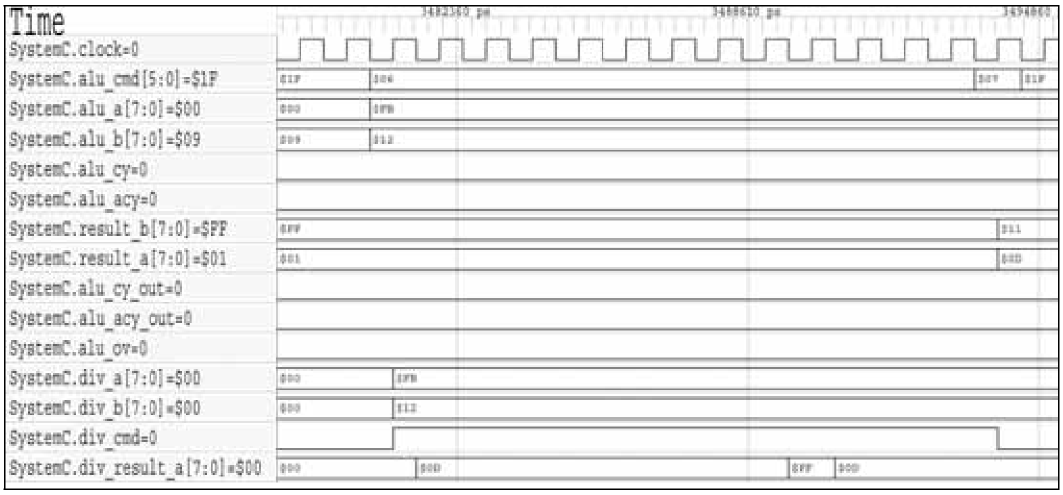


Figure 0.11 ALU Division.

Similar to multiplication operation, division is also a complicated instruction which is composed of eight successive compare subtract and shift operations. The timing diagrams between the Executer and ALU modules, is shown in Figure 5.11. As shown in division signal diagrams, Executer puts, FBh and 12h unsigned integer values to the operand ports and ALU_DIV_1 (06h) command to the command port. Beginning from the next clock cycle ALU module calculates this division. After 12 clock cycles passed, Executer module asks for the result by putting ALU_DIV_2 (07h) command. Then the ALU module stores the result to the result ports as 0Dh and 11h as shown in the figure. This is the correct result of the division of FBh by 12h, result is 0Dh and the remainder is 11h.

5.3 Arithmetic Instructions

8051's arithmetic instructions include add, subtract, increment, decrement, multiply and divide operations. There are a variety of these instructions depending on operands that are used. In order to verify execution of these instructions, one or more of the registers are set to predefined values, then the operation is executed with necessary operands and the result is compared to the solution.

In case of an error, error number is saved to Port1 which is not currently used by any other function, then jumps to an error reporting location where the error code is sent via serial port. In this section, the verification processes are explained by pseudo codes for simplicity, the complete assembly codes are given in Appendix C

The verified addition operations are;

```
ADD A,Rn
ADD A,direct
ADD A,@Ri
ADD A,#data
ADDC A,Rn
ADDC A,direct
ADDC A,@Ri
ADDC A,#data
```

In order to verify these operations, the following sample code is used;

```
MOV   A,#10        ; Load  A  by 10
MOV   R0,#10       ; Load  R0 by 10
ADD   A,R0         ; A = A + R0
SUBB  A,#20        ; A = A - 20
JZ    DONE_1       ; Goto DONE_1 if A == 0
MOV   P1,#1        ; Error Code
JMP   FAILED       ; Goto End
```

ADDC type instructions are using the carry input so for the verification of those instructions, carry is set to one before addition is done so that adding of the carry is also verified. .

The verified subtraction operations are;

```
SUBB A,Rn
SUBB A,direct
SUBB A,@Ri
```

In order to verify these operations, the following sample code is used;

```
MOV   PSW,#0       ; Clear Status Register
MOV   A,#10        ; Load  A  by 10
MOV   R0,#10       ; Load  R0 by 10
SUBB  A,R0         ; A = A - R0
JZ    DONE         ; Goto DONE if A == 0
MOV   P1,#2        ; Error Code
JMP   FAILED       ; Goto End
```

The verified increment operations are;

```
INC A
INC Rn
INC direct
INC @Ri
INC DPTR
```

To verify increment operations, the following sample code is used;

```
MOV   PSW,#0       ; Clear Status Register
MOV   A,#10        ; Load  A  by 10
INC   A            ; Increment A by 1
SUBB  A,#11        ; A = A - 11
JZ    DONE         ; Goto DONE if A == 0
MOV   P1,#2        ; Error Code
JMP   FAILED       ; Goto End
```

The verified decrement operations are;

```
DEC A
DEC Rn
DEC direct
DEC @Ri
```

To verify decrement operations, the following sample code is used;

```
MOV   PSW,#0      ; Clear Status Register
MOV   A,#10       ; Load  A  by 10
DEC   A           ; Increment A by 1
SUBB  A,#9        ; A = A - 9
JZ    DONE        ; Goto DONE if A == 0
MOV   P1,#4       ; Error Code
JMP   FAILED      ; Goto End
```

Multiplication and division operations are verified as in the sample code below;

```
MOV   PSW,#0      ; Clear Status Register
MOV   A,#50h            ; Load  A  by 80
MOV   A,#A0h            ; Load  A  by 160
MUL   AB          ; Result = 3200h, A=00h B=32h
JNZ   ERROR       ; Goto ERROR if A is not zero
MOV   A,B
SUBB  A,#32h            ; A = A - 32h
JZ    DONE        ; Goto DONE if A == 0
MOV   P1,#5       ; Error Code
JMP   FAILED      ; Goto End
```

There is one very special instruction in 8051 processor, the DA , decimal adjust instruction. This will adjust two nibbles of a byte such that each representing BCD values. DA instruction is verified as shown in the pseudo code below.

```
MOV   PSW,#0      ; Clear Status Register
MOV   A,#80h      ; Load  A  by 80h
ADD   A,#99h      ; A = A + 99h
DA    A           ; Decimal Adjust
SUBB  A,#78H      ;Will clr ACC if C set
JZ    DONE        ; Goto DONE if A == 0
MOV   P1,#4       ; Error Code
JMP   FAILED      ; Goto End
```

5.4 Logical Instructions

Logical and shift operations in 8051 processor are verified with the same method used for arithmetic operations. The following and, or, exclusive or, clear, complement, rotate left, rotate right, and swap instructions are verified

```
ANL A,Rn
ANL A,direct
ANL A,@Ri
ANL A,#data
ANL direct,A
ANL direct,#data
ANL C,bit
ANL C,/bit
ORL A,Rn
ORL A,direct
ORL A,@Ri
ORL A,#data
ORL direct,A
ORL direct,#data
ORL C,bit
ORL C,/bit
XRL A,Rn
XRL A,direct
XRL A,@Ri
XRL A,#data
XRL direct,A
XRL direct,#data
CLR A
CLR C
CPL C
CLR bit
CPL bit
CPL A
RL A
RLC A
RR A
RRC A
SWAP A
```

5.5 Data Transfer Instructions

There are basically three types of data transfer operations, first one is MOV operations which moves data between registers, internal RAM, external RAM and also code memory, second type is XCH operations which will exchange values of two data operands and lastly the push/pop operations that saves and restores data values to/from stack. Verification of these instructions is done by the same method with arithmetic instructions. The following instructions are verified

```
MOV A,Rn
MOV A,direct
MOV A,@Ri
MOV A,#data
MOV Rn,A
MOV Rn,direct
MOV Rn,#data
MOV direct,A
MOV direct,Rn
MOV direct,direct
MOV direct,@Ri
MOV direct,#data
MOV @Ri,A
MOV @Ri,direct
MOV @Ri,#data
MOV C,bit
MOV bit,C
MOVX @DPTR,A
MOVX A,DPTR
MOVC A,@A+DPTR
MOVC A,@A+PC
```

In order to verify the data transfer operations, a predefined byte is first loaded to one of the registers, then the content of this register is moved to another register with the MOV instruction under test. Then either an addition or a subtraction is done to have zero content in the register. Now it is time to compare the register with zero, and jump to either error location or test passed location. The whole assembly code for testing these instructions are given in Appendix C

5.6 Flow Control Instructions

Flow control instructions like call, jump and return are indeed already tested with in tests of other instructions because every fragment of a test code need to jump some certain location that will indicate whether the test is correct or not. But more complex instructions like “compare and jump if equals to zero” are verified with the code given in Appendix C. Verified instructions are given below;

```
AJMP direct
CJNE A,direct,rel
CJNE A,#data,rel
CJNE Rn,#data,rel
CJNE @Ri,#data,rel
DJNZ Rn,rel
DJNZ direct,rel
JB bit,rel
JBC bit,rel
JC rel
JMP @A+DPTR
JNB bit,rel
JNC rel
JNZ rel
JZ rel
LJMP direct
```

5.7 Interrupts

There are five interrupt resources in 8051 CPU, two external, two timers and one from serial port events. In this thesis a test bench is designed in SystemC design language, as well as the 8051 CPU core. The stimulator module generates necessary test signals such as interrupt and reset, to 8051 processor. External interrupt is generated by this stimulator module, when interrupt event occurs, program counter jumps to interrupt vector after completing the current instructions

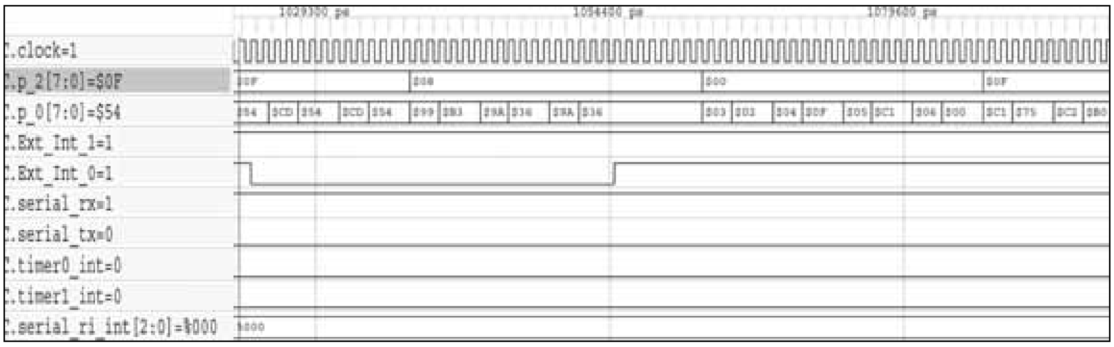


Figure 0.12 External Interrupt Operation.

A fragment of the trace output file of SystemC simulation of 8051 core is given in Figure 5.12. As easily can be followed, before the interrupt event occurs, 8051 core is fetching instructions at address 0x0899 and 0x089A and executing those instructions. At that time an interrupt event occurs, the CPU completes the current instruction and then jumps to the interrupt vector table. Since External Interrupt-0 occurred, the PC is loaded with 0x0003 and the instruction is fetched there. As seen from the external Port-0 and Port-2 signals, CPU reads the jump instruction with operands 0x0F and 0xC1 which is the address of relevant interrupt service routine. In the next machine cycle PC is loaded with 0x0FC1 and 8051 fetches the instruction over there.

In order to test the interrupt operation a test code is written such that, it changes a predefined register of the 8051 CPU with a predefined value. Since interrupt is asynchronous to the running program, the exact time is not known by the software.

While CPU is running the test software for other instructions, an interrupt is generated by the external circuit, then the program branches to the interrupt service routine executes the commands there, and finally return back to the test code as if it had not been interrupted. Before ending the program the predefined register is checked whether it is been altered by the Interrupt Service Routine (ISR) or not. If the expected value is read from the register the interrupt test is passed otherwise error code is given. This test code is given in Appendix C.

5.8 Serial Port

Two different test programs are prepared for testing the serial port operations; one of them is used for verifying the serial transmit operations, the other is used for serial receive operations. A Serial Monitor module is implemented in the test bench to have the serial transmit and receive functions. The Serial Monitor module sends a serial data to the 8051 core when it is started. This operation will be asynchronous to the 8051 core. Serial Port module of the 8051 core is always watching for a serial data.

When the transmitted byte is captured in the Serial Port module of 8051, a Serial Interrupt with a RI flag is generated to the processor core. At this point, Executer module completes the execution of current instruction and then jumps to interrupt vector table. There it fetches the address of serial port interrupt service routine, then CPU branches to that routine. In this ISR routine, a predefined register is set to a predefined value to indicate that ISR has been run. Finishing the ISR, CPU returns to the software program it has been executing just before the serial interrupt event occurred. This is shown in Figure 5.13.

At the end of the test program, the predefined register is read and compared to the predefined test value. The program branches to a passed location if the serial interrupt service routine has been run or to a failed location if not.

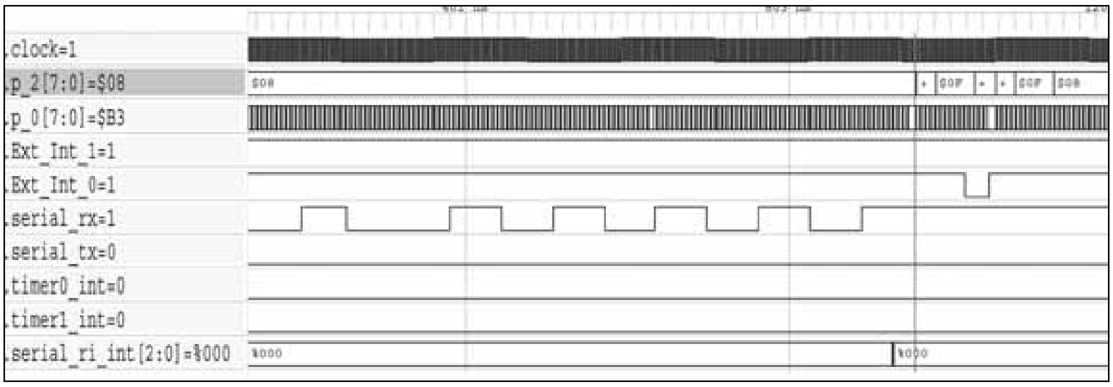


Figure 0.13 Serial Port Receive Operation.

The transmit operation of the serial port is tested by sending some data byte to the remote serial monitor module. At the end of the assembly test program, an information byte about the internal tests, is written to SBUF register. If any data is written to SBUF, the serial port is activated if it has been configured.

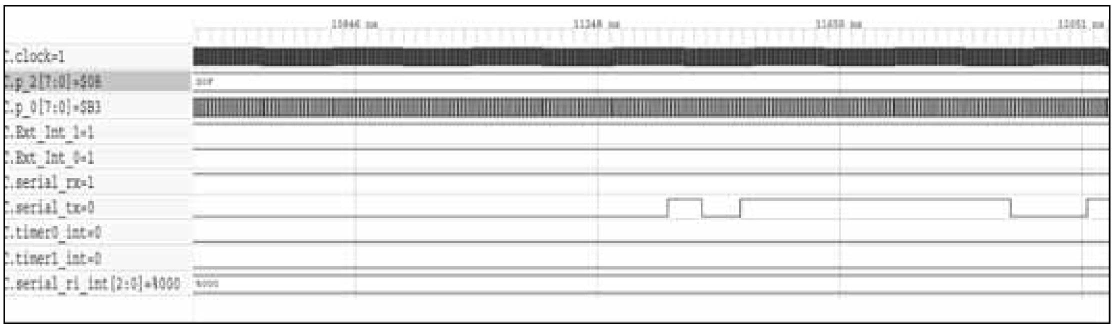


Figure 0.14 Serial Port Transmit Operation.

As seen on the signal diagram in Figure 5.14, SystemC simulation of 8051 core and its test bench, near to the end of the running program, a serial byte is sent to the Serial Monitor module.

5.9 Timer Operations

In order to verify the timer operations, Timer-0 is configured for mode-2, reload mode, with a reload value of 0x10 in the assembly test file. This will generate periodic timer interrupts as shown in Figure 5.15. When a timer interrupt occurs, the program will branch to the interrupt vector, there it will pick the address of timer interrupt service routine, then finally branch to that location. In this function a register is set to a predefined value to indicate that timer interrupt service routine has been executed. After completing the timer ISR, 8051 core returns back to running program. Finally just before the end of the program, the register is tested whether it has been altered by the ISR or not. If the expected value is found a pass mark is given otherwise an error code is given.

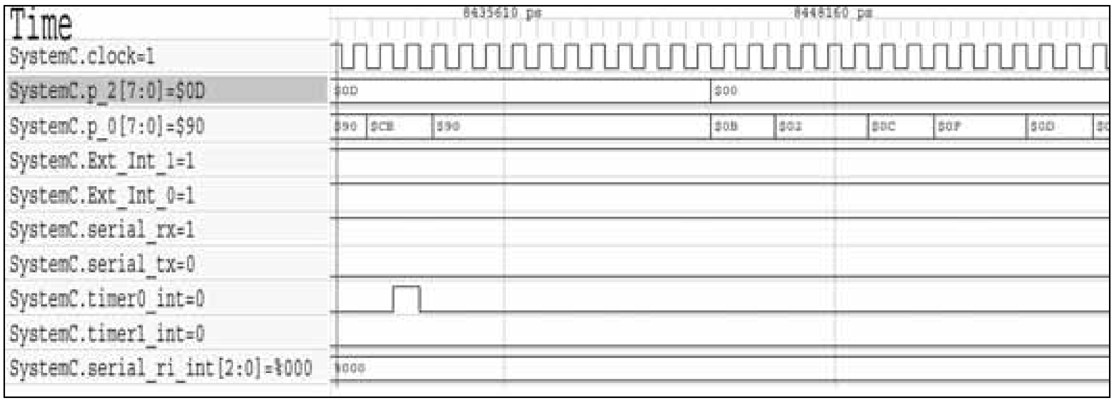
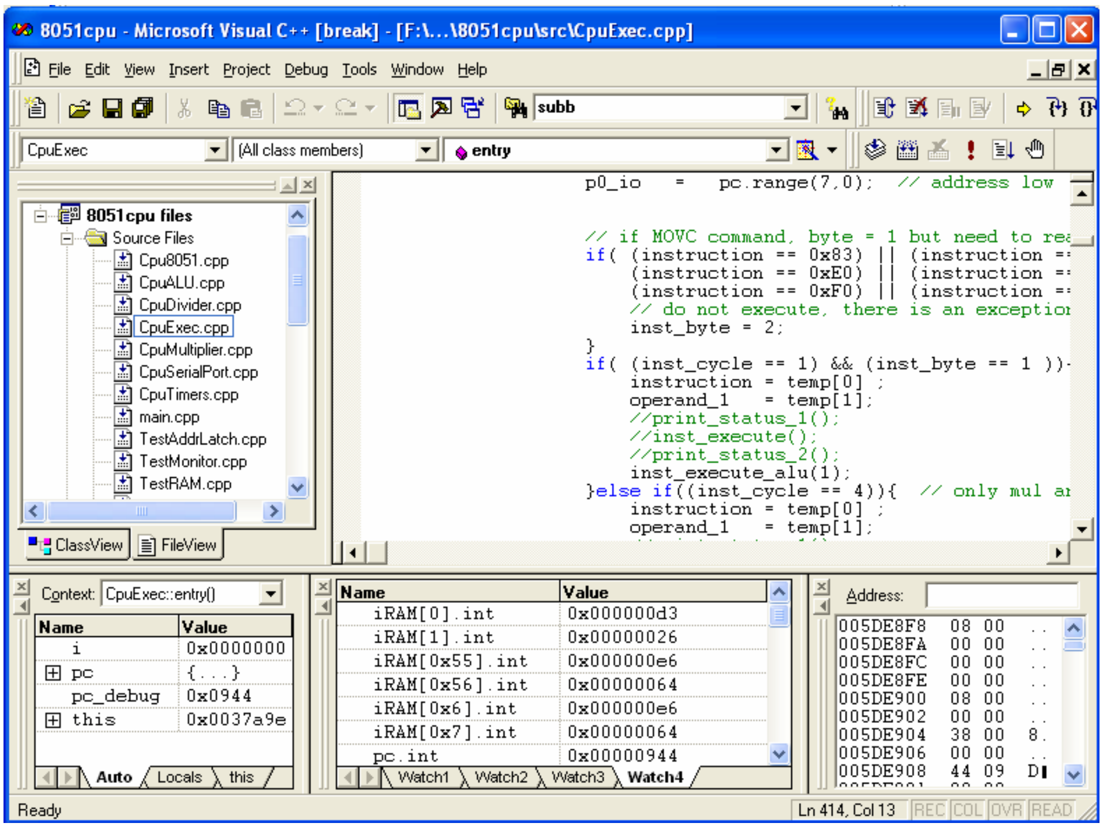


Figure 0.15 Timer-0 Operation.

5.10 Testing The Whole Design

During the verification of 8051 core, a test software is used to span all possible states of the microcontroller. Although this software has already tested how 8051 core executes a software program, a standard code from an independent source would be very beneficial for verifying the design under development.

For this purpose, standard 16-bit Cyclic Redundancy Check (CRC) algorithm is selected because of its very well known structure. The code is downloaded from the internet site of Keil Elektronik GmbH, which is one of the largest suppliers of 8051 microcontroller development environments. 8051 core is capable of doing 8 bit arithmetic operations but there are library functions for handling 16 bit data sizes or more. One of the reasons for selecting the 16 bit CRC algorithm for testing the designed core, is to have a more complicated software using those type of library functions. These codes are supplied in Appendix-E.



```
                p0_io    =    pc.range(7,0);  // address low

                // if MOVC command, byte = 1 but need to rea
                if( (instruction == 0x83) || (instruction ==
                    (instruction == 0xE0) || (instruction ==
                    (instruction == 0xF0) || (instruction ==
                    // do not execute, there is an exceptio
                    inst_byte = 2;
                }
                if( (inst_cycle == 1) && (inst_byte == 1 ))
                    instruction = temp[0] ;
                    operand_1    = temp[1];
                    //print_status_1();
                    //inst_execute();
                    //print_status_2();
                    inst_execute_alu(1);
                }else if((inst_cycle == 4)){  // only mul ar
                    instruction = temp[0] ;
                    operand_1    = temp[1];
```

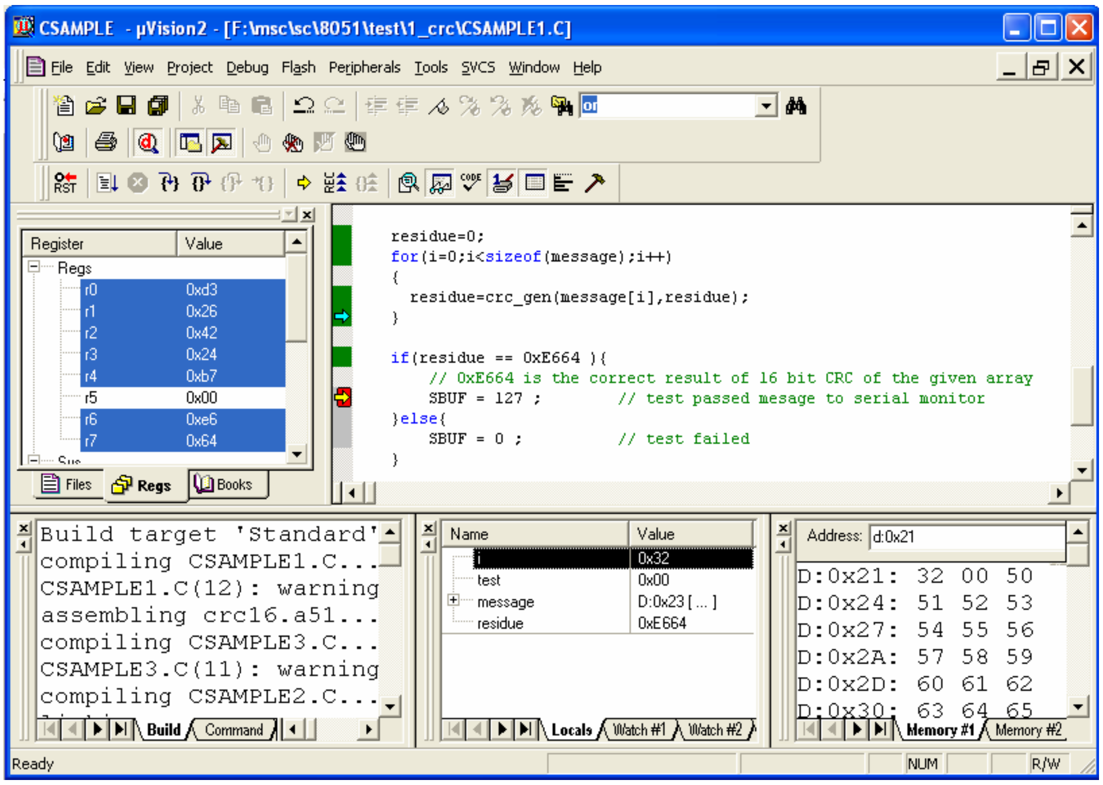
Name	Value
i	0x0000000
⊞ pc	{...}
pc_debug	0x0944
⊞ this	0x0037a9e

Name	Value
iRAM[0].int	0x000000d3
iRAM[1].int	0x00000026
iRAM[0x55].int	0x000000e6
iRAM[0x56].int	0x00000064
iRAM[0x6].int	0x000000e6
iRAM[0x7].int	0x00000064
pc.int	0x00000944

```
005DE8F8  08 00  ..
005DE8FA  00 00  ..
005DE8FC  00 00  ..
005DE8FE  00 00  ..
005DE900  08 00  ..
005DE902  00 00  ..
005DE904  38 00  8.
005DE906  00 00  ..
005DE908  44 09  D▮
```

Figure 0.16 Simulation of 8051 in SystemC.

The software is compiled and linked in Keil µVision2 development and simulation environment. The output hex file is loaded to both Keil's simulator and to our 8051 core designed for this thesis. Then the two simulators are executed with the same input streams, and happily they generated exactly the same output. Additionally, the intermediate values of the variables and states of the 8051 CPU registers are also checked with the Keil's simulator and found hundred percent compatible. This is shown in Figure 5.16 and Figure 5.17.


CSAMPLE - µVision2 - [F:\msc\sc\8051\test\1_crc\CSAMPLE1.C]
File Edit View Project Debug Flash Peripherals Tools SVCS Window Help
Files
Regs
Books
Build
Command
Locals
Watch #1
Watch #2
Memory #1
Memory #2
Ready
NUM
R/W

Register	Value
Regs	
r0	0xd3
r1	0x26
r2	0x42
r3	0x24
r4	0xb7
r5	0x00
r6	0xe6
r7	0x64

```
residue=0;
for(i=0;i<sizeof(message);i++)
{
  residue=crc_gen(message[i],residue);
}

if(residue == 0xE664 ){
    // 0xE664 is the correct result of 16 bit CRC of the given array
    SBUF = 127 ;        // test passed mesage to serial monitor
}else{
    SBUF = 0 ;          // test failed
}
```

```
Build target 'Standard'
compiling CSAMPLE1.C...
CSAMPLE1.C(12): warning
assembling crc16.a51...
compiling CSAMPLE3.C...
CSAMPLE3.C(11): warning
compiling CSAMPLE2.C...
```

Name	Value
i	0x32
test	0x00
message	D:0x23 [...]
residue	0xE664

Address: d:0x21

```
D:0x21:  32  00  50
D:0x24:  51  52  53
D:0x27:  54  55  56
D:0x2A:  57  58  59
D:0x2D:  60  61  62
D:0x30:  63  64  65
```

Figure 0.17 Simulation of 8051 in Keil uVision.

CONCLUSIONS

Today’s high performance devices and systems force the designers to use multi disciplinary development methodologies. SystemC provides a compound design environment for both hardware and software modules of a system. In other words, SystemC is used for co-design and co-simulation processes of the development cycle for highly complicated systems.

In this thesis, the experience of modeling an 8 bit microcontroller using SystemC is discussed. Before going deep in the design process, the basics of SystemC design environment is given and the original 8051 core is explained. Then the design of the 8051 core is described in details and the verification and test issues are discussed.

The top level design of the 8051 core includes independent modules such as the executer unit, arithmetic logic unit (ALU), timer modules and serial communication unit. This modular design approach provides flexibility both to the designer and to the future users of the core. Although all the modules are developed by the author of the thesis in this work, SystemC provides a modular design environment so that the designers can merge their individual works easily. Moreover, each of these modules can be improved or completely changed to keep up with the developing requirements without affecting other modules.

Verification of a certain designed system is an important problem and must be handled carefully as a part of the development process. SystemC provides a perfectly integrated and easy to use verification method. Similar to the development of the system under focus, a suitable and on purpose test bench can be easily created using SystemC. The value of the verification process is getting higher as early it became available to the designer, in this sense; SystemC adds a significant worth to the whole design process.

In our work in this thesis, a test bench is developed using SystemC to verify the operation of the designed 8051 core. During the development cycle, simulation results such as signal traces and viewing internal register contents helped to improve the design. These are compared with the original signal traces supplied by the vendor companies. The results are also compared with the results of the simulations run on industry proven simulators. After having completely compatible results, a standard 16 bit CRC code, which is downloaded from one of the vendors companies, is loaded and ran on our 8051 core giving successful results.

Although SystemC is proven to be a good language for system level modeling which may include both hardware and software tasks, it should not be thought as a replacement for the current HDLs. Instead, it is a development environment on a higher level of abstraction which may use microprocessors, DSPs and complicated bus and bridge systems to achieve complex systems consist of hardware and software modules as well.

The designer can make executable specifications with low simulation times, and construct very well integrated test benches for the verification of the system. As the language is based on C++, the learning curve is relatively short. The support for SystemC in the EDA industry is widening, which means more SystemC familiar development and simulation tools will be on the market in the near future. Also the support for SystemC synthesis is growing with giving the signs of more featured SystemC Compiler tools in coming years.

The 8051 core and external memory elements designed in SystemC can be used as a general platform for designing various ‘System-on-Chip’s (SoC), which may include software or “firmware” as it is so called in embedded systems world, to achieve complicated jobs, such as bus controllers and communication controllers.

REFERENCES

[1] J. Tirado, M, Serra, A. Portero, Q. Saiz, Rapid Prototyping Platform for Reactive Systems with a POLIS based HW-SW Co-design Approach, IP Based Design Seminars, 2000.

[2] Overview of the Open SystemC Initiative, SystemC Organization, 1999

[3] Guido Arnout, C for System Level Design, Design, Automation and Test in Europe (DATE), March 09, 1999.

[4] Synopsis CoCentric SystemC Compiler, Synopsis Inc, 2002.

[5] John Connell, ARM, Bruce Johnson, Synopsis, Early Hardware/ Software Integration Using SystemC 2.0, 2002.

[6] SystemC User's Guide for Version 2.0, SystemC Organization,2002.

[7] Functional Specification For SystemC Version 2.0, SystemC Organization,2002.

[8] P NC-SystemC Simulator Data Sheet, Cadence Design Systems Inc 4538C 10/03, 2003.

[9] Mustafa Badaroğlu, Design of a 8-Bit CMOS Embedded Microcontroller Chip, A Thesis Submitted To The Graduate School of Natural and Applied Sciences of Middle East Technical University in The Department of Electrical and Electronics Engineering,1998.

[10] Describing Synthesizable RTL in SystemC™ Version 1.2, Synopsis Inc, November 2002.

[11] 80C51 Based 8 bit Microcontrollers, Philips Data Handbook, 1998

[12] M.Morris Mano, Digital Design, Second Edition, Prentice Hall International Editions, 1991.

[13] I. Scott MacKenzie, The 8051 Microcontroller, Second Edition, Prentice Hall, 1995.

[14] 80C51 Family Hardware Description, SU00557, SU00558, SU00559, Philips 8051 Datasheet, 1998.

[15] Seamless C-Bridge Technology Enabling C in Hardware/Software Co-Verification Datasheet, Mentor Graphics, 2-02 HDG 1020010, 2002.

[16] Prosilog's SystemC Compiler Datasheet, Prosilog Inc, 2002.

[17] 8051 IP Core 8 Bit Micro-Controller Datasheet, Aldec Design Verification Company, 2004.

[18] DW8551_DS Datasheet, The Synthesizable VHDL 8051 Core, Synopsis Inc, 2002.

[19] Smart Card IC features 256 kbyte of pure flash memory, Thomas Net, March 2004.

[20] MIFARE PROX 8-bit Dual Interface IC Datasheet, Philips Inc., 2004.

[21] A Varma, J.R. Armstrong, J.M. Baker, A SystemC GSM Model For Hardware/Software Co-Design International HDL Conference and Exhibition (HDLCon), 2002.

[22] K. Bartleson, A New Standard for System-Level Design. SystemC Organization, Design, Automation and Test in Europe (DATE), 2003.

[23] Allan Cochrane, John Connell, Andy Nightingale, Capturing Design Intent and Evaluating Performance with SystemC, ARM, 2003.

[24] Ramaswamy Ramaswamy, Russel Tessier, Department of Electrical and Computer Engineering, University of Massachuattes, The Integration of SystemC and Hardware Assisted Verification, Proceedings of the Reconfigurable Computing, 2002.

[25] Jan Lundgren, Bengt Oelmann, Behavioral Simulation of Power Line Noise Coupling in Mixed Signal Systems using SystemC, Proceedings. IEEE Computer Society, 2003.

[26] George Economakos, Petros Oikonomakos, Behavioral Synthesis with SystemC, National Technical University of Athens, Department of Electrical and Computer Engineering, Proceedings in , Design, Automation and Test in Europe (DATE), 2001

[27] Tim Kogel Andreas Wieferink, Heinrich Meyr, Andrea Kroll, SystemC Based Architecture Exploration of a 3D Graphic Processor, Workshop on Signal Processing Systems, IEEE, 2001.

[28] Luca Benini, Davide Bertozzi, Davide Bruni, University Di Bologna, SystemC Co-simulation and Emulation of Multiprocessor SoC Designs, IEEE Computer Society, April 2003

[29] Efficient Automatic Visualization of SystemC Designs, Institute of Computer Science, University of Bremen, Germany, 2003.

[30] Rolf Drechsler, Daniel Grobe, Connecting the Value Chain with SystemC, Rick Jamison, Geoffrey Moore, Synopsis Inc, September 1999.

[31] Sudep Pasricha, Transaction Level Modeling for SoC with SystemC 2.0, Design Flow and Reuse/ R&D ST Microelectronics Ltd, Synopsys User Group Conference, 2002.

[32] Joachim Gerlach, University of Tubingen, Germany, System Level design using the SystemC Modeling Platform, Worshop on System Design Automation SDA, 2000.

[33] C Norris Ip, Stuart Swan, Cadance Design Systems, A tutorial Introduction on the New SystemC Verification Standard, Design, Automation and Test in Europe (DATE) 2003.

[34] Ando Ki, Empirical Study of SystemC, R&D Center Dynalith Systems, April 2003.

APPENDICES

A. Instruction Set Of 8051

OpCode	**Operation**		**Bytes**	**Clocks**
00 H	NOP	NOP	1	1
01 H	AJMP addr11	AJMP addr11	2	3
02 H	LJMP addr16	LJMP addr16	3	4
03 H	RR A	RR A	1	1
04 H	INC A	INC A	1	1
05 H	INC direct	INC direct	2	3
06 H	INC @R0	INC @Ri	1	3
07 H	INC @R1		1	3
08 H	INC R0	INC Rn	1	2
09 H	INC R1		1	2
0A H	INC R2		1	2
0B H	INC R3		1	2
0C H	INC R4		1	2
0D H	INC R5		1	2
0E H	INC R6		1	2
0F H	INC R7		1	2
10 H	JBC bit,rel	JBC bit,rel	3	4
11 H	ACALL addr11	ACALL addr11	2	6
12 H	LCALL addr16	LCALL addr16	3	6
13 H	RRC A	RRC A	1	1
14 H	DEC A	DEC A	1	1
15 H	DEC direct	DEC direct	1	2
16 H	DEC @R0	DEC @Ri	2	3
17 H	DEC @R1		2	3
18 H	DEC R0	DEC Rn	1	1
19 H	DEC R1		1	1
1A H	DEC R2		1	1
1B H	DEC R3		1	1
1C H	DEC R4		1	1
1D H	DEC R5		1	1

1E H	DEC R6		1	1
1F H	DEC R7		1	1
20 H	JB bit,rel	JB bit,rel	3	4
21 H	AJMP addr11		2	3
22 H	RET	RET Return	1	4
23 H	RL A	RL A	1	1
24 H	ADD A,#data	ADD A,#data	2	2
25 H	ADD A,direct	ADD A,direct	2	2
26 H	ADD A,@R0	ADD A,@Ri	1	2
27 H	ADD A,@R1		1	2
28 H	ADD A,R0	ADD A,Rn	1	1
29 H	ADD A,R1		1	1
2A H	ADD A,R2		1	1
2B H	ADD A,R3		1	1
2C H	ADD A,R4		1	1
2D H	ADD A,R5		1	1
2E H	ADD A,R6		1	1
2F H	ADD A,R7		1	1
30 H	JNB bit,rel	JNB bit,rel	3	4
31 H	ACALL addr11		2	6
32 H	RETI	RETI Return	1	4
33 H	RLC A	RLC A	1	1
34 H	ADDC A,#data	ADDC A,#data	2	2
35 H	ADDC A,direct	ADDC A,direct	2	2
36 H	ADDC A,@R0	ADDC A,@Ri	1	2
37 H	ADDC A,@R1		1	2
38 H	ADDC A,R0	ADDC A,Rn	1	1
39 H	ADDC A,R1		1	1
3A H	ADDC A,R2		1	1
3B H	ADDC A,R3		1	1
3C H	ADDC A,R4		1	1
3D H	ADDC A,R5		1	1
3E H	ADDC A,R6		1	1
3F H	ADDC A,R7		1	1
40 H	JC rel	JC rel	2	3
41 H	AJMP addr11		2	3
42 H	ORL direct,A	ORL direct,A	2	3
43 H	ORL direct,#data	ORL direct,#data	3	4

44 H	ORL A,#data	ORL A,#data	2	2
45 H	ORL A,direct	ORL A,direct	2	2
46 H	ORL A,@R0	ORL A,@Ri	1	2
47 H	ORL A,@R1		1	2
48 H	ORL A,R0	ORL A,Rn	1	1
49 H	ORL A,R1		1	1
4A H	ORL A,R2		1	1
4B H	ORL A,R3		1	1
4C H	ORL A,R4		1	1
4D H	ORL A,R5		1	1
4E H	ORL A,R6		1	1
4F H	ORL A,R7		1	1
50 H	JNC rel	JNC rel	2	3
51 H	ACALL addr11		2	6
52 H	ANL direct,A	ANL direct,A	2	3
53 H	ANL direct,#data	ANL direct,#data	3	4
54 H	ANL A,#data	ANL A,#data	2	2
55 H	ANL A,direct	ANL A,direct	2	2
56 H	ANL A,@R0	ANL A,@Ri	1	2
57 H	ANL A,@R1		1	2
58 H	ANL A,R0	ANL A,Rn	1	1
59 H	ANL A,R1		1	1
5A H	ANL A,R2		1	1
5B H	ANL A,R3		1	1
5C H	ANL A,R4		1	1
5D H	ANL A,R5		1	1
5E H	ANL A,R6		1	1
5F H	ANL A,R7		1	1
60 H	JZ rel	JZ rel	2	3
61 H	AJMP addr11		2	3
62 H	XRL direct,A	XRL direct,A	2	3
63 H	XRL direct,#data	XRL direct,#data	3	4
64 H	XRL A,#data	XRL A,#data	2	2
65 H	XRL A,direct	XRL A,direct	2	2
66 H	XRL A,@R0	XRL A,@Ri	1	2
67 H	XRL A,@R1		1	2
68 H	XRL A,R0	XRL A,Rn	1	1
69 H	XRL A,R1		1	1
6A H	XRL A,R2		1	1

6B H	XRL A,R3		1	1
6C H	XRL A,R4		1	1
6D H	XRL A,R5		1	1
6E H	XRL A,R6		1	1
6F H	XRL A,R7		1	1
70 H	JNZ rel	JNZ rel	2	3
71 H	ACALL addr11		2	6
72 H	ORL C,direct	ORL C,direct	2	2
73 H	JMP @A+DPTR	JMP @A + DPTR	1	2
74 H	MOV A,#data	MOV A,#data	2	2
75 H	MOV direct,#data	MOV direct,#data	3	3
76 H	MOV @R0,#data	MOV @Ri,#data	2	3
77 H	MOV @R1,#data	MOV @Ri,#data	2	3
78 H	MOV R0,#data	MOV Rn,#data	2	2
79 H	MOV R1,#data	MOV Rn,#data	2	2
7A H	MOV R2,#data	MOV Rn,#data	2	2
7B H	MOV R3,#data	MOV Rn,#data	2	2
7C H	MOV R4,#data	MOV Rn,#data	2	2
7D H	MOV R5,#data	MOV Rn,#data	2	2
7E H	MOV R6,#data	MOV Rn,#data	2	2
7F H	MOV R7,#data	MOV Rn,#data	2	2
80 H	SJMP rel	SJMP rel	2	3
81 H	AJMP addr11		2	3
82 H	ANL C,bit	ANL C,bit	2	2
83 H	MOVC A,@A+PC	MOVC A,@A + PC	1	3
84 H	DIV AB	DIV A,B	1	5
85 H	MOV direct,direct	MOV direct,direct	3	4
86 H	MOV direct,@R0	MOV direct,@Ri	2	4
87 H	MOV direct,@R1		2	4
88 H	MOV direct,R0	MOV direct,Rn	2	3
89 H	MOV direct,R1		2	3
8A H	MOV direct,R2		2	3
8B H	MOV direct,R3		2	3
8C H	MOV direct,R4		2	3
8D H	MOV direct,R5		2	3
8E H	MOV direct,R6		2	3
8F H	MOV direct,R7		2	3
90 H	MOV DPTR,#data16	MOV DPTR,#data16	3	3

91 H	ACALL addr11		2	6
92 H	MOV bit,C	MOV bit,C	2	3
93 H	MOVC A,@A+DPTR	MOVC A,@A + DPTR	1	3
94 H	SUBB A,#data	SUBB A,#data	2	2
95 H	SUBB A,direct	SUBB A,direct	2	2
96 H	SUBB A,@R0	SUBB A,@Ri	1	2
97 H	SUBB A,@R1		1	2
98 H	SUBB A,R0	SUBB A,Rn	1	1
99 H	SUBB A,R1		1	1
9A H	SUBB A,R2		1	1
9B H	SUBB A,R3		1	1
9C H	SUBB A,R4		1	1
9D H	SUBB A,R5		1	1
9E H	SUBB A,R6		1	1
9F H	SUBB A,R7		1	1
A0 H	ORL C,bit	ORL C,bit	2	2
A1 H	AJMP addr11		2	3
A2 H	MOV C,bit	MOV C,bit	2	2
A3 H	INC DPTR	INC DPTR	1	3
A4 H	MUL AB	MUL A,B	1	5
A5 H1	–	A5 H1	–	-
A6 H	MOV @R0,direct	MOV @Ri,direct	2	5
A7 H	MOV @R1,direct	MOV @Ri,direct	2	5
A8 H	MOV R0,direct	MOV Rn,direct	2	4
A9 H	MOV R1,direct	MOV Rn,direct	2	4
AA H	MOV R2,direct	MOV Rn,direct	2	4
AB H	MOV R3,direct	MOV Rn,direct	2	4
AC H	MOV R4,direct	MOV Rn,direct	2	4
AD H	MOV R5,direct	MOV Rn,direct	2	4
AE H	MOV R6,direct	MOV Rn,direct	2	4
AF H	MOV R7,direct	MOV Rn,direct	2	4
B0 H	ANL C,bit	ANL C,bit	2	2
B1 H	ACALL addr11		2	6
B2 H	CPL bit	CPL bit	2	3
B3 H	CPL C	CPL C	1	1
B4 H	CJNE A,#data,rel	CJNE A,#data,rel	3	4
B5 H	CJNE A,direct,rel	CJNE A,direct,rel	3	4
B6 H	CJNE @R0,#data,rel	CJNE @Ri,#data,rel	3	4
B7 H	CJNE @R1,#data,rel		3	4

B8 H	CJNE R0,#data,rel	CJNE Rn,#data rel	3	4
B9 H	CJNE R1,#data,rel		3	4
BA H	CJNE R2,#data,rel		3	4
BB H	CJNE R3,#data,rel		3	4
BC H	CJNE R4,#data,rel		3	4
BD H	CJNE R5,#data,rel		3	4
BE H	CJNE R6,#data,rel		3	4
BF H	CJNE R7,#data,rel		3	4
C0 H	PUSH direct	PUSH direct	2	4
C1 H	AJMP addr11		2	3
C2 H	CLR bit	CLR bit	2	3
C3 H	CLR C	CLR C	1	1
C4 H	SWAP A	SWAP A	1	1
C5 H	XCH A,direct	XCH A,direct	2	3
C6 H	XCH A,@R0	XCH A,@Ri	1	3
C7 H	XCH A,@R1		1	3
C8 H	XCH A,R0	XCH A,Rn	1	2
C9 H	XCH A,R1		1	2
CA H	XCH A,R2		1	2
CB H	XCH A,R3		1	2
CC H	XCH A,R4		1	2
CD H	XCH A,R5		1	2
CE H	XCH A,R6		1	2
CF H	XCH A,R7		1	2
D0 H	POP direct	POP direct	2	3
D1 H	ACALL addr11		2	6
D2 H	SETB bit	SETB bit	2	3
D3 H	SETB C	SETB C	1	1
D4 H	DA A	DA A	1	1
D5 H	DJNZ direct,rel	DJNZ direct,rel	3	4
D6 H	XCHD A,@R0	XCHD A,@Ri	1	3
D7 H	XCHD A,@R1		1	3
D8 H	DJNZ R0,rel	DJNZ Rn,rel	2	3
D9 H	DJNZ R1,rel		2	3
DA H	DJNZ R2,rel		2	3
DB H	DJNZ R3,rel		2	3
DC H	DJNZ R4,rel		2	3
DD H	DJNZ R5,rel		2	3
DE H	DJNZ R6,rel		2	3

DF H	DJNZ R7,rel		2	3
E0 H	MOVX A,@DPTR	MOVX A,@DPTR	1	4
E1 H	AJMP addr11		2	3
E2 H	MOVX A,@R0	MOVX A,@Ri	1	4
E3 H	MOVX A,@R1		1	4
E4 H	CLR A	CLR A	1	1
E5 H	MOV A,direct	MOV A,direct	2	2
E6 H	MOV A,@R0	MOV A,@Ri	1	2
E7 H	MOV A,@R1		1	2
E8 H	MOV A,R0	MOV A,Rn	1	1
E9 H	MOV A,R1		1	1
EA H	MOV A,R2		1	1
EB H	MOV A,R3		1	1
EC H	MOV A,R4		1	1
ED H	MOV A,R5		1	1
EE H	MOV A,R6		1	1
EF H	MOV A,R7		1	1
F0 H	MOVX @DPTR,A	MOVX @DPTR,A	1	4
F1 H	ACALL addr11		2	6
F2 H	MOVX @R0,A	MOVX @Ri,A	1	4
F3 H	MOVX @R1,A		1	4
F4 H	CPL A	CPL A	1	1
F5 H	MOV direct,A	MOV direct,A	2	3
F6 H	MOV @R0,A	MOV @Ri,A	1	3
F7 H	MOV @R1,A	MOV @Ri,A	1	3
F8 H	MOV R0,A	MOV Rn,A	1	2
F9 H	MOV R1,A	MOV Rn,A	1	2
FA H	MOV R2,A	MOV Rn,A	1	2
FB H	MOV R3,A	MOV Rn,A	1	2
FC H	MOV R4,A	MOV Rn,A	1	2
FD H	MOV R5,A	MOV Rn,A	1	2
FE H	MOV R6,A	MOV Rn,A	1	2
FF H	MOV R7,A	MOV Rn,A	1	2

B. Nonsynthesizable SystemC And C++ Constructs

Category	**Comment**	**Construct**	**Corrective action**
Thread process	Used for modeling and test benches	SC_THREAD SC_CTHREAD	SC_METHOD
Main function	Used for simulation.	sc_main()	
Clock generators	Used for simulation	sc_start()	
Communication	Used for modeling communication	sc_interface sc_port sc_mutex sc_fifo“	
Watching	Not supported for RTL synthesis	watching() W_BEGIN, W_END, W_DO, W_ESCAPE	
Synchronization	Used for synchronization of events	Master-slave library of SystemC	
Tracing	Creates waveforms of signals, channels, and variables for simulation.	sc_trace, sc_create trace_file	
Local class declaration	Not allowed.		Replace global class
Nested class declaration	Not allowed.		Replace global class
Derived class	Not allowed.		Replace global class
Dynamic storage allocation	Not allowed. The new construct is allowed only to instantiate a module to create hierarchy.	malloc(), free(), new, new[], delete, delete[]	Use static memory allocation
Exception handling	Not allowed		
Recursive function call	Not allowed		
Function overloading	Not allowed		Unique function calls
C++ built-in functions	Math library, I/O library, file I/O, and similar built-in C++ functions not allowed.		Replace with synthesizable functions
Virtual function	Not allowed		Replace with a nonvirtual function.
Inheritance	Not allowed		Create independent modules.

Multiple inheritance	Not allowed		Create independent modules.
Member access control specifiers	Allowed in code but ignored for synthesis. All member access is public.	public, protected, private, friend	
the (->) operator	Not allowed, except for module instantiation.		Replace with access using the period (.) operator.
Static member	Not allowed.		Replace with non-static member
Dereferenceoperator	Not allowed.	* and & operators	Use array accessing
Unbounded loop	Not allowed.		Replace with a bounded loop,
Out-of-bound array access	Not allowed.		Replace with in-bound array access.
Operator overloading	Not allowed (except the classes overloaded by SystemC).		Replace with unique function calls.
Operator, sizeof	Not allowed.		Determine size statically for use in synthesis.
Pointer	Pointers are allowed only in hierarchical modules to instantiate other modules.	*	Replace all other pointers with accessto array elements orindividual elements.
Pointer type conversions	Not allowed.		
this pointer	Not allowed.		
Reference	Allowed only for passing parameters to functions.	&	Replace in all other cases.
Static variable	Not allowed in functions.		
User-defined template class	Only SystemC templates classes such as sc_int<> are supported.		
Type casting at runtime	Not allowed		
Type identification at runtime	Not allowed		
Unconditional branching	Not allowed	goto	
Unions	Not allowed		Use structs
Global Variables	Not allowed		Use local variables
Member variable	access to member variables by only one process is supported.		Use signals instead of variables for communication between processes.
Volatile variable	Not allowed		Use only non-volatile variables

C. Assembly Verification Code

ADD Instructions

To verify ADD instructions, following code is used;

```
;///////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 1 // ADD A,Rn (1)
        MOV  A,#10
        MOV  R0,#10
        ADD  A,R0
        SUBB A,#20
        JZ   DONE_1
        MOV  P1,#1
        LJMP FAILED
DONE_1:
;///////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 2 // ADD A,direct (2)
        MOV  A,#10
        MOV  100,#10
        ADD  A,100
        SUBB A,#20
        JZ   DONE_2
        MOV  P1,#2
        LJMP FAILED
DONE_2:
;///////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 3 // ADD A,@Ri (3)
        MOV  A,#10
        MOV  R0,#100
        MOV  100,#10
        ADD  A,@R0
        SUBB A,#20
        JZ   DONE_3
        MOV  P1,#3
        LJMP FAILED
DONE_3:
;///////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 5 // ADD A,#data (5)
        MOV  A,#10
        ADD  A,#5
        SUBB A,#15
        JZ   DONE_5
        MOV  P1,#5
        LJMP FAILED
DONE_5:
;///////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 6 // ADDC A,Rn (6)
        MOV  A,#10
        MOV  R0,#10
        CPL  C
        ADDC A,R0
        SUBB A,#21
        JZ   DONE_6
        MOV  P1,#6
        LJMP FAILED
DONE_6:
;///////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 7 // ADDC A,direct (7)
        MOV  A,#10
        MOV  100,#10
        CPL  C
        ADDC A,100
        SUBB A,#21
        JZ   DONE_7
        MOV  P1,#7
        LJMP FAILED
DONE_7:
;//////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 8 // ADDC A,@Ri (8)
        MOV  A,#10
        MOV  R0,#100
        MOV  100,#10
        CPL  C
        ADDC A,@R0
        SUBB A,#21
        JZ   DONE_8
        MOV  P1,#8
        LJMP FAILED
DONE_8:
;//////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 9 // ADDC A,#data (9)
        MOV  A,#10
        CPL  C
        ADDC A,#5
        SUBB A,#16
        JZ   DONE_9
        MOV  P1,#9
        LJMP FAILED
DONE_9:
```

Subtract Instructions

Subtract instructions are tested with the code below;

```
;//////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 97 // SUBB A,Rn (97)
        MOV  A,#10
        MOV  R0,#10
        SUBB A,R0
        JZ   DONE_97
        MOV  P1,#97
        LJMP FAILED
DONE_97:
;//////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 98 // SUBB A,direct (98)
        MOV  A,#10
        MOV  127,#10
        SUBB A,127
        JZ   DONE_98
        MOV  P1,#98
        LJMP FAILED
DONE_98:
;//////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 99 // SUBB A,@Ri (99)
        MOV  A,#10
        MOV  R0,#127
        MOV  127,#10
        SUBB A,@R0
        JZ   DONE_99
        MOV  P1,#99
        LJMP FAILED
DONE_99:
```

Increment Operations

Increment operations are verified with the code below ;

```
;//////////////////////////////////////////////////
```

```
        MOV  PSW,#0              ;INST 37 // INC A (37)
        MOV  A,#10
        INC  A
        SUBB A,#11
        JZ   DONE_37
        MOV  P1,#37
        LJMP FAILED
DONE_37:
;///////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 38 // INC Rn (38)
        MOV  R0,#10
        INC  R0
        MOV  A,R0
        SUBB A,#11
        JZ   DONE_38
        MOV  P1,#38
        LJMP FAILED
DONE_38:
;///////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 39 // INC direct (39)
        MOV  127,#10
        INC  127
        MOV  A,127
        SUBB A,#11
        JZ   DONE_39
        MOV  P1,#39
        LJMP FAILED
DONE_39:
;///////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 40 // INC @Ri (40)
        MOV  127,#10
        MOV  R0,#127
        INC  @R0
        MOV  A,@R0
        SUBB A,#11
        JZ   DONE_40
        MOV  P1,#40
        LJMP FAILED
DONE_40:
;///////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 41 // INC DPTR (41)
        MOV  DPTR,#12FFH
        INC  DPTR
        MOV  A,DPH
        SUBB A,#13H
        JZ   DPH_OK_41
        MOV  P1,#41
        LJMP FAILED
DPH_OK_41:
        MOV  A,DPL
        JZ   DONE_41
        MOV  P1,#41
        LJMP FAILED
DONE_41:
```

Decrement Operations

Decrement operations are tested with the code below ;

```
;///////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 30 // DEC A (30)
        MOV  A,#10
        DEC  A
        SUBB A,#9
        JZ   DONE_30
        MOV  P1,#30
        LJMP FAILED
DONE_30:
;///////////////////////////////////////////////////
        MOV  PSW,#0
        MOV  R0,#10              ;INST 31 // DEC Rn (31)
        DEC  R0
        MOV  A,R0
        SUBB A,#9
        JZ   DONE_31
        MOV  P1,#31
        LJMP FAILED
DONE_31:
;//////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 32 // DEC direct (32)
        MOV  127,#10
        DEC  127
        MOV  A,127
        SUBB A,#9
        JZ   DONE_32
        MOV  P1,#32
        LJMP FAILED
DONE_32:
;//////////////////////////////////////////////////
        MOV  PSW,#0
        MOV  R0,#127     ;INST 33 // DEC @Ri (33)
        MOV  127,#10
        DEC  @R0
        MOV  A,@R0
        SUBB A,#9
        JZ   DONE_33
        MOV  P1,#33
        LJMP FAILED
DONE_33:
```

Multiplication and Division Operations

Multiplication and division operations are verified as follows;

```
;//////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 76 // MUL AB (76)
        MOV  A,#80
        MOV  B,#160
        MUL  AB                  ; = 3200H
        JNZ  ERROR_76
        MOV  A,B
        SUBB A,#32H
        JZ   DONE_76
ERROR_76:
        MOV  P1,#76
        LJMP FAILED
DONE_76:
;//////////////////////////////////////////////////
        MOV  PSW,#0
        MOV  A,#251              ;INST 34 // DIV AB (34)
        MOV  B,#18
        DIV  AB
        SUBB A,#13
        JZ   CHECK_B_34
        MOV  P1,#34
        LJMP FAILED
CHECK_B_34:
        MOV  A,B
        SUBB A,#17
        JZ   DONE_34
        MOV  P1,#34
        LJMP FAILED
DONE_34:
```

Logical and Shift Operations

Logical and shift operations in 8051 processor are verified with the code below.

```
;//////////////////////////////////////////////////
        MOV  PSW,#0             ;INST 11 // ANL A,Rn (11)
        MOV  R0,#255
        MOV  A,#170
        ANL  A,R0
        SUBB A,#170
        JZ   DONE_11
        MOV  P1,#11
        LJMP FAILED
DONE_11:
;//////////////////////////////////////////////////
        MOV  PSW,#0             ;INST 12 // ANL A,direct (12)
        MOV  127,#0
        MOV  A,#255
        ANL  A,127
        JZ   DONE_12
        MOV  P1,#12
        LJMP FAILED
DONE_12:
;//////////////////////////////////////////////////
        MOV  PSW,#0             ;INST 13 // ANL A,@Ri (13)
        MOV  R0,#127
        MOV  127,#1
        MOV  A,#254
        ANL  A,@R0
        JZ   DONE_13
        MOV  P1,#13
        LJMP FAILED
DONE_13:
;//////////////////////////////////////////////////
        MOV  PSW,#0             ;INST 14 // ANL A,#data (14)
        MOV  A,#255
        ANL  A,#255
        SUBB A,#255
        JZ   DONE_14
        MOV  P1,#14
        LJMP FAILED
DONE_14:
;//////////////////////////////////////////////////
        MOV  PSW,#0             ;INST 15 // ANL direct,A (15)
        MOV  50,#255
        MOV  A,#0
        ANL  50,A
        MOV  A,50
        JZ   DONE_15
        MOV  P1,#15
        LJMP FAILED
DONE_15:
;//////////////////////////////////////////////////
        MOV  PSW,#0             ;INST 16 // ANL direct,#data (16)
        MOV  25,#128
        ANL  25,#255
        MOV  A,25
        SUBB A,#128
        JZ   DONE_16
        MOV  P1,#16
        LJMP FAILED
DONE_16:
;//////////////////////////////////////////////////
        MOV  PSW,#0             ;INST 17 // ANL C,bit (17)
        MOV  A,#128
        CPL  C
        ANL  C,ACC.7
        SUBB A,#127
        JZ   DONE_17
        MOV  P1,#17
        LJMP FAILED
DONE_17:
;//////////////////////////////////////////////////
        MOV  PSW,#0             ;INST 18 // ANL C,/bit (18)
        MOV  A,#128
        CPL  C
        ANL  C,/ACC.7
        SUBB A,#128
        JZ   DONE_18
        MOV  P1,#18
        LJMP FAILED
DONE_18:
```

```
;//////////////////////////////////////////////////
        MOV  PSW,#0               ;INST 78 // ORL A,Rn (78)
        MOV  A,#90H
        MOV  R0,#9H
        ORL  A,R0
        SUBB A,#99H
        JZ   DONE_78
        MOV  P1,#78
        LJMP FAILED
DONE_78:
;//////////////////////////////////////////////////
        MOV  PSW,#0               ;INST 79 // ORL A,direct (79)
        MOV  A,#9H
        MOV  127,#90H
        ORL  A,127
        SUBB A,#99H
        JZ   DONE_79
        MOV  P1,#79
        LJMP FAILED
DONE_79:
;//////////////////////////////////////////////////
        MOV  PSW,#0               ;INST 80 // ORL A,@Ri (80)
        MOV  A,#90H
        MOV  R0,#127
        MOV  127,#06H
        ORL  A,@R0
        SUBB A,#96H
        JZ   DONE_80
        MOV  P1,#80
        LJMP FAILED
DONE_80:
;//////////////////////////////////////////////////
        MOV  PSW,#0               ;INST 81 // ORL A,#data (81)
        MOV  A,#11H
        ORL  A,#22H
        SUBB A,#33H
        JZ   DONE_81
        MOV  P1,#81
        LJMP FAILED
DONE_81:
;//////////////////////////////////////////////////
        MOV  PSW,#0               ;INST 82 // ORL direct,A (82)
        MOV  A,#90H
        MOV  127,#9H
        ORL  127,A
        CLR  A
        MOV  A,127
        SUBB A,#99H
        JZ   DONE_82
        MOV  P1,#82
        LJMP FAILED
DONE_82:
;//////////////////////////////////////////////////
        MOV  PSW,#0               ;INST 83 // ORL direct,#data (83)
        MOV  127,#90H
        ORL  127,#9H
        MOV  A,127
        SUBB A,#99H
        JZ   DONE_83
        MOV  P1,#83
        LJMP FAILED
DONE_83:
;//////////////////////////////////////////////////
        MOV  PSW,#0               ;INST 84 // ORL C,bit (84)
        ORL  C,ACC.0
        JC   ERROR_84
        MOV  A,#1
        ORL  C,ACC.0
        JNC  ERROR_84
        ORL  C,ACC.1
        JC   DONE_84
ERROR_84:
        MOV  P1,#84
        LJMP FAILED
DONE_84:
;//////////////////////////////////////////////////
        MOV  PSW,#0               ;INST 85 // ORL C,/bit (85)
        MOV  A,#1
        ORL  C,/ACC.0
        JC   ERROR_85
        ORL  C,/ACC.1
        JNC  ERROR_85
        ORL  C,/ACC.0
        JC   DONE_85
ERROR_85:
        MOV  P1,#85
        LJMP FAILED
DONE_85:

;///////////////////////////////////////////////////
        MOV  PSW,#0               ;INST 106 // XRL A,Rn (106)
        MOV  A,#35H
        MOV  R0,#53H
        XRL  A,R0
        SUBB A,#66H
        JZ   DONE_106
        MOV  P1,#106
        LJMP FAILED
DONE_106:
;///////////////////////////////////////////////////
        MOV  PSW,#0               ;INST 107 // XRL A,direct (107)
        MOV  A,#53H
        MOV  127,#35H
        XRL  A,127
        SUBB A,#66H
        JZ   DONE_107
        MOV  P1,#107
        LJMP FAILED
DONE_107:
;///////////////////////////////////////////////////
        MOV  PSW,#0               ;INST 108 // XRL A,@Ri (108)
        MOV  A,#35H
        MOV  R0,#127
        MOV  127,#53H
        XRL  A,@R0
        SUBB A,#66H
        JZ   DONE_108
        MOV  P1,#108
        LJMP FAILED
DONE_108:
;///////////////////////////////////////////////////
        MOV  PSW,#0               ;INST 109 // XRL A,#data (109)
        MOV  A,#35H
        XRL  A,#53H
        SUBB A,#66H
        JZ   DONE_109
        MOV  P1,#109
        LJMP FAILED
DONE_109:
;///////////////////////////////////////////////////
        MOV  PSW,#0               ;INST 110 // XRL direct,A (110)
        MOV  A,#35H
        MOV  127,#53H
        XRL  127,A
        CLR  A
        MOV  A,127
        SUBB A,#66H
        JZ   DONE_110
        MOV  P1,#110
        LJMP FAILED
DONE_110:
;///////////////////////////////////////////////////
        MOV  PSW,#0               ;INST 111 // XRL direct,#data (111)
        MOV  127,#35H
        XRL  127,#53H
        MOV  A,127
        SUBB A,#66H
        JZ   DONE_111
        MOV  P1,#111
        LJMP FAILED
DONE_111:

;///////////////////////////////////////////////////
        MOV  A,#128               ;INST 23 // CLR A (23)
        CLR  A
        JZ   DONE_23
        MOV  P1,#23
        LJMP FAILED
```

```
DONE_23:
;//////////////////////////////////////////////////
        CLR  C                  ;INST 25 // CLR C (24)
        CPL  C                  ;INST 24 // CPL C (24)
        JC   DONE_24
        MOV  P1,#24
        LJMP FAILED
DONE_24:
;//////////////////////////////////////////////////
        CLR  ACC.6              ;INST 26 // CLR bit (26)
        CPL      ACC.6          ;INST 27 // CPL bit (27)
        JNZ  DONE_26
        MOV  P1,#26
        LJMP FAILED
DONE_26:
;//////////////////////////////////////////////////
        MOV  PSW,#0             ;INST 28 // CPL A (28)
        MOV  A,#255
        CPL  A
        JZ   DONE_28
        MOV  P1,#28
        LJMP FAILED
DONE_28:

;//////////////////////////////////////////////////
        MOV  PSW,#0             ;INST 90 // RL A (90)
        MOV  A,#129
        RL   A
        SUBB A,#3
        JZ   DONE_90
        MOV  P1,#90
        LJMP FAILED
DONE_90:
;//////////////////////////////////////////////////
        MOV  PSW,#0             ;INST 91 // RLC A (91)
        MOV  A,#129
        RLC  A
        SUBB A,#1       ;A(2)-C(1)-1
        JZ   DONE_91
        MOV  P1,#91
        LJMP FAILED
DONE_91:
;//////////////////////////////////////////////////
        MOV  PSW,#0             ;INST 92 // RR A (92)
        MOV  A,#129
        RR   A
        SUBB A,#192
        JZ   DONE_92
        MOV  P1,#92
        LJMP FAILED
DONE_92:
;//////////////////////////////////////////////////
        MOV  PSW,#0             ;INST 93 // RRC A (93)
        MOV  A,#3
        RRC  A
        SUBB A,#0       ;A(1)-C(1)-0
        JZ   DONE_93
        MOV  P1,#93
        LJMP FAILED
DONE_93:

;//////////////////////////////////////////////////
        MOV  PSW,#0             ;INST 101 // SWAP A (101)
        MOV  A,#23H
        SWAP A
        SUBB A,#32H
        JZ   DONE_101
        MOV  P1,#101
        LJMP FAILED
DONE_101:
```

Data Transfer Operations

Following assembly code fragment is used to verify various data transfer instructions

```
;//////////////////////////////////////////////////
        MOV  PSW,#0
        MOV  R0,#10              ;INST 52 // MOV A,Rn (52)
        MOV  A,R0
        SUBB A,#10
        JZ   DONE_52
        MOV  P1,#52
        LJMP FAILED
DONE_52:
;//////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 53 // MOV A,direct (53)
        MOV  127,#10
        MOV  A,127
        SUBB A,#10
        JZ   DONE_53
        MOV  P1,#53
        LJMP FAILED
DONE_53:
;//////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 54 // MOV A,@Ri (54)
        MOV  R0,#127
        MOV  127,#10
        MOV  A,@R0
        SUBB A,#10
        JZ   DONE_54
        MOV  P1,#54
        LJMP FAILED
DONE_54:
;//////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 55 // MOV A,#data (55)
        MOV  A,#10
        SUBB A,#10
        JZ   DONE_55
        MOV  P1,#55
        LJMP FAILED
DONE_55:
;//////////////////////////////////////////////////
        MOV  PSW,#0              ; INST 56  // MOV Rn,A (56)
        MOV  A,#10
        MOV  R0,A
        CLR  A
        MOV  A,R0
        SUBB A,#10
        JZ   DONE_56
        MOV  P1,#56
        LJMP FAILED
DONE_56:
;//////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 57 // MOV Rn,direct (57)
        MOV  127,#10
        MOV  R0,127
        MOV  A,R0
        SUBB A,#10
        JZ   DONE_57
        MOV  P1,#57
        LJMP FAILED
DONE_57:
;//////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 58 // MOV Rn,#data (58)
        MOV  R0,#10
        MOV  A,R0
        SUBB A,#10
        JZ   DONE_58
        MOV  P1,#58
        LJMP FAILED
DONE_58:
;//////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 59 // MOV direct,A (59)
        MOV  A,#10
        MOV  127,A
        CLR  A
        MOV  A,127
        SUBB A,#10
        JZ   DONE_59
        MOV  P1,#59
        LJMP FAILED
DONE_59:
;///////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 60 // MOV direct,Rn (60)
        MOV  R0,#10
        MOV  127,R0
        MOV  A,127
        SUBB A,#10
        JZ   DONE_60
        MOV  P1,#60
        LJMP FAILED
DONE_60:
;///////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 61 // MOV direct,direct (61)
        MOV  127,#10
        MOV  126,127
        MOV  A,126
        SUBB A,#10
        JZ   DONE_61
        MOV  P1,#61
        LJMP FAILED
DONE_61:
;/////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 62 // MOV direct,@Ri (62)
        MOV  127,#10
        MOV  R0,#127
        MOV  126,@R0
        MOV  A,126
        SUBB A,#10
        JZ   DONE_62
        MOV  P1,#62
        LJMP FAILED
DONE_62:
;///////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 63 // MOV direct,#data (63)
        MOV  127,#10
        MOV  A,127
        SUBB A,#10
        JZ   DONE_63
        MOV  P1,#63
        LJMP FAILED
DONE_63:
;///////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 64 // MOV @Ri,A (64)
        MOV  A,#10
        MOV  R0,#127
        MOV  @R0,A
        CLR  A
        MOV  A,127
        SUBB A,#10
        JZ   DONE_64
        MOV  P1,#64
        LJMP FAILED
DONE_64:
;///////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 65 // MOV @Ri,direct (65)
        MOV  127,#10
        MOV  R0,#126
        MOV  @R0,127
        MOV  A,126
        SUBB A,#10
        JZ   DONE_65
        MOV  P1,#65
        LJMP FAILED
DONE_65:
;///////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 66 // MOV @Ri,#data (66)
        MOV  R0,#127
        MOV  @R0,#10
        MOV  A,127
        SUBB A,#10
        JZ   DONE_66
        MOV  P1,#66
        LJMP FAILED
DONE_66:
;//////////////////////////////////////////////////
        MOV  PSW,#0               ;INST 67 // MOV C,bit (67)
        MOV  A,#1
        MOV  C,ACC.0
        JC   DONE_67
        MOV  P1,#67
        LJMP FAILED
DONE_67:
;//////////////////////////////////////////////////
        MOV  PSW,#0               ;INST 68 // MOV bit,C (68)
        CPL  C
        MOV  ACC.0,C
        CPL  C
        SUBB A,#1
        JZ   DONE_68
        MOV  P1,#68
        LJMP FAILED
DONE_68:
;//////////////////////////////////////////////////
        MOV  PSW,#0               ;INST 69 // MOVX @DPTR,A (69)
        MOV  DPTR,#1234H ;           MOVX A,DPTR
        MOV  A,#55h
        MOVX @DPTR,A
        MOV  A,#0
        MOVX A,@DPTR
        SUBB A,#55h
        JZ   DONE_69
ERROR_69:
        MOV  P1,#69
        LJMP FAILED
DONE_69:
;//////////////////////////////////////////////////
        MOV  PSW,#0               ;INST 70 // MOVC A,@A+DPTR (70)
        MOV  A,#0
        MOV  DPTR,#DB_TBL
        MOVC A,@A+DPTR
        SUBB A,#66H
        JZ   DONE_70
DB_TBL:
        DB   66H
        DB   77H
ERROR_70:
        MOV  P1,#70
        LJMP FAILED
DONE_70:
;//////////////////////////////////////////////////
        MOV  PSW,#0               ;INST 71 // MOVC A,@A+PC (71)
        MOV  A,#4
        MOVC A,@A+PC
        SUBB A,#66H
        JZ   DONE_71
        DB   66H
ERROR_71:
        MOV  P1,#71
        LJMP FAILED
DONE_71:
```

Flow Control Instructions

To test the flow control instructions following assembly codes are used;

```
;//////////////////////////////////////////////////
        MOV  PSW,#0               ;INST 10 // AJMP (10)
        AJMP DONE_10
        MOV  P1,#10
        LJMP FAILED
DONE_10:

;//////////////////////////////////////////////////
        MOV  PSW,#0               ;INST 19 // CJNE A,direct,rel (19)
        MOV  A,#128
        MOV  100,#128
        CJNE A,100,ERROR_19
        MOV  A,#127
        CJNE A,100,CHECK_C_19
ERROR_19:
        MOV  P1,#19
        LJMP FAILED
CHECK_C_19:
        JC   DONE_19
        MOV  P1,#19
        LJMP FAILED
DONE_19:
;///////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 20 // CJNE A,#data,rel (20)
        MOV  A,#128
        CJNE A,#128,ERROR_20
        MOV  A,#127
        CJNE A,#128,CHECK_C_20
ERROR_20:
        MOV  P1,#20
        LJMP FAILED
CHECK_C_20:
        JC   DONE_20
        MOV  P1,#20
        LJMP FAILED
DONE_20:
;///////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 21 // CJNE Rn,#data,rel (21)
        MOV  R1,#128
        CJNE R1,#128,ERROR_21
        MOV  R1,#127
        CJNE R1,#128,CHECK_C_21
ERROR_21:
        MOV  P1,#21
        LJMP FAILED
CHECK_C_21:
        JC   DONE_21
        MOV  P1,#21
        LJMP FAILED
DONE_21:
;///////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 22 // CJNE @Ri,#data,rel (22)
        MOV  R1,#100
        MOV  100,#128
        CJNE @R1,#128,ERROR_22
        MOV  100,#127
        CJNE @R1,#128,CHECK_C_22
ERROR_22:
        MOV  P1,#22
        LJMP FAILED
CHECK_C_22:
        JC   DONE_22
        MOV  P1,#22
        LJMP FAILED
DONE_22:

;///////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 35 // DJNZ Rn,rel (35)
        MOV  R0,#10
        DJNZ R0,JUMP_35          ;Should jump
        MOV  P1,#35
        LJMP FAILED
JUMP_35:
        MOV  R0,#1
        DJNZ R0,NOT_JUMP_35      ;Should not jump
        AJMP DONE_35
NOT_JUMP_35:
        MOV  P1,#35
        LJMP FAILED
DONE_35:
;///////////////////////////////////////////////////
        MOV  PSW,#0              ;INST 36 // DJNZ direct,rel (36)
        MOV  127,#10
        DJNZ 127,JUMP_36 ;Should jump
        MOV  P1,#36
        LJMP FAILED
JUMP_36:
        MOV  127,#1
        DJNZ 127,NOT_JUMP_36      ;Should not jump
        AJMP DONE_36
NOT_JUMP_36:
        MOV  P1,#36
        LJMP FAILED
DONE_36:

;///////////////////////////////////////////////////
        MOV  A,#16                ;INST 42 // JB bit,rel (42)
        JB   ACC.4,DONE_42
        MOV  P1,#42
        LJMP FAILED
DONE_42:
;///////////////////////////////////////////////////
        MOV  A,#8                 ;INST 43 // JBC bit,rel (43)
        JBC  ACC.3,CHECK_BIT_43
        MOV  P1,#43
        LJMP FAILED
CHECK_BIT_43:
        JZ   DONE_43
        MOV  P1,#43
        LJMP FAILED
DONE_43:
;///////////////////////////////////////////////////
        MOV  PSW,#0               ;INST 44 // JC rel (44)
        JC   ERROR_44
        CPL  C
        JC   DONE_44
ERROR_44:
        MOV  P1,#44
        LJMP FAILED
DONE_44:

;///////////////////////////////////////////////////
        MOV  A,#4                 ;INST 45 // JMP @A+DPTR (45)
        MOV  DPTR,#JMP_TBL
        JMP  @A+DPTR
JMP_TBL:
        AJMP JUMP_0
        AJMP JUMP_2
        AJMP JUMP_4
        AJMP JUMP_6
JUMP_0:
JUMP_2:
JUMP_6:
        MOV  P1,#43
        LJMP FAILED
JUMP_4:
;///////////////////////////////////////////////////
        MOV  A,#4                 ;INST 46 // JNB bit,rel (46)
        JNB  ACC.2,ERROR_46
        JNB  ACC.1,DONE_46
ERROR_46:
        MOV  P1,#46
        LJMP FAILED
DONE_46:
;///////////////////////////////////////////////////
        MOV  PSW,#0               ;INST 47 // JNC rel (47)
        CPL  C
        JNC  ERROR_47
        CPL  C
        JNC  DONE_47
ERROR_47:
        MOV  P1,#47
        LJMP FAILED
DONE_47:
;///////////////////////////////////////////////////
        MOV  PSW,#0               ;INST 48 // JNZ rel (48)
        MOV  A,#0
        JNZ ERROR_48
        MOV A,#1
        JNZ DONE_48
ERROR_48:
        MOV  P1,#48
        LJMP FAILED
DONE_48:
;///////////////////////////////////////////////////
        MOV  PSW,#0               ;INST 49 // JZ rel (49)
        MOV  A,#2
        JZ   ERROR_49
        MOV  A,#0
        JZ   DONE_49
ERROR_49:
        MOV  P1,#49
        LJMP FAILED
DONE_49:
;///////////////////////////////////////////////////
    LJMP DONE_51 ;INST 51 // LJMP (51)
        MOV  P1,#51
        LJMP FAILED
DONE_51:
```

ISR Routines and CPU Configuration

```
;///////////////////////////////////////////////////
        RSEG  STACK
        DS    10H  ; 16 Bytes Stack

        CSEG  AT   0     ; Reset
        USING   0  ; Register-Bank 0
; Execution starts at address 0 on power-up.
        JMP   START

        ; 0 EXTERNAL INT 0 0003h
        ; 1 TIMER/COUNTER 0 000Bh
        ; 2 EXTERNAL INT 1 0013h
        ; 3 TIMER/COUNTER 1 001Bh
        ; 4 SERIAL PORT 0023h
        CSEG  AT   0x03          ;//Interrupt-0
        JMP   ISR_INT0

        CSEG  AT   0x0B          ;//Timer Interrupt-0
        JMP   ISR_TINT0

        CSEG  AT   0x23          ;//Serial Interrupt
        JMP   ISR_SINT

        RSEG  PROG
START:
       MOV  SP,#STACK-1    ; first set Stack Pointer
        MOV  IE,#0x93         ; Enable ints, enable serial int, tint-0, ext-0
        MOV  SCON,#0x90     ; mode 2: 9-bit UART, fixed rate,  enable receiver
        MOV  TH1,#0xF3      ; reload value 2400 baud
        MOV  TH0,#0x08      ; reload value 2400 baud
        MOV  TMOD,#0x22     ; timer 1 mode 2: 8-Bit reload
        MOV  P2,#01           ; this is to check Serial Int
        MOV  P3,#01           ; this is to check Int-0
        MOV  P0,#01           ; this is to check Timer Int-0
        MOV  TCON,#0x15     ; enable Timer 0, Ext-Ints are Edge Sensitive

;/////////////// INTERRUPTS //////////////////////////////
        RSEG PROG

ISR_INT0:
        MOV  P3,#02
        RETI

ISR_TINT0:
        MOV  P0,#02
        RETI

ISR_SINT :
        MOV  P2,#02
        RETI


;///////////////// DONE  /////// All instructions passed
        MOV  P1,#127             ;
        MOV  A,P2                ; check serial int result
        SUBB A,#2
        JZ   S_DONE
        MOV  P1,#112
S_DONE:
        MOV  A,P3                ; check int-0 result
        SUBB A,#2
        JZ   INT0_DONE
        MOV  P1,#113
INT0_DONE:

        MOV  A,P0                ; check timer int-0 result
        SUBB A,#2
        JZ   TINT0_DONE
        MOV  P1,#114
TINT0_DONE:


;///////////////// FAIL  //////  P1 shows which one failed
FAILED:  NOP
        MOV  SBUF, P1

WAIT_HERE:
        MOV  A,#127
WAIT1:   DEC  A
        JNZ   WAIT1

        NOP
```

D 8051 Variants

Variant	Pins	Mfg	RAM	CODE	XRAM	Notes
C8051F0X	64	Cygnal	256	32KF	0	20 Mips,12bADC.DAC,SPI,i2c,PCA
MCS251	44	Intel	1K	16K	0	16 Bit 80x51FX, also Temic
MCS151	44	Intel	1K	16K	0	Fast 80x51FX
SABC509L	100qf	Siemens	256	64Kx	3K	ALU, PWM, CaptComp 2UART, 10b A/D
SABC517A	84	Siemens	256	64Kx	2K	ALU, 8 PWM, 2UART, 10b A/D
SABC515	80qf	Siemens	256	64Kx	2K	10A/D,XRAM,OWD,CAN V2B, Xt2
73D2910	100qfp	SSI	256	128Kx	0	80C52+Ports+HDLC
78C438	84.100	Winbond	256	64Kx	1Mx	40Mhz, more ports C52
78E354	68.48	Winbond	256	16KF	256	20MHz, Video Monitor
SABC515A	68	Siemens	256	64Kx	1K	515+10bA/D,1K XRAM,BRG,OWD
MAX7651	Qfp64	Maxim	256	16KF	256	Turbo+ 12 Bit ADC, Dual UART
SABC508	64	Siemens	256	32K	1K	
SS89C578	68	Siliconians	256	32KF	256	10ChA/D,256XRAM,SPIx2 FLASH
DS87C550	PLC68	Dallas	256	8K	1K	Turbo 80C552, -i2c,+UART,PWM
80CE558	80qfp	Philips	256	64Kx	768	Enhanced 80C552, Sep i2c, RSO
80C535A	68	Siemens	256	32K	1K	515+10bA/D,1K XRAM,BRG,OWD
80C592	68	Philips	256	64Kx	256	NOT FOR NEW DESIGNS!!
80C552	68	Philips	256	64Kx	0	10 Bit A/D, WDOG, PWM
87C552	68	Philips	256	8K	0	10b A/D, i2c, CaptComp, PWM
80C562	68	Philips	256	64Kx	0	8b A/D, i2c, CaptComp, PWM
SABC505	44	Siemens	256	64Kx	256	8bA/D,XRAM,OWD,CAN V2B, Xt2
SABC504	44	Siemens	256	64Kx	256	10bA/D,XRAM,OWD,DC Motor PWM
ADUC812	44	AnaDev	256	8KF	0	c51+FLash+EE+ADC+DAC
SABC541	44	Siemens	256	8K	256	USB Bus Controller 1.5/12MHz
87C451	68	Philips	128	4K	0	7 Ports, 1 Handshake
80C451	68	Philips	128	64Kx	0	7 Ports, 1 Handshake
87C453	68	Philips	256	8K	0	7 Ports, 1 Handshake
83CL580	56,64	Philips	256	6K	0	LV 8052+ADC+i2c+More INTs, WDOG
W77LE58	40	Winbond	256	64Kx	1K	FAST, 2 DPTR 2 UART P4
80C320	40	Dallas	256	64Kx	0	FAST, 2 DPTR 2 UART VRST
80C310	40	Dallas	256	64Kx	0	Simpler 80C320 e62.5Mhz
87C520	40	Dallas	256	16K	1K	16K OTP enhanced 80C320
T89C51RD	40	Temic	256	64KF 1K	X	
P89C51RD	40	Philips	1K	64KF	0	80C51FX+PCA,1K, Flash
P89C51RC	40	Philips	512	32KF	0	80C51FX+PCA,512 Flash
87C51FX	40	Philips	256	32K	0	87C51FA,FB,FC FAMILY
T89C51CC	44	Temic	256	32KF	1K	2K EE, and PCA ISP.IAP
T89C51RB	40	Temic	256	16KF	1K	SPI, PCA ISP.IAP
T89C51RC	40	Temic	256	32KF	1K	SPI, PCA ISP.IAP
80C575	40	Philips	256	64Kx	0	8052+PCA,AnalogComp,WDOG,RSTLo
87C575	40	Philips	256	8K	0	8052+PCA,AnalogComp,WDOG,RSTLo
80C576	40	Philips	256	8K	0	8052+PCA,UPI,A/D,PWM,WDOG,VRSTLo
87C576	40	Philips	256	8K	0	8052+PCA,UPI,A/D,PWM,WDOG,VRSTLo
SABC501	40	Siemens	256	64Kx	0	40MHz Enhanced 8052 U/D
SABC502	40	Siemens	256	64Kx	256	8052+XRAM+8DP+WD+BRG+OWD
80C528	40	Philips	256	64Kx	256	8052+Wdog, XRAM
87C528	40	Philips	256	32K	256	8052+Wdog, XRAM
87F51RC	44	Atmel	256	32KF	256	OTP Flash, XRAM
87C524	40	Philips	256	16K	256	16K 87C528
80C550	40	Philips	128	4K	0	8b A/D WDog
80CL781	40	Philips	256	64Kx	0	Low Voltage 8052, More INTs, WDOG
83CL781	40	Philips	256	16K	0	Low Voltage 8052, More INTs, WDOG

80CL782	40	Philips	256	64Kx	0	Low Voltage, faster 781
89S8252	40.44	Atmel	256	8KF	2KE	FLASH,8K+2KEE,WDOG,SPI,ISP
89S53	40.44	Atmel	256	12KF	0	89S8252 minus EEPROM
89S52	40.44	Atmel	256	8KF	0	WDog ISP Std C52
89S51	40.44	Atmel	256	4KF	0	WDog ISP Std C51
89C55	40.44	Atmel	256	20KF	0	FLASH, Fast,LV 87C52+20K
89C52	40.44	Atmel	256	8KF	0	FLASH, Fast,LV 87C52
87C54	40	Intel	256	16K	0	16K 87C52i
87C58	40	Intel	256	32K	0	32K 87C52i
87C52	40	Intel	256	8K	0	8052+U/D+OscO+4Li
80C154	40	Temic	256	64Kx	0	Enhanced 8052 (also OKI)
83C154D	40	Temic	256	32K	0	Enhanced 8052
83C154	40	OKI	256	16K	0	Enhanced 8052
80C654	40	Philips	256	64Kx	0	i2c
87C652	40	Philips	256	8K	0	i2c
87C654	40	Philips	256	16K	0	i2c
83CE654	44qfp	Philips	256	16K	0	i2c, low RFI 654
DS5000	40	Dallas	128	32KR	32K	80x51 Secure+ NV support, BootLdr
DS2250	40sim	Dallas	128	32K	32K	As 5000, but smarter package
DS5001	80qfp	Dallas	128	64Kx	64K	Better 5000, + RPC + BatSw
80C851	40	Philips	128	64Kx	0	8051+256B EEPROM
83C852	6	Philips	256	6K	0	2K EEPROM SmartCard 80x51, Die, ALU
8052	40	All	256	64Kx	0	8051+Timer2
8752	40	Intel	256	8K	0	8051+Timer2
80C52	40	Siemens	256	64Kx	0	8051+Timer2,Philips,Oki,Temic
78E52	40	Winbond	256	8KF	0	40Mhz, FLASH C52
78C32	40	Winbond	256	64Kx	0	40MHz C32, Static
80CL410	40	Philips	128	64Kx	0	Low Voltage, More INTs i2c-UART
80CL31	40	Philips	128	64Kx	0	Low Voltage, More Ints 80x51
80CL610	40	Philips	256	64Kx	0	Low Voltage, More INTs i2c-UART
83CL411	40	Philips	256	64Kx	0	80CL31 with 256 RAM, No T2
89C51	40.44	Atmel	128	4KF	0	FLASH,Fast,LV 87C51
87C51	40	All	128	4K	0	Core processor,UART,Tmr0,Tmr1
78E51	40	Winbond	128	4KF	0	40MHz FLASH C51
T87C5112	52	Temic	256	16K	0	LPC with ADC,PCA
T87C5111	24	Temic	256	4K	0	LPC with ADC,PCA
87C752	28	Philips	64	2KE	0	87751+ A/D, PWM
87C749	28	Philips	64	2KE	0	87C752 - i2c
87C751	24	Philips	64	2KE	0	Small size, bit i2c
87C748	24	Philips	64	2KE	0	87C751 - i2c
87C750	24	Philips	64	1KE	0	Small size,
87LPC76X	20	Philips 128	4K	0	X	
89C4051	20	Atmel	128	4KF	0	4K version of 89C2051
89C2051	20	Atmel	128	2KF	0	20Pin 89C51,+AnaComp+LED
89C1051U	20	Atmel	64	1KF	0	20Pin 1051+UART
89C1051	20	Atmel	64	1KF	0	20Pin 2051 -uart,timer1

E Standard 16-Bit CRC Code

```
;// crc16.a51
;// ASSEMBLY CODE FOR CRC-16 FOR SDLC
;// http://www.keil.com/support/docs/488.htm
;// Copyright 1995-1999 Keil Software, Inc.

$title(16 bit crc for polynomial X16+X12+X5+1 for SDLC)
public  crc_gen,?crc_gen?byte,crc16
        ;CRC subroutine
        ;CRC uses most significant bit
        ;r6, r7 is CRC residue
        ;acc is data byte

crc_code segment code
crc_data segment data

        rseg    crc_data
?crc_gen?byte:
cdat:   ds      1
cres:   ds      2

        rseg    crc_code
        using   0

crc_gen:mov     a,cdat
        mov     r1,cres
        mov     r0,cres+1
crc16:  xrl     a,r1            ; xor data byte
        mov     r3,a            ; temp store in r3
        swap    a               ; rotate right four bits
        mov     r2,a            ; temp save in r2
        xrl     a,r3
        anl     a,#0f0h
        xrl     a,r0
        mov     r4,a            ; temp save
        mov     a,r2
        rl      a
        anl     a,#1fh          ; mask
        xrl     a,r4
        mov     r4,a            ; save
        mov     a,r3
        rl      a
        anl     a,#1
        xrl     a,r0
        xrl     a,r4
        mov     r7,a            ;low byte is complete
        mov     a,r2
        anl     a,#0fh
        xrl     a,r3
        mov     r1,a
        mov     a,r2
        xrl     a,r3
        rl      a
        anl     a,#0e0h
        xrl     a,r1
        mov     r6,a

        ret
        end


// C sample code for using CRC functions
// http://www.keil.com/support/docs/488.htm
// Copyright 1995-1999 Keil Software, Inc.

#include <reg51.h>
#include <stdio.h>

extern alien unsigned int crc_gen(unsigned char b,unsigned int residue);

void main (void)  {                            /* main program */

        unsigned char i, test;
        unsigned char message[50] =
                       {0x50,0x51,0x52,0x53,0x54,0x55,0x56,0x57,0x58,0x59,
                        0x60,0x61,0x62,0x63,0x64,0x65,0x66,0x67,0x68,0x69,
                        0x70,0x71,0x72,0x73,0x74,0x75,0x76,0x77,0x78,0x79,
                        0x80,0x81,0x82,0x83,0x84,0x85,0x86,0x87,0x88,0x89,
                        0x90,0x91,0x92,0x93,0x94,0x95,0x96,0x97,0x98,0x99 };
                        // 16 bit CRC of given array is 0xE664

        unsigned int residue;

        SCON = 0x90;    /* SCON */                   /* setup serial port control */
        TMOD = 0x20;    /* TMOD */
        TCON = 0x69;    /* TCON */
        TH1 =  0xf3;    /* TH1 */


        residue=0;
        for(i=0;i<sizeof(message);i++)
        {
          residue=crc_gen(message[i],residue);
        }

        if(residue == 0xE664 ){
                // 0xE664 is the correct result of 16 bit CRC of the given array
                SBUF = 127 ;           // test passed mesage to serial monitor
        }else{
                SBUF = 0 ;             // test failed
        }

        // delay elements
        for(i = 0; i <255; i++ ){
                residue ++ ;
        }
}
```