SYSTEMC IMPLEMENTATION WITH ANALOG MIXED SIGNAL MODELING FOR A MICROCONTROLLER

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

YAKUP MURAT MERT

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

MAY 2007

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. Hasan Güran	(METU, EE)	_____
Prof. Dr. Murat Aşkar	(METU, EE)	_____
Prof. Dr. Tayfun Akın	(METU, EE)	_____
Assist. Prof. Dr Cüneyt Bazlamaçcı	(METU, EE)	_____
M.Sc. Lokman Kesen	(ASELSAN)	_____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name: Yakup Murat MERT
Signature

ABSTRACT

SYSTEMC IMPLEMENTATION WITH ANALOG AND MIXED SIGNAL MODELING FOR A MICROCONTROLLER

MERT, Yakup Murat
M.Sc., Department of Electrical and Electronics Engineering
Supervisor: Prof. Dr. Murat Aşkar

May 2007, 129 pages

In this thesis, an 8-bit microcontroller, PIC 16F871, has been implemented using SystemC with classical hardware design methods. Analog modules of the microcontroller have been modeled behaviorally with SystemC-AMS which is the analog and mixed signal extensions for the SystemC. SystemC-AMS provides the capability to model non-digital modules and synchronization with the SystemC kernel. In this manner, electronic systems that have both digital and analog components can be described and simulated very effectively.

The PIC 16F871 is a well known and very common microcontroller. Its architecture, peripheral modules and analog components makes this microcontroller pretty good model for a System on Chip (SoC) concept. Designed microcontroller's peripheral modules, instruction set and addressing modes have been verified utilizing the test codes. Besides, designed microcontroller has been tested with 16-bit CRC code. Moreover, a synchronous demodulator system that involves designed microcontroller and additional analog units has been constructed and simulated. Finally, SystemC to hardware flow has

been demonstrated with implementation of arithmetic logic unit of the 16F871 into FPGA based hardware.

Keywords: SystemC, SystemC-AMS, PIC 16F871, Microcontroller

ÖZ

BİR MİKRODENETLEYİCİNİN ANALOG VE KARMA SİNYAL OLARAK MODELLENMESİ VE SYSTEMC İLE GERÇEKLEŞTİRİLMESİ

MERT, Yakup Murat
Yüksek Lisans., Elektrik Elektronik mühendisliği bölümü
Tez Yöneticisi: Prof. Dr. Murat Aşkar

Mayıs 2007, 129 sayfa

Bu tezde 8 bitlik mikro denetleyici olan PIC 16F871, SystemC dilinin klasik donanım tasarlama yöntemleriyle tasarlanmıştır. Mikrodenetleyicinin analog bileşenleri ise SystemC dilinin analog ve karma sinyal eklentisi olan SystemC-AMS ile modellenmiştir. SystemC-AMS, analog ve sayısal olmayan modüllerin modellenmesini ve de SystemC çekirdeği ile eşgüdümü sağlamaktadır. Böylelikle hem sayısal hem de analog bileşenleri içeren elektronik sistemler etkili bir şekilde modellenebilir ve çalıştırılabilir.

PIC 16F871, çok iyi bilinen ve oldukça yaygın bir mikrodenetleyicidir. Mimarisi, çevresel birimleri ve analog bileşenleri onu SoC kavramı için oldukça iyi bir örnek yapmaktadır. Tasarlanan mikrodenetleyicinin çevresel birimleri, komut seti ve adresleme yöntemlerinin doğru çalıştığı sınama kodlarıyla gösterilmiştir. Bunun yanında, 16 bitlik CRC koduyla ayrıca test edilmiştir. Bunun yanında, tasarlanan mikrodenetleyici ve ek analog birimler kullanılarak eşzamanlı demodulator örneği çalıştırılmıştır. Son olarak,

SystemC dilinden donanım sentezleme yöntemi, bu mikrodenetleyicinin aritmetik ve mantık biriminin FPGA tabanlı bir donanım ile sentezlenmesiyle gösterilmiştir.

Anahtar kelimeler: SystemC, SystemC-AMS, PIC 16F871, Mikrodenetleyici

ACKNOWLEDGEMENTS

I would like to thank my supervisor Prof. Dr. Murat Aşkar for his support throughout my studies at Middle East Technical University. His ideas and experiences guided me during the entire thesis work.

I would like to extend my thanks to all lecturers at the Department of Electrical and Electronics Engineering, who greatly helped me to store the basic knowledge onto which I have built my thesis.

I would like to forward my appreciation to all my friends particularly Salih Zengin, who contributed to my thesis with their continuous encouragement.

I would like to thank my parents and my brother for standing by me through these years. Their constant encouragement and moral support is the primary reason for whatever little I have achieved so far in my life.

TABLE OF CONTENTS

ABSTRACT iv
ÖZ vi
ACKNOWLEDGEMENTS viii
TABLE OF CONTENTS ix
LIST OF TABLES xii
LIST OF FIGURES xiv
CHAPTERS
1. INTRODUCTION 1
2. SYSTEMC AS AN INTEGRATED ANALOG AND DIGITAL DESIGN ENVIRONMENT 5
2.1 Analog And Mixed Signal Extensions To SystemC 5
2.2 Synchronous Data Flow (SDF) Module 7
2.2.1 Synchronous Data Flow (SDF) Module Member Functions 8
2.2.2 Synchronous Data Flow (SDF) Module Ports 8
2.3 Analog Linear Behavioral Models 10
2.4 Frequency Domain Specifications 12
2.5 Linear Electrical Networks 14
2.6 A Design Example 15
3. MAIN PROPERTIES OF THE PIC 16F871 CORE 20
3.2 Type of Memories 22
3.1.1 Program Memory Organization 22
3.1.2 Data Memory Organization 23
3.2 Special Function Registers 26
3.3 Addressing Modes Of The 16F871 26
3.3.1 Direct Addressing 26
3.3.2 Indirect Addressing 27
3.4 Program Flow Control Operations 28

3.4.1 Unconditional Branching 28
3.4.2 Direct Calls 29
3.4.3 Computed GOTO 30
3.4.4 Conditional Skipping 30
3.4.5 Interrupts 31
3.4.6 Sleep 31
4. DESIGN OF THE EXTENDED PIC 16F871 32
4.1 The Main State Machine 32
4.2 Instruction Set 36
4.3 Arithmetic Logic Unit (ALU) 37
4.5 Data Memory and Special Function Registers 38
4.6 The Parallel Slave Port (PSP) 42
4.7 The Universal Synchronous/Asynchronous Receiver Transmitter (USART) 43
4.8 Timers 45
4.8.1 The Timer 0 45
4.8.2 The Timer 1 45
4.8.3 The Timer 2 46
4.8.4 The Watch-dog Timer 46
4.9 The Capture/Compare/PWM module (CCP) 47
4.10 The A/D Converter 48
4.11 The D/A Converter 49
4.12 Interrupt Controller 50
5. VERIFICATION OF THE DESIGNED CORE 52
5.1 The Main Core 54
5.1.1 The Instruction Architecture Test 56
5.1.2 Addressing Mode Verifications 59
5.2 The Parallel Slave Port (PSP) 60
5.3 The Serial Port (USART) 61
5.4 Timer Modules 62
5.6 The Capture, Compare, PWM Module (CCP) 64
5.7 The A/D Converter Module 66
5.8 The D/A Converter Module 68
5.9 Interrupts 69

5.10 CRC Algorithms 70
5.11 A Demodulator Example 72
6. CONCLUSIONS 74
REFERENCES 77
APPENDICES
A. THE PIC 16F871 INSTRUCTION SET SUMMARY 80
B. ASSEMBLY VERIFICATION CODES 89
B.1 Instruction Set Verification Code 89
B.2 Addressing Modes Verification Codes 97
C. SYSTEMC-AMS MODEL DESCRIPTIONS 100
C.1 Analog Linear Behavioral Models 100
C.2 Frequency Domain Specification Functions 101
C.3 SystemC-AMS Linear Electrical Library Elements 102
D. BIT MAPS OF SOME SPECIAL FUNCTION REGISTERS 107
E. HARDWARE SYNHESIS OF ARITHMETIC LOGIC UNIT 119
F. ASSEMBLY CODE OF THE DEMODULATOR EXAMPLE AND THE 16 BIT CRC TEST PROGRAM 123

LIST OF TABLES

Table 3-1: Bank selection using RP0 and RP0 bits 23
Table 4-1: Special Function registers reside in (a) Bank 0 (b) Bank 1 (c) Bank 2 (d) Bank 3 39
Table 5-1: Expected results from 16-bit CRC algorithm..... 71
Table A- 1: Descriptions of (a) ADDLW (b) ADDWF (c) ANDLW (d) ANDWF instructions. 80
Table A- 2 :Descriptions of (a) BCF (b) BTFSS (c) BSF (d) ANDWF instructions. 81
Table A- 3 :Descriptions of (a) CALL (b) CLRWDT (c) CLRF (d) CLRW instructions. 82
Table A- 4: Descriptions of (a) COMF (b) SLEEP (c) DECF (d) DECFSZ instructions. 83
Table A- 5: Descriptions of (a) INCF (b) INCFSZ (c) GOTO (d) IORLW instructions. 84
Table A- 6: Descriptions of (a) IORWF (b) MOVLW (c) MOVF (d) MOVWF instructions. 85
Table A- 7: Descriptions of (a) NOP (b) RETLW (c) RETFIE (d) RETURN instructions. 86
Table A- 8: Descriptions of (a) RLF(b) SUBLW (c) RRF (d) SUBWF instructions. 87
Table A- 9: Descriptions of (a) XORWF (b) XORLW (c) SWAPF instructions..... 88
Table D- 1: Bit map of `STATUS` register 107
Table D- 2: Bit map of `OPTION_REG` register..... 108
Table D- 3: Bit maps of (a) `TRISE` (b) `EECON1` registers 109
Table D- 4: Bit map of `TXSTA` register 110
Table D- 5: Bit map of `RCSTA` register..... 111
Table D- 6: Bit map of (a) `T1CON` (b) `T2CON` registers 112
Table D- 7: Bit map of `CCPCON` register 113
Table D- 8: Bit map of (a) `ADCON0` (b) `DACON` registers 114
Table D- 9: Bit map of (a) `ADCON1` register (b) `ADCON`<3:0> bits 115

Table D- 10: Bit map of (a) INTCON register 116
Table D- 11: Bit map of (a) PIR1 (b)PIR2 registers 117
Table D- 12: Bit map of (a) PIE1 (b)PIE2 registers 118
Table E- 1: Device Utilization Summary 120
Table E- 2: Timing Summary 121
Table E- 3: MAP Report Summary 121
Table E- 4 : Summary of Post Place and Route Static Timing Report 122

LIST OF FIGURES

Figure 2-1: Layered approach of SystemC-AMS 5
Figure 2-2: SystemC and SystemC-AMS design flow 6
Figure 2-3: General form of SDF module [9] 9
Figure 2-4 : (a) Schematic description and transfer function of Low-Pass filter (b) SystemC-AMS model of Low-Pass filter [9] 11
Figure 2-5: (a) Schematic description and transfer function of digital comb filter (b) SystemC-AMS model of digital comb filter [9] 13
Figure 2-6: (a) SystemC-AMS model of a linear network (b) schema of the network [9] 14
Figure 2-7: Test-bench of the low-pass filter 15
Figure 2-8: SystemC-AMS model of the low-pass filter [9] 16
Figure 2-9: SystemC-AMS model of (a) sinus signal source (b) digital gain controller module [9] 17
Figure 2-10: Main source file of the LPF [9] 17
Figure 2-11: Output waveform at (a) 5 kHz (b) 20 kHz 18
Figure 2-12: Effect of the digital gain controller. 19
Figure 3-1 : Block digram of PIC 16F871 [23] 21
Figure 4-1:Block diagram of extended 16F871 microcontroller 33
Figure 4-2:Clock/Instruction cycle of main core 34
Figure 4-3: Block diagram of ALU 35
Figure 4-4 : Instruction format of the 16F871 37
Figure 5-1: Testbench and the extended PIC 16F871 design for testing 53
Figure 5-2: Test-ROM generation flow 54
Figure 5-3: ROM signals of 16F871 55
Figure 5-4: RAM signals of 16F871 55
Figure 5-5: SLEEP instruction and power-down mode 58
Figure 5-6: Addressing modes (a) direct addressing (b) indirect addressing 59
Figure 5-7: Parallel port write operation 60

Figure 5-8: Parallel port read operation 61
Figure 5-9: USART transmission operation 62
Figure 5-10: USART reception operation 62
Figure 5-11:Timer 1 operation 63
Figure 5-12:Timer 2 operation 63
Figure 5-13:CCP capture mode 65
Figure 5-14:CCP compare mode 65
Figure 5-15:CCP PWM mode 65
Figure 5-17:A/D converter transfer function (a) -1V / 6V range view (b) Close view of transfer function around 4V. Vref+ is equal to 5V and Vref- is equal to 0. 67
Figure 5-16:A/D conversion process 68
Figure 5-18: D/A converter sinus signal output 69
Figure 5-19: Interrupt and awakening from SLEEP mode. 70
Figure 5-20: Obtained results from 16-bit CRC verification code 71
Figure 5-21 Demodulator setup 72
Figure 5-22 Demodulator output for different phase angles 73
Figure E- 1: SystemCrafter Design flow 120

CHAPTER 1

INTRODUCTION

Today, application specific integrated circuits (ASIC) contain millions of transistors and embrace several end product components integrated into the same chip. As technology enhances, design complexity also grows in parallel with the increasing complexity of the ASICs and design process becomes harder to deal with. Besides, handling hardware and software design processes separately fuels the complexity of the design process and increases the design flow. The only way to overcome this problem is combining both hardware and software design processes in the same framework. SystemC is one of the tools that satisfies this important need and closes the gap between hardware and software with co-design and co-simulation concepts. It utilizes C++ syntax for design and supports different levels of abstraction.

Integrating ASICs within a system is known as system-on-chip (SoC). On the other hand, system-on-chip implementations are becoming more and more complex, heterogeneous and include not only software and digital hardware, but also analog/RF and non-electronic components such as sensors or actuators. The design and verification of such complex systems require appropriate design environment and efficient simulation [1]. Current languages and tools such as VHDL-AMS (analog and mixed signal extensions for VHDL) [2], Modelica [3] and Matlab/Simulink [4] are very useful to support the system level design of mixed analog-digital systems. However, they do not offer a single consistent framework in which complex heterogeneous systems can be designed. They are neither simulation efficient enough nor sufficient to interact with the discrete models [5]. These factors imply that, it is crucial to include analog and mixed signal components for any design environment that claims to be applicable to system-on-chip design. SystemC is becoming leading environment for system design but, current SystemC versions still lack support for continuous-time systems [6].

Idea of SystemC-AMS, analog and mixed signal extensions for SystemC, emerged from the need for system level design and verification of heterogeneous systems easily and efficiently. SystemC is already an unmatched environment for digital system modeling since it supports hardware and software co-simulation and co-design, but continuous time systems can not be modeled with SystemC. Design of such systems require additional model of computations (MoC). The rules to model a specified system are known as the model of computation [1]. SystemC currently supports only discrete-event models of computation which is ideal to describe digital systems. However, this model of computation is not appropriate for modeling continuous time systems. Analog and mixed signal extensions aim to close this gap by means of applying another computation method named synchronous (or static) dataflow (SDF) [7]. With libraries provided by SystemC-AMS, SystemC gains capability to design and simulate multiple domain systems in the same framework with very high level abstraction and very good simulation performance.

SystemC-AMS libraries have been developed considering the three main application areas, namely, (i) signal processing applications (telecommunication and multimedia), (ii) RF applications and (iii) power electronics and automotive [5] [7]. Signal processing and RF applications require both time and frequency domain modeling and simulation capabilities. On the other hand, power electronics applications call for electrical network components and nonlinear system modeling. Also, automotive applications imply non-electronic component modeling such as mechanic and fluidic [8] [10] [11] [12]. Current SystemC-AMS release does not support all requirements. However, it supports modeling in time and frequency domains. Also linear systems can be described behaviorally via transfer functions or differential equation sets. Besides, linear networks can be constructed using linear elements such as resistors and capacitors etc. SystemC-AMS is planned to cover the nonlinear description capabilities described above [5]. Also, its non-electronic modeling capabilities will be extended with mechanical elements library that will allow to model sensors and actuators explicitly [22].

Although SystemC-AMS is rather new concept and it does not cover all design areas, most of the continuous time systems can be designed and simulated with it. There are

already reported case studies of modeled hybrid systems in SystemC-AMS [13] [14] [15].

The aim of the thesis is to study the digital and analog SystemC descriptions through the design of an 8-bit microcontroller. In this study, an extended PIC 16F871 microcontroller and its peripheral modules are implemented with SystemC using register-transfer-level (RTL) and behavioral-level description. Its core and peripherals are very well known member of the PIC micro midrange microcontroller family. With its 8-bit main core, special function registers, several peripheral units and analog inputs, it is a very good model to study system on chip concept. The analog to digital converter (A/D) of the microcontroller is modeled behaviorally using SystemC-AMS. In order to extend the study, a digital to analog converter (D/A) which, normally the PIC 16F871 does not involve, is added and also modeled in the same way.

Currently there isn't any EDA tool that directly synthesizes SystemC codes to hardware [16]. In order to employ the SystemC codes for hardware synthesizing, it should be converted to traditional hardware description languages (HDL) using some programs such as SystemCrafter [17], CoCentric SystemC compiler [18] or Prosilog [19]. This hardware synthesis flow of SystemC is studied with synthesis of arithmetic logic unit of PIC 16F871.

In this thesis, SystemC 2.0.1 release is used with Microsoft Visual 6.0 C++ compiler during the design, debugging and simulation stages. Results of the simulations are received in .vcd format and *SynaptiCad WaveViewer 10.20b* tool is utilized to observe the digital waveforms. Assembly codes are compiled using *MPLAB 7.40* tool and necessary .hex files for the simulations are generated. Behavioral design and simulations of analog modules have been done using SystemC-AMS 0.15 RC1 and SystemC 2.1.1 libraries. However, current SystemC-AMS version can only be compiled with *gnu gcc* compiler on linux environment. For this purpose, *cygwin* which is linux emulator for windows is used as console. Also, analog waveforms are traced using *qplot 1.2* or *octave workshop 1.0* which is known as the MATLAB clone. Besides, trial version of *SystemCrafter 2.0* is utilized for SystemC to VHDL synthesis of the arithmetic logic unit

of the PIC 16F871. In order to perform FPGA synthesis of the generated code, *Xilinx ISE 8.1i* is employed.

This thesis consists of six chapters. Chapter 1 is devoted to introductory study. Chapter 2 describes the specifications of the SystemC-AMS language in detail. In order to clarify the concept of the analog and mixed signal modeling, some examples are provided. At the end of the chapter, a complete analog behavioral design example is presented with the simulation results.

In Chapter 3 fundamental features of the main core of the PIC 16F871 microcontroller is given without involving peripheral modules. This chapter aims to introduce the designed digital core. In Chapter 4, SystemC implementation of the extended 16F871 microcontroller is covered including microcontroller structure, register set, and information about the peripherals that accompany their implementation details.

Chapter 5 is allocated for simulations and verification of the design. Test-bench environment, instruction set verification, addressing mode simulations and verification of each peripheral module are explained separately. This chapter also covers the simulations of the CRC test code. At the end of the chapter, synchronous demodulator example that utilizes the designed microcontroller is given as a hybrid system modeling application of SystemC-AMS.

Chapter 6 concludes the thesis. An overview of the thesis and important issues encountered during the study is stated in this chapter.

Appendix A covers the details of the instruction set summary. Assembly test codes of the main core are given in Appendix B. In Appendix C, embedded functions and linear network elements of the SystemC-AMS are described in detail. Special function registers of the main core which is very important to understand the features of the microcontroller are given in Appendix D. FPGA synthesis of the arithmetic logic unit is described briefly and synthesis results are given in Appendix E. Finally, CRC test code and assembly code of the synchronous demodulator example is provided in Appendix F.

CHAPTER 2

SYSTEMC AS AN INTEGRATED ANALOG AND DIGITAL DESIGN ENVIRONMENT

2.1 Analog And Mixed Signal Extensions To SystemC

SystemC supports very high level design and simulation of digital systems. However, today's electronic system designs, especially multimedia and communication systems contain significant analog components. On the other hand, analog system design environments are separated from digital design tools and vice versa. Analog and mixed signal extensions (AMS) aim to close this gap for SystemC. Design environment that includes AMS libraries in the design process for SystemC is named as SystamC-AMS.

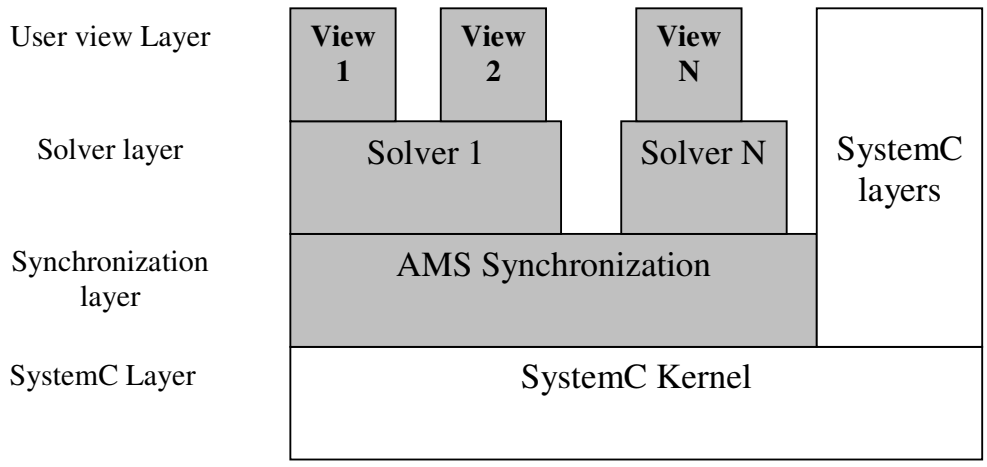


Figure 2- 1: Layered approach of SystemC-AMS

The AMS extensions to SystemC are being defined using the layered approach as given in Figure 2.1. They are built on SystemC kernel which constitutes the base layer. On top of the base layer there are two sets of layers. One of them covers the existing SystemC layers and other one is for the new set of layers related with AMS extensions. The user view layer provides different descriptive methods to write executable continuous-time models such as transfer functions and state space formulation. The solver layer provides different implementations of solvers that are required to simulate specific AMS descriptions. The synchronization layer implements a mechanism to organize the simulation of the SystemC-AMS model that may include different continuous time views and discrete-event parts [7].

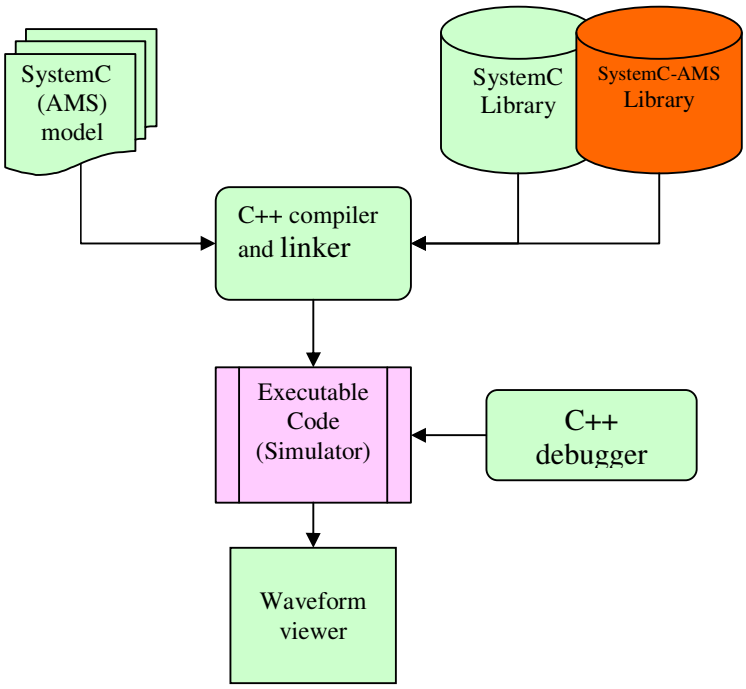


Figure 2- 2: SystemC and SystemC-AMS design flow

2.2 Synchronous Data Flow (SDF) Module

An SDF module is a basic building block that has continuous time behavior in SystemC-AMS. An SDF module is a container class that holds the member of the module which may not instantiate other modules [21]. SystemC-AMS uses the Synchronous Data Flow (SDF) formalism to model the continuous time systems. Processes which are described with SDF formalism communicate with each other using unidirectional FIFO channels. They receive tokens, process it and send it to output channel. In SystemC-AMS environment, tokens correspond to sampled data and processes correspond to modules. For this reason, analog modules are named as SDF modules. Typical SDF modules contain:

- Port declarations. Decelerated ports maybe either SDF ports which provide communication between other SDF modules or converter type ports provide communication between SystemC modules (Discrete Events).

- Data member declarations of the module.

- SDF member functions or other functions.

SDF modules do not express a real analog circuit and they are not synthesizable models. They can be used to describe an idealized system. For this reason data members can be manipulated in anyway. In other words, all data transfers are immediate assignments. Moreover, any data member can be re-assigned several times during process.

Hierarchical SDF modules follow the standard SystemC hierarchical channel instantiation procedure. This standard procedure is as followed,

1. Create component instance declarations at the top module.
2. Allocate these instantiated modules by giving them different name.
3. Bind ports of instantiated modules via sdf signals.

Primitive sdf modules can be connected to each other in order to construct hierarchical modules through `sca_sdf_signal`, which is quite similar to `sc_signal` of SystemC.

Behavior of the module is described using embedded sdf member functions. In the following part SDF functions are described.

2.2.1 Synchronous Data Flow (SDF) Module Member Functions

Unlike SystemC modules, SDF modules do not have sensitivity list or processes like SC_METHOD. Behavior of SDF module is described using SDF member functions.

Port attributes like sampling rate, sampling period and delay are declared in `attributes()` member function. Details about this function and its embedded functions will be explained in Section 2.2.2. This function is executed in elaboration time. In other words, attributes like sampling rate can not be changed during simulation. Initial values for input and output ports are announced in an optional `init()` member function. This function is executed just before the simulation. Continuous time behavior of the SDF module is described in `sig_proc()` function. It can be considered as the heart of the module. This function is executed each time module needs to be evaluated [21]. For frequency domain models, `ac_sig_proc()` member function is used. Details about this function will be given in Section 2.4. `Post_proc()` is another optional member function that may be used for post processing tasks such as FFT (Fast Fourier Transformation). This function is executed just after the simulation, but it should be called with `sca_terminate()` function in the main file.

2.2.2 Synchronous Data Flow (SDF) Module Ports

Each module has ports for communication with other modules or reception of data from the surrounding. Modules may have any number of ports. In contrast to SystemC modules, SDF modules may have only unidirectional SDF ports. To read data from another SDF module *sca_sdf_in* port is used while *sca_sdf_out* port is used for delivering data. Addition to these ports, SDF modules have particular converter ports to

interact with SystemC modules. For discrete event inputs *sca_scsdf_in* ports, for discrete event outputs *sca_scsdf_out* ports are utilized [17].

```
SCA_SDF_MODULE(module_name)
{
//Port declarations

sca_sdf_in<double> in;
sca_sdf_out<double> out;

void attributes( ) {
// Sampling period, rate declarations
}

void init( ) {
// initial values of output ports
}

void sig_proc( ) {
// Continuous time domain behavior
}

void post_proc( ) {
//Post processing tasks
}

SCA_CTOR(module_name){ }
```

Figure 2- 3: General form of SDF module [9]

In contrast to SystemC, some properties of SDF modules such as sampling rate depend on the ports. SDF ports' attributes are controlled in `attributes()` member function and their initial values are assigned in `init()` member function.

Sampling periods of ports are set using `port.set_T()` function. Here, `port` denotes the name of the port. In brackets period duration should be given in Systemc time domain. SDF module must have at least one port with this attribute. Ports that are connected with *sca_signal* must have sampling rates commensurate with each other.

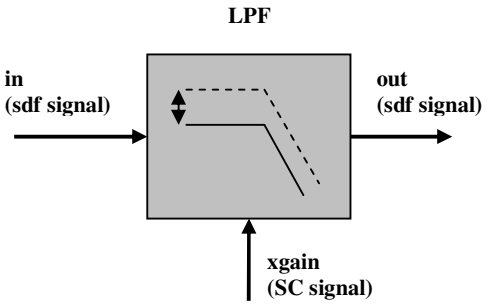
Otherwise an error will arise. Number of the samples to be read or written per evaluation duration is set using `port.set_rate()` function. An integer number should be placed in the brackets to express sampling rate. Default value is 1 for this function. Normally, modules begin to sample at time "0". This can be modified using `port.set_t0()` function which sets absolute time of first sample to be processed at the port[21]. Time value for first sample should be written in brackets in SystemC time domain. Another attribute for a port is waiting for a definite number of samples before reading or writing to port again. For this purpose, `port.set_delay()` function should be used. Default value for this function is zero. Otherwise, it should be revealed with an integer value between the brackets.

There are also some functions that returns the port attribute values that can be used in main code. For example, `port.get_T()` function returns the sampling period of that port. Similarly, `port.get_t0()` and `port.get_delay()` functions return the absolute time of the first sample and delay respectively. For multi-rate ports `port.get_time()` function is used instead of `port.get_t0()` function. Finally, `Port.get_sample_cnt()` function returns number of the samples has processed by port since beginning of the simulation.

A module can read from or write to a port by simply using `read()` or `write()` functions. But for multi-rate functions (sampling rate greater than 1) read and write functions can be used for reading or writing nth sample, where n is an integer which should be smaller or equal to sampling rate of that port. `Port.read(n)` instruction returns nth sample of the current module evaluation. It can also be written as port[n]. In order to write nth sample to the output `port.write(val,n)` or `Port[n]=var` instructions should be used, where `val` denotes the data that will be written to output.

2.3 Analog Linear Behavioral Models

Linear analog systems can be described in SystemC-AMS with their; (i) Laplace transfer functions in polynomial form, (ii) transfer functions in zero-pole representations or (iii) state-space equations [21]. Z domain modeling may also be used as explained in section 2.4.



$$H(s) = \frac{1}{1 + \frac{1}{2\pi f_c} s}$$

(a)

```
//--------------------------------
//  Low-pass filter with gain controller
//--------------------------------

SCA_SDF_MODULE(prefi_ac){
sca_sdf_in<double> in;
sca_sdf_out<double> out;
sca_scsdf_in<bool> xgain;

// parameter

double prefi_fc;
double prefi_gain0;
double prefi_gain1;

// states

sca_ltf_nd ltf_1;
sca_vector<double> A, B;


void init() {
B(0) = 1.0;
A(0) = 1.0;
A(1) = 1.0/(2.0*M_PI*prefi_fc); }
```

```
void sig_proc() {

double tmp = ltf_1(B,A,in.read());

if (xgain.read())
{out.write(tmp*prefi_gain1);}
else
{ out.write(tmp * prefi_gain0);}

}

SCA_CTOR(prefi_ac) {

prefi_fc = 1.0e3;
prefi_gain0 = 2.5;
prefi_gain1 = 2.5 * 4;

}
};
```

(b)

Figure 2- 4: (a) Schematic description and transfer function of Low-Pass filter (b) SystemC-AMS model of Low-Pass filter [9]

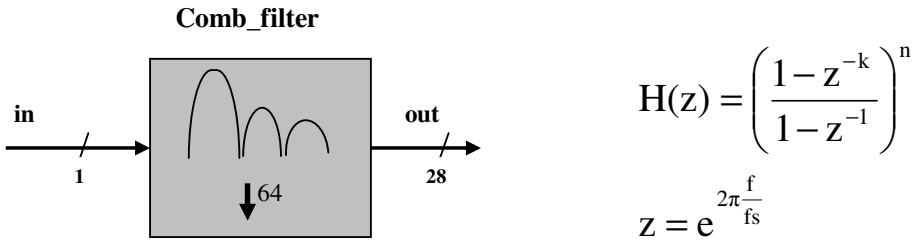
Each method has particular data class. These data classes contain the parameter variables for coefficients of the functions or equations which may be numerator or denominator coefficients. Each method should be defined as an object of class. In order to receive system output, class should be loaded with parameter values. For transfer functions, these parameters are defined with `sca_vector` while state-space description parameters are defined with `sca_matrix` variables. Besides, parameters of the models can be configured during simulation where needed. Returned output response can also be manipulated as an ordinary variable. Formalisms of these methods are given in APPENDIX C.

In Figure 2.4, behavioral description of a low-pass filter in SystemC-AMS is given.

2.4 Frequency Domain Specifications

Frequency domain specifications provide to model the continuous time system with set of equations in frequency domain. During frequency domain analysis, this equation system is solved for given frequencies [21]. As mentioned previously, this specifications must be described in optional `ac_sig_proc()` member function. This member function and its members are quite similar with `sig_proc()` function. However, functions used in `ac_sig_proc()` member functions return complex values. Moreover, each member function has additional prefix "ac". General method is describing a model in time domain in `sig_proc()` function and describing in frequency domain in `ac_sig_proc()` function so that both time and frequency domain responses of same system could be viewed. Each function is simulated separately and their results should be saved in different files. Frequency domain simulations can be performed for a single frequency point or for a specified frequency range. Simulation will return the real and imaginary components of the signal for corresponding frequencies.

Members of `ac_sig_proc()` function and their specifications are given in APPENDIX C. In Figure 2.5 frequency domain description of a comb filter in SystemC-AMS is presented.



(a)

```
//-------------------------------------
//  Frequency domain desctiption of comb
//   filter
//-------------------------------------

SCA_SDF_MODULE(ac_tx_comb)
{
sca_sdf_in<bool> in;
sca_sdf_out<sc_int<28> > out;

void attributes() {
in.set_rate(64);
out.set_rate(1);
}
void ac_sig_proc()  {
sca_complex z;
z = sca_ac_z(in.get_T().in_seconds() , 1 );
double k = 64.0; //decimation factor
double n = 3.0; //order of comb filter

// complex transfer function:

sca_complex h;
h = pow((1.0-pow(z,-k)) / (1.0-1.0/z), n);
sca_ac(out) = h * sca_ac(in) ;  }

// Rest of the code ….
```

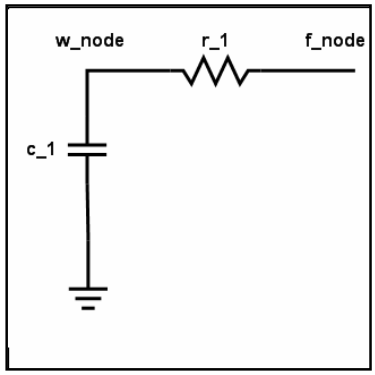
(b)

Figure 2- 5: (a) Schematic description and transfer function of digital comb filter (b) SystemC-AMS model of digital comb filter [9]

2.5 Linear Electrical Networks

In order to describe analog systems, SystemC-AMS also contains linear elements library. These elements are most common elements used on analog systems such as resistors, capacitors and inductors. Exact list of linear elements are listed in APPENDIX C. It is reported that next versions of SystemC-AMS will also have mechanical elements, such as mass, spring and dumper, for MEMS applications and modeling [9].

```
//------------------------------
//   RC Network
//------------------------------

// Rest of the code….

sca_elec_node w_node;  //electrical node
sca_elec_node f_node;    //electrical node
sca_elec_ref gnd;          //reference node

sca_r r_it("r_it");

r_it.value = 2e3;
r_it.p(w_node);
r_it.n(f_node);

sca_c c_it("c_it");

c_it.value=100e-9;
c_it.p(w_node);
c_it.n(gnd);

// Rest of the code …
```



(a) **(b)**

Figure 2- 6: (a) SystemC-AMS model of a linear network (b) Schema of the network [9]

Electrical elements are connected to each other as in SPICE. There are reference nodes that every element should be connected. Also, there can't be any dangling port for linear elements. Linear network library also includes converter elements in order to connect these elements to SDF modules. In this way, linear elements can be controlled by pure

digital blocks via SDF modules, while some elements can be directly controlled by SystemC modules. Linear elements can be described in a single module. In this case, they have particular ports that linear elements terminate named as `sca_elec_port`. Linear elements can also be tied to other modules in main file as if independent modules. In Figure 2.6, SystemC-AMS description of a simple RC network is given.

2.6 A Design Example

In order to demonstrate the design of an analog system with SystemC-AMS, a simple example will be given. A low-pass filter (LPF) has been designed and simulated in SystemC-AMS environment. This may be the simplest model that can clarify the linear modeling approach. Described LPF has both analog and digital ports. It will be simulated using *gnu gcc* compiler on Linux environment. Schematic description the testbench model is given in Figure 2.7.

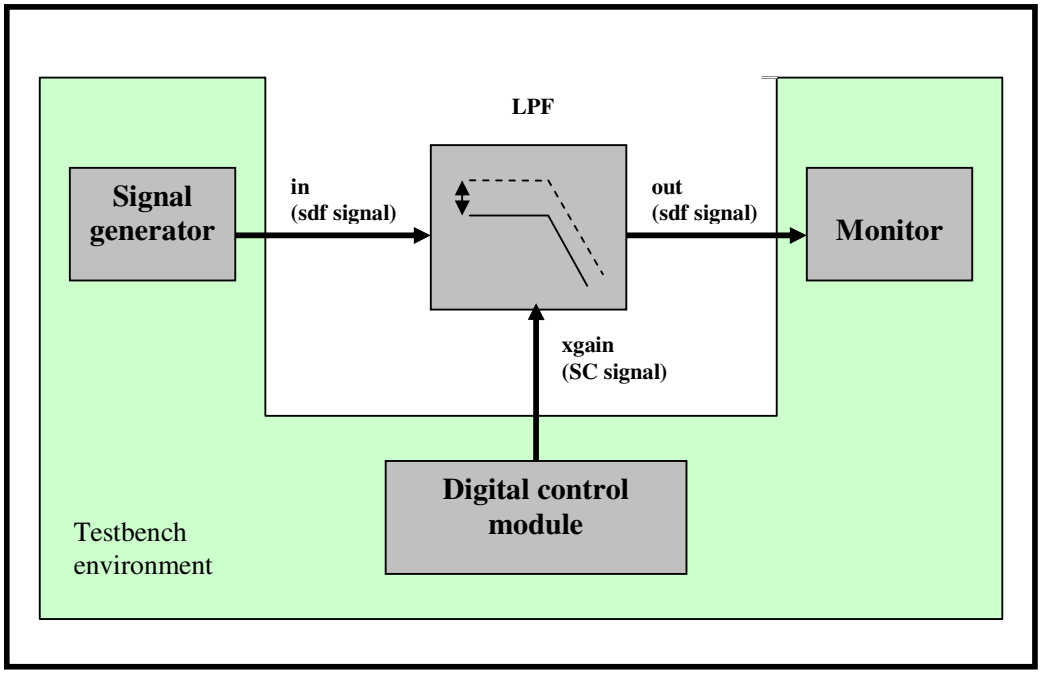


Figure 2- 7: Test-bench of the low-pass filter

LPF module has two inputs and one output port. Analog signal is applied to “in” port and filtered signal is sent to *out* port. The signal *xgain*, which is connected to a digital module, controls the amplification factor. This filter is described via laplace transfer function of a low-pass filter whose cut-off frequency is 10 kHz.

```
//---------------------------------------
//  Low-pass filter
//---------------------------------------

#include "systemc-ams.h"

SCA_SDF_MODULE(lp1)  {
sca_sdf_in<double> in;
sca_sdf_out<double> out;
sca_scsdf_in<bool> xgain;

// parameter

double prefi_fc;
double prefi_gain0;
double prefi_gain1;

sca_ltf_nd ltf_1;
sca_vector<double> A, B;

void init() {
B(0) = 1.0;
A(0) = 1.0;
A(1) = 1.0/(2.0*M_PI*prefi_fc);  }
```

```
void sig_proc() {

double tmp = ltf_1(B,A,in.read());
if (xgain.read())
{out.write(tmp*prefi_gain1);}
else
{ out.write(tmp * prefi_gain0);}

}
SCA_CTOR(lp1) {

prefi_fc = 1.0e4;
prefi_gain0 = 2.;
prefi_gain1 = 1;

}

}; // End of module
```

Figure 2- 8: SystemC-AMS model of the low-pass filter [9]

```
//-----------------------------------------------
// Sinus signal source  (stimulus)
//-----------------------------------------------


#include "systemc-ams.h"

SCA_SDF_MODULE(src_sin) {
  sca_sdf_out<double>  out;

  double ampl, freq;

  void sig_proc() {

out.write(ampl*sin(2*M_PI*freq*sc_time_st
amp().to_seconds()));

 }

  SCA_CTOR(src_sin) {}

};  // End of module
```

(a)

```
//---------------------------------------
// Digital gain controller
//---------------------------------------

#include "systemc-ams.h"

SC_MODULE(control) {

  sc_in<bool> CLK;
  sc_out<bool> xgain_o;



  SC_CTOR(control) {
  SC_METHOD(entry)
    sensitive<<CLK.pos();
  }

  void entry()
{
            xgain_o=true;
  }

}; // End of module
```

(b)

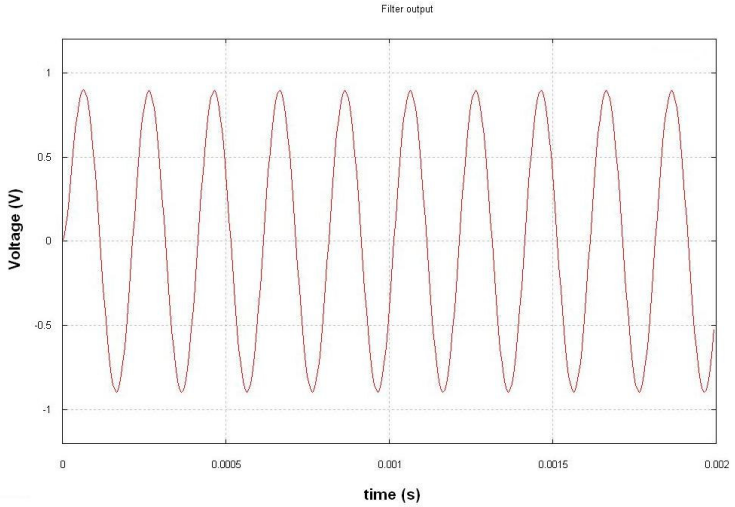
Figure 2- 9: SystemC-AMS model of (a) sinus signal source (b) digital gain controller module [9]

```
//---------------------------------------
//  Main file
//---------------------------------------

#include "systemc-ams.h"
#include "lp1.cpp"
#include "src_sin.cpp"
#include "control.cpp"


int sc_main(int argc, char* argv[])
{
  sca_sdf_signal<double>  src, lpo;
  sc_signal<bool> gain;
  sc_clock clock;

  lp1 i_lp1("uut_lp1");
    i_lp1.in(src);

i_lp1.in.set_T(sc_time(0.005,SC_MS));
    i_lp1.out(lpo);
    i_lp1.xgain(gain);

control i_control ("control");
    i_control.xgain_o(gain);
    i_control.CLK(clock);
```

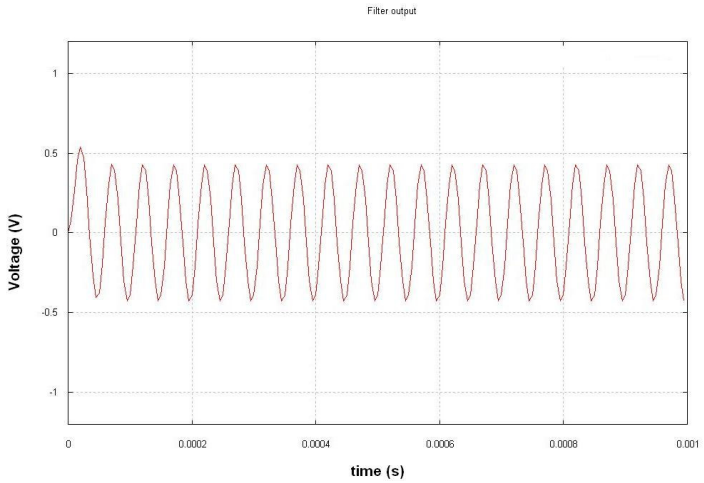
```
src_sin i_src("src");
    i_src.out(src);

i_src.out.set_T(sc_time(0.005,SC_MS));
    i_src.ampl = 1.0;
    i_src.freq = 1e4;

 trace tr_src("tr_src");
   tr_src.in(src);

  trace tr_lpo("tr_lpo");
    tr_lpo.in(lpo);


 sc_start(2.0, SC_MS);

  return 0;

}  // End of main file
```

Figure 2- 10: Main source file of the LPF [9]

Signal generator provides a stable sinus signal which oscillates between -1 and 1 volt in order to test the filter. This module can be considered as an analog stimulus. For this simulation 5 kHz and 20 kHz sinus signals are generated to observe the response of filter to different frequencies.



(a)



(b)

Figure 2- 11: Output waveform at (a) 5 kHz (b) 20 kHz

The digital control module configures the gain of the output by means of switching the port `xgain` . When this port is low, unity gain will be observed. If it becomes high, magnitude of the output signal will be amplified by two. This module can be considered as a digital stimulus.

Monitor module collects the output of the low-pass filter and directly writes to a file. SystemC-AMS does not have an integrated analog waveform viewer. Saved file should be read by another program like Matlab. In this simulation, octave, which is known as Matlab clone, is used to trace the analog waveform of the output. In Figure 2.11 simulation results for different input frequencies are given. Figure 2.12 depicts the effect of the digital gain controller.

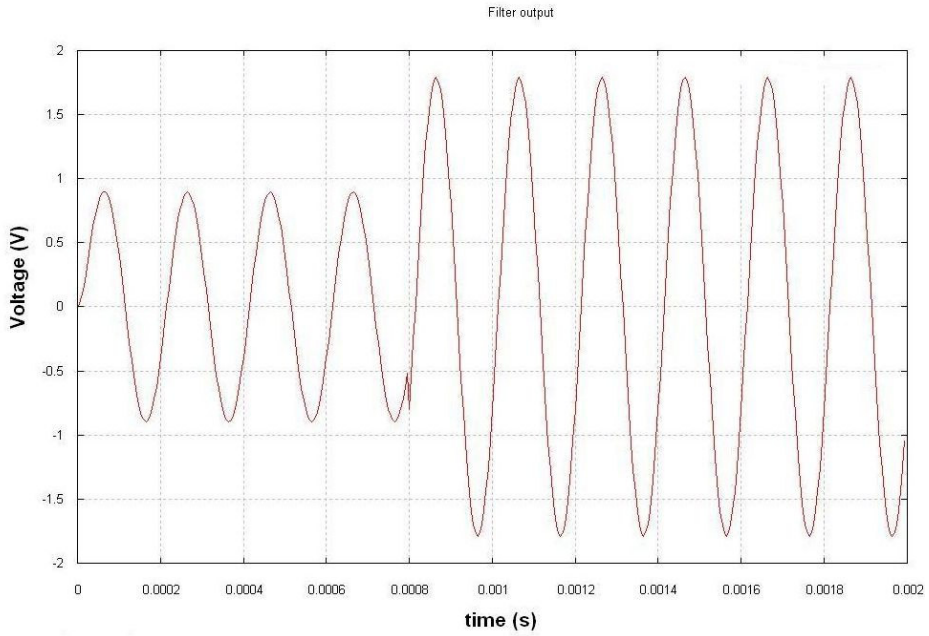


Figure 2- 12: Effect of the digital gain controller.

CHAPTER 3

MAIN PROPERTIES OF THE PIC 16F871 CORE

Microcontrollers are today's one of the most important digital structures. Since the first digital microprocessors, there is still a need for microcontrollers. Fundamental difference between microcontrollers and microprocessors is their architecture. Microcontrollers are simpler processors that have sub-components integrated on same chip. Besides, main task of the microcontrollers is supervising the peripheral elements during their operations. Currently, microcontrollers have several digital peripheral modules that increase their capabilities. In addition to digital elements, recent microcontrollers have analog peripherals in order to improve their interaction with analog systems.

PIC midrange family is the one of the most common microcontroller group in use. There are several different members of family that are specified for the user's needs. Main core is common in different models of the midrange microcontrollers while peripheral modules changes from model to model.

The PIC 16F871 microcontroller is the one of the most known member of the mid range family. It has 35 instructions set which is very easy to learn. Fundamental operation instructions are executed in four oscillator cycle while branching instructions need eight oscillator cycles or two instruction cycles. The 16F871 has on-chip 2 kilobyte flash program memory, Data-RAM and Data-EEPROM memories. Data RAM has four banks and plenty of register set. Architecture of the PIC 16F871 microcontroller is given in Figure 3.1.

Main reason for selecting the 16F871 microcontroller as a case study is its architecture. Its main core is very well known and it houses several peripherals. Despite to its reduced instruction set, it can perform several operations which make it one of the unique

controllers on market. Secondly, although there are many models for main core of the mid-range family in different hardware description languages, there is no particular PIC 16F871 model in SystemC language.

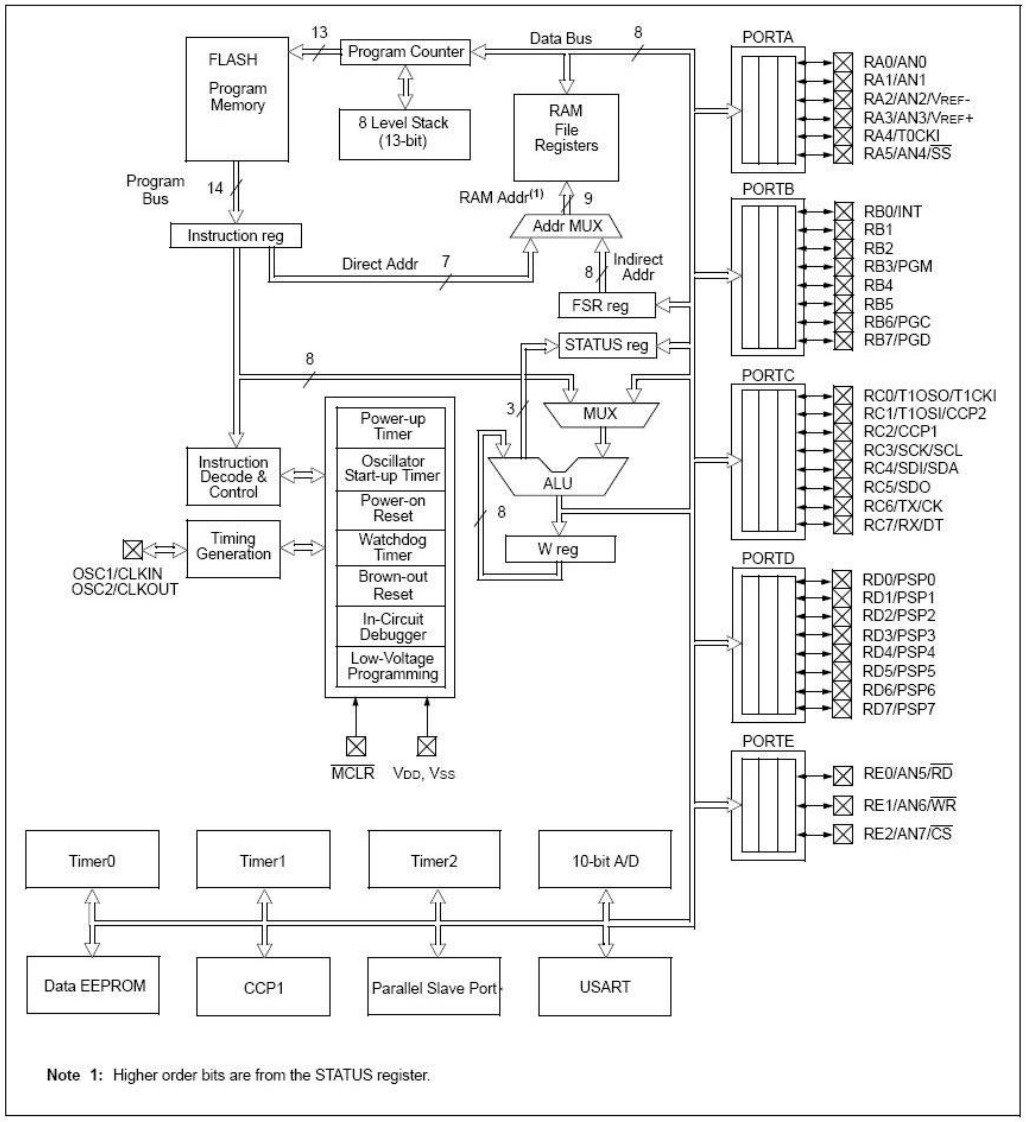


Figure 3- 1: Block digram of PIC 16F871 [23]

A SystemC model of a mid-range family member may also helpful for undergraduate project applications of PIC microcontroller. Considering the development of the SystemC-AMS, the16F871 model in SystemC may also be a platform for AMS applications. Modeled analog modules can easily be connected directly to main core or through A/D converter. An appropriate assembly code would be enough to simulate the whole system described with SystemC and SystemC-AMS. In this chapter, main features of the PIC 16F871 microcontroller core will be given.

3.2 Type of Memories

The 16F871 microcontroller has three different memory resources. They are listed and explained below

- On-chip Program memory; It is a Flash memory in this device. It can be ROM or EPROM in different PIC micro family.
- On-chip Data Memory (RAM).
- On-chip EEPROM data storage memory.

Main core does not directly support external data or program memory. External memory usages must be supported with software.

3.1.1 Program Memory Organization

The program memory stores the programs that main core is to be execute. The PIC mid-range devices have a 13 bit counter capable of addressing 8 Kbytes x 14 program memory spaces. The 16F871 has 2 Kbytes x 14 on-chip flash program memory [23]. Devices have higher memory capacities than 2 Kbytes need memory paging. Memory paging implies that 8 Kbytes memory is divided into 4 equal pages. In order to start a code pack in other pages, `ORG` directive must be employed that points the starting address of program code. Program memory paging will be explained in later sections.

Main core begins to read program memory from 0x0000 location which is also Reset vector address. In other words, when reset occurs program counter is cleared and it

jumps to main code starting address location. In Program memory, 0x0004 address is allocated for interrupt vector. When an interrupt occurs program counter is loaded with 0x0004, then it begins to execute interrupt vector code.

3.1.2 Data Memory Organization

The data memory is partitioned into multiple banks which contain General purpose registers (GPR) and Special function registers (SFR) [23]. The 16F871 microcontroller has 4 banks and each bank extends up to 0x7F (128 bytes) and 4 banks offers 0x1FF (512 bytes) memory area. However, all data locations are not implemented. Lower portion of the memory banks are reserved for special function registers while upper locations are reserved for general purpose registers.

```
BSF STATUS, RP0
BCF STATUS, RP1   ;//  Bank 1
```

Table 3- 1: Bank selection using RP0 and RP0 bits

RP<1:0>	BANK
00	BANK 0
01	BANK 1
10	BANK2
11	BANK3

Bank selection is achieved via setting or clearing bank selection bits, RP1 and RP0, which are 6^{th} and 5^{th} bits of `STATUS` register respectively. Instruction given above selects Bank 1 by means of setting RP0 bit and clearing RP1 bits. Bank selection configurations are given in Table 3.1. In order to manipulate correct registers on memory bank, bank selection should be carried out done properly. However, some configuration

registers, such as STATUS register have reserved locations in each bank for convenience.

The Working Register (w)

The working register (w) is the only and unique accumulator of the 16F871 microcontroller that holds 8 bit data. Most of command directly involves the working register into operation since literal values can only be loaded to it. After that, literal value can be transferred to target special function registers. Data transfers between registers and the working register are bi-directional. However, a special function register cannot be assigned to another without assistance of the working register. The working register is also an invariable source for all arithmetic operations, and second target option for arithmetic results.

Bit Addresses

The PIC 16F871 allows user to reach and manipulate any register's any bit, if it is writable. BSF and BCF commands are particular instructions that set or clear declared register' target bit respectively. For this purpose, 3 bits are allocated in instruction word of BSF and BCF in order to address bit locations.

```
BSF   w, 0
BSF   w, 1
BCF   w, 2
BCF   w, 3
```

First two lines of code that is given above sets 0^{th} and 1^{st} bits of working register (w). Following 2 lines clears 2^{nd} and 3^{rd} bits of w register. This property is important since peripherals are started, stopped or configured using dedicated bit of the appropriate control register.

The Program Counter

The program counter of the PIC 16F871 is a 13 bit counter that is incremented regularly after every 4th clock pulse. Initial value for program counter is 0x0000 and it is incremented by one after execution of each instruction. Subroutine calls or unconditional `GOTO` commands load program counter with starting address of sub-code. In case of interrupts, program counter is loaded with 0x0004 hexadecimal value, which is the start address of the interrupt service routine (ISR).

Lower byte of the program counter, PCL register, is a readable and writable register. However, higher 5 bits of program counter is neither directly readable nor writable. Higher 5 bits come from the `PCLATH` register. Contents of the `PCLATH` register is transferred to upper byte of program counter when the PC is loaded with new value [23]. In section 3.4, different situations for program counter manipulations will be explained in detail.

The Stack

The PIC 16F871 core has 13 bit 8 level stack. The stack is controlled by hardware in case of either subroutine calls or interrupts. There are no commands such as `PUSH` or `POP`. Besides, stack pointer cannot be controlled by software. When a subroutine is called using `CALL` command, present Program Counter value is transferred to stack. At the end of the subroutine `RETURN` or `RETLW` commands dictates main core to return where it was called from.

Similarly, when interrupt occurs program counter value is pushed to stack. At the end of the interrupt service routine, the `RETFIE` command pops the address properly so that microcontroller can continue the normal process from where it was interrupted.

The stack of the 16F871 has a circular structure. That is, 8 consecutive subroutine or interrupt operations can be evaluated properly. After that, stack pointer will overflow and become zero again which will lead to inconvenient processes.

3.2 Special Function Registers

Since the main core of the PIC 16F871 has RISC architecture, it has very large set of registers. RISC architecture reduces instruction set, but increase in register set is a compromise for this configuration. These registers control almost every property of the device, even interrupts. Before main part of the code, microcontroller should be configured by means of adjusting special function registers (SFR) in order to make it operate in an appropriate way. SFRs can be re-assigned during program execution. In this way, device can be re-configured in a completely different manner during any process.

3.3 Addressing Modes Of The 16F871

The 16F871 offers two main methods to reach the memory locations and their values which is called addressing modes. These addressing modes are,

- Direct Addressing
- Indirect addressing

3.3.1 Direct Addressing

In direct addressing mode, name of special function registers come after specific instructions. This provides only 7 bit address information that comes from an op-code. However, 7 bit only allows addressing up to 0x7F. In order to address further memory locations, 2 more bits are needed. These bits come from Bank Selection bits which are stored in `STATUS` register's 5^{th} and 6^{th} bits. Before modifying value of any register, Bank selection should be performed properly. Otherwise, another register will be evaluated as the operand. Special Function Registers can only be loaded with content of the working register since they cannot be immediate addressed as described in previous section.

First example given below describes an assignment. Content of the working register will be transferred to T2CON register. When this instruction compiled and linked, assembly compiler will automatically generate lower 7 bit of address of the T2CON register. For this reason, first and second expressions are completely identical. Similarly, when 3rd instruction is compiled, assembly compiler will generate lower 7 bit of address of the PR2 register which is same with the T2CON. Switching to Bank 1 will yield to proper operation. Otherwise 3 instructions will have same effect.

```
MOVWF T2CON
MOVWF 0x12
MOVWF PR2
```

3.3.2 Indirect Addressing

Indirect addressing offers easier and more flexible way to reach a memory range in the RAM. There are two specific registers for the indirect addressing. First one is the file select register (FSR), which holds the lower 8 bit of address of any memory location. Other one is INDF register which is not a physical register. Any instruction followed by INDF will lead to an access to the location that FSR points to. STATUS register's 7th bit holds the bank selection bit for indirect addressing mode.

Indirect addressing allows reaching any point in RAM. It is not limited with General purpose registers. Though, it is mostly used for GPR registers for wide memory range accesses. Assembly script below describes how to clear the RAM locations between 0x20-0x2F using indirect addressing [23]. Please note that, same purpose can be achieved with very long code with direct addressing.

```
        MOVLW 0X20    ; Starting point
        MOVWF FSR
NEXT:
        CLRF INDF     ; Clear indirect addressed location
        INCF FSR,F
        BTFSS FSR,4   ; Equal to 0x2F ?
        GOTO NEXT     ; No
CONTINUE:             ; Yes
```

3.4 Program Flow Control Operations

As mentioned in Section 3.1, the program counter starts with 0x0000 value and increments sequentially unless program counter is altered. There are particular instructions that change program flow. These instructions may lead to unconditional jumping, branching to subroutines, conditional skipping the next instruction and freeze in program counter. Additionally, interrupts, if enabled, changes the program counter and jumps to dedicated address in order to perform pre-defined tasks. If the program flow alters, the program counter is loaded with new program memory address and main core continues to execute from this new location. In next sections details about program flow control operations will be given.

3.4.1 Unconditional Branching

The unconditional jump is one of the most important operations for a microprocessor. It is generally needed several times to fetch new instruction address during execution which is referred as unconditional jumping. This operation has higher importance for the PIC mid-range devices since each conditional skipping instruction is followed by unconditional jumps. It is carried out using the `GOTO` instruction. This instruction can be used in two different forms with same effect. A label name may point instruction location as given in first line. In this case program execution continues from that location. Second line describes the other method that tells main core to go back to "n" lines above, where n is an integer. In order to jump "n" line below, with "-" sign must be replaced with "+" sign. Forever loop will be generated If "n" is zero.

```
GOTO   Label_name
GOTO   $-n
```

`The GOTO` instruction is performed by loading the program counter with 11 bit address information comes from op-code, and 2 bits come from the `PCLATH` register. 3 bits of the op-code provide recognition of the `GOTO` instruction; other 11 bits point an address location. The `GOTO` instruction also supports long jumps. Long jumps refer to branching operations to other program memory pages. In order to jump to other pages, upper 2 bits

of the PCLATH register should be configured for proper page selection, before the GOTO instruction.

```
        ORG 0x500      ; Page 0 (0x000 – 0x7FF )
        ....
        BCF PCLATH,4
        BSF PCLATH,3   ; Select Page 1
        GOTO label_1   ; Jumpt to Label_1 at page 1
        ....

        ORG 0x900      ; Page 1 (0x800 – 0xFFF)
        ....
label_1:
        ....
```

Instructions given above perform a GOTO operation to a location in different program memory page [23].

3.4.2 Direct Calls

Direct calls are another important tasks that main core must perform. The CALL instruction is used to achieve this purpose. In contrast to unconditional jumps, call operations are needed to execute short subroutines. After execution of the subroutine, program flow returns to the main program.

The CALL instruction is accomplished in similar way with the GOTO instruction. However, before loading the program counter with new program memory address, its present value is transferred to stack not to loose it. Afterwards, the program counter is loaded with 11 bit address information of subroutine code. Upper two bits are for the page selection. Page selection is performed literally explained in previous section. After execution of a subroutine, program counter is loaded with former address saved in stack. Subroutines must end with either RETURN or RETLW instructions. Only difference between these two instructions is RETLW loads working register with a literal value

before leaving the subroutine. RETLW command also provides computed GOTO operation which will be described in next section.

3.4.3 Computed GOTO

The main core has also unique property which is names as computed goto. This goto operation is achieved by adding an offset value to program counter [23]. Computed goto operation is needed for table reading or data decoding. The table consists of several RETLW instructions. The first instruction in the table computes the target line by means of adding offset to the PCL register and consequently, the program branches to the appropriate RETLW instruction line [25]. This is also known as relative addressing.

```
....
        MOVLW  offset_value
        CALL   TABLE
        ....
TABLE:
        ADDLW PCL, offset_value
        RETLW    0xAA
        RETLW    0xCC
        RETLW    0xFF
        ....
```

Example given above briefly describes computed goto. If the offset_value is 0, program will branch to first line which will load working register with 0xAA. If the offset_value is 1, program will branch to second line and so on.

3.4.4 Conditional Skipping

The PIC 16F871 device has particular commands called as conditional skipping instructions. These instructions skip a program line if specific conditions are met. The BTFSS instruction tests target bit of a register if it is set then main core skips next program line. Target bit is declared by user. Similarly, the BTFSC instruction skips a line if target bit of register is zero. On the other hand, the INCFSZ and the DECFSZ

instructions increments or decrements any register and after each incrementing or decrementing they check register if it is overflowed. Similarly, if condition is satisfied, main core skips the next program line. As mentioned previously, each conditional skipping operation is followed by a `GOTO` instruction. In this way, the IF…ELSE… structure can be built with combination of a couple of instructions.

3.4.5 Interrupts

Interrupts are special events that changes program flow if enabled. In case of interrupt, main core breaks normal code execution and performs a special task. When an interrupt event arises, the program counter is immediately pushed to stack and loaded with the address of interrupt service routine (ISR). The ISR routine's beginning address is 0x0004. End of ISR is declared by the `RETFIE` instruction. Then, the program counter pops original value from the stack and resumes its operation.

3.4.6 Sleep

The `SLEEP` instruction does not change program flow as same as previous commands. However, the `SLEEP` instruction freezes program flow via locking state machine in one state. This instruction is applied in order to reduce power consumption when microcontroller is idle. System can only saved from this condition with an external interrupt. This implies that, with the `SLEEP` instruction the main core is forced to wait an external event without any operation while consuming little power

CHAPTER 4

DESIGN OF THE EXTENDED PIC 16F871

The PIC 16F871 microcontroller consists of several peripherals including analog and digital modules and main core. Digital components are divided into modules and implemented with SystemC while analog modules are described behaviorally using SystemC-AMS.

The biggest digital module named as Core includes main state machine, arithmetic logic unit (ALU), parallel slave port (PSP), Timer 0 and watch-dog timer (WDT) units. Serial port (USART) is implemented as a separate module. Due to the strong relation among them, Timer 1, Timer 2 and capture, compare and pulse width modulation (PWM) unit (CCP) are implemented as a single module. Analog components such as A/D converter and additional D/A converter modules are also described behaviorally as separate modules.

4.1 The Main State Machine

PIC midrange devices has main state machine that has four instruction execution states namely Q1, Q2, Q3 and Q4 states. Each clock pulse drives main core into next state and four oscillator cycles corresponds one instruction cycle. The instruction fetch and execute is pipelined such that fetch takes one instruction cycle while decode and execute takes another instruction cycle. However, due to the pipelining, each instruction is effectively executed in one instruction cycle [24].

Execution starts with fetching the instruction and saving it in Instruction Register (IR). This instruction is decoded and executed in next oscillator cycles. Data memory is read during Q2 cycle and written during Q4 cycle .

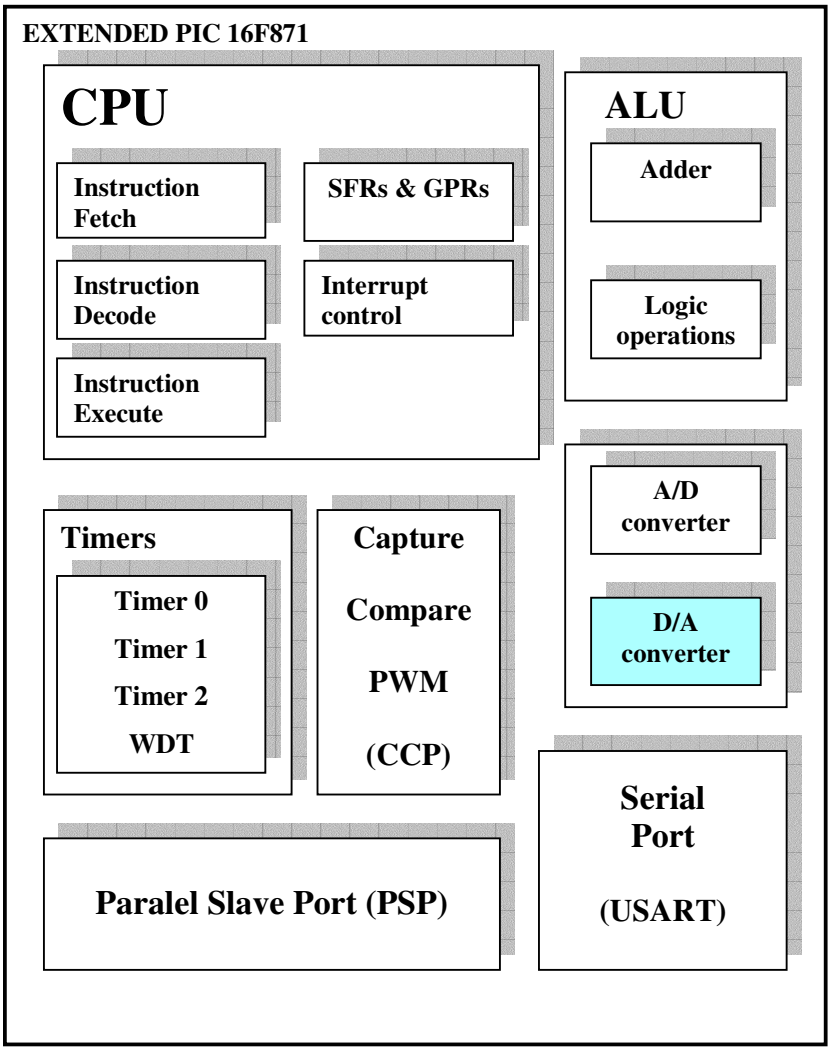


Figure 4- 1: Block diagram of the extended 16F871 microcontroller

Majority of the instructions can be fetched, decoded and executed in one instruction cycle. Only branching and conditional skipping instructions spends two instruction cycles to complete the operation. However, for these commands second cycle is a no-operation cycle. In other words state machine does not perform any operation. At the end of the second instruction cycle, main core resumes to normal operation.

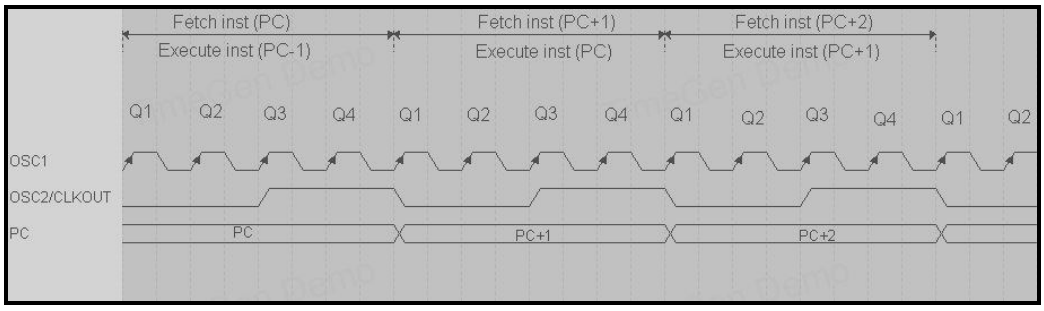


Figure 4- 2: Clock/Instruction cycle of main core

Main core has five different operation modes and each of them has different behavior.

1. **Reset mode:** Reset mode drives the CPU into a known state which is called RESET state. In this state each register is set to preset values. Program Counter is cleared and it begins to increment from the 0x0000 value. Main core returns to normal mode when reset conditions are killed.
2. **Normal mode:** In this mode, state machine fetches, decodes and executes instructions as pictured above. Four states follow each other continuously during normal mode.
3. **No-operation mode:** During this mode main core changes its state from one to another. But it does not perform any operation. This mode is applied for conditional skipping or `NOP` instruction. It is also a stop for every program flow changing operation.
4. **Interrupt start mode:** Normal mode is still valid even an interrupt occurs. However, it spends two dummy cycles before branching to interrupt service routine. These dummy cycles are equivalent with `NOP` instruction except all interrupts are disabled by clearing global interrupt enable flag (GIE) during this mode in order to avoid another interrupt. Interrupt start mode ends when program counter jumps to interrupt vector.
5. **Sleep mode:** This mode starts with when `SLEEP` instruction executed. The 16F871 does not have an idle mode. So, in sleep mode state machine is stuck in Q1 state. Peripheral modules also stop during this mode since they use

internal oscillator unless they can be used with an external clock and external clock is utilized. Only an external interrupt can make state machine switch to normal mode.

All main core components such as state machine, program counter and bus structure are compatible with the original microcontroller. Designed main core supports all features discussed in previous chapter. Moreover, designed main core supports program memory paging feature which implies that designed main core can handle 8 Kbytes program memory while original device has 2 Kbytes program memory.

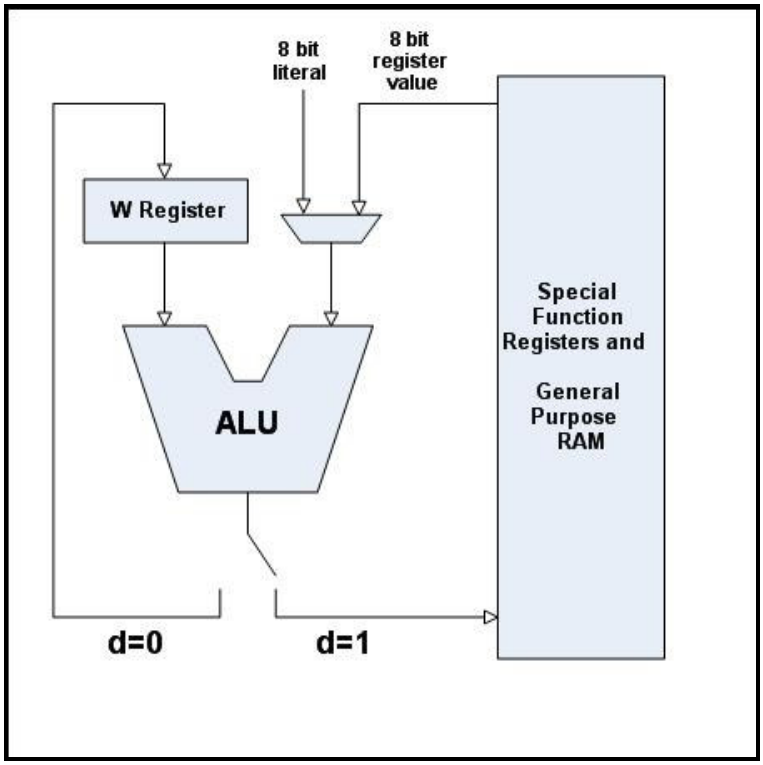


Figure 4- 3: Block diagram of ALU

4.2 Instruction Set

As a PIC micro family member, 16F871 has 14-bit instruction code and 35 instructions as other family members. The instruction set is highly orthogonal and can be divided into three main categories [24].

- Byte oriented file register operations
- Bit oriented file register operations
- Literal an control operations

For byte oriented instructions, “f” represents the target register and “d” represents destination designator. As mentioned previously if destination designator is equal to ‘1’, ALU result is placed into target register; if destination designator is equal to ‘0’ ALU result is written into working register. For bit oriented instructions, ‘b’ represents the bit select designator. For this purpose, three bits are allocated for bit select designator to be able to reach any bits. Also, “k” represents 8 bit literal value while for `CALL` and `GOTO` instructions ‘k’ represents the 11 bit literal value.

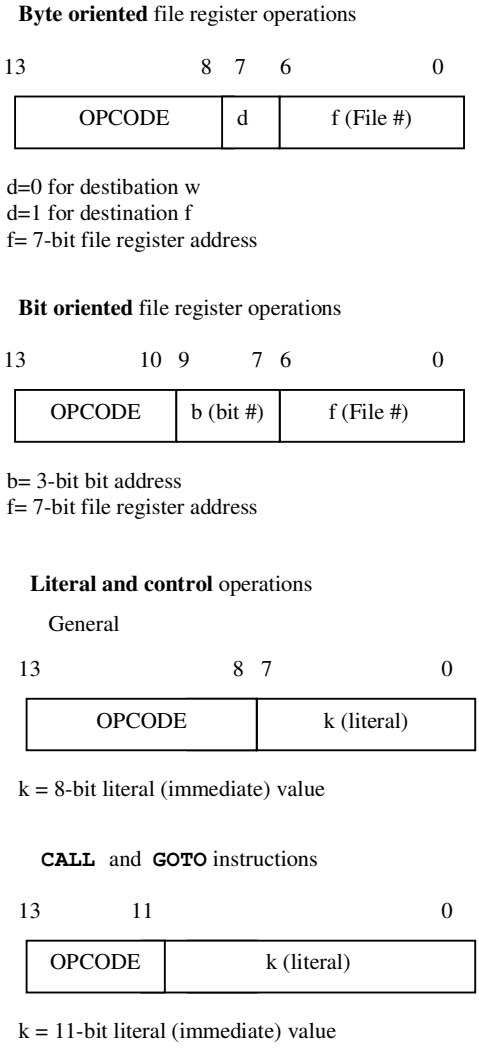


Figure 4- 4: Instruction format of the 16F871

4.3 Arithmetic Logic Unit (ALU)

The 16F871 has 8-bit Arithmetic Logic unit (ALU) that can perform arithmetic and Boolean logic operations between working register and any register. Instructions can process only one register as an operand at most. This implies that, for every arithmetic and logic operation, working register (w) is a fix operand while other operand may be a

literal value or a register. ALU module is capable of addition, subtraction, shift and logic operations. It is also an unchangeable stop for each data since all data buses terminate at this unit. Destination of data is determined by second operand of an instruction. If destination operand is equal to '1', result of the ALU is written to destination register. If destination operand is '0', result of the ALU is transferred to working register. By means of this switching methodology, any data can be transferred to any memory location. Schematic description of ALU is given in Figure 4.3.

ALU interacts with main core strongly and it is an inevitable stop for every data. For this reason designed ALU module is mounted into main core. Besides, it is hundred percent compatible with original Arithmetic Logic Unit that the16F871 involves.

4.5 Data Memory and Special Function Registers

Data memory houses the Special Function Registers (SFR) and General purpose registers (GPR). General purpose registers are implemented as RAM module while special function registers are designed as internal registers. Data memory is divided into four equal RAM Banks.

The 16F871 core has very large set of special function registers due to main core has RISC architecture and several peripheral modules. Implemented SFRs and their addresses are given in Table 4.1 which is grouped according to their banks.

Table 4- 1: Special Function registers reside in (a) Bank 0 (b) Bank 1 (c) Bank 2 (d) Bank 3

Address	Name	Description
0x00	INDF	Used for indirect addressing. Not a physical register
0x01	TMR0	Timer 0 module Register
0x02	PCL	Lower 8 bits of Program Counter
0x03	STATUS	Status Register
0x04	FSR	File Select Register. Indirect Data Memory access pointer
0x05	PORTA	Port A Data Latch
0x06	PORTB	Port B Data Latch
0x07	PORTC	Port C Data Latch
0x08	PORTD	Port D Data Latch
0x09	PORTE	Port E Data Latch
0x0A	PCLATH	Upper 5 bits of Program Counter
0x0B	INTCON	Interrupt control Register
0x0C	PIR1	Peripheral Interrupts Flag Register 1
0x0D	PIR2	Peripheral Interrupts Flag Register 2
0x0E	TMR1L	Holding Register for Least Significant byte of 16-bit TMR1 Register
0x0F	TMR1H	Holding Register for Most Significant byte of 16-bit TMR1 Register
0x10	T1CON	Timer 1 Control Register
0x11	TMR2	Timer 2 Module Register
0x12	T2CON	Timer 2 Control Register
0x15	CCPR1L	Capture/Compare/PWM Register 1 (LSB)
0x16	CCPR1H	Capture/Compare/PWM Register 1 (MSB)
0x17	CCP1CON	Capture/Compare/PWM Module Control Register
0x18	RCSTA	USART Receive Status and Control Register
0x19	TXREG	USART Transmit Data Register
0x1A	RCREG	USART Receive Data Register
0x1E	ADRESH	A/D Result Register High Byte
0x1F	ADCON0	A/D converter Module Control Register

(a)

Table 4.1: (Continued)

Address	Name	Description
0x80	INDF	Used for indirect addressing. Not a physical register
0x81	OPTION_REG	Option Register
0x82	PCL	Lower 8 bits of Program Counter
0x83	STATUS	Status Register
0x84	FSR	File Select Register. Indirect Data Memory access pointer
0x85	TRISA	Port A Data Direction Register.
0x86	TRISB	Port B Data Direction Register.
0x87	TRISC	Port C Data Direction Register.
0x88	TRISD	Port D Data Direction Register.
0x89	TRISE	Port E Data Direction Register.
0x8A	PCLATH	Upper 5 bits of Program Counter
0x8B	INTCON	Interrupt control Register
0x8C	PIE1	Peripheral Interrupts Enable Register 1
0x8D	PIE2	Peripheral Interrupts Enable Register 2
0x92	PR2	Timer 2 Period Register
0x98	TXSTA	USART Transmit Status and Control Register
0x9E	ADRESL	A/D Result Register Low Byte
0x9F	ADCON1	A/D Module control register

(b)

Table 4.1: (Continued)

Address	Name	Description
0x100	INDF	Used for indirect addressing. Not a physical register
0x101	TMR0	Timer 0 module Register
0x102	PCL	Lower 8 bits of Program Counter
0x103	STATUS	Status Register
0x104	FSR	File Select Register. Indirect Data Memory access pointer
0x106	PORTB	Port B Data Latch
0x10A	PCLATH	Upper 5 bits of Program Counter
0x10B	INTCON	Interrupt control Register
0x10C	EEDATA	EEPROM Data Register
0x10D	EEADR	EEPROM Address Register
0x11D	DACON0	D/A module control register
0x11E	DADATH	D/A data register high byte
0x11F	DADATL	D/A data register low byte

(c)

Address	Name	Description
0x180	INDF	Used for indirect addressing. Not a physical register
0x181	OPTION_REG	Option Register
0x182	PCL	Lower 8 bits of Program Counter
0x183	STATUS	Status Register
0x184	FSR	File Select Register. Indirect Data Memory access pointer
0x186	TRISB	Port B Data Direction Register.
0x18A	PCLATH	Upper 5 bits of Program Counter
0x18B	INTCON	Interrupt control Register
0x18C	EECON1	EEPROM Read/Write Configuration Register 1
0x18D	EECON2	EEPROM Control Register 2. Not a Physical Register

(d)

Only unimplemented registers are EEDATH and EEADRH registers which are used for flash memory (program memory) read and write operations. Addition to original register set, there are three more registers namely DACON0, DACDATH and DACDATL which are the control and data registers of D/A converter.

4.6 The Parallel Slave Port (PSP)

The 16F871 has parallel Slave Port module for parallel communication. Slave term implies that, data read or write operations are performed if external master device orders it. When PSP module is activated, data transmission and reception is performed through Port D pins, and synchronization controls are performed via Port E pins. For reading or writing operations PORTD register is used as a buffer register. PSP module can directly interface an 8 bit microprocessor data bus [26]. Parallel slave port module is first designed separately from main core. But, Since it is strongly synchronous with main core, data read and write operations are performed with PORTD register and in order to reduce hand-shakings between these modules, designed PSP module is mounted to main core and it is hundred percent compatible with the original parallel port that the 16F871 has.

Parallel Slave port is controlled using TRISE register. Lower 3 bits of TRISE register determines the direction of Port E which is a three pin port. Other bits are Parallel slave port control and status bits. Bit map of the TRISE register is given in Table D.3.

When PSP module is enabled, in other words PSP mode is activated, Port D becomes a parallel data bus as mentioned. Also, enabling PSP module turns Port E pins into $\overline{CS}$, $\overline{RD}$ and $\overline{WR}$ input pins, which control chip selection, read and write operations respectively.

Write operation

A write to PSP from external system occurs when both $\overline{CS}$ and $\overline{WR}$ pins are first detected low [26]. This is the start signal for write operation. When either $\overline{CS}$ or $\overline{WR}$ pins are high again input buffer full status flag (IBF) is set which declares that, data write is completed and 8 bit data is waiting to be read. This flag is cleared when `PORTD` register is read. If another write operation is attempted before previous data read, Input buffer overflow detect bit (IBOV) will become high that indicates inappropriate operation. Parallel slave port interrupt flag (PSPIF) is set after write operation.

Read Operation

A read from the PSP from the external system, occurs when both $\overline{CS}$ and $\overline{RD}$ pins are first detected low. This is a start signal for read operation. Then, Output buffer full flag (OBF) is cleared indicating that `PORTD` output latch is read by the external module. In other words, PSP output port is ready for read operation. When either $\overline{CS}$ or $\overline{RD}$ pins become high, Parallel Slave Port interrupt flag (PSPIF) becomes high indicating that read operation is complete. OBF bit will remain clear until another data is written to output data latches by software [26].

4.7 The Universal Synchronous/Asynchronous Receiver Transmitter (USART)

Designed serial port (USART) is implemented as a separate module. This is the most complicated module after main core. It has four state machines with nine states. State machines perform synchronous/asynchronous transmission and reception operations. Designed serial port is completely compatible with original one and it exactly follows the original procedure during data transmission and reception.

USART module supports following modes

- Asynchronous (full duplex)
- Synchronous – master (half duplex)
- Synchronous – slave (half duplex)

Synchronous mode implies that, data transmission is performed in accordance with clock. In slave mode synchronization is achieved by external device's supervision while in master mode clock pulses are generated by USART module. Asynchronous data transmission or reception begins after detection of start bit which is '0' and finishes after detection of stop bit which is '1'. Full-duplex term means transmission and reception can be carried out simultaneously, while synchronous mode can either transmit data or receive data at a time. USART module is configured with `TXSTA` and `RCSTA` registers which correspond to transmission and reception control registers respectively. `TXSTA` and `RCSTA` registers are given with details in Table D.4, D.5. Data that will be transmitted should be transferred to a buffer register which is called `TXREG`. Received data is transferred to another buffer register which is named as `RCREG` register. This register is a double buffered register, in other words it is a two deep FIFO [32].

USART data transmission or reception speed depends on the Baud rate generator. Baud rate generator makes interfaced devices remain in accordance. Baud rate clock speed depends on the speed of the devices and baud rate value which is saved in `SPBRG` register. Before any operation, proper baud rate clock values must be calculated for each device.

USART module supports the 8-bit or 9-bit data transmission. 9^{th} bit is generally used as parity bit. Parity bits are stored in 0^{th} bit of `TXSTA` register while transmitting a data or in 0^{th} bit of `RCSTA` register while receiving a data. Data receptions can be continuous or single byte in synchronous mode.

Interrupt behavior of serial port a little bit different than the other modules. Transmission or reception interrupt flags are read only bits and they cannot be cleared by software. Transmission interrupt flag bit remains high till `TXREG` is written. In other words,

transmission interrupt occurs when serial transmission is started. When a new data is written to TXREG, interrupt flag is cleared until transmission starts. Reception interrupt occurs when data receive is complete and it is cleared when RCREG has been read [32].

4.8 Timers

The 16F871 microcontroller has four timers namely Timer 0, Timer 1, Timer 2 and Watch-dog timer (WDT). Timer 0, Timer 1 and Timer 2 can be used as a timer or synchronous/asynchronous counter. Two timer modules, Timer 1 and Timer 2 are designed as a separate module while Timer 0 and Watch-dog timer modules are embedded to main core. Except some limitations of the WDT, timer modules are compatible with original PIC 16F871 timers. Details about each timer will be discussed separately.

4.8.1 The Timer 0

Timer 0 is an eight bit timer/ counter module which can be used with external clock source with edge sensitivity selection option. It has also 8 bit programmable pre-scalar. Timer 0 overflows at 0xFF and when overflow occurs, the Timer 0 interrupt flag (T0IF) is set. OPTION_REG register is a control and configuration register for Timer 0.

When Timer 0 is activated, it will increment regularly at every instruction cycle (without pre-scalar). Timer0 and watchdog timer (WDT) shares the same pre-scalar mutually exclusively [23]. Prescalar assignment is configured using PSA bit. Assigned prescalar values are different for WDT and Timer 0. Please refer to Table D.2 for details and bit descriptions of OPTION_REG register.

4.8.2 The Timer 1

Timer 1 is a 16 bit timer/counter consists of two 8 bit registers namely TMR1H and TMR1L which are readable and writable registers. Timer 1 register pair increments from 0x0000 to 0xFFFF (if another value is not written) and rolls-over to 0x0000 [27].

Timer 1 is configured via T1CON register and it can be used as

- Synchronous timer
- Synchronous counter
- Asynchronous counter

In timer mode, Timer 1 module uses the internal clock which is equal to $f_{OSC}/4$. In synchronous counter mode it is triggered by rising edge of the external clock source. Synchronization implies that, in SLEEP mode Timer 1 will not increment. If $\overline{\text{T1SYNC}}$ bit is set, Timer 1 is in asynchronous counter mode. Asynchronous mode ensures increment of Timer 1 even in SLEEP mode. Timer 1 has a three bit pre-scalar and it can be used for different frequency values for either internal or external clock sources. Another duty of Timer 1 module is controlling the Capture and Compare modes of CCP module.

4.8.3 The Timer 2

Timer 2 is an 8-bit timer with pre-scalar and post-scalar. TMR2 register which is a readable and writable register holds the value to be incremented. Although it is an 8-bit counter and with highest pre-scalar and post-scalar configurations, overflow time can be configured as same as a 16 bit counter's overflow time [28]. It only uses internal clock as clock source. For this reason in sleep mode it does not operate.

Timer 2 overflows when TMR2 register is equal to PR2 register, and then Timer 2 interrupt flag (TMR2IF) is set. PR2 is also a readable and writable register. Timer 2 module can be configured via T2CON register which is given in Table D.6. It also controls the PWM mode of the CCP module.

4.8.4 The Watch-dog Timer

Watch-dog timer is a free running eight bit timer. WDT enable or disable is declared in software during program loading stage. WDT generates a reset signal when it overflows.

Thanks to this operation, it protects main core from dead-locks. For this purpose, during program execution WDT must be cleared regularly by `CLRWDT` instruction. The $\overline{TO}$ bit of the `STATUS` register will be cleared upon WDT reset [23]. WDT is configured using `OPTION_REG` register. As mentioned in previous sections, Timer 0 and WDT modules are embedded into main core. Timer 0 is fully compatible with original module while designed WDT module is excluded from pre-scalar which Timer 0 and WDT modules are supposed to share. In other words, designed WDT does not utilize prescalar.

4.9 The Capture/Compare/PWM module (CCP)

CCP module is designed in a super module that contains Timer 1 and Timer 2 since capture and compare modes are controlled by Timer 1 and PWM mode is controlled by Timer 2. Designed CCP module is hundred percent compatible with the CCP module of the 16F871.

CCP module contains 16-bit register which can operate as a 16-bit capture register, as a 16-bit compare register or 10-bit pulse width modulation (PWM) master/slave duty cycle register [31]. This 16-bit register named as `CCPR1` register consists of two separate 8-bit registers namely `CCP1H` and `CCP1L`. CCP module is configured using `CCPCON` register which is given in Table D.7.

Capture mode

In capture mode, 16-bit `TMR1` register is captured and transferred to `CCP1H` and `CCP1L` registers when an event detected on CCP1 pin. An event is defined as

- Every falling edge
- Every rising edge
- Every 4^{th} rising edge
- Every 16^{th} rising edge

Event selection can be handled through CCPCON register. After a capture operation, hardware sets the CCP interrupt flag (CCPIF). Capture mode is quite useful to measure the duration between two consecutive events.

Compare mode

In compare mode, 16-bit CCPR1 register is constantly compared with TMR1 register pair. When a match occurs, the CCP1 pin is [31].

- Driven high
- Driven low
- Remains unchanged.

An interrupt generation accompanies the match. Behavior of the compare mode is configured with CCPCON register.

Pulse width modulation mode (PWM)

In pulse width modulation mode, CCP module produces up to 10-bit resolution PWM output. As mentioned previously Timer 2 module controls the PWM mode of the CCP module. When TMR2 register is equal to PR2 register, PWM duty cycle is latched from CCPR1L into CCPR1H and CCP1 pin is driven high. In next cycle of TMR2 register, when concatenation of TMR2 and T2CON<1:0> is equal to duty cycle register, CCP1 pin is driven low. Duty cycle register is concatenation of CCPR1H register and CCP1CON<5:4>. PWM mode does not generate an interrupt and continuously produce modulated signal.

4.10 The A/D Converter

A/D converter is described behaviorally using SyctemC – AMS library. Designed core has 14 bit A/D converter despite to original A/D converter that 16F871 involves is a 10 bit module. A/D converter is designed and simulated separately, and then it is connected

to main core. Designed converter module operates exactly same with described one. But in this design data acquisition module is not used. In other words, analog voltage is directly applied to A/D converter module. There is single channel for analog voltage input and reference voltages. Last restriction is, designed A/D converter can only be used with common clock source which implies that it cannot be used with an external RC clock. At first sight there seems several deviations from original module but these differences do not directly affect the operation of the A/D conversion. Fundamental properties of the A/D converter module are saved.

A/D converter module is controlled and configured with two registers namely ADCON0 and ADCON1. ADCON0 register configures the conversion clock frequency, analog channel selection and A/D conversion start/stop operation. ADCON1 register controls the result format and channel selection. In Table D.8a and D.9, A/D converter control registers are given in detail. However, ADCON1 register's channel selection bits are not applicable since single channel is defined in this design. High byte of converted digital data is saved in ADRESH while low byte of converted digital data is saved in ADRESL register [29].

For A/D conversion operation A/D converter module must be turned on via setting ADON bit of ADCON0 register. A/D conversion is started by means of setting GO/$\overline{\text{DONE}}$ bit. When the conversion is over GO/$\overline{\text{DONE}}$ bit is cleared by hardware automatically. Conversion of each bit calls for at least 2 clock cycles if A/D converter clock is configured as fosc/2. If clock selection of A/D converter module is configured as fosc/4, in order to convert each bit, at least 4 clock cycles are needed and so on. When the conversion is over, A/D conversion interrupt flag (ADIF) is set. Besides, A/D converter module can operate even the main core is in the SLEEP mode.

4.11 The D/A Converter

Normally, the PIC 16F871 microcontroller device does not have a D/A converter. This module is implemented in order to enhance the properties of main core and extend the SystemC-AMS study for the thesis. Before D/A converter design, several D/A converters are evaluated.

Implemented module is a 12-bit D/A converter. For this purpose, three additional registers are defined namely DACON0, DADATH and DADATL. DACON0 is the control register of the D/A converter module while the following two registers hold the data to be converted to analog signal.

Clock frequency of the D/A converter is the half of the main core. It samples and updates the digital data at Q1 cycle, then updates the analog output signal at Q4 cycle. Data conversion is completed in one instruction cycle. Thanks to this way, consecutive data can be sent to D/A converter in order to gain a signal waveform. 12-bit digital data transferred through DADATH and DADATL registers where lower byte is stored in DADATL. 1st bit of the DACON0 register is cleared when a data is written to DADATH register and it is set when 4 bit data is written to DADATL register. This bit allows digital data update and hinders erroneous results. Otherwise when low byte of the 12-byte data is assigned with new data output will change in same instruction cycle. D/A converter module doesn't generate an interrupt.

4.12 Interrupt Controller

As the name implies, an interrupt is some event that interrupts normal program execution. As stated earlier, program flow is always sequential and it can be altered only by those instructions that expressly cause program flow to deviate in some way [30]. But interrupts change program flow temporarily. They force program flow to branch and execute a subroutine code, and then it resumes normal program flow. Interrupts provide ability to main core to interact with the external events. Designed main core handles the interrupt operations and sources exactly same with the original PIC 16F871.

The 16F871 microcontroller has following interrupt sources

1. RB Port change interrupt
2. RB0/INT external interrupt
3. Timer 0 overflow interrupt
4. Timer 1 overflow interrupt

5. Timer 2 overflow interrupt
6. EEPROM read/write complete interrupt
7. Parallel Slave port read/write interrupt
8. A/D converter interrupt
9. USART Receive interrupt
10. USART Transmit interrupt
11. CCP module interrupts.

Interrupt sources given above have equal priority. `INTCON` register is the status and control register for interrupts given in 1, 2 and 3. These are generally called main core interrupts while others are known as peripheral interrupts. Each interrupt has a declaration flag bit and an enable bit on the control registers. Peripheral interrupts are declared via `PIR1` and `PIR2` registers and each peripheral interrupt is enabled or disabled via `PIE1` and `PIE2` registers. In Table D.11 and D.12 these registers and task of each bit are given in detail.

CHAPTER 5

VERIFICATION OF THE DESIGNED CORE

In order to verify the design, it is needed to perform some simulations to observe the response of the modeled system. For this purpose, a test-bench setup is built that consists of stimulus modules and display modules. Test bench architecture is depicted in Figure 5.1.

Stimulus module sends appropriate signals to main core through data ports (Port A, Port B etc.) which main core collects information from outer world. Also reset and interrupt pins which changes program execution flow in case of asynchronous events are connected to stimulus module.

TestROM module contains the program code in hexadecimal format that main core will execute. It can be considered as a pseudo EEPROM which consists of a C++ code that reads .hex file which is generated by compiler, and transfers this data to an array conveniently. When address of a memory location is sent to TestROM by means of program address ports, it releases the hex code of corresponding instruction via program data ports.

Some digital peripheral modules have particular display module which simulates corresponding system they should be connected to. In the beginning, peripheral modules are designed and simulated independently. Then, they are wired to main core using SystemC signals. Second simulation also checks accordance with main core and re-simulates the peripheral if module operates accurately. For second simulations, a particular test program is written and executed for each peripheral. Besides, second verification checks if control registers' of the peripheral module operates conveniently. This is important since there is a set of registers associated with each module. Analog

components have separated stimulus and display modules as well. Analog stimulus modules provide reference voltages and channel voltages while analog displays collect analog output data and writes to a file for tracing.

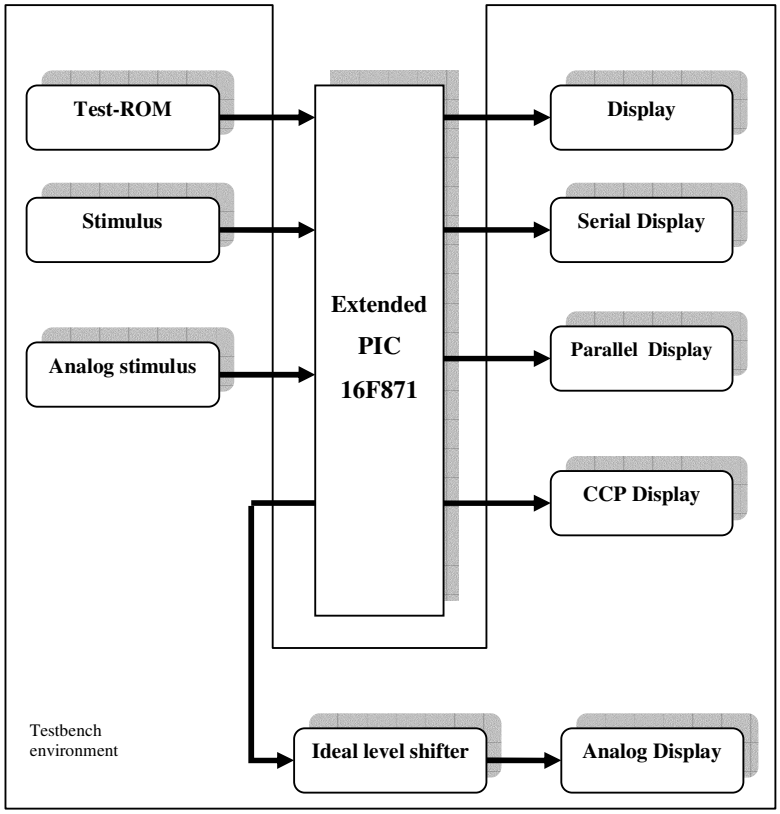


Figure 5- 1: Testbench and the extended PIC 16F871 design for testing

Main point of verifying the design is building a testbench and TestROM such that, it contains all possible combinations of events that can be executed. For this purpose, several test programs have been written. One of this program checks almost all instructions whether the system performs appropriate operation. Other programs are either for checking asynchronous operations such as resets and interrupts or for peripheral components' verification. This test programs are written in assembly

language, compiled and linked with MPLAB tool. After compilation the .hex file is received which contains the hexadecimal values correspond to written instructions. TestROM module directly reads this file and generates a pseudo ROM array as mentioned previously.

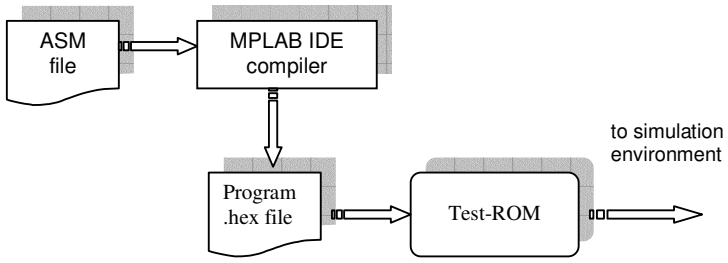


Figure 5- 2: Test-ROM generation flow

Results of simulations are received in .vcd format for digital modules and in .dat format for analog ones. First group can be observed using VCD viewers while additional programs are needed like “octave” for .dat files.

5.1 The Main Core

The PIC 16F871 has an 8 bit core. This core performs several operations in 4 oscillator cycles which is called 1 execution cycle. Branching operations are performed in 2 execution cycles which corresponds to 8 oscillator cycles.

Main core receives instructions from program memory. It directly sends program counter’s value through *prog_adr_o* pin and receives corresponding instruction. Instructions are in hexadecimal format and received through prog_dat_i pin. For this study, a test_ROM module is implemented which is a C++ code that reads .hex form of a compiled and linked assembly code. Communication signal waveforms between main core and test_ROM are given in Figure 5.3.

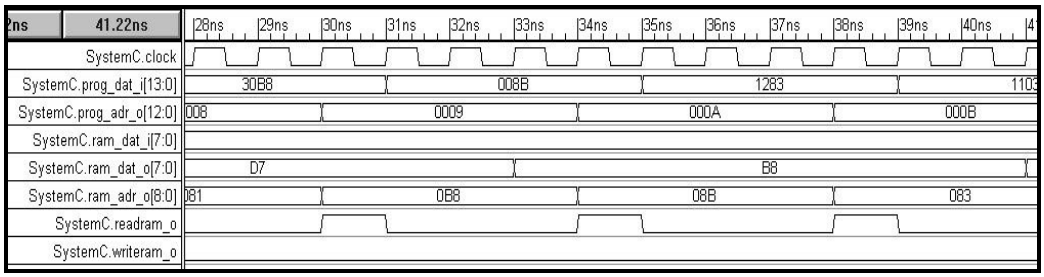


Figure 5- 3: ROM signals of the extended 16F871

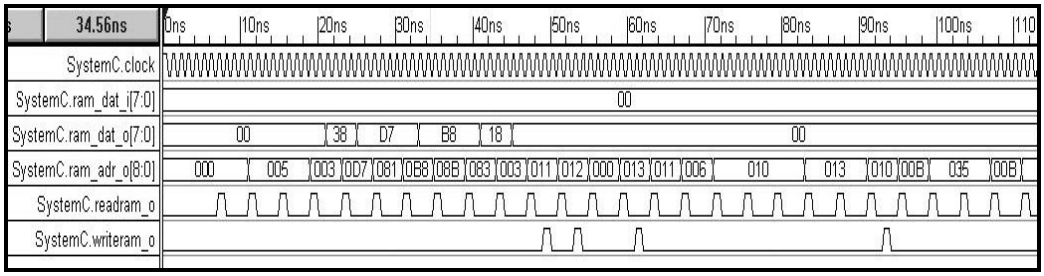


Figure 5- 4: RAM signals of the extended 16F871

Handshaking of main core and the data RAM are a little bit different. Before reading or writing operations, main core should send signal that informs RAM memory about the process whether it is a read or write operation. When main core attempts to write to data RAM, it asserts write_ram_o pin then sends address and data through ram_adr_o and ram_dat_o respectively. Similarly, read operation starts with assertion of read_ram_o pin. Then, main core sends address of memory location and receives data from ram_dat_i pin. Waveform of the communication between RAM and main core is given in Figure 5.4. The 16F871 has on-chip ROM and RAM modules. External RAM or ROM memory usages require specified software for handshaking between microcontroller and external device.

5.1.1 The Instruction Architecture Test

The 16F871 has instruction set with 35 instructions. All instructions except `NOP`, `RETFIE` and `CLRWDT` instructions are tested individually and their operations are approved by the test program given in APPENDIX B. Idea of this test program is giving initial values to the registers, comparing the results after execution of the instruction and checking the result if it is the expected value. If the result is correct, code jumps to next command's block. Otherwise, assembly code returns a decimal number corresponds to that instruction so that instruction which works improperly can be determined. If all instructions pass, the examination assembly code returns 0xFF value which is received after execution of the test code.

5.1.1.1 Arithmetic & Logic Instructions

The 16F871 can perform arithmetic and logic instructions including addition, subtraction, increment, decrement, logical and, logical inclusive or, logical exclusive or, byte swap, left and right shift and complement. Instructions are executed after assigning some values to target registers. Expected result is subtracted from the register and then zero flag is checked. Zero flag becomes high if result of the arithmetic operation is zero. Each instruction passed the test program which implies that arithmetic and logic instructions operate correctly. List of verified arithmetic and logic instructions are given below.

```
ADDWF   Register1,W
ADDLW   literal_value
ANDWF   Register1,W
ANDLW   literal_value
SUBLW   literal_value
SUBWF   Register1,W
COMF    Register1
DECF    Register1
INCF     Register1
IORWF   Register1,W
IORLW   literal_value
```

```
XORLW   literal,value
XORWF   Register1,W
RLF     Register1
RRF     Register1
SWAPF   Register1
```

5.1.1.2 Data Transfer Instructions

Data transfer instructions can be grouped into two main categories. Byte oriented instructions and bit oriented instructions. Byte oriented data instructions perform 8-bit data transfer between registers. Special function registers can only be loaded with working register whereas working register can be loaded with either a literal value or special function register. Bit oriented instructions assign either 0 or 1 value to target bit of destination register.

There are also particular commands such as `CLRW` that directly assigns 0 to the working register (w). Verified data transfer instructions are listed below.

```
CLRW
CLRF   Register1
MOVF   Register1
MOVWF  Register1
MOVLW  Literal_value
BCF    Register1,bit_number
BSF    Register1,bit_number
```

5.1.1.3 Flow Control Instructions

Flow control instructions changes the normal sequence of the assembly code by means of changing the content of program counter. The PIC 16F871 does not have conditional branching operation. Instead, it skips next line of code if a certain condition is satisfied. This can be formulized as, condition1 - next line1, else - next line2. `GOTO` command should be used in line1 to retain examination of the condition. A real conditional branching can be emulated with combination of a couple of commands. These conditions

are based on contents of registers or their specified bits. List of verified flow control instructions are given below.

```
INCFSZ    Register1
DECFSZ    Register1
BTFSC     Register1,Bit number
BTFSS     Register1,Bit number
CALL      Label_name
GOTO      Lable_name
RETURN
RETLW
```

5.1.1.4 Other Instructions

One of the important instructions of 16F871 is `SLEEP` command. This command traps state register in Q1 so that main core and peripherals that are synchronous with main core would stop temporarily which aims to reduce power consumption of controller when it is idle. There is no command that makes the device awake. This can only be achieved with a PORTB external interrupt. Response of the device to SLEEP command is given in Figure 5.5. Awakening of device will be covered in Section 5.9.

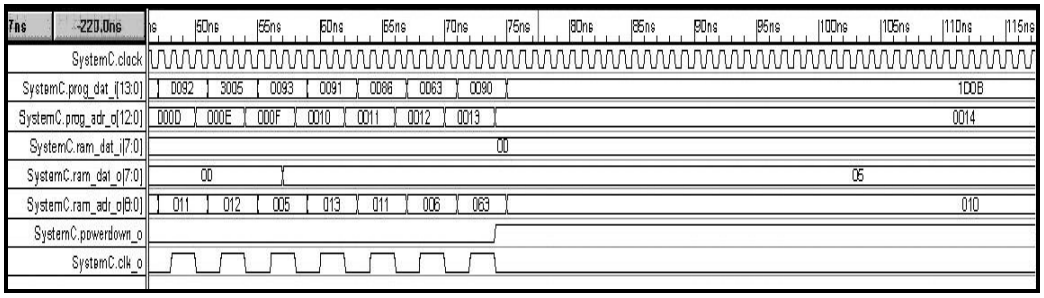
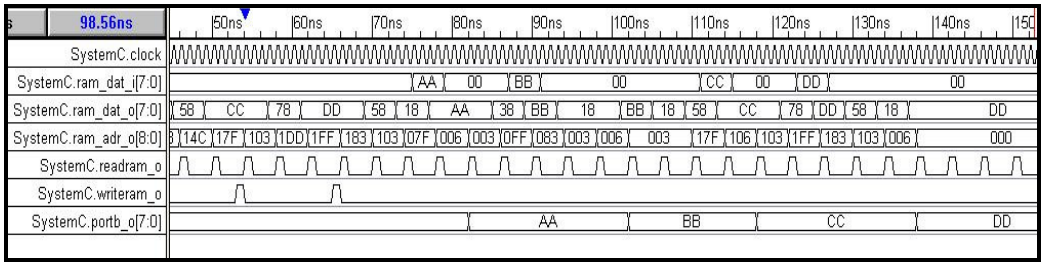


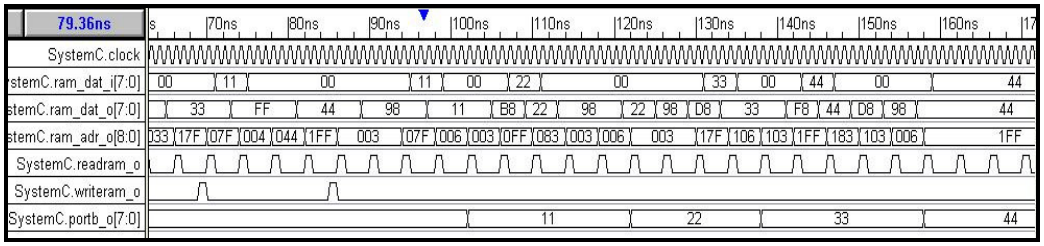
Figure 5- 5: SLEEP instruction and power-down mode

5.1.2 Addressing Mode Verifications

The 16F871 microcontroller supports two addressing modes which are described in Chapter 3 previously. In order to verify if designed microcontroller performs the addressing modes properly two verification programs are prepared. Since addressing mode is handled during code decoding, there is no need to check each instruction with both addressing modes. For this purpose, different memory locations in different memory banks are written using direct and indirect addressing modes. Then, same memory locations are read again and same results are received. Waveforms of addressing mode verifications are given in Figure 5.6.



(a)



(b)

Figure 5- 6: Addressing modes (a) direct addressing (b) indirect addressing

5.2 The Parallel Slave Port (PSP)

In order to verify parallel slave port (PSP), parallel port is enabled. For this purpose, 5th bit of `TRISE` register is set. Parallel port supports read and write operations. Write mode allows a write operation to be performed by master processor. Transferred data will be written into the 16F871's `PORTD` register. When PSP module detects both chip select ($\overline{CS}$) and Write ($\overline{WR}$) pins low a write operation occurs. When either $\overline{CS}$ or $\overline{WR}$ pin becomes high again, IBF becomes high and in same cycle interrupt flag (PSPIF) is set. In Figure 5.7 write operation of PSP port is shown. 0xDE value was received from parallel display module and transferred to `PORTD` register.

A read operation will begin when both $\overline{CS}$ and $\overline{RD}$ pins become low. After that output buffer full (OBF) becomes low immediately. This indicates that data at the Port D was read by the master. When either $\overline{CS}$ or $\overline{RD}$ pins become high again, read operation will be completed after interrupt flag becomes high (PSPIF). Read operation of Parallel port is shown in Figure 5.8. 0x50 hexadecimal value was read by the parallel display module.

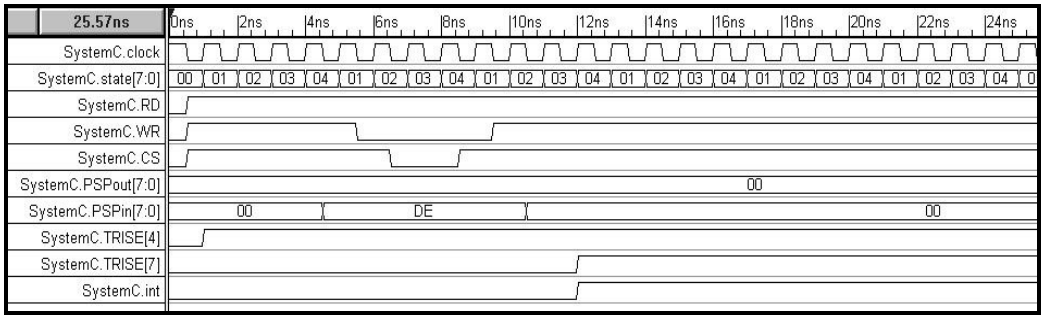


Figure 5- 7: Parallel port write operation

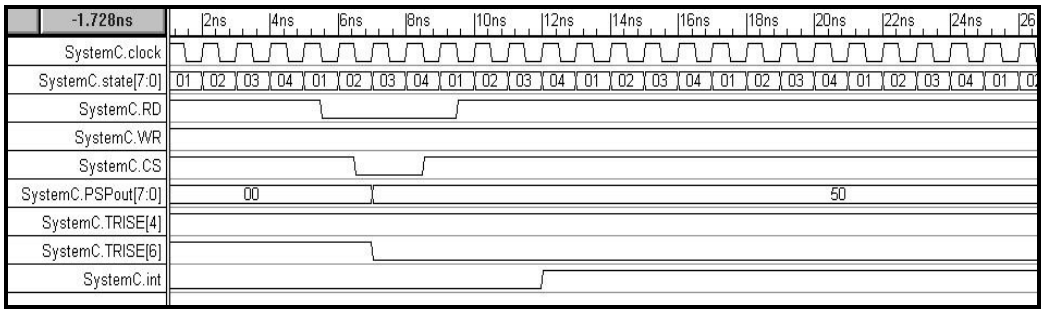


Figure 5- 8: Parallel port read operation

5.3 The Serial Port (USART)

Serial port was verified by means of configuring it for transmission and reception separately. Serial port is controlled by two main registers; `TXSTA`, transmit status and control register and `RCSTA`, receive status and control register. `TXREG` and `RCREG` keep the data that will be transmitted or received respectively. This device supports full-duplex operation for asynchronous mode which means it can perform transmission and reception simultaneously.

For Transmission operation `RCSTA` and `TXSTA` registers are loaded with 0x90 and 0x24 hexadecimal values respectively. This configuration enables asynchronous serial transmission and reception. Serial output generates continuous high value when it is idle. To inform the receiver module that an 8-bit data is coming it asserts serial out low. This is called start bit. After sending start bit, it begins to send the data bit by bit starting from LSB. When data transmit is complete, serial output is driven high again in order to inform the receiver that data transmission is completed. In this simulation `TXREG` was loaded with 0x8 and 0xCC values in sequence. For this reason after completing the first data transmission, serial output was driven high first, then driven low again to start the second data's transmission. Transmission operation of designed USART is as in Figure 5.9.

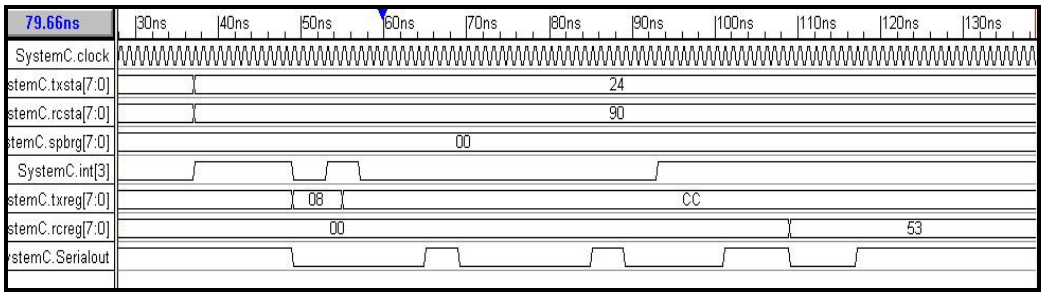


Figure 5- 9: USART transmission operation

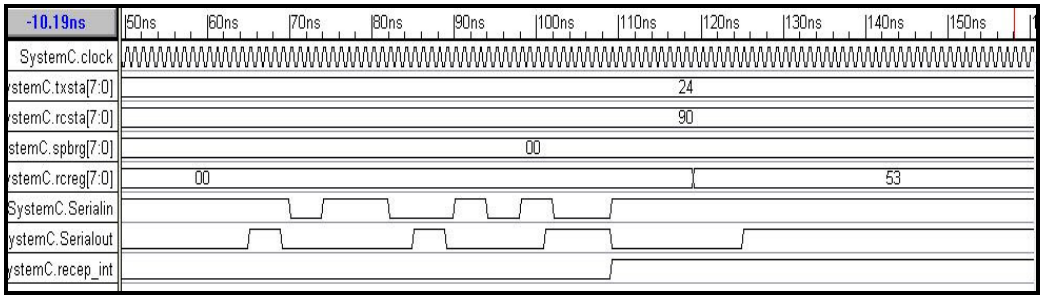


Figure 5- 10: USART reception operation

In order to verify reception operation a serial display which is simple serial transmitter is connected to Serialin port. Since this module can perform full–duplex operation configurations of control registers were saved. Serial display module was configured to send 0x53 hexadecimal data. It detects the start bit first, and then gains the serial data bit by bit. After receiving the stop bit, serial data is loaded into `RCREG`. Serial reception operation is shown in Figure 5.10.

5.4 Timer Modules

The 16F871 device has 3 timers. In this chapter Timer1 (16 bit timer) and Timer2 (8 bit timer) operations will be verified for convenience. For this purpose, Timer1 module was set as synchronous counter and `TIMER1H` and `TIMER1L` registers are loaded with

values 0xFF and 0xFA respectively. Internal clock (fosc/4) was selected and it is divided by 8 using prescaler. VCD waveform of simulation result is depicted in Figure 5.11. Timer1 register was overflowed at 0xFFFF and generated Timer1 interrupt which made Timer1 interrupt flag set. It should be noted that after overflow timer1 still keeps counting. For this reason, it should be turned off at ISR or in normal program code.

In contrast to Timer1, Timer2 overflows when `TMR2` register matches `PR2` register since Timer2 module also controls PWM block. `PR2` register was loaded with 0x7F hexadecimal value. When these two register matches, Timer2 module generates an interrupt that sets Timer2 interrupt flag. If PWM was enabled Timer2 register increments till it is stopped by software. For this simulation prescaler value is assigned as 1:16 and postscaler value assigned as 1:1. VCD waveform of Timer2 operation is given in Figure 5.12.

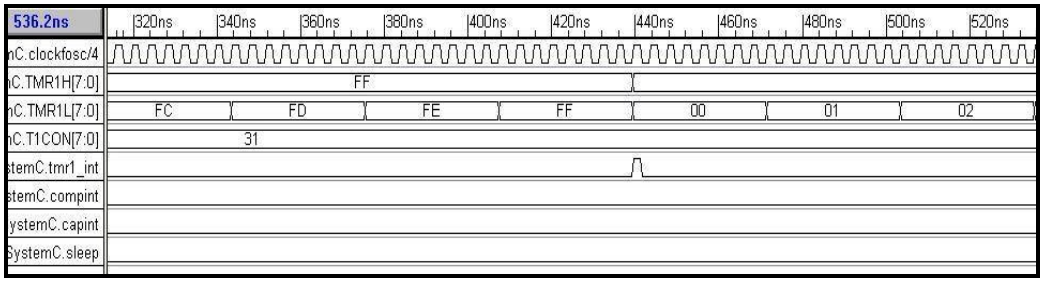


Figure 5- 11: The Timer 1 operation

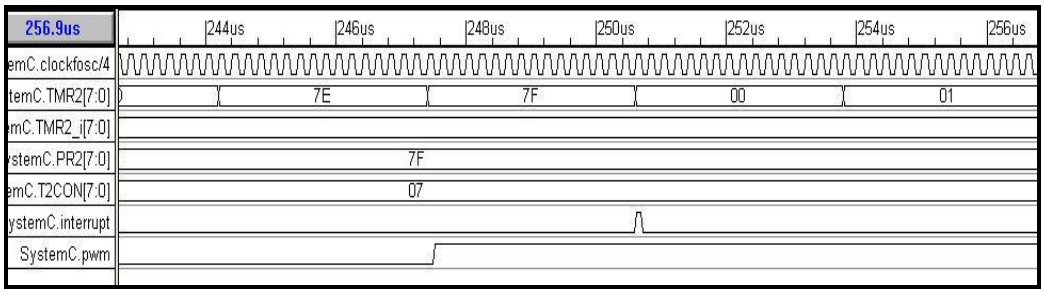


Figure 5- 12: The Timer 2 operation

5.6 The Capture, Compare, PWM Module (CCP)

The Capture, Compare and PWM (CCP) module supports different modes and each mode was simulated separately. For convenience, CCP module was configured for Capture mode operation first. CCPCON register was loaded with 0x5 for rising edge capturing. When a rising edge comes to CCP pin, contents of TIMER1H and TIMER1L registers were assigned to CCPR1H and CCPR1L registers respectively. Then, capture mode was switched to each falling edge capturing and CCPCON register was loaded with 0x4. When falling edge comes to CCP pin, contents of TIMER1H and TIMER1L registers were transferred as described. This procedure is for measuring duration between two events. In software only thing that the programmer should do is subtracting the captured results. Of course CCPR1H and CCPR1L registers should be assigned to temporary registers. Otherwise former values will be lost. Verification of capture mode is depicted in Figure 5.13.

Secondly, CCP module was configured for compare mode such that CCP pin would be forced to be low when a match occurs between TIMER1H, TIMER1L and CCPR1H, CCPR1L registers respectively. For this purpose, CCPCON register was loaded with 0x9. After the match, capture interrupt flag was set. Simulation waveform of compare mode is given in Figure 5.14.

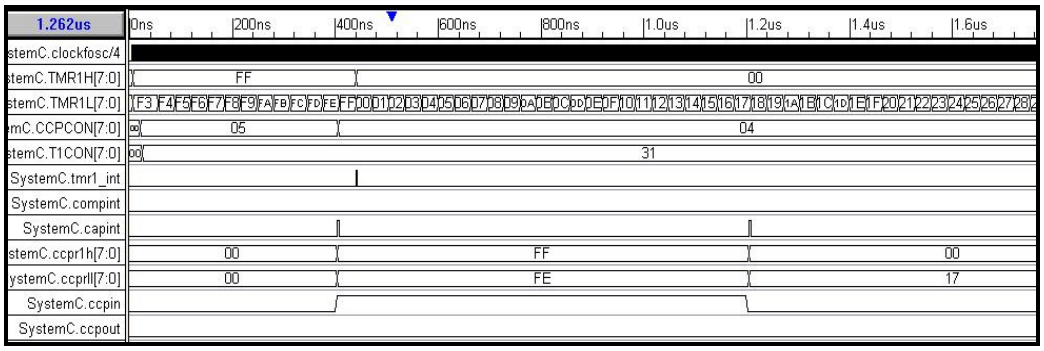


Figure 5- 13:The CCP capture mode

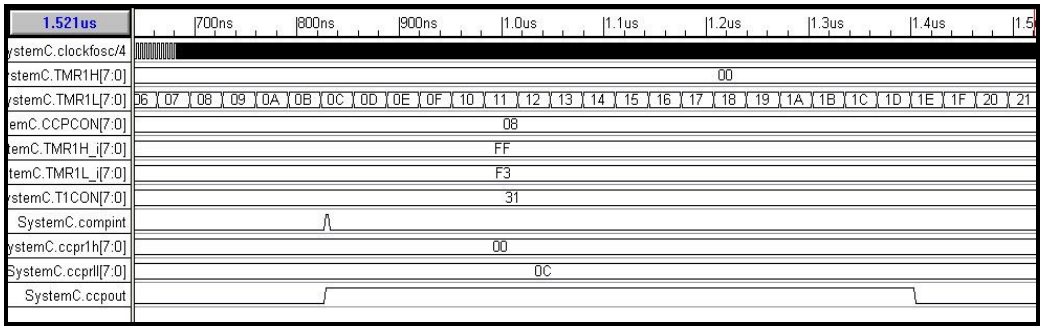


Figure 5- 14: The CCP compare mode

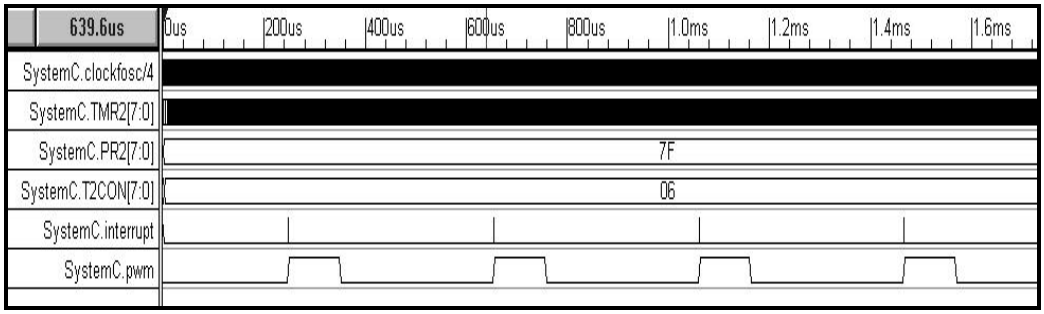


Figure 5- 15: The CCP PWM mode

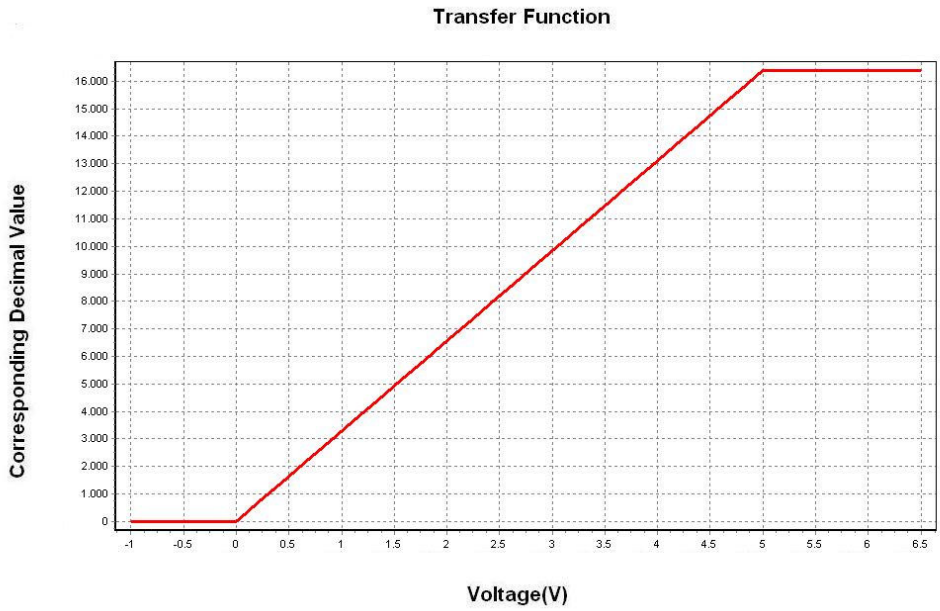
To generate a pulse width modulated output, at first PWM output period and duty cycles should be determined. Formula of the PWM period is given in equation 5.1 and PWM duty cycle is given in equation 5.2. Clock cycle of main core was adjusted to 20 Mhz and Prescaler of the Timer2 was configured to 1:16 and `PR2` register's value was assigned as 0x7F as mentioned previously. Expected PWM output period is 409.6 us (~2441 Hz) and aimed duty cycle is %25. For this purpose, 10 bit PWM register was assigned with 0x82. This register's upper 8-bit comes from `CCPR1L` register and lower 2 bit comes from `CCP1CON` [5:4]. Assigned values are calculated from equation 5.1 and equation 5.2. Output waveform of the PWM simulation is given in Figure 5.15.

PWM period = [(PR2) + 1] • 4 • TOSC• (TMR2 prescale value), specified in units of time (5.1)

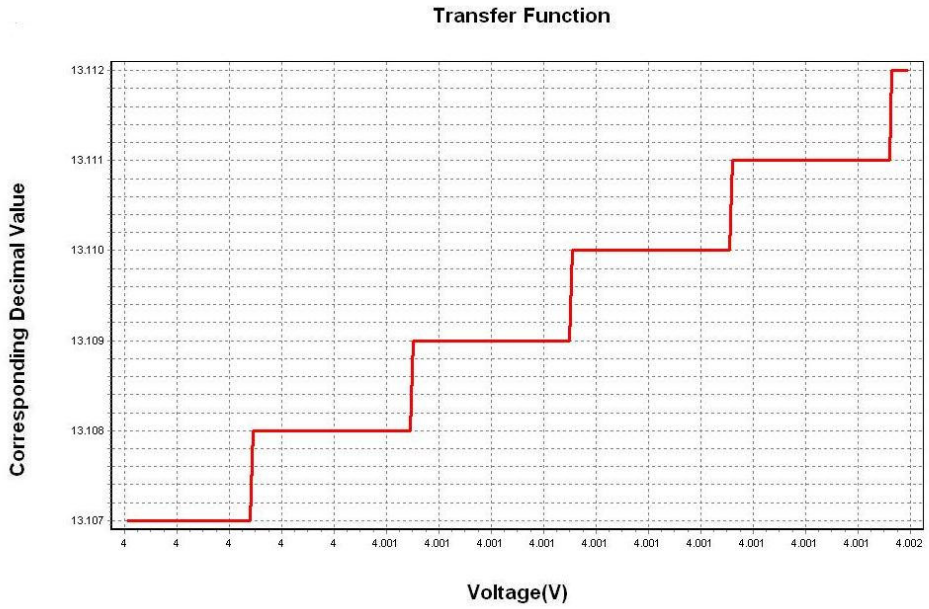
PWM duty cycle = (DCxB9:DCxB0 bits value) • Tosc • (TMR2 prescale value), in units of time (5.2)

5.7 The A/D Converter Module

Main point of verification of A/D converter is simply applying voltage and receiving digital value. First of all, reference voltages Vref+ and Vref- were assigned as 5 V and 0 V respectively. 1.3 V was applied to analog input channel. 0^{th} bit of ADCON turns A/D converter on and 2^{nd} bit expresses the status of converter. If this bit is 1, A/D converter is in progress otherwise not. Conversion of each bit requires at least 1 clock cycle. Since pre-scalar was configured as 1:2, conversion took 28 clock cycles. Expected hexadecimal value for the applied potential is 0x10A3. After conversion A/D interrupt flag was set and `ADCON` register's 2^{nd} bit became low which means A/D controller was turned off. Hence, `ADCON` register's value became 0x1. Simulation of A/D conversion is depicted in Figure 5.16. Also a sweep analysis for A/D converter was performed in order to observe transfer function of A/D converter. It is given in Figure 5.17.



(a)



(b)

Figure 5- 16: A/D converter transfer function (a) -1V / 6V range view (b) Close view of transfer function around 4V. Vref+ is equal to 5V and Vref- is equal to 0.

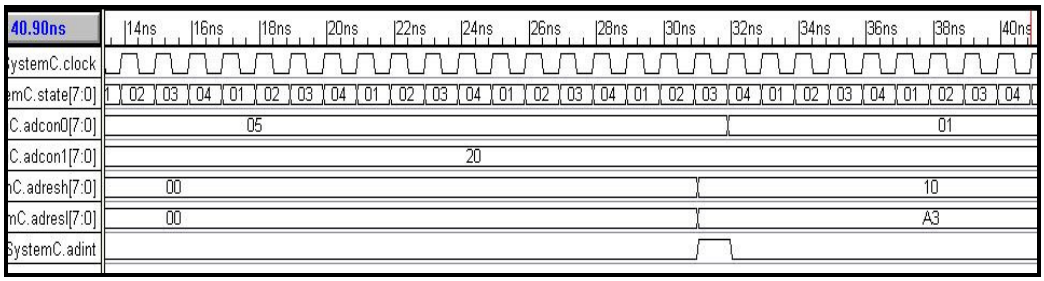


Figure 5- 17:A/D conversion process

5.8 The D/A Converter Module

In order to simulate the D/A converter another code was written that generates a data table of sinus wave. Since it is a 12-bit converter, number of the samples can be 4096 at most. For this simulation, number of the samples was selected as 1000 and main core frequency was configured to 40 MHz. Another factor that should be mentioned is the ideal level shifter.

D/A converters cannot generate negative voltages. As a result, in order to observe a bipolar waveform, a level shifter is attached to the analog output. D/A converter is turned on by means of setting 0^{th} and 1^{st} bits of `DACON0` register. For convenient digital to analog conversion, `DADATL` and `DADATH` registers should be written consecutively. Generated sinus waveform is given in Figure 5.18,.

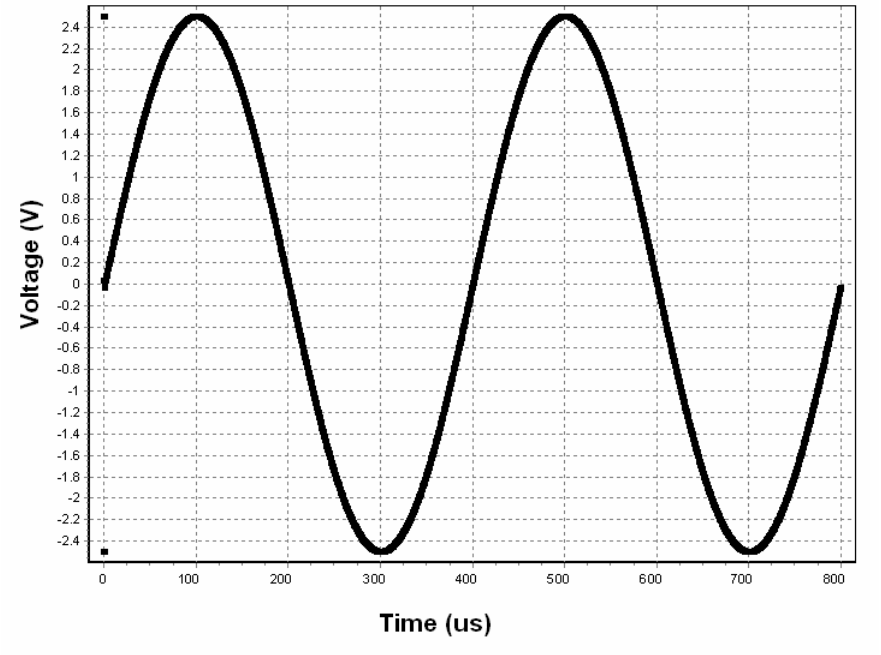


Figure 5- 18: D/A converter sinus signal output

5.9 Interrupts

The 16F871 has several interrupt sources. All peripheral modules generate an interrupt after completing their task. Also microcontroller's PORTB has 4 pins sensitive to changes and one pin is sensitive to rising or falling edges depending on the control bit. All interrupts can be masked using enable/disable bits. If an interrupt source enabled and that interrupt occurs, program counter value will pushed to stack and its value becomes 0x004. This address is the starting address of the interrupt vector table. Interrupt vector table routine must end with a `RETFIE` command. This instruction pops the former content of program counter and microcontroller returns to normal operation. All interrupt sources have equal priority and leads to same result.

As mentioned previously, Port B interrupts awaken the microcontroller from SLEEP. In Figure 5.19, result of an external interrupt can be seen while the 16F871 is in SLEEP mode.

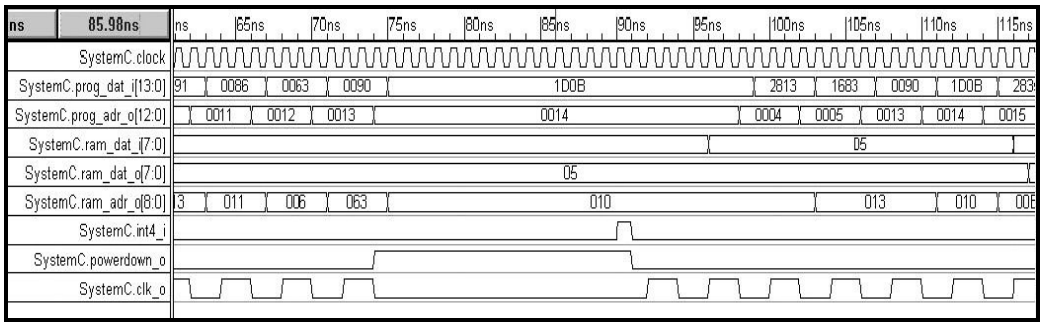


Figure 5- 19: Interrupt and awakening from SLEEP mode.

5.10 CRC Algorithms

Instruction set, special function registers and addressing modes of main core was verified with test assembly codes. However, all test programs are written particularly for this study. Hence, executing other test programs will be beneficial for the verification of the designed microcontroller.

For this purpose, 16-bit CRC code was employed which is downloaded from internet site of Charles Ader that provides several codes for the all microcontrollers [33]. This code is adapted from CRC code released by Dallas Semiconductor [34]. In other words, this CRC code is actually written for 8051 and translated for PIC midrange microcontrollers. Downloaded CRC code is given in Appendix F.

This CRC code loads seed value to working register and calls the CRC subroutine. After each call, subroutine returns 16-bit hexadecimal result whose high byte is saved in `crc_hi` register and low byte is saved in `crc_lo` register. In order to observe these registers they are transferred to Port B and Port C respectively. Expected results of CRC code and obtained simulation results are given in Table 5.1 and Figure 5.20 respectively.

Table 5- 1: Expected results from 16-bit CRC algorithm

Initial value of accumulator	Expected CRC Result
0x80	crc_hi:crc_lo = 00 00
0x75	crc_hi:crc_lo = A0 01
0x8A	crc_hi:crc_lo = 27 A0
0x0B	crc_hi:crc_lo = DF A6
0x75	crc_hi:crc_lo = BD 1E
0xC7	crc_hi:crc_lo = EF FC
0xAA	crc_hi:crc_lo = D3 AE
0x75	crc_hi:crc_lo = C3 D2
0xC7	crc_hi:crc_lo = BA 82
0x55	crc_hi:crc_lo = F3 7B
0x43	crc_hi:crc_lo = 1C 73
0x1C	crc_hi:crc_lo = 14 1C
0x14	crc_hi:crc_lo = 00 00

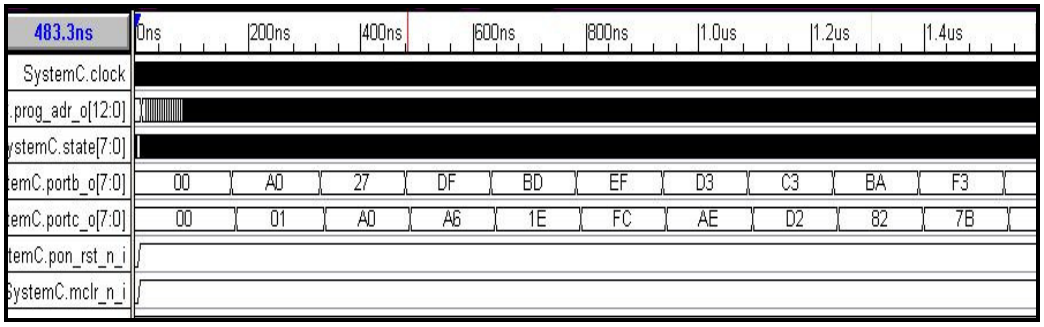


Figure 5- 20: Obtained results from 16-bit CRC verification code

5.11 A Demodulator Example

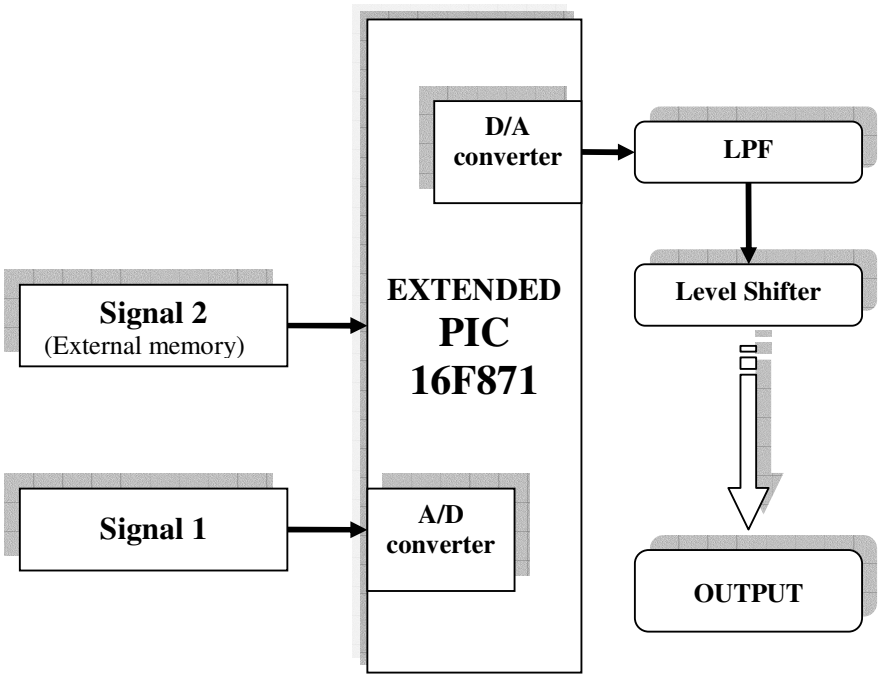


Figure 5- 21: Demodulator setup

In order to demonstrate the system level simulation capabilities of the Systemc-AMS and verify the designed microcontroller, a demodulator set up is constructed. This is a well known synchronous demodulator system. In this model, the PIC 16F871 samples signal 1 via its A/D converter module and reads the external data memory which houses the signal 2, connected through its Port B. Using a multiplication algorithm, the 16F871 multiplies these two discrete signals. Afterwards, it generates resultant continuous time signal via its D/A converter. This multiplied signal carries dc, high and low frequency components. Low pass filter extracts the both dc component and message which is the low frequency signal. Unfortunately, it also attenuates the message signal. Then, level shifter kills the all dc components. Hence, at the output only message signal will be observed.

Frequency of message signal is a cosines signal with 50 Hz frequency while the carrier signal is another cosines signal with 4 KHz frequency.

Signal1: $(1+0.5m(t))\cos w_2 t$ where $m(t) = \cos w_1 t$ (5.3)

Signal2: $\cos(w_2 t + \theta)$ (5.4)

Signal at the output of the level shifter is given in Equation 5.5. Output of message signal is identical with the original message except the magnitude. Results of the level shifter are given in Figure 5.22 for θ is equal to 0°, 30°, 60° and 90° respectively.

$(0.2m(t))\cos\theta$ (5.5)

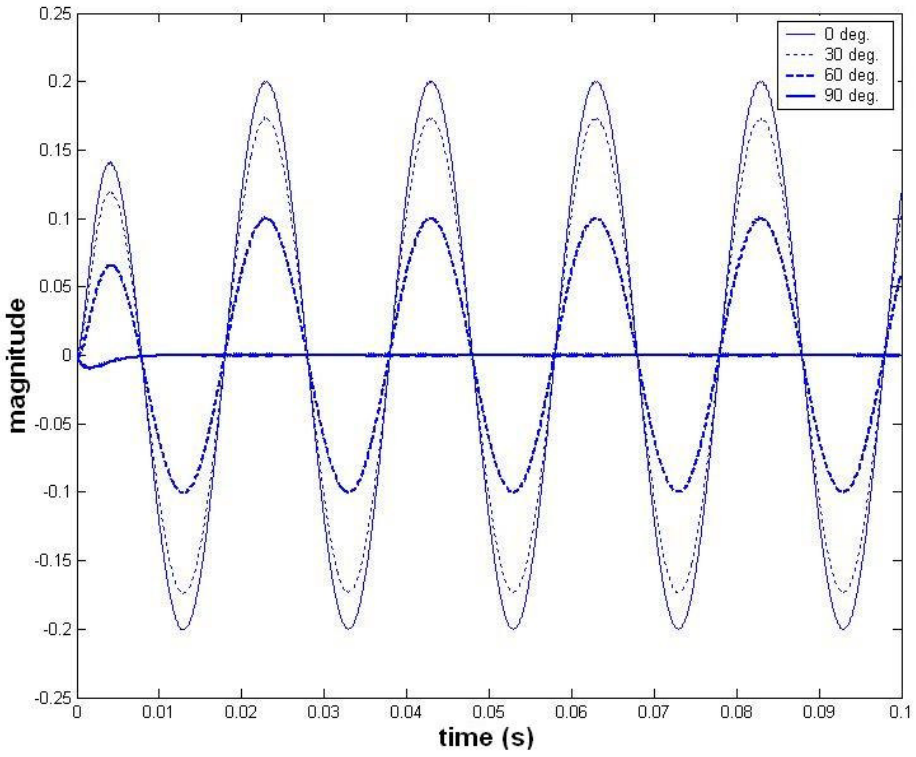


Figure 5- 22: Demodulator output for different phase angles

CHAPTER 6

CONCLUSIONS

In this thesis, an 8-bit microcontroller whose architecture, peripheral modules and instruction set is fully compatible with the PIC 16F871 microcontroller is implemented in SystemC language. Designed microcontroller involves an additional digital to analog converter (D/A) besides the core of the PIC 16F871. The original and the additional analog components of the microcontroller are described behaviorally using analog and mixed signal extensions of the SystemC, named as SystemC-AMS. Since it is rather new approach, before the design procedure, fundamentals of the SystemC-AMS have been discussed. Afterwards, main features of the core are given in details. Finally, verification approach and simulation results of the designed extended PIC 16F871 microcontroller are presented.

The designed microcontroller consists of 8-bit RISC core, parallel slave port, serial port, timers and capture, compare and pulse-width modulation unit. It also involves behaviorally described analog to digital and digital to analog converters. Main core of the microcontroller is a state machine that has five states including reset state. It has 35 instructions and more than forty special function registers. Besides, main core supports two addressing modes namely direct and indirect addressing. Designed peripheral units that are strongly synchronized with the main core such as parallel slave port (PSP), are integrated with main core. However, other modules such as serial port and timers are implemented as independent modules. These peripheral modules communicate with the main core through the data-bus. Utilizing this, new peripheral modules can easily be added to the main core if needed.

In order to verify the design, a particular test environment has been constructed. This testbench consists of several stimulus and display units. TestROM, which is one of the

stimulus units, pretends the ROM memory that houses the hexadecimal form of the assembly code to be executed. Other stimulus units imitate the environments that main core and peripherals interact. On the other hand, display units collect the responses of the main core or peripherals. Utilizing this testbench, several test codes in assembly language are prepared and applied. In particular, verification programs that test instruction set and addressing modes are fixed. Besides, each peripheral module is simulated with its particular verification code based on the assembly codes that are taken from the sources provided by the manufacturer. In addition to the output waveforms, the contents of the internal registers are also tracked. Obtained simulation results are compared with the signal waveforms given in the datasheets. Furthermore, designed main core is also verified using 16-bit CRC algorithm for the PIC midrange family. All simulation results were literally compatible with the expected results.

In this study, assembly codes are transferred to simulation environment through test-ROM module. This module reads the compiled .hex file and generates pseudo ROM array. In this way, different modules and different assembly codes could be simulated without any change on SystemC code. This provides ease of simulation of different software.

The 16F871 core, designed testbenches and verification codes, which have been developed in this study, can also be used as a design and simulation platform for hardware and software projects in future. Especially, designed new analog or mixed signal modules can easily be attached to main core directly or through another analog module such as A/D converter. A synchronous demodulator model including designed core and analog modules is built as an example.

SystemC-AMS is a new concept for system on chip design. Since non-digital and non-electronic components are becoming more common for systems on chip, it is essential to include these components into the same design and simulation environment for easier and flexible modeling. In this way, system domain simulation results can be easily observed for systems that have several digital, analog and non-electrical units.

Simulation time is another vital factor for such design environment that provides hybrid system modeling in unique framework. It is observed that simulation performance of SystemC significantly remains even after addition of analog and mixed signal extensions. Assembly code used for simulating the D/A converter has more than four thousands of lines and simulation duration is only on the order of seconds. This implies that very complex hybrid systems can be simulated within a reasonable time period.

Normally, PIC 16F871 microcontroller doesn't involve a digital to analog converter. However, an additional D/A converter module has been modeled using SystemC-AMS and added to the PIC 16F871 structure. Described D/A converter module can generate analog output up to 12-bit resolution.

In this study, SystemC to hardware flow is also demonstrated with synthesis of the Arithmetic logic unit of the 16F871. Synthesis flow begins with conversion of the SystemC code to VHDL code and ends with synthesizing generated VHDL code using the FPGA synthesis tool. In this study, as the tool for the SystemC to VHDL conversion, the trail version of *SystemCrafter 2.0* is employed. This tool is currently devoid of high level synthesis capabilities. Therefore, rest of the design cannot be synthesized due to the limitations of the SystemCrafter. It also does not support combinational logic systems. For this reason, a register is added to output of the arithmetic logic unit. With this small modification, SystemC to VHDL conversion of arithmetic logic unit is achieved without a glitch. Generated code is a gate level VHDL code and ready for synthesis. However, this VHDL code can not be modified by user after conversion since it is very low level and complex. Corrections should be performed on the SystemC code and whole procedure should be repeated.

After conversion of the SystemC code to VHDL, *Xilinx ISE 8.1i* is employed for the synthesis operation and synthesized arithmetic logic unit is embedded into XC3S200 FPGA. Mapped logic uses 168 slices which corresponds the 8% of the available slices and maximum clock frequency for the synthesized arithmetic logic unit is 73.556 MHz. Although the clock frequency achieved in this study is found almost twice that of the commercial controllers, synthesized VHDL code is poor in terms of the chip area. This is caused by the low performance of the SystemC to VHDL synthesis tool *SystemCrafter*.

REFERENCES

1. A. Vachoux, C. Grimm, K. Einwich, " *Extending SystemC to Support Mixed Discrete-Continuous System Modeling and Simulation"* , IEEE, 2005
2. E. Christien, K. Bakalar, " *VHDL-AMS, A Hardware description language for analog and mixed signal applications*", Transactions on circuits and systems II, Vol 46, No 10, 1999
3. www.dynasim.com
4. www.mathworks.com
5. A.Vachoux, C. Grimm, K. Einwich, "*SystemC-AMS Requirements, Design objectives and rationales"*, Proc. 2003 Design Automation and Test in Europe (DATE 2003), Munich, Germany, 2003.
6. T. Grötker *et al.*, "*System Design with SystemC"*, Kluwer,2002.
7. Vachoux, C. Grimm, K. Einwich, *"Analog and Mixed Signal Modeling with SystemC-AMS",* IEEE ISCAS'03 Conference, May 2003.
8. K. Einwich, P. Schwarz, Ch. Grimm, K. Waldschimidt, *" Mixed Signal Extensions for SystemC",* FDL'02, Marseille, France.
9. *SystemC-AMS Version 0.15.* Fraunhofer Institut für Integrierte Schaltungen, Dresden, 2006. Presentation slides.
10. K. Kundert, "Simulation Methods for RF Integrated Circuits", Proc. IEEE ICCAD'97, pp. 752-765, 1997.
11. P. Lieverse *et al.*, "*A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems*" Proc. IEEE Workshop on Signal Processing Systems, pp. 181-190, 1999.
12. G. Vanderspeeten *et al.*, "A methodology for efficient high-level dataflow simulation of mixed-signal front-ends of digital telecom transceivers",Proc. IEEE DAC 2000, pp. 440-445, 2000.

13. A.Varma, M.Y. Afridi, A. Akturk, P. Klein, A. R. Hefner, B. Jacob, " *Modeling heterogeneous SoCs with SystemC: A Digital/MEMS case study*", CASES'06 ,2006, Seoul, Korea.
14. E. Markert, M. Dienel, G. Herrmann, D. Mueller, U. Heinkel, " *Modeling of a new 2D acceleration sensor array using SystemC-AMS*", International MEMS conference 2006.
15. M. Alassir, J. Denoulet, O. Romain & P. Garda," *A SystemC-AMS model of an I2C bus controller*", IEEE, 2006.
16. Synthesis Working Group of OSCI (2004). *SystemC Synthesizable Subset*. Website: http://www.SystemC.org
17. SystemCrafter Inc (2005). *SystemCrafter SC User Manual Version 2.0.0.3*. Website: http://www.systemcrafter.com
18. Synopsys (2001). *Synopsys CoCentric SystemC Compiler: RTL User and Modeling Guideline*.
19. Prosilog Inc. (2002). *Prosilog's SystemC Compiler Datasheet.*
20. Xilinx (2006). *XST User Guide 8.1i*. Website: http://www.xilinx.com
21. K. Einwich, C. Grimm, A. Vachoux, "*SystemC-AMS Reference manual, version 1.0*." (2005)
22. T. Mahne, *SystemC-AMS Proposal of naming scheme and extensions for the Linear network element models* (2005)
23. *PIC 16F870/871 Data sheet*, Microchip inc.
24. *PIC micro Mid-Range MCU family Reference manual*, Microchip inc.
25. Stan D'Souza, *Implementing a table read, AN556*, Microchip inc.
26. *Parallel Slave Port reference manual*, Microchip inc.
27. *Timer 1 reference manual,* Microchip inc.
28. *Timer 2 reference manual,* Microchip inc.
29. *A/D converter reference manual*, Microchip inc.
30. Kesen, L. (2004) *Implementation of an 8-bit Microcontroller with SystemC*, Ankara, Middle East Technical University in The Department of Electrical and Electronics Engineering
31. *CCP reference manual*, Microchip inc.
32. *USART reference manual,* Microchip inc.
33. http://www.piclist.com/techref/scenix/lib/io/osi2/crc16ca_sx.htm

34. *Understanding and using cyclic redundancy checks with Dallas Semiconductor iButton products*, *Application note 27*, Dallas Semiconductor. Website: http://www.maxim-ic.com

APPENDIX A

THE PIC 16F871 INSTRUCTION SET SUMMARY

Table A- 1: Descriptions of (a) ADDLW (b) ADDWF (c) ANDLW (d) ANDWF instructions.

ADDLW	
Syntax:	ADDLW k
Operands:	0 ≤ k ≤ 255
Operation:	(W)+k →(W)
Status Affected :	C, DC, Z
Description:	The contents of w register are added to 8 bit literal 'k' and result is placed in w register.

(a)

ADDWF	
Syntax:	ADDWF f, d
Operands:	0 ≤ f ≤ 127 d ϵ [0,1]
Operation:	(W)+f →(destination)
Status Affected :	C, DC, Z
Description:	Add the contents of w register with register 'f'. If d is '0' result is stored in w register. If d is '1' result is stored back in 'f' register.

(b)

ANDLW	
Syntax:	ANDLW k
Operands:	0 ≤ k ≤ 255
Operation:	(W).AND.(k) →(W)
Status Affected :	Z
Description:	The contents of w register are AND'ed with 8 bit literal 'k' and result is placed in w register.

(c)

ANDWF	
Syntax:	ADDWF f, d
Operands:	0 ≤ f ≤ 127 d ϵ [0,1]
Operation:	(W).AND.(k) →(destination)
Status Affected :	Z
Description:	AND the w register with register 'f'. If d is '0' result is stored in w register. If d is '1' result is stored back in 'f' register.

(d)

Table A- 2 :Descriptions of (a) BCF (b) BTFSS (c) BSF (d) ANDWF instructions.

BCF	
Syntax:	`BCF f, d`
Operands:	`0 ≤ f ≤ 127` `0 ≤ b ≤ 7`
Operation:	`0 → f<b>`
Status Affected :	None
Description:	Bit 'b' in register 'f' is cleared

(a)

BTFSS	
Syntax:	`BTFSS f, b`
Operands:	`0 ≤ f ≤ 127` `0 ≤ b ≤ 7`
Operation:	`Skip if f<b> = 1`
Status Affected :	None
Description:	If bit 'b' in register 'f' is '0', the next instruction is executed. If bit 'b' is 1, then next instruction is discarded and NOP is executed instead, making this is $2T_{CY}$ instruction.

(b)

BSF	
Syntax:	`BSF f, d`
Operands:	`0 ≤ f ≤ 127` `0 ≤ b ≤ 7`
Operation:	`1 → f<b>`
Status Affected :	None
Description:	Bit 'b' in register 'f' is cleared

(c)

BTFSC	
Syntax:	`BTFSS f, b`
Operands:	`0 ≤ f ≤ 127` `0 ≤ b ≤ 7`
Operation:	`Skip if f<b> = 1`
Status Affected :	None
Description:	If bit 'b' in register 'f' is '1', the next instruction is executed. If bit 'b' is 0, then next instruction is discarded and NOP is executed instead, making this is $2T_{CY}$ instruction.

(d)

Table A- 3 :Descriptions of (a) CALL (b) CLRWDT (c) CLRF (d) CLRW instructions.

CALL	
Syntax:	CALL k
Operands:	0 ≤ k ≤ 2047
Operation:	PC+1 → T_{os} k → PC<10:0> PCLATH<5:4> → PC<12:11>
Status Affected :	None
Description:	Call subroutine. First, return address is pushed (PC+1) onto stack. The eleven bit immediate address is loaded into PC bits <10:0>. The upper bits of PC are loaded from PCLATH. Call is a 2 cycle instruction

(a)

CLRWDT	
Syntax:	CLRWDT
Operands:	None
Operation:	0x00 → WDT 0 → WDT pre-scalar 1 → TO 1 → PD
Status Affected :	TO, PD
Description:	CLRWDT resets the Watch-dog timer. It also resets the WDT pre-scalar. TO and PD bits are set.

(b)

CLRF	
Syntax:	CLRF f
Operands:	0 ≤ f ≤ 127
Operation:	0 → (f) 1 → Z
Status Affected :	Z
Description:	The contents of 'f' register are cleared and Z bit is set.

(c)

CLRW	
Syntax:	CLRW
Operands:	None
Operation:	0 → (W) 1 → Z
Status Affected :	None
Description:	W register is cleared and zero bit (Z) is set

(d)

Table A- 4: Descriptions of (a) COMF (b) SLEEP (c) DECF (d) DECFSZ instructions.

COMF	
Syntax:	`COMF f, d`
Operands:	$0 \leq f \leq 127$ $d \in [0,1]$
Operation:	$(\overline{f}) \rightarrow$(destination)
Status Affected :	Z
Description:	The contents of register 'f' are complemented. If d is '0' result is stored in w register. If d is '1' result is stored back in 'f' register.

(a)

SLEEP	
Syntax:	`SLEEP`
Operands:	None
Operation:	0 → WDT 0 → WDT pre-scalar 1 → TO 0 → PD
Status Affected :	TO, PD
Description:	The processor is put in the SLEEP mode with oscillator stopped.

(b)

DECF	
Syntax:	`DECF f, d`
Operands:	$0 \leq f \leq 127$ $d \in [0,1]$
Operation:	(f) − 1 →(destination)
Status Affected :	Z
Description:	Decrement register 'f'. If d is '0' result is stored in w register. If d is '1' result is stored back in 'f' register.

(c)

DECFSZ	
Syntax:	`ADDWF f, d`
Operands:	$0 \leq f \leq 127$ $d \in [0,1]$
Operation:	(f) − 1 →(destination) skip if result = 0
Status Affected :	None
Description:	The contents of register 'f' decremented. If d is '0' result is stored in w register. If d is '1' result is stored back in 'f' register. If the result is '1', the next instruction executed. If the result is '0' then NOP executed instead making it $2T_{CY}$.

(d)

Table A- 5: Descriptions of (a) INCF (b) INCFSZ (c) GOTO (d) IORLW instructions.

INCF	
Syntax:	INCF f, d
Operands:	0 ≤ f ≤ 127 d ∈ [0,1]
Operation:	(f) + 1 →(destination)
Status Affected :	Z
Description:	The contents of 'f' register are incremented. If d is '0' result is stored in w register. If d is '1' result is stored back in 'f' register.

(a)

INCFSZ	
Syntax:	INCFSZ f, d
Operands:	0 ≤ f ≤ 127 d ∈ [0,1]
Operation:	(f) + 1 →(destination) skip if result = 0
Status Affected :	None
Description:	The contents of register 'f' incremented. If d is '0' result is stored in w register. If d is '1' result is stored back in 'f' register. If the result is '1', the next instruction executed. If the result is '0' then NOP executed instead making it $2T_{CY}$.

(b)

GOTO	
Syntax:	GOTO k
Operands:	0 ≤ k ≤ 2047
Operation:	k → PC<10:0> PCLATH<4:3> → PC<12:11>
Status Affected :	None
Description:	GOTO is unconditional branch. The eleven bit immediate value is loaded into PC bits<10:0>. The upper bits of PC are loaded from PCLATH<4:3>. GOTO is a two cycle instruction.

(c)

IORLW	
Syntax:	ADDWF f, d
Operands:	0 ≤ k ≤ 255
Operation:	(W).OR.(k) →(destination)
Status Affected :	Z
Description:	The contents of the w register are OR'ed with 8 bit literal 'k'. Result is placed back in w register.

(d)

Table A- 6: Descriptions of (a) IORWF (b) MOVLW (c) MOVF (d) MOVWF instructions.

IORWF	
Syntax:	IORWF f, d
Operands:	0 ≤ f ≤ 127 d ∈ [0,1]
Operation:	(W).OR.(f) →(destination)
Status Affected :	Z
Description:	Inclusive or the w register with register 'f'. If d is '0' result is stored in w register. If d is '1' result is stored back in 'f' register.

(a)

MOVLW	
Syntax:	MOVLW f, d
Operands:	0 ≤ k ≤ 255
Operation:	k →(W)
Status Affected :	None
Description:	8 bit literal is loaded to w register.

(b)

MOVF	
Syntax:	MOVF f, d
Operands:	0 ≤ f ≤ 127 d ∈ [0,1]
Operation:	(f) →(destination)
Status Affected :	Z
Description:	The contents of register 'f' are moved to destination depending upon 'd'. If d=0destination is w register. If d=1 destination file register f itself. D=1 is useful to test a file register since Z is affected.

(c)

MOVWF	
Syntax:	MOVWF f
Operands:	0 ≤ f ≤ 127
Operation:	(W) →(f)
Status Affected :	None
Description:	The contents of w register is moved to register 'f'

(d)

Table A- 7: Descriptions of (a) NOP (b) RETLW (c) RETFIE (d) RETURN instructions.

NOP	
Syntax:	NOP
Operands:	None
Operation:	No operation
Status Affected :	None
Description:	No operation

(a)

RETLW	
Syntax:	RETLW k
Operands:	0 ≤ k ≤ 255
Operation:	(k) → (W) TOS → PC
Status Affected :	NONE
Description:	The w register is loaded with 8 bit literal 'k'. The program counter is loaded from top of the stack. This is 2 cycle instruction.

(b)

RETFIE	
Syntax:	RETFIE
Operands:	None
Operation:	TOS → (PC) 1 → GIE
Status Affected :	None
Description:	Return from interrupt

(c)

RETURN	
Syntax:	RETURN
Operands:	None
Operation:	TOS → PC
Status Affected :	None
Description:	Return from subroutine. The stack is POP'ed and TOP of to stack is loaded into PC. This is a 2 cycle instruction.

(d)

Table A- 8: Descriptions of (a) RLF(b) SUBLW (c) RRF (d) SUBWF instructions.

RLF	
Syntax:	`RLF f, d`
Operands:	`0 ≤ f ≤ 127` `d ∈ [0,1]`
Operation:	
Status Affected :	C
Description:	The contents of 'f' register are rotated one bit left through the carry flag. If 'd' is '0' , result is placed in w register. If 'd' is 1 result is placed back in register 'f'

(a)

SUBLW	
Syntax:	`SUBLW k`
Operands:	`0 ≤ k ≤ 255`
Operation:	`k - (W) →(W)`
Status Affected :	C, DC, Z
Description:	The W register is subtracted(2's complement method) from the 8 bit literal. 'k'. The result is placed back into W register.

(b)

RRF	
Syntax:	`RRF f, d`
Operands:	`0 ≤ f ≤ 127` `d ∈ [0,1]`
Operation:	
Status Affected :	C
Description:	The contents of 'f' register are rotated one bit right through the carry flag. If 'd' is '0' , result is placed in w register. If 'd' is 1 result is placed back in register 'f'

(c)

SUBWF	
Syntax:	`SUBWF f, d`
Operands:	`0 ≤ f ≤ 127` `d ∈ [0,1]`
Operation:	`f - (W) →(destination)`
Status Affected :	C, DC, Z
Description:	Subtract (2's complement method) w register from 'f'. If 'd' is '0' , result is placed in w register. If 'd' is 1 result is placed back in register 'f'

(d)

Table A- 9: Descriptions of (a) XORWF (b) XORLW (c) SWAPF instructions

XORWF	
Syntax:	`XORWF f, d`
Operands:	`0 ≤ f ≤ 127` `d ∈ [0,1]`
Operation:	`(W).XOR.(k) →(destination)`
Status Affected :	Z
Description:	Exclusive OR the w register with register 'f'. If d is '0' result is stored in w register. If d is '1' result is stored back in 'f' register.

(a)

XORLW	
Syntax:	`XORLW k`
Operands:	`0 ≤ k ≤ 255`
Operation:	`(W).XOR.(k) →(W)`
Status Affected :	Z
Description:	The contents of w register are XOR'ed with 8 bit literal 'k' and result is placed in w register.

(b)

SWAPF	
Syntax:	`SWAPF f, d`
Operands:	`0 ≤ f ≤ 127` `d ∈ [0,1]`
Operation:	`(f<3:0>)→(destination<7:4>)` `(f<7:4>)→(destination<3:0>)`
Status Affected :	None
Description:	The upper and lower nibbles of 'f' register is exchanged. If 'd' is '0' , result is placed in w register. If 'd' is 1 result is placed back in register '

(c)

APPENDIX B

ASSEMBLY VERIFICATION CODES

B.1 Instruction Set Verification Code

```
LIST            P=PIC16F871
INCLUDE         P16F871.INC
__CONFIG _HS_OSC & _WDT_OFF & _PWRTE_ON & _CP_OFF
ERRORLEVEL      -302    ;ELIMINATE BANK WARNING

ORG 0X0
       GOTO LBL_1   ; LBL_1 = 0X5
ORG 0X4
       GOTO SON     ; LBL_2 = 0X13

LBL_1
       BSF STATUS,RP1      ; SWITCH TO BANK 2
       MOVLW 0xD7   ; D'215' B'11010111'
       MOVWF OPTION_REG
       MOVLW 0xB8   ; D'184' B'10111000'
       MOVWF INTCON
       BCF STATUS,RP0      ; SWITCH TO BANK 0

;////////////////////////////////// ADDLW
       MOVLW 0X15
       ADDLW 0X10
       SUBLW 0X25
       BTFSC STATUS,Z
       GOTO DONE1
       MOVLW D'1'
       GOTO FAIL

;////////////////////////////////// ADDWF
DONE1
       MOVLW 0X15
```

```
       MOVWF EEDATA
       MOVLW 0X10
       ADDWF EEDATA,W
       SUBLW 0X25
       BTFSC STATUS,Z
       GOTO DONE2
       MOVLW D'2'
       GOTO FAIL
;////////////////////////////////////ANDLW
DONE2
       MOVLW 0X75
       ANDLW 0XFF
       SUBLW 0X75
       BTFSC STATUS,Z
       GOTO DONE3
       MOVLW D'3'
       GOTO FAIL
;///////////////////////////////////ANDWF
DONE3
       MOVLW 0X30
       MOVWF EEDATA
       MOVLW 0XFF
       ANDWF EEDATA,W
       SUBLW 0X30
       BTFSC STATUS,Z
       GOTO DONE4
       MOVLW D'4'
       GOTO FAIL
;////////////////////////////////////IORLW
DONE4
       MOVLW 0X9A
       IORLW 0X35
    SUBLW 0XBF
       BTFSC STATUS,Z
       GOTO DONE5
       MOVLW D'5'
       GOTO FAIL
;////////////////////////////////////IORWF
DONE5
       MOVLW 0X13
       MOVWF EEDATA
       MOVLW 0X91
       IORWF EEDATA,W
```

```
       SUBLW 0X93
       BTFSC STATUS,Z
       GOTO DONE6
       MOVLW D'6'
       GOTO FAIL
;////////////////////////////////////XORLW
DONE6
       MOVLW 0XB5
       XORLW 0XAF
       SUBLW 0X1A
       BTFSC STATUS,Z
       GOTO DONE7
       MOVLW D'7'
       GOTO FAIL
;////////////////////////////////////XORWF
DONE7
       MOVLW 0XAF
       MOVWF EEDATA
       MOVLW 0XB5
       XORWF EEDATA,1
       MOVF  EEDATA,W
       SUBLW 0X1A
       BTFSC STATUS,Z
       GOTO DONE8
       MOVLW D'8'
       GOTO FAIL
;////////////////////////////////////////SWAPF
DONE8
       MOVLW 0XA5
       MOVWF EEDATA
       SWAPF EEDATA,1
       MOVF  EEDATA,W
       SUBLW 0X5A
       BTFSC STATUS,Z
       GOTO DONE9
       MOVLW D'9'
       GOTO FAIL
;////////////////////////////////////////RLF
DONE9
       MOVLW 0
       ADDLW 0   ; clear carry flag
       MOVLW B'11100110'
       MOVWF EEDATA
```

```
       RLF  EEDATA,W
       SUBLW B'11001100'
       BTFSC STATUS,Z
       GOTO DONE10
       MOVLW D'10'
       GOTO FAIL
;//////////////////////////////////////RRF
DONE10
       MOVLW 0
       ADDLW 0   ; clear carry flag
       MOVLW B'11100110'
       MOVWF EEDATA
       RRF EEDATA,W
       SUBLW B'01110011'
       BTFSC STATUS,Z
       GOTO DONE11
       MOVLW D'11'
       GOTO FAIL
;//////////////////////////////////////COMF
DONE11
       MOVLW 0X13
       MOVWF EEDATA
       COMF EEDATA,W
       SUBLW 0XEC
       BTFSC STATUS,Z
       GOTO DONE12
       MOVLW D'12'
       GOTO FAIL
;//////////////////////////////////////INCF
DONE12
       MOVLW 0XFF
       MOVWF EEDATA
       INCF EEDATA,W
       BTFSC STATUS,Z
       GOTO DONE13
       MOVLW D'13'
       GOTO FAIL
;//////////////////////////////////////DECF
DONE13
       MOVLW 0X1
       MOVWF EEDATA
       DECF EEDATA,W
       BTFSC STATUS,Z
```

```
       GOTO DONE14
       MOVLW D'14'
       GOTO FAIL
;////////////////////////////////////////SUBLW
DONE14
       MOVLW 0X10
       SUBLW 0X10
       BTFSC STATUS,Z
       GOTO DONE15
       MOVLW D'15'
       GOTO FAIL
;////////////////////////////////////////SUBWF
DONE15
       MOVLW 0X20
       MOVWF EEDATA
       SUBWF EEDATA,W
       BTFSC STATUS,Z
       GOTO DONE16
       MOVLW D'16'
       GOTO FAIL
;////////////////////////////////////////CLRW
DONE16
       MOVLW 0X20
       CLRW
       BTFSC STATUS,Z
       GOTO DONE17
       MOVLW D'17'
       GOTO FAIL
;////////////////////////////////////////CLRF
DONE17
       MOVLW 0X50
       MOVWF EEDATA
       CLRF EEDATA
       BTFSC STATUS,Z
       GOTO DONE18
       MOVLW D'18'
       GOTO FAIL
;////////////////////////////////////////BCF
DONE18
       MOVLW 0XC7
       MOVWF EEDATA
       BCF EEDATA,7
       MOVF EEDATA,W
```

```
       SUBLW 0X47
       BTFSC STATUS,Z
       GOTO DONE19
       MOVLW D'19'
       GOTO FAIL
;//////////////////////////////////////////BSF
DONE19
       MOVLW 0X0A
       MOVWF EEDATA
       BSF EEDATA,7
       MOVF EEDATA,W
       SUBLW 0X8A
       BTFSC STATUS,Z
       GOTO DONE20
       MOVLW D'20'
       GOTO FAIL
;//////////////////////////////////////////MOVLW
DONE20
       MOVLW 0X30
       SUBLW 0X30
       BTFSC STATUS,Z
       GOTO DONE21
       MOVLW D'21'
       GOTO FAIL
;//////////////////////////////////////////MOVWF
DONE21
       MOVLW 0X50
       MOVWF EEDATA
       SUBWF EEDATA,W
       BTFSC STATUS,Z
       GOTO DONE22
       MOVLW D'22'
       GOTO FAIL
;//////////////////////////////////////////MOVF
DONE22
       MOVLW 0X65
       MOVWF EEDATA
       MOVLW 0X30
       MOVF EEDATA,W
       SUBLW 0X65
       BTFSC STATUS,Z
       GOTO DONE23
       MOVLW D'23'
```

```
       GOTO FAIL
;//////////////////////////////////////////BTFSC
DONE23
       MOVLW 0X7
       MOVWF EEDATA
       BTFSC EEDATA,3
       GOTO  ERROR1
       MOVLW 0X8
       MOVWF EEDATA
       BTFSC EEDATA,3
       GOTO DONE24
       GOTO ERROR1
ERROR1
       MOVLW D'24'
       GOTO FAIL
;//////////////////////////////////////////BTFSS
DONE24
       MOVLW 0X8
       MOVWF EEDATA
       BTFSS EEDATA,3
       GOTO ERROR2
       MOVLW 0X7
       MOVWF EEDATA
       BTFSS EEDATA,3
       GOTO DONE25
       GOTO ERROR2
ERROR2
       MOVLW D'25'
       GOTO FAIL
;//////////////////////////////////////////INCFSZ
DONE25
       MOVLW 0XFF
       MOVWF EEDATA
       INCFSZ EEDATA
       GOTO ERROR3
       MOVLW 0XAA
       MOVWF EEDATA
       INCFSZ EEDATA
       GOTO DONE26
       GOTO ERROR3
ERROR3
       MOVLW D'26'
       GOTO FAIL
```

```
;////////////////////////////////////////////DECFSZ
DONE26
       MOVLW 0X1
       MOVWF EEDATA
       DECFSZ EEDATA
       GOTO ERROR4
       MOVLW 0XAA
       DECFSZ EEDATA
       GOTO DONE27
       GOTO ERROR4
ERROR4
       MOVLW D'27'
       GOTO FAIL
;////////////////////////////////////////////GOTO
DONE27
       GOTO DONE28
       MOVLW D'28'
       GOTO FAIL
;////////////////////////////////////////////CALL+RETURN
DONE28
       MOVLW 0XAA
       CALL C1  ;// 0XBB VALUE WİLL BE ASSIGNED
       SUBLW 0XBB
       BTFSC STATUS,Z
       GOTO DONE29
       MOVLW D'29'
       GOTO FAIL
;////////////////////////////////////////////CALL+RETLW
DONE29
       MOVLW 0XAA
       CALL C2 ;// 0XBB VALUE WILL BE ASSIGNED
       SUBLW 0XBB
       BTFSC STATUS,Z
       GOTO DONE
       MOVLW D'30'
       GOTO FAIL

C1
       MOVLW 0XBB
       RETURN

C2:
       RETLW 0XBB
```

```
DONE

       MOVLW 0XFF
       MOVWF PORTB
       GOTO SON

FAIL
       MOVWF PORTB
SON
       END
```

B.2 Addressing Modes Verification Codes

```
LIST            P=PIC16F871
#INCLUDE         "P16F871.INC"
__CONFIG _HS_OSC & _WDT_OFF & _PWRTE_ON & _CP_OFF
ERRORLEVEL      -302    ;ELIMINATE BANK WARNING

ORG 0X0
       GOTO LBL_1   ; LBL_1 = 0X5
LBL_1
       MOVLW 0xAA    ;////////////////////////// DIRECT ADDRESSING
       MOVWF 0x7F
      BSF STATUS, RP0 ; Bank1
       MOVLW 0xBB
       MOVWF 0x7F
       BCF STATUS, RP0 ;
       BSF STATUS, RP1 ; Bank2
       MOVLW 0xCC
       MOVWF 0x7F
       BSF STATUS, RP0 ; Bank3
       MOVLW 0xDD
       MOVWF 0x7F
        BCF STATUS, RP0 ; Bank0
       BCF STATUS, RP1 ; Bank0
       MOVF 0x7F,W
       MOVWF PORTB
       BSF STATUS, RP0 ; Bank1
```

```
 MOVF 0x7F,W
 BCF STATUS, RP0 ; Bank0
 BCF STATUS, RP1 ; Bank0
 MOVWF PORTB
 BCF STATUS, RP0 ;
 BSF STATUS, RP1 ; Bank2
 MOVF 0x7F,W
 MOVWF PORTB
 BSF STATUS, RP0 ; Bank3
 MOVF 0x7F,W
 BCF STATUS, RP0 ; Bank0
 BCF STATUS, RP1 ; Bank0
 MOVWF PORTB

 ;////////////////////////////// INDIRECT ADDRESSING
MOVLW 0x7F
 MOVWF FSR
 MOVLW 0x11
 MOVWF INDF
 MOVLW 0xFF
 MOVWF FSR
 MOVLW 0x22
 MOVWF INDF
  BSF STATUS, IRP ; Bank2,3
 MOVLW 0x7F
 MOVWF FSR
 MOVLW 0x33
 MOVWF INDF
 MOVLW 0xFF
 MOVWF FSR
 MOVLW 0x44
 MOVWF INDF
  BCF STATUS, RP0 ; Bank0
 BCF STATUS, RP1 ; Bank0
 MOVF 0x7F,W
 MOVWF PORTB
 BSF STATUS, RP0 ; Bank1
 MOVF 0x7F,W
 BCF STATUS, RP0 ; Bank0
 BCF STATUS, RP1 ; Bank0
 MOVWF PORTB
 BCF STATUS, RP0 ;
 BSF STATUS, RP1 ; Bank2
```

```
MOVF 0x7F,W
MOVWF PORTB
BSF STATUS, RP0 ; Bank3
MOVF 0x7F,W
BCF STATUS, RP0 ; Bank0
BCF STATUS, RP1 ; Bank0
MOVWF PORTB

END
```

APPENDIX C

SYSTEMC-AMS MODEL DESCRIPTIONS

C.1 Analog Linear Behavioral Models

Transfer function: General form of the transfer function is given as below. Numerator and denominator coefficients are described using vectors. Transfer function is described with sca_ltf_nd class.

$$H(s) = \frac{b_n.s^n + b_{n-1}.s^{n-1} + ... + b_0}{a_m.s^m + a_{m-1}.s^{m-1} + ... + a_0} \quad \text{(C.1)}$$

```
sca_ltf_nd    ltf1;       //decleration
out  =  ltf1(NUM,DEN,input);
```

Zero-pole: Zero pole description is another form of the s domain transfer functions which is the ratio of the two factored polynomials. Necessary coefficients are roots of the denominator, roots of the numerator and a coefficient.

$$H(s) = k.\frac{(s - z_0).(s - z_1).....(s - z_n)}{(s - p_0).(s - p_1).....(s - p_n)} \quad \text{(C.2)}$$

```
sca_ltf_zp   ltf2;        //decleration
out =ltf2 (Z,P,K,input);
```

State-space: SystemC-AMS supports state-space descriptions. This model can be used with sca_ss class. In this model coefficients are given in matrix form.

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du \qquad \text{(C.3)}$$

```
sca_ss    stspc1;    //decleration
out = stspc1(A,B,C,D,input) ;
```

C.2 Frequency Domain Specification Functions

- **Frequency functions:** It returns the current frequency

```
sca_ac_freq()
```

- **Fundamental operators:** Used to describe z and s in frequency domain. Letter n represents the oversampling rate whose default value is 1.

```
sca_ac_z(period,n)
sca_ac_s(n)
```

- **Linear description functions:** These functions employed for linear descriptions in frequency domain. They are similar to ones in the time domain.

```
sca_ac_ltf_nd (NUM,DEN,input)
sca_ac_ltf_zp(Z,P,K,input)
sca_ac_ss (A,B,C,D,input)
```

- **Simulation functions:** This function starts the frequency domain simulation and specifies the simulation frequencies. Simulation frequency range can be linearly or logarithmically divided to n points.

```
Sca_ac_domain_simulate(startf, endf, npoints, {SCA_LIN | SCA_LOG } )
```

C.3 SystemC-AMS Linear Electrical Library Elements

1.Resistor :

Model name: sca_r

Parameters: value=0 / Ω

Ports: p, n

2.Inductor

Model name: sca_l

Parameters: value=1 / H

Ports: p,n

3.Capacitor

Model name: sca_c

Parameters: value=1 / F

Ports: p,n

4.Voltage controlled voltage source

Model name: sca_vcvs

Parameters: value=1 / gain

Ports: ncp - positive control terminal , ncn - negative control terminal , np electrical - positive terminal of source , nn -negative terminal of source

5.Voltage controlled current source

Model name: sca_vccs

Parameters: value=1 / transconductance

Ports: ncp - positive control terminal , ncn -negative control terminal, np - positive terminal of source, nn - negative terminal of source

Note: current =value · voltage ctrl

6.Current controlled voltage source

Model name: sca_l

Parameters: value=1 / transresistance

Ports: ncp -positive control terminal , ncn- negative control terminal, np - positive terminal of source , nn - negative terminal of source .

7.Current controlled current source

Model name: sca_cccs

Parameters: value=1 / current gain

Ports: ncp - positive control terminal , ncn - negative control terminal, np - positive terminal of source , nn - negative terminal of source

Note: current = value · current ctrl

8. Nullor (nullator - norator pair)

Model name: sca_nullor

Parameters: -

Ports: nip -positive terminal of nullator, nin - negative terminal of nullator , nop - positive terminal of norator, non - negative terminal of norator

9. Gyrator

Model name: sca_gyrator

Parameters: g1=1 / gyration conductance in S , g2=1 / gyration conductance in S

Ports: p1 -positive terminal of primary port , n1- negative terminal of primary port, p2 -positive terminal of secondary port , n2 -negative terminal of secondary port

10. Voltage source driven by SDF signal

Model name: sca_sdf2v

Parameters: nominal_voltage gain=1 / voltage scaling factor

Ports: p-positive terminal , n - negative terminal , ctrl -static dataflow control input

Note: voltage = nominal voltage + gain· sdf in

11. Current source driven by SDF signal

Model name: sca_sdf2i

Parameters: gain=1 /current scaling factor, nominal current

Ports: p -positive terminal, n - negative terminal , ctrl - static dataflow control input

Note: current = nominal current + gain · sdf in

12. Variable resistor controlled by SDF signal

Model name: sca_sdf2r

Parameters: : init_value, conversion factor.

Ports: p - positive terminal , n- negative terminal , ctrl -static dataflow control input

Note: resistance = init_value + conversion factor · sdf in

13. Variable inductor controlled by static dataflow signal

Model name: sca_sdf2l

Parameters: nominal_inductance=0.0 / H, conversion_factor=1.0 / H, initial_current=0.0 / A

Ports: p-positive terminal , n- negative terminal , ctrl - static dataflow control input

Note: inductance = nominal inductance + conversion factor · sdf_in

14. Variable capacitor controlled by SDF signal

Model name: sca_sdf2c

Parameters: -

Ports: p - positive terminal, n -negative terminal, ctrl - static dataflow control input

Note: capacitance = nominal capacitance + conversion factor · sdf in

15. Voltage source driven by SystemC signal

Model name: sca_sc2v

Parameters: gain=1 / voltage scaling factor

Ports: p- positive terminal , n - negative terminal , ctrl - static dataflow control input

Note: voltage = nominal voltage + gain · sc in

16. Current source driven by SystemC signal

Model name: sca_sc2i

Parameters: gain=1 / current scaling factor, Nominal current

Ports: p - positive terminal, n - negative terminal , ctrl - static dataflow control input

Note: current = nominal current + gain · sc in

17. Variable resistor controlled by SystemC signal

Model name: sca_sc2r

Parameters: init_value, conversion factor

Ports: p - positive terminal, n - negative terminal , ctrl - static dataflow control input

Note: resistance = init_value + conversion factor · sc in

18. Variable inductor controlled by SystemC signal

Model name: sca_sc2l

Parameters: nominal_inductance=0.0 / H, conversion_factor=1.0 / H, initial_current=0.0 / A

Ports: p - positive terminal, n - negative terminal , ctrl - static dataflow control input

Note: inductance = nominal inductance + conversion factor · sc in

19. Variable capacitor controlled by SystemC signal

Model name: sca_sc2c

Parameters: nominal_capacitance=0.0 / F, conversion_factor=1.0 / F, initial_voltage=0.0 / V

Ports: p - positive terminal, n - negative terminal , ctrl - static dataflow control input

Note: capacitance = nominal capacitance + conversion factor · scin

20. Constant voltage source

Model name: sca_vconst

Parameters: value=1 / V

Ports: p,n

21. Constant constant source

Model name: sca_iconst

Parameters: value=1 / A

Ports: p,n

22. Voltage to static dataflow converter (reference node: gnd)

Model name: sca_v2sdf

Parameters: scale=1 / voltage scaling factor

Ports: p, sdf_voltage

Note: sdf out = scale · voltage

23. Differential voltage to static dataflow converter

Model name: sca_vd2sdf

Parameters: scale=1 / voltage scaling factor

Ports: p - positive terminal, n - negative terminal , sdf_voltage - static dataflow output

Note: sdf out = scale · voltage

24. Current to static dataflow converter

Model name: sca_i2sdf

Parameters: scale=1 /current scaling factor

Ports: p - positive terminal, n - negative terminal , sdf_current - static dataflow output

Note: sdf out = scale · current

25. Switch controlled by static dataflow signal

Model name: sca_sdf_rswitch

Parameters: ron =1.0e-6 / ON resistance , roff = 1.0e12 / OFF resistance, off_val= false /decides which position is the off-position

Ports: p - positive terminal, n - negative terminal , ctrl - static dataflow control input

26 Switch controlled by SystemC signal

Model name: sca_sc_rswitch

Parameters: ron =1.0e-6 / ON resistance , roff = 1.0e12 / OFF resistance, off_val= false /decides which position is the off-position

Ports: p - positive terminal, n - negative terminal , ctrl - static dataflow control input

27. Ideal linear transformer

Model name: sca_ideal_transformer

Parameters: n=1 /turns ratio

Ports: p1 -positive terminal of primary port, n1-negative terminal of primary port, p2 - positive terminal of secondary port, n2 -negative terminal of secondary port

28.Ground

Model name: gnd

Parameters: -

Ports: -

APPENDIX D

BIT MAPS OF SOME SPECIAL FUNCTION REGISTERS

Table D- 1: Bit map of `STATUS` register

Bit #	Bit name	Description
7	IRP	Register Bank select bit. (For indirect Addressing) --(R/W) 1 : Bank 2,3 0 : Bank 0,1
6-5	RP1-RP0	Register Bank select bit. (For indirect Addressing)--(R/W) 00 : Bank 0 01 : Bank 1 10 : Bank 2 11 : Bank 3
4	TO	Time-out bit --(R) 1 : After power-up, `CLRWDT` instruction or SLEEP instruction 0 : A WDT time-out occurred
3	PD	Power-down bit. --(R) 1 : After power-up by `CLRWDT` instruction 0 : By execution SLEEP instruction
2	Z	Zero bit. --(R/W) 1 : Result of arithmetic operation is zero. 0 : Result of arithmetic operation is NOT zero.
1	DC	Digit carry/borrow bit (`ADDWF`, `ADDLW`, `SUBWF`, `SUBLW` instructions) (for borrow polarity is reversed) --(R/W) 1 : A carry-out from the 4^{th} lower bit of result occurred 0 : No carry-out from the 4^{th} lower bit of result .
0	C	Digit carry/borrow bit (`ADDWF`, `ADDLW`, `SUBWF`, `SUBLW` instructions) (for borrow polarity is reversed) --(R/W) 1 : A carry-out from the Most significant bit of result occurred 0 : No carry-out from the Most significant lower bit of result .

Table D- 2: Bit map of `OPTION_REG` register

<table>
<tr><th>Bit #</th><th>Bit name</th><th>Description</th></tr>
<tr><td>7</td><td>RBPU</td><td>Port B pull-up enable bit --(R/W)
1 : Port B pull-ups disabled
0 : Port B pull-ups enabled by individual port latch enables</td></tr>
<tr><td>6</td><td>INTEDG</td><td>Interrupt edge select bit --(R/W)
1 : Interrupt on rising edge of RB0/INT pin
0 : Interrupt on falling edge of RB0/INT pin</td></tr>
<tr><td>5</td><td>T0CS</td><td>Timer 0 clock source select bit --(R/W)
1 : Transition on RA4/T0CKI pin
0 : Internal instruction cycle clock</td></tr>
<tr><td>4</td><td>T0SE</td><td>Timer 0 source edge select bit --(R/W)
1 : Increment on high-to-low transition on RA4/T0CKI pin
0 : Increment on low-to-high transition on RA4/T0CKI pin</td></tr>
<tr><td>3</td><td>PSA</td><td>Pre-scalar assignment bit --(R/W)
1 : Pre-scalar assigned to WDT
0 : Pre-scalar assigned to Timer 0</td></tr>
<tr><td>2-1-0</td><td>PS2:PS0</td><td>Pre-scalar rate selection bits --(R/W)
<table>
<tr><th>Bit value</th><th>TMR0 rate</th><th>WDT rate</th></tr>
<tr><td>000</td><td>1:2</td><td>1:1</td></tr>
<tr><td>001</td><td>1:4</td><td>1:2</td></tr>
<tr><td>010</td><td>1:8</td><td>1:4</td></tr>
<tr><td>011</td><td>1:16</td><td>1:8</td></tr>
<tr><td>100</td><td>1:32</td><td>1:16</td></tr>
<tr><td>101</td><td>1:64</td><td>1:32</td></tr>
<tr><td>110</td><td>1:128</td><td>1:64</td></tr>
<tr><td>111</td><td>1:256</td><td>1:128</td></tr>
</table></td></tr>
</table>

Table D- 3: Bit maps of (a) `TRISE` (b) `EECON1` registers

Bit #	Bit name	Description
7	IBF	Input buffer full status bit --(R) 1 : A word has been received and waiting to be read by CPU 0 : No word has been received
6	OBF	Output buffer full status bit --(R) 1 : Output buffer still holds previously written word 0 : Output buffer has been read
5	IBOV	Input buffer overflow detect bit --(R/W) 1 : A write occurred when a previously input word has not been read (must be cleared in software) 0 : No overflow occurred
4	PSPMODE	Parallel Slave Port mode select bit. --(R/W) 1 : Parallel slave port mode 0 : General purpose I/O mode
3	-	Unimplemented. Read as zero
2	TRISE2	RE2 direction control bit --(R/W) 1 : Input 0 : Output
1	TRISE1	RE1 direction control bit --(R/W) 1 : Input 0 : Output
0	TRISE0	RE0 direction control bit --(R/W) 1 : Input 0 : Output

(a)

Bit #	Bit name	Description
7	EEPGD	Program/Data EEPROM select bit --(R/W) 1 : Access program memory 0 : Access Data memory
6-5-4	-	Unimplemented .Read as 0.
3	WRERR	EEPROM Error flag bit --(R/W) 1 : A write operation prematurely terminated.(Any reset during operation) 0 : The write operation completed
2	WREN	EEPROM write enable bit --(R/W) 1 : Allows write cycles 0 : Inhibits write to EEPROM
1	WR	Write control bit --(R/W) 1 : Initiates a write cycle.(software can only set it) 0 : Write cycle to EEPROM is complete
0	RD	TMR2 PR2 match interrupt enable bit --(R/W) 1 : Initiates a EEPROM read. (software can only set it.) 0 : Does not initiate an EEPROM read.

(b)

Table D- 4: Bit map of `TXSTA` register

Bit #	Bit name	Description
7	CSRC	Clock source select bit --(R/W) Asynchronous mode Don't care Synchronous mode 1 : Master mode (Clock generated internally from BRG) 0 : Slave mode (Clock from external source)
6	TX9	9 bit transmit enable bit --(R/W) 1 : Selects 9 bit transmission 0 : Selects 8 bit transmission
5	TXEN	Transmit enable bit --(R/W) 1 : Transmit enabled 0 : Transmit disabled
4	SYNC	USART mode select bit. --(R/W) 1 : Synchronous mode 0 : Asynchronous mode
3	-	Unimplemented. Read as zero
2	BRGH	High Baud rate select bit --(R/W) Asynchronous mode 1 : High speed 0 : Low speed Synchronous mode Unused in this mode
1	TRMT	Transmit Shift register status bit --(R) 1 : TSR empty 0 : TSR full
0	TX9D	9^{th} bit of transmit data. Can be parity bit --(R/W)

Table D- 5: Bit map of RCSTA register

Bit #	Bit name	Description
7	SPEN	Serial port enable bit --(R/W) 1 : Serial port enabled 0 : Serial port disabled
6	RX9	9 bit receive enable bit --(R/W) 1 : Selects 9 bit reception 0 : Selects 8 bit reception
5	SREN	Single Receive enable bit --(R/W) Asynchronous mode Don't care Synchronous mode – master 1 : Enables single receive 0 : Disables single receive This bit is cleared after reception is complete Synchronous mode – receive Don't care
4	CREN	Continuous receive enable bit. --(R/W) Asynchronous mode 1 : Enables continuous receive 0 : Disables continuous receive Synchronous mode 1 : Enables continuous receive until CREN bit is cleared (CREN overrides SREN) 0 : Disables continuous receive
3	-	Unimplemented. Read as zero
2	FERR	Framing error bit --(R) 1 :Framing error 0 : No framing error
1	OERR	Overrun error bit --(R) 1 : Overrun error(can be cleared by clearing CREN) 0 : No overrun error
0	TX9D	9^{th} bit of received data. Can be parity bit --(R)

Table D- 6: Bit map of (a) `T1CON` (b) `T2CON` registers

Bit #	Bit name	Description
7	-	Unimplemented. Read as zero
6	-	Unimplemented. Read as zero
5-4	T1CKPS1: T1CKPS0	Timer1 input clock pre-scale select bits --(R/W) 11 : 1:8 Pre-scale value 10 : 1:4 Pre-scale value 01 : 1:2 Pre-scale value 00 : 1:1 Pre-scale value
3	T1OSCEN	Timer 1 oscillator enable bit --(R/W) 1 : Oscillator is enabled 0 : Oscillator is shut-off .
2	T1SYNC	Timer 1 external clock input synchronization select bit--(R/W) When TMR1CS =1 1 : Do not synchronize external clock input 0 : Synchronize external clock input When TMR1CS =0 This bit is ignored
1	TMR1CS	Timer 1 clock source select bit --(R/W) 1 :External clock from pin T1OSO/T1CKI (on the rising edge) 0 : Internal clock (fosc/4)
0	TMR1ON	Timer 1 on bit --(R/W) 1 : Enables Timer 1 0 : Stops Timer 1

(a)

Bit #	Bit name	Description
7	-	Unimplemented. Read as 0.
6-5-4-3	TOUTPS3: TOUTPS0	Timer 2 output post-scale select bit --(R/W) 0000 = 1:1 Post-scale 0001 = 1:2 Post-scale …. …. 1111 = 1:16 Post-scale
2	TMR2ON	Timer 2 on bit --(R/W) 1 : Timer 2 is on 0 : Timer 2 is off
1-0	T2CKPS1: T2CKPS0	Timer 2 pre-scale select bits. --(R/W) 00 : Pre-scalar is 1 01 : Pre-scalar is 4 1x : Pre-scalar is 16

(b)

Table D- 7: Bit map of CCPCON register

Bit #	Bit name	Description
7-6	-	Unimplemented. Read as 0
5-4	DC1B1: DC1B0	PWM duty cycle bit 1 and bit 0 --(R/W) Capture mode Unused Compare mode Unused PWM mode These bits are the LSBs (bit1 and bit 0) of the 10 bit PWM duty cycle. Upper 8 bits are stored in CCPR1L register.
3-2-1-0	CCP1M3: CCP1M0	TCCP1 mode select bits --(R /W) 0000 = Capture/Compare/PWM module is off 0100 = Capture mode, every falling edge 0101 = Capture mode, every rising edge 0110 = Capture mode, every 4th rising edge 0111 = Capture mode, every 16th rising edge 1000 = Compare mode, Initialize CCP1 pin low, on match force CCP pin high 1001 = Compare mode, Initialize CCP1 pin high, on match force CCP pin low 1010 = Compare mode Generate software interrupt on match 1011 = Compare mode, Trigger special event. (CCPIF is set) 11xx = PWM mode

Table D- 8: Bit map of (a) ADCON0 (b) DACON registers

Bit #	Bit name	Description
7-6	ADCS1: ADCS0	A/D conversion clock select bits --(R/W) 00 : fosc/2 01 : fosc/8 10 : fosc/32 11 : F_{RC}(clock derived from internal RC oscillator)
5-4-3	CHS2: CHS0	Register Bank select bit. (For indirect Addressing)--(R/W) 000 : Channel 0 (AN0) 001 : Channel 1 (AN1) 010 : Channel 2 (AN2) 011 : Channel 3 (AN3) 100 : Channel 4 (AN4) 101 : Channel 5 (AN5) 110 : Channel 6 (AN6) 111 : Channel 7 (AN7)
2	GO/DONE	A/D conversion status bit --(R/W) When ADON = 1 1 : A/D conversion in progress(cleared automatically by hardware when conversion completed) 0 : A/D conversion not in progress
1	-	Unimplemented. Read as 0
0	ADON	A/D on bit. --(R/W) 1 : A/D module powered up. 0 : A/D converter module turned-off.

(a)

Bit #	Bit name	Description
7-6-5-4-3-2	-	Unimplemented. Read as 0
1	DAEN	D/A conversion enable bit. --(R/W) 1 : D/A conversion is enabled. 0 : D/A conversion is disabled.
0	DAON	D/A on bit. --(R/W) 1 : D/A module powered up. 0 : D/A converter module turned-off.

(b)

Table D- 9: Bit map of (a) ADCON1 register (b) ADCON<3:0> bits

Bit #	Bit name	Description
7-6	-	Unimplemented. Read as 0.
5	ADFM	A/D format select 1 : Right justified. 6 Most significant bits of ADRESH are read as '0' 0 : Left justified. 6 Least significant bits of ADRESL are read as '0'
4	-	Unimplemented read as '0'.
3-2-1-0	PCFG3: PCFG0	A/D configuration control bits See Table D-9 (b)

(a)

PCFG	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0	VREF+	VREF-	C/R
0000	A	A	A	A	A	A	A	A	AVDD	AVSS	8/0
0001	A	A	A	A	VREF+	A	A	A	AN3	AVSS	7/1
0010	D	D	D	A	A	A	A	A	AVDD	AVSS	5/0
0011	D	D	D	A	VREF+	A	A	A	AN3	AVSS	4/1
0100	D	D	D	D	A	D	A	A	AVDD	AVSS	3/0
0101	D	D	D	D	VREF+	D	A	A	AN3	AVSS	2/1
011X	D	D	D	D	D	D	D	D	-	-	0/0
1000	A	A	A	A	VREF+	VREF-	A	A	AN3	AN2	6/2
1001	D	D	A	A	A	A	A	A	AN3	AVSS	6/0
1010	D	D	A	A	VREF+	A	A	A	AN3	AVSS	5/1
1011	D	D	A	A	VREF+	VREF-	A	A	AN3	AN2	4/2
1100	D	D	D	A	VREF+	VREF-	A	A	AN3	AN2	3/2
1101	D	D	D	D	VREF+	VREF-	A	A	AN3	AN2	2/2
1110	D	D	D	D	D	D	D	A	AVDD	AVSS	1/0
1111	D	D	D	D	VREF+	VREF-	D	A	AN3	AN2	1/2

(b)

Table D- 10: Bit map of (a) `INTCON` register

Bit #	Bit name	Description
7	GIE	Global interrupt enable bit --(R/W) 1 : Enables all unmasked interrupts 0 : Disables all interrupts
6	PEIE	Peripheral interrupt enable bit--(R/W) 1 : Enables all unmasked peripheral interrupts 0 : Disables all peripheral interrupts
5	T0IE	Timer 0 overflow interrupt enable bit --(R/W) 1 : Enables TMR0 interrupt 0 : Disables TMR0 interrupt
4	INTE	RB0/INT external interrupt enable bit. --(R/W) 1 : Enables RB0/INT external interrupt 0 : Disables RB0/INT external interrupt
3	RBIE	RB port change interrupt enable bit. --(R/W) 1 : Enables RB port change interrupt 0 : Disables RB port change interrupt
2	T0IF	Timer 0 overflow interrupt flag bit --(R/W) 1 : Timer 0 register has overflowed (must be cleared in software) 0 : Timer 0 did NOT overflow
1	INTF	RB0/INT external interrupt flag bit. --(R/W) 1 : RB0/INT external interrupt occurred(must be cleared in software) 0 : RB0/INT external interrupt did NOT occur
0	RBIF	RB port change interrupt flag bit. --(R/W) 1 : At least one of the RB7:RB4 pins changed state (must be cleared in software) 0 : None of the RB7:RB4 pins have changed state

Table D- 11: Bit map of (a) `PIR1` (b)`PIR2` registers

Bit #	Bit name	Description
7	PSPIF	Parallel slave port Read/Write interrupt flag bit --(R/W) 1 : A read/write operation takes place (must be cleared in software) 0 : No read/write has occurred
6	ADIF	A/D converter interrupt flag bit --(R) 1 : An A/D conversion completed 0 : The A/D conversion is NOT complete
5	RCIF	USART Receive interrupt flag bit --(R) 1 : USART Receive buffer full 0 : USART Receive buffer is Empty
4	TXIF	USART transmit interrupt enable bit --(R) 1 : USART Transmit buffer is empty 0 : USART Transmit buffer full
3	-	Unimplemented .Read as 0.
2	CCPIF	CCP1 interrupt flag bit --(R/W) Capture mode 1 : A Timer 1 register capture occurred (must be cleared in software) 0 : No Timer 1 register capture occurred Compare mode 1 : A Timer 1 reg. compare match occurred (must be cleared in software) 0 : No Timer 1 register compare match occurred PWM mode Unused in this mode
1	TMR2IF	TMR2 PR2 match interrupt flag bit --(R/W) 1 : TMR2 to PR2 match occurred (must be cleared in software) 0 : No TMR2 to PR2 match occurred
0	TMR1IF	Timer 1 overflow interrupt flag bit --(R/W) 1 : Timer 1 register overflowed (must be cleared in software) 0 : Timer 1 did NOT overflow

(a)

Bit #	Bit name	Description
7-6-5	-	Unimplemented. Read as 0
4	EEIF	EEPROM write operation interrupt flag bit --(R/W) 1 : EEPROM write operation completed (must be cleared in software) 0 : EEPROM write operation is NOT complete or has NOT been started
3-2-1-0	-	Unimplemented. Read as 0

(b)

Table D- 12: Bit map of (a) `PIE1` (b)`PIE2` registers

Bit #	Bit name	Description
7	PSPIE	Parallel slave port Read/Write interrupt enable bit --(R/W) 1 : Enables PSP read/write interrupts 0 : Disables PSP read/write interrupts
6	ADIE	A/D converter interrupt enable bit --(R/W) 1 : Enables A/D converter interrupt 0 : Disables A/D converter interrupt
5	RCIE	USART Receive interrupt enable bit --(R/W) 1 : Enables USART Receive interrupt 0 : Disables USART Receive interrupt
4	TXIE	USART transmit interrupt enable bit --(R/W) 1 : Enables USART Transmit interrupt 0 : Disables USART Transmit interrupt
3	-	Unimplemented .Read as 0.
2	CCPIE	CCP1 interrupt enable bit --(R/W) 1 : Enables CCP1 interrupt 0 : Disables CCP1 interrupt
1	TMR2IE	TMR2 PR2 match interrupt enable bit --(R/W) 1 : Enables TMR2 to PR2 match interrupt 0 : Disables TMR2 to PR2 interrupt
0	TMR1IE	Timer 1 overflow interrupt enable bit --(R/W) 1 : Enables Timer 1 overflow interrupt 0 : Disables Timer 1 overflow interrupt

(a)

Bit #	Bit name	Description
7-6-5	-	Unimplemented. Read as 0
4	EEIE	EEPROM write operation interrupt enable bit --(R/W) 1 : Enable EEPROM write interrupt 0 : Disable EEPROM write interrupt
3-2-1-0	-	Unimplemented. Read as 0

(b)

APPENDIX E

HARDWARE SYNHESIS OF ARITHMETIC LOGIC UNIT

Currently SystemC design language cannot be directly synthesized like VHDL or Verilog. Only method for hardware synthesis is converting SystemC code to other synthesizable hardware description languages. *SystemCrafter 2.0* is one of the commercial tools that perform this operation. However, latest version does not support all properties of SystemC for synthesis. Main limitations of the SystemCrafter are listed below [17].

- It does not support SC_METHOD process which implies that it can not synthesize combinational digital circuits
- Modules must be sensitive only to clock source and all sub-modules must be sensitive to same edge of the clock.
- Global variables are not allowed. This property restricts the high level abstraction. For this reason, every sub-modules must be connected to each other with signals explicitly.

Due to these limitations, in order to illustrate a SystemC to hardware synthesis, only arithmetic logic unit (ALU) of the 16F871 will be converted to VHDL. Design flow of the Systemcrafter can be summarized as given in Figure E-1.

SystemCrafter generates gate level VHDL code both in C++ and VHDL. It also compiles SystemC code as C++ compiler and builds executable file. Generated VHDL file is compatible with Xilinx ISE 8.1i [20] FPGA synthesis tool. For this purpose, user only needs another library comes with Systemcrafter named *craftgatelibrary.vhd.* This library contains the functions that are used during the conversion.

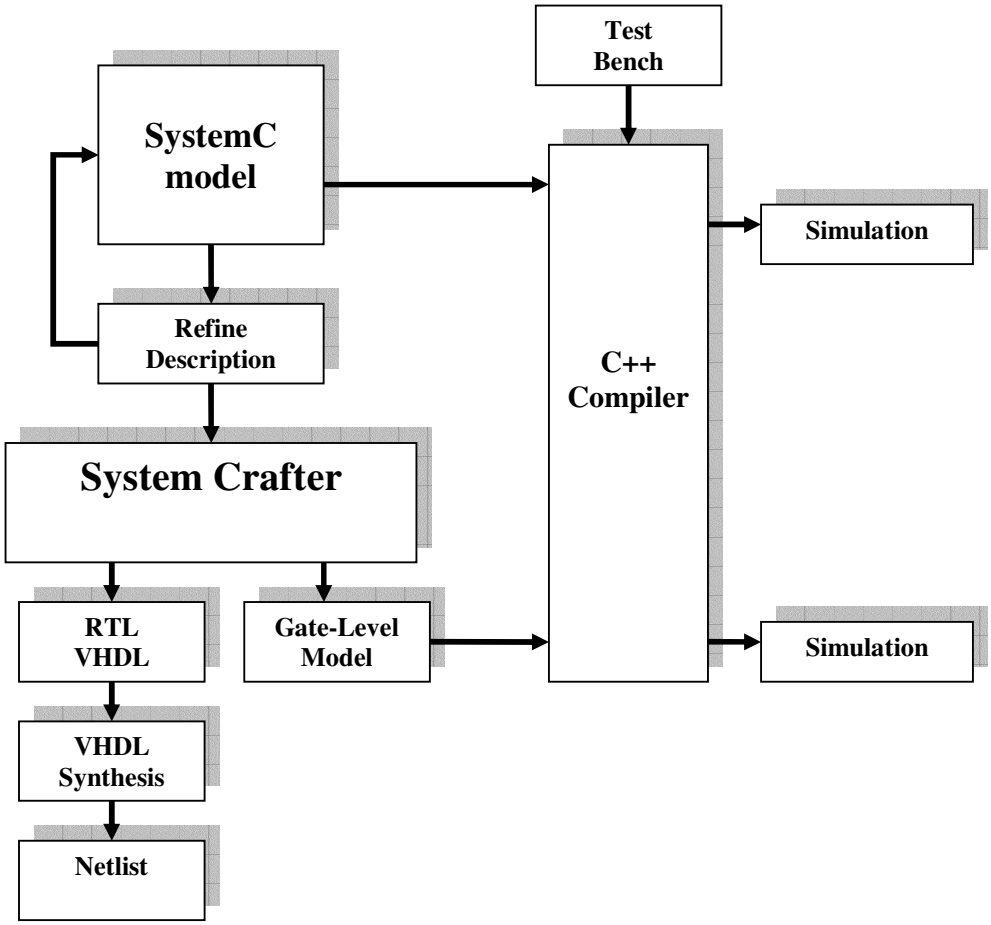


Figure E- 1: SystemCrafter Design flow

Synthesis Results:

Table E- 1: Device Utilization Summary

Logic Utilization	Used	Available	Utilization
Number of Slices	152	1920	7%
Number of Slice Flip Flops	87	3840	2%
Number of 4 input LUTs	290	3840	7%
Number of bonded IOBs	34	173	19%
Number of GCLKs	1	8	12%

Table E- 2: Timing Summary

Minimum period	12.964ns
Maximum Frequency	77.134MHz
Minimum input arrival time before clock	1.825ns
Maximum output required time after clock	19.206ns
Maximum combinational path delay	No path found

Implementation results:

Table E- 3: MAP Report Summary

Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	65	3,840	1%
Number of 4 input LUTs	290	3,840	7%
Logic Distribution			
Number of occupied Slices:	168	1,920	8%
Number of Slices containing only related logic	168	168	100%
Number of Slices containing unrelated logic	0	168	0%
Total			
Number 4 input LUTs	290	3,840	7%
Number of bonded IOBs	34	173	19%
IOB flip flops	22		
Number of GCLKs	1	8	12%
Total equivalent gate count for design	2,529		
Additional JTAG gate count for IOBs	1,632		

Table E- 4 : Summary of Post Place and Route Static Timing Report

Minimum period	13.595ns
Maximum Frequency	73.556 MHz
Minimum input required time before clock	3.019ns
Minimum output required time after clock	19.982ns

APPENDIX F

ASSEMBLY CODE OF THE DEMODULATOR EXAMPLE AND THE 16 BIT CRC TEST PROGRAM

Demodulator example Program

```
;*****************************************************************
;               Demodulator Example Program
;*****************************************************************

LIST            P=PIC16F871
INCLUDE         P16F871.INC
__CONFIG _HS_OSC & _WDT_OFF & _PWRTE_ON & _CP_OFF
ERRORLEVEL      -302        ;ELIMINATE BANK WARNING

MUL1    equ     0X20        ; 8 bit Multiplicand
MUL2    equ     0X21        ; 8 bit Multiplier
HBYTE   equ      0X22       ; High byte of the 16 bit result
LBYTE   equ      0X23       ; Low byte of the 16 bit result

DACON0      EQU     0x11D
DACDATAH    EQU     0x11E
DACDATAL    EQU     0x11F
SAME        EQU     0X1

ORG 0X0
       GOTO LBL_1    ; LBL_1 = 0X5
LBL_1
       BSF STATUS, RP0     ; Select Bank1
       CLRF TRISA
       CLRF TRISD
       MOVLW 0xFF
       MOVWF TRISB
       BCF STATUS, RP0     ; BANK0
```

```
        CLRF PORTA
        MOVLW 0x01
        MOVWF ADCON0
LOAD
        BSF ADCON0, GO       ; Start A/D Conversion
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP

        BSF PORTA, 0
        MOVF ADRESH,W
        MOVWF MUL1      ; MUL1
        MOVWF PORTC
        MOVF PORTB,W
        BCF PORTA,0
        MOVWF MUL2     ; MUL2
        CALL MUL
        SWAPF HBYTE,F
        MOVF HBYTE,W
        BSF STATUS,RP1 ; Go to Bank2
        BSF DACON0,0
        BSF DACON0,1
         MOVWF DACDATAH;
        BCF STATUS,RP1 ; BANK 0

        MOVLW 0XF0   ; ARRANGE FOR dac
        ANDWF HBYTE,F
        ANDWF LBYTE,F
        SWAPF LBYTE,F
        MOVF HBYTE,W
        IORWF LBYTE,F
        MOVF LBYTE,W
        MOVWF PORTD

        MOVF LBYTE,W
        BSF STATUS,RP1 ; Go to Bank2
        MOVWF DACDATAL;
        BCF STATUS,RP1 ; BANK 0
```

```
       GOTO LOAD

;****   Define a macro for adding & right shifting  **
MULT    MACRO   bit             ; Begin macro
       BTFSC   MUL2,bit
       ADDWF   HBYTE,SAME
       RRF     HBYTE,SAME
       RRF     LBYTE,SAME
       ENDM                    ; End of macro
; ****************************         Begin Multiplier Routine
MUL
       CLRF    HBYTE
       CLRF    LBYTE
       MOVF    MUL1, W        ; move the MULTiplicand to W reg.
       BCF     STATUS, C      ; Clear the carry bit in the status Reg.
       MULT    0
       MULT    1
       MULT    2
       MULT    3
       MULT    4
       MULT    5
       MULT    6
       MULT    7
       RETLW   0

       END
```

CRC test Program

```
;*****************************************************************
;               CRC Test Program
;*****************************************************************

LIST             P=PIC16F871
#INCLUDE         "P16F871.INC"
__CONFIG _HS_OSC & _WDT_OFF & _PWRTE_ON & _CP_OFF
ERRORLEVEL      -302    ;ELIMINATE BANK WARNING

CRC_HI  EQU     00EH
CRC_LO  EQU     00FH

ORG 0X0
       GOTO LBL_1    ; LBL_1 = 0X5
```

```
LBL_1
       CLRF    CRC_HI
       CLRF    CRC_LO
       MOVLW   0X80    ; CRC_HI:CRC_LO = 00 00
       CALL    CRC16
       MOVF    CRC_HI,W
       MOVWF   PORTB
       MOVF    CRC_LO,W
       MOVWF   PORTC
       CLRW
       MOVLW   0X75    ; CRC_HI:CRC_LO = A0 01
       CALL    CRC16
       MOVF    CRC_HI,W
       MOVWF   PORTB
       MOVF    CRC_LO,W
       MOVWF   PORTC
       MOVLW   0X8A    ; CRC_HI:CRC_LO = 27 A0
       CALL    CRC16
       MOVF    CRC_HI,W
       MOVWF   PORTB
       MOVF    CRC_LO,W
       MOVWF   PORTC
       MOVLW   0X0B    ; CRC_HI:CRC_LO = DF A6
       CALL    CRC16
       MOVF    CRC_HI,W
       MOVWF   PORTB
       MOVF    CRC_LO,W
       MOVWF   PORTC
       MOVLW   0X75    ; CRC_HI:CRC_LO = BD 1E
       CALL    CRC16
       MOVF    CRC_HI,W
       MOVWF   PORTB
       MOVF    CRC_LO,W
       MOVWF   PORTC
       MOVLW   0XC7    ; CRC_HI:CRC_LO = EF FC
       CALL    CRC16
       MOVF    CRC_HI,W
       MOVWF   PORTB
       MOVF    CRC_LO,W
       MOVWF   PORTC
       MOVLW   0XAA    ; CRC_HI:CRC_LO = D3 AE
       CALL    CRC16
       MOVF    CRC_HI,W
```

```
        MOVWF   PORTB
        MOVF    CRC_LO,W
        MOVWF   PORTC
        MOVLW   0X75    ; CRC_HI:CRC_LO = C3 D2
        CALL    CRC16
        MOVF    CRC_HI,W
        MOVWF   PORTB
        MOVF    CRC_LO,W
        MOVWF   PORTC
        MOVLW   0XC7    ; CRC_HI:CRC_LO = BA 82
        CALL    CRC16
        MOVF    CRC_HI,W
        MOVWF   PORTB
        MOVF    CRC_LO,W
        MOVWF   PORTC
        MOVLW   0X55    ; CRC_HI:CRC_LO = F3 7B
        CALL    CRC16
        MOVF    CRC_HI,W
        MOVWF   PORTB
        MOVF    CRC_LO,W
        MOVWF   PORTC
        MOVLW   0X43    ; CRC_HI:CRC_LO = 1C 73
        CALL    CRC16
        MOVF    CRC_HI,W
        MOVWF   PORTB
        MOVF    CRC_LO,W
        MOVWF   PORTC
        MOVLW   0X1C    ; CRC_HI:CRC_LO = 14 1C
        CALL    CRC16
        MOVF    CRC_HI,W
        MOVWF   PORTB
        MOVF    CRC_LO,W
        MOVWF   PORTC
        MOVLW   0X14    ; CRC_HI:CRC_LO = 00 14
        CALL    CRC16
        MOVF    CRC_HI,W
        MOVWF   PORTB
        MOVF    CRC_LO,W
        MOVWF   PORTC
        GOTO    LBL_1   ; CRC_HI:CRC_LO = 00 00
;*****************************************************************
; CRC-16  (X^16+X^15+X^2+X^0)
; NO TABLES, NO LOOPS, NO TEMPORARY REGISTERS USED.
```

```
;
; INPUT:  W = DATA BYTE FOR CRC CALCULATION
;         CRC_HI:CRC_LO 16 BIT CRC REGISTER
; OUTPUT: CRC_HI:CRC_LO UPDATED.
; NOTES:  CARRY IS TRASHED.
;         DIGIT CARRY IS TRASHED.
;         ZERO IS TRASHED.
;         W IS ZERO ON EXIT.
; 30 INSTRUCTIONS, 31 MACHINE CYCLES PER BYTE.
;
; COPYRIGHT (C) FEBRUARY 8, 2000. ALL RIGHT RESERVED.
; CHARLES ADER, PO BOX 940 PLEASANTON, CALIFORNIA, USA.
;
; THIS CODE STARTED OUT AS AN EXAMPLE PROGRAM FOUND IN
; DALLAS SEMICONDUCTOR APPLICATION NOTE 27:
; UNDERSTANDING AND USING CYCLIC REDUNDANCY CHECKS
; WITH DALLAS SEMICONDUCTOR IBUTTON(TM) PRODUCTS.
;
; THE APPLICATION NOTE SHOWS AN 8051 ASSEMBLY LANGUAGE
; ROUTINE THAT CALCULATES THE SAME CRC AS THE HARDWARE
; IN THE DS5001/2 SECURE MICRO.
;*****************************************************************
CRC16  XORWF   CRC_LO,W        ;W = INPUT XOR OLD CRC_LO
       XORWF   CRC_HI,W        ;SWAP OLD CRC_HI WITH W
       XORWF   CRC_HI,F        ;
       XORWF   CRC_HI,W        ;NEW CRC_HI = INPUT XOR OLD CRC_LO
       MOVWF   CRC_LO          ;NEW CRC_LO = OLD CRC_HI
;*****************************************************************
       MOVF    CRC_HI,W        ;SAVE CRC_HI IN W
       SWAPF   CRC_HI,F        ;TRADE NIBBLES
       XORWF   CRC_HI,F        ;XOR HIGH HALF BYTE WITH LOW
       RRF     CRC_HI,F        ;INITIALIZE CARRY
       BTFSC   CRC_HI,0
       INCF    STATUS,F        ;COMPLIMENT CARRY
       BTFSC   CRC_HI,1
       INCF    STATUS,F        ;COMPLIMENT CARRY
       BTFSC   CRC_HI,2
       INCF    STATUS,F        ;COMPLIMENT CARRY
       MOVWF   CRC_HI          ;RESTORE CRC_HI FROM W
;
; USE THE PARITY OF CRC_HI, (INPUT XOR CRC_LO),
; TO COMPLETE THE CRC CALCULATION.
;
```

```
MOVLW   001H
BTFSC   STATUS,C        ; IF CARRY
XORWF   CRC_LO,F        ; FLIP BIT 0 OF CRC_LO
MOVLW   040H
RRF     CRC_HI,F        ; SHIFT PARITY INTO CRC_HI
BTFSC   STATUS,C        ; IF SHIFT OUT IS ONE
XORWF   CRC_LO,F        ; FLIP BIT 6 OF CRC_LO
RLF     CRC_HI,W        ; UNSHIFT CRC_HI INTO W
XORWF   CRC_HI,F        ; COMBINE THEM
RRF     CRC_HI,F        ; SHIFT PARITY BACK INTO CRC_HI
MOVLW   080H
BTFSC   STATUS,C        ; IF SHIFT OUT IS ONE
XORWF   CRC_LO,F        ; FLIP BIT 7 OF CRC_LO

RETLW   0
END
```