SYSTEMC IMPLEMENTATION OF A RISC-BASED MICROCONTROLLER ARCHITECTURE

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

BY

SALİH ZENGİN

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

DECEMBER 2006

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: Salih ZENGİN
Signature :

ABSTRACT

SYSTEMC IMPLEMENTATION OF A RISC-BASED MICROCONTROLLER ARCHITECTURE

ZENGİN, Salih
M.S., Department of Electrical and Electronics Engineering
Supervisor: Prof. Dr. Murat Aşkar

December 2006, 174 Pages

Increasing the complexity of modern electronic systems leads to Electronic System Level (ESL) modeling concept, which supports hardware and software co-design and co-verification environment in a single framework. SystemC language, which is an IEEE approved electronic design standard for system design and verification processes, provides such an environment by supporting a wide range of abstraction levels from system-level to register-transfer level (RTL). In this thesis, two different models of a processor core, whose instruction set architecture (ISA) is compatible with 16-bit TI MSP430 microcontroller, are designed by employing the classical hardware modeling capability of the SystemC language. With its well-designed orthogonal instruction set, elegant addressing modes, useful constant generators and flexible von-Neumann architecture, 16-bit RISC-like processor of the MSP430 microcontroller is an ideal selection for the system-on-a-chip (SoC) designs. Instruction set and addressing modes of the designed processors are simulated thoroughly. In addition, original 16-bit and 32-bit cyclic redundancy code (CRC) programs are used in order to verify the processor cores. In this study, SystemC to hardware flow is also illustrated by synthesizing the Arithmetic and Logic Unit (ALU) part of the processor into a Xilinx-based hardware.

Keywords: SystemC, SystemC Synthesis, MSP430, Computer Design

ÖZ

RISC TABANLI MİCRODENETLEYİCİ YAPISININ SYSTEMC İLE GERÇEKLENMESİ

ZENGİN, Salih
Yüksek Lisans., Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi : Prof. Dr. Murat Aşkar

Aralık 2006, 174 Sayfa

Elektronik sistemlerin gittikçe karmaşıklaşması, tek bir ortamda yazılım ve donanım birimlerinin birlikte tasarlanmasını ve doğrulanmasını destekleyen, sistem düzeyinde modelleme kavramını ortaya çıkarmıştır. SystemC, böyle bir ortamı sağlayan, sistem seviyesinden yazmaç transfer seviyesine kadar (RTL) geniş bir soyutlama aralığını destekleyen, IEEE onaylı, sistem tasarımı ve doğrulanması için geliştirilmiş elektronik tasarım standardıdır . Bu tezde, SystemC dilinin klasik donanım modelleme yeteneği kullanılarak, 16-bit'lik TI MSP430 mikrodenetleyicisinin komutlarına uyumlu, iki adet farklı işlemci modelleri tasarlanmıştır. İyi tasarlanmış ortagonal komutlarıyla, kullanışlı adresleme şekilleriyle, yararlı sabit sayı üreteçleriyle ve esnek von-Neumann mimarisi ile, 16-bit'lik RISC-benzeri MSP430 mikrodenetleyicileri, SoC tasarımları için ideal bir seçimdir. Tasarlanan işlemcilerin komut kümesi ve adresleme şekilleri tamamiyle denenmiştir. Bunun yanı sıra, orjinal 16-bit ve 32-bit'lik CRC programları da işlemcilerin çalışmalarını doğrulamak için kullanılmıştır. Ayrıca, SystemC'den donanıma geçiş, bahsedilen işlemcilerin arithmetik ve mantık birimi (ALU), Xilinx tabanlı donanıma dönüştürülerek bu çalışmada gösterilmiştir.

Anahtar Kelimeler: SystemC, SystemC Sentezlenmesi, MSP 430, Bilgisayar Tasarımı

To my family,

To my wife

ACKNOWLEGMENTS

I am most thankful to my supervisor Prof. Dr. Murat Aşkar for sharing his invaluable ideas and experiences on the subject of my thesis.

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 am very grateful to TÜBİTAK-SAGE for providing tools and other facilities throughout the production of my thesis.

I would like to forward my appreciation to all my friends and colleagues who contributed to my thesis with their continuous encouragement.

I would like to express my deep gratitude to my family, who has always provided me with constant support and help.

Special thanks to my wife for all her help and showing great patience during the writing process of my thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

➔	Write Operation
←➔	Exchange Fields
*	Affected Bit
-	Not Affected Bit
0	Logically Cleared Bit
1	Logically Set Bit
#N	Immediate Mode
@PC+	Immediate Mode
@R	Register Indirect Mode
@R+	Register Indirect Auto-increment Mode
X (0)	Absolute Mode
X (PC)	Symbolic Mode
X (R)	Indexed Mode
	SystemC Port

LIST OF ABBREVIATIONS

Ad	Addressing Mode Field for Destination Operand
Addr	Address
ALU	Arithmetic and Logic Unit
As	Addressing Mode Field for Source Operand
ASIC	Application Specific Integrated Circuits
ASM	Assembly Language
BW	Byte or Word field
C	Carry bit of the status register
CD-ROM	Compact Disk-Read Only Memory
CG	Constant Generator
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CU	Control Unit
DCO	Digitally Controlled Oscillator
DSP	Digital Signal Processing
DST	Destination Bus
ESL	Electronic System Level
EDA	Electronic Design Automation
FIFO	First In First Out
GIE	Global Interrupt Enable
HDL	Hardware Description Language
HW	Hardware
IC	Integrated Circuit
ID	Instruction Decode
IE	Instruction Execute
IF	Instruction Fetch
Inst	Instruction
IP	Intellectual Property
ISDB	Integrated Signal Data Base
MAB	Memory Address Bus

MCU	Microcontroller Unit
MDB	Memory Data Bus
N	Negative bit of the status register
Op	Operator
OP-CODE	Operation Code
OSCI	Open SystemC Initiative
PC	Program Counter
PCB	Printed Circuit Board
R	Register or Register Direct Mode
Rdst	Destination Register
Reg	Register
RISC	Reduced Instruction Set Architecture
RTL	Register Transfer Level
SoC	System on a Chip
SP	Stack Pointer
SR	Status Register
SRC	Source Bus
SW	Software
TFM	Timed Functional Model
UTFM	Untimed Functional Model
V	Overflow bit of the status register
VCD	Value Change Dump
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
WIF	Waveform Intermediate Format
Z	Zero bit of the status register

CHAPTER 1

INTRODUCTION

Today, modern electronic systems include many integrated circuits (ICs) such as processors, memories and application specific ICs (ASICs) in order to implement complicated functions. Technological advances in chip process technology enable the designers to integrate all of these components into a single chip, which can be referred as a system on a chip (SoC). The advantages of SoC implementation far outweigh that of classical printed circuit board (PCB) base systems, since SoC designs provide the opportunity of the low cost, low power consumption, faster operation, small size, and reliable implementations.

With the increasing complexity of this kind of heterogeneous systems, their implementation has become much more difficult to handle. The central motive is the inappropriate methodology and the limitations of the current electronic design automation (EDA) tools used. Besides, traditional approaches separate the electronic design process into two main parts, namely, software (SW) and hardware (HW). These two main groups work separately from each other, which brings some significant disadvantages for the large-scale systems.

To cope with the problems occurred as a result of the high complex embedded systems to a certain extent, electronic system level (ESL) modeling platform that brings the HW and SW developers in a single framework is needed to be used. The co-design and co-verification platforms such as SystemC and SystemVerilog are discussed in [1]. SystemC, having the advantage based on C++, provides such an environment.

Since the system-level design and verification use C++, SystemC has drawn the attention of the designers after it was first introduced in 1999 by Open SystemC Initiative (OSCI) [2, 3, 4]. SystemC ver. 2.1 was approved by IEEE in December

2005 [5]. SystemC is indeed a C++ class library built on top of the standard C++ together with an event-driven simulation kernel. It enables the implementation of the transaction level models and hardware-related constructs in the C++ environment. Thus, SystemC is an ideal candidate for system modeling language that brings the software and hardware designers into a single framework. As being a system design language, SystemC facilitates a wide range of abstraction levels for the hardware, software and communication channels. A communication channel constructs a bridge between the HW and SW, or HW and HW, or SW and SW by using transaction level modeling. As a result, refinement process from the system-level to register transfer or behavioral level can be performed easily, which is a unique quality specific to SystemC. For instance, instruction set simulators of a specific processor is used to verify the functionality of the processor within the high-level abstraction of the design. For real implementation, register transfer level (RTL) model of the processor is required. The complete top-to-bottom design process so called refinement process can be realized in a single design environment.

System designers need intellectual property (IP) core modules to analyze system-level design alternatives, explore architecture and examine performance at the early stage of the design process for the complex systems. Moreover, IP cores are essential for the industry adaptation of SystemC since design reuse is the dominant technique to decrease the design and production time and hence increase the productivity. SystemC IP cores that are available at various abstraction levels are being developed all over the world. PowerPC SystemC IP core licensed by Summit Design, Inc. [6], and ARM processors that are important parts of the CoWare's ConvergenSC model library [7] are two significant representative examples. In addition, the SystemC implementation of the DES algorithm [8], standard MD5 hash code [8] and JPEG encoder [9] are some other IP cores. Furthermore, there are also some studies about the conversion of the existing IP modules implemented in the classical HDL languages (Verilog and VHDL) into the SystemC code so as to reuse these designs in new SystemC projects [10, 11].

SystemC IP core library has been set up and it is being improved in the Electrical and Electronics Engineering at Middle East Technical University (METU). There are two more theses about the SystemC IP modules serving the purpose of the library, namely, SystemC implementation of the 8051 microcontroller [12] and optimized reconfigurable Viterbi decoders [13].

The purpose of the thesis is first and foremost to develop a 16-bit RISC processor core in the SystemC language for the SoC designs. In this thesis, two different implementations of a processor, which is compatible with the instruction set and timing of the 16-bit TI MSP430 microcontroller, are realized by using the classical hardware modeling (RTL/Behavioral level) capability of the SystemC language. MSP430 is relatively a new microcontroller and new versions are currently being designed. With its well-designed orthogonal instruction set, elegant addressing modes, useful constant generators and flexible von-Neumann architecture, 16-bit RISC-like processor of the MSP430 microcontroller is an ideal selection for the SoC applications. In this study, first a processor was designed by using the instruction-pipelining technique. Then, by analyzing the drawbacks of the first approach, the second processor was developed. Instruction set and addressing modes of the processors were simulated thoroughly. In addition, original 16-bit and 32-bit cyclic redundancy code (CRC) programs represented by Texas Instrument Inc. (TI) were employed in order to verify the processor cores [14].

Current technology requires the use of VHDL/Verilog as a mid step to realize hardware (netlist) implementation from the SystemC descriptions [15]. Programs such as SystemCrafter[16], CoCentric SystemC Compiler [17] and Prosilog [18] convert SystemC modules into VHDL and/or Verilog. Besides, in this study, SystemC to hardware flow is also illustrated by synthesizing the Arithmetic and Logic Unit part of the processor cores into a Xilinx-FPGA based hardware.

SystemC 2.0.1 release is used in all of the SystemC implementation parts. At the early stage of the thesis, some small designs such as adders, counters, flip-flops were designed with SystemC_Win 1.0 Beta[1] to explore the SystemC semantics. SystemC_Win, which relies on C++ Builder V5.5[1], provides a user-friendly

(1) Free version

environment to develop and verify the SystemC models. Users can easily create their own projects in SystemC_Win and run them in the context of the SystemC_Win environment. This environment provides a virtual console and a waveform viewer to display the messages and the results of the simulations.

SystemC 2.0.1 release is employed with the Microsoft Visual C++ 6.0[1] environment for the design and verification stages of the processor cores. The simulation waveforms can be stored in ASCII, WIF, ISDB and VCD formats. VCD trace files are preferred, which can be observed by means of GTKWave Analyzer v1.3.19[2] and SynaptiCad WaveViewer 11.03b[2]. As a MSP430 cross compiler, IAR Assembler V3.40A[1] is used for generating test programs in assembly language. Besides, trial version of SystemCrafter[3] is employed to convert the SystemC modules to VHDL descriptions. Generated VHDL codes are synthesized and implemented to a Xilinx-based FPGA via Xilinx ISE 8.1i[1]. To simulate the behavior of the VHDL codes and mapped physical designs, Xilinx ISE Simulator[1] is preferred. Logical netlists are synthesized from the VHDL descriptions by Xilinx synthesis tool (Xilinx XST[1]).

The thesis is composed of six chapters. Chapter 2 first introduces the SystemC design language and then discusses traditional and SystemC design flows. An overview of the MSP430 microcontroller family, processor registers, instruction set architecture and addressing modes of the processor are explained in detail employing some illustrative examples.

Chapter 3 presents the implementation of the processor cores. Firstly, the processor based on the instruction-pipeline scheme is introduced briefly (The details are given in Appendix-B). Then, the second approach is discussed by detailing the design of the datapath architecture, control unit and stimulus parts of the second processor.

([1]) Licensed to Tübitak-Sage

([2]) Free version

([3]) Trial version

Chapter 4 provides an explanation of the verification phase of the two processor cores. All the instructions and addressing modes are comprehensively simulated. Besides, the original CRC algorithms represented by Texas Instrument Inc. (TI) are employed so as to verify the overall functionality of the processor cores designed.

Chapter 5 deals with the synthesis process of the SystemC codes into hardware modules. In this chapter, following an overview of the SystemC to hardware design flows, the path and the tools used in order to generate hardware from the SystemC descriptions are given in adequate detail. Firstly, translation of the SystemC constructs into VHDL by the SystemCrafter synthesis program is explained. Then, Xilinx design flow, which is used to create Xilinx-based FPGA designs, is stated. In this chapter, the conversion of the SystemC model of the *Arithmetic and Logic Unit* of the processors into an FPGA-based hardware design is demonstrated as a case study.

Chapter 6 concludes the thesis. Following an overview of the thesis, most significant advantages and disadvantages of the MSP430 processor core are stated. Besides, some of the crucial points experienced while performing SystemC to hardware flow are mentioned.

Appendix-A covers the detailed instruction set, addressing modes and timing diagrams of the original MSP430 microcontroller. The design details of the first processor based on the instruction-pipelined architecture are presented in Appendix-B. The details of the simulation of the synthesized ALU on the Xilinx environment can be found in Appendix-C.

Microsoft Visual C++ projects of the SystemC processor cores, IAR embedded workbenches of the test programs, the SystemCrafter project of the arithmetic and logic unit, Xilinx ISE project files including VHDL testbench and soft version of the thesis are presented in the CD-ROM, which was attached to back cover of the thesis.

CHAPTER 2

SYSTEMC AND MSP430 MICROCONTROLLER

This chapter first introduces the SystemC design language and then discusses traditional and SystemC design flows. After that, following an overview of the MSP430 microcontroller family, processor registers, instruction set architecture and addressing modes of the processor core are mentioned in detail.

2.1 SystemC

SystemC 2.1 is an IEEE approved electronic design standard for system design and verification processes [5]. It has been developed by industry-sponsored consortium called the Open SystemC Initiative (OSCI), which is an independent not-for-profit organization made up of a wide range of companies, universities and individuals dedicated to contributing to and improving SystemC as an open source standard.

Strictly speaking, SystemC itself is not a language, and it is a C++ class library entirely built on top of standard object-oriented C++ language with an event-driven simulation kernel. SystemC enables the designers to model both software and hardware parts together using C++. It primarily focuses on hardware-oriented constructs such as concurrent behavior, notion of time, special HW data types, modules and hierarchy concepts because C++ already finds the answers of the most of the software concerns. Besides, SystemC 2.0 and upper versions provide also high-level abstraction of communication. The SystemC class libraries are available free via an open source license available at www.systemc.org.

A brief history of SystemC is illustrated in Table 2-1 below. In the design automation conference (DAC) in 1997, it was declared that efficient reactivity (waiting and watching) and concurrency were implemented in Scenic design

framework that allows the system designers to model hardware and software modules in C++ [19]. After the foundation of OSCI by Synopsys and CoWare in 1999, SystemC 0.9 that is the first version of SystemC was relased in September 1999. Version 0.9 was based on work by Synopsys, augmented by Register Transfer C (RTC) that is a proprietary CoWare language [20]. Then, SystemC 1.0 was introduced in 2000. By facilitating basic extensions in C++ for hardware design such as the notion of time, hardware-related data types, concurrency, modules, signals and I/O ports, it enables the designers to model at RT and behavioral abstraction levels, which are similar to hardware description languages (HDLs) such as Verilog and VHDL. Therefore, HW and SW co-design and co-verification environment was produced in the SystemC environment. However, it is lack of modeling capability at high abstraction levels. For example, the only way to communicate two modules is by using the pin-accurate models, which brings important limitations to modify the high-level system models. Moreover, the version included good support for fixed-point modeling in digital signal processing applications, which is difficult to do in traditional HDLs. Besides, SystemC 1.1 beta and 1.2 beta provided some limited communication refinement capabilities.

SystemC 2.0, which is a super set of SystemC 1.0, was publicized in 2002. The capability of this version was increased by adding generalized modeling of the communication and synchronization constructs into older releases such as channels, interfaces, events, FIFOs and transaction level modeling (TLM) concept. In the TLM concept, communication channels are modeled by employing interface function calls [21, 22]. High-level abstraction interface between the modules allows the designer to explore the system and determine the trade-offs at the early state of the system design process. The latest version SystemC 2.1 brings some important improvements such as increased modularity for IP delivery and better support for the TLM design. It was announced that the IEEE Standards organization ratified SystemC 2.1 as standard 1666-2005 on December 12, 2005 [5].

Table 2-1: A Brief History of SystemC

Date	Notes
1997	Scenic Design Framework was introduced by Synopsys in UC Irvine, DAC'97.
1999	Open SystemC Initiative was founded by Synopsys and CoWare
September 1999	SystemC 0.9, first version of SystemC, was published. It supported cycle-accurate and pin-accurate modeling.
2000	Cadence joined OSCI in 2000.
February 2000	SystemC 0.91, fixed some bugs.
March 2000	SystemC 1.0 that widely accessed major release was announced.
October 2000	SystemC 1.0.1, fixed some bugs.
2001	Mentor Graphics joined OSCI.
February 2001	SystemC 1.2, provided various improvements in the language.
August 2002	SystemC 2.0, channels & event constructs were added. It has more clear syntax.
April 2002	SystemC 2.0.1 fixed some bugs. It is widely used version.
2003	SystemC Verification Library was published.
June 2003	Language reference manual (LRM) was in review.
Spring 2004	LRM of SystemC 2.1 was submitted for IEEE standard
December 2005	SystemC 2.1 was approved by the IEEE.

SystemC is a new system-level design standard and it provides significant improvements with respect to the classical approach. The following two subchapters discuss the traditional and SystemC-based system design flows.

2.1.1 Traditional System Design Flow

With the increasing complexity of the heterogeneous systems that consist of hardware and software components, their implementation has become much

more difficult to handle. The central motive is the inappropriate methodology and the limitations of the current electronic design automation (EDA) tools used. Figure 2-1 below illustrates the design flow of the traditional approach to cope with such complex systems.

As displayed in figure below, firstly, the system is modeled at high abstraction level by considering the system requirements. High-level programs such as MatLab and high-level programming languages such as C, C++, and Java are employed for this purpose. At the end of this stage, the system can be depicted by means of functional blocks.

Following the verification of the high-level model of the system, the design is divided up into hardware (HW) and software (SW) parts. This process is called *hardware-software partitioning*. This phase follows the implementation stage, where making changes can be difficult and expensive. Consequently, partitioning is crucial to obtain effective and efficient outcomes. Besides, it is often not clear that which part of the system can be mapped to HW and which part can be SW. Generally, high-speed functions are mapped to relatively more expensive hardware parts and software routines are selected for comparatively slow tasks. At the end of the system partitioning, all of the tasks will be assigned to two main groups -hardware and software.

The tasks assigned to hardware groups are separated into manageable sub-modules and all of the related functions are *manually* translated into register transfer level (RTL) descriptions via hardware-description languages (HDLs) such as Verilog and VHDL, which is a time-consuming and error prone process. Once the HDL descriptions are synthesized into logic circuits, the implementation process follows this synthesis so as to generate working hardware parts.

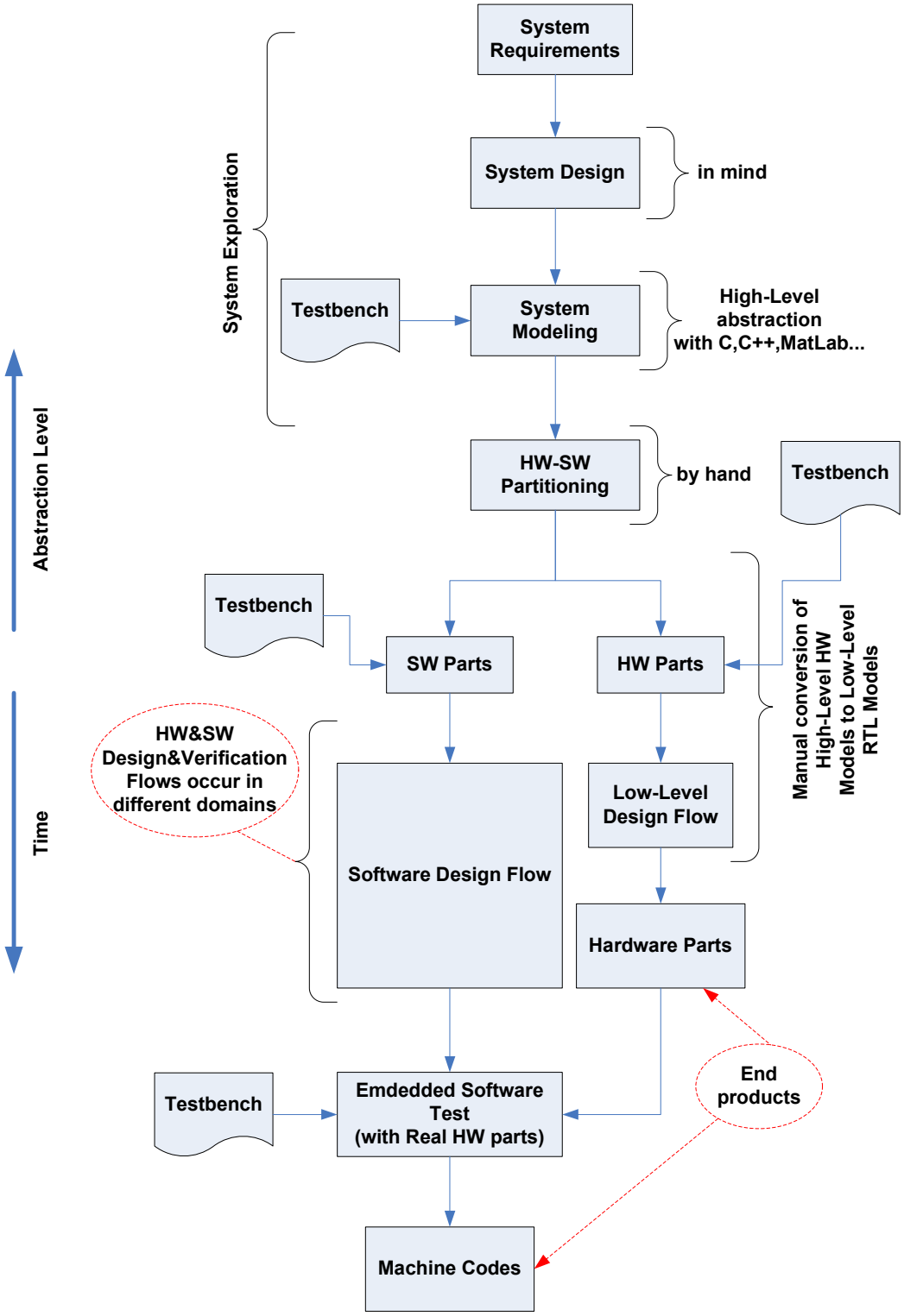

Figure 2-1: Traditional System Design Flow

Software-mapped functions are designed by the help of high-level languages. C and C++ are the most prominent languages for embedded applications. However, some subroutines can be realized with the assembly language. The high-level programs and assembler codes are then translated into machine codes by the use of cross compilers. After the verification phase of the peripheral hard cores, the embedded codes then become ready to be run on DSP, general purpose or multimedia processors.

The increasing complexity of the electronic systems creates some weaknesses in the conventional design flow. Some of the problems that arose are listed below:

- There is a gap between functional models and RTL models. It means that system architects, software and hardware designers follow separate design environments. Particularly, traditional approach lacks co-design and co-verification capabilities for the HW and SW modules, which results in some vital difficulties in testing, debugging and optimization of the overall system.
- Multiple system testbenches are employed throughout the design stages.
- The design flow costs high and long development cycles are needed.
- The software developer has to wait until the design of the first hardware prototype so as to complete his/her tasks.
- The entire system cannot be verified adequately before the first hardware prototype is ready. Consequently, problems or incompetence cannot be identified early in the design process, which may result in a decrease in system performance, or even a need to rebuild the prototype.
- HDL codes are not easy to change. Thus, if any high-level model of the system is altered, the propagation delay of the upgrading phase turns out to be so slow at the RTL design stage.
- RTL descriptions trigger so many events to be monitored. Thus, simulation of the large RTL designs takes so much time to be verified, which slows down the system verification process.

To cope with all of these types of problems, electronic system level (ESL) modeling platform that brings the HW and SW developers in the same framework

needs to be used [1]. SystemC, having the advantage based on C++, provides such an environment.

2.1.2 System Design with SystemC

SystemC is an ideal candidate for system-level modeling language since it brings the software and hardware designers into a *single framework*. Figure 2-2 below illustrates the system-design flow with SystemC.

With regard to the interpretation of the figure shown below, following the determination of the system requirements, system is designed in mind and then untimed functional model (UTFMs) is created. Next, by adding timing information to the processes, timed functional model (TFM) is generated. In this respect, it is worth noting that functional modeling in SystemC uses *finite sized* FIFO channels as different from the classical approach.

The phase of the system exploration results in SystemC models with an *executable specification* created in the C++ environment. Indeed, this model reflects a virtual system. Following the HW-SW partitioning, the implementation-level refinement of the SW and HW tasks commences.

The programmers of the embedded software design the software assignments in C/C++ environment and they can simulate the embedded software with virtual (high-level abstract) HW models. Thus, the programmer does not need the first hardware prototype for the verification purpose.

The hardware engineers refine the functional descriptions of the HW parts into RTL models, which can be performed manually or with an EDA tool. A HW designer can simulate a refined RTL module with the virtual system by the help of *adapters*, which are used to connect the pin-accurate model to high-level abstraction models of the communication parts.

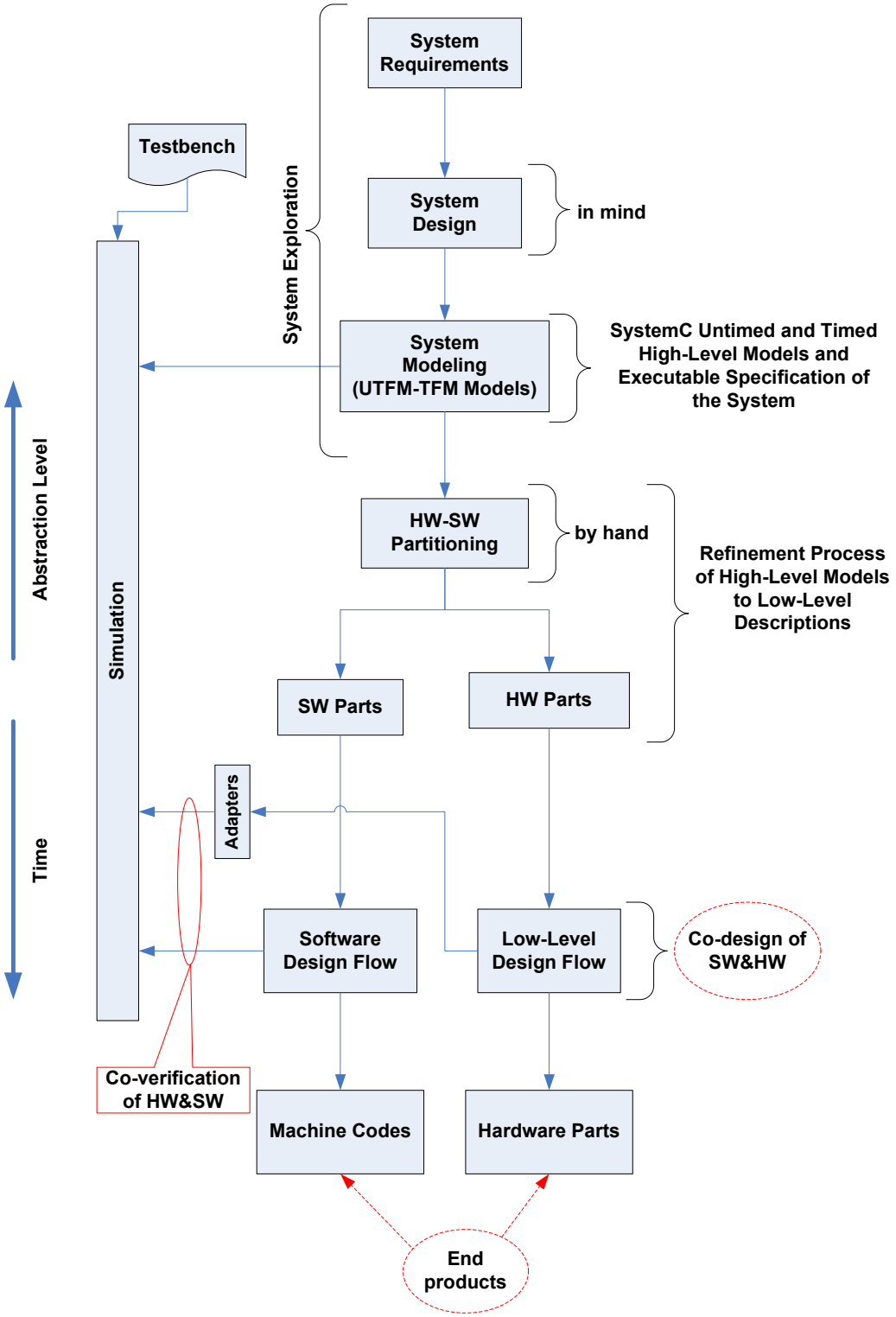

Figure 2-2: System Design Flow with SystemC

2.1.2.1 Advantages of SystemC Design Flow

System-level design flow with SystemC has some significant advantages compared to the traditional system design approach. Some of them are summarized below. SystemC

- makes the system design flow smooth, efficient, easy and less expensive.
- shortens development cycles.
- maximizes processing power and component reuse.
- offers co-design and co-verification environment, whereby there is no isolation between the HW and SW engineers.
- provides various abstraction levels and also having a single language allows the testbenches to be reused at various modeling levels.
- It is easier to explore various algorithms, architectures and implementation alternatives. Thus, it is easier to optimize the design.
- creates the HW and SW development team with an executable specification of the system. An executable specification, which acts as a strong reference for the system designers throughout the design stage of the system, is essentially a C++ program that exhibits the same behavior as the system when executed.
- Standard TLM interface increases IP reuse.
- There is no need to convert C/C++ level descriptions into RTL HDL (Verilog, VHDL…) codes. SystemC provides RTL/Behavioral level abstractions.
- smoothes the flow from system models to RTL models by supporting TLM and implementation through refinement.

For more information about SystemC, [12] can be referred. Besides, the steps needed to configure the Microsoft Visual Studio 6.0 as a SystemC compiler are summarized in [13]. In addition, more formal information can be founded in the SystemC user guide [3] and language reference manuals [4, 5], which can be downloaded from www.systemc.org. Other literature can provide useful explanatory understanding of SystemC as well [23, 24].

2.2 MSP430 Microcontroller

MSP430 is an ultra-low power, high performance, easy-to-use microcontroller with a modern 16-bit RISC-like CPU and memory-mapped peripherals [25, 26, 27]. Although 64kB memory can be accessed entirely by 16-bit registers, memory extended version of MSP430, namely MSP430X (announced in April 2006), can address up to 1MB address space without paging [28]. All of the microcontroller modules are combined together by using the von-Neumann architecture where a common memory bus is shared as illustrated in Figure 2-3 below.

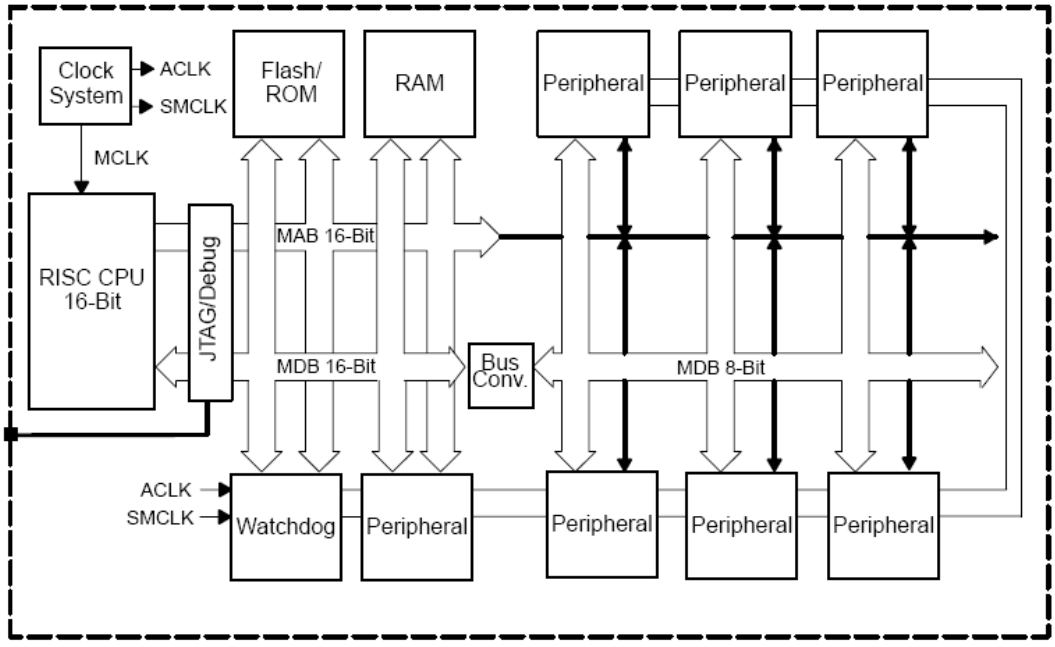

Figure 2-3: MSP430 Architecture [25]

The ultra-low-power feature of the controller stem from two main reasons. The first reason is the digitally controlled oscillator (DCO) start up time. The DCO can start up within 1 µs as depicted in Figure 2-4 below, and this feature allows the system to stay in low-power mode for long idle conditions. The second one is that

effectively utilizing the peripherals permits the CPU to be turned off to save power. With its ultra-low power architecture, the MSP430 microcontrollers are suitable for battery-powered applications.

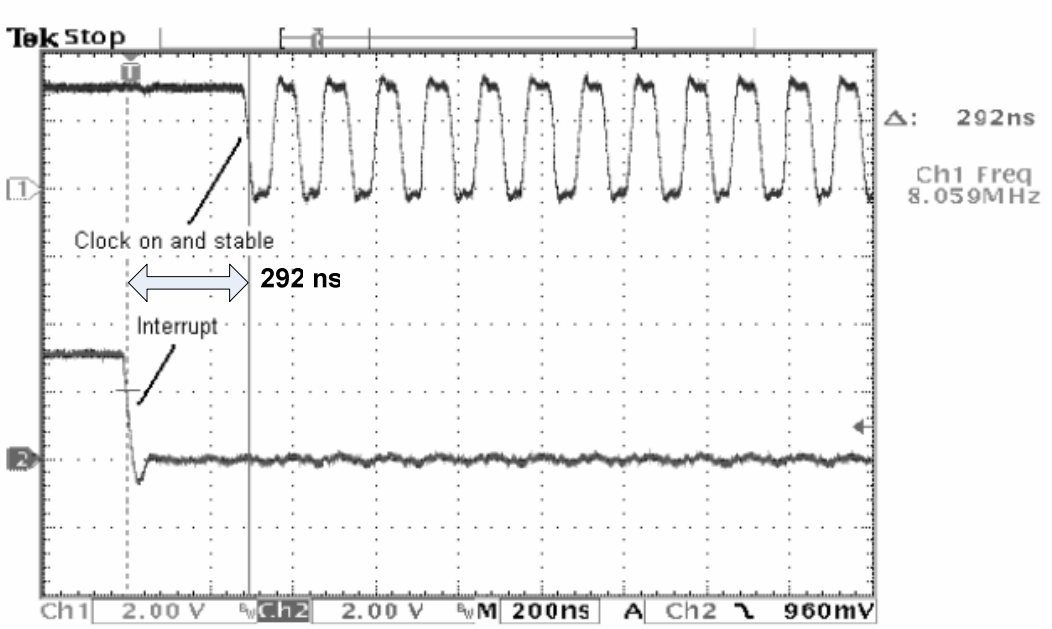

Figure 2-4: MSP430 Clock Startup [29]

Modern RISC-like CPU with fully addressable, single cycle, sixteen 16-bit CPU registers improves the performance of the MCU. In addition, it is allowed to access all of the registers including program counter, status register and stack pointer. Some commonly used constants are also generated by the constant registers and this feature reduces the code size of a given task and enables the assembler to emulate some of the instructions.

Small number of instructions, orthogonal feature of the instruction set and linear memory space make the MCU programmed easily in assembly language. Besides, it is optimized for modern high-level programming.

All intended peripheral modules can be connected easily to the MSP430 system. The modular construction that significantly depends on the von-Neumann architecture and memory-mapped peripherals, whose data and control registers are located in the memory space are as shown in Figure 2-3 and Figure 2-5 respectively. This architecture enables the microcontroller designers to extend the product portfolio without difficulty.

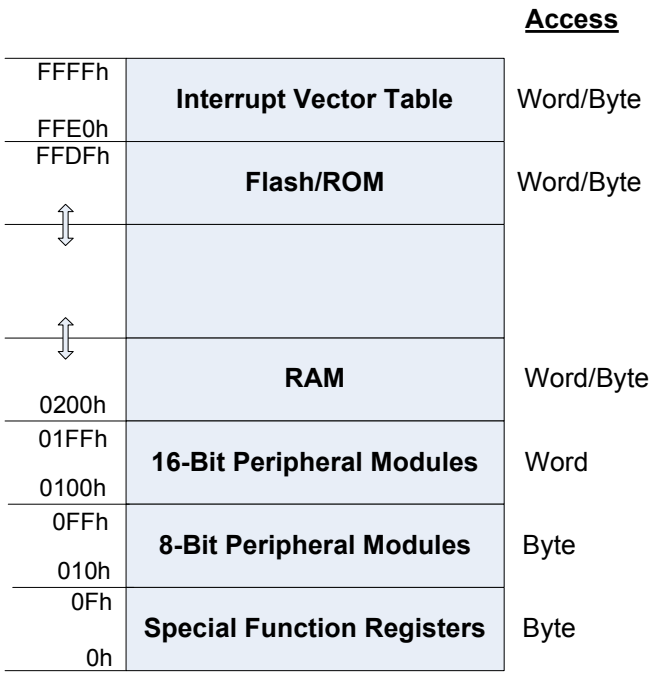

Figure 2-5: Memory Map [25]

RISC-like construction of the MSP430 processor core has some important advantages in some designs intended for the real-time applications. In typical RISC architectures, to increase the performance, total number of memory accesses is reduced by performing most of the computations within the processor registers. However, input and output of the calculation process are fetched from the memory by the help of some limited instructions such as LOAD and STORE. Besides, most of the instructions can only access to the internal registers. On the other hand, the MSP430 processor enables all of the instructions to be used with all of the possible addressing modes (with insignificant exceptions). This architecture speeds up the calculations related with RAM and it enables frequent and random accesses to the memory, which is critical for the real-time applications.

Following this introduction part, the CPU features of the MCU will be focused on. The chapter proceeds detailed explanation of the available CPU registers and instruction set architecture (ISA), then it ends up with an explanation of the available addressing modes reinforced through examples.

2.2.1 Central Processing Unit Registers

There are sixteen 16-bit registers, four of which, namely, PC (Program Counter), SP(Stack Pointer), SR (Status Register),CG2 (Constant Generator-2)-have dedicated purposes, and the rest are classified as general purpose registers. There are also some hidden registers, which are invisible to the programmers. These secret registers, which will be discussed in the implementation part of the processor cores, have special tasks to perform some micro operations.

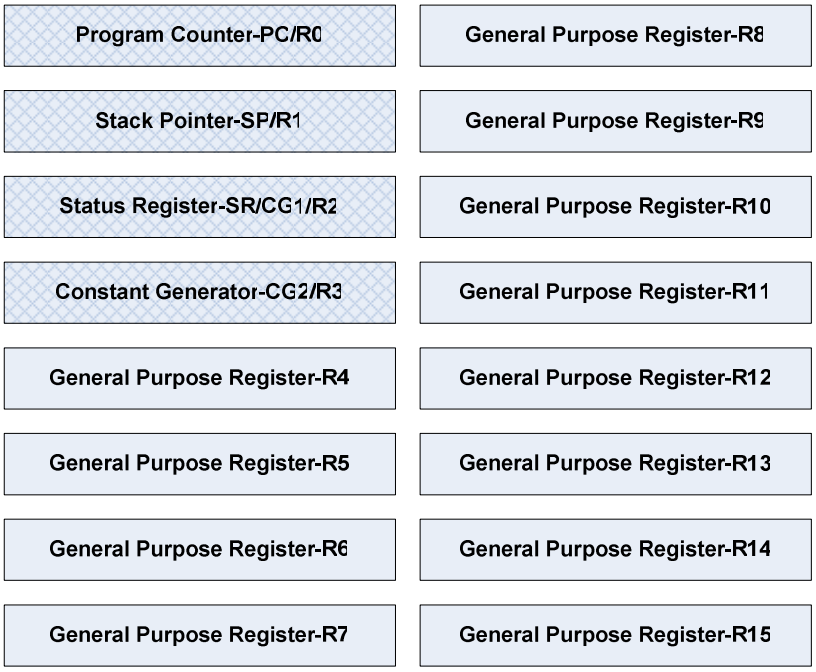

Figure 2-6: MSP430 CPU Registers

2.2.1.1 Registers That Have Pre-defined Functions

There are four registers that have dedicated functions, which can be listed as follows:

- Keeping the sequence of the program execution,
- Holding the address of the stack,
- Storing the status information, and
- Generating some commonly used constants.

2.2.1.1.1 Program Counter-PC/R0

Program Counter register stores the address of the next instruction. It behaves similar to an output register of a 16-bit loadable even counter. Since the instructions are located at even addresses of the memory, the least significant bit

of the PC is logic zero as shown in Figure 2-7. Moreover, all of the instructions and addressing modes that are applied to the general-purpose registers also can be applied to PC. However, some cases refer to special meanings. For instance, when the PC is used as a source register with indirect register auto-increment addressing mode (@PC+), this condition implies another addressing mode-immediate mode (#N). Here are some code examples:

MOV @PC+, R5; *Copy the immediate constant following the instruction word to R5*
XOR #1200h(PC), R5; *PC relative symbolic addressing mode is inferred.*
MOV #ODTU, PC; Branch *to address ODTU*
MOV @R5, PC; *Branch to address pointed by R5 register*

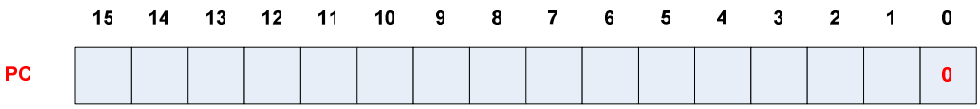

Figure 2-7: Program Counter, PC

2.2.1.1.2 Stack Pointer-SP/R1

The stack pointer (SP/R1) register is employed to store the return addresses of subroutine calls and interrupts. It is used by the PUSH, RETI, and CALL core instructions and POP, and RET emulated instructions. Moreover, the SP can be used with all of the instructions and addressing modes. As the SP points the instructions like PC, the least significant bit of the register is logic zero as illustrated in Figure 2-8 below.

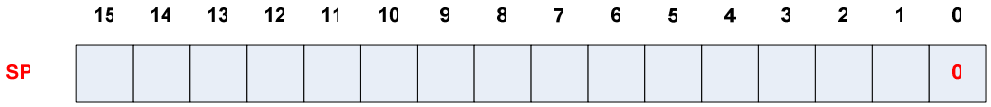

Figure 2-8: Stack Pointer, SP

2.2.1.1.3 Status Register (Constant Generator1)-SR/CG1/R2

Status register (SR) holds some control variables and keeps the status information, which is denoted by V (Overflow), N (Negative), Z (Zero), and C (Carry/Barrow) bits. Descriptions of these bits are as expressed in Table 2-2, and the usage of the register is as given in Table 2-3 below. In addition, the constants 0, +4 and +8 are generated by this register. That's why this register is also called as a *constant register*. Addressing mode bits for source operand (*As* field in the instruction word) determine which constant value is generated. Note that the absolute addressing mode is inferred by SR when it is referenced as an operand register in the indexed addressing mode. Thus, the register can only be used with the register direct mode. Other modes are not available with SR.

Note that some of the instructions do not affect status bits, or some logic operations affect these bits in a different way; for XOR, BIT and AND logical instructions, the carry bit contains the inverted zero status bit, for the sake of flexibility. Pease refer to the instruction set description that is given in Appendix A to see how an instruction effects the status information.

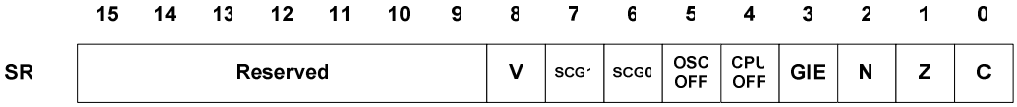

Figure 2-9: Status Register, SR

Table 2-2: Description of the Status Register Bits [25]

Bit	Description
Reserved	Reserved for the processor.
V	Overflow bit. For all instructions, it can be different usage. Thus, please see each instruction for the meaning of the overflow bit.
SCG1	System clock generator-1. When it has “1” value, the SMCLK turns off.
SCG0	System clock generator-0. When it has “1” value and DCOCLK is not used for MCLK or SMCLK, DCO dc generator turns off.
OSC OFF	Oscillator off. When it has “1” value and LFXT1CLK is not use for MCLK or SMCLK, LFXT1 crystal oscillator turns off.
CPU OFF	CPU off. When set, turns off the CPU.
GIE	General interrupt enable. When set, all maskable interrupts are enabled. When reset, all maskable interrupts are disabled.
N	Negative bit. This bit is set when the result of a byte or word operation is negative and cleared when the result is not negative. Word operation: N is set to the value of bit 15 of the result Byte operation: N is set to the value of bit 7 of the result.
Z	Zero bit. This bit is set when the result of a byte or word operation is 0 and cleared when the result is not 0.
C	Carry bit. This bit is set when the result of a byte or word operation produced a carry and cleared when no carry occurred.

Table 2-3: Constant Values Produced by the Status Register

As	Constant	Notes
00	-	Register mode
01	(0)	Absolute addressing mode is realized by using the SR with As equal “00”
10	0x0004	Acts as a constant generator. Constant +4 is produced.
11	0x0008	Acts as a constant generator. Constant +8 is produced.

2.2.1.1.4 Constant Generator2-CG2/R3

R3 is a constant generator. The constant value generated by the R3 is determined by the source addressing bits (As) located in the instruction word as shown in Table 2-4. Note that writing to the SR is not forbidden but the value indented to be written on the register will be ignored.

Table 2-4: Constant Values Produced by R3

As	Constant	Notes
00	0x0000	Constant 0
01	0x0001	Constant +1
10	0x0002	Constant +2
11	0xFFFF	Constant -1

2.2.1.2 General Purpose Registers (from R4 to R15)

There are twelve general-purpose registers. All of them can be employed to store data, addresses of the operands, or index values. In addition, byte, word and bit operations can be performed on data stored on these registers.

2.2.2 Instruction Set Architecture

Instruction set is defined as a set of instructions, which can be executed by a specific processor. MSP430 assembler has 51 instructions in total, which are divided into two main groups. These are 27 *core instructions* decoded by the control unit and 24 *emulated instructions*. Moreover, the core instructions are divided into three sub groups, namely *double operand*, *single operand* and *jump*

instructions as depicted in Figure 2-10 below. All of these instructions have constant bit length and they have highly orthogonal architecture.

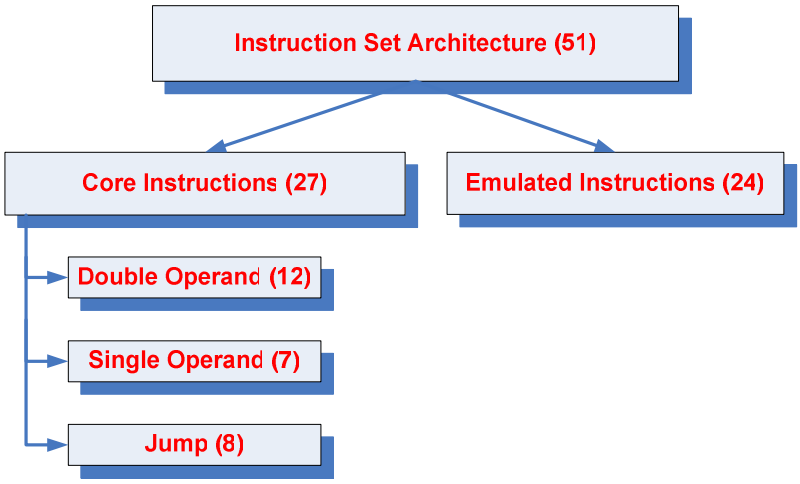

Figure 2-10: Instruction Set Architecture of the MSP430 Processor Core

2.2.2.1 Core Instructions

Core instructions could be described as instructions, which have unique op-codes decoded by the control unit of the CPU. The processor core of MSP430 owns 24 core instructions, whose instruction words are constant 16-bit long. There are three types of core instructions as listed below:

- Double operand instructions
- Single operand instructions, and
- Jump Instructions.

2.2.2.1.1 Double Operand Instructions

As the name suggests, double operand instructions have two operands, namely, source and destination operands, in order to perform the defined function. Figure 2-11 below displays the format of the instruction word.

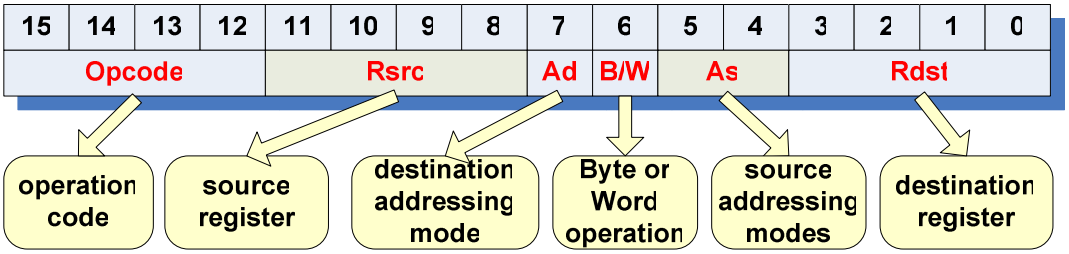

Figure 2-11: Double Operand Instruction Format

As shown in the chart above, there are six parts to define the instruction completely. To begin with, the *op-code* field determines the operation of the instruction. It is directly related with the input that is in charge of determining the function of the arithmetic and logic unit. Secondly, the part *As* together with the source register determines the source-addressing mode. If the *As* part itself determined the source-addressing mode, there would be only four source addressing modes. However, as a result of the fact that both the part *As* and *source register* decide on the addressing mode cooperatively, there are seven source-addressing modes, namely,

- Register direct addressing mode (R)
- Register indirect addressing mode (@R)
- Register indirect addressing with auto-increment mode (@R+)
- Immediate addressing mode (@PC+ or #N)
- Indexed mode addressing mode (X(R))
- Symbolic addressing mode (X(PC))
- Absolute addressing mode (X(0))

Thirdly, the part *Ad* and *destination register* together determine the destination-addressing mode. The number of addressing modes of the destination operand increases from two to four by the help of destination register. The destination-addressing modes are listed below:

- Register direct addressing mode (R)
- Indexed mode addressing mode (X(R))
- Symbolic addressing mode (X(PC))
- Absolute addressing mode (X(0))

Next part is called *B/W*, which denotes the byte or word operation. What is more, byte operations are active high, and word operations are active-low. In assembly language, if an instruction is used without an extension or it is used with ending ".w", it means that the instruction is in *word* form. Moreover, if the suffix ".b" is used, it refers to the *byte* operation.

Apart from *B/W*, another part is named *Rsrc* that stands for the source register. As mentioned above, the source register determines the source-addressing mode by being in cooperation with *As* part in the instruction word.

Lastly, *Rdst* denotes the destination register. As explained before, destination register determines the destination operand-addressing mode together with *Ad*.

Please note that the detailed explanations of all addressing modes are given in *Addressing Modes* part of the thesis.

There are 12 double operand core instructions whose mnemonics, explanations of operation, effects on status bits are given in Appendix A.1. BIT(.B) and CMP(.B) instructions only affect the status bits which are abbreviated by V, N, Z, C letters. Besides, these three instructions do not initiate any micro-operation to write back to the destination operand, which means that they do not affect the destination operand whether it is located in memory or register. In addition, all the instructions can be used in byte form as symbolized with ending "(.B)" in the table given in Appendix A.1.

Execution cycles and necessary memory locations of the double operand instructions are expressed in Appendix A.1. Register-to-register operation takes only one clock cycle. When the program counter is targeted except from *symbolic, absolute, and indexed* source-addressing modes, then one extra clock cycle is required to complete the execution of the instruction. When a memory location is referenced, then one more cycle is necessary to fetch the data from the memory. In addition, if the effective address is calculated by using an offset value, then one more cycle is considered essential to fetch this offset value. Moreover, when a memory location is selected as a destination, then one more clock cycle is required to write the result in the memory. As shown in the table, one word memory location is needed to store the instruction and one more word is required for immediate, indexed, symbolic or absolute addressing modes.

2.2.2.1.2 Single Operand Instructions

As the name suggests, these instructions have only one operand. There are seven single operand instructions as given in Appendix A.2 and the format of the instruction word is as illustrated in Figure 2-12 .This format is made up of five major parts.

Firstly, the word begins with a *constant* located in the most significant six bits position. Secondly, the *op-code* part distinguishes the single operand instructions from one another. The op-code is not directly related with ALU operation. However, the op-code of RRC(.B), RRA(.B), SWPB, and SXT instructions are directly used to generate the function selection input of the arithmetic and logic unit. The third part is *B/W*, which is used to select the byte or word operations as in the case of the double operand instructions. Moreover, SWPB, CALL, RETI, and SXT instructions do not have any byte forms.

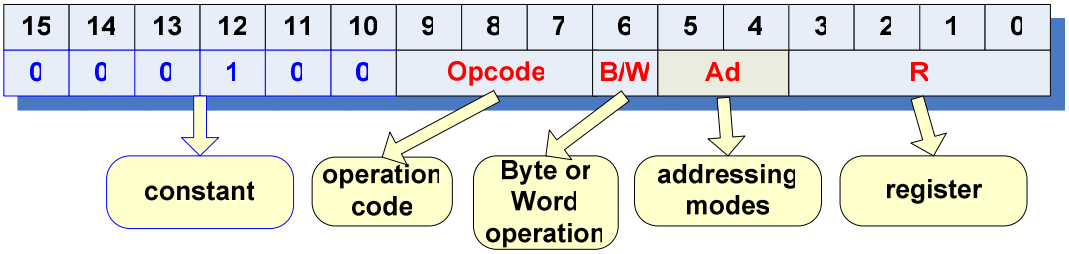

Figure 2-12: Single Operand Instruction Format

The next part is named Ad representing the addressing mode when it is used together with the register. All of the seven addressing modes are possible to be used with the single operand instructions.

The last part located in the least significant four bits in the instruction word is *R* that symbolizes the number of the register used. Rsrc or Rdst symbols are not preferred, because RRC(.B), RRA(.B), SWPB, CALL and SXT instructions regard the register shown in part *R* as a destination register whereas PUSH instruction consider it as a source register.

Appendix A.2 also depicts the execution clock cycles and necessary memory locations of single operand instructions. However, RETI instruction, which needs 5 cycles to complete the execution and requires only one memory word location, is not placed in the table. As displayed in appendix, instruction word occupies one word location in memory, besides; one more word is required to store the immediate constant and offset values of indexed, symbolic and absolute addressing modes.

2.2.2.1.3 Jump Instructions

Jump instructions provide Program Counter (PC)-relative program branching. They can be decoded by inspecting the most significant 4-bits, which start with “0010” or “0011” bits as shown in Appendix A.3. There are, in total, eight jump

instructions as listed in Appendix A.3. This group consists of seven conditional jump instructions and one unconditional jump instruction. They do not change the status information of the processor. On the other hand, the conditional jump instructions use the status bits.

The format of the jump instructions is presented in Figure 2-13. There are three major fields, namely, *constant*, *C*, and *offset*. The constant field is used to distinguish the jump instructions from other instructions. The *C* field refers to the conditions employed by conditional jumps. The offset value is used to calculate the jump address. The 10-bit program counter offset is treated as a signed 10-bit value, which is doubled and then added to the program counter to get effective jump address. The possible jump range from –511 to +512 words relative to the PC value that points the next instruction.

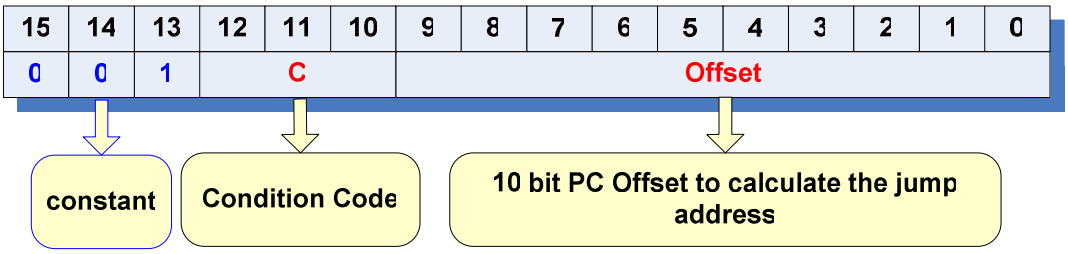

Figure 2-13: Jump Instruction Format

All of the jump instructions can be executed in two clock cycles independent of the condition. It means that whether the condition is met or not, the execution cycles of the jump instructions are two clock cycles. Besides, all of them occupy only one word in the memory.

2.2.2.2 Emulated Instructions

There are 27 emulated instructions as given in Appendix A.4. Emulated instructions makes the instruction set of the processor more flexible and the size of the program denser. Besides, they are generated by using core instructions with constant registers (R2 and R3), programs counter (PC) and stack pointer (SP). Please note that the derivatives of the NOP instruction, which enable more than one clock cycle delay, can be easily constructed. For example, “ADD X(R4), X(R4)” instruction provides six clock cycle delay and the instruction occupies three word locations in the memory.

2.2.3 Addressing Modes

In this part, initially a short description of the addressing mode structure of the CPU is given. Next, all of the possible addressing modes are explained in detail accompanied by examples.

Addressing mode determines how the instructions access their operand or operands. For double operand instructions, there are seven addressing modes for source and four addressing modes for destination operand as seen in Table 2-5 below. At this point, it is right time to explain orthogonality feature. Two or more sets are named *orthogonal* if the elements of sets are unrelated to that of the other sets although the elements of a set may have a relationship one another within the same set. As demonstrated in the Figure 2-14 below, there exist three separate and independent groups, which are made up of double operand instructions, destination operand addressing modes and source operand addressing modes. These three groups can be regarded as orthogonal since they are totally unrelated to each other. That is, all of the double operand instructions can be used with all possible combinations of source and destination addressing modes. As a result of the emergence of 100% orhtogonality, combination of three groups fill the complete possible space as illustrated in Figure 2-14 below.

Table 2-5: Addressing Modes for the Double Operand Instructions

Addressing Modes	Source Operand	Destination Operand
Register Direct	√	√
Indexed Mode	√	√
Symbolic Mode	√	√
Absolute Mode	√	√
Indirect Register Mode	√	X
Indirect Auto-increment Register Mode	√	X
Immediate Mode	√	X

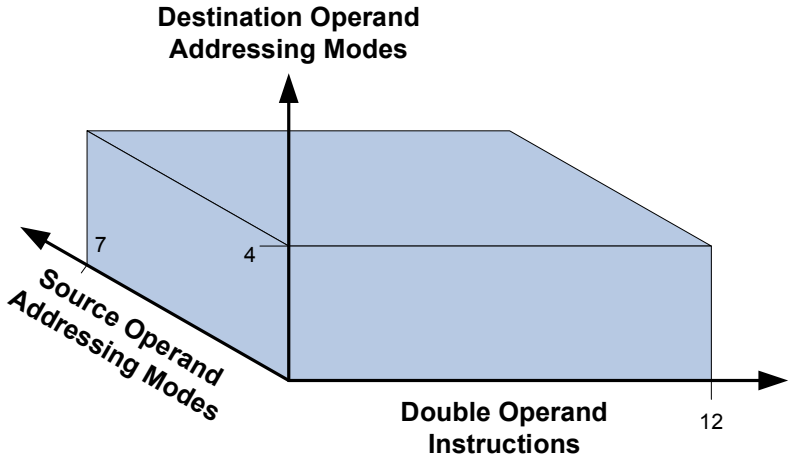

Figure 2-14: Orthogonality Feature of the Double Operand Instructions

Similarly, single operand instructions can be used with all of the seven possible addressing modes as given in Table 2-6. Nevertheless, there are two exceptions, which strays the single operand instruction and addressing mode sets from 100% orthogonality feature as depicted Figure 2-15 below. Firstly, RETI instruction which brings the program execution back to where it was interrupted does not need any addressing mode, because, the operands are implicitly defined in the

instruction word. This kind of addressing mode is defined as *implied mode*. Secondly, RRA, RRC, SWPB, and SXT instructions are not recommended to be employed with immediate addressing mode. In fact, using these four instructions with immediate constants does not make so much sense. Because, there are only a few exceptions, single operand instructions may be called *highly orthogonal*.

Table 2-6: Addressing Modes for the Single Operand Instructions

Addressing Modes	Operand
Register Direct	√
Symbolic Mode	√
Indexed Mode	√
Symbolic Mode	√
Absolute Mode	√
Indirect Register Mode	√
Indirect Auto-increment Register Mode	√
Immediate Mode	√

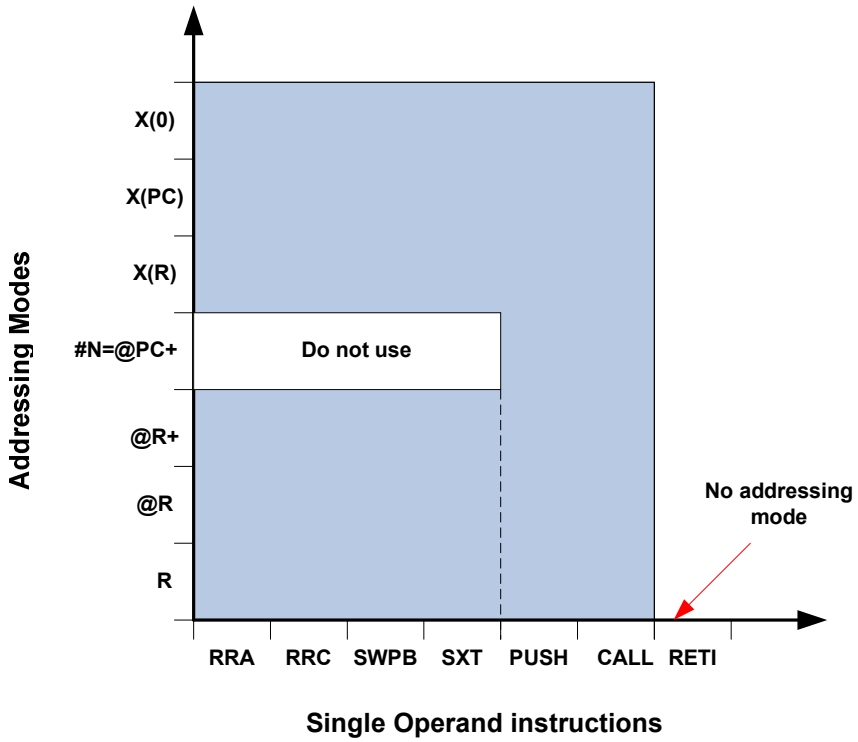

Figure 2-15: Orthogonality Feature of the Single Operand Instructions (Please note that the RETI instruction does not have any operand and immediate mode is not allowed for the RRA, RRC, SWPB, SXT instructions)

2.2.3.1 Decoding Addressing Modes

Addressing modes are determined by the addressing mode fields in the instruction word and the register used. There are two addressing mode fields named as *As* and *Ad* for double operand instructions. Besides, only one field called as *Ad* exists for single operand instructions. The control unit of the processor interprets these fields and the registers together to take necessary actions to fetch operands. Table A-7 and Table A-8 in Appendix A show how the addressing mode fields and registers affect the addressing modes of the double operand instructions. As quite evident in these tables, the symbolic and absolute

modes are special cases of the indexed mode where PC and R2 are used as a register, respectively. Moreover, immediate addressing mode is implied by using the PC with indirect register auto-increment mode. These observations can be used to simplify the control unit of the CPU. Please note that addressing modes of the single operand instructions are same with the source operand addressing modes of double operand instructions. Thus, by replacing As field with Ad and ignoring the *source* word in register part of Table A-7, we get all possible addressing modes for single operand instructions.

As it could be noticed from the information given above, the addressing mode architecture is highly orthogonal where all the instructions within the same group use the same addressing mode structure. It makes the microcontroller unit (MCU) to be programmed easily.

2.2.3.2 Register Direct Addressing Mode

As the name *Register Direct Mode* suggests, the register content is *directly* used as an operand. Register mode is available for all operands. What is more, it is the fastest one as well as being the one that needs the least memory. There are also some other addressing modes where the content of the registers are used indirectly so as to fetch the operands.

If the registers R2 or R3 are used as a *constant source register*, then the source-addressing mode is automatically converted into register direct mode in order to use the generated constant values. On the other hand, the constant “0” generated by R2 is to be used so as to realize absolute mode addressing.

Register mode can be employed for both source and destination operands of the double operand instructions. An example of adding the content of R7 and R8 to R8 is provided below in Figure 2-16 and Figure 2-17. The assembler code for the instruction is “ADD R7, R8”.

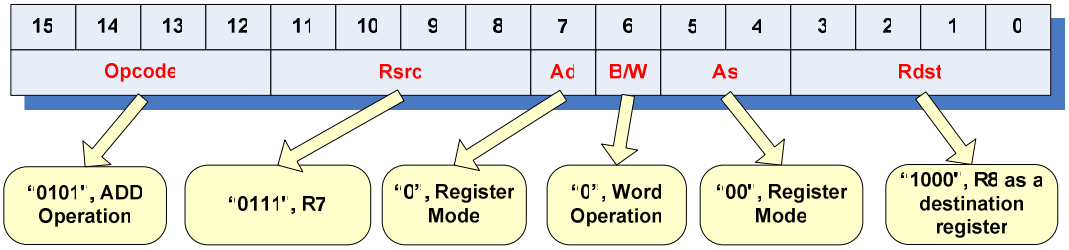

Figure 2-16: Instruction Word of the "ADD R7, R8"

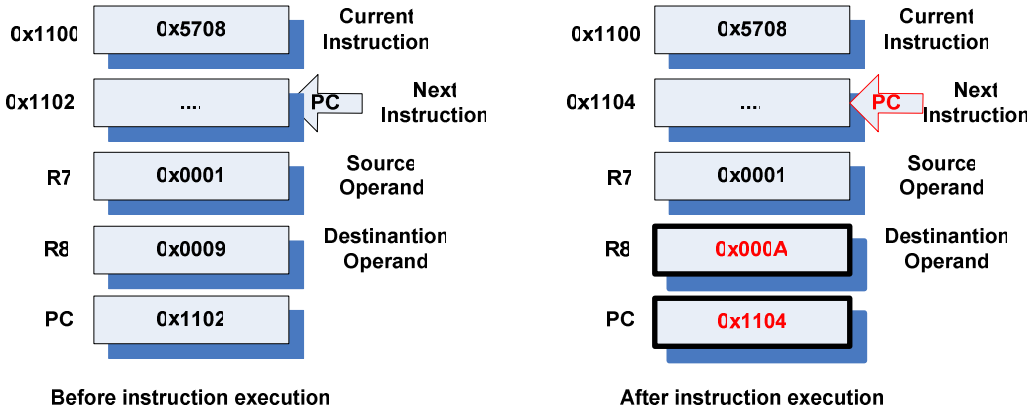

Figure 2-17: Execution of the "ADD R7, R8" Instruction (Rectangles in Bold Depict Changed Locations)

Another example of the register mode used in single operand instructions is given below in Figure 2-19. The content of the R8 register is swapped and the result is stored back to register R8. The assembler code of the operation is "SWPB R8". The instruction word of the instruction is as represented in Figure 2-18 below. The addressing mode filed gets "00" vale for register direct mode. Besides, SWPB instruction can be used only with word operation; byte operation is not available.

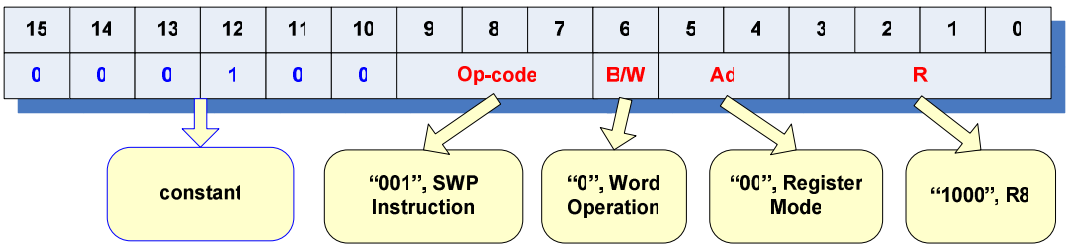

Figure 2-18: Instruction Word of the "SWPB R8"

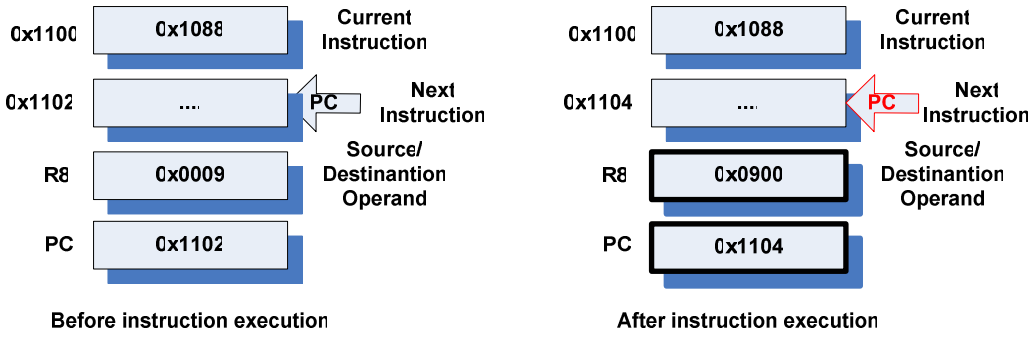

Figure 2-19: Execution of the "SWPB R8" Instruction

2.2.3.3 Indexed Addressing Mode

For indexed addressing mode, an index value is added to the register value to determine the effective address of the operand. This mode requires extra word for index value. Moreover, two memory read operations are needed to fetch both the index and operand values. Furthermore, an addition is to be performed to get the effective address of the operand. As seen in Figure 2-20, *A*d field of the instruction word gets "1", and *As* gets "01" values for indexed source and destination addressing modes.

There is an example to demonstrate the index mode used as a source and as a destination addressing modes. In the example, the content of the memory locations, namely, 0x1800 and 0x1900, are summed and the result is written back to the 0x1900. The instruction words as well as the contents of the related registers and memory locations are depicted in Figure 2-20. Please note that after the instruction word (0x5596), index value for source and destination operands follow respectively.

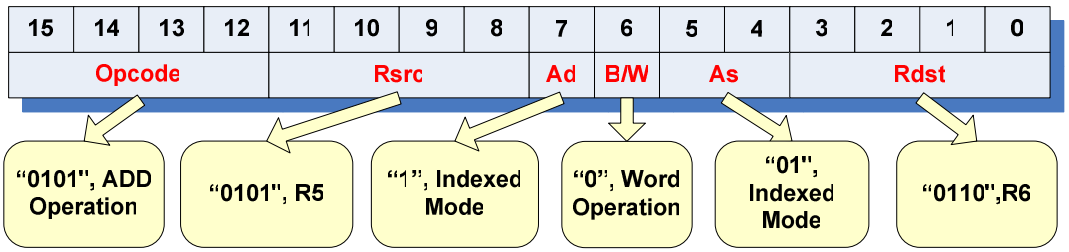

Figure 2-20: Instruction Word of the "ADD 0x0300(R5), 0x0700 (R6)"

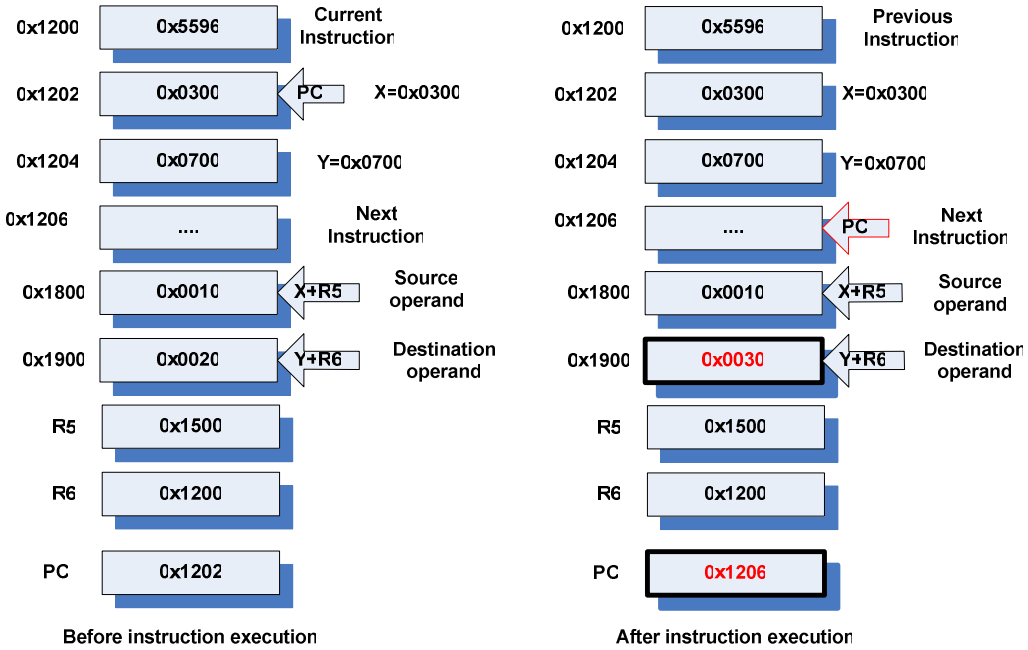

Figure 2-21: Execution of the "ADD 0x0300(R5), 0x0700(R6)" Instruction

2.2.3.4 Symbolic Addressing Mode

Symbolic addressing mode addresses the operands located in the memory relative to the program counter. In fact, it is a special case of the indexed mode where PC is implemented as a source register. The index values, namely, X and Y, are calculated by compilers and these offset values stored sequentially after the instruction word. Symbolic mode brings no difficulty to the hardware implementation of the CPU.

There is an example of the symbolic mode. Because this mode is a special case of the indexed mode, the addressing mode fields are same with indexed mode but the source and destination registers are equal to PC as illustrated in Figure 2-22. The contents of the memory locations 0x1502 and 0x1602 are summed and the result is stored back to the 0x1602 location as given in Figure 2-23.

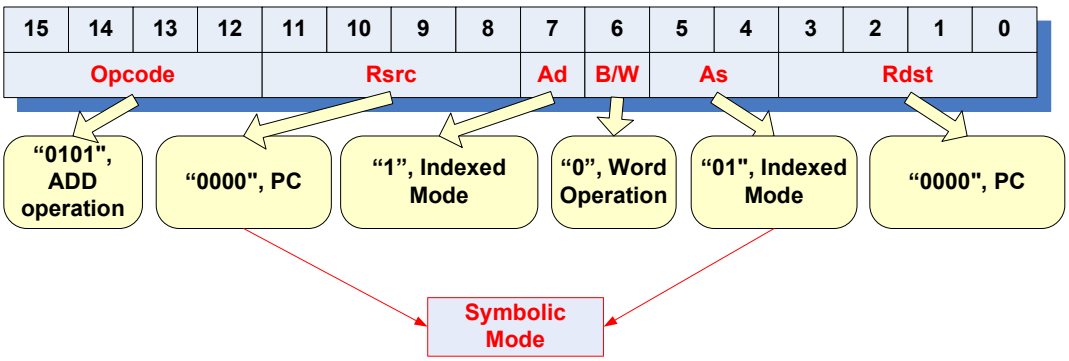

Figure 2-22: Instruction Word of the “ADD ODTU1, ODTU2” Where ODTU1=0x1502=X+PC and ODTU2=0x1602=Y+PC

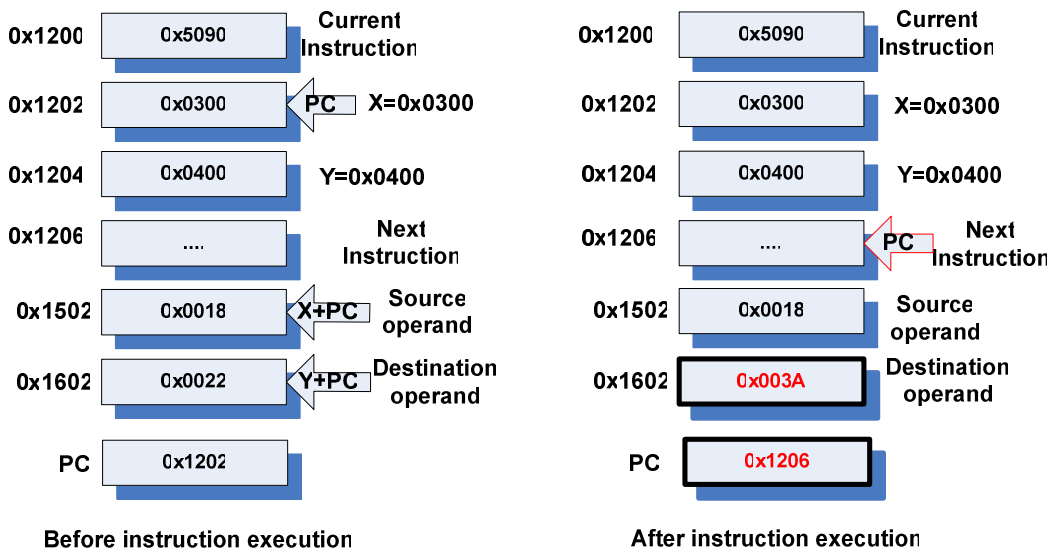

Figure 2-23: Execution of the "ADD ODTU1, ODTU2" Instruction Where ODTU1=0x1502=X+PC and ODTU2=0x1602=Y+PC

2.2.3.5 Absolute Addressing Mode

In absolute mode, the address of the operand is stored in memory location followed by the instruction word. In fact, it is similar to indexed mode where SR is implemented as a register. Note that SR gets zero value for absolute mode (As=1and/or Ad=01). The absolute addressing mode brings no difficulty to the hardware implementation of the CPU. Index values, namely, X and Y, directly represent the effective address of the operands.

Absolute mode (indexed mode with 0 "0(Ry)") can be used as a substitution of indirect register mode with one extra data word for "0" value and one extra cycle to fetch the zero index value from the memory.

An example of the absolute mode is provided below. The addressing mode fields are same with indexed mode but SR is selected as a register as illustrated in

Figure 2-24. The contents of the memory locations 0x1500 and 0x1600 are added and the result is written back to the 0x1600 location as shown in Figure 2-25.

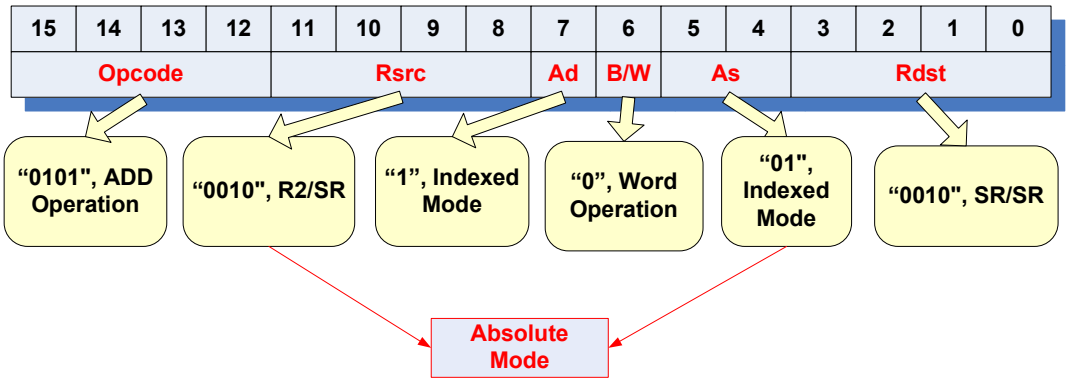

Figure 2-24: Instruction Word of the "ADD &ODTU1, &ODTU2" Where ODTU1=0x1500=X and ODTU2=0x1600=Y

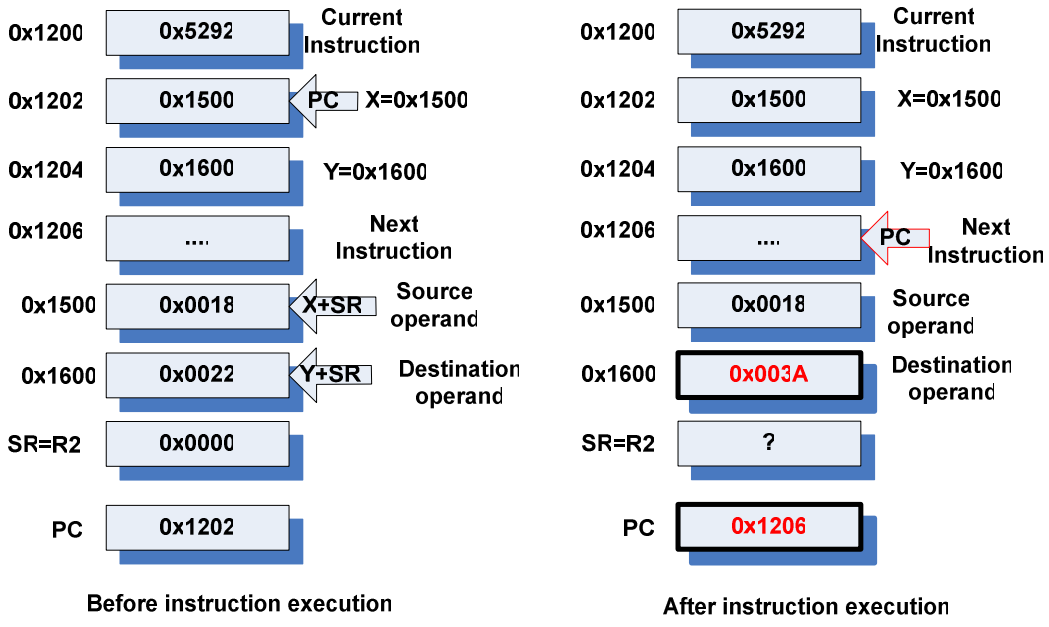

Figure 2-25: Related Register and Memory Contents for the "ADD &ODTU1, &ODTU2" Where ODTU1=0x1502=X and ODTU2=0x1602=Y

2.2.3.6 Indirect Register Addressing Mode

In this mode, the address of the operand is equal to the content of one of the visible CPU registers. Because the registers are 16-bit wide, they can point all the 64KByte memory space. This addressing mode is not available for destination operands of the double operand instructions. Instead of using indirect register mode, indexed mode with zero “0(Rdst)” can be used with one extra data word for “0” value and one extra clock cycle to fetch this value from memory.

An example of the indirect register addressing mode is presented above in Figure 2-27. The instruction is denoted as “ADD @R5, 0(R6)” in assembly language and Figure 2-26 shows the instruction word. The contents of memory locations 0x1500 and 0x1600 are added and the result is written back to the memory location 0x1600 as in Figure 2-27.

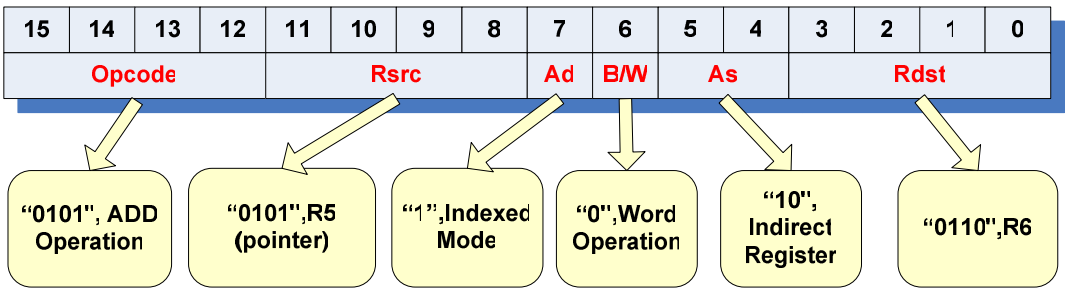

Figure 2-26: Instruction Word of the “ADD @R5, 0(R6)”

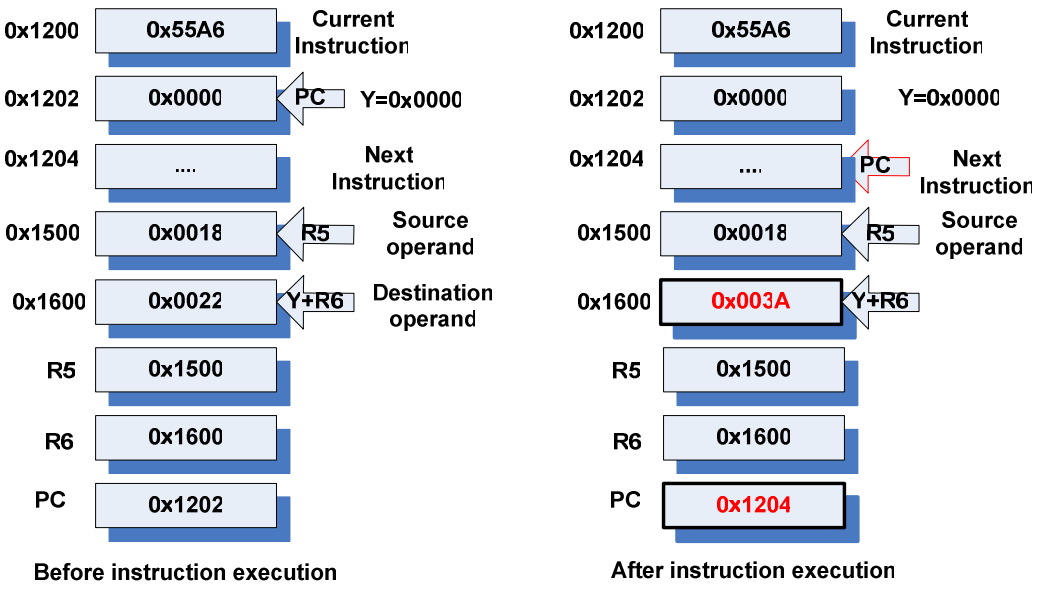

Figure 2-27: Related Register and Memory Contents for the "ADD @R5, 0(R6)" Instruction

2.2.3.7 Indirect Register Auto-increment Addressing Mode

This addressing mode is similar with the indirect register mode but the source operand is incremented by "2" for word operations, and incremented by "1" for byte operations. This mode is suitable for table based processing operations. This addressing mode is not available for the destination operands of the double operand instructions as indirect register addressing mode. There is not any direct compatible form of this addressing mode.

An example of the addressing mode is given below. The instruction word is as illustrated in Figure 2-28. The contents of the memory locations 0x1500 and 0x1600 are added and the result is written back to the memory location 0x1600. Besides, the content of the source register is incremented by two as in Figure 2-29. Please note that for word operations, the source register is incremented by two.

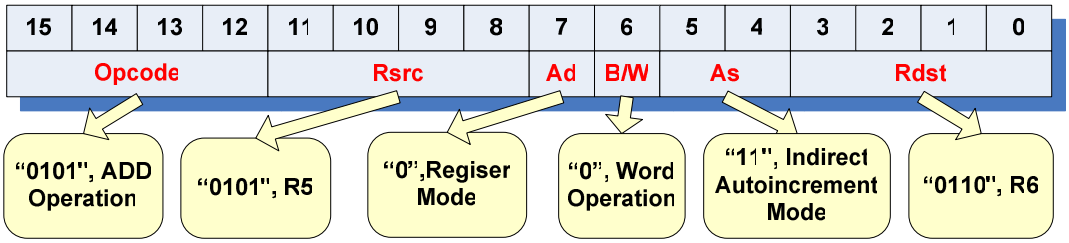

Figure 2-28: Instruction Word of the "ADD @R5+, R6"

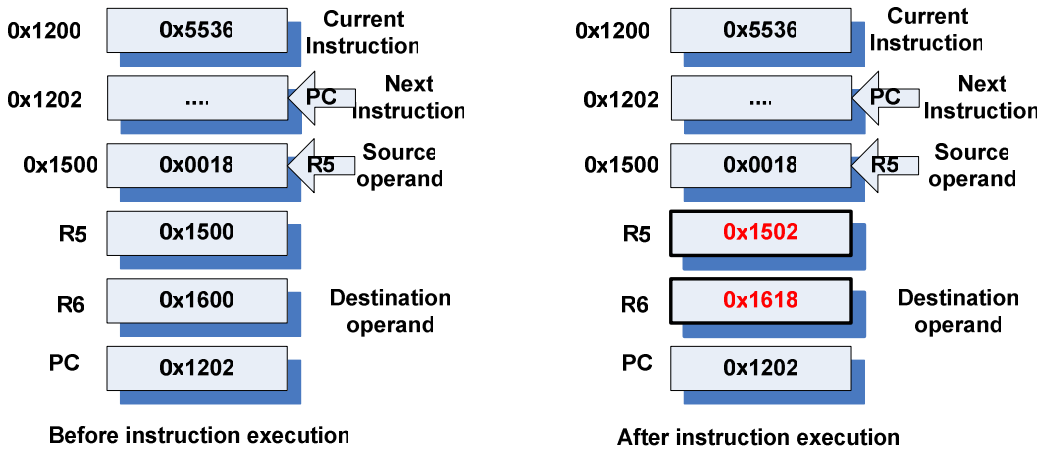

Figure 2-29: Related Register and Memory Contents for the "ADD @R5+, R6"

2.2.3.8 Immediate Addressing Mode

Immediate addressing mode refers the immediate constant that follows the instruction word as an operand. Indeed, the immediate constant is implicitly addressed by using PC as a source register of the indirect register auto-increment mode. It is not available for the source operand of the double operand instructions. Indirect auto increment mode with PC as a source register is used to denote this mode.

An instruction, which uses immediate mode, is given as an example below. The format of the instruction is as presented in Figure 2-30. In this example, #0x1234 constant value is added to the content of the register R6, and then the result is stored back to the R6 register as in Figure 2-31.

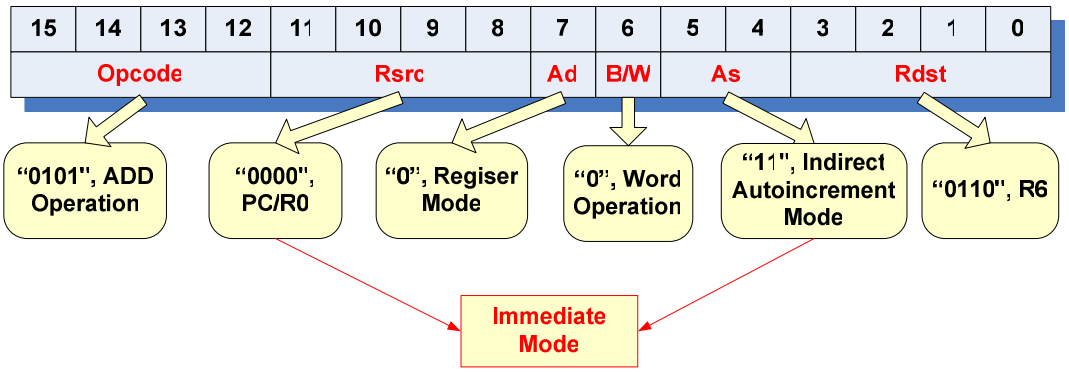

Figure 2-30: Instruction Word of the "ADD #0x1234, R6"

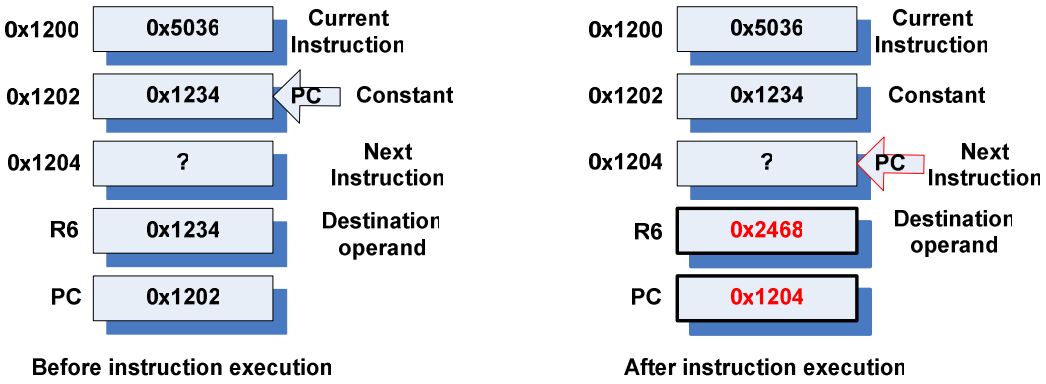

Figure 2-31: Execution of the "ADD #0x1234, R6" Instruction

CHAPTER 3

DESIGN OF MSP430 CPU CORE MODELS WITH SYSTEMC

3.1 Introduction

Two processors, which are compatible with the instruction set of the original MSP430 CPU, are implemented with SystemC. Classical hardware modeling approach, which means RTL/behavioral abstraction, is used throughout the design cycles.

Design flow of the processors is as depicted in Figure 3-1 below. Firstly, the instruction set architecture (ISA) and available addressing modes of the processor were inspected. The instruction set was divided into manageable groups by considering the implementation state. After that, the datapath requirements were determined. Following the verification of the prototype in terms of time, design process of the control unit was started. Lastly, the designed processors were implemented in SystemC. Verification phase was performed by comparing the results of the simulation in the SystemC environment and the simulator of MSP430, which is embedded into IAR MSP430 Cross Complier. The verification phase of the processors is explained in the *Verification of the SystemC CPU Cores* part of the thesis.

In this chapter, the first processor, namely, processor-1, is mentioned briefly and the disadvantages of the first approach are expressed. Note that the details of the processor-1 are given in Appendix B. Next, the second CPU (processor-2) is explained in detail.

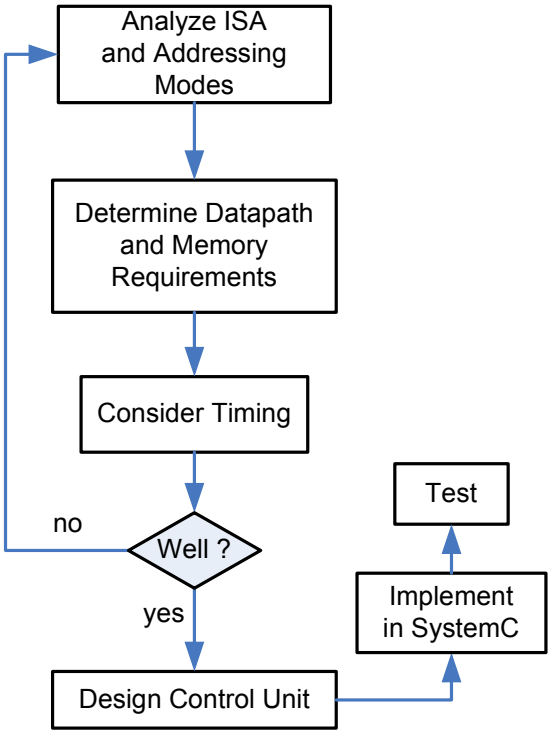

Figure 3-1: Design Approach

3.2 Processor Cores

The first processor, processor-1, suggests instruction-pipelining scheme. To be clearer, while the current instruction is in its last execution phase (IE), the next instruction word is fetched (IF) and decoded (ID) by the control unit. Figure 3-2 illustrates the execution cycles and phases.

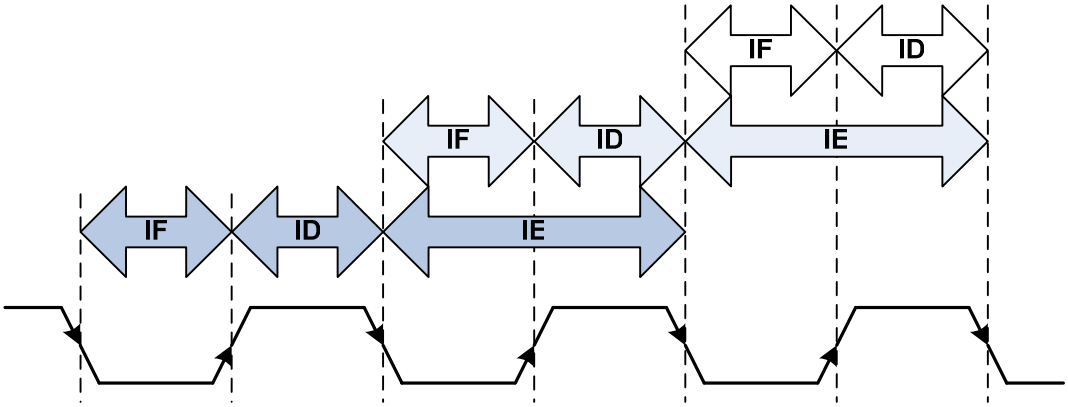

Figure 3-2: Execution Cycles of the Processor-1

Figure 3-3 illustrates the execution of the processor-1 with two instructions. Initially, the instruction word 4324h (mov #2, R4) is fetched from the memory, which lasts within one low part of the clock period. Next, fetched instruction is decoded by the control logic. This phase is also completed within one-half of the clock period. Then, instruction is executed in arithmetic and logic unit, which needs one clock period. At the falling edge of the clock, the calculated data, number 2 in this example, is stored in the destination register (R4). Beside, at the execution phase of the instruction “mov #2, R4”, next instruction (4225h, “mov #4, R4”) is fetched and decoded.

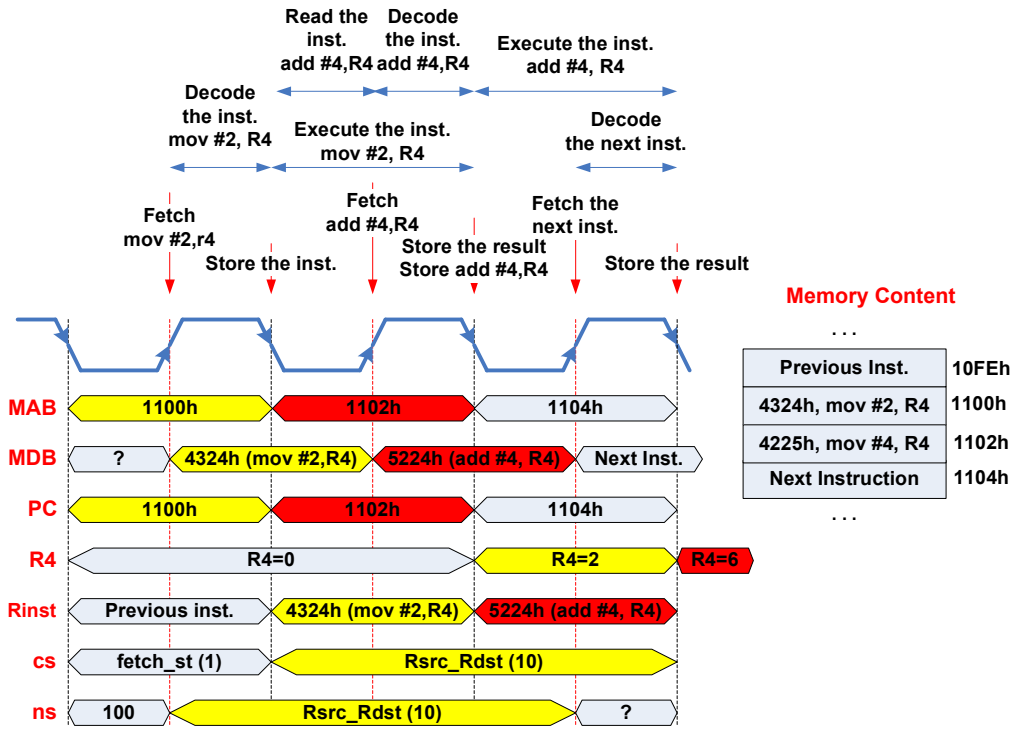

Figure 3-3: Example for the Processor-1

Figure 3-4 illustrates the original datapath of the MSP430 processor core. Besides, the datapath architecture of the first CPU is shown in Figure 3-5. In conclusion, pipelining issue makes the control unit and datapath more complex as will be explained in the following two paragraphs.

All of the general-purpose registers can be used as a pointer with the indirect register auto-increment addressing mode. If a common internal register could be used to load the memory address bus and if it was possible to copy the content of the pointer register to this register, then there might be no reason to enable all of the registers to access the memory address bus. It means that the entire pointer registers, except the program counter, could access the memory address bus with a common internal register. However, according to the designed first CPU, while performing instructions with the indirect register auto-increment addressing mode (@Rsrc+), both of the source and the destination busses are in use at *fetch* state.

Thus, there is no path to copy the content of the pointer register. In addition, there is no event that the content of the pointer register can be incremented. As a result, entire registers have to access the memory data bus with three-state buffers, which results in 15 more buffers and one more decoder in datapath logic.

Two major problems make the control unit more complex. Firstly, the instruction fetch and decode operations are to be performed at two more states in addition to the *fetch* state. Secondly, the necessity of the existence of the memory address bus buffers results in more complex control logic.

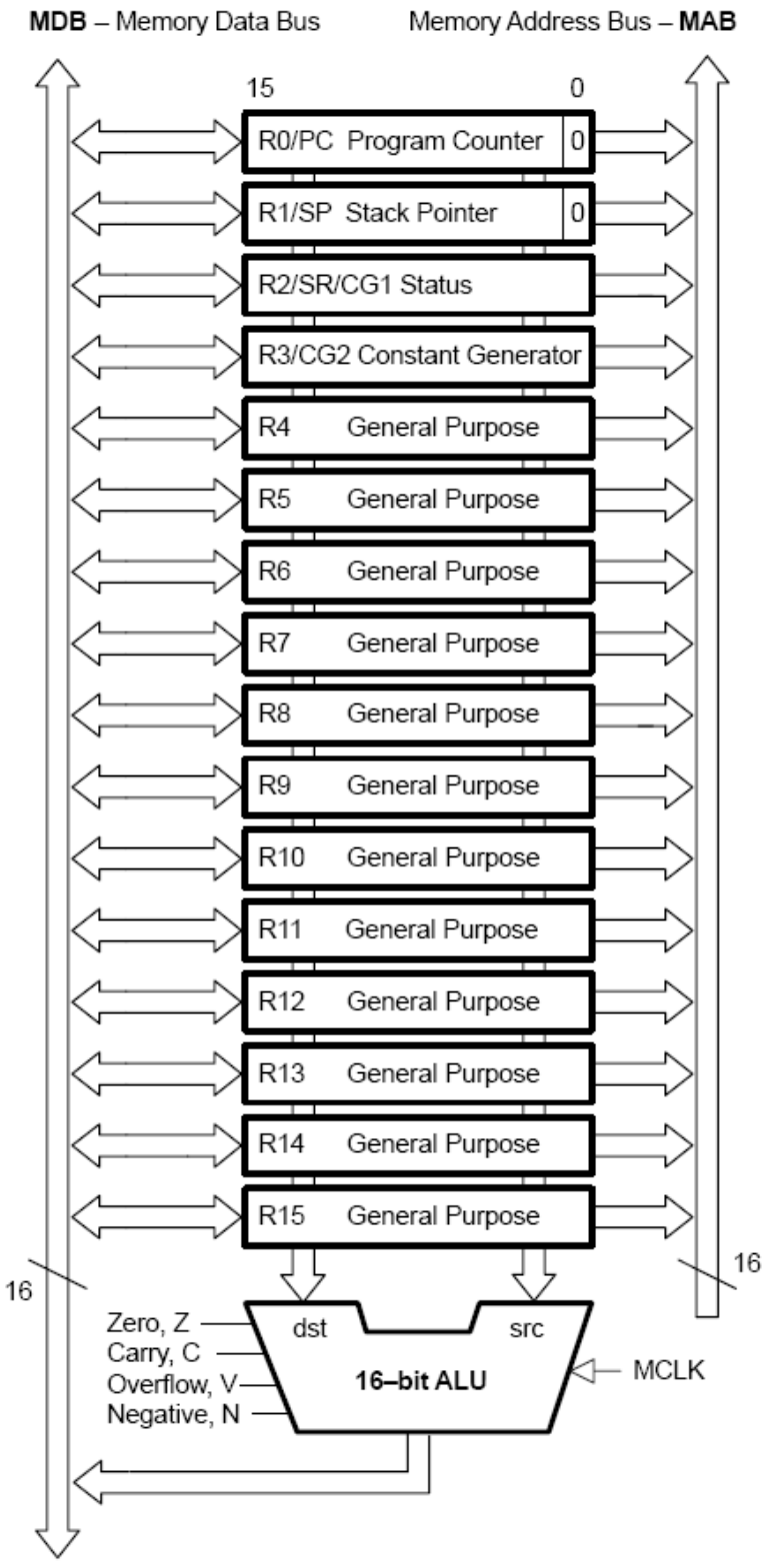

Figure 3-4: Original Datapath Architecture [25]

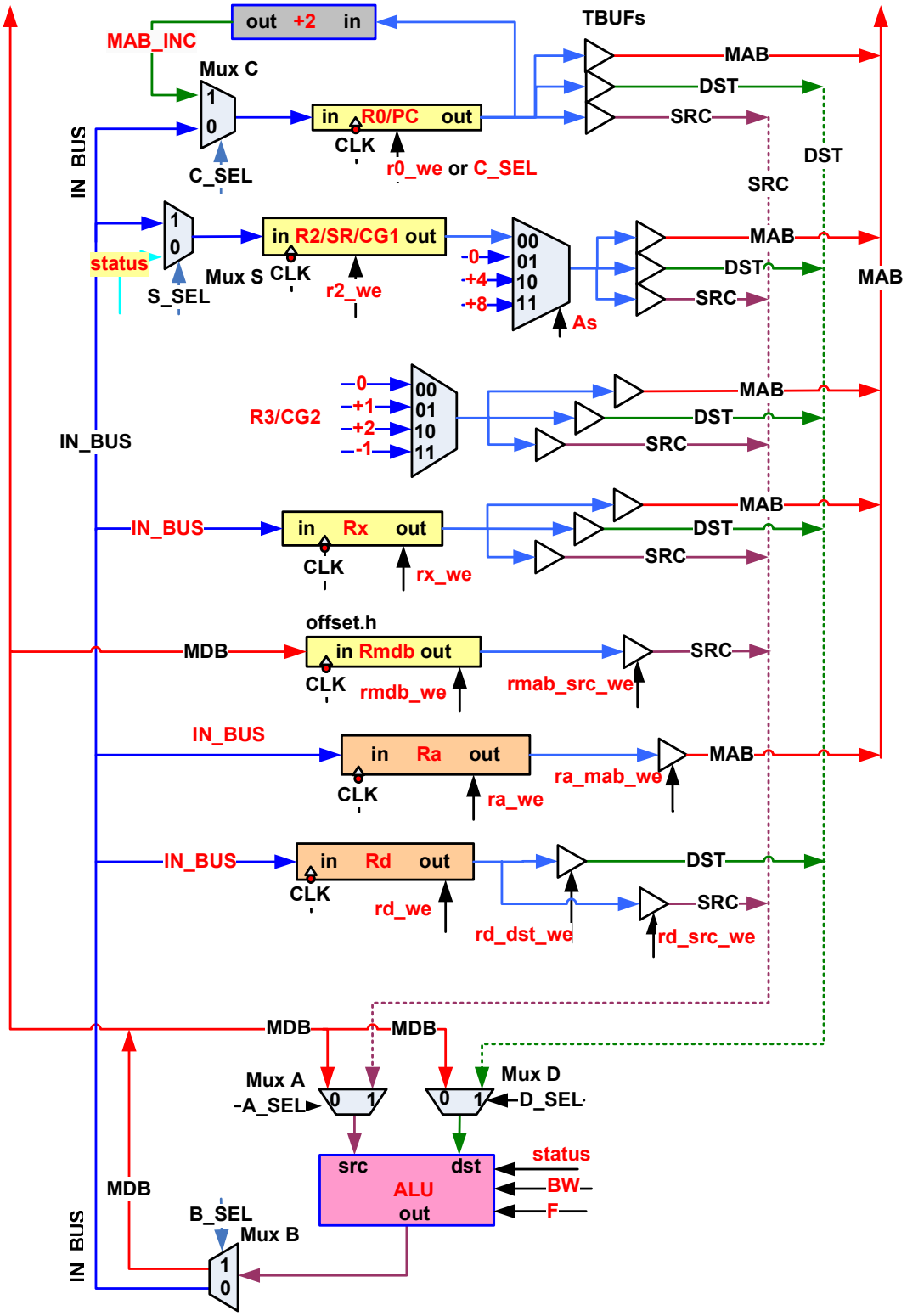

Figure 3-5: Datapath of the Processor-1

Some important simplifications are done with the second approach (processor-2), namely, modifying datapath and control logic. The improvement results that while the instruction is fetched from the memory at one-half of the clock period, the instruction decode and the execution phases can be done at the other half of the clock period as demonstrated in Figure 3-6 below.

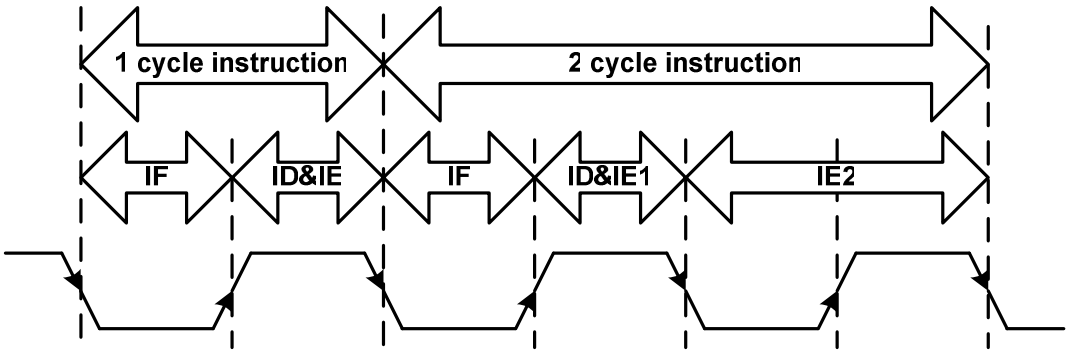

Figure 3-6: Timing of the Execution Phases of the Processor-2

To illustrate this approach Figure 3-7 displays the timing diagram of the execution of the double operand instruction- “AND Rsrc, Rdst” through the use of the letters from “A” to “E”. The sequential operations that are represented by the letters are explained in order as follows:

Firstly, when the falling edge “A” occurs, the content of the program counter (PC) is loaded with the address of the instruction. Secondly, during the time interval “B”, memory read operation is initiated by means of loading the memory address bus and applying appropriate control signals to the memory. Thirdly, at the rising edge event “C”, fetching operation is completed, which results loading the instruction to the memory data bus (MDB). Next, during the time interval “D”, control logic decodes the instruction and generates the required control signals to perform the single-cycle instruction. The dataflow through the datapath is expected to end before the falling edge, event “E”. Lastly, at the falling edge

event, namely, “E”, the calculated result is loaded to the destination register. Besides, the content of the program counter is incremented by two to point the next instruction. Taking the information given above into consideration, we can draw the conclusion that there is no clock latency between the operations; fetch of the instruction and start of the execution. Moreover, this architecture makes it possible for the single-cycle instructions to be carried out easily. Furthermore, as will be seen this approach simplifies the control and datapath logic significantly.

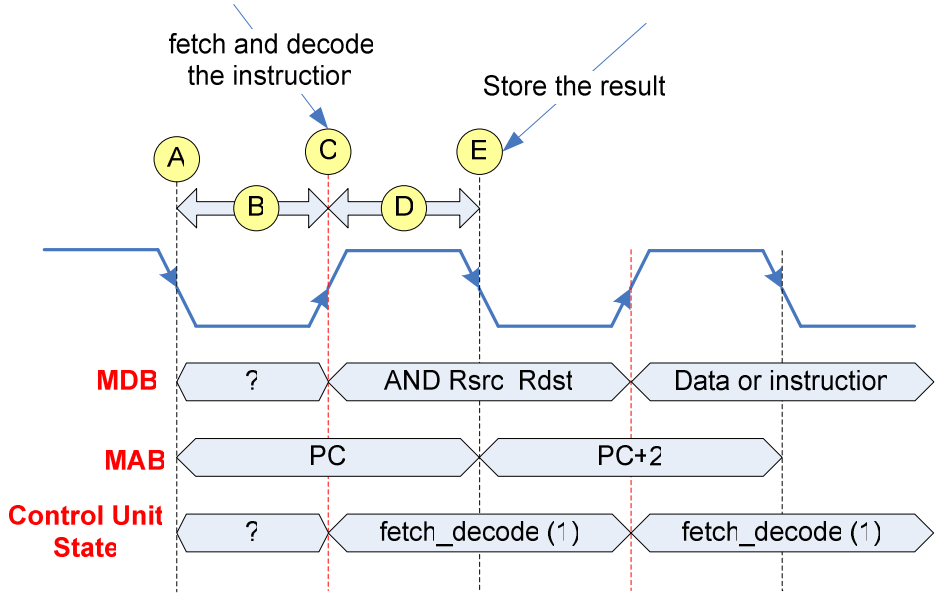

Figure 3-7: Timing Waveforms of the Instruction Execution Cycles

Depending on the second approach (processor-2), a CPU is designed and implemented with SystemC. The CPU implementation can be divided into three major groups as *datapath*, *control unit* and, *memory/stimulus* which are displayed in Figure 3-8 including the dataflow among these groups. Besides, datapath includes *register block, multiplexers* and *ALU* as sub modules. These five major parts are connected to each other in the “main.cpp” part of the SystemC code. (Please refer to “main. cpp” file of processor code).

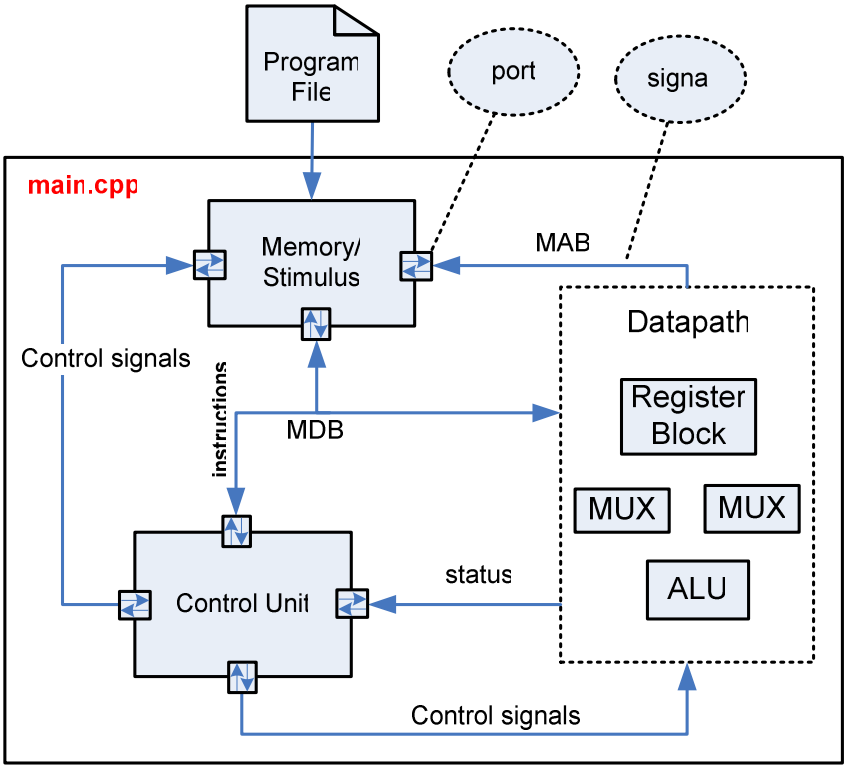

Figure 3-8: Overview of the Designed Architecture

3.3 Datapath Architecture

Datapath is the path along which data flows inside the CPU while being processed. The datapath architecture of the CPU is designed and implemented as two main parts: (Figure 3-9)

- Register block and
- Arithmetic-logic unit

In this section, the block function of the datapath and the interaction between the register block and ALU are briefly summarized. Next, these two main parts are explained in detail in the following subsections (please refer to “main.cpp” file of processor code).

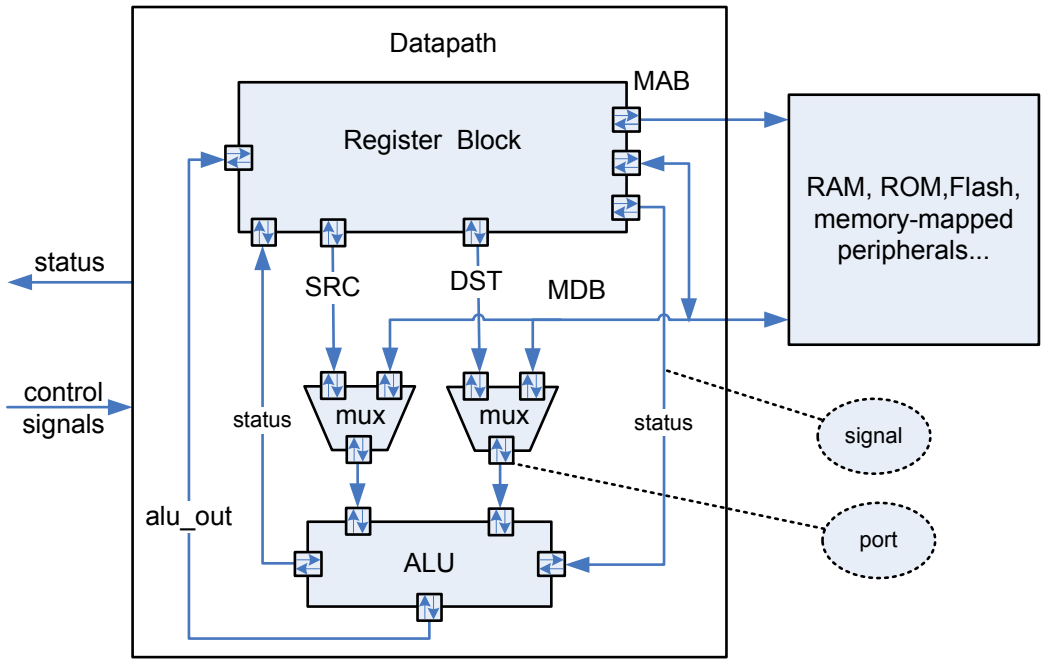

Figure 3-9: Dataflow Diagram of the Datapath

The data flow through the datapath is controlled by the *control unit* and the *status information* of the CPU is feedback to the control logic. Besides, the operand that is going to be processed in the ALU comes from the CPU registers, or outside the CPU. Similarly, the processed data can be written to a CPU register or can be written to somewhere outside the CPU. As seen in Figure 3-9 below, the datapath interacts with the peripherals and the memory through the memory bus. The data from outside the CPU is multiplexed with the source (SRC) and the destination (DST) busses. If the memory data bus was registered and then it was loaded to the ALU, one clock-cycle delay would occur then, which is not proper according to the original instruction timing. Dataflow from the registers to the ALU is performed with the source bus and the destination bus. Besides, the output of the ALU is feedback to the registers with a signal called *alu_out*. A bus system can be constructed with three-state buffers or with multiplexer. The three-state buffer method is used to construct buses in the implementation stage of the SystemC processor cores.

3.3.1 Register Block

Because registers are regarded as the most important part of this module, it is called “register block”. Besides, it consists of an increment module, buffers, multiplexers, decoders, and some basic logic gates as shown in Figure 3-10. After a short overview of the register block, CPU register architectures and decoders are explained in detail as separate subsections (please refer to “reg_block.h” file of the SystemC project files).

All of the registers load the busses with three-state buffers. Buffer enable signals of the CPU registers are directly connected to the control unit or they are connected to the outputs of the decoders. As seen in Figure 3-10, there are only two decoders named source decoder and destination decoder. Source decoder outputs are directly connected to the appropriate enable signals of the output buffers. Outputs of the destination decoder are directly connected to the enable signals of the destination buffers. On the other hand, outputs of the destination decoder are logically *and*ed with *register write enable signal* and the outputs of the logic *and* gates are connected to the enable inputs of the visible registers. Registers as well as the components related to them are explained in detail in the next subsection entitled *Registers of Central Processing Unit*. Decoders, which are discussed in detail in the subsection *Register Block Decoders*, are used to generate enable signals for the output buffers and the datapath registers.

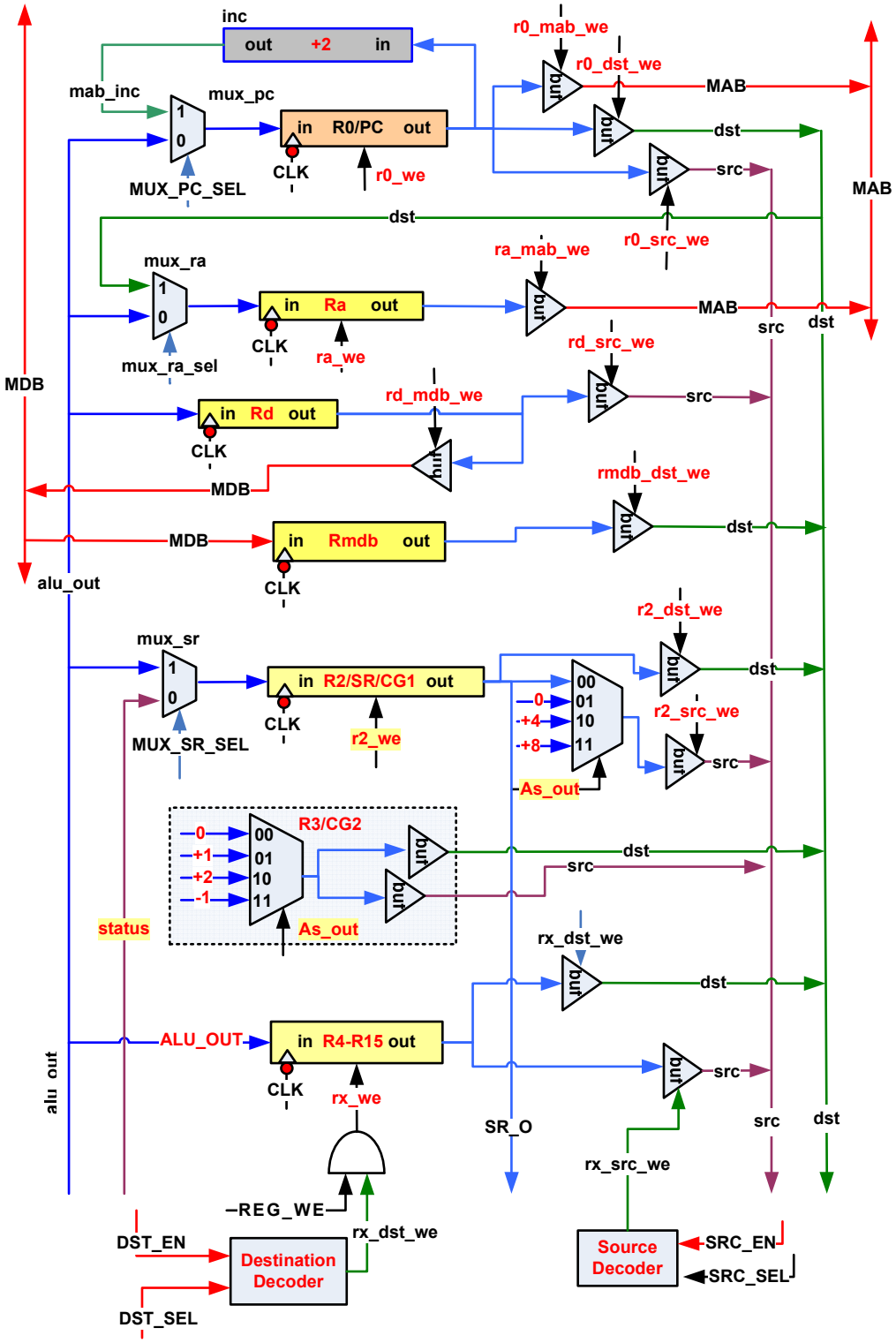

Figure 3-10: Register Block

3.3.1.1 Registers of the Central Processing Unit

MSP430 CPU core implemented in SystemC has 19 registers. All of the registers are 16-bit long and they are triggered with the falling edge of the clock. There are two main kinds of registers with respect to the programmer as shown in Figure 3-11. The first one is visible registers, which can be seen by the programmer. The functions of these registers are as mentioned before in SYSTEMC AND *MSP430 MICROCONTROLLER* part of the thesis. The latter one is invisible registers, which are hidden and they are not visible by the programmer. Invisible registers are employed by the control unit to store some intermediate values while executing some of the instructions. Besides, the only way to access the memory data bus and the memory address bus is to exploit the invisible registers.

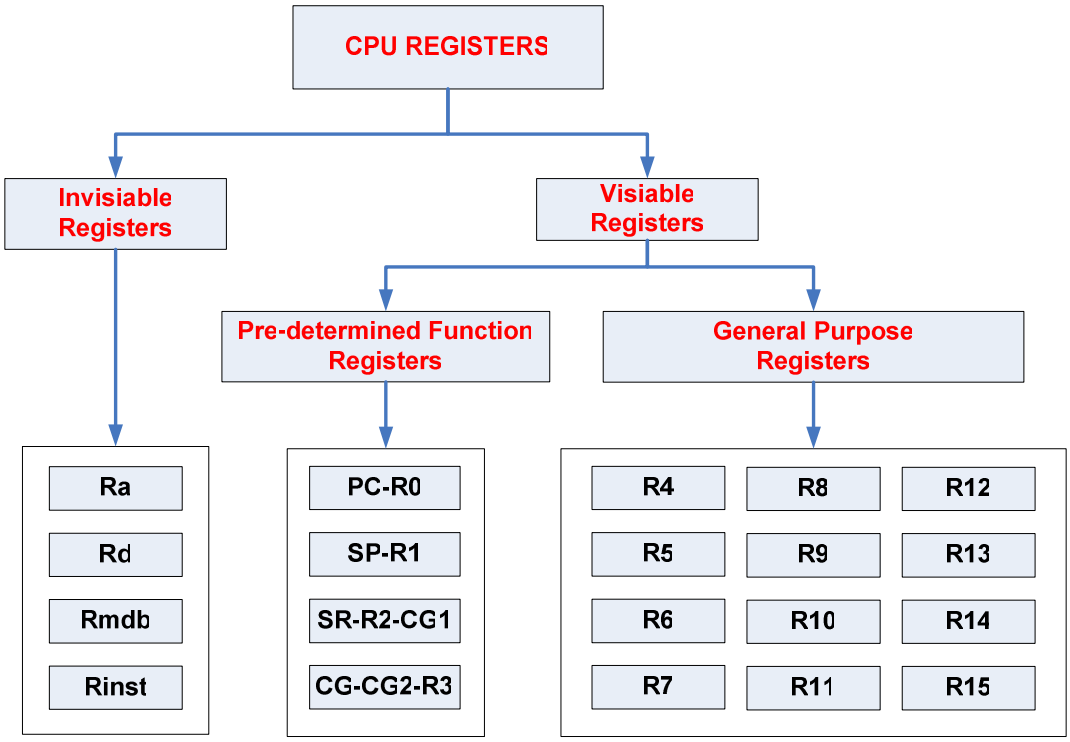

Figure 3-11: Central Processing Unit Registers

3.3.1.1.1 Address Register (Ra)

Address register is symbolized with Ra, and the register and its related components are as shown in Figure 3-12. Ra is one of the two registers that can load to the memory address bus (MAB). The reason for the existence of this register is to access to the memory address bus in order to perform operations that have memory-addressed operands. In other word, this register is used to memory write and memory read operations. Nevertheless, the instruction fetch operations are also performed by the program counter.

The register has a multiplexer called *mux_ra* in front of the data input port. The address content that will be loaded to the register-Ra depends on the addressing mode as shown in Table 3-1. The output of the register is loaded with a tree-state buffer called *ra_ma_buf*. While the select input signal of the multiplexer *mux_ra* is shortened as *mux_ra_sel*, write enable signal of the register-Ra is symbolized as *ra_we*. Moreover, write enable signal of the buffer *ra_mab_buf* is represented by *ra_mab_we*. All three signals mentioned above are directly controlled by the control unit.

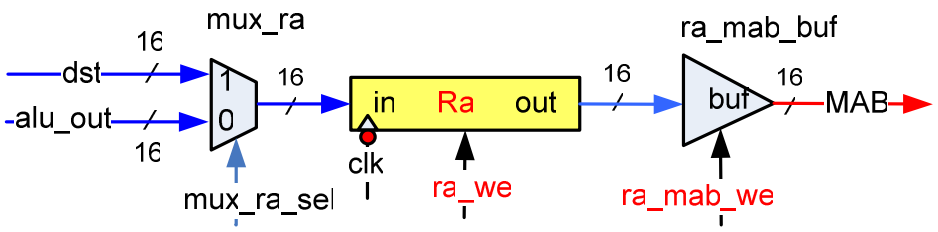

Figure 3-12: Address Register

Regarding Figure 3-12, Ra register can be loaded directly with the content of the destination bus. At first glace, it can be seen that there is no need to use “dst” bus to move some register contents to Ra, because ALU can be used to perform this function. Nevertheless, whiling copying some information to the Ra via ALU, we

do not have any chance to use ALU for other operations. Shortly, this architecture allows the control unit to employ ALU for some operations during a *move* action in order to copy some register contents to Ra. These operations occur while executing indirect auto-increment addressing mode operations. To be clearer, whilst the content of the pointer register is being copied to address register, it is being incremented by the ALU.

Table 3-1: Loading Types of the Address Register with the Addressing Modes

Addressing Modes	Used Bus	Note
Register Mode	-	Ra is not used
Indexed Mode	“alu_out” bus	calculated address is loaded to Ra
Symbolic Mode	“alu_out” bus	calculated address is loaded to Ra
Absolute Mode	“alu_out” bus	calculated address is loaded to Ra
Indirect Register Mode	“dst” bus	direct move from register to Ra
Indirect Auto Increment Mode	“dst” bus	direct move from register to Ra
Immediate Mode	“dst” bus	direct move from register to Ra

3.3.1.1.2 Data Register (Rd)

Data register is the unique register that can drive the *memory data bus* (MDB). Hence, this register loads the MDB bus when *writing something to the memory*. As well, some intermediate values that are calculated while executing some instructions can be loaded to the data register. Then, this intermediate values can be feed back to ALU by using source bus (SRC). The structure of the data register with two output buffers is as displayed in Figure 3-13. *rd_src_we* and *rd_mdb_we* are enable input signals of source and destination buffers respectively. Write enable signal of the data register is called *rd_we*. These three enable signals are directly controlled by the control logic.

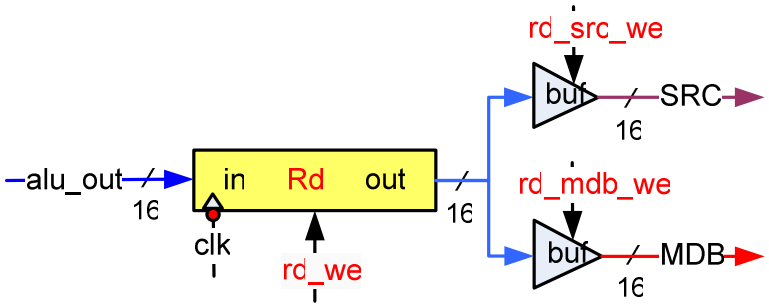

Figure 3-13: Data Register

3.3.1.1.3 Memory Data Bus Register (Rmdb)

Memory Data Bus Register (Rmdb) is *not a pure register* itself as demonstrated in Figure 3-14 below. The combinational logic, which is placed in front of the 16-bit register, shifts the content of the memory data bus and then extends the sign of the shifted value. This register is used to calculate the offset values for the *jump* instructions.

The output buffer of the register loads the destination bus (DST) and the enable signal of the buffer is directly controlled by the control unit of the processor-2.

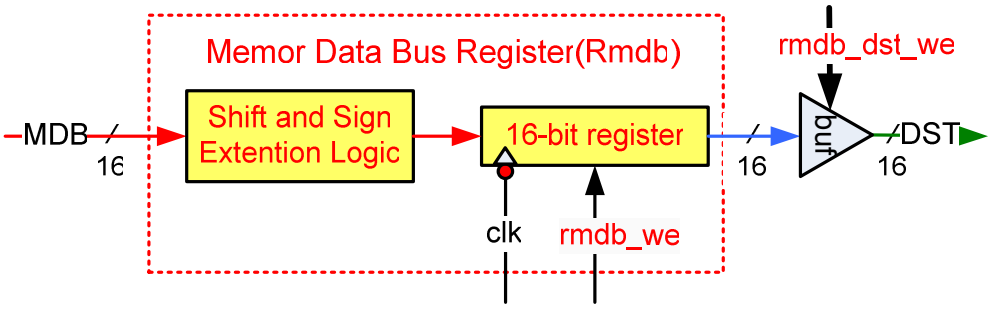

Figure 3-14: Rmdb Register

3.3.1.1.4 Instruction Register (Rinst)

For some instructions, the control unit needs to remember the fetched instruction word. Thus, Rinst register is constructed to store the fetched instruction. During every *fetch and decode* cycle, this register content is updated with the falling edge of the clock. Please note that Rinst register is not an *implemented* register as others; in fact, it is an *inferred* register in the control unit.

3.3.1.1.5 Program Counter (PC)

Program counter stores the address of the next instruction to be executed. One of the two registers that can load memory address bus is PC. Besides, for the *immediate addressing mode* operations, it can also hold the address of the immediate value, which is located in a memory location following the instruction word.

Figure 3-15 shows the register and the related logic components. Like general purpose registers, PC can be loaded with the *alu_out* signal content, which enables the branch instruction to be emulated from the core move (MOV) instruction. For example, “MOV R5, PC” instruction redirects the execution of the program to the location pointed by the R5 register. The output of the register content can be loaded to the memory address bus (MAB), destination bus (DST) and source bus (SRC). PC also can be loaded with the output of the increment module, which is a combinational logic. By the use of this module, the PC content can be incremented by two.

The logic *or* gate enables the PC to be addressed by the op-code with the help of the destination decoder or it allows the control logic to select PC directly as a destination register.

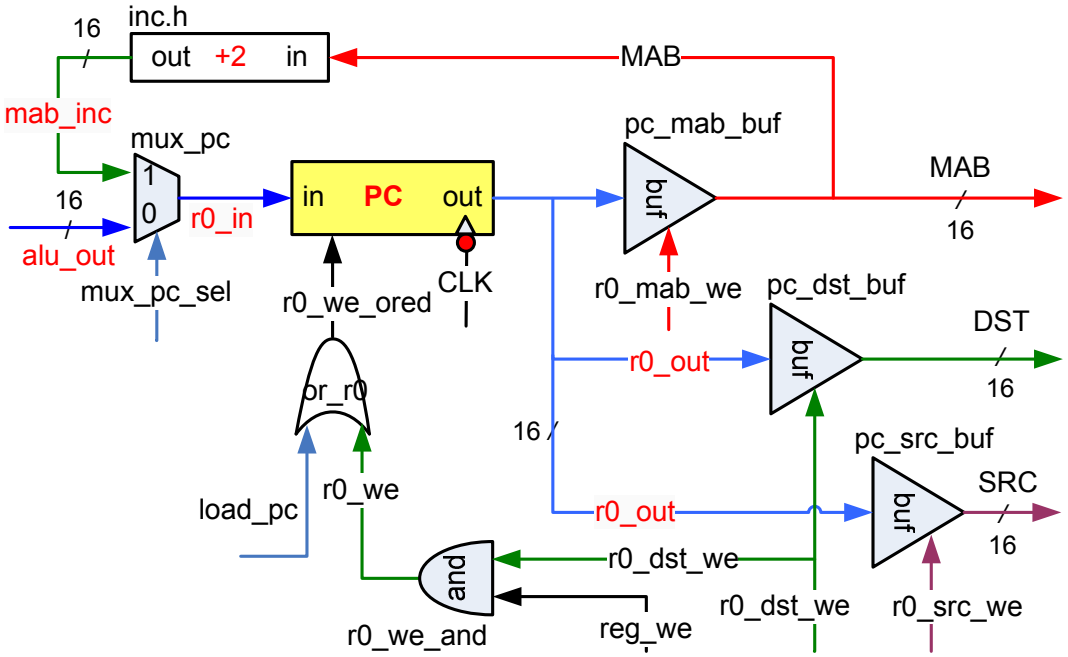

Figure 3-15: Program Counter Register

3.3.1.1.6 Stack Pointer (SP)

The stack pointer is used for the stack operations and it points to the top of the stack. The peripheral architecture of the SP and the operations that can be performed on are entirely same with general-purpose registers. Figure 3-16 illustrates the register architecture.

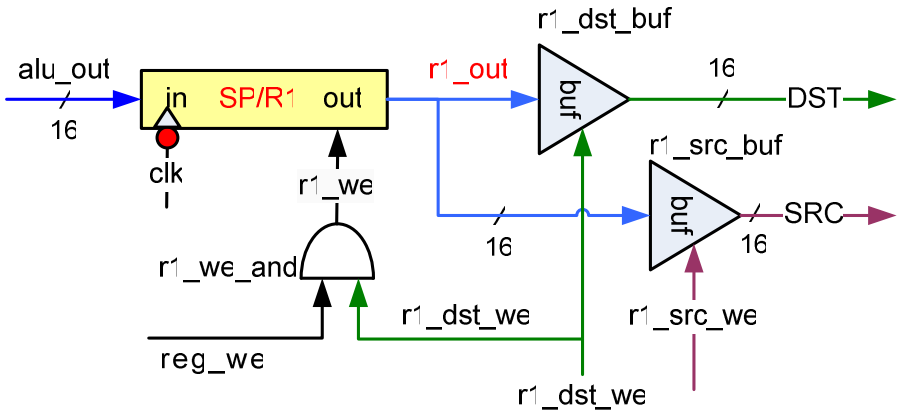

Figure 3-16: Stack Pointer

3.3.1.1.7 Status Register (SR-R2-CG1-Constant Register1)

Status register has two important duties, one of which is storing the status and control information of the processor and the other one is generating 0, +4, and +8 constants. The peripheral architecture of the status register is as shown in Figure 3-17. The constant values can be accessible when the status register is addressed as a source register. That is why, the output of the multiplexer is connected to the source bus buffer. On the other hand, when SR is addressed as a destination register, the content of the SR will be output. In fact, only register direct addressing mode is accessible for SR. Moreover, the constant value “0” generated by the SR is used to realize the *absolute mode* operations.

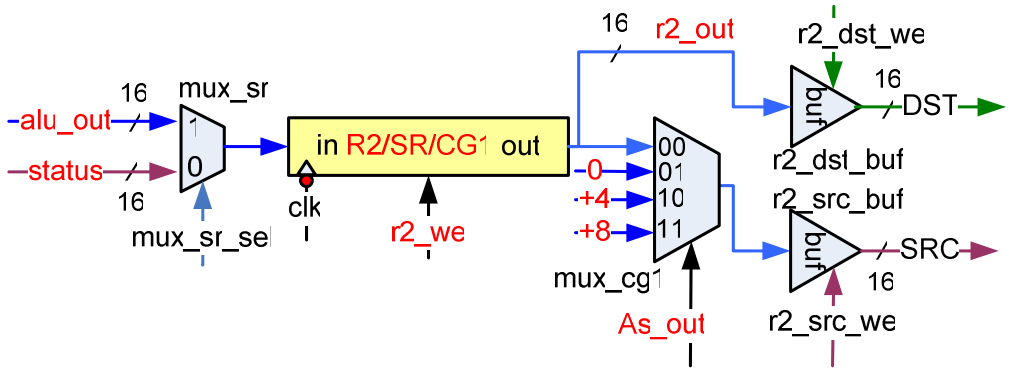

Figure 3-17: Status Register

3.3.1.1.8 Constant Register (R3-CG2)

This register, in fact, has no memory unit. Thus, no value can be hold by the register. It has only a multiplexer that has four constants as inputs as presented in Figure 3-18. These constant values can be addressed as source or destination operand in the register mode. However, the value indented to be written in the register will be ignored. Besides, only register direct mode is available for R3.

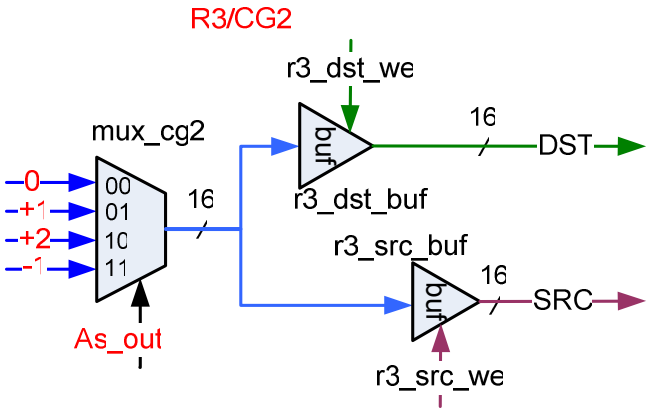

Figure 3-18: Constant Register

3.3.1.1.9 General Purpose Registers (R4-to-R15)

The CPU has 12 general-purpose registers. Figure 3-19 shows the architecture of the registers. Rx refers to the registers numbered from R4 to R15. These registers are used by the programmer to store some calculated values to perform a defined function. Write enable signals of the output buffers and enable signals of the registers are generated by the decoders. Besides, the control unit controls the write operations by the help of *reg_we* signal.

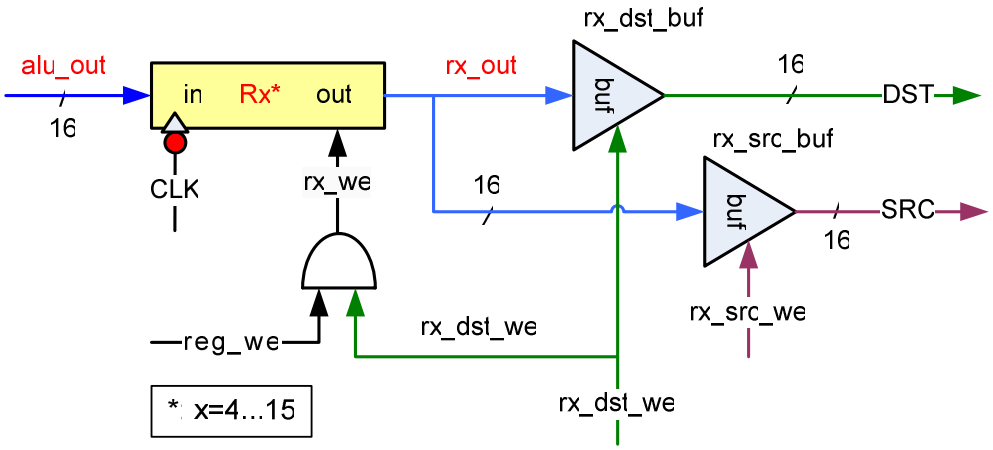

Figure 3-19: General Purpose Registers

3.3.1.2 Register Block Decoders

Decoders are important parts of the register block of the datapath. There are two decoders named *source decoder* and *destination decoder* as in depicted Figure 3-20 below. Decoders are 4-to-16-line, because the source and destination visible registers are coded in the instruction word by using 4 bits.

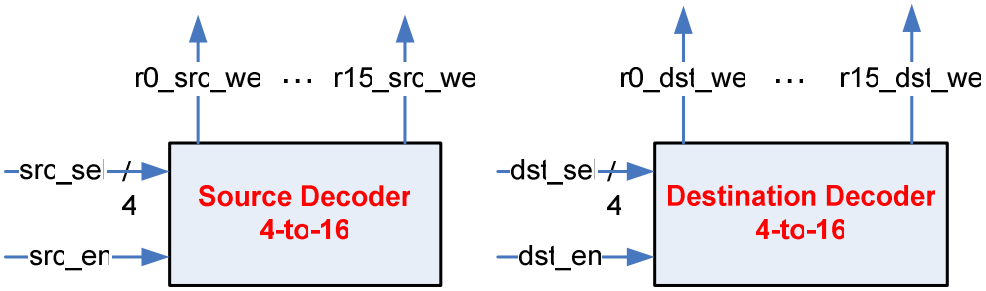

Figure 3-20: Source and Destination Decoders

Source decoder is used to select which register loads to the source bus (SRC). The outputs of the source decoder are directly connected to the enable signals of the source bus buffers.

Destination decoder is used to select which register loads to the destination bus (DST). Enable signals of the destination buffers are directly connected with the outputs of this decoder. Besides, the decoder outputs are logically *and*ed and then connected to the write enable signals of the registers as demonstrate in Figure 3-21 below. Nevertheless, there is no write enable signal connected to the R3 register because this register is not writable. Moreover, as mentioned before, enable signal of the program counter (r0_we_ored) is generated by logically "or"ing the *load_pc* signal with *ro_we* signal to make the PC loaded independently.

Lastly, select and enable inputs of the decoders (src_sel, dst_sel, src_en, dst_en) are controlled by the control unit and the enable signals of the decoders are used to disable the decoders when one of the *invisible registers* are required to load the source or the destination busses.

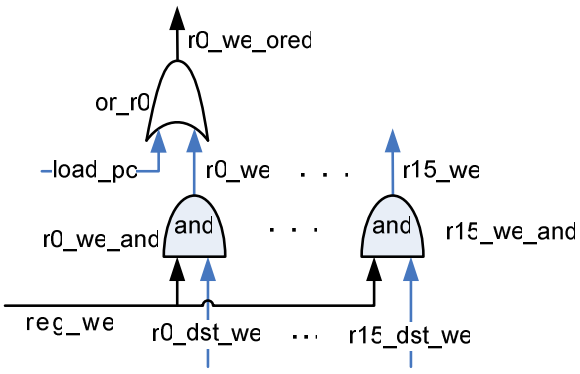

Figure 3-21: Register Write Enable Signals

3.3.2 Arithmetic and Logic Unit

Arithmetic logic unit is where the arithmetic and logic operations are performed within the CPU. Besides, this module also generates status information, which is employed to execute the conditional branch instructions. There are sixteen possible operations, which can be performed by ALU with 4-bit function select input.

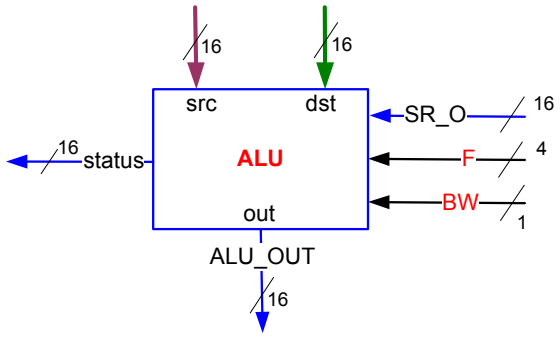

Figure 3-22: Arithmetic and Logic Unit

ALU is not only employed to perform functions determined in the op-code field of the instruction word; it also can be used by the control unit to calculate the increment values and to transfer information. Thus, we cannot directly relate the op-codes with decoding of ALU functions. The tables below list all the operations performed by ALU. In addition, it displays how the status bits are affected by the ALU operations.

Arithmetic-logic unit performs only four single operand operations that are supposed to be located on the destination port, because these four instructions have op-codes directly related with the ALU functions as shown in Table 3-2. As observed from the table only least significant two bits are related with the op-code of the instruction word. In fact, two constant zeros are appended to the least significant two bits of the op-code to generate function selection input of the ALU for single operand instructions-SWPB, SXT, RRA, and RRC as in Figure 3-23. Other four single operand instructions, whose op-codes are not directly linked to ALU functions, are not carried out directly by ALU, instead by the help of ALU.

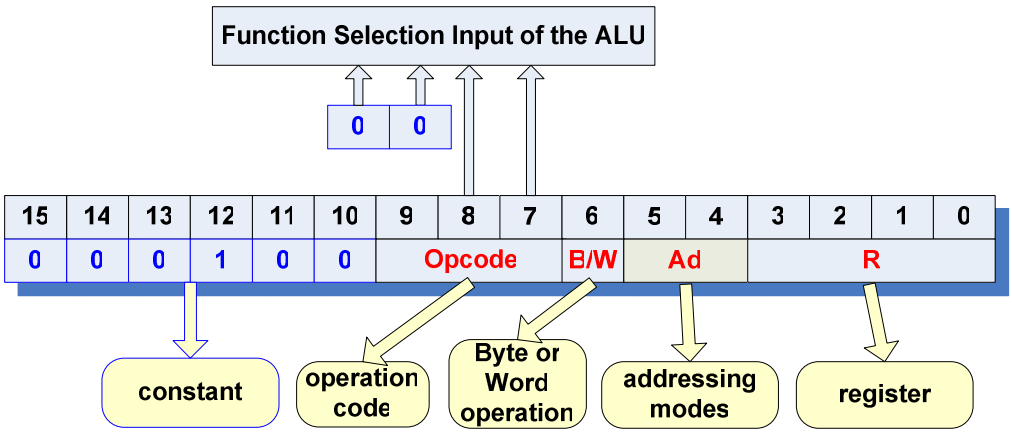

Figure 3-23: Function Selection Input of the ALU for the Single Operand Instructions

Without any exception, operation codes of the double operand instructions are directly related with the ALU functions. Thus, ALU is expected to have 12 operations related with the 12 double operand instructions. The Table 3-3 displays all of these functions. Please note that *BIT* and *CMP* instructions do not initiate a write operation to the destination register. To realize these feature, the output of the ALU, while performing these two instructions, is equal to the data on the destination port. Thus, the content of the destination operand is copied to itself. In fact, these instructions only affect the status information of the processor.

Byte or word operations affect the way to generate the status information and the output of ALU. Note that when byte operations are performed, the most significant 8 bits of the output of the ALU are assigned to zero values.

Table 3-2: Single Operand Instructions Performed by the ALU

Single Operand Operations								
F	**Function**	**Description**	**Operation**	**Output of the ALU**	**Status Bits**			
					V	**N**	**Z**	**C**
0000	RRC dst	Rotate right through C	C➔MSB➔.....LSB➔C	*operator* (dst)	*	*	*	*
0001	SWPB	Swap bytes	(Bit 15...Bit 8)↔(Bit 7 ...Bit 0)		-	-	-	-
0010	RRA dst	Rotate right arithmetically	MSB➔MSB➔ .... ➔LSB ➔C		**0**	*	*	*
0011	SXT	Extend sign	Bit 7 ➔ (Bit 8.....Bit 15)		**0**	*	*	*

Table 3-3: Double Operand Instructions Performed by the ALU

Double Operand Operations								
F	**Function**	**Description**	**Operation**	**Output of the ALU**	**Status Bits**			
					V	**N**	**Z**	**C**
0100	MOV src, dst	Move source to dst	src	src	**-**	**-**	**-**	**-**
0101	ADD src, dst	Add source to dst	dst + src	dst + src	*	*	*	*
0110	ADDC src, dst	Add source and C to dst	dst + src + C	dst + src + C	*	*	*	*
0111	SUBC src, dst	Subtract source and not(C) from destination	dst + ***not***(src) + C	dst + ***not***(src) + C	*	*	*	*
1000	SUB src, dst	Subtract source from destination	dst + ***not***(src) +1	dst + ***not***(src) +1	*	*	*	*
1001	CMP src, dst	Compare source and destination	dst + ***not***(src) +1	dst	*	*	*	*
1010	DADD src, dst	Add source and C decimally to dst.	dst+src+C, BCD	dst+src+C, BCD	*	*	*	*
1011	BIT src, dst	Test bits in destination	dst ***and*** src	dst	**0**	*	*	*
1100	BIC src, dst	Clear bits in destination	dst ***and*** (***not***(src))	dst ***and*** (***not***(src))	**-**	**-**	**-**	**-**
1101	BIS src, dst	Set bits in destination	src ***or*** dst	src ***or*** dst	**-**	**-**	**-**	**-**
1110	XOR src, dst	Exclusive OR source and destination	src ***xor*** dst	src ***xor*** dst	*	*	*	*
1111	AND src, dst	AND source and dst	src ***and*** dst	src ***and*** dst	**0**	*	*	*

3.4 Control Unit

The control unit controls the dataflow through the datapath and the memory activation by generating control signals. It is a *hardwired* control unit, where the control signals to perform microoperations are generated by conventional logic design techniques.

Control unit uses *the instruction word* and the *status information* as inputs and generates control signals for the memory and the datapath units as illustrated in Figure 3-24.

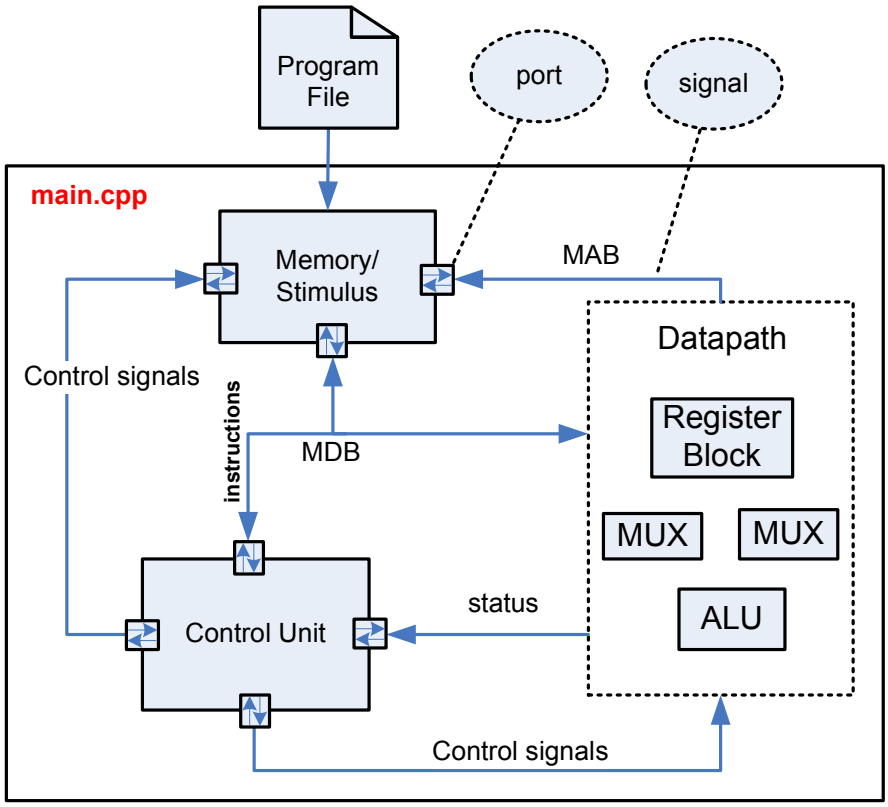

Figure 3-24: General CPU Architecture

The control unit consists of a *control state register*, an *instruction register*, and a *combinational logic* as displayed in Figure 3-25. The control state register, which is triggered with the rising edge of the clock, keeps the current state of the control unit. The size of the register depends on the implementation method and the size of the state machine. The combinational logic part generates control signals by the help of the status information, data of the memory data bus and current instruction loaded into the instruction register (Rinst). Please refer to “cu.cpp” and “cu.h” files in order to see the SystemC implementation of the control unit.

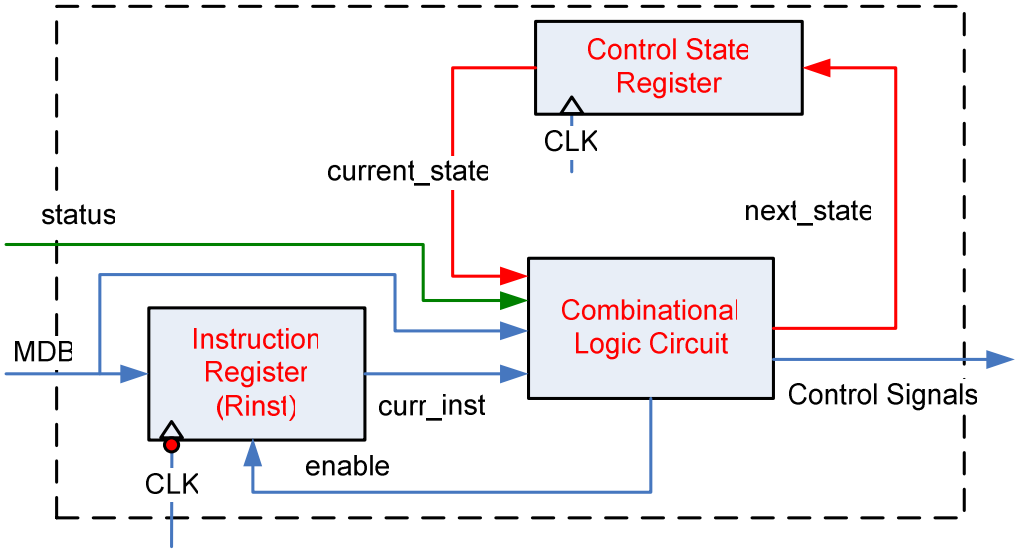

Figure 3-25: Block Schematic of the Control Unit

There are 20 states summarized in Table 3-4. Although SystemC supports tracing of enumerated types, VCD trace files do not. Consequently, all of the control states are assigned to *state numbers* to trace the execution of the control unit. The most essential and thereby the most important of all states for all instructions is called *fetch and decode state*, which is shortened as *fetch_decode.* Fetch and decode operations are performed when the control unit is at the state where types of the instructions are determined as shown in Table 3-5. On the other hand, some other operations can also be done at *fetch_decode* state. Depending on the

instruction and used operand addressing mode, the next state is decided by the next state logic.

Table 3-4: States of the Finite State Machine of the Control Unit

State Name	State Number	Operation
reset_add	900	Ra<=0xFFFE=(-2)
reset_vector	901	Rd<=M[Ra]=M[0xFFFE=(-2)]
rd2pc	902	PC<=Rd
fetch_decode	1	fetch and decode operations
calc_source2	20	Rdst<=Rdst operator M[Ra or PC], Rd<=Rdst operator M[Ra or PC], if #N, PC+
source_add	30	Ra<=M[PC] + Rsrc, PC+
dest_add	40	Ra<=M[PC]+ Rdst, PC+
calc_source	41	Rd<=M[Ra] operator Rd
Rd2Mra	2	M[Ra]<=Rd,…
Read2Rd	50	Rd<=mov M[Ra or PC],..
SP_2	400	SP<=SP-2, Ra<=SP-2
pc2rd	4	Rd<=mov(PC)
to_pc	40	pc<=mov(Rdst) if CALL Rdst,As=00 pc<=mov M[Ra] else
inc_pc	5	pc<=pc+2, rd<=pc+2
off	7	Ra<=M[Ra]+Rdst
calc_ssr	8	Rd<=operator M[Ra], PC+, if #N
calc_pc	434	pc<=pc+Rd
to_SR	671	SR<=mov M[Ra]
inc_sp	435	SP<=SP+2, Ra<=SP+2
inc_sp_2	436	SP<=SP+2, Ra<=SP+2

Control unit is designed by considering the instruction format and the addressing mode architecture of the processor. As mentioned before, there are three instruction formats. The following subsections explain how the control unit works and how it is designed under the heading of each instruction type.

Table 3-5: Core Instruction Map

Bits [15, 14, 13,12]	Instruction Type
0xxx	Not used
1xxx	Single Operand Instruction
2xxx-3xxx	Jump Instruction
4xxx-Fxxx	Double Operand Instruction

3.5 Implementation of the Addressing Modes

This part explains how the processor-2 handles the addressing modes. The state transitions of the control unit and the employed micro operations are also summarized. For the sake of clarity, this section is divided into three parts. The first part deals with the combination of the addressing modes for the double operand instructions. The second part illustrates the single operand instructions. The last part is about the jump instructions. Please note that although the jump instructions do not have any addressing modes, they have intrinsic operands. Thus, the executions of these instructions are also explained in this section.

3.5.1 Double Operand Instructions

Double operand instructions, as its name infers, have double operands named as source and destination. For source operand, there are seven possible addressing

modes, and for destination operand, there are four addressing modes. The addressing mode combinations of double operand instructions are given in Table 3-6 below. Note that the letter *X* denotes the *don't care* condition. As obvious, some exceptions hinder the uniformality. As displayed in the Table 3-6 below, the use of the constant registers R2 and R3 as a source or destination results in disorder.

Table 3-6: Addressing Mode Combinations for the Double Operand Operations (no simplification)

As	Rsrc	Ad	Rdst	Addressing Mode Combinations
00	X	0	X	Register to register
00	X	1	R2=SR=0	Register to absolute
00	X	1	R0=PC	Register to symbol
00	X	1	R1, R3, R4,…R15	Register to indexed
01	R0=PC	0	X	Symbolic to register
01	SR=R2	0	X	Absolute to register
01	R3	0	X	Register to register
01	R1,R4,…R15	0	X	Indexed to register
01	R0=PC	1	R0=PC	Symbolic to symbolic
01	R0=PC	1	R2=SR=0	Symbolic to absolute
01	R0=PC	1	R1, R3, R4,…R15	Symbolic to indexed
01	R2=SR	1	R0=PC	Absolute to symbolic
01	R2=SR	1	R2=SR=0	Absolute to absolute
01	R2=SR	1	R1, R3, R4,…R15	Absolute to indexed
01	R3	1	R0=PC	Register to symbolic
01	R3	1	R2=SR=0	Register to absolute
01	R3	1	R1, R3, R4,…R15	Register to indexed
01	R1,R4,…R15	1	R0=PC	Indexed to symbolic
01	R1,R4,…R15	1	R2=SR=0	Indexed to absolute
01	R1,R4,…R15	1	R1, R3, R4,…R15	Indexed to indexed
10	R2,R3	0	X	Register to register
10	R0,R1,R4,.R15	0	X	Indirect register to register
10	R2,R3	1	R0=PC	Register to symbolic

Table 3-6 (continued)

10	R2,R3	1	R2=SR=0	Register to absolute
10	R2,R3	1	R1, R3, R4,…R15	Register to indexed
10	R0,R1,R4,.R15	1	R0=PC	Indirect register to symbolic
10	R0,R1,R4,.R15	1	R2=SR=0	Indirect register to absolute
10	R0,R1,R4,.R15	1	R1, R3, R4,…R15	Indirect register to indexed
11	R0=PC	0	X	Immediate to register
11	R0=PC	1	R0=PC	Immediate to symbolic
11	R0=PC	1	R2=SR=0	Immediate to absolute
11	R0=PC	1	R1, R3, R4,…R15	Immediate to indexed
11	R2,R3	0	X	Register to register
11	R2,R3	1	R0=PC	Register to symbolic
11	R2,R3	1	R2=SR=0	Register to absolute
11	R2,R3	1	R1, R3, R4,…R15	Register to indexed
11	R1,R4,…R15	0	X	Indirect Auto-increment to register
11	R1,R4,…R15	1	R0=PC	Indirect Auto-increment to symbolic
11	R1,R4,…R15	1	R2=SR=0	Indirect Auto-increment to absolute
11	R1,R4,…R15	1	R1, R3, R4,…R15	Indirect Auto-increment to indexed

The addressing mode combinations above can be simplified through grouping by looking for the answer to the question *how we can implement the addressing mode combinations in hardware.*

As obvious, the *absolute* and *symbolic* addressing modes are special cases of the *indexed* mode. In detail, symbolic mode uses the program counter (PC) as a register and the absolute mode uses SR=0 as a register. Hardware implementation of these addressing modes cannot bring so much difference. Thus, we can group these three addressing mode as a whole.

Similarly, the only difference between the *indirect register mode* and the *indirect register auto-increment mode* is that the latter increments the register and the first

one do not. Besides, the *immediate mode* is a special case of the indirect register auto-increment mode that the program counter is used as a register to point the next address for the immediate constant. As you can see, the hardware implementation of these three addressing modes is not much different. Thus, we can also take these modes as a whole.

Simplified addressing mode combinations for double operand operations are as illustrated in Table 3-7. As presented in Table 3-8, there are six addressing mode groups. Note that the destination register does not have any effect, nevertheless when the constant register R3 is used in the absolute mode, the zero constants, must be generated by R3.

Table 3-7: Simplified Addressing Mode Combinations for the Double Operand Operations

As	Rsrc	Ad	Rdst	Group Symbol	Reduced Addressing Mode Combinations
00	X	0	X	Rsrc_Rdst	Register to register
00	X	1	X	Rsrc_YRdst	Register to indexed
01	Except R3	0	X	XRsrc_Rdst	Indexed to register
01	R3	0	X	Rsrc_Rdst	Register to register
01	Except R3	1	X	XRsrc_YRdst	Indexed to indexed
01	R3	1	X	Rsrc_YRdst	Register to indexed
10	R2,R3	0	X	Rsrc_Rdst	Register to register
10	Except R2, R3	0	X	@Rsrc_Rdst	Indirect register to register
10	R2,R3	1	X	Rsrc_YRdst	Register to indexed
10	Except R2, R3	1	X	@Rsrc_YRdst	Indirect register to indexed
11	R2,R3	0	X	Rsrc_Rdst	Register to register
11	Except R2,R3	0	X	@Rsrc_Rdst	Immediate to register
11	R2,R3	1	X	Rsrc_YRdst	Register to indexed
11	Except R2,R3	1	X	@Rsrc_YRdst	Immediate to indexed

For six addressing mode combinations, the sequences of the microoperations are explained in detail as follows. Please note that register transfer language (RTL) is used.

Table 3-8: Another View of the Simplified Addressing Mode Combinations for the Double Operand Operations

Simplified Addressing Mode Combinations	Group Symbol	As	Rsrc	Ad
Register to register	Rsrc_Rdst	00	X	0
		01	R3	
		10	R2,R3	
		11	R2,R3	
Register to indexed	Rsrc_YRdst	00	X	1
		01	R3	
		10	R2,R3	
		11	R2,R3	
Indexed to register	XRsrc_Rdst	01	Except R3	0
Indexed to indexed	XRsrc_YRdst	01	Except R3	1
Indirect register to register	@Rsrc_Rdst	10	Except R2, R3	0
		11	PC	
Indirect register to indexed	@Rsrc_YRdst	10	Except R2, R3	1
		11	PC	

3.5.1.1 Register to Register Combination Group

This group is the *fastest* one in which the *register contents* are used as operands of the arithmetic logic unit and the calculated result is stored back into the destination register. As well, the combination properties, the appropriate RTL descriptions of the control unit states, and the state transitions are presented in the table below. Note that state numbers related with the current state is also given. As displayed, there are only two states in order to perform this addressing mode combination.

Table 3-9: RTL Descriptions and State Transitions of the Rsrc_Rdst Combination Group

Rsrc_Rdst		
Source Operand Addressing Mode	**Destination Operand Addressing Mode**	
Register Mode	Register Mode	
As	"00"	
Ad	"0"	
Execution Clock Cycles	1, 2	
Needed Memory Word	1	
Current State	**Operations**	**Next State**
fecth_decode (1)	PC ← PC+2, Rdst ←Rdst *operator* Rsrc, **Rd** ← Rdst *operator* Rsrc	Rdst≠PC → fecth_decode
		Rdst=PC → rd2pc
rd2pc (902)	PC ← *mov* (**Rd**)	fecth_decode

As regards the first state shown in the table below, it is named *fetch_decode* state, at which the instruction is fetched from memory and decoded in the control unit. Moreover, at this state, PC is incremented by two and the operation defined in the op-code field of the instruction is put into practice by ALU. As you can see,

fetching, decoding, execution, and write-back phases can be performed within a one-clock cycle. Thus, the CPU can perform single instruction within one clock cycle. Furthermore, the calculated result is also copied in *Rd* register. If the destination register is PC, then the second state transition comes out so as to move the *Rd* content to PC. As seen in the table below, PC cannot be loaded with the output of the ALU at *fetch_decode* state.

As to second state, it is named *rd2pc* state, at witch the content of the Rd register is copied to PC register by using the “move” operation of the ALU. This state is used when the PC is selected as a destination register.

3.5.1.2 Indirect Register to Register Combination Group

This group consists of the indirect register, indirect register auto-increment, and immediate addressing modes for source operand and only register direct mode for destination. In this combination, source operand is fetched form the memory, and the destination operand is an internal CPU register. As well, the combination properties, the RTL descriptions of the control unit states, and the state transitions are shown in the table below. As displayed, there are three states to perform these addressing mode combinations within this group.

At *fetch_decode* state, the instruction fetch and decode operations are performed and PC is incremented by two. In addition, source register content is copied to the address register (Ra) through the destination bus (DST). Thus, source register (Rsrc) content can be used to fetch the operand located in the memory. As can be seen from the data path structure, none of the visible registers, except PC, can load the MAB. Therefore, when a register is used as a pointer, the content of this register must be copied to the address register that can load the MAB. Furthermore, while performing *indirect register auto-increment* mode, the source register content is incremented by one for *byte* operations and by two for *word* operations. This auto-increment operation is executed in ALU by using the constant register R3.

At *calc_source2* state, the operation that is defined in the op-code field is performed in ALU; the result is stored back to the destination and the data register (Rd). On the other hand, for the *immediate addressing mode*, the PC is incremented by two. Because, for this addressing mode, the program counter points to the next location to address the immediate operand, thus the content of the PC must be incremented by two for the location of the next instruction.

Table 3-10: RTL Descriptions and State Transitions of the @Rsrc_Rdst Combination Group

@Rsrc_Rdst		
Source Operand Addressing Mode		**Destination Operand Addressing Mode**
Indirect register mode, indirect auto increment mode, immediate mode		Register Mode
As	"10", "11"	
Ad	"0"	
Execution Clock Cycles	2, 3	
Needed Memory Words	1, 2	

Current State	**Operations**	**Next State**
fecth_decode (1)	PC ← PC+2, **Ra** ← *dst_bus*(Rsrc), if @Rsrc+, Rsrc ← Rsrc+2 or 1	calc_source2
calc_source2 (20)	if #N, PC← PC+2 Rdst ← Rdst *op*.M[**Ra or PC**], **Rd** ← Rdst *op*.M[**Ra** or **PC**]	Rdst≠PC ➔ fecth_decode
		Rdst=PC ➔ rd2pc state
rd2pc (902)	PC ← *mov* (**Rd**)	fecth_decode

If the destination register is PC, then one more state transition occurs to move the data register (Rd) content to PC. This operation is performed at *rd2pc* state.

3.5.1.3 Indexed to Register Combination Group

At these addressing mode combinations, the source operands are addressed by the indexed mode and its derivatives, namely, the symbolic and absolute modes. On the other hand, only the register mode is available for the destination operand. In short, the destination operand is an internal CPU register, and the source operand is fetched from the memory. The operand located in memory is addressed by the source register and a memory content, which follows the instruction word. The sum of the index X and the value of the source register point the source operand location. As well, the combination properties, the RTL descriptions of the control unit states, and the state transitions are shown in the table below. As displayed in the chart, there are three states to perform these addressing mode combinations within this group.

At *fecth_decode* state, the instruction fetch and decode operations are performed and the PC is incremented by two to point the X index value.

Table 3-11: RTL Descriptions and State Transitions of the X(Rsrc)_Rdst Combination Group

X(Rsrc)_Rdst		
Source Operand Addressing Mode		**Destination Addressing Mode**
indexed mode, symbolic mode, absolute mode		Register Mode
As	“01”	
Ad	“0”	
Execution Clock Cycle	3	
Needed Memory Word	2	
Current State	**Operations**	**Next State**
fecth_decode (1)	PC ← PC+2	source_add
source_add (30)	PC ← PC+2, **Ra** ← M[PC]+Rsrc	calc_source2
calc_source2 (20)	Rdst ← Rdst *op.* M[**Ra**]	fecth_decode

At *source_add* state, address of the source operand is calculated by adding the source registers and the index values. The sum is also copied to the address register (Ra). Note that the index value X follows the instruction. Thus, PC must be incremented by two to point next instruction word in the memory.

At *calc_source2* state, the operation determined in the op-code field in the instruction word is performed and the result is stored back to the destination CPU register. Please note that the source operand is the data on the memory data bus (MDB).

3.5.1.4 Register to Indexed Combination Group

This group consists of the addressing mode combinations where the destination operands are addressed by the indexed mode and its derivatives, namely, the symbolic and absolute modes, as well as only the register mode is available for the source operand. In this group, the source operand is one of the CPU registers, and the destination operand is located in the memory. The destination operand is addressed by the sum of the index and the destination register values. As well, the combination properties, the RTL descriptions of the control unit states, and the state transitions are shown in the table below. As displayed in the chart, there are four states to perform these addressing mode combinations within this group.

At *fecth_decode* state, the instruction fetch and decode operations are performed and the content of PC is incremented by two in order to point the location of the index Y.

At *dest_add* state, the address of the destination operand located in the memory is calculated by summing the offset value Y and the destination register contents. Moreover, at this state, PC is incremented by two to point the next instruction word, because the index follows the next instruction word.

At *calc_source* state, the operation defined in the op-code field is performed and the computed value is stored to the data register (Rd). Please not that destination port of the ALU is loaded by the memory data bus (MDB).

At *Rd2Mra* state, the content of the data register (Rd) is copied to the memory location pointed by the address register (Ra).

Table 3-12: RTL Descriptions and State Transitions of the Rsrc_Y(Rdst) Combination Group

Rsrc_Y(Rdst)		
Source Operand Addressing Mode	**Destination Operand Addressing Mode**	
Register Mode	Indexed mode, symbolic mode, absolute mode	
As	"00"	
Ad	"1"	
Execution Clock Cycle	4	
Needed Memory Word	2	
Current State	**Operations**	**Next State**
fecth_decode (1)	PC← PC+2,	dest_add
dest_add (40)	PC← PC+2, **Ra** ← M[PC]+Rdst	calc_source
calc_source (41)	**Rd** ← M[**Ra**] *op.* Rsrc	Rd2Mra
Rd2Mra (2)	M[**Ra**] ← **Rd**	fecth_decode

3.5.1.5 Indirect Register to Indexed Combination Group

In this group, the source and the destination operands are stored in the memory. The source operand is pointed by the source register (Rsrc). As well, the address

of the destination operand is the sum of the index value Y and the destination register (Rdst). Moreover, the combination properties, the appropriate RTL descriptions of the control unit states, and the state transitions are shown in the table below. As displayed, there are five states to execute the instructions with this addressing mode combination.

At *fetch_decode* state, the instruction is fetched from the memory and decoded by the control unit. Besides, PC is also incremented, and the source register content is copied to Ra register by the help of the destination bus (DST). Moreover, to perform *the indirect auto-increment mode*, the source register content is incremented by one for *byte* operations and by two for *word* operations.

Table 3-13: RTL Descriptions and State Transitions of the @Rsrc_Y(Rdst) Combination Group

@Rsrc_Y(Rdst)		
Source Addressing Modes		**Destination Addressing Modes**
Indirect register mode, indirect-auto increment mode, immediate mode		indexed mode, symbolic mode, absolute mode
As	“10”, “11”	
Ad	“1”	
Execution Clock Cycles	5	
Needed Memory Words	2	
Current State	**Operations**	**Next State**
fecth_decode (1)	PC← PC+2, **Ra** ← *dst_bus*(Rsrc), if @Rsrc+, Rsrc ← Rsrc+2 or 1	Read2Rd
Read2Rd (50)	**Rd** ← *mov* M[**Ra**] if not #N	to_pc
	Rd ← *mov* M[**PC**], PC+ if #N	
to_pc (40)	PC← PC+2, **Ra** ← M[PC]+Rdst	calc_source
calc_source (41)	**Rd** ← M[**Ra**] *operator* **Rd**	Rd2Mra
Rd2Mra (2)	M[**Ra**] ← **Rd**	fecth_decode

At *Read2Rd* state, the source operand is fetched from the memory and stored into the Rd register. For *the immediate mode*, PC points to the source operand. Furthermore, for the *indirect register* and *indirect auto-increment* modes, address register (Ra) points to the source operand.

At *to_pc* state, the address of the destination operand is calculated by summing the index value and the destination register contents. Then, the address information is stored into the address register-Ra.

At *calc_source* state, the operation determined by the op-code is performed, and the result is stored to the data register (Rd). The index value follows the instruction word; therefore, PC must be incremented by two to point the next instruction located in the memory.

At *Rd2Mra* state, the content of the data register is copied to the memory location pointed by the address register (Ra) register.

3.5.1.6 Indexed to Indexed Combination Group

This group takes *the longest* time compared to other combinations of the double operand addressing modes. On the other hand, in this group, both of the source and the destination operands are stored in the memory. Besides, the two operands are addressed by the sum of the related index and the register contents. It means that, the source operand address equals to the sum of the index value X and the content of source register-Rsrc. Similarly, the address of the destination operand is estimated by adding the values of index Y and destination register-Rdst. Note that, the index of source operand (X) follows the instruction, and the index of the destination operand, which is denoted with Y, follows the index X. Furthermore, the combination properties, the appropriate RTL descriptions of the control unit states, and the state transitions are shown in the table below. As displayed, there are six states in order to execute the instructions with these addressing mode combinations.

At *fetch_decode* state, the instruction is fetched form the memory, and it is decoded in the control unit. In addition, PC is incremented by two to point the next memory location. No other operation is needed at this state.

At *source_add* state, the source operand address is calculated and the result is stored to the address register (Ra). Note that, at this state, PC already points the source index-X. In addition, PC is incremented by two to point the next index value Y.

At *Read2Rd* state, the source operand is fetched from the memory and it is copied to the data register (Rd).

Table 3-14: RTL Descriptions and State Transitions of the X(Rsrc)_Y(Rdst) Combination Group

X(Rsrc)_Y(Rdst)		
Source Addressing Modes	**Destination Addressing Modes**	
indexed mode, symbolic mode, absolute mode	indexed mode, symbolic mode, absolute mode	
As	"01"	
Ad	"1"	
Execution Clock Cycles	6	
Needed Memory Words	3	
Current State	**Operations**	**Next State**
fecth_decode (1)	PC← PC+2	source_add
source_add (30)	PC← PC+2, **Ra** ← M[PC]+Rsrc	Read2Rd
Read2Rd (50)	**Rd** ← *mov* M[**Ra**]	to_pc
	Rd ← *mov* M[**PC**]	
to_pc (40)	PC← PC+2, **Ra** ← M[PC]+Rdst	calc_source
calc_source (41)	**Rd** ← M[**Ra**] *operator* **Rd**	Rd2Mra
Rd2Mra (2)	M[**Ra**] ← **Rd**	fecth_decode

At *to_pc* state, the address of the destination operand is calculated by summing the contents of the index and the destination register. In addition, PC is incremented by two to point the next instruction.

At *calc_source* state, the operation determined by the op-code is performed, and the result is stored to the data register (Rd).

At *Rd2Mra* state, the content of the data register is copied to the memory location pointed by the address register (Ra).

3.5.2 Single Operand Instructions

As mentioned before, there are seven single operand instructions that can be divided into four groups. The members of the first group, namely, SWPB, SXT, RRC, and RRA instructions, are performed in the ALU, so the execution cycles and addressing mode implementations of these instructions are same. Different approaches are needed to implement CALL, PUSH, and RETI instructions as shown in the following sub sections.

3.5.2.1 SWPB, SXT, RRC, and RRA Instructions

SWPB, SXT, RRC, and RRA instructions can be used with all of the seven addressing modes. In fact, these instructions have assigned functions in the ALU.

3.5.2.1.1 Register Addressing Mode

The register addressing mode takes the *shortest time* among other addressing modes of the single operand instructions. The operand is one of the CPU registers. Moreover, the addressing mode properties, the appropriate RTL descriptions of the control unit states, and the state transitions are shown in the table below. As displayed, there is only one state to execute the instructions with this addressing mode combination.

At *fetch_decode* state, instruction is fetched from the memory and then it is decoded by the control logic. Besides, the operation coded in the instruction word is performed at this state, and the result is stored back to the register with the falling edge of the clock.

Table 3-15: RTL Descriptions and State Transitions of the SWPB, SXT, RRC, and RRA Instructions with the Register Addressing Mode

SSRR R		
Execution Clock Cycle	1	
Needed Memory Word	1	
As	"00"	
Operand Addressing Mode	register mode	
Current State	**Operations**	**Next State**
fecth_decode (1)	PC← PC+2, Rdst ← *operator*(R)	fecth_decode

3.5.2.1.2 Indirect Register Addressing Mode Group

In this group, the operand is located in the memory. One of the CPU registers points to the operand. Moreover, the addressing mode properties, the appropriate RTL descriptions of the control unit states, and the state transitions are shown in the table below. As displayed, there are there states to execute the instructions used with this addressing mode combination.

At *fetch_decode* state, the instruction fetch and decode operations are performed. Besides, the content of the pointer register is copied to the address register (Ra) via the destination bus. For the *immediate mode*, the content of the PC must be incremented and copied to the address register-Ra since in this mode the PC

points to the immediate constant. Moreover, during the execution of the *indirect register auto-increment mode* the content of the pointer register (Rdst) must be incremented by one for *byte* operations and by two for *word* operations.

At *calc_ssr* state, the operation is performed in the ALU. In addition, for the *immediate mode*, the PC is incremented by two to point the next instruction word located in the memory.

At *Rd2Mra* state, a memory write operation is performed. The content of the data register (Rd) is copied to the memory location pointed by the address register (Ra).

Table 3-16: RTL Descriptions and State Transitions of the SWPB, SXT, RRC, and RRA Instructions with the Indirect Register, Indirect Auto-Increment, and Immediate Modes

SSRR @R, SSRR @R+, SSRR @PC+/#N	
Execution Clock Cycles	3
Needed Memory Word(s)	1,2
As	"10", "11"
Operand Addressing Modes	Indirect register mode, indirect auto increment mode, immediate mode

Current State	**Operations**	**Next State**
fecth_decode (1)	PC← PC+2	calc_ssr
	Ra ← *dst_bus*(R), if not #N	
	Ra ← PC+2, if #N	
	R ← R+2/1, if @R+ and not #N	
calc_ssr (8)	**Rd** ← *operator*(M[**Ra**]), PC← PC+2, if #N	Rd2Mra
Rd2Mra (2)	M[**Ra**] ← **Rd**	fecth_decode

3.5.2.1.3 Indexed, Symbolic and Absolute Modes

In this group, the operand is placed in the memory, and the address of the operand is the summation of the offset value X and the register content. Moreover, the addressing mode properties, the appropriate RTL descriptions of the control unit states, and the state transitions are exposed in the table below. As displayed, there are four states to execute the instructions with these addressing modes.

At *fetch_decode* state, the single operand instruction is fetched from the memory, and it is decoded by the control logic. In addition, the PC is incremented by two to point the index value-X.

At *dest_add* state, the address of the operand is calculated by summing the index X and the register content. Besides, the address information is stored in the address register (Ra).

Table 3-17: RTL Descriptions and State Transitions of the SWPB, SXT, RRC, and RRA Instructions with the Indexed, Symbolic, and Absolute Modes

SSRR X(R)		
Execution Clock Cycles	4	
Needed Memory Words	2	
As	"01"	
Operand Addressing Modes	indexed mode, symbolic mode, absolute mode	
Current State	**Operations**	**Next State**
fecth_decode (1)	PC← PC+2	dest_add
dest_add (40)	**Ra** ← M[PC]+R	calc_ssr
calc_ssr (8)	**Rd** ← *operator*(M[**Ra**])	Rd2Mra
Rd2Mra (2)	M[**Ra**] ← **Rd**	fecth_decode

At *calc_ssr* state, the instruction is executed ant the result is stored in the data register (Rd).

At *Rd2Mra* state, the content of the data register is copied to the memory location pointed by address register.

3.5.2.2 PUSH Instruction

PUSH instruction can be used with all of the addressing modes, which maybe divided into three groups.

3.5.2.2.1 Register Addressing Mode

In register addressing mode, the value that is stored at the top of the stack is the content of the one of the CPU registers. Pushing a register content into the stack takes three clock cycles as displayed in the table below. Besides, the addressing mode properties, the appropriate RTL descriptions of the control unit states, and the state transitions can be learned from the table below.

At *fetch_decode* state, the instruction is fetched and decoded. Besides, the PC is incremented to point the next instruction word. Moreover, the register content that will be copied to the stack is copied into the data register (Rd), which is the unique register that can load the MDB.

At *SP_2* state, the stack pointer is decremented by two. This location is where the pushing data will be stored. The decremented value of the SP is also copied into the address register (Ra) to perform the *memory write* operation.

At *Rd2Mra* state, the *memory write* operation is performed. In detail, the content of the data register (Rd) is stored in the memory location pointed by the address register (Ra).

Table 3-18: RTL Descriptions and State Transitions of the PUSH Instruction with the Register Direct Mode

PUSH R		
Execution Clock Cycles	3	
Needed Memory Word	1	
As	"00"	
Operand Addressing Mode	Register mode	
Current State	**Operations**	**Next State**
fecth_decode (1)	PC← PC+2, **Rd** ← *mov*(R)	SP_2
SP_2 (400)	SP ← SP-2, **Ra** ← SP-2	Rd2Mra
Rd2Mra (2)	M[**Ra**] ← **Rd**	fecth_decode

3.5.2.2.2 Indirect Register Addressing Mode Group

The push operation with these addressing modes uses a memory location, which is pointed by one of the CPU register, as an operand. In addition, the addressing mode properties, the appropriate RTL descriptions of the control unit states, and the state transitions are shown in the table below. As displayed, there are four states to execute the addressing modes.

At *fetch_decode* state, the instruction fetch and decode operations are performed. Besides, the content of the pointer register is copied to the address register via the destination bus. Moreover, for the *immediate mode*, the content of the PC must be incremented. Furthermore, during the execution of the *indirect register auto-increment mode* the content of the pointer register must be incremented by one for *byte* operations and by two for *word* operations.

At *Read2Rd* state, the memory content, which is pointed by the address register, is fetched and it is copied to the data register (Rd) with the falling edge of the clock.

At *SP_2* state, the stack pointer is decremented by two. The location, which is pointed by the SP, is the address where the data will be stored into the stack. In addition, the decremented value of the SP is copied into the Ra register to perform the *memory write* operation. Besides, for the *immediate mode*, the PC is incremented by two to point the next instruction word.

At *Rd2Mra* state, a memory write operation is performed. In detail, the content of the data register (Rd) is stored in the memory location pointed by the address register (Ra).

Table 3-19: RTL Descriptions and State Transitions of the PUSH Instruction with the Indirect Register, Indirect Auto-increment and Immediate Modes

PUSH @R , PUSH @R+, PUSH #N	
Execution Clock Cycles	4
Needed Memory Word(s)	1 or 2
As	“10”, “11”
Operand Addressing Modes	Indirect register mode, indirect register auto increment mode, immediate mode

Current State	**Operations**	**Next State**
fecth_decode (1)	PC← PC+2 **Ra** ← *dst*(R), R ← R+2, if @R+ and not #N	Read2Rd
Read2Rd (50)	**Rd** ← M[**Ra** or **PC**]	SP_2
SP_2 (400)	SP ← SP-2, **Ra** ← SP-2 if #N, PC← PC+2	Rd2Mra
Rd2Mra (2)	M[**Ra**] ← **Rd**	fecth_decode

3.5.2.2.3 Indexed, Symbolic and Absolute Modes

The PUSH instruction fetches its operand from the memory in these addressing modes. The operand is addressed by the sum of values of the index X and the content of the destination register. In addition, the addressing mode properties, the appropriate RTL descriptions of the control unit states, and the state transitions are shown in the table below. As displayed, there are five states to execute the addressing modes.

Table 3-20: RTL Descriptions and State Transitions of the PUSH Instruction with the Indexed, Symbolic and Absolute Addressing Modes

PUSH X(R)		
Execution Clock Cycles	5	
Needed Memory Words	2	
As	"01"	
Operand Addressing Modes	indexed mode, symbolic mode, absolute mode	
Current State	**Operations**	**Next State**
fecth_decode (1)	PC← PC+2, **Ra** ← *dst*(R)	dest_add
dest_add (40)	**Ra** ← M[PC] + R PC← PC+2	Read2Rd
Read2Rd (50)	**Rd** ← M[**Ra**]	SP_2
SP_2 (400)	SP ← SP-2, **Ra** ← SP-2	Rd2Mra
Rd2Mra (2)	M[**Ra**] ← **Rd**	fecth_decode

At *fetch_decode* state, the instruction fetch and decode operations are performed. Besides, the content of the pointer register is copied to the address register (Ra) by the use of the destination bus.

At *dest_add* state, the address of the operand is calculated by summing the content of the index value and the destination register.

At *Read2Rd* state, the memory content, which is pointed by the Ra register is fetched and copied to the data register (Rd) with the falling edge of the clock.

At *SP_2* state, the stack pointer is decremented by two in order to point a memory location into which the pushing data will be stored. In addition, the decremented value of the SP is copied into the Ra register to perform the memory write operation.

At *Rd2Mra* state, the memory write operation is performed. In detail, the content of the Rd register is stored into the memory location pointed by the Ra register.

3.5.2.3 CALL Instruction

CALL instruction can be used with all of the seven addressing modes, which are divided into four groups.

3.5.2.3.1 Register Addressing Mode

CALL instruction with the register mode, directs the program execution to another memory location pointed by the register. Besides, the addressing mode properties, the appropriate RTL descriptions of the control unit states, and the state transitions are shown in the table below. As displayed, there are four states to execute CALL instruction with the direct register addressing mode.

At *fetch_decode* state, the instruction is fetched and decoded. Besides, the content of the program counter is incremented and stack pointer is decremented by two. Moreover, the new value of the SP is also copied to the address register (Ra).

At *pc2rd* state, the PC value, which points to the next instruction word to be executed after returning from the call procedure, is copied to the data register (Rd).

At *Rd2Mra* state, the return address is copied to the memory location pointed by the address register.

Table 3-21: RTL Descriptions and State Transitions of the CALL Instruction with the Register Direct Mode

CALL R		
Execution Clock Cycles	4	
Needed Memory Word	1	
As	"00"	
Operand Addressing Mode	Register mode	
Current State	**Operations**	**Next State**
fecth_decode (1)	PC← PC+2, SP ← SP-2, **Ra** ← SP-2	pc2rd
pc2rd (4)	**Rd** ← *mov*(PC)	Rd2Mra
Rd2Mra (2)	M[**Ra**] ← **Rd**	to_pc
to_pc (6)	PC ← *mov*(R)	fecth_decode

At *to_pc* state, the register content is copied to the PC in order to redirect the program execution to the intended subroutine.

3.5.2.3.2 Indirect Register Addressing Mode

CALL instruction with indirect register mode, directs the program execution to another memory location pointed by a memory location. In addition, the addressing mode properties, the appropriate RTL descriptions of the control unit states, and the state transitions are shown in the table below. As displayed, there are four states to execute the addressing mode.

Table 3-22: RTL Descriptions and State Transitions of the CALL Instruction with the Indirect Register Mode

CALL @R		
Execution Clock Cycles	4	
Needed Memory Word	1	
As	"10"	
Operand Addressing Mode	Indirect register mode	
Current State	**Operations**	**Next State**
fecth_decode (1)	PC← PC+2, SP ← SP-2, **Ra** ← SP-2	pc2rd
pc2rd (4)	**Rd** ← *mov*(PC)	Rd2Mra
Rd2Mra (2)	M[**Ra**] ← **Rd,** **Ra** ← *mov***(**R)	to_pc
to_pc (6)	PC ←*mov* M[**Ra**]	fecth_decode

At *fetch_decode* state, the instruction is fetched and decoded. Besides, PC is incremented and SP is decremented by two. Moreover, the new value of the stack pointer is copied to the Ra register.

At *pc2rd* state, the PC value, which points the return address, is copied to the data register (Rd).

At *Rd2Mra* state, the address of the next instruction is copied to the memory location pointed by address register (Ra). Besides, the register content is copied to the address register.

At *to_pc* state, the memory content is copied to the PC to direct the execution of the program to the subroutine.

3.5.2.3.3 Indirect Register Auto-increment and Immediate Modes

CALL instruction with indirect register auto-increment mode and immediate mode directs the program execution to indented subroutine whose start address is pointed by a memory location, or an immediate constant. In addition, the addressing mode properties, the appropriate RTL descriptions of the control unit states, and the state transitions are shown in the table below. As displayed, there are five states to execute the mentioned addressing modes.

At *fetch_decode* state, the instruction is fetched and decoded. Besides, the content of the PC is incremented and the SP is decremented by two. Moreover, the new value of the stack pointer is copied to the address register.

At inc_pc state, if the addressing mode is immediate mode, then PC is incremented by two to point the next instruction word in the memory.

At *pc2rd* state, the PC value, which points to the next instruction to be executed after returning from the call procedure, is copied to the data register (Rd).

At *Rd2Mra* state, the address of the next instruction is copied to the memory. Besides, the register content is copied to the Ra register. Besides, indirect register auto-increment mode, the address register is incremented by one for *byte*, and by two for *word* operations. Moreover, for the immediate mode the PC is decremented to fetch the immediate value from the memory.

At *to_pc* state, the memory content is copied to the PC to direct the CPU to the subroutine.

Table 3-23: RTL Descriptions and State Transitions of the CALL Instruction with the Indirect Register and Immediate Modes

CALL @R+ / CALL #N			
Execution Clock Cycles	5		
Needed Memory Word(s)	1 or 2		
As	"11"		
Operand Addressing Mode	Indirect register auto-increment mode, Immediate mode		
Current State	**Operations**		**Next State**
fecth_decode (1)	PC← PC+2, SP ← SP-2, **Ra** ← SP-2		inc_pc
inc_pc (5)	if #N, PC+		pc2rd
pc2rd (4)	**Rd** ← *mov*(PC)		Rd2Mra
Rd2Mra (2)	M[**Ra**] ← **Rd**		to_pc
	if @R+ and not #N	**Ra** ← R, R ← R+2	
	if #N	**Ra** ← PC-2, PC ← PC-2	
to_pc (6)	PC ←*mov* M[**Ra**]		fecth_decode

3.5.2.3.4 Indexed, Absolute, and Symbolic Addressing Modes

CALL instruction with the indexed, absolute, symbolic modes, jumps the program execution to a new location that is pointed by a memory location. As obvious, the address of the memory location is the sum of the offset and register contents. In addition, the addressing mode properties, the appropriate RTL descriptions of the

control unit states, and the state transitions are shown in the table below. As displayed, there are five states to execute the addressing modes.

At *fetch_decode* state, the instruction is fetched and decoded. Besides, the PC is incremented and the SP is decremented by two. Moreover, the new value of the stack pointer is copied to the Ra register.

At inc_pc state, the PC is incremented to sore the return address from the subroutine call.

At *Rd2Mra* state, the address of the next instruction is copied to the memory via the Ra and Rd registers. Besides, the PC is decremented by two and the decreased value of the PC is copied to the address register.

Table 3-24: RTL Descriptions and State Transitions of the CALL Instruction with the Indexed, Absolute, Symbolic Modes

CALL X(R)		
Execution Clock Cycles	5	
Needed Memory Words	2	
As	"01"	
Operand Addressing Modes	Indexed Mode, Absolute Mode, Symbolic Mode	
Current State	**Operations**	**Next State**
fecth_decode (1)	PC← PC+2, SP ← SP-2, **Ra** ← SP-2	inc_pc
inc_pc (5)	PC← PC+2	Rd2Mra
Rd2Mra(2)	M[**Ra**] ← **Rd** , **Ra** ← PC-2, PC ← PC -2	off
off (7)	**Ra** ← M[**Ra**] + R	to_pc
to_pc (6)	PC ← *mov* M[**Ra**]	fecth_decode

At *off* state, the address of the subroutine is calculated by summing the offset value and the register value.

At *to_pc* state, the memory content is copied to the PC to direct the CPU to the called subroutine.

3.5.2.4 RETI Instruction

RETI instruction does not have any addressing mode. In addition, the appropriate RTL descriptions of the control unit states and the state transitions are shown in the table below. As displayed, there are five states to execute the instruction.

At *fetch_decode* state, the instruction is fetched and decoded. Besides, the PC is incremented and the stack pointer content is copied to the address register.

Table 3-25: RTL Descriptions and State Transitions of the RETI Instruction

RETI		
Execution Clock Cycles	5	
Needed Memory Word	1	
As	-	
Operand Addressing Mode	no addressing mode	
Current State	**Operations**	**Next State**
fecth_decode (1)	PC← PC+2, **Ra** ← *mov* SP	to_SR
to_SR (671)	SR ← *mov* M[**Ra**]	inc_sp
inc_sp (435)	SP← SP+2 , **Ra** ← SP+2	to_pc
to_pc (6)	PC ← *mov* M[**Ra**]	inc_sp_2
inc_sp_2 (436)	SP← SP+2	fecth_decode

At *to_SR* state, the content of the status register is copied to the top of the stack.

At *inc_sp* state, the stack pointer content is incremented by two and the new value is copied to the address register in order to point the new top of the stack.

At *to_pc* state, the memory content is copied to the PC to direct the program execution.

At *inc_cp_2* state, the stack pointer is incremented by two. *fetch_decode* state follows this state.

3.5.3 Jump Instructions

None of the jump instructions is used with any addressing modes. There are only two state transitions to execute all of the jump instructions even if the jump condition is not met. In addition, the appropriate RTL descriptions of the control unit states and the state transitions are illustrated in the table below.

Table 3-26: RTL Descriptions and State Transitions of the Jump Instructions

JUMP		
Execution Clock Cycles	2	
Needed Memory Word	1	
Current State	**Operations**	**Next State**
fecth_decode (1)	PC ← PC+2, Rd← Rmdb	calc_pc
calc_pc (434)	PC ← Rd + PC	fecth_decode

At *fetch_decode* state, the jump instruction is fetched from the memory and it is decoded by the control logic. Moreover, the content of the memory data bus register (Rmdb), which equals the left shifted and sign extended version of the offset value, is copied to the data register (Rd).

At *calc_pc* state, the effective jump address is calculated by summing the content of the data register (Rd) and the address of the next instruction pointed by the program counter. If the jump condition is met, then the PC is loaded with this value. Otherwise, the content of the program counter remains as it is.

3.6 Stimulus or Memory Unit

Stimulus unit is an external part of the CPU as seen in Figure 3-26. In fact, this part acts as a memory part for the processor core. The program file, which is generated by the cross compilers, is loaded to the memory locations.

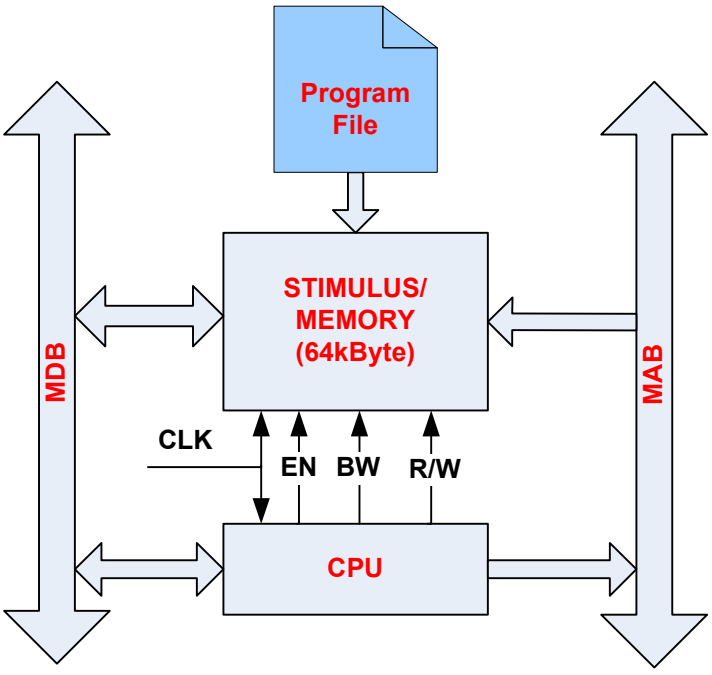

Figure 3-26: Memory Unit and CPU Interface

The memory unit is composed of a read and a write memory processes as seen in Figure 3-27 (Please refer to the “stimulus.cpp” and “stimulus.h” files of the SystemC projects). Before the instantiation of these parts, a function (Load_Mem) is called to load the memory locations.

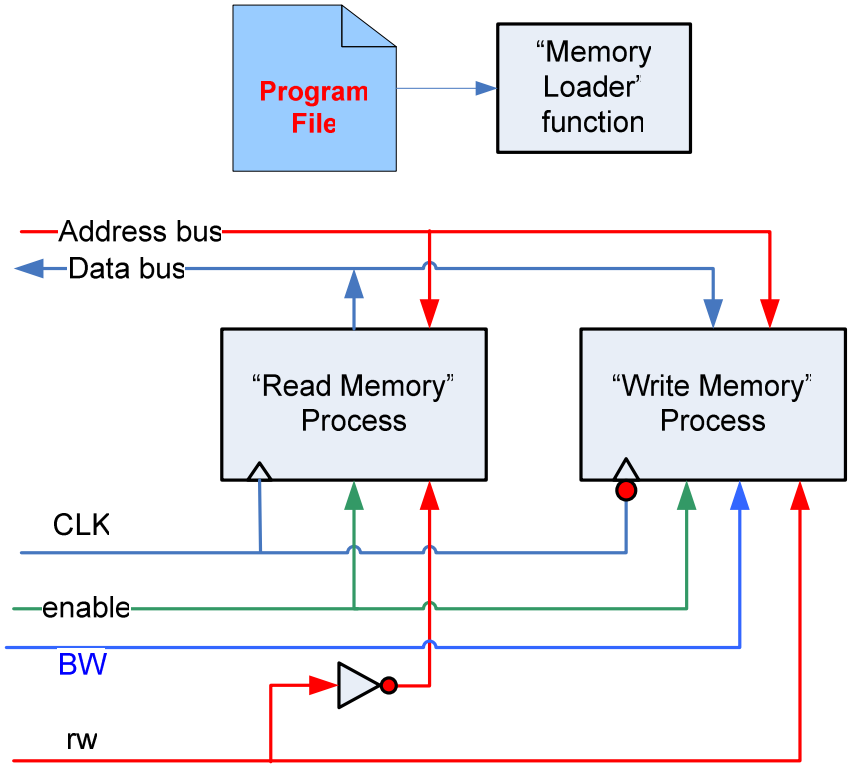

Figure 3-27: Signal Details of the Memory Unit

The read process enables the control unit to read any of the memory word locations. The process ignores the BW signal. When the memory is enabled (enable=1) and a read operation is initiated (rw=0), then the word content pointed by the memory address bus is loaded to the memory data bus. In the example given in Figure 3-28 below, data value-3001h is fetched from the memory

location-3000h. When the memory is disabled or a write operation is initiated, then read process does not loads the memory data bus.

Writing to the memory is performed via the write process. Depending on the value of the BW signal, write process writes to byte or word locations of the memory. When the memory is enabled and a write operation (rw=1) is initiated, then the write process samples the data on the memory data bus with the falling edge of the clock. In the example given in Figure 3-28 below, data value-7002h is written to the memory location-3000h. When the memory is disabled or a read operation is initiated, then read process does not load the memory data bus.

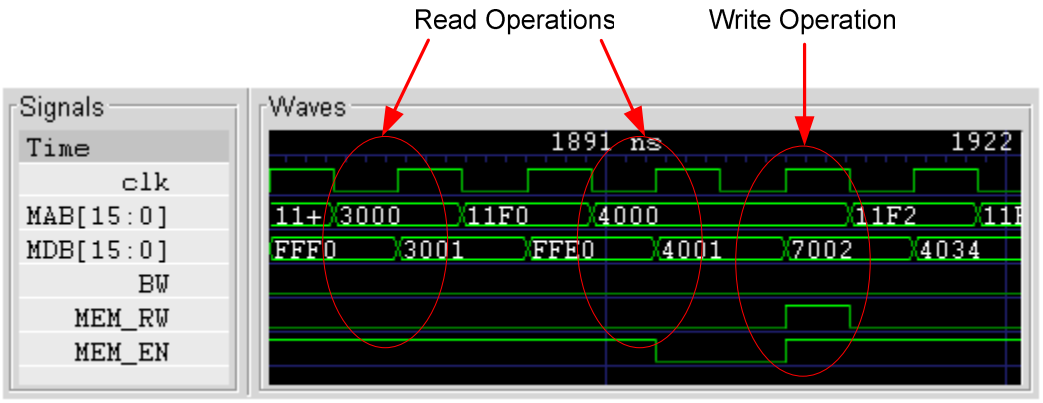

Figure 3-28: Read and Write Operations of the Memory

CHAPTER 4

VERIFICATION OF THE SYSTEMC CPU CORES

The structure of the verification platform is as shown in Figure 4-1. The IAR cross compiler is used to transform the source codes that are written in assembly (ASM), C, or C++ languages into machine codes. The program file, which consists of machine codes, is an input for the testbench of the SystemC designs. Firstly, the stimulus part loads its memory with this program, and then sends stimulus inputs for the processor core. The results of the SystemC simulations are saved in VSD file format, which can be inspected by GTKWave Analyzer v1.3.19 or SynaptiCad WaveViewer 11.03b.

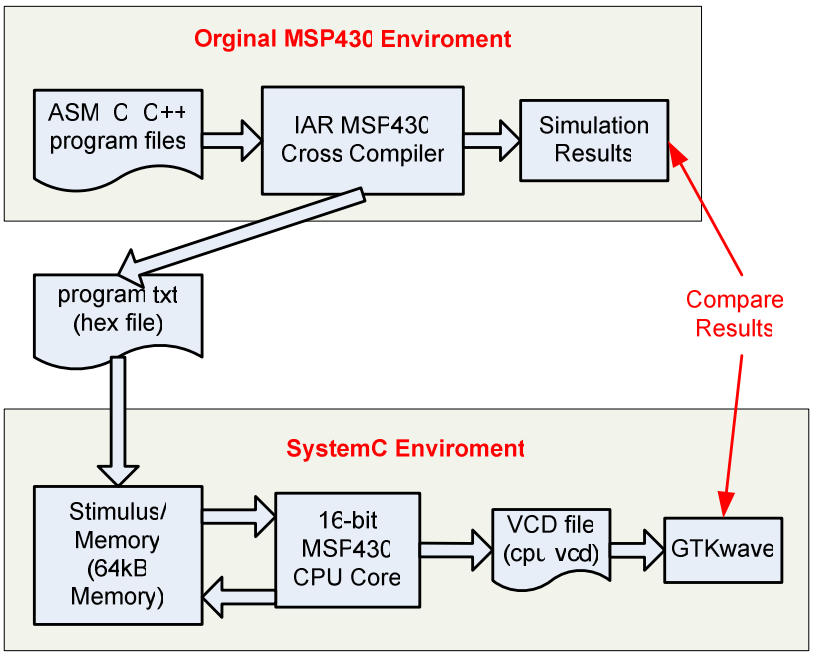

Figure 4-1: Testbench Architecture for the Processor Cores

All of the instructions and addressing modes were simulated via the testbench given above. Besides, cyclic redundancy code (CRC) algorithms are used to verify the overall functionality of the processors. To simulate the processor cores, programs are written in assembly language in IAR compiler environment. Note that, all of the simulation programs are presented in the CD-ROM, which is attached to back cover of the thesis.

4.1 Verification of the Instruction Set Architecture

In this part, all of the 27 core instructions are simulated for two processor cores. The simulation structure includes both the *functional test* and all possible effects of the instruction executions on the *status information* of the processors. Besides, the *byte* and *word* operations are examined. Only one data sample is employed to verify each of the test combinations. The instruction XOR is given as an example here. The assembler test codes related with the XOR instruction and the simulation results are illustrated below.

```
;XOR(.B) src, dst Exclusive OR source and destination, "src XOR dst => dst"
  ;word operation
    ;Functional test
      mov #0x3AAA, r4
      xor #0xFFFF, r4 ;r4=0xC555
    ;Status bit test
      clrn
      clrz
      clrc
      mov #0x8000,r4
      xor #0xF000,r4 ;r4=0x7000,SR=0x0101 (V=1,C=1=not(Z))
      xor #0x8000,r4 ;r4=0xF000 SR=0x0005 (N=1,C=1=not(Z))
      xor #0xF000,r4 ;r4=0x0000 SR=0x0102 (V=1,Z=1)

   ;Byte operation
    ;Functional test
      mov.b #0x3A, r4
      xor.b #0xFF, r4 ;r4=0xC5
    ;Status bit test
      clrn
      clrz
      clrc
      mov.b #0x80,r4
      xor.b #0xF0,r4 ;r4=0x0070,SR=0x0101 (V=1,C=1=not(Z))
      xor.b #0x80,r4 ;r4=0x00F0 SR=0x0005 (N=1,C=1=not(Z))
      xor.b #0xF0,r4 ;r4=0x0000 SR=0x0102 (V=1,Z=1)
```

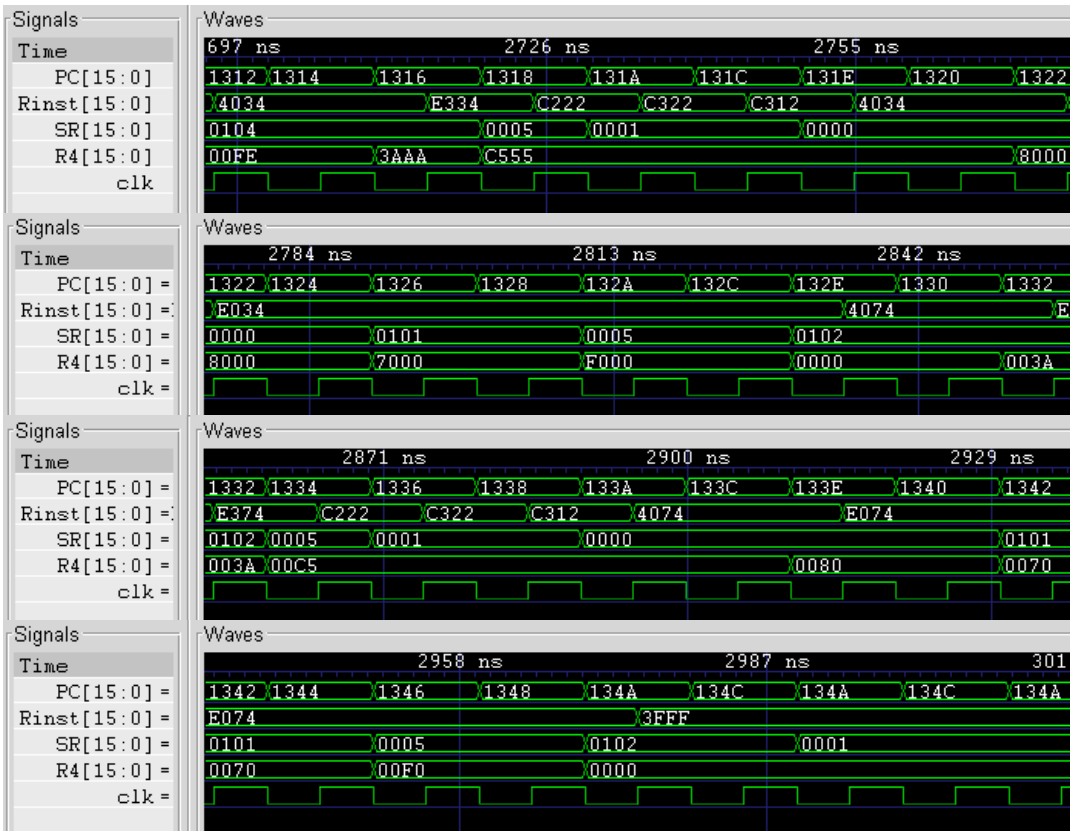

Figure 4-2: Simulation Results for the XOR Instruction

4.1.1 Double Operand Instructions

So as to simulate all of the double operand instructions, a test program is designed. Functional behavior of the instructions, their effects on status bits and byte/word issues are examined. Please note that, since double operand instruction have opcodes that are directly related with the ALU functions, verification of these instructions also proofs the related functions defined in the ALU.

4.1.2 Single Operand Instructions

The single operand instructions are simulated by a test project created with IAR cross compiler. The expected results verify the single operand instructions. Besides, this verification stage shows that the related functions performed by the ALU works well.

4.1.3 Jump Instructions

Jump instructions are simulated, which verifies the execution of the all jump instructions and related components.

4.2 Verification of the Addressing Modes

In this part, all of the possible addressing modes are simulated for all instructions. As mentioned before, jump instructions are not used with the addressing modes. Thus, this section only deals with the double and single operand instructions. Note that only one data sample is employed to verify each of the test combinations. The combination of the indirect register mode and the indexed addressing mode is presented here as an example. The assembler codes and simulation waveforms are given below.

```
//"Indirect Register Mode" for source operand and "Indexed Mode" for destination operand
//instruction @Rsrc,Y(Rdst)
    mov #2300h,r4
    mov #2400h,r5
    mov #0011h,0(r4); M[r4=2300h]=0011h
    mov @r4,300h(r5);M[r5+300h]=M[2700h]=0011h
```

Figure 4-3: Simulation Results for the Indirect Register to Indexed Addressing Mode

4.2.1 Double Operand Instructions

An assembly program is designed in IAR 3.40A cross compiler to simulate all possible addressing mode combinations for the double operand instructions. Each addressing mode test is intent to be independent to other parts. The expected results and simulation outputs show that the addressing mode combinations for the double operand instructions are executed correctly in the CPU design cycle.

4.2.2 Single Operand Instructions

To simulate all seven possible addressing modes, the single operand instructions are divided into four groups as done in the design stage. Please refer to Table 4-1 below.

The first group contains the instructions SXT, SWPB, RRA and RRC. The four instructions in this group can be distinguished between each other with the ALU

functions. Thus, there is no need to simulate all these four instructions separately. The second and third groups have only the PUSH and CALL instruction, respectively. These two instructions have different sequence of micro-operations. Therefore, they are handled independently. The last group named *Group-4* consists of the RETI instruction that has no addressing mode and it is skipped at this stage.

Table 4-1: Grouping the Single Operand Instructions

Group Name	Instructions	Addressing Modes
Group-1	SXT, SWPB, RRC, RRA	All of the seven addressing modes are possible
Group-2	PUSH	
Group-3	CALL	
Group-4	RETI	No addressing mode is available

4.3 CRC Algorithms

In order to verify the processors thoroughly, original cyclic redundancy code (CRC) algorithms are preferred because they can be implemented without the need of the peripherals of the MSP430 microcontroller. Thus, three different kinds of CRC algorithms, namely, b*itwise*, *reflected bitwise* and *table based* are employed with 16 and 32-bit domains. Applied algorithms, which are written in the MSP430 assembly, were quoted from the Texas Instruments Inc. [14].

The cyclic redundancy codes are employed so as to have an idea with regard to whether transmitted data is correct or not. Thus, the transmitter generates redundant data and concatenates the redundant code to the original message. Besides, the receiver side interpreters the CRC and the pure data to determine the correctness of the data transmission. Computing CRCs are based on polynomial division, where each bit of the digital message is mapped to the binary

coefficient of a polynomial. In order to compute CRCs, bit-by-bit division and table-based algorithms can be applied.

The bitwise and reflected bitwise algorithms need less memory but longer computation time. Thus, bitwise method can be proper for low cost and slow applications. 16-bit and 32-bit CRC algorithms with bitwise and reflected bitwise implementations are employed in order to verify the RTL SystemC models of the processor cores. Note that, in bitwise method, the incoming data and the resulting CRC must be bit reflected. Applied input data sequence is “123456789” which can be represented by “31 32 33 34 35 36 37 38 39” as hexadecimal notation. Important parameters, namely, CRC polynomial, initial and expected CRC values, are given in Table 4-3 below. The verification phase ended with successful results as depicted in Figure 4-4, Figure 4-5, Figure 4-6 and Figure 4-7.

Table 4-2: Bitwise CRC Algorithms

	Algorithm Type	Expected CRC Value	Initial CRC Value	Polynomial
16 bit	Bitwise	R12=0xFEE8	R12=0x0000	R14=0x8005
	Reflected bitwise	R12=0xBB3D	R12=0x0000	R14=0xA001
32 bit	Bitwise	R13=0xFC89, R12=0x1918	R13=0xFFFF, R12=0xFFFF	R15=0x04C1 R14=0x1DB7
	Reflected bitwise	R13=0xCBF4, R12=0x3926	R13=0xFFFF, R12=0xFFFF	R15=0xEDB8 R14=0x8320

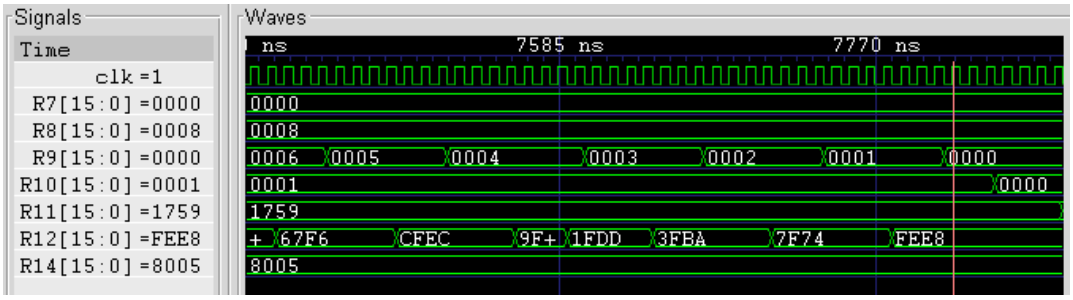

Figure 4-4: 16-bit Bitwise CRC Algorithm

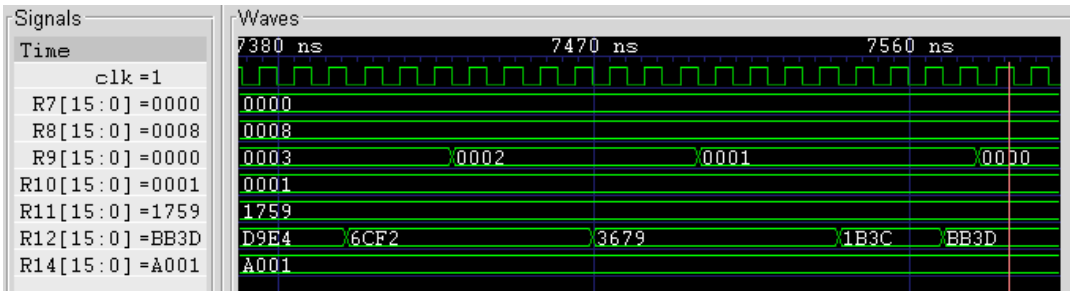

Figure 4-5: 16-bit Reflected Bitwise CRC Algorithm

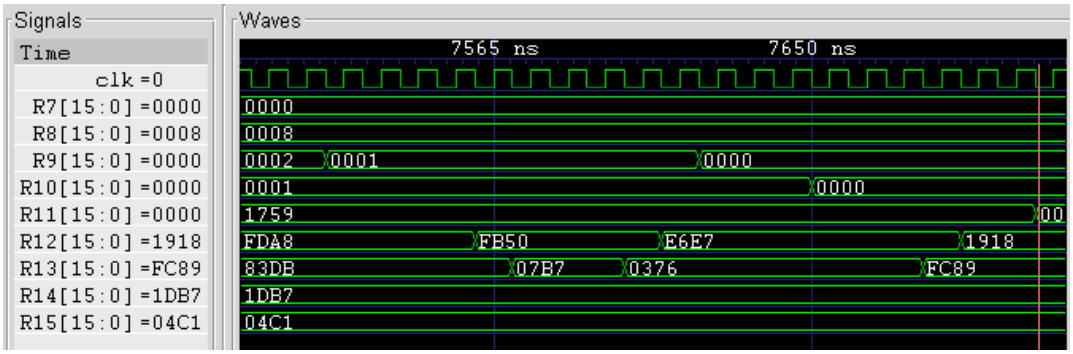

Figure 4-6: 32-bit Bitwise CRC Algorithm

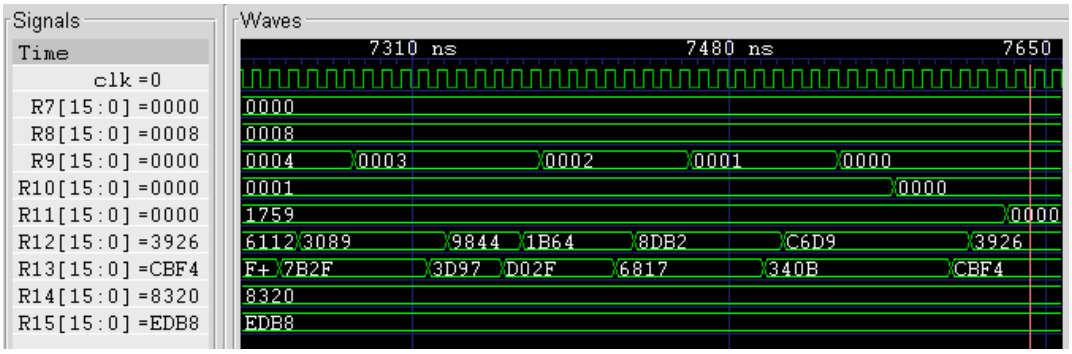

Figure 4-7: 32-bit Reflected Bitwise CRC Algorithm

Table-based CRC algorithms need low number of cycles, which result in low power consumption. On the other hand, this approach requires calculating and storing the CRC values in a table before the execution. Note that 512-byte tables are used for the table-based 16-bit and 32-bit CRC implementations. The applied input data sequence is the numbers from 1 to 9 which can be represented by “31 32 33 34 35 36 37 38 39” in the hexadecimal notation. Initial and expected CRC values are given in Table 4-3 below. The simulation results are same with the expected values as shown in Figure 4-8 and Figure 4-9.

Table 4-3: Table–Based CRC Algorithms

Algorithm Type	Expected CRC Value	Initial CRC Value
16-bitTable-based	R12=0xFEE8	R12=0x0000
32-bitTable-based	R13=0xFC89, R12=0x1918	R13=0xFFFF, R12=0xFFFF

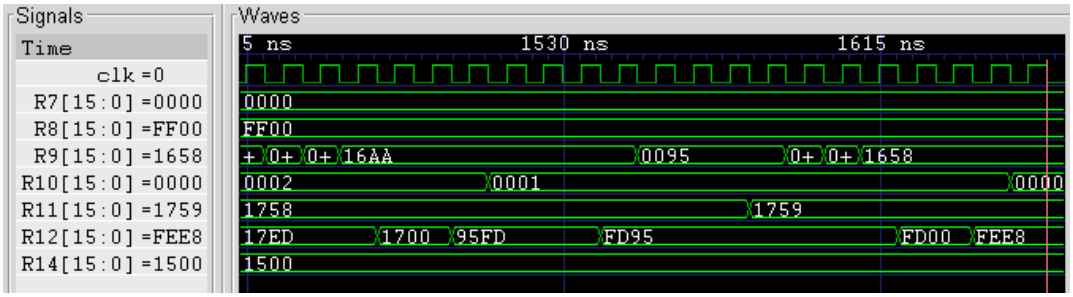

Figure 4-8: 16-bit Table Based CRC Algorithm

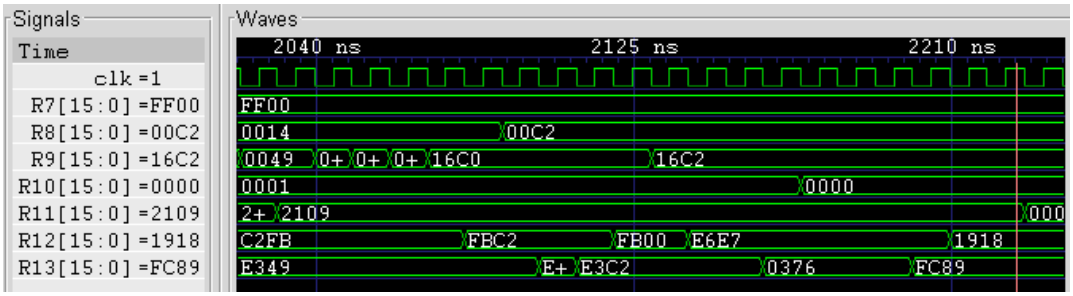

Figure 4-9: 32-bit Table Based CRC Algorithm

The employed CRC algorithms cover 2.68% of the double operand, 5.1% of the single operand and 25% of the jump instructions. These computations includes all possible combination of the addressing modes and operation types, namely, byte and word operations.

CHAPTER 5

SYSTEMC TO HARDWARE FLOW

In this part, following an overview of the SystemC to hardware design flows, the path and the tools used in order to generate hardware from the SystemC descriptions are mentioned in enough detail. Converting the SystemC model of the *Arithmetic and Logic Unit* to an FPGA-based hardware design was performed as a case study.

5.1 Introduction

Following the design and verification of a system in the SystemC environment, the hardware-software partitioning phase takes place. In fact, the partitioning phase can also be handled at the design stage. No matter how the hardware parts are generated, the other important problem, which is the conversion of the SystemC description into hardware parts, arises. Various design flows can be traced to produce real working hardware parts from the SystemC hardware descriptions. Alternative two design flows are presented in Figure 5-1 below.

Firstly, SystemC models can be directly synthesized to netlist. Most efficient netlists are expected to be produced by this approach. On the other hand, the second approach depends on translating the SystemC descriptions into traditional HDLs and generating netlists with traditional HDL synthesis process. This indirect approach brings some important advantages. For instance, HDL synthesis tools are so mature and powerful. Besides, combining some HDL IPs to a project at the HDL level is possible.

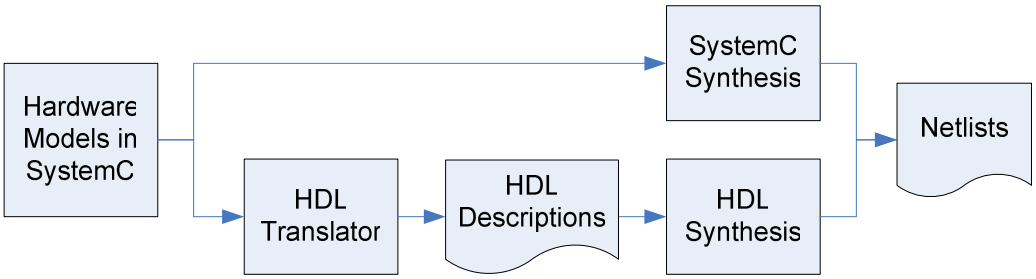

Figure 5-1: Alternative SystemC to Hardware Flows

In this study, the SystemC description of the *Arithmetic and Logic U*nit is firstly translated into VHDL by the SystemCrafter synthesis tool and then classical Xilinx design flow is kept track of generating real working hardware of the ALU.

5.2 SystemC to VHDL Conversion

SystemC is a system modeling and verification environment. The designer describes the system by means of the SystemC methodology, and then verifies the model whether it works properly or not. After the verification and refinement stages of the model, the designer translates the hardware parts into traditional HDL codes manually, which is very tedious and error prone process. SystemCrafter SC, which is a SystemC synthesis tool, automates this process by synthesizing the RTL VHDL descriptions from original SystemC hardware models. The synthesis tool also generates gate-level SystemC description, which is used to verify the translation process. Figure 5-2 below displays the design flow of SystemCrafter. There are two types of design flows available:

- System-level design flow
- Gate-level design flow

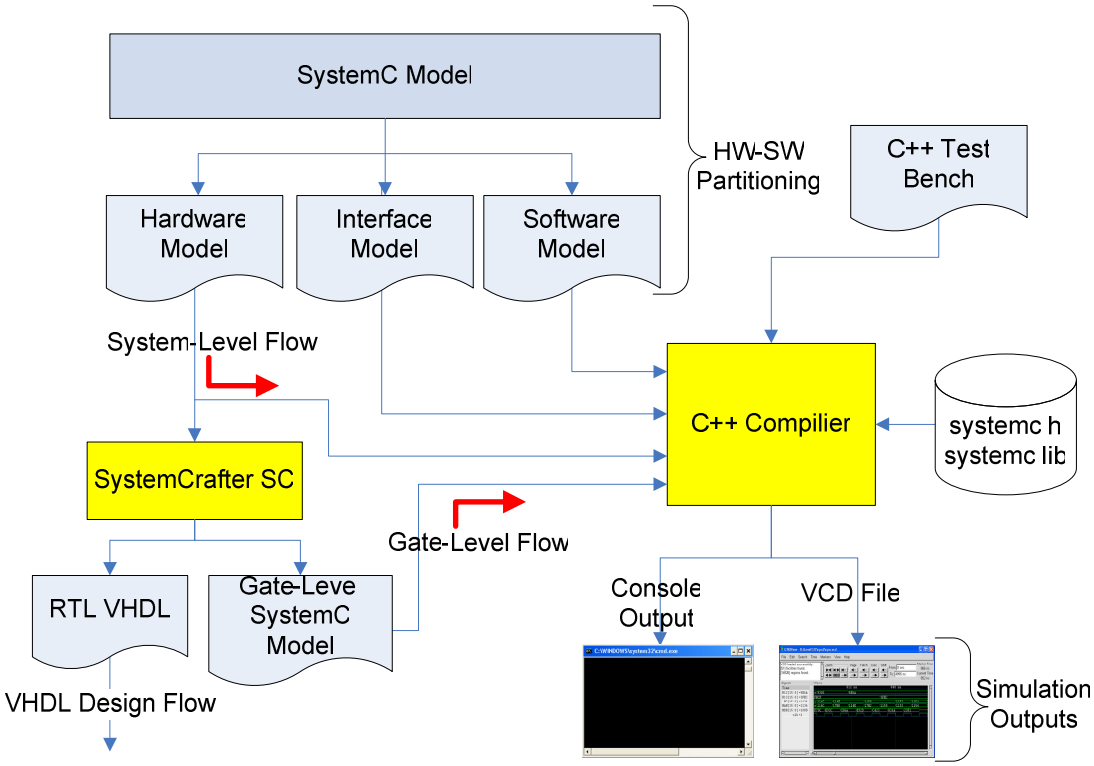

Figure 5-2: SystemCrafter Design Flow

5.2.1 System-Level Flow

The system-level flow uses a C++ compiler such as Microsoft Visual C++ or GNU GCC to simulate the SystemC constructs. This flow is the same with the original SystemC design flow.

5.2.2 Gate-Level Flow

The gate-level design flow converts the SystemC hardware descriptions to *gate-level SystemC models* and *RTL VHDL codes*. Generated gate-level SystemC cores are, in fact, in the form of C++ files (.cpp) and they can be simulated via a C++ compiler. Generated VHDL codes require other tools to be synthesized down

to FPGA primitives. Please note that VHDL descriptions can be generated independently from the gate-level design flow. Note that the details of the synthesis tool can be found in the SystemCrafter user manual for version 2.0.0 [16].

5.2.3 Synthesizable Subset of SystemC

SystemC was originally designed as a system-level simulation language, and it is not possible to compile all of the SystemC constructs to hardware [15], which is similar to the traditional hardware description languages (HDLs). Similarly, a subset of SystemC, which are described in the *Language Reference* part of the user manual [16], is supported by SystemCrafter. Because of the limitations of the tool, original SystemC model of the *Arithmetic and Logic Unit* was slightly modified. The most significant change done was that synchronous design of the original ALU had to be converted to synchronous one. Because SystemCrafter does not support SC_METHOD process and only clocked SC_THREAD processes are supported.

5.2.4 Performed Operations with SystemCrafter

Although SystemCrafter can be used to simulate the SystemC codes in the system-level and gate-level design flows, it is only used in this study to compile the SystemC description of the Arithmetic-Logic Unit of the processors into RTL VHDL codes. In fact, the system-level design flow is exactly same with the original SystemC flow, which was already employed at the design and verification stages of the processor cores. Besides, gate-level flow cannot manage with the ALU constructs and the compiler gives a *compiler limit* error because it expects more simple functions as shown below:

> “C:\samples\alu64\GateLevel\alu.cpp(469): fatal error C1509: compiler limit: too many exception handler states in function 'alu_do_alu::alu_do_alu'. simplify function”

The generated VHDL codes are synthesized to netlist and they are implemented into hardware by using standard VHDL design flows of the Xilinx FPGAs.

5.3 VHDL to HW Flow

In this part, synthesis, implementation and simulation operations are explained. Xilinx ISE 8.1i design and verification environment is employed throughout all of the stages [30, 31, 32]. Figure 5-3 below illustrates the followed steps.

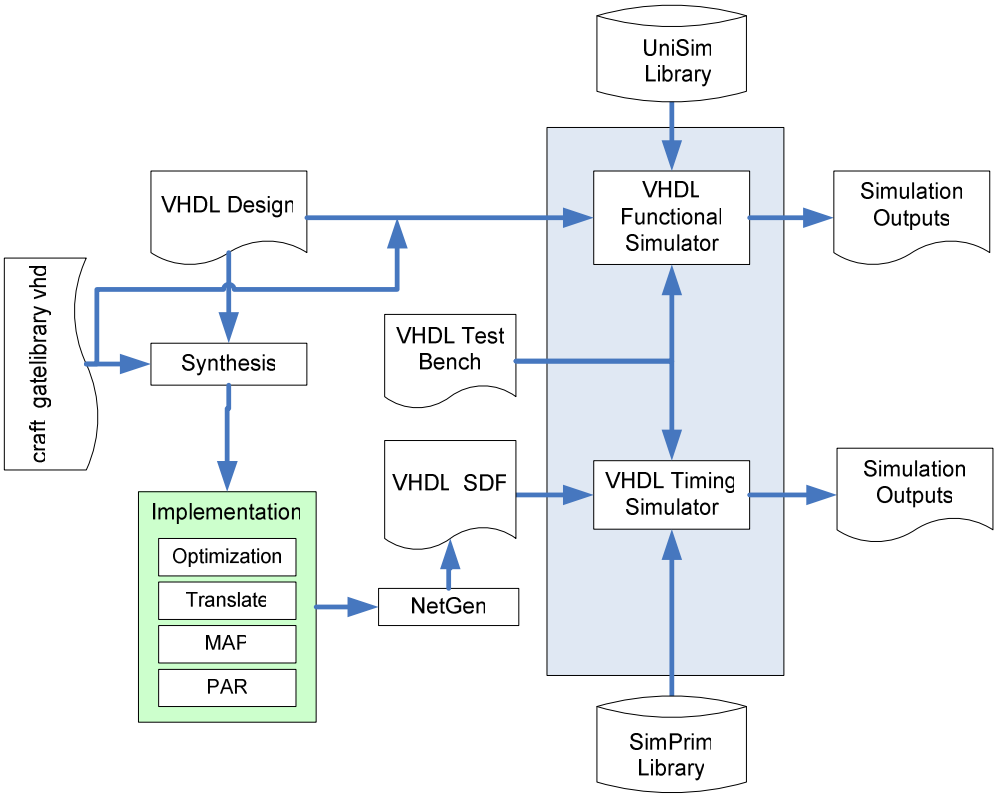

Figure 5-3: Xilinx Design and Verification Flows

5.3.1 Synthesis Process

Synthesis means that functional information in a VHDL file is translated into a gate-level description (netlist). Generated structural netlist, which is a hardware representation of a circuit, is also optimized for XCV300E Xilinx device at this state. Although the Xilinx design environment supports other VHDL synthesis tools such as LeonardoSpectrum and Synplify, its own program, Xilinx Synthesis Technology (XST), is used.

Figure 5-4 below shows the XST flow. VHDL design file and SystemCrafter gate-level library (crafter_gatelibrary.vhd) are inputs to the synthesis tool. A NGC file, which is a file format of Xilinx netlist, is produced as an output. Note that *speed* is chosen as the optimization goal type and *high* optimization effort is selected. Default values are used for the remaining options.

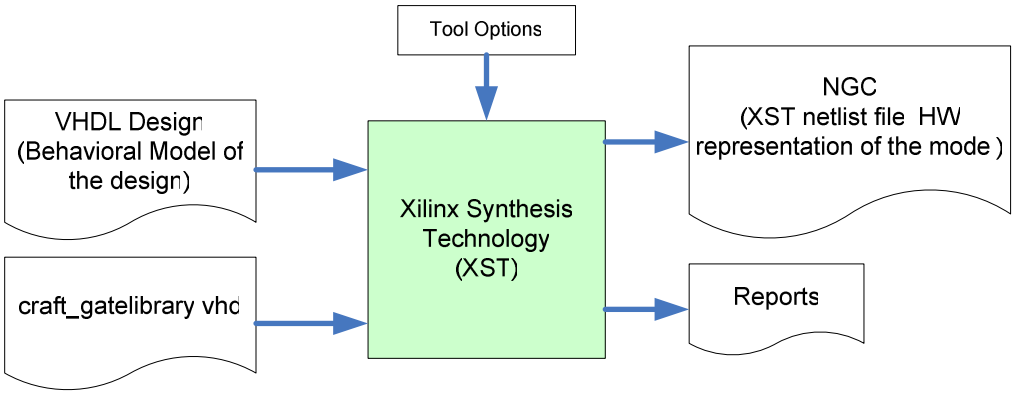

Figure 5-4: Xilinx Synthesis Technology (XST) Flow

Summary of the synthesis reports are as shown in Table 5-1, Table 5-2 and Figure 5-5 below. Please note that, synthesis report is just estimation. The final circuit architecture and clock speed is determined after the *place and route*

processes. According to the report summaries, 1022 slices (33% of XCV300E) are occupied and maximum clock frequency is 31.960MHz.

Table 5-1: Device Utilization Summary

Logic Utilization	Used	Available	Utilization
Number of Slices	1022	3072	33%
Number of Slice Flip Flops	258	6144	4%
Number of 4 input LUTs	1917	6144	31%

Table 5-2: Timing Summary

Minimum period	31.289ns
Maximum Frequency	31.960MHz
Minimum input arrival time before clock	3.082ns
Maximum output required time after clock	35.895ns
Maximum combinational path delay	No path found

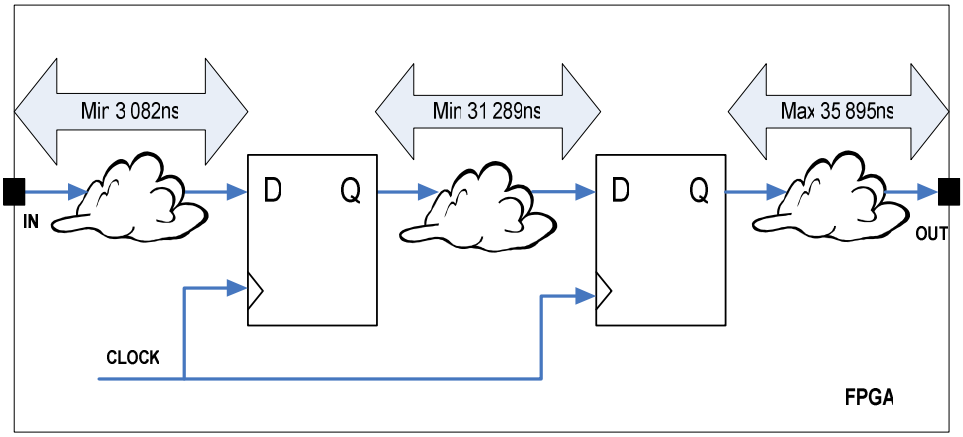

Figure 5-5: Timing Summary

5.3.2 Implementation Process

Implementation process consists of mapping, placement and routing of a logical design into the targeted Xilinx device. At this stage, the logical design file, which is output of the synthesis stage, is converted into a native circuit description (NCD file), which contains hierarchical components used to develop the design and the Xilinx primitives.

The process includes three major parts:

1. Translate (NGDBuild)
2. Mapping (MAP)
3. Placement and Routing (PAR)

The t*ranslation stage* merges multiple design netlist files into a single NGD file, which describes the logic design with Xilinx Native Generic Database (NGD) primitives and original design hierarchy. Simplified translation flow is shown in Figure 5-6 below.

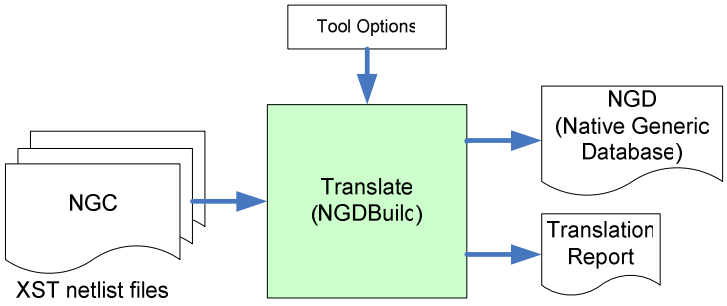

Figure 5-6: Translation Stage

At the *mapping state*, logical components from the netlist are assigned to physical elements which are Configurable Logic Blocks (CLBs) and Input-Output Blocks (IOBs) within the FPGA. NGD is an input file to MAP process and the output is a NCD (Native Circuit Description) file, which represents the design mapped to the components in the target Xilinx FPGA. Figure 5-7 below displays the MAP process. In addition, Table 5-3 below shows the summary of the performed MAP report.

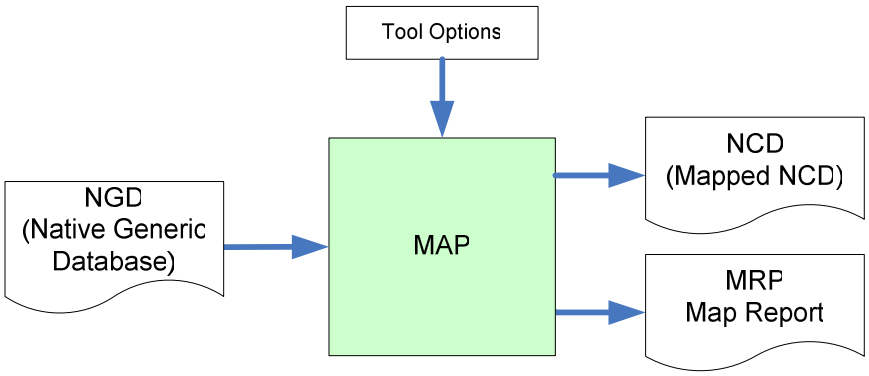

Figure 5-7: MAP Process

At place and route stage, logic components are placed onto the target chip, and they are connected (routed) to each other. Besides, timing information is extracted into PAR report file. As shown in Figure 5-8, mapped NCD file is converted to the placed and routed NCD that is used to generate bit stream file by BitGen program. A summary of the post place and route static timing report is shown in Table 5-4 below.

Table 5-3: MAP Report Summary

Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	217	6,144	3%
Number of 4 input LUTs	1,905	6,144	31%
Logic Distribution			
Number of occupied Slices:	1,020	3,072	33%
Number of Slices containing only related logic	1,020	1,020	100%
Number of Slices containing unrelated logic	0	1,020	0%
Total			
Number 4 input LUTs	1,930	6,144	31%
Number used as logic	1,905		
Number used as a route-thru	25		
Number of bonded IOBs	61	158	38%
Number of GCLKs	1	4	25%
Number of GCLKIOBs	1	4	25%

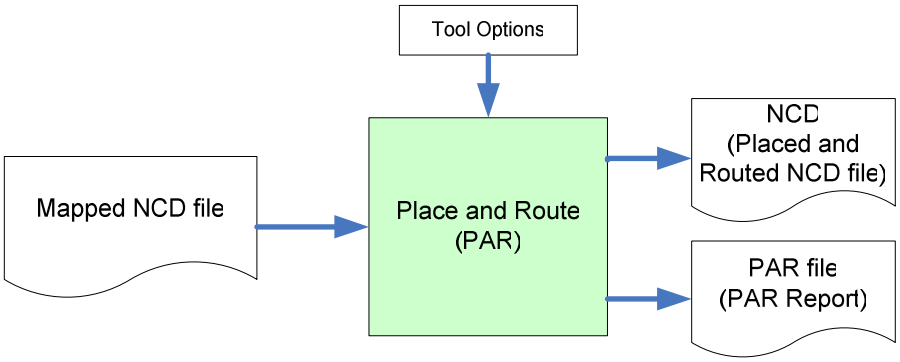

Figure 5-8: Place and Route Stage

Table 5-4: Summary of the Post Place and Route Static Timing Report

Minimum period	29.052ns
Maximum frequency	34.421MHz
Longest delay 8.593ns logic (29.8%) 20.278ns route (70.2%)	28.871ns

5.3.3 Simulations

Simulation can be defined as an execution of a design to emulate the real-word functionality. Generated VHDL codes were simulated via the integrated Xilinx ISE Simulator. Xilinx supported simulators such as Model Technology ModelSim, Cadence NC-SIM and Synopsys VCS-MX, which can also be used for this purpose. Functional and timing simulations are two main categories. A testbench (stimulus), which is a VHDL file containing test vectors, is designed to drive all the simulations. VHDL testbench can be found in the CD-ROM attached to back cover of the thesis.

5.3.3.1 Functional (Behavioral) Simulation

The behavioral simulation is handled prior to the synthesis and implementation processes. This type of early stage simulation provides an idea about the functionality of the VHDL modules.

To verify the VHDL codes generated by SystemCrafter, first behavioral simulation is performed. The simulation results prove the functionality of the generated VHDL codes that describe the Arithmetic and Logic Unit of the processor cores. The results of the simulation can be found in Appendix-C.

5.3.3.2 Timing Simulation

The timing simulation is applied to verify the timing behavior of the circuit, which is implemented into a specific FPGA. For this purpose, a timing simulation netlist is generated from the mapped, placed or routed design. This process is called *back-annotation process*, which is performed by NetGen program.

Post-place and route timing simulation is used to verify the timing characteristics of the implemented ALU design, which verifies the execution of the actual Arithmetic and Logic Unit. Timing simulation flow is displayed in Figure 5-9 below and the results of the simulation can be found in Appendix-C.

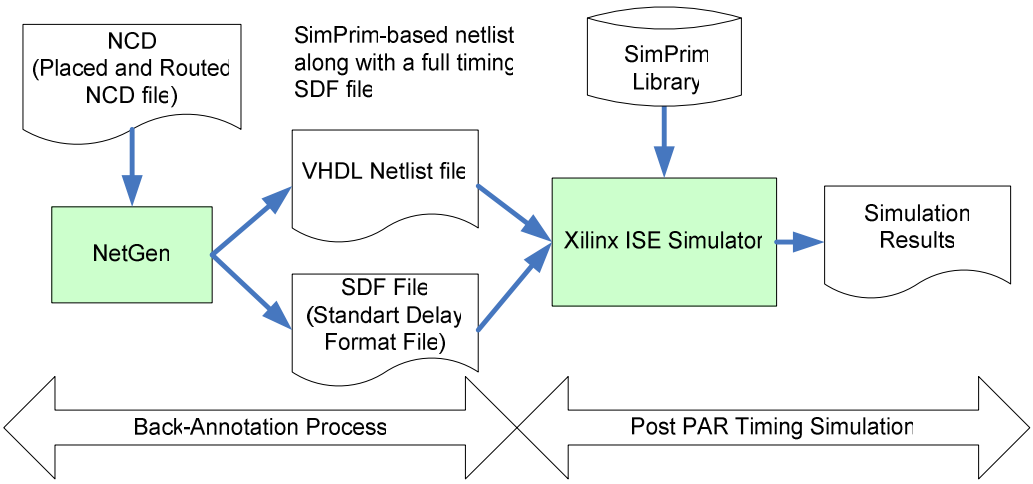

Figure 5-9: Timing Simulation Flow

CHAPTER 5

CONCLUSIONS

In this thesis, two SystemC RTL models of a processor core, whose instruction set and execution cycles are compatible with that of the TI MSP430 microcontroller family, are implemented by exerting classical hardware modeling ability of the SystemC language. In addition, adequate testbenches are supplied in all of the design stages. Besides, SystemC to hardware flow is illustrated by implementing the arithmetic and logic unit of the processor into a Xilinx-FPGA device.

The MSP430 low power microcontroller family mainly consists of 16-bit RISC –like CPU, memory-mapped peripherals, a flexible clock system and linear memory space. All of the modules are combined together via the von-Neumann architecture where common memory data and address busses are shared. With its bus structure and easily connected analog and digital peripherals, MSP430 is considered a good solution for mixed-signal applications. In this study, processor core of MSP430 is focused on. As well, the most remarkable impression of the processor probably results from the instruction set design, excellent addressing modes and registers that generate commonly used constants.

MSP430 has easy to use 27 core instructions. The double operand core instructions are 100% orthogonal. On the other hand, the single operand instructions may be classified as highly orthogonal since some exceptions exist. Besides, 24 instructions can be emulated by utilizing constant registers and core instructions.

In the implementations, some special shortcuts are used to infer three of the addressing modes because there is not enough space in the instruction word; only two bits for seven possibilities. To become clearer, the immediate addressing

mode is coded by employing the program counter (PC) as a source register with the register indirect auto-increment mode. Similarly, when PC is applied as a source or destination register with the indexed addressing mode, the symbolic mode is inferred. Besides, by exploring the fact that using the status register with addressing modes other than the register direct makes little sense, status register is modified in order to generate some constants and to infer the absolute addressing mode instead.

In addition to its several advantages, the processor part of the MSP430 microcontroller has some disadvantages. Firstly, fetching, decoding and execution phases of the single-cycle instructions are performed within one-clock cycle. This architecture is expected to limit the clock speed dramatically. Secondly, destination operands cannot be addressed with the indirect register or indirect register with auto-increment addressing modes, which stems from the limitation of the instruction word length. The destination can be targeted indirectly only via the indexed addressing mode, which needs one more instruction word for the index 0 and an extra clock cycle to fetch this value from the memory. Thirdly, the auto-decrement addressing mode is not supported completely, which results in the need of some more design to implement the PUSH instruction. Lastly, genereted constants are only employed in the register direct mode. It means that, for example, they can not be used for the index values.

Before the implementation stage of the SystemC processor cores, the instruction set architecture and addressing modes are inspected in terms of the execution cycles and desirable memory locations. Next, some design alternatives are explored to realize processor activities. Design decisions, which are made at the early stage of the design, impact on the cost and the performance of the result. In this study, the first approach brings some difficulties to the control and datapath units in terms of the logic and design complexity. However, the first processor can tolerate more logic delay at the instruction decode and instruction execution phases. The second processor, which has been produced depending on the outcomes of the study done on the first approach, has proved to be more convenient with respect to logic area.

The instruction set and addressing modes of the RTL SystemC processor cores were simulated thoroughly. In addition, original program codes for 16-bit and 32-bit cyclic redundancy code (CRC) algorithms delivered by Texas Instrument Inc. were used to verify the functionality of the processor models regarding whether they are working properly or not. All of the assembler codes are compiled by IAR Embedded Workbench. Then, generated machine codes were transferred to the stimulus part of the SystemC project. The simulation results performed both in SystemC and IAR workbench environment were compared, which verifies the functionality of the SystemC processor cores.

SystemC to hardware flow is also illustrated by synthesizing the *Arithmetic and Logic Unit* part of the processor into hardware. For this purpose, SystemCrafter synthesis tool is utilized to convert the SystemC descriptions into VHDL codes. However, SystemC is originally designed as a system-level simulation language, and it is not possible to compile all of the SystemC constructs to hardware, which is similar to the traditional hardware description languages (HDLs). Similarly, only a small subset of the SystemC language is supported by SystemCrafter. Hence, original SystemC model of the *Arithmetic and Logic Unit* is slightly modified. The most significant change is the conversion of the asynchronous process of the ALU model into synchronous process, since the current version of SystemCrafter does not support asynchronous operations. At this point, it is advised to make a start on the SystemC project by employing the supported subset of the language instead of full language constructs.

Although SystemCrafter was successfully employed to generate VHDL codes, it is almost impossible to trace the codes generated. Besides, these VHDL outputs are not recommended to be used as a useful metric to have an idea about the size of the design because the synthesis process introduces redundant logic, which is expected to be removed by downstream tools.

Following the SystemC to VHDL conversion process, Xilinx ISE, which is FPGA design and verification environment, was employed. Firstly, behavioral simulation of the generated VHDL codes is performed in order to verify RTL VHDL descriptions. Then, the VHDL codes are synthesized into a gate-level description

(netlist). After that, synthesized logic is implemented into XCV300E FPGA. Mapped logic occupies 1020 slices (33% of XCV300E), and maximum clock frequency is 34.421 MHz, which seems the generated VHDL codes are poor in terms of chip area and clock speed. This problem can be solved by direct synthesizing the RTL SystemC descriptions into netlist. Lastly, timing simulation is applied to verify the timing behavior of the implemented circuit. Hence, SystemC RTL model of the *Arithmetic and Logic Unit* is successfully converted into working real hardware and execution of the logic is verified.

Last of all, with its true von-Neumann architecture, SystemC IP cores of the MSP430 processor can be successfully applied to a variety of SoC applications. After the verification of the interaction of the MSP430 compatible core with its peripherals by the help of instruction set simulator, designed RTL SystemC models of the processor can be used in the implementation state of the system. In addition, modular architecture of the processor cores can be easily improved to support more addressing modes and expand the use of the constant registers. Besides, clock speed can be increased by employing the pipelining concept for the fetch, decode and execution phases. Finally, the MSP430 processor cores can be a reference study to explore and design the extended version, namely, MSP430X.

REFERENCES

[1] Bacchini, F. Smith, G. et al. , “Building a Common ESL Design and Verification Methodology - Is it Just a Dream?,” Panel summary. 43rd Design Automation Conference, pp.370-371, 24-28 July 2006

[2] Open SystemC Initiative, “Functional Specification for SystemC Version 2.0”, 2002

[3] Open SystemC Initiative, “SystemC Version 2.0 User’s Guide, Update for SystemC 2.0.1”, 2002

[4] Open SystemC Initiative, “SystemC 2.0.1 Language Reference Manual Revision 1.0”, 2003

[5] IEEE Computer Society, “IEEE Standard SystemC Language Reference Manual”, New York, USA, March 2006

[6] Summit Design Inc., www.summit-design.com , Last accessed; December 2006

[7] CoWare Inc., www.CoWare.com, Last accessed; December 2006

[8] Organization of the open cores, www.opencores.org, Last accessed; December 2006

[9] Jonsson, B., “A JPEG Encoder in SystemC”, A thesis submitted to Lulea University of Technology in the Department of Computer Science and Electrical Engineering, Tokyo, 2005

[10] Mahmoudi, L.; Ayough, A.; Abutalebi, H.; Nadjarbashi, O. F.; Hessabi S., “Verilog2SC: A Methodology for Converting Verilog HDL to SystemC,” Proceedings of the 11th International HDL Conference, pp. 211-217, 2002

[11] Fin, A.; Fummi, F.; Pravadelli, G. , “AMLETO: A Multi-language Environment for Functional Test Generation”, IEEE, pp. 821-829, 2001

[12] Kesen, L. , “Implementation of an 8-bit Microcontroller with SystemC”, A thesis submitted to the Graduate School of Natural and Applied Sciences of Middle East Technical University in The Department of Electrical and Electronics Engineering, Ankara, November 2004

[13] Sözen, S., “A Viterbi Decoder Using SystemC for Area Efficient VLSI Implementation”, A thesis submitted to the Graduate School of Natural and Applied Sciences of Middle East Technical University in The Department of Electrical and Electronics Engineering, Ankara, September 2006

[14] Lenchak, E., “CRC Implementation with MSP430”, Texas Instruments Inc., 2004

[15] Synthesis Working Group of OSCI, “SystemC Synthesizable Subset”, 2004

[16] SystemCrafter Inc., “SystemCrafter SC User Manual Version 2.0.0.3”, 2005

[17] Synopsys, “Synopsys CoCentric SystemC Compiler: RTL User and Modeling Guideline”, 2001

[18] Prosilog Inc., “Prosilog’s SystemC Compiler Datasheet”, 2002

[19] Liao, S.; Tjiang, S.; Gupta. R., “An efficient implementation of reactivity for modeling hardware in the Scenic design environment,” Proceedings of the Design Automation Conference, pp. 70-75, 1997

[20] Wolfe, V.; Davidson, S.; Lee, I., “RTC: language support for real-time concurrency,” Proceeding of the Real-Time Systems Symposium, pp.43-52, 1991

[21] Cai, L.; Gajski, D., “Transaction Level Modeling: An Overview,” Proceedings of the International Conference on Hardware/Software Codesign & System Synthesis, pp.19-24, 1-3 October 2003

[22] Swan, S., “SystemC Transaction Level Models And RTL Verification, ” Proceedings of the 43rd annual conference on Design Automation Conference, pp.90-92, 24-28 July 2006

[23] Grötker,T.; Liao, S. et al. ”System Design with SystemC”. Boston: Kluwer Academic Bublishers, 2002

[24] Black, D.C.; Donovan, J., “SystemC: From The Ground Up”, USA: Eklectic Ally, Inc., 2004

[25] Texas Instruments Inc., “MSP430x1xx Family User’s Guide”, 2003

[26] Underwood, S., “mspgcc-A port of the GNU tools to the Texas Instruments MSP430 microcontrollers” ,2003

[27] Bierl, L. “MSP430 Family Mixed-Signal Microcontroller Application Reports”, USA:Texas Instruments Inc.

[28] Texas Instruments Inc. ,"MSP430x4xx Family User's Guide", pp.41-76, 2006

[29] Texas Instruments Inc., "Application Report: Choosing An Ultralow-Power MCU", 2004

[30] Xilinx, "XST User Guide 8.1i", 2006

[31] Xilinx, "Synthesis and Simulation Design Guide", 2006

[32] Xilinx, "Development System Reference Guide", 2006

[33] Maxfield, C.M. "The Design Warrior's Guide to FPGAs", USA: Newnes, 2004

[34] Saul, J. "Using SystemC and SystemCrafter to Implement Data Encryption", Xcell Journal, Issue 58, pp.32-34, 2006

[35] Mano, M.M. "Computer System Architecture". USA: Prentice-Hall International, 1993

[36] Mano, M.M.; Kime, C.R., "Logic and Computer Design", USA: Prentice-Hall International, 1997

[37] Morton, G.; Venkat, K., "Application Report: MSP430 Competitive Benchmarking", Texas Instruments, 2006

APPENDIX A

INSTRUCTION SET OF MSP430

A.1 Double Operand Instructions

Table A-1: Double Operand Instructions

Mnemonics	Operation		Status Bits			
	In words	In RTL	V	N	Z	C
ADD(.B)	Addition	src + dst → dst	*	*	*	*
ADDC(.B)	Addition with carry	src+dst+C→dst	*	*	*	*
AND(.B)	Logic “and” operation	src and dst → dst	0	*	*	*
BIT(.B)	Bit test operation	src and dst	0	*	*	*
BIC(.B)	Bit clear operation	not(src) and dst→dst	-	-	-	-
BIS(.B)	Bit set operation	src or dst → dst	-	-	-	-
DADD(.B)	Decimally add operation	src + dst + C → dst (decimally)	*	*	*	*
MOV(.B)	Copy the content of source to destination	src → dst	-	-	-	-
SUB(.B)	Subtraction	dst+not(src)+1→dst	*	*	*	*
CMP(.B)	Compare operation	dst-src	*	*	*	*
SUBC(.B)	Subtraction with carry	dst+not(src)+C→dst	*	*	*	*
XOR(.B)	Logic “xor” operation	src xor dst → dst	*	*	*	*

Table A-2: Execution Cycle / Needed Word Locations for the Double Operand Instructions

		Destination Addressing Modes	
		Register Direct	Indexed, Symbolic, Absolute
Source Addressing Modes	Register Direct Mode	1*/1	4/2
	Indirect Register Mode, Indirect Reg. with Auto-increment,	2*/1	5/2
	Immediate Mode	2*/2	2*/3
	Indexed, Symbolic, Absolute Modes	3/2	6/3

***: Add one cycle if destination register is program counter (PC)**

A.2 Single Operand Instructions

Table A-3: Single Operand Instructions

Mnemonics	Operation		Status Bits			
			V	N	Z	C
RRC(.B)	Rotate Right through carry	C➔MSB➔ .... ➔LSB➔C	*	*	*	*
RRA(.B)	Rotate Right arithmetically	MSB➔MSB➔ .... ➔LSB➔C	0	*	*	*
PUSH(.B)	push	SP-2➔SP, src➔@SP	-	-	-	-
SWPB	Swap bytes	Operand[15:8] ←➔Operand[7:0]	-	-	-	-

Table A-3 (continued)

Mnemonics	Operation		Status Bits			
			V	N	Z	C
CALL	CALL a subroutine	SP-2➔SP, Addr. of next inst. ➔@SP, dst➔PC	-	-	-	-
RETI	Return from interrupt	TOS➔ SR, SP+2➔ SP TOS➔PC, SP+2➔ SP	*	*	*	*
SXT	Sign extension	Bit7➔Bit8...Bit15	0	*	*	*

Table A-4: Execution Cycle / Needed Word Locations for the Single Operand Instructions

Addressing Mode	Execution Clock Cycles/ Necessary Memory Locations		
	RRA, RRC, SWPB, SXT	PUSH	CALL
Register Direct Mode	1/1	3/1	4/1
Indirect Register Mode	3/1	4/1	4/1
Register indirect addressing with auto-increment mode	3/1	4/1	5/1
Immediate addressing mode	*3/2	4/2	5/2
Indexed mode addressing mode	4/2	5/2	5/2
Symbolic addressing mode	4/2	5/2	5/2
Absolute addressing mode	4/2	5/2	5/2

(*) It is not recommended to use immediate mode with RRA, RRC, SWPB, SXT instructions.

A.3 JUMP Instructions

Table A-5: Jump Instructions

Mnemonics	Operation	Application
JNE/JNZ	Jump if Z==0 (if !=)	After comparisons: src ≠ dst Test for nonzero contents
JEQ/JZ	Jump if Z==1 (if ==)	After comparisons: src = dst Test for zero contents
JNC/JLO	Jump if C==0 (if unsigned <)	After unsigned comparisons: dst < src Test for a reset carry
JC/JHS	Jump if C==1 (if unsigned >=)	After unsigned comparisons: dst ≥ src Test for a set carry
JN	Jump if N==1 Note there is no "JP" if N==0!	Test for the sign of a result: dst < 0
JGE	Jump if N==V (if signed >=)	After signed comparisons: dst ≥ src
JL	Jump if N!=V (if signed <)	After signed comparisons: dst ≤ src
JMP	Jump unconditionally	Program control transfer

A.4 Emulated Instructions

Table A-6: Emulated Instructions

Mnemonics	Used Core Instruction	Description	Operation	Status Bits			
				V	N	Z	C
ADC(.B)	ADDC(.B) #0,dst	Add C to destination	dst + C → dst	*	*	*	*
BR, BRANCH	MOV dst, PC	Branch to destination	dst → PC	-	-	-	-
CLR(.B)	MOV(.B) #0,dst	Clear destination	0 → dst	-	-	-	-
CLRC	BIC #1,SR	Clear C	0 → C	-	-	-	0
CLRN	BIC #4,SR	Clear N	0 → N	-	0	-	-
CLRZ	BIC #2,SR	Clear Z	0 → Z	-	-	0	-
DADC(.B)	DADD(.B) #0,dst	Add C decimally to destination	dst+C→dst (decimally)	*	*	*	*
DEC(.B)	SUB(.B) #1,dst	Decrement destination	dst-1 → dst	*	*	*	*
DECD(.B)	SUB(.B) #2,dst	Decrement destination	dst-2 → dst	*	*	*	*
DINT	BIC #8,SR	Disable interrupts	0 → GIE	-	-	-	-
EINT	BIS #8,SR	Enable interrupts	1 → GIE	-	-	-	-

Table A-6 (continued)

Mnemonics	Used Core Instruction	Description	Operation	Status Bits			
				V	N	Z	C
INC(.B)	ADD(.B) #1,dst	Increment destination	dst +1 → dst	*	*	*	*
INCD(.B)	ADD(.B) #2,dst	Double-increment destination	dst +2 → dst	*	*	*	*
INV(.B)	XOR(.B) #0FFh,dst	Invert destination	not(dst) → dst	*	*	*	*
NOP	MOV #0, R3	No operation	0 → R3	-	-	-	-
POP(.B)	MOV(.B) @SP+, dst	Pop item from stack to destination	@SP → dst, SP+2→SP	-	-	-	-
RET	MOV @SP+,PC	Return from subroutine	@SP→PC, SP+2→SP	-	-	-	-
RLA(.B)	ADD(.B) dst, dst	Rotate left arithmetically	C←MSB←…←LSB ←0	*	*	*	*
RLC(.B)	ADDC(.B) dst, dst	Rotate left through C	C←MSB←…←LSB ←C	*	*	*	*
SBC(.B)	SUBC(.B) #0,dst	Subtract not(C) from destination	dst+FFFFh+C→dst	*	*	*	*
SETC	BIS #1,SR	Set C	1 → C	-	-	-	1
SETN	BIS #4,SR	Set N	1 → N	-	1	-	-
SETZ	BIS #2,SR	Set Z	1 → C	-	-	1	-
TST(.B)	CMP(.B) #0,dst	Test destination	dst+FFFFh+1	0	*	*	1

Table A-7: Decoding the Source Operand Addressing Modes

As	Source Reg.	Description	Addressing Mode	
00	R0-PC	PC content is used as operand	Register Direct Mode	
00	R1-SP	SP content is used as operand		
00	R2-SR	SR content is used as operand		
00	R3-CG	Constant “0” is used as operand		
00	Rn*	Rn content is used as operand		
01	R3-CG	Constant “+1” is used as operand	Register Direct Mode	
01	R0-PC	X(PC) , effective address=X+PC	Symbolic Mode	Indexed Mode
01	R1-SP	X(SP), effective address=X+SP	Indexed Mode	
01	R2-SR	X(0), effective address=X	Absolute Mode	
01	Rn*	X(Rn), effective address=X+Rn	Indexed Mode	
10	R0-PC	@PC, dangerous to use, not recommended	Indirect Register Mode	
10	R1-SP	@SP, effective address=SP		
10	R2-SR	Constant “+4” is used as operand	Register Direct Mode	
10	R3-CG	Constant “+2” is used as operand		
10	Rn*	@Rn, effective address=Rn	Indirect Register Mode	
11	R0-PC	@PC+, immediate constant follows the instruction.	Immediate Mode	
11	R1-SP	@SP+, effective address=SP	Indirect Auto-increment	
11	R2-SR	Constant “+8” is used as operand	Register Direct Mode	
11	R3-CG	Constant “-1” is used as operand		
11	Rn*	@Rn+, effective address=Rn	Indirect Auto-increment	

(*) n=4 to 15

Table A-8: Decoding the Destination Operand Addressing Mode

Ad	Dst. Reg.	Description	Addressing Mode	
0	R0-PC	PC content is used as operand	Register Direct Mode	
0	R1-SP	SP content is used as operand		
0	R2-SR	SR content is used as operand		
0	R3-CG	R3 is non-writeable	-	
0	Rn*	Register contents are operand	Register Direct Mode	
1	R3-CG	Do not use	-	
1	R0-PC	X(PC), effective address=X+PC	Symbolic Mode	Index Mode
1	R1-SP	X(SP), effective address=X+SP	Indexed Mode	
1	R2-SR	X(0), effective address=X	Absolute Mode	
1	Rn*	X(Rn), effective address=X+Rn	Indexed Mode	

(*) n=4 to 15

APPENDIX B

INSTRUCTION-PIPELINED PROCESSOR

In this part, RTL descriptions of the first processor, processor-1, are explained. For details of the addressing modes, please refer to the Addressing Modes part of the thesis. The section is divided into three parts, namely, double operand instructions, single operand instructions and jump instructions.

B.1 Double Operand Instructions

Table B-1: Register to Register Addressing Mode Combination (Rsrc_Rdst)

Rsrc_Rdst		
Current State	**Operations**	**Next State**
Rsrc_Rdst (10)	Rdst ←Rdst *op.* Rsrc, if Rdst≠PC, PC ← PC+2 Fetch&decode the next inst.	Rdst≠PC → depends on the next instruction
		Rdst=PC → fetch_st
fetch_st (1)	Fetch&decode the next inst.	depends on the next inst.

Table B-2: Indirect Register, Indirect Register Auto-increment, and Immediate to Register Addressing Mode Combinations

@Rsrc_Rdst, @Rsrc+_Rdst		
Current State	**Operations**	**Next State**
at_Rsrc_Rdst (20)	Rdst← Rdst *op.* M[Rsrc], Rd← Rdst *op.* M[Rsrc], if #N (@PC+), PC←PC+2	Rdst≠PC ➔ fecth_st
		Rdst=PC➔at_Rsrc_Rdst_1
at_Rsrc_Rdst_1 (21)	PC←Rd	fecth_st
fetch_st(1)	if @Rscr+, Rsrc←Rsrc+2or1, Fetch&decode the next inst.	depends on the next inst.

Table B-3: Indexed, Symbolic, Absolute to Register Addressing Mode Combinations (X(Rsrc)_Rdst)

X(Rsrc)_Rdst		
Current State	**Operations**	**Next State**
XRsrc_Rdst (30)	Ra←M[PC] + Rsrc , PC←PC+2	XRsrc_Rdst_1
XRsrc_Rdst_1 (31)	Rdst←M[Ra] op. Rdst	fetch_st
fetch_st (1)	PC←PC+2, Fetch&decode the next inst.	depends on the next inst.

Table B-4: Register to Indexed, Symbolic, Absolute Addressing Mode Combinations (Rsrc_YRdst)

Rsrc_YRdst		
Current State	**Operations**	**Next State**
Rsrc_YRdst (40)	Ra←M[PC]+Rdst	Rsrc_YRdst_1
Rsrc_YRdst_1(41)	Rd←M[Ra] *op.* Rsrc	Rsrc_YRdst_2
Rsrc_YRdst_2(42,64)	M[Ra]←Rd	fetch_st
fetch_st (1)	PC←PC+2 Fetch&decode the next inst.	depends on the next inst.

Table B-5: Indirect Register, Indirect-Auto Increment, Immediate to Indexed, Symbolic, Absolute Addressing Mode Combinations (@Rsrc_YRdst)

@Rsrc_YRdst, @Rsrc+_YRdst		
Current State	**Operations**	**Next State**
at_Rsrc_YRdst(50)	Rd←M[Rsrc]	at_Rsrc_YRdst_1
at_Rsrc_YRdst_1 (51)	Ra←M[PC]+Rdst, if #N (@PC+), PC←PC+2	at_Rsrc_YRdst_2
at_Rsrc_YRdst_2 (52,63)	Rd←Rd *op.* M[Ra]	at_Rsrc_YRdst_3
at_Rsrc_YRdst_3 (53, 64)	M[Ra]←Rd	fetch_st
fetch_st (1)	PC←PC+2 if @Rscr+, Rsrc←Rsrc+2 Fetch&decode the next inst.	Depends on the next inst.

Table B-6: Indexed, Symbolic, Absolute to Indexed, Symbolic, Absolute Addressing Mode (XRsrc_YRdst)

XRsrc_YRdst		
Current State	**Operations**	**Next State**
XRsrc_YRdst(60)	Ra←M[PC] + Rsrc, PC←PC+2	XRsrc_YRdst_1(61)
XRsrc_YRdst_1(61)	Rd←M[Ra]	XRsrc_YRdst_2(62)
XRsrc_YRdst_2(62)	Ra←M[PC] + Rdst, PC←PC+2	XRsrc_YRdst_3(63)
XRsrc_YRdst_3(63)	Rd←Rd *operator* M[Ra]	XRsrc_YRdst_4(64)
XRsrc_YRdst_4(64)	M[Ra]←Rd	fetch_st
fetch_st(cs=1)	PC←PC+2 Fetch&decode the next inst.	Depends on the next inst.

B.2 Single Operand Instructions

Single operand instructions are divided into four groups, namely,

- SWPB, SXT, RRA and RRC Instructions (SSRR)
- PUSH Instruction
- CALL Instruction
- RETI Instruction

SWPB, SXT, RRA and RRC Instructions

Table B-7: SWPB, SXT, RRC, and RRA Instructions with the Register Addressing Mode

SSRR Rdst		
Current State	**Operations**	**Next State**
Rdst_SSR (100)	Rdst ← *op.*(Rdst) PC← PC+2, Fetch&decode the next inst.	fecth_st

Table B-8: SWPB, SXT, RRC, and RRA Instructions with the Indirect Register Mode, Indirect Auto-Increment Mode, Immediate mode

SSRR @Rdst, SSRR @Rdst+, SSRR @PC+/#N		
Current State	**Operations**	**Next State**
at_Rdst_SSR(200)	Rd←operator(M[Rdst])	at_Rdst_SSR_1
at_Rdst_SSR_1 (201)	M[Rdst]←Rd, if #N, PC← PC+2	fetch_st
fetch_st (1)	If @Rdst+, Rdst←Rdst+2 PC←PC+2 Fetch&decode the next inst.	Depends on the next inst.

Table B-9: RTL Descriptions and State Transitions of the SWPB, SXT, RRC, and RRA Instructions with the Indexed Mode, Symbolic Mode, and Absolute Mode

SSRR X(Rdst)		
Current State	**Operations**	**Next State**
XRdst_SSR(300)	Ra←M[PC]+Rdst, PC←PC+2	XRdst_SSR_1
XRdst_SSR_1(301)	Rd←op.M[Ra]	XRdst_SSR_2
XRdst_SSR_2(302,64)	M[Ra]←Rd	fetch_st
fetch_st	PC←PC+2 Fetch&decode the next inst.	Depends on the next inst.

PUSH Instruction

Table B-10: RTL Descriptions and State Transitions of the PUSH Instruction with the Register Direct Mode

PUSH Rdst		
Current State	**Operations**	**Next State**
SP_2 (400)	SP←SP-2	Rsrc_PUSH
Rsrc_PUSH(401)	M[SP]←Rsrc	fetch_st
fetch_st(1)	PC←PC+2 Fetch&decode the next inst.	Depends on the next inst.

Table B-11: RTL Descriptions and State Transitions of the PUSH Instruction with the Indirect Register, Indirect Auto-Increment, and Immediate modes

PUSH @Rdst , PUSH @Rdst+, PUSH #N		
Current State	**Operations**	**Next State**
SP_2 (400)	SP←SP-2	at_Rsrc_PUSH
at_Rsrc_PUSH(500)	If #N, PC←PC+2 Rd←M[Rsrc]	at_Rsrc_PUSH_1
at_Rsrc_PUSH_1(501)	M[SP]←Rd	fetch_st
fetch_st(1)	if @Rsrc+, Rsrc←Rsrc+2 Fetch&decode the next inst.	Depend on the next inst.

Table B-12: RTL Descriptions and State Transitions of the PUSH Instruction with the Indexed, Symbolic, Absolute Modes

PUSH X(Rdst)		
Current State	**Operations**	**Next State**
SP_2(400)	SP←SP-2	XRsrc_PUSH
XRsrc_PUSH(600)	Ra←M[PC]+Rsrc, PC←PC+2	XRsrc_PUSH_1
XRsrc_PUSH_1(601)	Rd←M[Ra]	at_Rsrc_PUSH_1
at_Rsrc_PUSH_1(501)	M[SP]←Rd	fetch_st
fetch_st(1)	PC←PC+2, Fetch&decode the next inst.	Depends on the next inst.

CALL Instruction

Table B-13: RTL Descriptions and State Transitions of the CALL Instruction with the Register Direct Mode

CALL Rdst		
Current State	**Operations**	**Next State**
CALL_Rdst(800)	Ra←Rdst Rd←Rdst	CALL_SP_2
CALL_SP_2(801)	SP←SP-2	CALL_2_TOS
CALL_2_TOS(802)	M[SP]←PC	fetch_st
fetch_st(1)	PC←Rd+2 Fetch&decode the instruction pointed by Ra	Depends on the next inst.

Table B-14: CALL Instruction with the Indirect Register Mode

CALL @Rdst		
Current State	**Operations**	**Next State**
CALL_at_Rdst,(810)	Ra←M[Rdst] Rd←M[Rdst]	CALL_SP_2
CALL_SP_2(801)	SP←SP-2	CALL_2_TOS
CALL_2_TOS(802)	M[SP]←PC	fetch_st
fetch_st	PC←PC+2, Fetch&decode the instruction pointed by Ra	Depends on the following inst.

Table B-15: CALL Instruction with the Indirect Register Mode

CALL @Rdst+ / CALL #N		
Current State	**Operations**	**Next State**
CALL_at_Rdst (810)	Ra←M[Rdst], Rd←M[Rdst], if @PC+(=#N), PC←PC+2	CALL_2_TOS
CALL_SP_2 (801)	SP←SP-2	CALL_2_TOS
CALL_2_TOS (802)	M[SP]←PC	CALL_at_Rdstp_3
CALL_at_Rdstp_3 (823)	PC←Rd	fetch_st
fetch_st(1)	if @Rdst+, Rdst←Rdst+2 PC←PC+2, Fetch&decode the next inst.	Depends on the next inst.

Table B-16: CALL Instruction with the Indexed, Absolute, Symbolic Modes

CALL X(Rdst)		
Current State	**Operations**	**Next State**
CALL_XRdst (830)	PC←PC+2, Ra←M[PC]+Rdst	CALL_XRdst_1
CALL_XRdst_1(831)	Ra←M[Ra] Rd←M[Ra]	CALL_SP_2
CALL_SP_2 (801)	SP←SP-2	CALL_2_TOS
CALL_2_TOS (802)	M[SP]←PC	fetch_st
fetch_st (1)	PC←Rd+2, Fetch&decode the inst. pointed by Ra	Depends on the fetched inst.

RETI Instruction

Table B-17: RTL Descriptions and State Transitions of the RETI Instruction

RETI		
Current State	**Operations**	**Next State**
RETI_st (840)	SR←M[SP]	RETI_st_1
RETI_st_1 (841)	SP←SP+2	RETI_st_2
RETI_st_2 (842)	PC←M[SP]	RETI_st_3
RETI_st_3 (843)	SP←SP+2	fetch_st
fetch_st	PC←PC+2 Fetch&decode the inst.	Depends on the fetched inst.

B.3 JUMP Instructions

Table B-18: RTL Descriptions and State Transitions of the Jump Instructions

JUMP		
Execution Clock Cycles	2	
Needed Memory Word	1	
Current State	**Operations**	**Next State**
fecth_decode (1)	PC ← PC+2, Rd← Rmdb	calc_pc
calc_pc (434)	PC ← Rd + PC	fecth_decode

APPENDIX C

XILINX ISE SIMULATOR RESULTS OF THE ALU SIMULATION

This part clarifies the simulation stimulus and expected simulation outputs of the Arithmetic and Logic Unit (ALU). As mentioned before, behavioral and post route timing simulations were performed. The instructions were ordered with respect to the function select input of the ALU. The input values and expected outputs, which are same for both of the simulations, were given with the waveform outputs so as to be verified easily. Although, some explanations were made when necessary, the *instruction set* part of the MSP430 datasheet is suggested being read.

C.1 RRC and SWPB Instructions

RRC and SWPB instructions are single operand instructions and their operands are the data on the destination (DST) port of the ALU.

SWPB instruction does not have byte format and it does not affect the status bits-V, N, Z, and C.

Table C-1: Simulation Inputs and Expected Outputs for the RRC and SWPB Instructions

Input Values					Expected Outputs	
F	B/W	DST	SRC	VNZC	VNZC	Output
0 (RRC.W)	0	x"0000"	-	0000	0010	x"0000"
0 (RRC.W)	0	x"5555"	-	0000	0001	x"2AAA"
0 (RRC.W)	0	x"0001"	-	0000	0011	x"0000"
0 (RRC.W)	0	x"2222"	-	0001	1100	x"9111"
0 (RRC.B)	1	x"5555"	-	0000	0001	x"002A"
0 (RRC.B)	1	x"0001"	-	0000	0011	x"0000"
0 (RRC.B)	1	x"2222"	-	0001	1100	x"0091"
1 (SWPB)	-	x"1234"	-	0000	0000	x"3412"

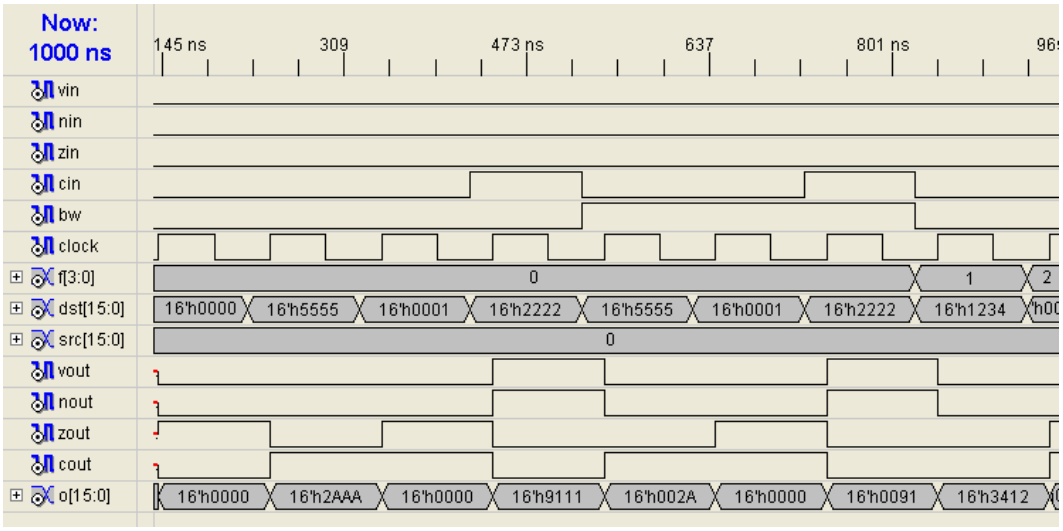

Figure C-1: Behavioral Simulation Waveforms of the RRC and SWPB Single Operand Instructions

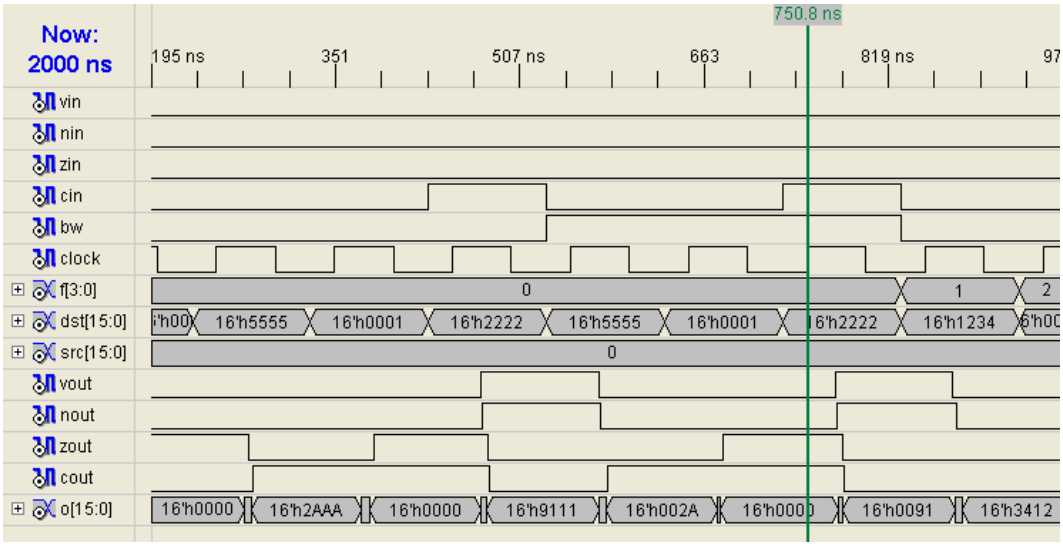

Figure C-2: Post Route Timing Simulation Waveforms of the RRC and SWPB Instructions

C.2 RRA, SXT and MOV Instructions

RRA and SXT are single operand instructions, and the operand is the data on the destination (DST) port of the ALU.

The execution of the RRA instruction resets the overflow status bit (V).

SXT does not have *byte* form and overflow status bit (V) always gets zero value.

The operand of the MOV instruction is the data on the source (SRC) port of the ALU. MOV instruction does not affect the status bits. Thus, previous status information is copied to the status ports.

Table C-2: Simulation Inputs and Expected Outputs for the RRA, SXT and MOV Instructions

Input Values					Expected Outputs	
F	**B/W**	**DST**	**SRC**	**VNZC**	**VNZC**	**Output**
2 (RRA.W)	0	x"0001"	-	0000	0011	x”0000”
2 (RRA.W)	0	x"8888"	-	0000	0100	x”C444”
2 (RRA.B)	1	x"0001"	-	0000	0011	x”0000”
2 (RRA.B)	1	x"8888"	-	0000	0100	x”00C4”
3 (SXT)	0	x"008A"	-	0000	0100	x"FF8A"
3 (SXT)	0	x"0070"	-	0000	0100	x"0070"
4 (MOV)	0	-	x”1982”	0000	0100	x”1982”
4 (MOV)	1	-	x”AB34”	0000	0100	x”0034”

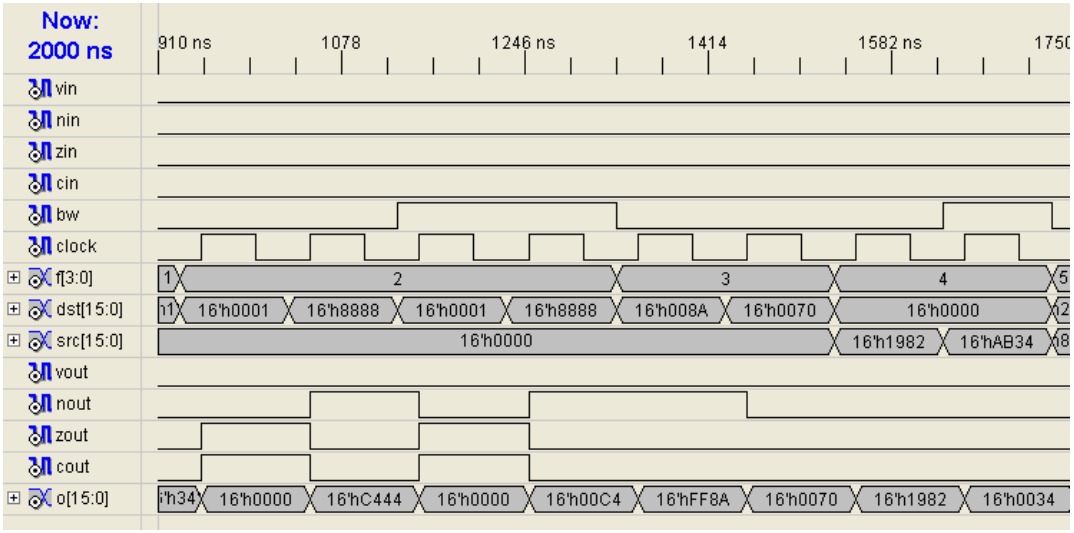

Figure C-3: Behavioral Simulation Waveforms of the RRA, SXT, and MOV Instructions

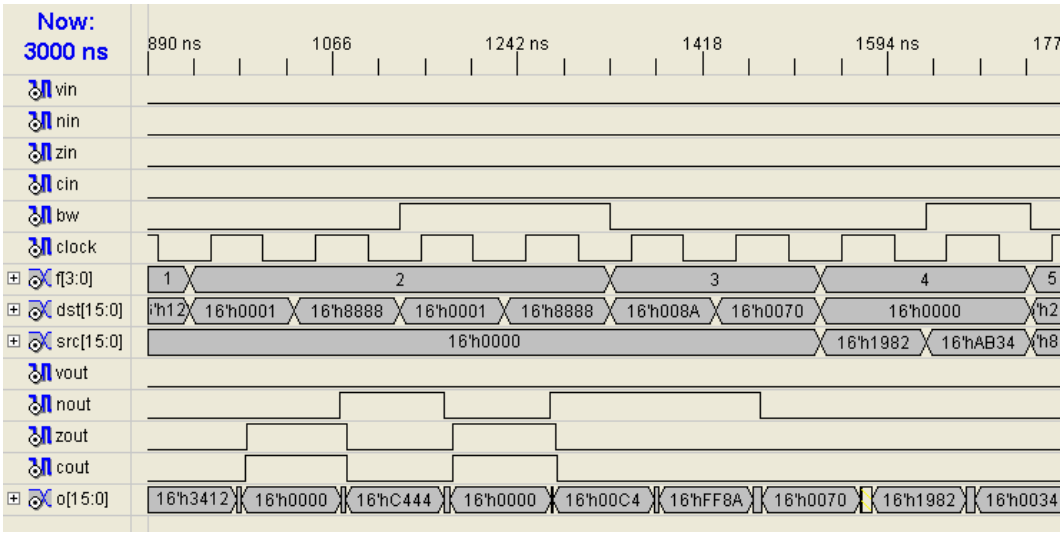

Figure C-4: Post Route Timing Simulation Waveforms of the RRA, SXT, and MOV Instructions

C.3 ADD Instruction

Please note that 2's complement representation is used for arithmetic operands.

Table C-3: Simulation Inputs and Expected Outputs for the ADD Instruction

Input Values					Expected Outputs	
F	B/W	DST	SRC	VNZC	VNZC	Output
5 (ADD.W)	0	x"2222"	x"8888"	0000	0100	x"AAAA"
5 (ADD.W)	0	x"7000"	x"7FFF"	0000	0101	x"EFFF"
5 (ADD.W)	0	x"8001"	x"8001"	0000	1001	x"0002"
5 (ADD.W)	0	x"FFE0"	x"0020"	0000	0011	x"0000"
5 (ADD.B)	1	x"2222"	x"8888"	0000	0100	x"00AA"
5 (ADD.B)	1	x"7E7E"	X"7F7F"	0000	1100	x"00FD"
5 (ADD.B)	1	x"9081"	x"9081"	0000	1001	x"0002"
5 (ADD.B)	1	x"FFE0"	x"0020"	0000	0011	x"0000"

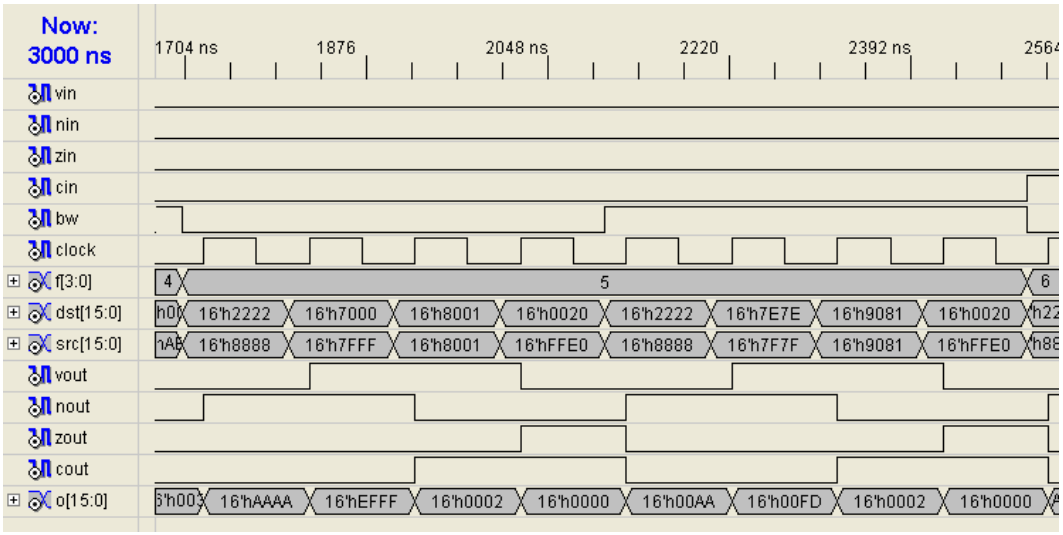

Figure C-5: Behavioral Simulation Waveforms of the ADD Instruction

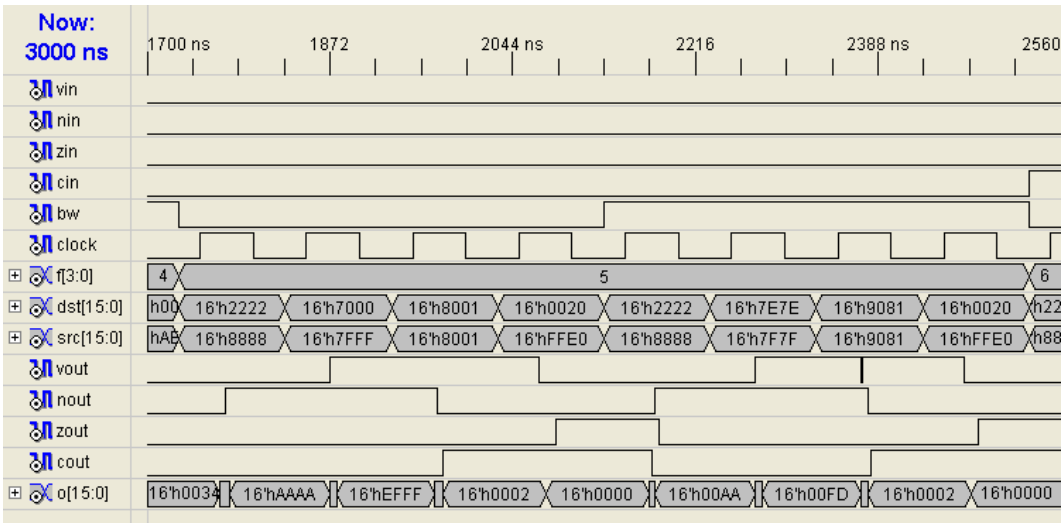

Figure C-6: Post Route Timing Simulation Waveforms of the ADD Instruction

C.4 ADDC Instruction

Table C-4: Simulation Inputs and Expected Outputs for the ADDC Instruction

Input Values					Expected Outputs	
F	B/W	DST	SRC	VNZC	VNZC	Output
6 (ADDC.W)	0	x”2222”	x”8888”	0001	0100	x”AAAB”
6 (ADDC.W)	0	x”7000”	x”7FFE”	0001	0101	x”EFFF”
6 (ADDC.W)	0	x”8001”	x”8002”	0001	1001	x”0004”
6 (ADDC.W)	0	x"001F"	x"FFE0"	0001	0011	x”0000”
6 (ADDC.B)	1	x”2222”	x”8888”	0001	0100	x”00AB”
6 (ADDC.B)	1	x”7E7E”	x”7F7F”	0001	1100	x”00FD”
6 (ADDC.B)	1	x”9081”	x”9082”	0001	1001	x”0004”
6 (ADDC.B)	1	x”001F”	x”FFE0”	0001	0011	x”0000”

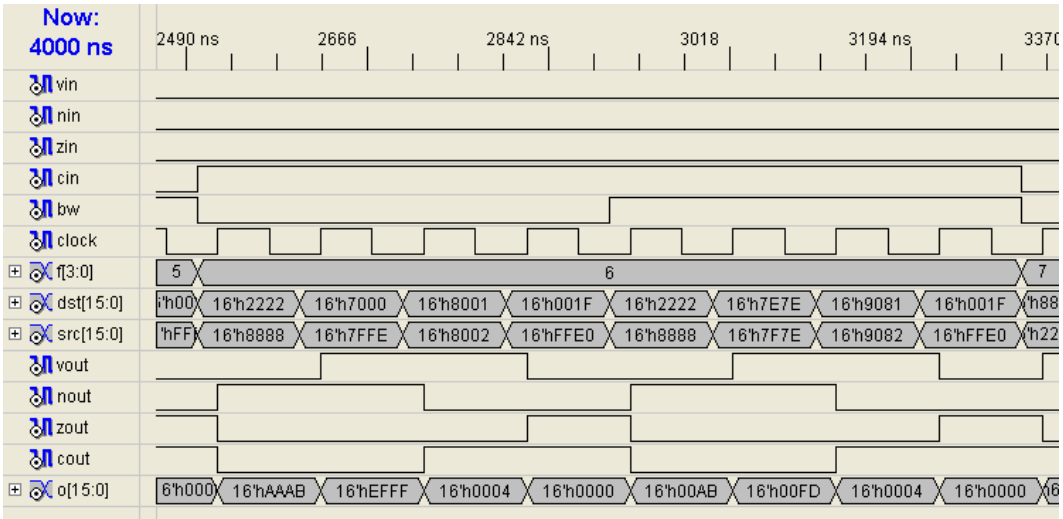

Figure C-7: Behavioral Simulation Waveforms of the ADDC Instruction

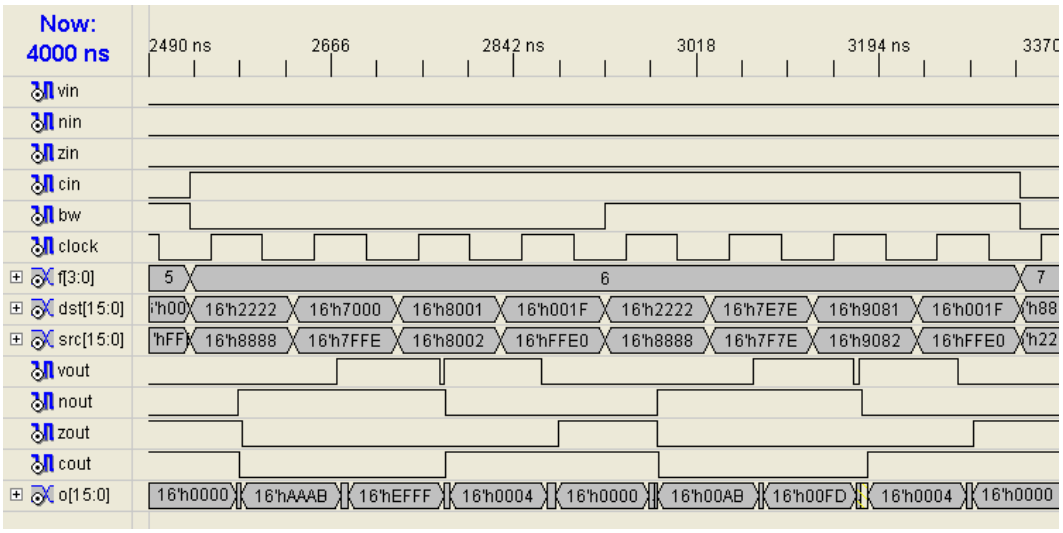

Figure C-8: Post Route Timing Simulation Waveforms of the ADDC Instruction

C.5 SUBC Instruction

Borrow value is equal to inverted value of the carry bit.

Table C-5: Simulation Inputs and Expected Outputs for the SUBC Instruction

Input Values					Expected Outputs	
F	B/W	DST	SRC	VNZC	VNZC	Output
7 (SUBC.W)	0	x”8888”	x”2222”	0000	1001	x”6665”
7 (SUBC.W)	0	x”4444”	x”4444”	0001	0011	x”0000”
7 (SUBC.W)	0	x”7FFF”	x”000F”	0000	0001	x”7FEF”
7 (SUBC.W)	0	x”7FFF”	x”8001”	0001	1100	x”FFFE”
7 (SUBC.B)	1	x”8888”	x”2222”	0000	1001	x”0065”
7 (SUBC.B)	1	x”1144”	x”7744”	0001	0011	x”0000”
7 (SUBC.B)	1	x”7FFF”	x”000F”	0000	0001	x“006F”
7 (SUBC.B)	1	x”7F7F”	x”8081”	0000	1100	x”00FD”

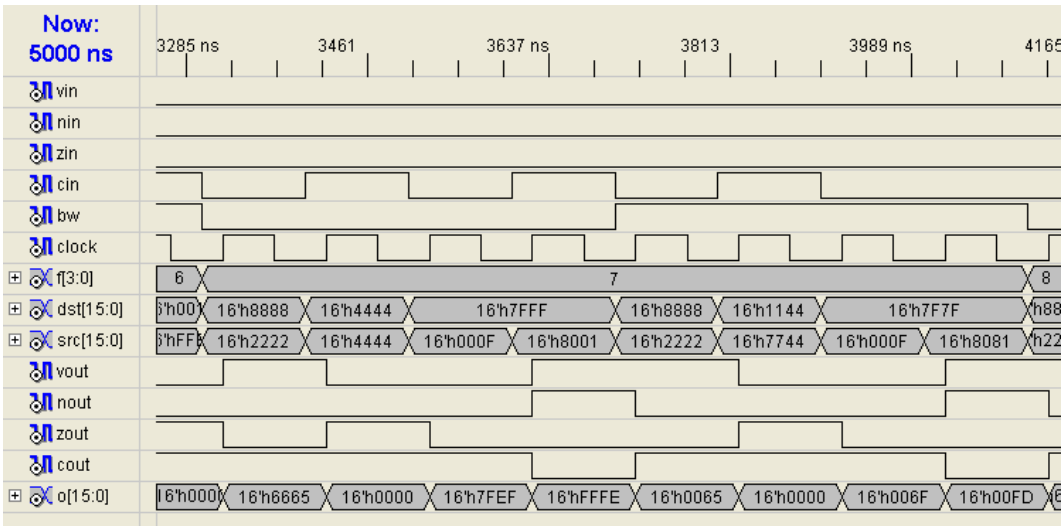

Figure C-9: Behavioral Simulation Waveforms of the SUBC Instruction

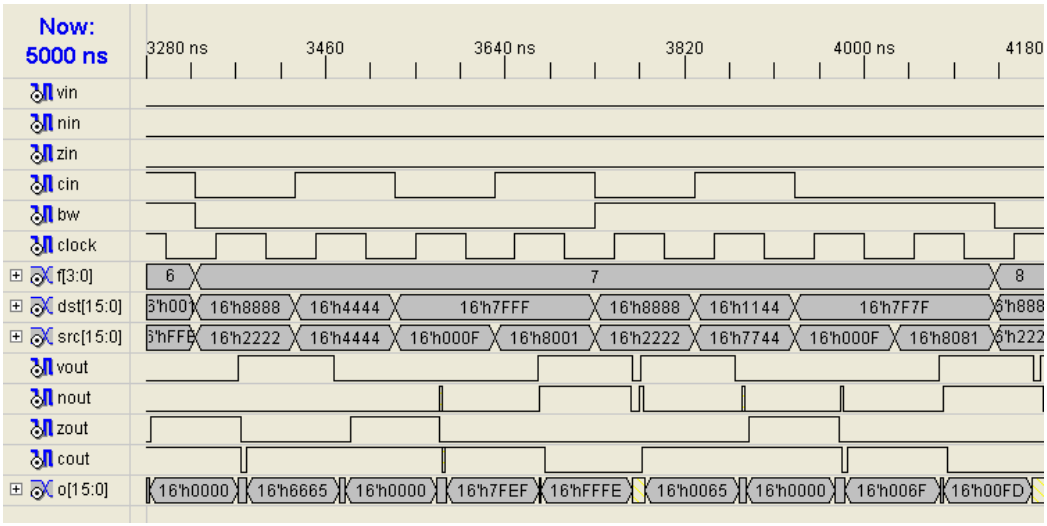

Figure C-10: Post Route Timing Simulation Waveforms of the SUBC Single Operand Instruction

C.6 SUB Instruction

Table C-6: Simulation Inputs and Expected Outputs for the SUB Instruction

Input Values					Expected Outputs	
F	B/W	DST	SRC	VNZC	VNZC	Output
8 (SUB.W)	0	x”8888”	x”2222”	0000	1001	x”6666”
8 (SUB.W)	0	x”4444”	x”4444”	0000	0011	x”0000”
8 (SUB.W)	0	x”7FFF”	x”000F”	0000	0001	x”7FF0”
8 (SUB.W)	0	x”7FFF”	x”8001”	0000	1100	x”FFFE”
8 (SUB.B)	1	x”8888”	x”2222”	0000	1001	x”0066”
8 (SUB.B)	1	x”1144”	x”7744”	0000	0011	x”0000”
8 (SUB.B)	1	x”7F7F”	x”000F”	0000	0001	x“0070”
8 (SUB.B)	1	x”7F7F”	x”8081”	0000	1100	x”00FE”

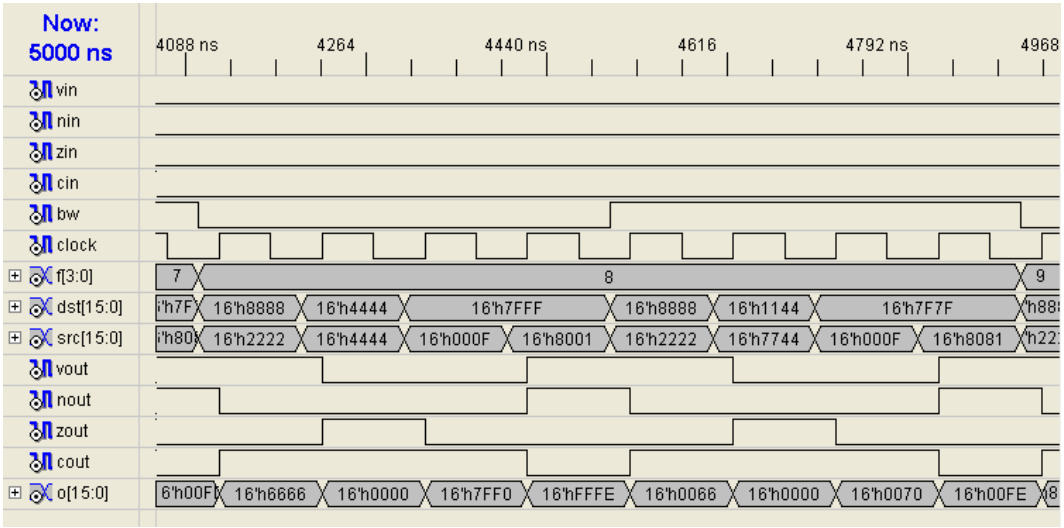

Figure C-11: Behavioral Simulation Waveforms of the SUB Instruction

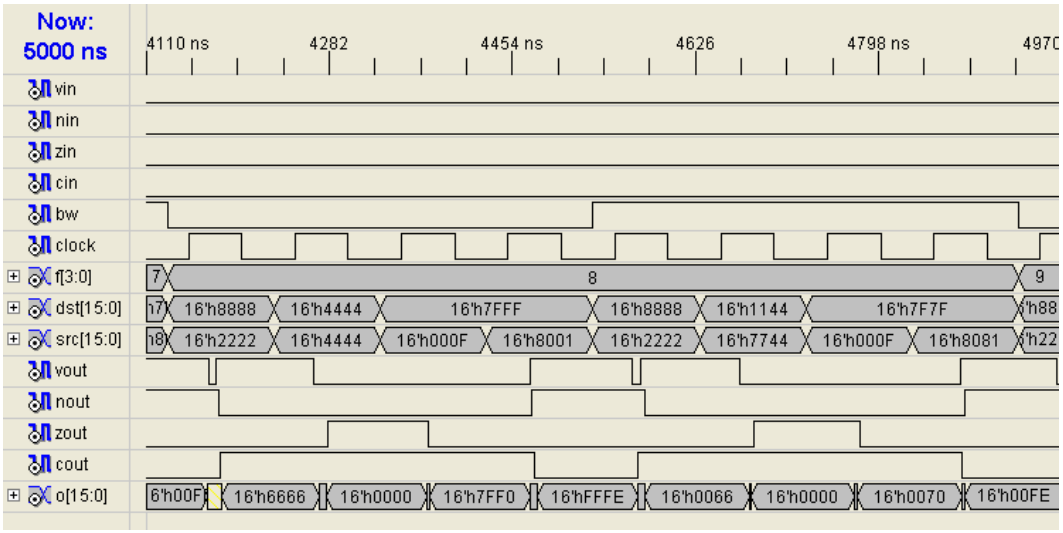

Figure C-12: Post Route Timing Simulation Waveforms of the SUB Instruction

C.7 CMP Instruction

CMP instruction moves the data on the destination (DST) port to the output.

Table C-7: Simulation Inputs and Expected Outputs for the CMP Instruction

Input Values					Expected Outputs	
F	B/W	DST	SRC	VNZC	VNZC	Output
9 (CMP.W)	0	x”8888”	x”2222”	0000	1001	x”8888”
9 (CMP.W)	0	x”4444”	x”4444”	0000	0011	x”4444”
9 (CMP.W)	0	x”7FFF”	x”000F”	0000	0001	x”7FFF”
9 (CMP.W)	0	x”7FFF”	x”8001”	0000	1100	x”7FFF”
9 (CMP.B)	1	x”8888”	x”2222”	0000	1001	x”8888”
9 (CMP.B)	1	x”1144”	x”7744”	0000	0011	x”1144”
9 (CMP.B)	1	x”7F7F”	x”000F”	0000	0001	x”7F7F”
9 (CMP.B)	1	x”7F7F”	x”8081”	0000	1100	x”7F7F”

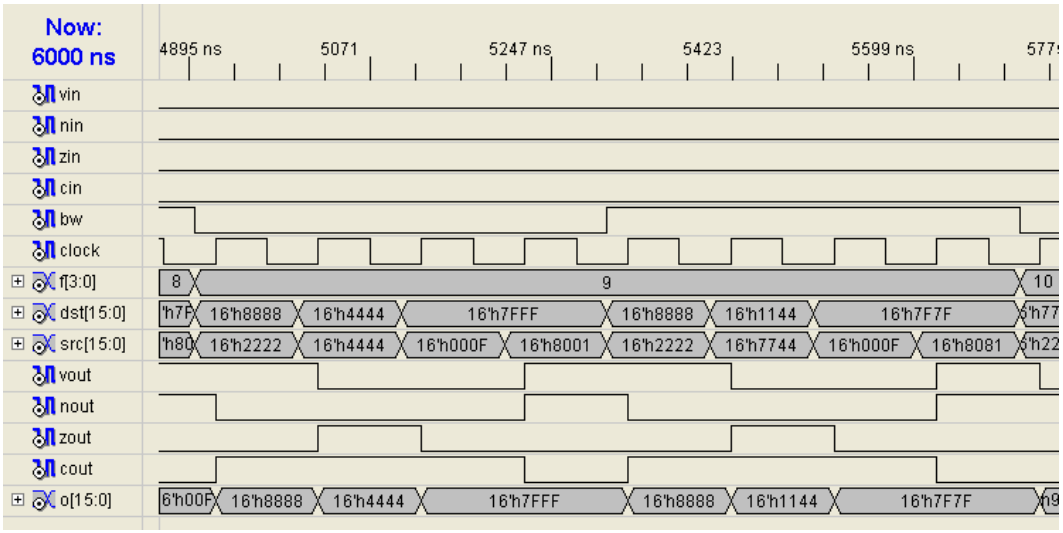

Figure C-13: Behavioral Simulation Waveforms of the CMP Instruction

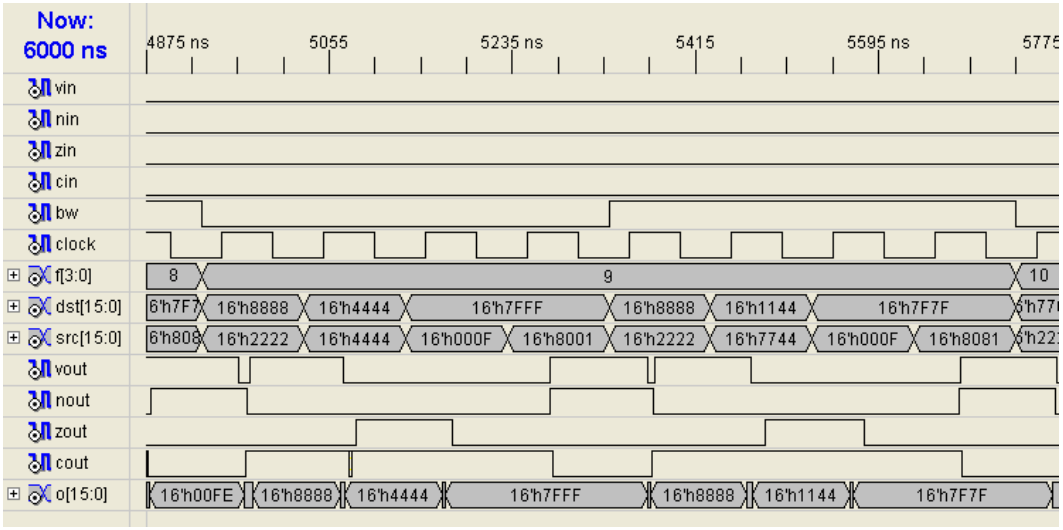

Figure C-14: Post Route Timing Simulation Waveforms of the CMP Instruction

C.8 DADD, BIT, BIC Instructions

Execution of the BIT instruction resets the overflow bit (V).

MOV instruction does not affect status bits.

Table C-8: Simulation Inputs and Expected Outputs for the Instructions DADD, BIT and BIC

Input Values					Expected Outputs	
F	**B/W**	**DST**	**SRC**	**VNZC**	**VNZC**	**Output**
10 (DADD.W)	0	x”7768”	x”2222”	0000	0100	x”9990”
10 (DADD.W)	0	x”9999”	x”0001”	0000	0011	x”0000”
10 (DADD.B)	1	x”7768”	x”2222”	0000	0100	x”0090”
10 (DADD.B)	1	x”9999”	x”0001”	0000	0011	x”0000”
11(BIT.W)	0	x”FAC3”	x”85CC”	0000	0101	x”FAC3”
11(BIT.W)	0	x”5555”	x”AAAA”	0000	0010	x”5555”
11(BIT.B)	1	x”1AC3”	x”15CC”	0000	0101	x”1AC3”
11(BIT.B)	1	x”1155”	x”11AA”	0000	0010	x”1155”
12 (BIC.W)	0	x”ABCD”	x”1234”	0000	1111	x”A9C9”
12 (BIC.B)	1	x”ABCD”	x”1234”	0000	1111	x”00C9”

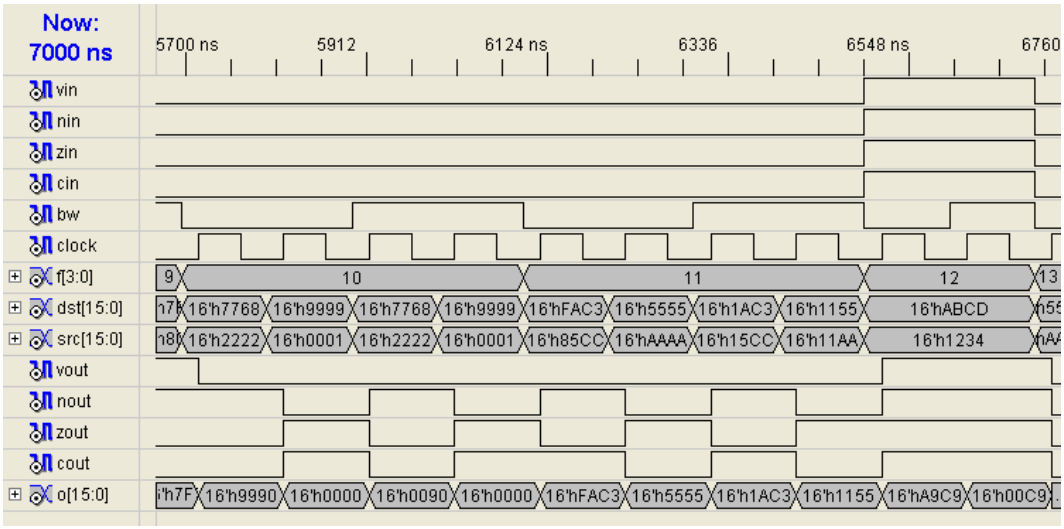

Figure C-15: Behavioral Simulation Waveforms of the DADD, BIT, and BIC Instructions

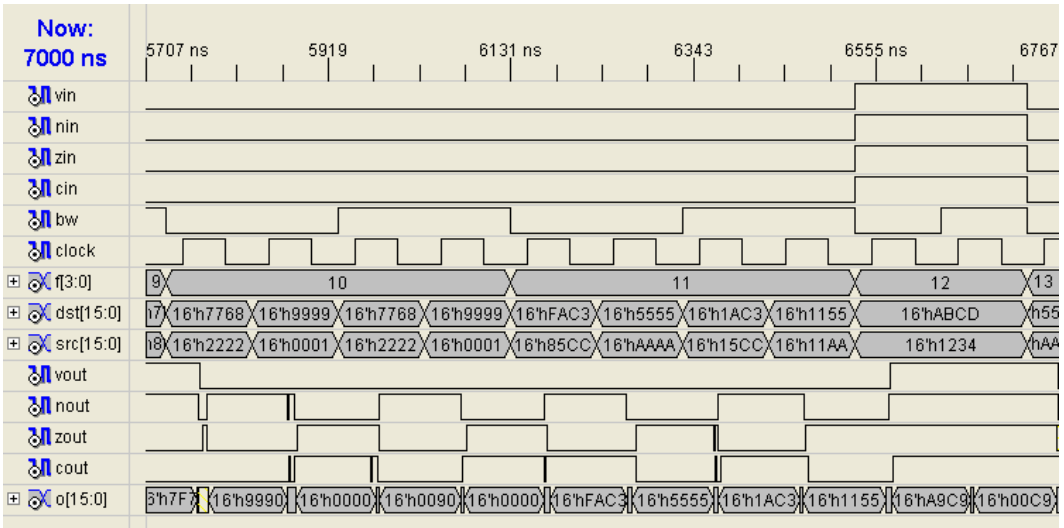

Figure C-16: Post Route Timing Simulation Waveforms of the DADD, BIT, and BIC Instructions

C.9 BIS, XOR, and AND Instructions

BIS instruction does not affect status bits. Besides, execution of the AND instruction resets the overflow bit (V).

Table C-9: Simulation Inputs and Expected Outputs for the Instructions BIS, XOR and AND

Input Values					Expected Outputs	
F	**B/W**	**DST**	**SRC**	**VNZC**	**VNZC**	**Output**
13 (BIS.W)	0	x”5555”	-	0000	0000	x”FFFF”
13 (BIS.B)	1	x”5555”	-	0000	0000	x”00FF”
14(XOR.W)	0	x”A5AA”	-	0000	0101	x”0F00”
14(XOR.W)	0	x”AAAA”	-	0000	1010	x”0000”
14(XOR.B)	1	x”5A5A”	-	0000	0101	x”00F0”
14(XOR.B)	1	x”A5AA”	-	0000	1010	x”0000”
15 (AND.W)	0	x”5555”	x”AAAA”	0000	0010	x”0000”
15 (AND.W)	0	x”9999”	x”AAAA”	0000	0101	x”8888”
15 (AND.B)	1	x”AA55”	x”AAAA”	0000	0010	x”0000”
15 (AND.B)	1	x”9999”	x”AAAA”	0000	0101	x”0088”

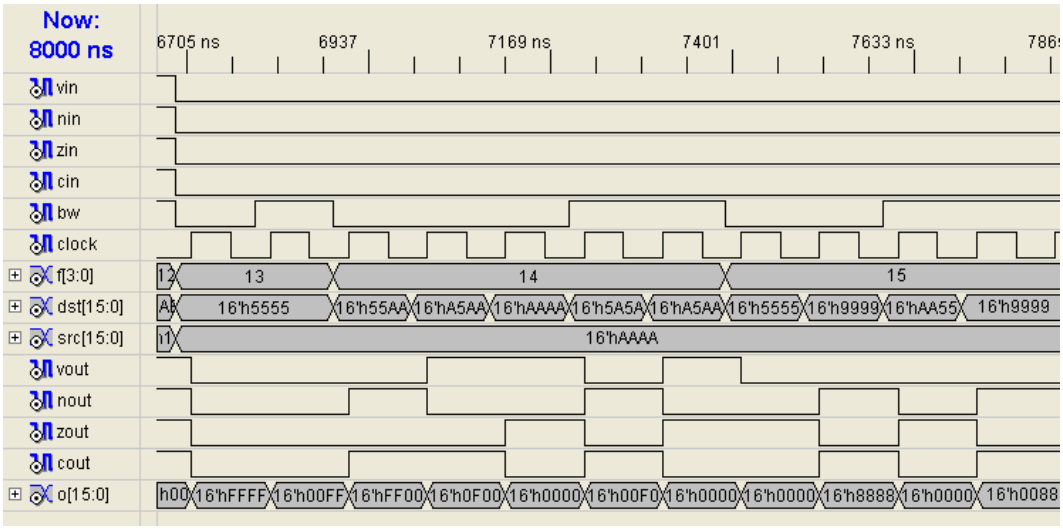

Figure C-17: Behavioral Simulation Waveforms of the BIS, XOR, and AND Instructions

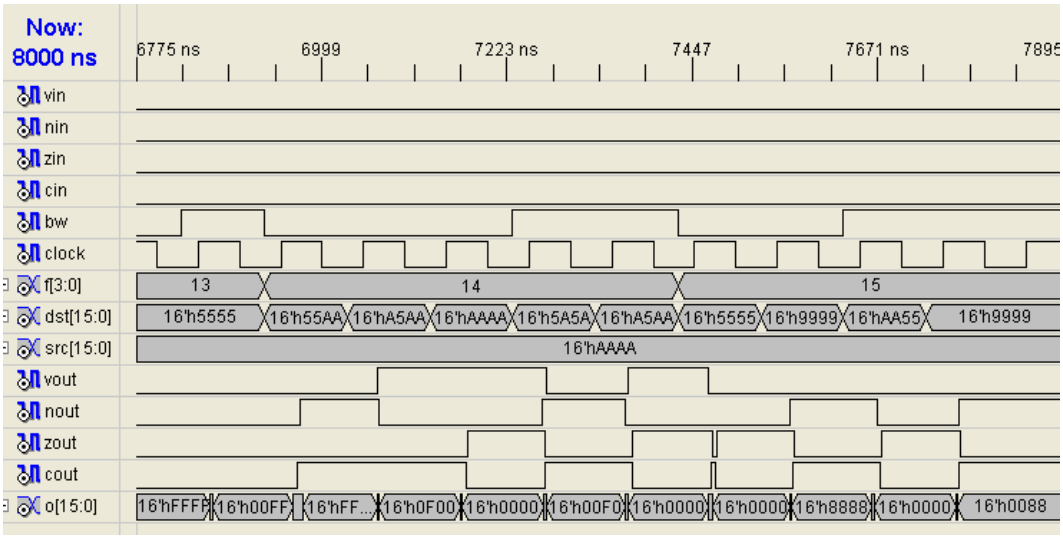

Figure C-18: Post Route Timing Simulation Waveforms of the Instructions BIS, XOR, and AND

APPENDIX D

STRUCTURE OF THE CD-ROM DIRECTORY

The directory structure of the attached CD-ROM is as illustrated below:

Table D-1: Directory Structure of the Attached CD-ROM

SystemC_models/	Directory of MS VC++ project workspace for SystemC processor models		
	cpu1/	SystemC model of the first processor	
	cpu2/	SystemC model of the second first processor	
IAR_works/	Directory of IAR compiler programs		
	AddrMode/	Simulation of Addressing modes	
		AddrMode_DOI/	Simulation of double operand addressing modes
		AddrMode_SOI/	Simulation of single operand addressing modes
	ISA/	Simulation of Instruction set architecture	
		DOI/	Simulation of double operand ISA
		SOI/	Simulation of single operand ISA
		JI/	Simulation of jump ISA
	CRC/	Simulation of CRC algorithms	
		crc16bitwise/	16-bit bitwise CRC
		crc16Rbitwise/	16-bit Reverse bitwise CRC
		crc16table/	16-bit table-based CRC
		crc32bitwise/	32-bit bitwise CRC
		crc32Rbitwise/	32-bit Reverse bitwise CRC
		crc32table/	32-bit table-based CRC
SystemCrafter/	SystemCrafter Project Directory		
Xilinx8.1i/	Xilinx ISE project navigator directory		