Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

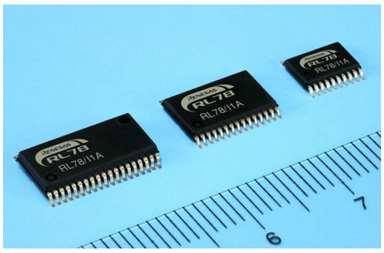
2011

MSc THESIS

MePoEfAr: Memory and Power Efficient Architecture for Embedded Microcontrollers

Imran Ashraf

Abstract



CE-MS-2011-17

Microcontroller based embedded systems have witnessed enormous growth in recent decades. Microcontrollers are the most versatile products found in most of the market segments and in several product families spanning from 4-bit to 64-bit processors. The application domain is such that for some applications only a little functionality is required; for instance, when used as a controller for a simple user interface. In other applications, the functionality demands are high, such as the demand for floating-point calculations and signal processing. Microcontrollers have to meet these demands, while being smaller in size and power efficient. Since memory occupies a large share of area in a microcontroller and contributes the most towards power consumption, the architecture has to be memory efficient. Particularly, for applications using a matrix of processors (as in multi-core architectures), each with its own program memory, the program memory and power efficiencies are a major design goal. The memory efficiency of the instruction set, which also implies power efficiency, is an important factor which needs to be taken into account in the design of microcontroller architectures.
In this thesis, we propose a **Me**mory and **Po**wer **Ef**ficient **Ar**chitecture (**MePoEfAr**) for embedded microcontrollers. MePoEfAr is intended as an improvement of the class of architectures represented by the ATMEL AVR, Texas Instruments MSP430 and the ARM Cortex-M3 microcontrollers. These architectures were designed to be used as embedded controllers. They often have on-board SRAM for data storage and ROM/Flash for program storage. This property demands a memory-efficient architecture, because a small savings of the on-chip program memory area quickly offsets the gates required for extra processor functionality. In addition, due to power aspects, especially for hand-held devices, the clock frequencies used are not very high, so that the instruction decoding time is less critical.
A source level profiler has been developed to get the statistics of various C language constructs for the representative programs used in embedded applications. These statistics were used in making various trade-offs to tune this architecture. An assembler and Interpretive simulator was developed to perform assembler level benchmarking for performance evaluation and comparison with three embedded architectures. Results show the improvement of MePoEfAr performance by 70% and 17% when compared to TI MSP430 and ARM Coretex-M3 microcontrollers, respectively. Furthermore, MePoEfAr outperforms Atmel AVR by a factor of 2.32.
Efficiency of MePoEfAr comes from its more orthogonal architecture, its memory efficient and rich instruction set, efficient support for immediate values and displacements, efficient instruction encoding with variable length instructions of 1 to 4 bytes. Moreover, availability of large number of registers, and the possibility of large number of operations on these registers add to the efficiency of the architecture.

TU Delft Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

MePoEfAr: Memory and Power Efficient Architecture for Embedded Microcontrollers

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Imran Ashraf
born in Mansehra, Pakistan

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

MePoEfAr: Memory and Power Efficient Architecture for Embedded Microcontrollers

by Imran Ashraf

Abstract

Microcontroller based embedded systems have witnessed enormous growth in recent decades. Microcontrollers are the most versatile products found in most of the market segments and in several product families spanning from 4-bit to 64-bit processors. The application domain is such that for some applications only a little functionality is required; for instance, when used as a controller for a simple user interface. In other applications, the functionality demands are high, such as the demand for floating-point calculations and signal processing. Microcontrollers have to meet these demands, while being smaller in size and power efficient. Since memory occupies a large share of area in a microcontroller and contributes the most towards power consumption, the architecture has to be memory efficient. Particularly, for applications using a matrix of processors (as in multi-core architectures), each with its own program memory, the program memory and power efficiencies are a major design goal. The memory efficiency of the instruction set, which also implies power efficiency, is an important factor which needs to be taken into account in the design of microcontroller architectures.

In this thesis, we propose a **Me**mory and **Po**wer **Ef**ficient **Ar**chitecture (**MePoEfAr**) for embedded microcontrollers. MePoEfAr is intended as an improvement of the class of architectures represented by the ATMEL AVR, Texas Instruments MSP430 and the ARM Cortex-M3 microcontrollers. These architectures were designed to be used as embedded controllers. They often have on-board SRAM for data storage and ROM/Flash for program storage. This property demands a memory-efficient architecture, because a small savings of the on-chip program memory area quickly offsets the gates required for extra processor functionality. In addition, due to power aspects, especially for hand-held devices, the clock frequencies used are not very high, so that the instruction decoding time is less critical.

A source level profiler has been developed to get the statistics of various C language constructs for the representative programs used in embedded applications. These statistics were used in making various trade-offs to tune this architecture. An assembler and Interpretive simulator was developed to perform assembler level benchmarking for performance evaluation and comparison with three embedded architectures. Results show the improvement of MePoEfAr performance by 70% and 17% when compared to TI MSP430 and ARM Coretex-M3 microcontrollers, respectively. Furthermore, MePoEfAr outperforms Atmel AVR by a factor of 2.32.

Efficiency of MePoEfAr comes from its more orthogonal architecture, its memory efficient and rich instruction set, efficient support for immediate values and displacements, efficient instruction encoding with variable length instructions of 1 to 4 bytes. Moreover, availability of large number of registers, and the possibility of large number of operations on these registers add to the efficiency of the architecture.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2011-17

Committee Members :

Advisor:	Dr. Said Hamdioui, CE, TU Delft
Advisor:	Ad J. van de Goor, CE, TU Delft
Chairperson:	Dr. ir. Koen L. M. Bertels, CE, TU Delft
Member:	Dr. Ir. G. Kuzmanov, CE, TU Delft
Member:	Dr. Alexandru Iosup, PDS, TU Delft
Member:	Ir. A.C. de Graaf, CE, TU Delft

To the hug of my son Usman
&
To the smile of my niece Simra Khan

Contents

List of Figures

List of Tables

Listings

Acknowledgements

First of all, I would like to express my gratitude to my supervisors, Ad van de Goor and Said Hamdioui for giving me a chance to work under their kind supervision. Special thanks to Ad van de Goor for his valuable guidance and precious time, throughout this work. He has always come down at my level and helped me to understand the architecture related concepts. It is really an honor for me to work with a member of PDP-11 architecture team.

I would also like to thank a number of people in the CE group for their help and support. Thanks to Georgi Kuzmanov, Nadeem and Fakhar for their useful discussions. Roel for allowing me to use his QUIPU profiler and providing me a quick start for its modifications. Anca Molnos for providing me EEMBC benchmarks. Thanks to Max Ferger (from ACE BV) for giving me a chance to attend the CoSy training. Laiq, Faisal, Mottaqiallah for proof reading parts of my thesis and their friendly support throughout my MSc studies.

Among my friends at TU Delft, I would like to thank Di and Wu, for their wonderful company thought my stay here at Delft. I would also like to thank Husnul Amin, Mehfooz, Hamayun and Seyab for their help in finding a wonderful accommodation for me and for their help in setting it up.

Last but not least, I would like to thank my family, especially, my parents for their love and support throughout my good and bad times, and for making me who I am. Sincere thanks to my wife, for her care, patience and encouragement throughout my MSc studies and especially during my thesis work. She helped me a lot by taking good care of home and kid, and sparing me completely for my studies.

Imran Ashraf
Delft, The Netherlands
September 8, 2011

1 Introduction

This chapter provides an introduction to the work presented in this thesis. Section 1.1 highlights some the applications of microcontroller with some statistics from an industry research for the year 2010. Section 1.2 presents the motivation behind this thesis work. Section 1.3 lists the main contributions of this thesis. Section 1.4 outlines the remaining content of this thesis.

1.1 Introduction

One of the important aspects of modern electronic technology is embedded systems based on microcontrollers. According to the Microchip ISA Vision Summit 2011 [17], 10 billion microcontroller units are produced per year for embedded applications as compared to 400 million units per year for general purpose microprocessor based applications. This growth in microcontroller industry is derived by the huge application domains where they can be used. Figure 1.1 provides a brief list of applications which use microcontrollers. Among other applications, consumer application alone have utilized about 3.39 billion microcontrollers in the year 2010, as can be seen from Figure 1.2.

Consumer	Automotive	Office Automation	Telecom	Industrial
High definition TV	CDI	Computer mouse	Cellular telephone	Power Inverter
Stereo receiver	Body Control	Laptop trackball	Cordless telephone	Motor control
DVD player	Infotainment	Computer keyboard	Feature phone	Compressor
Universal remotes	Keyless entry	Handheld scanner	Answering machine	Thermostat
Cable TV converter	Radar detector	Laser printer interface board	Pay phone	Postage meter
Video game systems	Cruise control	Wireless LANs	Pager	Utility meter
Cameras	Anti-lock braking	Printer cartridges	Modem	Robotics
Garage opener	Speedometer	Hi-res scanner	Caller ID	Process control
Carbon Monoxide detector	Climate control	Bar code reader	Line cards	Gas pump
Microwave oven	Security System	Disk drive	Hands-free kits	Smoke detector
Smoke detector	Active suspension	Tape back-up unit	Long distance service router	Credit card reader
Water filters	Fuel pump control	US bus hubs	Power Amp	Access verification and control
Cordless tools	Fuel injection	Facsimile machine		Lighting sensors and control
Vacuum cleaner	Air bag sensor	CD/DVD writer		Ballast control
Electric blanket	Power seats			
	Compass			

Figure 1.1: Various Microcontroller Applications

The vast diversity of the microcontroller applications, demands a variety of microcontroller architectures satisfying the needs of these application domains. Most of these devices are aimed at small size and low power consumption, for instance, hand held devices such as cell phones, digital watches, pagers etc. Figure 1.3 provides the statistics of the annual cellular handset sales. It can be seen from these statistics that about 1.5 billion cellular phone units have been sold in the year 2010.

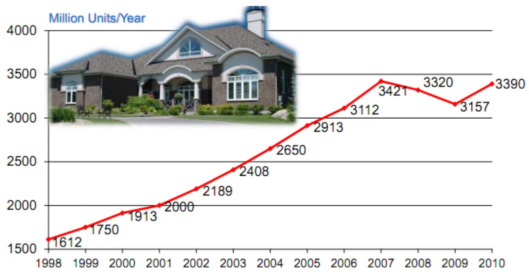


Figure 1.2: Microcontrollers in Consumer Applications [17]

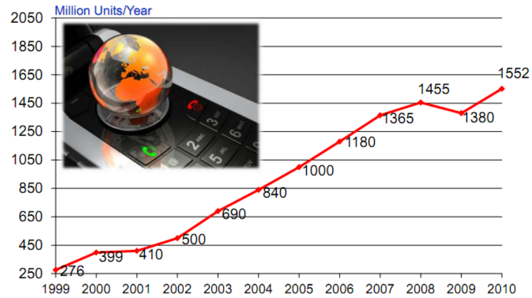


Figure 1.3: Annual Cellular Handset Sales [17]

Memory and power efficiency can be achieved in several ways at different design levels. This thesis discusses the details of an embedded microcontroller in which memory and power efficiency is achieved at the architecture level.

1.2 Motivation

Key points which motivated the design of this memory and power efficient microcontroller architecture are:

- Embedded microcontrollers are found inside another system where their smaller size is important. They often have on-board SRAM for data storage and ROM/Flash for program storage. Memory occupies a substantial area on the chip. This property demands a memory-efficient architecture, because small savings of the on the on-chip program memory area quickly offsets the gates required for extra processor functionality, reducing the size and cost of the microcontroller.
- Power consumption is an important criteria in the design of microcontrollers, particularly for hand held devices running on batteries. In some cases, replacing the batteries is very costly, for instance, in case of underground water meters and heart pace makers. So these devices have to be power efficient.
- Because of power aspects, especially for hand-held devices, the clock frequencies used are not very high, so that the instruction decoding time is less critical. This means that for these devices, design choices can be made in favor of power efficiency as compared to clock speed.
- In applications using a matrix of processors (as in multi-core architectures), each with its own program memory, the program memory and power efficiencies are a major design goal.
- The memory efficiency of the instruction set also implies power efficiency, which was the key motivation behind this architecture.

1.3 Main Thesis Contributions

This thesis makes the following main contributions:

1. Provides the instruction set architecture of a memory and power efficient embedded microcontroller

2. Provides the static profiling statistics to fine tune the architecture for memory and power efficiency
3. Provides the details of the software tool chain including:
   (a) An assembler to translate the assembly programs into machine code
   (b) An interpreter to model the architecture to simulate the execution of machine code
4. Provides the details of performance evaluation of this architecture
5. Provides the static and dynamic results of performance comparison with the three well known embedded microcontrollers

1.4 Outline of Thesis

An outline of this thesis is presented here to give an overview of the whole thesis.

Chapter 2 presents an overview of microcontroller architectures. A classification of microcontroller architecture based on several criteria is presented. Three well know microcontroller architectures are discussed in detail, which we have used for performance comparison.

Chapter 3 discusses the static profiling. The statistics of high level language constructs such as statements, operations, constants are are provided to show their frequency distributions in four *C* language benchmark programs.

Chapter 4 provides the details of MePoEfAr architecture. It starts with overall architecture properties. Issues, like type of architecture, bit and byte numbering, data types, instruction classification and register sets are discussed. Global architecture issues such as layout of the program status word and Memory Map are provided. Various instruction formats in MePoEfAr architecture with examples are provided. Furthermore, operation sets supported by these instruction formats are also discussed with the a description on how these operations affect the condition codes. A brief description of exceptional conditions like traps and interrupt vectors are provided followed by a discussion of extension of Program and Data Memory. The summary of encoding cost and feasibility of MePoEfAr architecture are provided to show the space for future extension in the architecture.

Chapter 5 gives the implementation details of MePoEfAr assembler. It covers the details of the intermediate steps involved to translate the assembly program to machine code. Instruction bit assignments are provided to showing the bit patterns used to represent assembly instructions.

Chapter 6 discusses the MePoEfAr interpreter which simulates the MePoEfAr microcontroller. It discusses the two main parts of MePoEfAr interpreter. First part which loads the machine code to memory and performs some book keeping for debugging information. Second part is the microcontroller model which fetches the instructions from memory, decodes and executes them.

Chapter 7 covers the assembler level benchmarking details, which we performed to evaluate the performance of MePoEfAr architecture. Furthermore, it provides the results of static and dynamic comparison of performance with three well known microcontrollers.

Chapter 8 provides the conclusions and recommendations for future work. This chapter is followed by the bibliography and appendices. The scanner and parser generator codes for MePoEfAr assembler are provided in Appendix A and Appendix B respectively. Appendix C provides the assembly codes of the benchmark programs we have used for performance comparison. Details of these calculations are provided in Appendix D.

2 Overview of Microcontroller Architectures

In this chapter an overview of microcontroller architectures is presented. Microcontroller architectures can be classified based on a number of factors such as the architectural style, memory interfaces, word-size and operand specification. Section 2.1 provides the classification of microcontroller architectures based on these criteria. A brief description of three example architectures is given in Section 2.2. Properties of an ideal microcontroller are discussed in Section 2.3. Finally, Section 2.4 summarizes this chapter.

2.1 Classification of Microcontroller Architectures

Large number of microcontrollers are designed to fulfill the requirements for their diverse application area [19]. These microcontrollers can be classified based on various criteria. Figure 2.1 provides an overview of a classification of these microcontrollers based on architectural aspects. The details of each category in this classification is provided one by one in sub-sections.

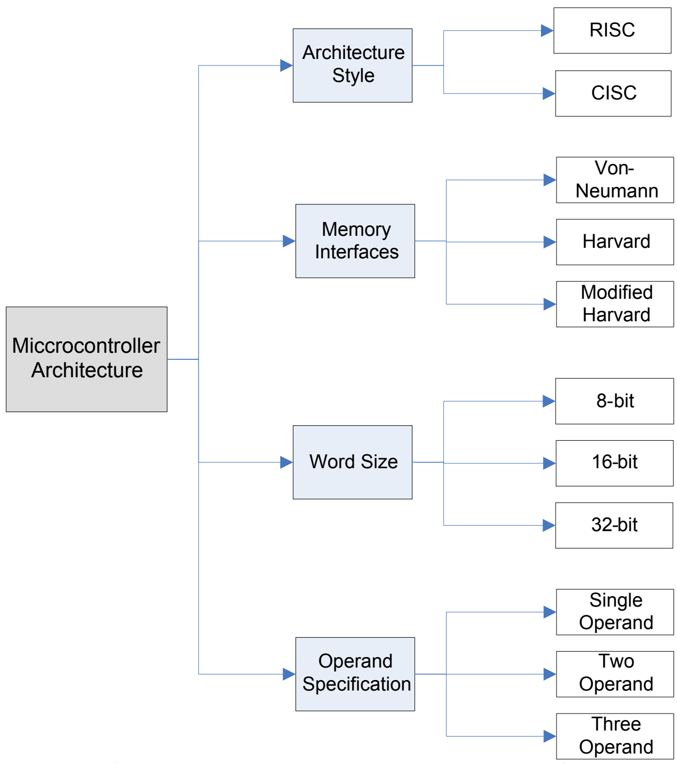


Figure 2.1: A Classification of Microcontroller Architectures

2.1.1 Classification Based on Architectural Style

Based on the architectural style, microcontrollers can be classified into simple and fixed size instructions or complex variable length instructions as described below:

Reduced Instruction Set Computer (RISC) style architectures have simple instructions [31]. Most of the instructions in these architectures execute in a single cycle, as these instructions involve register to register operations. Data fetch from the memory is performed only with Load and Store instructions with simple addressing modes. This is the reason they are also known as Load-Store architectures. From the performance point of view, in the design of RISC architectures trade-offs are made in favor of a lower **C**ycles **P**er **I**nstruction (**CPI**), at the expense of increased code size. The reason for the increased code size is that the complexity of the system is shifted from hardware to software as most of the high level language support is provided in software [30]. So more number of assembly instructions are required to do some HLL operation, resulting in the increased code size. Examples of microcontrollers based on RISC architecture are:

- ARM Cortex-M3 series microcontrollers
- Atmel AVR AT90S851
- PIC microcontrollers by Microchip e.g. PIC16F84
- MSP430 Family by Texas Instruments

Complex Instruction Set Computer (CISC) architecture style is characterized by having a large number of instructions, with most of the instructions requiring a number of cycles for execution. Instructions are variable length instruction. CISC architecture supports register to register, register to memory and memory to memory operand specification in instructions. Normally there is a variety of addressing modes available in these architectures. The advantage of the CISC architecture is that most of the instructions are powerful, allowing the programmer/compiler to use one instruction in place of many simpler instructions, resulting in a reduced code size. Examples of microcontrollers based on CISC architecture are:

- Intel 8051, 8052 and 8096 family
- Motrola 68000 family (designed and marketed by FreeScale Semiconductor)
- M16C/60 and H8SX cores by Renesas Electronics
- TLSC 870 C1, TLCS 900 L1, TLCS 900 H1 core families by Thoshiba

2.1.2 Classification Based on Memory Interfaces

Microcontroller architectures can either have a single memory for instructions and data or physically separate memories to hold program and data. Based on these memory interfaces, architectures are classified as:

Von Neumann architectures store both program and data in the common main memory [33]. This means that either instruction is read from memory or data is read/written from/to this memory. The Von-Neumann architectures main advantage is the simplification of the microcontroller design because of a single memory access. The disadvantage is because of the same bus system, both instruction

cycle and data cycle cannot occur at the same time. This is known as Von Neumann Bottleneck as pointed out by Backus Naur [18]. Examples of microcontroller architectures based on Von Neumann style are:
- Texas Instrument MSP430
- Motorola 68HC11

Harvard architectures are characterized by having two physically separate memories and pathways for program and data. Instructions can be stored in read-only memory and data in read-write memory. This means that attributes of instruction and data memory can be different. For instance, they may have different word width, access timings, implementation technology or memory address structure. Harvard architectures have distinct instruction space and data space. As instruction fetches and data access do not contend for a single memory pathway, a Harvard architecture microcontroller can thus be faster for a given circuit complexity. Examples of Harvard microcontroller architectures are:
- Renesas RX600 Series microcontrollers
- Microchip PIC microcontrollers

Modified Harvard architectures have the characteristic that they have unified instruction and data space. They have separate path ways for instructions and data which is implemented by instruction and data caches. Examples of modified Harvard microcontroller architectures are:
- Atmel AVR AT90S851 microcontroller
- ARM Cortex M3 Series

2.1.3 Classification Based on Word Size

Although there are 4-bit (COP400 by National Semiconductor) and 24-bit (PIC24 by Microchip) microcontroller architectures as well but the most common word sizes are 8-, 16- and 32-bit.

8-bit Architecture performs arithmetic and logical operations on 8-bits. Examples of 8-bit microcontrollers are:
- Intel 8051 family
- Motorola MC68HC11 family
- Atmel AVR AT90S851

16-bit Architecture performs arithmetic and logical operations on 16-bits. Examples of 16-bit microcontrollers are:
- MSP430 Family by Texas Instruments
- S12 and S12X families by Freescale
- Motorola MC68HC12 and MC68332 families

32-bit Architecture performs arithmetic and logical operations on 32-bits. Examples of 32-bit microcontrollers are:
- ARM Cortex-M based family
- Atmel AVR32
- Microchip PIC32 based on MIPSM4K architecture

2.1.4 Classification Based on Operand Specification

Operands in a single instruction vary from a single operand to multiple operands. The work presented in [24] provides taxonomy of architectures based on operands. The most common [1] architectures based on number of operands are:

1-Operand Architectures specify one operand explicitly in the instruction and the other operand is the implicit accumulator operand. This accumulator register is a special register to accumulate the temporary results of computation. In order to perform an operations, instructions are required to move the operands to accumulator and move the result back to where it is required. Intel 8051 architecture is an example of 1-operand architectures. In these architecture, A = B + C is implemented as:

```
load B
add C
store A
```

2-Operand Architectures: have two operands explicitly specified in the instruction. One of the operand serves as both source and destination. The statement A = B + C in these architectures is implemented as:

```
load r1, B
load r2, C
add r1, r2
store r1, A
```

In these examples ri are general purpose registers. Examples of 2-operand microcontroller architectures are:

- Atmel AVR AT90S851
- Texas Instrument MSP 430
- Microcontrollers based on ARM Thumb [2] architecture

3-Operand Architectures: have an explicit mention of one destination and two source operands in the instructions. So A = B + C will be implemented as:

```
load r1, B
load r2, C
add r3, r1, r2
store r3, A
```

Specification of three operands in an instruction requires relatively large encoding space. Most of the 3-operand architectures are 32-bit or higher architectures. Examples of 3-operand architectures are:

- Atmel AVR32 architecture
- Microcontrollers based on ARM Architectures

[1] 0-operand architectures also known as stack-based architectures, have their operands implicitly on stack. Java Virtual Machine is an example of stack based architecture. These architectures are not common for microcontrollers.

[2] Thumb instructions are 16-bit instructions accommodating the specification of only two operands.

2.2 Example Architectures

In this section, we provide the details of the three example architectures based on the classification we have described in this chapter. These example architectures are later used for performance comparison in assembler level benchmarking (Chapter 7). These three microcontroller architectures are:

1. Atmel AVR AT90S851
2. TI MSP430G2231
3. ARM Cortex-M3 LPC1342

Table 2.1 provides an overview of this classification. For the sake of brevity in this table, TI, ARM and AVR refers to TI MSP430G2231, ARM Cortex-M3 LPC1342 and AVR AT90S851 microcontrollers respectively.

Table 2.1: Classification of Three Microcontroller Architectures Based on the Categories Described in This Chapter

	Classification Criteria			
Name	**Architectural Style**	**Word Size**	**Memory Interface**	**Operand Specification**
AVR	RISC	8	Modif. Harvard	2-operand
TI	RISC	16	von Neumann	2-operand
ARM	RISC	32	Modif. Harvard	3-operand

2.2.1 Atmel AVR AT90S851

AT90S8515 is a low power, CMOS, 8-bit microcontroller based on the AVR RISC architecture [15] developed by Atmel [14]. It utilizes modified Harvard architecture concept. Although it is an 8-bit microcontroller, each instruction takes one or two 16-bit words. It has 32 single-byte general purpose registers with single clock cycle access time. It supports five addressing modes.

2.2.2 TI MSP430G2231

Second candidate is MSP430G2231 [3], a 16-bit RISC architecture developed by Texas Instruments [2]. It has been designed for low cost and low power embedded application. It uses von-Neumann architecture with a single instructions and data memory space. Instructions generally take one cycle per word fetched or stored. It has 27 core instruction and 24 emulated instructions. It supports seven addressing modes for source operands and four addressing modes for the specification of destination operands in instructions. It has the following 16-bit registers:

R0: Program counter
R1: Stack pointer
R2: Status register (only in register addressing with word data type)
R2 and R3: are used as constant generators for the most frequent constants (0,1,2,4,8)

R4-R15: General purpose registers

The user guide found here [4] provide further details of MSP430 microcontroller architecture.

2.2.3 ARM LPC1342 Cortex-M3

Third candidate is the LPC1342 [13] developed by NXP (founded by Phillips) [12]; a Cortex-M3 based low power 32-bit RISC m1icrocontroller. ARM is a fab-less company which designs these architectures as **I**ntellectual **P**roperty (**IP**) modules and sells licenses to other companies which actually manufacture the chips, in the case of LPC1342, the manufacturing company is NXP. There are various architectures provided by ARM targeting various application areas, such as:

- ARM Cortex-A series targets the general purpose processor cores
- ARM Cortex-R series is a family of processors for real time systems
- ARM Cortex-M series processors are designed for low-power, memory efficient embedded applications

Among this M-series processors, Cortex-M3 processors are especially designed for embedded microcontrollers. It is based on modified Harvard architecture concept. It supports Thumb-2 instruction set to reduce the instruction memory requirements by including the support for 16-bit instructions. It has following general purpose and special purpose registers:

R0-R12: General purpose registers
R13: Stack pointer
R14: Link registers used by subroutines for return address
R15: Program counter
xPSR Program Status Register

Registers R0-R7 are accessible by all instructions, whereas, registers R8-R12 are only accessible by 32-bit instructions and 16-bit instructions cannot access them. The technical reference manual of ARM Cortex-M3 architecture (as well as other ARM architectures) can be found here [5] for further details.

2.3 Ideal Properties of a Microcontroller Architecture

Ideal properties of a microcontroller architecture refer to the properties which are not realizable in a single architecture. These properties are inter-related such that making a design trade-off to improve certain property may adversely affect the other property(ies). For instance, making a choice in favor of simple fixed width instructions favors higher clock speed at which these designs can be run. The down side of this choice is the increased program footprint. Ideal properties of microcontrollers are briefly discussed below.

2.3.1 Program Memory Size

Microcontrollers are normally embedded inside other systems. Size of microcontroller is important so that it can fit in the system. Program memory occupies a major share in the chip area. So, ideally, program memory size should be negligible in microcontrollers. In other words, architecture should be memory efficient such that program size for a given application should ideally be negligible.

2.3.2 Power Consumption

Power consumption is an important criteria in the design of microcontrollers, particularly for hand held devices running on batteries. In some cases, replacing the batteries is very costly, for instance, in the case of underground water meters and heart pace makers. Ideal microcontrollers should consume negligible amount of power.

2.3.3 Speed

Due to the diverse application areas where microcontroller can be used, the demand on processing speed is also diverse. There are application which require high processing speed, such as streaming applications. Ideally, the processing speed should be very high.

2.3.4 Modularity

Ideally, microcontroller architecture should be highly modular, such that any type of modification in one aspect should not bring change to the rest of the architecture. The modularity of an architecture helps in development and testing of the individual subsystems, which results in reduced time to market. During the life of the architecture, modularity assists in evolution of the architecture, resulting in variants of the architecture satisfying certain application requirements. This modularity can further be classified as:

Modularity w.r.t. instruction and data address range: Architecture should be modular such that at any stage in the life of microcontroller, it is possible to extend the instruction address space without impacting data memory address space.

Modularity w.r.t. data types and no of registers in different data types: In this respect, microcontroller architecture should be such that a variety of data types should be supported without modifying the architecture. Furthermore, It should be possible to change the number of registers in a particular data type depending upon the nature of an application.

2.4 Summary

Demand of microcontroller based embedded systems is increasing every year. This is the result of a large number of applications using microcontroller as embedded processing units. The diversity of applications has resulted in a large variety of microcontroller architectures. In this chapter we provided an overview of microcontroller architectures.

Microcontroller architectures are based on RISC or CISC philosophy depending upon the choice to be high processing requirement or smaller code size. These architectures are 4-bit to 64-bit architectures, while 8, 16 and 32 to be the most common word size found now-a-days. Architectures are found to be having single storage and single address space for program and data favoring the Von-Neumann style. Harvard architectures, having distinct program and data memory, or modified Harvard architectures, having single address space but separate buses for instructions and data are commonly used in microcontrollers. Very few architectures are single operand (accumulator based) architectures. 2-operand microcontroller architectures are commonly used by 16-bit architectures. Because of the high encoding space requirement, 3-operand architectures are mostly 32-bit architectures. This classification is further summarized for three microcontroller architectures which we have used in benchmark for performance comparison.

Ideally, microcontroller architecture should be such that program memory size should be negligible, processing speed should be very high at the cost of negligible power consumption. Ideal microcontroller architecture should be modular such that variants can easily be produced and evolution of architecture should be possible without modifying the rest of the architecture.

Before diving into the details of MePoEfAr microcontroller architecture, statistics of high level language constructs are presented in the next chapter.

Statistics of C Language 3

In the previous chapter, an overview of microcontroller architecture was discussed. Before diving into the details of our architecture in the next chapter, frequency distributions of various *C* language constructs are presented in this chapter. An important rule for the design of a microcontroller architecture is to efficiently implement the most frequent cases. In order to know the frequency of different constructs in the language, four applications namely Coremark and AutoBench (EEMBC benchmarks), assembler and interpreter of our architecture have been profiled. The results of different types of statements, operations and operands are tabulated. These results are then utilized in the design of the architecture presented in next chapter.

This chapter opens with the section on list of language constructs to give an overview of what we are going to analyze in this chapter. Section 3.2 briefly discusses the profiling, developed profiler and application programs used for profiling. Section 3.3 provides the profiling results with analysis. Section 3.4 concludes the chapter.

3.1 List of Language Constructs

This section provides a list of *C* language constructs for which the frequency distributions are presented. The results are divided in four groups; namely, statements, operations, operands and miscellaneous measurements. The detailed list of these constructs is given below:

1. Statements
   (a) Assignments
      i. Assignment Types based on LHS
         A. Variable
         B. Array Element
         C. Structure/Union Field
      ii. Assignment Types based on complexity of RHS expression
         A. A = Constant
         B. A = A op Constant
         C. A = B
         D. A = B op Constant
         E. A = A op B
         F. A = B op C
         G. Others (with complex RHS)
   (b) *if* statements
      i. If-only statements
      ii. If-else statements

- (c) *switch* statements
- (d) *break* statements
- (e) *continue* statements
- (f) *goto* statements
- (g) Loops
- (h) Function calls
- (i) *return* statements

2. Operations
   - (a) Arithmetic operations
     - i. $+$
     - ii. $-$
     - iii. $*$
     - iv. $/$
     - v. $\%$
   - (b) Address Arithmetic operations
     - i. $+$
     - ii. $-$
   - (c) Relational operations
     - i. $==$
     - ii. $!=$
     - iii. $<$
     - iv. $>$
     - v. $<=$
     - vi. $>=$
   - (d) Bitwise operations
     - i. and
     - ii. or
     - iii. xor
     - iv. not
   - (e) Shift operations
     - i. Shift left
     - ii. Shift right
     - iii. Arithmetic Shift right
   - (f) Complement operations
   - (g) Absolute operations
   - (h) Type conversions
     - i. 8 to 16
     - ii. 8 to 32
     - iii. 8 to 64
     - iv. 16 to 8
     - v. 16 to 32
     - vi. 16 to 64
     - vii. 32 to 8
     - viii. 32 to 16
     - ix. 32 to 64

x. 64 to 8
xi. 64 to 16
xii. 64 to 32
xiii. integer to address
xiv. address to integer
xv. integer to real
xvi. real to integer
xvii. others

3. Operands
   (a) Constants
      i. -1, 0, 1, 2, ..., 14, 15
      ii. 16-31
      iii. 32-63
      iv. 64-127
      v. 128-255
      vi. 256-65535
      vii. others
   (b) Variable accesses
      i. 8-bit variable access
      ii. 16-bit variable access
      iii. 32-bit variable access
      iv. 64-bit variable access
   (c) Array accesses
   (d) Structure/union Field accesses
   (e) Function calls
4. Miscellaneous
   (a) Average number of function parameters
   (b) Average number of function locals
   (c) Average number of globals used in a function
   (d) Frequency Distribution of Parameters Based on Data Types
   (e) Frequency Distribution of Locals Based on Data Types

3.2 Profiling

Profiling is the program analysis carried out for a number of purposes, for instance, to measure different metrics. Operation frequencies, operand frequencies, function calls are a few examples of such metrics. This analysis can be static or dynamic. Static analysis is performed on the application without actually running it. On the other hand, dynamic profiler analyzes the program during execution. From program memory point of view, results of static profiling are important, which we have provided in this chapter.

3.2.1 Profiler

Profilers are the software tools which are used to automate profiling; in other words, to create the profile of the application program. We have modified the *Quipu* [20] static profiler to obtain all the results as listed in Section 3.1. *Quipu* is a part of Q^2 profiling framework which is developed in the context of **D**elft **W**ork**B**ench (DWB) [21]. This tool is developed as an engine in the CoSy compiler system [6] developed by ACE Associated Compiler Experts.

3.2.2 Profiler Benchmark Applications

An important step in statistical analysis of various language constructs is the selection of the applications to be profiled. We have profiled following four applications:

1. Coremark
2. AutoBench
3. Assembler
4. Interpreter

Table 3.1 provides information about the number of lines of code and number of functions in selected four applications. Blank lines and comments are also counted towards lines of code in these numbers.

Table 3.1: Application Programs Used for Profiling

S.No.	Application	Lines of Code	No. of Functions
1	Coremark	892	27
2	AutoBench	1986	26
3	Assembler	8194	104
4	Interpreter	5597	214

A brief description of these applications is given below:

Coremark: Coremark [7] is an **E**mbedded **M**icroprocessor **B**enchmark **C**onsortium (**EEMBC**) benchmarks [10]. Unlike synthetic benchmarks, EEMBC benchmarks are real application programs. Coremark is freely available from EEMBC website and is used for a quick comparison of embedded processor and microcontroller core functionality. Coremark suit contains three applications as listed below:

1. *core_matrix* performs common matrix operations like additions, multiplications on integer and floating point data.
2. *core_state* determines if an input stream contains valid numbers.
3. *core_list* performs list processing as searching and sorting the linked list.

AutoBench: AutoBench [9] is another EEMBC benchmark suite. AutoBench is a suite of benchmarks that allow users to predict the performance of microprocessors and microcontrollers in automotive, industrial, and general-purpose applications. It is not a free benchmark, but Computer Engineering Lab has the license to use it. It involves matrix operations, bit manipulation, arithmetic operations, table look up and singal processing like filtering.

Assembler: The assembler application is the assembler developed for our architecture. It has the lexical analysis code generated by Flex (a general purpose lexical analyzer generator) [1], parser code generated by Bison (parser generator) [11], code for tree traversals for analysis, symbol table generation. At the end, machine code is generated for our architecture which involves bitwise operations. Further details are provided in Chapter 5.

Interpreter: This application is the interpreter developed for our architecture. It reads the machine code in, decodes the instructions and executes it to produce the results based on the semantics of the instruction. Further details of this interpreter can be seen in Chapter 6.

3.3 Frequency Distribution of *C* Language Constructs

Frequency distributions of different *C* language constructs presented in the list in Section 3.1 obtained by our profiler for selected applications are presented in this section.

3.3.1 Frequency Distribution of Statements

Frequency distribution of various *C* statements is given in Table 3.2 for the selected four application programs. It can be seen from this table that assignment statements constitute the bulk of statements with a frequency of 58.96%. The second most frequent statement is the *if* statement with an average of 19.73%. Similarly, statistics for other statements are also tabulated. Frequency of *break* statement is about 9% which majorly comes from the cases in *switch* statements, especially in *assembler* and *interpreter*.

Table 3.2: Frequency Distribution of Statements

Statement	**Percentage**				
	Coremark	**AutoBench**	**Assembler**	**Interpreter**	**Average**
Assignments	62.69	67.96	53.99	51.18	58.96
if else	18.98	16.87	27.13	15.94	19.73
switch	0.64	0	1.4	3.66	1.43
goto	0	0	1.21	0	0.3
Loops	8.1	9.29	2.18	2.37	5.49
Function Calls	1.92	1.7	1.04	1.07	1.43
return	3.84	3.56	2.58	4.15	3.53
break	3.84	0.62	10.37	21.62	9.11
continue	0	0	0.1	0	0.03
Total	**100**	**100**	**100**	**100**	**100**

As assignment statements have the highest frequency of occurrence among the statements, so let us see the details of assignment statements. Assignment statements can have a simple variable, an array element or a structure/union field on **L**eft **H**and **S**ide (**LHS**). Frequency distribution of assignment statements based on LHS expression is given in Table 3.3. As can be seen from the results that assignments with a variable on left hand side are the most frequent with an average frequency of about 73%.

Table 3.3: Frequency Distribution of Assignment Statements Based on LHS

Assignment Type	**Percentage**				
	Coremark	**AutoBench**	**Assembler**	**Interpreter**	**Average**
variable assignments	83.46	56.35	56.62	96.25	73.17
array assignments	1.5	17.55	1.73	2.71	5.87
struct/union assignments	15.04	26.1	41.65	1.05	20.96
Total	**100**	**100**	**100**	**100**	**100**

Assignments statements can also be classified based on the complexity of expression on **R**ight **H**and **S**ide(**RHS**). Frequency distribution of *C* assignment statements based on complexity of the expression on RHS is given in Table 3.4. Results show that most of the assignment statements have simple RHS expression, that is a constant or a simple variable. These operations correspond to moves. On average, 33% of the assignments have a constant on RHS. Expressions with a variable on RHS make up about 22%.

Table 3.4: Distribution of Assignments Based on Complexity of RHS Expression

Assignment Type	**Percentage**				
	Coremark	**AutoBench**	**Assembler**	**Interpreter**	**Average**
A = Const	28.46	28.61	45.62	29.42	33.03
A = B	31.09	20.57	27.01	8	21.67
A = A op Const	13.11	11.35	8.33	14.93	11.93
A = B op Const	0.37	0.24	1.23	1.33	0.79
A = A op B	0.37	0	0.22	2.04	0.66
A = B op C	0.37	0	0.58	3.47	1.11
others (different complexity)	26.22	39.24	17.02	40.8	30.82
Total	**100**	**100**	**100**	**100**	**100**

The six simple cases listed in the table constitute 70% on average. The *other* expressions with different complexity make up rest of 30%. The RHS expressions in these cases have more than 2 operands on RHS. These operands can be constants, variables, array accesses, structure or union field or return value from a function, involved in various operations.

3.3.2 Operations

In order to know the importance of different operations, frequency distribution of different operations in selected programs is given in Table 3.5. This table summarizes the frequency distribution of all operations for integer and floating point numbers. Statistics from this table show that arithmetic operations are the most frequent operations, wherein, addition and multiplication have a frequency of 24% and 14% respectively.

Address arithmetic refers to arithmetic operations carried out to compute the addresses of data, which corresponds to *C* pointer arithmetic. These operations have a frequency of about 10% in total, where most of the operations are additions.

Relational operations on the average, make up about 22% from the whole operation space. Among relational operations, equality (==) , inequality (! =) and less than (<) operations are frequent operations. Equality and inequality operations are frequent because they are used as test conditions in selection statements. Less than (<) comparison is mostly used in loop statements, where a loop counter is initialized and incremented till this counter is less than certain count value. In bitwise operations, and (&) operation has highest frequency of about (3.25%), which is used in bit masking.

Data type conversion takes place when the operations involve operands of different data types. For instance, in an operation involving integer and floating point data, type conversion takes place. This type conversion can be explicit (type casting) or implicit (operations involving different data types) in *C* language. Conversion operations have an average frequency of 14.8%. Detailed frequency distribution for different conversion

Table 3.5: Frequency Distribution of Operations

Operation Type		**Percentage**									
		Coremark		**AutoBench**		**Assembler**		**Interpreter**		**Average**	
Arithmetic	+	15.82	30.89	16.37	40.09	5.75	18.07	10.35	21.78	12.07	27.71
	-	1.88		3.74		2.82		3.72		3.04	
	*	12.62		18.31		8.98		3.54		10.86	
	/	0.38		1.53		0.42		2.54		1.22	
	%	0.19		0.14		0.1		1.63		0.52	
Address Arithmetic	+	13.56	13.56	9.85	9.85	14.76	15	0.09	0.09	9.57	9.63
	-	0		0		0.24		0		0.06	
Relational	==	4.14	27.31	4.85	27.18	30.61	53.13	20.62	53.05	15.06	40.17
	! =	8.29		4.99		11.11		13.26		9.41	
	<	10.55		13.18		5.29		13.17		10.55	
	>	2.07		3.19		1.32		3.45		2.51	
	<=	0.94		0.14		1.88		0.73		0.92	
	>=	1.32		0.83		2.92		1.82		1.72	
Shift	<<	0.75	3.01	0	4.72	0.03	0.1	2.27	4.36	0.76	3.05
	>>	0		0		0		0		0	
	Arith >>	2.26		4.72		0.07		2.09		2.29	
Bitwise	and	5.46	7.9	0.14	0.14	1.04	1.04	6.36	9.08	3.25	4.54
	or	1.69		0		0		2.18		0.97	
	not	0		0		0		0.18		0.05	
	xor	0.75		0		0		0.36		0.28	
Complement		0.19	0.19	0	0	0.24	0.24	0	0	0.11	0.11
Absolute		0	0	0	0	0	0	0	0	0	0
Type Conversion	8 to 16	0.38	17.14	0.55	18.03	0.28	12.39	0.54	11.63	0.44	14.8
	8 to 32	0.19		1.39		3.31		1.73		1.66	
	8 to 64	0		0		0		0		0	
	16 to 8	0.19		0		0		0.82		0.25	
	16 to 32	2.82		3.61		0.52		2.18		2.28	
	16 to 64	0		0		0		0		0	
	32 to 8	0		0		0.24		5.45		1.42	
	32 to 16	13.18		10.54		7.24		0.82		7.95	
	32 to 64	0		0		0		0		0	
	64 to 8	0		0		0		0		0	
	64 to 16	0		0		0		0		0	
	64 to 32	0		0		0		0		0	
	int to addr	0.38		1.94		0.8		0.09		0.8	
	add to int	0		0		0		0		0	
	int to float	0		0		0		0		0	
	float to int	0		0		0		0		0	
Total		100	**100**	100	**100**	100	**100**	100	**100**	100	**100**

operations are also provided. Integer to address conversions takes place when an integer is operated with a pointer (pointing to some data). It can be seen from the statistics that integer to address conversion occurs frequently, whereas address to integer conversion never occurred. This is because of that fact that computed addresses are saved in pointers and not transferred to integer variables.

Among integer data type conversion operations, 32- to 16-bit and 16- to 32-bit conversions are the most frequent. An operand is promoted to higher size when it is operated with an operand of higher size, for instance 16-bit variable will be converted to 32-bit representation when it will be added to 32-bit data. Conversion of data from higher size to lower size takes place when it is explicit in the language or when a statement involves assignment of data of larger size than the destination. As an example, addition of two 32-bit variables will result in 32-bit result, but when this 32-bit result is assigned to a 16-bit variable, 32-bit to 16-bit conversion will take place.

Although 8-bit variables are accessed but these are not frequently used in operations involving 16 and 32 bit operands. So, type promotion does not take place so frequently. 16- to 32-bit conversion is frequent because, these two data types are frequently used in operations with each other. 32- to 16-bit conversion takes place, because 16-bit operands are frequent (as can be seen from Table 3.13). So the assignments having 16-bit variables cause these conversions.

Operations operate on data and the data can be of integer or floating point type. Table 3.6 provides the statistics of integer type of operations. Overall, 58% operations on average are integer data type operations. On the other hand, frequency of floating point operations is about 1%.

Table 3.6: Frequency Distribution of Integer Operations

Operation Type		**Percentage**									
		Coremark		**AutoBench**		**Assembler**		**Interpreter**		**Average**	
Arithmetic	+	14.69	29	16.37	39.12	5.75	18.04	10.26	21.15	11.77	26.83
	-	1.88		3.74		2.82		3.54		3	
	*	11.86		17.48		8.95		3.36		10.41	
	/	0.38		1.39		0.42		2.36		1.14	
	%	0.19		0.14		0.1		1.63		0.52	
Relational	==	3.2	16.94	2.64	14.43	17.06	28.58	12.08	34.06	8.75	23.5
	!=	5.65		2.64		5.71		10.54		6.14	
	<	5.46		6.8		2.65		7.72		5.66	
	>	1.13		1.66		0.66		2.36		1.45	
	<=	0.56		0.14		1.04		0.45		0.55	
	>=	0.94		0.55		1.46		0.91		0.97	
Shift	<<	0.75	3.01	0	4.72	0.03	0.1	2.27	4.36	0.76	3.05
	>>	0		0		0		0		0	
	Arith >>	2.26		4.72		0.07		2.09		2.29	
Bitwise	and	5.46	7.9	0.14	0.14	1.04	1.04	6.36	9.08	3.25	4.54
	or	1.69		0		0		2.18		0.97	
	not	0		0		0		0.18		0.05	
	xor	0.75		0		0		0.36		0.28	
Complement		0	0	0	0	0.24	0.24	0	0	0.06	0.06
Absolute		0	0	0	0	0	0	0	0	0	0
Total		56.85	**56.85**	58.41	**58.41**	48	**48**	68.7	**68.7**	58	**58**

Table 3.7 provides the statistics of floating point operations. It can be seen that most of the floating point operations are the arithmetic operations. Relational operations never involved floating point data, whereas, complement operations still had an occurrence.

Statistics from the previous tables show that most of the operations (58%) involve integer operands. Ineger operations are applied on different sizes of integers. In order to have support for integers of different sizes or to make some trade-offs in design of architecture , it is important to see the frequency distribution of integer type operations based on size.

Frequency distribution of operations applied to 8-bit data type is given in Table 3.8. It

Table 3.7: Frequency Distribution of Floating Point Operations

Operation Type		Percentage									
		Coremark		AutoBench		Assembler		Interpreter		Average	
	+	1.13		0		0		0.09		0.31	
	-	0		0		0		0.18		0.05	
Arithmetic	*	0.75	1.88	0.83	0.97	0.03	0.03	0.18	0.63	0.45	0.88
	/	0		0.14		0		0.18		0.08	
	%	0		0		0		0		0	
	==	0		0		0		0		0	
	! =	0		0		0		0		0	
Relational	<	0	0	0	0	0	0	0	0	0	0
	>	0		0		0		0		0	
	<=	0		0		0		0		0	
	>=	0		0		0		0		0	
Complement		0.19	0.19	0	0	0	0	0	0	0.05	0.05
Absolute		0	0	0	0	0	0	0	0	0	0
Total		2.07	**2.07**	0.97	**0.97**	0.03	**0.03**	0.63	**0.63**	0.94	**0.93**

can be seen that, about 10% of the operations are the operations on 8-bit data. Most of the 8-bit operations are relational operations making up 5% on average. This is because, 8-bit data is the *char* data type in *C*, which is used for byte level processing. For instance, in EEMBC *core_state* program, there are comparisons, if the character is a decimal point (**.**), an *e* or *E* for floating point exponantial representation etc. Furthermore, this is the reason that most of the relational operations are equality and inequality comparisons.

Table 3.8: Frequency Distribution of 8-bit Integer Operations

Operation Type		Percentage									
		Coremark		AutoBench		Assembler		Interpreter		Average	
	+	0		3.61		0.07		0.54		1.06	
	-	0		0.55		0.1		0.18		0.21	
Arithmetic	*	0	0	3.74	7.9	0	0.17	0.73	1.99	1.12	2.52
	/	0		0		0		0.54		0.14	
	%	0		0		0		0		0	
	==	1.88		0		3.48		2.18		1.89	
	! =	2.45		0.28		0.17		7.08		2.5	
Relational	<	0	4.71	0	0.42	0	3.86	1.09	10.98	0.27	4.99
	>	0		0.14		0		0.54		0.17	
	<=	0.19		0		0.21		0.09		0.12	
	>=	0.19		0		0		0		0.05	
	<<	0		0		0		0.09		0.02	
Shift	>>	0	0	0	4.16	0	0	0	0.36	0	1.13
	Arith >>	0		4.16		0		0.27		1.11	
	and	0.19		0.14		0		0.82		0.29	
Bitwise	or	0	0.19	0	0.14	0	0	0	1	0	0.33
	not	0		0		0		0.18		0.05	
	xor	0		0		0		0		0	
Complement		0	0	0	0	0	0	0	0	0	0
Absolute		0	0	0	0	0	0	0	0	0	0
Total		4.9	**4.9**	12.62	**12.62**	4.03	**4.03**	14.33	**14.33**	9	**8.97**

Table 3.9 provides the statistics of 16-bit integer operations. 16-bit operations have an overall frequency of 8%, where arithmetic operations have the contribution (3.27%).

Frequency distribution of operations applied to 32-bit integers is given in Table 3.10. It

Table 3.9: Frequency Distribution of 16-bit Integer Operations

Operation Type		Percentage									
		Coremark		**AutoBench**		**Assembler**		**Interpreter**		**Average**	
Arithmetic	+	1.69	2.81	1.8	7.49	0.24	0.48	0.64	2.28	1.09	3.27
	-	0.56		0.97		0.24		0.18		0.49	
	*	0.56		4.72		0		0.73		1.5	
	/	0		0		0		0.73		0.18	
	%	0		0		0		0		0	
Relational	==	0.38	1.89	0.42	1.26	0.03	0.17	1.36	4.09	0.55	1.85
	!=	0.56		0		0.14		0.73		0.36	
	<	0.38		0.42		0		1.18		0.5	
	>	0.19		0		0		0.73		0.23	
	<=	0		0.14		0		0.09		0.06	
	>=	0.38		0.28		0		0		0.17	
Shift	<<	0.56	2.44	0	0.28	0	0	0.45	1.09	0.25	0.95
	>>	0		0		0		0		0	
	Arith >>	1.88		0.28		0		0.64		0.7	
Bitwise	and	3.77	5.28	0	0	0	0	1	1.18	1.19	1.62
	or	1.13		0		0		0		0.28	
	not	0		0		0		0		0	
	xor	0.38		0		0		0.18		0.14	
Complement		0	0	0	0	0	0	0	0	0	0
Absolute		0	0	0	0	0	0	0	0	0	0
Total		12.42	**12.42**	9.03	**9.03**	0.65	**0.65**	8.64	**8.64**	7.69	**7.69**

can be seen that, 41.62% of the operations involve 32-bit data. Arithmetic and relational operations are the most frequent 32-bit operations with an average frequency of 21.04% and 16.67%, respectively.

Table 3.10: Frequency Distribution of 32-bit Integer Operations

Operation Type		Percentage									
		Coremark		**AutoBench**		**Assembler**		**Interpreter**		**Average**	
Arithmetic	+	12.99	26.18	10.96	23.73	5.43	17.37	9.08	16.89	9.62	21.04
	-	1.32		2.22		2.47		3.18		2.3	
	*	11.3		9.02		8.95		1.91		7.8	
	/	0.38		1.39		0.42		1.09		0.82	
	%	0.19		0.14		0.1		1.63		0.52	
Relational	==	0.94	10.36	2.22	12.77	13.54	24.55	8.54	18.98	6.31	16.67
	!=	2.64		2.36		5.4		2.72		3.28	
	<	5.08		6.38		2.65		5.45		4.89	
	>	0.94		1.53		0.66		1.09		1.06	
	<=	0.38		0		0.84		0.27		0.37	
	>=	0.38		0.28		1.46		0.91		0.76	
Shift	<<	0.19	0.57	0	0.28	0.03	0.1	1.73	2.91	0.49	0.97
	>>	0		0		0		0		0	
	Arith >>	0.38		0.28		0.07		1.18		0.48	
Bitwise	and	1.51	2.45	0	0	1.04	1.04	4.54	6.9	1.77	2.6
	or	0.56		0		0		2.18		0.69	
	not	0		0		0		0		0	
	xor	0.38		0		0		0.18		0.14	
Complement		0	0	0	1.32	0.24	0	0	0	0.06	0.33
Absolute		0	0	0	0	0	0	0	0	0	0
Total		39.56	**39.56**	36.78	**38.1**	43.3	**43.06**	45.68	**45.68**	41.36	**41.61**

3.3.3 Operands

Operations operate on operands and operands in *C* language can be of various types. Frequency distribution of various operands in selected programs is given in the Table 3.11. It can be seen from these statistics that constants and simple variables occur most frequently with an average frequency of about 32% and 44% respectively.

Table 3.11: Frequency Distribution of Operands

Operand	Percentage				
	Coremark	**AutoBench**	**Assembler**	**Interpreter**	**Average**
Constants	25.52	33.57	31.36	37.21	31.92
Simple Variables	55.6	38.87	38.71	44.22	44.35
Array Access	1.01	8.77	2.24	1	3.26
Struct/union Field Access	8.89	10.92	18.3	4.38	10.62
Function Calls	3.25	5.44	7.07	12.72	7.12
Pointers	5.71	2.43	2.31	0.47	2.73
Total	**100**	**100**	**100**	**100**	**100**

Because of the high frequency of constants, their further analysis is performed. Frequency distribution of different constants is given in the Table 3.12. It can be seen that small constants are the most frequent ones. Among the 4-bit constants, 0, 1, 2, 4, 8 are the most frequent. Constant 0 is frequent as it is used in initialization and comparison operations. 1 is also used frequently in increment/decrement operations like $i++$, $--i$ in loops. Overall, 4-bit constants have an accumulative frequency of about 87%.

Table 3.12: Frequency Distribution of Constants

Constant	Coremark		AutoBench		Assembler		Interpreter		Average	
	%	**Cum. %**	**%**	**Cum. %**	**%**	**Cum. %**	**%**	**Cum. %**	**%**	**Cum. %**
-1	0.11	0.11	0	0	0.95	0.95	1.41	1.41	0.62	0.62
0	18.44	18.55	16.63	16.63	18.61	19.56	30.06	31.47	20.94	21.56
1	21.75	40.3	33.69	50.32	18.58	38.14	19.71	51.18	23.43	44.99
2	2.13	42.43	4.8	55.12	8.18	46.32	7.26	58.44	5.59	50.58
3	3.78	46.21	3.94	59.06	7.62	53.94	10.66	69.1	6.5	57.08
4	13.24	59.45	8.96	68.02	13.29	67.23	4.97	74.07	10.12	67.2
5	2.13	61.58	1.39	69.41	1	68.23	3.32	77.39	1.96	69.16
6	1.18	62.76	0.64	70.05	1.3	69.53	4.97	82.36	2.02	71.18
7	2.36	65.12	0.96	71.01	1.43	70.96	4.32	86.68	2.27	73.45
8	21.28	86.4	11.19	82.2	1.59	72.55	2.33	89.01	9.1	82.55
9	0.24	86.64	0.43	82.63	1.4	73.95	0.42	89.43	0.62	83.17
10	0.24	86.88	0.75	83.38	1.7	75.65	0.27	89.7	0.74	83.91
11	0	86.88	0.43	83.81	1.22	76.87	0.38	90.08	0.51	84.42
12	0.95	87.83	0.43	84.24	0.65	77.52	0.23	90.31	0.57	84.99
13	0	87.83	0.43	84.67	0.76	78.28	0.15	90.46	0.34	85.33
14	0	87.83	0.43	85.1	0.76	79.04	0.15	90.61	0.34	85.67
15	0.71	88.54	0.64	85.74	1.32	80.36	0.34	90.95	0.75	86.42
16-31	3.07	91.61	1.92	87.66	8.61	88.97	1.38	92.33	3.75	90.17
32-63	3.55	95.16	5.65	93.31	3.4	92.37	0.92	93.25	3.38	93.55
64-127	1.18	96.34	0.75	94.06	4.62	96.99	2.02	95.27	2.14	95.69
128-256	1.42	97.76	0.75	94.81	0.49	97.48	1.57	96.84	1.06	96.75
256-65535	1.65	93.26	3.3	90.96	1.62	90.59	1.57	93.9	2.04	92.21
others	0.47	93.73	1.92	92.88	0.92	91.51	1.6	95.5	1.23	93.44

In order to see the frequency of size of operands, frequency distribution of 8-, 16-, 32-

and 64-bit operands appearing in different operations is given in the Table 3.13. 32- and 16-bit are the most frequent operand sizes with an average frequency of about 60% and 34%, respectively.

Table 3.13: Frequency Distribution of Operand Accesses Based on Size

Size (Bits)	Percentage				
	Coremark	AutoBench	Assembler	Interpreter	Average
8	3.11	7.63	3.7	11.81	6.56
16	41.01	40	33.1	20.91	33.76
32	55.87	52.37	63.2	67.28	59.68
64	0	0	0	0	0
Total	**100**	**100**	**100**	**100**	**100**

3.3.4 Miscellaneous

Table 3.14 gives the average number of variables based on locality per function. These variables can be of global scope, passed to this function as an argument or local variables of the function. It can be seen that on average, a function uses 7 locals. Furthermore, on average 2 arguments are passed to a function. Operands with global scope used inside a function are about 2.33 on average.

Table 3.14: Average (per Function) of Variables Based on Locality

Locality	Average				
	Coremark	AutoBench	Assembler	Interpreter	Average
parameters	3.04	1.42	1.13	1.53	1.78
locals	5	10.23	9.5	4.02	7.19
globals	0.16	1.81	5.31	2.02	2.33

The local variables and the arguments to the function can be simple variables, arrays, struct/union field or a pointer. Table 3.15 provides the frequency distribution of the arguments of a function based on data types. It can be seen that, most of the parameters passed to functions are either simple variables or pointers. Among simple variables, 32-bit integer variables are the most frequent data type passed as an argument to the function with a percentage distribution of 39% on average.

Array is never passed as argument to function. This is because most of the time the base address is passed as a pointer pointing to these data structures. Arguments containing struct/union are not frequently used as well, as they are also frequently passed as reference. In short, about 50% of the function parameters are pointers.

Table 3.15: Frequency Distribution of Parameters Based on Data Types

Operand Type		Percentage				
		Coremark	AutoBench	Assembler	Interpreter	Average
Simple Variable	Integer 8-bit	1.14	2.44	0	3.89	1.87
	Integer 16-bit	10.23	2.44	0	4.38	4.26
	Integer 32-bit	27.27	19.51	42.95	65.69	38.86
	Integer 64-bit	0	0	0	0	0
	Floating Point	4.55	0	1.34	8.27	3.54
Array		0	0	0	0	0
Struct/union		0	0	1.34	10.46	2.95
Pointer		56.82	75.61	54.36	7.3	48.52
Total		**100**	**100**	**100**	**100**	**100**

Locals to a function can also be of various types as given in Table 3.16 with their frequency distributions. Statistics show that, on the average about 88% of the locals are simple variables. Among simple variables, 32-bit and 16-bit integer varaibles are the frequent data types, with an average frequency of 61% and 13% respectively. About 13% of the locals are pointers.

Table 3.16: Frequency Distribution of Locals Based on Data Types

Operand Type		Percentage				
		Coremark	**AutoBench**	**Assembler**	**Interpreter**	**Average**
Simple Variable	Integer 8-bit	7.2	4.89	0.61	9.05	5.44
	Integer 16-bit	18.4	23.68	0	10.04	13.03
	Integer 32-bit	42.4	51.13	79.86	70.3	60.92
	Integer 64-bit	0	0	0	0	0
	Floating Point	5.6	2.26	2.53	7.78	4.54
Array		1.6	4.14	0.4	0.14	1.57
Struct/union		1.6	0	3.44	2.69	1.93
Pointer		23.2	13.91	13.16	0	12.57
Total		**100**	**100**	**100**	**100**	**100**

3.4 Conclusions

In order to see the characteristics of *C* language programs, this chapter discussed the static frequency distribution of various constructs in *C* language for embedded applications. Four *C* applications, namely EEMBC Coremark, EEMBC AutoBench, assembler and interpreter or our architecture were profiled. From the statistics, it can be concluded that among the statements, assignment statements are the most frequent statements. Most of these assignment statements are simple assignments with a variable on left hand side. Similarly, based on the complexity of expression on right hand side of assignments, constants and simple variables make up the most frequent cases. These assignments are translated to move and move immediate operations, which should be efficiently implemented. For the efficiency of memory accesses, there should be a support for efficient addressing modes. An interesting conclusion is that most of the simple assignments with an operand on right hand side have the same operation on left hand side destination. This shows the importance of 2-operand instructions, where one operand, while being a part of the operation, also serves as the destination to hold result.

Arithmetic operations have a higher frequency among all the operations, where in addition and multiplication having the major contributions. Bulk of operations involve integers of 16-bit and 32-bit sizes. Operations involving 8-bit size also have reasonable frequency, whereas, 64-bit operations almost never occur. This shows that architecture should have a support for 8-, 16- and 32-bit sizes, especially for memory efficient architecture.

Relational operations are the second highest frequent operations, as these are used to make decision for branches in selection and repetition instructions. This highlights the importance of conditions, which should be efficiently implemented for conditional control transfer instructions.

Type conversions are also frequent operations following arithmetic and relational operations. Most of the conversions are between 16- and 32-bit integer data type. It can be concluded that, support of type conversion with different operations will result in an efficient architecture.

Most of the operands in these operations are simple variables and constants. Based on the size of the operands, 16-bit and 32-bit operands are most frequent ones. In a memory efficient architecture, there should be special support for constants, especially 4-bit constants. Statistics showed that 4-bit constants make up about 87% of the total constants used in operations, 0 and 1 being the most frequent constants.

Statistics presented in this chapter showed that frequency distribution of *C* language constructs (statements, operations, operands etc) do not have a uniform distribution over the complete range. These results are utilized in making trade-off in the design of our microcontroller architecture discussed in next chapter.

4 MePoEfAr Architecture

This chapter contains the architectural details about the MeFoEfAr, which are confidential. Hence, it is not included in this public version of thesis.

MePoEfAr Assembler 5

In the previous chapter MePoEfAr architecture was discussed. To evaluate the efficiency of MePoEfAr architecture and have a comparison of performance with existing microcontrollers, benchmark programs need to be run on our architecture. In order to automate this task, MePoEfAr assembler and simulator was developed. Assembler is the focus of discussion in this chapter while interpreter will be discussed in the next chapter.

This chapter starts with the a brief introduction of assemblers. Section 5.2 discusses MePoEfAr assembler with the details of the intermediate steps involved to translate the assembly program to machine code. Section 5.3 discusses instructions bit assignments. Finally, Section 5.4 summarizes the whole chapter .

5.1 Introduction to Assemblers

Assembler is a utility program which translates the machine instruction written in the form of English mnemonics (assembly instructions) into binary patterns which machine can understand (machine instructions). This translation process is a one to one mapping of mnemonics to stream of bits representing the machine instruction and data. An important task of assemblers is to resolve symbolic names used in the assembly programs representing variables and memory locations. In order to resolve these references, assembler has to pass the assembly program once or twice depending upon the complexity of the assembly language. In this context, assemblers are generally classified as follows:

One-Pass Assemblers reads the source code once and preform the translations. The assumption is that all the references will be defined before their use. If they are not so, an error is generated. In short, One-Pass assembler cannot handle forward referencing.

Two-Pass Assemblers makes two passes over the assembly code. In the first pass it creates a symbol table. The values of the references are used in the second pass for the machine code generation. MePoEfAr assembler is a Two-Pass assembler.

In short, assembler has to perform a number of tasks. It has to perform lexical analysis, syntactic analysis, semantic analysis, maintain symbol table to resolve references and emit the machine code at the end.

5.2 MePoEfAr Assembler

MePoEfAr assembler is a Two-Pass assembler. It is written in *C* language and has 8194 lines of code, out of which 1944 lines of *C* code is generated by Flex and Bison from the description of lexical syntax and grammar as discussed in Section 5.2.1 and Section 5.2.2

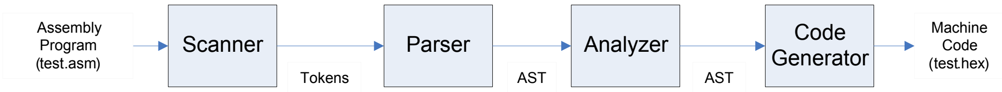


Figure 5.1: Block Diagram of MePoEfAr Assembler Showing Various Steps Performed in the Assembly Process

respectively. Based on the tasks performed by MePoEfAr assembler, it has been divided into following stages:

1. Scanner
2. Parser
3. Analyzer
4. Code Generator

Figure 5.1 shows the overview of the assembler. These stages are described one by one in the following sections. Listing 5.1 provides an example MePoEfAr assembly program which will be used in the description in the following sections.

```
;test.asm
;Simple MePoEfAr Assembly Program

MAIN:   MOVw    #2, W3      ;W3 = 2
        ADDw    #5, W3      ;W3 = W3 + 5
        SUBw    W3, 4(X5)   ;M[(X5)+4] = M[(X5)+4] - W3
END:    RTS                 ;return to caller
```

Listing 5.1: MePoEfAr Example Assembly Program used for Illustration of Various Assembler Stages in this Chapter

5.2.1 Scanner

Scanner is the first stage of the assembler to perform the lexical analysis. In this analysis, the assembly program in the file is scanned and broken down into tokens, leaving out the white spaces and comments. Lexical Analyzers can be generated by hand but pretty much efficient tools are available to generate the lexical analyzers. One such tool is **Flex** (**F**ast **Lex**) [1] which we have used to generate the lexical analyzer of MePoEfAr and is freely available. Flex code for the scanner is given in Appendix A. Figure 5.2 shows the block diagram of Scanner. It reads the assembly programs and generates the Tokens as shown.

Flex Code (*.l* extension) is compiled by *flex* to generate the *C* code (*.yy.c* extension) for the lexical analyzer based on the lexical description in Flex Code. The tokens generated by this *C* program are given as input to Parser. For instance, the tokens generated by our scanner for the example program given in Listing 5.1 are as given in Figure 5.3. It can bee seen that comments and white spaces are ignored. Newline is used to have a record of number of line in the source code for generating error messages. It can be seen from this figure that the first token is the *LABEL* corresponding to the label *MAIN*. Next is the *SYMBOL* token corresponding to *MOVw* instruction mnemonic in

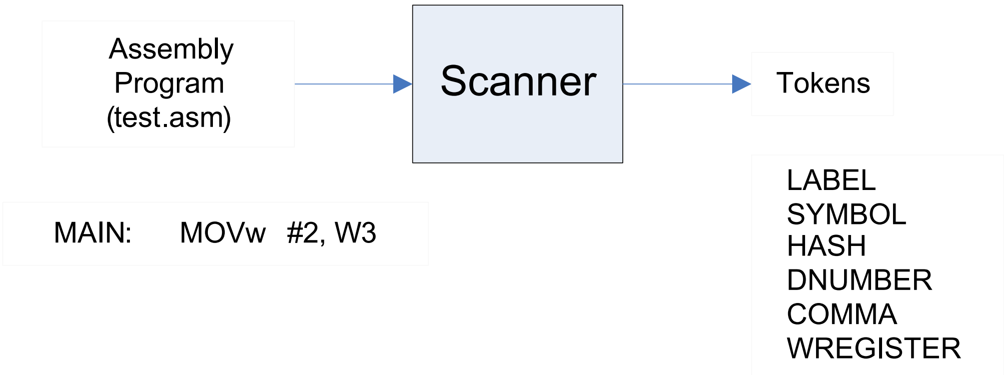


Figure 5.2: Block Diagram of Scanner, which Reads the Input Assembly Instructions and Produces the Tokens

LABEL SYMBOL HASH DNUMBER COMMA WREGISTER SYMBOL HASH DNUMBER COMMA WREGISTER SYMBOL WREGISTER COMMA WREGISTER NEWLINE SYMBOL WREGISTER COMMA DNUMBER LBRACK XREGISTER RBRACK LABEL SYMBOL

Figure 5.3: Tokens generated by Scanner for the Example Program in Listing 5.1

the first instruction at Line 4. Next token is the *HASH* symbol corresponding to # symbol for immediate value. Next a *COMMA* is seen and following *COMMA* is the *WREGISTER* token corresponding to *W3* in the assembly program. On the same lines, other instructions are also tokenized as shown in Figure 5.3.

5.2.2 Parser

Parser or Syntax Analyzer is the part of Assembler which determines the syntax or structure of a program based on the specified rules. These rules are called the grammar of the language. We have used Bison [11], a free parser generator, to generate the parser for MePoEfAr . Appendix B provides the grammar which we have used to generate the parser for MePoEfAr assembler. So, the tokens provided by Scanner are considered to make sentences according to this grammar. If the assembly program does not satisfy this grammar, a syntax error is generated.

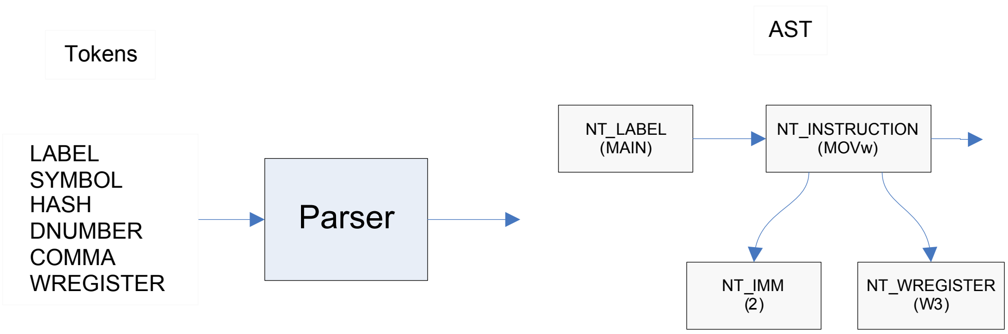


Figure 5.4: Block Diagram of Parser. Tokens are taken as Input from the Scanner and Parser Performs Syntactic Analysis and Constructs the Abstract Syntax Tree as an Output

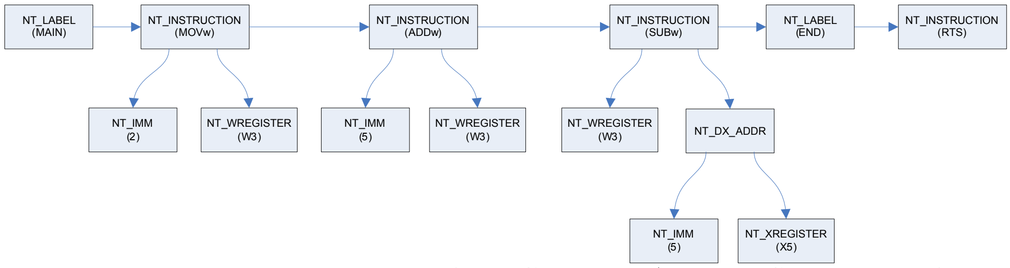


Figure 5.5: Visual Representation of the Complete Abstract Syntax Tree for the Example Program given in Listing 5.1

The output of the Parser is the abstract representation of the assembly program known as **A**bstract **S**yntax **T**ree (**AST**). Figure 5.4 shows the block diagram of Parser where it is shown that it takes the Tokens as input and generates the AST. The visual representation of the complete AST for the example program provided in Listing 5.1 is shown in Figure 5.5. In this AST, each right arrow is a pointer to next instruction. So, nodes in AST are linked together by next pointer as a linked list. Similarly, downward arrows indicate pointers to child. The first box represents the first node *NT_LABEL* which corresponds to Label *MAIN*. It does not have any child so there are no downward arrows. The next pointer points to next node *NT_INSTRUCTION* representing the instruction *MOVw*. This node has two children corresponding to the immediate field (*NT_IMM*) and destination register field (*NT_WREGISTER*). Similar explanation hold for the rest of the nodes in the figure. This AST is used in later phases to do semantic analysis and code generation.

5.2.3 Analyzer

At this stage, the AST generated by parser is traversed to perform semantic analysis. In the first phase, instruction groups are identified and symbols are added to symbol table. In order to know the location of various symbols in the assembly program, a *location* variable is updated according to the length of the instructions in the tree.

An crucial task in this analysis is regarding the maintenance of symbol table and to know the size of instructions. In MePoEfAr , instructions are variable length, so information about the length of the instruction is important to update the location counter. Interesting part is, length of the branch instruction depends up the branch displacement and to know the branch displacement we need to know the length of the instructions. For instance, consider the code segment given in Listing 5.2. The instruction *BRlt* in Line 6 has a 8-bit field for the branch target address(shown as *D8* in Table **??**). If the branch target address is greater than or equal to -128 and less than or equal to $+127$ then it can be accommodated in the first instruction word and size of the instruction will be 2 bytes. Otherwise, branch target address will be provided in the next instruction word, making it a 4-byte instruction. So, the size of this instruction depends upon the location of Lable *NoSWAP* which is a forward reference and has not been resolved yet (in the first pass). Furthermore, location of the Label *NoSWAP* depends upon the size of all the instruction proceeding it including the *BRlt* instruction at Line 6. This issue is resolved by assuming the worst case offsets for branch instruction and hence maximum size of the instruction (4 bytes) in the first pass. These are finalized in the symbol table based on the actual value in the second pass.

```
L1:      MOVw    W0,W1
L2:      MOVd    (X4)+,D2
         ...
         ...
         CPAd    D2, D3       ; compare D2 with D3
         BRlt    NoSwap       ; if(D3 < D2) then no swamping required
         ;otherwise swap here
         ...
         ...
NoSwap:  S1BR    W1, L2       ;loop if j>0
         ...
```

Listing 5.2: MePoEfAr Example Code Used for the the Illustration of Branch Instruction Size and Update of Location Counter

Table 5.1 shows the visual representation the Symbol Table for the example assembly program given in Listing 5.1. This table has two entries for the two symbols found in this example program. The names of these symbols are provided in first Column. Values of symbols are given in second column. Line number of use is also stored for generating the error and warning messages, as shown in the 3rd column of the table. For instance, the first symbol is *MAIN* which has a value 0 as it is the address of the first instruction. The column Line number shows us that it has been accessed at Line 4 in the source code (See Listing 5.1). Similarly, the Symbol *END* has the value 9 and is available at line 7 in the source code.

Table 5.1: Visual Representation of the Symbol Table for the Example Program in Listing 5.1

Symbol Name	**Symbol Value**	**Line Number**
MAIN	0	4
END	9	7

Type analysis is also performed in this stage. Data types are explicit in MePoEfAr assembly mnemonics, so it is checked if this data type matches the type specified by operand(s). For instance, the instruction *ADDb* #13, $B3$ expects the second operand to be a byte register. An error is generated, with the information about the line number of the instruction which caused this error, if types does not match. Similarly, error message is also generated if an operation is not defined in that instruction sub group. For instance, the instruction $MULb$ #13, $B3$ will cause an error as multiplication is not defined for integer Immediate to Register (*IR*) instruction format (See Table **??**).

5.2.4 Code Generator

In this phase of assembler, AST is traversed and binary patterns corresponding to assembly mnemonics are emitted. All the information required to generate the machine code

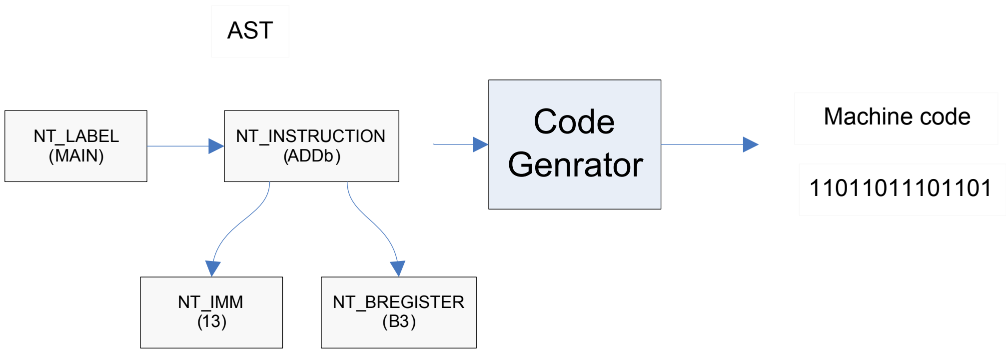


Figure 5.6: Block Diagram of Code Generator which Generates the Machine Code at the Output for the Abstract Syntax Tree of a Single Instruction at the Input

is present in the AST nodes, which is collected by earlier stages. Consider the example of code generation for instruction $ADDb$ #13, $B3$. The machine code generated for this instruction will be 11011011101101 in binary format or $DBED$ in hex format as shown in Figure 5.6 . This is because this instruction belongs to the Sub Group Immediate to Register (*SG IR*) (See Table **??**). So the binary code to represent *SG IR* for byte data type is 110110 as shown by Entry 16 in Table 5.2. The *OiIR* field will be 10 for the *ADD* operation (See Table **??**). *Rd* field will get the value 11 representing the Register *B3*. Immediate field I will get the value 1101 representing the immediate value 13.

The generated machine code for the given assembly program is written to a file in hex format, which will be given as input to the MePoEfAr interpreter.

5.3 Instruction Bit-assignment

The last phase in MePoEfAr assembler is the code generator stage. Binary patterns corresponding to assembly program for data, addresses and instructions is emitted. An important task in this stage is the assignment of binary patterns to mnemonics. This task is not trivial in MePoEfAr , as we have variable number of bits for the representation of mnemonics. We have utilized the concept of variable length coding to represent instruction sub groups.

The bit-assignment is based on the implementation assumption that after instruction-fetch, the instruction decode cycle will take place. During this cycle three register pre-fetches will take place, regardless of the details of the instruction. The three fields to be pre-fetched are:

Rs: the source register of a possible RR or MR instruction
Rd: the destination register of a possible MR, IR or R instruction
AX: the addressing mode and index register combination which may be used in a memory referencing instruction

The above logic requires that the fields of Rs, Rd and AX in the instruction layout are always in the same position of the 16-bit instruction word; regardless of the operation to be performed. In other words, the fields Rs, Rd and AX are always assigned to the same bit positions in the instruction.

Table 5.2: A Possible Bit Assignment for Various MePoEfAr Instruction Formats

<table>
<tr><th>#</th><th>SG</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>1</td><td>RR(b)</td><td colspan="4">0000</td><td colspan="3">OiRR</td><td colspan="4">Rs</td><td>OiRR</td><td colspan="4">Rd</td></tr>
<tr><td>2</td><td>RR(w)</td><td colspan="4">0001</td><td colspan="3">OiRR</td><td colspan="4">Rs</td><td>OiRR</td><td colspan="4">Rd</td></tr>
<tr><td>3</td><td>RR(d)</td><td colspan="4">0010</td><td colspan="3">OiRR</td><td colspan="4">Rs</td><td>OiRR</td><td colspan="4">Rd</td></tr>
<tr><td>4</td><td>MR(b)</td><td colspan="4">0011</td><td colspan="3">OiMR</td><td colspan="4">Rd</td><td colspan="5">AX</td></tr>
<tr><td>5</td><td>MR(w)</td><td colspan="4">0100</td><td colspan="3">OiMR</td><td colspan="4">Rd</td><td colspan="5">AX</td></tr>
<tr><td>6</td><td>MR(d)</td><td colspan="4">0101</td><td colspan="3">OiMR</td><td colspan="4">Rd</td><td colspan="5">AX</td></tr>
<tr><td>7</td><td>RM(b)</td><td colspan="4">0110</td><td colspan="3">OiRM</td><td colspan="4">Rs</td><td colspan="5">AX</td></tr>
<tr><td>8</td><td>RM(w)</td><td colspan="4">0111</td><td colspan="3">OiRM</td><td colspan="4">Rs</td><td colspan="5">AX</td></tr>
<tr><td>9</td><td>RM(d)</td><td colspan="4">1000</td><td colspan="3">OiRM</td><td colspan="4">Rs</td><td colspan="5">AX</td></tr>
<tr><td>10</td><td>MF</td><td colspan="4">1001</td><td colspan="3">OfMF</td><td colspan="4">Fd</td><td colspan="5">AX</td></tr>
<tr><td>11</td><td>FM</td><td colspan="4">1010</td><td colspan="3">OfFM</td><td colspan="4">Fs</td><td colspan="5">AX</td></tr>
<tr><td>12</td><td>CB</td><td colspan="4">1011</td><td colspan="4">CC</td><td colspan="8">D8</td></tr>
<tr><td>13</td><td>MX</td><td colspan="5">11000</td><td colspan="3">OxMX</td><td colspan="3">Xd</td><td colspan="5">AX</td></tr>
<tr><td>14</td><td>FF</td><td colspan="5">11001</td><td colspan="2">OfFF</td><td colspan="4">Fs</td><td>OfFF</td><td colspan="4">Fd</td></tr>
<tr><td>15</td><td>S1</td><td colspan="5">11010</td><td colspan="2">RG</td><td colspan="2">R</td><td colspan="7">D7</td></tr>
<tr><td>16</td><td>IR(b)</td><td colspan="6">110110</td><td>OiIR</td><td colspan="4">Rd</td><td>OiIR</td><td colspan="4">I</td></tr>
<tr><td>17</td><td>IR(w)</td><td colspan="6">110111</td><td>OiIR</td><td colspan="4">Rd</td><td>OiIR</td><td colspan="4">I</td></tr>
<tr><td>18</td><td>IR(d)</td><td colspan="6">111000</td><td>OiIR</td><td colspan="4">Rd</td><td>OiIR</td><td colspan="4">I</td></tr>
<tr><td>19</td><td>XX</td><td colspan="7">1110010</td><td colspan="3">OxXX</td><td colspan="3">Xs</td><td colspan="3">Xd</td></tr>
<tr><td>20</td><td>IX</td><td colspan="7">1110011</td><td colspan="2">OxIX</td><td colspan="4">I</td><td colspan="3">Xd</td></tr>
<tr><td>21</td><td>SAV</td><td colspan="7">1110100</td><td>S/R</td><td colspan="3">#RegPairs</td><td colspan="2">DT</td><td colspan="3">Rstart</td></tr>
<tr><td>22</td><td>SAVx</td><td colspan="7">1110101</td><td>S/R</td><td colspan="8">Mask</td></tr>
<tr><td>23</td><td>R(b)</td><td colspan="7">1110110</td><td colspan="4">Rd</td><td>0</td><td colspan="4">OiR</td></tr>
<tr><td>24</td><td>R(w)</td><td colspan="7">1110110</td><td colspan="4">Rd</td><td>1</td><td colspan="4">OiR</td></tr>
<tr><td>25</td><td>R(d)</td><td colspan="7">1110111</td><td colspan="4">Rd</td><td>0</td><td colspan="4">OiR</td></tr>
<tr><td>26</td><td>F</td><td colspan="7">1111011</td><td colspan="4">Fd</td><td>1</td><td colspan="4">OfF</td></tr>
<tr><td>27</td><td>M(b)</td><td colspan="8">11110000</td><td colspan="3">OiM</td><td colspan="5">AX</td></tr>
<tr><td>28</td><td>M(w)</td><td colspan="8">11110001</td><td colspan="3">OiM</td><td colspan="5">AX</td></tr>
<tr><td>29</td><td>M(d)</td><td colspan="8">11110010</td><td colspan="3">OiM</td><td colspan="5">AX</td></tr>
<tr><td>30</td><td>M</td><td colspan="8">11110011</td><td colspan="3">OfM</td><td colspan="5">AX</td></tr>
<tr><td>31</td><td>M</td><td colspan="8">11110100</td><td colspan="3">OxM</td><td colspan="5">AX</td></tr>
<tr><td>32</td><td>NO</td><td colspan="8">11110101</td><td colspan="8">NOOP only takes 8 bits</td></tr>
<tr><td>33</td><td>InXS</td><td colspan="8">11110110</td><td colspan="8">Mask</td></tr>
<tr><td>34</td><td>InM</td><td colspan="9">111101110</td><td colspan="2">DT</td><td colspan="5">AX</td></tr>
<tr><td>35</td><td>InMS</td><td colspan="9">111101111</td><td colspan="2">DT</td><td colspan="5">AX</td></tr>
<tr><td>36</td><td>X</td><td colspan="10">1111100000</td><td colspan="3">OxX</td><td colspan="3">Xd</td></tr>
<tr><td>37</td><td>Ju</td><td colspan="10">1111100001</td><td colspan="2">OC2</td><td>@</td><td colspan="3">Xd</td></tr>
<tr><td>38</td><td>InR</td><td colspan="10">1111100010</td><td colspan="2">DT</td><td colspan="4">R</td></tr>
<tr><td>39</td><td>InRS</td><td colspan="10">1111100011</td><td colspan="2">DT</td><td colspan="4">Rstart</td></tr>
<tr><td>40</td><td>BitOP</td><td colspan="10">1111100100</td><td colspan="2">OBit</td><td colspan="4">Bit#</td></tr>
<tr><td>41</td><td>InX</td><td colspan="12">1111100101</td><td colspan="4">X</td></tr>
<tr><td>42</td><td>InS</td><td colspan="16">1111100110</td></tr>
<tr><td>43</td><td>InSS</td><td colspan="16">1111100111</td></tr>
<tr><td>44</td><td>RTS</td><td colspan="16">1111101000</td></tr>
<tr><th>#</th><th>SG</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
</table>

Table 5.2 shows a possible bit-assignment for the MePoEfAr instructions. The columns numbered from 15 through 0 denote the instruction bits of the first instruction word. The column *SG* lists the Sub Group which is implemented by the corresponding Table entry. The SGs are taken from Table **??**(See Column SG). For example, Entry No 4 in Table 5.2 is the bit assignment for the SG MR which has instruction code 0011 (i.e.,

two zeros and two one's in binary, and not eleven) specified by bit positions 12-15. This pattern is for the memory register operations for byte data type. The OiMR stands for the Operations on integers Memory to Register format as specified in Table **??**. Rd is the destination register and AX represent the addressing mode and index register combination for the specification of source operand which is in memory.

From the Table 5.2 it is clear that the instructions are systematic, such that simple and fast encoding is possible. In addition, a fair amount of unused opcode space is available for future requirements.

One idea which may be mentioned at this point is that it may be better to have two sets of indeX registers: one set which is used in User Mode and one set which is used in Supervisor mode; hence the selection is done by the Mode bit of the Status Register. The context switching can be very fast because interrupt handlers can have their private register sets and use the supervisor indeX registers.

5.4 Summary

Assembler is a piece of code which translates assembly instructions to machine code. In this chapter we have discussed the MePoEfAr Two-Pass assembler assembler. Figure 5.7 shows the summary of the steps taken by MePoEfAr Assembler for the translation assembly program to machine code. It can be seen from this figure that scanner is the first stage of MePoEfAr Assembler. Whitespace and comments are left out by scanner and tokens are passed to parser. Parser performs the syntactic analysis and constructs the Abstract Syntax Tree (AST) based on the defined grammar. An important task in this translation process is maintaining the symbol table which is done by traversing the AST. This process was involved for MePoEfAr assembly language because of two reasons. Firstly, instructions in MePoEfAr are variable length instructions. Secondly, size of the branch instructions depends upon the offset used for branch displacements. We resolved this issue by assuming the worst size of branch instructions in first pass and updating the proper instruction lengths and hence the location counter in the second pass.

The last stage in this translation process was generating the binary patterns for the instructions. For this, variable length instruction subgroups were assigned the bit patterns based on the concept of variable length coding. Fast encoding of instructions was taken into account during this bit assignment process. The machine code generated by assembler will be fed to MePoEfAr Interpreter which is discussed in the next chapter.

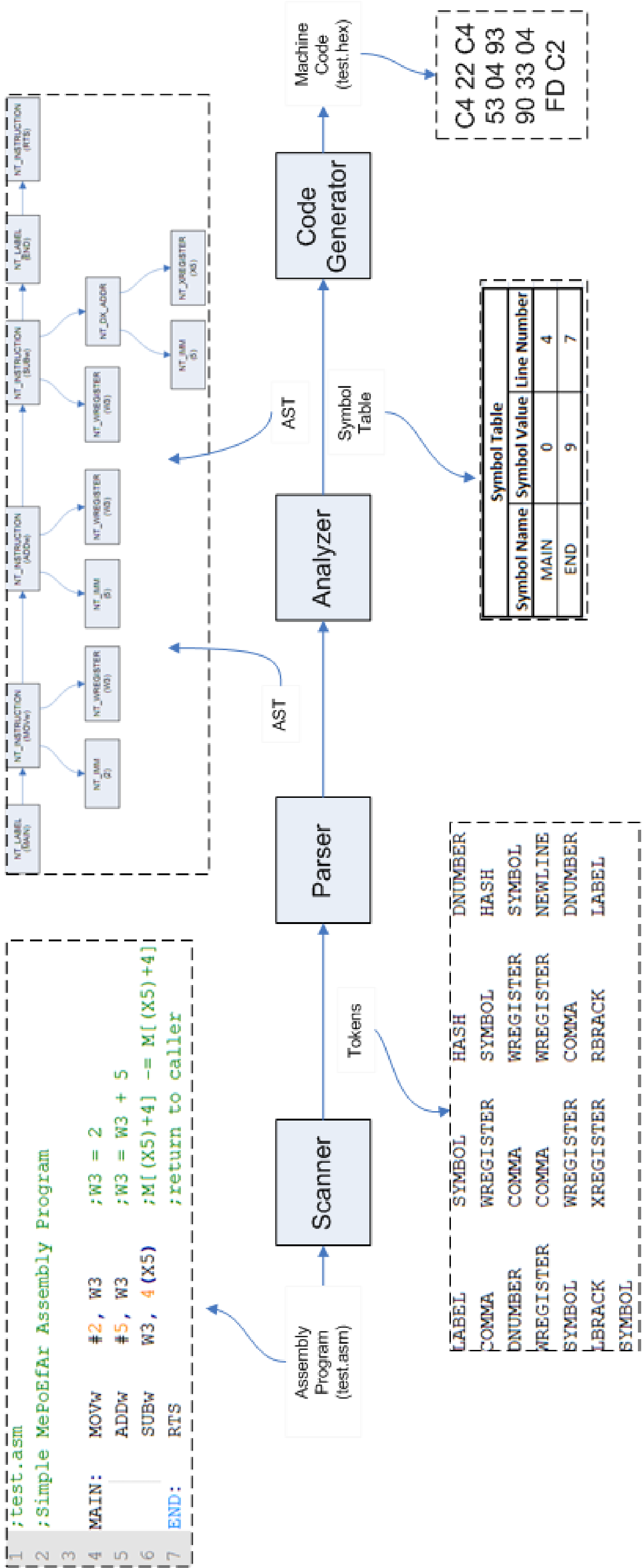


Figure 5.7: Summary of MePoEfAr Assembler Showing Various Steps Performed in the Assembly Process

6 MePoEfAr Interpreter

Machine code gets executed either on a real architecture or on its abstract model. Architecture models are utilized in simulators in the early design phase for a number of reasons. Firstly, simulators are used to run benchmark programs and obtain performance results for an architecture in the early design phase. Secondly, microcontroller architecture is a collection of sub-systems. Simulators are developed to verify the conformance of these sub-systems to the functionality as described in the architecture document. Thirdly, simulators can be utilized to debug and test development and application programs targeted for the new architecture. This implies that the software tool chain (compiler, assembler, linker) can be developed and tested in parallel with the development of the hardware platform.

In the previous chapter, we discussed our MePoEfAr assembler which we developed to generate the machine code from MePoEfAr assembly programs. In order to debug and test the functionality of MePoEfAr assembly programs, we developed the MePoEfAr simulator, which is discussed in this chapter.

This chapter starts with a brief overview of simulators in Section 6.1. Section 6.2 provides a high level description of the MePoEfAr Interpreter. Part of the interpreter working as supervisor program, is discussed in Section 6.3. This section also discusses the way source line numbers of the instructions in the assembly programs are mapped to the memory address of instructions. The MePoEfAr microcontroller model is described in Section 6.4, which actually executes the programs. Finally, Section 6.5 summarizes the whole chapter.

6.1 Overview of Simulators

Architecture models are developed in the early design phase of the architecture for a number of reasons [8]. The models are know as simulators or cross simulators as they simulate the functionality of the target architecture on a host machine. When these models are used to test the instruction set of an architecture, they are referred to as as **I**nstruction **S**et **S**imulators (**ISS**). The ISS of an architecture can be designed in two ways [26]:

1. Interpretive Simulators [27], [22], [29] in which the machine code of the program is loaded in to the memory of the architecture. Instructions are fetched, decoded and executed one by one much like the real architecture. Interpretive Simulators have the advantage that simulator does not need to be re-generated when the application program is modified (as is required in the compiled simulators, discussed below). The disadvantage is the low speed of interpretive simulators. [27] discusses an

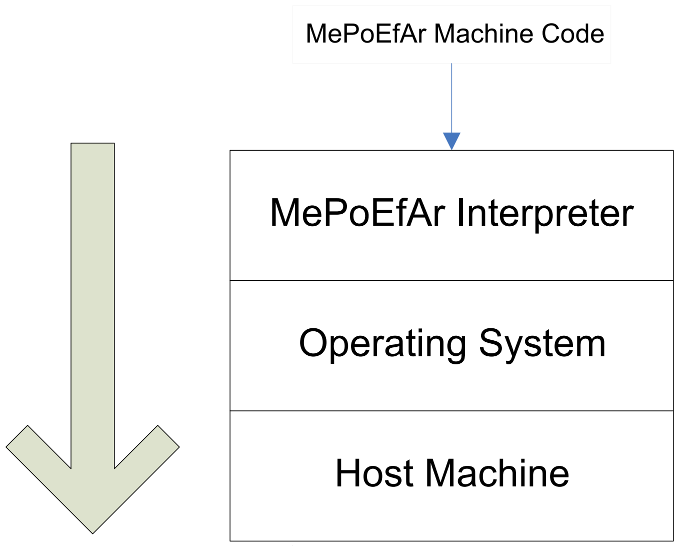


Figure 6.1: Block Diagram of MePoEfAr Interpreter Showing its Position in Relation to the Host Machine

ARM interpretive simulator. We have developed an interpretive style simulator for our MePoEfAr architecture.

2. Compiled Simulator [28], [32] which generates an executable simulation file per application program. It has the advantage of speed, because the instruction decoding overhead moves to simulator generation time. The disadvantage is that it requires a recompilation of the whole simulator in order to simulate a different file.

6.2 MePoEfAr Interpreter

MePoEfAr simulator has been developed as interpretive simulator to closely resemble the instruction fetch, decode and execute stages of the architecture. MePoEfAr interpreter program is 5597 lines of code written in *C* language. The advantage of developing it in a high level language like *C* is that it is easily portable to other platforms with a recompilation of the interpreter. This interpreter reads the machine code generated by assembler and executes these instructions on host PC. Figure 6.1 shows the relation of MePoEfAr interpreter with respect to host machine. It can be seen from this figure that MePoEfAr Interpreter reads the machine code and communicates with the operating system layer for its execution. Next, operating system sends instructions to the host machine to execute this program.

Figure 6.2 shows the block diagram of MePoEfAr Interpreter. It can be seen from this figure that interpreter consists mainly of two blocks, as listed below:

Supervisor (main()) Program which loads the program to memory and instructs the microcontroller to RUN the program

MePoEfAr Microcontroller Model which executes the program

These two parts are discussed one by one in detail in the next two sections.

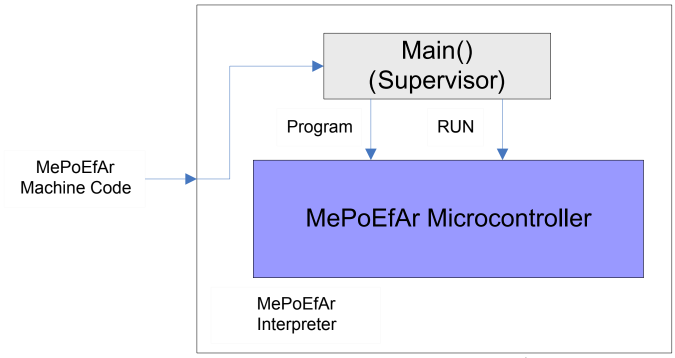


Figure 6.2: Block Diagram of the MePoEfAr Interpreter

6.3 Supervisor Program (*main()*)

Supervisor or *main()* program is the part of MePoEfAr interpreter which supervises the interpretation process. It reads in the machine code and subsequently loads it into the data structure which represents the program memory of MePoEfAr architecture. Next it instructs the microcontroller to execute the loaded program. These tasks of the *main()* program are depicted in 6.1.

```
1  int main(int argc, char * argv[])
2  {
3
4      if( argc > 1) //if there is a command line argument for the name of file
5          strcat(inputFileName, argv[1]); //use this name
6      else
7          strcat(inputFileName,"test.hex"); //otherwise default test.hex will be used
8
9      printf("\n  MePoEfAr Interpreter    \n");
10
11     initFiles();        //utility function to initalize files
12                         //for different purposes
13
14     printf("\n Hex file (%s) will be loaded to Program Memory \n",inputFileName);
15     printf("\n Loading program in to memory ... \n");
16     loadPM();        //call this function to load the machine code into the
17                      //program memory. The actual bytes representing the
18                      //machine code will be loaded and the information about
19                      //the line numbers of source program will be used for
20                      //the mapping of program memory and line numbers
21
22     runProgram();    //start the show
23                      //execute the program
24
25     printf("\n Finished Program Execution ... \n");
26
27     //interact with the user
28     printf("\n Select a choice from the follwoing menu to see results ... \n");
29     interact();         //prompt the user if he wants to monitor
30                         //registers, memory etc
31
32     closeFiles();       //close the files opened for internal use
33     return 0;
34 }
```

Listing 6.1: MePoEfAr main() Interpreter *C* Code. It Prompts the User for Input Hex File, Calls *loadPM()* to load it into memory. *runProgram()* Executes the Loaded Program

An important task which is performed by program load function (*loadPM()* at Line 16 in Listing 6.1) is the mapping of source line number of the instructions to their memory addresses. This is important because the feedback provided by the interpreter in the form of errors and warnings becomes very helpful if it points to the source line number. Next sub section describes in detail how we have achieved this in our MePoEfAr interpreter.

6.3.1 Memory Address to Source Line Number Mapping

MePoEfAr Interpreter is developed such that user is able to see the contents of internals of MePoEfAr architecture. An important feature of MePoEfAr Interpreter is its ability to give the information about the running instruction with the source line number of the original assembly program. This is achieved by putting the line numbers of the source assembly instructions inside the generated machine code. On the Interpreter side, the *main* program, which acts as the supervisor will call the function *loadPM()* to load the program. This function is designed such that it serves two purposes. At first, It will load the actual machine code into the program memory. Secondly, it will store the mapping of memory addresses and source line numbers inside a list. The list to hold these mapping entries, is implemented utilizing the hash function concept. Listing 6.2 shows the code for this mapping. Two important functions in this regard are:

1. *insertMapping()* which inserts a mapping entry in the list.
2. *searchMapping()* which searches for entry that contains source line number of the requested memory address.

```
/*
Structure to represent a node in the mapping list.
PMAddress is mapped to lineNo, which is represented
by the entry of this node in the list.
*/
typedef struct node
{
    int lineNo;              //line number in source program
    int PMAddress;           //program memory address
    struct node *next;       //pointer to next node
}*mapList;                   //pointer to a list of such nodes

/* the hash table */
static mapList hashTable[SIZE];

/*
Function insertMapping
input is the pmAddr and lno which is to be mapped
returns 0 on success and mapping entry is inserted successfully
return -1 if mapping is already there, or it cannot be inserted
because of memory allocation problem
*/
int insertMapping( int pmAddr, int lno)
{
    int h = hash(pmAddr);        //temporary to hold the key from hash function
    mapList l = hashTable[h];    //get the key from the hash function

    //loop till mapping found or till the end of list
    while ((l != NULL) && (pmAddr != l->PMAddress) )
        l = l->next;

    if (l != NULL)  // found in list
        return -1;  //unsuccessful return
    else // mapping not in list
    {
        l = (mapList) malloc(sizeof(struct node));  //allocate memory
        if(l != NULL)
        {
            l->PMAddress = pmAddr;       //save memory address for this entry
            l->lineNo = lno;             //save the corresponding line no
            l->next = hashTable[h]; //pointer to next, get from hash func
            hashTable[h] = l;            //
            return 0;        //successful return
        }
        else
            return -1;  //unsuccessful return
                  //memory allocation problem
```

```
    }
}

/* Function searchMapping
searches the map entry of pmAddr and corresponding lno.
If found, this lno is written as its address is the argument to the function.
returns 0 if mapping found
returns -1 if mapping not found
*/
 int searchMapping( int pmAddr, int * lno )
{
    mapList l =  hashTable[hash(pmAddr)];   //hash table entry

    //loop till mapping found or till the end of list
    while ((l != NULL) && (pmAddr != l->PMAddress) )
        l = l->next;

    if (l == NULL)              //not found till the end
        return -1;              //signal failure

    else                        //found
    {
        *lno = l->lineNo;       //write the lno
        return 0;               //signal success
    }
}
```

Listing 6.2: Code Used to Store the Mapping of Program Memory Address and Line Numbers in MePoEfAr Interpreter

6.4 MePoEfAr Microcontroller Model

The main part of MePoEfAr Interpreter is the *microcontroller*. After the program is loaded to program memory, *runProgram()* function is called to execute the loaded program. This program execution is done in a loop as shown in Listing 6.3. The body of this loop consists of four main functions as discussed below:

fetchInstruction() fetches the first word (2 bytes) of instruction from the location pointed by the program counter and copies it into a temporary data structure (*instrTemp*) for later processing. This *instrTemp* is passed to it by reference as can be seen from Line 13 in Listing 6.3.

decodeInstruction() decodes the instruction passed to by reference as can be seen from Line 16 in Listing 6.3. This is the function which identifies the **S**ub **G**roup (**SG**) of the instruction. Details of the decoding process are provided later in a separate section.

executeInstruction() executes the instruction passed to it as argument. In this function, a *switch* statement selects the function corresponding to its SG to execute it. The instruction gets executed and changes (if needed) the state of the microcontroller based on its operation.

interact() interacts with the user in case the step mode is enabled as can be seen from Line 32 in Listing 6.3. After each instruction is executed, interpreter prompts the user whether he wants to see the internals of the architecture. In case the step mode is disabled, complete program gets executed and the user can interact only at the end of the program.

```
1 /*
2 Function runProgram(), which executes the program
3 instruction are fetched, decoded and executed one by one.
4 If step by step mode is defined, then
```

```
*/
void runProgram()
{
    int lno;    //temp to hold lno of current instruction
    Instruction instrTemp; //temp to hold current instruction

    while(PC<noOfBytes) //loop till complete program
    {
        fetchInstruction(&instrTemp);        //fetch insruction
        instrTemp=swapInstrBytes(instrTemp); //this will swap the bytes for proper endianess

        decodeInstruction(&instrTemp);  //decode instruction
        printf("Current Instruction SG = %s \n", SG_TYPE_TITLE [instrTemp.SG] );

        //first of all read the source line no from the linked list
        if( searchMapping(PC,&lno) == 0 )   //search for mapping
            printf("\n Executing instruction from line %d \n", lno);
        else
            printf("\n Could not find the Mapping for PM Address : %d\n",PC);

        executeInstruction(instrTemp);  // execute instruction
                                        // update PSW
                                        // update PC accordingly in case if more bytes fetched

        //if step by step mode is active then ask user to continue or
        //if he wants to have a look at some registers or memory or...
        #ifdef STEP_MODE
        interact();
        #endif
    }
}
```

Listing 6.3: *runProgram()* Function in which Instructions are Fetched, Decoded and Executed

In order to achieve this instruction fetch, decode and execute, internal components of MePoEfAr architecture were modeled as data structures. These components are listed below:

1. Program Counter
2. Program Status Word
3. Registers
4. Program Memory
5. Data Memory
6. Stack and Stack Pointer
7. Decoder
8. Arithmetic and Logic Unit

the following is a brief description of the implementation of each of these components.

6.4.1 Program Status Word

Four condition code bits from the **P**rogram **S**tatus **W**ord (**PSW**) namely zero flag, sign flag, carry flag and over flow flag are modeled as global integers which are updated after an instruction which affects these flags is executed [1].

6.4.2 Program Counter

Program **C**ounter (**PC**) is modeled as a global counter pointing to the address of the next instruction to be executed. It is updated after each instruction fetch (or fetching of

[1] These flags are always visible at the terminal showing the updated status of condition codes based on the status of recently executed instruction.

instruction bytes with size larger than two bytes), or execution of instructions operating on PC (Branch and Jump instructions).

6.4.3 Registers

Registers are modeled as arrays of corresponding data type. Functions are provided to read from and write to these registers.

6.4.4 Program Memory

Program Memory is modeled as an array of *int8_t* [2] data type. A variable indicates the number of bytes of program loaded into program memory, which is updated during the program load. Functions are provided to fetch instruction bytes from program memory.

6.4.5 Data Memory

Data Memory is modeled as an array of *int8_t* data type. Basic functions are provided to read and write the data memory. These functions are then utilized to define functions to read and write data as integer and floating point values.

6.4.6 Stack and Stack Pointer

Stack area is a part of data memory and starts from highest memory address and grows towards the lower memory address. A pointer pointing to current position on stack, known as **S**tack **P**ointer (**SP**) is implemented which is used in stack related operations (subroutine call and return). SP is initialized to highest data memory address, and whenever something is pushed on stack, SP is decremented and vice versa.

6.4.7 Decoder

Instruction decoder is implemented as nested *switch* statements as can be seen from Listing 6.4. The outer *switch* statement (Line 17) selects the case based on the number of bits to be considered. The starting value is 4 as it is the minimum number of bits to identify an SG. The inner *switch* statement matches the proper SG among the options available based on the match of these bits value to code of that SG. These *switch* statements execute inside a *while* loop which iterates until instruction SG is identified or no of bits to be considered for making the decision equals 16 (bits in one instruction word). On each iteration of the loop, the number of bits to be considered for decoding are incremented as can be seen from Line 43.

```
1 /*
2 Function decodeInstruction() decodes the instruction.
3 Input is a pointer to the instruction and based upon
4 the decoding logic described in the instruction bit
5 assignment, Sub Group of instruction will be updated.
6 */
```

[2] *int8_t* is always an 8-bit data type which is defined in *stdint.h*

```
void decodeInstruction(Instruction *instrTemp)
{
    int bitsToConsider=4;   //start with considering 4 bits
    unsigned int bitsValue; //value of the considered bits

    while(bitsToConsider<=16)   //maximum bits in instruction is 16
    {
        //slice the bits which we want to consider to compare its value
        bitsValue = sliceBits(instrTemp->shortInstr,bitsToConsider);

        switch (bitsToConsider)
        {
            case 4: //instructions with 4 bit SG field
                switch(bitsValue)
                {
                    case RRbCode:   instrTemp->SG=SG_RRb;   return;
                    case RRwCode:   instrTemp->SG=SG_RRw;   return;
                    case RRdCode:   instrTemp->SG=SG_RRd;   return;
                    case MRbCode:   instrTemp->SG=SG_MRb;   return;
                    case MRwCode:   instrTemp->SG=SG_MRw;   return;
                    case MRdCode:   instrTemp->SG=SG_MRd;   return;
                    /*  and so on other sub groups with 4-bit SG field
                    are decoded */
                }
            break;
            case 5: //instructions with 5 bit SG field
                switch(bitsValue)
                {
                    case MXCode:    instrTemp->SG=SG_MX;    return;
                    case FFCode:    instrTemp->SG=SG_FF;    return;
                    case S1Code:    instrTemp->SG=SG_S1;    return;
                }
            break;

            /*  and so on other sub groups are also decoded */
        }
    bitsToConsider++;       //increment bits to consider if not found
    }
}
```

Listing 6.4: Code for Instruction Decoding

The end result of this decoding is that either the instruction is identified correctly and SG field is updated with the proper sub group, or instruction SG field is updated with *SG_NA* indicating a **N**ot **A**pplicable SG for the execute stage.

6.4.8 Arithmetic and Logic Unit

Arithmetic and Logic operations constitute the bulk of operations operating on various data types as defined in MePoEfAr architecture. An **A**rithmetic and **L**ogic **U**nit (**ALU**) is modeled as a number of functions to execute these operations for all the data types. During the execution phase, based on the data type and operation the corresponding function is selected by a *switch* statement, which will perform the operation and update the condition codes as defined in the architecture.

6.5 Summary

In order to test and debug the programs written for a specific architecture, these programs are translated into the form understandable by machine. This machine can be the real machine or a model of the machine implemented in a high level language. Simulators are developed in the early architecture design phase to model these architectures. Interpretive simulator is the style of MePoEfAr interpreter which we have discussed in this chapter. From the two main parts of this interpreter, one part of MePoEfAr interpreter is the *main* program which loads the machine code in the program memory, maps the memory address of instructions to their source line numbers (in the original assembly program). This mapping is important for testing and debugging the assembly programs, as the feed back given by interpreter in form of error and warning messages are useful if they have the information of the source line numbers.

The second part of the interpreter is the model of the MePoEfAr *microcontroller*. Various components of MePoEfAr architecture are modeled inside this microcontroller. These components are utilized in the loop which is executing the instructions one by one. In this loop, instructions are fetched, decoded and executed. If step mode is active, the interpreter interacts with the user in case he is interested to examine the state of the microcontroller. The interpreter described in this chapter is used to debug and test the functionality of benchmark programs used to evaluate and compare the performance of MePoEfAr architecture. Benchmarking details are provided in next chapter.

7 Assembler Level Benchmarking

In this chapter, performance of MePoEfAr architecture is analyzed and compared to three other well known microcontroller architectures. Performance of an architecture for the given benchmark is also dependent upon the quality of code generated by the compiler. In order to have a comparison solely based on architectural capabilities, we have performed our first round of benchmarking at the assembler level. Assembler and Interpreter of MePoEfAr have been discussed in previous two chapters.

Six benchmark programs are selected to test different aspects of architecture. These application programs are hand assembled and optimized for MePoEfAr architecture, as well as, for the other three candidate architectures for a fair comparison. Appendix C contains the hand assembled optimized programs for all the four architectures considered for comparison.

This chapter starts with the description of the evaluation criteria. Candidate architectures are briefly mentioned in Section 7.2. Benchmark application programs are described in Section 7.3. Performance results with comparison and evaluation are discussed in Section 7.4. Section 7.5 summarizes the chapter with a table of combined results to give the overall impression of performance comparison.

7.1 Evaluation Criteria

Performance has been evaluated based on efficiency of architecture in terms of number of instructions, program memory size (bytes) and execution time (cycles). To estimate the power consumption, number of instructions executed, program/data memory traffic (cycles) has also been calculated. These results can be classified in two main categories. Figure 7.1 shows the classification of results which are calculated for evaluation and comparison.

7.2 Candidate Architectures for Comparison

Performance of MePoEfAr architecture is compared with three famous architectures which are being widely used as embedded microcontrollers. These architectures are:

1. Atmel AVR AT90S851 (8 Bit)
2. TI MSP430G2231 (16 Bit)
3. ARM LPC1342 (32 Bit)

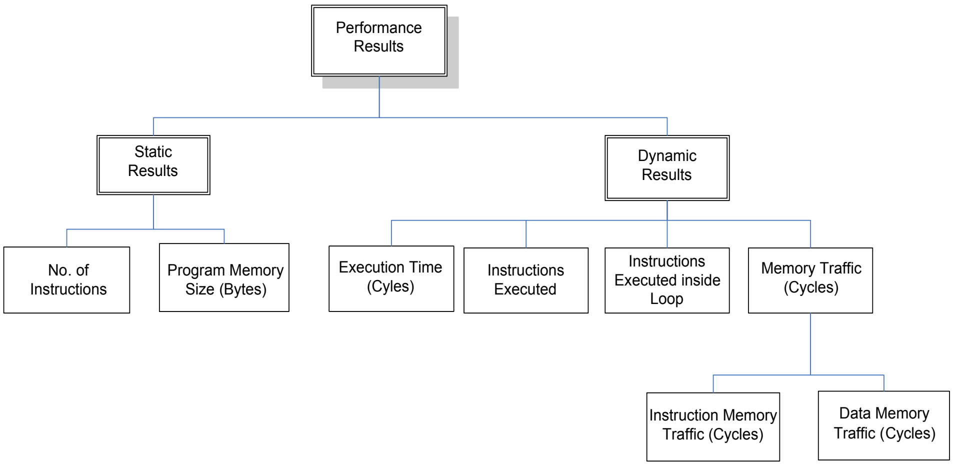


Figure 7.1: Classification of Evaluation Criteria

7.2.1 Atmel AVR AT90S851

AT90S8515 is a low power, CMOS, 8-bit microcontroller based on the AVR RISC architecture [15] developed by Atmel [14]. It utilizes modified Harvard architecture concept. Although it is an 8-bit microcontroller, each instruction takes one or two 16-bit words. It has 32 single-byte general purpose registers with single clock cycle access time. It supports five addressing modes.

7.2.2 TI MSP430G2231

Second candidate is MSP430G2231 [3], a 16-bit RISC architecture developed by Texas Instruments [2]. It has been designed for low cost and low power embedded application. It uses von-Neumann architecture with a single instructions and data memory space. Instructions generally take one cycle per word fetched or stored. It has 27 core instruction and 24 emulated instructions. It supports seven addressing modes for source operands and four addressing modes for the specification of destination operands in instructions. It has the following 16-bit registers:

R0: Program counter
R1: Stack pointer
R2: Status register (only in register addressing with word data type)
R2 and R3: are used as constant generators for the most frequent constants (0,1,2,4,8)
R4-R15: General purpose registers

The user guide found here [4] provide further details of MSP430 microcontroller architecture.

7.2.3 ARM LPC1342

Third candidate is the LPC1342 [13] developed by NXP (founded by Phillips) [12]; a Cortex-M3 based low power 32-bit RISC m1icrocontroller. ARM is a fab-less company which designs these architectures as **I**ntellectual **P**roperty (**IP**) modules and sells licenses to other companies which actually manufacture the chips, in the case of LPC1342, the manufacturing company is NXP. There are various architectures provided by ARM targeting various application areas, such as:

- ARM Cortex-A series targets the general purpose processor cores
- ARM Cortex-R series is a family of processors for real time systems
- ARM Cortex-M series processors are designed for low-power, memory efficient embedded applications

Among this M-series processors, Cortex-M3 processors are especially designed for embedded microcontrollers. It is based on modified Harvard architecture concept. It supports Thumb-2 instruction set to reduce the instruction memory requirements by including the support for 16-bit instructions. It has following general purpose and special purpose registers:

R0-R12: General purpose registers
R13: Stack pointer
R14: Link registers used by subroutines for return address
R15: Program counter
xPSR Program Status Register

Registers R0-R7 are accessible by all instructions, whereas, registers R8-R12 are only accessible by 32-bit instructions and 16-bit instructions cannot access them. The technical reference manual of ARM Cortex-M3 architecture (as well as other ARM architectures) can be found here [5] for further details.

For the sake of brevity, in rest of the chapter, MePoEfAr , TI, ARM and AVR refers to our architecture, TI MSP430G2231, ARM Cortex-M3 LPC1342 and AVR AT90S851 microcontrollers respectively.

7.3 Selected Benchmark Programs

In this section a brief description of the benchmark programs is presented. The central idea of each algorithm is summarized and the features of microcontroller architecture which will be tested by each application are also mentioned.

Three types of Microprocessor /Microcontroller/DSP benchmarks are known in general [25], [34]:

1. 1. Synthetic benchmarks (e.g. Whetstone Benchmark [23], Dhrystone Benchmark [16]) developed to measure system specific parameters (CPU, Compiler, and so on)
2. Application based benchmarks (real world benchmarks) developed to compare different system architectures in the same real fields of application, for instance EEMBC benchmarks [10] such as AutoBench [9], Coremark [7]
3. Algorithm based benchmarks (a compromise between the first and the second type) developed to compare different system architectures in special (synthetic) fields of application.

The benchmark code used to test the processor architecture and compilers can be separated into eight different modules:

1. Fixed-point math algorithms
2. Floating-point math algorithms
3. Logic calculations
4. Digital control
5. Fast Fourier Transform
6. Field processing
7. Loops and conditional jumps
8. Recursion and stack tests

At the assembler level, writing hand assembled codes for full fledged benchmarks for these different architectures is a time consuming process. So we picked up some part of these benchmarks (which are doing the real computations inside) and used them for our assembler level performance evaluation and comparison. Following programs have been used for our assembler level benchmarking:

1. Recursive Factorial Algorithm
2. String Copy Function
3. Bubble Sort Algorithm
4. Sensor Structure Program
5. Matrix Multiplication
6. FIR Algorithm

The above mentioned applications cover most of the features mentioned in above 8 modules. A brief description of these benchmark programs is given below.

7.3.1 Benchmark Application 1: Recursive Factorial Program

This program is the recursive factorial calculation program. It is based on the concept that factorial of a number n is the number times the factorial of previous number $(n-1)$. This implies, factorial of n can be calculated if we know the factorial of $n-1$. This divide and conquer approach is continued till number is reduced to 1. Factorial of 0 and 1 is 1, which is the base case of recursion. A number is passed to *factorial()* function from the *main()*. This function calculates the factorial and returns the result to main function. Listing 7.1 provides the commented *C* code of this program.

```
/*
FactRec Benchmark Program
C Program implementing recursive factorial function.
A number is passed as an argument to this function and
factorial of the number is returned after calculations.

Factorial of a positive integer n, denoted by n!, is the
product of all positive integers less than or equal to n.
For example, 5! = 5 X 4 X 3 X 2 X 1.
0! is defined to be 1.
*/

//prototype of the factorial function
```

```c
long factorial(int );

void main(void)
{
    //call the factorial function
    factorial(5);
}

long factorial(int n)
{
    if(n<=1)        //i.e. if the number is less than or equal to 1
        return 1;   //then return 1
    else                            //otherwise factorial will
        return n * factorial(n-1);  //be n times factorial of n-1
}
```

Listing 7.1: Benchmark Application 1: Recursive Factorial Program

7.3.2 Benchmark Application 2: String Copy Program

This benchmark application performs simple string copy operation. *StrCpy()* function is called from *main()*. Source and destination string addresses are passed as arguments to this function. *StrCpy()* does the copy operation and returns back to main. This program will test the conditional branching and data memory access capability. Listing 7.2 is the *C* code of this benchmark.

```c
/*
StringCopy Benchmark Program
C Program implementing string copy For testing, source string is
    initialized to "Super Scalar". the address of source and destination
    strings are passed to StrCpy which will copy the string from source to
     destination
*/

//prototype of the string copy function
void strCopy(char * ,char * );

void main(void)
{
    //initialization of source string
    char *strSrc  = "Super Scalar";

    //destination string
    char strDest[25];


    //now call the copy function
    strCopy(strSrc,strDest);

}

//string copy function
```

```
void StrCpy(char * src,char *dest)
{
    int i=0; //index variable

    while(src[i] != NULL)   //loop until null character is not seen
    {
        dest[i]=src[i];    //copy a character from source to
                              //destination
        i++;               //increment the index
    }
    dest[i] = src[i];     //copy the last character, which is null
                    //character

    return;        //return to calling method (done copying)
}
```

Listing 7.2: Benchmark Application 2: String Copy Program

7.3.3 Benchmark Application 3: Bubble Sort Program

This application program is the famous bubble sort algorithm. An array of 10 numbers is initialized in the *main()* function with the elements in the ascending order. The base address of this array is passed to *BSort()* function. This function sorts the array in descending order. This program will test the performance regarding array handling, conditions and loops. *C* code of this benchmark is given in Listing 7.3.

```
/*
BubbleSort Benchmark Program
Program to sort the array in ascending order. Bubble sort is used as the
    sorting algorithm. Bubble sort, also known as sinking sort, is a simple
     sorting algorithm that works by repeatedly stepping through the list
    to be sorted, comparing each pair of adjacent items and swapping them
    if they are in the wrong order. The pass through the list is repeated
    until no swaps are needed, which indicates that the list is sorted. The
     algorithm gets its name from the way smaller elements "bubble" to the
    top of the list.
*/

//size of the array
#define Arr_Size 10

void BSort(int a[Arr_Size]);

void main(void)
{
    int Array[10];
    int i;

    //fill array with numbers
    for(i=0; i<10; i++)
    {
        Array[i]=i;
```

```
    }

    //call the sorting function
    BSort(Array);

}

//bubble sort function
void BSort(int a[Arr_Size])
{
    int i,j,temp;

        for(i=Arr_Size-2;i>=0;i--)  //Array size is 10, 9 passes
                                     //needed to completely sort
                                    //array
        {
            for(j=0;j<=i;j++)
            {
                if(a[j]<a[j+1])  //if a number is greater than
                                      //its next number
                {
                temp=a[j];  //then swap to bring them in
                                 //descending order
                    a[j]=a[j+1];
                    a[j+1]=temp;
                }
            }//end for j

        }//end for i

}//end function.
```

Listing 7.3: Benchmark Application 3: Bubble Sort Program

7.3.4 Benchmark Application 4: Sensor Structure Program

This application implements a record (known as structure in *C* language) to store the data for sensor values, hence will test structure handling. A structure used to store sensor value contains 3 members:

1. 1 *char* byte Flag indicating if sensor has been calibrated or not.
2. 1 *short* int containing the offset to be adjusted
3. 1 *long int* containing the actual sensor value

An array of five sensor values is declared. *InitSensors()* function initializes these values to some arbitrary numbers. *CalibrateSensors()* function will subtract the offset from the value of the sensors and set the *Flag*. *main()* will call these two functions to initialize and calibrate sensor data. Listing 7.4 provides the *C* code of this benchmark.

```
/*
SensorStruct Benchmark Program
```

```c
C Program implementing a structure for sensor values.Structure contains 3 elements:
1 char byte Flag indicating if sensor has been calibrated or not.
1 short int containing the offset to be adjusted
1 long int containing the actual sensor value
An array of 5 sensors is declared. InitSensors() will initialize these values to some numbers. CalibrateSensors() will subtract
the offset from the value of the sensors and set the Flag. main() will call these two functions to initialize and calibrate sensor data.
*/

// sensor initialization function
void InitSensors();

// sensor calibration function
void CalibrateSensors();

// structure to hold sensor data
typedef struct
{
    char Flag;
    short Offset;
    long Value;
}Sensor;

// array of 5 sensor values
Sensor sensors[5];

void main()
{
    InitSensors();
    CalibrateSensors();
}

// sensor initialization function
void InitSensors()
{
    short i;
    i=0;
    while(i<5)
    {
        sensors[i].Flag = 0;
        sensors[i].Offset = i;
        sensors[i].Value = i+3;
        i++;
    }
}

// sensor calibration function
void CalibrateSensors()
{
    short i=0;
    while(i<5)
    {
        sensors[i].Flag = 1;
```

```
        sensors[i].Value=sensors[i].Value - sensors[i].Offset;
         i++;
    }
}
```

Listing 7.4: Benchmark Application 4: Sensor Structure Program

7.3.5 Benchmark Application 5: Matrix Multiplication Program

This benchmark application performs the matrix multiplication algorithm. In the main function two matrices of order 3 *by* 4 and 4 *by* 5, respectively; are initialized. Later standard matrix multiplication is performed to get the product matrix of order 3 *by* 5. This application will be able to test the capability of the architecture to handle integer math, nested loops with conditions and address calculations for matrix elements. Listing 7.5 is the *C* code of this benchmark.

```
/*
MatrixMul Benchmark Program
Matrix Multiplication is the implementation of multiplication of a
3X4 matrix by 4X5 matrix to get a product 3X5 matrix. Both the matrixes
are initialized with some values. Later actual multiplication is
performed to get the product matrix.
*/

int main(void)
{
    short m, n, p;
    long m1[3][4]; //matrix 1
    long m2[4][5]; //matrix 2
    long m3[3][5]; //product matrix

    //fill the first array with some numbers
     //(m+p values for testing)
    for(m = 0; m < 3; m++)
    {
        for(p = 0; p < 4; p++)
        {
            m1[m][p]=m+p;
        }
    }

    //fill the second array with some numbers
     //(m+p values for testing)
    for(m = 0; m < 4; m++)
    {
        for(p = 0; p < 5; p++)
        {
            m2[m][p]=m+p;
        }
    }

    //perform multiplication
```

```c
    for(m = 0; m < 3; m++)
    {
        for(p = 0; p < 5; p++)
        {
            m3[m][p] = 0;
            for(n = 0; n < 4; n++)
            {
                m3[m][p] += m1[m][n] * m2[n][p];
            }
        }
    }
}
```

Listing 7.5: Benchmark Application 5: Matrix Multiplication Program

7.3.6 Benchmark Application 6: FIR Program

This application is the algorithm of 17th order FIR filter. This algorithm is used to test the math calculations capability of the architecture involved in these types of applications. Similar applications widely implemented on microcontrollers are PID control algorithms. In both types of applications the output is a weighted sum of the current and a finite number of previous values of the input. In this example, input values for the filter is an array of 51 16-bit arbitrary values representing discrete input signal. Calculations are performed and results are stored in the output array representing the discrete output signal. Performance calculations for this benchmark are based on the assumption that all the architectures have floating point hardware as there was huge difference in results because of floating point calculations involved in the program. Listing 7.6 shows the *C* code of this benchmark.

```c
/*
FIR Benchmark Program
The output of a filter is a weighted sum of the current and a finite number of previous values of the input. For testing in this example, input values for the filter is an array of 51 16-bit values. The order of the filter is 17.
*/
void main(void)
{
    int i, y; /* Loop counters */
    float COEFF[17]; //to hold the coefficients of the filter
    int INPUT[67];   //to hold the input (A/D converted values)
    float OUTPUT[36]; //to hold the (filtered) output values
    float sum;        //temporary used for sum

    //fill the coefficient array with some values
    for(i=0;i<17;i++)
    {
        COEFF[i]=1/(i+5.0);
    }
    //fill in the input values
    for(i=0;i<67;i++)
```

```
    {
        INPUT[i]=i;
    }
    //apply filtering
    for(y = 0; y < 36; y++)
        {
        sum=0.0;
        for(i = 0; i < 8; i++)
            {
            sum=sum + COEFF[i]*(INPUT[y+16-i]+INPUT[y + i]);
            }
        OUTPUT[y] = sum + INPUT[y + 8] * COEFF[8];
        }

}
```

Listing 7.6: Benchmark Application 6: FIR Program

7.4 Result Evaluation and Comparison

In this section, performance results are summarized for the above mentioned benchmark programs. Static and dynamic results are tabulated and a comparison ratio of MePoEfAr architecture with selected candidate architectures is also provided. Last rows in all these tables present the mean value of the column. In case of actual values, for instance number of instructions, execution cycles etc; arithmetic mean is calculated. Arithmetic mean of the ratios does not show consistency, as mean of the ratios depends upon the reference architecture. So for the mean of ratios, this number represents the geometric mean of that column to show the overall comparison. The detailed internal calculations performed to get these results are given in Appendix D.

7.4.1 Static Results

Total number of instructions required to implement some functionality by an *architecture* is a measure of capability of instruction set of that architecture. Table 7.1 below shows the total number of instructions required by the four microcontrollers. Figure 7.2 shows this information graphically.

Table 7.1: Number of Instructions Required for Benchmark Programs

#	Benchmark	MePoEfAr	TI	TI/MePoEfAr	ARM	ARM/MePoEfAr	AVR	AVR/MePoEfAr
1	FactRec	14	26	1.86	14	1.00	53	3.79
2	StringCopy	8	9	1.13	10	1.25	11	1.38
3	BubbleSort	20	33	1.65	24	1.20	42	2.10
4	SensorStruct	23	29	1.26	29	1.26	39	1.70
5	MatrixMul	33	56	1.70	43	1.30	105	3.18
6	FIR	45	76	1.69	51	1.13	189	4.20
	Mean	24	38	1.52	29	1.19	73	2.51

As can be seen from the these results, for the simple 8-bit string copy benchmark, all the controllers require same number of instructions. As the complexity of application and number of bits in data types involved in programs increases, this gap increases.

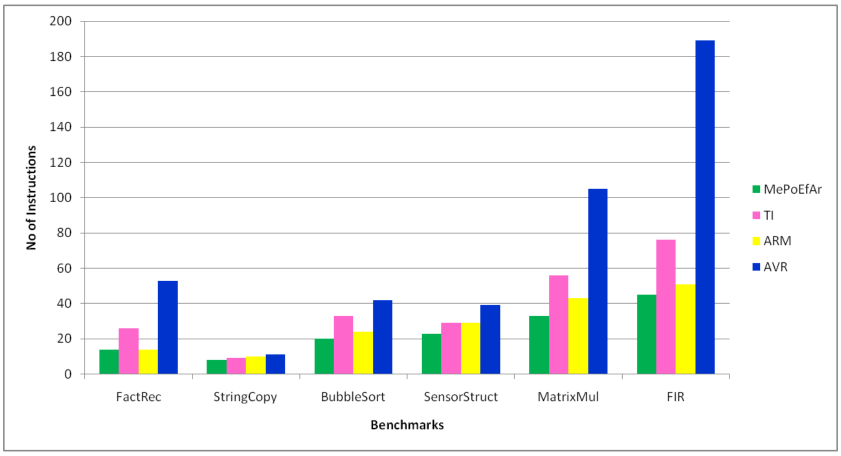


Figure 7.2: Number of Instructions Required for Benchmark Programs

AVR and TI require large number of instructions as most of the calculations involved are for more than 8 and 16 bits. On the other hand, MePoEfAr has several register sets (data types) and same instructions operate on these different register sets. Choice of data types resulted in less number of instructions to implement the same benchmarks in MePoEfAr .

Large number of operations is possible in MePoEfAr due to efficient instruction encoding in spite of being 16-bit architecture. Support of large number of operations implies less number of instructions required in benchmarks to achieve some functionality. As, otherwise, operations need to be emulated with more instructions. For instance, TI suffers because multiply instruction is not a part of instruction set. Although, it has 16-bit hardware multiplier but it is available as memory mapped peripheral, which means multiple instructions for reading from and writing to those registers for multiplication. In case of AVR, though it has multiply instruction, it requires more instructions, because registers are only 8-bit wide. So, more instructions are needed to achieve, for example, 32-bit addition/subtraction.

AVR also requires large number of instructions because of register pressure. This means more data move instructions and memory spills because of register shortage. This is because the data is moved to registers for some operational steps. If in case, before all these steps are complete, more data needs to be fetched which requires even more registers, previously occupied registers are stored in memory (spilled) and fetched back later resulting in extra instructions. This is one of the reasons for large number of instructions required by AVR in FIR benchmark in which 4 byte variables need to be processed in the internal loop.

ARM also performs operations only on registers (load-store architecture). This means instructions are required for moving data to registers before operations can be performed on data. Similarly, store instructions are explicitly needed for storing the results back to memory. This is the reason that, in spite of being 32-bit architecture, it requires, on the average, 19% more instructions as compared to MePoEfAr .

Another reason for reduced number of instructions required by MePoEfAr is the variety

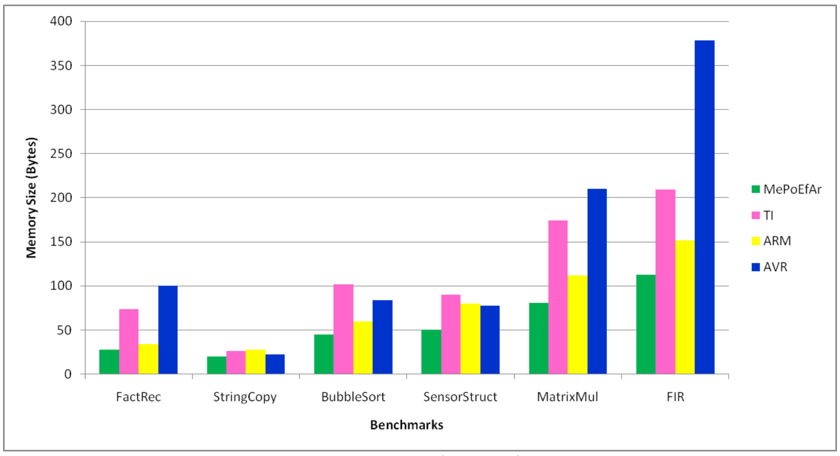


Figure 7.3: Program Memory Size (Bytes) for Selected Benchmarks

of addressing modes possible in the architecture. Lesser instructions are required for address computation as compared to other architectures because of variety of addressing modes available in MePoEfAr . Although TI and AVR also have auto increment and decrement addressing modes, but in orthogonal MePoEfAr , these modes work on all the data types available in the architecture. On the average, for the given benchmarks, ARM and TI require 19% and 52% more instructions than MePoEfAr respectively, while instruction ratio of AVR to MePoEfAr is 2.51.

Memory efficiency of an Instruction set architecture is compared by calculating the total number of bytes of program memory required for each benchmark application. Table 7.2 summarizes the total number of bytes required for all the six benchmark applications by the four microcontroller architectures. These results are graphically plotted in Figure 7.3.

Table 7.2: Program Memory Size (Bytes) for Selected Benchmarks

#	Benchmark	MePoEfAr	TI	TI/MePoEfAr	ARM	ARM/MePoEfAr	AVR	AVR/MePoEfAr
1	FactRec	28	74	2.64	34	1.21	100	3.57
2	StringCopy	20	26	1.30	28	1.40	22	1.10
3	BubbleSort	45	102	2.27	60	1.33	84	1.87
4	SensorStruct	50	90	1.80	80	1.60	78	1.56
5	MatrixMul	81	174	2.15	112	1.38	210	2.59
6	FIR	113	209	1.85	152	1.35	378	3.35
	Mean	56	113	1.95	78	1.37	145	2.15

It can be seen From the results that ARM has better memory efficiency as compared to TI and AVR; but MePoEfAr has a small memory footprint as compared to ARM with an overall difference of 37%. ARM Cortex M3 supports Thumb-2 instruction set which means support for 16-bit instructions; still 32-bit instructions are needed resulting in large memory size.

For string copy program requiring 8-bit operations, AVR is close in memory efficiency to MePoEfAr but for other applications which operate on higher data types, this difference increases. AVR also requires large number of instructions for these applications which directly means more instruction memory requirement. In short, MePoEfAr outperforms AVR by a factor of 2.15.

Memory efficiency of MePoEfAr is mainly because of the variable length instructions resulting in instructions of 1, 2, 3 or 4 bytes depending upon their frequency of occurrence. Another reason for the memory efficiency of MePoEfAr is efficient support for small immediate values and short displacements. In Thumb mode, ARM supports 3-bit immediate values if two registers are specified and 8-bit immediate if a single register operand is specified. TI has reserved two registers namely R2 and R3 as constant generators to generate five most frequent constants -1, 0, 1, 2, 4 and 8. In case of MePoEfAr , 4-bit immediate and 8-bit offsets are accommodated directly inside first instruction word with both operands specified and without reserving any registers.

7.4.2 Dynamic Results

Dynamic results describe the dynamic nature of the architecture. These results highlight the aspects of architecture for the benchmark applications, the way these programs are actually executed on it. For instance, the total number of instructions executed for a given program gives an idea about the total power requirements and the instruction-memory- CPU traffic. Table 7.3 summarizes the total number of instructions executed by each of the architecture for the selected six benchmark applications.

Table 7.3: Total Number of Instructions Executed

#	Benchmark	MePoEfAr	TI	TI/MePoEfAr	ARM	ARM/MePoEfAr	AVR	AVR/MePoEfAr
1	FactRec	42	87	2.07	42	1.00	81	1.93
2	StringCopy	44	57	1.30	58	1.32	59	1.34
3	BubbleSort	371	779	2.10	523	1.41	908	2.45
4	SensorStruct	66	101	1.53	97	1.47	131	1.98
5	MatrixMul	599	1442	2.41	852	1.42	1662	2.77
6	FIR	5229	9331	1.78	6282	1.20	24349	4.66
	Mean	1059	1966	1.83	1309	1.29	4532	2.33

Initializing instructions are executed only once in the application. They may require a large part of program memory. From the execution point of view, the total number of instructions executed inside the loop is important and has major contribution in the execution time and total power consumption. Table 7.4 shows the total number of instructions executed inside the loop.

Table 7.4: Total Number of Instructions Executed inside Loop

#	Benchmark	MePoEfAr	TI	TI/MePoEfAr	ARM	ARM/MePoEfAr	AVR	AVR/MePoEfAr
1	FactRec	39	84	2.15	39	1.00	76	1.95
2	StringCopy	39	52	1.33	52	1.33	52	1.33
3	BubbleSort	363	771	2.12	517	1.42	897	2.47
4	SensorStruct	55	90	1.64	85	1.55	115	2.09
5	MatrixMul	582	1420	2.44	833	1.43	1632	2.80
6	FIR	5218	9324	1.79	6278	1.20	24342	4.67
	Mean	1049	1957	1.87	1301	1.31	4519	2.37

Figure 7.4 and Figure 7.5 show the graph of total number of instructions executed inside the loop for these benchmark programs by the four microcontrollers.

MePoEfAr requires less number of instructions for an application as compared to other architectures which means less number of instructions executed for an application. This is evident from Table 7.4 and graph in Figure 7.5. Because of larger instruction set of

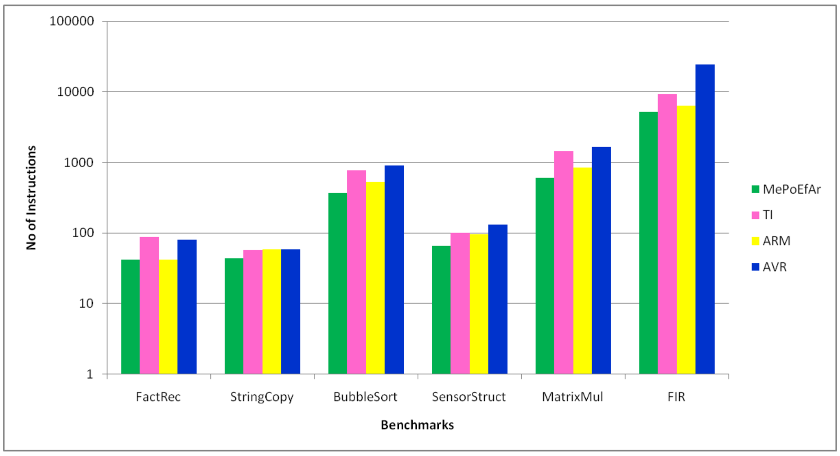


Figure 7.4: Total Number of Instructions Executed

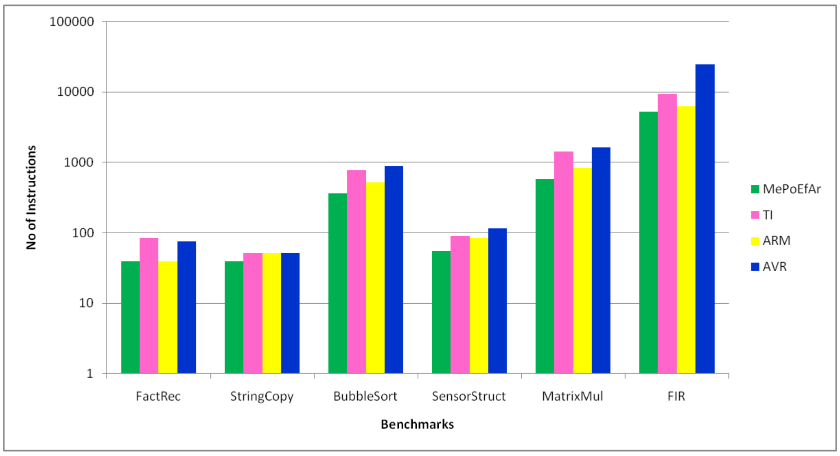


Figure 7.5: Total Number of Instructions Executed inside Loop

MePoEfAr , fewer instructions are required for an application. For example, loop control instruction is a single instruction in MePoEfAr , but for other architectures two or three instructions are required.

On the average for the given benchmarks, TI executes 87% more instructions than MePoEfAr . For operations higher than 16-bits, TI has to perform multiple operations which for instance, can be done with a single instruction in MePoEfAr and ARM. In case of ARM, 31% more instructions are executed as compared to MePoEfAr . In case of AVR, an 8-bit architecture, this requirement is even more and ratio of instructions executed on AVR to MePoEfAr is about 2.37.

Although speed is not the main design consideration, we have also performed a comparison of execution time. In order to compare the architectures based on execution time, we need the information about the number of cycles required for the execution of benchmark programs. For our architecture we have assumed that if the instructions do not require extra operands to be fetched from memory then it is executed in one clock cycle. Otherwise extra cycle is added for each of the extra operand fetched from memory. For arithmetic operations, Table 7.5 gives the number of cycles assumed for

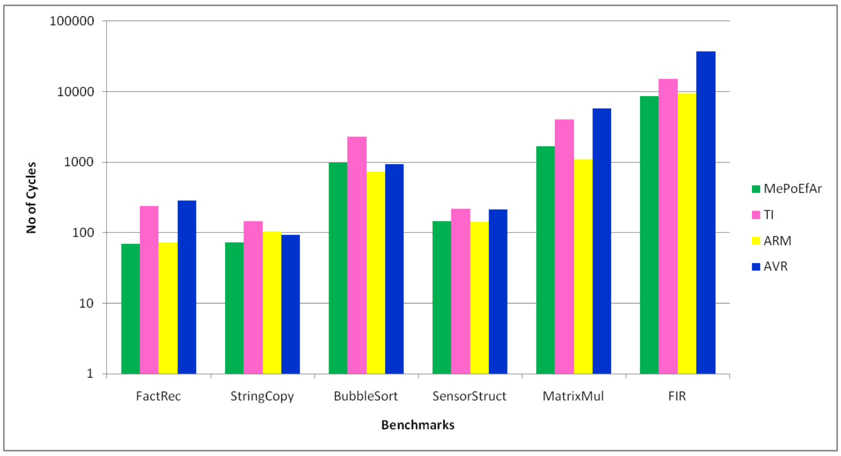


Figure 7.6: Total Number of Execution Cycles

integer and floating point operations for different data types supported by MePoEfAr . 16-bit hardware is assumed in for the numbers in this table. For 24-bit operations, a little more than 1.5 is assumed for the overhead. For floating point operations, the extra cycles have been assumed for the pre- and post-processing involved in these calculations like scaling, normalization, alignment etc.

Table 7.5: Number of Cycles for Arithmetic Operations for Supported Data Types

	Operation								
	ADD/SUB			MUL			DIV		
Data Type	**16**	**24**	**32**	**16**	**24**	**32**	**16**	**24**	**32**
Byte	1	1.6	2	1	1.6	2	1	1.6	2
Word/Index	1	1.6	2	1	1.6	2	1	1.6	2
Double Word	1	1.6	2	2	2.6	3	2	2.6	3
Floating Point	4	4.6	5	6	6.6	7	8	8.6	9

Table 7.6 summarizes the total number of execution cycles consumed by six benchmarks. Figure 7.6 graphically shows the number of execution cycles required by four architectures for selected benchmarks. In order to make the comparison evident for all applications, the vertical axis in this graph is on log scale because of large number of cycles required by AVR especially for FIR application.

Table 7.6: Total Number of Execution Cycles

#	Benchmark	MePoEfAr	TI	TI/MePoEfAr	ARM	ARM/MePoEfAr	AVR	AVR/MePoEfAr
1	FactRec	70	241	3.44	73	1.04	285	4.07
2	StringCopy	73	145	1.99	103	1.41	93	1.27
3	BubbleSort	982	2299	2.34	728	0.74	936	0.95
4	SensorStruct	145	218	1.50	142	0.98	214	1.48
5	MatrixMul	1679	4061	2.42	1087	0.65	5733	3.41
6	FIR	8683	15182	1.75	9502	1.09	37138	4.28
	Mean	1939	3691	2.16	1939	0.95	7400	2.18

In order to make a fair comparison, it is assumed that all the architectures have floating point hardware unit. AVR and ARM executes all instructions in single cycle but TI and MePoEfAr require multiple cycles for different instructions. It can be seen from the results of Table 7.6 and graphs of Figure 7.6 that number of cycles required by TI and AVR are more than two times that of MePoEfAr . This primarily is because of more

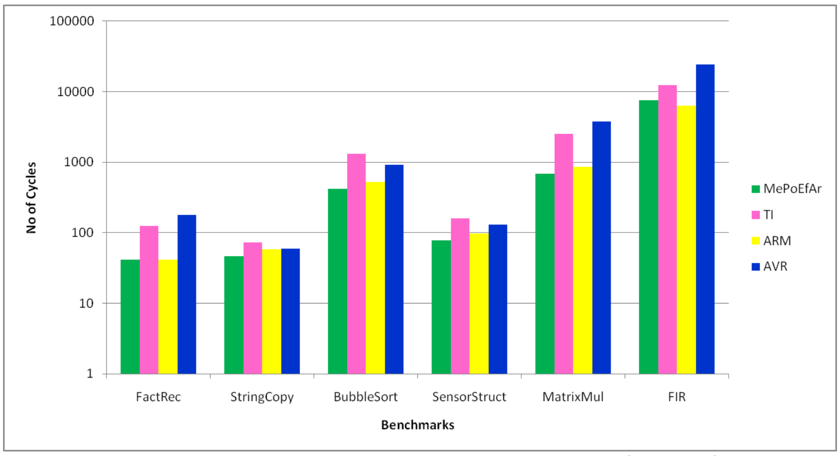


Figure 7.7: Instruction Memory Traffic (Cycles)

number of instructions required for these benchmarks which results in more number of instructions executed. In other words, for MePoEfAr , fewer instructions have to be fetched and processed. Furthermore, TI has von Neumann architecture so it cannot perform instruction and memory accesses in parallel.

In case of ARM, most of the instructions are executed in a single cycle as it is specifically designed for speed. ARM has a modified Harvard architecture, so it has a single address space but physically two memories which in turn facilitates parallel memory accesses. But, ARM is a load-store architecture, it needs instructions to load data from memory to perform operations on this data. Similarly, when the results need to be written to memory, store instructions are explicitly required. This means more instruction fetches and decodes. On the average for all the benchmarks, ARM requires 5% less execution cycles as compared to MePoEfAr .

Memory access consumes considerable amount of power. Table 7.7 below summarizes the instruction memory traffic in cycles and Figure 7.7 shows the same information graphically. Vertical axis in this graph is on log scale.

Table 7.7: Instruction Memory Traffic (Cycles)

#	Benchmark	MePoEfAr	TI	TI/MePoEfAr	ARM	ARM/MePoEfAr	AVR	AVR/MePoEfAr
1	FactRec	42	125	2.98	42	1.00	177	4.21
2	StringCopy	46	73	1.59	58	1.26	59	1.28
3	BubbleSort	418	1308	3.13	523	1.25	908	2.17
4	SensorStruct	78	161	2.06	97	1.24	131	1.68
5	MatrixMul	685	2499	3.65	852	1.24	3762	5.49
6	FIR	7473	12436	1.66	6282	0.84	24349	3.26
	Mean	1457	2767	2.39	1309	1.13	4898	2.66

ARM has 16- or 32-bit instruction which implies a single instruction memory cycle to fetch an instruction. But, ARM requires more number of instructions on the average, so instruction memory cycles consumed by ARM are 13% more than MePoEfAr on the average as can be seen from Table 7.7.

AVR also executes instructions in a single cycle but it requires higher number of instructions than MePoEfAr which directly translates to higher instruction memory traffic by a factor of 2.66.

In case of TI, as the number of instructions fetched from the memory is large, and on top of it, most of the instructions require multiple cycles. This results in higher instruction memory traffic by a factor of 2.39 as compared to MePoEfAr .

In order to compare data memory traffic, data memory cycles are also computed. In order to have a fair comparison, same width of MePoEfAr is assumed as used by the architecture in consideration. This mean 16-bit path from data memory is considered when comparing with TI and AVR, whereas, 32-bit bus width is assumed for the comparison with ARM.

Table 7.8 summarizes the data memory traffic in cycles. Although, same input data is processed still there is a variation in data memory cycles for some benchmarks.

Table 7.8: Data Memory Traffic (Cycles)

#	Benchmark	MePoEfAr		TI	TI/MePoEfAr	ARM	ARM/MePoEfAr	AVR	AVR/MePoEfAr
		16-bit	32-bit						
1	FactRec	10	10	10	1.00	10	1.00	20	2.00
2	StringCopy	26	26	26	1.00	26	1.00	26	1.00
3	BubbleSort	380	190	380	1.00	190	1.00	380	1.00
4	SensorStruct	50	35	50	1.00	35	1.00	90	1.80
5	MatrixMul	334	167	424	1.27	167	1.00	908	2.72
6	FIR	1536	1106	1536	1.00	1106	1.00	10600	6.90
	Mean	389	256	404	1.04	256	1.00	2004	2.02
	Mean	388	255	403	1.04	256	1.03	2004	2.09

An interesting point worth mentioning is that, though MePoEfAr and TI can access 16-bits data in single cycle still in case of *MatrixMul* TI requires 27% more data memory cycles as compared to MePoEfAr . This is because one element of matrix is needed twice as multiplier is 16-bit wide. Furthermore, inner loop needs register to calculate addresses of matrix elements as well as for the actual multiplication of elements. Programs are optimized considering instruction memory as first goal. So if we place this data in registers once, and for later operations, then data memory cycles will become same but program memory size and number of instructions executed will be adversely affected. But, in case of MePoEfAr , availability of large number of registers of different sizes facilitates storage of intermediate results in registers and operations are possible on these registers. This results in reduced data memory access even for complex applications.

For AVR, in case of *StringCopy* benchmark, data memory cycles are same as required by other architectures, but for other benchmarks it requires far more data memory cycles. Furthermore, Registers are 8-bits wide so, multiple registers required for operations because of which limited data can be kept in registers. Especially in case of *FIR* application, data needs to be stored back to memory because of unavailability of registers, and fetched back later (spills) which caused considerable data memory traffic.

7.5 Summary

In order to have the overall impression of the architectures under discussion, all the results discussed above are summarized in the Table 7.9. These numbers are ratios and mean of all the ratios is also given at the bottom of table to show the overall comparison.

Table 7.9: Performance Comparison Summary

#	Benchmark	TI/MePoEfAr	ARM/MePoEfAr	AVR/MePoEfAr
1	No of Instructions	1.52	1.19	2.51
2	Program Size	1.95	1.37	2.15
3	Instructions Executed	1.83	1.29	2.33
4	Instructions Executed in Loop	1.87	1.31	2.37
5	Execution Cycles	2.16	0.95	2.18
6	Instruction Memory Traffic	2.39	1.13	2.66
7	Data Memory Traffic	1.04	1.03	2.09
	Mean	1.77	1.17	2.32

In summary it can be concluded from the above table that MePoEfAr architecture has better performance in all respects as compared to TI architecture. Overall MePoEfAr architecture performance is 77% better than TI microcontroller.

MePoEfAr is better than ARM in most of the cases, while being same for data memory cycles. ARM has winning situation based on the execution cycles. This gain is because of the instructions to calculate array address in single cycle by a single instruction which utilizes the shifter. This can be seen from the bubble sort and matrix multiplication benchmark results for execution cycles. ARM is a 32-bit architecture and can represent these type of instructions. Overall MePoEfAr outperforms ARM by 17%.

There is a considerable difference in performance results of AVR as compared to other architectures in all respects. On the average for the given benchmarks, MePoEfAr performance is better than AVR by a factor of 2.31.

8 Conclusion and Future Work

This chapter starts with a brief summary of the whole thesis in Section 8.1. We highlight the conclusions of this work in Section 8.2. Finally, Section 8.3 provides some recommendations for future work.

8.1 Summary

This section gives a brief summary of the work presented in this thesis. We provide short description of each chapter as follows:

Chapter 1 provided an introduction to the work presented in this thesis. It discussed the key motivation behind the thesis and enlisted the main contributions of this work.

Chapter 2 presented an overview of microcontroller architectures and their classification, which are based on several criteria. Three well-known embedded microcontroller architectures were discussed in detail, which were used for the performance comparison.

Chapter 3 discussed the static profiling. The statistics of high level language constructs obtained from the developed profiler were provided. These statistics show the frequency distributions of the *C* language constructs in four benchmark programs.

Chapter 4 provided the details of MePoEfAr architecture. It started with overall architecture properties, type of architecture, bit and byte numbering, data types, instruction classification and register sets. Global architecture issues such as layout of the program status word and Memory Map were provided. Various instruction formats in MePoEfAr architecture with examples were detailed. Furthermore, operation sets supported by these instruction formats were also tabulated with a description on how these operations affect the condition codes. A brief description of exceptional conditions like traps and interrupt vectors were provided followed by a discussion of extension of program and data Memory. The summary of encoding cost and feasibility of MePoEfAr architecture were discussed, in order to show the availability of the encoding space in the architecture, for future extensions.

Chapter 5 gave the implementation details of MePoEfAr assembler. It covered the details of the intermediate steps involved to translate the assembly program to machine code. Instruction bit assignments were provided which we used to represent assembly instructions as bit patterns.

Chapter 6 discussed MePoEfAr interpreter which has been used for the simulation of the MePoEfAr microcontroller. It discussed the two main parts of MePoEfAr interpreter. First part loads the machine code to memory and performs some book keeping for debugging information. Second part is the microcontroller model which fetches the instructions from memory, decodes and executes them.

Chapter 7 covered the assembler level benchmarking details, which we performed to evaluate the performance of MePoEfAr architecture. Furthermore, it provided the results of static and dynamic comparison of performance with three well known embedded microcontrollers.

This chapter, that is Chapter 8, summarizes the thesis. Conclusions drawn from our work are provided followed by some recommendations for future work.

8.2 Conclusions

Conclusions drawn based on the work presented in this thesis are enumerated below. For the sake of brevity, in rest of the chapter, TI, ARM and AVR refers to Texas Instruments MSP430G2231, ARM Cortex-M3 LPC1342 and Atmel AVR AT90S851 microcontrollers respectively.

- Statistics presented in this thesis show that frequency distribution of *C* language constructs (statements, operations, operands etc.) do not have a uniform distribution over the complete range. Furthermore, a single architecture cannot satisfy the demands of all the applications, so intelligent trade-offs must be made in favor of the most frequent constructs. Conclusions drawn from the static analysis of the benchmarks programs are:
    1. Assignments are the most frequent statements. About 60% of the statments are assignments.
    2. 73% of assignments have a simple variable on the left hand side of assignments.
    3. Most of the assignments have a simple expression on the right hand side. About 55% of assignments have either a constant or a simple variable on right hand side.
    4. Arithmetic operations are the most frequent operations. Among arithmetic operations, addition and multiplication are the most frequent operations.
    5. After relational operations, type conversion operations are also frequent. Most of the conversions are between 16 and 32 bit integer data types.
    6. Based on data type, 32-bit operations are the most frequent operations.
    7. Small constants are the most frequent ones. 4-bit constants have an accumulative frequency of about 87%. 0, 1, 2, 4 and 8 are the most frequent constants.
    8. Among local variables, 32-bit integers, 16-bit integers, pointers and 8-bit integers have a frequency distribution of about 61%, 13%, 12%, and 5%, respectively.
- The results of assembler level benchmarking show that MePoEfAr architecture is 77% and 17% better than TI and ARM, respectively. Furthermore, MePoEfAr outperforms AVR by a factor of 2.31. Following conclusions can be drawn from the detailed analysis of these results:
    1. *Number of instructions* required to implement some functionality by an architecture is a measure of capability of instruction set of that architecture. On average, for the given benchmarks, ARM and TI require 19% and 52% more

instructions than MePoEfAr respectively. Furthermore, the instruction ratio of AVR to MePoEfAr is 2.51. The effeciency of MePoEfAr compared to other architectures is because of the follwoing reasons:

(a) Operations normally involve 16 and 32-bit data types and AVR and MSP430 need multiple instructions for these operations whereas MePoEfAr has 8, 16 and 32-bit data types.
(b) Availability of large number of operations in MePoEfAr architecture as compared to other architectures, requires no emulation of these operations by extra instructions.
(c) ARM is a load store architecture, which required instructions to load data in registers, perform operations and later instructions to store the results back to memory.
(d) AVR and TI have auto-increment and auto-decrement addressing modes, requiring less number of instructions for address computations. But, in MePoEfAr, these modes work on all the data types available in the architecture.

2. *Memory efficiency* of an Instruction set architecture is compared by calculating the total number of bytes of program memory require for each benchmark application. MePoEfAr is 37%, 95% and 115% more memory effecient than ARM, TI and AVR, respectively. This memory effeciency is achieved as follows:
   (a) Although ARM supports thumb-2 instruction set, which means the support for 16-bit instructions in addition to the 32-bit instructions. Despite of these 16-bit instructions, 32-bit instructions are alse needed in these benchmarks increasing the program memory size.
   (b) Variable length instructions in the MePoEfAr architecture has proven to be more memory efficient. Frequently occurring instructions are short 2-byte instructions. On the other hand 3 to 4 byte instructions are not very frequent.
   (c) MePoEfAr provides efficient support for small immediate values and short displacements. In thumb mode, ARM supports 3-bit immediate values if two registers are specified and 8-bit immediate if a single register operand is specified. TI has reserved two registers namely R2 and R3 as constant generators to generate frequent constants (0, 1, 2, 4 and 8). In case of MePoEfAr , 4-bit immediate values and 8-bit offsets are accommodated directly inside the first instruction word, with both operands specified and without reserving any registers.
3. *Instructions executed* for a given program give an idea about the total power requirements and the instruction- memory- CPU traffic. TI and ARM executes 87% and 31% more instructions than MePoEfAr. Furthermore, ratio of instructions executed on AVR to MePoEfAr is about 2.37. This efficiency of MePoEfAr is due to the following reasons:
   (a) Larger instruction set of MePoEfAr resulted in fewer instructions for an application. For example, loop control instruction is a single instruction in MePoEfAr, but for other architectures two or three instructions are

required.

(b) For operations higher than 16-bits, TI and AVR perform operations with multiple instructions which can be performed with a single instruction in MePoEfAr.

(c) TI, AVR and ARM require a large number of instructions, which result in large number of instructions executed by these architectures.

4. *Execution cycles* required by TI and AVR are more than two times as compared to MePoEfAr. This is because of more number of instructions required for these benchmarks which resulted in more number of instructions executed. In other words, for MePoEfAr , fewer instructions have to be fetched and processed. Furthermore, TI has von Neumann architecture so it cannot perform instruction and memory accesses in parallel.
5. *Instruction memory accesses* consume power. ARM required 13% more instruction cycles as compared to MePoEfAr. TI and AVR required higher instruction memory cycles by a factor of 2.39 and 2.66, respectively. More number of instructions required by these architecture result in higher instruction memory traffic.
6. In case of *data memory traffic*, TI, ARM and MePoEfAr require almost same number of cycles. In case of AVR, registers are 8-bit wide. So multiple registers are required for operations which results in limited data to be kept in registers. Due to register spills, data must be stored back to memory because of unavailability of registers, and fetched back later, resulting in increased data memory traffic.

8.3 Future Work

Some recommendations for the future work are enlisted as follows:

1. In this work, we have performed static profiling analysis to obtain the frequency distributions of various *C* language constructs. Static results are important for the design of a memory efficient architecture. In contrast to static analysis, dynamic profiling is performed during the program execution. The results of dynamic profiling are also important, as they point out the most frequently executed constructs in the benchmarks. Hence, there is a need of dynamic profiling, which can be given the second priority in making design decisions and to fine tune the architecture.
2. Cost of 8-bit instructions (in the units of 1024) is 216. This is 21% of the total encoding space available. From the results of static analysis, 8-bit data type is not so frequent. Hence, further analysis is required to probably remove the support of this data type and use this encoding space to make the architecture more efficient.
3. The variable length instructions used in MePoEfAr architecture proved to be more memory efficient. This efficiency has its cost in terms of complex decoding logic required by instructions. Further work is required to synthesize the decoding logic to obtain some numbers for the area overhead introduced by this decoding logic.
4. The interpretive simulator which we have developed, does not incorporate the information about the number of cycles consumed by individual instructions and

the overall execution cycles of the complete benchmark. Hence, there is a need to add the information about the execution cycles to make it a cycle accurate simulator, or to perform an RTL simulation (VHDL simulation). This will help in running larger benchmarks and will save the time consumed in performing the calculations for comparison manually.

5. Another important property of an instruction set architecture is its support for compilers. Hence high level language compiler is required to further ease the benchmarking process. Furthermore, results from the compiler writing process can prove to be another important feedback for the architecture.

Bibliography

[1] *http://flex.sourceforge.net/*.

[2] *http://focus.ti.com/*.

[3] *http://focus.ti.com/docs/prod/folders/print/msp430g2231.html*.

[4] *http://focus.ti.com/general/docs/lit/getliterature.tsp?literatureNumber=slau144h&fileType=pdf*.

[5] *http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0337i/index.html*.

[6] *http://www.ace.nl/compiler/cosy.html*.

[7] *http://www.coremark.org/home.php*.

[8] *http://www.design-reuse.com/articles/21745/interpretive-instruction-set-simulator.html*.

[9] *http://www.eembc.org/benchmark/automotive_sl.php*.

[10] *http://www.eembc.org/home.php*.

[11] *http://www.gnu.org/software/bison/*.

[12] *http://www.nxp.com/*.

[13] *http://www.nxp.com/#/pip/pip=[pip=LPC1311_13_42_43,pfp=71567]|pp=[t=pip,i=LPC1311_13_42_43]*.

[14] *www.atmel.com*.

[15] *www.atmel.com/dyn/resources/prod_documents/DOC0841.pdf*.

[16] *Dhrystone benchmark: Rationale for version 2 and measurement rules*, SIGPLAN Notices **23** (1988), 49–62.

[17] *Microchip industry research*, 2011.

[18] John Backus, *Can programming be liberated from the von neumann style?: a functional style and its algebra of programs*, Commun. ACM **21** (1978), 613–641.

[19] R. Bannatyne and G. Viot, *Introduction to microcontrollers. i*, Wescon/98, sep 1998, pp. 350 –360.

[20] K.L.M. Bertels, S. A. Ostadzadeh, and R. J. Meeuws, *Advanced profiling of applications for heterogeneous multi-core platforms*, July 2011, p. 13.

[21] Koen Bertels, Stamatis Vassiliadis, Elena Moscu Panainte, Yana Yankova, Carlo Galuzzi, Ricardo Chaves, and Georgi Kuzmanov, *Developing applications for polymorphic processors: The delft workbench*, 2006.

[22] Robert F. Cmelik and David Keppel, *Shade: A fast instruction set simulator for execution profiling*, Tech. report, Mountain View, CA, USA, 1993.

[23] B A Wichmann H. J. Curnow, *A synthetic benchmark*, Computer Journal **19** (1976), 1.

[24] John L. Hennessy and David A. Patterson, *Computer architecture: A quantitative approach, 3rd edition*, Computer Architecture: A Quantitative Approach, 3rd Edition, Morgan Kaufmann, 3rd Edition, May 2002.

[25] K.-D. Kramer, T. Stolze, and T. Banse, *Benchmarks to find the optimal microcontroller-architecture*, Computer Science and Information Engineering, 2009 WRI World Congress on, vol. 2, 31 2009-april 2 2009, pp. 102 –105.

[26] R. Leupers, J. Elste, and B. Landwehr, *Generation of interpretive and compiled instruction set simulators*, Design Automation Conference, 1999. Proceedings of the ASP-DAC '99. Asia and South Pacific, jan 1999, pp. 339 –342 vol.1.

[27] Mingsong Lv, Qingxu Deng, Nan Guan, Yaming Xie, and Ge Yu, *Armiss: An instruction set simulator for the arm architecture*, Embedded Software and Systems, 2008. ICESS '08. International Conference on, july 2008, pp. 548 –555.

[28] Christopher Mills, Stanley C. Ahalt, and Jim Fowler, *Compiled instruction set simulation*, 1991.

[29] Achim Nohl, Gunnar Braun, Oliver Schliebusch, Rainer Leupers, Heinrich Meyr, and Andreas Hoffmann, *A universal technique for fast and flexible instruction-set architecture simulation*, Proceedings of the 39th annual Design Automation Conference (New York, NY, USA), DAC '02, ACM, 2002, pp. 22–27.

[30] David A. Patterson and David R. Ditzel, *The case for the reduced instruction set computer*, SIGARCH Comput. Archit. News **8** (1980), 25–33.

[31] David A. Patterson and Carlo H. Sequin, *Risc i: A reduced instruction set vlsi computer*, 25 years of the international symposia on Computer architecture (selected papers) (New York, NY, USA), ISCA '98, ACM, 1998, pp. 216–230.

[32] H. Meyr V. Zivojnovic, S. Tjiang, *Compiled simulation of programmable dsp architectures*, IEEE Workshop on VLSI Signal Processing (1995).

[33] John von Neumann, *First draft of a report on the evdac, charles babbage institute reprint series for the history of computing, mit press*, vol. 12, 1987.

[34] Reinhold P. Weicker, *An overview of common benchmarks*, Computer **23** (1990), 65–75.

Appendix A: Lexical Analyzer Generator Code

```
%option nounput

%{
#include "globals.h"
#include "grammar.tab.h"     /* import token definitions from Yacc */
#include <stdio.h>
#include <string.h>

void cstyle_comment();       /* Multiline C-Style comment */
int lineno = 1;
%}
digit        [0-9]
hdigit       [a-fA-F0-9]
odigit       [0-7]
letter       [a-zA-Z]
newline      \n
whitespace   [ \t]+
hash         "#"
atrate       "@"
valid_char   {letter}|"_"|"."|{digit}
symbol       ({letter}|"_"|".")){valid_char}*
label        {symbol}":"
comment      ";".*\n
hnumber      0[xX]{hdigit}+
bnumber      0[bB][01]+
onumber      0[oO]{odigit}+
dnumber      {digit}+
fnumber      {digit}+\.{digit}+
breg         B{digit}+
wreg         W{digit}+
dreg         D{digit}+
freg         F{digit}+
xreg         X{digit}+
norm_char    '(\\.|\\\\n|[^\\])[']?
character    {norm_char}
string       \"(\\.|\\\\n|[^\n\\"])*\"
%%
"/*"             {cstyle_comment();}
{comment}        {lineno++; return NEWLINE;}
{breg}           {return BREGISTER;}
{wreg}           {return WREGISTER;}
{dreg}           {return DREGISTER;}
{freg}           {return FREGISTER;}
{xreg}           {return XREGISTER;}
```

```
{symbol}          {return SYMBOL;}
{label}           {return LABEL;}
{bnumber}         {return BNUMBER;}
{onumber}         {return ONUMBER;}
{dnumber}         {return DNUMBER;}
{hnumber}         {return HNUMBER;}
{fnumber}         {return FNUMBER;}
{character}       {return CHARACTER;}
{string}          {return STRING;}
{hash}            {return HASH;}
{atrate}          {return ATRATE;}
","               {return COMMA;}
";"               {return SEMI_COLON;}
":"               {return COLON;}
"+"               {return PLUS;}
"-"               {return MINUS;}
"*"               {return MULTIPLY;}
"/"               {return DIVIDE;}
"("               {return LBRACK;}
")"               {return RBRACK;}
{newline}         {lineno++; return NEWLINE;}
{whitespace}      {/* ignore whitespaces */}
%%
void cstyle_comment()
{
    char c;
    int done = FALSE;
    char * text = yytext + 2;
    int i = 0;

    while(!done)
    {
        while ((c = input()) != '*')
        {
            if ( c == EOF )
                return;
            text[i++] = c;
            if ( c == '\n' )
                lineno++;
        }
        text[i++] = c;
        while ((c = input()) == '*')
        {
            if ( c == EOF )
                return;
            text[i++] = c;
        }
        text[i++] = c;
        if ( c == '\n' )
            lineno++;
        if (c == '/')
        {
            done = TRUE;
            text[i] = '\0';
        }
```

```
    }
}

#ifdef _MSC_VER
int  yywrap( )
{
    return 1;
/*hex_char     '(\\)(x){hdigit}{hdigit}
oct_char     '(\\){odigit}{odigit}{odigit}
character    {hex_char}|{oct_char}|{norm_char}*/


}
#endif
```

Listing A.1: Flex Code for the Lexical Analyzer Generator for MePoEfAr Assembler

B Parser Generator Code

```
%{
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include "ast.h"
#include "globals.h"

#define YYSTYPE TreeNode *

TreeNode * AST;
extern int lineno;

TreeNode *tmpNodes[NUM_OF_CHILDREN] = {NULL};    // Temperory Nodes
int tni = 0;                // Temperory Nodes index
char tempStr[20] = "-" ;

extern char * yytext;        // Token string from Scanner
extern int yylex();
void yyerror( const char * msg );

#define ADD_TO_LIST(ss, s1, s2)      \
{                    \
    YYSTYPE t = s1;          \
    if ( t == NULL )         \
        ss = s2;         \
    else                     \
    {                    \
        while( t->next != NULL )\
            t = t->next;     \
        t->next = s2;        \
        ss = s1;         \
    }                    \
}
%}

%token BREGISTER WREGISTER DREGISTER FREGISTER XREGISTER
%token HASH ATRATE SYMBOL LABEL BNUMBER ONUMBER DNUMBER HNUMBER FNUMBER CHARACTER STRING
%token COMMA SEMI_COLON PLUS MINUS MULTIPLY DIVIDE
%token LBRACK RBRACK NEWLINE COLON

%left PLUS MINUS
%left MULTIPLY DIVIDE
```

```

%%
program     : stmt_seq { AST = $1;}
            ;

stmt_seq    : stmt_seq stmt { ADD_TO_LIST( $$, $1, $2 ) }
            | stmt { $$ = $1; }
            ;
stmt        : statement NEWLINE { $$ = $1; }
            | statement SEMI_COLON { $$ = $1; }
            | NEWLINE
            | SEMI_COLON
            ;
statement   : labels operation
                {
                    YYSTYPE t = $1;

                    while( t->child[0] != NULL )
                        t = t->child[0];
                    $1->next = $2;
                    $$ = $1;
                }
            | operation { $$ = $1; }
            | labels { $$ = $1; }
            ;
labels      : labels label
                {
                    YYSTYPE t = $1;
                    if ( t == NULL )
                        $$ = $2;
                    else
                    {
                        while( t->child[0] != NULL )
                            t = t->child[0];
                        t->child[0] = $2;
                        $$ = $1;
                    }
                }
            | label { $$ = $1; }
            ;
operation   : instruction { $$ = $1; }
            | label num_const { ($$ = $1)-> nodeType = NT_DIRECTIVE; ($$->child[0]) = ($2);}
            ;
instruction : symbol operands
                {
                    int i;

                    ($$ = $1)->nodeType = NT_INSTRUCTION;
                    ($$)->SG =SG_NA; //SG not applicable or not yet specified
                    ($$)->op = OP_INVALID;  //not valid or not yet specified
                    ($$)->instrSize = -1; //not yet specified so represented by -1
```

```
                        for( i = 0; i < tni && i < NUM_OF_CHILDREN; i++ )
                            $$->child[i] = tmpNodes[i];
                        tni = 0;
                    }
                | symbol {
                        ($$ = $1)->nodeType = NT_INSTRUCTION;
                        ($$)->SG =SG_NA; //SG not applicable or not yet specified
                        ($$)->op = OP_INVALID;
                        ($$)->instrSize = -1;
                    }
                ;
operands        : operands COMMA operand
                    {
                        if ( tni < NUM_OF_CHILDREN )
                            tmpNodes[tni++] = $3;
                    }
                | operand { tmpNodes[tni++] = $1; }
                ;
operand         : reg { $$ = $1; }
                | imm { $$ = $1; }
                | mem_ref { $$ = $1; }
                | symbol { $$ = $1; }
                ;
mem_ref         : DX_addr { $$ = $1; }
                | Ptr_addr { $$ = $1; }
                | SMptr_addr { $$ = $1; }
                | Abs_addr { $$ = $1; }
                ;
DX_addr         : num_const LBRACK reg RBRACK               /* D(X) Addressing */
                    {
                        $$ = newTreeNode( NT_DX_ADDR, NULL );
                        $$->child[0] = $1;
                        $$->child[1] = $3;
                    }
                | symbol LBRACK reg RBRACK                  /* D(X) Addressing */
                    {
                        $$ = newTreeNode( NT_DX_ADDR, NULL );
                        $$->child[0] = $1;
                        $$->child[1] = $3;
                    }
                ;
Ptr_addr        : ATRATE reg                                /* Pointer Addressing */
                    {
                        ($$ = $2)->nodeType = NT_PTR_ADDR;
                    }
                ;
SMptr_addr      : LBRACK reg RBRACK PLUS                    /* Self Modifying ptr Addressing */
                    {                               /*Post increment (X)+       */
                        ($$ = $2)->nodeType = NT_SMPTR_POST_INC_ADDR;
                    }
```

```
              | MINUS LBRACK reg RBRACK                    /*Pre decrement -(X)       */
                {
                    ($$ = $3)->nodeType = NT_SMPTR_PRE_DEC_ADDR;
                }
              ;
Abs_addr      : ATRATE num_const { $$ = $2; ($$->nodeType) = NT_ABSADDRIMM;}
              | ATRATE symbol { $$ = $2; ($$->nodeType) = NT_ABSADDRID;}
              ;
imm           : HASH num_const { $$ = $2; ($$->nodeType) = NT_IMM;}
              | HASH symbol { $$ = $2; ($$->nodeType) = NT_ID;}
              ;
symbol        : SYMBOL { $$ = newTreeNode( NT_ID, yytext ); }
              ;
label         : LABEL { $$ = newTreeNode( NT_LABEL, yytext); }
              ;
reg           : BREGISTER { $$ = newTreeNode( NT_BREGISTER, yytext ); /* Changed yytext+1 to yytext */ }
              | WREGISTER { $$ = newTreeNode( NT_WREGISTER, yytext ); /* Changed yytext+1 to yytext */ }
              | DREGISTER { $$ = newTreeNode( NT_DREGISTER, yytext ); /* Changed yytext+1 to yytext */ }
              | FREGISTER { $$ = newTreeNode( NT_FREGISTER, yytext ); /* Changed yytext+1 to yytext */ }
              | XREGISTER { $$ = newTreeNode( NT_XREGISTER, yytext ); /* Changed yytext+1 to yytext */ }
              ;
num_const     : BNUMBER { $$ = newTreeNode( NT_CONSTANT, yytext ); ($$->value) = other2dec($$->name,2);}
              | MINUS BNUMBER
                {
                    $$ = newTreeNode( NT_CONSTANT, yytext );
                    ($$->value) = -1*other2dec($$->name,2);
                    strcat(tempStr,$$->name);
                    tempStr[strlen(tempStr)]='\0';
                    strcpy($$->name,tempStr);
                    strcpy(tempStr,"-");
                }
              | ONUMBER { $$ = newTreeNode( NT_CONSTANT, yytext ); ($$->value) = other2dec($$->name,8);}
              | MINUS ONUMBER
                {
                    $$ = newTreeNode( NT_CONSTANT, yytext );
                    ($$->value) = -1*other2dec($$->name,8);
                    strcat(tempStr,$$->name);
                    tempStr[strlen(tempStr)]='\0';
                    strcpy($$->name,tempStr);
                    strcpy(tempStr,"-");
                }
              | DNUMBER { $$ = newTreeNode( NT_CONSTANT, yytext ); ($$->value) = atoi($$->name);}
              | MINUS DNUMBER
                {
                    $$ = newTreeNode( NT_CONSTANT, yytext );
                    ($$->value) = -1*atoi($$->name);
```

```
                        strcat(tempStr,$$->name);
                        tempStr[strlen(tempStr)]='\0';
                        strcpy($$->name,tempStr);
                        strcpy(tempStr,"-");
                    }
                | HNUMBER { $$ = newTreeNode( NT_CONSTANT, yytext ); ($$->value) = other2dec($$->name,16);}
                | MINUS HNUMBER
                    {
                        $$ = newTreeNode( NT_CONSTANT, yytext );
                        ($$->value) = -1*other2dec($$->name,16);
                        strcat(tempStr,$$->name);
                        tempStr[strlen(tempStr)]='\0';
                        strcpy($$->name,tempStr);
                        strcpy(tempStr,"-");
                    }
                | FNUMBER { $$ = newTreeNode( NT_CONSTANT, yytext ); ($$->value) = strtofp($$->name);}
                | MINUS FNUMBER
                    {
                        $$ = newTreeNode( NT_CONSTANT, yytext );
                        ($$->value) = -1*strtofp($$->name);
                        strcat(tempStr,$$->name);
                        tempStr[strlen(tempStr)]='\0';
                        strcpy($$->name,tempStr);
                        strcpy(tempStr,"-");
                    }

                ;
%%

void yyerror( const char * msg )
{
    printf("%s on line %d\n", msg, lineno );
    exit(1);
//| CHARACTER { $$ = newTreeNode( NT_CONSTANT, yytext ); }
/*
directive       : num_const { $$ = newTreeNode( NT_DIRECTIVE, yytext); }
                ;

*/

}
```

Listing B.1: Bison Code for the Parser Generator for MePoEfAr Assembler

Assembly Codes for the Selected Benchmarks

C

C.1 MePoEfAr Assembly Codes

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   MePoEfAr Recursive Factorial Benchmark Program
;   This program recursively calculates the factorial
;   of a number (n). A number is passed to this subroutine
;   by main for factorial calculation.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; main subroutine
;
;   D0  -> number for which factorial needs to be calculated
;   D1  -> to hold the result i.e. calculated factorial
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Main:   MOVd    #5, D0   ;D0 = 5 i.e. n = 5, number for factorial calculation
        BRS     Fact     ;call to fact subroutine
End:    HALT             ;end

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   factorial subroutine.
;   D0 contanins number 'n' for which the factorial needs to be calculated
;   D1  -> to hold the result i.e. calculated factorial
;   D2  -> a temporary for internal calculations
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Fact:   MOVd    D0,-(SP);save D0 on stack as it will be modified
        MOVd    D0,D2    ;D2 = n
        SUBd    #1, D0   ;D0 = n-1
        BRgt    L0       ;branch to L0 if(D0 > 0)
        MOVd    #1,D1    ;otherwise D1 = 1, which is factorial of 1 (result)
        MOVd    (SP)+,D0;restore the register pushed earlier on stack
        RTS              ;return to caller

        ;program will come here if n <= 1
L0:     BRS     Fact     ;call subroutine Fact for n-1
        MULd    D2,D1    ;perform multiplication i.e. n * fact (n - 1)
        MOVd    (SP)+,D0;restore the register pushed earlier on stack
        RTS              ;return to caller
```

Listing C.1: MePoEfAr Assembly Code for Benchmark 1: Recursive Factorial

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   MePoEfAr String Copy Benchmark Program
;   In this program, main subroutine passes the addresses of source and
;   destination strings to the StrCpy subroutine to copy the chracters
;   from source to the destination string.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Main Subroutine
;
;   X4  -> Source String index
;   X5  -> Destination String index
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Main:   MOVx    #Src,X4 ;Address of source string in the memory
        MOVx    #Dst,X5 ;Address of dest string in the memory
        BRS     StrCpy  ;call the StrCpy subroutine
End:    HALT            ;Halt at the end

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   StrCpy Subroutine
;
```

```
;   B0  ->  temporary byte to hold the value to be copied
;           from source to destination
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
StrCpy: MOVb    (X4)+, B0    ;copy a character from source to B0
                             ;and increment the address counter for next byte
        MOVb    B0, (X5)+    ;now paste it at the destination address
                             ;and increment the address counter for next byte
        BRne    StrCpy       ;branch back to do this byte copying again
                             ;if this byte is not Null

        RTS                  ;return to caller
```

Listing C.2: MePoEfAr Assembly Code for Benchmark 2: String Copy

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   MePoEfAr Bubble Sort Benchmark Program
;   In this program, an array of 10 elements is initialized
;   in the main subroutine. Base address of this array is
;   passed to BSort subroutine to sort the numbers in
;   descending order.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Main Subroutine
;   START   ->  Base Address of Array
;   X4      ->  Index for number Array
;   D1      ->  i.e. loop counter
;   D2      ->  Data with which array will be initialized
;   X4      ->  is used to index Array
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

Main:   MOVd    #10, D1      ; j = # elements
        MOVx    #START,X4    ; X4 = base address of array
        MOVd    #0,D2        ; data with wihich array will be
                             ; initialized

L1:     MOVd    D2, (X4)+    ; Array[i] = D2

        ADDd    #1,D2        ; D2+1
        DECBRn  D1, L1       ; decrement element counter and
                             ; branch to next element if not done

        BRS     BSort        ; call the BSort routine

End:    HALT                 ; Halt at the end

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   BSort Subroutine
;   Actual subroutine used to implement sorting Algorithm

;   START   ->  Base Address of Array
;   X4  ->  used to index the Array
;   W0  ->  i i.e. loop counter
;   W1  ->  j i.e. loop counter
;   D2  ->  used to store the value of current element in the Array
;   D3  ->  used to store the value of next element in the Array
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
BSort:  MOVx    #START, X4   ; base address
        MOVw    #9,W0        ; i = 9
L1:     MOVw    W0,W1        ; j = i

L2:     MOVd    (X4)+,D2     ; D2 = arr[j]

        MOVd    @X4,D3       ; D3 = arr[j+1]

        CPAd    D2, D3       ; compare D2 with D3
        BRlt    NoSwap       ; if(D3 < D2) then no swaping required

        ;otherwise swap here
        MOVd    D2,@X4       ;arr[j+1] = D2 = arr[j]
        MOVd    D3,-4(X4)    ;arr[j]   = D3 = arr[j+1]

NoSwap: S1BR    W1, L2       ;loop if j>0

        S1BR    W0,L1        ;loop till i>0

        RTS                  ;return to caller (sorting done)
```

Listing C.3: MePoEfAr Assembly Code for Benchmark 3: Bubble Sort

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   MePoEfAr Assembly Program implementing a structure for sensor values.
;   Structure contains 3 elements:
;   1 char byte Flag indicating if sensor has been calibrated or not.
;   1 short int containing the offset to be adjusted
;   1 long int containing the actual sensor value
;
;   An array of 5 sensors is declared. InitSensors() will initialize
;   these values to some numbers. CalibrateSensors() will subtract
;   the offset from the value of the sensors and set the Flag.
;   main() will call these two functions to initialize and calibrate
;   sensor data.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Main subroutine
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Main:   BRS     Init    ;call to Init subroutine
        BRS     Calib   ;call to Calib subroutine
End:    RTS             ;end

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Init subroutine
;
;   START   ->  base address of first struct member
;   X4      ->  index struct array
;   B0      ->  loop counter
;   W0      ->  i
;   D0      ->  Data with which Value will be initialized
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Init:   MOVd    #3, D0          ;sensor value will be initialized with D0
                                ;3 is added to every value of i, so
                                ;initialized D0 with 3

        MOVx    #START, X4      ;X4 = Starting address of struct
        MOVw    #0, W0          ;i = 0
        MOVb    #5, B0          ;loop counter

L0:     MOVb    #0, (X4)+       ;sensors[i].Flag = 0
        MOVw    W0, (X4)+       ;sensors[i].Offset = i
        ADDdws  W0, D0          ;D0 = i+3
        MOVd    D0,(X4)+        ;sensors[i].Value = i+3

        ADDb    #1, W0          ;i++

        S1BR    B0, L0          ;loop back 5 times

        RTS                     ;return to caller

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Calib subroutine
;
;   START   ->  Starting address of struct array
;   X4      ->  index struct array
;   D0      ->  sensors[i].Value
;   W0      ->  sensors[i].Offset
;   B0      ->  loop counter
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Calib:  MOVx    #START, X4      ;X4 = Starting address of struct
        MOVb    #5, B0          ;loop counter

L1:     MOVb    #1, (X4)+       ;sensors[i].Flag = 1
        MOVw    (X4)+, W0       ;W0 = sensors[i].Offset
        MOVwds  W0, D0          ;D0 = sensors[i].Offset
        SUBd    D0,(X4)+        ;sensors[i].Value =   sensors[i].Value
                                ;                   - sensors[i].Offset
        S1BR    B0, L0          ;loop back 5 times
        RTS                     ;return to caller
```

Listing C.4: MePoEfAr Assembly Code for Benchmark 4: Sensor Structure

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   MePoEfAr Matrix Multiplication Benchmark Program
;   This program multiplies two matrices of order 3X4 and 4X5
;   to give a product matrix of order 3X5. Both the matrices
;   are initialized with some numbers and then multiplication
;   is performed to get product.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Main Subroutine
;   Base Address of matrix m1 -> M1
;   Base Address of matrix m2 -> M2
;   Base Address of matrix m3 -> M3
;   X4 is used to index the elements of matrix m1
;   X1 is used to index the elements of matrix m2
;   X5 is used to index the elements of matrix m3
;   B6  ->  no of rows
;   B7  ->  no of columns
;   B1,B4,B5    ->  loop counters
;   Note: Arrays are stored in memory in Row Major Order
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Main:   MOVd    #nRows1, B6     ; B6 = no of rows of m1
        MOVd    #nCols1, B7     ; B7 = no of cols of m1
        MOVx    #M1, X4         ;base address of m1
        BRS     INIT            ;call initialize subroutine
                                ;for m1

        MOVd    #nRows2, B6     ; B6 = no of rows of m2
        MOVd    #nCols2, B7     ; B7 = no of cols of m2
        MOVx    #M2, X4         ;base address of m2
        BRS     INIT            ;call initialize subroutine
                                ;for m2

        ;now perform multiplication

        ;Initialize base address of m1, m2, m3
        INx     X4, #3, #M1, #M2, #M3

        MOVd    #5, B5          ;nCols2
L3:     MOVd    #3, B4          ;nRows1
L2:     MOVd    #4, B1          ;nCols1 (or nRows2 is same)
        MOVd    #0, D3          ;D3 = 0 (accumulator for one element)
L1:     MOVd    (X4)+, D2       ;D2 = m1[m][n]
        MULd    @X1, D2         ;D2 = m1[m][n] * m2[n][p]
        ADDd    D2, D3          ;D3 = D3 + m1[m][n] * m2[n][p]
        ADDx    #20, X1         ;X1 += nCols * size
                                ;it will point to first element of next row
        S1BR    B1, L1          ;repeat 4 times

        MOVd    D3, (X5)+       ;m3[m][p] = D3
        SUBx    #56, X1         ;X1 now points to first element of next column
        S1BR    B4, L2          ;repeat this 5 times

        MOVx    #M2, X1         ;X1 now points to base address of m2
        S1BR    B5, L3          ;repeat this 3 times

        RTS                 ;multiplication done, stop

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   INIT Subroutine
;   D2  ->  row number
;   D3  ->  value to be assigned
;   X4  ->  array index
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
INIT:   MOVd    #0, D2          ;D2 = 0, row number
        MOVd    #0, D3          ;D3 = 0
L1:     MOVd    D3, (X4)+       ;Matrix[r][c] = D3
        ADDd    #1, D3          ;D3++
        S1BR    B7, L1          ;repeat this for nCols

        ADDd    #1, D2          ;increment row
        MOVd    D2, D3          ;D3 = D2 = row number

        S1BR    B6, L1          ;repeat this for nRows

        RTS                     ;return to caller
```

Listing C.5: MePoEfAr Assembly Code for Benchmark 5: Matrix Multiplication

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   MePoEfAr FIR Filter Benchmark Program
;   This program is an implmentation of an FIR filter.
;   COEFF and INPUT arrays are initialized with some data and
;   then FIR caculations are performed to get the OUTPUT array.
;   These calculations are basically integer and floating point
;   calculations performed on these arrays to get floating
;   results in OUTPUT array.
;
;   Total Number of Instruction: 45
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;    Main Subroutine
;
;    COEFF   ->  base address of COEFF array
;    INPUT   ->  base address of INPUT array
;    OUTPUT  ->  base address of OUTPUT array
;    X4  ->  index of COEFF array
;    X2  ->  index of INPUT array
;    X6  ->  index of OUTPUT array
;    F1  ->  sum
;    F0, F2, F3, F5  ->  floating point temporary results
;    D0, D1, D2, D3 are used for integer temporary calculations
;    B1, B2  ->  loop counters
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        ;initialize COEFF array
Main:   MOVx    #COEFF, X4   ;base address of COEFF
        MOVb    #18, B0      ;no of coeff
        MOVf    #5, F0       ;F0 = i+5
L1:     MOVf    #1, F2       ;F2 = 1
        DIVf    F0, F2       ;F2 = 1/(i+5)

        MOVf    F2, (X4)+    ;COEFF[i] = F2

        ADDf    #1, F0       ;update value of i+5 in F0

        S1BR    B0, L1       ;loop back for all coeffecients

        ;initialize INPUT array
        MOVx    #INPUT, X4   ; base address of INPUT
        MOVb    #68, B0      ; no of INPUT samples
        MOVd    #2, D2       ; value to be stored

L2:     MOVw    W2, (X4)+    ; INPUT[i] = 2

        S1BR    B0, L2       ; loop back for all INPUT samples

        ;Perform FIR Calculations
        MOVx    #COEFF, X4
        MOVx    #INPUT, X2
        MOVx    #OUTPUT, X6

        MOVb    #36, B1      ;y = 36

L4:     MOVb    #8, B2       ;i = 8
        MOVf    #0, F1       ;sum = 0

L3:     MOVx    #16, X3      ;X3 = 16
        SUBbxs  B2, X3       ;X3 = 16-i
        ADDbxs  B1, X3       ;X3 = y+16-i
        MULx    #2, X3       ;X3 = (y+16-i) * 2
        ADDx    X2, X3       ;X3 = start + (y+16-i) * 2
        MOVw    @X3, W3      ;W3 = INPUT[y+16-i]

        MOVbxs  B1, X3       ;X3 = y
        ADDbxs  B2, X3       ;X3 = y+i
        MULx    #2, X3       ;X3 = (y+i) * 2
        ADDx    X2, X3       ;X3 = start + (y+i) * 2
        ADDd    @X3, W3      ;W3 = INPUT[y+16-i]+ INPUT[y+i]

        MOVwfs  W3, F3       ;convert to float

        MULdfs  (X4)+, F3    ;F3 = COEFF[i] * (INPUT[y+16-i]+ INPUT[y+i])

        ADDf    F3, F1       ;F1 = sum + F3

        S1BR    B2, L3       ;loop back 8 times

        MOVf    32(X0),F5    ;F5 =  COEFF[8]

        MOVx    #8, X3       ;X3 = 8
        ADDbxs  B1, X3       ;X3 = y+8
        MULx    #4, X3       ;X3 = (y+8) * size
        ADDx    X2, X3       ;X3 = start + (y+8) * size
        MOVw    @X3, W3      ;W3 = INPUT[y+8]

        MULwfs  W3, F5       ;F5 = INPUT[y+8] * COEFF[8]
        ADDf    F1, F5       ;F5 = sum + INPUT[y+8] * COEFF[8]

        MOVf    F5,(X6)+     ;OUTPUT[y] = F5

        S1BR    B1, L4       ;loop back 36 times

        ;otherwise we are done
End:    HALT                 ; Halt the program
```

Listing C.6: MePoEfAr Assembly Code for Benchmark 6: FIR

C.2 Atmel AVR AT90S851 Assembly Codes

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Atmel AVR Recursive Factorial Benchmark Program
;   This program recursively calculates the factorial
;   of a number (n). A number is passed to this subroutine
;   by main for factorial calculation.
;
;   Total Number of Instruction: 53
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   main subroutine
;
;   R18,R19 contain n
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

Main:   LDI     R18,0x05        ;n=5;
        LDI     R19,0x00
        RCALL   Fact            ;call factorial

        RET                     ;end of main

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Factorial subroutine
;
;   R18,R19 contain n
;   R22-R25 will hold the calculated factorial
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ;push registers on stack
Fact:   PUSH    R18             ;Push register on stack
        PUSH    R19             ;Push register on stack

        ;if(n<=1)       //i.e. if the number is 0 or 1
        CPI     R18,0x02        ;Compare with 2
        CPC     R19,R1          ;Compare with carry
        BRGE    L0              ;Branch if(n>=2)

        ;return 1;   //then return 1
        LDI     R22,0x01        ;Load immediate
        LDI     R23,0x00        ;Load immediate
        LDI     R24,0x00        ;Load immediate
        LDI     R25,0x00        ;Load immediate
        RJMP    L1              ;to return 1
                                ;go down to L1 retore registers and return
        ;otherwise
        ;return n * factorial(n-1);  //this n times factorial of n-1
L0:     MOV     R20,R18         ;Copy register
        MOV     R21,R19         ;Copy register
        ;R20-R21 contain n

        SBIW    R18,0x01        ;Subtract immediate from word
        ;R18 now contain n-1

        ;call factorial(n-1)
        RCALL   Fact             ;Relative call subroutine
        ;result is in R22-R25

        ;copy n back to R18-R21 for multiplication
        MOV     R18,R20         ;Copy register
        MOV     R19,R21         ;Copy register
        CLR     R20             ;Clear Register
        CLR     R21             ;Clear Register

        RCALL   Mult32          ;Mult32
        ;result of multiplication is R22-R25

        ;restore registers
L1:     POP     R19             ;Pop register from stack
        POP     R18             ;Pop register from stack
        RET                     ;Subroutine return

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Mult32 subroutine
;
;   R18-R21 first operand
```

```
;    R22-R25 second operand
;    R22-R25 result of multiplication
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Mult32: CLR       R31              ;Clear Register
        CLR       R30              ;Clear Register
        CLR       R27              ;Clear Register
        CLR       R26              ;Clear Register
M0:     SBRS      R22,0            ;Skip if bit in register set
        RJMP      M1               ;Relative jump
        ADD       R26,R18          ;Add without carry
        ADC       R27,R19          ;Add with carry
        ADC       R30,R20          ;Add with carry
        ADC       R31,R21          ;Add with carry
M1:     LSL       R18              ;Logical Shift Left
        ROL       R19              ;Rotate Left Through Carry
        ROL       R20              ;Rotate Left Through Carry
        ROL       R21              ;Rotate Left Through Carry
        LSR       R25              ;Logical shift right
        ROR       R24              ;Rotate right through carry
        ROR       R23              ;Rotate right through carry
        ROR       R22              ;Rotate right through carry
        BRNE      M0               ;Branch if not equal
        SBIW      R24,0x00         ;Subtract immediate from word
        CPC       R23,R22          ;Compare with carry
        BRNE      M0               ;Branch if not equal
        MOV       R25,R31          ;Copy register
        MOV       R24,R30          ;Copy register
        MOV       R23,R27          ;Copy register
        MOV       R22,R26          ;Copy register
        RET                        ;Subroutine return
```

Listing C.7: Atmel AVR AT90S851 Assembly Code for Benchmark 1: Recursive Factorial

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;    Atmel AVR String Copy Assembly Program
;    In this program, main subroutine passes the addresses of source and
;    destination strings to the StrCpy subroutine to copy the chracters
;    from source to the destination string.

;    Total Number of Instructions: 11
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;    main subroutine
;
;    R30,R31 contain address of strSrc
;    R28,R29 contain address of strDest
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

Main:        LDI     R30, 0x60            ;address of strSrc
             LDI     R31, 0x00
             LDI     R28, 0x70            ;address of strDest
             LDI     R29, 0x00

             RCALL   strCopy              ;call string copy subroutine

             RET                          ;end of main

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;    strCopy subroutine
;
;    R24 is used as temp to hold current character
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
strCopy:     LD      R24, Z+              ;R24 = strSrc[i]
             ST      Y+, R24              ;strDest[i] = R24

             TST     R24                  ;test if character is null
             BRNE    strCopy              ;if not then loop back for next

             RET                          ;done copying, return to caller
```

Listing C.8: Atmel AVR AT90S851 Assembly Code for Benchmark 2: String Copy

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;    Atmel AVR  Bubble Sort Benchmark Program
;    In this program, an array of 10 elements is initialized
```

```
;   in the main subroutine. Base address of this array is
;   passed to BSort subroutine to sort the numbers in
;   descending order.

;   Total Number of Instruction: 42
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   main subroutine
;
;   R30,R31 contain address of array
;   R24,R25 for i
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

Main:   LDI       R30,0x00        ;base address of array
        LDI       R31,0x00        ;base address of array

        LDI       R24,0x00        ;i=0
        LDI       R25,0x00

        ;Array[i]=i;
L0:     ST        Z+,R24          ;Store indirect and postincrement
        ST        Z+,R25          ;Store indirect and postincrement

        ADIW      R24,0x01        ;i++
        CPI       R24,0x0A        ;Compare with 10
        CPC       R25,R1          ;Compare with carry
        BRNE      L0              ;loop back if(i<10)


        RCALL     BSort        ;call BSort subroutine

        RET                    ;Subroutine return

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   BSort subroutine
;
;   R30,R31 contain address of array
;   R18,R19 for i
;   R20,R21 for j
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
BSort:  MOV       R30,R24         ;base address of a[]
        MOV       R31,R25         ;base address of a[]

        LDI       R18,0x08        ;i=0
        LDI       R19,0x00        ;i=0

L2:     LDI       R20,0x00        ;j=0
        LDI       R21,0x00        ;j=0


        ;a[j]
L1:     LDD       R22,Z+0         ;Load indirect with displacement
        LDD       R23,Z+1         ;Load indirect with displacement

        ;a[j+1]
        LDD       R26,Z+2         ;Load indirect with displacement
        LDD       R27,Z+3         ;Load indirect with displacement

        ;if a number is greater than its next number
        CP        R26,R22         ;if(a[j]>a[j+1])
        CPC       R27,R23         ;if(a[j]>a[j+1])
        BRGE      NoSwap          ;then no swap required

        ;otherwise we need to swap
        ;a[j]=a[j+1]
        STD       Z+1,R27         ;Store indirect with displacement
        STD       Z+0,R26         ;Store indirect with displacement

        ;a[j+1]=a[j]
        STD       Z+3,R23         ;Store indirect with displacement
        STD       Z+2,R22         ;Store indirect with displacement

        ;loop condition for j
NoSwap: SUBI      R20,0xFF        ;Subtract immediate
        SBCI      R21,0xFF        ;Subtract immediate with carry
        ADIW      R30,0x02        ;Add immediate to word
        CP        R18,R20         ;Compare
        CPC       R19,R21         ;Compare with carry
        BRGE      L1              ;Branch if greater or equal, signed

        ;loop condition for i
        SUBI      R18,0x01        ;Subtract immediate
        SBCI      R19,0x00        ;Subtract immediate with carry
        SER       R20             ;Set Register
        CPI       R18,0xFF        ;Compare with immediate
        CPC       R19,R20         ;Compare with carry
```

```
        BRNE      L2             ;loop back if(i>=0)

        ;done with sorting
        RET                      ;Subroutine return
```

Listing C.9: Atmel AVR AT90S851 Assembly Code for Benchmark 3: Bubble Sort

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Atmel AVR Assembly Program implementing a structure for sensor values.
;   Structure contains 3 elements:
;   1 char byte Flag indicating if sensor has been calibrated or not.
;   1 short int containing the offset to be adjusted
;   1 long int containing the actual sensor value
;
;   An array of 5 sensors is declared. InitSensors() will initialize
;   these values to some numbers. CalibrateSensors() will subtract
;   the offset from the value of the sensors and set the Flag.
;   main() will call these two functions to initialize and calibrate
;   sensor data.
;
;   Total Number of Instruction: 39
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Main subroutine
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Main:   RCALL   Init    ;call to Init subroutine
        RCALL   Calib   ;call to Calib subroutine
End:    RET             ;end

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Init subroutine
;
;   STHi,STLo   ->  starting address of struct array
;   R30, R32    ->  pointer to current element
;   R15,R16     ->  loop counter, i
;   R20-R23     ->  data with which Values will be initialized
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Init:   MOV     R30, #STLo  ;R30,R31 contain starting address of struct
        MOV     R31, #STHi

        MOV     R20, #3     ;R20 = 3
                            ;sensor value will be initialized with D0
                            ;3 is added to every value of i, so
                            ;initialized D0 with 3
        CLR     R21
        CLR     R22
        CLR     R23

        MOV     R15, #0     ;i = 0
        MOV     R16, #0

L0:     STD     Z+, #0      ;Flag = 0

        STD     Z+, R15     ;Offset = i
        STD     Z+, R16


        INC     R20         ;R20 = i + 3
        ADC     R21, R16
        ADC     R22, #0
        ADC     R23, #0

        STD     z+, R20     ;Value = i+3
        STD     z+, R21
        STD     z+, R22
        STD     z+, R23

        INC     R15         ;i++

        CPI     R15, #5     ;compare with 5
        BRNE    L0          ;loop back 5 times

        RET                 ;return to caller


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Calib subroutine
;
;   STHi,STLo   ->  starting address of struct array
;   R30, R32    ->  pointer to current element
;   R15         ->  loop counter, i
```

```
;    R18,R19      ->  to hold offset
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Calib:  MOV     R30, #STLo   ;R30,R31 contain starting address of struct
        MOV     R31, #STHi

        MOV     R15, #5      ;i = 5

L1:     STD     Z+0, #1      ;Flag = 1

        MOV     R18, z+      ;R18,R19 = offset
        MOV     R19, z+

        SUB     z+, R18      ;value = value - offset
        SBC     z+, R19
        SBCI    z+, #0
        SBCI    z+, #0

        DEC     R15          ;decrement loop counter

        BRNE    L1           ;loop back for 5 sensors

        RET                  ;return to caller
```

Listing C.10: Atmel AVR AT90S851 Assembly Code for Benchmark 4: Sensor Structure

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;    Atmel AVR Matrix Multiplication Benchmark Program
;    This program multiplies two matrices of order 3X4 and 4X5
;    to give a product matrix of order 3X5. Both the matrices
;    are initialized with some numbers and then multiplication
;    is performed to get product.
;
;    Total Number of Instruction: 105
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;    Main Subroutine
;    Base Address of matrix m1 -> M1Lo, M1Hi
;    Base Address of matrix m2 -> M2Lo, M2Hi
;    Base Address of matrix m3 -> M3Lo, M3Hi
;    R26,R27      ->  pointer to m1
;    R28,R29      ->  pointer to m2
;    R30,R31      ->  pointer to m3
;    R18-R21      ->  hold current element of m1
;    R22-R25      ->  hold current element of m2
;    R15,R16,R17 ->  temporaries for passing values
;                    and loop couters
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ;initialize m1
Main:   MOV     R15, #nRows1        ;rows of m1
        MOV     R16, #nCols1        ;cols of m1

        MOV     R30, #M1Lo      ;base address m1 low
        MOV     R31, #M1Hi      ;base address m1 high

        RCALL   INIT                ;call initialization subroutine

        ;initialize m2

        MOV     R15, #nRows2        ;rows of m2
        MOV     R16, #nCols2        ;cols of m2

        MOV     R30, #M2Lo      ;base address m2 low
        MOV     R31, #M2Hi      ;base address m2 high

        RCALL   INIT                ;call initialization subroutine

        ;perform multiplication
        MOV     R26, #M1Lo      ;base address m1 low
        MOV     R27, #M1Hi      ;base address m1 high

        MOV     R28, #M2Lo      ;base address m2 low
        MOV     R29, #M2Hi      ;base address m2 high

        MOV     R30, #M3Lo      ;base address m3 low
        MOV     R31, #M3Hi      ;base address m3 high

        MOV     R15, #5      ;nCols2
L3:     MOV     R16, #3      ;nRows1
L2:     MOV     R17, #4      ;nCols1

```

```
        LDI     Z+0,#0          ;m3[m][p] = 0
        LDI     Z+1,#0
        LDI     Z+2,#0
        LDI     Z+3,#0

L1:     MOV     R18, X+         ;R18-R21 = m1[m][n]
        MOV     R19, X+
        MOV     R20, X+
        MOV     R21, X+

        MOV     R22, Y+         ;R22-R25 = m2[m][n]
        MOV     R23, Y+
        MOV     R24, Y+
        MOV     R25, Y+

        ;perform m1[m][n] * m2[n][p]
        RCALL     Mult32        ;Relative call subroutine

        ADD     Z+0,R22         ;m3[m][p] += m1 * m2
        ADD     Z+1,R23
        ADD     Z+2,R24
        ADD     Z+3,R25

        ADIW    R29:R28,#20     ;pointer for m2
                                ;it will point to first element of next row

        DEC     R17             ;decrement
        BRNE    L1              ;loop back 4 times

        ADIW    R31:R30,#4      ;now point to the next element of m3
                                ;as we are done with current element

        SBIW    R29:R28,#56     ;pointer for m2
                                ;now points to first element of next column

        DEC     R16             ;decrement
        BRNE    L2              ;loop back 3 times

        MOV     R28, #M2Lo      ;base address m2 low
        MOV     R29, #M2Hi      ;base address m2 high
                                ;now points to base address of m2

        DEC     R15             ;decrement
        BRNE    L3              ;loop back 5 times

        RET                     ;Subroutine return

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Matrix INIT Subroutine
;
;   R23-R26 ->  Data to be assigned
;   R15     ->  has the nRows
;   R16     ->  has the nCols
;   R30,R31 ->  element pointer
;   R17     ->  Row number
;
;   Instructions    = 22
;   Bytes           = 44
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
INIT:   CLR       R23             ;data = 0
        CLR       R24
        CLR       R25
        CLR       R26

        CLR       R17             ;row counter


L1:     ;mat[m][p] = data
        STD       Z+0,R23         ;Store indirect with displacement
        STD       Z+1,R24         ;Store indirect with displacement
        STD       Z+2,R25         ;Store indirect with displacement
        STD       Z+3,R26         ;Store indirect with displacement

        ;increment data = data + 1
        ADD       R23,#1        ;Add without carry
        ADC       R24,#0        ;Add with carry
        ADC       R25,#0        ;Add with carry
        ADC       R26,#0        ;Add with carry

        ;R30,R31 =  address of m1[m][p]
        ;now point to next element in the matrix
        ADI       R30,#4        ;Copy register
        ADC       R31,#0        ;Copy register

        DEC       R16
        BRGT      L1            ;repeat it for all columns

```

```
        ;row ++
        INC       R17           ;Add immediate to word

        MOV       R17,R23       ;R23 = row number (data for new row)

        DEC       R15
        BRGT      L1            ;repeat it for all rows

        RET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   32-bit Multiplication Subroutine
;   R18-R21 ->  First Number
;   R22-R25 ->  Second Number
;   R22-R25 ->  Product
;
;   Instructions    = 35
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Mult32: PUSH    R26             ;push on stack
        PUSH    R27             ;these are used as pointers in main
        PUSH    R30
        PUSH    R31

        EOR     R31, R31        ;clear registers
        EOR     R30, R30        ;for results
        EOR     R27, R27
        EOR     R26, R26

M1:     SBRS    R22, 0
        RJMP    M2
        ADD     R26, R18
        ADC     R27, R19
        ADC     R30, R20
        ADC     R31, R21

M2:     ADD     R18, R18
        ADC     R19, R19
        ADC     R20, R20
        ADC     R21, R21
        LSR     R25
        ROR     R24
        ROR     R23
        ROR     R22
        BRNE    M1
        SBIW    R24, 0X00
        CPC     R23, R22
        BRNE    M1
        MOV     R25, R31
        MOV     R24, R30
        MOV     R23, R27
        MOV     R22, R26

        POP     R31             ;restore registers
        POP     R30
        POP     R27
        POP     R26

        RET
```

Listing C.11: Atmel AVR AT90S851 Assembly Code for Benchmark 5: Matrix Multiplication

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Atmel AVR FIR Filter Benchmark Program
;   This program is an implmentation of a 17 order FIR filter.
;   COEFF and INPUT arrays are initialized with some data and
;   then FIR caculations are performed to get the OUTPUT array.
;   These calculations are basically integer and floating point
;   calculations performed on these arrays to get floating
;   result samples in OUTPUT array.
;
;   Total Number of Instruction: 189 + 530 = 719
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Main Subroutine
;
;   Y+1 to Y+4  -> sum
;   Y+5 , Y+6   -> y
;   Y+7 , Y+8   -> i
;   COEFF at address  0x0000
;   INPUT at address  0x0100
;   OUTPUT at address 0x0200
```

```
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ;COEFF initialization
Main:   STD     Y+8,R1          ;i = 0
        STD     Y+7,R1

L0:     LDD     R16,Y+7         ;R16,R17 = i
        LDD     R17,Y+8

        LDD     R22,Y+7         ;R22,R25 = i
        LDD     R23,Y+8
        CLR     R24
        CLR     R25

        RCALL   INT2FLOAT       ;convert i to floating point
                                ;so now R22-R25 = float(i)

        LDI     R18,0x00        ;R18-R21 = 5.0
        LDI     R19,0x00
        LDI     R20,0xA0
        LDI     R21,0x40

        RCALL   FADD            ;floating point add
                                ;so now R22-R25 = (i+5.0)

        LDI     R18,0x00        ;R18-R22 = 1.0
        LDI     R19,0x00
        LDI     R20,0x80
        LDI     R22,0x3F

        RCALL   FDIV            ;floating point divide
                                ;so now R22-R25 = 1.0 / (i+5.0)

        ;now compute the address of COEFF[i]
        MOV     R30,R16         ;R30,R19 = i
        MOV     R31,R17
        LSL     R30             ;R30,R31 = i * 4
        ROL     R31
        LSL     R30
        ROL     R31
        LDI     R20,0x00        ;R20,R21 = base address of COEFF
        LDI     R21,0x00        ;i.e 0x00
        ADD     R30,R20         ;R30,R31 = base + i * 4
        ADC     R31,R21
        STD     Z+0,R24         ;COEFF[i] = 1 / (i+5.0)
        STD     Z+1,R25
        STD     Z+2,R26
        STD     Z+3,R27

        LDD     R24,Y+7         ;R24,R25 = i
        LDD     R25,Y+8
        ADIW    R24,0x01        ;i++
        STD     Y+8,R25         ;Store back i
        STD     Y+7,R24
        CPI     R24,0x11        ;Compare with 17
        CPC     R25,R1
        BRLT    L0              ;loop back if(i<17)

        ;INPUT array initialization
        STD     Y+8,R1          ;i = 0
        STD     Y+7,R1

L1:     LDD     R30,Y+7         ;R30,R31 = i
        LDD     R31,Y+8

        SUBI    R18,0x00        ;base address of INPUT i.e 0x0100
        SBCI    R19,0x01
        LSL     R30             ;R30,R31 = i * size
        ROL     R31
        ADD     R30,R18         ;Add base address
        ADC     R31,R19         ;i.e. R30,R31 = base + i * size
        LDI     R18,0x02        ;R18,R19 = 2
        LDI     R19,0x00
        STD     Z+1,R19         ;INPUT[i] = 2
        STD     Z+0,R18

        LDD     R24,Y+7         ;R24,R31 = i
        LDD     R25,Y+8
        ADIW    R24,0x01        ;i++
        STD     Y+8,R25         ;Store i back
        STD     Y+7,R24
        CPI     R24,0x43        ;Compare i with 67
        CPC     R25,R1
        BRLT    L1              ;loop back if(i<67)

        ;perform filtering
        STD     Y+6,R1          ;y=0
```

```
        STD     Y+5,R1

L2:     LDI     R24,0x00        ;R24-R27 = 0
        LDI     R25,0x00
        LDI     R26,0x00
        LDI     R27,0x00
        STD     Y+1,R24         ;sum = 0
        STD     Y+2,R25
        STD     Y+3,R26
        STD     Y+4,R27

        STD     Y+8,R1          ;i=0
        STD     Y+7,R1

        ;inner loop which will iteratively compute
        ;sum = sum + COEFF[i] * ( INPUT[y + 16 - i] + INPUT[y + i] )
L3:     LDD     R30,Y+7         ;R30,R31 = i
        LDD     R31,Y+8
        LSL     R30             ;R30,R31 = i * size
        ROL     R31
        LSL     R30
        ROL     R31
        ADDI    R30,0x00        ;R30,R31 = base + i * size
        ADIC    R31,0x00
        LDD     R14,Z+0         ;R14,R17 = COEFF[i]
        LDD     R15,Z+1
        LDD     R16,Z+2
        LDD     R17,Z+3

        LDD     R24,Y+5         ;R24,R25 = y
        LDD     R25,Y+6
        LDD     R24,Y+7         ;R24,R25 = i
        LDD     R25,Y+8
        SUB     R30,R24         ;y-i
        SBC     R31,R25
        ADDI    R30,0x00        ;R30,R31 = y - 1 + 16
        ADCI    R31,0x10
        LSL     R30             ;R30,R31 = (y - 1 + 16)*size
        ROL     R31
        ADDI    R30,0x00        ;add base address of INPUT
        ADCI    R31,0x01        ;i.e. R30,R31 = base + (y - 1 + 16)*size
        LDD     R18,Z+0         ;R18,R19 = INPUT[y+16-i]
        LDD     R19,Z+1

        LDD     R20,Y+5         ;R20,R21 = y
        LDD     R21,Y+6
        LDD     R30,Y+7         ;R30,R31 = i
        LDD     R31,Y+8
        ADD     R30,R20         ;y+i
        ADC     R31,R21
        LSL     R30             ;R24,R25 = (y+i) * size
        ROL     R31
        ADDI    R30,0x00        ;add base address of INPUT
        ADCI    R31,0x01        ;i.e. R30,R31 = base + (y+i) * size
        LDD     R24,Z+0         ;R24,R25 = INPUT[y+i]
        LDD     R25,Z+1
        ADD     R18,R24         ;R18,R19 = INPUT[y+16-i] + INPUT[y+i]
        ADC     R19,R25

        MOV     R22,R18         ;R22-R25 = INPUT[y+16-i] + INPUT[y+i]
        MOV     R23,R19
        CLR     R24
        SBRC    R23,7           ;Skip if bit in register cleared
        LAT     R24             ;Load and Toggle
        MOV     R25,R24

        RCALL   INT2FLOAT       ;call to int2float subroutine
                                ;so R22-R25 will be converted to float
                                ;i.e. R22-R25 = float(INPUT[y+16-i] + INPUT[y+i])

        MOV     R18,R14         ;R18-R21 = COEFF[i]
        MOV     R19,R15
        MOV     R20,R16
        MOV     R21,R17

        RCALL   FMUL            ;call to floatin point multiplication routine
                                ;R22-R25 = COEFF * ( INPUT[y+16-i] + INPUT[y+i])

        LDD     R18,Y+1         ;R18-R21 = sum
        LDD     R19,Y+2
        LDD     R20,Y+3
        LDD     R21,Y+4

        RCALL   FADD            ;call to floating point addition routine

        STD     Y+1,R22         ;store back the value of sum
        STD     Y+2,R23
```

```
        STD     Y+3,R24
        STD     Y+4,R25

        LDD     R24,Y+7         ;R24,R25 = i
        LDD     R25,Y+8
        ADIW    R24,0x01        ;i++
        STD     Y+8,R25         ;store i back
        STD     Y+7,R24
        CPI     R24,0x08        ;compare i with 8
        CPC     R25,R1          ;Compare with carry
        BRGE    L3              ;loop back if(i<8)

        ;outer loop which will make output samples
        ;OUTPUT[y] = sum + INPUT[y + 8] * COEFF[8];
        LDD     R30,Y+5         ;R30,R31 = y
        LDD     R31,Y+6
        ADIW    R30,0x08        ;R30,R31 = y+8
        LSL     R30             ;R30,R31 = (y+8) * size
        ROL     R31
        ADDI    R30,0x00        ;add base address
        ADCI    R31,0x01        ;R30,R31 = base + (y+8) * size

        LDD     R22,Z+0         ;R22-R25 = INPUT[y+8]
        LDD     R23,Z+1
        CLR     R24             ;Clear Register
        SBRC    R23,7           ;Skip if bit in register cleared
        LAT     R24             ;Load and Toggle
        MOV     R25,R24

        RCALL   INT2FLOAT       ;call int2float subroutine
                                ;i.e. R22-R25 = float(INPUT[y+8])

        LDD     R18,Y+41        ;R18-R21 = COEFF[8]
        LDD     R19,Y+42        ;substrict is constant
        LDD     R20,Y+43        ;so calculated at assemble time
        LDD     R21,Y+44

        RCALL   FMUL            ;call floating point multiplication
                                ;so R22-R25 = INPUT[y + 8] * COEFF[8]

        LDD     R18,Y+1         ;R18-R21 = sum
        LDD     R19,Y+2
        LDD     R20,Y+3
        LDD     R21,Y+4

        RCALL   FADD            ;call floating point addition routine
                                ;so R22-R25 = sum + INPUT[y + 8] * COEFF[8]

        LDD     R30,Y+5         ;R30,R31 = y
        LDD     R31,Y+6
        LSL     R30             ;R30,R31 = y * size
        ROL     R31
        LSL     R30
        ROL     R31
        ADDI    R30,0x00        ;add base address
        ADCI    R31,0x02        ;base address of OUTPUT is 0x0200
        STD     Z+0,R22         ;as R22-R25 = sum + INPUT[y + 8] * COEFF[8]
        STD     Z+1,R23         ;assign to OUTPUT[y]
        STD     Z+2,R24         ;i.e.
        STD     Z+3,R25         ;OUTPUT[y] = sum + INPUT[y + 8] * COEFF[8]

        LDD     R24,Y+5         ;R24,R25 = y
        LDD     R25,Y+6
        ADIW    R24,0x01        ;y++
        STD     Y+6,R25         ;Store back y
        STD     Y+5,R24
        CPI     R24,0x24        ;y Compare with 36
        CPC     R25,R1
        BRLT    L2              ;loop back if(y<36)

        ;done with filtering
        RET                     ;Subroutine return
```

Listing C.12: Atmel AVR AT90S851 Assembly Code for Benchmark 6: FIR

C.3 TI MSP430 Assembly Codes

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;    TI MSP430 Recursive Factorial Assembly Program
```

```
;   This program recursively calculates the factorial
;   of a number (n). A number is passed to this subroutine
;   by main for factorial calculation.
;
;   Total No of Instruction = 26
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   main subroutine
;   r12      ->  n for which factorial is to be calculated
;   r14,r15 ->  calculated factorial
;   r4,r5   ->  temporaries
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Main:   MOV.W     #5,r12      ;r12 = 5

        CALL      #Fact       ;call Fact subroutine

End:    RET                   ;end of main

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Fact subroutine which will calculate factorial of 5
;   r10   assigned to n
;   r12 and r13 will hold resulting factorial
;   r14 and r15 are temporaries for multiplication
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Fact:   PUSH.W    r12         ;r12 will be modified so save it
        MOV.W     r12,r10     ;r10 = n
        SUB.W     #1,r12      ;r12 = n-1
        JGE       L1          ;jump to L1 if(n>=2)

        ;base case is the case for 0 and 1
        ;factorial of which is 1
        MOV.W     #1,r14      ;r14 = 1
        MOV.W     #0,r15      ;r15 = 0
        POP       r12         ;restore r12
        RET                   ;return to caller

        ;if not base case then we need to find factorial(n-1)
        ;and multiply with n to get result
L1:     CALL      #Fact       ;call factorial for n-1
        MOV.W     r14,r4      ;put lower 16 bits of result in r14
        MOV.W     r15,r5      ;put upper 16 bits of result in r15

        ;As r10 = n
        ;Now to calculate  n * factorial(n-1)
        ;perform multiplication
        CLR       r14             ; prod hi = 0
        CLR       r15             ; prod low = 0

        ;LSBs * LSBs
        MOV       r4,&0130h       ; copy to multiplier registers (OP1LO)
        MOV       r10,&0138h      ; OP2LO
        MOV       &SumLo,r14      ; Add product to result (Sum0)
        MOV       &SumHi,r15      ; Sum1

        MOV       r10,&0130h      ; copy to multiplier registers (OP2LO)
        MOV       r5,&0138h       ; OP1HI

        ADD       &SumLo,r14      ; Add product to result (Sum0)
        ADDC      &SumHi,r15      ; Sum1


        ; so now r14 and r15 contain the result
        POP       r12             ;restore r10
        RET                       ;return to caller
```

Listing C.13: TI MSP430 Assembly Code for Benchmark 1: Recursive Factorial

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   TI MSP430 String Copy Assembly Program
;   In this program, main subroutine passes the addresses of source and
;   destination strings to the StrCpy subroutine to copy the chracters
;   from source to the destination string.
;
;   Total No of Instructions = 9
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Main subroutine
;   r12   assigned to strSrc
;   r15   assigned to strDest
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
Main:       MOV.W     #strSrc,r12     ;address of source string
            MOV.W     #strDest,r15    ;address of destination string
            CALL      #strCopy        ;call the copy subroutine
End:        RET                       ;end of main

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   strCopy subroutine
;* r12   has strSrc base address
;* r15   has strDest base address
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
strCopy:    MOV.B     @r12+,0(r15)    ;copy a byte from
                                      ;source to destination
            ADD.B     #1,r15          ;increment destination address
            TST.B     0(r15)          ;test for null character
            JNE       strCopy         ;loop back if not null

            RET                       ;return to caller
```

Listing C.14: TI MSP430 Assembly Code for Benchmark 2: String Copy

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   TI MSP430 Bubble Sort Assembly Program
;   In this program, an array of 10 elements is initialized
;   in the main subroutine. Base address of this array is
;   passed to BSort subroutine to sort the numbers in
;   descending order.
;
;   Total No of Instruction = 33
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Main Subroutine
;   START   ->  starting address of array
;   r15     ->  pointer to current element in the array
;   r13     ->  loop counter i
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Main:   MOV.W     #0,r13         ;i = 0
        MOV.W     #0,r14
        MOV.W     #START,r15     ;address of array element

L1:     MOV.W     r13,0(r15)     ;arr[i] = i
        MOV.W     r14,2(r15)

        ADD.W     #4,r15         ;point to next element of array
                                 ;size of each element is 4

        ADD.W     #1,r13         ;i++

        CMP.W     #10,r13        ;compare with 10
        JL        L1             ;loop back if(i<10)
        ;otherwise we are done with all the elements in the array

        CALL      #BSort         ;call the BSort routine

End:    RET                      ;end of main

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   BSort Subroutine
;   START   ->  starting address of array
;   r15     ->  pointer to current element in the array
;   r13     ->  loop counter i
;   r9      ->  loop counter j
;   r10,r11 ->  temporary
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
BSort:  MOV.W     #8,r13                 ; i = 8
        MOV.W     #START,r15             ; starting address of array
L3:     MOV.W     #0,r9                  ; j = 0
L4:     ;perform the comparison of arr[j] and arr[j+1]
        ;both are 32 bit values
        ;r10,r11  = arr[j] which will be used for comparison and swaping
        MOV.W     @r15,r10               ; r10 = high arr[j]
        MOV.W     2(r15),r11             ; r11 = low arr[j]

        CMP.W     6(r15),r11             ; compare higher 16 bits
        JL        L5                     ; if(arr[j] < arr[j+1] )
                                         ; then swaping is required
        JNE       L6                     ; otherwise no swaping required

        ;if higher 16 bits are same then
        ;we need to compare lower 16 bits
        CMP.W     4(r15),r10             ; now compare lower 16 bits
        JHS       L6                     ; if they are same then

```

```
        ;swap arr[j] and arr[j+1]
        ;r10,r11 contain arr[j]
L5:     ;arr[j] = arr[j+1]
        MOV.W    4(r15),0(r15)        ; move low 16 bits
        MOV.W    6(r15),2(r15)        ; move high 16 bits

        ;arr[j+1] = arr[j]
        MOV.W    r10,4(r15)           ; copy high 16 bits
        MOV.W    r11,6(r15)           ; copy low 16 bits

L6:     ADD.W    #4,r15               ;point to next element of array
                                      ;size of each element is 4

        ADD.W    #1,r9                ; j++
        CMP.W    r9,r13               ; compare with i
        JGE      L4                   ; loop back if(i>= j)

L7:     SUB.W    #1,r13               ; i--
        TST.W    r13                  ; compare with 0
        JGE      L3                   ; loop back if(i>=0)

L8:     RET                           ;return to caller
```

Listing C.15: TI MSP430 Assembly Code for Benchmark 3: Bubble Sort

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   TI MSP430 Assembly Program implementing a structure for sensor values.
;   Structure contains 3 elements:
;   1 char byte Flag indicating if sensor has been calibrated or not.
;   1 short int containing the offset to be adjusted
;   1 long int containing the actual sensor value
;
;   An array of 5 sensors is declared. InitSensors() will initialize
;   these values to some numbers. CalibrateSensors() will subtract
;   the offset from the value of the sensors and set the Flag.
;   main() will call these two functions to initialize and calibrate
;   sensor data.
;
;   Total Number of Instruction: 29
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Main subroutine
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Main:   CALL    Init    ;call to Init subroutine
        CALL    Calib   ;call to Calib subroutine
End:    RET             ;end

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Init subroutine
;
;   START   ->  base address of first struct member
;   r15     ->  index struct array
;   r4      ->  i
;   r8,r9   ->  Data with which Value will be initialized
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Init:   MOV.W   #0, r8       ;sensor value will be initialized with r8,r9
        MOV.W   #0, r9

        MOV.W   #START, r15  ;r15 = Starting address of struct
        MOV.W   #0, r4       ;i = 0

L0:     MOV.B   #0, 0(r15)   ;sensors[i].Flag = 0
        INC.B   r15          ;increment the index

        MOV.W   r4, 0(r15)   ;sensors[i].Offset = i

        MOV.W   r4, r8       ;r8 = i
        ADC.W   #3, r9       ;r8,r9 = i+3
        MOV.W   r8,2(r15)    ;sensors[i].Value = i+3
        MOV.W   r8,4(r15)
        ADD.W   #6,r15       ;;increment the index to point to next struct element

        ADD.W   #1, r4       ;i++
        CMP.W   #5, r4       ;loop back 5 times
        JL      L0

        RTS                  ;return to caller

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Calib subroutine
;
```

```
;   START   ->  Starting address of struct array
;   r15     ->  index struct array
;   r4      ->  loop counter, i
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Calib:  MOV.W   #START, r15 ;r15 = Starting address of struct
        MOV.W   #5, r4      ;loop counter

L1:     MOV.B   #0, 0(r15)  ;sensors[i].Flag = 1
        INC.B   r15         ;increment the index

        SUB.W   0(r15),2(r15);sensors[i].Value -= sensors[i].Offset
        SUBC.W  #0,4(r15)

        ADD.W   #6,r15      ;increment the index to point to next struct element

        DEC.W   r4          ;i--
        JG      L0          ;loop back 5 times

        RTS                 ;return to caller
```

Listing C.16: TI MSP430 Assembly Code for Benchmark 4: Sensor Structure

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   TI MSP430 Matrix Multiplication Assembly Program
;   This program multiplies two matrices of order 3X4 and 4X5
;   to give a product matrix of order 3X5. Both the matrices
;   are initialized with some numbers and then multiplication
;   is performed to get product.
;
;   Total No of Instructions = 56
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Main Subroutine
;   r4    assigned to n
;   r5    assigned to m
;   r6    assigned to p
;
;   M1  ->  Base Address of matrix m1
;   M2  ->  Base Address of matrix m2
;   M3  ->  Base Address of matrix m3
;   r13 ->  index for m1
;   r14 ->  index for m2
;   r15 ->  index for m3
;   r9,r10,r11,r12  ->  temporaries
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        ;initialize m1
Main:   MOV.W     #nRows1, r6          ; r6 = no of rows
        MOV.W     #nCols1, r7          ; r6 = no of cols
        MOV.W     #M1, r12             ; base address of m1
        CALL      #INIT                ; call init routine

        ;initialize m2
        MOV.W     #nRows2, r6          ; r6 = no of rows
        MOV.W     #nCols2, r7          ; r6 = no of cols
        MOV.W     #M2, r12             ; base address of m2
        CALL      #INIT                ; call init routine

        ;perform multiplication
        MOV.W     #M1, r13             ; base address of m1
        MOV.W     #M2, r14             ; base address of m2
        MOV.W     #M3, r15             ; base address of m3

        MOV.W     #5, r4               ;nCols2
L3:     MOV.W     #3, r5               ;nRows1
L2:     MOV.W     #4, r6               ;nCols1

        MOV.W     #0, r9               ;accumulator
        MOV.W     #0, r10

L1:     ;multiplication of m1[m][n] * m2[n][p]
        CLR    r11                     ;temporary to hold product to
        CLR    r12                     ;hold m1[m][n] * m2[n][p]

        ;LSBs * LSBs
        MOV 0(R13),&0130h              ;copy to multiplier registers
        MOV 0(R14),&0138h
        ADD &SumLo,R11                 ; Add product to result
        ADDC &SumHi,R12

        ;LSBs * MSBs
        MOV 0(R13),&0130h              ;copy to multiplier registers
```

```
        MOV 2(R14),&0138h
        MOV 0(R14),&0134h                ;multiplication with accumlation
        MOV 2(R13),&0138h                ;copy to multiplier registers
        ADD &SumLo,R12                   ;Add accumulated products
        ;R11 and R12 contain product i.e. m1 * m2
        ADD.W    r11, r9                 ;accumulate products
        ADDC.W   r12, r10
        ADD.W    #20, r14                ;for the next element it should point
                                         ;to first element of next row
                                         ;nCols * size
        DEC.W    r6                      ;done with 1 row
        JG       L1
        MOV.W      r9,0(r15)             ; m3[m][p] = r9,r10
        MOV.W      r10,2(r15)
        SUB.W      #56, r14              ; decrement pointer for m1 to point
                                         ; to first element of next row
        DEC.W    r5                      ; repeat this for all the columns
        JG       L2
        MOV.W      #M2, r14              ; base address of m2
        DEC.W    r4                      ; repeat for all rows
        JG       L3
        RET`                             ; return to caller
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   INIT subroutine
;
;   r6      ->  nRows
;   r7      ->  nCols
;   r4,r5   ->  m+p value, the data to be assigned
;   r12     ->  current element address pointer
;   r9      ->  row counter
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
INIT:   MOV.W      #0, r4                ;r4,r5 represent m+p
        MOV.W      #0, r5
        MOV.W      #0, r9                ;row counter required for m+p
L9:     MOV.W      r4, 0(r12)
        MOV.W      r5, 2(r12)            ;mat[m][p] = m+p
        ADD.W      #4, r12               ;point to next array element
        ADD.W      #1, r4                ;increment m+p
        ADDC.W     #0, r5
        DEC.W      r6                    ;decrement row
        JG         L9                    ;loop if > 0
        ADD.W      #1, r9                ;increment row counter
        MOV.W      r9, r4                ;assign it to r4 (m+p)
        DEC.W      r7                    ;decrement column
        JG         L9                    ;loop back if > 0
        RET
```

Listing C.17: TI MSP430 Assembly Code for Benchmark 5: Matrix Multiplication

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   TI MSP430 FIR Assembly Program
;   This program is an implmentation of an FIR filter.
;   COEFF and INPUT arrays are initialized with some data and
;   then FIR caculations are performed to get the OUTPUT array.
;   These calculations are basically integer and floating point
;   calculations performed on these arrays to get floating
;   results in OUTPUT array.
;
;   Total No of Instructions = 76 + 256 = 332
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Main Subroutine
;
```

```
;   COEFF   ->  base address of COEFF  array
;   INPUT   ->  base address of INPUT array
;   OUTPUT  ->  base address of OUTPUT array
;
;   r7     assigned to i
;   r8,r9  assigned to sum
;   r10    assigned to y
;
;   floating point calculations
;   fs_add       304 bytes,  110 Instructions
;   fs_mpy       194 bytes,  64  Instructions
;   fs_div       154 bytes,  52  Instructions
;   fs_itof      86  bytes,  30  Instructions
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        ;initialize COEFF array
main:   MOV.W    #COEFF,r15
        MOV.W    #0,r7                  ; i=0
L1:     MOV.W    r7,r12                 ; r12= i
        CALL     #__fs_itof             ; convert i to float

        MOV.W    #0,r14                 ; r14 and r15 will store 5.0 in float
        MOV.W    #16544,r15             ;
        CALL     #__fs_add              ; i + 5.0

        MOV.W    r12,r14                ;
        MOV.W    r13,r15                ; r14, r15 contain (i+5.0)
        MOV.W    #0,r12                 ;
        MOV.W    #16256,r13             ; r12 and r13 get 1.0
        CALL     #__fs_div              ; perform 1/(i+5.0)
                                        ; r12 and r13 contain result

        ;COEFF[i] = 1/(i+5.0)
        MOV.W    r12,0(r15)             ; lower 16 bits
        MOV.W    r13,2(r15)             ; upper 16 bits

        ADD.W    #4,r15                 ; r15 now points to next element

        ADD.W    #1,r7                  ; i++
        CMP.W    #17,r7                 ; compare with 17
        JL       L1                     ; loop back if(i<17)

        ;initialize INPUT array
        MOV.W    #0,r7                  ; i = 0
        MOV.W    #2,r8                  ; r8 = 2
L3:     MOV.W    r7,r15                 ; r15 = i
        RLA.W    r15                    ; r15 = i*size

        MOV.W    r8,68(r15)             ; INPUT[i] = 2

        ADD.W    #1,r7                  ; i++
        CMP.W    #67,r7                 ; compare with 67
        JL       L3                     ; loop back if(i<67)

        ;perform filtering
        MOV.W    #0,r10                 ; y = 0
L5:     MOV.W    #0,r7                  ; i = 0

        MOV.W    #0,r8                  ; sum = 0
        MOV.W    #0,r9                  ;

L6:     MOV.W    r7,r15                 ; r15 = i
        MOV.W    r10,r13                ; r13 = y
        SUB.W    r15,r13                ; r13 = y-i
        ADD.W    #16,r13                ; r13 = y-i+16
        RLA.W    r13                    ; r13 = (y-i+16) * size
        ;r13 now ocntains address of INPUT[y+16-i]

        MOV.W    r10,r15                ; r15 = y
        MOV.W    r7,r14                 ; r14 = i
        ADD.W    r15,r14                ; r14 = y + i
        RLA.W    r14                    ; r14 = (y + i) * size
        ;r14 now ocntains address of INPUT[y+i]

        MOV.W    68(r14),r12            ; r12 = INPUT[y+i]
        ADD.W    68(r13),r12            ; r12 = INPUT[y+16-i] + INPUT[y+i]
        CALL     #__fs_itof             ; r12, r13 now contain float representation
                                        ; of INPUT[y+16-i] + INPUT[y+i]

        MOV.W    r7,r15                 ; r15 = i
        RLA.W    r15                    ; r15 = i*2
        RLA.W    r15                    ; r15 = i*size

        MOV.W    0(r15),r14             ; r14 = lower 16 bits of COEFF[i]
        MOV.W    2(r15),r15             ; r15 = upper 16 bits of COEFF[i]
                                        ; r14, r15 now contain float representation
```

```
                                        ; of COEFF[i]

        CALL      #__fs_mpy             ; COEFF[i] * (INPUT[y+16-i] + INPUT[y+i])
                                        ; r12 and r13 contain result

        MOV.W     r8,r14                ; r14 = lower 16 bits of sum
        MOV.W     r9,r15                ; r15 = upper 16 bits of sum

        ; now perform addition sum + COEFF[i] * (INPUT[y+16-i] + INPUT[y+i])
        CALL      #__fs_add             ; result will be in r12 and r13

        ;sum is being accumulated
        ;so store back the sum for next calculation
        MOV.W     r12,r8                ; r8 = lower 16 bits of sum
        MOV.W     r13,r9                ; r9 = upper 16 bits of sum

        ADD.W     #1,r7                 ; i++
        CMP.W     #8,r7                 ; compare with 8
        JL        L6                    ; loop back if(i<8)

        MOV.W     r10,r15               ; r15 = y
        ADD.W     #8, r15               ; r15 = y+8
        RLA.W     r15                   ; r15 = (y+8) * size

        MOV.W     68(r15),r12           ; r12 = INPUT[y+8]
        CALL      #__fs_itof            ; convert r12 to float

        ;r3 is 0
        ;COEFF is at address 0
        ;so COEFF[8] will be at address 32
        MOV.W     32(R3),r14            ; r14 = lower 16 bits of COEFF[8]
        MOV.W     34(R3),r15            ; r14 = upper 16 bits of COEFF[8]

        ; now perform INPUT[y + 8] * COEFF[8]
        CALL      #__fs_mpy             ; result will be stored back in r12,r13

        MOV.W     r8,r14                ; r14 = lower 16 bits of sum
        MOV.W     r9,r15                ; r15 = lower 16 bits of sum

        ;sum + INPUT[y + 8] * COEFF[8]
        CALL      #__fs_add             ; r12 and r13 contain result

        MOV.W     r10,r15               ; r15 = y
        RLA.W     r15                   ; r15 = y*2
        RLA.W     r15                   ; r15 = y*size

        ;store sum + INPUT[y + 8] * COEFF[8] back to OUTPUT[y]
        MOV.W     r12,202(r15)          ; lower 16 bits
        MOV.W     r13,204(r15)          ; upper 16 bits

        ADD.W     #1,r10                ; y++
        CMP.W     #36,r10               ; compare with 36
        JL        L5                    ; loop back if(y<36)

        ;done with filtering
        RET                             ;return to caller
```

Listing C.18: TI MSP430 Assembly Code for Benchmark 6: FIR

C.4 ARM Cortex-M3 Assembly Codes

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   ARM Cortex-M3 Recursive Factorial Benchmark Program
;   This program recursively calculates the factorial
;   of a number (n). A number is passed to this subroutine
;   by main for factorial calculation.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   main subroutine
;
;   R0  -> number for which factorial needs to be calculated
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

Main:       MOV R0, #5      ;n=5
            BL  fact        ;call factorial subroutine
End:        BX  R14         ;return to caller

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;    Factorial subroutine
;
;    R4  ->  calculated factorial
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Fact:        PUSH    R0              ;store register on stake
             MOV     R5, R0          ;R4 = n
             SUB     R0, R0,#1       ;R0 = n-1
             BGT     L0              ;if greater then jump

             ;otherwise come here to base case calculations
             MOV     R4, #1          ;fact = 1
             POP     R0              ;restore register
             BX      R14             ;return to caller

L0:          BL      Fact            ;call factorial recursively
             MUL     R4,R4,R5        ;n*fact(n-1)
             POP     R0              ;restore register
             BX      R14             ;return to caller
```

Listing C.19: ARM Cortex-M3 Assembly Code for Benchmark 1: Recursive Factorial

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;    ARM Cortex-M3 String Copy Benchmark Program
;    In this program, main subroutine passes the addresses of source and
;    destination strings to the StrCpy subroutine to copy the chracters
;    from source to the destination string.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;    Main Subroutine
;
;    R1  ->  Source String address
;    R2  ->  Destination String address
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

Main:   MOV      R1,#Src        ;R1 = Address of source string
        MOV      R2,#Dest       ;R2 = Address of destin string

        BL       strCpy         ;call string copy subroutine

End:    BX       R14            ;return to caller

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;    String Copy Subroutine
;
;    R3  ->  temporary for current byte
;    R0  ->  loop counter
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
StrCpy: MOV      R0,#0          ;i=0
L0:     LDRB     R3,[R2,R0]     ;R3 = SrcStr[i]
        STRB     R3,[R1,R0]     ;DestStr[i] = R3

        ADD      R0,R0,#1       ;i++
        CBNZ     L0             ;loop till not null

        BX       R14            ;return to caller
```

Listing C.20: ARM Cortex-M3 Assembly Code for Benchmark 2: String Copy

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;    ARM Cortex-M3 Bubble Sort Benchmark Program
;    In this program, an array of 10 elements is initialized
;    in the main subroutine. Base address of this array is
;    passed to BSort subroutine to sort the numbers in
;    descending order.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;    Main Subroutine
;    START    ->  Base Address of Array
;    R4       ->  j i.e. loop counter
;    R1       ->  to index the array
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

Main:   MOV      R4,#0                  ;j=0
        MOV      R1,#START              ;R1 = base address of Array

L0:     STR      R4,[R1,R4,LSL #2]      ;Array[j] = j
```

```
        ADD     R4,R4,#1                ;j++
        CMP     R4,#10                  ;compare with 10
        BLT     L0                      ;loop back if(j<10)

        BL      BSort                   ;call sorting routine

End:    BX      R14                     ;return to caller

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   BSort Subroutine
;   Actual subroutine used to implement sorting Algorithm
;   Array is at Address 0X0000
;   R0  ->  j i.e. loop counter
;   R2  ->  i i.e. loop counter
;   R1  ->  has the base address of array
;   R5  ->  holds Array[j]
;   R12 ->  holds Array[j+1]
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
BSort:  MOV     R2,#8           ;i=8
L1:     MOV     R0,#0           ;j=0

L2:     LDR     R12,[R1,R0,LSL #2]      ;R12 = Array[j]
        ADD     R4,R0,#1                ;R4 = j+1
        LDR     R5,[R1,R4,LSL #2]       ;R5 = Array[j+1]
        CMP     R12,R5                  ;comparre Array[j] with Array[j+1]
        BLE     L1                      ;if less then or equal
                                        ;then no swap required

        ;otherwise swap here
        STR     R5,[R1,R0,LSL #2]       ;Array[j] = Array[j+1]
        STR     R12,[R1,R4,LSL #2]      ;Array[j+1] = Array[j]

L1:     ADD     R0,R0,#1                ;j++
        CMP     R0,R2                   ;compare with i
        BLE     L2                      ;loop back if(j<=i)

        SUB     R2,R2,#1                ;i--
        CBZ     R2,L1                   ;compare to 0 and loop if(i>=0)

        BX      R14                     ;done sorting
                                        ;return to caller
```

Listing C.21: ARM Cortex-M3 Assembly Code for Benchmark 3: Bubble Sort

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   ARM Cortex-M3 Sensor Struct Benchmark Program
;   Structure contains 3 elements:
;   1 char byte Flag indicating if sensor has been calibrated or not.
;   1 short int containing the offset to be adjusted
;   1 long int containing the actual sensor value
;
;   An array of 5 sensors is declared. InitSensors() will initialize
;   these values to some numbers. CalibrateSensors() will subtract
;   the offset from the value of the sensors and set the Flag.
;   main() will call these two functions to initialize and calibrate
;   sensor data.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Main subroutine
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Main:   BL      Init    ;call to Init subroutine
        BL      Calib   ;call to Calib subroutine
End:    BX      R14     ;return to caller

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Init subroutine
;
;   START   ->  base address of first struct member
;   R1      ->  index struct array
;   R2      ->  Data with which Value will be initialized
;   R3      ->  Flag will be initialized by R3
;   R4      ->  loop counter
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Init:   MOV     R1,#START           ;R1 = base address of Array
        MOV     R4,#0               ;i = 0
        MOV     R2,#0               ;sensor value will be initialized with R2
        MOV     R3,#0               ;R3 = 0 for flag

L0:     ADD     R2,R4,#3            ;R2 = i + 3
        STRB    R3,[R1,#0x00]       ;sensors[i].Flag = 0
        STRH    R4,[R1,#0x02]       ;sensors[i].Offset = i
```

```
        STR        R2,[R1,#0x04]      ;sensors[i].Value = i+3

        ADD        R1,R1,#6           ;point to next element

        ADD        R4,R4,#1           ;i++
        CMP        R4,#10             ;compare with 5
        BLT        L0                 ;loop back if(i<5)

        BX         R14                ;return to caller

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Calib subroutine
;
;   START    ->  Starting address of struct array
;   R1       ->  index struct array
;   R2       ->  sensors[i].Offset
;   R3       ->  sensors[i].Value
;   R4       ->  loop counter
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Calib:  MOV      R1,#START                ;R1 = base address of Array
        MOV      R4,#0                    ;i = 0
        MOV      R3,#1                    ;R3 = 1 for flag

L1:     STRB     R3,[R1,#0x00]            ;sensors[i].Flag = 1

        LDRH     R2,[R1,#0x02]            ;R2 = sensors[i].Offset
        LDR      R3,[R1,#0x04]            ;R3 = sensors[i].Value
        SUB      R2, R3, R2               ;R2 = sensors[i].Value - sensors[i].Offset
        STR      R2,[R1,#0x04]            ;sensors[i].Value = R2

        ADD      R1,R1,#6                 ;point to next element

        ADD      R4,R4,#1                 ;i++
        CMP      R4,#10                   ;compare with 5
        BLT      L1                       ;loop back if(i<5)

        BX       R14                      ;return to caller
```

Listing C.22: ARM Cortex-M3 Assembly Code for Benchmark 4: Sensor Structure

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   ARM Cortex-M3 Matrix Multiplication Benchmark Program
;   This program multiplies two matrices of order 3X4 and 4X5
;   to give a product matrix of order 3X5. Both the matrices
;   are initialized with some numbers and then multiplication
;   is performed to get product.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Main Subroutine
;   Base Address of matrix m1 -> M1
;   Base Address of matrix m2 -> M2
;   Base Address of matrix m3 -> M3
;   R10 is used to index the elements of matrix m1
;   R11 is used to index the elements of matrix m2
;   R12 is used to index the elements of matrix m3
;   R1,R3,R4,R5 ->  loop counters and temporaries
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ;fill first matrix
Main:   MOV      R12, #M1                 ;R12 =base address of m1
        MOV      R6, #nRows1              ;R6 = no of rows of m1
        MOV      R7, #nCols1              ;R7 = no of cols of m1
        BL       INIT                     ;call to INIT subroutine

        ;fill second matrix
        MOV      R12, #M2                 ;R12 =base address of m2
        MOV      R6, #nRows2              ;R6 = no of rows of m2
        MOV      R7, #nCols2              ;R7 = no of cols of m2
        BL       INIT                     ;call to INIT subroutine

        ;perform multiplication
        MOV      R10, #M1                 ;R10 =base address of m1
        MOV      R11, #M2                 ;R11 =base address of m2
        MOV      R12, #M2                 ;R12 =base address of m3

        MOV      R5,#0                    ;nCols2
L6:     MOV      R4,#0                    ;nRows1
L5:     MOV      R1,#0                    ;nCols1 (or nRows2 is same)
        MOV      R3,#0                    ;R3 = 0 (accumulator for one element)

L4:     LDR      R7,[R10]                 ;R7 = m1[m][n]
        LDR      R8,[R11]                 ;R8 = m2[n][p]
        MLA      R3,R7,R8                 ;R3 += m1[m][n] * m2[n][p]
```

```

        ADD     R11,R11,#20         ;it will point to first element of next row
        ADD     R10, R10, #4        ;points to next m1 element
        SUB     R1, R1, #1
        BLT     L4                  ;repeat 4 times

        STR     R3,[R11]            ;m3[m][p] = R3

        SUB     R11, R11, #56       ;now points to first element of next column
        ADD     R12, R12, #4        ;points to next m3 element
        SUB     R1, R4, #1
        BLT     L5                  ;repeat 5 times
        MOV     R11, #M2            ;R11 =base address of m2
        SUB     R5, R5, #1
        BLT     L6                  ;repeat 3 times

        BX      R14                 ;return to caller

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   INIT Subroutine
;   R0  -> value to be assigned
;   R1  -> row number
;   R12 -> array index
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
INIT:   MOV     R0, #0              ;R0 = 0, m+p
        MOV     R1, #0              ;R1 = 0, row number
L1:     STR     R0, [R12]           ;mat[m][p] = m+p
        ADD     R0, #1              ;R0++
        SUB     R7, #1              ;decrement col
        ADD     R12,#4              ;point to next element
        BLT     L1                  ;repeat this for nCols

        ADD     R1, #1              ;increment row
        MOV     R0, R1              ;R0 = row number, for next row

        SUB     R6, #1              ;decrement rows
        BLT     L1                  ;repeat this for nRows

        BX      R14                 ;return to caller
```

Listing C.23: ARM Cortex-M3 Assembly Code for Benchmark 5: Matrix Multiplication

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   ARM Cortex-M3 FIR Filter Benchmark Program
;   This program is an implmentation of a 17 order FIR filter.
;   COEFF and INPUT arrays are initialized with some data and
;   then FIR caculations are performed to get the OUTPUT array.
;   These calculations are basically integer and floating point
;   calculations performed on these arrays to get floating
;   result samples in OUTPUT array.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Main Subroutine
;
;   COEFF   -> starting address of COEFF Array
;   INPUT   -> starting address of INPUT Array
;   OUTPUT  -> starting address of OUTPUT Array
;
;   R10     -> to hold sum for accumulation
;   R8,R9   -> loop counters i and y respectively
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        ;initialize COEFF array
Main:   MOV     R8,#0               ;i=0
L0:     ADD     R0,R8,5             ;R0 = i+5
        BL      int2float           ;R0 = float(i+5)
        MOV     R4,#0x3f800000      ;R4 = 1.0
        BL      fdiv                ;R0 = 1/(i+5.0)

        MOV     R1,#COEFF           ;R1 = base address of COEFF
        STR     R0,[R1,R8,LSL #2]   ;COEFF[i] = 1/(i+5.0)

        ADD     R8,R8,#1            ;i++
        CMP     R8,#17              ;compare with 17
        BLT     L0                  ;loop back if(i<17)

        ;initialize INPUT array
        MOV     R8,#0               ;i = 0
L1:     MOV     R0,#2               ;R0 = 0

        MOV     R1,#INPUT           ;R1 = base address of INPUT
```

```
        STR     R0,[R1,R8,LSL #2]  ;INPUT[i] = 2

        ADD     R8,R8,#1           ;i++
        CMP     R8,#67             ;compare with 67
        BLT     L1                 ;loop back if(i<67)


        ;Perform FIR Calculations
        MOV     R9,#0                  ;y = 0
L2:     MOV     R10,#0                 ;sum = 0
        MOV     R8,#0                  ;i = 0

L3:     ADD     R1,R9,#16              ;R1 = y+16
        SUB     R1,R1,R8               ;R1 = y+16-i
        MOV     R2,#INPUT              ;R2 = base address of INPUT
        LDR     R1,[R2,R1,LSL #2]      ;R1 = INPUT[y+16-i]

        ADD     R2,R9,R8               ;R2 = y+i
        LDR     R2,[R3,R2,LSL #2]      ;R2 = INPUT[y+i]

        ADD     R0,R1,R2               ;R0 = INPUT[y+16-i] + INPUT[y+i]

        BL      int2float              ;R0 = float(R0)

        MOV     R6,#COEFF              ;R6 = base of COEFF
        LDR     R1,[R6,R8,LSL #2]      ;R1 = COEFF[i]

        ;R0 = COEFF[i] * INPUT[y+16-i] + INPUT[y+i]
        BL      fmul()

        MOV     R1,R10                 ;R1 = sum

        ;R0 = sum + COEFF[i] * INPUT[y+16-i] + INPUT[y+i]
        BL      fadd()

        MOV     R10,R0                 ;R10 = sum
                                       ;sum accumulation

        ADD     R8,R8,#1               ;i++
        CMP     R8,#8                  ;compare with 8
        BLT     L3                     ;loop back if(i<8)

        MOV     R1,#INPUT              ;R1 = base address of INPUT
        ADD     R2,R9,#8               ;R2 = y+8
        LDR     R0,[R1,R2,LSL #2]      ;R0 = INPUT[y+8]

        BL      int2float              ;R0 = float(INPUT[y+8]

        LDR     R1,[#Addr(COEFF[8]]    ;R1 = COEFF[8]

        BL      fmul                   ;R0 = INPUT[y+8] * COEFF[8]

        MOV     R1,R10                 ;R1 = sum

        BL      fadd                   ;R0 = sum + INPUT[y+8] * COEFF[8]

        MOV     R1,#OUTPUT             ;R1 = base address of OUTPUT

        ;OUTPUT[y] = sum + INPUT[y+8] * COEFF[8]
        STR     R0,[R1,R9,LSL #2]

        ADD     R9,R9,#1               ;y++
        CMP     R9,#36                 ;compare with 36
        BLT     L2                     ;loop back if(y<36)

End:    BX      R14                    ;return to caller
```

Listing C.24: ARM Cortex-M3 Assembly Code for Benchmark 6: FIR

Calculations Details D

D.1 MePoEfAr Calculations Details

Table D.1: MePoEfAr Calculations

#	Instruction			Static Results: Instr Bytes	Static Results: Instr Cycles	Static Results: DBytes Moved	Dynamic Results: # of Exec.	Dynamic Results: Exec. Cycles	Memory Traffic (Cycles): IM	Memory Traffic (Cycles): DM16	Memory Traffic (Cycles): DMem32
MePoEfAr Calculations for Benchmark 1: Recursive Factorial											
1	Main:	MOVd	#5, D0	2	1		1	1	1	0	0
2		BRS	Fact	2	2		1	2	1	0	0
3	End:	RTS		2	2		1	2	1	0	0
4	Fact:	MOVd	D0,-(SP)	2	2	2	5	10	5	5	5
5		MOVd	D0,D2	2	1.6		5	8	5	0	0
6		SUBd	#1, D0	2	1		5	5	5	0	0
7		BRgt	L0	2	1		5	5	5	0	0
8		MOVd	#1,D1	2	1		1	1	1	0	0
9		MOVd	(SP)+,D0	2	2	2	1	2	1	1	1
10		RTS		2	2		1	2	1	0	0
11	L0:	BRS	Fact	2	2		4	8	4	0	0
12		MULd	D2,D1	2	2		4	8	4	0	0
13		MOVd	(SP)+,D0	2	2	2	4	8	4	4	4
14		RTS		2	2		4	8	4	0	0
Total				**28**	**23.6**	**6**	**42**	**70**	**42**	**10**	**10**
MePoEfAr Calculations for Benchmark 2: String Copy											
1	Main:	MOVx	#Src,X4	4	2		1	2	2	0	0
2		MOVx	#Dst,X5	4	2		1	2	2	0	0
3		BRS	StrCpy	2	2		1	2	1	0	0
4	End:	HALT		2	1		1	1	1	0	0
5	StrCpy:	MOVb	(X4)+, B0	2	2	2	13	26	13	13	13
6		MOVb	B0, (X5)+	2	2	2	13	26	13	13	13
7		BRne	StrCpy	2	1		13	13	13	0	0
8		RTS		2	1		1	1	1	0	0
Total				20	13	4	44	73	46	26	26
MePoEfAr Calculations for Benchmark 3: Bubble Sort											
1	Main:	MOVd	#10, D1	2	1		1	1	1	0	0
2		MOVx	#START,X4	4	2		1	2	2	0	0
3		MOVd	#0,D2	2	1		1	1	1	0	0
4	L1:	MOVd	D2, (X4)+	2	4	4	10	40	10	20	10
5		ADDd	#1,D2	2	2		10	20	10	0	0
6		DECBRn	D1, L1	2	3		10	30	10	0	0
7		BRS	BSort	2	2		1	2	1	0	0
8	End:	HALT		2	1		1	1	1	0	0
9	BSort:	MOVx	#START,X4	4	1		1	1	2	0	0
10		MOVw	#9,W0	2	1		1	1	1	0	0

Continued on Next Page...

#	Instruction			Static Results			Dynamic Results				
				Instr	Instr	DBytes	# of	Exec.	Memory Traffic (Cycles)		
				Bytes	Cycles	Moved	Exec.	Cycles	IM	DM16	DMem32
11	L1:	MOVw	#W0,W1	2	1		9	9	9	0	0
12	L2:	MOVd	(X4)+,D2	2	4	4	45	180	45	90	45
13		MOVd	@X4,D3	2	3	4	45	135	45	90	45
14		CPAd	D2, D3	2	1		45	45	45	0	0
15		BRlt	NoSwap	2	1		45	45	45	0	0
16		MOVd	D2,@X4	2	3	4	45	135	45	90	45
17		MOVd	D3,-4(X4)	3	5	4	45	225	90	90	45
18	NoSwap:	DECBRn	W1, L2	2	2		45	90	45	0	0
19		DECBRn	W0,L1	2	2		9	18	9	0	0
20		RTS		2	1		1	1	1	0	0
Total				45	41	20	371	982	418	380	190
MePoEfAr Calculations for Benchmark 4: Sensor Structure											
1	Main:	BRS	Init	2	2		1	2	1	0	0
2		BRS	Calib	2	2		1	2	1	0	0
3	End:	RTS		2	1		1	1	1	0	0
4	Init:	MOVd	#3, D0	2	1		1	1	1	0	0
5		MOVx	#START, X4	4	2		1	2	2	0	0
6		MOVw	#0, W0	2	1		1	1	1	0	0
7		MOVb	#5, B0	2	1		1	1	1	0	0
8	L0:	MOVb	#0, (X4)+	2	2	2	5	10	5	5	5
9		MOVw	W0, (X4)+	2	2	2	5	10	5	5	5
10		ADDwds	W0, D0	3	4		5	20	10	0	0
11		MOVd	D0,(X4)+	2	4	4	5	20	5	10	5
12		ADDb	#1, W0	2	1		5	5	5	0	0
13		DECBRn	B0, L0	2	2		5	10	5	0	0
14		RTS		2	1		1	1	1	0	0
15	Calib:	MOVx	#START, X4	4	2		1	2	2	0	0
16		MOVb	#5, B0	2	1		1	1	1	0	0
17	L1:	MOVb	#1, (X4)+	2	2	2	5	10	5	5	5
18		MOVw	(X4)+, W0	2	2	2	5	10	5	5	5
19		MOVwds	W0, D0	3	2		5	10	10	0	0
20		SUBd	D0, (X4)+	2	3	4	5	15	5	10	5
21		DECBRn	B0, L0	2	2		5	10	5	0	0
22		RTS		2	1		1	1	1	0	0
Total				50	41	16	66	145	78	40	30
MePoEfAr Calculations for Benchmark 5: Matrix Multiplication											
1	Main:	MOVb	#nRows1, B6	2	1		1	1	1	0	0
2		MOVb	#nCols1, B7	2	1		1	1	1	0	0
3		MOVx	#M1, X4	4	2		1	2	2	0	0
4		BRS	INIT	2	2		1	2	1	0	0
5		MOVb	#nRows2, B6	2	1		1	1	1	0	0
6		MOVb	#nCols2, B7	2	1		1	1	1	0	0
7		MOVx	#M2, X4	4	2		1	2	2	0	0
8		BRS	INIT	2	2		1	2	1	0	0
9		Inx X4,#3	#M1,#M2,#M3	9	5		1	5	5	0	0
10		MOVb	#5, B5	2	1		1	1	1	0	0
11	L3:	MOVb	#3, B4	2	1		5	5	5	0	0
12	L2:	MOVb	#4, B1	2	1		15	15	15	0	0
13		MOVd	#0, D3	2	2		15	30	15	0	0
14	L1:	MOVd	(X4)+, D2	2	4	4	60	240	60	120	60

Continued on Next Page...

#	Instruction			Static Results			Dynamic Results				
				Instr Bytes	Instr Cycles	DBytes Moved	# of Exec.	Exec. Cycles	Memory Traffic (Cycles) IM	DM16	DMem32
15		MULd	@X3, D2	2	8	4	60	480	60	120	60
16		ADDd	D2, D3	2	2		60	120	60	0	0
17		ADDx	#20, X3	3	2		60	120	120	0	0
18		DECBRn	B1, L1	2	2		60	120	60	0	0
19		MOVd	D3, (X5)+	2	4	4	15	60	15	30	15
20		SUBx	#56, X3	3	2		15	30	30	0	0
21		DECBRn	B4, L2	2	2		15	30	15	0	0
22		MOVx	#M2, X3	4	2		5	10	10	0	0
23		DECBRn	B5, L3	2	2		5	10	5	0	0
24		RTS		2	1		1	1	1	0	0
25	INIT:	MOVd	#0, D2	2	1		2	2	2	0	0
26		MOVd	#0, D3	2	1		2	2	2	0	0
27	L1:	MOVd	D3, (X4)+	2	4	4	32	128	32	64	32
28		ADDd	#1, D3	2	1		32	32	32	0	0
29		DECBRn	B7, L1	2	2		32	64	32	0	0
30		ADDd	#1, D2	2	2		32	64	32	0	0
31		MOVd	D2, D3	2	1		32	32	32	0	0
32		DECBRn	B6, L1	2	2		32	64	32	0	0
33		RTS		2	1		2	2	2	0	0
Total				81	68	16	599	1679	685	334	167
MePoEfAr Calculations for Benchmark 6: FIR											
1	Main:	MOVx	#COEFF,X4	4	2		1	2	2	0	0
2		MOVb	#18,B0	3	2		1	2	2	0	0
3		MOVf	#5,F0	2	1		1	1	1	0	0
4	L1:	MOVf	#1,F2	2	1		18	18	18	0	0
5		DIVf	F0,F2	2	1		18	18	18	0	0
6		MOVf	F2,(X4)+	2	2	4	18	36	18	36	18
7		ADDf	#1,F0	2	1		18	18	18	0	0
8		DECBRn	B0,L1	2	1		18	18	18	0	0
9		MOVx	#INPUT,X4	4	2		1	2	2	0	0
10		MOVb	#68,B0	3	2		1	2	2	0	0
11		MOVd	#2,D2	2	1		1	1	1	0	0
12	L2:	MOVw	W2,(X4)+	2	3	2	68	204	68	68	68
13		DECBRn	B0,L2	2	1		68	68	68	0	0
14		MOVx	#COEFF,X4	4	2		1	2	2	0	0
15		MOVx	#INPUT,X2	4	2		1	2	2	0	0
16		MOVx	#OUTPUT,X6	4	2		1	2	2	0	0
17		MOVb	#36,B1	3	2		1	2	2	0	0
18	L4:	MOVb	#8,B2	2	1		36	36	36	0	0
19		MOVf	#0,F1	2	1		36	36	36	0	0
20	L3:	MOVx	#16,X3	3	2		304	608	608	0	0
21		SUBbx	B2,X3	3	2		304	608	608	0	0
22		ADDbx	B1,X3	3	2		304	608	608	0	0
23		MULx	#2,X3	2	1		304	304	304	0	0
24		ADDx	X2,X3	2	1		304	304	304	0	0
25		MOVw	@X3,W3	2	2	2	304	608	304	304	304
26		MOVbxs	B1,X3	3	2		304	608	608	0	0
27		ADDbxs	B2,X3	3	2		304	608	608	0	0
28		MULx	#2,X3	2	1		304	304	304	0	0
29		ADDx	X2,X3	2	1		304	304	304	0	0

Continued on Next Page…

#	Instruction			Static Results			Dynamic Results				
				Instr	Instr	DBytes	# of	Exec.	Memory Traffic (Cycles)		
				Bytes	Cycles	Moved	Exec.	Cycles	IM	DM16	DMem32
30		ADDd	@X3,W3	2	2	2	304	608	304	304	304
31		MOVwfs	W3,F3	3	2		304	608	608	0	0
32		MULdfs	(X4)+,F3	3	2	4	304	608	608	608	304
33		ADDf	F3,F1	2	1		304	304	304	0	0
34		DECBRn	B2,L3	2	2		304	608	304	0	0
35		MOVf	32(X0),F5	3	4	4	36	144	72	72	36
36		MOVx	#8,X3	2	1		36	36	36	0	0
37		ADDbxs	B1,X3	3	2		36	72	72	0	0
38		MULx	#4,X3	2	1		36	36	36	0	0
39		ADDx	X2,X3	2	1		36	36	36	0	0
40		MOVw	@X3,W3	2	2	4	36	72	36	72	36
41		MULwfs	W3,F5	3	1		36	36	72	0	0
42		ADDf	F1,F5	2	1		36	36	36	0	0
43		MOVf	F5,(X6)+	2	2	4	36	72	36	72	36
44		DECBRn	B1,L4	2	2		36	72	36	0	0
45	End:	RTS		2	1		1	1	1	0	0
Total				113	73	26	5229	8683	7473	1536	1106

D.2 Atmel AVR AT90S851 Calculations Details

here will be the code for benchmark 2

Table D.2: Atmel AVR Calculations

#	Instruction			Static Results			Dynamic Results			
				Instr.	Instr.	DBytes	No. of	Exec.	Memory Traffic (Cycles)	
				Bytes	Cycles	Moved	Exec.	Cycles	Instr. Mem	Data Mem
Atmel AVR Calculations for Benchmark 1: Recursive Factorial										
1	Main:	LDI	R18,0x05	2	1		1	1	1	0
2		LDI	R19,0x00	2	1		1	1	1	0
3		RCALL	Fact	2	3		1	3	1	0
4		RET		2	4		1	4	1	0
5	Fact:	PUSH	R18	2	2	1	5	10	5	5
6		PUSH	R19	2	2	1	5	10	5	5
7		CPI	R18,0x02	2	1		5	5	5	0
8		CPC	R19,R1	2	1		5	5	5	0
9		BRGE	L0	2	1		5	5	5	0
10		LDI	R22,0x01	2	1		1	1	1	0
11		LDI	R23,0x00	2	1		1	1	1	0
12		LDI	R24,0x00	2	1		1	1	1	0
13		LDI	R25,0x00	2	1		1	1	1	0
14		RJMP	L1	2	1		1	1	1	0
15	L0:	MOV	R20,R18	2	1		4	4	4	0
16		MOV	R21,R19	2	1		4	4	4	0
17		SBIW	R18,0x01	2	1		4	4	4	0
18		RCALL	Fact	2	3		4	12	4	0
19		MOV	R18,R20	2	1		4	4	4	0

Continued on Next Page...

#	Instruction			Static Results			Dynamic Results			
				Instr. Bytes	Instr. Cycles	DBytes Moved	No. of Exec.	Exec. Cycles	Memory Traffic (Cycles) Instr. Mem	Data Mem
20		MOV	R19,R21	2	1		4	4	4	0
21		CLR	R20	2	1		4	4	4	0
22		CLR	R21	2	1		4	4	4	0
23	(27,50)	RCALL	Mult32	50	43		4	172	100	0
24	L1:	POP	R19	2	2	1	5	10	5	5
25		POP	R18	2	2	1	5	10	5	5
53		RET		2	4		1	4	1	0
Total				100	82	4	81	285	177	20
Atmel AVR Calculations for Benchmark 2: String Copy										
1	Main:	LDI	R30,0x60	2	1		1	1	1	0
2		LDI	R31,0x00	2	1		1	1	1	0
3		LDI	R28,0x70	2	1		1	1	1	0
4		LDI	R29,0x00	2	1		1	1	1	0
5		RCALL	strCopy	2	3		1	3	1	0
6		RET		2	4		1	4	1	0
7	strCopy:	LD	R24,Z+	2	2	1	13	26	13	13
8		ST	Y+,R24	2	2	1	13	26	13	13
9		TST	R24	2	1		13	13	13	0
10		BRNE	strCopy	2	1		13	13	13	0
11		RET		2	4		1	4	1	0
Total				22	21	2	59	93	59	26
Atmel AVR Calculations for Benchmark 3:Bubble Sort										
1	Main:	LDI	R30,0x00	2	1		1	1	1	0
2		LDI	R31,0x00	2	1		1	1	1	0
3		LDI	R24,0x00	2	1		1	1	1	0
4		LDI	R25,0x00	2	1		1	1	1	0
5	L0:	ST	Z+,R24	2	2	1	10	20	10	10
6		ST	Z+,R25	2	2	1	10	20	10	10
7		ADIW	R24,0x01	2	1		10	10	10	0
8		CPI	R24,0x0A	2	1		10	10	10	0
9		CPC	R25,R1	2	1		10	10	10	0
10		BRNE	L0	2	1		10	10	10	0
11		RCALL	BSort	2	3		1	3	1	0
12		RET		2	4		1	4	1	0
13	BSort:	MOV	R30,R24	2	1		1	1	1	0
14		MOV	R31,R25	2	1		1	1	1	0
15		LDI	R18,0x08	2	1		1	1	1	0
16		LDI	R19,0x00	2	1		1	1	1	0
17	L2:	LDI	R20,0x00	2	1		9	9	9	0
18		LDI	R21,0x00	2	1		9	9	9	0
19	L1:	LDD	R22,Z+0	2	1	1	45	45	45	45
20		LDD	R23,Z+1	2	1	1	45	45	45	45
21		LDD	R26,Z+2	2	1	1	45	45	45	45
22		LDD	R27,Z+3	2	1	1	45	45	45	45
23		CP	R26,R22	2	1		45	45	45	0
24		CPC	R27,R23	2	1		45	45	45	0
25		BRGE	NoSwap	2	1		45	45	45	0
26		STD	Z+1,R27	2	1	1	45	45	45	45
27		STD	Z+0,R26	2	1	1	45	45	45	45
28		STD	Z+3,R23	2	1	1	45	45	45	45

Continued on Next Page...

#	Instruction			Static Results			Dynamic Results			
				Instr.	Instr.	DBytes	No. of	Exec.	Memory Traffic (Cycles)	
				Bytes	Cycles	Moved	Exec.	Cycles	Instr. Mem	Data Mem
29		STD	Z+2,R22	2	1	1	45	45	45	45
30	NoSwap:	SUBI	R20,0xFF	2	1		45	45	45	0
31		SBCI	R21,0xFF	2	1		45	45	45	0
32		ADIW	R30,0x02	2	1		45	45	45	0
33		CP	R18,R20	2	1		45	45	45	0
34		CPC	R19,R21	2	1		45	45	45	0
35		BRGE	L1	2	1		45	45	45	0
36		SUBI	R18,0x01	2	1		9	9	9	0
37		SBCI	R19,0x00	2	1		9	9	9	0
38		SER	R20	2	1		9	9	9	0
39		CPI	R18,0xFF	2	1		9	9	9	0
40		CPC	R19,R20	2	1		9	9	9	0
41		BRNE	L2	2	1		9	9	9	0
42		RET		2	4		1	4	1	0
Total				84	52	10	908	936	908	380
Atmel AVR Calculations for Benchmark 4: Sensor Structure										
1	Main:	RCALL	Init	2	3		1	3	1	0
2		RCALL	Calib	2	3		1	3	1	0
3	End:	RET		2	4		1	4	1	0
4	Init:	MOV	R30,#STLo	2	1		1	1	1	0
5		MOV	R31,#STHi	2	1		1	1	1	0
6		MOV	R20,#3	2	1		1	1	1	0
7		CLR	R21	2	1		1	1	1	0
8		CLR	R22	2	1		1	1	1	0
9		CLR	R23	2	1		1	1	1	0
10		MOV	R15,#0	2	1		1	1	1	0
11		MOV	R16,#0	2	1		1	1	1	0
12	L0:	STD	Z+,#0	2	2	1	5	10	5	5
13		STD	Z+,R15	2	2	1	5	10	5	5
14		STD	Z+,R16	2	2	1	5	10	5	5
15		INC	R20	2	1		5	5	5	0
16		ADC	R21,R16	2	1		5	5	5	0
17		ADC	R22,#0	2	1		5	5	5	0
18		ADC	R23,#0	2	1		5	5	5	0
19		STD	z+,R20	2	2	1	5	10	5	5
20		STD	z+,R21	2	2	1	5	10	5	5
21		STD	z+,R22	2	2	1	5	10	5	5
22		STD	z+,R23	2	2	1	5	10	5	5
23		INC	R15	2	1		5	5	5	0
24		CPI	R15,#5	2	1		5	5	5	0
25		BRNE	L0	2	1		5	5	5	0
26		RET		2	4		1	4	1	0
27	Calib:	MOV	R30,#STLo	2	1		1	1	1	0
28		MOV	R31,#STHi	2	1		1	1	1	0
29		MOV	R15,#5	2	1		1	1	1	0
30	L1:	STD	Z+,#1	2	2	1	5	10	5	5
31		MOV	R18,z+	2	2	1	5	10	5	5
32		MOV	R19,z+	2	2	1	5	10	5	5
33		SUB	z+,R18	2	2	2	5	10	5	10
34		SBC	z+,R19	2	2	2	5	10	5	10

Continued on Next Page. . .

#	Instruction			Static Results			Dynamic Results			
				Instr. Bytes	Instr. Cycles	DBytes Moved	No. of Exec.	Exec. Cycles	Memory Traffic (Cycles) Instr. Mem	Data Mem
35		SBCI	z+,#0	2	2	2	5	10	5	10
36		SBCI	z+,#0	2	2	2	5	10	5	10
37		DEC	R15	2	1		5	5	5	0
38		BRNE	L1	2	1		5	5	5	0
39		RET		2	4		1	4	1	0
Total				78	66	18	131	214	131	90
Atmel AVR Calculations for Benchmark 5: Matrix Multiplication										
1	Main:	MOV	R15,#nRows1	2	1		1	1	1	0
2		MOV	R16,#nCols1	2	1		1	1	1	0
3		MOV	R30,#M1Lo	2	1		1	1	1	0
4		MOV	R31,#M1Hi	2	1		1	1	1	0
5		RCALL	INIT	2	3		1	3	1	0
6		MOV	R15,#nRows2	2	1		1	1	1	0
7		MOV	R16,#nCols2	2	1		1	1	1	0
8		MOV	R30,#M2Lo	2	1		1	1	1	0
9		MOV	R31,#M2Hi	2	1		1	1	1	0
10		RCALL	INIT	2	3		1	3	1	0
11		MOV	R26,#M1Lo	2	1		1	1	1	0
12		MOV	R27,#M1Hi	2	1		1	1	1	0
13		MOV	R28,#M2Lo	2	1		1	1	1	0
14		MOV	R29,#M2Hi	2	1		1	1	1	0
15		MOV	R30,#M3Lo	2	1		1	1	1	0
16		MOV	R31,#M3Hi	2	1		1	1	1	0
17		MOV	R15,#5	2	1		1	1	1	0
18	L3:	MOV	R16,#3	2	1		5	5	5	0
19	L2:	MOV	R17,#4	2	1		15	15	15	0
20		LDI	Z+0,#0	2	1	1	15	15	15	15
21		LDI	Z+1,#0	2	1	1	15	15	15	15
22		LDI	Z+2,#0	2	1	1	15	15	15	15
23		LDI	Z+3,#0	2	1	1	15	15	15	15
24	L1:	MOV	R18,X+	2	2	1	60	120	60	60
25		MOV	R19,X+	2	2	1	60	120	60	60
26		MOV	R20,X+	2	2	1	60	120	60	60
27		MOV	R21,X+	2	2	1	60	120	60	60
28		MOV	R22,Y+	2	2	1	60	120	60	60
29		MOV	R23,Y+	2	2	1	60	120	60	60
30		MOV	R24,Y+	2	2	1	60	120	60	60
31		MOV	R25,Y+	2	2	1	60	120	60	60
32	(35,50)	RCALL	Mult32	72	53		60	3180	2160	0
33		ADD	Z+0,R22	2	2	1	60	120	60	60
34		ADD	Z+1,R23	2	2	1	60	120	60	60
35		ADD	Z+2,R24	2	2	1	60	120	60	60
36		ADD	Z+3,R25	2	2	1	60	120	60	60
37		ADIW	R29:R28,#20	2	2		60	120	60	0
38		DEC	R17	2	1		60	60	60	0
39		BRNE	L1	2	1		60	60	60	0
40		ADIW	R31:R30,#4	2	2		15	30	15	0
41		SBIW	R29:R28,#56	2	2		15	30	15	0
42		DEC	R16	2	1		15	15	15	0
43		BRNE	L2	2	1		15	15	15	0

Continued on Next Page. . .

#	Instruction			Static Results			Dynamic Results			
				Instr.	Instr.	DBytes	No. of	Exec.	Memory Traffic (Cycles)	
				Bytes	Cycles	Moved	Exec.	Cycles	Instr. Mem	Data Mem
44		MOV	R28,#M2Lo	2	1		5	5	5	0
45		MOV	R29,#M2Hi	2	1		5	5	5	0
46		DEC	R15	2	1		5	5	5	0
47		BRNE	L3	2	1		5	5	5	0
48		RET		2	4		1	4	1	0
49	INIT:	CLR	R23	2	1		2	2	2	0
50		CLR	R24	2	1		2	2	2	0
51		CLR	R25	2	1		2	2	2	0
52		CLR	R26	2	1		2	2	2	0
53		CLR	R17	2	1		2	2	2	0
54	L1:	STD	Z+0,R23	2	2	1	32	64	32	32
55		STD	Z+1,R24	2	2	1	32	64	32	32
56		STD	Z+2,R25	2	2	1	32	64	32	32
57		STD	Z+3,R26	2	2	1	32	64	32	32
58		ADD	R23,#1	2	1		32	32	32	0
59		ADC	R24,#0	2	1		32	32	32	0
60		ADC	R25,#0	2	1		32	32	32	0
61		ADC	R26,#0	2	1		32	32	32	0
62		ADI	R30,#4	2	1		32	32	32	0
63		ADC	R31,#0	2	1		32	32	32	0
64		DEC	R16	2	1		32	32	32	0
65		BRGT	L1	2	1		32	32	32	0
66		INC	R17	2	1		32	32	32	0
67		MOV	R17,R23	2	1		32	32	32	0
68		DEC	R15	2	1		32	32	32	0
69		BRGT	L1	2	1		32	32	32	0
105		RET		2	4		2	8	2	0
Total				210	151	20	1662	5733	3762	908
Atmel AVR Calculations for Benchmark 6: FIR										
1	Main:	STD	Y+8,R1	2	2	1	1	2	1	1
2		STD	Y+7,R1	2	2	1	1	2	1	1
3	L0:	LDD	R16,Y+7	2	2	1	18	36	18	18
4		LDD	R17,Y+8	2	2	1	18	36	18	18
5		LDD	R22,Y+7	2	2	1	18	36	18	18
6		LDD	R23,Y+8	2	2	1	18	36	18	18
7		CLR	R24	2	1		18	18	18	0
8		CLR	R25	2	1		18	18	18	0
9		RCALL	INT2FLOAT	2	3		18	54	18	0
10		LDI	R18,0x00	2	1		18	18	18	0
11		LDI	R19,0x00	2	1		18	18	18	0
12		LDI	R20,0xA0	2	1		18	18	18	0
13		LDI	R21,0x40	2	1		18	18	18	0
14		RCALL	FADD	2	3		18	54	18	0
15		LDI	R18,0x00	2	1		18	18	18	0
16		LDI	R19,0x00	2	1		18	18	18	0
17		LDI	R20,0x80	2	1		18	18	18	0
18		LDI	R22,0x3F	2	1		18	18	18	0
19		RCALL	FDIV	2	3		18	54	18	0
20		MOV	R30,R16	2	1		18	18	18	0
21		MOV	R31,R17	2	1		18	18	18	0

Continued on Next Page...

#	Instruction			Static Results			Dynamic Results			
				Instr. Bytes	Instr. Cycles	DBytes Moved	No. of Exec.	Exec. Cycles	Memory Traffic (Cycles) Instr. Mem	Memory Traffic (Cycles) Data Mem
22		LSL	R30	2	1		18	18	18	0
23		ROL	R31	2	1		18	18	18	0
24		LSL	R30	2	1		18	18	18	0
25		ROL	R31	2	1		18	18	18	0
26		LDI	R20,0x00	2	1		18	18	18	0
27		LDI	R21,0x00	2	1		18	18	18	0
28		ADD	R30,R20	2	1		18	18	18	0
29		ADC	R31,R21	2	1		18	18	18	0
30		STD	Z+0,R24	2	2	1	18	36	18	18
31		STD	Z+1,R25	2	2	1	18	36	18	18
32		STD	Z+2,R26	2	2	1	18	36	18	18
33		STD	Z+3,R27	2	2	1	18	36	18	18
34		LDD	R24,Y+7	2	2		18	36	18	0
35		LDD	R25,Y+8	2	2		18	36	18	0
36		ADIW	R24,0x01	2	1		18	18	18	0
37		STD	Y+8,R25	2	2	1	18	36	18	18
38		STD	Y+7,R24	2	2	1	18	36	18	18
39		CPI	R24,0x11	2	1		18	18	18	0
40		CPC	R25,R1	2	1		18	18	18	0
41		BRLT	L0	2	1		18	18	18	0
42		STD	Y+8,R1	2	2	1	1	2	1	1
43		STD	Y+7,R1	2	2	1	1	2	1	1
44	L1:	LDD	R30,Y+7	2	2	1	36	72	36	36
45		LDD	R31,Y+8	2	2	1	36	72	36	36
46		SUBI	R18,0x00	2	1		36	36	36	0
47		SBCI	R19,0x01	2	1		36	36	36	0
48		LSL	R30	2	1		36	36	36	0
49		ROL	R31	2	1		36	36	36	0
50		ADD	R30,R18	2	1		36	36	36	0
51		ADC	R31,R19	2	1		36	36	36	0
52		LDI	R18,0x02	2	1		36	36	36	0
53		LDI	R19,0x00	2	1		36	36	36	0
54		STD	Z+1,R19	2	2	1	36	72	36	36
55		STD	Z+0,R18	2	2	1	36	72	36	36
56		LDD	R24,Y+7	2	2	1	36	72	36	36
57		LDD	R25,Y+8	2	2	1	36	72	36	36
58		ADIW	R24,0x01	2	1		36	36	36	0
59		STD	Y+8,R25	2	2	1	36	72	36	36
60		STD	Y+7,R24	2	2	1	36	72	36	36
61		CPI	R24,0x43	2	1		36	36	36	0
62		CPC	R25,R1	2	1		36	36	36	0
63		BRLT	L1	2	1		36	36	36	0
64		STD	Y+6,R1	2	2		1	2	1	0
65		STD	Y+5,R1	2	2		1	2	1	0
66	L2:	LDI	R24,0x00	2	1		36	36	36	0
67		LDI	R25,0x00	2	1		36	36	36	0
68		LDI	R26,0x00	2	1		36	36	36	0
69		LDI	R27,0x00	2	1		36	36	36	0
70		STD	Y+1,R24	2	2	1	36	72	36	36
71		STD	Y+2,R25	2	2	1	36	72	36	36

Continued on Next Page...

#	Instruction			Static Results			Dynamic Results			
				Instr. Bytes	Instr. Cycles	DBytes Moved	No. of Exec.	Exec. Cycles	Memory Traffic (Cycles) Instr. Mem	Data Mem
72		STD	Y+3,R26	2	2	1	36	72	36	36
73		STD	Y+4,R27	2	2	1	36	72	36	36
74		STD	Y+8,R1	2	2	1	36	72	36	36
75		STD	Y+7,R1	2	2	1	36	72	36	36
76	L3:	LDD	R30,Y+7	2	2	1	304	608	304	304
77		LDD	R31,Y+8	2	2	1	304	608	304	304
78		LSL	R30	2	1		304	304	304	0
79		ROL	R31	2	1		304	304	304	0
80		LSL	R30	2	1		304	304	304	0
81		ROL	R31	2	1		304	304	304	0
82		ADDI	R30,0x00	2	1		304	304	304	0
83		ADIC	R31,0x00	2	1		304	304	304	0
84		LDD	R14,Z+0	2	2	1	304	608	304	304
85		LDD	R15,Z+1	2	2	1	304	608	304	304
86		LDD	R16,Z+2	2	2	1	304	608	304	304
87		LDD	R17,Z+3	2	2	1	304	608	304	304
88		LDD	R24,Y+5	2	2	1	304	608	304	304
89		LDD	R25,Y+6	2	2	1	304	608	304	304
90		LDD	R24,Y+7	2	2	1	304	608	304	304
91		LDD	R25,Y+8	2	2	1	304	608	304	304
92		SUB	R30,R24	2	1		304	304	304	0
93		SBC	R31,R25	2	1		304	304	304	0
94		ADDI	R30,0x00	2	1		304	304	304	0
95		ADCI	R31,0x10	2	1		304	304	304	0
96		LSL	R30	2	1		304	304	304	0
97		ROL	R31	2	1		304	304	304	0
98		ADDI	R30,0x00	2	1		304	304	304	0
99		ADCI	R31,0x01	2	1		304	304	304	0
100		LDD	R18,Z+0	2	2	1	304	608	304	304
101		LDD	R19,Z+1	2	2	1	304	608	304	304
102		LDD	R20,Y+5	2	2	1	304	608	304	304
103		LDD	R21,Y+6	2	2	1	304	608	304	304
104		LDD	R30,Y+7	2	2	1	304	608	304	304
105		LDD	R31,Y+8	2	2	1	304	608	304	304
106		ADD	R30,R20	2	1		304	304	304	0
107		ADC	R31,R21	2	1		304	304	304	0
108		LSL	R30	2	1		304	304	304	0
109		ROL	R31	2	1		304	304	304	0
110		ADDI	R30,0x00	2	1		304	304	304	0
111		ADCI	R31,0x01	2	1		304	304	304	0
112		LDD	R24,Z+0	2	2	1	304	608	304	304
113		LDD	R25,Z+1	2	2	1	304	608	304	304
114		ADD	R18,R24	2	1		304	304	304	0
115		ADC	R19,R25	2	1		304	304	304	0
116		MOV	R22,R18	2	1		304	304	304	0
117		MOV	R23,R19	2	1		304	304	304	0
118		CLR	R24	2	1		304	304	304	0
119		SBRC	R23,7	2	1		304	304	304	0
120		LAT	R24	2	1		304	304	304	0
121		MOV	R25,R24	2	1		304	304	304	0

Continued on Next Page...

#	Instruction			Static Results			Dynamic Results			
				Instr. Bytes	Instr. Cycles	DBytes Moved	No. of Exec.	Exec. Cycles	Memory Traffic (Cycles) Instr. Mem	Data Mem
122		RCALL	INT2FLOAT	2	3		304	912	304	0
123		MOV	R18,R14	2	1		304	304	304	0
124		MOV	R19,R15	2	1		304	304	304	0
125		MOV	R20,R16	2	1		304	304	304	0
126		MOV	R21,R17	2	1		304	304	304	0
127		RCALL	FMUL	2	3		304	912	304	0
128		LDD	R18,Y+1	2	2	1	304	608	304	304
129		LDD	R19,Y+2	2	2	1	304	608	304	304
130		LDD	R20,Y+3	2	2	1	304	608	304	304
131		LDD	R21,Y+4	2	2	1	304	608	304	304
132		RCALL	FADD	2	3		304	912	304	0
133		STD	Y+1,R22	2	2	1	304	608	304	304
134		STD	Y+2,R23	2	2	1	304	608	304	304
135		STD	Y+3,R24	2	2	1	304	608	304	304
136		STD	Y+4,R25	2	2	1	304	608	304	304
137		LDD	R24,Y+7	2	2	1	304	608	304	304
138		LDD	R25,Y+8	2	2	1	304	608	304	304
139		ADIW	R24,0x01	2	1		304	304	304	0
140		STD	Y+8,R25	2	2	1	304	608	304	304
141		STD	Y+7,R24	2	2	1	304	608	304	304
142		CPI	R24,0x08	2	1		304	304	304	0
143		CPC	R25,R1	2	1		304	304	304	0
144		BRGE	L3	2	1		304	304	304	0
145		LDD	R30,Y+5	2	2	1	36	72	36	36
146		LDD	R31,Y+6	2	2	1	36	72	36	36
147		ADIW	R30,0x08	2	1		36	36	36	0
148		LSL	R30	2	1		36	36	36	0
149		ROL	R31	2	1		36	36	36	0
150		ADDI	R30,0x00	2	1		36	36	36	0
151		ADCI	R31,0x01	2	1		36	36	36	0
152		LDD	R22,Z+0	2	2	1	36	72	36	36
153		LDD	R23,Z+1	2	2	1	36	72	36	36
154		CLR	R24	2	1		36	36	36	0
155		SBRC	R23,7	2	1		36	36	36	0
156		LAT	R24	2	1		36	36	36	0
157		MOV	R25,R24	2	1		36	36	36	0
158		RCALL	INT2FLOAT	2	3		36	108	36	0
159		LDD	R18,Y+41	2	2	1	36	72	36	36
160		LDD	R19,Y+42	2	2	1	36	72	36	36
161		LDD	R20,Y+43	2	2	1	36	72	36	36
162		LDD	R21,Y+44	2	2	1	36	72	36	36
163		RCALL	FMUL	2	3		36	108	36	0
164		LDD	R18,Y+1	2	2	1	36	72	36	36
165		LDD	R19,Y+2	2	2	1	36	72	36	36
166		LDD	R20,Y+3	2	2	1	36	72	36	36
167		LDD	R21,Y+4	2	2	1	36	72	36	36
168		RCALL	FADD	2	3		36	108	36	0
169		LDD	R30,Y+5	2	2	1	36	72	36	36
170		LDD	R31,Y+6	2	2	1	36	72	36	36
171		LSL	R30	2	1		36	36	36	0

Continued on Next Page...

#	Instruction			Static Results			Dynamic Results			
				Instr.	Instr.	DBytes	No. of	Exec.	Memory Traffic (Cycles)	
				Bytes	Cycles	Moved	Exec.	Cycles	Instr. Mem	Data Mem
172		ROL	R31	2	1		36	36	36	0
173		LSL	R30	2	1		36	36	36	0
174		ROL	R31	2	1		36	36	36	0
175		ADDI	R30,0x00	2	1		36	36	36	0
176		ADCI	R31,0x02	2	1		36	36	36	0
177		STD	Z+0,R22	2	2	1	36	72	36	36
178		STD	Z+1,R23	2	2	1	36	72	36	36
179		STD	Z+2,R24	2	2	1	36	72	36	36
180		STD	Z+3,R25	2	2	1	36	72	36	36
181		LDD	R24,Y+5	2	2	1	36	72	36	36
182		LDD	R25,Y+6	2	2	1	36	72	36	36
183		ADIW	R24,0x01	2	1		36	36	36	0
184		STD	Y+6,R25	2	2	1	36	72	36	36
185		STD	Y+5,R24	2	2	1	36	72	36	36
186		CPI	R24,0x24	2	1		36	36	36	0
187		CPC	R25,R1	2	1		36	36	36	0
188		BRLT	L2	2	1		36	36	36	0
189		RET		2	4		1	4	1	0
Total				378	294	80	24349	37138	24349	10600

D.3 TI MSP430G2231 Calculations Details

Table D.3: TI MSP430 Calculations

#	Instruction			Static Results			Dynamic Results			
				Instr.	Instr.	DBytes	No. of	Exec.	Memory Traffic (Cycles)	
				Bytes	Cycles	Moved	Exec.	Cycles	Instr. Mem	Data Mem
TI MSP430 Calculations for Benchmark 1: Recursive Factorial										
1	Main:	MOV.W	#5,r12	4	2		1	2	2	0
2		CALL	#Fact	4	5		1	5	2	0
3	End:	RET		2	3		1	3	1	0
4	Fact:	PUSH.W	r12	2	3	2	5	15	5	5
5		MOV.W	r12,r10	2	1		5	5	5	0
6		SUB.W	#1,r12	2	1		5	5	5	0
7		JGE	L1	2	2		5	10	5	0
8		MOV.W	#1,r14	2	1		1	1	1	0
9		MOV.W	#0,r15	2	1		1	1	1	0
10		POP	r12	2	3	2	1	3	1	1
11		RET		2	3		1	3	1	0
12	L1:	CALL	#Fact	4	5		4	20	8	0
13		MOV.W	r14,r4	2	1		4	4	4	0
14		MOV.W	r15,r5	2	1		4	4	4	0
15		CLR	r14	2	1		4	4	4	0
16		CLR	r15	2	1		4	4	4	0
17		MOV	r4,&0130h	4	4		4	16	8	0
18		MOV	r10,&0138h	4	4		4	16	8	0
19		MOV	&SumLo,r14	4	4		4	16	8	0

Continued on Next Page. . .

#	Instruction			Static Results			Dynamic Results			
				Instr.	Instr.	DBytes	No. of	Exec.	Memory Traffic (Cycles)	
				Bytes	Cycles	Moved	Exec.	Cycles	Instr. Mem	Data Mem
20		MOV	&SumHi,r15	4	4		4	16	8	0
21		MOV	r10,&0130h	4	4		4	16	8	0
22		MOV	r5,&0138h	4	4		4	16	8	0
23		ADD	&SumLo,r14	4	4		4	16	8	0
24		ADDC	&SumHi,r15	4	4		4	16	8	0
25		POP	r12	2	3	2	4	12	4	4
26		RET		2	3		4	12	4	0
Total				74	72	6	87	241	125	10
TI MSP430 Calculations for Benchmark 2: String Copy										
1	Main:	MOV.W	#strSrc,r12	4	2		1	2	2	0
2		MOV.W	#strDest,r15	4	2		1	2	2	0
3		CALL	#strCopy	4	5		1	5	2	0
4	End:	RET		2	3		1	3	1	0
5	strCopy:	MOV.B	@r12+,0(r15)	4	5	2	13	65	26	13
6		ADD.B	#1,r15	2	1		13	13	13	0
7		TST.B	0(r15)	2	2	2	13	26	13	13
8		JNE	strCopy	2	2		13	26	13	0
9		RET		2	3		1	3	1	0
Total				26	25	4	57	145	73	26
TI MSP430 Calculations for Benchmark 3: Bubble Sort										
1	Main:	MOV.W	#0,r13	2	1		1	1	1	0
2		MOV.W	#0,r14	2	1		1	1	1	0
3		MOV.W	#START,r15	4	2		1	2	2	0
4	L1:	MOV.W	r13,0(r15)	4	2	2	10	20	20	10
5		MOV.W	r14,2(r15)	4	4	2	10	40	20	10
6		ADD.W	#4,r15	2	1		10	10	10	0
7		ADD.W	#1,r13	2	1		10	10	10	0
8		CMP.W	#10,r13	4	2		10	20	20	0
9		JL	L1	2	2		10	20	10	0
10		CALL	#BSort	4	5		1	5	2	0
11	End:	RET		2	3		1	3	1	0
12	BSort:	MOV.W	#8,r13	4	2		1	2	2	0
13		MOV.W	#START,r15	4	2		1	2	2	0
14	L3:	MOV.W	#0,r9	2	1		9	9	9	0
15	L4:	MOV.W	@r15,r10	6	6	2	45	270	135	45
16		MOV.W	2(r15),r11	2	2	2	45	90	45	45
17		CMP.W	6(r15),r11	2	2	2	45	90	45	45
18		JL	L5	6	6		45	270	135	0
19		JNE	L6	2	2		45	90	45	0
20		CMP.W	4(r15),r10	2	2	2	45	90	45	45
21		JHS	L6	4	3		45	135	90	0
22	L5:	MOV.W	4(r15),0(r15)	6	6	2	45	270	135	45
23		MOV.W	6(r15),2(r15)	6	6	2	45	270	135	45
24		MOV.W	r10,4(r15)	4	4	2	45	180	90	45
25		MOV.W	r11,6(r15)	4	3	2	45	135	90	45
26	L6:	ADD.W	#4,r15	2	1		45	45	45	0
27		ADD.W	#1,r9	2	1		45	45	45	0
28		CMP.W	r9,r13	2	1		45	45	45	0
29		JGE	L4	2	2		45	90	45	0
30	L7:	SUB.W	#1,r13	2	1		9	9	9	0

Continued on Next Page...

#	Instruction			Static Results			Dynamic Results			
				Instr.	Instr.	DBytes	No. of	Exec.	Memory Traffic (Cycles)	
				Bytes	Cycles	Moved	Exec.	Cycles	Instr. Mem	Data Mem
31		TST.W	r13	2	1		9	9	9	0
32		JGE	L3	2	2		9	18	9	0
33	L8:	RET		2	3		1	3	1	0
Total				102	83	20	779	2299	1308	380
TI MSP430 Calculations for Benchmark 4: Sensor Struct										
1	Main:	CALL	Init	4	5		1	5	2	0
2		CALL	Calib	4	5		1	5	2	0
3	End:	RET		2	3		1	3	1	0
4	Init:	MOV.W	#0, r8	2	1		1	1	1	0
5		MOV.W	#0, r9	2	1		1	1	1	0
6		MOV.W	#START, r15	4	2		1	2	2	0
7		MOV.W	#0, r4	2	1		1	1	1	0
8	L0:	MOV.B	#0, 0(r15)	4	2	2	5	10	10	5
9		INC.B	r15	2	1		5	5	5	0
10		MOV.W	r4, 0(r15)	4	2	2	5	10	10	5
11		MOV.W	r4, r8	2	1		5	5	5	0
12		ADC.W	#3, r9	4	2		5	10	10	0
13		MOV.W	r8,2(r15)	4	4	2	5	20	10	5
14		MOV.W	r8,4(r15)	4	4	2	5	20	10	5
15		ADD.W	#6,r15	4	2		5	10	10	0
16		ADD.W	#1, r4	2	1		5	5	5	0
17		CMP.W	#5, r4	4	2		5	10	10	0
18		JL	L0	2	2		5	10	5	0
19		RET		2	3		1	3	1	0
20	Calib:	MOV.W	#START, r15	4	2		1	2	2	0
21		MOV.W	#5, r4	4	2		1	2	2	0
22	L1:	MOV.B	#0, 0(r15)	2	2	2	5	10	5	5
23		INC.B	r15	2	1		5	5	5	0
24		SUB.W	0(r15),2(r15)	6	6	4	5	30	15	10
25		SUBC.W	#0,4(r15)	4	2	2	5	10	10	5
26		ADD.W	#6,r15	4	1		5	5	10	0
27		DEC.W	r4	2	1		5	5	5	0
28		JG	L0	2	2		5	10	5	0
29		RET		2	3		1	3	1	0
Total				90	66	16	101	218	161	40
TI MSP430 Calculations for Benchmark 5: Matrix Multiplication										
1	Main:	MOV.W	#nRows1, r6	2	1		1	1	1	0
2		MOV.W	#nCols1, r7	2	1		1	1	1	0
3		MOV.W	#M1, r12	4	2		1	2	2	0
4		CALL	#INIT	4	5		1	5	2	0
5		MOV.W	#nRows2, r6	2	1		1	1	1	0
6		MOV.W	#nCols2, r7	2	1		1	1	1	0
7		MOV.W	#M2, r12	4	2		1	2	2	0
8		CALL	#INIT	4	5		1	5	2	0
9		MOV.W	#M1, r13	4	2		1	2	2	0
10		MOV.W	#M2, r14	4	2		1	2	2	0
11		MOV.W	#M3, r15	4	2		1	2	2	0
12		MOV.W	#5, r4	4	2		1	2	2	0
13	L3:	MOV.W	#3, r5	4	2		5	10	10	0
14	L2:	MOV.W	#4, r6	2	1		15	15	15	0

Continued on Next Page. . .

#	Instruction			Static Results			Dynamic Results			
				Instr.	Instr.	DBytes	No. of	Exec.	Memory Traffic (Cycles)	
				Bytes	Cycles	Moved	Exec.	Cycles	Instr. Mem	Data Mem
15		MOV.W	#0, r9	2	1		15	15	15	0
16		MOV.W	#0, r10	2	1		15	15	15	0
17	L1:	CLR	r11	2	1		60	60	60	0
18		CLR	r12	2	1		60	60	60	0
19		MOV	0(R13),&0130h	6	6	2	60	360	180	60
20		MOV	0(R14),&0138h	6	6	2	60	360	180	60
21		ADD	&SumLo,R11	4	4		60	240	120	0
22		ADDC	&SumHi,R12	4	4		60	240	120	0
23		MOV	0(R13),&0130h	6	6	2	60	360	180	60
24		MOV	2(R14),&0138h	6	6	2	60	360	180	60
25		MOV	0(R14),&0134h	6	6	2	60	360	180	60
26		MOV	2(R13),&0138h	6	6	2	60	360	180	60
27		ADD	&SumLo,R12	4	3		60	180	120	0
28		ADD.W	r11, r9	2	1		60	60	60	0
29		ADDC.W	r12, r10	2	1		60	60	60	0
30		ADD.W	#20, r14	4	2		60	120	120	0
31		DEC.W	r6	2	1		60	60	60	0
32		JG	L1	2	2		60	120	60	0
33		MOV.W	r9,0(r15)	4	2	2	15	30	30	15
34		MOV.W	r10,2(r15)	4	4	2	15	60	30	15
35		SUB.W	#56, r14	4	2		15	30	30	0
36		DEC.W	r5	2	1		15	15	15	0
37		JG	L2	2	2		15	30	15	0
38		MOV.W	#M2, r14	4	2		5	10	10	0
39		DEC.W	r4	2	1		5	5	5	0
40		JG	L3	2	2		5	10	5	0
41		RET‘		2	1		1	1	1	0
42	INIT:	MOV.W	#0, r4	2	1		2	2	2	0
43		MOV.W	#0, r5	2	1		2	2	2	0
44		MOV.W	#0, r9	2	1		2	2	2	0
45	L9:	MOV.W	r4, 0(r12)	4	2	2	2	4	4	2
46		MOV.W	r5, 2(r12)	4	2	2	32	64	64	32
47		ADD.W	#4, r12	2	1		32	32	32	0
48		ADD.W	#1, r4	2	1		32	32	32	0
49		ADDC.W	#0, r5	2	1		32	32	32	0
50		DEC.W	r6	2	1		32	32	32	0
51		JG	L9	2	2		32	64	32	0
52		ADD.W	#1, r9	2	1		32	32	32	0
53		MOV.W	r9, r4	2	1		32	32	32	0
54		DEC.W	r7	2	1		32	32	32	0
55		JG	L9	2	2		32	64	32	0
56		RET		2	3		1	3	1	0
Total				174	125	20	1442	4061	2499	424
TI MSP430 Calculations for Benchmark 6: FIR										
1	main:	MOV.W	#COEFF,r15	4	2		1	2	2	0
2		MOV.W	#0,r7	2	1		1	1	1	0
3	L1:	MOV.W	r7,r12	2	1		18	18	18	0
4		CALL	#itof	4	5		18	90	36	0
5		MOV.W	#0,r14	2	1		18	18	18	0
6		MOV.W	#16544,r15	4	2		18	36	36	0

Continued on Next Page...

#	Instruction			Static Results			Dynamic Results			
				Instr. Bytes	Instr. Cycles	DBytes Moved	No. of Exec.	Exec. Cycles	Memory Traffic (Cycles) Instr. Mem	Data Mem
7		CALL	#fadd	4	5		18	90	36	0
8		MOV.W	r12,r14	2	1		18	18	18	0
9		MOV.W	r13,r15	2	1		18	18	18	0
10		MOV.W	#0,r12	2	1		18	18	18	0
11		MOV.W	#16256,r13	4	2		18	36	36	0
12		CALL	#fdiv	4	5		18	90	36	0
13		MOV.W	r12,0(r15)	4	2	2	18	36	36	18
14		MOV.W	r13,2(r15)	4	2	2	18	36	36	18
15		ADD.W	#4,r15	2	1		18	18	18	0
16		ADD.W	#1,r7	2	1		18	18	18	0
17		CMP.W	#17,r7	4	2		18	36	36	0
18		JL	L1	2	2		18	36	18	0
19		MOV.W	#0,r7	2	1		1	1	1	0
20		MOV.W	#2,r8	2	1		1	1	1	0
21		MOV.W	r7,r15	2	1		1	1	1	0
22	L3:	RLA.W	r15	2	1		68	68	68	0
23		MOV.W	r8,68(r15)	4	2	2	68	136	136	68
24		ADD.W	#1,r7	3	1		68	68	136	0
25		CMP.W	#67,r7	4	2		68	136	136	0
26		JL	L3	2	1		68	68	68	0
27		MOV.W	#0,r10	2	1		1	1	1	0
28	L5:	MOV.W	#0,r7	2	1		36	36	36	0
29		MOV.W	#0,r8	2	1		36	36	36	0
30		MOV.W	#0,r9	2	1		36	36	36	0
31	L6:	MOV.W	r7,r15	2	1		304	304	304	0
32		MOV.W	r10,r13	2	1		304	304	304	0
33		SUB.W	r15,r13	2	1		304	304	304	0
34		ADD.W	#16,r13	4	2		304	608	608	0
35		RLA.W	r13	2	1		304	304	304	0
36		MOV.W	r10,r15	2	1		304	304	304	0
37		MOV.W	r7,r14	2	1		304	304	304	0
38		ADD.W	r15,r14	2	1		304	304	304	0
39		RLA.W	r14	2	1		304	304	304	0
40		MOV.W	68(r14),r12	4	2	2	304	608	608	304
41		ADD.W	68(r13),r12	4	2	2	304	608	608	304
42		CALL	#itof	4	2		304	608	608	0
43		MOV.W	r7,r15	2	1		304	304	304	0
44		RLA.W	r15	2	1		304	304	304	0
45		RLA.W	r15	2	1		304	304	304	0
46		MOV.W	0(r15),r14	4	2	2	304	608	608	304
47		MOV.W	2(r15),r15	4	2	2	304	608	608	304
48		CALL	#fmpy	4	5		304	1520	608	0
49		MOV.W	r8,r14	2	1		304	304	304	0
50		MOV.W	r9,r15	2	1		304	304	304	0
51		CALL	#fadd	4	5		304	1520	608	0
52		MOV.W	r12,r8	2	1		304	304	304	0
53		MOV.W	r13,r9	2	1		304	304	304	0
54		ADD.W	#1,r7	2	1		304	304	304	0
55		CMP.W	#8,r7	2	1		304	304	304	0
56		JL	L6	2	2		304	608	304	0

Continued on Next Page...

#	Instruction			Static Results			Dynamic Results			
				Instr. Bytes	Instr. Cycles	DBytes Moved	No. of Exec.	Exec. Cycles	Memory Traffic (Cycles) Instr. Mem	Data Mem
57		MOV.W	r10,r15	2	1		36	36	36	0
58		ADD.W	#8, r15	2	1		36	36	36	0
59		RLA.W	r15	2	1		36	36	36	0
60		MOV.W	68(r15),r12	4	2	4	36	72	72	72
61		CALL	#fitof	4	5		36	180	72	0
62		MOV.W	32(R3),r14	4	2	2	36	72	72	36
63		MOV.W	34(R3),r15	4	2	2	36	72	72	36
64		CALL	#fmpy	4	5		36	180	72	0
65		MOV.W	r8,r14	2	1		36	36	36	0
66		MOV.W	r9,r15	2	1		36	36	36	0
67		CALL	#fadd	4	5		36	180	72	0
68		MOV.W	r10,r15	2	1		36	36	36	0
69		RLA.W	r15	2	1		36	36	36	0
70		RLA.W	r15	2	1		36	36	36	0
71		MOV.W	r12,202(r15)	4	4	2	36	144	72	36
72		MOV.W	r13,204(r15)	4	4	2	36	144	72	36
73		ADD.W	#1,r10	2	1		36	36	36	0
74		CMP.W	#36,r10	4	2		36	72	72	0
75		JL	L5	2	2		36	72	36	0
76		RET		2	3		1	3	1	0
Total				209	137	26	9331	15182	12436	1536

D.4 ARM Cortex-M3 LPC1342 Calculations Details

Table D.4: ARM Cortex M3 Calculations

#	Instruction			Static Results			Dynamic Results			
				Instr. Bytes	Instr. Cycles	DBytes Moved	No. of Exec.	Exec. Cycles	Memory Traffic (Cycles) Instr. Mem	Data Mem
ARM Cortex M3 Calculations for Benchmark 1: Recursive Factorial										
1	Main:	MOV	R0,#5	2	1		1	1	1	0
2		BL	fact	4	3		1	3	1	0
3	End:	BX	R14	2	2		1	2	1	0
4	Fact:	PUSH	R0	2	2	4	5	10	5	5
5		MOV	R5,R0	2	1		5	5	5	0
6		SUB	R0,R0,#1	2	1		5	5	5	0
7		BGT	L0	2	1		5	5	5	0
8		MOV	R4,#1	2	1		1	1	1	0
9		POP	R0	2	2	4	1	2	1	1
10		BX	R14	2	3		1	3	1	0
11	L0:	BL	Fact	4	3		4	12	4	0
12		MUL	R4,R4,R5	4	1		4	4	4	0
13		POP	R0	2	2	4	4	8	4	4
14		BX	R14	2	3		4	12	4	0
Total				34	26	12	42	73	42	10
ARM Cortex M3 Calculations for Benchmark 2: String Copy										
1	Main:	MOV	R1,#Src	4	1		1	1	1	0

Continued on Next Page...

#	Instruction			Static Results			Dynamic Results			
				Instr. Bytes	Instr. Cycles	DBytes Moved	No. of Exec.	Exec. Cycles	Memory Traffic (Cycles) Instr. Mem	Data Mem
2		MOV	R2,#Dest	4	1		1	1	1	0
3		BL	strCpy	4	3		1	3	1	0
4	End:	BX	R14	2	3		1	3	1	0
5	StrCpy:	MOV	R0,#0	2	1		1	1	1	0
6	L0:	LDRB	R3,[R2,R0]	2	2	4	13	26	13	13
7		STRB	R3,[R1,R0]	4	2	4	13	26	13	13
8		ADD	R0,R0,#1	2	1		13	13	13	0
9		CBNZ	R3,L0	2	2		13	26	13	0
10		BX	R14	2	3		1	3	1	0
Total				28	19	8	58	103	58	26
ARM Cortex M3 Calculations for Benchmark 3: Bubble Sort										
1	Main:	MOV	R4,#0	2	1		1	1	1	0
2		MOV	R1,#START	4	1		1	1	1	0
3	L0:	STR	R4,[R1,R4,LSL #2]	4	2	4	10	20	10	10
4		ADD	R4,R4,#1	2	1		10	10	10	0
5		CMP	R4,#10	2	1		10	10	10	0
6		BLT	L0	2	1		10	10	10	0
7		BL	BSort	4	3		1	3	1	0
8	End:	BX	R14	2	3		1	3	1	0
9	BSort:	MOV	R2,#8	2	1		1	1	1	0
10	L1:	MOV	R0,#0	2	1		9	9	9	0
11	L2:	LDR	R12,[R1,R0,LSL #2]	4	2	4	45	90	45	45
12		ADD	R4,R0,#1	2	1		45	45	45	0
13		LDR	R5,[R1,R4,LSL #2]	4	2	4	45	90	45	45
14		CMP	R12,R5	2	1		45	45	45	0
15		BLE	L3	2	1		45	45	45	0
16		STR	R5,[R1,R0,LSL #2]	4	2	4	45	90	45	45
17		STR	R12,[R1,R4,LSL #2]	4	2	4	45	90	45	45
18	L3:	ADD	R0,R0,#1	2	1		45	45	45	0
19		CMP	R0,R2	2	1		45	45	45	0
20		BLE	L2	2	1		45	45	45	0
21		SUB	R2,R2,#1	2	1		9	9	9	0
22		CBZ	R2,L1	2	2		9	18	9	0
23		BX	R14	2	3		1	3	1	0
Total				60	35	20	523	728	523	190
ARM Cortex M3 Calculations for Benchmark 4: Sensor Structure										
1	Main:	BL	Init	4	3		1	3	1	0
2		BL	Calib	4	3		1	3	1	0
3	End:	BX	R14	2	3		1	3	1	0
4	Init:	MOV	R1,#START	4	1		1	1	1	0
5		MOV	R4,#0	2	1		1	1	1	0
6		MOV	R2,#0	2	1		1	1	1	0
7		MOV	R3,#0	2	1		1	1	1	0
8	L0:	ADD	R2,R4,#3	2	1		5	5	5	0
9		STRB	R3,[R1,#0x00]	4	2	4	5	10	5	5
10		STRH	R4,[R1,#0x02]	4	2	4	5	10	5	5
11		STR	R2,[R1,#0x04]	4	2	4	5	10	5	5
12		ADD	R1,R1,#6	2	1		5	5	5	0
13		ADD	R4,R4,#1	2	1		5	5	5	0
14		CMP	R4,#10	2	1		5	5	5	0

Continued on Next Page. . .

#	Instruction			Static Results			Dynamic Results			
				Instr. Bytes	Instr. Cycles	DBytes Moved	No. of Exec.	Exec. Cycles	Memory Traffic (Cycles) Instr. Mem	Data Mem
15		BLT	L0	2	1		5	5	5	0
16		BX	R14	2	3		1	3	1	0
17	Calib:	MOV	R1,#START	4	1		1	1	1	0
18		MOV	R4,#0	2	1		1	1	1	0
19		MOV	R3,#1	2	1		1	1	1	0
20	L1:	STRB	R3,[R1,#0x00]	4	2	4	5	10	5	5
21		LDRH	R2,[R1,#0x02]	4	2	4	5	10	5	5
22		LDR	R3,[R1,#0x04]	2	2	4	5	10	5	5
23		SUB	R2,R3,R2	4	1		5	5	5	0
24		STR	R2,[R1,#0x04]	4	2	4	5	10	5	5
25		ADD	R1,R1,#6	2	1		5	5	5	0
26		ADD	R4,R4,#1	2	1		5	5	5	0
27		CMP	R4,#10	2	1		5	5	5	0
28		BLT	L1	2	1		5	5	5	0
29		BX	R14	2	3		1	3	1	0
Total				80	46	28	97	142	97	35
ARM Cortex M3 Calculations for Benchmark 5: Matrix Multiplication										
1	Main:	MOV	R12,#M1	4	1		1	1	1	0
2		MOV	R6,#nRows1	2	1		1	1	1	0
3		MOV	R7,#nCols1	2	1		1	1	1	0
4		BL	INIT	4	3		1	3	1	0
5		MOV	R12,#M2	4	1		1	1	1	0
6		MOV	R6,#nRows2	2	1		1	1	1	0
7		MOV	R7,#nCols2	2	1		1	1	1	0
8		BL	INIT	4	3		1	3	1	0
9		MOV	R10,#M1	4	1		1	1	1	0
10		MOV	R11,#M2	4	1		1	1	1	0
11		MOV	R12,#M2	4	1		1	1	1	0
12		MOV	R5,#0	2	1		1	1	1	0
13	L6:	MOV	R4,#0	2	1		5	5	5	0
14	L5:	MOV	R1,#0	2	1		15	15	15	0
15		MOV	R3,#0	2	1		15	15	15	0
16	L4:	LDR	R7,[R10]	2	2	4	60	120	60	60
17		LDR	R8,[R11]	2	2	4	60	120	60	60
18		MLA	R3,R7,R8	4	2		60	120	60	0
19		ADD	R11,R11,#20	4	1		60	60	60	0
20		ADD	R10,R10,#4	2	1		60	60	60	0
21		SUB	R1,R1,#1	2	1		60	60	60	0
22		BLT	L4	2	1		60	60	60	0
23		STR	R3,[R11]	4	2	4	15	30	15	15
24		SUB	R11,R11,#56	4	1		15	15	15	0
25		ADD	R12,R12,#4	2	1		15	15	15	0
26		SUB	R1,R4,#1	2	1		15	15	15	0
27		BLT	L5	2	1		15	15	15	0
28		MOV	R11,#M2	4	1		5	5	5	0
29		SUB	R5,R5,#1	2	1		5	5	5	0
30		BLT	L6	2	1		5	5	5	0
31		BX	R14	2	1		1	1	1	0
32	INIT:	MOV	R0,#0	2	1		2	2	2	0
33		MOV	R1,#0	2	1		2	2	2	0

Continued on Next Page...

#	Instruction			Static Results			Dynamic Results			
				Instr.	Instr.	DBytes	No. of	Exec.	Memory Traffic (Cycles)	
				Bytes	Cycles	Moved	Exec.	Cycles	Instr. Mem	Data Mem
34	L1:	STR	R0,[R12]	4	2	4	32	64	32	32
35		ADD	R0,#1	2	1		32	32	32	0
36		SUB	R7,#1	2	1		32	32	32	0
37		ADD	R12,#4	2	1		32	32	32	0
38		BLT	L1	2	1		32	32	32	0
39		ADD	R1,#1	2	1		32	32	32	0
40		MOV	R0,R1	2	1		32	32	32	0
41		SUB	R6,#1	2	1		32	32	32	0
42		BLT	L1	2	1		32	32	32	0
43		BX	R14	2	3		2	6	2	0
Total				112	54	16	852	1087	852	167
ARM Cortex M3 Calculations for Benchmark 6: FIR										
1	Main:	MOV	R8,#0	2	1		1	1	1	0
2	L0:	ADD	R0,R8,5	2	1		18	18	18	0
3		BL	int2float	4	3		18	54	18	0
4		MOV	R4,#0x3f800000	4	1		18	18	18	0
5		BL	fdiv	4	3		18	54	18	0
6		MOV	R1,#COEFF	4	1		18	18	18	0
7		STR	R0,[R1,R8,LSL #2]	4	2	4	18	36	18	18
8		ADD	R8,R8,#1	2	1		18	18	18	0
9		CMP	R8,#17	2	1		18	18	18	0
10		BLT	L0	2	1		18	18	18	0
11		MOV	R8,#0	2	1		1	1	1	0
12	L1:	MOV	R0,#2	2	1		68	68	68	0
13		MOV	R1,#INPUT	4	1		68	68	68	0
14		STR	R0,[R1,R8,LSL #2]	4	2	4	68	136	68	68
15		ADD	R8,R8,#1	2	1		68	68	68	0
16		CMP	R8,#67	2	1		68	68	68	0
17		BLT	L1	2	1		68	68	68	0
18		MOV	R9,#0	2	1		1	1	1	0
19	L2:	MOV	R10,#0	2	1		36	36	36	0
20		MOV	R8,#0	2	1		36	36	36	0
21	L3:	ADD	R1,R9,#16	2	1		304	304	304	0
22		SUB	R1,R1,R8	4	1		304	304	304	0
23		MOV	R2,#INPUT	4	1		304	304	304	0
24		LDR	R1,[R2,R1,LSL #2]	4	2	4	304	608	304	304
25		ADD	R2,R9,R8	4	1		304	304	304	0
26		LDR	R2,[R3,R2,LSL #2]	4	2	4	304	608	304	304
27		ADD	R0,R1,R2	2	1		304	304	304	0
28		BL	int2float	4	3		304	912	304	0
29		MOV	R6,#COEFF	2	1		304	304	304	0
30		LDR	R1,[R6,R8,LSL #2]	4	2	4	304	608	304	304
31		BL	fmul	4	3		304	912	304	0
32		MOV	R1,R10	2	1		304	304	304	0
33		BL	fadd	4	3		304	912	304	0
34		MOV	R10,R0	2	1		304	304	304	0
35		ADD	R8,R8,#1	2	1		304	304	304	0
36		CMP	R8,#8	2	1		304	304	304	0
37		BLT	L3	2	1		304	304	304	0
38		MOV	R1,#INPUT	4	1		36	36	36	0

Continued on Next Page...

#	Instruction			Static Results			Dynamic Results			
				Instr.	Instr.	DBytes	No. of	Exec.	Memory Traffic (Cycles)	
				Bytes	Cycles	Moved	Exec.	Cycles	Instr. Mem	Data Mem
39		ADD	R2,R9,#8	4	1	4	36	36	36	36
40		LDR	R0,[R1,R2,LSL #2]	4	2		36	72	36	0
41		BL	int2float	4	3	4	36	108	36	36
42		LDR	R1,[#Addr(COEFF[8])]	4	2		36	72	36	0
43		BL	fmul	4	3		36	108	36	0
44		MOV	R1,R10	2	1		36	36	36	0
45		BL	fadd	4	3		36	108	36	0
46		MOV	R1,#OUTPUT	4	1		36	36	36	0
47		STR	R0,[R1,R9,LSL#2]	4	2	4	36	72	36	36
48		ADD	R9,R9,#1	2	1		36	36	36	0
49		CMP	R9,#36	2	1		36	36	36	0
50		BLT	L2	2	1		36	36	36	0
51	End:	BX	R14	2	3		1	3	1	0
Total				152	77	32	6282	9502	6282	1106

Curriculum Vitae

Imran Ashraf