C Compiler for a Parallel Processor

By
Ahsan Kamal and Husain Ahmad

Project report for partial fulfillment of the requirements of MCS/NUST for award of the B.E degree in Software Engineering

Department of Computer Science
Military College of Signals
Rawalpindi

April 2002

Syndicate Members

Ahsan Kamal
NC # 143
BESE-4

Husain Ahmed
NC # 154
BESE-4

Directing Staff

Lt Col Muhammad Tufail Malik
HOD CS Department

Dr. Shoab A Khan *(external)*
President CET

Abstract

In this dissertation, we present a dual-module C compiler for the Media Engine (ME-2), being developed in Communications Enabling Technology (CET). The first module takes the C code and converts it into the serial assembly, referred hereon as Front-end Serial Assembly Generator (FSAG). We have only implemented a minimal functionality prototype of FSAG, whereas in actuality the GNU C compiler is being used as the serial code generator. The second module is the Serial Assembly Optimizer (SAO). This module makes use of advanced optimization techniques to generate a parallel and optimized code.

Declaration

"No portion of the work presented in this dissertation has been submitted in support of another award or qualification either at this institution or elsewhere"

Acknowledgements

The final year project has been a great experience for us. During the course of its duration we have learned a lot. We have been able to apply the knowledge and skills that we have developed during our three and a half years of stay in MCS. However, this humble effort of ours would not have been fruitful if it were not for the guidance and support many people. We would like to thank those who, often despite of their own commitments, have taken their time and effort to help us.

First of all we would like to thank Allah Almighty for his guidance, without which we would have been lost. Then we would like to thank our Directing Staff, Col Tufail for his help, understanding and considerations at difficult times. We would also like to thank Dr. Shoab, Madam Durdana, Salma baji, Mr. Tahir Awan and all the people in CET who helped us in every possible way. Our colleagues in MCS have been really supportive, especially Omair, Zunair, Imran Siddique, Khurram, Sarmad, Imran Ashraf and all the inlivings. Finally, we would like to thank NUST, MCS and the CS department, as they have made it all possible.

Ahsan Kamal

Husain Ahmed

Table Of Contents

List of Figures/Table

Section one

Introduction

In recent years the advances in process technology and chip design have led to many high performance chips on the market, at more and more affordable prices per computational power. At the same time, the “smart devices" model of computation has also been gaining ground, giving rise to a large variety of embedded systems utilizing high performance, often highly specialized chips. In particular, many of those chips have been designed to address the needs of signal processing application such as digital communications and digital television, computer graphics, simulations, etc. Being intended for large production volumes, cost has been an even more pressing issue and a lot of the designs do not have the complicated optimization circuitry doing branch prediction, pre-fetching and caching, associated with modern general-purpose microprocessors. Because of the simplicity of the hardware design, which makes parallelism very explicit to the user, most of the optimizations have been left to the programmer. Still, providing optimal or nearly optimal solutions to the optimization problems in the transition to high performance chips has been interesting, challenging and difficult. This work presents an attempt at optimal or nearly optimal utilization of computing capabilities for high-performance chips. Any DSP algorithm written in simple non-parallel C language is first converted into un-optimized serial assembly. Given this serial code, or for that matter an un-optimized or partially optimized assembly code by a programmer, the software tries to produce optimal or nearly optimal version of the same code for a specific DSP/VLIW chip.

The real motivation behind this methodology lies due to the following reasons.

I- Hand Optimization Techniques are not Scalable

While code parallelization of assembly code for VLIW/DSP chips has been an active research issue for decades, in most cases even now, assembly code for powerful DSP and other VLIW chips is handcrafted and optimized. Unfortunately, a lot of the techniques employed by humans in this process (if any systematic techniques are employed at all) do not scale very well. For example, the introduction of the Texas Instruments TMS320C6000 series, capable of issuing up to 8 instructions with variable pipeline lengths per cycle, marks a new era, where it becomes increasingly hard to hand-optimize code with that much parallelism, and in fact, to even write remotely optimal code. In fact, one can argue that software such as the one presented here, extended with more high-level optimizations, and could often produce better results than humans on long and algorithmically complicated code. Thus, given the tendency of creation of even more powerful chips and the use of more convoluted algorithms, scalability becomes more and more important issue.

II- Hand-Optimized Code is not Portable

Not only techniques for hand optimization are not scalable, but also they are not portable. Historically, each generation of DSP chips has been taking advantage of process technology to optimize the instruction set architecture, because it is too costly in terms of power and gates to emulate a single instruction set architecture. Thus, with the introduction of new more powerful and different chips, all optimizations to existing code need to be re-done, which is a long and difficult process, involving both coding, optimization and validation.

III- Related Work

There has been a fair amount of work in the field with many semiconductor companies releasing powerful chips and compilers that do some optimizations as commercial solutions. In fact some of the inspiration behind this work come from looking of older version of assembly optimizer tools running in real time and attempting to achieve better and more general solutions.

Solutions like these have more potential for overall improvement, but employ heuristic solutions for code compaction. Other relevant publications include work on various techniques ([1-10]) for code compaction. Many of those techniques are computationally infeasible (such as solving the entire optimization problem as a single integer linear program) or partially used in this work (linear programming for obtaining bounds), or inapplicable, because of different computation framework.

Chapter 1

Introduction To The Dissertation

This chapter gives a brief introduction to this document. It states the purpose and scope of this document, explaining what this document is for as well as who should use this document. It also gives a brief introduction to our project. However at this point we have avoided any complex details. This chapter prepares the readers mind for the things coming next.

1.1) Purpose of Dissertation

This dissertation is provided to fulfill the requirements of the Final Year project for the completion of the BE Software Engineering Degree. The purpose of this Dissertation is to provide an insight into the various phases and semantics of this project. It describes the complete development process including the research, analysis, and design specific documentation. It provides an insight into the working and semantics of the ME-2 compiler, which can help both the users and the future developers.

1.2) Scope of Dissertation

The dissertation has been created in order to meet the requirements of the Department. It shall be providing all the necessary information about the project. However, we have also kept in mind the users and any future developers. It will be helpful in understanding the workings of the compiler.

1.3) Layout of the Dissertation

The dissertation has been divided into various sections. A brief section wise description follows.

The first section starts by introducing the reader to the project by giving a brief description to the project.

The second section contains some background material regarding the project. This material is provided so the reader can develop a sound understanding of the actual project. First we discuss the architecture of ME-2 processor. Then we move to the various optimization techniques that are used for mapping of DSP algorithms on VLIW. In the last part of this section, the various techniques discovered and used by us are listed.

The next section contains the Software Specifications. It contains the detailed analysis documentation including the Flow Diagrams, Process specifications, Class diagrams etc.

The fourth and the final section contain the various results and conclusion. It also mentions certain areas, which we were not able to implement into our product, and gives recommendation of any future work in this field.

Chapter 2
Project Description

This chapter lists the features of ME-2 compiler. It starts of with a brief description of the project. Then an initial specification listing the main features of the ME-2 compiler is given. In the end the life cycle model used is explained.

2.1) Description

Media Engine 2 (ME-2) is a fixed-point DSP processor based on ***Very Long Instruction Word (VLIW)*** architecture. It is being developed at ***Communication Enabling Technologies (CET)***. It has 9 functional units and support for parallel execution of a maximum of 5 instructions. (The detailed overview of ME-2 is given in the next chapter)

Our project was to develop a C compiler for this ME-2 processor. It has been a strong combination of both research work and product development. During the several phases of the project we had to go through a lot of research materials, researching and then adopting various state of the art compiler techniques.

The project is divided into two independent modules, the Front-end Serial Assembly Generator (FSAG), which produces the serial assembly, and the Serial Assembly Optimizer (SAO), which optimizes the serial assembly into parallel assembly. This modular approach later allowed CET to use the C compiler from the **G**NU **C**ompiler **C**ollection (GCC) as the front-end code generator.

2.2) Features

ME-2 C Compiler contains the following features.

2.2.1) Modular Approach

A modular approach is used for the developing of the compiler. It is divided into the following main modules

1. Front-end Serial Assembly Generator
2. Serial Assembly Optimizer

These modules were developed separately, with strong emphasis on a common interface between the two.

2.2.2) Front-end Serial Assembly Generator

The Front-end Serial Assembly Generator (FSAG) should transform all types of C-language statements and declarations into serial assembly code. The ME-2 assembly contains both serial and parallel instructions. The FSAG should only generate serial assembly instructions.

2.2.3) Serial Assembly Optimizer

The Serial Assembly Optimizer (SAO) should transform the serial assembly instruction into parallel assembly. It should perform advance optimization on the serial code fed into it. These optimization techniques shall concentrate on the time optimization of the code. Mechanism shall be provided so that certain optimizations can be switched off.

Note: *all of these features are requirement as specified by CET. However, as we shall see farther on, that some of these features required a different approach and a separate team was dispatched to implement those features. This was particularly the case regarding the FSAG.*

2.3) Life Cycle Model

As specified, a modular approach was used for developing the ME-2 Compiler. Hence the whole development process was divided into two distinct phases or modules; the development of the FSAG and SAO.

The development was done using the classical compiler writing techniques merged with the software engineering models. The different phases listed below describe the whole life cycle of the development process.

2.3.1) PHASES

- Understanding the architecture and instruction set of ME-2.
- Implementation of front-end code generator prototype.
- Research of optimization techniques for mapping DSP algorithms on VLIW machines.
- Selection and modification of techniques.
- Hand coded testing of techniques.
- Design.
- Implementation.
- Testing in user environment.
- Integration of Serial Assembly Optimizer ***(SAO)*** with GCC serial code generator.

2.3.2) FSAG

Initially while taking up the project we were told that the Front-end Serial Assembly Generator FSAG should be developed using the GNU Compiler

Collection GCC. A team of two people was already working for the development of FSAG.

We developed a small subset of the FSAG using rapid development method. Thus a small prototype was completed and shown to our Directing Staff. However, we were encouraged not to build on that prototype. The use of modular approach enabled us to develop both FSAG and SAO in parallel with each other.

2.3.3) SAO

The serial assembly optimizer required a lot of research regarding the advanced optimization techniques for mapping of DSP algorithms on VLIW based processors. Thus its development started with this research. The next step was to test these techniques and short-list them to the few suiting our particular needs. This was followed by the design and implementation of the algorithm based on these techniques.

Section two

Background Knowledge

The full awareness of the VLIW architecture and the various compiler optimization techniques is of utmost importance for grasping the essence of this project. This section shall prove to be a learning experience for a layman, starting with the very basic concepts and moving onto the most technical aspects. For the able reader, however, this shall prove to be a mere revision of some interesting concepts and techniques.

The first chapter in this section provides a brief, yet insightful introduction to the VLIW architecture and then specifically to the ME-2 architecture.

The second chapter explains the various optimization techniques that are widely used and implemented in such optimizing compilers.

Chapter 3
ME-2 Architecture

This chapter gives the user with the background knowledge. It starts with an introductory detail of the ME-2 architecture. After the architectural features are understood it moves on to the various optimization techniques. In the end we list the techniques that were used in the implementation of SAO. We also discuss the various constraints and limitations that set the selection criteria.

3.1) Introduction

Real-time digital signal processing applications require processing rates of millions of instructions per second. A single digital signal processor cannot handle such high-speed computation rates. Since digital signal processing algorithms possess high degree of parallelism, parallel processing is used to increase the computational capability of DSP based systems.

DSP chips with multiple FUs can exploit both fine-grain parallelism and coarse-grain parallelism. Multiple FUs on single chip architecture is called Very Long Instruction Word (VLIW) architecture.

The VLIW architecture employs multiple pipelined FUs, multi-ported register files, multiple data paths and a global clock. It takes advantage of both temporal and spatial parallelism [4]. This machine reduces the Clocks Per Instructions (CPI) by executing several operations concurrently. Parallel operations are embedded in horizontal instruction format. Thus one long instruction word specifies completely the operations to be performed by each of the FUs in each cycle. RISC-like instruction sets are used so that dependency checking becomes easier for the compiler [2]. The compiler detects parallel operations in a program and embeds them in the very long instruction words.

3.2) VLIW Architecture

The Very Long Instruction Word (VLIW) architectures form the basis for an alternative way to organize processors. They are derived from the concept of horizontal micro coding and multiple instruction execution. They are designed to exploit the instruction level parallelism inherent in programs. These processors employ multiple pipelined functional units, multi-ported register file, multiple data paths and a global clock. As such they take advantage of both temporal and spatial parallelism [4]. All the functional units share the use of the register file. The operations to be concurrently executed by the functional units are synchronized in a VLIW instruction. Figure 3.1 shows a typical VLIW processor and its instruction format. Different fields of the long instruction word carry the opcodes to be dispatched to different functional units. For example, I_1 would be executed by FU_1, I_2 executed by FU_2 and so on. VLIW machines are expected to provide ten to thirty times the performance of a more conventional machine built of the same implementation technology [11].

In comparison, the presence of high-level regularity in user's code is essential if a SIMD processor is to be employed. VLIW machines can exploit even irregular forms of parallelism for achieving speedup. Similarly, an MIMD solution imposes synchronization and communication penalties. Whereas in a VLIW processor all functional units run completely synchronized, directly controlled in each clock cycle by the compacting compiler. Because the *compiler handles the arbitration*, the buses are fast, simple and cheap.

Similar to superscalar architecture, the VLIW architecture can reduce the clocks per instruction (CPI) factor by executing several operations concurrently. However, superscalar machines need more complex hardware for run-time resource scheduling and synchronization. Simplicity in hardware makes VLIW processors easier to design and enhances their efficiency. These processors use

RISC-like instruction sets. While VLIW architectures permit static extraction of fine grain parallelism, their major drawback lies in the considerable code memory size requirements, due to the horizontal nature of the instruction set.

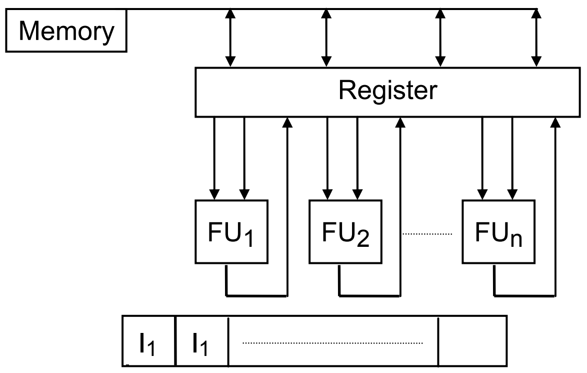


Figure 3.1 Block diagram of an ideal VLIW and its instruction word

Multipurpose VLIW architectures are now being built for multimedia, video and Digital Signal Processing (DSP) applications. Mapping of DSP algorithms was chosen because efficient exploitation of concurrency available in these algorithms is of prime importance to synthesize high throughput systems. These algorithms are characterized by iterative sequences of operations, representing arithmetic parallelism. Furthermore, the number of times these iterations are executed is predictable. This makes the use of static scheduling feasible. VLIW machines provide an effective platform as its multiple functional units can extract temporal parallelism. Pipelining within functional units allows issuing of new operations in each cycle exploiting the spatial parallelism. Highly concurrent implementations can be obtained by using global optimization techniques.

Some of the early examples of VLIW processors are the Intel i860 that can issue two operations per cycle, the IBM System 6000 with four concurrent operations, and the Multiflow TRACE, which was designed to allow the concurrent execution of up to twenty-eight operations per cycle. The most recent commercially available VLIWs are TriMedia-1 by Philips and TMS320C6200 from Texas Instruments.

3.2.1) Terminology

Let P be a program consisting of a set of operations $\{s_1,s_2,s_3\ldots.s_n\}$. These operations require resources for their execution such as

Functional units $\{fu_1,fu_2,fu_3\ldots.fu_n\}$, which can be adders, multipliers, load/store ports etc.

Registers $\{r_1,r_2,r_3\ldots.r_n\}$, which are, used for read/write of data values. A register is *live* for the duration between which data is written and its corresponding read.

A compacted program P_c represents a set of operations $\{s_1,s_2,s_3\ldots.s_n\}$ with the independent operations scheduled in the same cycle. It is equivalent to P, but takes fewer cycles to execute.

Iteration means one pass through the loop.

Intra-iteration dependences are the precedence constraints within iteration.

Inter-iteration dependence is a dependence on the result of a previous iteration.

3.2.2) Principles Behind VLIWs

The basic building blocks behind these architectures are:

- Datapaths
- Pipelines
- Functional units

3.2.2.1) Datapaths

A large number of datapaths characterize the VLIW architecture. These datapaths support the simultaneous access of operands by each of the functional units for parallel operation. Figure 3.2 shows the datapaths for a generic VLIW machine. A large number of registers help in fetching the operands from memory and using them for fast access. Thus the register file must have two read ports and one write port for each one of the functional units. One of the most critical constraints in the implementation of the ideal VLIW model is the inability to build a register file with very large number of ports [13]. The number of ports of the registers usually limits multiple instruction execution, in practice, and resource limitations simply prevent the writing of multiple results simultaneously to a single register set. Consequently, the actual VLIW implementations often use partitioned register files and functional units. A number of these partitions may then be used to resolve the need for more functional units without increasing the number of ports in the register file. In case a functional unit needs to read the register file of another partition cross paths are used.

3.2.2.2) Pipelines

Pipelining implies the segmenting in time of a computational function into several sub functions. Figure 3.3 shows a function partitioned in time into k different stages. If each stage is a physically distinct piece of hardware, then they can operate concurrently allowing up to k parallel operations after the filling of the entire pipeline. Suppose time for each stage to operate is T, and then the first output will appear after a delay of kT time units. However, successive outputs can be obtained every T time units.

The main advantage of pipelining is that its hardware cost is quite low. However, if any of the constituent stages fails to produce an output at the end of T time units, the entire pipeline will stall. In a VLIW machine, instructions using different functional units can proceed simultaneously through the pipeline phases. The pipeline operation for different instructions can be categorized according to the number of CPU cycles or delay slots. During delay slots, results from the instruction cannot be read. To optimize a program for speed, one must understand the sequence of the program fetch; data load requests the program makes and how they might stall the CPU.

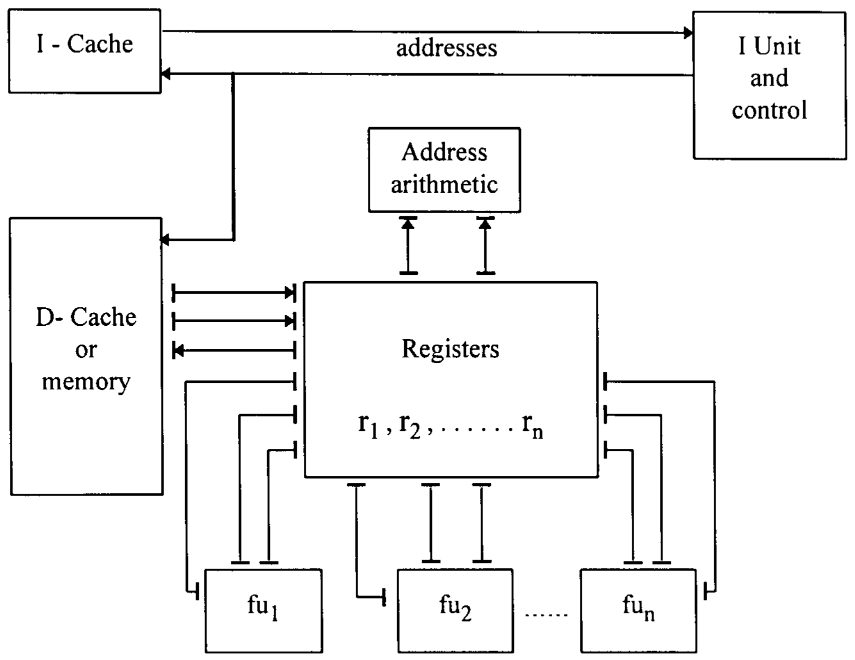


Figure 3.2 Datapaths of a generic machine

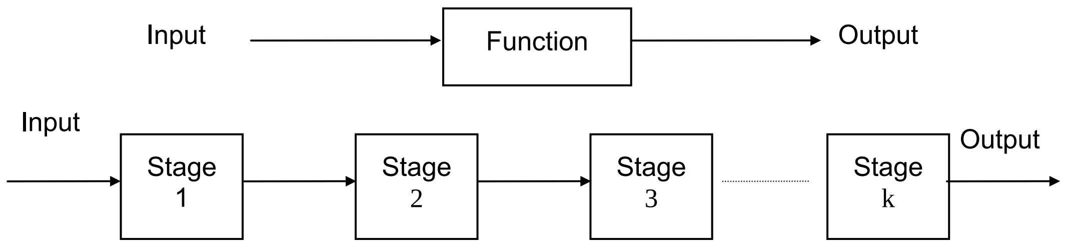


Figure 3.3 Pipelining example

3.2.2.3) Functional units

All VLIW machines rely on the use of several functional units to achieve speedup. Pipelined functional units are employed so that instructions can be dispatched in each cycle. All of these functional units operate synchronously using one global clock. Each segment of the long instruction word controls one particular functional unit. The enhancement achieved by increasing the number of functional units against cost is still a topic of current research. The maximum number of concurrent instructions is at most equal to the number of functional units. Since all VLIWs use RISC-like instruction sets, they have nondestructive triadic register files. A large number of these registers help in fetching the operands from memory and using them for fast access. Thus the register file must be multiported so that several different functional units can access operands simultaneously, and it should have enough memory bandwidth to balance the maximum operand usage rate of the functional units.

3.3) ME-2 Architecture

The VZM 2000 TXP/RXP also known as ME-2 is a fixed-point digital signal processor (DSP) core. It is a high performance, very long instruction word (VLIW) architecture, specially suited to media-specific applications like G.729a, G.723.1.

The salient features of the architecture are:

- Four arithmetic/MAC operations in parallel with 64-bit load/stores.
- 32-bit arithmetic, logical, shift and normalization operations.
- 40/32-bit MACs
- Multi-cycle double precision operations
- Data Pointer Registers for intensive DSP loops
- Delay line registers for efficient FIR, convolution and correlation processing
- Data Alignment Buffers (DABs) for unaligned loads
- Advanced addressing modes (Bit reversed, circular, pre/post modify etc.)
- Zero-overhead Looping and predicated execution
- Multi-cycle fractional division
- Right pre-shifts of 1, 2, or 3 bits with register loads
- Quadruple test bits for testing of four simultaneous conditions
- Four simultaneous Add-Compare-Select (ACS) operations

3.3.1) Central Processing Unit (CPU)

The TXP/RXP CPU contains:

- Program Control Unit
- Instruction Dispatch Unit
- Data Path Control Logic
- Address Generation Unit
- Control Register
- Nine Data Paths (Execution Blocks)
- Debug Logic
- 16 14-bit Address Register File
- 16 32-bit Data Register File

3.3.2) Internal Memory

The TXP/RXP has separate data and program memories. The Data memory is treated as 4K x 64-bit words and Program memory is treated as 2K x 128-bit words.

3.3.3) TXP/RXP Pipeline

There are six pipeline stages.

- The pipeline can dispatch five parallel instructions every cycle.
- Parallel instructions proceed simultaneously through the same pipeline phases.

3.4) Instruction Set Overview

The assembly instructions have been designed with a C-like syntax to provide ease of programming. The extensive range of instruction groupings allowed further aids the programmer in coding media specific applications. The CPU allows parallel execution of 5 instructions, with certain restrictions.

3.4.1) Instruction Types

Instructions for the VZM 2000 TXP/RXP can be divided in to the following categories.

- Load/Store instructions
- AGU arithmetic instructions
- Stack support instructions
- Program Control and Loop instructions
- Logic instructions
- Shift and Normalization instructions
- Mac unit instructions
- Arithmetic instructions

All instructions of Media Engine-2 (Engine0) can be categorized as AGU and DATAPATH instructions.

3.4.1.1) AGU Instructions

All instruction related to the address generation unit are categorized into AGU instruction. These include load/store, stack support, AGU arithmetic and control flow instructions. For a brief overview, refer to [20].

3.4.1.2) DataPath Instructions

The logical instructions, shift and normalization instructions, arithmetic instructions, and MAC instructions are included in this category. See [20] for reference.

3.4.2) Registers

There are 16 x 32-bit data registers from R0 to R15, and 16 x 14-bit address registers from A0 to A15. The data registers can be used as separate 32 x 16-bit registers with **R*i*.f** implying the upper 16 bits and **R*i*.i** the lower part of the data register **R***i*

3.4.3) Addressing Modes

When upper 16-bits of a data register are loaded, lower 16-bits are zero-filled. When lower 16-bits are loaded, upper 16-bits contain sign extension. Load is done through a pre-shifter or a data alignment buffers. The processor supports the following addressing modes.

- ➢ Register addressing with no update, post-increment or decrement on address register.
- ➢ Register addressing with post offset update or indexing (pre-increment by offset register without update) on address register.
- ➢ Register addressing with post offset update or indexing (pre-increment without update) by an immediate offset on address register)

Any register from A8-A15 with the exception of A11 (reserved for stack operation) can be used for specifying an offset.

3.4.4) Use of data pointer registers

3. If a DPR points to a register apart from R0, R4, R8, R12 for load or store of four 16 bit operands, data will be loaded to /stored from the start of that group. It is true for other group and MAC operation, e.g. if DPR points to R3 and four loads are performed at that DPR, operands will be loaded to R0, R1, R2, R3. Same is the case of load/store of two operands.

4. Any AGU or execution block instruction that uses dprs, is 32 bit. Four dprs, at maximum can be updated. Most of the 16 bit instructions have a corresponding 32-bit DPR version. Dprs are not only used in loops but are also to provide flexibility in operand specification. For example, AND operation requires one source and destination to be the same when its 16 bit version (with registers only) is used. However, with dprs, all three operands become independent.

3.4.5) Execution Block Packet Composition

The following grouping restrictions apply to execution block instructions within one packet:

- 48-bit EB0 instruction, 16-bit EB1 instruction, 16-bit EB2 instruction, 16-bit EB3 instruction
- 32-bit EB0 instruction, 32-bit EB1 instruction, 16-bit EB2 instruction, 16-bit EB3s instruction
- 32-bit EB0 instruction, 16-bit EB1 instruction, 16-bit EB2 instruction, 16-bit EB3 instruction
- 16-bit EB0 instruction, 16-bit EB1 instruction, 16-bit EB2 instruction, 16-bit EB3 instruction
- 32-bit Dual instruction at EB0, 32-bit Dual instruction at EB2
- 32-bit Dual instruction at EB0, 16-bit Dual instruction at EB2
- 32-bit Dual instruction at EB0, 16-bit EB2 instruction, 16-bit EB3 instruction
- 16-bit Dual instruction at EB0, 16-bit Dual instruction at EB2
- 16-bit Dual instruction at EB0, 16-bit EB2 instruction, 16-bit EB3 instruction
- 32-bit Quad instruction at EB0
- 16-bit Quad instruction at EB0

3.4.6) VLIW Grouping Restrictions

These restrictions define the composition of a valid VLIW packet for Engine 0.

1. An instruction word is defined as 16 bits long.
2. Maximum instructions words in a packet can be 8.
3. All atomic instructions comprise of 1 instruction word.
4. Long instruction can be 32-bit wide for AGU, and 48-bit wide for execution blocks. In a long instruction, instruction words beyond the first one are called extension words.
5. Prefix holds additional information about the VLIW a packet. A prefix can be either 32 or 16 bit long.
6. There cannot be a 32-bit instruction and a dual instruction in the same packet.
7. If there is a dual instruction in a packet, it should be the first execution block instruction.
8. If there are two dual instructions in a packet, they cannot be any other execution block instructions in that packet.

3.4.7) Looping restrictions

1. The minimum size for hardware loops is two VLIW packets.
2. The first packet of a loop instruction cannot contain a crossover. This implies that to push the first packet until it has no crossovers, the assembler places nops, each with MSBs 11 to indicate a continuing packet of nops with a repeat instruction and any other parallel instructions.

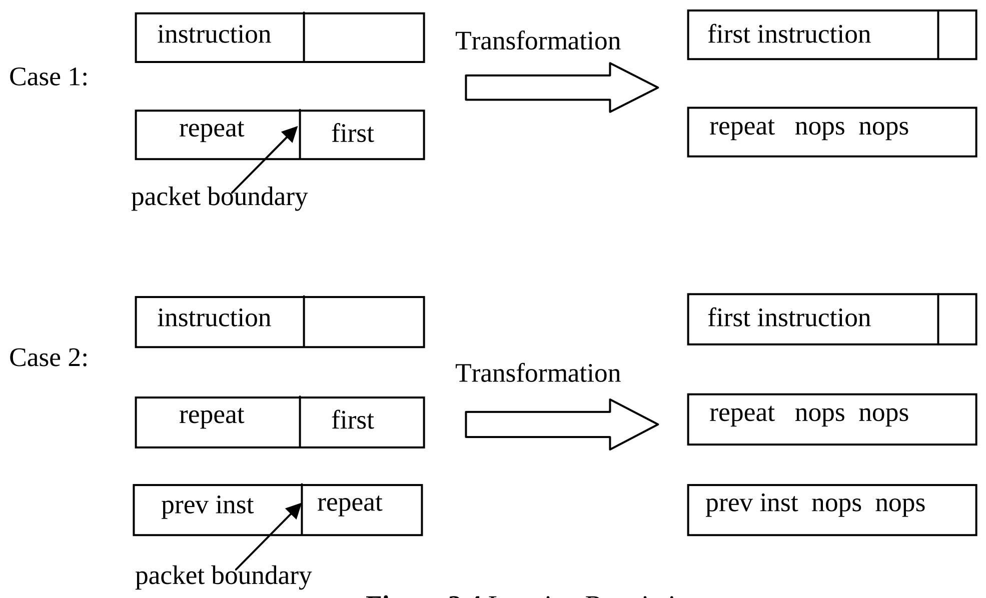


Figure 3.4 Looping Restrictions

3. A loop that has a nested loop within cannot be a short loop.
4. Continue and Break instructions always have two delay slots. Non-delayed Continue and Breaks cannot be executed.
5. Loop markers cannot be placed within the delay slots of change of flow instructions.
6. Nested loops cannot end at the same address. This restriction arises from the fact that two loop-end markers for different loops cannot coincide.
7. Loop markers for short loops are placed before the last instruction.
8. Loop markers for long loops are placed before the second to last instruction. If nth instruction is the last instruction, loop marker is required with (n-2)th instruction(assembler)
9. The last four execution packets of a loop cannot contain the REPEAT instruction of a nested loop.
10. Outer loops should contain at least three instructions, after the end of inner loop if continue or break instruction is used in inner loop (in order to jump outside the inner loop without missing loop marker).

3.4.8) Conditional Execution

It is possible to include any mix of instructions in any combination of IFT/IFF, Ifany, ifall, or caseT. An execution packet may have the combinations ift/ iff, ift / ifaand iff / ifa.

There are however restrictions on the operations that can be conditionally executed. These operations are few and are rarely required to be conditionally executed. More important restrictions are the scheduling constraints on predicated packets. These constraints occur because of the latency of test instructions.

Possible constraint conditional AGU instructions can only be executed on $T_{1.}$

3.4.9) Latencies

Instruction	Latency
Ri = Aj	1 cycle
pop $(R_i)/(R_i, R_{i+1})$	3 cycles
pop $(A_i)/(A_i, A_{i+1})$	3 cycles
Change of Flow Instructions	3 cycles
Ri=Ri/Rj	18 cycles

Table 3.1 Latencies

Chapter 4

Optimizations Techniques

Compiler optimizations are designed to reduce a program's execution time. Traditionally, these optimizations are customized for a given machine model. Classical optimizations are designed to improve the program's efficiency for a machine model that has one thread of execution and can issue one instruction per cycle. Superscalar optimizations are designed for a machine model with a single thread of execution and a limited instruction issue rate. Multiprocessors are built using either uni-processors or superscalar processors and thus there is more than one machine model to optimize for. Therefore, it is important to understand the interactions of these optimizations and their effect on available parallelism and speedup.

4.1) Parallelism in Programs

First of all lets see what type of parallelism is available in programs. Parallelism can be divided broadly into

4.1.1) Coarse-grain parallelism

This type of parallelism refers to the ability to divide a large program into smaller modules and then to dispatch these modules in a multiprocessor environment. This type of parallelism is exploited at the operating system level by some sort of scheduler.

4.1.2) Fine-grain of Instruction Level Parallelism

Instruction Level Parallelism (ILP) refers to the parallelism available at the finest levels in which multiple instructions are executed in a single cycle. Modern CPUs can execute multiple instructions concurrently. Two sources of parallelism are exploited:

- Some machines issue multiple instructions in one cycle $\Rightarrow$ superscalar machine
- Some machines overlap various execution phases of different instructions $\Rightarrow$ pipelining

ILP can be improved by reordering instructions known as instruction scheduling. During the process of instruction scheduling we select multiple instructions for parallel execution. However this selection is constrained by certain factors.

In the following pages we shall be considering the various types of methods and ways to enhance the optimization of the programs. We shall start with a general description of the compiler optimization techniques. Then we move on to the optimization techniques specific for the VLIW architecture. After discussing the most popular and widely adopted techniques we shall move on to discuss the optimization techniques that were used in the ME-2 C compiler.

4.2) Types of Optimizations

Compiler optimizations remove artificial constraints imposed by the programmer and the programming language, in order to increase the program's efficiency and expose its inherent parallelism. We have classified these optimizations into three levels: classical, superscalar, and multiprocessor.

4.2.1) Classical Optimizations

Classical optimizations are made up of two components, local and global optimizations. Local optimizations are applied to instructions within a basic block, and use no knowledge of the program as a whole (e.g., data flow analysis) to make optimization decisions. The local optimizations considered here are constant propagation, copy propagation, common sub expression elimination, redundant load/store elimination, constant folding, strength reduction, operation folding, constant combining, and code reordering. On the other hand, global optimizations are applied among operations within the same function.

The goal of classical optimizations is to reduce the execution time of a program by eliminating redundant instructions and replacing a set of instructions with a more efficient set. The effect of these optimizations on the available parallelism is not clear.

4.2.2) Superscalar Optimizations

Superscalar optimizations combine and enlarge basic blocks to expose more parallelism. The following superscalar optimizations are considered:

Superblock formation, loop unrolling, loop peeling, branch target expansion, induction variable expansion, memory disambiguation, and register renaming. A superblock is the basic scope for optimizations. Superblock formation consists of first combining basic blocks that tend to execute in sequence into a trace, and then performing code duplication to eliminate all side entrances from the trace. Loop unrolling replicates the body of a superblock loop several times. Loop peeling fully unrolls loops with small numbers of iterations. Branch target expansion copies the target superblock of a frequently taken branch into its fall-through path. Induction variable expansion removes the dependencies between induction variables in unrolled copies of a loop body. Memory disambiguation and register renaming are used to remove artificial dependencies between instructions.

Superblock formation and optimizations add additional bookkeeping instructions to the less frequently

4.2.3) Multiprocessor Optimizations

Memory renaming and data migration to high-speed memory are powerful compiler optimizations that uncover the inherent parallelism within an application program. Memory renaming refers to renaming all memory variables such that they only hold one value. Since a memory variable is never written more than once, all memory output and anti-dependencies are removed. Data migration refers to loading frequently used memory variables into high-speed memory such as registers. It is obvious that memory renaming will improve the parallelism because it removes data dependencies. However, the effect of data migration on parallelism depends on the level of data migration to high-speed memory.

This was a generalized discussion of the compiler optimization techniques. From here on we shall focus on the optimization techniques specifically used for the parallel mapping of DSP algorithms on VLIW architecture processors.

As mentioned before, there are two types of parallelism available. We shall be considering only the ILP. ILP can be improved by various techniques including loop unrolling, software pipelining, predicated execution and instruction scheduling. The first step, however is to analyze the source code and determine the dependencies that exist between the various instructions

4.3) Dependence Analysis

Determination of data dependences is a task typically performed with high-level language source code in today’s optimizing and parallelizing compilers. Very little work has been done in the field of data dependence analysis on assembly language code, but this area will be of growing importance, e.g. for increasing ILP. A central element of a data dependence analysis in this case is a method for memory reference disambiguation that decides whether two-memory operations may/must access the same memory location.

Let us define some notations that will be helpful towards the understanding of different types of dependencies. If we have two instruction S1 and S2 then the notation, S1 ◄ S2, implies that S1 precedes S2 in order of execution.

Dependence can be divided broadly into the following three categories.

- Resource Dependence
- Control Dependence
- Data Dependence

4.3.1) Resource Dependencies

It means that two instructions that use the same functional unit cannot execute at the same time.

4.3.2) Control Dependencies

It occurs as a result of the control flow of the program. So if S1 is control dependent on S2, we write

$$S1 \ \delta^{c} \ S2$$

4.3.3) Data Dependencies

This can be further classified [21] into the following four categories.

4.3.3.1) Flow Dependence:

Known as the read-after-write ***(RAW)*** hazard. Two instructions are said to be Flow dependent if *S1* ◄ *S2* and the former sets a value later uses. So we can say that

if **S1:** ***d*** = b*e

and **S2:** e = ***d***+1

then since S1 sets the value of ***d*** that is being used by S2 for calculating the value of e, so

$\boldsymbol{S1\ \delta^{F} S2}$

Here we can see that if S2 was to be executed before S1 that would change that outcome of the result.

4.3.3.2) Anti Dependence: Known as the write-after-read ***(WAR)*** hazard. Two instructions are said to be anti-dependent if *S1* ◄ *S2* and S1 uses a variable that is updated by S2. So we can say that

if **S1:** d = b****e***

and **S2:** ***e*** = d+1

then since S1 is using ***e*** which is changed or set by S2, so

$\boldsymbol{S1\ \delta^{A}\ S2}$

Here we can see that if S2 was to be executed before S1, that would change the outcome of the result and *d* would be holding an incorrect value.

4.3.3.3) Output Dependence: *K*nown as the write-after-write ***(WAW)*** hazard*.* Two instructions are said to be output dependent if *S1* ◄ *S2* and S1, S2 both set the same variable. So we can say that

if **S1:** ***d*** = b*e

and **S2:** ***d*** = e/2

then since S1 and S2 both set the variable ***d***, so

$\boldsymbol{S1\ \delta^{A}\ S2}$

Again the order of execution is of great importance.

4.3.4) Dependence Graphs

These dependencies are calculated in the analysis phase and dependence graphs are constructed. Directed Acyclic Graph ***(DAG)*** can be used to show the dependence between several operations. A node represents an operation and the edges represent dependencies between nodes. The type of dependency represented by an edge is unimportant, so we omit it. A dependence graph corresponding to each basic block of instructions is produced. A basis block ***(BB)*** is a structure holding instruction grouped together such that there is on control path into or out of the basic block except for the first and the last instruction. Thus the BB does not contain any jump or call instruction except for not necessarily the last instruction of the BB.

These dependence graphs are then used as the basic unit on which various optimizations are performed. Let us now briefly see the various optimization techniques for exploiting the ILP.

4.4) VLIW Compilers

Compilers for VLIW processors play a pivotal role in exposing instruction-level parallelism (ILP) for the effective utilization of hardware. Exploitation of fine-grain parallelism is a critical part of exploiting all of the parallelism available in a given program. The real challenge to using these architectures lies in compacting the code such that the semantics of the program are preserved. This is unlikely to be possible without a general solution to the “optimization” or “compaction” problem. Parallelizing compilers have yet to become as effective as programmers in their transformation task [8]. A compiler has to rely on user assertions and/or source code modifications to improve the quality of the code it generates.

The actual parallelism available in a program is limited by its dependences. Dependence between two program statements is a conflict that prevents the statements from executing concurrently. Dependences can be categorized into three types: resource, data and control. Resource dependence between two statements is usually a consequence of the limited hardware available in any physical computer system. The data dependences exist when the current instruction is dependent on the result of a previous instruction. Control dependences represent the conditional execution such as the if-then-else statements. Branch outcomes also fall into this category.

VLIW machines need low level programming. While developing parallel schedules, a programmer must keep in mind all the details of hardware design. The execution time for each instruction must be known prior to scheduling, so that optimal instruction scheduling can be done [4]. Since each instruction specifies multiple operations, these operations must have resources allocated separately. This is a time-consuming exercise and is very much prone to error. Compilation techniques are needed to relieve the programmer and make the entire process faster and less error-prone. Thus, the success of a VLIW processor depends heavily on the efficiency in code compaction.

Efficient compilers should be able to translate serial programs written at a reasonably high level into good parallel schedules. The optimality criteria generally used to judge these schedules are:

- $D_{i/o}$ input to output delay should be minimum.
- T_{ii} iteration initiation interval should be minimum.
- The number of processors/functional units should be minimum.

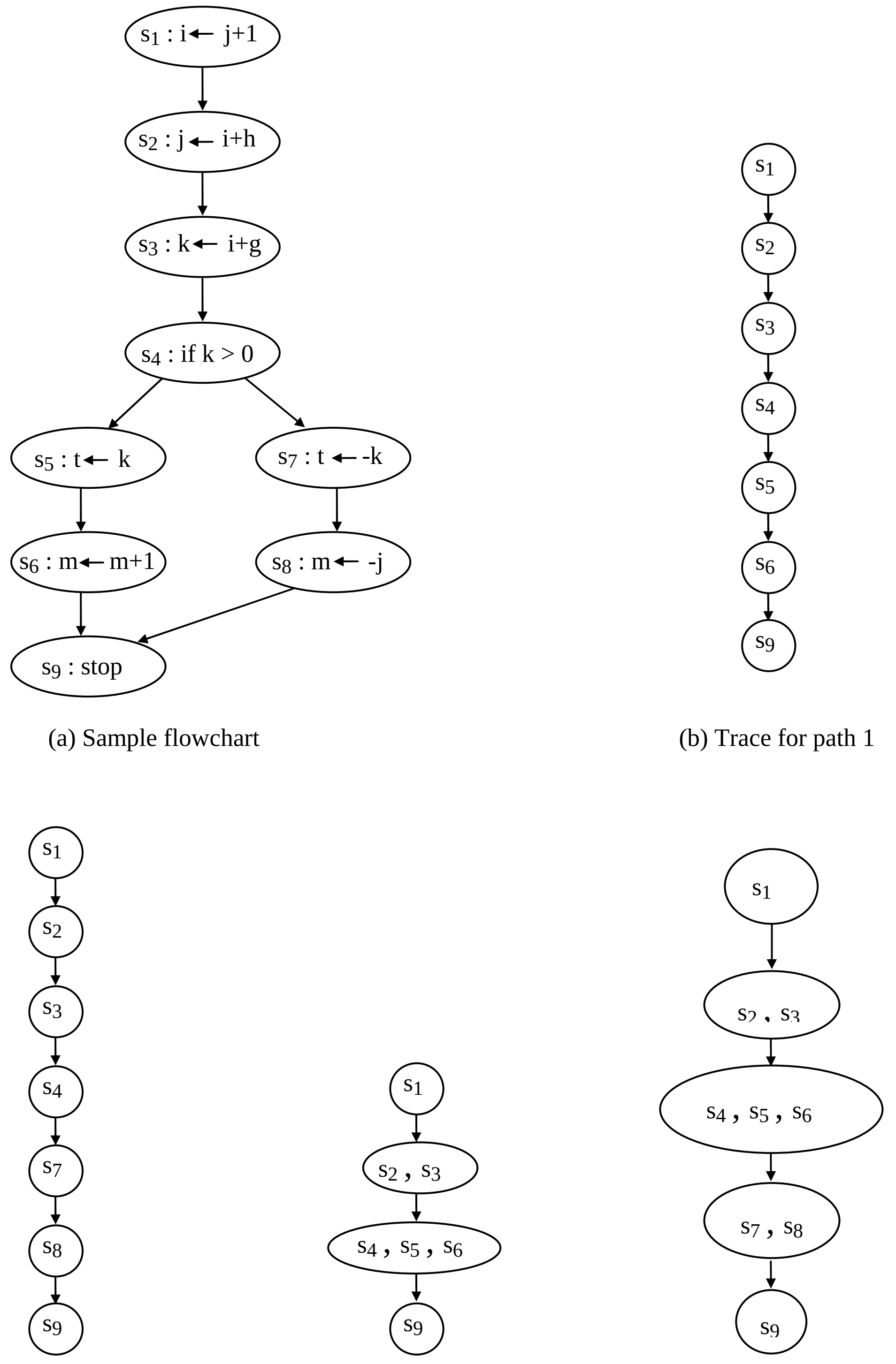
Scientific computations spend a significant amount of time in executing loops; therefore loops represent a critical component. Minimum value of iteration interval guarantees highest speed of execution of a loop. This speed often places a higher limit on the rate at which real-time processing can be done. Keeping the number of processors/functional units to a minimum ensures the lowest possible cost with which an optimal schedule can be achieved. Since optimal solutions are relatively more expensive in terms of time and resources, near-optimal solutions are used for faster and economical implementations [15].

4.5) Optimization Techniques For a VLIW Compilers

4.5.1) Trace Scheduling

Joseph Fisher offered Trace scheduling as a solution to the “optimization problem” [7] when efforts to use basic blocks for compaction did not prove very effective. This technique compacts large sections of code containing several basic blocks obtaining an overview of the program. Scheduling is then carried out, giving operations with longer delays a higher priority for placement. Otherwise these delays will percolate through the program. For example, load and branch operations are usually the most time consuming ones; these are scheduled as early as possible.

Trace scheduling operates on traces, which may consist of many basic blocks. It uses branch prediction and loop unrolling to statically look several basic blocks ahead for parallelism opportunities. The trace contains operations from the most probable path. Directed Acyclic Graphs (DAGs) are built for each path through the trace. These are also called the trace data precedence graph. These have been shown to contain all of the necessary restrictions on inter block motion and only those restrictions [7]. Sequence of instructions is then ordered to minimize the execution time of the most probable path. However, this speed up is usually at the expense of increasing the execution time of less frequently used paths through the program. Independent trace operations can be packed into the long instruction words providing simultaneous issuing of multiple operations per cycle. In case of conditional execution, compensation code is added to preserve the program's semantics when a branch prediction turns out to be incorrect. For example, Figure 4.1a shows sample code. A DAG is shown in Figure 4.1b and c for the two possible paths. Figure 4.1b depicts path 1, the more likely path through the program, while Figure 4.1c shows path 2, and the less likely taken path. Both the paths are identical up to s_4. Instructions s_5 and s_6 are scheduled in the same cycle as s_4 assuming the branch outcome to be true, while s_7 and s_8 are scheduled in the next cycle and will be conditional on the branch outcome being false. So path 2 will take 1 cycle longer to execute. If the false outcome had been the more probable path, we could have scheduled s_4, s_7 and s_8 in one cycle eliminating the extra cycle.



(a) Sample flowchart (b) Trace for path 1

(c) Trace for path 2 (d) Path 1 compacted. (e) Path 2 compacted.

Figure 4.1 Trace scheduling example.

4.5.2) Software Pipelining

Software pipelining techniques compute a static parallel schedule that overlaps the operations of several iterations analogous to a hardware pipeline that overlaps operation in a dynamic instruction stream. The schedule so computed is suitable for execution on VLIW machines. Most signal processing applications have static loops that are known to execute a certain number of times. These comprise a large portion of a program’s parallelism. Software pipelining concentrates on extracting this potential parallelism. Maximum parallelism available in a loop is limited by its data-dependencies and the target machine’s resource dependencies [12]. Before software pipelining is applied, a single loop is compacted to minimize its execution time. The compaction is limited by intra-iteration dependences. Then software pipelining is carried out to overlap operations from successive iterations.

The time that elapses between the issues of two iterations is called the initiation interval. The value of iteration initiation interval t_{ii} is chosen such that a new copy of the loop schedule can be issued every t_{ii} cycles to overlap the execution of operations from different iterations. This interval must be long enough to satisfy inter-iteration dependencies before dependent operations from subsequent iterations begin executing. In the worst case, dependence from the last instruction of one iteration to the first instruction of the next iteration could limit the execution to one iteration at a time. In this case, the entire iteration needs to be completed before the next one can be started.

The smallest acceptable initiation interval leads to the highest number of simultaneous operations and thus the maximum performance. An example of the software pipelining technique is shown in Figure 4.2. The C code loop is to be software pipelined for execution on a VLIW processor with two memory ports, one adder and one multiplier. Figure 4.3 shows the pseudo-assembly code for the loop. Since this loop has no inter-iteration dependences, only the available resources limit its maximum execution time. In the next step, a schedule is generated, in Table 2.1, for a single iteration that does not violate any of the dependences within an iteration. The resources are shown along the y-axis and the cycles along x-axis. Using the resource usage column we find that the minimum iteration period is three.

Table 4.2 shows the software-pipelined schedule for the loop or the steady state. The time required to reach the steady state is called the *pipeline fill* time. It is represented by the cycles labeled 0-5. Inside the steady state, a new iteration is started every three cycles. After the eighth cycle, the loop is repeated from cycle six again. In the steady state s_1, s_4, and s_5, are executing for the third iteration, s_2 is executing for the second iteration and s_3, s_6, and s_7, are executing for the first iteration. Thus the result of the first iteration will be available after eight cycles but subsequent iterations will complete every three cycles. If the number of iterations is known at compile time, the code size of a pipelined loop is within three times the code size for one iteration of the loop [11]. This owes to the fact that each pipeline must have a *pipeline fill* time and a *pipeline drain* time. Either of these can at most be equal to one iteration. Although software pipelining increases the total code size, compared to the unpipelined loop version, the steady state is typically much shorter than the length of the unpipelined loop. Thus it can be concluded that the increase in code size due to software pipelining is not an issue.

```
for(i=n;i>0;i--)
  {
     a[i]= e[i]+6;
     c[i]=b[i]*d[i];
  }
```

Figure 4.2. Example loop

		Operation	Latency
s_1	:	load e [i]	5
s_2	:	add e [i] + 6	1
s_3	:	store a [i]	1
s_4	:	load b [i]	5
s_5	:	load d [i]	5
s_6	:	multiply b [i] * d [i]	2
s_7	:	store c [i]	1

Figure 4.3. Pseudo-assembly code.

Cycles ⟶

Resource ↓

	0	1	2	3	4	5	6	7	8	Resource usage
Memory port 1	s_1	s_5								2
Memory port 2	s_4							s_3	s_7	3
Multiplier							s_6			1
Adder						s_2				1

Table 4.1. Single loop iteration

Cycles ⟶

Loop

Resource ↓

	0	1	2	3	4	5	6	7	8
Memory port 1	s_1	s_5		s_1^{*}	s_5^{*}		s_1^{**}	s_5^{**}	
Memory port 2	s_4			s_4^{*}			s_4^{**}	s_3	s_7
Multiplier							s_6		
Adder						s_2			s_2^{*}

Table 4.2. Software pipelined schedule.

4.5.3) Loop Unrolling

When the resources of the target machine are not fully utilized, the performance can be improved by unrolling the loop. The unrolling process exposes more instructions for parallel execution in a loop and hence utilizes the resources in a more efficient way [12]. For example, if the target machine is capable of executing four concurrent instructions but the three-cycle loop is such that only seven instructions are executing then we are using 58% of the available resources. But if we unroll the loop and execute 14 instructions every five cycles then resource usage increases to 70%. Now two iterations are being completed every five cycles instead of six. This means a 17% improvement in performance over the previous loop.

Figure 4.4 shows the previous loop unrolled. Figure 4.5 is the pseudo-assembly code. As Table 4.3 shows, the minimum iteration interval is now five cycles. Table 4.4 shows fourteen instructions are executing in the loop and two iterations are being computed in each loop execution. A loop can be unrolled completely so that the successive computations implied by the loop appear sequentially, or it can be partially unrolled as shown in the above example. Loop unrolling has two major advantages. First, the number of increments and tests is cut down by the unrolling factor. Secondly, more instructions are exposed for parallel execution. The disadvantage of loop unrolling is that it improves performance at the cost of code expansion. For this reason, the criteria for unrolling loops should include the size of the loop and the relative frequency of executing the loop [15]. A compiler needs complete knowledge of the hardware resources to sensibly unroll loops. Loops should only be unrolled by a factor that would result in a maximal usage of the resources. For example, consider a machine that has two multipliers and two adders. A loop that has one multiplication and one addition should not be unrolled by a factor of more than two. This would give maximum resource usage with minimum code expansion.

```
for ( i = n ;  i >o ;  i --)
{
a[i] = e[i] +6
a[i+1] = e[i+1]+6
c[i] = b[i] * d[i]
c[i+1] = b[i+1] * d[i+1]
}
```

Figure 4.4 Unrolled loop

```
s1  :  load e[i]
s2  :  load e[i+1]
s3  :  add e[i] +6
s4  :  store a[i]
s5  :  add e[i+1]+6
s6  :  store a[i+1]
s7  :  load b[i]
s8  :  load d[i]
s9  :  load b[i+1]
s10 :  load d[i+1]
s11 :  mul b[i] * d[i]
s12 :  mul b[i+1] * d[i+1]
s13 :  store c[i]
s14 :  store c[i+1]
```

Figure 4.5 Pseudo- assembly code

Cycles $\longrightarrow$

Resource $\downarrow$

	0	1	2	3	4	5	6	7	8	9	Resource usage
Memory port1	s_1	s_7	s_9						s_4	s_{13}	5
Memory port 2	s_2	s_8	s_{10}						s_6	s_{14}	5
Multipliers							s_{11}	s_{12}			2
Adder						s_3	s_5				2

Table 4.3 Single loop iteration

Cycles $\longrightarrow$

Resource $\downarrow$

Loop

	0	1	2	3	4	5	6	7	8	9
Memory port1	s_1	s_7	S_9			s_1^*	s_7^*	s_9^*	s_4	s_{13}
Memory port 2	s_2	s_8	S_{10}			s_2^*	s_8^*	s_{10}^*		s_{14}
Multipliers								s_{11}	s_{12}	
Adder							s_3	s_5		

Table 4.4. Software pipelined schedule

4.5.4) Register Scheduling

Traditional instruction scheduling methods minimize the number of registers used, which also reduces the degree of parallelism exploited. One key factor in this optimization is to make effective use of the target machine's registers. Registers provide fast access to operands as compared to retrieval from memory. The goal should, therefore, be to keep the most frequently accessed operands in the registers.

Allocating registers is a rather difficult optimization to perform. If register assignment is performed before scheduling, then software pipelining may produce poor results, because the register allocator may unnecessarily reuse registers, thus adding data dependences to the program [3]. The approach should be to start with an arbitrary allocation and then modify the register allocation during software pipelining.

Consider now the program fragment in Figure 4.6. In this example, operation s_2 is not available for scheduling at the start because its target register is one of the operand registers of operation s_1. However, if there is a spare register then the dependence can be broken by renaming the destination register of s_2 as in Figure 4.6(b). Now operation s_2 and s_1 can be scheduled in parallel. It is necessary to insert a register move s_2' into the program to restore the machine state after these operations. Here, the assumption is that the advantage gained in eliminating the dependence outweighs the cost of the extra copy.

Several different schemes are used for performing this allocation:

- A round-robin scheme can be used to allocate registers while the schedule is being generated.
- One can assume infinite number of registers to produce a schedule first and then allocate registers and add spill code.
- Another approach is to integrate register allocation with scheduling by keeping track of the *liveness* of registers.

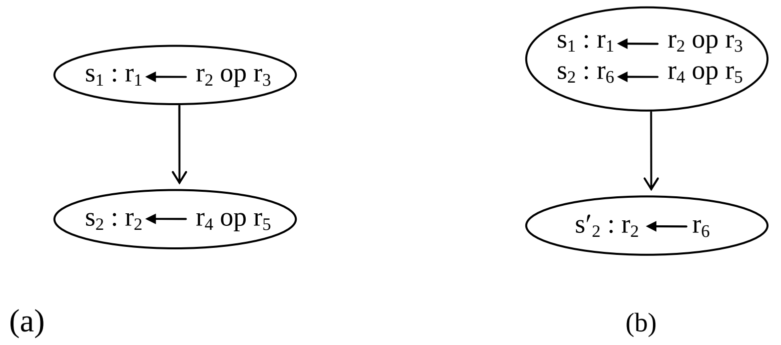

$s_1 : r_1 \leftarrow r_2$ op r_3
$s_2 : r_2 \leftarrow r_4$ op r_5
(a)
$s_1 : r_1 \leftarrow r_2$ op r_3
$s_2 : r_6 \leftarrow r_4$ op r_5
$s'_2 : r_2 \leftarrow r_6$
(b)

Figure 4.6. Register renaming

Section three

Project Specifications

The last section gave an overview of things that the reader must know in order to be aware of the project semantics. It also described the complete research phase of the project. After the research phase was over, we moved onto the next phase that comprised of the analysis and design phase. This section describes the project semantics and specifications. It includes the following

- Environmental Model
- Data Flow Diagrams
- Process Specifications
- Actors and Use Cases
- Class Diagram
- Sequence Diagram
- Classes—Responsibility, Collaborators (CRC) Cards
- Classes—Attributes, Operations

Chapter 5
Analysis and Design Specification

5.1) Environmental Model

The environmental model gives a brief introduction to the project. It gives the system, as it should appear in its final user environment

5.1.1) Statement of Purpose

“Implementation of a C/C++ compiler, with strong emphasis on low level optimization, for the VLIW-Architecture based ME-2 processor ”

The compiler shall take as input the standard C/C++ source code and convert it into fully optimized parallel assembly code. The compiler also caters for verifying the correctness of the input Source Code and performs Error-Detection, indicating the possible causes of the errors.

It is also required that the compiler shall be developed using modular approach so that several teams can work on different modules simultaneously. Our part deals mostly with the Serial Assembly Optimizer. The C/C++ source code is converted in to serial assembly with some high level optimization. Then this serial assembly is fed into the SAO module, which converts it into fully optimized parallel assembly

5.1.2) Context Diagram

This is the Top-level Data Flow Diagram, showing only at the top most level the different modules of the project.

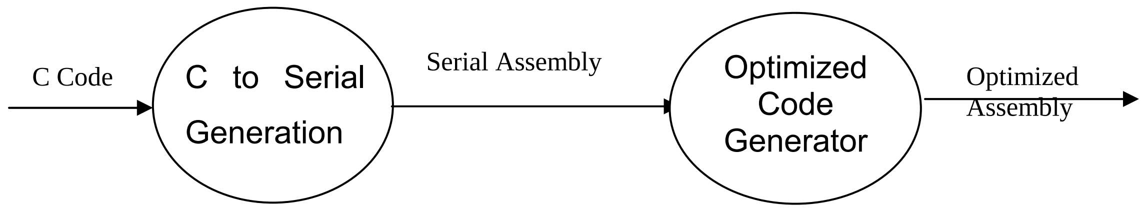


Figure 5.1 Context Diagram—*Level 0*

5.2) Data Flow Diagrams

LEVEL 1

1) Serial Assembly Generator

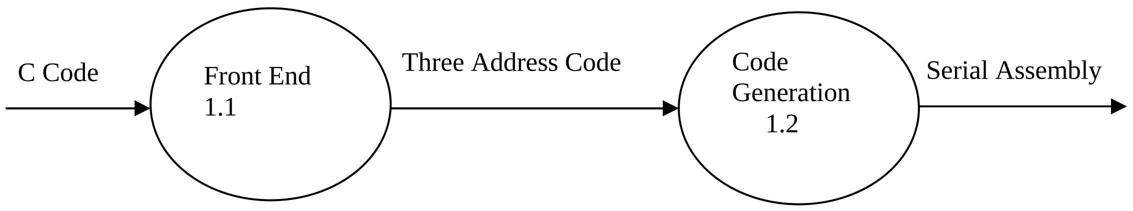


Figure 5.2 DFD (1)

2) Optimized Assembly Generator

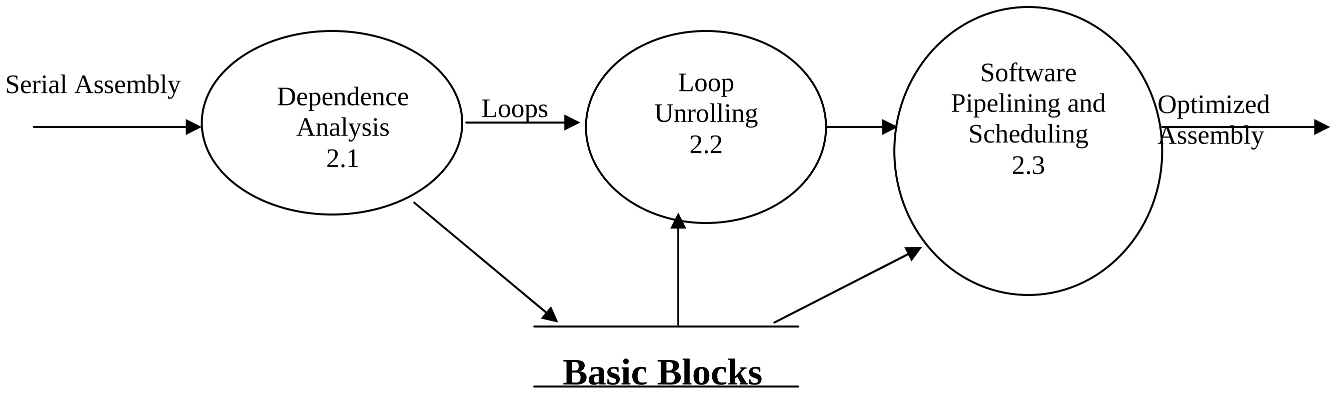


Figure 5.3 DFD (2)

LEVEL 2

1.1) Front End

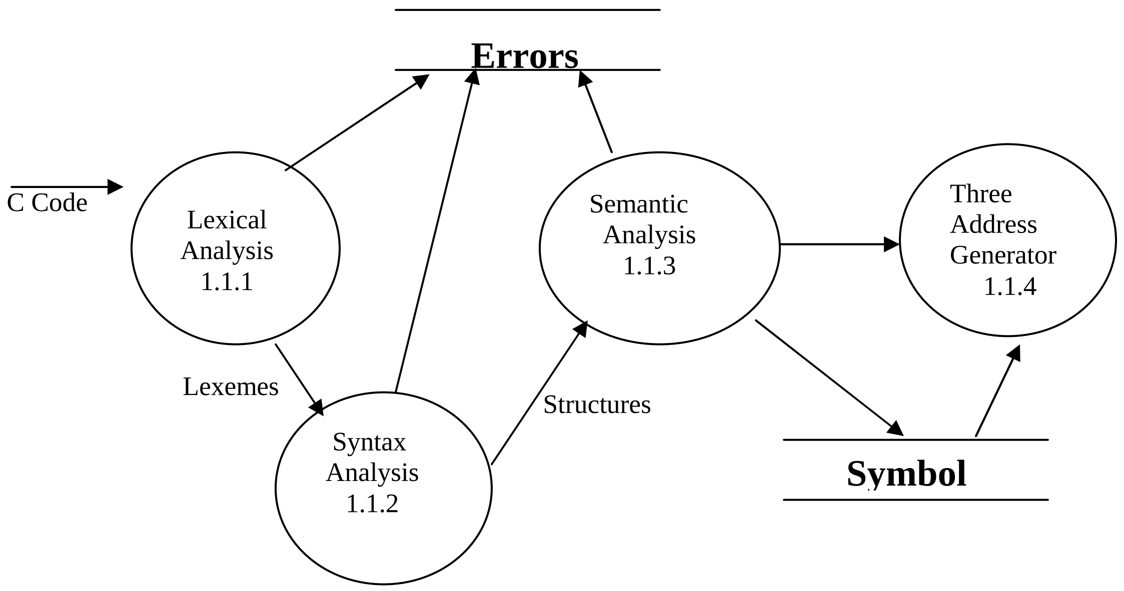


Figure 5.4 DFD (1.1)

1.2) Code Generation

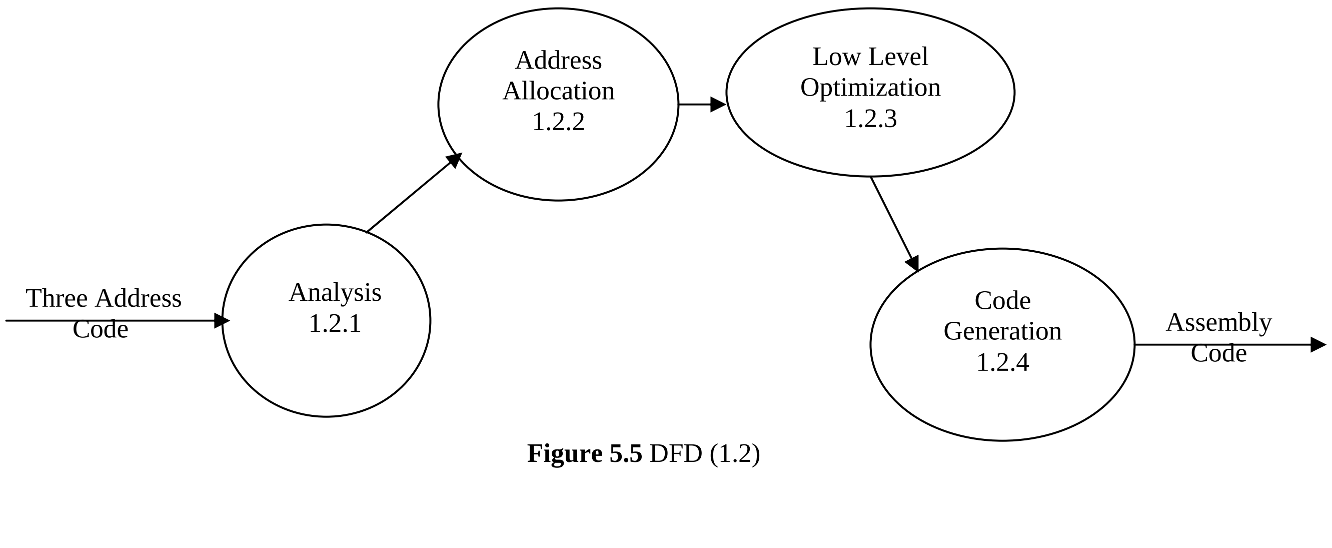


Figure 5.5 DFD (1.2)

5.3) Process Specification

5.3.1) Dependence Analysis (2.1)

The following steps are performed during the dependence analysis of the serial assembly code

- Flow graph construction
- Basic block construction
- DAG or other dependence graph construction
- Live variable analysis for each BB

5.3.2) Loop Unrolling (2.2)

The process of unrolling includes writing the code a repeated number of times and reducing the loop count by the unroll factor. This optimization helps to increase the ILP since there are more instructions available for scheduling now.

5.3.3) Software Pipelining and Scheduling (2.3)

The instruction-scheduling phase selects the various instructions for parallel execution. Software pipelining attempts to rearrange the sequence of instructions inside a loop, in order to minimize dependencies between such instructions, thus increasing the level of parallelization. The iteration of a software pipeline loop may contain instructions from a different iteration of the original loop.

This optimization is only applied to the innermost loops of small or moderate size, which contain no branches or function calls within the loop.

5.3.4) Lexical Analysis (1.1.1)

During this, the stream of characters making up the source program is read from left-to-right and grouped into *tokens* that are sequence of characters having collective meaning. The blanks separating the characters of these tokens are eliminated during lexical analysis.

5.3.5) Syntax Analysis (1.1.2)

Characters of tokens are grouped hierarchically into nested collections with collective meaning. The grammatical phrases of the source program are represented by a parsing structure, which describes the syntactic structure of the input.

5.3.6) Semantic Analysis (1.1.3)

Certain checks are performed during this process to ensure that the components of a program fit together meaningfully. This process checks the source program for semantic errors and gathers type information for the subsequent code-generation phase. It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operands of expressions and statements.

5.3.7) Three-address Generation (1.1.4)

After syntax and semantic analysis, an explicit intermediate representation of the source program is generated. This is sort of a program representation for an abstract machine.

5.3.8) Analysis (1.2.1)

This phase gathers up the information about the flow of the program from the three-address code. This information divides the program into basic blocks of code with links showing the flow between these basic blocks.

5.3.9) Address Allocation (1.2.2)

During this phase the addresses are assigned to the different variables used in the program. This includes the transient variables as well as the temporary variables generated by the three-address code generation phase. Mostly we try to keep all the variables in the register memory.

5.3.10) Low-Level Optimization (1.2.3)

This phase attempts to improve the intermediate code, so that faster-running machine code will result. This optimization are most of the times trivial but are necessary in order to remove the redundant code introduced by automated generation of code.

5.3.11) Code Generation (1.2.4)

The final phase is the generation of the target code, consisting of assembly code. Intermediate instructions are translated into a sequence of machine instructions that perform the same task.

These process narratives help us define the flow of data, and what transformations are performed on the data during this flow.

5.4) Use Case Diagram

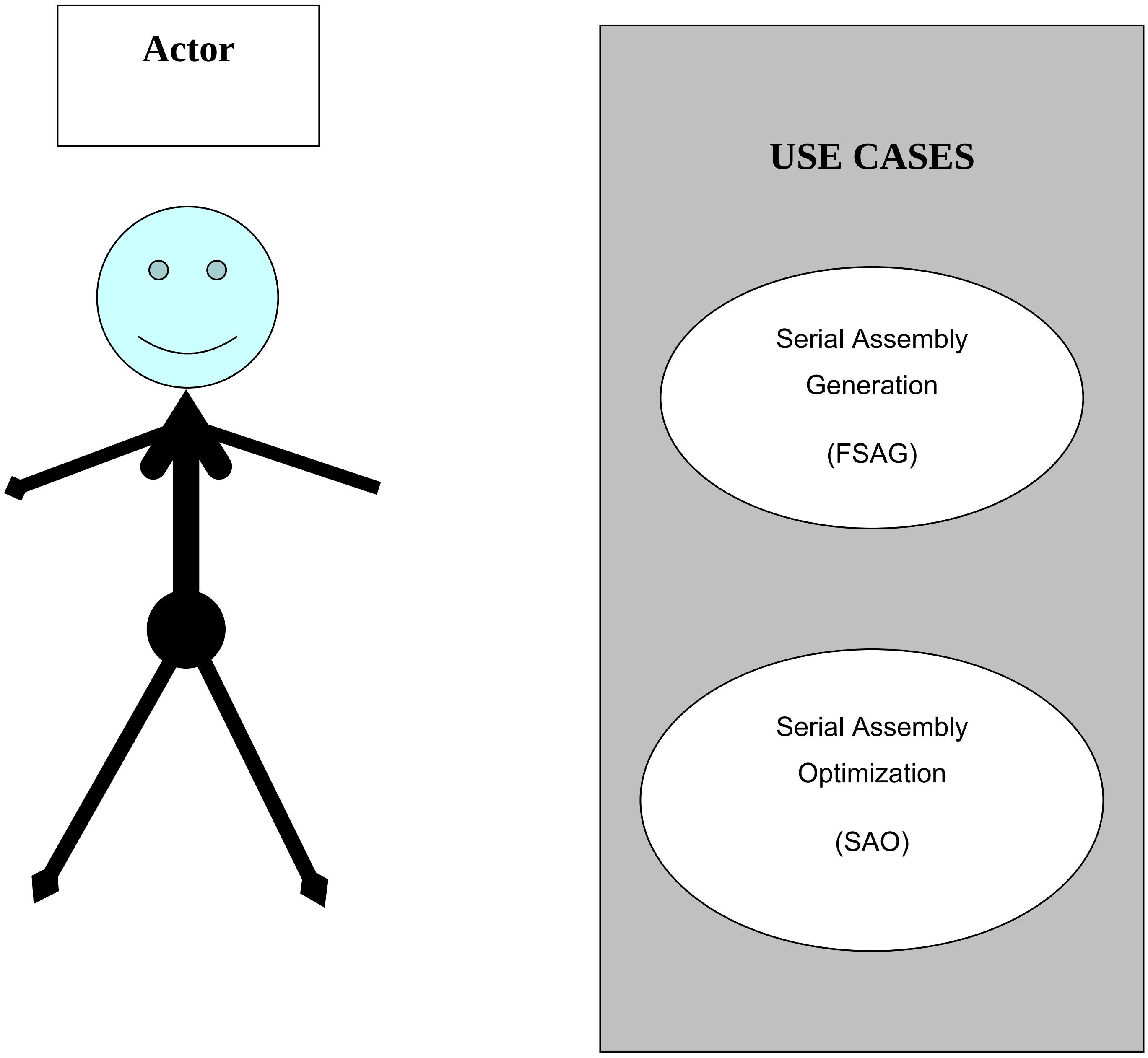

Actor
USE CASES
Serial Assembly Generation (FSAG)
Serial Assembly Optimization (SAO)

Figure 5.6 Use case Diagram

5.4.1) Actor—Programmer

An actor is a user of the system in a particular role. In our system there is only one actor that is the user of the compiler. From here on the actor is called the programmer.

5.4.2) Use Case Description

Use cases describe the system from the user point of view. The working and functionality of any compiler is hidden from the user. Most of the times it's just a text editor with a toolbar that allows the user to compile the source code. Our system consists of the following high-level use cases

5.4.2.1) Serial Assembly Generation

The programmer writes the C code in the text editor. He has the option to save or edit existing works. He then selects the compile command. In reaction to this command, the FSAG compiles the source code. First it analyzes the input code. During this analysis it performs lexical analysis, syntax analysis and semantic analysis. Then it generates the intermediate code. This intermediate code is used to generate the serial assembly of ME-2.

5.4.2.2) Serial Assembly Optimization

The input is fed into the SAO. This input comes from either the GCC or directly from the programmer. The programmer then selects the desired optimizations from the menu. He then hits the optimization command. This results in the analysis of the code, during which dependencies are calculated. This dependency information is then used for the various optimizations.

5.5) Class Relationship Collaborators

Class Name:	Lexical Analyzer
Class Type:	Generic
Derived From:	
Class Characteristics:	Sequential, Transient
Responsibilities:	**Collaborators:**
Getting Input	
Generating Tokens	Token
Setting Token Types	Token
Setting Numerals	

Class Name:	Parser
Class Type:	Generic
Derived From:	
Class Characteristics:	Sequential, Transient
Responsibilities:	**Collaborators:**
Grouping of Tokens	Token, Instruction
Checking for Errors	
Syntax analysis	
Semantic Analysis	
Identification of variables	Symbol Table

Class Name:	Symbol Table
Class Type:	Generic
Derived From:	
Class Characteristics:	Sequential
Responsibilities:	**Collaborators:**
Maintains record if variables	

Class Name:	Descriptor
Class Type:	Generic
Derived From:	
Class Characteristics:	Transient
Responsibilities:	**Collaborators:**
Maintain information about data register allocation	
Maintain information about Address register allocation	

Class Name:	Token
Class Type:	Generic
Derived From:	
Class Characteristics:	Transient
Responsibilities:	**Collaborators:**
Keeps track of Token lexemes	
Keeps track of Token Types	
Contains Numeral Values	

Class Name:	Instruction
Class Type:	Generic
Derived From:	
Class Characteristics:	Sequential, Transient
Responsibilities:	**Collaborators:**
Maintains information about instruction type	
Information about Register used by the instruction	Token

Class Name:	Basic Block
Class Type:	Generic
Derived From:	
Class Characteristics:	Sequential, Transient
Responsibilities:	**Collaborators:**
Contains List of Instruction	Instruction
Information about the Control Flow	
Live variables	

Class Name:	Code Generator
Class Type:	Generic
Derived From:	Object
Class Characteristics:	Sequential, Transient
Responsibilities:	**Collaborators:**
Allocation of variables to Address Registers	Register Descriptor
Allocation of values to Data Registers	Register Descriptor
Translation of intermediate instruction into Assembly Instructions	

Class Name:	Register
Class Type:	Generic
Derived From:	Token
Class Characteristics:	Sequential, Transient
Responsibilities:	**Collaborators:**
Maintains information about availability	
Maintains information about register contents	

Class Name:	Optimizer
Class Type:	Generic
Derived From:	
Class Characteristics:	Sequential, Transient
Responsibilities:	**Collaborators:**
Performs Loop Unrolling	
Software pipelining of the inner loops	
Instruction Scheduling of the basic blocks	
Performs live variable analysis	

5.6) Class Relationship Diagram

Figure 5.7 Class Diagram

5.7) Sequence Diagrams

5.7.1) Serial Assembly Generation

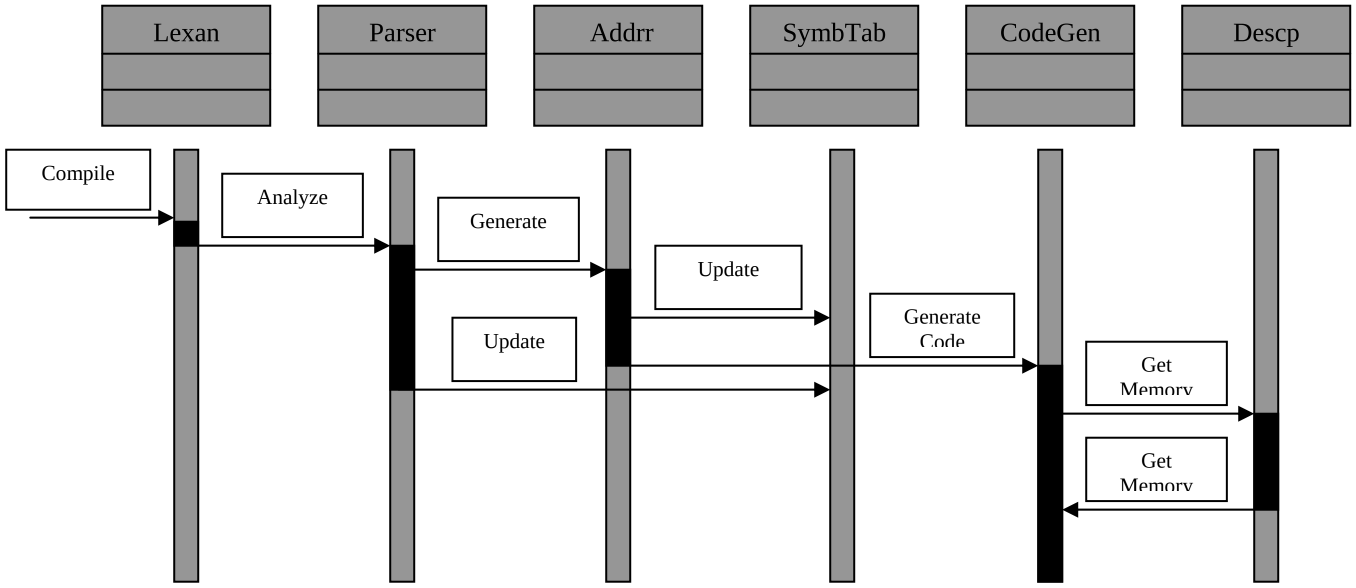


Figure 5.8 Sequence Diagram—Serial Assembly Generation

5.7.2) Serial Assembly Optimization

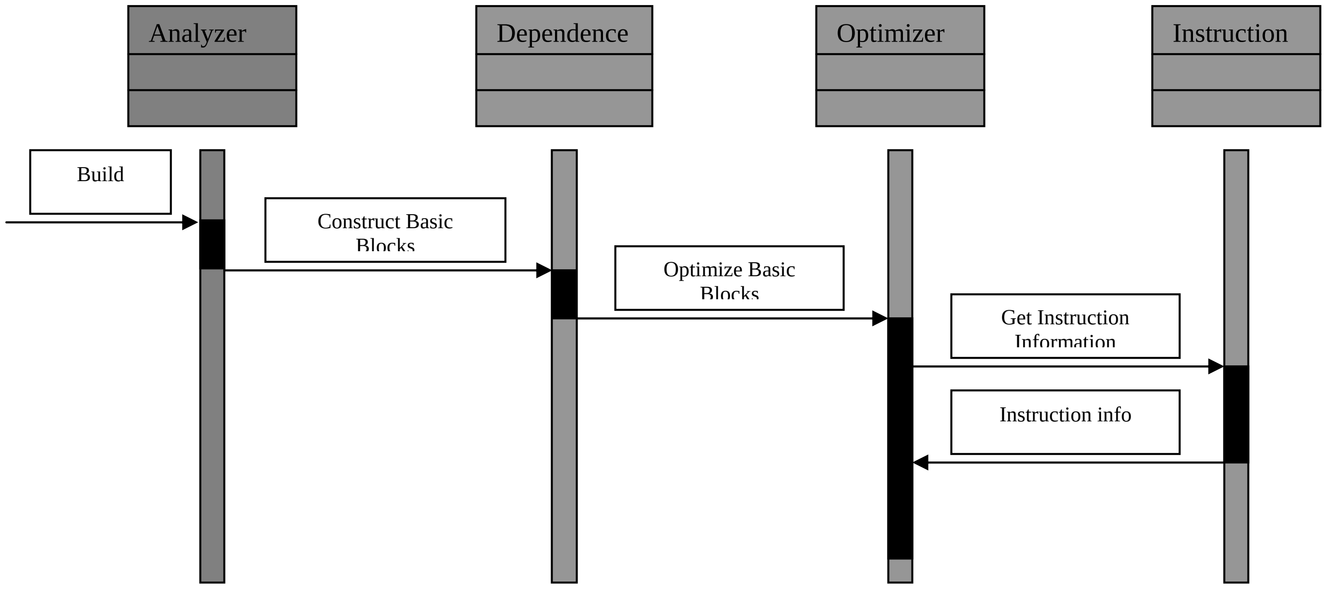


Figure 5.9 Sequence Diagram—Serial Assembly Optimization.

5.8) Class—Attributes, Methods

CInstruction

ATTRIBUTES

CString	m_InstructionString
CLinkList <CToken>	m_LLRegistersUsed
CLinkList <CToken>	m_LLRegistersRead
CLinkList <CToken>	m_LLRegistersWritten

METHODS

CInstruction ()
~CInstruction()
void UpdateRegistersInInstructionString( )
UpdateLoopCountInRepeatInstructions()
CInstruction()
operator= (CInstruction Inst)

CLinkList

ATTRIBUTES

Derived data members only

METHODS

CLinkList( )
CLinkList(CLinkList & L1)
~CLinkList( )
operator += (CLinkList<T> L1)
operator = (CLinkList<T> L1)
CheckLinkList( )

COptimizer

ATTRIBUTES

CBasicBlock	m_BasicBlock
CBasicBlock	m_PrologueForUnroll
CBasicBlock	m_EpilogueForUnroll
CLinkList<CFinalNodeForSP>	m_PrologueForSP
CBasicBlock	m_EpilogueForSP
CLinkList <CToken>	m_LLOfTokens
CLinkList <CAccumulatorInstruction>	m_LLOfAccumulatorInstructions
CUIntArray	m_InstScheduledForRes
CLinkList<CInitialNodeForSP>	m_LinkListForSP
CUIntArray	m_LinkListOfAvailable
CLinkList<CFinalNodeForSP>	m_LLOfFinalNodeForSP
CLinkList<COperationScheduled>	m_LLOfOperationsScheduled
ResourceTable	m_ResourceTable
CLinkList<CCombinedInstructions>	m_LinkListOfCombinedInstructions
CCombinedInstructions*	m_CombinedInstructionsForANode

METHODS

COptimizer( )
~COptimizer( )
DecrementDelay( )
IfNoRestriction( )
UpdateRestrictions( )
InitializeAccumulatorInstructions( )
FindAccumulatorRegisters( )
FindStartRegister( )
MINUS( )
UNION( )
INTERSECTION( )
REMOVECOMMON( )
MINUS( )
UNION( )
INTERSECTION( )
REMOVECOMMON( )
ArrangeInAscendingOrder ( )
UnRollInnerMostLoop( )
InitializeBasicBlock( )
RemoveDigits( )
SoftwarePipelineInnerMostLoop( )
SchedulingSimpleBasicBlock( )
CalculateNodeps( )

Available( )
GetNodeFromLinkListForSP( )
Depends( )
CalculateLiveNX( )
CopyInitialNodeForSP( )
ScheduledBefore( )
AddScheduledOperations( )
ScheduleState( )
ScheduleStateForSBB( )
Schedule( )
ScheduleForSBB( )
UpdateOne( )
GetMinimumIteration( )
GetMaximumIteration( )

ClexicalAnalyser

ATTRIBUTES

CString	m_strInput
int	strIndex
CArray<CStmt,CStmt&>	stmtArray

METHODS

CLexan()
~CLexan()
GetBlockCount()
SetID()
GetBlockLeaders()
SetBasicBlockLeaders()
SetInput(CString str)
FormatInput()
GenerateTokens()
FormatInput()
NextToken()

Cparser

ATTRIBUTES

CSymbolTable*	SymbolTable
CLexan*	lex
CString	tracefilename
CString	output
int	aux_index
int	lebel
FILE*	trace
FILE*	fcode
bool	m_btrace_open
bool	m_bcode
CCode	code
CToken	lookahead
CArray<CToken,CToken&>	tary

METHODS

CParser()
~CParser()
Parse()
ShowTrace()
MakeTokens()
ClearAllTokens()
AddToken(
GetAt()
GetTarySize()
f_data()
f_body(
type()
var_decl(int)
match_Id_data(int)
match_Id_body()
stmtp()
assg()
boolexp()
Isboolexp()
Isboolexpp()
expr()
exprp()
term()
factor()

match()
AuxToken()
CallExpr()
expr1()
term1()
factor1()
match1()
ErrorMsg()
NextToken()
NewLebel()
OpenFiles()
CloseFiles()
tmsg(CString)

CDescriptor

ATTRIBUTES

RegInfo	addrReg[16]
AddressInfo	addrRec[100]
int	memory[2000]
int	index
int	memIndex

METHODS

getMem()
getAReg()
updateAddressRec()
addAddressInfo()
addAddressInfo()
getAddressInfo()
delAddressInfo()
getIDAt()
getReg(g)
getID()
addRegInfo()
delRegInfo()

CSymbolTable

ATTRIBUTES

CArray<CToken,CToken&>	stary

METHODS

lookup()
lookup(
Insert()
Insert()
GetLastEntry()
GetSize()
GetAt()
ClearAllTokens()
InitSymbolTable()

CCell

ATTRIBUTES

CString m_id	
int	m_val
int	m_flag

METHODS

CCell()
CCell(CString,int=1)
CCell(int,int)
GetStr()
GetFlag()
GetString()
GetVal()
operator=()

Resource Table

ATTRIBUTES

CUIntArray	m_Agu
CUIntArray	m_Mac
CUIntArray	m_Shift

METHODS

ResourceTable( )
InitializeResourceTable( )
IsResourceFree(int type)

CToken

ATTRIBUTES

UINT	ID
int	m_nValue
CString	m_strTitle

METHODS

CToken()
CToken()
CToken(
~CToken()
IsConstant()
GetID()
GetValue()
GetTitle()
SetValue()
SetID()
SetTitle()

CThreeAddress

ATTRIBUTES

CCell	m_arg1
CCell	m_arg2
int	m_op

METHODS

- CThreeAddress()
- CThreeAddress(int,CCell&,CCell&)
- ~CThreeAddress()
- GetStr()

CBasicBlock

ATTRIBUTES

CLinkList<CInstruction>	m_LLInstruction
CLinkList<CToken>	m_LLLiveVariables
int	m_nNextBlock//
int	m_nBno

METHODS

- CBasicBlock()
- ~CBasicBlock()
- CBasicBlock(CBasicBlock &bb)
- operator =()
- SetBlockNum()
- GetBlockNum()
- SetNextBlockNum(int num)
- GetNextBlockNum()

CCodeGenerator

ATTRIBUTES

CLexan*	pLexan
CAddressDescp*	pAddD
CRegisterDescp*	pRegD
VarInfo	varLive[100
CArray<CBasicBlock,CBasicBlock&>	bbArray
CStringArray	strFinal

METHODS

CCodeGenerator()
~CCodeGenerator()
SetStmtTypes()
CodeGeneration()
GenerateCode()
SetBasicBlocks()
LoopCode()
AssgtCode()
RelOpCode()
ConditionalCode()
BinOpCode()
LabelCode()
JumpCode()
UpdateVarInfo()
VarDeclaration()

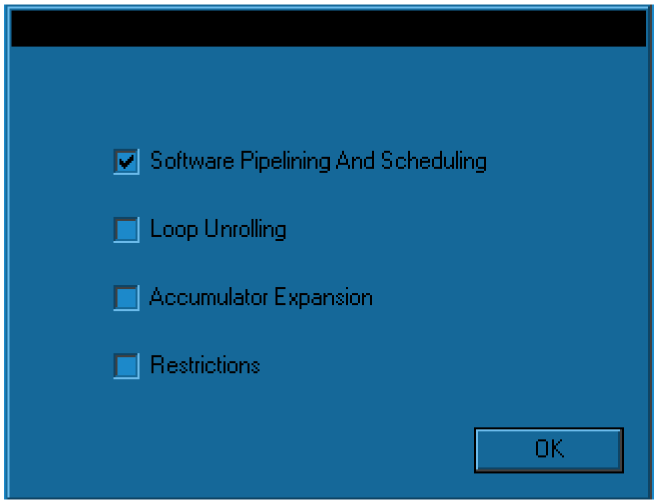
Section four

Results and Conclusion

Here we list the results we were able to achieve due to the various optimizations. The results are confined only to the SAO since the FSAG was only a prototype and no results were expected out of it. This section provides recommendation for anyone who is interested in any such future work.

I- Results

The following results show how much optimization was achieved when different optimization levels were selected.

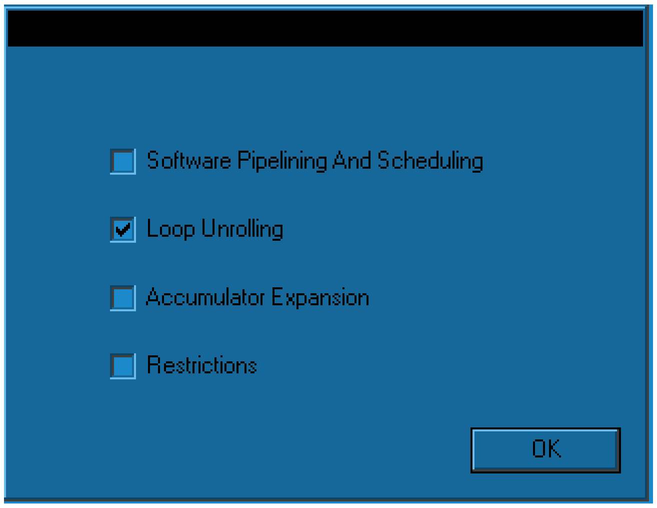

[✓] Software Pipelining And Scheduling
[ ] Loop Unrolling
[ ] Accumulator Expansion
[ ] Restrictions

```
////////////////////////Input Code/////////////////////////
          repeat(40)
          {
            r0 = *(a1)++;
            r1 = *(a2)++;
            r2 += r0 * r1;
          }
```

Simple Serial Code
Total Cycle Count: 40 * 3 = 120

```
//////////////////////Output///////////////////////////////
  r0=*(a1)++       ;
  repeat(39)
  {
                  r1=*(a2)++    ;
                  r2+=r0*r1     ||
                  r0=*(a1)++    ;
  }
  r1=*(a2)++
  r2+=r0*r1
```

Software Pipelined Code
Total Cycle Count: 39 * 2 = 78



☐ Software Pipelining And Scheduling

☑ Loop Unrolling

☐ Accumulator Expansion

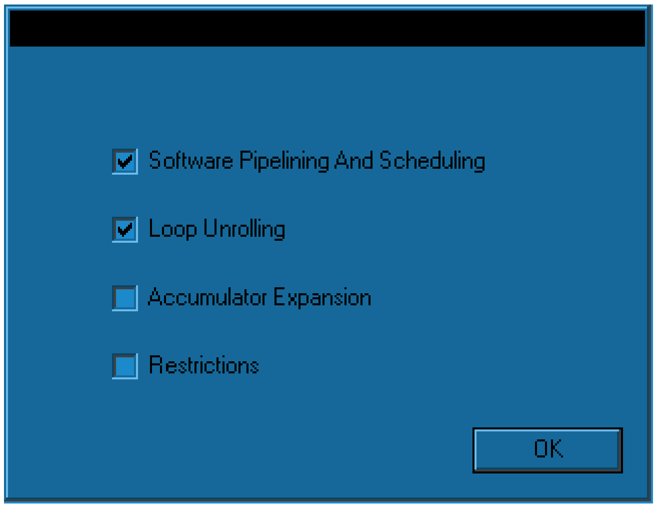
☐ Restrictions

```
////////////////////////Input Code////////////////////////
          repeat(40)
          {
            r0 = *(a1)++;
            r1 = *(a2)++;
            r2 += r0 * r1;
          }
------------------------Output---------------------
  repeat(10)
  {
    r4=*(a1)++
    r8=*(a2)++
    r2+=r4*r8

    r5=*(a1)++
    r9=*(a2)++
    r2+=r5*r9

    r6=*(a1)++
    r10=*(a2)++
    r2+=r6*r10

    r7=*(a1)++
    r11=*(a2)++
    r2+=r7*r11
  }
```

Simple Serial Code
Total Cycle Count: 40 * 3 = 120

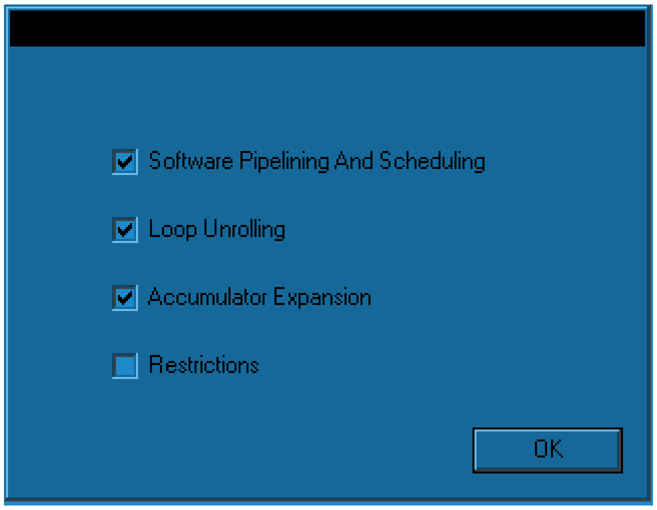
Unrolled Code
Here the code is not parallelized.

It is only shown here to show the result of this particular algorithm.


[✓] Software Pipelining And Scheduling
[✓] Loop Unrolling
[ ] Accumulator Expansion
[ ] Restrictions

```
r4=*(a1)++    ||
r5=*(a1)++    ||
r6=*(a1)++    ||
r7=*(a1)++     ;

repeat(9)
{
  r8=*(a2)++    ||
  r9=*(a2)++    ||
  r10=*(a2)++   ||
  r11=*(a2)++    ;
  r2+=r4*r8      ;
  r2+=r5*r9      ;
  r2+=r6*r10     ;
  r2+=r7*r11    ||
  r4=*(a1)++    ||
  r5=*(a1)++    ||
  r6=*(a1)++    ||
  r7=*(a1)++     ;
}
r8=*(a2)++
r2+=r4*r8
r9=*(a2)++
r2+=r5*r9
r10=*(a2)++
r2+=r6*r10
r11=*(a2)++
r2+=r7*r11
```

Software Pipelined and Unrolled Code
Total Cycle Count: 9 * 5 = 45


[✓] Software Pipelining And Scheduling

[✓] Loop Unrolling

[✓] Accumulator Expansion

[ ] Restrictions

```
r4=r2
r5=0
r6=0
r7=0
r8=*(a1)++    ||
r9=*(a1)++    ||
r10=*(a1)++   ||
r11=*(a1)++    ;
repeat(9)
{
  r0=*(a2)++    ||
  r1=*(a2)++    ||
  r2=*(a2)++    ||
  r3=*(a2)++     ;
  r4+=r8*r0     ||
  r5+=r9*r1     ||
  r6+=r10*r2    ||
  r7+=r11*r3    ||
  r8=*(a1)++    ||
  r9=*(a1)++    ||
  r10=*(a1)++   ||
  r11=*(a1)++    ;
}
r0=*(a2)++
r4+=r8*r0
r1=*(a2)++
r5+=r9*r1
r2=*(a2)++
r6+=r10*r2
r3=*(a2)++
r7+=r11*r3
r4=r4+r6
r5=r5+r7
r4=r4+r5;   r2=r4
```

Software Pipelined, Unrolled and Accumulator Expanded Code

Total Cycle Count: 9 * 2 = 18

II- Analysis and Conclusions

From the above shown results it is quite obvious that the targeted time optimization levels have been achieved. However, if the user is constrained by any memory limitations, then the optimizations can be turned off which would result in space optimized code. It is important to note that we have aimed only for time optimizations, code size was not a limiting factor. The increase in code size is usually by a known factor; hence it can be estimated at run-time if the code size would reach unwanted limits.

We were able to achieve most of our goals. While setting these goals, we kept in our mind the time limitations. There are some optimization features that can still be incorporated in the SAO. These are mentioned in the recommendations.

III- Future Recommendations

Compiler optimization is an ever-progressing field. Researchers all over the world are in the process of discovering and developing new techniques. These different optimization techniques work miraculously on different VLIW processors. There are some advanced features provided in ME-2 that would make time optimizations even more efficient. However, we have tried to follow suit of some already implemented standard optimization techniques.

Following are some of the interesting advancement that can be made into the ME-2 compiler

- ➢ The GNU c compiler used as the front-end serial code generator contains a lot of built in optimization. However, due to poor documentation and help material its very difficult to use them efficiently. Some work can be done to extract and use the built-in optimizations from the GNU C Compiler.
- ➢ ME-2 has some advanced architectural features like the delay line and the data pointer registers. These have so far not been incorporated into the SAO. Though there is no standard technique for implementation of these

techniques, however, if time and resources are given, new techniques can be and should be developed.

This brings us to the end of this dissertation. We have tried our best to present it in the truest form what our whole project was all about.

Appendix A

Serial Assembly Format

The Serial Assembly Optimizer (SAO) is the low-level optimizing module of the ME 2 C compiler. It converts the input serial assembly into Optimized Parallel Assembly. Since ME-2 is a VLIW processor designed to run DSP algorithms, thus SAO strongly consider the architectural features of ME 2 and implements optimization techniques used for efficient mapping of DSP Algorithms on VLIW processors.

The input to the SAO may come from:

- The front-end C compiler converting C/C++ code into serial assembly
- Hand coded Serial Code from a Programmer

In both the cases the compiler or the programmer do not have to concern with the various issues regarding

- Parallel execution of the various instructions
- Instruction Latencies

However, a standard format of the Serial Assembly must be established. There is the issue of whether the intermediate form is appropriate for the kinds and degree of optimization to be performed. Some optimizations may be hard to do at all on a given intermediate representations (IR), and some may take much longer to do than they would on another representation. In the present case IR is actually the serial assembly, which uses the same instruction set as the final optimized parallel assembly. The problem arises due to the fact that there are too many ways of doing the same thing and everyone was born with a unique mind. The instruction set has ample instructions to confuse the best amongst us. So it is quite natural that the two programmers, hand coding the same C code, may translate it into completely different Serial Assembly code, thus justifying the need for this document.

The following pages contain several examples showing the C code as well as the expected input serial assembly. This format is optimal for the implementation of the architecture specific optimization techniques. The following discussion also includes why, in some cases, the specified format is desired.

Note: *this document at present does not cover the various issues regarding the specific use of registers. It has been assumed that no registers are being used for special purposes. However, it is expected that these issues shall be covered*

A-1) Register Allocation

Registers should be used with as little re-use of register as possible. Some high level optimization may try to reuse register by saving the contents of a register in memory. This should be avoided and the registers may only be reused when there is no empty register available.

It may not be necessary to declare certain variables in the memory. Instead, the variables can be assigned to registers directly and used/modified without any load store from the memory. If a variable is declared and used within the same function or block, then there is no need for its declaration in the assembly code. We can directly use a register. How ever, if the variable is a pointer, or an array or a reference, than it needs to be stored at the end of the block or procedure.

A-2) For Loops

For loops should be replaced with repeat statements of the serial assembly. The loop count can be a constant value, or a value in a register. The value in a register can be loaded from a memory location or it can be decided at run-time.

A-3) If-else and Predicated Execution

While dealing with if-else statements, it is desirous to have as much use of IFT, IFF and IFA constructs as possible. However in certain unavoidable circumstances, conditional jumps may also be used. But the top priority should go to predicated execution of such statements. There is a dependency that if any AGU instruction is dependent on a ‘t’ bit, then there should be at least a difference of two cycles between the setting of ‘t’ bit and the use of that AGU instruction. So if most of the AGU instructions in the code are dependent on ‘t’ bit, then instead of ‘t’, ‘b’ bits should be used. But if we have only the MAC instructions that depend upon the ‘t’ bit, then ift, ifj and ifa should be used.

Here are some code samples

A-4) Auto Correlation

The C code below might seem quite frightening and the Serial Assembly surprisingly simpler. The arrays should be handled with the following shown procedures. The address registers are assigned to specific locations (other address registers) in the outer loop and the loop counter

```
for (i = 1; i <= m; i++)
{
          sum = 0;
    for(j=0; j<L_WINDOW-i; j++)
      sum = L_mac(sum, y[j], y[j+i]);

    sum = L_shl(sum, norm);
    r_32[i] = sum;
}
///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
here   sum = L_mac(sum, y[i], y[i+j] ) means
  sum += y[i] * y[i+j]
/////////////////////////////////////////////////////////////////

short y[240];
int r[12];

A0=&y;
A5=&r;
R12=L_WINDOW
```

```
A3  = A0;

REPEAT (#20)
{
  A1  = A3++;
  A2  = A0;
  R0  = #0;
  R12 --;

  REPEAT (R12)
  {
    R1  =  *(A2) ++;
    R2  =  *(A1) ++;
    R0  +=  R1*R2;
  }
  R0  = norm(R0);
  *(A5)++ = R0;
}
```

C Code	Assembly

Loading of Short and Integer Values
short a;
int main()
{
  a = 3;
  return 0;
}
	`//A0->a`
	
	`main()`
	`{`
	`  int a;`
	`  A0   = &a;`
	`  R0   =  #3`
	`  *A0  =  R0.i`
	
	`/* Use a new register even if R0 is free for use now*/`
	`  R1   =  #0`
	`  return  R1`
	`}`

Addition of Short Values
 	
short a,b,c;
int main()
{
  a = 3;
  b=13;
  c=a+b;
  return 0;
}
	`// A2->c`
	
	`main()`
	`{`
	`  int c;`
	`  A2  = &c`
	`  R0  = #3`
	`  R1  = #13`
	`  R2  = R0 + R1`
	`  *A2 = R2.i`
	`  R3  =  #0`
	`  return  R3`
	`}`

Multiple Expressions
 	
short a,b,c,d;
 	
int main()
{
  a = 3;
  b=13;
 	
  d=a+b+c;
  return 0;
}
	`//A0->c , A2->d`
	
	`main()`
	`{`
	`  int c,d;`
	`  A0  = &c`
	`  A2  = &d`
	`  R0  = #3`
	`  R1  = #13`
	`  R2.i= *A0`
	`  R0  = R0 + R1`
	`  R3  = R0 + R2`
	`  *A2 = R3.i`
	`  R4  = #0`
	`  return R4`
	`}`

C Code | **Assembly**

If else

C Code:

```
short a,b,c,d;

int main()
{
  a = 3;
  b=13;

  if (a>b)
    b=5;
  else
    a=5;
  return 0;
}
```

Assembly:

```
main()
{
  int a ;
  int b ;
  A0  = &a
  A1  = &b
  R0  = 3
  R1  = 13
  T1  = R0>R1

  IFT1 R1 = 5;
  IFF1 R0 = 5;

  R0  = *A0
  R1  = *A1

  RETURN 0;
}
```

Nested If

C Code:

```
short a,b,c,d;


int main()
{
  a = 3;
  b=13;

  if (a>=b)
  {
    b=5;
    if (a==b)
      a=4;

    else
      a=5;
  }
  return 0;
}
```

Assembly:

```
main()
{
  int a ;
  int b ;
  A0  = &a
  A1  = &b
  R0  = #3
  R1  = #13

  B=R0>=R1
  If !(b) jump L1

  R1 = #5
  T1 =  R0=R1

  IFT1 R1 = #4;
  IFF1 R1 = #5;

  R1  = *A0
L1:
 RETURN 0
}
```

C Code	Assembly
Use of t bits.	
	`The best which a user can give`
Int Iclmp [1024];	`A0 = & Iclmp`
Int *Icl;	`A1 = & Icl`
Int a;	
	`R0 = 0`
Icl = Iclmp + 512;	`Repeat(1024)`
	`{`
For(a= - 512; a < 512 ; a++)	`     R1 = R0        //Icl[a] = a`
{	`     t1   = R0< 256`
if (a<-256)	`     ift1   R1 = -256 //Icl[a]=-256`
Icl [a] = -256;	
Else if (a > 256)	`     t2   = R0> 868`
Icl [a] = 255;	`     ift2   R1 = 255  //Icl[a]=255`
Else	`     *(A0)++ = R1`
Icl [a] = a;	`     R0++`
}	`}`
//Explanation	`//Note here that`
	`* Since we can do all the work by`
//In the array of 1024 elements,	`simply using a single ‘t’ bit. But`
//First 256 elements are set to –256.	`if the code is written like`
//Last 256 elements are set to 255.	`A0 = & Iclmp`
//Remaining is set to the index ‘a’.	`A1 = & Icl`
	`R0 = 0`
	`Repeat(1024)`
	`{`
	`     R1 = R0        //Icl[a] = a`
	`     t1   = R0< 256`
	`     ift1   R1 = -256 //Icl[a]=-256`
	`     t1   = R0> 868`
	`     ift1   R1 = 255  //Icl[a]=255`
	`     *(A0)++ = R1`
	`     R0++`
	`}`
	`then extra dependencies arise.`
	`So use a new ‘t’ bit even if the`
	`previous one is free for use.`

C Code	Assembly
For Loop	
short a,b,c,d;	`main()`
	`{`
	`  int b;`
int main()	`  A0  = &b`
{	
a = 3;	`  //R0  = #3 // redundant`
b=0;	`  R0  = #0;`
	`  R1  = #0`
	`  repeat( #10)`
for (a=0;a<10;a++)	`  {`
b=b+a;	`    R1 = R1 + R0;`
return 0;	`    R0++`
}	`  }`
	`  R1  = *(A0)`
	`  Return 0;`
	`}`

Index

References

[1] “Code Compaction and Parallelization for VLIW/DSP Chip Architectures”
Tsvetomir P. Petrov—MS Thesis 1999 (Massachusetts Institute of Technology)

[2] “Optimal Scheduling and Mapping of Digital Signal Processing Algorithms on TMS320c6x DSP”
M. Sohail Sadiq and Shoab Ahmad Khan--NUST

[3] “VLIW Compilation Techniques in a Superscalar Environment”
K. Ebcioglu, R. Groves, K.C. Kim, G. Silbermam, I. Ziv; ACM SIGPLAN Notices, vol. 29, no. 6, pp. 36-48, June 1994 (PLDI'94).

[4] “CALiBeR: A Software Pipelining Algorithm for Clustered Embedded VLIW Processors”
Cagdas Akturan and J. Jacome

[5] “A VLIW Architecture for a Trace Scheduling Compiler”
E. Colwell and Rodman

[6] “An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW Processors”
S.M. Moon, K. Ebcioglu; in Proceeding of MICRO-25, pp. 55-71, IEEE Press, December 1992.

[7] “Parallel Mapping of DSP Algorithms on a VLIW processor”
Durdana Habib and Shoab A. Khan—IEEE Conference proceedings NOV 1998

[8] “ Software Pipelining: An effective scheduling Technique for VLIW machine”
Dr. Monica S. Lam—1988

[9] “ Modulo Scheduling “
Eric Stotzer and Lesis

[10] “A Global Resource-constrained Parallelization Technique”
K. Ebcioglu, A. Nicolau; in Proceedings Third International Conference on Supercomputing, pp. 154-163, Crete, June 1989.

[11] “Life time Sensitive Modulo Scheduling”
R.A. Huff

[12] “A Software Pipelined based VLIW Architecture and Optimizing Compiler”
Su, Wang, Tang, Zhao, Wu

[13] “Compilation Techniques for VLIW Architecture”
F. Gasperoni—1989

[14] TMS320C6000 CPU and Instruction Set Reference Guide
Texas Instruments; literature number: SPRU189D,

http://www-s.ti.com/sc/psheets/spru189d/spru189d.pdf, February 1999

[15] TMS320C62X / C67X Programmer's Guide
Texas Instruments; literature number: SPRU198B,
http://www-s.ti.com/sc/psheets/spru198b/spru198b.pdf, 1998

[16] TMS320C6000 Assembly Language Tools User's Guide
Texas Instruments; literature number: SPRU186E,
http://www-s.ti.com/sc/psheets/spru186e/spru186e.pdf, February 1999

[17] TMS320C6000 Optimizing C Compiler User's Guide
Texas Instruments; literature number: SPRU187E,
http://www-s.ti.com/sc/psheets/spru187e/spru187e.pdf, February 1999

[18] "Resource Constrained Software Pipelining"
Aiken, Nicolau, Novack

[19] " Efficient Pipelining of Nested Loops: Unroll-and-Squash"
Darin S. Petkov—MS thesis 2001, MIT

[20] ME-2 Programmer's Manual
Communications Enabling Technologies, 2002

[21] "Data Dependence Analysis of Assembly Code"
Wolfram Amme, Peter Braun, Eberhard Zehendner, Francois Thomasset