12-1997

Microprogramming a Writeable Control Memory using Very Long Instruction Word (VLIW) Compilation Techniques

Randall S. Whitman

Recommended Citation

Whitman, Randall S., "Microprogramming a Writeable Control Memory using Very Long Instruction Word (VLIW) Compilation Techniques" (1997). *Theses and Dissertations*. 5785.
https://scholar.afit.edu/etd/5785

Microprogramming a Writeable Control Memory

using

Very Long Instruction Word (VLIW) Compilation Techniques

THESIS
Randall S. Whitman
Capt, USAF

AFIT/ENG/GCS/97D-19

19980127 076

DTIC QUALITY INSPECTED 3

Approved for public release; distribution unlimited

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

Microprogramming a Writeable Control Memory

using

Very Long Instruction Word (VLIW) Compilation Techniques

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Randall S. Whitman, B.S.

Capt, USAF

December, 1997
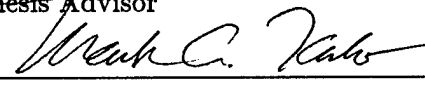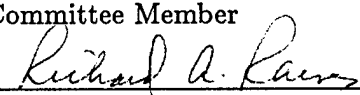
Microprogramming a Writeable Control Memory

using

Very Long Instruction Word (VLIW) Compilation Techniques

Randall S. Whitman, B.S.

Capt, USAF

Approved:

_____   _____
Lt Col David M. Gallagher   25 Nov 97
Thesis Advisor   Date

_____   _____
Maj Mark A. Kanko   25 Nov 97
Committee Member   Date

_____   _____
Maj Richard A. Raines   25 Nov 97
Committee Member   Date

Acknowledgements

I would like to thank my wonderful wife Susan for her loving support and continued hard work during this thesis effort. Her patience, cheerfulness, and love provided the strength and encouragement I needed to pursue this research. My children, Bethany, Randy Jr., Amy, Emily, Elizabeth, and William have been a wonderful blessing during these past 18 months. Their love and never-ending hugs and kisses will never be forgotten.

I would also like to my thesis advisor, Lt Col David Gallagher, for his help and guidance provided from the beginning. He has taught me much in the area of computer architectures and the process of performing difficult research for what appeared to be an overwhelming effort. I would also like to thank my committee members, Major Richard Raines and Maj Mark Kanko, for their assistance in making this thesis a reality.

I would most importantly like to thank the Lord Jesus Christ for the uncountable times He has picked me up from moments of discouragement. His faithfulness to my family and I remains a cherished cornerstone of support. The success of this thesis is surely a mark of His committment to those He loves. He is ever faithful!

<div align="right">Randall S. Whitman</div>

Table of Contents

List of Figures

List of Tables

Abstract

Microprogrammed Digital Signal Processors (DSP) are frequently used as a solution to embedded processor applications. These processors utilize a control memory which permits execution of the processor's instruction set architecture (ISA). The control memory can take the form of a static, read only memory (ROM) or a dynamic, writeable control memory (WCM), or both. Microcoding the WCM permits redefining the processor's ISA and provides speedup due to its instruction level parallelism (ILP) potential. In the past, code generation efforts for microprogrammable processors focused on creating assembly and microcode as two separate steps.

In this thesis, an alternative approach was chosen which combines the separate code generation steps into one automated, dual-target compilation process using the advanced techniques of VLIW compiler technology. The architecture chosen for this effort is a microprogrammable DSP being developed by Rome Labs, New York. The prototype compiler developed in this effort has demonstrated the potential for speedup of microcoded program portions over their assembly code counterparts. Therefore, the feasibility of program speedup produced by a dual-target compiler using VLIW compilation techniques has been validated.

Microprogramming a Writeable Control Memory

using

Very Long Instruction Word (VLIW) Compilation Techniques

I. Introduction

1.1 Background

During the 1970s and most of the 1980s, complex instruction set architectures and microprogramming were the standard programming practices most processors were designed around. During the 1980s, however, individuals like John Hennessy and David Patterson began a revolution of sorts to reduce the complexity of the instruction set into smaller, more easily digestible bites for a computer to handle. The result was the reduced instruction set computer (RISC) and the pipelined processors found in most applications of today. Interestingly though, microprogrammed processors are making their way into the marketplace again in the area of digital signal processors (DSPs). The renewed interest is mainly due to the decrease in the cost and size of memory and their low power usage - ideal for embedded processors like DSPs.

Typically, microcoded processors use a read only memory (ROM) which contains the microcode to execute the instruction set architecture (ISA) of the processor. Some architectures also provide a writeable portion of control memory, the writeable control memory (WCM), which can be modified to adapt to application-specific requirements. Some work has been accomplished in the area of exploiting the WCM recently, but most has been accomplished in the later 1970s and early 1980s. These efforts had focused mainly on generating microcode using dated compiler methods and has left out the assembly code portion as a separate step, and therefore a separate tool in the process. Another point to be made is that the previous efforts did not have the added advantage of the improved compiler technology of today. One of these improvements is in the area of Very Long Instruction Word (VLIW) compilers. This method of compilation provides a means of multiple instruction issue in the same cycle to gain a program speedup. VLIW compila-

tion techniques may therefore be beneficial to a processor which posses the potential of multiple instruction issue due to the format of the microinstruction itself.

1.2 Problem Statement

Rome Labs has created a microprgrammed DSP for which no microcode compiler currently exists. Therefore, the goal of this effort is to generate a dual target compiler, generating both assembly and microcode, for Rome Lab's Wafer Scale Signal Processor (WSSP) which encorporates many of the recent advances achieved in VLIW compiler technology.

1.3 Scope

The scope of this effort will be a code generator which transforms C language programs into their WSSP assembly and microcode functional equivalents. This requires a methodology to handle the two separate architecture targets, assembly code and microcode, in a single pass rather than employing the multiple tools used by previous efforts. Creating a compiler to translate C language programs into WSSP code is a scope requiring much more time than has been alloted for this effort; therefore, a compiler front end had to be chosen to bring the C code into some intermediate language. This effort will translate the intermediate language into the target processor's assembly and microcode.

1.4 Summary of Current Knowledge

Microcoded processors exist on the market today, but the methods companies most commonly use is to still hand microcode the chosen portions of the program [11]. Also, much work related to microcode compilation is designed toward the goal of creating a customized processor architecture which optimizes the generated microcode as in [6]. Therefore most of the research information gathered for this effort resides in the work done in the late 1970s and early 1980s as well as the current WSSP specifications provided by Rome Labs.

The WSSP processor actually had its origins here at AFIT during the late 1980s from

the efforts of Dr Richard Linderman, currently the lead engineer for the WSSP at Rome Labs, Lt Col David Gallagher [4], and Capt John Comtois [2]. The later two were thesis efforts which created the floating point application specific processor (FPASP) incorporated into the WSSP of today. This processor is explained in more detail in Chapter 2.

1.5 Assumptions

A simulator was developed to emulate the WSSP until it has actually been manufactured and tested. The WSSP compiler will be created to meet the WSSP specifications, and the resulting code tested on the simulator. Changes to the WSSP may be made during actual testing of the chips, but may not be encorporated into this effort in time.

1.6 Approach

Since generating a compiler from scratch is not feasible in the time alloted for this effort, a compiler front end was chosen from which the code generator was created. The compiler generates the intermediate code which is converted into both assembly and microcode. The code generator for the assembly code was previously created and so the goal now was to modify the current code generator to generate the microcode in one compilation pass. This was done in three steps to be explained later in the thesis.

1.7 Design Environment

The code generator is written in the C programming language. The resulting assembly and microcode is assembled and linked using the SAS Assembler provided by Rome Labs, and execution of the code occurs on the VSIM simulator located at Rome Labs.

1.8 Organization of Thesis

Chapter 1 has provided the background to this thesis effort and has defined the particular problem to be solved. The scope of the problem and an approach to its solution were also presented.

Chapter 2 addresses efforts which have been made in the past to create microcode

for processors using a WCM, what microcoding entails, and an introduction to the WSSP.

Chapter 3 describes the methodology used in solving the dual compilation problem for generating microcode and assembly code. It speaks to the three phases involved in a code generator to produce the target processor specific code.

Chapter 4 delves into much greater detail in describing the actual implementation modifications made to the WSSP code generator. This is done to benefit the interested reader, but more importantly to provide a foundation of understanding for future efforts and modifications to the code generator itself.

Chapter 5 describes the results and analysis of the two benchmarks used in verifying and validating the WSSP code generator. This chapter will also provide the conclusions from this effort and recommendations for future efforts.

II. Background

2.1 Introduction

Since the goal of this effort is to generate a compiler for a microprogrammed processor, much research went into two specific areas. The first was to understand what a microprogrammed processor is and how it functions. The second was to find out what efforts have been performed beforehand to exploit this type of processor. Therefore, this chapter explains the previous efforts employed in exploiting a WCM. It also explains how this type of processor operates and is microprogrammed. A background in code optimizations and VLIW techniques are also required to explain what role they play in the dual target compiler created. Finally, a brief description of the WSSP digital signal processor by Rome Labs is given.

2.2 Microprogramming

An excellent reference for microprogramming is given by [10]. The following is an explanation of a microprogram control unit and how it is programmed.

2.2.1 Physical Layout. A microprogram control unit is made of two parts: the control memory and the microprogram sequencer. The sequencer is composed of the circuits which handle determining the next address of the control memory. The sequencer contains a register called the control address register which holds the address of the control memory to be executed next. This address is determined from one of four possibilities which are a function of the current microinstruction being executed (except for the initial branch from a standard instruction to WCM execution. In this case, a memory mapping is used where the macro instruction specifies the address in the WCM execution should switch to). The control memory holds the control words, or microinstructions, to be executed. The control words are broken up into fields, each of which specifies some reaction of the processor components. It is determining the format of the microinstruction that is required in order to microprogram a processor. Microinstructions come in two formats, horizontal and vertical.

2.2.1.1 Horizontal Microinstructions.   Horizontal instructions are designed so that each bit controls each microoperation and therefore consists of long control words of 1s and 0s. This allows for controlling a variety of components in parallel but usually not all bits are utilized. Since the words are long, this can be an expensive design choice. For example, a four bit microinstruction may use the first bit to control an add operation, the next bit to control a multiply, and so on. Some encoding can be used as well to group mutually exclusive operations into fields. These fields must be decoded and so involves more hardware than the non-encoded microinstruction. This added hardware also increases the cost and propagation time of the signals. Though it may reduce the size of the memory required, the added hardware may offset this benefit. The WSSP utilizes this type of microinstruction.

2.2.1.2 Vertical Microinstructions.   Vertical instructions require decoding circuits external to the control memory and involve one or two levels of decoding. "Direct" decoding uses one level of decoding performed directly on the instruction. This means each field has its own decoder to control what ever hardware it's assigned. "Indirect" decoding uses two levels where the bits in one field determine the meaning of bits in another field. If the microinstruction controls a bus organized CPU (components are controlled by signals on the buses), this is classified as a vertical microinstruction with the decoders for the fields placed in the bus system itself.

2.2.2 Control Memory.   The control memory can be a read only memory (ROM) or a writeable control memory/store (WCM/WCS) or both. Typically a ROM is used, but occasionally a WCM is utilized. Since the ROM is static, it is very limited in its use in that it is microprogrammed once and remains static throughout execution of the processor. The WCM/WCS may be written to during execution and so can change throughout program execution to provide dynamic control of the hardware. This capability of the WCM provides an opportunity to be exploited to gain significant speedup in a much more flexible way than the ROM.

2.3 Exploiting the WCM: Past Efforts

2.3.1 Methods of Loading the WCM. Abd-Alla and Karlgaard discuss an optimization technique which uses trace information of program execution to optimize a microprogrammed processor's ISA [1]. This effort involved heuristic tuning in five phases to create the final microcode for the WCM. These phases were environment partitioning, execution tracing, architecture synthesis, microcode verification, and system translator communications. Their focus was an algorithm for the architecture synthesis phase which through successive iterations of program code on a general purpose processor, would eventually develop the optimal microcode architecture solution. Final results showed improvements of 7.9 and 4.4 times that of a general architecture for their two sample Fortran programs. Their final results were impressive, but too much overhead is incurred in attempting to generate the trace file through multiple iterations, and this is done on a separate processor. Liu and Mowle discuss four methods of generating microcode to exploit a WCM [9]. All four techniques utilize the inner most loops of a given program to be transformed into the microcode for the WCM. Each technique is described below.

2.3.1.1 Static Loading. The first technique, Static Loading of the inner loops, simply microcodes the inner loops in the program and loads them into the WCM (prior to execution) until no space is left. When the WCM is small, which it usually is, this technique is not very efficient.

2.3.1.2 Selective Loading. The second technique, Selective Loading, microcodes those inner loops with the best chance of gained speedup and loads them in the WCM first. The other loops follow until again the WCM is full. Their method of determining the "best chance of speedup" was to assign priorities to each loop based on the depth of the loop within all its outer loops. Therefore those loops most deeply nested are given the highest level of perceived gain and microcoded first.

2.3.1.3 Dynamic Overlaying. The third technique, Dynamic Overlaying, requires two steps. Step one is to identify all inner loops and convert them to microcode. Step two involves loading the microcoded inner loop into the WCM during runtime as

needed. This involves overhead and a bookkeeping mechanism which will be discussed in the next chapter.

2.3.1.4 User Aided.

The final method allows the user to make use of any execution knowledge he or she may have in choosing which loop will be microcoded and how (static or dynamic overlay) it will be loaded into the WCM.

2.3.2 WCM Code Compilation.

Rauscher and Agrawala demonstrated that compilers can generate microprograms automatically which improve the performance of a program on a given processor using a WCM (with the main memory holding the program portions that could not be translated into the WCM). Their work correlates very strongly with this effort since they used a compiler that would translate the high level language program code into an intermediate language, choose the program blocks which would produce the best gain in performance, and then microprogram those for use in the WCM [13]. What they did not have at their disposal were the advanced techniques available today. Also, their paper doesn't discuss code optimizations for increased performance due to improved code scheduling and instruction level parallelism. These code optimization techniques didn't emerge for a few years after the above efforts.

2.4 Code Optimizations

There are actually a multitude of code optimizations in existence today, but only two schemes - compaction and trace scheduling - will be focused on.

2.4.1 Compaction.

The desire of compaction is to reduce the overall number of cycles required for a given program or code sequence. Compaction achieves this by analyzing the instructions within a local block. A local block is a sequence of instructions where entrance into the block occurs only at the top instruction and exiting the block occurs at the end in a jump or return statement or possibly a merge into another block. To compact this code, dependency graphs are generated in order to determine which cycle each instruction may be executed while maintaining the functionality of the code. Instructions having no dependencies with each other and which don't require the same processor

resources may be assigned to the same cycle. Once this is accomplished, instructions may now be moved from blocks to other blocks to reduce overall cycle time. In general, cycle time is reduced, but the result is not optimal since the field of view is too narrow, looking at local blocks one at a time.

2.4.2 Trace Scheduling. Joseph Fisher proposed trace scheduling as a solution to this problem of nearsightedness [3]. In trace scheduling, the compiler determines the most frequently executed path through the code. This can be done a number of ways (like profiling). This trace is sequentially laid out by the compiler in what is termed a superblock. This superblock is then compacted. This whole process generates new blocks within the superblock. The optimization is good, however, it produces branches and/or jumps into or out of the trace. This may result in code not being executed due to a branch or jump. This problem must be handled through complex book keeping. The solution chosen is known as "tail duplication" where the necessary code is simply duplicated and placed at the end of the applicable block corresponding to the branch. The result is a savings in total cycles for the given code, over and above that of compaction. In Fisher's work, he provides an example where the code originally required 13 cycles to execute. Simple local block compaction reduced this to 11 cycles. However, trace scheduling brought it down to seven cycles. One drawback to compaction and trace scheduling is the potential increase in code size of the program. While this may not be a problem today due to the size of most memories, the problem can actually grow exponentially. Is this a problem when you're microprogramming for a limited-size WCM? To test this requires a space analysis which is discussed in the next chapter. In general, however, this shouldn't be an issue since inner loops, which are typically very small, are the primary pieces of code microcoded for the WCM.

2.5 VLIW Techniques

Modern processors attempt to issue multiple instructions each cycle. There are two approaches to accomplishing this - dynamic scheduling and VLIW. A machine implementing dynamic scheduling, which uses hardware to schedule instructions during runtime, can

issue multiple instructions in one clock cycle - but does so with a limited window of view. In VLIW architectures, the compiler generates the instruction scheduling prior to runtime. In doing so, the compiler ensures that there are no dependencies between instructions that issue at the same time and that there are sufficient hardware resources to execute them. This simplifies the instruction decoding and issue logic required in a machine using dynamic scheduling [12]. The benefit of a VLIW compiler is that it possesses full view of the source code as it generates each instruction word and has the number and type of all functional units of the targeted processor at its disposal. The parallelism needed for issuing multiple instructions is achieved through loop unrolling and scheduling code across basic blocks using a global scheduling technique as mentioned previously [7]. As the compiler processes the source code, it uses this global code access and processor architecture data to create a control flow graph of the code. Each node of the graph represents an instruction and carries with it the information of what resource it requires and what clock cycle it is free to issue. Based on the format of the VLIW (the fields of which specify the processor resources), the compiler groups instructions together that may issue in the same clock cycle. In this way, how the instructions flow through the machine is fully determined by the compiler [5]. Though VLIW techniques have these desirable traits, they are not without their problems. Cache misses can severely disrupt the compiler's plans when all following instructions must be stalled while the data is retrieved. This is due to the requirement to maintain the flow generated by the compiler for all instructions based on which clock cycle they issue and their latencies. Branches, 20-30% of code, create a similar problem as well. Fortunately much of this can be alleviated through trace scheduling and loop unrolling [5]. Another drawback to VLIW techniques is the nonportability of code. This is due to the processor-specific nature of the generated code. VLIW code must be recompiled before moving to another architecture.

Microinstruction and VLIW Comparison Results. Regardless of the drawbacks, it's obvious that microinstructions and VLIW compilation techniques possess similarities which could be combined to create processor-specific efficient microcode. And with this potential, all that is needed is a processor and the corresponding compiler to

demonstrate whether the concept is valid. The compiler chosen for this effort is the Impact compiler created by the University of Illinois. This will be examined in detail in Chapter 3.

2.6 The Wafer Scale Signal Processor

The WSSP is a programmable digital signal processor optimized for power efficient floating-point computations [8]. It can compute eight IEEE 32-bit or four IEEE 64-bit floating point operations per clock cycle and is designed to interface with PCI buses. Each WSSP chip is comprised of two identical processors, A and B, and a shared I/O interface. Each processor is made up of two 32-bit data paths (upper and lower), a control unit (ROM and WCM), and a floating point unit for addition and multiplication. The chip architecture can be broken into three sections: the integer datapath and control core, the floating point unit, and the external I/O bus interface.

2.6.1 Integer Datapaths and Control Core.
Each of the A and B processors contains dual 32-bit integer datapaths and control and memory interface circuits. Each integer datapath has two operand buses (A, B Buses) and one destination bus (C Bus) which connect the registers to an integer ALU to perform standard arithmetic and logical operations. Between the upper and lower datapaths, 44 general purpose registers are available as well as some special purpose capabilities like a barrel shifter. The WSSP uses external SRAM with no internal cache. This removes the cache miss problems associated with VLIW techniques.

2.6.2 Floating Point Units.
The floating point unit is composed of an adder and a multiplier capable of two IEEE 754-1985 single-precision operations or one IEEE 754-1985 double-precision operation per cycle. Both units are pipelined two levels deep to provide increased throughput.

2.7 Conclusion

Since such advanced compilation techniques are now at our disposal, we can apply them to this reemerging technology to improve its performance with a substantial speedup.

The following chapter describes the appreciable gains between execution in and out of the WCM as well as the specific issues that arise and must be handled because of it. It also spells out the changes made to the Impact compiler to generate microcode.

III. Methodology

3.1 Introduction

The approach explained in this thesis to exploit the WCM is to create a code generator for the WSSP which targets both standard machine instructions and microinstructions. The WSSP does not currently have a dual assembly and microinstruction code generator, and so a new tool must be created. Due to the smaller size of the WCM, and the usually smaller, yet repetitious, nature of loops in program code, this code generator will microcode the inner loops. The code generator will be based upon the Impact Compiler developed by the University of Illinois which compiles C programming language code. Once the code is generated, a VHDL simulator provided by Rome Labs will be used to determine the performance of the code. The following paragraphs will explain how micropgrogrammable loop identification can and will be done, methods of loading the WCM and the method chosen, a brief description of the VHDL simulator, and finally the modifications which need to be made to the Impact Compiler to generate both assembly instructions and microinstructions.

3.2 Programmable Loops

A few techniques exist to identify the inner loop. Abd-Alla and Karlgaard used post-execution analysis of a trace file created during execution by saving to file information about references to main memory. This trace file documents the frequency that each instruction is executed. From this file, program loops are identified and marked [1]. Another approach is using graph theory to determine the program loops. This approach provides a view of the structure of the code for loops to be identified.

Shin and Malek proposed a systematic approach which also considers the functional aspect of looping [14]. First they define a microprogrammable loop as one that has a single entry and contains no calls to a procedure or function which cannot be expanded in line (one exit). Single entry can be determined using graph representation and analysis. They provide a method of determining whether a function called within the loop is expandable and still allow the loop to fit within the size constraints of the WCM. Due to limited time

3-1

in completing this effort, I will manually determine if a loop is microprogrammable or not, tag that loop as such for later processing, and leave future efforts to focus on changes to Impact to choose what code should be microcoded.

3.3 Microcode Execution Time Requirements

Determining which loops should be placed in the WCM is essentially a time and space calculation. The actual number of cycles to execute the code in assembly language must be compared to the number of cycles required in microcode. Calculating the cycles for the assembly code is a simple matter given the cycles required for each instruction. However, it is not this simple when determining the execution time in microcode. Since this effort attempts to combine assembly instuctions into one microinstruction, accurate estimation of execution time cannot easily be done without fully generating the microcode. However, assuming about 80% of the assembly code can be combined, a reasonable estimate can be made on the merit of microprogramming the loop based on time savings. If this were the only issue to be considered for time, all loops should be microcoded provided they fit into the WCM. This is not the case, however, since there exists an overhead simply by attempting to use the WCM.

The overhead in execution time is generated for several reasons. When an assembly basic block (inner loop) is transformed into microcode, a few things must happen. One is that a mapping must be created pointing an assembly instruction to the microcode to be executed when called. The second is the actual microcode must be loaded into the WCM before it's executed. Both of these efforts are above and beyond what is required had the code simply been executed in assembly language. The key is that the loop must iterate enough times to overcome the cost of the overhead. The mapping effort costs 6 cycles and the loading of the microcode into the WCM costs 2*N + 6 cycles, where N is the number of microinstructions which can range from 1 to 64. Fifteen additional cycles are required due to four load instructions (3 cycles each) associated with loading and mapping the microcode instructions and three cycles for the prefetching required going into and returning from the WCM. Another reason for overhead is that labels, or rather

the addresses they represent, are not available to the microcode compiler. These addresses are generated by the assembler right before run time and so are not available to the compiler. Therefore ALL labels appearing in a microcode portion of code must be moved into registers in assembly code prior to the microcode. This costs 2*L cycles, where L is the number of labels. Therefore the range of the overhead for this processor is 6 + 2*1 + 6 + 2*L + 15 = 27 + 2*L cycles to a max of 6 + 2*64 + 6 + 2*L + 15 = 155 + 2*L cycles.

The following sections present an analysis of the different instruction types used by the WSSP.

3.3.1 Integer ALU and Non-Pipelined Instructions. Integer ALU instructions executed in assembly language require two cycles. This includes the cycle to prefetch the next assembly instruction before it's executed. If multiple ALU and non-pipelined instructions are sequentially executed in microcode, this prefetch overhead occurs only once since for every jump into microcode the next assembly instruction must only be fetched before returning to assembly code. Hence, the more of these type instructions that will fit into the microcode, the better you do on eliminating the additional prefetch cycles. Another point to make is that combining, or issuing on the same cycle, can occur as well when executing from microcode - so long as there are no resource conflicts between the operations. Due to the WSSP design, which can execute two integer instructions simultaneously, some combining can be accomplished of non-conflicting instructions. Thus two assembly instructions requiring two cycles each can be reduced into one microinstruction requiring only one cycle - achieving a best-case speedup of four. Consider the performance of five sequential ALU instructions, Table 3.1 shows the cycles required for each type of execution.

Table 3.1   Integer ALU Cycle Times

	No. of Cycles
Assembly	10
Microcode (no combining)	5
Microcode (80% combining)	3

The difference in execution between assembly and microcode with no combining is the simple difference in the number of cycles required for execution - two cycles for assembly instructions and one cycle for microcode instructions. The prefetch in microcode is combined within the five microinstructions and so occurs only once in the entire sequence. The 80% combined is the result of just three microinstructions requiring execution since four of the assembly instructions were combined into two microinstructions. Tables 3.2, 3.3, and 3.4 depict the cycles required for each method where ex stands for execution.

Table 3.2   Cycles ALU for Assembly Instructions

Instruction	1	2	3	4	5	6	7	8	9	10
1	ex	ex								
2			ex	ex						
3					ex	ex				
4							ex	ex		
5									ex	ex

Table 3.3   Cycles for ALU Microcode without Combining

Instruction	1	2	3	4	5
1	ex				
2		ex			
3			ex		
4				ex	
5					ex

Table 3.4   Cycles for ALU Microcode with 80% Combining

Instruction	1	2	3
1 & 2	ex		
3 & 4		ex	
5			ex

It can be seen that simply executing in microcode can potentially produce a significant speedup for that section of code. Obviously, any combining less than 80% will still produce a speedup.

3.3.2 Pipelined Instructions.

Floating Point ALU Instructions.   Floating point (FP) ALU operations on the WSSP are pipelined instructions. Therefore, if no data dependencies exist between two instructions, one can be started in the current cycle and the following instruction started in the next cycle. Operations performed on the FP ALU are FP adds and FP subtracts. Since integer mulitiplies, FP multiplies and FP divides have different resource requirements than FP ALU instructions, they will be discussed separately.

FP operations require three cycles to execute both in assembly language and microcode. Unlike the integer ALU operations, the instruction prefetch is merged into the three cycles and so cannot be removed simply by executing the instruction in microcode. Two FP single precision operations can be executed simultaneously if there are no data dependencies and no resource conflicts. They will be pipelined if zero data dependencies exist but resource conflicts do exist. Table 3.5 shows the cycles required by executing five FP ALU instructions in assembly code. It also represents the cycles for execution in microcode when 100% data dependence exists between all instructions. With this much dependence between instructions, there is no benefit from using the WCM over simple execution in assembly code. However, this is not usually the case, and so a significant speedup can be obtained simply by converting the instructions into microcode as can be seen in Tables 3.6 and 3.7. These tables show that speedup is improved by 36.4% with no combining and no data dependencies, and slightly more than doubled ($15/7 = 2.14$) when 80% combining is possible. The reason for the cycle skips is resource bus contention. The FP ALU functional unit takes inputs from registers on two busses - one source comes from either the upper or lower B bus, and the other source comes from either the upper or lower C bus. Since the result is written on one of the C busses, a conflict would arise between the result of the completing instruction and the sources of a new instruction starting on the same cycle. (Other source paths are possible to remove this restriction but a patent pending on the FP ALU prevents specifics.)

Floating Point Multiply Instructions.   The FP multiply instructions are essentially identical to the FP ALU instructions, except that there is no resource

Table 3.5   Cycles for FP ALU Assembly or Microcode with 100% data dependence

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	ex	ex	w												
2				ex	ex	w									
3							ex	ex	w						
4										ex	ex	w			
5													ex	ex	w

Table 3.6   Cycles for FP ALU Microcode - No Combining

Instruction	1	2	3	4	5	6	7	8	9	10	11
1	ex	ex	w								
2		ex	ex	w							
3					ex	ex	w				
4						ex	ex	w			
5									ex	ex	w

contention on the write back cycle. Therefore, these instructions can be truly pipelined from start to finish. The cycles required for assembly code and microcode with 100% dependencies is the same as the FP ALU instructions shown in Table 3.5. Tables 3.8 and 3.9 show the cycles required for execution in microcode with no combining and microcode with 80% combining respectively. The "w" represents the "write back" portion of the executing instruction.

Floating point multiplies produce a greater speedup than FP arithmetic operations since the resource conflict on the write back cycle doesn't exist. As long as there are zero data dependencies, pipelining alone more than halves the execution time. Achieving

Table 3.7   Cycles for FP ALU Microcode with 80% Combining

Instruction	1	2	3	4	5	6	7
1 & 2	ex	ex	w				
3 & 4		ex	ex	w			
5					ex	ex	w

Table 3.8   Cycles for FP MUL Microcode without Combining

Instruction	1	2	3	4	5	6	7
1	ex	ex	w				
2		ex	ex	w			
3			ex	ex	w		
4				ex	ex	w	
5					ex	ex	w

Table 3.9   Cycles for FP MUL Microcode with 80% Combining

Instruction	1	2	3	4	5
1 & 2	ex	ex	w		
3 & 4		ex	ex	w	
5			ex	ex	w

80% combining produces a speedup of three. As a note, integer multiplies, INT MULT, are performed using the FP multiplier, require three cycles to execute, and have one restriction - no combining may occur since only one 32 bit integer multiply operation can be started at a time.

3.3.3 Branches.   Branches in the WSSP are converted to (INT or FP) ALU compare and branch instructions. Both instructions can be pipelined with other instructions and are potentially combinable, providing an opportunity for speedup. The cycles for the compares were discussed in the INT ALU and FP ALU sections. The branch instruction itself requires two to four cycles. Four cycles are required for "less than or equal to" and "greater than or equal to" conditional branches. Performing branches in microcode generates some interesting problems.

Branching in Microcode.   Since branching normally requires the use of labels, branching in microcode isn't possible with a simple branch instruction (recall the discussion of labels for loads and stores and the need to move these labels into registers in assembly code prior to the jump to microcode) unless the target is also in microcode. What must be done is to insert jumps to branch handling portions of microcode which vector out

of microcode to a branch handler in assembly which jumps to the required assembly code destination branch label. When execution is complete in the destination branch, a return is performed to the calling assembly branch handler which in turn makes another jump to the microinstruction following the original branch call in microcode. Figure 3.1 provides a picture of actual code showing how this can be done. For example, code begins executing at label CB_1. The microinstructions on the right side of the figure are mapped to a location in the WCM (LOADMAP) and then loaded into that memory location (LOADRAM). The jump to microcode is made by executing JMP UCODE_1. When the first branch in microcode is encountered and taken, which originally was a branch to CB_5, the branch to the microcode branch handler (U_CB_5) occurs. The address of the PRE_CB_5 is loaded into the PC and the two microexit microinstructions are executed. The jump is made to the assembly code in PRE_CB_5 which simply jumps to CB_5. Once CB_5 is complete, the return to PRE_CB_5 is made. Finally, with the assembly portion complete, all that remains is a jump back to microcode - UCODE_2.

All this jumping obviously generates some overhead - about eight cycles total. But instances will arise when it becomes necessary. A specific example for a microcode branch would be integer divides or mod operations. The WSSP ISA doesn't have instructions of this type and so branches must be made to predefined assembly library units. Rather than preclude microcoding because an integer divide or mod instruction appears in the code, it can be handled the above way provided the timing and space analysis determines it to be worthwhile.

3.3.4 Loads and Stores. These instructions require from 3 cycles up to 12 cycles to execute both in assembly and microcode. Byte-size loads and stores are the most costly. They also require the use of the same ALU unit as the INT ALU instructions and so functional unit conflicts arise frequently. Speedup comes from the fact that pipelining of loads and stores with each other and other instruction types is possible in microcode (combining is not possible due to resource conflicts). The cycles required for these instructions are identical to the FP multiplies, though no combining can occur.
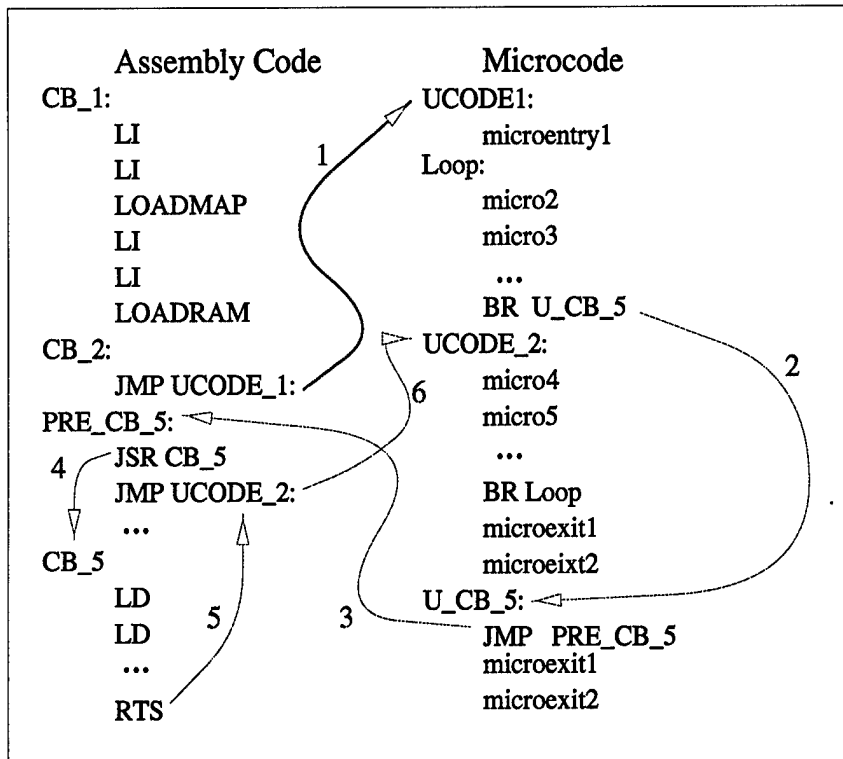
Figure 3.1   Handling Branches in Microcode

3.4 Instruction Mix Summary

Obviously code doesn't normally consist of only one type of instruction. And the analysis gets more complex as more instructions come into play. From the above analysis it can be seen that transforming the above instructions into microcode will produce a speedup over assembly language alone provided the overhead is overcome. The less data dependence and the more combining that exists, the more speedup that will occur. Rather than exhaust all the possibilities of instruction mixes, consider the following example.

Instruction Mix Example. Consider a code stream consisting of a mix of nine instructions of the types INT ALU, INT MULT, FP ALU, FP MULT, and BR (which becomes a compare/INT ALU operation and branch). For simplicity, assume there are no data dependencies between any instructions except the compare and branch. Table 3.10 shows the frequency of each instruction and the corresponding cycles in assembly and microcode.

Table 3.10   Example Instruction Mix

Instruction	Frequency	Assembly Cycles	Microcode Cycles
INT ALU	2	2	1
INT MUL	1	3	3
FP ALU	2	3	3
FP MUL	2	3	3
INT CMP	1	2	1
INT BR	1	2	2

Cycles in Assembly. In assembly code, no combining or pipelining is possible. Each instruction executes its necessary number of cycles and then the following instruction executes until the instruction stream is exhausted. Therefore the total number of cycles in assembly code would be

$$Assembly\ Cycles = I * \sum(F_i * C_i)$$

where I is the number of iterations, F_i is the frequency of the ith instruction, and C_i is the cycles required for the ith instruction. This code would then require $I * 23$ cycles to execute.

Cycles in Microcode. The ALU (including CMP) and FP instructions can potentially be combined, no more than two into one, and the BR instruction can probably be absorbed into the last instructions (due to scheduling) and so not incur an additional two cycles. Since it can't be said that all of these instructions can be combined due to potential resource conflicts, assume 80% can. Therefore seven instructions of the nine possible can be combined. Note that different times result depending on how the instructions are combined. A multitude of potential combinations exist for combining this code depending on resources needed, and many of them will result in the minimum number of cycles required. The scheduler will chose one of the minimum cycle count solutions. Table 3.11 illustrates how each instruction is combined and the cycles they're executed in.

Table 3.11   Cycles Required for Microinstructions

Instruction	1	2	3	4	5	6
2 INT ALUs	ex					
INT MUL & CMP		ex	ex	w		
2 FP ALUs			ex	ex	w	
2 FP MULs & BR				ex	ex	w

Microinstructions dictate the sequence of events in the processor for *each cycle* of execution. The columns of Table 3.11 provide a picture of this – not the rows as might be expected. Therefore, six microinstructions would be needed to execute the code and so the total cycles required for this arrangement would be $I*6$. The overhead cost, C, is $6 + 2*6 + 6 + 15 = 39$ cycles. Therefore the total cycles for one iteration and the overhead is $(39 + 6)$ 45 cycles. The number of iterations required to overcome the overhead costs can thus be determined by solving for the break-even point between assembly code and microcode – anything above this is a speedup. Let A be the number of Assembly cycles, I be the number of iterations, O the overhead, and M be the number of Microcode cycles.

Then

$$A * I \;=\; O + M * I$$
$$I \;=\; O \div (A - M)$$
$$I \;=\; 39 \div (23 - 6) = 2.29$$

Therefore, at least 3 iterations of the loop must occur in order to recoup the cycles lost in overhead.

Microcode Space Requirements. The space requirements must be met for microcoding the WCM as well. The WCM is capable of containing 64 microinstructions. Since each cycle required represents one microinstruction, the example above of six cycles would fit into the WCM and so pass the space test.

Example Summary. This type of comparison would need to be done for each loop. Due to the time constraints for this effort, automatically determining which loops are microprogrammable will be a follow on effort. It is not a simple issue since the microcode must be generated first by the compiler/scheduler to determine what pipelining and combining will occur. Based on this, the analysis to use microcode or not can be carried out. For this thesis, each loop will be manually evaluated to determine if a speedup is obtained through microcoding and if the loop will fit within the WCM. Since the percentage of combining cannot be determined prior to compilation, 80% combining will be assumed.

3.5 Methods to Load the WCM

Methods to load the WCM were discussed in the previous chapter and focus on the efforts of Liu and Mowle. The method chosen for this thesis is the user-aided with dynamic overlay. Each microcoded loop will be loaded into the WCM as needed during runtime. Any previous loops, including the same loop, will be overwritten. Time constraints for this effort are the only reasons this method was chosen since it is unnecessary to overwrite a loop which already exists in the WCM. One way of preventing this overwrite is a trap that

gets called if the opcode for the loop exists in the microcode map to prevent loading the data again. Other methods exist, and should be the focus of a follow-on effort.

3.6 The VHDL Simulator

The VHDL simulator to be used in this effort was developed by a contractor for Rome Labs. It exists only at Rome Labs and only two copies are available. It's designed to handle simulating not only the assembly code but the microcode as well. This is the simulator to be used to verify the correct operation of the WSSP code generator. A few benchmarks, each heavy in floating point execution since this is the primary purpose of the WSSP signal processor, will be compiled into assembly code and executed gathering the number of cycles executed. It will then be compared with the performance of its microcoded counterpart. The speed of the simulator limited what benchmarks were used.

3.7 The Impact Compiler

The Impact compiler is a VLIW compiler and so can optimize and schedule multiple instructions per cycle. The scheduling portion of the compiler operates on an intermediate instruction language known as Lcode to provide a unified interface to the code optimizers and code generators. Lcode stands for low-level intermediate code. In order to generate the assembly and microcode instructions required by the WSSP, the current assembly code generator must be modified.

3.7.1 Impact Code Generators. As a background, code generators for impact are comprised of three phases to bring the basic Lcode intermediate form of a program into the machine specific instruction format required by the target processor. This flow from Lcode to machine assembly code can be seen in Figure 3.2.

Impact generates optimized Lcode from a program that was originally written in the C programming language and places it in a file, <filename>.O. Phase 1 takes this file and begins the process of moving the Lcode into machine specific code. In this particular phase, instructions are created which map one-to-one to instructions the target processor
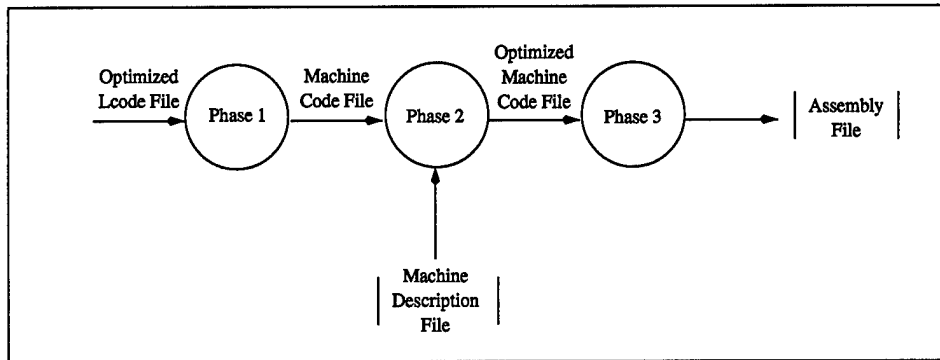
Figure 3.2   Code Generation Instruction Flow

can execute. As an example, Phase 1 would take the following Lcode instruction

  sub_f R1, R2, 1.0

and change it into two instructions of the form

  mov_f R3, 1.0

  sub_f R1, R2, R3

since the WSSP doesn't have a floating point subtract with immediate instruction. The WSSP also has an instruction that changes its mode of operation from integer, to single precision floating point, to double precision floating point. This is due to the use of the FP ALU and FP Multiplier (which is also used for integer multiplies). This instruction, SETFP, is inserted as needed during this phase. <filename>.mc is the output file of Phase 1.

Phase 2 then takes the output of Phase 1 and performs four operations on the code: instruction scheduling, register allocation, optimization, and changes offset references from four areas in the stack (incoming parameters, outgoing parameters, local variables, and register spill space) to offsets off the stack pointer (SP). Phase 2 sets up the stack for each function in the program as well. The output of this phase is an optimized machine code file <filename>.mco which becomes the input file for the next and last phase of the code generator.

Phase 3 is the simplest of the three phases and makes no changes to the code other than simple translation from intermediate form to assembly code. By this time, the input stream is in the format to simply print out to a file, <filename>.s, as assembly instructions.

All that is left is to assemble and link the code and execute it on the target processor, or in this case, on the simulator.

3.8 Dual Target Code Generator for Microcoding

Generating assembly code for the WSSP with Impact has already been accomplished using the above format as the template. However, the WSSP requires microcode in order to exploit the WCM - which in essence is generating code for two target architectures simultaneously. A method needed to be devised to generate the microcode as well as the assembly code in a single pass compilation step. The same three phases are used, but modifications needed to be made to each. Before discussing these modifications, recall that the marking of the microcode portions of code (control blocks) will be performed prior to Phase 1, and so <filename>.O, which is the input file to this phase, will already have these control blocks tagged.

3.8.1 Phase 1 Modifications.
Phase 1 will still perform the same purpose, which is to create a one-to-one mapping of Lcode instructions to match the target processor's ISA. Due to microcoding, new issues emerge that must be dealt with. For example, when creating assembly code, labels and their absolute addresses aren't a concern since the addresses are generated by the assembler when machine binary code is created just before runtime. This poses a problem when trying to move this code into microcode and the absolute address isn't available yet. In order to combat this and prevent labels from making it to Phase 3 where the microcode will be generated, each label must be moved once into a register in assembly code so that these addresses are available to the microcode. This is only done when microcoding is being accomplished and does add to the overhead cost (three cycles for each label) for moving assembly code into microcode.

A potential optimization would be to do address calculations for loads and stores outside of microcode which would be a one-time two cycle overhead. Loads and stores would then only be a two cycle operation (rather than the current three cycle operation) in microcode saving one cycle per iteration. If a new address calculation is needed while in microcode, however, this optimization would be nullified.

No other changes are necessary for Phase 1. However, it is anticipated that follow-on work will likely need to make further changes to aid optimization.

3.8.2 Phase 2 Modifications.   Phase 2 requires sweeping changes, or actually additions, in order to handle the dual compilation requirement. Since both types of code are to be created, each "target code" requires its own machine description file. This is true because the latencies, resource usages, and in some cases even the instruction format of assembly instructions change when creating a microinstruction from the corresponding assembly instruction. Defining every instruction's characteristics to ensure proper scheduling and optimization is a major undertaking. The target processor architecture must be fully understood in order to correctly create the mapping of latencies, resources, and formats per instruction. In addition to this, since the code generator is set up to go through one function at a time, and a function may contain portions targeted for assembly and portions to microcode, two passes must be made in order to handle both instances. The first pass schedules and optimizes those instructions destined for assembly code using the corresponding machine description file, while the second pass does the same for instructions to be microcoded using its machine description file. The truly amazing point behind this is that with the machine description files, the scheduler and optimizer will create extremely efficient, conflict free machine code. When programs get very large, this is a tremendous benefit. The scheduler then creates a table of these resources and determines which instructions can be issued simultaneously to gain the greatest speedup. For an individual to perform this task would be time consuming on a staggering scale. However, with the computer, this is a simple task.

In this particular instance, using the Lcode instructions for resource mapping wasn't sufficient, since resource mapping for microcode must get down to busses, functional units, and mutually exclusive fields required for each cycle. This being the case, a new opcode must be created for each corresponding Lcode instruction which takes into consideration whether the registers assigned are upper or lower registers and whether instructions using immediates require 16 bits or 32 bits to represent the immediate field (32 bit immediates require an extra cycle to load the upper 16 bits first followed by the lower 16 bits). Due to this level of detail, each Lcode instruction explodes into two to eight possible instruction

formats, depending on the mix of upper and lower registers in the instruction assigned during register allocation. The register allocator could be improved in this area by performing a dependence chain of code and assigning registers based on reduced conflict versus the current "minimal register use" algorithm it employs.

Another important point is that there may be a number of different ways to execute any given instruction due to the many paths the data may take. This is a tremendous opportunity to exploit the parallelism not always inherent in instruction flow. Provided the cycle and resources are mapped together correctly in the machine description file, the scheduler will choose the instruction option that will allow the greatest amount of instruction level parallelism (ILP). This allows for a greater chance of speedup. The difficulties associated with this potential will be dealt with in the next chapter, where more detail to these modifications will be explained.

The result of Phase 2 is that each instruction now carries with it the cycle it is to be issued in. This information is essential for Phase 3.

3.8.3 Phase 3 Modifications. Phase 3 still retains the purpose of printing out the instructions in the format the assembler needs to generate binary machine code. However, a simple printing scheme is no longer sufficient. Phase 3 underwent sweeping changes as well due to the complexity of generating microcode (see Figure 3.3).

Initially, a microinstruction output file is created and initialized. It's given the program name with "_micro.s" appended to it (Create Output File process). All portions of microcode are printed to this file. Once the output file initialization is accomplished, "entry" and "exit" instructions are inserted into the microinstruction stream (Create Entry and Create Exit Instructions processes). These instructions perform the prefetching of the next assembly instruction that will be executed upon exit of this loop. The assembly instructions tagged for microcoding are then converted into microinstructions (Create Microinstructions process). Since the last assembly instruction is a branch instruction, care must be taken here so as to ensure the branch is not executed before preceding instructions have completed executing (Handle Branch Instructions process). The resulting microinstructions are printed to the output file (Print Microinstructions process) in a for-
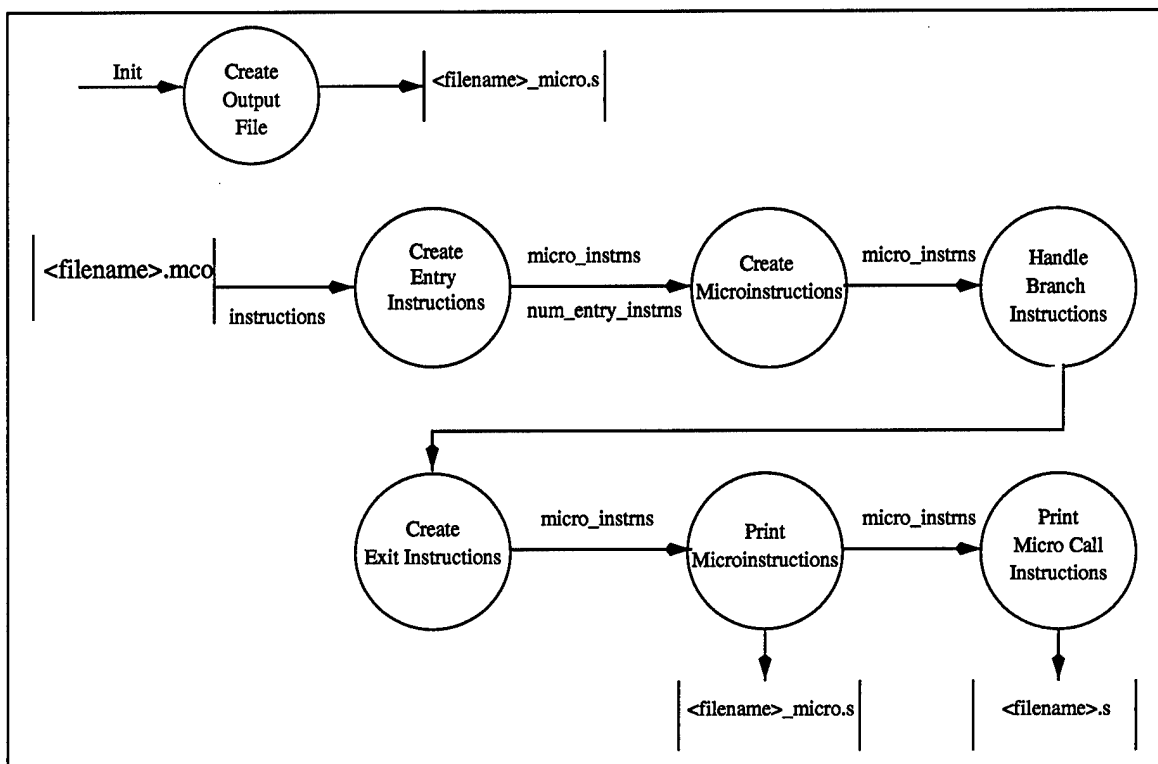
Figure 3.3   Phase 3 Process Overview

mat detailed in the next chapter. Finally, a few assembly instructions are inserted into the assembly code stream to set up the jump to microcode (Print Micro Call Instructions process). These instructions are already a part of the WSSP ISA and consist of the LOADMAP and LOADRAM. The first of these maps a newly defined opcode for the microcode loop to the address in microcode it will be loaded so that when this opcode is fetched in assembly language, the processor knows to jump to the corresponding address in the WCM. The LOADRAM is the instruction which will move the code from memory into the WCM before execution. In order to execute the microcode, an inline assembly instruction is created with the corresponding opcode and inserted in the assembly stream following the LOADRAM and LOADMAP instructions. When program execution reaches this point, a jump occurs to the WCM in order to execute the microcode stored there. Upon completion, execution returns to the assembly code following the jump. This sequence of instructions appears in the assembly code as follows:

```
LOADMAP R1, R2
LOADRAM R2, R1, R3
.int 00000000000000000000000000000000
.int 00000000000000000000000011111110
```

R1 and R2 contain the opcode and WCM address of the microcode respectively. Register R3 contains the number of microinstructions to load into the WCM. The .int statements are the inline assembly instruction stating to execute opcode 127, "111111", which causes the jump to the location in the WCM pointed to by R2.

3.9   Conclusion

This is the method chosen for creating the dual targetted compiler for the WSSP. Although some of the discussion was at a detail level to get a good handle on the task at hand and what will be accomplished, much of the specific details were left out in the implementation of the code for the code generator. The following chapter will remedy this situation for the benefit of the interested reader and any follow-on efforts building upon this foundation.

IV. Impact and the WSSP Code Generator: A Closer Look

4.1 Introduction

Whereas Chapter 2 described the background to using VLIW techniques to generating microcode and Chapter 3 explained the method chosen to solve the problem in some detail, this chapter delves into the specifics of the code generator implementation. It describes the key features and design decisions of the actual code that generates the microcode for the WSSP. As a reminder of the process, Figure 4.1 provides the Code Generator Flow. Only specific areas need to be explained in more depth, however. The first is the format and overall design of each code generation phase. The format of the source code files will be described and why a functional design was chosen. The Phase 1 discussion details the setting of the mode for the processor using an implementation already built into Impact, the creation of preheader control blocks for removing labels from microcode and placing the LOADMAP and LOADRAM instructions outside the loop. The details of the instruction flow in Phase 2 is expanded upon. This includes the separation of assembly code and microcode. A detailed explanation of the heart of Phase 2, the machine description file (MDF) is presented here as well. Finally, an explanation of Phase 3's method of generating microcode and the data structures used caps off the description of the code generator for the WSSP. A hardware description of the WSSP is somewhat limited due to patents pending on portions of the design.
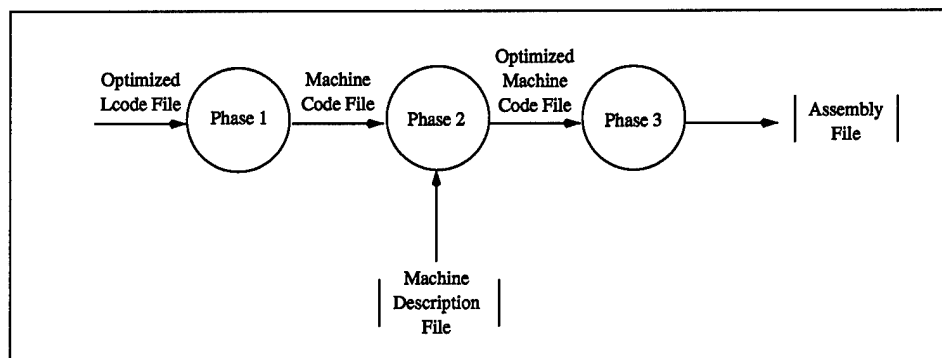

Figure 4.1   Code Generation Instruction Flow

4.2 Overall Design Decisions

The major design decision was to continue use of the existing functional design of the code generators rather than attempt a differing approach. The purpose of the code generator actually lends itself to a functional approach since its primary purpose is to handle incoming data, process it in some fashion, and pass it on to the next step. The end result is a series of functions that permit a continuous flow of data. This is not so much an interaction between objects, as would be the case for an Object Oriented approach. More important reasons for this choice, however, are derived from the purpose of the completed tool. It is planned to be used on the market for the WSSP. As such, there isn't enough time to perform a major re-engineering effort on the code. The desire is to design and code the tool to be complete and correct but to get to market quickly - this meant using the current design and delaying any major re-engineering efforts until a later time as needed. A final point to be made is that since the design method currently used has proven effective to many successful code generators already in existence, no convincing argument could be thought of to cause a choice to redesign the current architecture.

Impact is a very large compiler. It incorporates many features that interact with the code generators and is extremely complex. Therefore a design decision was made to make minimal changes to these interacting portions of Impact - the scheduler in particular. Changing this code could cause ripple effects to other code generators and also may not allow improvements to these units to be easily added for the benefit of all code generators. Nevertheless, some small changes appeared essential and have been implemented for use in Phase 2.

Other design decisions were to insure a sound level of software engineering principles were followed. These mainly consisted of file headers which contain the file's author, creation date, description, and history of changes to provide a greater level of program understandability and thereby easing the burden and shortening the time required for future modifications. It also provides a knowledge of the current version of the file at a glance.

Prototypes of all routines embedded in a file are up front to provide an interface and synopsis of the functions available for use and those local to a particular phase. This

also helps prevent interfacing problems with external routines. If changes are made to an existing method which differ from the prototype, chances are good that the change will impact external routines and careful research should be made before implementing that change.

Comments are added to all routines to clarify their purpose and reasons for particular implementation schemes. For example, in Phase 1 when an Lcode instruction must be changed to map to the corresponding WSSP instruction(s), a comment is added to show all instruction formats.

Errors are handled in Impact by making a call to a routine called Lpunt() and passing the name of the routine the error occurred in and a brief description of the problem. The program then gracefully halts execution rather than crashing, or worse, generating incorrect code. This design remains unchanged.

The structures used by Impact and the code generators are defined in the source file Lcode.h. These structures contain the necessary information for each instruction as well as the control block and function they are contained in. A hierarchy exists that maps a program into functions which contain control blocks which contain instructions which contain operands. These are linked lists that are manipulated as they are processed through Impact and the code generators. Figure 4.2 provides a picture of this.
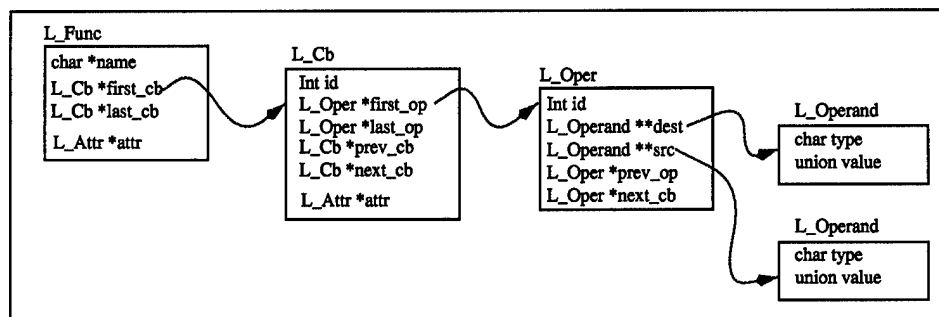

Figure 4.2   Lcode Data Structures

The L_Func is the structure to hold data related to functions. The name field is the function's name like "main" found in most C programs. The L_Attr component is also a linked list of additional information used to further explain a particular attribute of a data structure. The microcode control blocks are marked using this field. L_Cb holds data

related to control blocks. L_Oper holds data to instructions (or operations) and also uses two arrays of destination and source operands. Finally, L_Operand holds the data for the source and destination operands for each instruction. The control blocks and instructions are implemented as doubly linked lists. The id fields represent the number assigned to a particular control block or instruction.

4.3 Overall Code Generator Process

The control of each phase is performed by the main driver, L_gen_code() found in lwssp_main.c. After initialization, including a call to each phase's initialization, the main driver sets all the code generator optimization and process flags. It does this by reading a particular file, STD_PARMS, which contains all of the Impact and code generator flags and their state - either on or off. For example, if microcoding is to be done, this flag would be "on" in the STD_PARMS file and set when the code generator main driver reads it in. This flag is now available to each phase. This method of setting flags was done to keep all options (flags) in one location for quick and easy changes. Since the values are read in during run time, recompiling due to changing a flag isn't necessary - only making the change to the file and saving those changes. Once the flags are set, the main driver then calls each phase's driver to process the instruction stream. This is done one function at a time and one phase at a time.

4.4 Phase 1

This section will explain the overall flow of the instructions through Phase 1 and focus in on a few of the processes that occur. Figure 4.3 depicts pictorially what's happening inside Phase 1. The driver in Phase 1, L_process_func(), takes a function and processes each of its control blocks and their instructions. It does this through the three main processes Init Fpmode, Remove Labels, and Annotate Instructions. These three encompass the primary modifications to Phase 1 for generating WSSP assembly and microcode.

4.4.1 Set Fpmode.
The WSSP has three modes to correctly execute floating point, double, and integer multiply operations. This mode, represented in Phase 1 by
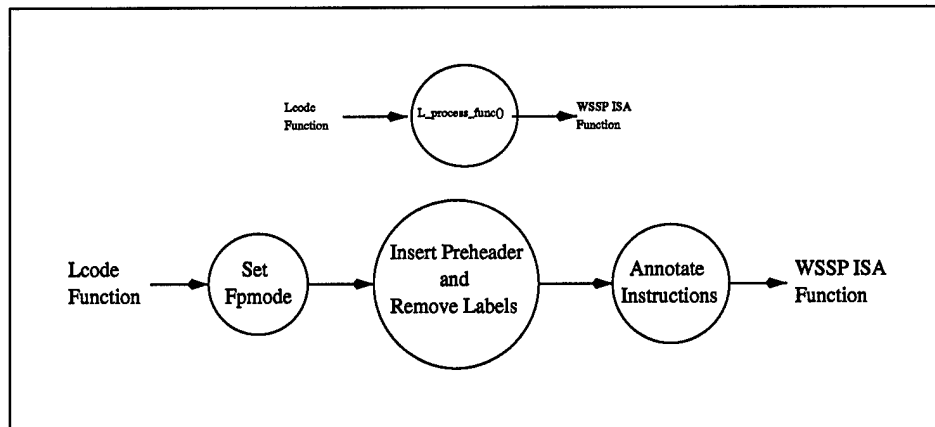
Figure 4.3   Phase 1 Main Processes

fpmode, must be set using the SETFP WSSP instruction prior to executing any of these instruction types. Therefore a separate routine was designed to create and insert this SETFP instruction and to maintain the current mode setting. Options to inserting the SETFP instruction is one of the STD_PARMS flags which, when turned on, would only insert a SETFP instruction when a mode change occured. Otherwise, the routine will insert a SETFP prior to all instructions that require the mode to be set. The mode is reset to the "unknown" state for each control block entry and when jumps are made to subroutines that may have changed the mode. To reduce the unnecessary mode resets, the routine Is_Safe_Fpmode_JSR(function_name) was created to return true if the function function_name does not change the mode. Reducing the insertion of this instruction in the code stream can save many cycles for programs that contain a high number of the same type instructions. A point to add is that a fpmode **destination** operand is created for the SETFP instruction and a fpmode **source** operand is created for each instruction requiring this mode. This is done to inform the scheduler of a data dependence in order to maintain the sequence of execution of these instructions to insure proper execution results.

4.4.2 Insert Preheader and Remove Labels.   This process is activated only if microcoding is to be done. The preheader control block is inserted in the code stream so that the LOADRAM and LOADMAP instructions can be executed outside of a loop. In this way, unecessary remapping and reloading of the WCM is prevented. As was explained

4-5

Instruction Flow Before	Instruction Flow After
cb 4	cb 4
mov R6, 10.0	mov R6, 10.0
cb 5 (microcode cb)	cb 8 (preheader)
load R5, label1,0	LOADMAP
add R5, R5, R6	LOADRAM
store label1,0,R5	cb 7
...	mov R7, label1
branch condition, cb5	cb 5 (microcode cb)
cb 6	load R5, R7, 0
	add R5, R5, R6
	store R7, 0, R5
	...
	branch condition, cb5
	cb 6

Figure 4.4   Preheader and Label Removal Result

in Chapter 3, labels must be removed from code targeted for the WCM. Therefore, the labels are loaded into registers in a new preheader and those registers are then used in the microcode. Figure 4.4 provides an example of the resulting control block after two preheaders have been inserted and the labels removed from a microcode tagged control block. Note that cb 7 and cb 8 are inserted between control blocks cb 4 and cb 5, where cb 5 is the control block tagged for microcoding. The label, label1, is moved into register 7 in the preheader cb 7 and the label sources replaced with register 7 in the load and store instructions in cb 5. One more function does occur however, and that's to change any branches originally to cb 5 to cb 7 to insure the label is moved into register before each jump to microcode.

4.4.3 Annotate Instructions.   This process handles transforming all Lcode instructions into instructions which match one-to-one to WSSP ISA instructions. It consists of another routine, L_annotate_oper(), which vectors all possible instruction types to the appropriate annotation routines. A routine exists for each type of instruction. For example, L_annotate_float_cond_branch() takes an Lcode conditional branch instruction, like blt (branch less than), and converts it into the two corresponding WSSP instructions. The

first instruction is a floating point compare and the second the actual branch based on the condition and the flags set from the compare. The WSSP ISA doesn't possess a single "compare and branch" instruction. So the Lcode instruction blt cb5, R5, R4 becomes two instructions - fps_cmp <flags>, R5, R4 and br lt, cb5, <flags>. The flags operand, as a destination and then source, is inserted to insure the data dependence is known between the compare and branch. Recall that a SETFP instruction could also be inserted before the compare instruction since it is a floating point operation. Therefore, this one Lcode instruction could create up to three instructions in the code stream. This process is repeated for all instructions in the function.

4.4.4 Phase 1 Conclusion.

This was a deeper look at Phase 1, but by no means comprehensive. Only a thorough perusal of the code would provide a complete picture of everything that occurs. However, the main processes of the phase were explained. Each function is passed into the driver which initializes the processor mode and then goes through each microcode control block to add the microcode preheader and remove any labels. Finally, each instruction is annotated to create a one-to-one mapping into the WSSP ISA.

4.5 Phase 2

Phase 2 is the key point at which dual compilation begins. As mentioned in the previous chapter, a definite splitting of assembly code and microcode occurs here. Figure 4.5 gives a picture of the processes and the order in which they occur. I will explain each of these processes as well as the machine description files (MDF) which are an integral part of Phase 2.

4.5.1 Machine Description Files.

The purpose of the machine description files is to provide the scheduler the necessary instruction resource mapping information required for each instruction being processed. In each file, every instruction has its latencies defined, the resources required and the cycle for that resource, what cycle each destination operand is available, and what cycle each source operand is required. For assembly code, this is a
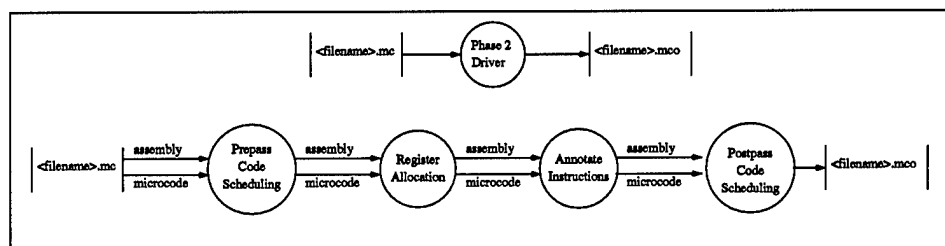
Figure 4.5   Phase 2 Main Processes

fairly straightforward effort to define since each instruction is executed autonomously and therefore no resource contention exists. Since this is the case, specifying the resources is unnecessary.

Resource contention is determined on a cycle-by-cycle basis and so cycles are specified beginning at zero and incremented there on. All instructions begin in cycle zero. For example, executing a floating point add assembly instruction, add_f R1, R2, R3, would have the following machine file specification:

>   Destination (R1): Available in cycle 3
>   Sources (R2, R3): Needed by cycle 0
>   Resource Required: One issue slot (cycle 0)
>   Latency: 3 cycles

In contrast, the MDF description for a microinstruction requires the knowledge of whether the registers are upper or lower to specify resources - this is why register allocation must be performed before microcode annotation. It is also why for every assembly instruction there are multiple possibilities for the corresponding microinstruction. In the add_f instruction example, there are eight possible formats for the operands in microcode since there are three operands and each can be either upper or lower. Which bus is used must be specified in the microcode and the scheduler must know this information to correctly schedule the microcode. Therefore, each possibility must be listed in the MDF and would look like the following:

>   Destination (UR1): Available in cycle 3
>   Sources (UR2, UR3): Needed by cycle 0
>   Resource Required: One issue slot (cycle 0),

4-8

Upper A bus (cycle 0),
Upper B bus (cycle 0),
Upper FP ALU (cycle 0),
Upper Result (C) bus (cycle 2)
Latency: 3 cycles

This must be accomplished for all possible operand formats. Another point to make is the case where one of the source operands is an immediate and not a register. For floating point operations, this adds one cycle to the operation since only 16 bits can be loaded on a bus via a microinstruction per cycle and floating point immediates require 32 bits. This penalty is due to the fact that only 16 bits of the microinstruction is allocated for an immediate. Therefore the upper 16 bits are loaded into a register on the first cycle followed by the second 16 bits in the next cycle. The total possibilities for these three operand instructions are four - two destination possibilities times two register source possibilities. The following example shows the four possibilities:

add_f UR1, UR2, immediate
add_f UR1, LR2, immediate
add_f LR1, UR2, immediate
add_f LR1, LR2, immediate

Each must have a unique specification in the microcode MDF to list all resources required and the cycle each is required.

For integer ALU operations with immediates, there are double the number of possibilities since the immediate could require only 16 bits (saving one cycle) or require the full 32 bits incurring the additional cycle. This means that although there is one MDF specification for a three operand integer ALU assembly instruction, there are 14 corresponding microinstruction possibilities which could appear in the code stream - each requiring slightly different resources and the cycles each resource is needed. Obviously the latency can change as well due to the size of immediates.

One final point to make here is that only one execution option has been defined per instruction in this effort. Multiple options actually exist and defining those will reduce the total number of cycles in a given loop since more ILP would be possible. However, since

the instruction format would be the same for all options, Phase 3 would have to determine which option was chosen for each instruction. A method would have to be devised to handle this case. The scheduler may be able to do this, but it would then require modifications and violate the original design decision to prevent these changes. As was stated before, however, exceptions are made to this rule. If Phase 3 must determine which option was chosen, it would need its own resource contention scheme. This isn't too complicated for combining two instructions, but as the number of instructions to combine grows, the more difficult the search becomes since its probable that only one combination of all the instruction options is possible.

4.5.2 Prepass Code Scheduling. The first step in Phase 2 is Prepass Code Scheduling. The scheduler takes an Lcode function and schedules each control block's instructions based on their data dependencies and processor resource needs. To do this, the scheduler must have a machine description file which dictates the resources required for each instruction. The MDF information is read into internal data structures to speed resource contention calculations. Prepass scheduling performs an upfront scheduling of assembly control blocks followed by a scheduling of the microcode control blocks. Impact is currently designed to hold one machine description file's contents per scheduling pass. To handle a dual target compilation architecture, Phase 2 declared two pointers for each MDF infomation. On the first pass, the Impact MDF data structure is filled with the data corresponding to assembly code. On the second pass the MDF data structure is filled with the data corresponding to microcode. Once the second pass completes, the MDF data structure is reset to the assembly MDF information. Prior to each transition however, the routine Lsched_reinit() is called to reinitialized parameters used by the scheduler. This is done to remove any side effects that could be caused from the previous scheduling pass.

4.5.3 Register Allocation. Once the initial scheduling passes are complete for a function, the Register Allocator process begins. When Lcode and Phase 1 generated required a new register, they were simply created and the register number merely incremented - appearing to have an infinite register pool. However, processors only have a limited size register bank and these must be mapped to the registers found in the instructions. It is the

register allocator's task to make this mapping. Currently, the allocator simply makes efficient use of the processor's registers by only allocating the minimum number required per task. This works fine temporarily, but it misses a potential speedup gained if the allocator would assign registers based on minimum resource conflict rather than maximum register reuse. This is an obvious target for future optimizations. An optimization of this type is dependent on the machine description file since registers drive data busses which are resources. If all upper or lower registers could be assigned per an instruction dependence flow, and a parallel, but orthogonal, instruction dependence flow existed, then each flow could be allocated registers from the opposite side of the processor. One dependence flow would then be assigned all upper registers and the other dependence flow assigned all lower registers. This may not be entirely possible, but the allocator should be able to maximize ILP with this technique rather than maximizing the register reuse.

4.5.4 Annotating Instructions.   Figure 4.5 brings out two types of annotation which must occur here due to the dual target compilation process. The first, offset annotation, is for both assembly and microcode instructions and the second is for microcode instructions alone.

Offset Annotation.   Up to this point, all offsets from base addresses found in the instruction stream are given relative to four special areas located in the stack frame.

These offsets must be converted to offsets from the stack pointer, which is a special register reserved in all processors, so that Phase 3 can process the instructions correctly. By assigning the positions of these four areas in the stack, the offset is now easily determined with respect to the stack pointer. As Figure 4.6 shows, space is reserved for incoming parameters (IP), outgoing parameters (OP), local variables (LV), and register spills (RS). Other areas are reserved but aren't applicable to the offsets in Phase 2 annotation. As was already mentioned, this type of annotation must be performed on all instructions.

```
                Incoming Parameters (IP)      Previous Stack Frame

old stack pointer ──▶
                        Callee Registers

                      Local Variables (LV)

                       Allocation Space        Current Stack Frame

                    Register Spill Space (RS)

                    Outgoing Parameters (OP)
New stack pointer ──▶

                                              Next Stack Frame
```
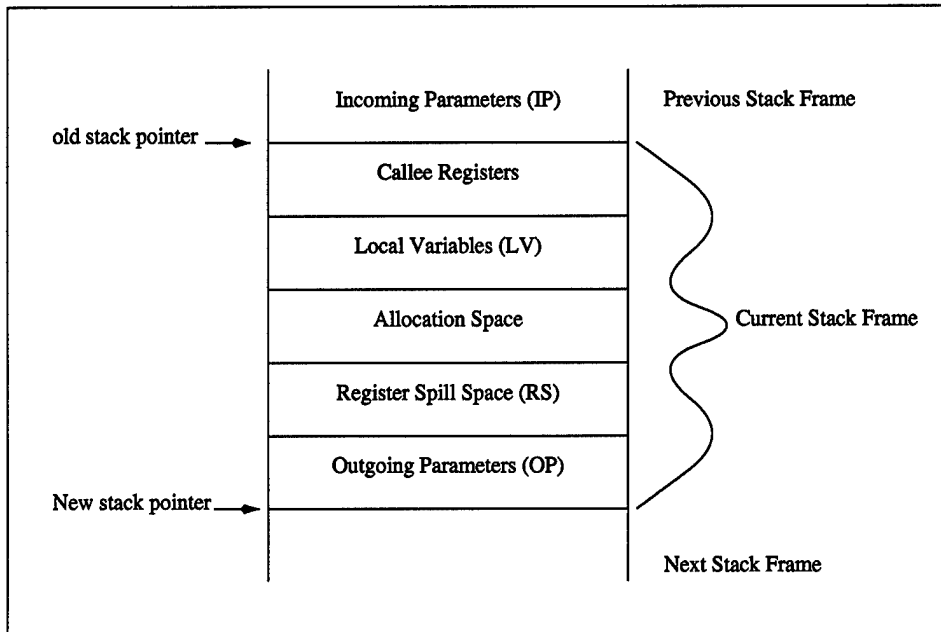
Figure 4.6   WSSP Phase 2 Stack

Microcode Annotation.   The microcode annotation is performed only on those portions of code tagged as being targeted for microcode and provided the microcode compilation flag is set. This process is directly tied to the microcode machine description file. Each instruction to be microcoded is identified by the type of operands, upper or lower, it has been allocated by the register allocator and the unique opcode value corresponding to that instruction format. As an example of defining a new opcode for a given microcode instruction, consider the mul_f UR1, UR2, UR3 again. The opcode created for it was LWSSP_op_FPMULT_uuu since it was a floating point multiply instruction and has been assigned all upper registers for operands. Phase 2 simply evaluates each instruction to determine the combination of its operands and assigns the opcode as appropriate. Many new opcodes were created for microcoding due to the many possible formats each instruction type can possess. Each of the opcodes correspond to one instruction type defined in the microcode MDF and are used to differentiate between instruction cases. This annotation must occur because the following process, Postpass Scheduling, will not function correctly without it. All defined opcodes can be found in the file lwssp_opc.h.

4.5.5 Postpass Scheduling. Similar to prepass scheduling, postpass scheduling occurs once for all the control blocks targetted for assembly code and a second time to process the microcode control blocks. It is initiated by a function call to Lsched_postpass_code_scheduling() in Lschedule.c. The assembly MDF is used for the first pass and the microcode MDF is used for the second pass. However, postpass performs a critical role in that it assigns the issue slot for each instruction. This issue slot represents the cycle the instruction will begin execution. Provided the MDF is correct, the resulting scheduling will produce no resource conflicts during Phase 3 microcode generation.

4.6 Phase 3

This section is designed to provide a closer look at Phase 3 which contains two main processes at the top level as shown in Figure 3.3. The Print Assembly Instructions process contains all the routines to handle printing out each instruction in the required WSSP ISA format. Given a function, routine P_process_func() processes each assembly code control block by passing each of its instructions to the routine P_print_oper(). This routine processes each instruction by passing it to the appropriate print routine which will print the instruction to <filename>.s. These routines can be found in the source code file for Phase 3, lwssp_phase3_func.c.

The handling of the microcode, however, requires a finer level of detail and complexity. Beginning from Figure 3.3, each of the six processes handling the microinstructions flow will be explained and the abstract data structures to hold them. But first, understanding the WSSP microinstruction format is necessary.

4.6.1 The WSSP Microinstruction Format. The WSSP microinstruction is 128 bits long and is composed of 35 fields. Each field represents the control of some portion of the processor. Each field has mnemonics associated with it and a corresponding binary value for all possible field values. Rome Labs and AFIT's Department of Electrical and Computer Engineering possess the specifications that describe each of these fields in detail. This information will not be duplicated here, but as an example, consider the integer ALU instruction ADD UR1, UR2, UR3. The corresponding microinstruction would have
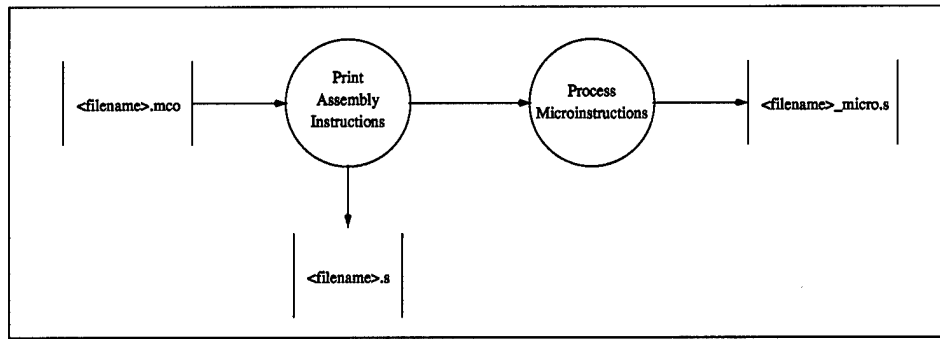
Figure 4.7   Phase 3 Top Level Processes

the appropriate fields set to the corresponding values for R1, R2, and R3. The field controlling the INT ALU would specify an add operation for this cycle. The result is placed on the upper result bus and latched into upper register one (UR1). All this is accomplished in one cycle, therefore only one microinstruction is needed. The resulting mnemonic microinstruction would look something like the following:

upA=R2 upB=R3 R1=upC upADD

The microinstruction itself would have the bit patterns filled into the corresponding fields. All fields not specified are filled with zeros - which represents a NOP control signal.

4.6.2   Microinstruction Data Structures.   The following three types and two constants were declared in order to model a microinstruction:

```
#define NUM_FIELDS 35
#define NUM_WORDS 4

typedef struct Micro_Field {
        int value;
        int size;
} Micro_Field;

typedef struct Micro_Word {
        int value;
        int size;
} Micro_Word;
```

```
typedef struct Micro_Instruction {
        struct Micro_Field field[NUM_FIELDS];
        struct Micro_Word word[NUM_WORDS];
} Micro_Instruction;
```

In addition to these declarations, a runtime variable is declared which consists of an array of 64 Micro_Instructions which is filled per microcode control block. This data structure is what is "filled" during Create Microinstructions by the individual microinstruction fill routines. Each element in the array represents one cycle of execution in microcode. If the array is too small to create all the microinstructions required for a particular control block, Lpunt() is called and Phase 3 exits gracefully.

4.6.3 Create Entry and Exit Instructions.   The entry and exit microinstructions correspond to the prefetch operation mentioned in Chapter 3. By performing these operations, the next assembly instruction is insured to be loaded on the required bus before execution of that instruction is initiated. The WSSP has two registers and an instruction bus to handle this prefetch requirement. Upon entering microcode, the IR, or instruction register, contains the next assembly instruction to be executed. The PREV_IR, or previous instruction register, holds the current assembly instruction being executed. The IR bus also contains the current assembly instruction being executed. This allows execution of the microcode which implements the current assembly instruction. However, before exiting microcode, the IR contents must be moved to the PREV_IR register which displaces the contents on the IR bus. The next assembly instruction can then be read in from memory into the IR register.

In order to retrieve the next assembly instruction before exiting microcode, the address of that instruction must be available. The PC register, also known as the A pointer, is incremented once so that it now points to the next assembly instruction. This value is then moved to the Memory Address Register (MAR) so that the instruction can be read in. This prefetching, or entry and exit process, can be done in three cycles as shown in Figure 4.8.

The first cycle is to increment the PC address. The second cycle is to move this address into the MAR as well as tell the processor to MAP off the instruction being loaded

4-15
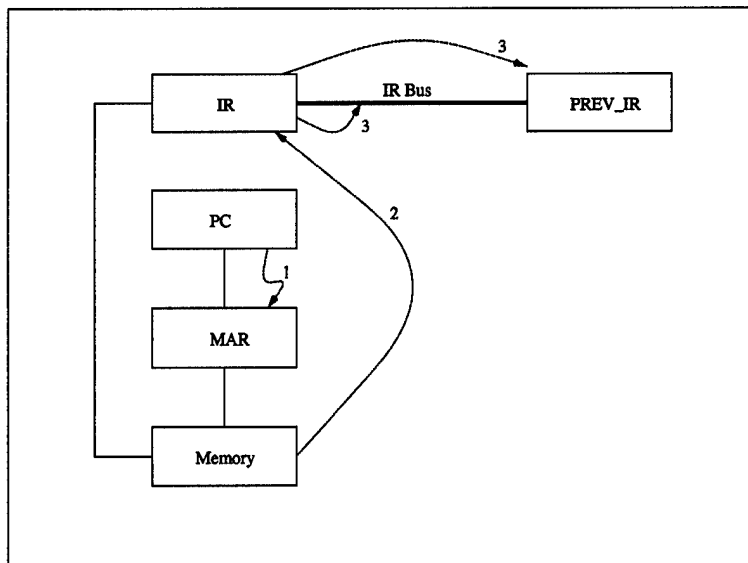
Figure 4.8   Entry and Exit Instructions Flow

in on the next cycle. The processor is now set up to read in the next assembly language instruction and begin execution from there and not from the following microinstruction - which is the default path of execution. The final cycle will move the contents of the IR into the PREV_IR and fetch the next assembly instruction placing it into the IR. The following code sequence shows where the Entry and Exit microinstructions are placed when microcoding a loop:

```
        entry micro
Loop:
        micro1
        micro2
        micro3
        ...
        BR Loop
        exit1 micro
        exit2 micro
```

When microinstructions are created for a straight-line portion of code, meaning no loop, the Entry and Exit instructions can be combined with the rest of the microinstructions and not kept separate as in the loop code. This is due to the fact that they will only be executed once, which is not the case if they're embedded in a loop.

4.6.4 Create Microinstructions. This process is implemented by the routine M_fill_micro_instructions() found in lwssp_micro_func.c. It takes the control block to be microcoded and the microinstructions array as input to generate all the microinstructions required for the given instruction stream. This routine passes each of the instructions in the control block to the corresponding microinstruction fill routine. For example, if a mul_f UR1, UR2, UR3 is encountered, the routine that handles floating point multiply instructions with all register operands, M_fill_FPMULT_instruction(), will be called. Since this type of instruction requires three cycles to execute, three microinstructions will be filled to implement it. Recall that Phase 2 establishes the slot each instruction is to begin execution. The first microinstruction will be placed at the predetermined slot, or cycle, in the array and the next two instructions immediately follow. So if our instruction stream contained only one instruction, mul_f UR1, UR2, UR3, then the microinstruction stream would look like the following:

    entry micro
    mul_f1 micro
    mul_f2 micro
    mul_f3 micro
    exit1 micro
    exit2 micro

Note that mul_f1, mul_f2, and mul_f3 would be in the microinstruction format discussed in the previous section.

To get a better understanding of how an assembly instruction is transformed into a microinstruction, I'll explain each cycle required for the example mul_f instruction.

The mul_f UR1, UR2, UR3 instruction can be broken down into three operations in the WSSP processor. The first operation is to place the source operands on the appropriate data busses for input into the FP MUL. The first operation also includes signaling the FP MUL to begin operation. The second operation is actually a wait, or NOP, while the FP MUL executes. The final operation, or cycle, specifies placing the FP MUL result onto the upper C bus and latching it into upper register one, UR1. The following microinstructions

are the result of microcoding the mul_f instruction using the mnemonics provided in the WSSP Microcode Specifications. The remainder of the fields are NOPs and are not shown.

    mul_f1: AU=R2 BU=R3 FP*
    mul_f2: NOP
    mul_f3: ,R1=CU CD_UP C=FP*

FP* means start the FP MUL functional unit, NOP means do nothing, and CD_UP and C=FP* mean drive the upper C bus with the FP MUL result. AU=R2 and BU=R3 declare placing those register values on the upper A bus and upper B busses respectively. R1=CU specifies latching the result into upper register one.

All instructions must be defined as the above in order to be microcoded and the MDF must match these instruction codings exactly to ensure no conflicts result. A microinstruction routine exists for each instruction type with a similar naming convention as for the fp_mul instruction. Once all instructions have been microcoded in this way, the Create Microinstructions process is complete and the next process, Handle Branch Instructions is begun.

4.6.5 Handle Branch Instructions.   Branch instructions are handled in the same manner as was discussed in Chapter 3. Having a basic understanding now of the WSSP microinstruction, it can be seen that to microcode any assembly instruction, all that is required is an understanding of the processor operations needed to specify for each cycle. Branch instructions are no different except that the last branch may not begin in the slot allocated by the Phase 2 scheduler. Recall that a FP MUL, or any floating point instruction, requires at least three cycles to execute. If the scheduler determines the branch can be issued in the same cycle the FP instruction is begun and the branch requires only one cycle, then the branch could occur before the FP instruction has the chance to complete. This means the third microinstruction specifying what to do with the FP result will not be executed and so an incorrect result occurs. This case is handled in the M_fill_branch_instruction() routine.

4.6.6 Print Microinstructions.   Print Microinstructions is a simple function performed by the routine M_print_microinstructions() in file lwssp_micro_func.c. It prints all

microinstructions to the file <filename>_micro.s. The important feature to bring out here is the format of the microinstructions when printed to file. The label of each microcode portion is named using the convention $_<function_name>_micro_x, where the name of the function the microcode control blocks resides in forms part of the label name and the x at the end of the label name is simply the x*th* control block of microcode generated. So if two loops are converted to microcode in function main(), then the two labels would be: $_main_micro_1 and $_main_micro_2. Table 4.1 shows the format of two microinstructions when printed to file.

Table 4.1   Microinstruction Print Format

```
.align 3
.global $_main_micro_1
$_main_micro_1:
.int 0B0000000000000000110000000000000, 0B0000000000000000000000000000000000
.int 0B0000000000000001000000000000000, 0B0000000000000000110000000000000
.int 0B0000000000000000000000000000000, 0B0000000000000000000000000000000000
.int 0B0000000000000000000000000000000, 0B0000000000000000000000000000000000
.int 0B0000000000000000000000010111110, 0B0000000000000000000000000000000000
.int 0B0000000000000000000000000000000, 0B0000000000000000000000000000000000
.int 0B0000000000000000011010000000000, 0B0000000000000000000000000000000000
.int 0B0000000000000000000000000000000, 0B0000000000000000000000000000000000
```

Note that each line above contains 64 bits, so that two lines make up one 128 bit microinstruction. However, there's a feature to this format that isn't obvious. The instructions are actually printed out in pairs, with the first line being the first 64 bits of the first microinstruction. The second line is the first 64 bits of the second microinstruction. The third line is the last 64 bits of the first micrinstruction, and the fourth line is the last 64 bits of the second micoinstruction. This pattern repeats itself for the remainder of the microinstructions. Another point to remember is that there must be an even number of microinstructions. If the control block only generates an odd number of instructions, then a NOP microinstruction must be tagged onto the end of the code stream. This is simply a WSSP implementation feature.

4.6.7 Print Micro Call Instructions.   This process is performed by the routine M_print_micro_call_instructions(). It prints the four load immediates, the LOADRAM and

LOADMAP instructions to the assembly file <filename>.s as well as the inline instruction to make the jump to microcode. This inline instruction has the following format:

```
.int 00000000000000000000000000000000
.int 00000000000000000000000xxxxxxx0
```

The sequence of seven x's represents the opcode for this "JSR" which was placed in the mapping table by the LOADMAP instruction. When this "inline" instruction is reached during execution, a jump to the loaded microcode is performed. For example, if the mapping instruction was LOADMAP 127, 4032, then the opcode would be 127 and the inline instruction would have the form shown below:

```
.int 00000000000000000000000000000000
.int 00000000000000000000000011111110
```

It was discussed previously that a preheader control block should be created during Phase 1 and to place the four LIs, LOADRAM, and LOADMAP instructions in it. This would prevent reloading of the map and microinstructions and generate a single occurence of this overhead cost. However, this feature has not been implemented due to a lack of simulator availability and adequate testing time to insure a correct implementation.

4.7 Conclusion

This chapter emphasized the details involved in the dual target code generator for the WSSP. Some important features were the splitting of the assembly and microcode portions of code and the need for a MDF to specifiy the resources required per cycle for each instruction to be microcoded. This required a major effort due to the multiple formats for each instruction type depending on the register used, whether upper or lower. Recall that the MDF and the scheduler of Phase 2 work hand-in-hand to generate the issue slot for all instructions destined for the WCM. This issue slot per instruction is essential for Phase 3 which creates the microinstructions for a control block and must know their position to be executed. The format of a microinstruction for the WSSP architecture and the data structures designed to model it were explained. Also explained were the specific design decisions made and the routines used in each Phase to provide the interested reader and

future efforts with a more complete understanding of the implementation of the WSSP code generator.

V. Analysis and Conclusions

5.1 Introduction

This chapter presents the results of the benchmarks used to demonstrate that the dual-targeted compiler using VLIW techniques is a viable method in creating microcode to produce a substantial speedup over assembly code. A discussion is presented on the benchmarks chosen and their resulting execution times both in assembly and microcode. The times are for the entire program as well as the total time spent in microcode in order to provide an overall speedup impact versus the speedup of the microcoded portions themselves. The results are based on simulator data since the WSSP is not currently available for use. The chapter concludes with a summary of research contributions and recommendations for future work.

5.2 The Benchmarks

The two benchmarks, Matrix Multiply and Precision Tracker, were chosen due to their planned use on the WSSP. Both benchmarks contain a mix of integer and floating point operations as well as possessing inner loops which could be translated into microcode. Matrix Multiply represents a common single instance of an inner loop, while Precision Tracker takes into consideration an entire program which utilizes multiple inner loops as part of its execution. In these cases, all inner loops are "for" loops which provided a foreknowledge of the number of iterations each execution of the inner loop would be. Without knowing this information, profiling would need to occur to determine if the number of iterations of the inner loop overall would be sufficient to warrant microcoding the inner loop. The following paragraphs explain the results of both benchmarks.

Matrix Multiply Results. The source code of Matrix Multiply and the assembly code of its inner loop are shown at Table 5.1. This code consists of three "for" loops each of which will iterate four times. The last "for" loop, the inner loop, was converted to microcode. Using the assembly code specifications from Rome Labs, this loop requires 34 cycles per iteration. Four iterations per call requires 136 cycles each for each execution of the inner loop. Since this loop is nested within two, four-iteration loops, it

will be called 16 times for a total of 2,176 cycles in assembly code. Table 5.2 reveals the resulting microcode generated for the Matrix Multiply routine. The corresponding assembly instructions are placed in the microinstruction they begin execution for easier comparison. NOP microinstructions are explicitly shown while microinstructions with no mnemonics are not NOPs, but represent additional cycles needed to execute the instructions shown above it. The microcode loop required 19 cycles per iteration. Adding in the three cycles for prefetching, the microcoded version requires 19*4 + 3 = 79 cycles for each call of the inner loop. Using the formulas from Chapter 3, the overhead associated with microcoding in this case is 68 cycles. This overhead occurs only once since the Matrix Multiply routine is called only once and no branches or jumps occur within the three "for" loops. Because of this, the overhead code was moved outside the outermost "for" loop to minimize its cost. Therefore the total microcode speedup gained is:

$$
\begin{aligned}
Speedup \quad &= \quad Assembly\ Cycles \div Microcode\ Cycles \\
&= \quad 2176 \div (68 + 79 * 16) \\
&= \quad 1.634 \\
&= \quad 63.4\%
\end{aligned}
$$

The asymptotic upper bound for speedup in this case is

$$
\begin{aligned}
UpperBound \quad &= \quad (136 * N) \div (O + 79 * N) \\
&= \quad 136 \div 79 \\
&= \quad 73\%
\end{aligned}
$$

where O represents the overhead, which is constant, and N represents the number of inner loop iterations which is also the dimension of the matrix. This is due to the fact that increasing the matrix size doesn't change the code to multiply matrices, only the loop iteration count changes. If the size is increased to 100, which is not unreasonable and is actually more realistic for applications on DSPs, the speedup would be (136 * 100 * 100)

/ (68 + 79 * 100 * 100) = 1.721, or a 72% speedup which approaches the upper limit. As a final note, the program time for Matrix Multiply in assembly code was 134,550 ns, while the program time with microcoding was 110,925 ns. Therefore, the program speedup was 134550 / 110925 = 21.3%. This result seems like a major drop in speedup for the program overall, however, in addition to and prior to the matrix multiply routine, two matrices are created and initialized. This requires the execution of two nested "for" loops. Since the size of the matrix is 4 x 4, the cost of initialization is 16 + 16 = 32 operations, whereas the cost for the multiply is 4 * 4 * 4 = 64. Half of the total execution time is spent in initialization. As the size of the matrices get much larger, however, this cost will become negligible. For example, if the size is 100, then the initialization cost is 1,000 + 1,000 = 2,000, and the multiply cost is 100 * 100 * 100 = 1,000,000. Therefore the initialization cost is reduced to only 2% of the total execution time.

Table 5.1   Matrix Multiply Code

Source Code	Assembly Code

for (i=0; i<4; i++)
    for (j=0; j<4; j++) {
        C[i][j] = 0.0;
        for (k=0; k<4; k++)
            C[i][j] = C[i][j] + A[i][k]*B[k][j];
    }
	Serial_matrix_mult_5:
	    LW LR7,LR5(UR6)
	    LW LR4,UR7(UR5)
	    SETFP 0x0
	    LW UR8,UR7(LR6)
	    ADDI LR9,LR9,1
	    ADDUI UR5,UR5,40
	    FPS_M LR7,LR7,LR4
	    ADDI LR5,LR5,4
	    CMP LR9,LR8
	    FPS_A LR4,UR8,LR7
	    SW UR7(LR6),LR4
	    BR UILT,Serial_matrix_mult_5
	    NOP

Precision Tracker Results.   The program Precision Tracker is a much larger program than Matrix Multiply and is currently being used as the test program for the newly manufactured chip. Due to the size of the program, the simulation would require over 300 hours to execute. Because of this, profiling of the code was performed and revealed

Table 5.2   Matrix Multiply Microcode

Microcode
$_matrix_mult_micro_1:
#0 Entry
#1 LW LR7,LR5(UR6)
.int 0B00000000000000000110000000000000, 0B00000000000000000000000000000000
.int 0B00000001100000001000000000000000, 0B00000000000000000000000000000000
.int 0B00000000000000000000000000000000, 0B00000000000000000000000000000000
.int 0B00000000000000101000000000000000, 0B00000000010110010001010000000000
#2 LW LR4,UR7(UR5), ADDI LR9,LR9,1
#3
.int 0B00101001110000001000000000111010, 0B00000000000000010000000000000000
.int 0B11111000000000000000000000111010, 0B00000000000000000000000000000000
.int 0B00101100101000000000000000000000, 0B00000000010110010000000100101001
.int 0B00000000000001000000000000000000, 0B00000000000000010111110000000111
#4
#5 SETFP 0x0
.int 0B11111000000000000000000000000000, 0B00000000000000000000000000000000
.int 0B00000011101000000001000000000000, 0B00111111111111110000000000000000
.int 0B00000000000001000000000000000000, 0B00000000000000010111110000000100
.int 0B00001000100110000000000000000000, 0B00000000000010001000001110100000
#6 LW UR8,UR7(LR6)
#7 FPS_M LR7,LR7,LR4
.int 0B00111000000000001000000000000000, 0B00000000000000000000000000000000
.int 0B00000000000000000000000000111010, 0B00000000000000000000000000000000
.int 0B00101100100000001000000000000000, 0B00000000000000000000000011000000
.int 0B00000000000001001100000000000000, 0B00000000000000000011100010000000
#8
#9 ADDUI UR5,UR5,40
.int 0B11111000000100000000000000000000, 0B00000000000000000000000000000000
.int 0B00000001010010100000010000000000, 0B00000000000101000000000000000000
.int 0B00000000010001000000000000000000, 0B00000000000000000111110000000000
.int 0B00101100101010000000000000000000, 0B00011110000000000000000000000111
#10 FPS_A LR4,UR8,LR7
#11 ADDI LR5,LR5,4
.int 0B01000000000000000000000000000000, 0B00000000000000000000000000000000
.int 0B00000000000000000000000000000000, 0B00000000000001000000000000000000
.int 0B00000000010000010101000010100000, 0B00000000000000000000000011100000
.int 0B00000000000100000000000000000000, 0B00000000010110010000000010100101
#12
#13 CMP LR9,LR8
.int 0B00000000000000000000000000000000, 0B00000000000000000000000000000000
.int 0B00000000000000000000000000000000, 0B00000000000000000000000000000000
.int 0B00000000000000000000000000000000, 0B00011111000000000000000000000100
.int 0B00000000000000000000000000000000, 0B00000000001101001001001010000000
#14 SW UR7(LR6),LR4
#15
.int 0B00000000001111100000000000000000, 0B00000000000000000000000000000000
.int 0B00111000000000001000010000000000, 0B00000000000000000000000000000000
.int 0B00000000000000010000000000000000, 0B00010010000000000000000000011111
.int 0B00101100100000001000000000000000, 0B00000000000000000000000011000000
#16
#17 BR UILT,Serial_matrix_mult_5
.int 0B00000000000000000000000000000010, 0B00000000000000000000000000000000
.int 0B00000000000000000000000000000000, 0B00000111111000001001011010000000
.int 0B00000000000000000000000000000000, 0B00000000000000000000000000000000
.int 0B00000000000101100000000000000000, 0B00000000000000000000000000000000
#18 NOP
#19 Exit1
.int 0B00000000000000000000000000000000, 0B00000000000000000000000000000000
.int 0B00000000000000010000000000000000, 0B00000000000000011000000000000000
.int 0B00000000000000000000000000000000, 0B00000000000000000000000000000000
.int 0B00000000000000000000000000000000, 0B00000000000000000000000000000000
#20 Exit2
#21 NOP
.int 0B00000000000000000000000010111110, 0B00000000000000000000000000000000
.int 0B00000000000000000000000010000000, 0B00000000000000000000000000000000
.int 0B00000000000000000011010000000000, 0B00000000000000000000000000000000
.int 0B00000000000000000000000000000000, 0B00000000000000000000000000000000

that approximately 80% of the total execution time occured in routines containing inner loops which could be microcoded. In all, 26 inner loops were microcoded. The speedup was determined for each inner loop and for the routines they were contained in. Amdahl's law was then used to determine the speedup of Precision Tracker overall. Table 5.3 shows the simulation results for this benchmark. The column "Loop Structure" depicts the number of loops involved and the number of times each loop is iterated. The table contents are provided in increasing order of total loop iterations. The structures range from a simple nine iteration "for" loop to the triple "for" loops iterated nine times each.

Table 5.3   Precision Tracker Results

	Loop Structure	Assembly Time (ns)	Microcode Time (ns)	Speedup
1	9	4,000	3,400	11.8%
2	9	4,225	3,650	15.8%
3.	9	5,800	4,750	22.1%
4	3x3	4,380	3,850	13.8%
5	3x3	6,150	5,100	20.6%
6	4x4	6,600	5,100	29.4%
7	9x9	44,100	26,600	65.8%
8	9x9	33,975	20,100	69.0%
9	9x9	33,975	20,100	69.0%
10	9x9	33,975	20,100	69.0%
11	9x9	54,225	30,750	76.3%
12	9x9	40,050	22,450	78.4%
13	9x9	36,000	20,100	79.1%
14	9x9	36,000	20,100	79.1%
15	9x9	36,000	20,100	79.1%
16	9x9	48,150	26,600	81.0%
17	9x9x1	70,425	40,850	72.4%
18	9x4x4	91,800	50,900	80.4%
19	9x9x4	230,850	139,300	65.7%
20	9x9x4	190,300	106,700	78.4%
21	9x9x9	524,475	301,300	74.1%
22	9x9x9	633,825	354,050	79.0%
23	9x9x9	488,025	264,750	84.3%
24	9x9x9	579,150	317,500	82.4%
25	9x9x9	524,475	280,950	86.7%
26	9x9x9	524,475	264,750	98.1%

5.3 Benchmarks Analysis and Conclusions

By observing the microcode of Matrix Multiply in Table 5.2, it can be seen why the speedup was less than optimal. Pipelining is occuring since instructions are executed one cycle after the previous one. However, only once did combining appear. In the third microinstruction, LW LR4,UR7(UR5) and ADDI LR9,LR9,1 were combined. This is due to the current design of the register allocator, maximum register reuse, and the fact that only one implementation option exists for each instruction. If the registers were allocated based on the dependence chains spoken of in Chapter 3 and the scheduler was given a wider range of resource assignment possibilities per instruction, a greater speedup could be attained. Also, the WSSP contains special registers for loop counts and address calculations which free up the source and result busses for use by other instructions. Use of these registers was beyond the scope of this thesis.

The results of Precision Tracker brought out the same conclusions found in the Matrix Multiply routine, however, another point was established in that the deeper the inner loop was nested or the more work the inner loop performed, the greater the potential for increased speedup. Note that more work performed in the inner loop does not *guarantee* a better speedup. This can be seen by comparing loop 22 which contained 13 assembly instructions and required 633,825 ns to execute with loop 23 which contained only 10 assembly instructions and required 488,025 ns to execute. A better speedup was gained while less work was involved (84.3% vs 79.0%). Loop structure 1 contained two labels which had to be removed prior to microcoding. The fact that the microcoding still provided a speedup demonstrates that labels aren't a reason to chose not to microcode a given inner loop, though it did reduce the overall speedup attained for that particular loop structure. As a final note, a greater speedup for Precision Tracker *could* be attained if microcoding wasn't limited to only 80% of the code.

The results of the benchmarks on the VSIM simulator demonstrate that the dual-target compiler methodology chosen for the WSSP can produce code that will correctly execute and achieve a reasonable speedup over simple execution in assembly code. This demonstrates that combining what was previously done with two or more separate tools can be achieved in one compilation pass with good results. The use of two machine description

files and generating separate opcodes for the microcode target proved effective in ensuring no resource conflicts existed and correctly scheduled code resulted.

5.4 Contributions

5.4.1 Dual-Target Compilation.
This effort has demonstrated the validity of such a compiler to generate correct code and achieve a performance gain as well. Although other methods may exist to create a dual target compiler, modifying a retargetable compiler which utilizes a machine desription file for resource mapping provides successful results.

5.4.2 A Microcode Compiler for the WSSP.
A microcode compiler did not exist for the WSSP designed by Rome Labs. This effort has successfully created one. Since this processor is destined for use in both military and commercial applications, this compiler provides a major milestone in moving the WSSP to these significant areas of application.

5.5 Recommendations and Future Work

5.5.1 Minimize Microcode Preloading.
Future work can be done in the area of reducing the overhead incurred when microcoding. A method does not currently exist in this implementation to minimize the loading of microcode into the WCM. A way of polling the internal map to see if the "branch to microcode" opcode is valid and matches this particular code may be possible. Care must be taken to insure the opcode wasn't reused for another portion of microcode however. Another possibility may be a software trap routine which could be used to reduce the preloading of microcode as well. Also, the overhead code can be moved from the point at which each inner loop is called to just outside the outermost loop. This can be combined with the above suggestion and greatly reduce the overhead. The deeper the inner loop is nested, the greater the speedup gained will be by moving these overhead instructions outside the outermost loop.

5.5.2 Marking the Microcode Control Blocks.
Future efforts must also eliminate the manual process of marking the control blocks to be microcoded. [Liu 79] presents four methods spoken of in Chapter 2 which could be implemented to chose the microcodeable

loop. This process should be implemented prior to Phase 1 which means a potential modification to Impact. To prevent this, another phase could be added to the code generator to perform this marking task.

5.5.3 Maximizing the Combining of Microinstructions.

Future efforts should include the modification of the scheduler and/or the register allocator to maximize the combining of code destined for microcode rather than register reuse as is currently being performed. Efforts need to be made to generate multiple dependence flows which may exist within the microcode control block and alternately allocate upper and lower registers to insure an optimal amount of combining is achieved.

5.5.4 Use of Special Purpose Hardware.

The WSSP possesses special purpose registers for use in loop optimizations. These components are utilized in the vector instructions already provided in the ISA, but have not been taken advantage of in this effort. Use of incrementor registers for loop counts and offset calculations would reduce the total cycles required by eliminating the need to add offsets to base addresses. Future work should make use of these special purpose hardware resources to reduce the total cycles required and hence achieve added speedup per loop.

5.5.5 WSSP Vector Instructions.

Since the vector instructions available in the WSSP ISA have not been implemented in this effort, work should entail adding these instructions to the list of microcodeable instructions or determining if another implementation would be more beneficial. The current implementation of these instructions in ROM may not be the optimal solution and generating microcode for the same routine in the WCM may prove to generate a greater speedup.

5.5.6 Microcode Branching.

One final effort for future work should address and implement branching in microcode. Currently the solution presented in Chapter 3 has not been implemented. A much broader scope of microcodeable control blocks would result if this scheme, or an improvement of it, were implemented for the WSSP. Since the branches out of microcode generate some overhead, a new calculation of the overhead would be

necessary and must be added to the marking of microcode control blocks effort mentioned above.

5.6 Closing Remarks

It can be seen that a dual-target compiler has great potential to make significant contributions to the current use of microprogrammed DSPs. Through the automated exploitation of the WCM, a significant speedup can be achieved thereby granting improved performance to the processor. Since these processors are destined for use in military applications, this effort can have a direct and positive impact on the effectiveness of future US warfighter capabilities.

Bibliography

1. Abd-Alla, A. M. and D. C. Karlgaard. "Heuristic Synthesis of Microprogrammed Computer Architecture," *IEEE Transactions on Computers, C-23*(8):802–807 (August 1974).

2. Comtois, John Henry. *Architecture and Design for a Laser Programmable Double Precision Floating Point Application Specific Processor*. MS thesis, AFIT/GE/ENG/88-5. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1988.

3. Fisher, Joseph A. "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers, C-30*(7):478–490 (July 1981).

4. Gallagher, David M. *Rapid Prototyping of Application Specific Processors*. MS thesis, AFIT/GE/ENG/87D-19. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1987.

5. Gallagher, Lt Colonel David M. Class Notes, CSCE 692, Computer Architectures, Fall Quarter 1996.

6. Goossens, Gert, et al. "An Efficient Microcode Compiler for Application Specific DSP Processors," *IEEE Transactions on Computer-Aided Design, 9*(9):925–937 (September 1990).

7. Hennessy, John L. and David A. Patterson. *Computer Architecture, A Quantitative Approach*. San Francisco, California: Morgan Kaufmann Publishers, Inc., 1996.

8. Linderman, R. W., et al., "A Full Custom DSP Optimized for MFLOPS/Watt." Unpublished Paper, 1997.

9. Liu, Phillip S. and Frederic J. Mowle. "Techniques of Program Execution with a Writable Control Memory," *IEEE Transactions on Computers, C-27*(9):816–827 (September 1978).

10. Mano, M. Morris. *Computer System Architecture*. Englewood Cliffs NJ: Prentic-Hall, Inc., 1982.

11. McGuire, Jerry, "For Efficient Signal Processing in Embedded Systems, Take a DSP, not a RISC." WWWeb, http://www.analog.com/publications/magazines/Dialogue/30-3/efficient.html, September 1997.

12. Patterson, David A. and John L. Hennessy. *Computer Organization and Design. The Hardware/Software Interface*. San Francisco, California: Morgan Kaufmann Publishers, Inc., 1994.

13. Rauscher, Tomlinson G. and Ashok K. Agrawala. "Dynamic Problem-Oriented Redefinition of Computer Architecture via Microprogramming," *IEEE Transactions on Computers, C-27*(11):1006–1014 (Nov 1978).

14. Shin, H. and M. Malek. "Identification of Microprogrammable Loops for Problem Oriented Architecture Synthesis." *MICRO 16. Proceedings of the 16th Annual Microprogramming Workshop.* 122–127. October 1983.

Vita

Captain Randall S. Whitman was born in ███████ on ███████. Following graduation from Southview High School in Lorain, Ohio, he enlisted in the Air Force as an Armament Systems Specialist and was assigned to Ramstein AB, Germany where he worked on the F-4E, F-15, F-16, and A-10 aircraft. While there he met his wife Susan and was shortly thereafter sent to Tyndall AFB, Florida. He applied and was accepted for the Airman's Education and Commissioning Program. After graduating from Wright State University in Dayton, Ohio with a Bachelor of Science in Computer Science degree, he was sent to Officer's Training School. Upon graduation he was assigned to Colorado Springs, Colorado where he oversaw installations of space systems into Cheyenne Mountain and maintained the hardware and software components of the Global Positioning System (GPS). Following graduation from AFIT, Captain Whitman will be assigned to USTRATCOM/J25 at Offutt AFB, Nebraska.

Permanent address: ███████

REPORT DOCUMENTATION PAGE		*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE December 1997	3. REPORT TYPE AND DATES COVERED Master's Thesis	

4. TITLE AND SUBTITLE MICROPROGRAMMING A WRITEABLE CONTROL MEMORY USING VERY LONG INSTRUCTION WORD (VLIW) COMPILATION TECHNIQUES	**5. FUNDING NUMBERS**
6. AUTHOR(S) Randall S. Whitman, Captain, USAF	

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology 2750 P Street WPAFB OH 45433-7126	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT/ENG/GCS/97D-19
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Dr Richard Linderman RL/OCSS 26 Electronic Pkwy (Bldg 106) Griffiss AFB, NY 13441	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for public release; distribution unlimited	**12b. DISTRIBUTION CODE**

13. ABSTRACT *(Maximum 200 words)*

Microprogrammed Digital Signal Processors (DSP) are frequently used as a solution to embedded processor applications. These processors utilize a control memory which permits execution of the processor's instruction set architecture (ISA). The control memory can take the form of a static, read only memory (ROM) or a dynamic, writeable control memory (WCM), or both. Microcoding the WCM permits redefining the processor's ISA and provides speedup due to its instruction level parallelism (ILP) potential. In the past, code generation efforts for microprogrammable processors focused on creating assembly and microcode as two separate steps.

In this thesis, an alternative approach was chosen which combines the separate code generation steps into one automated, dual-target compilation process using the advanced techniques of VLIW compiler technology. The architecture chosen for this effort is a microprogrammable DSP being developed by Rome Labs, New York. The prototype compiler developed in this effort has demonstrated the potential for speedup of microcoded program portions over their assembly code counterparts. Therefore, the feasibility of program speedup produced by a dual-target compiler using VLIW compilation techniques has been validated.

14. SUBJECT TERMS Microprogram, Digital Signal Processor, Compiler, Very Long Instruction Word, Instruction Level Parallelism, Dual-Target Compiler	**15. NUMBER OF PAGES** 76
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL