EVALUATION OF SOME SUPERSCALAR AND VLIW PROCESSOR DESIGNS

BY

JOHN GUSTAF HOLM

B.S.E., University of Michigan, 1989

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1992

Urbana, Illinois

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Wen-mei Hwu, for his technical advice and patience. I have learned more from Wen-mei than could ever be written in a thesis.

I wish to extend my appreciation to William Chen, Scott Mahlke, Tom Conte, Rick Hank, and Pohua Chang for their outstanding technical input on this project. William wrote the trace generator and a simulator for VLIW architectures. His understanding of trace generation proved to be very valuable. Scott wrote the optimizing compiler and was always there when I had questions. He also improved the code scheduler. Tom was very helpful in answering questions about simulation on micro-architectures. Rick wrote the register allocator. Pohua is responsible for writing most of the IMPACT-I C compiler and helped me start on this work. He is also responsible for the original code scheduler.

Finally, I wish to thank everyone above and all of my other friends who have made my life in graduate school very enjoyable.

TABLE OF CONTENTS

Page

1. INTRODUCTION 1

2. OVERVIEW OF IMPACT 2

3. CODE SCHEDULING 4

4. HARDWARE MODELS 12
   4.1 The VLIW Hardware Model 12
   4.2 Superscalar In-order Hardware Model 18
   4.3 Superscalar Out-of-order Hardware Model 19
   4.4 Register Renaming 23
   4.5 Branch Prediction 24

5. SIMULATOR 25
   5.1 Simulation of a VLIW Architecture 30
      5.1.1 Fetch stage 30
      5.1.2 Decode stage 32
      5.1.3 Execute stage 32
   5.2 Simulation of a Superscalar In-order Issue Architecture 33
      5.2.1 Fetch stage 33
      5.2.2 Decode stage 33
      5.2.3 Execute stage 34
   5.3 Simulation of a Superscalar Out-of-Order Issue Architecture 35
      5.3.1 Fetch stage 35
      5.3.2 Decode stage 35
      5.3.3 Execute stage 35

6. RESULTS 37
6.1 Resource Constraints 39
6.2 Effects of Window Size 40
6.3 Effects of Code Scheduling 41
6.4 Effects of Register Renaming 42

7. CONCLUSIONS 44

APPENDIX A. CODE SCHEDULER 46
A.1 Scheduler 46
A.1.1 Important directories 47
A.1.2 Syntax of the command 47
A.2 Instrumenting for Trace Generation 50
A.3 Merging the Information 50
A.4 Linking 50

REFERENCES 53

LIST OF TABLES

Table Page

3.1: Heuristic Weights of Nodes for Restricted Code Motion 9
3.2: Heuristic Weights of Nodes for General Code Motion 10
6.1: Benchmarks 38

LIST OF FIGURES

Figure Page

2.1: Example of a Superblock 3
3.1: DAG of Instructions 6
3.2: Levelized DAG Using Restricted Code Motion 7
3.3: Levelized DAG Using General Code Motion 8
3.4: Schedule for Restricted Code Motion 11
3.5: Schedule for General Code Motion 11
4.1: Basic Structure of the VLIW Pipeline (Single Issue) 13
4.2: Scoreboarded Register File 14
4.3: Example of Result Forwarding 15
4.4: Example of a Four-issue VLIW 16
4.5: Example of Multiple Result Forwarding 17
4.6: Extra Hardware in the In-order Superscalar Processor 19
4.7: Out-of-order Processor Hardware 21
4.8: Instruction Issue Window 21
4.9: Reorder Buffer 23
5.1: Instrumented Program Interfacing with Simulator 25
5.2: Instrumented Program Instruction Structure 26
5.3: Internal Structure Used in the Simulator 27
5.4: Indexed Linear List Queue Structure 29
5.5: Structure of the Simulator 31
5.6: Example of In-order Superscalar Decoding 34
6.1: Comparison of Different Hardware Models 38
6.2: Speedup from Unlimited Function Units 39
6.3: Effects of Smaller Window Sizes 40
6.4: Performance Benefit from Code Scheduling 41
6.5: Effects of Window Size on Code Scheduling Benefit 42
6.6: Effect of Register Renaming on Performance 43
A.1: Flow Chart of the Scheduler 46
A.2: Example Opcode File 47

A.3: Example Specification File 48
A.4: Example Option File 49
A.5: Shell Script to Run Lmips-trace 51

1. INTRODUCTION

Computer performance can be improved in many ways. Improvements in fabrication, layout, and low-level logic design often lead to faster computers. However, high-level design can also make great improvements. One way a design can improve the performance of a computer is to exploit parallelism in program instructions. Some research tries to find course-grained parallelism in programs by concentrating on loop-level parallelism or task-level parallelism. This thesis will concentrate on parallelism at the instruction level. Recent research has shown that many programs exhibit a good deal of parallelism at the instruction level [1].

In this thesis we present three microprocessor models designed to exploit instruction level parallelism. One is compiler based and the other two are hardware based. Chapter 2 gives an overview of of the optimized, trace based compiler. Chapter 3 gives a detailed description of the code scheduling algorithm. Chapter 4 presents the the three hardware models under consideration and some variations of those models. Chapter 5 describes the tool used to evaluate the hardware models through trace-driven event-based simulation. Chapter 6 presents the results of the study of these architectures.

2. OVERVIEW OF IMPACT

The IMPACT architectural framework [2] provides major investigative tools needed to carry out a study of different multiple instruction architectures. IMPACT is a complete compiler that consists of classical optimizations [3]. In addition to these optimizations, profile-based optimizations are also incorporated [4]. IMPACT provides complete support for profile-based decisions.

In addition to optimized scalar code generation, IMPACT also provides code scheduling tools to schedule code for very long instruction word (VLIW) and superscalar architectures. Since this feature is important to evaluating multiple instruction issue architectures, it will be discussed in detail in Chapter 3.

In addition to compiler support, IMPACT provides support for trace generation. Traces can be consumed by the simulator as they are generated. This will be discussed in Chapter 5.

Traditionally, compilers have used basic blocks as their unit of code optimization [3]. Since basic blocks are often small, finding parallel instructions to execute is difficult. IMPACT uses its own larger construct, which is termed the superblock. Superblocks have only one entry point, but may have multiple exit points from conditional branches.

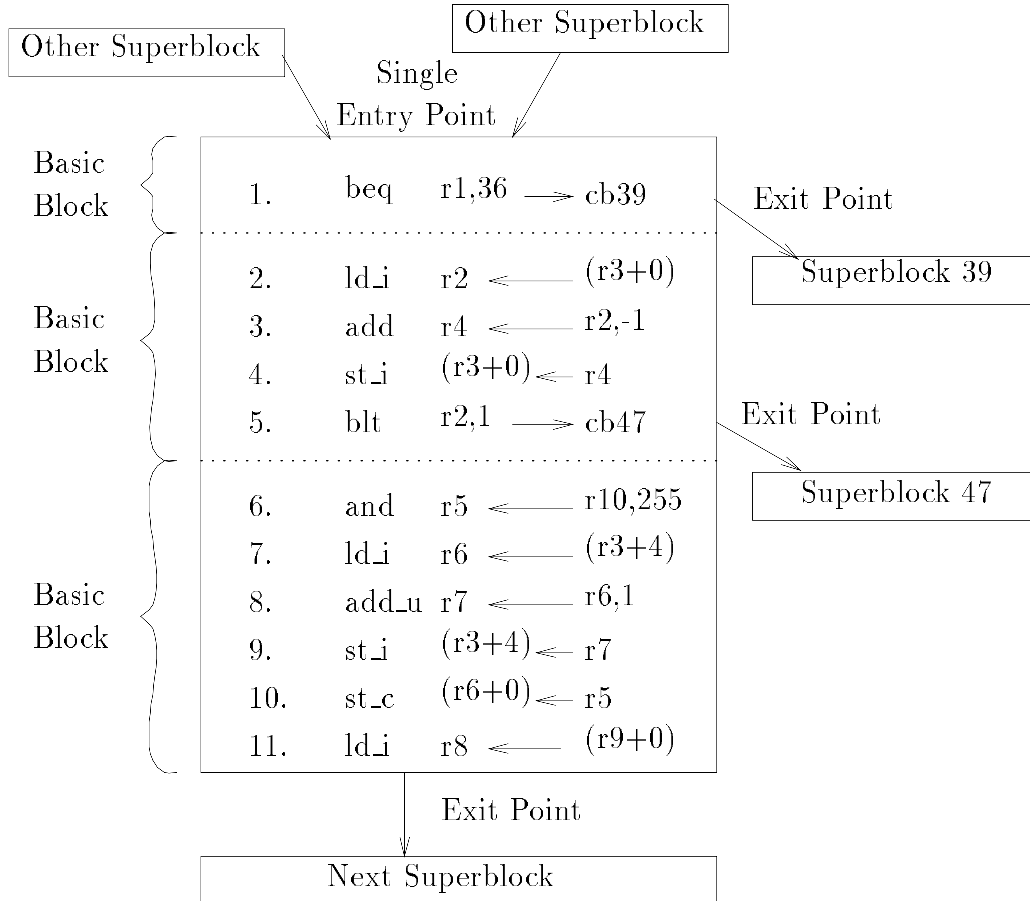

Figure 2.1: Example of a Superblock

The final exit point is always a fall-through path or an unconditional branch. Profile information is used to form superblocks such that the early exit points are more likely to fall through to the next instruction rather than branch to another superblock.

Figure 2.1 shows an example superblock. It consists of three basic blocks put one after another with the only early exit points coming from conditional branches. Superblocks allow a larger selection of code to be scheduled. Scheduling of superblocks will be discussed in the next chapter.

3. CODE SCHEDULING

Instruction scheduling is important for multiple instruction execution processors. It is necessary for VLIW machines and improves performance of superscalar machines. This chapter explains the code scheduling tool used to compile code for superscalar and VLIW machines.

To maximize utilization of the function units of a processor, instructions must be moved around so that they can be issued at the proper time. For instance, on a processor with a separate floating-point unit and integer unit, it is undesirable for the instruction stream to have long runs of only floating-point instructions. During these types of runs, the integer unit will be idle. In this case, it is much better to have integer instructions mixed in with floating-point instructions. The function of the code scheduler is to rearrange instructions to best suit the architecture.

Most scheduling algorithms use a dependence graph to help schedule instructions [5]. For the scheduler, each instruction is a node in the dependence graph. The arcs in the graph are dependences between instructions. A dependence between two instructions means that the first instruction must be executed before the second one to insure correctness of the program. Similarly, no dependence between two instructions means that

they can be executed in any order relative to one another. There are four kinds of dependences in the graph. Given two instructions i and j from an instruction sequence where i occurs before j, a description of these dependences follows:

1. If j uses the result of i, j is said to be *flow dependent* upon i. Sometimes these are called *true dependences*.

2. If the destination of j is the same as the destination of i, j is said to be *output dependent* upon i.

3. If i uses the result of j, j is said to be *anti-dependent* upon i. This type of dependence is the inverse of flow dependence in that the order of the instructions is reversed.

4. If i is a control instruction, j is said to be *control dependent* upon i.

With the exception of control dependences, all dependences can be either memory dependences or register dependences. Therefore, the scheduler has to distinguish between seven types of dependences. Conceptually, the graph is a directed acyclic graph (DAG). However, it is convenient to have arcs going in both directions.

Figure 3.1 shows a DAG for the example code of the superblock that is shown in Figure 2.1. For clarity, the redundant dependences have been removed. As shown in this graph, control dependences cause the graph to be mostly serial. Because node five is a branch, nodes one through four cannot be executed after it, and nodes six through eleven cannot be executed before it. In order to benefit from code motion, the definition of control dependence has to be relaxed. Chang *et al.* have proposed two models that effectively relax code motion requirements [2]. The two models are described below.

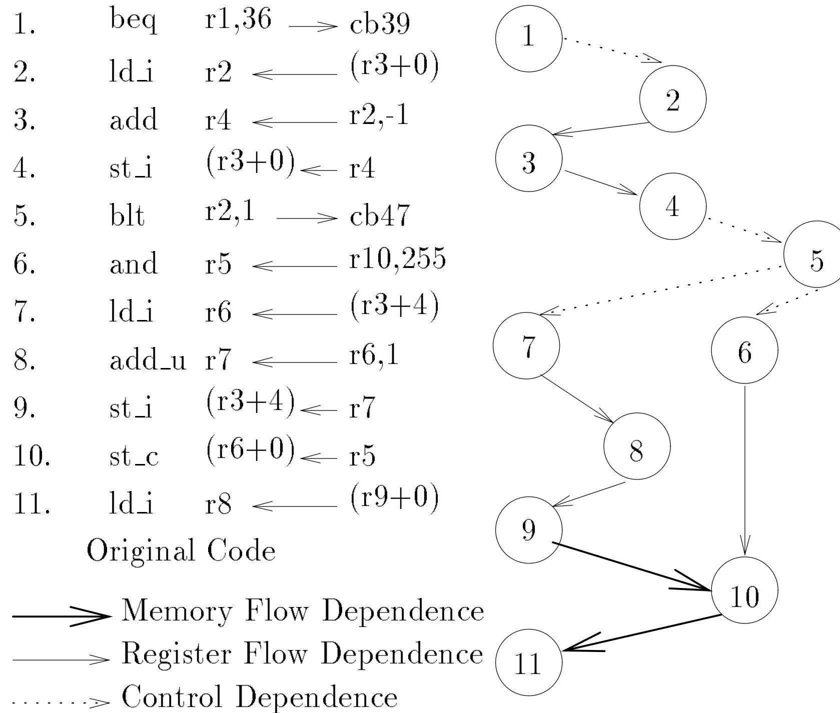

Figure 3.1: DAG of Instructions

Given two instructions i and j where i is a branch instruction, define $live_out(i)$ as the set of variables that could be defined before they are used if i is taken [3]. To move code across a branch two restrictions are put in place:

1. The destination register of j must not be in $live_out(i)$.

2. If the branch i precedes j, j must not cause an exception that could terminate program execution.

Restricted code motion requires that both restrictions are enforced. General code motion only requires that the first restriction is enforced. Special nontrapping support is required for this model. To support these two models, the scheduler removes control dependences from the dependence graph.

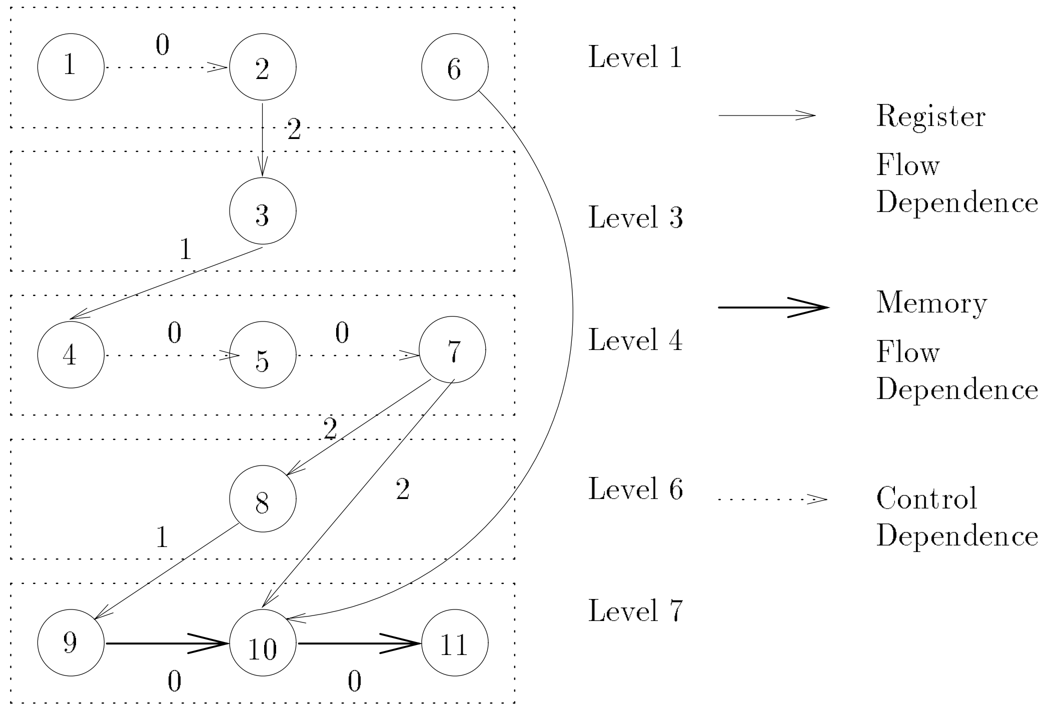

Figure 3.2: Levelized DAG Using Restricted Code Motion

Once the dependence graph has been derived, list scheduling is applied. The first step in list scheduling is to take the DAG and levelized it. Each instruction is assigned a level which indicates when it can be executed relative to the other instructions. Due to some special hardware support, such as squashing branches, there may be dependences among instructions on the same level. Figures 3.2 and 3.3 show the levelized graphs for restricted code motion and general code motion. Note that special hardware for branches and memory operations is implied by these graphs. An instruction that comes after a branch can be executed at the same time as the branch. If the branch is taken, the instruction will be squashed. Memory instructions can be issued simultaneously which assumes support for forwarding the result of a store to a load that was issued at the same cycle.

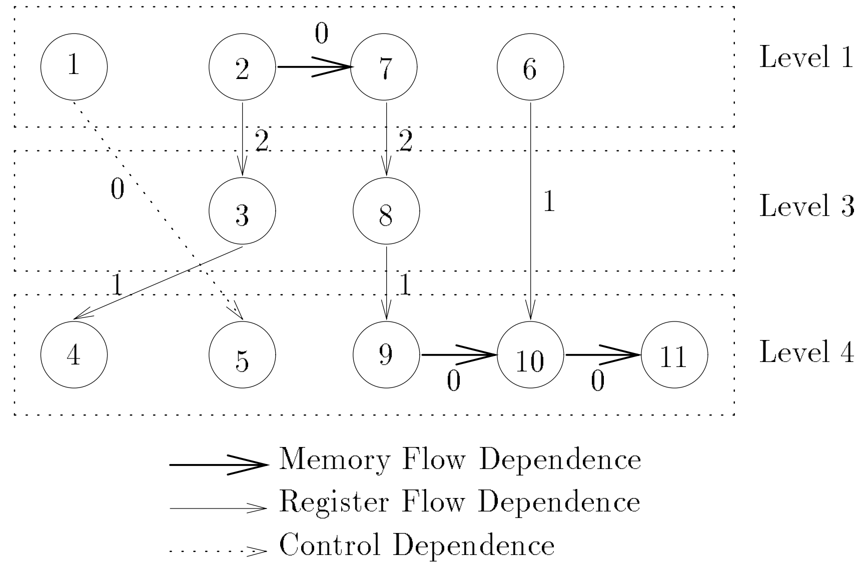

Figure 3.3: Levelized DAG Using General Code Motion

Once a levelized graph is obtained, weights are assigned to each of the nodes in the graph. These weights determine when the instructions are executed relative to the other instructions in a level. Five heuristic measures are used for determining the weights of a given node. They are described in order of importance as follows:

Frequency - How many times this instruction is executed. It is based on profile information.

Slackness - The square of the quantity maximum level minus how late an instruction can be issued. For example, node six in Figure 3.2 has slackness of one because it could be executed at level six. Lower priority implies later issue which shortens register lifetime.

Liveness - Counts the number of source registers and subtracts the number of destination registers. This gives higher priority to registers that use variables and lower priority to those that define them.

Table 3.1: Heuristic Weights of Nodes for Restricted Code Motion

	Instruction	Slackness	Liveness	Uncovering	Weight
	Multiplier →	1,000	1,000	100	
1.	beq r1,36→cb39	6^2	2	10	39,000
2.	ld_i r2←(r3+0)	6^2	1	9	37,900
3.	add r4←r2,-1	4^2	1	8	17,800
4.	st_i (r3+0)←r4	3^2	3	7	9,700
5.	blt r2,1→cb47	3^2	2	6	8,600
6.	and r5←r10,255	0	1	2	1,200
7.	ld_i r6←(r3+4)	3^2	1	4	7,400
8.	add_u r7→r6,1	1^2	1	3	2,300
9.	st_i (r3+4)←r7	0	3	2	3,200
10.	st_c (r6+0)←r5	0	3	1	3,100
11.	ld_i r8←(r9+0)	0	1	0	1,000

Uncovering - Counts the number of dependents of a node. For example, in Figure 3.2 nodes two through ten all depend on node one, giving node one an uncovering value of ten.

Original Order - If the previous measures do not distinguish between two nodes then the original order is used, which prevents ties.

Table 3.1 gives the values of the nodes in the restricted code motion DAG shown in Figure 3.2. Table 3.2 gives the values of the nodes in the general code motion DAG shown in Figure 3.3. Because no code profiling is done on the example code, only slackness, liveness, and uncovering are shown. Each measure has a constant multiplier associated with it that insures that the important measures are counted more.

Once the weights are calculated, the code can be scheduled. From each level the code is placed into instructions according to the weight of the node. Note that in Figure 3.2, node six could be scheduled with nodes three, four, five, or seven. However, it will not be scheduled. The levels are scheduled in order. This is one of the limitations of our

Table 3.2: Heuristic Weights of Nodes for General Code Motion

	Instruction	Slackness	Liveness	Uncovering	Weight
	Multiplier →	1,000	1,000	100	
1.	beq r1,36→cb39	1^2	2	1	3,100
2.	ld_i r2←(r3+0)	3^2	1	7	7,700
3.	add r4←r2,-1	1^2	1	1	2,100
4.	st_i (r3+0)←r4	0	3	0	3,000
5.	blt r2,1→cb47	0	2	0	2,000
6.	and r5←r10,255	1^2	1	2	2,200
7.	ld_i r6←(r3+4)	3^2	1	4	7,400
8.	add_u r7→r6,1	1^2	1	3	2,300
9.	st_i (r3+4)←r7	0	3	2	3,200
10.	st_c (r6+0)←r5	0	3	1	3,100
11.	ld_i r8←(r9+0)	0	1	0	1,000

heuristics. Figure 3.4 shows the code schedule for a dual issue machine. Note that the original eleven instructions can be run in eight cycles, which is 1.4 instructions per cycle. Figure 3.5 shows that general code motion does much better. Eleven instructions are executed in 6 cycles, which is 1.8 instructions per cycle and the best that can be obtained. Both forms of scheduling helped improve the scalar code. The DAG in Figure 3.1 would have a cycle count of nine.

Code scheduling is necessary to help improve performance of machines. Without code scheduling, the amount of instruction parallelism is greatly reduced and potentially useable hardware is unuseable. The next chapter discusses the hardware models.

Instruction 2	Instruction 1	Cycle
2. ld_i r2 ⟵ (r3+0)	1. beq r1,36 ⟶ cb39	1
	6. and r5 ⟵ r10,255	2
	3. add r4 ⟵ r2,-1	3
5. blt r2,1 ⟶ cb47	4. st_i (r3+0) ⟵ r4	4
	7. ld_i r6 ⟵ (r3+4)	5
	8. add_u r7 ⟵ r6,1	6
10. st_c (r6+0) ⟵ r5	9. st_i (r3+4) ⟵ r7	7
	11. ld_i r8 ⟵ (r9+0)	8

Figure 3.4: Schedule for Restricted Code Motion

Instruction 2	Instruction 1	Cycle
7. ld_i r6 ⟵ (r3+4)	2. ld_i r2 ⟵ (r3+0)	1
6. and r5 ⟵ r10,255	1. beq r1,36 ⟶ cb39	2
3. add r4 ⟵ r2,-1	8. add_u r7 ⟵ r6,1	3
5. blt r2,1 ⟶ cb47	4. st_i (r3+0) ⟵ r4	4
10. st_c (r6+0) ⟵ r5	9. st_i (r3+4) ⟵ r7	5
	11. ld_i r8 ⟵ (r9+0)	6

Figure 3.5: Schedule for General Code Motion

4. HARDWARE MODELS

In this thesis, we study three hardware models of execution. We study one VLIW model and two superscalar models. The VLIW model is a minimal hardware approach whereas the superscalar models require that the hardware do most of the work. Different instruction issue rates and register renaming are studied.

4.1 The VLIW Hardware Model

The VLIW hardware model is based upon the the IMPACT architectural model developed by Chang *et al.* [2]. Since the compiler does most of the work, we assume minimal, yet aggressive, hardware. There is support for out-of-order completion of instructions in the scoreboarded register file, but there is no support for out-of-order issue of instructions. All of the function units have results forwarding and are symmetric.

Figure 4.1 shows the basic structure of the single issue pipeline. Instructions are fetched from memory and latched in the first stage. The decode stage is responsible for fetching operands from either memory or the function units and latching them. To limit the number of branch delay slots to one, branching is also done in the decode stage. The

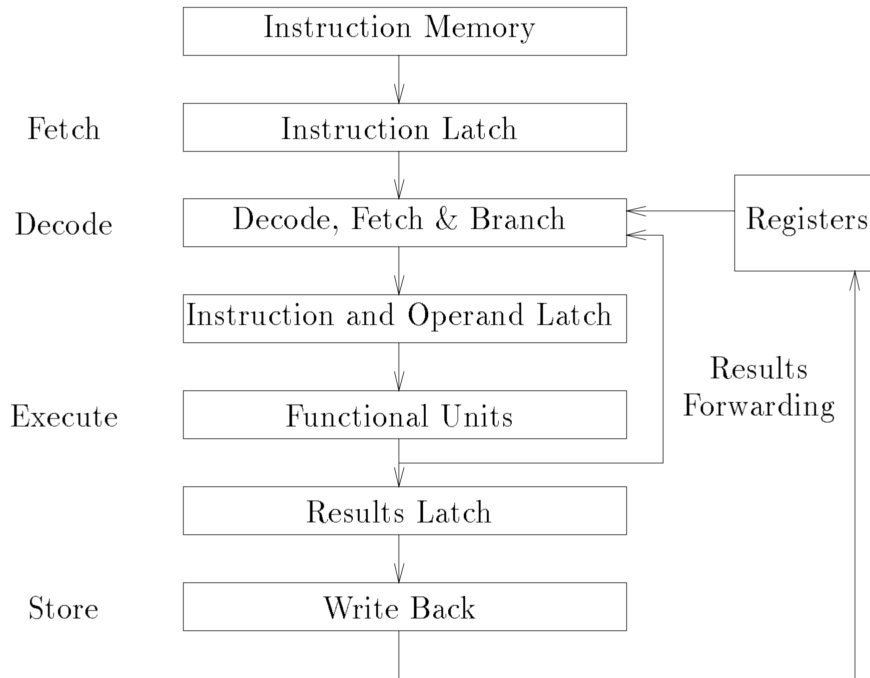

Figure 4.1: Basic Structure of the VLIW Pipeline (Single Issue)

execute stage consists of the function units which will be discussed later. The results from the function units are latched at the store stage.

The register file for the VLIW model is scoreboarded. Figure 4.2 shows a scoreboarded register file. In addition to the data that all registers contain, there is a valid bit indicated by V in the figure. The bit is set when an instruction uses the register for its destination. If this bit is set, the register can be used for a source in the instruction. If it is not set, then the processor stalls instruction issue until the result is ready.

The possibility also exists that the source of an instruction has not been written to the register file, but is available at the same time that another instruction requires it. Result forwarding allows the currently decoded instruction to use a result that was just generated. Figure 4.3 has an example of this. Instruction one is being processed in the ALU, and instruction two uses the results of instruction one. Without result forwarding,

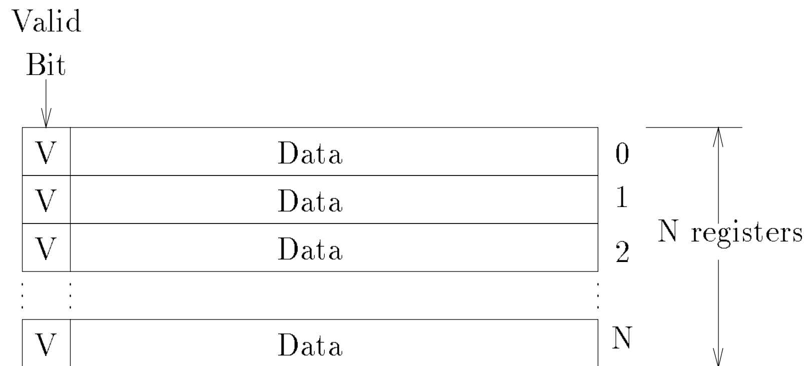

Figure 4.2: Scoreboarded Register File

instruction two would have to wait until instruction one had written the result in the register. This effectively adds a one-cycle delay between instructions one and two. With the addition of the compare circuits and the muxes, the results from instruction one can be forwarded to instruction two in the same cycle. This allows many instructions to be a single cycle instead of two cycles.

The difference between a VLIW and the processor we have described above is that a VLIW can issue multiple instructions at once. Since the compiler automatically makes sure that individual instructions are packed into large instructions, the VLIW machine is basically many single issue pipelines with the features described above. However, there are a few key differences. One is that the register file must be able to supply all of the source operands that the multiple issue needs and it must be able to write as many results as the machine can produce per clock cycle. Since branches are common in integer codes (often called control intensive because of their large number of branches), it should be possible to issue multiple branches in the same instruction. It is impossible to execute multiple taken branches. Therefore, a mechanism for ordering the branches must be provided. Result forwarding must also accommodate multiple results being retired simultaneously.

1. add r1 $\leftarrow$ r2,r4
2. add r5 $\leftarrow$ r1,r3

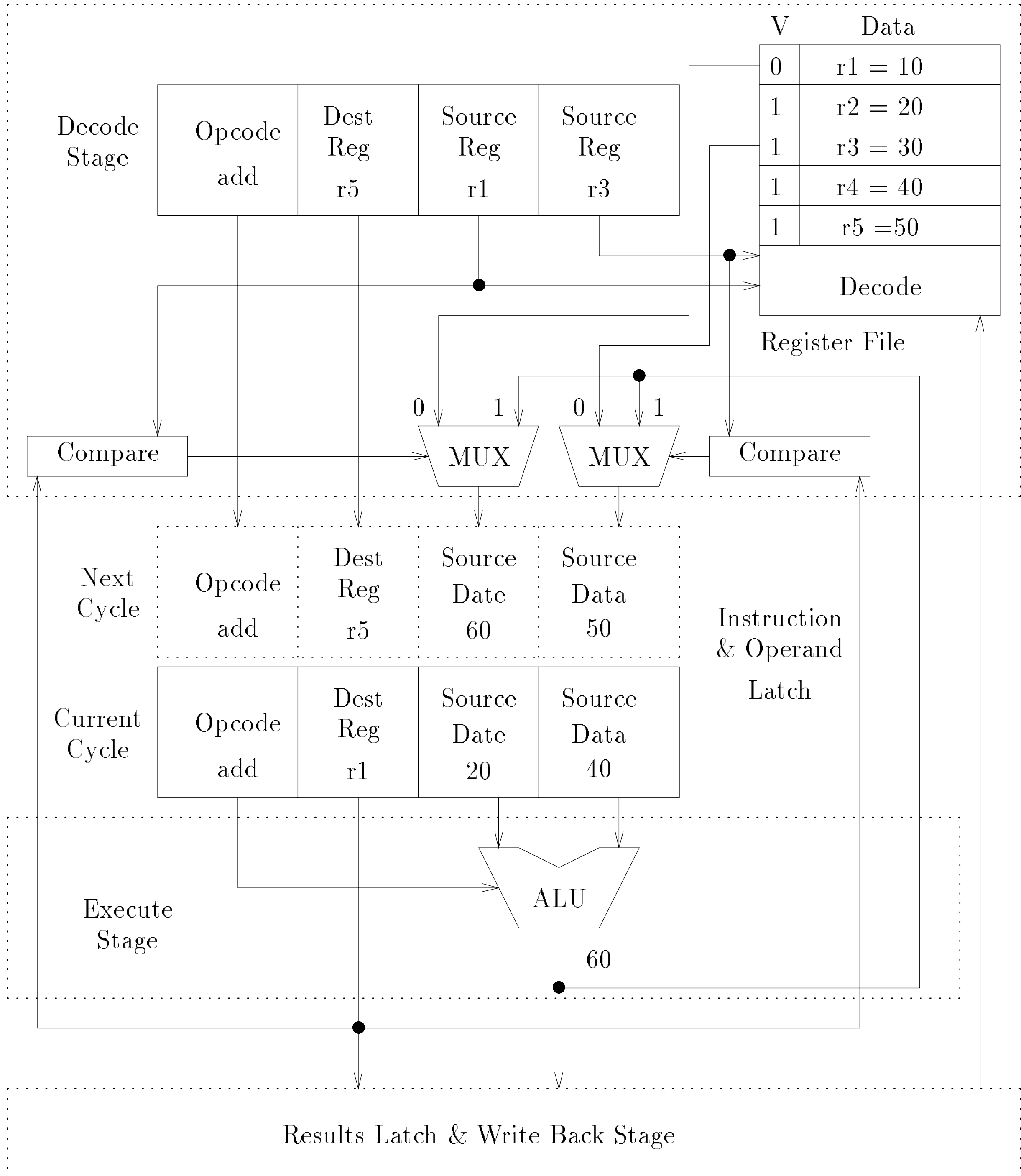

Figure 4.3: Example of Result Forwarding

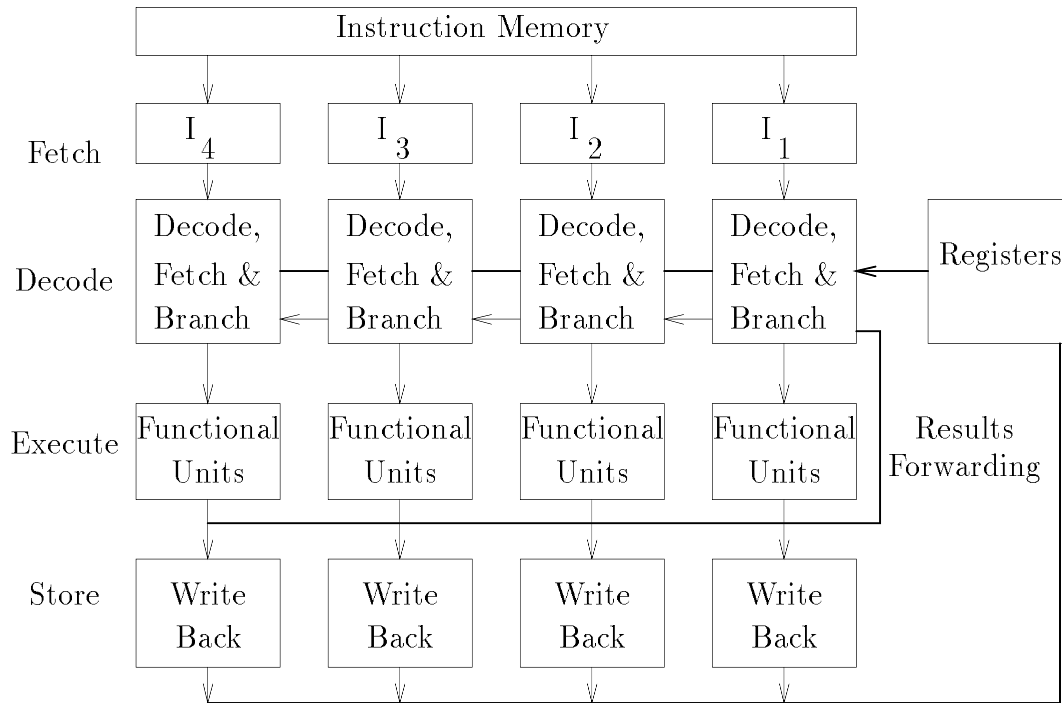

Figure 4.4: Example of a Four-issue VLIW

Figure 4.4 has an example of a four-issue VLIW machine. Each decoder requires a port to the register file indicated by the thick line going between the Decode, Fetch, and Branch boxes. To accommodate multiple branches, a taken branch squashes the instructions in the same word that would be executed sequentially after it. For example, if I_2 is a taken branch then I_3 and I_4 are squashed. The squashing line is indicated by the thin line with the arrow pointing left. Multiple result forwarding is not shown in the figure, but is discussed next.

If multiple function units execute concurrently, many results for many destinations may be generated at the same time. This requires that each source look at all of the results and make sure they do not include the value that the source requires. Figure 4.5 shows an example of multiple functional units. The machine generates four results at once, and the destination of each one is compared with each source of the next instruction.

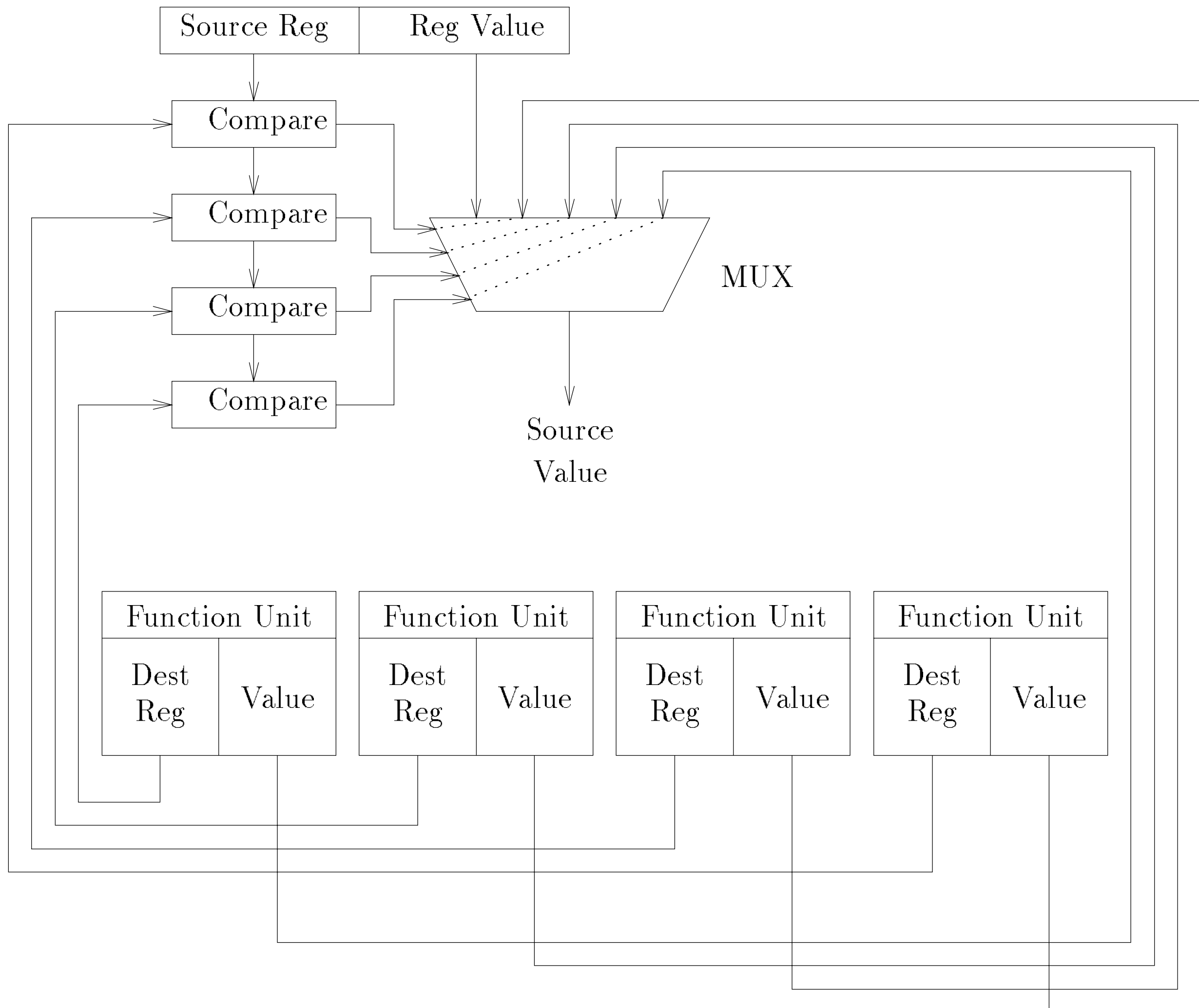

Figure 4.5: Example of Multiple Result Forwarding

The figure shows only one source. None or one of the destination registers can match the source. If no destination registers match, then the source value is selected. If the source value is selected and the valid bit is not set, the pipeline stalls and waits for the results to become available. If one of them matches, then it is selected. For clarity, a dashed line shows which value the control line on the mux selects.

The hardware to implement a VLIW processor is relatively simple. It is basically repeating the hardware of a single issue processor. Some of the features have to be expanded, but that is possible. However, a VLIW processor requires compilation for each different issue rate. Code compiled for a dual issue will not run on a four-issue and vice-versa. Recompilation is often undesirable. The next section describes a simple superscalar processor which eliminates the need to compile long instructions.

4.2 Superscalar In-order Hardware Model

The superscalar in-order model is very similar to the VLIW model except that dependencies between instructions must be detected by the hardware. The single issue pipeline of the in-order model and the VLIW single issue are identical. The register file and results forwarding are identical as are the execute and store stages. Only the decode logic for the multiple instruction issue is a little more complex.

In the VLIW model the instructions fetched from memory are guaranteed to be independent. In the in-order superscalar model, the instructions are fetched and then the analysis is done to see which instructions can proceed. Independent instructions are allowed to proceed while the ones that cannot proceed are refetched on the next cycle along with additional instructions.

Figure 4.6 shows the modification to the hardware necessary to make the VLIW processor of Figure 4.4 into a superscalar processor. This hardware makes sure that

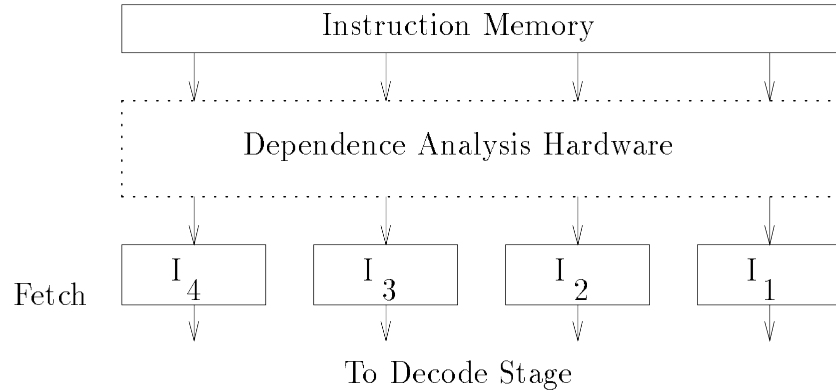

Figure 4.6: Extra Hardware in the In-order Superscalar Processor

no instruction depends on any previous instruction in the buffer. In addition to the scoreboard provided in the register file, the decode stage in this mode also has to take into account resource constraints that the VLIW hardware does not. (The compiler insures that these do no exist in the VLIW model.) For example, if three floating-point instructions are fetched, but only two units exist, fetching must be halted until at least one of the instructions can be issued into the floating-point pipelines. Resource interlock hardware and more complex fetching hardware free the compiler from having to resolve these conflicts.

4.3 Superscalar Out-of-order Hardware Model

The most aggressive hardware presented is the superscalar out-of-order execution model. Often, the words "superscalar processor" are used to refer to this processor model. In this model, instructions are fetched and are allowed to issue as long as their operands are available and there is a unit available to execute them. This requires the hardware to keep track of this information dynamically. In effect, the dynamic instruction stream

is scheduled by the processor. The advantage of this approach is that it maximizes the number of functional units that are busy.

As one might expect, the hardware support for this model of computation is more complex than for the in-order models. The hardware must keep track of the in-order execution pattern of the program while trying to keep the function units busy. There are many models and implementations of out-of-order issue of instructions. As early as 1967, IBM was using out-of-order issue for the floating-point pipeline in the 360-91 [6, 7]. Keller provides abstract models for out-of-order execution [8]. Hwu demonstrates the feasibility of out-of-order design of the complete HPSm single-chip micro-architecture [9]. The model used in this thesis most closely resembles the work of Johnson [10] who uses a reorder buffer [11].

In each cycle, the number of instructions fetched equals the issue rate. Unlike the previous hardware models, these instruction do not have to be independent. On the next cycle, these instructions are assigned tags and placed in the issue window. The issue window buffers instructions until they can issue. In the following cycle, instructions in the window that can execute are issued to the function units. For an instruction to issue, a function unit must be available, and the instruction's sources must be available. The results from the function units are forwarded to issue logic, and results are written into the register file and the reorder buffer. The reorder buffer is responsible for preserving correctness of the results (in terms of the sequential code) and allowing for exceptions. Figure 4.7 shows the block diagram for a four-issue out-of-order issue processor.

The instruction window is shown in Figure 4.8. When an instruction is placed into this window, the fields are filled. The decode logic figures out the type of function unit the instruction will run on and places that in the function unit field. The reorder buffer (explained below) assigns the destination tag. If the source value is available, the valid

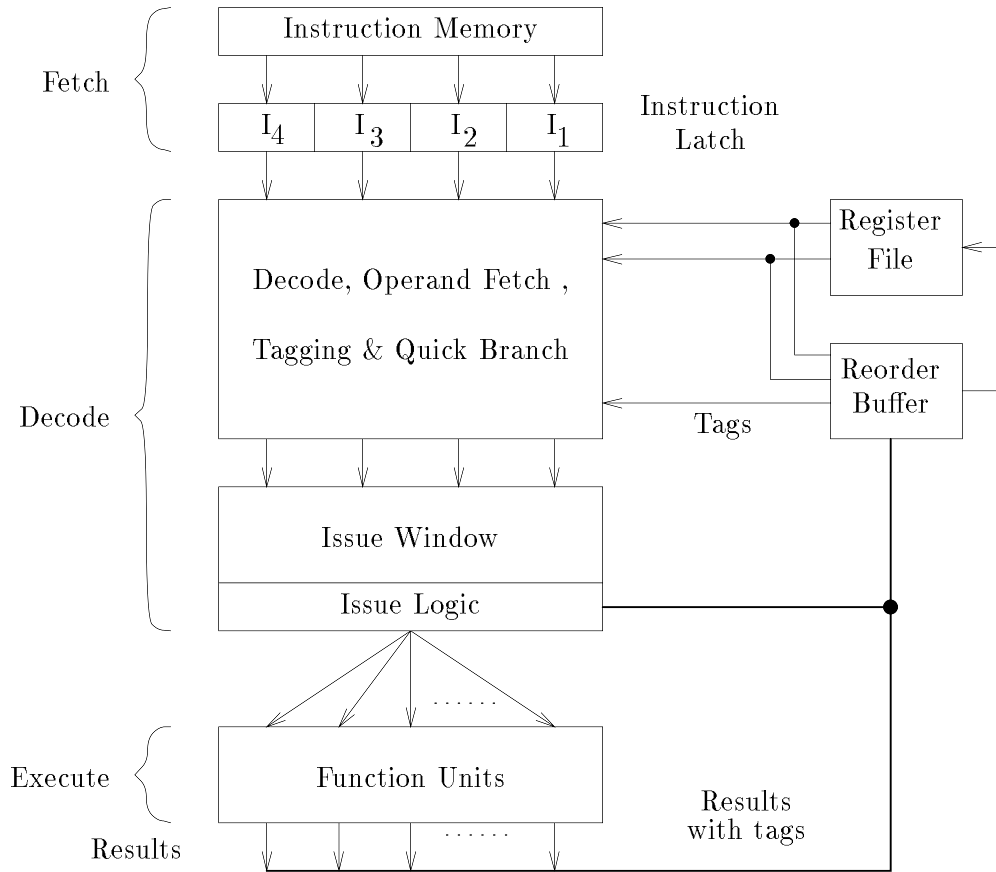

Figure 4.7: Out-of-order Processor Hardware

Function Unit	Dest. Tag	Source1			Source2		
		Tag	Value	Valid	Tag	Value	Valid
: :	: :	: :	: :	: :	: :	: :	: :

Figure 4.8: Instruction Issue Window

bit is set, and the value is moved into the valid field. The tag does not have to be assigned. If the source is not available, then the reorder buffer assigns the tag for that source, and the valid bit is not set.

At each cycle, the issue window checks which instructions are ready to fire by ensuring that both source valid bits are set, and then assigns them to function units. If too many instructions need the same function units, then the later instructions are not fired. The distance from the head of the buffer is used to determine which instructions fire first. The shorter the distance, the earlier the instruction fires.

At the end of each instruction, tags of results are sent to the issue window. These tags are compared to all of the sources that have valid bits that are not set. If the tags match, the value is placed in the source value field. This is similar to the write-back stage in the sequential pipeline. Since this effectively adds an extra stage to the pipeline, result forwarding is used in this model as well. This works by giving the issue logic the tags of results that will be available at the end of the cycle. The instructions with these tags are now valid and can be fired on the next cycle. When they fire, the forwarding hardware places the results in the source field.

Much of the function of the issue window depends upon the reorder buffer. Figure 4.9 shows the structure of the reorder buffer, which is a circular queue with a head and a tail pointer. When instructions are decoded, they are each assigned a tag based upon the tail pointer. If four instructions are issued, they would get destination tags, five, six, seven, and eight and the tail pointer would be moved to tag eight. The destination register number and program counter are saved. When the destination tags return from the function units, the valid bit is set and the exception bit is set appropriately. If the head entry is valid and no exception has occurred, then its result is sent to the register file, and the head pointer is moved one position closer to the tail pointer. If an exception

	Entry Number (Tag)	Destination Register	Result Value	Valid Bit	Exception Bit	Program Counter
	1					
Head→	2					
	3					
Tail→	4					
	:	:	:	:	:	:
	:	:	:	:	:	:
	N					

Figure 4.9: Reorder Buffer

has occurred, then the results in the reorder buffer are flushed and the exception routine is called with the return point as the program counter field of the instruction.

To make sure that memory is consistent, all stores are held in their pipeline until all of the preceding instructions are known to be free of exceptions. Effectively, stores are held in the reorder buffer until they reach the head, at which time, they are actually sent to memory. Loads are not constrained this way since they do not modify the state. However, loads must check the reorder buffer to see if the address that they are loading has been stored recently.

4.4 Register Renaming

Keller's abstract model of processors defines register renaming [8]. Register renaming eliminates anti-dependences and output dependences. The out-of-order model has register renaming already in the form of tags. The in-order superscalar and VLIW models could also use tags which point to a new register. Each destination referring to the original register would get the value associated with the new tagged register. This model would allow multiple instructions with output dependences or anti-dependences to be

issued in the same cycle. However, this would complicate the relatively simple hardware of the two in-order models.

4.5 Branch Prediction

Branches occur frequently in computer programs. They occur more frequently (in terms of cycles) when more instructions can be issued in a single cycle. Unconditional branches do not pose a problem because the fetching of the next instruction can start from the destination of the unconditional branch with no problem. Conditional branches are a problem because the hardware has to wait until the direction the branch takes is known in order to discern which instruction to fetch.

To minimize this problem, the IMPACT model has two types of conditional branches: normal and forward semantic. When a normal conditional branch is fetched, the next instruction fetched is the one which follows the branch in the code. When a forward semantic branch is fetched, the next instruction is fetched from the destination of the branch. The compiler decides which type of branch should be used. Our out-of-order hardware model allows as many outstanding conditional branches as can fit in the reorder buffer.

Other methods which can be used for branch prediction include branch target buffers [12]. These buffers use past information of branch behavior to predict future branch behavior. Hwu *et al.* show that software methods can perform better than the hardware method [13].

5. SIMULATOR

To evaluate the different machine models, a discrete event-driven simulator was constructed. The simulator consumes an instruction trace. To save the large amounts of storage space needed to store instruction traces, the program under investigation is instrumented to provide the address trace as it runs. The current instruction is passed to the simulator through a normal subroutine call. Conceptually, this is equivalent to the original program sending a message to the simulator. Realistically, the simulator is only another group of subroutines and functions. Figure 5.1 shows this system. This system requires that the simulator routines be self-contained. None of the variables or

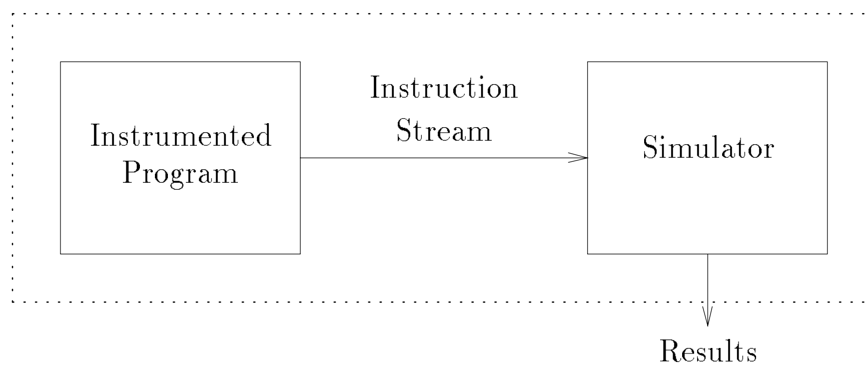

Figure 5.1: Instrumented Program Interfacing with Simulator

```
struct lx_Record {
  int  pe_id;       /* process id */
  int  fn_id;       /* function id */
  int  cb_id;       /* control block id */
  int  in_id;       /* instruction id */
  int  op_id;       /* operation id */
  int  opcode;      /* opcode */
  int  src1;        /* source 1 register */
  int  src2;        /* source 2 register */
  int  src3;        /* source 3 register */
  int  dest;        /* destination register */
  int  value;       /* cc / memory address */
  int  begin_addr;  /* starting address of operation */
  int  end_addr;    /* end address of operation */
} ;
```

Figure 5.2: Instrumented Program Instruction Structure

the subroutines can interfere with original program execution. In addition, none of the usual command line options are available to the simulator since the original program already uses them.

Each instruction has certain fields associated with it. The structure that the instrumented program sends to the simulator is shown in Figure 5.2. The simulator does not use all of these fields. The structure that the simulator uses is shown in Figure 5.3.

The simulator is a versatile tool that can be easily configured for different machines. The simulator can be reconfigured at run time through a file named PARFILE. The syntax for an entry in the PARFILE is "**option** VALUE" on its own line. Each option has default value. The list of options follows:

```
struct qu_struct{
  int issue_time;  /* Time the instruction will fire. */
  int comp_time;   /* Time the instruction will finish. */
  int src1;        /* Source 1 register. */
  int src2;        /* Source 2 register. */
  int src3;        /* Source 3 register. */
  int dest;        /* Destination register. */
  int type;        /* Execution unit type. */
  int address;     /* cc / memory address */
  int delay;       /* Delay of the instruction */
};
```

Figure 5.3: Internal Structure Used in the Simulator

issue_rate - The maximum number of instructions from memory that are fetched and decoded each cycle. Fewer instructions might actually be fetched due to dependencies between instructions. This variable is not used in VLIW mode since the long instructions are defined by the compiler.

model - Either VLIW or SUPERSCALAR. VLIW mode uses the compiler-generated information to issue instructions. SUPERSCALAR mode does run time analysis of the instructions and does not use the compiler-generated information.

register_renaming - Either YES or NO. YES implies that the machine has register renaming between stages of the pipeline.

rename_within_window - Either YES or NO. YES implies that instructions fetched into the issue buffer are renamed as well as between stages of the pipeline. Output dependencies do not stall the fetch on a superscalar machine. NO implies that output dependencies do stall the fetch.

dynamic_tagging - Either YES or NO. This parameter is used when the variable **hardware_schedule** is TOMASULO (see below). YES means always fetch as many as is allowed by the **issue_rate** variable even if there are dependencies between the instructions. The instructions will be automatically tagged. If this is NO, then only fetch until a dependence is found between an instruction in the issue buffer and the one fetched or until the issue buffer is full.

hardware_schedule - Either SCOREBOARD or TOMASULO. SCOREBOARD implies in-order issue but out-of-order completion of instructions. TOMASULO implies out-of-order issue and completion.

extra_delay - Any positive number. Some features require that an extra cycle be inserted for each instruction. This can be true of reservation stations [14].

max_qu - This is the maximum number of instructions that can be waiting to fire at one time.

resource_contraints - If this variable is YES, then user-defined resource constraints are enforced. Otherwise, as many instructions as possible are sent from the issue buffer to the execution units.

num_br_units - The maximum number of branches that can be fired at one time. **max_qu** is not affected by this.

num_alu_units - The maximum number of ALU instructions that can be fired at one time.

num_fp_units - The maximum number of floating-point instructions that can be fired at one time.

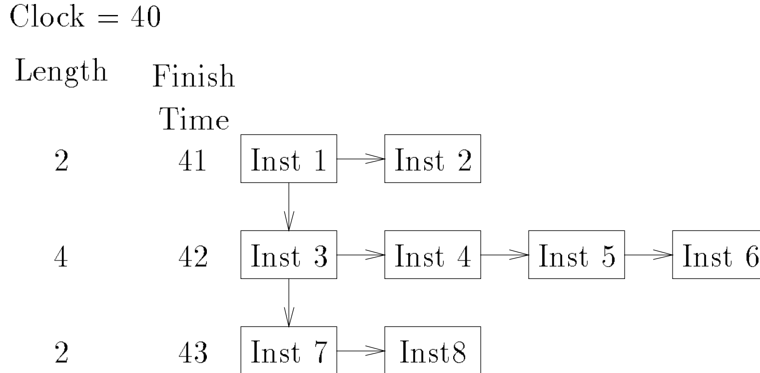

Figure 5.4: Indexed Linear List Queue Structure

num_mem_units - The maximum number of memory operations that can be fired at one time and includes both loads and stores.

num_dest - The maximum number of instructions that can write to the register file at one cycle. Effectively, this is the number of write ports to the register file.

Certain logical constraints are placed on the possible combinations of parameters in the simulator. If parameter **hardware_schedule** is set to TOMASULO, then **register_renaming** is automatically turned on. Likewise, **rename_within_window** set to YES also sets **register_renaming** to YES. The parameter **dynamic_tagging** automatically sets **rename_within_window** to YES and **hardware_schedule** to TOMASULO.

The discrete-event simulator is trace driven. The event queue consists of instructions waiting to be completed. Since the number of instructions in the queue is of moderate size, the queueing strategy used is an indexed linear list [15]. Figure 5.4 shows the structure of the queue. The actual implementation usesa two-dimensional array of instructions that wraps around in the form of a timing wheel. Each level of the queue represents the time at which an instruction finishes. Each level has a counter associated with it that tells how many instructions are at that level. When the clock is greater than or equal

to the finish time of a level, that level is deleted. The clock is advanced in the unit time approach [15]. However, certain special cases cause the clock to be more than one cycle.

The simulator has a basic structure which consists of fetching instructions, decoding, and executing. This structure mirrors the structure of a hardware pipeline. Figure 5.5 shows the structure of the simulator. The trace is sent to the fetch routine which places instructions in the issue buffer. The decode routine takes instructions from the issue buffer and places them in the instruction queue. The execute phase updates the clock and removes instructions from the issue buffer. The algorithms at each stage vary depending on the kind of machine simulated.

The specific simulations of VLIW, superscalar inorder issue, and superscalar out-of-order issue are discussed below.

5.1 Simulation of a VLIW Architecture

The VLIW simulator treats many instructions as a single long instruction. The latest issue time found among the individual instructions that make up the long instruction determines when all of them issue. The compiler determines which instructions can be issued together. The simulator is broken down into three sections: fetch, decode, execute. These three sections are discussed below.

5.1.1 Fetch stage

The fetch stage for a VLIW simulation is quite simple. Even though only one simple instruction is generated by the trace generator, each instruction carries a tag which indicates its long instruction.. In the fetch routine, the program continues to receive instructions and places them into the instruction buffer until the tag changes. The

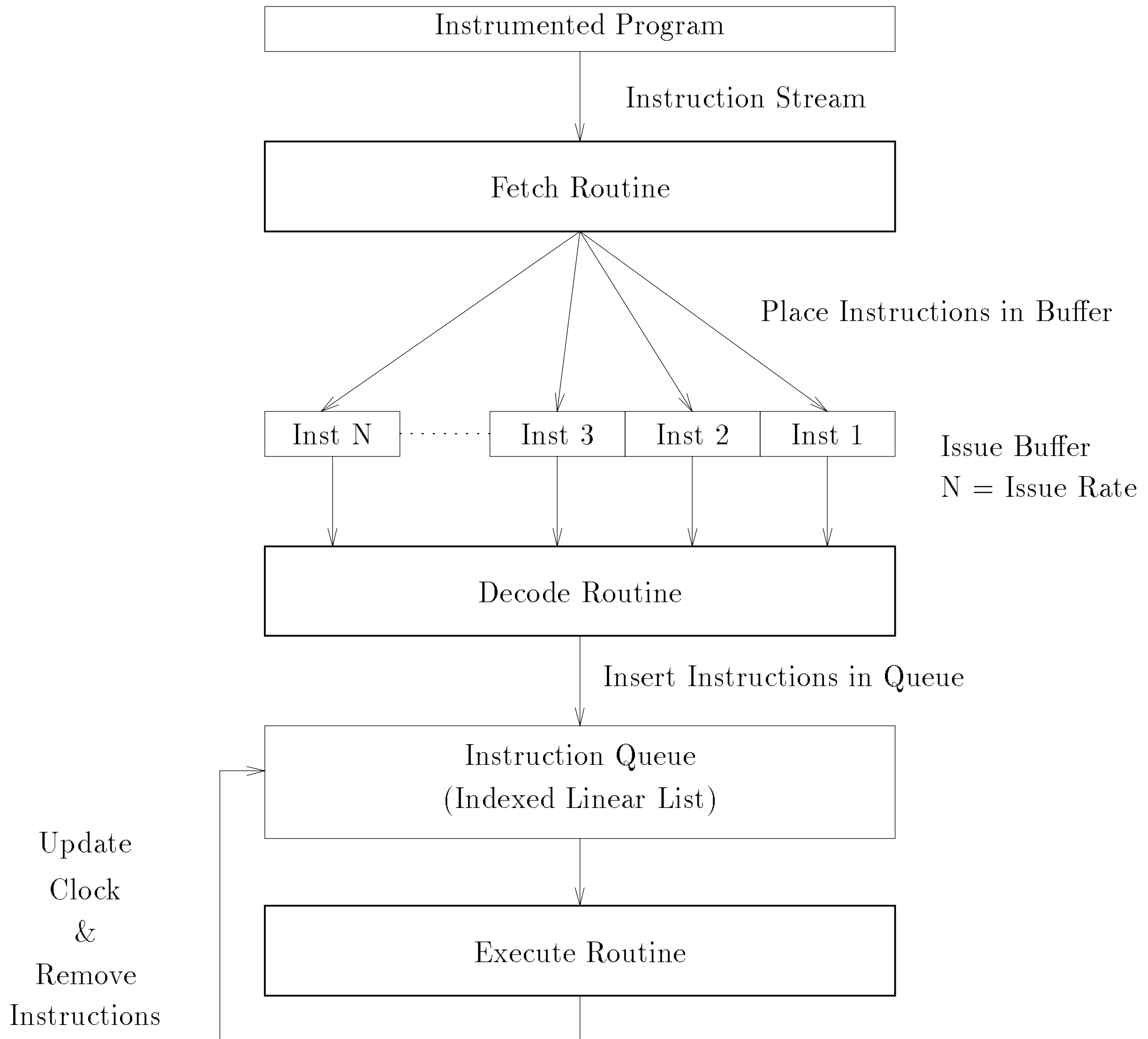

Figure 5.5: Structure of the Simulator

simple instruction that causes the tag to change is saved and added to the instruction buffer in the next clock cycle. Then the decode routine is called.

5.1.2 Decode stage

The decode stage finds dependencies between each instruction in the instruction buffer and instructions in the the instruction queue. The earliest time that each instruction in the instruction buffer can fire is found. The maximum of these times is taken to be the time that all of the instructions in the long instruction can fire. Data dependencies and resource constraints are detected in this stage.

5.1.3 Execute stage

The execute stage advances the clock and removes the instructions from the queue that have been completed on that clock cycle. Normally, the clock is advanced by one. However, under the following three conditions the clock can be advanced by more than one:

1. If either a resource constraint or **hardware_schedule** equals SCOREBOARD (which it usually does when **model** equals VLIW) and a data dependence is detected, the issue stalls and no instructions can be issued until the current instruction has fired. Therefore, the clock is fixed to be the issue time of the current instruction.

2. If the actual size of the queue becomes greater than the variable **max_qu**, the clock is changed to be the completion time of the first instruction in the queue that is less than **max_qu** away from the end of the queue.

3. If a mispredicted branch is encountered, the clock is changed to be the completion time of that branch.

The model for superscalar in-order issue is similar to that for the VLIW model and is described in the next section.

5.2 Simulation of a Superscalar In-order Issue Architecture

In the case of superscalar architectures, the information provided by the compiler is not taken into account. All the dependencies among instructions must be done by the hardware. Our simulator has to support these features. As before, the simulation is executed in the fetch, decode, and execute stages. The in-order model implies that **hardware_schedule** equals SCOREBOARD.

5.2.1 Fetch stage

The fetch stage for a superscalar machine is more complex than the VLIW fetch. Since everything is done by hardware, dependence analysis is done among instructions in the fetch buffer. In the fetch routines, the program gathers instructions until either the issue rate is reached or a dependence is found between the current instruction and one already in the issue window. If the latter is true, the current instruction is saved for issue on the next cycle.

5.2.2 Decode stage

The decode stage is a little different from that of the VLIW simulation. The earliest time that each instruction in the instruction buffer can fire is found. However, the maximum issue time to fire all of the instructions is no longer used. We use a local maximum. Since issue is in-order, each instruction in the fetch buffer must fire at least at the same time as the previous instruction in the buffer. For example, in Figure 5.6, instructions two and four could be issued before instructions one and three, respectively.

Clock = 1033	Instruction Fetch Buffer 4	3	2	1
Possible Issue Time	1034	1035	1033	1034
Inorder Issue Time	1035	1035	1034	1034

Figure 5.6: Example of In-order Superscalar Decoding

However, since this is an in-order model, two and four can only be issued at the same time as instructions three and one. As is true in the VLIW model, data dependencies and resource constraints are detected in this stage.

5.2.3 Execute stage

The execute advances the clock and removes the instructions from the queue that have been completed on that clock cycle. Normally, the clock is advanced by one. However, under the following three conditions the clock can be advanced by more than one:

1. If a data dependence or a resource constraint is detected, the issue stalls and no instructions can be issued until the current instruction has fired. Therefore, the clock is fixed to be the issue time of the current instruction.

2. If the actual size of the queue becomes greater than the variable **max_qu**, the clock is fixed to the completion time of the first instructions in the queue that are less than **max_qu** away from the end of the queue.

3. If a mispredicted branch is encountered, the clock is fixed to be the completion time of that branch.

The superscalar out-of-order issue is one step beyond the in-order issue. It is discussed in the next section.

5.3 Simulation of a Superscalar Out-of-Order Issue Architecture

This mode is turned on by having **hardware_schedule** equal to TOMASULO and **model** equal to SUPERSCALAR. As before, the hardware simulation has to support resource constraints and data dependencies among instructions. The fetch, decode, and execute stages are described below.

5.3.1 Fetch stage

As was the case for the in-order issue, the program gathers instructions until either the issue rate is reached or a dependence is found. The variable **rename_within_window** determines whether or not dependence information is used. If **rename_within_window** equals NO and a dependence is found between the current instruction and one already in the issue buffer, the current instruction is saved for issue on the next cycle. If **rename_within_window** equals YES, then only the number fetched is considered.

5.3.2 Decode stage

As in the VLIW and in-order models, the earliest time that each instruction in the instruction buffer can fire is found. However, we no longer use the maximum to fire the instructions. For example, in Figure 5.6, the possible issue times are these times the instructions actually fire. As is true in the other models, data dependencies and resource constraints are detected in this stage.

5.3.3 Execute stage

The execute advances the clock and removes the instructions from the queue that have been completed on that clock cycle. Normally, the clock is advanced by one. However, under the following three conditions the clock can be advanced by more than one:

1. If the actual size of the queue becomes greater than the variable **max_qu**, the clock is changed to be the completion time of the first instruction in the queue that is less than **max_qu** away from the end of the queue.

2. If a mispredicted branch is encountered, the clock is fixed to be the completion time of that branch.

Note that the processor no longer stalls for data dependencies or resource constraints. The instructions with these problems do issue later, but other instructions continue to be issued.

There is one other model that can be used, but is not presented in this thesis. That model is the out-of-order VLIW model. It is possible to set up the simulator to run this. It would behave like a VLIW in the fetch stage and an out-of-order machine in the decode and branch stages. In addition, register renaming can be turned on for the scoreboarded models of execution.

6. RESULTS

The simulator is used in evaluating the various hardware models. The eleven benchmarks used in evaluation are shown in Table 6.1. They are all common Unix programs which execute from 3,000,000 to 100 million instructions. Different issue rates are studied along with the general and restricted code motion. For example, a four-issue VLIW machine will have four integer units, four branch units, and four memory units. A two-issue VLIW would only have two of each type of unit. In order to do a fair comparison of the machine models, the in-order superscalar model and the out-of-order superscalar models have the same number as the VLIW model. In addition to this, different scheduling is applied to each different issue rate.

The two code motion models are studied because many of the past processors have assumed the restricted model. The general model requires special instructions that do not trap on exceptions [2]. The out-of-order model can execute instructions before branches because it has hardware to squash instructions for a mispredicted branch.

Figure 6.1 shows the results of running the eleven Unix benchmarks. The harmonic means of the speedups of each benchmark are given. For both the VLIW model and in-order superscalar models, general code motion greatly out performs restricted code

Table 6.1: Benchmarks

Benchmark	Description
cccp	GNU C preprocessor
cmp	Unix file comparison utility
compress	Unix file compression utility
eqn	Equation typesetter for TROFF
eqntott	Boolean minimizaation
espresso	Logic minimization
grep	String search in files
lex	Lexical analysis program generator
tbl	Table typesetter for TROFF
wc	Word counting
yacc	Parsing program generator

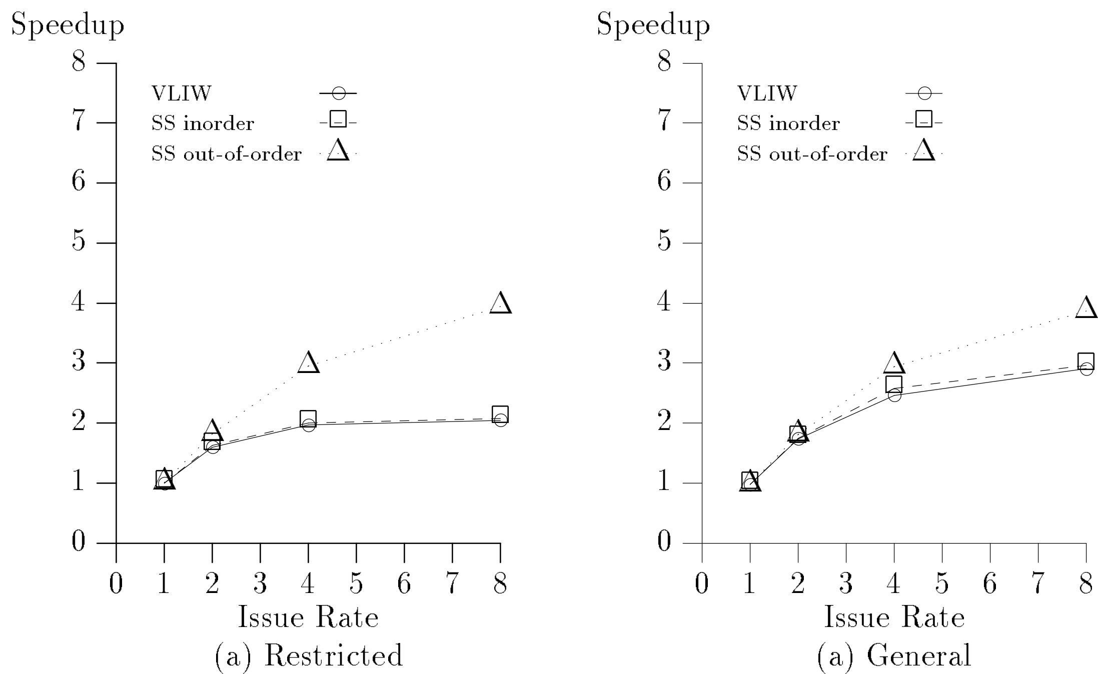

Figure 6.1: Comparison of Different Hardware Models

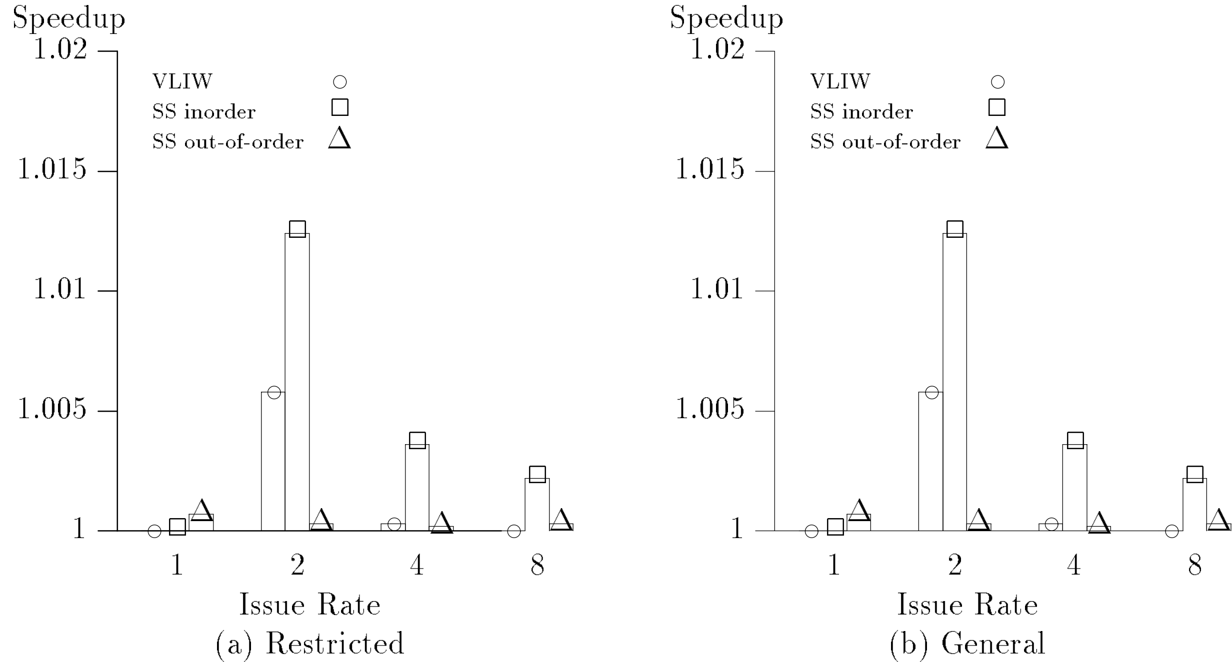

Figure 6.2: Speedup from Unlimited Function Units

motion. This confirms the results of Chang *et al.* [2]. The out-of-order model outperforms both models, but performs equally well for both code motion models. Out-of-order is the fastest because it has the advantage of being able to schedule the dynamic instruction stream. Both the VLIW and in-order models have to rely on static scheduling which, for example, restricts itself to superblocks. The out-of-order model can schedule across superblock boundaries.

6.1 Resource Constraints

The hardware model presented in the previous section assumed that the number of function units was equal to the issue rate. To make sure that this is not a bottleneck, we ran the same programs without constraints on function units. Figure 6.2 shows the results. Unlimited function units do not increase performance significantly. In fact, other

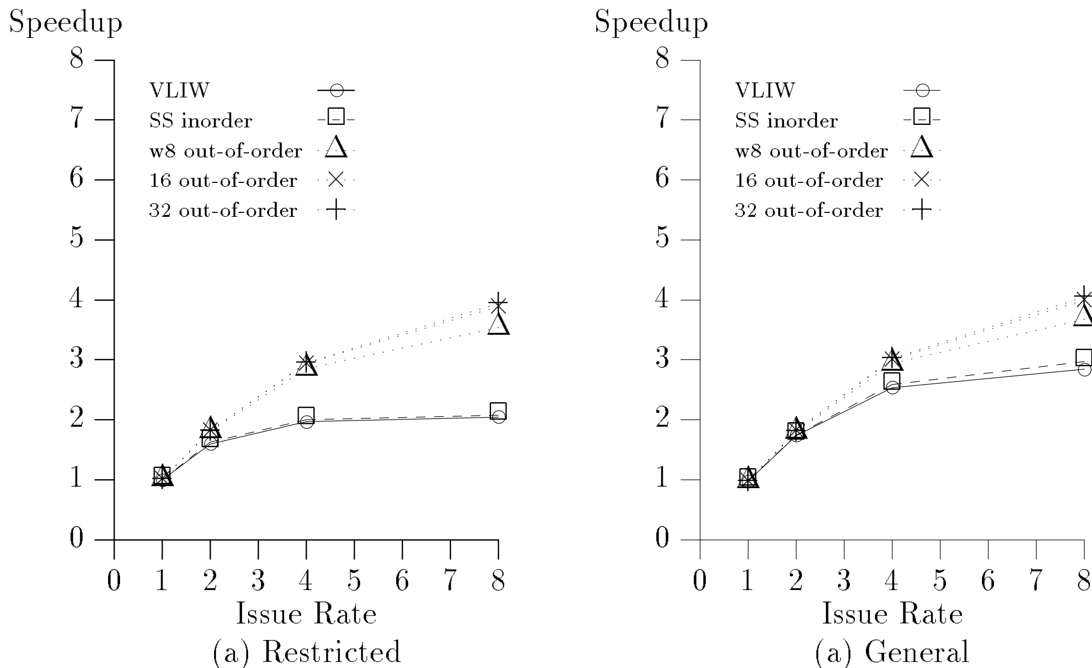

Figure 6.3: Effects of Smaller Window Sizes

researchers have indicated that the same speedups can be achieved with fewer function units [10, 16].

6.2 Effects of Window Size

Heretofore the instruction window size of 32 has been assumed for the out-of-order execution model. Since this size of window requires considerable associative search, smaller-sized windows were run for the out-of-order model. Figure 6.3 shows that the window size of 16 performs almost as well as the 32. The window size of 8 does degrade performance somewhat at an issue rate of 8. The VLIW and in-order models are shown in the figure to give points of reference.

A small window is adequate for the processor because the benchmarks are mostly integer programs. Since instructions are short, they don't spend much time in the window.

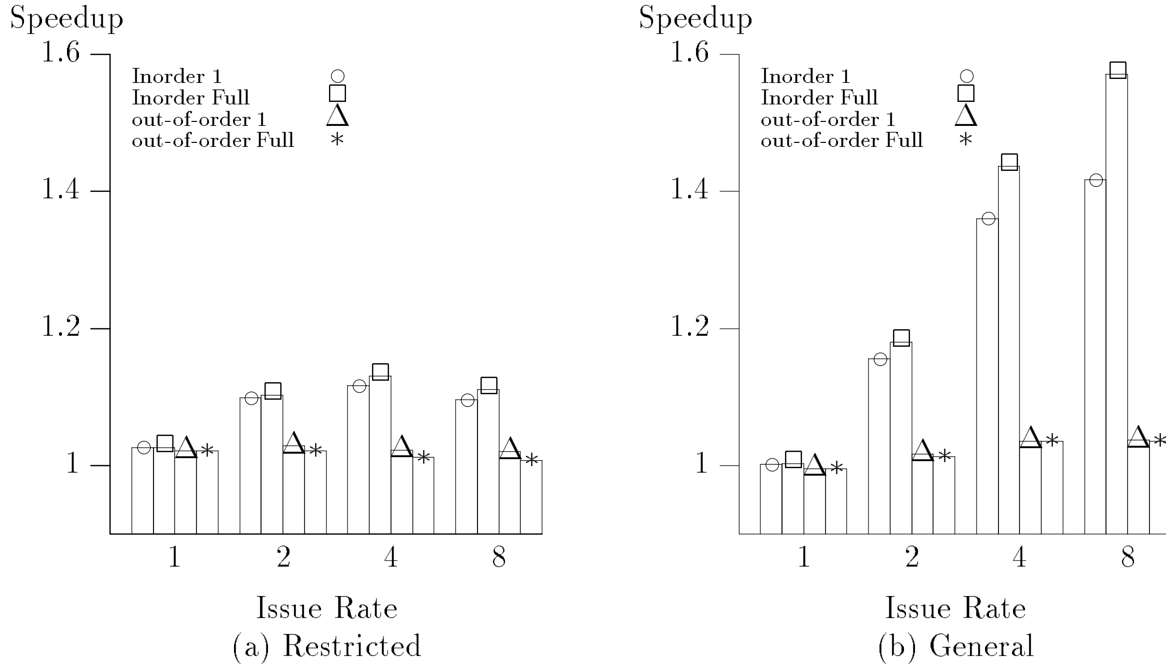

Figure 6.4: Performance Benefit from Code Scheduling

6.3 Effects of Code Scheduling

Code scheduling is not required for the in-order and out-of-order superscalar models to execute programs correctly, but it offers some improvement in performance. Figure 6.4 shows the speedup of scheduled code over nonscheduled, highly optimized code. The speedups are calculated relative to the same machine without scheduling. For each model, results are given for code scheduled for the single issue rate and for code scheduled for the proper issue rate. For the in-order model, performance improves significantly. This is expected, because the in-order model has many more hardware constraints which limit scheduling. For the out-of-order model, the performance increase is about 4% for the general code, but the restricted model offers little increase. In both code motion cases in the out-of-order model, the single issue schedule works almost as well as the schedule for the current issue, because the scheduler has a tendency to schedule according to

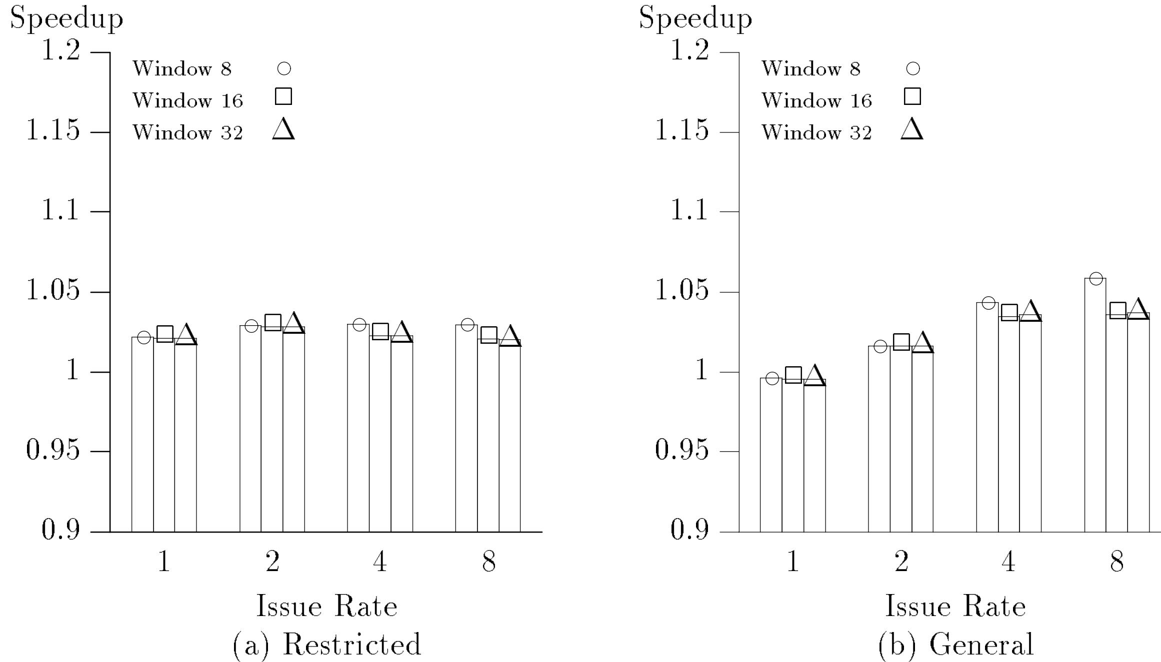

Figure 6.5: Effects of Window Size on Code Scheduling Benefit

heuristic weight rather than issue rate. Most instructions end up in the same place in the sequential instruction stream regardless of the issue rate at which they are scheduled.

The more aggressive the hardware is, the less scheduling helps. In the previous case, the window size was 32 with plenty of function units. If the window size is decreased, the benefits of scheduling are increased. Figure 6.5 shows the benefits of scheduling for different window sizes. The speedups are calculated relative to the out-of-order model with the same window size. For higher issue rates and smaller window sizes, the benefit of code scheduling is around 7%. This is two times the previous benefit.

6.4 Effects of Register Renaming

Out-of-order execution has many benefits over in-order execution. One of them is that registers are renamed eliminating output dependencies among instructions. It is

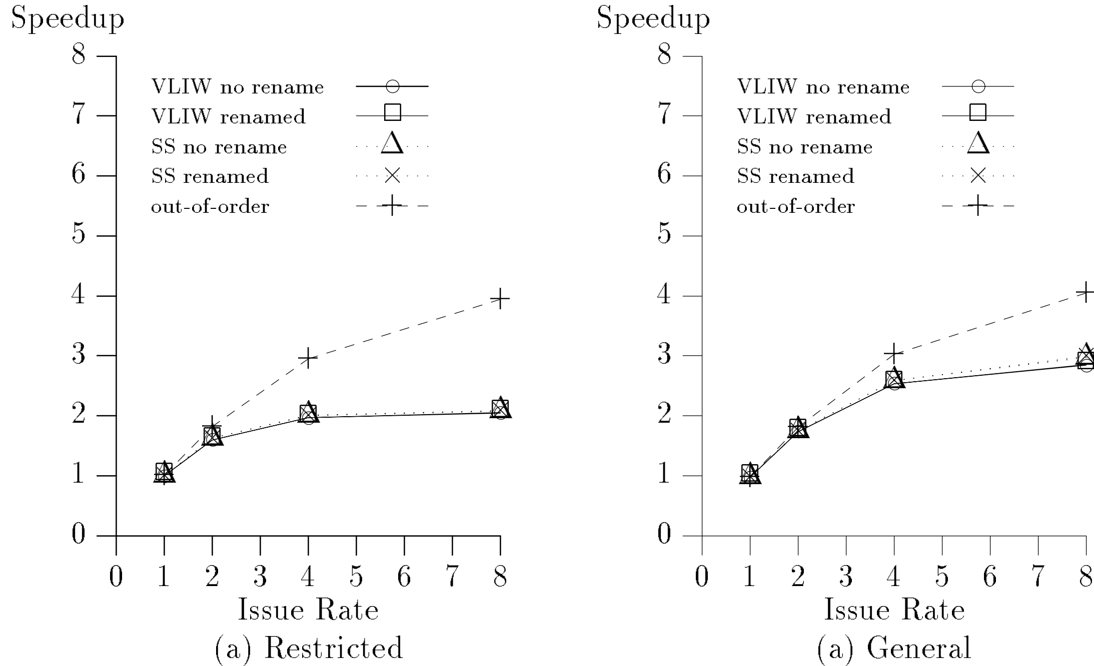

Figure 6.6: Effect of Register Renaming on Performance

also possible to rename registers in the in-order and VLIW models. Figure 6.6 shows the effect of register renaming relative to the other nonrenamed models and the out-of-order model. Basically, renaming does not increase performance significantly. (It does slightly for some of the benchmarks.) The reason that performance is not increased is that output dependencies rarely occur because the optimizing compiler usually removes them.

7. CONCLUSIONS

Exploitation of instruction-level parallelism can provide significant speedup of serial-based code. All three hardware models that utilize this parallelism provide good speedup over scalar code. The most complex out-of-order model performs the best, because it dynamically schedules the instructions of the programs. However, as more restrictions are placed on that model, the software support has a larger impact on performance and the hardware support becomes less useful. Since the out-of-order model presented here is highly aggressive, the simpler inorder models might perform the best when implementation costs are considered.

The out-of-order model requires significantly more hardware as the system scales up, whereas the inorder models seem to scale fairly well. One of the problems with the inorder models is that even the most aggressive compiler technology does not have the detailed information of the dynamic trace and thus is limited to scheduling superblocks and making many assumptions about branch directions. The static speedup is also limited by the size of the superblock. As compilers find more parallelism in programs, the hardware techniques are bound to fall behind due to the overhead cost of dynamically

scheduling code. Once this happens, the compiler-based approaches should be able to perform better than the out-of-order hardware approaches with the same hardware cost.

APPENDIX A. CODE SCHEDULER

This appendix gives the details of generating code for the trace generator.

A.1 Scheduler

The scheduler is necessary for the trace generator to output useable code. It is possible to run superscalar code without using the code scheduler, but registers will not be allocated. In order to have register allocation, the scheduler must be run. The register allocation also puts in spill code and callee-caller saves.

Figure A.1 shows the basic flowchart of the scheduler. The passes can be turned on and off by setting options in the L_OPTION file. This requires recompilation of the scheduler.

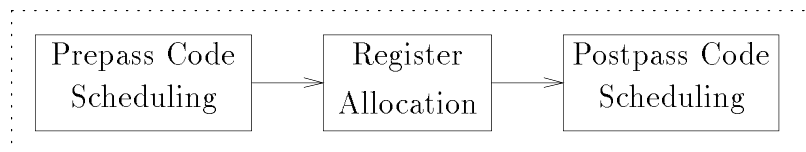

Figure A.1: Flow Chart of the Scheduler

```
(opcode jsr      1  (cnt) void 1)
(opcode beq_fs   14  (cnt) void 1)
(opcode mov      57  (arith) i 1)
(opcode add      60  (arith) i 1)
(opcode ld_uc   122  (load) i 2)
(opcode st_c    129  (store) void 1)
```

Figure A.2: Example Opcode File

A.1.1 Important directories

The main source directory: /home/polaris1/impcc/src/Limpact2/

The option files are in: /home/polaris1/impcc/option/

The default opcode file is: L_OPCODE_2

The default specification file is: L_IMPACT_1

The default option file is: L_OPTION_r

The executable for Limpact2 are in: /home/polaris1/impcc/bin/

Short examples of L_OPCODE, L_IMPACT, and L_OPTION are shown in Figures A.2, A.3, and A.4, respectively. The opcode file contains the names and types of instructions. The specification file contains groups of instructions and those instructions which can be issued together. The option file is a list of options for the scheduler.

A.1.2 Syntax of the command

Limpact2 −i <input_file> −o <output_file> −opcode <opcode_file>
−spec <machine_spec_file> −option <compile_options_file>

There are shell scripts in the /home/polaris1/impcc/option directory which run the scheduler on a group of files. The scheduler produces a file called IMPACT_001 which

```
/*-------------------------------------------------------------*/
(define_register_type sp1 (INT) 64 0 1)
(define_register_type fp1 (INT FLOAT) 64 0 1)
(define_register_type fp2 (DOUBLE) 64 0 2)
/*-------------------------------------------------------------*/
(define_predicate_operand_type  PRED_1          0 1)
(define_int_operand_type  INT_1    -1000000 +1000000)
(define_int_operand_type  INT_2    0 64)
(define_float_operand_type  FLOAT_1    -1.0e50 +1.e50)
(define_double_operand_type  DOUBLE_1   -1.0e100 +1.e100)
(define_label_operand_type  LABEL_1    offset)
(define_label_operand_type  LABEL_2    direct)
(define_register_operand_type  SR1    sp1)
(define_register_operand_type  FR1    fp1)
(define_register_operand_type  FR2    fp2)
/*-------------------------------------------------------------*/
(define_operand_mode DEST (SR1 FR1 FR2))
(define_operand_mode SRC (
   SR1 FR1 FR2 PRED_1 INT_1 INT_2 FLOAT_1 DOUBLE_1 LABEL_1 LABEL_2
))
/*-------------------------------------------------------------*/
(define_operation_type jsr (jsr () (SRC)))
(define_operation_type beq_fs (beq_fs () (SRC SRC SRC)))
(define_operation_type mov (mov (DEST) (SRC)))
(define_operation_type add (add (DEST) (SRC SRC)))
(define_operation_type ld_uc (ld_uc (DEST) (SRC SRC)))
(define_operation_type st_c (st_c () (SRC SRC SRC)))
/*-------------------------------------------------------------*/
(define_operation_group ALU (
    mov add ld_uc st_c
))
(define_operation_group BR (
    jsr beq_fs
))
/*-------------------------------------------------------------*/
(define_instruction_type T1 (ALU ALU))
(define_instruction_type T2 (BR ALU))
(define_instruction_type T3 (ALU BR))
/*-------------------------------------------------------------*/
```

Figure A.3: Example Specification File

```
/*      SUPERBLOCK CREATION OPTIONS.  */
(L_min_trace_copy_weight                50)
(L_max_trace_size                       100)
(L_min_fs_weight                        100)
(L_min_jump_opti_weight                 100)
/*      SUPERBLOCK SCHEDULNG OPTIONS.  */
(L_prepass_scheduling                   on)
(L_register_allocation                  on)
(L_postpass_scheduling                  on)
(L_optimize_for_parallel_issue          on)
(L_num_predicate_registers              64)
(L_num_predicate_spill_registers        1)
(L_num_integer_registers                64)
(L_num_integer_spill_registers          4)
(L_num_float_registers                  32)
(L_num_float_spill_registers            4)
(L_num_double_registers                 32)
(L_num_double_spill_registers           4)
(L_restricted_code_percolation          on)
(L_general_code_percolation             off)
(L_sentinel_code_percolation            off)
(L_boosted_code_percolation             off)
(L_allow_speculative_stores             off)
(L_boosting_limit                       32)
(L_unsafe_mem_dep_resolving             1)
(L_resolve_all_memory_dep               off)
(L_resolve_all_anti_output_dep          off)
(L_resolve_all_control_dep              off)
```

Figure A.4: Example Option File

contains information on the run time of the program if the profile-based execution model is used. It is not useful for trace-based simulation.

A.2 Instrumenting for Trace Generation

Once the code has been scheduled, it has to be instrumented to generate a trace. The syntax of the command follows:

```
Lmips−trace −trace 2 −opti 1 −target dec−mips
<input_file> −o <output_file>
```

A shell script for running this on multiple files is given in Figure A.5. Lmips−trace produces files called fnid_file and fnid_num, which contain the function names and the number of the functions, respectively.

A.3 Merging the Information

Lmerge is used to merge the information of the entire program and create information files used in trace generation. All of the files created by Limpact2 are piped through Lmerge. The usual syntax of Lmerge follows:

```
cat <files> | Lmerge −target dec−mips −trace 2
```

This command creates IMPACT_FN_INDEX, IMPACT_LCODE_BREAK, and IMPACT_LCODE_INFO. These files contain information that is used to generate traces.

A.4 Linking

After all of the previous steps have been completed, the code is then compiled using the C compiler to produce object files. (Compile using cc −c.) Then all of the files are

```
#!/bin/sh
#  Generate machine code for impact
#
#  Usage: gen_Limpact "directory" < "list"

if [ $# -gt 0 ]; then
  PREFIX="$1";
fi;

echo "> prefixing all files with [${PREFIX:=.}/]";

\rm fnid_file fnid_num

while read FILE;
do
  echo "> processing $PREFIX/$FILE"
  FILEOUT=`echo $FILE | awk -F. '{print $1}'`
  Lmips-trace -trace 2 -opti 1 -target dec-mips -i ${PREFIX}/${FILE} \
    -o ${PREFIX}/${FILEOUT}.s
  sync
  ERROR=$?;
  if [ $ERROR -ne 0 ]; then
      echo "> Limpact failed on input file: ${FILE}";
      echo "> error_code= ${ERROR}";
      exit 0200;
  fi;
done;
```

Figure A.5: Shell Script to Run Lmips-trace

linked together to create and execute. This includes the simulator object files and three files which contain routines for generating traces. These files are merge.o, trace.o, and general.o. They are currently available in ~jgholm/impcc/sssim1/.

REFERENCES

[1] M. Butler, T. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, "Single instruction stream parallelism is greater than two," in *The 18th Annual Symposium on Computer Architecture*, pp. 276–286, May 1991.

[2] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, Nancy J. Warter, and Wen-mei W. Hwu, "Impact: An architectural framework for multiple-instruction-issue processors," in *The 18th Annual International Symposium on Computer Architecture*, vol. 19, pp. 266–286, May 1991.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers, Principles, Techniques, and Tools.* Reading, Massachusetts: Addison-Wesley, 1986.

[4] S. A. Mahlke, "Design and implementation of a portable global code optimizer," M.S. thesis, University of Illinois at Urbana-Champaign, 1991.

[5] Edward G. Coffman, Jr., Ed., *Computer and Job/Shop Scheduling Theory.* New York, New York: John Wiley & Sons, 1976.

[6] R. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, pp. 25–33, Jan. 1967.

[7] S.F. Anderson, J.G. Earle, R.E. Goldschmidt, and D.M. Powers, "The IBM system/360 model 91 floating-point execution unit," *IBM Journal of Research and Development*, vol. 11, pp. 34–53, Jan. 1967.

[8] R.M. Keller, "Look ahead processors," *Computing Surveys*, vol. 7, pp. 177–195, Dec. 1975.

[9] Wen-mei William Hwu, Exploiting concurrency to achieve high performance in a single-chip microarchitecture. Ph.D. dissertation, University of California at Berkeley, 1988.

[10] Mike Johnson, *Superscalar Microprocessor Design.* Englewood Cliffs, New Jersey: Prentice-Hall, 1991.

[11] J.E. Smith and A.R. Pleszkun, "Implementation of precise interrupts in pipelined processors," in *The 11th Annual Symposium on Computer Architecture*, June 1985.

[12] J.K.F. Lee and A.J. Smith, "Branch prediction strategies and branch taget buffer design," *IEEE Computer*, pp. 6–22, Jan. 1984.

[13] W.W. Hwu, T.M. Conte, and P.P. Conte, "Comparing software and hardware schemes for reducing the cost of branches," in *The 16th Annual International Symposium on Computer Architecture*, May 28-June 1, 1989.

[14] Shlomo Weiss and James E. Smith, "Instruction issue logic in pipelined supercomputers," *IEEE Transactions on Computers*, vol. c-33, pp. 1013–1022, Nov. 1984.

[15] Raj Jain, *The Art of Computer Systems Performance Analysis.* New York, New York: John Wiley & Sons, 1991.

[16] T. Conte, Systematic computer architecture prototyping. Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.