EVALUATION OF A DECOUPLED COMPUTER ARCHITECTURE AND THE DESIGN OF A VECTOR EXTENSION

by

Honesty Cheng Young

Computer Sciences Technical Report #603

July 1985

EVALUATION OF A DECOUPLED COMPUTER ARCHITECTURE AND THE DESIGN OF A VECTOR EXTENSION

by

HONESTY CHENG YOUNG

Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706

May 1985

to my Parents
for their love, faith, support,
encouragement, and inspiration

to my wife
for her love, patience,
sacrifice, and support

ABSTRACT

To meet increasing demands for computing power, systems must exploit parallelism at all levels. A balance between the processing speeds of a machine's subsystems is critical for overall system performance. A high performance system must have a fast scalar mode as well as a vector mode which reduces the fraction of work that the scalar mode must process.

PIPE (Parallel Instructions and Pipelined Execution) is a very high performance computer architecture intended for a heavily pipelined VLSI implementation. The primary goal of the PIPE architecture is fast execution of scalar programs. PIPE has several novel features in order to meet this goal. These features include *architectural queues* which reduce the effective memory accessing delay; a *prepare-to-branch* instruction which decreases the penalty incurred by branches; and *decoupled execution mode* which increases the instruction initiation rate.

In this study, a compiler (for a subset of Pascal) has been developed to evaluate the effectiveness of these architectural features. A code scheduler takes advantage of the architectural queues and the prepare-to-branch instruction. Software pipelining (which can prefetch operands across loop boundaries) also takes advantage of these features. A code generator generates two instruction streams (of sequential tasks), that run on the two processors required for PIPE's decoupled mode.

A queue-based vector extension is proposed. This extension has the following characteristics: (1) two level control used to support out-of-order instruction initiation, (2) multiple classes of registers, (3) a very short vector start-up time (one clock period), (4) branch instructions used only for implementing high level language control structures, (5) elements of a queue may be read repeatedly, and (6) easily vectorizable property.

We demonstrate that the original PIPE architecture supports fast execution of scalar programs, and that the vector extension facilitates vectorization.

TABLE OF CONTENTS

ABSTRACT iii

1. Introduction 1

1.1. The Demand for High Performance Systems 1

1.2. Terminology 2

1.3. Research Goal and Plan 3

1.4. Thesis Organization 5

2. Related Work 6

2.1. Decoupled Architectures 6

2.2. Code Scheduling Methods 9

2.3. Vector Supercomputers 11

3. The PIPE Architecture 14

3.1. Load/Store Instructions 16

3.2. Branch Instructions 18

4. The Code Scheduler 23

4.1. The PIPE Pascal Compiler 23

4.2. Code Scheduler for the Intermediate Language 25

4.2.1. The Algorithm 25

4.2.2. Examples 32

4.3. Code Scheduler at the Machine Code Level 33

4.4. One-Level Code Scheduling 40

4.5. Remarks 43

4.6. Simulation Studies 44

5. Software Pipelining 50

5.1. Software Pipelining Results 57

6. Code Generation for Decoupled Mode 59

6.1. Code Generation Methods for Decoupled Mode 59

6.2. Simulation Results 61

7. The Design of a Vector Extension 65

7.1. Motivation for Vector Instructions 67

7.2. Desired Features 69

7.3. Implementation 72

7.3.1. Two-Level Control 72

7.3.2. Queue Registers 74

7.3.3. Vector Load/Store 79

7.3.4. Mask Registers 80

7.3.5. Comparison Instructions 80

7.3.6. Vector Editing Instructions 81

7.4. Performance Evaluation 82

7.5. Intrinsic Functions 85

7.6. Summary 86

8. Conclusions 88

REFERENCES 91

LIST OF FIGURES

3.1. PIPE: A Decoupled Architecture 15

4.1. The *WDG* of Example 4.1 39

4.2. Relative Performance for Different LDQ Sizes 48

7.1. The Organization of a Processor with Two Functional Units 71

7.2. The Organization of a Queue Register 75

LIST OF TABLES

4.1. Relative Performance Gain by Adding the LDQ and the PBR Instructions to Bare PIPE Machine 44

4.2. Relative Speedup Due to the LDQ Only 45

4.3. Relative Performance for Different LDQ Size 46

4.4. The Dynamic Branch Count Distribution 47

5.1. The Speedup of Software Pipelined Code 58

6.1. Performance Simulation Results for Decoupled Mode 62

6.2. Performance Simulation Results of All Features 64

7.1. Timing Chart for Livermore Loop 12 74

7.2. Execution Time (of One Iteration) of Five Livermore Loops 84

ACKNOWLEDGEMENTS

I am indebted to Professor James. R. Goodman, my advisor and committee chairman, for his invaluable support, guidance, and encouragement in all aspects of my research, as well as his precious assistance in improving my technical writing skills. I would like to thank the members of my committee, Professors Michael J. Carey, Charles N. Fischer, Andrew R. Pleszkun, Marvin H. Solomon, James E. Smith, and Mary K. Vernon for their extremely helpful comments and suggestions.

Thanks are also due to members in the PIPE group. In particular, M.A. Holliday, J.-T. Hsieh, and P.B. Schechter who developed the dataflow analyzer, the PIPE performance simulator, and the PIPE functional interpreter, respectively.

I wish to express my deepest gratitude to the members of my family for their support throughout my graduate studies; especially to my wife, Emma, who sacrificed so much to make this dream become reality.

This work was supported by National Science Foundation under Grant MCS-8202952 and MCS-8105904.

Chapter 1

Introduction

1.1. The Demand for High Performance Systems

The computing requirements for solving all kinds of problems are increasing rapidly. In order to get high computing power, systems must exploit parallelism at all levels. It is well known [Amdahl67] [Rudsinksi77] [Bucher83] [Worlton84] that for a system with two processing speed modes, the effective execution speed is limited by the slower mode, unless the fraction of workload that has to be done in this mode is nominal. Put differently, a good balance between different processing speeds is critical to the overall system performance. In the numerical supercomputer environment, the *high-speed mode* corresponds to vector operations, the *low-speed mode* to scalar instructions. We must push the limits on both ends to meet ever-increasing computing requirements. An ideal high performance system must not only have a fast scalar mode, but also a vector mode that reduces the fraction of work that must be processed in the scalar mode. The latter can be achieved by architectural innovations which facilitate the vectorization of most array operations.

Pipelining has been used to build processors with a fast scalar mode for quite some time [Kogge81]. The performance of a pipelined machine, however, is often limited by memory accessing, conditional branching, and the *Flynn bottleneck.* Flynn [Flynn66] observed that there is always some point in the instruction fetch/decode path through which instructions pass at the maximum rate of one per clock period. A survey of commercially available high performance computer systems (*e.g.*, Cray-1 [Russell78], Cyber 205 [Lincoln82]) supports this observation.

A new computer architecture, called PIPE (Parallel Instructions and Pipelined Execution) [Smith83] [Craig83] [Goodman85] is a VLSI-oriented, high performance architecture project at the University of Wisconsin-Madison. The primary goal of the PIPE architecture is fast execution of sequential programs. Architectural data queues are included in the PIPE architecture to reduce the influence of delay due to accessing memory. The *prepare-to-branch* (PBR) instruction is a mechanism to decrease the penalty incurred by branches.

In order to achieve a high instruction issue rate, the major design principle of PIPE is simple issue conditions: only a few simple conditions must be checked to initiate an instruction. The instruction itself (*e.g.*, floating point multiplication), however, may take several clock periods to complete. The PIPE architecture goes one step further, attaining a higher instruction initiating rate by allowing two cooperative instruction streams of a sequential task to run on two processors. Thus, the bandwidth of the instruction initiating rate is doubled by having two instruction issue units on the system--one on each processor.

1.2. Terminology

A *computer architecture* includes: (a) the specification and design of the components (building blocks) of a computer system, (b) the interactions between the components, and (c) the evaluation of the entire system or its components. An architecture that has multiple processors running multiple instruction streams of a sequential single task is called a *decoupled architecture* [Smith84]. The decoupled architecture we are interested in is one that separates a task into two parts: operand access and algorithmic computation. The instruction stream associated with operands access, termed the *access instruction stream*, runs on an *access processor* (A-processor or AP). On the other hand, the algorithmic computation instruction stream, termed the *execute instruction stream*, runs on an *execute processor* (E-

processor or EP). The execution mode of running two instruction streams on two processors is called the *Access/Execute (AE) mode* or the *decoupled mode*, while its counterpart (if it exists), which runs one stream on a single processor, is called the *Single Processor (SP) mode* or the *single mode*.

Code scheduling is a special kind of code motion. It is useful only for architectures with simultaneous operations, such as pipelined processors or multiple functional unit systems. The function of code scheduling is to define a semantically equivalent code sequence by reordering instructions. Its purpose is to utilize the parallelism provided by the underlying hardware, avoiding unnecessary resource conflicts in order to reduce the running time of compiled programs.

Vectorization is the task performed by a compiler to recognize parallelism lost in the use of sequential programming languages. *Vector computers* can process streams of elements, rather than a few simple operands, within a single instruction. The *vector startup time* is the execution time difference between using a vector instruction to process an array of length one and processing the same item with scalar instructions.

1.3. Research Goal and Plan

The goal of this work is to evaluate the effectiveness of PIPE's novel features in supporting a fast scalar mode, and to design an extension which provides short vector startup time and furnishes the capability of vectorizing most array-oriented programs to improve the total system performance.

A new architectural feature is efficacious if (a) the "loaded version" of the architecture provides a reasonable performance gain over its "bare version" counterpart (an equivalent architecture without this special feature); (b) with reasonable effort, an optimizing compiler can be constructed to utilize the special feature; and (c) it can be implemented for a reasonable cost. In this study, we will use the

comparison between the loaded version and the bare version as the measurement.

A compiler for a subset of Pascal has been developed to evaluate aspects of the PIPE architecture. A code scheduler is used to reshape the compiled code to take advantage of the data queues and the prepare-to-branch instruction. Software pipelining [Charlesworth81] (also known as loop folding [Weiss84a]) is a technique to schedule the code sequence of the inner-most loops by rearranging the code across basic block[1] boundaries. The scheduling methods described in this thesis are applicable to most register-register pipelined architectures by simply changing the cost table.

The potential speedup for decoupled mode execution has been shown by using hand-written code [Smith84] [Hsieh84]. The Pascal compiler for PIPE is capable of generating the aforementioned two instruction streams automatically. The methods of generating the decoupled code are explained. The benchmark programs used are the first 14 Livermore loops [McMahon72] [Riganati84] and a few procedure-call intensive programs.

Many array-oriented programs cannot be vectorized, partly because the underlying vector computers do not provide the adequate primitive instructions to generate vector code for many commonly seen vector operations, such as vector inner product. The major goal of the proposed two level control, queue-based vector extension is to ease the task of the vectorization of array-oriented programs. Some limitations of the original PIPE architecture are also eliminated.

---

[1] A basic block is a code sequence with no jumps in, except at the beginning, and no jumps out, except at the end [Aho77].

1.4. Thesis Organization

In chapter 2, we compare some decoupled architectures, delineate the heuristics used for scheduling methods, and contrast second generation vector supercomputers [Kozdrowicki80] with the Cray-1.

PIPE is the underlying architecture of this research. In chapter 3, we describe the interesting features of the PIPE architecture and their rationales. In the subsequent two chapters, we emphasize the code scheduling methods used to utilize PIPE's novel features. The effectiveness of the scheduling methods is demonstrated by running a set of benchmark programs. The time complexity of the scheduling methods is also discussed.

In chapter 6, the code generation method for the decoupled mode is explained. The speedup of the compiled code is compared with that of the hand-written code.

In chapter 7, we point out some of PIPE's limitations. A vector extension is suggested, which makes more array-oriented operations vectorizable. Meanwhile, this extension also remedies most limitations of the original PIPE architecture. We conclude this thesis in chapter 8. Possible directions for future research are also outlined.

Chapter 2

Related Work

In this chapter, work is presented that relates to the work that will be discussed in subsequent chapters. In section 2.1, we compare several decoupled architectures. In section 2.2, we discuss some scheduling methods. In section 2.3, we contrast the special vector processing features of state-of-the-art vector supercomputers and point out some of the limitations.

2.1. Decoupled Architectures

The major common feature among different decoupled architectures is the partitioning of multiple units. Some earlier high performance systems (*e.g.*, IBM 360/91 [Anderson67] decouple instruction fetch/decode (I-unit) from their execution (E-unit). The execution unit may consist of multiple functional units [Thornton70] or a pipelined processor or the combination of both (*i.e.*, multiple pipelined functional units [Anderson67] [Cray82]). The introduction of decoupled architectures reveals a new level of parallelism by using multiple units executing multiple instruction streams for a single task.

The CSPI MAP-200 [Cohler81] is the first commercial machine which enhances the I-unit/E-unit decoupling by running two instruction streams on two processors--one for data access and the other for execution. Each processor can proceed at its own pace without waiting for another processor to complete its task. Queues provide *elastic coupling* [Cohler81] between processors. Branch synchronization is done by setting/clearing corresponding flags between processors. A third memory transfer controller is included in the system. The memory transfer controller, however, does not execute a user program stream. It simply surveys the

status of various queues and does memory operations as promptly as possible. The memory transfer controller gives the read operation priority; requiring software solutions to prevent some memory hazards.

A class of DAE (decoupled access/execute) computer architectures has been proposed by Smith [Smith84]. Different organizations to implement a decoupled computer architecture are compared. Branch queues are included between processors to pass branch outcomes; allowing one processor to run several *branches* ahead of the other. Thus, instead of setting and clearing flags, synchronization between processors is provided by the branch queues. Memory conflicts are resolved by a hardware associative search on the address queues.

The SMA (Structured Memory Access) [Pleszkun82] [Pleszkun83] also has two processors. All memory requests, including instructions, are generated and controlled by one processor. Except for the loop mode, the two processors in the SMA are similar to the I-unit/E-unit decoupling described earlier in this section. During loop mode, however, two instruction steams run asynchronously on two processors. The memory access processor of the SMA also has several sophisticated mechanisms to generate addresses efficiently and to make fewer memory references.

The FOM (Fortran Optimized Machine) [Brantley82] [Brantley83] is a processor with multiple functional units. The instruction decode/issue is done in two stages. The high level instruction dispatch unit sends instructions to the instruction queues of the appropriate function units. Instructions decode/execute independently in their individual units. Therefore, instructions that use different functional units may be initiated out of the order that they passed the instruction dispatch unit. Architectural FIFO queues are used for communications between units.

The VLIW (Very Long Instruction Word) machine [Fisher84b], which has multiple CPUs, pushes the decoupling even farther, by issuing more than two instructions, running on multiple processors, every clock period. Its function is similar to a horizontal microcode engine. However, there are no queues in the VLIW computer. The operations on different processors are executed synchronously.

The SDP (Synchronous Distributed Processor) [Shively82] uses multiple microprocessors to implement two subunit functions in addition to the I/O unit. Interlocks are enforced by operand queues between two subprocessors. The instructions and data for the arithmetic unit are supplied by the index arithmetic unit. Hence, the arithmetic unit works like an attached processor, and the index arithmetic unit works like a host.

Among the decoupled architectures described above, the MAP-200, the Mark IIA, and the SDP are working machines, while the DAE, the SMA, the FOM, and the VLIW are paper machines at the time of this writing.

Many commercially available attached processors (*e.g.*, AP-120B [Charlesworth81], FPS-164 [Charlesworth81], IBM 3838 [IBM76], and MATP [Datawest79]) also have some of the characteristics of decoupled architecture, except that there is normally one instruction stream.

The PIPE (Parallel Instructions and Pipelined Execution) [Smith83] [Craig83] [Goodman85] is a VLSI experimental processor proposed at the University of Wisconsin-Madison. Its novel features include (a) architectural queues, (b) generalized delayed branch, (c) branch target registers, and (d) identical processors which imply the possibility of several execution modes. We will describe the relevant information about the PIPE architecture in the next chapter.

2.2. Code Scheduling Methods

The purpose of code scheduling is to create a semantically equivalent code sequence that minimizes total execution time. It is known [Ullman75] [Abdel-Wahab76] [Garey79] [Hennessy83] that code scheduling is an *NP*-complete problem. Normally, heuristics are used to get suboptimal solutions. The steps in code scheduling are: (a) perform dependency analysis, (b) keep a list of instructions ready to be issued, and (c) pick an instruction from the ready-list. Different architectures may have different structural dependencies. Hence, part of the dependency analysis is a processor dependent activity. The major difference among scheduling methods lies in the heuristics used to select an instruction from the ready-list. A branch-and-bound algorithm, which has exponential worst-case performance, can achieve the optimal solution [Gonzalez77]. In practice, however, it is too expensive to use the branch-and-bound method. The heuristic used by Hennessy and Gross [Hennessy83] is to choose the node farthest from the root (*i.e.*, the end of the basic block). Their hope is that the node farthest from the root is on the critical path. The worst case time complexity of their scheduling algorithm is $\mathbf{O(n^4)}$, where **n** is the size of the current basic block in terms of number of instructions. Their algorithm does software interlocking by inserting NOPs whenever necessary. Another heuristic used by Wood [Wood78] is to choose the node with the most descendants (direct or indirect). Adam, Chandy, and Dickson [Adam74] have compared, through extensive simulations, the performance of five heuristics used in an unrestricted environment. The heuristics studied are:

(1) HLFET (Highest Levels First with Estimated Times),

(2) HLFNET (Highest Levels First with no Estimated Times),

(3) RANDOM,

(4) SCFET (Smallest Colevels First with Estimated Times),

(5) SCFNET (Smallest Colevels First with No Estimated Time).

They concluded that the accuracy under their test cases is ordered: HLFET, HLFNET, SCFNET, Random, and SCFET. The idea behind HLFET (Highest Levels First with Estimated Times) is to issue the node with the longest-path. The method used by Hennessy and Gross [Hennessy83] is a variation of HLFNET. A compiler [Thorlin67] for the 6600 uses a version of HLFET as its heuristic. The heuristic used in this thesis is a modified HLFET.

Loop unrolling [Dongarra79] is a technique to increase the size of the basic block of a loop. It is a routine practice on some supercomputers [Cray85]. Fisher [Fisher81] proposed "trace scheduling" to reorder code across basic block boundaries (including loop unrolling), which, in some sense, is a way to increase the size of basic blocks. Rau and Glaeser [Rau81] explain some optimal scheduling methods for limited programs under their polycyclic architecture which avoid many resource conflicts by providing delay elements in the interconnection network.

The optimization of delayed branch [Patterson81] [Patterson82] [Hennessy81] is a special form of code scheduling. It is done by moving instruction(s) to the locations following a delayed branch instruction. In RISC, the movement of code to follow a delayed branch is done by a peephole optimizer [Patterson81]. In MIPS, Gross [Gross82] considered moving an instruction not only within the basic block, but also from the two possible branch targets.

The scheduling methods described in chapters 4 and 5 are used to utilize some special architectural features provided by the underlying PIPE architecture.

2.3. Vector Supercomputers

The second generation of vector supercomputers[2] [Kozdrowicki80] began when Cray Research, Inc., announced the Cray-1 [Russell78]. We will describe the Cray-1 [Cray82] in some detail and contrast features in other vector supercomputers with those of the Cray-1.

In addition to two classes of general purpose scalar registers and their backup counterparts, the Cray-1 has eight vector registers. With the "chaining" of pipelined functional units, scalar temporary results are used immediately after they become available. The vector chaining, however, is limited by the "chain slot time," when the reservation on a destination vector register is lifted for one clock period. Below is an example to explain the chaining in the vector mode of Cray-1. Given two vector instructions, $\mathbf{Vop_1}$ and $\mathbf{Vop_2}$, where $\mathbf{Vop_2}$ uses the results generated by $\mathbf{Vop_1}$. If all other issue conditions of $\mathbf{Vop_2}$ are met before the chain slot time of $\mathbf{Vop_1}$, $\mathbf{Vop_2}$ can be issued at the chain slot time of $\mathbf{Vop_1}$. Otherwise, $\mathbf{Vop_2}$ cannot be issued until $\mathbf{Vop_1}$ is completed. A vector store instruction is blocked from chain slot execution. If speed control is in effect (speed control is caused by systematic memory bank conflicts), a vector read cannot chain. Chaining can happen only during the chain slot time because only one *component counter* is associated with each vector register. When a vector register serves as both source and destination register, its component counter is not updated until the first result is written to the vector register. That is, chaining is a synchronous operation--a result operand must be generated every clock period and that operand must be used immediately after it has been computed. All twelve functional units, which perform twelve different operations, are fully pipelined and some of them are shared by the

---

[2] The two major first generation vector supercomputers are the CDC Star-100 [CDC73] and the TI-ASC [Watson72].

scalar and the vector portions of the machine. The startup time for vector operations is nominal. For typical operations, vectors of length of three or fewer run faster in scalar mode, while those of four elements or more run faster in vector mode. Vector performance, however, is often bounded by the memory bandwidth because there is one port between the CPU and the memory system. Elements of an array can be loaded with a constant stride apart, while elements of sparse vectors must be accessed and processed by scalar mode instructions. Each vector register has sixty-four elements. A vector longer than sixty-four entries must be split into several smaller ones to fit into the vector registers.

The Cray X-MP series [Cray85] is the successor of the Cray-1. The major differences between the Cray X-MP and the Cray-1 are described below. Multiple CPUs are included in the Cray X-MP series, so another dimension of parallelism (*i.e.*, multi-tasking) is possible. Shared registers are included between CPUs. Three memory ports are provided for vector operations. Thus, two load operations and one store operation can occur simultaneously. The software, however, must prevent memory overlap hazard conditions caused by concurrent block reads/writes. The "hardware automatic chaining" feature allows chaining to happen any time after the element 0 of the result arrives at the destination vector registers. The flexible chaining is achieved by using two pointers (component counters) with each vector register. More loops may be vectorizable because gather/scatter can be used to handle sparse vectors.

The Cyber 205 [Lincoln82] is a memory-memory vector processor where two adjacent elements of a vector must be in consecutive memory locations to avoid the complexity of detecting possible memory bank conflicts. Elements of non-unit stride apart have to be moved to consecutive locations using special vector editing instructions. On the other hand, the vector length can be as long as 64K elements. Various vector macro instructions are in the instruction set in order to cover many

hard-to-vectorize operations, such as vector inner product and maximum function. A family of sparse vector instructions is included to handle sparse vectors. Scalar chaining is done by using a special data path (called *shortstop*) between the output and input areas of the scalar arithmetic unit. Vector chaining is accomplished by configuring the data interchange to connect the input and output "trunks" (buses) to the appropriate processing unit.

The Facom VP-200 [Miaru83] is furnished with a relatively complete family of vector editing instructions. It not only provides gather/scatter instructions, but also the indirect vector loads/stores. The vector registers are dynamically reconfigurable in the sense that choices range from many small vector registers to a few large ones, which gives flexibility for vector register allocation.

All vector data on the S-810 [Nagashima84] have a "data valid" signal for each element. An element synchronizing circuit, belonging to each functional unit, detects the availability of the needed vector elements. This one data valid signal per element scheme accomplishes chaining between instructions with different (and even irregular) execution times. In particular, memory accessing activities and CPU operations can be chained together.

The primary limitations of the commercially available supercomputers are the following.

(1) The vector registers are used by the vector instructions only. Thus, it is impossible to use elements of a vector register by several scalar instructions.

(2) The instruction set must include many vector macros to capture most hard-to-vectorize operations, such as inner product.

The vector extension described in chapter 7 includes some of the properties of the commercially available vector supercomputers. This extension also remedies the two limitations just mentioned.

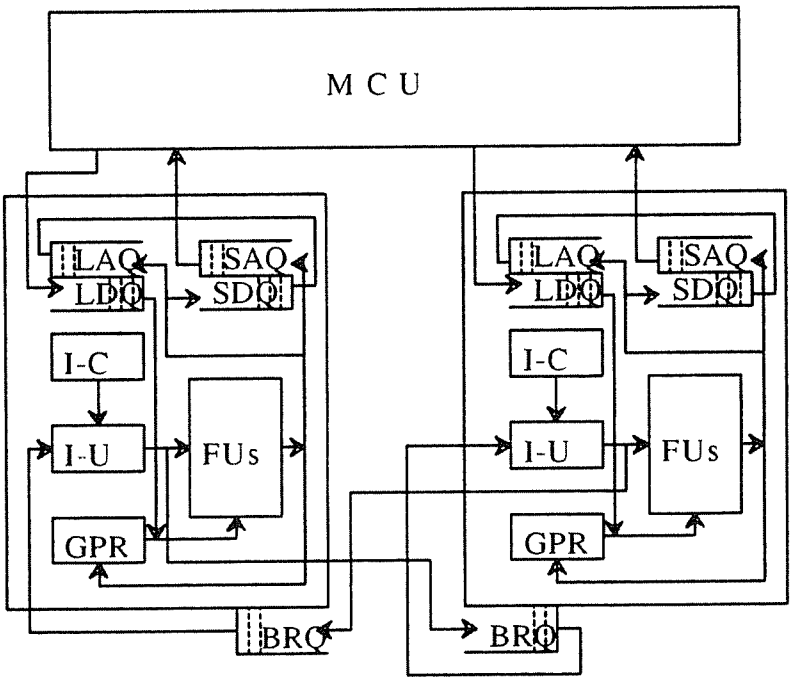
Chapter 3

The PIPE Architecture

In this chapter, we briefly describe the relevant details of the PIPE architecture to make this thesis self-contained. A more complete description is available elsewhere [Smith83] [Craig83] [Goodman85].

PIPE uses two identical co-processors that communicate via hardware queues (see Figure 3.1). Each of PIPE's processors is capable of executing an entire program by itself (in SP--single processor--mode), or the two processors may work together on a program (in access/execute mode). The instruction set for PIPE in SP mode is comparable in style to the CDC-6600 [Thornton70], with the addition of the architectural queues and the prepare-to-branch instruction (to be covered in later sections). The tasks run on the two processors are quite different, so that a viable system would almost certainly have two, quite different processors for the two functions. Under the decoupled execution mode, two identical processors are used for several reasons. This approach considerably reduced the work involved in building such a system; but more importantly, it provided us a way to evaluate the effectiveness of the decoupled approach (*e.g.*, the performance of the AE mode can be compared with that of the SP mode). We will first define the single processor architecture, and then the extensions necessary to support decoupled operation.

One decision that was made early in the design was that pipeline interlocks should be in hardware. This is in contrast to the MIPS project [Hennessy81], where the interlocks internal to the CPU are in software. By designing an architecture for pipelining implementation, we assure that nearly all interlocks involve register use, and that they can be resolved at one point in the instruction pipeline



Terms:
LAQ: Load Address Queue
LDQ: Load Data Queue
SAQ: Store Address Queue
SDQ: Store Data Queue
I-C: Instruction Cache
I-U: Issue Unit
GPR: General Purpose Registers
FUs: Function Units
BRQ: Branch Queue
MCU: Memory Control Unit

Figure 3.1. PIPE: A Decoupled Architecture.

with relatively simple logic. In other words, it is possible to generate code with software interlocks for non-memory activities [Hennessy83], but we feel that it is not necessary to do so; hardware interlocks in a well-designed architecture are straightforward to implement.

3.1. Load/Store Instructions

The interface between a processor and the memory subsystem is a set of architectural queues: the Load Address Queue (LAQ), the Load Data Queue (LDQ), the Store Address Queue (SAQ), and the Store Data Queue (SDQ). When a LOAD instruction is executed, the specified address is put on the Load Address Queue. The requested operand will be returned via the Load Data Queue as its last element. An additional instruction is needed to move an operand to one of the general purpose registers if the compiler decides to keep that operand in a register. A STORE operation is accomplished by the following two primitive instructions: (1) the store address is put on the Store Address Queue, and (2) the data is put on the Store Data Queue. The order of these two instructions is irrelevant, and other instructions may occur between them, including multiple instances of the first instruction. Having a Store Address Queue makes code scheduling more flexible than in conventional architectures, because a STORE instruction can be issued before the corresponding result has been computed. In most conventional architectures, the STORE instruction cannot be issued until the evaluation of the result has been completed.

Among all the queues, the Load Data Queue is of particular interest. In general, the execution time of a memory reference instruction (LOAD/STORE) is longer than that of a register-register instruction. In particular, the delay due to a load in a pipelined computer has great influence on the performance of the entire system. The LDQ is a mechanism for separating the request of an operand (*i.e.*, the LOAD instruction) from its use, thus reducing the urgency of memory read operations. Throughout this thesis, the LOAD instruction is used to represent memory accessing operations, and the LDQ is used to indicate relevant queues, whenever appropriate. The LDQ has the following advantages:

(1) It serves as a window between the CPU and the memory system. One can view the LDQ as a queue of registers in addition to the general purpose registers.

(2) There is no need to allocate registers for *use-once operands.* The use-once operands are taken out of the LDQ directly. In many programming environments, most operands are used either once or repeatedly. A smart compiler should allocate registers only for the latter type of operands. An additional move instruction is needed for operands which are allocated registers.

(3) Data can be requested several cycles before it is needed. The code scheduler can reorder the code to overlap the data transmission and access time with the execution of other instructions. This overlap is achieved by moving instructions between the LOAD instruction and the instruction that uses the operand.

(4) It is more flexible for the code scheduler to schedule the code when the LDQ is present than when the scheduling is limited to a set of general purpose registers. There is no data dependence, at the register lever, between use-once operands (except that they must be fetched in the order they are used). The same variable for different iterations can be fetched without register conflicts. The software pipelining mechanism, to be described in chapter 5, is an automatic way of prefetching operands across basic block boundaries.

(5) The slowness and irregularity of memory accesses can be hidden, and bursts of memory requests can be smoothed out.

(6) At issue time, LOADs need not reserve an input path to the register file. Ordinarily, this path is reserved, at instruction issue time, for use during the clock period when the instruction is completed. LOAD instructions pose a problem, however, because they may have unpredicatable completion times due to variations in memory access time. Hence, they either need their own path into the

register file (which then must be able to handle two writes [to the register file] simultaneously), or they must reserve the result path later in their execution, when the time of the data's arrival from memory has been determined. The LDQ provides a dedicated path for load data (which do not go into the register file except when an instruction specifically moves them there), so we can resolve interlocks during the one-clock-period instruction-issue time.

The major disadvantage of the LDQ is that processor deadlock is possible for a queue of finite length. The code scheduler must guarantee a deadlock-free code sequence. Also, twice-used variables are relatively expensive.

These architectural queues serve as elastic coupling between processors during the decoupled execution mode. An *alternative* LOAD instruction makes a memory request in one processor (normally, the A-processor), and the data is placed in the LDQ of the other processor (normally, the E-processor). Similarly, a store address generated by an *alternative* STORE instruction is paired with an operand from the other processor's SDQ.

3.2. Branch Instructions

In a pipelined architecture employing condition codes, many instructions may be in process at the same time, so careful bookkeeping is necessary to assure that the most recent condition code values are used in determining a conditional branch (*e.g.*, [Anderson67]). In PIPE, the instruction set includes a set of conditional branch instructions that explicitly test the sign and value of a register. Thus, one may consider that either PIPE has no condition codes, or PIPE has one condition code for each register; but PIPE does not have a shared condition code register. Thus, the hazard conditions of testing/setting the condition code are converted into testing hazard conditions when reading/writing the general purpose registers. Hazards on the registers have to be detected anyway.

In general, a branch instruction can be subdivided into three parts:

(1) Calculate the branch target address.

(2) Determine the branch outcome.

(3) Transfer control (if branch is taken).

One of the major goals in designing a CPU pipeline is to ensure a steady flow of instructions to the issue logic [Lee84]. It is well known [Anderson67] [Schorr71] [Flynn72] [Riseman72] [Smith81] [Lee84] that in a highly parallel computer system, branch instructions generally break the smooth flow of instruction fetching and execution. This results in delay, because a successful branch (a branch that is taken) changes the location of instruction fetches, and because the issuing of instructions must often wait until conditional branch decisions are made. Even an unsuccessful branch (a branch not taken) or an unconditional branch usually interrupts the smooth flow of instructions.

PIPE was defined to allow the separation of these three parts. The first part is actually performed by a separate instruction so that it can be moved outside a loop. The branch target address is stored in a branch register via a move instruction. (In PIPE, in addition to the general purpose registers, there is a set of branch registers. The branch registers are used to hold the branch targets.) The last two parts are a single instruction, but the effects are separated in time. The *prepare-to-branch* (PBR) instruction decides the branch outcome and specifies a delay (in terms of number of subsequent instruction parcels[1] to be issued) before transfer of control. The number of instruction parcels that should be executed unconditionally after a PBR is specified by a field of the PBR instruction (called the *branch count* (BC) field). The transfer of control occurs after BC instruction parcels following the

[1] For the current implementation, an instruction parcel is a 16-bit quantity. Instructions are one or two parcels.

PBR have been issued. We insert a pseudo-instruction, XBR (denotes the eXit point of the BRanch instruction), at the assembly language level to indicate the exit point. That is, the transfer of control for a successful branch takes place at the XBR. The prepare-to-branch instruction is simply a generalized form of the delayed branch (or branch with execution) used in many machine organizations (*e.g.*, 801 [Radin82], MIPS [Hennessy81], RISC [Patterson81] [Patterson82]). In the IBM 801 [Radin82], no instructions will be unconditionally executed after a *normal* branch instruction. Its issue logic, however, always initiates the instruction that has been placed immediately after a *branch with execute* instruction. Thus, the 801 permits the flexibility of allowing up to one instruction to be unconditionally executed after a branch instruction.

One of the reasons that the 801, RISC, and MIPS do not include larger branch delays may be that there are few stages in their execution pipelines. Short pipelines imply that branch instructions finish in very few clock periods--possibly one. The execution time of a (successful) branch instruction is the time from the issuing of that instruction to the issuing of the first instruction from the branch target. When there is only one stage in the pipeline (the special case of a non-pipelined machine), a branch delay of zero is always optimal because a branch instruction finishes in the same clock period it is issued. In PIPE, however, several clock periods are required to complete a prepare-to-branch instruction. This is due to PIPE's fast clock rate and (relatively) long pipelines. (For comparison, the execution time in the Cray-1 for a successful branch ranges from 5 clock periods (branch address is in a buffer) to 14 clock periods (branch address is not in a buffer) given that two parcels of the branch instruction are in the same instruction buffer [Cray82].) Two instructions, called the *subject instructions*, are always executed after a branch instruction in FOM [Brantley82]. The branch instruction in FOM indicates that there are cases where more than one useful instruction can be moved to the

locations after a branch instruction.

The advantages of the PBR instruction are:

(1) It is a mechanism to separate the branch decision from the transfer of control. Thus, it is possible to do *guaranteed pre-decoding* in the sense that the pre-decoded instructions are always executed. In the case of an instruction cache miss, the cache controller can do *guaranteed (cache) prefetching*.

(2) There is no penalty on the program size in using this generalized delayed branch. That is, it is never necessary to insert NOPs after the PBR instruction. For some architectures with a delayed branch (*e.g.*, RISC-I [Patterson81]), the number of instructions that must be executed after a delayed jump is fixed. Under certain circumstances, NOPs have to be inserted after the delayed jump instructions.

(3) The hardware will not fetch instructions that will not be executed.

(4) Most branch targets, stored in the branch registers, are loop invariants. Thus, standard optimization techniques [Aho77] can be used to move the calculation of branch targets and assignment of (loop-invariant) branch registers out of the loops.

For decoupled execution, the two processors must make the same decision on corresponding branches. To assure this, PIPE has two sets of branch instructions: *internal* branches that determine instruction execution only for the processor in which they are executed, and *external* branches that behave just like internal ones, but that also notify the other processor of the branch outcome, via a branch queue. Both processors also have a "prepare-to-branch-from-queue" instruction that checks the head of the branch queue (which is loaded by the other processor by executing an external branch) to see if a branch should be taken.

PIPE does not have a single instruction for call or return; rather, it provides several primitives from which a compiler may generate code specialized for any particular language or situation. In particular, PIPE uses the (*un*conditional) prepare-to-branch instruction for both call and return.

Chapter 4

The Code Scheduler

4.1. The PIPE Pascal Compiler

Currently, there is one high level language compiler for the PIPE architecture. The source language is a subset of Pascal. (The major unimplemented features are formal procedures, and "**goto** out of procedure.") The compiler has five phases. Phase one generates intermediate language (IL) in quintuple form, which is the quadruple form [Aho77] augmented with a fifth attribute-tuple. The attributes are used to keep track of the original high level language structures. Phase two performs data flow optimizations [Aho77] on the IL. Phase three schedules the IL. Phase four generates machine code. Phase five does the machine code level scheduling to take advantage of the architectural queues and the prepare-to-branch instruction.

We do two-level scheduling for the following reasons:

(1) There are fewer structural dependencies at the IL level than at the machine code level. Since we assume an infinite number of pseudo registers at the IL level, all pseudo registers are assigned at most once. (We shall see in chapter 5 that the software pipelining method duplicates some of the code within the body of a loop. If we do software pipelining at the IL level, a given pseudo register may be systematically assigned more than once in different basic blocks.) Thus, read-after-write (RAW) and write-after-read (WAR) hazards due to register allocation are deferred until register binding time. The register allocation algorithm can take the variable hazards into account and allocate registers accordingly.

(2) The time complexities are $O(n\log n)$[†] and $O(n)$ for scheduling methods at the IL level and the machine code level, respectively. The time complexity of one-level machine code scheduling is $O(n^2)$ (see section 4.4). Under the two-level scheduling scheme, the sequence of operand requests is reordered at the IL level. Thus, there is no need to change the order of memory-accessing instructions at the machine code level.

Although the IL was designed with the PIPE architecture in mind, the novel features of PIPE are not expressed explicitly. The code generated by the front-end, however, must meet the following criteria to allow effective scheduling of the IL.

(1) The branch condition has to be in a pseudo register. That is, the branch condition is computed before the branch instruction itself. Thus, it is possible to separate the evaluation of the branch condition from the branch instruction.

(2) The index to access an array element has to be in a pseudo register, rather than in a program variable. Thus, it is possible to separate the index calculation from the array element accessing.

(3) If any of the source operands of an arithmetic/logic operation are in memory, the destination must be in a pseudo register to facilitate the scheduling of memory accessing operations. (That is, the compiler won't generate memory-memory arithmetic/logic expressions.) An additional MOVE intermediate language statement moves the contents from the pseudo register to its memory location.

A complex (high level language) expression is subdivided into multiple simple IL instructions; each of them is either a binary or a unary operation. Thus, independent complex expressions may be scheduled to execute in an interleaved

---

[†] Base 2 logarithm is used through out this thesis.

fashion. The code scheduler at the IL level makes a simple assumption about real variables (as opposed to pseudo registers): that one object is not referenced by two or more different names. We also assume that an array, not each element of an array, is an object. Thus, we treat $A[i]$ and $A[j]$ as the same object, regardless of the relationship between i and j. That is, we ignore the aliasing problem for the purpose of this study. Some of the restrictions mentioned above can be lifted by using sophisticated algorithms, such as the disambiguator described by Fisher, *et. al.* [Fisher84a].

4.2. Code Scheduler for the Intermediate Language

The code scheduling method used at the IL level is explained in this section. This scheduling method is similar to that at the machine level. We show the function of the scheduler by an example.

4.2.1. The Algorithm

> *Definition* 4.1. A *weighted dependency graph* (WDG) G = $\{V,E,T,C\}$ is a weight directed acyclic graph where (a) V is a set of vertices corresponding to instructions, (b) E is a set of edges expressing data/control dependencies between pairs of vertices, (c) T is a function mapping from E to a set of non-negative integers, and (d) C is another function mapping from V to a set of non-negative integers. □

We will use the terms "vertex" and "the instruction represented by that vertex" interchangeably, when the meaning is clear from the context. The edges are the different dependencies (to be defined later) between vertices. An edge e_{ijk} is the k-th dependency from vertex v_i to v_j. If there is an edge e_{ijk} pointing from vertex v_i to vertex v_j the value t_{ijk} associated with the edge is the time[1] from the issuing of v_i

[1] This is an estimated time. It is very difficult, if not impossible, to get the actual execution time of each instruction because of some asynchronous operations (*e.g.*, memory bank conflicts, cache misses).

till the k-th dependency of v_i has been lifted. Note that it is possible to have more than one edge between a pair of vertices. Each edge represents a different dependency, and the costs with these edges may vary. All dependency edges point forward in the sense that a given instruction depends only on instructions that appear earlier in the textual order. The cumulative cost c_i associated with the vertex v_i is the minimum amount of time from the issuing of v_i to the end of the WDG.

Definition 4.2. Given two vertices (instructions) v_i and v_j in a weighted dependency graph, and v_i precedes v_j in the original sequence.
(1) V_j is *RAR* (read-after-read) dependent on v_i iff both instructions get values from the same object.
(2) V_j is *RAW* (read-after-write) dependent on v_i iff v_i assigns a value to an object and v_j gets a value from the same object.
(3) V_j is *WAR* (write-after-read) dependent on v_i iff v_j assigns a value to an object and v_i gets a value from the same object.
(4) V_j is *WAW* (write-after-write) dependent on v_i iff both instructions assign values to the same object.
(5) V_j is *CON* (control) dependent on v_i iff v_i must precede v_j logically.
□

The objects at the IL level, such as variables in a program and pseudo registers, are independent of the organization of the processor. The objects at the machine code level, however, are processor resources, such as registers, queues and the memory system. It is relatively difficult to distinguish one arbitrary memory element from another at the machine code level. Hence, the memory system, rather than the elements of the memory system, is considered a single object. This constraint will not affect the code quality reordered by the machine level scheduler because the IL scheduler has put all memory references in the proper order.

For a basic block which ends with a branch, the branch has to be the last instruction, even after the scheduling. Thus, there are control dependencies between all other instructions in such a basic block and branch. In other words, a control transfer point induces a control dependency between instructions before and after it. As mentioned before, we introduce a pseudo instruction XBR as a place

holder for the exit point of the prepare-to-branch instruction. At the machine language level, all other instructions within the same basic block are *CON* dependent on the XBR pseudo instruction (not the PBR instruction). Thus, no instructions will be moved to follow the XBR instruction. Instructions, however, may be moved to follow the PBR. In fact, one of the purposes of the machine code scheduler is to move suitable instructions to follow the PBR instruction.

RAR and WAW dependencies are for the machine code scheduling method only, for the reasons explained below. In general, an object can be read many times and still hold the same value. Hence, the order of reading from that object is irrelevant. Therefore, RAR does not exist between objects at the IL level. There is at least one exception in that if the object is a queue, the order of reading is important. Suppose both v_i and v_j read from the same queue and v_i precedes v_j. Then v_i gets the first element of that queue where v_j gets the second one. The effect is different if we exchange the order of v_i and v_j. The same argument holds for WAW hazard on writing to a queue. If an object at the IL level is assigned twice without being used between the two assignments, the first assignment is useless, hence is removed (by the compiler). That is, the WAW hazard does not exist at the IL level, either. Consequently, we don't have to check for RAR and WAW dependencies for scheduling done at the IL level.

We mentioned in chapter 2 that code scheduling is *NP*-complete. Thus, heuristics are used in developing the following algorithms. In algorithm 4.1, we outline the code scheduling method at the IL (intermediate language) level.

Algorithm 4.1. Intermediate Language Code Scheduling.
foreach basic block **do**
(4-1-1) build a data/control dependency graph;
(4-1-2) assign cost to each instruction in reversed topological sort order;
(4-1-3) issue instructions according to *weighted topological sort order*;
od;

□

The steps in algorithm 4.1 are explained by the following three algorithms.

Algorithm 4.2. Build Data/Control Dependency Graph.
foreach instruction in the original textual order **do**
(4-2-1) add a vertex corresponding to this instruction to the dependency graph;
(4-2-2) **forall** objects the current instruction defines (writes to) **do**
(4-2-2-1) add a WAR edge from the last previous instruction that uses (reads) the object to the current instruction;
(4-2-2-2) assign a cost to the edge;
od;
(4-2-3) **forall** objects the current instruction uses (reads from) **do**
(4-2-3-1) add a RAW edge from the last previous instruction that defines (writes) the object to the current instruction;
(4-2-3-2) assign a cost to the edge;
od;
(4-2-4) **if** the last instruction of the current basic block is a branch instruction **then**
(4-2-4-1) add a CON edge from the current instruction to the branch instruction;
fi;
od;

□

The following lemmas state some facts about the algorithms shown above. These lemmas will be used to prove some theorems of the code scheduling method.

Lemma 4.1. There is a one-to-one correspondence between input instructions and vertices of the dependency graph.

Proof. From statement (4-2-1).

□

Lemma 4.2. All dependencies, at the IL level, are included in the dependency graph constructed by Algorithm 4.2.

Proof. From Definition 4.2 and statement (4-2-2), (4-2-3), and (4-2-4).

□

For the discussion of this section, we assume the current basic block size is n. That is, there are n instructions in the current basic block.

Lemma 4.3. The time complexity of Algorithm 4.2 is linear in the size of input.

Proof. Since an instruction defines only a (small) finite number of objects, the maximal number of objects an instruction can define is dependent on the design of the instruction set but independent of the input program size. Let the maximum numbers of objects that can be defined and used by an instruction be D and U, respectively. Thus, statement (4-2-2) is executed at most $D \cdot U$ times per iteration. Similarly, statement (4-2-3) is executed at most $D \cdot U$ times per iteration. Statement (4-2-4) is executed once per iteration. A scheme similar to hashing can be used to find the last previous definition (use) of an object in constant time. Let C_s be the time required to find the last previous define (use). Therefore, the running time of Algorithm 4.2 is bounded by $(2 \cdot D \cdot U \cdot C_s + C_c) \cdot n$, where C_c is the time to add a control dependency. In other words, the time complexity of Algorithm 4.1 is $O(n)$.

□

For practical purposes, the values of D and U are small. In a typical three-address form, the values for D and U are 1 and 2, respectively. Thus, $D \cdot U$ is still a very small number.

Algorithm 4.3. Assign Cost to a Dependency Graph.
{ The cumulative cost of an instruction v_i is designated by $C(v_i)$. }
(4-3-1) **foreach** instruction v_i **do**
(4-3-1-1) $C(v_i) \leftarrow 0$;
od;
(4-3-2) do the topological sort;
(4-3-3) **foreach** instruction v_i in the reversed topological sort order **do**
(4-3-3-1) **foreach** edge e_{ij} pointing out from v_i to v_j with cost $T(e_{ij})$ **do**
(4-3-3-1-1) $C(v_i) \leftarrow Max2(C(v_i), C(v_j) + T(e_{ij}))$;
{ Max2 returns the larger one of its two arguments }
od;
od;

□

The reason we use the reversed topological sort order is that the cumulative costs of instructions that depend on v_i are evaluated before we compute the cumulative cost of v_i . In other words, in statement (4-3-3-1-1), $C(v_j)$ is known and will not be changed when we are calculating $C(v_i)$.

Lemma 4.4. The time complexity of Algorithm 4.3 is linear in the size of input.

Proof. Statement (4-3-1) is executed n times. Except for the branch instructions, there are at most $D \cdot U$ edges pointing to any vertex. There are $n-1$

edges pointing to the branch instruction. Therefore, there are at most $((n-1)\cdot D\cdot U)+(n-1)$ edges in the WDG. The time to do the topological sort on the WDG is $O(|\,|V|+|E|\,|)$, hence $O(n)$ [Knuth73a]. The time to execute statement (4-3-3-1-1) is constant and statement (4-3-3-1-1) is executed at most $((n-1)\cdot D\cdot U)+(n-1)$ times. (Recall that $(n-1)\cdot D\cdot U)+(n-1)$ is the number of edges in the WDG.) Thus, the time complexity of (4-3-3) is also $O(n)$. Therefore, the time complexity of Algorithm 4.3 is linear in the size of input.

□

Definition 4.3. A *leader* of a WDG in a topological sort is a vertex with no predecessors. That is, either this vertex does not depend on any other vertices, or all its predecessors have been removed (issued).

□

Definition 4.4. A *weighted topological sort order* of a WDG is topological sort order subject to the following constraint. V_i and v_j are two vertices, if v_i precedes v_j, then $C(v_i)\geq C(v_j)$, where C is the cumulative cost function of a vertex.

□

Algorithm 4.4. Issue Instructions by Weighted Topological Sort Order.

(4-4-1) calculate the leader set;
(4-4-2) **repeat**
    (4-4-2-1) pick up an instruction (v_i) from the leader set with maximum cumulative cost;
    (4-4-2-2) issue v_i;
    (4-4-2-3) remove v_i from the leader set;
    (4-4-2-4) insert new leaders to the leader set;
until the leader set is empty;
(4-4-3) **if** some instructions have not been issued **then**
    (4-4-3-1) error condition;
fi;

□

The error condition, (*i.e.*, statement (4-4-3)) should not happen at the IL level, because the original input is a legal sequence.

Lemma 4.5. The time complexity of Algorithm 4.4 is $O(n\log n)$.

Proof. The data structure used for the leader-set is an AVL tree [Knuth73b], which has $O(\log m)$ time complexity to insert, delete, and find the maximal element, where m is the size of the AVL tree. The size of the leader set is no bigger than n and each vertex (instruction) is put on the leader set only once. Similarly, a vertex is removed from the set only once. Thus, each of the statements (4-4-2-1), (4-4-2-3) and (4-4-2-4) is executed, at most, n times. The

original leader set can be built with time complexity $O(n\log n)$. Therefore, the time complexity of Algorithm 4.4 is $O(n\log n)$.

□

Lemma 4.6. If Algorithm 4.4 terminates successfully, there is a one-to-one correspondence between vertices of the WDG and the output instructions. In other words, the output instructions are a permutation of vertices of the WDG.

Proof. If Algorithm 4.4 terminates successfully, then all vertices have been issued. On the other hand, instructions issued by Algorithm 4.4 are members of the leader set which is constructed from vertices of the WDG.

□

Lemma 4.7. All dependencies represented by the WDG are preserved in the output instruction stream.

Proof. From the properties of topological sort.

□

Theorem 4.1. The time complexity of Algorithm 4.1 is $O(n\log n)$.

Proof. From Lemma 4.2, 4.3, and 4.4, the time complexity for one iteration of the loop in Algorithm 4.1 is $O(n\log n)$. Let n_i be the size of the i-th basic block and N be the size of input program. That is

$$N = \sum_i n_i$$

Thus, the time complexity of Algorithm 4.1 is

$$O(\sum_i n_i \log n_i) \leq O(\sum_i n_i \log N) = O(N \log N)$$

□

Definition 4.5. For two code sequences, C_i and C_j, C_i is said to be semantically equivalent to C_j iff C_i is a permutation of C_j and all dependencies in C_i are preserved in C_j.

□

Theorem 4.2. The scheduled code (the output of Algorithm 4.1) C_{sch} is semantically equivalent to the original code (the input to Algorithm 4.1) C_{in}.

Proof.
If-part:
[C_{sch} is a permutation of C_{in}] From Lemma 4.1 and Lemma 4.6.
[All dependencies in C_{in} are preserved in C_{sch}]: From Lemma 4.2 and Lemma 4.7.
Only-if-part: Similar to that of the if-part by introducing comparable lemmas.

4.2.2. Examples

For the discussion of examples in this thesis, we assume the following timing for the relevant functions.

Function	Time (clock periods)
Load from Memory	6
Store to Memory	1
Branch (taken)	6
Integer Add	1
Floating Point Add	4
Floating Point Multiplication	4
Load Branch Register	1

It may take more than one clock period for an operand to be written to the memory system. As far as the issue logic is concerned, however, a store operation does not cause any subsequent instructions to be blocked owing to data dependency. For a successful branch, the time for a branch is the number of clock periods required from the issue time of the branch instruction to the issue time of the first instruction of the branch target. We also assume that the issue logic is capable of issuing one instruction every clock period, providing there are no dependencies between instructions.

The primary purpose of the IL level code scheduling is to arrange all memory accessing instructions (LOADs/STOREs) in the desired order. The compiler keeps enough semantic information at the IL level to distinguish one object (variable) from another. The entire array, rather then elements of an array is considered as one object. We do not intend to include many architectural features within the IL. In particular, the concepts of prepare-to-branch and data queues are not delineated. We shall see in the next section that the code scheduler at the machine code level

takes advantage of these special architectural features.

4.3. Code Scheduler at the Machine Code Level

The algorithm used for the machine code level scheduler is similar to that used in the IL level scheduler. The major differences are (a) all objects are processor resources, such as registers; (b) RAR and WAW hazards are used to enforced the first-in-first-out (FIFO) order of queues for the reasons explained earlier; and (c) an additional restriction (described below) is imposed for a queue-filling instruction (an instruction that puts an element on a queue) to become a leader, viz., this queue-filling instruction should not overflow the queue. The last difference guarantees that the scheduled code is deadlock-free. It is, however, possible to have an error condition at the machine code level because of different assumptions about queue sizes. For instance, the code scheduler may not be able to find a code sequence for queue size of one, if the original code is generated assuming queue size of two. The functions of the machine code scheduler are: (1) to utilize the architectural queues; (2) to utilize the prepare-to-branch instruction; (3) to enforce the FIFO nature of the queues; and (4) to guarantee a deadlock free code for queues of finite size. From the different constraints at the machine code level, we have the following lemma and theorem.

Lemma 4.8. The time complexity of Algorithm 4 at the machine level is linear in the size of input.

Proof. The number of elements in a leader-set, for the scheduling method at the machine code level, is dependent on the number of objects (thus processor resources) in the system but independent of the input program size. Thus, the size of the leader-set in Algorithm 4.4 is bounded by a constant which depends only on the processor organization. Therefore, it takes constant time to insert an element, delete an element, and find the maximal element from the leader set. We have to do the insert/delete/find operations at most n time in Algorithm 4. Hence, the time complexity of Algorithm 4 is linear in the size of input program.

□

Theorem 4.3. The time complexity of scheduling method used at the machine language level is linear in the size of the input program.

Proof. From Lemma 4.3, 4.4, 4.8, and superposition.

□

The weighted topological sort method described in Algorithm 4.4 does not take the Flynn limit [Flynn66] [Smith84] [Goodman85] into account. Consequently, there are cases where many independent instructions are crowded waiting to be issued near the end of a basic block. This clustering of instructions results in unnecessarily prolonged execution time due to the Flynn limit. A simple scenario of this situation follows. Suppose there are five independent instructions with cumulative costs of two, and the cumulative costs of any other instructions are larger than two. Hence, the aforementioned instructions will be the last ones to be issued according to Algorithm 4.4. The relative issuing order of these five instructions, however, is unimportant. As mentioned above, the cumulative cost of a given instruction is the least time (in terms of clock periods) from issuing of that instruction to the end of the basic block. Thus, the best case is to finish the basic block two clock periods after issuing any of these five instructions. The underlying hardware, confined by the Flynn limit, requires 6 clock periods to complete all five instructions, which is 4 clock periods longer than we would like it to be. Some of these five instructions can be issued earlier if a slightly different scheduling method is used.

We introduce a modified scheduling method to take the Flynn limit into account. The basic idea behind the *modified weighted topological sort order* is explained here. Consider two consecutive instructions in the weighted topological sort order with cumulative costs of n and $n-k (k \geq 1)$, respectively. It is possible to issue $k-1$ other instructions, between these two instructions, from the leader set without increasing the total execution time of the basic block. In other words, $k-1$ *free* time slots are available to issue instructions with smaller cumulative costs

without execution time penalty. We then introduce another attribute to an instruction, called the *earliest issue time* (EIT). As the name suggests, the earliest issue time of an instruction indicates the earliest possible issue time of that instruction given the issue times and execution times of all instructions it depends on. Consider three instructions, v_i, v_j, and v_k, and the following conditions: (a) v_k depends on both v_i and v_j; (b) the execution times of v_i and v_j are T_i and T_j, respectively; and (c) v_i and v_j are issued at it_i and it_j, respectively. The EIT of v_k, denoted as EIT_k, is set to $max2((it_i + T_i),(it_j + T_j))$. That is, the underlying hardware interlock mechanism will block the issue of v_k until EIT_k. When an instruction is issued, the EITs of all it successor instructions will be updated accordingly. The EITs of different instructions are used in Algorithm 4.6 (SN_issue) to decide which instructions can be issued slightly out of the weighted topological sort order without increasing the total execution time. The modified topological sort order, which takes into consideration the bandwidth of the issue logic, is detailed in Algorithm 4.5 4.5.

Algorithm 4.5. Issue Instructions by Modified Weighted Topological Sort Order.

(4-5-1) calculate the leader set with the earliest issue time being zero;
(4-5-2) p_cc ← 0; { previous cumulative cost }
(4-5-3) c_cc ← 0; { current cumulative cost }
(4-5-4) **repeat**
    (4-5-4-1) pick up an instruction (v_i) from the leader set with maximal cumulative cost;
    (4-5-4-2) p_cc ← c_cc;
    (4-5-4-3) c_cc ← $\bar{C}(v_i)$; { the cumulative cost of v_i }
    (4-5-4-4) **if** (p_cc − c_cc) ≥ 2 **then**
        (4-5-4-4-1) SN_issue (p_cc − c_cc − 1, v_i);
    else
        (4-5-4-4-2) issue v_i;
        (4-5-4-4-3) update the EITs of all successor instructions of v_i;
        (4-5-4-4-4) remove v_i from the leader set;
        (4-5-4-4-5) insert new leaders to the leader set;
    fi;
until the leader set is empty;
(4-5-5) **if** some instructions have not been issued **then**

(4-5-5-1) error condition;
fi;

□

SN_issue issues at most n instructions out of the weighted topological sort order in the n free time slots introduced by dependencies.

Algorithm 4.6. SN_issue (n : integer; v : vertex);
{ issuing at most ''n'' instructions, other than ''v,'' according to the EIT }
{ ''v'' must be in the leader set }
(4-6-1) **while** (n > 0) **and** (|leader set| > 1) **do**
(4-6-1-1) pick up an instruction (v_j) from the leader set with maximal EIT;
(4-6-1-2) **if** $EIT_{v_j} > EIT_v$ **then**
return;
{ no instruction can be issued earlier than v can }
(4-6-1-3) **if** ($v_j \neq$ v) **then**
(4-6-1-3-1) issue v_j;
(4-6-1-3-2) update the EITs of all successor instructions of v_j;
(4-6-1-3-3) remove v_j from the leader set;
(4-6-1-3-4) c_cc ← $C(v_j)$; { cumulative cost of v_j }
(4-6-1-3-5) n ← n − 1;
(4-6-1-3-6) insert new leaders to the leader set;
fi;
od;

□

In Algorithm 4.6, when the size of the leader set is equal to one, the only element remaining in the leader set is the argument ''v.''

Many variations of SN_issue are possible. We will not discuss alternative versions of SN_issue in this thesis. The Flynn limit applies only to the hardware instruction issue unit. Thus, we have to apply this modified method to the machine code level scheduler alone. The bounded leader set assumption, used in Lemma 4.8, still holds. Therefore, we have the following theorem.

Theorem 4.4. The time complexity of the modified Algorithm 4.4 at the machine code level is linear in the size of input.

Proof. The proof is similar to that of Lemma 4.8 and Theorem 4.3.

□

With the exception of the result bus (the bus that sends the result to the register file), the pipeline in the PIPE architecture is almost linear in the sense that new operands can be put into the pipe every clock period in the absence of a result bus conflict. The result bus is used at the end of an instruction that sends the computed value to the register file. Thus, two instructions with different execution times may want to use the result bus at the same time. In PIPE, this conflict is detected by hardware, and subsequent instructions will be blocked to avoid result bus conflict. Two possible hardware solutions, which can alleviate the delay due to a result bus conflict, are described below. The first method is including multiple result buses. The second method uses the issue logic to detect the result bus conflicts at program execution time and inserts delays whenever necessary. This is similar to a scheme proposed by Patel and Davidson [Patel76]. Normally, a bus conflict will block the issuing of an instruction for one clock period. Thus, the delay due to a result bus conflict with a simple blockage scheme is relatively smaller for long pipelines (pipelines with many stages) than in the case of short pipelines. The detection of possible result bus conflicts at code scheduling time, however, can be used to break the tie when two instructions in the leader set have the same cumulative costs. That is, when two instructions in the leader set have the same cumulative cost, the one that does not cause a result bus conflict with previous instructions should be taken out of the leader set (*i.e.*, issued) first.

We will demonstrate the function of code scheduling by an example.

Example 4.1. We will use the following example [Hwang84] in chapters 4 and 5:

```
      DO 999 I = 1, 1000
999         Y(I) = F(I) × Y(I-1) + G(I)
```

This loop is hard to vectorize on some supercomputers because of the first order linear recurrence. Some notations used in the IL are outlined below: (a) %R*n* stands for the *n*-th pseudo register at the IL level; (b) $x[y]$ denotes an access to array x with offset y. The compiled IL before scheduling is:

```
        MOV     %R4,1000,,          /* loop bound
        MOV     %R2,y-1[%R1],,M     /* Y(0)
LOOP
        MULF    %R3,f[%R1],%R2,     /* F(I)×Y(I-1)
        ADDF    %R2,%R3,g[%R1],     /* ...+G(I)
        MOV     y[%R1],%R2,,        /* store the result
        ADDI    %R1,%R1,1,          /* increment loop count
        BRLE    %R1,%R4,LOOP,       /* check loop bound
```

The code generated from the above IL is:

```
        ....
        R0      ←   1000            /* loop bound
        BR0     ←   LOOP            /* branch target
        LDQ     ←   R1, y-1         /* load Y(0)
        R3      ←   LDQ             /* move Y(0) to R3
LOOP
S1      LDQ     ←   R1, f           /* load F(I)
S2      R2      ←   LDQ ×f R3       /* F(I)×Y(I)
S3      LDQ     ←   R1, g           /* load G(I)
S4      SAQ     ←   R1, y           /* addr of Y(I)
S5      R3      ←   R2 +f LDQ       /* ...+G(I)
S6      SDQ     ←   R3              /* value for Y(I)
S7      R1      ←   R1 + 1          /* increment loop count
S8      R4      ←   R1 − R0         /* test bound
S9      PBRLE       R4, BR0, 0      /* branch back
S10     XBR
```

The issue time and the completion time of the above code sequence are:

Statement Number	Issue Time	Completion Time
S1	0	6
S2	6	10
S3	7	13
S4	8	9
S5	13	17
S6	17	18
S7	18	19
S8	19	20
S9	20	*26

As a convention used in this thesis, the number following an asterisk in the completion-time column indicates the effective execution time (in terms of clock periods) per iteration. Since the loop is so simple, the scheduling method at the IL level will not change the order of memory accessing

instructions. The *WDG* of the machine code is shown in Figure 4.1. The number associated with an edge (in boldface) is the execution time (in terms of clock periods) of that particular instruction, where the number of an instruction (in italic) is the number of clock periods from the issuing of that instruction to the end of the basic block. The scheduled code of the machine code is

```
        ....
LOOP
        S1    LDQ     ←   R1, f
        S3    LDQ     ←   R1, g
        S4    SAQ     ←   R1, y
        S7    R1      ←   R1 + 1
        S8    R4      ←   R1 − R0
        S9    PBRLE         R4, BR0, 2
        S2    R2      ←   LDQ ×f R3
        S5    R3      ←   R2 +f LDQ
        S6    SDQ     ←   R3
        S10   XBR
```

S2 cannot be issued until 6 clock periods after S1 has been issued. Though S7, S8 and S9 have smaller cumulative costs than that of S2, they (S7, S8 and

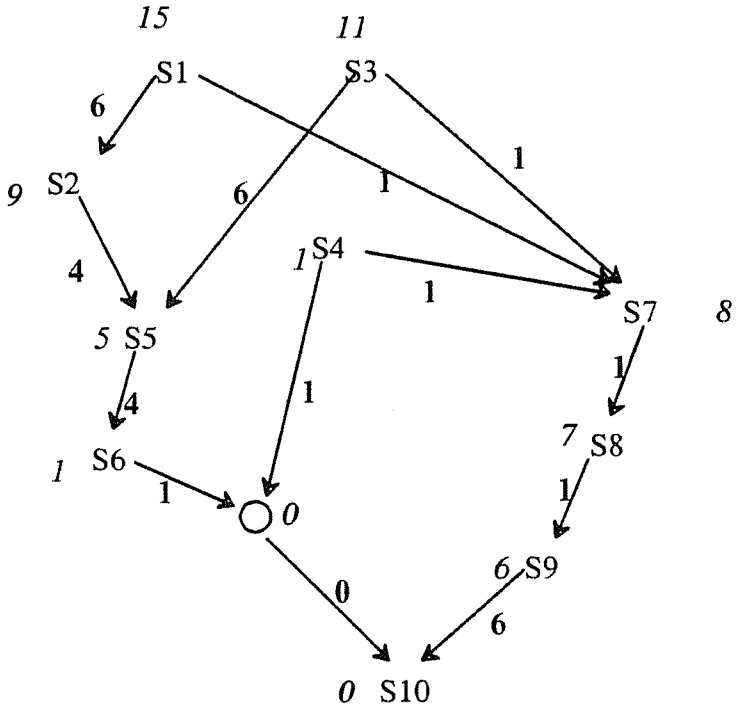


Figure 4.1. The *WDG* of Example 4.1.

S9) are moved to the locations before S2 by SN_issue. The issue time and completion time of the code sequence are

Statement Number	Issue Time	Completion Time
S1	0	6
S3	1	7
S4	2	3
S7	3	4
S8	4	5
S9	5	11
S2	6	10
S5	10	14
S6	14	*15

It takes 15 clock periods to execute one iteration of the loop.

□

4.4. One-Level Code Scheduling

As long as the high level language is available, the two-level scheduling methods proposed are applicable. If we must schedule an assembly language program to take advantage of architectural features, a more expensive (in the sense of execution time) scheduling method must be used. The expense comes from the reordering of memory accessing instructions without deadlock. Since the compiler does not keep enough semantic information at the machine code level, the scheduler will only schedule the order of LOADs between two consecutive STOREs. A single level scheduling method is described below. For simplicity of discussion, we will only elucidate the scheduling of load instructions and instructions that consume operands from the LDQ. Without loss of generality, we assume that a given instruction takes either no operands or exactly m operands from the LDQ. We will use *queue-draining instructions* to refer to instructions that take operands from the LDQ. The corresponding load instructions are termed *queue-filling instructions.* We enumerate all queue-draining instructions from D_1 to D_n. Similarly, queue-

filling instructions associated with D_i are numbered from F_{i1} to F_{im}. That is, F_{ij} loads the j-th operand for the i-th queue-draining instruction. In order to enforce the FIFO nature of the queues without leading to deadlock, all queue-draining instructions have to be strung together by appropriate RAR dependency links. The corresponding queue-filling instructions are chained by suitable WAW dependency links. The FIFO nature is enforced by finding a total order of all queue-draining instructions. The corresponding queue-filling instructions are arranged accordingly. The goal of finding this total order is to minimize the execution time of a basic block subject to the system time constraints and dependencies.

Definition 4.6. A queue-draining instruction and its queue-filling instructions form a *FD-unit.* That is, $F_{i1}, F_{i2}, \cdots F_{im}$ and D_i form the i-th FD-unit of a basic block.

□

We observe the following:

(1) Since a queue-draining instruction takes m consecutive elements from the LDQ, queue-filling instructions from the same FD-unit do not intermix with queue-filling instructions from any other FD-units.

(2) The total order of queue-filling instructions of the i-th FD-unit is $F_{i1}, F_{i2}, \cdots F_{im}$. In other words, operands used by a given queue-draining instruction are always requested in the desired order.

Thus, the total order of all FD-units leads to the total order of queue-draining instructions as well as that of queue-filling instructions. The cost of the i-th FD-unit is the cumulative cost of D_i. That is, the minimum cumulative cost within that FD-unit. The following algorithm finds the total order of all FD-units.

Definition 4.7. Let $M_{i1}, M_{i2}, \ldots, M_{im}$, be members of a FD-unit FD_i and $M_{j1}, M_{j2}, \ldots, M_{jm}$, be members of a FD-unit FD_j. FD_j is dependent on FD_i iff there exist a x and a y, and M_{ix} is in FD_i and M_{jy} is in FD_j, such that M_{jy} transitively depends (*i.e.*, directly or indirectly depends) on M_{ix}. A *coalesced transitive closure* among FD-units is formed by the dependency relations

among all FD-units.

□

Algorithm 4.7. Find the Total Order of All FD-units.
(4-7-1) find the transitive closure of the WDG;
(4-7-2) find the *coalesced transitive closure* among all FD-units;
(4-7-3) build the FD-WDG according to the coalesced transitive closure;
{ the FD-WDG consists of all FD-units }
(4-7-4) do weighted topological sort on the FD-WDG;

□

The output order from Algorithm 4.7 is a legal total order among all FD-units. If FD_i and FD_j are two adjacent FD-units according to the total order and FD_i proceeds FD_j, the following dependencies are added to the original WDG.

(1) Add a WAW link between F_{im} and F_{j1}.

(2) Add a RAR link between D_i and D_j.

This algorithm can always find a legal total order among FD-units, because the original text order is a legal one.

Theorem 4.5. The time complexity of Algorithm 4.7 is $O(n^2)$.

Proof. Since there are $O(n)$ edges for a WDG of n vertices, the algorithm of finding the transitive closure for use with *sparse relations* [Hunt77] applies. The transitive closure of the original WDG can be computed with time complexity of $O(n^2)$. The size of any given FD-unit is bounded by a small constant. Thus, the dependency between any two FD-units can be computed in constant time. There are, at most, $O(n)$ FD-units for a basic block of size n. Therefore, the dependency relations among all pair of FD-units (*i.e.*, the coalesced transitive closure) can be computed with time complexity of $O(n^2)$. The FD-WDG can also be built with time complexity of $O(n^2)$. The weighted topological sort can be done in time complexity $O(n\log n)$. Hence, the time complexity of Algorithm 4.7 is $O(n^2)$.

□

The scheduling method of Algorithm 4.9 moves appropriate instructions, according to their cumulative costs, between the PBR and XBR, subject to dependencies and system constraints. It does not move as many as possible. There is no execution time penalty, however, because instructions not moved are issued on cycles where following instruction could not be issued anyway. The same argument

holds for the queue-filling/draining instructions.

The branch count field of the PBR instruction should be updated in accordance with the number of instruction parcels being moved between the PBR and the XBR. This updating is done by the assembler.

4.5. Remarks

In AE mode of PIPE (the decoupled mode), the access processor (AP) calculates all memory addresses and initiates all memory references for both processors. The execute processor (EP) does all algorithmic computations. Since the IL does not know different execution modes of PIPE, we don't have to introduce a new scheduling method at the IL level for different execution modes. The scheduling method, for the AE mode, at the machine code level is similar to that for the SP mode, except that the scheduler for the AE mode has to look at two instruction streams at the same time to avoid deadlock.

All code scheduling methods discussed so far are applied to one basic block at a time. Sometimes, there are not many instructions within a small basic block. It is obvious that code scheduling can be more effective for large basic blocks than for small ones. One example of an inherently small basic block comes from the high level language *while*-statement, in that the evaluation of the Boolean expression associated with the *while*-statement forms a small basic block. However, it is possible for a compiler to convert a *while*-statement into an *if*-statement followed by a *repeat-until*-statement with proper adjustment to the Boolean expression, which reduces the number of small basic blocks. This *while*-statement example demonstrates that a compiler can do necessary transformations to make other optimization methods (in this case, code scheduling) more effective. In the next chapter, we will introduce a method which does code scheduling across the basic block boundary.

4.6. Simulation Studies

In this section, we show some experimental results concerning the effectiveness of the code scheduling methods in utilizing the data queues and the prepare-to-branch instruction during the single mode execution of the PIPE architecture.

A functional interpreter and a performance simulator have been built to evaluate the performance of the PIPE architecture. Our measurement method is to compare the performance of the current PIPE architecture with that of a "bare PIPE." By "bare PIPE," we mean the PIPE architecture with the degenerate

Table 4.1. Relative Performance Gain by Adding the LDQ and the PBR Instructions to the Bare PIPE Machine.

Programs	Bare PIPE (cycles)	Loaded PIPE (cycles)	Speed Up
ACK	715	539	1.33
FACT	952	663	1.44
HEAP	1918	1609	1.19
LLL1	1412	811	1.74
LLL2	22617	15820	1.43
LLL3	28015	23015	1.22
LLL4	24492	21913	1.12
LLL5	29976	17987	1.67
LLL6	36642	21324	1.72
LLL7	22452	9853	2.28
LLL8	21568	17965	1.20
LLL9	17612	7812	2.25
LLL10	29712	20813	1.43
LLL11	26985	20990	1.29
LLL12	26985	20990	1.29
LLL13	43660	30348	1.44
LLL14	37512	28363	1.32

case for each special feature: in particular, the LDQ size is 1, and the branch count field is always 0 (*i.e.*, no instructions are moved to follow the PBR instruction). We use the fourteen Livermore loops [McMahon72] [Riganati84], two highly recursive programs--ACK (Ackermann function with arguments 1,1) and FACT (calculate the factorial of 5 by using recursive calls), and one sorting program--HEAP (heap sort).

Though it is impractical to build a very large on-chip instruction cache for the fabrication technology available to universities, we assume a large, fully-associative instruction cache with block size of eight parcels. The instruction cache is much

Table 4.2. Speedup Due to the LDQ Only.

Programs	Bare PIPE (cycles)	Loaded PIPE (cycles)	Average Load Dist.	Speed Up
ACK	715	544	7.55	1.31
FACT	952	663	8.04	1.44
HEAP	1918	1623	4.56	1.18
LLL1	1412	812	8.40	1.74
LLL2	22617	17416	5.40	1.30
LLL3	28015	28015	15.67	1.00
LLL4	24492	24459	10.00	1.00
LLL5	29976	18321	7.39	1.64
LLL6	36642	22656	10.67	1.62
LLL7	22452	10332	12.11	2.17
LLL8	21568	18048	12.65	1.20
LLL9	17612	8212	14.15	2.14
LLL10	29712	21312	15.67	1.39
LLL11	26985	22989	7.18	1.17
LLL12	26985	22989	11.00	1.17
LLL13	43660	30860	11.00	1.41
LLL14	37512	29112	7.33	1.29

larger than any of the programs tested. Therefore, we may consider the size of the instruction cache to be infinite. Thus, when a line (cache block) is referenced for the first time, an instruction cache miss occurs. All subsequent execution of the instructions within the same line (block) will not cause any further cache misses.

In table 4.1, we present the performance gain by adding the LDQ and PBR instruction to the bare PIPE. The speed-up column is the ratio of "the bare PIPE" column to "the loaded PIPE" column. The speed-up ranges from 1.12 (LLL4) to 2.28 (LLL7). The mean and variance of the speed-up are 1.49 and 0.33, respectively. In table 4.2, we show the relative speed-up owing to the LDQ alone. The

Table 4.3. Relative Performance for Different LDQ Size.

Programs	LDQ size 1	2	3	4	6	∞
ACK	715	635	595	579	555	539
FACT	952	802	742	718	682	663
HEAP	1918	1633	1623	1619	1613	1609
LLL1	1412	1071	971	831	811	811
LLL2	22617	21820	20220	17420	16020	15820
LLL3	28015	27015	23015	23015	23015	23015
LLL4	24492	23941	21913	21913	21913	21913
LLL5	29976	24981	17987	17987	17987	17987
LLL6	36642	26652	21324	21324	21324	21324
LLL7	22452	14533	10933	10453	9853	9853
LLL8	21568	21045	19205	18285	17965	17965
LLL9	17612	13712	11812	8912	8112	7812
LLL10	29712	20813	20813	20813	20813	20813
LLL11	26985	20990	20990	20990	20990	20990
LLL12	26985	20990	20990	20990	20990	20990
LLL13	43660	30988	30348	30348	30348	30348
LLL14	37512	30613	29413	28363	28363	28363

The numbers are execution time in terms of clock periods.

load distance is the number of instruction parcels between the instruction that loads the operand and the instruction that takes that particular operand out of the LDQ. The average load distance is also included in table 4.2. In table 4.3, we show the relative performance (with respect to an infinite LDQ) of different LDQ sizes. In table 4.4, we list the dynamic branch count distribution.

From this experiment, we make the following observations.

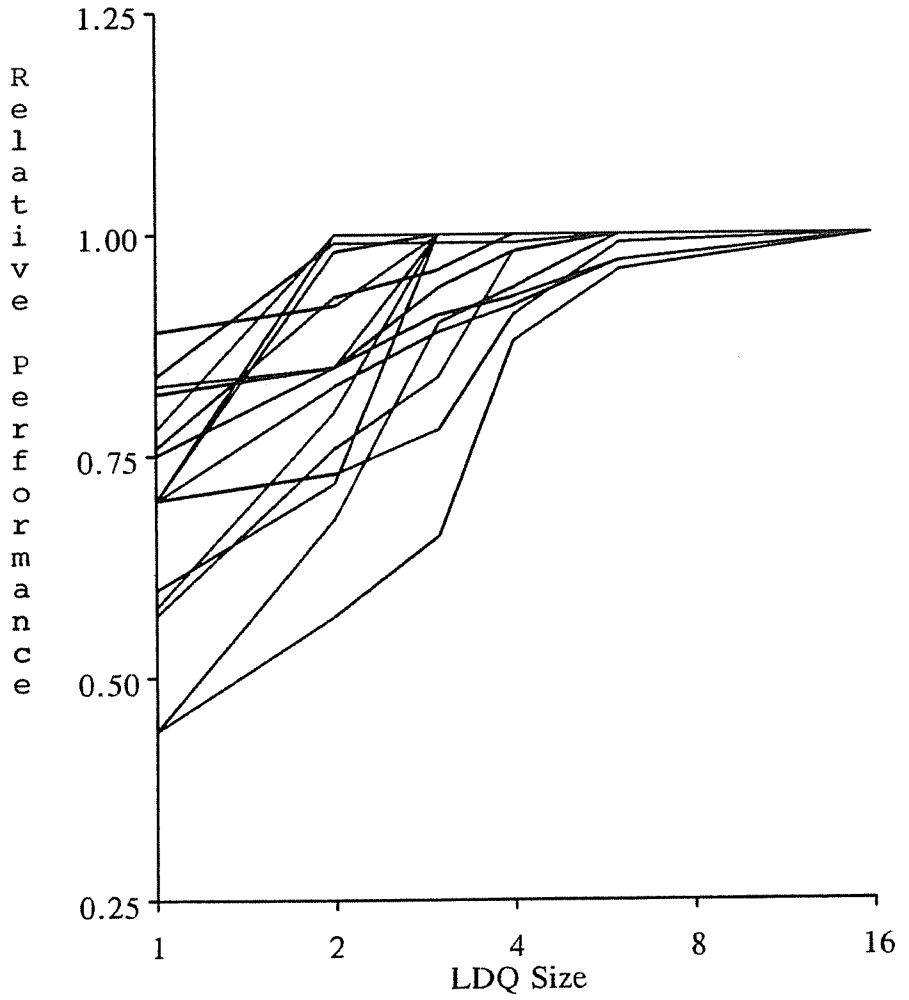
(1) Comparing tables 4.1 and 4.2, we conclude that most of the speed-up is due to the LDQ.

Table 4.4. The Dynamic Branch Count Distribution.

Programs	Branch Count (BC) Mean	Median
ACK	2.23	0
FACT	2.23	4
HEAP	1.79	0
LLL1	0.10	0
LLL2	3.49	3
LLL3	4.00	4
LLL4	4.00	4
LLL5	2.00	2
LLL6	3.99	4
LLL7	2.99	3
LLL8	5.51	6
LLL9	4.97	5
LLL10	2.99	3
LLL11	1.00	1
LLL12	1.00	1
LLL13	3.98	4
LLL14	1.01	1

The numbers for the branch count field are in terms of instruction parcels.

(2) From table 4.3, we conclude that the performance gains, due to the LDQ, level off when the LDQ size is about 6. (This, of course, is dependent on memory access time. We ran the same set of benchmark programs and found out that the performance levels off when the queue size is about the memory access time [in terms of clock periods].) The relative performance of these



This figure is constructed using the numbers in Table 4.4.

Figure 4.2. Relative Performance for Different LDQ Sizes.

programs for different LDQ sizes is shown in Figure 4.1, where the y-axis is the the relative performance with respect to an infinite LDQ. (The curves correspond to the benchmark programs and we drop the labels intentionally.)

(3) From tables 4.1 and 4.2, we can calculate that the performance gain due to the PBR instruction alone is not as good as that due to the LDQ alone. This can be explained by the following: (a) For a computer with relatively simple instructions, branch instructions occur less frequently than load instructions; (b) since a relatively large instruction cache is used, it takes only a few cycles to complete a branch instruction, even if the branch count field is zero, and (c) there are not always enough instructions to follow the PBR instruction if we do the assembly language level code scheduling within a basic block. Other possibilities of doing code scheduling are discussed in chapter 8.

(4) From table 4.4, we learn that the average branch count for all the benchmark programs is 2.79 with variance 1.48. The number of instructions that can be moved after a branch instruction depends on (a) the algorithm(s) used to schedule the code; and (b) the nature of the instruction set. From tables 4.1 and 4.2, we calculate that the average speedup owing to the PBR alone is about 10%. Though our algorithm is not as aggressive as that used with MIPS [Gross82], our results are comparable to theirs. We have shown that the generalized delayed branch is worthwhile for a reduced instruction set computer such as PIPE. Simple algorithms, such as the code scheduler described in this chapter, can be used to take advantage of the generalized delayed branch. Exotic algorithms, such as software pipelining (to be described in the next chapter), can make even better use of the PBR instruction.

Chapter 5

Software Pipelining

Software pipelining [Charlesworth81] is the deliberate partitioning of a program loop, carried out by the compiler, into load/computation/store sequences. This allows the overlapping of the execution of these operations in a fashion similar to hardware pipelining. The effect of software pipelining is to increase the throughput of the hardware pipeline. This is accomplished by anticipating operand requests and overlapping their access time with computations on previously fetched operands.

Essentially, the function of code scheduling is to identify the critical path in the data dependency graph, and to reorder the code sequence in order to overlap the operations of non-critical paths with the ones in the critical path. On the other hand, the function of software pipelining is to reshape the dependency graph by reconstructing the body of a loop in order to form a shorter critical path. The control dependency (*i.e.*, the branch instruction) is placed in the most appropriate location, and is often overlapped with some data dependencies.

The idea of software pipelining can be explained by an example. Consider the following code segment:

```
for i := 1 to Max do
begin
  Load_i;
  Computation_i;
  Store_i;
end;
```

The space-time diagram of this loop is:

L_{i-1}			L_i			L_{i+1}		
	C_{i-1}			C_i			C_{i+1}	
		S_{i-1}			S_i			S_{i+1}

(The x-axis is the time axis where the y-axis is the space axis. L, C, and S stand for Load, Computation, and Store, respectively.)

There are three "stages" in the loop shown above. There are data dependencies between $Load_i$ and $Computation_i$, as well as between $Computation_i$ and $Store_i$. Consequently, there are "bubbles" in the hardware pipeline. That is, some stages in the pipeline are not filled with useful work. An equivalent code sequence is:

$Load_1$;
$Computation_1$;
$Load_2$;
for i := 2 **to** Max − 1 **do**
$Store_{i-1}$;
$Computation_i$;
$Load_{i+1}$;
od;
$Store_{Max-1}$;
$Computation_{Max}$;
$Store_{Max}$;

For the next space-time diagram, we ignore the time to issue instructions (*i.e.*, only the execution time of instructions are shown). The space-time diagram for the new sequence looks like this:

L_{i-1}	L_i	L_{i+1}		
	C_{i-1}	C_i	C_{i+1}	
		S_{i-1}	S_i	S_{i+1}

There may be fewer data dependencies in the latter code sequence. Thus, the efficiency of the issue unit for the latter code sequence is higher. Consequently, the execution time of this particular loop can be reduced.

In PIPE, the only windows between CPU and the memory system are the architectural queues. These queues make it easier to do software pipelining because registers need not be allocated for loads and stores. Also, with queues, it is possible to load the data several iterations ahead of time (if it is beneficial to do so). This software pipelining technique can be applied to either the IL level or the machine level. Some semantic information, such as the loop boundary, may have to be kept around to simplify the job of software pipelining.

The following symbols are used for the discussion of the algorithms described in this section. Assume there are M_l instructions in the body of the loop, M of which are unrelated to loop control. That is, there are $M_l - M$ instructions which are used for loop control. Let S_{ij} be the i-th instruction from the j-th iteration of the loop.

Definition 5.1. *Momentous instructions* within a loop are instructions other than the loop control ones. □

Definition 5.2. The *n-th order unrolled sequence* is a sequence of momentous instructions formed by unrolling the body of a loop n times. □

Definition 5.3. The *cost of a code sequence* is the time between the issuing of the first instruction in that sequence and the completion of the last one. □

Definition 5.4. The *core* of a scheduled n-th order unrolled sequence is a code sequence that satisfies the following conditions.
(1) For all $1 \leq i \leq M$, there exists a unique j such that S_{ij} is in the sequence.
(2) For all code sequences that satisfy condition (1), choose the one with minimum cost.

□

We will explain the software pipelining algorithm below.

Algorithm 5.1. Software Pipelining.
(5-1-1) get the *n-th order unrolled sequence*;
(5-1-2) do code scheduling;
(5-1-3) find the *core*;
(5-1-4) reconstruct the loop;

□

The way to find the *core* is explained by the following algorithm.

Algorithm 5.2. Find the *core*.
foreach instruction S_{ij} **do**
(5-2-1) find a sequence, starting with S_{ij}, that satisfies condition (1) of a core **or** *end-of-input-stream*;
(5-2-2) **if** *end-of-input-stream* **then**
(5-2-2-1) **exit**;
else
{ call this sequence Q_{ij} }
(5-2-2-2) calculate the cost of Q_{ij};
(5-2-2-3) **if** Q_{ij} has the minimum cost up to this point **then**
(5-2-2-3-1) record Q_{ij};
fi;
od;
{ the last recorded Q_{ij} is the core found by this algorithm }

□

For practical purposes, the degree of unrolling (the value of n) should be at least two to make software pipelining more effective than simple scheduling method. It is easy to see that a core can always be found as long as the degree of unrolling is no less than one.

Lemma 5.1. The time complexity of Algorithm 2 is $O(n^2)$.

Proof. There are $n \cdot M$ instructions in the n-th order unrolled sequence, where n is the degree of unrolling. Normally, the degree of unrolling (*i.e.*, the value of n) is a small constant (*e.g.*, 3), and is independent of M. Thus, statement (5-2-1) is executed $O(M)$ times. Similarly, statement (5-2-2) is executed

$O(M)$ times. Statement (5-2-2-2) and (5-2-2-3) can be done in constant time. Consequently, the time complexity of Algorithm 2 is $O(M^2)$.

□

Algorithm 5.3. Reconstruct the loop.

(5-3-1) put all instructions before the *core* before the body of the newly constructed loop;
(5-3-2) put all instructions after the *core* after the body of the newly constructed loop;
(5-3-3) adjust loop bounds;
(5-3-4) add loop control instructions back;

□

Lemma 5.2. The time complexity of Algorithm 3 is constant.

Proof. Trivial.

□

Theorem 5.1. The time complexity of Software Pipelining is $O(M^2)$, where M is the number of momentous instructions within the loop.

Proof. It takes constant time to do the following two operations: (a) to get the n-th order unrolled sequence; and (b) to reconstruct the loop (Lemma 5.2). The time complexity of code scheduling is no worse than $O(M\log M)$ (Theorems 4.1 and 4.3). From Lemma 5.9, the time complexity of finding the core is $O(M^2)$. Therefore, the time complexity of the Software Pipelining algorithm is $O(M^2)$.

□

The software pipelining algorithm can be applied to both the IL level and the machine code level. In either case, the time complexity is $O(M^2)$.

With minor modification, this software pipelining method can be applied to loops where the number of iterations is not known at compile time (*e.g.*, *for*-loops having variables as loop bounds; *while* or *repeat-until* loops).

Example 5.1. The software pipelined code of Example 4.1 is:

```
      ....
LOOP
      S1    SDQ     ←   R3
      S2    SAQ     ←   R1, y
      S3    R2      ←   LDQ ×f R3
      S4    R1      ←   R1 + 1
      S5    R4      ←   R1 − R0
      S6    PBRLE       R4, BR0, 5
      S7    R3      ←   R2 +f LDQ
      S8    LDQ     ←   R1, f
      S9    LDQ     ←   R1, g
      S10   XBR
```

The issue time and completion time are:

Statement Number	Issue Time	Completion Time
S1	0	1
S2	1	2
S3	2	6
S4	3	4
S5	4	5
S6	5	*11
S7	6	10
S8	7	13
S9	8	14

Although S9 completes at time 14, the operand loaded by S9 will not be used until S7 of the next iteration. The effective speed of this loop is 11 clock periods per iteration. If the memory load delay were much longer, the software pipelining method could generate code to do operand prefetching two or more iterations ahead.

□

The efficiencies of the issue unit (*i.e.*, the issue rate) for the compiled unscheduled code, the scheduled code, and the software pipelined code are 35%, 69%, and 82%, respectively. The efficiency of the issue unit is a good indication of the throughput, hence, speed up. Thus, for this particular example, the speed-up of software pipelining over straight forward scheduling is about 19%.

It is relatively difficult to do software pipelining on loops with if-statements, because it is hard for the compiler (code scheduler) to know whether operands in the if-part should be prefetched. The compiler, however, can at least prefetch operands that are used to determined the Boolean condition of the if-statement. If the *true-ratio* of the if-statement is known, the compiler can do appropriate prefetching accordingly. This is often the case, for example, in testing for error conditions.

Software pipelining has marginal performance effects on loops where the execution time of an iteration is much longer than that of a memory reference. Thus, it is advisable to apply the software pipelining method only to loops where the execution time of an iteration is comparable with that of a memory reference. Intuitively, it is beneficial only to do software pipelining on small loops. There may not be many small loops in programs. A big loop may be split (by the compiler) into a few smaller loops to fit the loop bodies into the hardware instruction buffer. In the supercomputing environment, however, a relatively big loop may be divided into vectorizable parts and non-vectorizable parts by applying high level language program transformation techniques, such as the ones in Parafrase [Kuck80]. The non-vectorizable parts of a big loop may consist of a few small loops intermixed with some vectorizable loops which may also be small. The execution time of these non-vectorizable loops tends to dominate the total execution time. Software pipelining is a good way to reduce the execution of the non-vectorizable small loop, hence, total execution time. Because the software pipelining method is invoked only for small loops, the quadratic execution time of the software pipelining algorithm will not introduce excessive overhead to the entire scheduling process.

Loop unrolling is another method to speed up loops. The major difficulties of loop unrolling are [Weiss84a] (a) register allocation, and (b) code size of the loop. The code size has dramatic impact on the performance of loops. When the code size of a loop exceeds the hardware instruction buffer size, the system performance

degrades significantly, due to excessive instruction buffer misses. The software pipelining method, on the other hand, does not affect the code size of a loop body (though the preamble may be larger). Thus, software pipelined code does not have optimization anomalies in that the execution time of the software pipelined code is no longer than that of the original code. In the worst case, the software pipelined code is the same as the original code. Since the execution time of each instruction is used to guide software pipelining, the degree of prefetching (*e.g.*, the number of preload operands) is flexible in the sense that different loops may have different degrees of prefetching. The five loop unrolling (software pipelining) methods discussed by Weiss [Weiss84a] are just special cases of code sequences the software pipelining method may produce.

5.1. Software Pipelining Results

In this section, some of the Livermore Loops are used as sample programs to compare the performance of software pipelined code with scheduled code. The loops used are loops 5†, 6†, 11, 13 and 14. These loop are chosen because their tested performance on many state-of-the-art supercomputers (*e.g.*, S-810, VP-200, Cray 1, Cray X-MP) was below 20 MFLOPS[1] (million floating point operations per second) [Riganati84]. In other words, the execution speed of these five loops dominates the total system performance.

The performance comparison of the aforementioned loops is shown in 5.1. As we pointed out earlier that it takes six clock periods to do a memory load. If the memory load time were much longer, the software pipelining method might preload

---

† In the original Livermore loops, loops 5 and 6 are unrolled three times. For the discussion of this section, however, we used the version without the unrolling.

[1] In most cases, the performance was below 10 MFLOPS.

Table 5.1. The Speedup of Software Pipelined Code.

loops	scheduled	software pipelined	speed up
5	26	24	1.08
6	26	24	1.08
11	9	6	1.33
13	47	39	1.21
14	54	44	1.23

operands multiple iterations ahead of time. Thus, software pipelining is more effective when the memory access time is longer. With minor modification, software pipelining is also applicable to *while*-loops where the number of iteration is not a compile time constant.

Chapter 6

Code Generation for Decoupled Mode

Decoupled architectures, such as PIPE, divide the instruction stream into two parts: addressing and algorithmic computing. The two parts are put on separate processors. Previous studies [Smith84] [Hsieh84] about different decoupled architectures used hand-written benchmark programs to evaluate performance. In this chapter, we describe the methods the compiler uses to generate code for the AE mode of PIPE. Although the compiler does not generate as good a code as hand-written code, we compare the performance of a compiled decoupled code with a compiled single mode code to demonstrate the speedup attained by using high level languages. The only optimization used in running these benchmark programs is code scheduling[2]. The compilers for both the single mode and the decoupled mode share many modules. Only the code generating phase employs different routines. We expect similar speedup on both execution modes by applying other optimization methods, such as common subexpression elimination.

We will first describe the methods used in generating code for the decoupled mode. Then, we will show the simulation results of the compiled code.

6.1. Code Generation Methods for Decoupled Mode

The basic structure of the Pascal compiler has been discussed in chapter 4. In this section, we emphasize the code generation methods for the decoupled mode. As we mentioned before, the idea behind a decoupled architecture is that one pro-

[2] Since software pipelining favors smaller loops, we do not use it in running these programs.

cessor (the A-Processor) runs ahead and loads operands before they are needed by the other processor (the E-processor). Thus, all memory accessing instructions are generated on the A-processor. Consequently, address calculations are also done on the A-processor. The A-processor must make as many branch decisions as possible to run ahead of the E-processor. In the current implementation, the A-processor makes all branch decisions. Thus, all expressions participating in a branch decision are evaluated on the A-processor. In short, the A-processor generates code to do the following:

(1) Address calculation.

(2) Branch determination.

(3) Memory accessing (*i.e.*, load/store instructions).

To retain the source language level structures to aid the code generation (both for the single mode and the decoupled mode), several pseudo instructions are included in the intermediate language. The characteristic of an intermediate language statement, such as an expression associated with a branch decision, is denoted by a fifth tuple of the IL statement; the attribute tuple. The attributes tell the code generator to generate code for expressions on the appropriate processor.

In the front-end of the compiler, a binary flag, called inBranchExp, is turned on at the beginning of a Boolean expression for a control structure (*e.g.*, if-statement). This flag is turned off when the Boolean expression has been completely compiled. Intermediate language statements which are generated with the "inBranchExp" flag on have an attribute saying so. On the other hand, the array index calculation may be nested (*e.g.*, a[b[i]]). Thus, a counter, called IdxDepth, is incremented at the beginning of each index evaluation (*i.e.*, encounter a "["). This counter is decremented when an index calculation has been compiled (*i.e.*, encounter a "]"). The IdxDepth has an initial value of zero. An attribute associ-

ated with the index calculation is appended to each intermediate language statement generated with a positive IdxDepth.

The generation of load/store instructions for real variables (as opposed to pseudo registers), which are allocated in memory, does not pose many problems for the code generator. The alternative load is generated in the A-processor for an operand which is used by the E-processor. Similarly, the alternative store is used to store a result computed by the E-processor. If the operand is a pseudo register, there are three cases that the code generator must consider. Case 1, the pseudo register is mapped into a memory location, which is the same as accessing a memory operand. Case 2, the pseudo register is in a real register of the processor which needs that operand: the code generator simply uses the real register. Case 3, the pseudo register is in a real register of the wrong processor: in this case, two memory accessing operations have to be done to convey the requested value to the correct processor. Case 3 comes from the use of shared variables. Because of the shared variable problem, it is not always beneficial to allocate a register for a variable. This situation makes register allocation more complex than for the single processor case.

6.2. Simulation Results

We use the same set of benchmark programs described in chapter 4 to evaluate the effectiveness of decoupled code generation. The speedup for the decoupled execution mode (compared with the best single mode code) is shown in Table 6.1.

Loops 11 and 12 do not experience any performance gain because they are tight loops and their performance is bounded by the time to "execute" the branch instruction. As we expected, the non-numerical programs do not get as large a speedup as most numerical programs. In particular, the recursive programs do not reveal a significant speedup. The reason for this is that the compiler uses a naive

Table 6.1. Performance Simulation Results for Decoupled Mode.

programs	loaded single PIPE (cycles)	loaded decoupled PIPE (cycles)	speed up
ACK	539	502	1.07
FACT	663	659	1.01
HEAP	1609	1371	1.17
LLL1	811	558	1.45
LLL2	15820	13662	1.16
LLL3	23015	19020	1.21
LLL4	21913	15815	1.39
LLL5	17987	14327	1.26
LLL6	21324	18009	1.18
LLL7	9853	6742	1.46
LLL8	17965	14772	1.22
LLL9	7812	5320	1.47
LLL10	20813	11449	1.82
LLL11	20990	20992	1.00
LLL12	20990	20992	1.00
LLL13	30348	22164	1.37
LLL14	28363	18916	1.50

scheme to save/restore all registers during procedure calls which offsets some of the benefits gained by decoupled mode execution. Note there are twice as many registers to be saved and therefore twice as many addresses to be generated by the A-processor. For the fourteen Livermore loops, which are procedure-less programs, the code is generated assuming enough registers on each processor so that no spilling code is ever needed.

Comparing our results with studies done by others [Smith84] [Hsieh84], we find that, in general, we achieve a lower speedup. We offer the following reasons:

(1) Register allocation. Better register allocation, which keeps some global variables in registers, may help in balancing the load (*i.e.*, the time required to perform a function) on both processors.

(2) Underlying architecture. Different timing assumptions may lead to different loads on both processors. Some advantages of the decoupled mode (*e.g.*, possibly hidden memory delay) may be taken care of by a carefully designed single mode processor. In general, the single mode can take more advantage of the PBR instruction than the decoupled mode can, because in most basic blocks, more instructions may be placed after a PBR instruction in the single mode than in the decoupled mode. The sizes of the blocks associated with procedure prologue/epilogue are larger in the decoupled mode than in the single mode. The PBR instruction, however, is well utilized in the prologue/epilogue blocks, even in the single mode. Thus, during decoupled mode execution, the extra instructions in the prologue/epilogue blocks do not take further advantage of the PBR instruction.

(3) Code scheduling. The decoupled mode provides the effect of dynamic code scheduling. Some hand-coded programs concern more about the number of instructions generated than the their order. In other words, those programs are not well scheduled to take advantage of the underlying architecture. Poorly scheduled code benefits more from the dynamic code scheduling capability provided by the decoupled execution mode than well scheduled code does.

Table 6.2 shows the speedup comparing the "loaded decoupled mode PIPE" (*i.e.*, the PIPE architecture with all aforementioned special features running in decoupled mode) with the "bare single mode PIPE." The speedup shown here is the cumulative effect of all three special features (*i.e.*, data queues, prepare-to-branch instruction, and decoupled mode) introduced in PIPE.

Table 6.2. Performance Simulation Results of All Features.

program	bare single PIPE (cycles)	loaded decoupled PIPE (cycles)	speed up
ACK	715	502	1.42
FACT	952	659	1.44
HEAP	1918	1371	1.40
LLL1	1412	558	2.53
LLL2	22617	13662	1.66
LLL3	28015	19020	1.47
LLL4	24492	15815	1.55
LLL5	29976	14327	2.09
LLL6	36642	18009	2.03
LLL7	22452	6742	3.33
LLL8	21568	14772	1.46
LLL9	17612	5320	3.31
LLL10	29712	11449	2.60
LLL11	26985	20992	1.29
LLL12	26985	20992	1.29
LLL13	43660	22164	1.97
LLL14	37512	18916	1.98

Chapter 7

The Design of a Vector Extension

The major limitations of the existing PIPE architecture are the following:

(1) Each processor has only one queue in each queue class (*e.g.*, there is only one LDQ on each processor). There are application programs (*e.g.*, sorting) where multiple queues (in particular, multiple LDQs) are desirable. The single queue scheme also implies that load/store instructions must be generated in one processor during decoupled mode execution, which makes load balancing between processors less flexible than does a multiple queue organization.

(2) It does not provide vector processing capability. Thus, all operations must be performed with scalar instructions, which makes the instruction issue unit a noticeable bottleneck (*i.e.*, the Flynn bottleneck). This particularly limits the A-Processor from staying ahead of the E-processor.

(3) In the decoupled mode, the values of shared variables have to be transmitted via the memory system. Hence, the use of shared variables is extremely inefficient in terms of execution time.

(4) Some instruction initiation blockages are due to the single result bus between the functional units and registers.

(5) It enforces the use of branch instructions to evaluate Boolean expressions. An example of Boolean expression in the C language is "(a>b)?a:b". As we pointed earlier, branches break the instruction flow. Hence, we want to eliminate unnecessary branches. The contents comparison instruction described later in this chapter can be used to evaluate Boolean expressions without using branches.

(6) Each element of a queue can only be read once. A use-many-times operand must be moved explicitly to a general purpose register if we don't want to reload it from the memory every time we use its value.

In this chapter, we propose an extension to the existing PIPE architecture to remedy these limitations and to utilize the on-chip area by increasing the gate/pin ratio (*i.e.*, use more gates without requiring more pins).

One of the limitations of VLSI is the number of available pins on a single chip. As we mentioned above, pipelining and VLSI are a good match. VLSI provides us with many transistors per single chip. Even simple pipelining (*e.g.*, a pipeline that consists of simple "instruction fetch," "instruction decode," and "execution" stages) may not use up the available transistors. Some kind of instruction buffer (*e.g.*, instruction cache) is included in VLSI processors to reduce the off-chip communication and to utilize the on-chip area. But the performance gain levels off when the instruction buffer is larger than a certain limit [Smith82]. Thus, we do not want to include an extremely large on-chip instruction buffer, either. On the other hand, it may not be desirable to have an on-chip data cache for the following reasons:

(1) A local (on-chip) data cache introduces the data coherence problem in a multiprocessor environment. A decoupled architecture, which uses two processors, is a multiprocessor system.

(2) The use of a data cache for scientific applications is known to be of questionable value [Axelrod83]. Registers (*i.e.*, a programmer controllable cache) may prove more effective than a simple data cache.

Most, if not all, current supercomputers have some kind of vector processing capability, in additional to their powerful scalar units. We propose an extension to the existing PIPE architecture, which includes vector processing capability, to use the on-chip area provided by advanced VLSI technology and potentially enhance the

system performance without requiring more pins. The characteristics of this vector extension are:

(1) Multiple functional units are included, each of them highly pipelined.

(2) Instruction initiation is done by two-level issue logic.

(3) Multiple classes of registers are provided for different usage requirements.

(4) There is a non-blocking interconnection between the functional units and different classes of registers.

(5) It caters to the easily vectorizable property in that more array-oriented operations can be vectorized under this extension than many other state-of-the-art vector supercomputers (*e.g.*, Cray-1 [Cray82], Cyber 205 [Lincoln82]).

The function of these features will be explained in the subsequent sections. We will concentrate on the desired features rather than the detailed instruction format or implementation.

7.1. Motivation for Vector Instructions

The major considerations behind the vector instructions are the following:

(1) Flynn bottleneck. The performance of a computer system is sometimes limited by the instruction initiation rate. An issue unit is capable of initiating at most a fixed number (normally, one) of instructions every clock period. If each instruction does too simple an operation, this Flynn limit may become a bottleneck. A decoupled architecture does increase the bandwidth of the instruction issue capability by including multiple (two) issue units. Another way of increasing the effective issue bandwidth is to include powerful instructions. Each powerful instruction is able to do many operations. Hence, issuing one powerful instruction can achieve the same effect as issuing multiple relatively simple ones. The issue condition of a powerful instruction, however,

does not have to be complex. The major goal of vector extension is to accommodate powerful instructions in the existing PIPE architecture. Each powerful instruction (vector instruction) is just a repetition of a simple operation. These augmented vector instructions still enjoy the simple issue condition property of the original PIPE.

(2) Out-of-order execution. The floating point unit of the IBM 360/91 [Tomasulo67] and the scoreboard of the CDC 6600 [Thornton70] are examples of hardware methods to achieve out-of-order execution. These methods were abandoned because of the hardware complexity of implementing those algorithms and because of the emergence of comparable software solutions [Thorlin67]. Out-of-order execution is an essential part of vector instructions because of the drastically different execution time of vector instructions with different vector length as well as the simultaneous execution of vector and scalar instructions. The vector extension proposed in this study, however, provides out-of-order initiation of scalar instructions (almost) for free, in the sense that a scalar instruction is just a special case of vector instruction. That is, a scalar is just a vector of length one.

It is unreasonable to have several vector instructions compete for a single ALU. In such a case, the throughput of the system would be limited by the ALU regardless of the efficiency of the issue unit. Thus, for the discussion of this chapter, we will consider a multiple functional unit system. Each function unit is implemented by a linear (fully segmented) pipeline [Kogge81], which eliminates the necessity of doing pipeline reorganization and avoids most structural hazards (collisions). Each functional unit may either perform a single task (*e.g.*, floating point addition) or be capable of executing several operations. The features required to support decoupled execution (*e.g.*, alternate load/store, external branch) are included in the original PIPE. We only have to include corresponding alternate

loads/stores when we introduce new load/store instructions.

7.2. Desired Features

The desired vector properties include flexible memory-referencing instructions, arithmetic/logic operations, vector editing instructions and the easily-vectorizable properties. Regular arrays should be treated effectively, while sparse arrays must be handled with reasonable efficiency. To avoid the overhead introduced by using shared variables in AE mode, we suggest that all the registers be shared by all functional units. We also want to be able to intermix vector instructions with scalar ones. For example, vector A is loaded into a queue register by a vector load instruction, but entries of A are consumed by different scalar instructions.

A vector instruction has the following format:

repeat *n*
inst;

The semantics of the above repeat statement are to execute *inst n* times, where *inst* represents a simple scalar instruction. The length of a vector is specified by one of the queue length (QL) registers and the "repeat" instruction selects the appropriate queue length register. We will demonstrate in a later section that most vector operations (*e.g.*, inner product) can be realized by the combination of several repeat statements.

Four sets of operand registers are included:

(1) Constant scalar registers. These registers are intended to be used in a read-only fashion during normal program execution. They, however, are writable just like ordinary registers except that the underlying hardware is not furnished with interlocks to prevent possible hazards. Software methods, such as the one proposed in MIPS [Hennessy81], should be used when writing to these

registers. Normally, the values of the constant registers are assigned in an initialization block near the beginning of a program execution. Their usages, however, are in subsequent blocks. Here, the scheduling for constant registers is a lot easier than the general case required by MIPS. The simplification of interlocks may imply fast access. Constant registers also make the processor easily scalable in the sense that the instruction format remains unchanged when we change the data format. We only have to widen each individual constant register when the data format becomes longer. As a side effect, the existence of constant registers may also reduce processor-memory bus traffic. That is, except for the initialization of constant registers, the constants need not be loaded from the literal pool in the main memory.

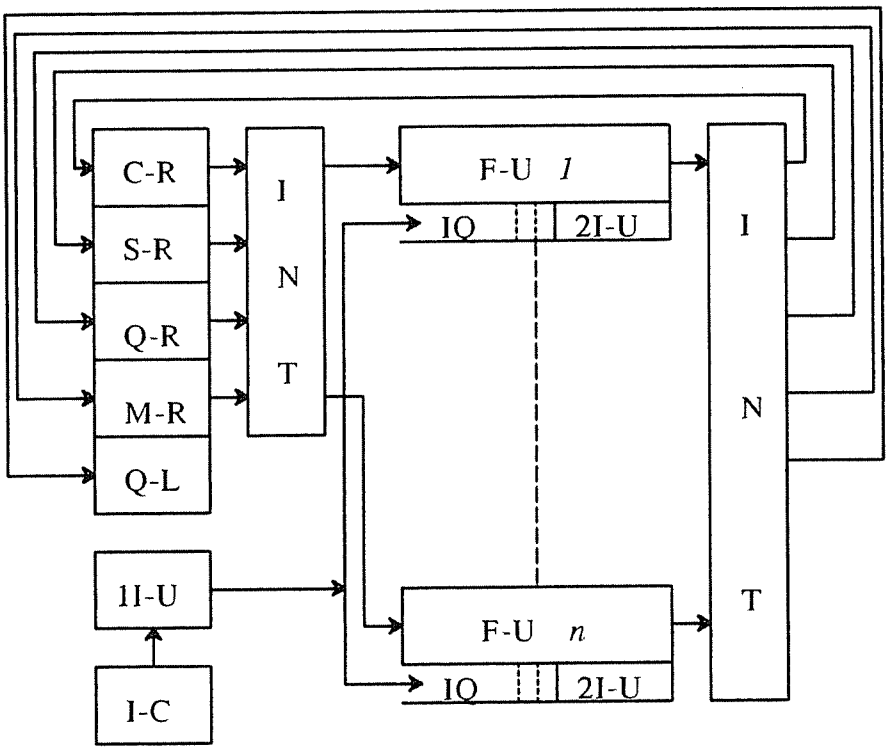
(2) Scalar registers. They are just like general purpose registers in most computer architectures.

(3) Queue registers. Queue registers are a set of registers, each of which is a queue. We may view a queue register as an array of registers. Only limited elements (*i.e.*, the head and tail of the queue) are accessible from outside. These queue registers will be completely specified in section 7.3.2. A queue register is the hardware implementation of a "stream." That is, operands in a queue register must be accessed in a predetermined order.

(4) Mask registers. Mask registers are similar to the queue registers in that they are also implemented as queues. The major difference is that each element of a mask register has only one bit.

Any one of the scalar registers or the queue registers can be used as a destination/source register of memory accessing instructions. Hence, an additional field is required in the load/store instructions to specify the corresponding destination/source register. Figure 7.1 illustrates a processor with two functional

units. The memory interface is treated as one functional unit. In this chapter, we will use %C*c*, %S*s*, %Q*q*, and %M*m* to denote the *c*-th constant register, the *s*-th scalar register, the *q*-th queue-register, and the *m*-th mask-register, respectively.



Terms:
C-R: Constant Registers
S-R: Scalar Registers
Q-R: Queue Registers
M-R: Mask Registers
Q-L: Qurue Length Registers
I-C: Instruction Cache
INT: Interconnection Network
1I-U: First Level Issue Unit
2I-U: Second Level Issue Unit
IQ: Instruction Queue
F-U*i*: the *i*-th Functional Unit

Figure 7.1. The Organization of a Processor with Two Functional Units.

The exact number of elements of a given register set is not included in this study. We made no attempt to design an arbitrarily scalable processor. In other words, we don't intend to put an extremely large number of elements in a register file, nor do we intend to include many of functional units. However, we do not exclude the possibility of having multiple function units for the same operation (*e.g.*, floating point addition).

7.3. Implementation

The realization of some unconventional functions are explained in this section.

7.3.1. Two-Level Control

A two-level instruction initiating scheme is adapted in this design. The two-level issue logic units are termed the *first level* issue logic and the *second level* issue logic, respectively. The first level issue logic decides branch outcomes and sends non-branch instructions to the proper instruction queues which are associated with the second level issue logic. For each functional unit, there is an instruction queue and second level issue logic. Each second level instruction issue logic initiates instructions in the order they were sent to the instruction queue (*i.e.*, the original program textual order). Instructions in different second level instruction queues, however, may be initiated in a different order than that in which they passed the first level issue logic. The second level issue logic checks for data availability and issue instructions accordingly. The implication of this two-level initiating scheme is that it takes two clock periods to issue a non-branch instruction. That is, the work of instruction initiating is divided into two stages of the pipeline. However, this scheme does not introduce extensive delay. The delay of the additional clock period in the decoding logic slows down a section of straight line code by at most one clock period. Since the branch decision is made by the first level issue logic, there is no

execution time penalty even across basic blocks, as long as the expressions that participate in the branch decision are evaluated earlier, which can normally be done by properly scheduling the code. Thus, in the best case, the penalty introduced by this two-level control scheme is one clock period per program. The worst case penalty, however, is one clock period per basic block. The first level issue unit sends a partially decoded instruction to the instruction queue associated with the appropriate function unit. In other words, the decoding is done in two stages (the first level issue unit and the second level issue unit). If the instruction decoding time of the original design determines the clock period, the two level decoding scheme may imply a faster clock rate, because the instruction decoding is done in two pipeline stages. The startup time of a vector instruction is just the time for the first level issue logic to send a "repeat" statement to the appropriate second level issue logic, which normally takes one clock period.

This two-level scheme does provide some out-of-order instruction initiation, for the first level issue logic is blocked only because of either (1) a branch instruction, which can be alleviated by the prepare-to-branch scheme, or (2) a full instruction queue of the corresponding functional unit. The functionality of this proposed scheme is similar to that of Tomasulo's algorithm [Tomasulo67] except that instructions that use the same functional unit are executed in program order. Table 7.1 compares the performance of the two-level scheme with that of Tomasulo's algorithm using the example (written in CAL [Cray83]) given by Weiss [Weiss84b] to compare different issue algorithms. In this example, the two-level scheme and the Tomasulo's algorithm perform equally well. The former, however, avoids the potentially expensive associative search used in the Tomasulo's algorithm.

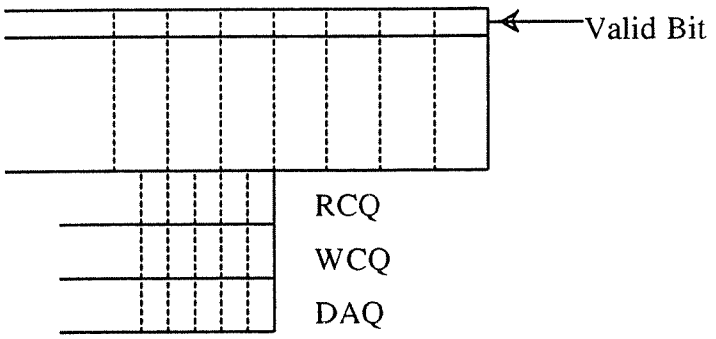
Table 7.1. Timing Chart for Livermore Loop 12.

```
1:
        S5    T00       COPY T00 TO S5
        A1    S5        COPY S5 TO A1
        S6    off1,A1   LOAD S6 (ADDRESS INDEXED BY A1)
        S1    off2,A1   LOAD S1 (ADDRESS INDEXED BY A1)
        S4    S6-fS1    FLOATING DIFFERENCE OF S6 AND S1 TO S4
        S3    S5+S7     INTEGER SUM OF S5 AND S7 TO S3
        A2    B02       COPY B02 TO A2
        A0    A2+1      INTEGER SUM A2 AND 1 TO A0
     Q3,A1    S4        STORE S4 (ADDRESS INDEXED BY A1)
       T00    S3        COPY S3 TO T00
       B02    A0        COPY A0 TO B02
       JAM    1         BRANCH TO LOOP ENTRY
```

Loop 12		Tomasulo's Algorithm		Two-level Scheme		
		issue	complete	first level	second level	complete
1:						
S5	T00	0	1	0	1	2
A1	S5	1	2	1	2	3
S6	off1,A1	2	13	2	3	14
S1	off2,A1	3	14	3	4	15
S4	S6-fS1	4	20	4	15	21
S3	S5+S7	5	8	5	6	9
A2	B02	6	7	6	7	8
A0	A2+1	7	9	7	8	10
Q3,A1	S4	8	21	8	9	10
T00	S3	9	10	9	10	11
B02	A0	10	11	10	11	12
JAM	1	11	*16	11	-	*16

7.3.2. Queue Registers

Each queue register (see Figure 7.2) is implemented as an array of registers. There is a *valid bit* associated with each element of the queue register to indicate the



Terms:
RCQ: Read-Control-Queue
WCQ: Write-Control-Queue
DAQ: Data-Available-Queue

Figure 7.2. The Organization of a Queue Register.

---

data availability of the designated element. Two pointers are associated with each queue register (not shown in the figure). They point to the head and the tail of that particular queue. These pointers are termed *head-pointer*, and *tail-pointer*, respectively. A queue may have three possible states--Full, Empty, and Normal--which can be checked by examining the appropriate valid bits. When a queue is full, an attempt to put an operand onto that queue will be blocked. Similarly, reading from an empty queue is also blocked. One of the advantages in having one valid bit per element is that a queue register can temporarily hold a vector which is longer than the queue size, as long as there are other instructions that take operands out of the same queue register. This valid bit per element scheme also makes chaining more flexible, because the speed of the consumer and the producer does not have to be identical. That is, chaining can be performed in an asynchronous fashion. The flexibility provided by the valid bit scheme is desirable in the following scenario. Suppose *A* and *B* are two back-to-back vector instructions and *B* follows *A*. *A* is a

vector load instruction with the stride being the number of memory banks. Thus, *A* cannot produce an operand every clock period, because of bank conflicts. *B*, on the other hand, is a register-register vector instruction, which is capable of consuming one element from a queue register every clock period. A simple scheme is to complete *A* before initiating *B*, which leads to the non-overlapping execution of vector instructions. Using the valid bit per element scheme, *B* can consume an operand immediately after *A* has produced it. The total execution time under this scheme is the execution time of *A* plus the time for *B* to process the last element. The saving here can be significant if the the vector is long or if *B* is chained to another vector instruction *C*. A similar scheme is used by the Hitachi S-810 [Nagashima84] to allow flexible chaining. This queue-based vector extension has the flavor of dataflow machines [Dennis80] except that the issue logic checks for data available, rather than the data availability "firing" the operation. Thus, function units can operate in parallel as much as possible. We call this organization a *control-driven dataflow* computer architecture. This control-driven dataflow scheme is more flexible than that of most traditional supercomputers in the sense that data availability is checked on an element by element basis, even for vector operands. Hence, the operation of chaining is more flexibility than in the case where data availability is checked at the vector level. On the other hand, the overhead of the dataflow approach [Gajski82] is avoided. This scheme supports complex data structures (*e.g.*, arrays) well, which have not been adopted successfully in a pure dataflow fashion [Gajski82].

One of the characteristics of queues is that an element of a queue is discarded after it has been read. There are cases in which we want to use an operand (a scalar or a vector) more than once. Therefore, we include three access modes in reading from a queue: (1) *destructive* mode; (2) *non-destructive* mode; and (3) *circular* mode. In destructive mode, a read operation removes the first element from the

queue register. In non-destructive mode, the first element of a queue remains after a read operation. That is, a queue can be (non-destructively) read many times while its contents are not changed. In circular mode, the entire queue remains unchanged after each element has been accessed. Since a queue register is implemented as an array of elements with two pointers, a different access mode simply suggests a slightly different interpretation in updating the pointers. For example, the circular mode is implemented by restoring the header pointer after the entire vector has been read. Different access modes may be specified as part of the register designator. In this chapter, we will use %Q*n*.D, %Q*n*.N, and %Q*n*.C to indicate access mode to the *n*-th queue register being destructive, non-destructive, and circular, respectively. (That is, the appended letter distinguishes the access mode.) No access mode is specified if a queue register is used as the designation register. One restriction of the circular mode is that a vector which is longer than the size of a queue will not fit into one queue. The compiler has to split a long vector into several smaller ones if it chooses to employ the circular mode.

No hardware interlock mechanism is provided for the constant registers. The scalar register is just a queue register of size one. The interlocks on the scalar registers are similar to those on the queue registers. The mechanism to handle different data dependency hazards (RAW, WAR, WAW) for queue/mask registers is relatively uniform and is described below. There are three queues (shift registers) associated with each queue/mask register, one corresponding to the read[1] operation (termed *read-control-queue*), the second corresponding to the write operation (termed *write-control-queue*), and the third detecting write-after-write hazards (termed *data-available-queue*). As we shall explain later in this chapter, the data-available-queue is also used to facilitate data forwarding (chaining). The first level

---

[1] A read operation to a register is to get the value from the register.

issue logic puts the tag of the functional unit on the corresponding read/write-control-queue. If that particular register is used more than once within a single instruction (*e.g.*, in the case where a register is used as both the source registers), only one entry is put in the control-queue. The second level issue logic checks the heads of the corresponding control-queues. When the heads of the relevant control-queues match the functional unit, the issue logic of that functional unit can check for other issue conditions. The data-available-queue is included with each queue/mask register. The second level issue logic sets the proper entries of the data-available-queue of the destination register, which is used to reveal when a result is available. A read operation checks for the data-available-queue to employ result forwarding (chaining) whenever possible. That is, when the first element of the data-available-queue is set with a special tag, a result for that register will be available in the next clock period. The control flag in the read-control-queue is removed after the contents have been latched, which is normally done at instruction issue time. In the case of a vector read, the control flag will not be removed from the read-control-queue until the content of the last element has been latched elsewhere. The corresponding write flag, however, is removed from the write-control-queue when the last write operation has been issued. The synchronization of simultaneous write operations is done by the data-available-queue, which is explained by the example below.

> *Example* 7.1. An instruction I that uses $\%Qi$ as its destination register takes n clock periods to complete. At instruction issue time, the (second level) issue logic checks the n-th[2] entry in $\%Qi$'s data-available-queue. A busy n-th entry (the n-th entry is set) indicates that an instruction which has been issued earlier and uses $\%Qi$ as its destination register, will not be completed until at least n clock periods later. When I is ready to issue, the issue logic sets the first n entries of the result shift register and puts a special tag on the n-th entry. This special tag is checked by a subsequent instruction that uses the result generated by I to do proper operand forwarding. A special tag in the first element of the

[2] The n-th element from the head of the queue.

data-available-queue means a result from this queue/mask register will be available in the next clock period.

□

7.3.3. Vector Load/Store

Studies [Bucher83] [Matsuura84] have shown that the relative ratio of vector load/store with unit stride, non-unit stride, and random access is about 70% : 20% : 10%. Thus, the vector load/store instruction has to support accessing random elements of an array. A repeated load/store with autoincrement can be used to access elements with a constant stride apart. A random access is normally represented by an additional level of indirection. That is, the addresses of the needed elements are put in another array. This random access is supported by the following two vector instructions: (a) put addresses of needed elements onto a queue register (call it %Q*q*); and (b) do a vector load using addresses in %Q*q*. An example of random vector load is shown below:

Example 7.2. Suppose array **A** is a sparse array. The (absolute) addresses of non-zero elements are stored in **B**, and there are *n* non-zero elements.
The following vector instructions are used to implement the required function:

```
repeat n
        LOAD %Qx  0        B
/* load addresses to Q-reg x */
repeat n
        LOAD %Qy  0        %Qx.D
/* load the desired elements to Q-reg y */
```

□

Simultaneous vector load/store operations may cause undesirable memory overlap hazard conditions (that is, read before write or write before read). One solution for eliminating such hazards is to have a smart memory system examine for the conflicts. Another way is to have the software detect the cases where the hazards may occur and assure sequential execution whenever necessary.

7.3.4. Mask Registers

There are cases when we want to do conditional vector operations, which may correspond to the conditional statements within loops. This feature is normally supported by having some mask registers which are widely used in SIMD machines [Cray82] [Lincoln82]. The entries of a mask register are either set to one or cleared to zero. There are many mask registers, and frequently used mask patterns can be kept in a mask register. The mask register allocation is similar to that of the branch registers (see chapter 3 for the function of the branch registers). As we will explain shortly, these mask registers are also used in vector editing instructions (*e.g.*, gather, scatter). The contents of a mask register may be determined either by the compiler or by executing a vector comparison instruction, which is the topic of the next section.

7.3.5. Comparison Instructions

The contents of a mask register can be loaded by doing an element-by-element vector comparison. Given two vectors **A** and **B** of length n, the result of the vector comparison is stored in **M**. The semantics are that $\mathbf{M}_i$ gets the comparison result of $\mathbf{A}_i$ and $\mathbf{B}_i$. That is, the result of a comparison is treated as an ordinary operand. We term such comparison a *logical* one in the sense the result of the comparison, rather than one of the operands, is returned. This scheme is used by some supercomputers (*e.g.*, Cray-1) to set up a mask register.

An application of vector logical comparison is given below. The application is to determine the zero entries in an array of floating point numbers. The value of an entry is treated as zero if its floating point representation is less than a given threshold value. Thus, a logical comparison of a queue (corresponding to the array) with a threshold value scalar returns a mask which shows the locations of non-zero entries. The scalar counterpart is useful in evaluating Boolean expressions without

using branches. That is, logical instructions such as AND, rather than branches, are used to evaluate complex Boolean expressions. Branches are needed only to construct high level program structures. This approach of evaluating Boolean expressions, however, may conflict with some language semantics, such as the Boolean short-circuit evaluation [Logothetis81].

There are cases where one of the operands involved in the comparison, rather than the logical comparison result, is needed. We propose another set of comparisons called *contents* comparisons. One such example is the function of Max2 used in the code scheduler. Max2 returns the bigger number of its two arguments, which is exactly the semantics provided by the contents comparison. Having this comparison, functions such as the Max2 can be realized without using branch instructions. This corresponds to the if-expression in some programming languages. The (vector) contents comparison is generally useful in coping with sorting-related problems. In a later section, we will sketch the steps of the merge sort algorithm by using the contents comparison instruction. The accessing modes of the operands participating in the contents comparison may depend on the comparison outcome and are specified as part of the contents comparison instruction.

7.3.6. Vector Editing Instructions

Vector editing instructions are used for conditional vector operations, sparse matrix computations, and other data editing applications. Examples of vector editing instructions include gather/scatter (sometimes called compress/expand). Gathering a vector **A** means that the elements of **A**, marked with 1s in the corresponding locations of the specified mask register, are copied into another vector **B**, where these elements are stored in an order-preserved manner. Scattering a vector is the opposite of gathering. An example of gathering is shown below.

Example 7.3. This example explains the gather operation.

mask register	1	1	0	0	1	1
A	A_1	A_2	A_3	A_4	A_5	A_6
B (compressed A)	A_1	A_2	A_5	A_6		

□

Different vector editing instructions can be realized by the repeat statements (vector instructions) with slightly different accessing modes (destructive, non-destructive, circular) described earlier. We made no attempt to completely specify all vector editing instructions. In the above example, either the destructive mode or the circular mode can be used, depending on whether the compiler decides to keep the vector **A** in a register or not. It is also possible to include scalars within a vector editing instruction. An example of value broadcasting is given below.

Example 7.4. In this example, we present the realization of the selective broadcasting. An entry of the result vector **B** gets the value of S, provided the corresponding mask bit is set. Otherwise, the corresponding entry in **A** is assigned.

mask register	1	1	0	0	1	1
S (a scalar)	S					
A	A_1	A_2	A_3	A_4	A_5	A_6
B	S	S	A_3	A_4	S	S

In this example, either the destructive mode or the circular mode is applicable. □

Merge (combining two vectors into one) and split (the opposite of merge) can be carried out by two repeat-statements with proper adjustments to the mask register.

7.4. Performance Evaluation

The five Livermore loops (loops 5†, 6†, 11, 13, 14) used to compare the speedup of software pipelining are used again as sample programs to compare the

† In the original Livermore loops, loops 5 and 6 are unrolled three times. For the discussion of this section, however, we again used the version without the unrolling.

performance of the vector extension described in this chapter with the methods described in the previous chapters.

Livermore loop 11 reads as:

```
      do 11 k = 2, 1000
11        x(k) = x(k-1) + y(k)
```

The semi-vectorized code generated for the 11-th loop is shown below:

```
S1 ENTER     %S1      Y                    /* addr of Y
S2 REPEAT    1000                          /* load Ys
S3   LDLO    %Q1      %S1      1           /* auto_inc load to %Q1
S4 MOVNFF    %Q2      %Q1.D                /* move X[1] := Y[1]
S5 REPEAT    999                           /* do the computation
S6   ADDF    %Q2      %Q2.C    %Q1         /* %Q2 in circular mode
S7 ENTER     %S2      X                    /* addr of X
S8 REPEAT    1000                          /* store Xs
S9   STLO    %Q2.D    %S2      1           /* auto_inc store from %Q2
```

We assume an infinitely long queue register (*i.e.*, %Q2) in generating this code segment. In the steady state, one result will be generated every four clock periods, which is the time needed to do a floating point multiplication. The steady state execution speed is limited by the data dependencies imposed by the program. Because the repeat statement applies to the above example successfully, we consider loop 11 as a vectorizable loop under this extension. The performance comparison of the aforementioned loops is shown in Table 7.2.

For loops 5, and 6, the software pipelined code, the decoupled code, and the vector code require the same execution time, which is limited by data-dependence. The vectorized code does not generate results any faster. The advantage, however, is that the first level controller is available to issue instructions following the loop. For loop 11, the execution speed of non-vectorized code is limited by the execution time of a branch instruction. On the other hand, the execution speed of vectorized code is bounded by data-dependence.

Table 7.2. Execution Time (of One Iteration) of Five Livermore Loops.

loops	scheduled	software pipelined	decoupled	vectorized
5	26	24	24	24
6	26	24	24	24
11	9	6	6	5
13	47	39	40	9
14	54	44	50	11

For the vector performance of loops 13 and 14, we assume that the memory system does not cause any extra delay other than memory bank busy time. In other words, we assume a memory system with an infinite number of banks and unlimited memory ports, which may be overly optimistic. Except for the memory accessing operations, we assume there is one functional unit for each distinct function[3]. Under the assumption of such a memory system, the bottleneck is the floating-point adder. It is also interesting to note that the decoupled mode for loops 13 and 14 runs slower than the well-scheduled software pipelined code because of the shared variable problem. That is, there is some execution time penalty if a variable is computed on one processor and will be used by another processor immediately after its value has become known. This penalty comes from the fact that in the original PIPE architecture, the value of a shared variable is passed via the memory system. The vector extension described in this section does have the flavor of decoupled architecture in that it uses multiple functional units. Since all registers in this extension are shared among all functional units, there is no execution time penalty if an operand is computed by one functional unit and will immediately be used by a

[3] However, we assume floating point addition and subtraction share the same functional unit.

different unit. As explained before, the result can even be bypassed using the data-available-queue.

7.5. Intrinsic Functions

Some vector operations, such as vector sum (the summation of all elements in a vector), are inherently difficult to vectorize. One possible way to cope with these hard-to-vectorize operations is for the compiler to generate scalar instructions [Cray82], which implies relatively low performance. Another possible approach is to include a large set of vector macro instructions and hope that most hard-to-vectorize operations are covered by the vector macros [Lincoln82]. In this extension, most vector macros can be composed by several of PIPE's vector instructions. We will show the realization of the following vector macros.

(1) Vector Sum.

We do the following steps to carry out the vector sum operation:

```
{ assume that VL has the vector length }
REPEAT     VL
    LD     %Q1  vector
{ clear %S1 }
ENTER      %S1  0
REPEAT     VL
    ADD    %S1  %S1  %Q1.D
```

(2) Inner Product.

Inner product is an element-by-element vector multiplication followed by a vector sum.

(3) Linear recurrence.

Loop 11 of the Livermore loops (which has been shown in an earlier section) is an example of first order linear recurrence.

(4) Minimum/Maximum function

We can use a contents comparison of a stream and a scalar to get an extreme value.

(5) Search.

We can use a logical comparison of a stream and a scalar followed by a mask count instruction (which is just a repeated addition) to tell if a particular value is in a given stream.

(6) Sorting.

We can use a contents comparison to merge two sorted streams. A sentinel must be used at the end of each stream to guard against the case where all elements of a stream have been consumed. The above method is the well-known merge sort.

We believe that most, if not all, programs can take advantage of this extension. However, we need advanced compiler techniques to fully utilize the potential parallelism provided by this extension.

7.6. Summary

In this section, we summarize the suggested features in this vector extension to amend the limitations of the original PIPE listed at the beginning of this chapter. Multiple queues are included, each of which can be the destination of a load instruction. During decoupled execution mode, the load instructions issued at different processors may choose not use the same queue register as the destination register. Thus, any processor can load operands for any other processor, which provides the flexibility of doing better load balancing between processors.

This vector extension uses a simple vector instruction format, which implies very short vector-startup time (one clock period). This simple format, however, is

very powerful to use in conjunction with the queue registers and different access modes. As demonstrated earlier in this chapter, many array-oriented programs are vectorizable under this extension. The memory is still needed to pass the values of shared variables between processors in the decoupled mode. Shared variables in a single processor of this PIPE extension use the same sets of registers. The destination non-blocking interconnection network avoids the unnecessary result bus conflicts. With the advent of VLSI, it is generally believed that non-block interconnection networks can be built within a reasonable cost for the size of machine we are designing. Branch instructions are needed only to carry out the high level language control structures by introducing the logical/contents comparison. Elements of a queue can be read repeatedly, provided an adequate access mode is specified.

The preliminary study suggests that this extension is a cost-effective way to implement a processor for array-oriented programs. The major remaining problems include (a) investigating its support for non-numerical applications; and (b) designing compiler techniques to fully utilize the suggested features automatically.

Chapter 8

Conclusions

We have demonstrated the feasibility of using the LDQ to reduce the impact of memory delay and of using the PBR instruction as a generalized delayed jump. A simple code scheduler is capable of reordering the compiled code to take advantage of these special features automatically. Software pipelining can take advantage of the aforementioned features even further (in particular, the PBR instruction). The degree of prefetching (*i.e.*, the number of prefetches across the loop boundary) is determined by the execution times of different pipes in a processor. Our scheduling methods are applicable to most register-register pipelined architectures by simply changing the cost table which shows the execution times and issue conditions of instructions. The scheduling methods described in this paper will be less effective for memory-memory pipelined processors. Incidentally, but not accidentally, register-register architectures are prevalent among the scalar mode of most high performance computers (*e.g.*, Cray-1 [Russell78], Cyber 205 [Lincoln82]) for control simplicity and other reasons. Even in the 360/370 family, where upward compatibility is critical, the internal organization of some high-end pipelined processors is of the register-register form (*e.g.*, the floating point unit of the 360/91 [Tomasulo67]). Thus, we believe that queues and PBR instructions are compatible with super-computers and that the scheduling methods described in this paper are effective in utilizing special features available in PIPE. The code scheduling methods described in this thesis are also applicable to the proposed extension.

The scheduling methods described above assume that all results computed by instructions in a basic block are needed at the end of the basic block. This is true

only for a branch instruction where the instructions from the branch target cannot be issued until the transfer control (the XBR point) completes (for a successful branch). This assumption leads to the conclusion that all cumulative costs have non-negative values. It is possible to consider the uses of an object in all its successor basic blocks. If an object A will not be used, in all its successor blocks, until at least n clock periods later and it takes m clock periods to compute A, the instruction that computes A would have a cumulative cost of $m-n$, rather than m. If $n>m$, the cumulative cost can have a negative value. This modified cumulative cost reflects the urgency of objects across the basic block boundaries. The effects of this modification are a subject for future research.

The code generator for the decoupled execution mode generates correct code for the decoupled mode. A static strategy is used in deciding where an instruction should go. That is, the code generated on A-processor does the following:

(1) Address calculation.

(2) Branch determination.

(3) Load/store instructions.

The proposed extension provides other alternatives for generating the code. The investigation of compiler techniques to balance the load of the two processors according to different applications would be intriguing.

As demonstrated in chapter 7, many programs which must be executed in the slow mode on most supercomputers can be executed in the fast mode of the proposed extension. However, compiler techniques that recognize parallelism within sequential programming languages and take advantage of the underlying architectural features, are yet to be developed. One may also look into new programming languages which facilitate the compilation for the control-driven dataflow architecture.

We conclude that the data queues, the prepare-to-branch instruction, and the decoupled execution mode of the PIPE architecture are useful features. Compilers can be constructed to take advantage of these features effectively. The performance of the PIPE architecture can be substantially enhanced by introducing the vector extension.

Some related research topics are outlined below:

(1) Register allocation. The interaction between the code scheduling and register allocation is worth studying. A new strategy is needed to allocate shared-variables in the decoupled code. Sometimes, a shared variable assigned to a register might degrade the entire system performance. Register allocation for special registers (*e.g.*, branch registers in the original PIPE, the queue registers in the PIPE extension) is also important.

(2) Different configurations for decoupled mode. One alternative configuration is to use multiple AP's and/or multiple EP's for a sequential task. Another functional arrangement is to put multiple AP's and EP's together to build a multiprocessor system where each node is a PIPE machine or its extension.

(3) Procedure call/return. It is important to support fast procedure call/return for non-numerical applications. The original PIPE architecture provides two sets of general purpose registers as a degenerated sliding window scheme used in RISC [Patterson81] [Patterson82]. It worth studying the effectiveness of the two-set-register scheme in supporting fast procedure call/return.

(4) Languages/Algorithms. New languages which express some architectural features explicitly may simplify the task of generating efficient code. Different algorithms may take advantage of the system configuration and result in fast execution.

REFERENCES

[Abdel-Wahab76] Abdel-Wahab, H.M., *Scheduling with Applications to Register Allocation and Deadlock Problems*, Doctoral Thesis, Department of Electrical Engineering, University of Waterloo, Waterloo, Ontario.

[Adam74] Adam, Thomas L., K.M. Chandy, and J.R. Dickson "A Comparison of List Schedules for Parallel Processing Systems," *Communication of ACM 17*, 12, pp. 685-690, December, 1974.

[Aho77] Aho, Alfred V., and Jeffrey D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass., 1977.

[Amdahl67] Amdahl, Gene, "The Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities," *Proceedings, AFIPS Spring Joint Computer Conference*, pp. 483-485, April 1967.

[Anderson67] Anderson, D.W., F.J. Sparacio, and R.M. Tomasulo, "The IMB System/360 Model 91: Machine Philosophy and Instruction-Handling," *IBM Journal of Research & Development, 11*, 1, pp. 8-24, January 1967.

[Axelrod83] Axelrod, T.S., P.F. Dubois, and P.G. Eltgroth, "A Simulator for MIMD Performance Prediction--Application to the S-1 Mark IIA Multiprocessor," *Proceedings, 1983 International Conference on Parallel Processing*, pp. 350-357, August, 1983.

[Brantley82] Brantley, William C., and Joseph Weiss, "FOM: A Fortran Optimized Machine--A High Performance, High Level Language Machine," *IBM Research Report RC 9640 (#40815)* March 1982.

[Brantley83] Brantley, William C., and Joseph Weiss, "Organization and Architecture Trade-offs in FOM," *IEEE International Workshop on Computer Systems Organization*, pp. 139-143, March 1983.

[Bucher83] Bucher, Ingrid Y., "The Computational Speed of Supercomputers," *Proceedings, ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pp. 151-165, August 1983.

[CDC73] Control Data Corporation, *Control Data STAR-100 Features Manual*, St. Paul, MN., pub. no. 60425500, October, 1973.

[Charlesworth81] Charlesworth, Alan E., "An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family," *IEEE Computer 14*, 9, pp. 18-27, September 1981.

[Cohler81] Cohler, Edmund. U., and James. E. Storer, "Functionally Parallel Architecture for Array Processors," *IEEE Computer 14*, 9, pp. 28-36, September 1981.

[Craig83] Craig, Gary L., James R. Goodman, Randy H. Katz, Andrew R. Pleszkun, Kishore Ramachandran, John Sayah, and James E. Smith, "PIPE: A High Performance VLSI Processor Implementation," *University of Wisconsin-Madison, Computer Sciences Department Technical Report # 513*, September 1983.

[Cray82] Cray Research, Inc. *Cray-1 Computer Systems* S Series Mainframe Reference Manual (HR-0029), 1982.

[Cray83] Cray Research, Inc., *Cray-1 and Cray X-MP Computer Systems, CAL Assembler Version 1 Reference Manual*, SR-0000, 1983.

[Cray85] Cray Research, Inc., *Advanced Large-scale and High-Speed Multiprocessor System for Scientific Applications*, Cray X-MP-4 Series, 1985.

[Datawest79] Datawest Corp., *Real Time Series of Microprogrammable Array Transform Processors*, Prod. Bulletin Series B, 1979.

[Dennis80] Dennis, Jack B., "Data Flow Supercomputers," *IEEE Computer 13*, 11, pp. 48-56, November, 1980.

[Dongarra79] Dongarra, J.J., and A.R. Jinds, "Unrolling Loops in Fortran," *Software Practice and Experience 9*, 3, pp. 219-226, March 1979.

[Fisher81] Fisher, Joseph A., "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers C-30*, 7, pp. 478-490, July, 1981.

[Fisher84a] Fisher, Joseph A., John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau, "Parallel Processing: A Smart Compiler and a Dumb Machine," *Proceedings, ACM SIGPLAN'84 Symposium on Compiler Construction*, pp. 7-47, June 1984.

[Fisher84b] Fisher, Joseph A., "The VLIW Machine: A Multiprocessor for Compiling Scientific Code," *IEEE Computer 17*, 7, pp 45-53, July, 1984.

[Flynn66] Flynn, Michael J., "Very High-Speed Computing Systems," *Proceedings of the IEEE 54*, 12, pp. 1901-1909, December 1966.

[Flynn72] Flynn, Michael J., "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers C-21*, 9, pp. 948-960, September 1972.

[Gajski82] Gajski, D.D., D.A. Padua, D.J. Kuck, and R.H. Kuhn, "A Second Opinion on Data Flow Machines and Languages," *IEEE Computer 15*, 2, pp. 58-70, February, 1982.

[Garey79] Garey, Michael R., and David S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, 1979.

[Gonzalez77] Gonzalez, Mario J. Jr., "Deterministic Processor Scheduling," *ACM Computing Survey 9*, 3, pp. 173-204, September, 1977.

[Goodman85] Goodman, James R., Jian-tu Hsieh, Koujuch Liou, Andrew R. Pleszkun, P.B. Schechter, and Honesty C. Young, "PIPE: a Decoupled Architecture for VLSI," to appear *Proceedings, the 12th International Symposium on Computer Architecture*, June, 1985.

[Gross82] Gross, Thomas R., and John L. Hennessy, "Optimizing Delayed Branches", *Proceedings, 15th Annual Workshop on Microprogramming*, pp. 114-120, October 1982.

[Hennessy81] Hennessy, John, Norman Jouppi, Forest Baskett, and John Gill, "MIPS: A VLSI Processor Architecture," *Technical Report No. 223, Computer Systems Laboratory, Stanford University*, November 1981.

[Hennessy83] Hennessy, John, and Thomas Gross, "Postpass Code Optimization of Pipeline Constraints," *ACM Transactions on Programming Languages and Systems 5*, 3, pp. 422-448, July 1983.

[Hsieh84] Hsieh, Jian-tu, Andrew R. Pleszkun, and James R. Goodman, "Performance Evaluation of the PIPE Computer Architecture," University of Wisconsin-Madison, *Computer Sciences Department Technical Report #566*, November, 1984.

[Hunt77] Hunt, H.B. III, T.G. Szymanski, and J.D. Ullman, "Operations on Sparse Relations," *Communication of ACM 20*, 2, pp. 171-176, March 1977.

[Hwang84] Hwang, Kai, and Fayé A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill Book Company, 1984.

[IBM76] IBM Corp., *IBM 3838 Array Processor Functional Characteristics*, October, 1976.

[Knuth73a] Knuth, Donald E., *The Art of Computer Programming. Volume 1: Fundamental Algorithms*, 2nd Ed., Addison-Wesley Publishing Company, pp. 258-268, 1973.

[Knuth73b] Knuth, Donald E., *The Art of Computer Programming. Volume 3: Sorting and Searching*, 2nd printing, Addison-Wesley Publishing Company, pp. 451-471, 1973.

[Kogge81] Kogge, Peter M. *The Architecture of Pipelined Computers*, McGraw-Hill, New York, 1981.

[Kozdrowicki80] Kozdrowicki, Edward W., and Douglas J. Theis, "Second Generation of Vector Supercomputers," *IEEE Computer 13*, 11, pp. 71-83, November, 1980.

[Kuck80] Kuck, David J., Robert H. Kuhn, Bruce Leasure, and Michael Wolfe, "The Structure of an Advanced Retargetable Vectorizer," *Proceedings, IEEE COMPSAC*, pp. 709-715, October 1980.

[Lee84] Lee, Johnny K.-F., and Alan Jay Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer 17*, 1, pp. 6-22, January 1984.

[Lincoln82] Lincoln, Neil R., "Technology and Design Tradeoffs in the Creation of a Modern Supercomputer," *IEEE Transactions on Computers C-31*, 5, pp. 349-362, May 1982.

[Logothetis81] Logothetis, George, and Prateek Mishra, "Compiling Short-Circuit Boolean Expressions in One Pass," *Software Practice and Experience 11*, pp. 1197-1241, 1981.

[Matsurra84] Matsurra, Toshihiko, Sachio Kamiya, and Masaaki Takiuchi, "Design Concept of the Facom VP Based on Extensive Analysis of Applications," *Proceedings, International Conference on Computer Design: VLSI in Computers*, pp. 232-237, October, 1984.

[McMahon72] McMahon, F.H., "Fortran CPU performance Analysis," *Lawrence Livermore Laboratories*, 1972.

[Miura83] Miura, Kenichi, and Keiichiro Uchida, "FACOM Vector Processor System:VP-100/VP-200," *Proceedings, NATO Advanced Research Workshop on High Speed Computing*, West Germany, June 1983, reprinted in *IEEE, Tutorial Supercomputers: Design and Applications*, Kai Hwang (Ed.), pp. 59-73, August, 1984.

[Nagashima84] Nagashima, Shigeo, and Yasuhiro Inagami, "Design Consideration for a High-Speed Vector Processor: The HITACHI S-810," *Proceedings, IEEE International Conference on Computer Design: VLSI in Computers*, pp. 238-243, October, 1984.

[Patel76] Patel, Janak H., and Edward S. Davidson, "Improving the Throughput of a Pipeline by Insertion of Delays," *Proceedings, IEEE/ACM the Third Annual International Symposium on Computer Architecture*, pp. 159-163, 1976.

[Patterson81] Patterson, David A., and Carlo H. Séquin, "RISC I: A Reduced Instruction Set VLSI Computer," *Proceedings, IEEE/ACM the Eight Annual International Symposium Computer Architecture*, pp. 443-457, April 1981.

[Patterson82] Patterson, David A., and Carlo H. Séquin, "A VLSI RISC," *IEEE Computer, 15*, 9, pp. 8-21, September 1982.

[Pleszkun82] Pleszkun, Andrew R., "A Structured Memory Access Architecture," *Computer Systems Group Report CSG-10*, Coordinate Science Laboratory, University of Illinois, Urbana, October, 1982.

[Pleszkun83] Pleszkun, Andrew R., and Edward S. Davison, "A Structured Memory Access Architecture," *Proceedings, IEEE International Conference on Parallel-Processing*, pp. 461-471, August 1983.

[Radin82] Radin, George, "The 801 Minicomputer," *ASPLOS SIGARCH Computer News, 10*, March 1982. Reprinted in *IBM Journal of Research and Development 27*, 3, pp. 39-47, May 1983.

[Rau81] B. Ramakrishna Rau, and Christopher D. Glaeser, "Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High

Performance Scientific Computing," *Proceedings, 14th Annual Workshop on Microprogramming*, pp. 183-198, October 1981.

[Riganati84] Riganati, John P., and Paul B. Schneck, "Supercomputing," *IEEE Computer 17*, 10, pp. 97-113, October, 1984.

[Riseman72] Riseman, Edward M., and Caxton C. Foster, "The Inhibition of Potential Parallelism by Conditional Jumps," *IEEE Transactions on Computers C-21*, 12, pp. 1405-1411, December 1972.

[Rudsinski77] Rudsinski, L., and J. Worlton, "The Impact of Scalar Performance on Vector and Parallel Processors", *Proceedings, the Symposium on High Speed Computer and Algorithm Organization*, pp. 451-452, April 1977.

[Russell78] Russell, Richard M., "The Cray-1 Computer System," *Communications of ACM 21*, 1, pp. 63-72, January 1978.

[Schorr71] Schorr, H., "Design Principles for a High-Performance System," *Proceedings, Symposium on Computers and Automata*, April, 1971.

[Shively82] Shively, Richard R., "Architecture of a Programmable Digital Signal Processor," *IEEE Transactions on Computers C-31*, 1, pp. 16-22, January, 1982.

[Smith81] Smith, James E., "A Study of Branch Prediction Strategies," *Proceedings, IEEE/ACM the Eighth Annual International Symposium on Computer Architecture*, pp. 135-142, May 1981.

[Smith82] Smith, Alan Jay, "Cache Memories," *ACM Computing Surveys 14*, 3, pp 473-530, September, 1982.

[Smith83] Smith, James. E., Andrew R. Pleszkun, Randy H. Katz, and James R. Goodman, "PIPE: A High Performance VLSI Architecture," *Proceeding, IEEE International Workshop on Computer Systems Organization*, pp. 131-138, March 1983. Also available as *University of Wisconsin-Madison Computer Sciences Department Technical Report # 512*, September, 1983.

[Smith84] Smith, James E. "Decoupled Access/Execute Computer Architecture," *ACM Transactions on Computer Systems 2*, 4, pp. 289-308, November 1984.

[Thorlin67] Throlin, J.F. "Code Generation for PIE (Parallel Instruction Execution) Computers," *Proceedings ,AFIPS Spring Joint Computer Conference*, pp. 641-643, April, 1967.

[Thornton70] Thornton, J. E., *Design of a Computer, The Control Data 6600*, Scott, Foresman and Co., Glenview, Ill. 1970.

[Tomasulo67] Tomasulo, R. M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development 11*, 1, pp.25-33, January 1967.

[Ullman75] Ullman, J.D. "NP-complete Scheduling Problems," *J. Comput.*

Syst. Sci. 10, 3, pp. 84-393, June, 1975.

[Watson72] Watson, W.J., "The TI ASC: A Highly Modular and Flexible Super Computer Architecture," *Proceedings, AFIPS Fall Joint Computer Conference 41*, pt. 1, pp. 221-228, 1972.

[Weiss84a] Weiss, Shlomo, "Very High Performance Scalar Processing," *Technical Report ECE-84-22, Electrical and Computer Engineering Department*, University of Wisconsin-Madison, September 1984.

[Weiss84b] Weiss, Shlomo, and James R. Smith, "Instruction Issue Logic in Pipelined Supercomputers," *IEEE Transactions on Computers C-33*, 11, pp. 10133-1022, November, 1984.

[Wood78] Wood, Graham, "On the Packing of Micro-instruction Words," *Proceedings, 11th Annual Workshop on Microprogramming*, pp. 51-55, December 1978.

[Worlton84] Worlton, Jack, "Understanding Supercomputer Benchmarks," *Datamation*, pp. 121-130, September, 1984.