

# Architectural Design and Analysis of a VLIW Processor

*Arthur Abnous and Nader Bagherzadeh*

Department of Electrical and Computer Engineering

University of California, Irvine

Irvine, CA 92717

Phone: (714) 856-8720

e-mail: aabnous@balboa.eng.uci.edu, nader@balboa.eng.uci.edu

## Abstract

Architectural design and analysis of VIPER, a VLIW processor designed to take advantage of instruction level parallelism, are presented. VIPER is designed to take advantage of the parallelizing capabilities of Percolation Scheduling. The approach taken in the design of VIPER addresses design issues involving implementation constraints, organizational techniques, and code generation strategies. The hardware organization of VIPER is determined by analyzing the efficiency of various organizational strategies. The relationships that exist among the pipeline structure, the memory addressing mode, the bypassing hardware, and the processor cycle time are studied. VIPER has been designed to provide support for multiway branching and conditional execution of operations. An integral objective of the design was to develop the code generator for the target machine. The code generator utilizes a new code scheduling technique that is devised to reduce the frequency of pipeline stalls caused by data hazards.

## 1 Introduction

Concurrency is a key element in achieving high performance in a processor. In order to speed up program execution, different parts of the computation should be executed in parallel. Traditionally, parallel processing has been applied to the execution of high-level language constructs such as loops [1]. This type of parallelism is known as coarse-grain parallelism whereby one tries to overlap the execution of different parts of the computation at the level of operations seen by the high-level language programmer. MIMD parallel architectures have traditionally taken advantage of this type of parallelism [2].

With the recent development of advanced compilation techniques such as Trace Scheduling [3] and Percolation Scheduling [4], it has become feasible to exploit parallelism at the level of machine instructions. This type of parallelism is known as fine-grain parallelism. VLIW (Very Long Instruction Word) architectures exploit fine-grain parallelism in order to speed up program execution.

The objective of this paper is to present the architectural design and analysis of VIPER (VLIW Integer ProcEссор), a VLIW processor designed to exploit fine-grain parallelism, and investigate its effectiveness for integer processing tasks. The architecture of the processor is based on the Percolation Scheduling (PS) parallelizing compiler that has been developed at the University of California at Irvine [5]. An important characteristic of VLIW processors is that the compiler has full knowledge of the micro-architecture of the

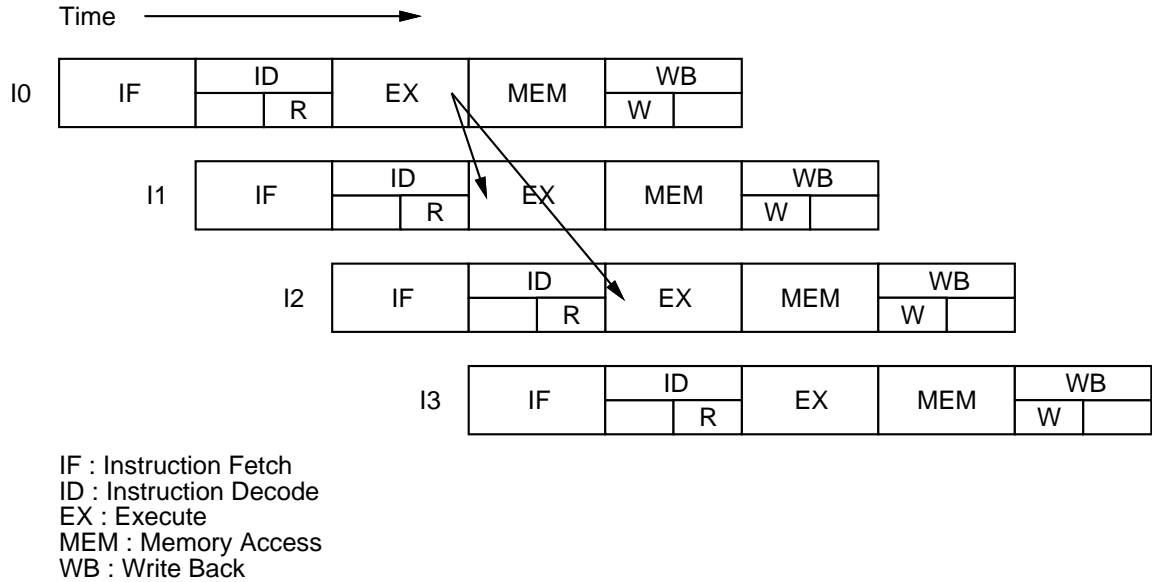


Figure 1: Pipeline Structure of a Typical RISC Processor

processor. This means that the code generation strategy, the architecture, and the organization of the processor are very closely related to each other. The architecture of the processor was developed with a CMOS VLSI implementation in mind. An integral part of the design was the development of the code generator module of the PS compiler for the target architecture. Hardware/software trade-offs were studied at several points during the design process. This approach allows the architect to address design problems with a combination of hardware *and* software solutions and results in improved performance.

## 1.1 Pipelining and RISC processors

One of the most important goals of RISC (Reduced Instruction Set Computer) processors is efficient pipelining [7, 8]. The instruction sets of RISC processors are simple and are designed with efficient pipelining and decoding in mind. Because of the simplicity of the instruction set, the underlying hardware of a RISC processor is simple and can run at high speeds. Because of efficient pipelining, the CPI factor of RISC processors comes very close to one. This characteristic is responsible for the performance advantage of RISC processors compared to traditional CISC (Complex Instruction Set Computer) processors.

Figure 1 shows the pipeline structure typically used in RISC processors [9, 10, 11]. The pipeline consists of five stages: IF (Instruction Fetch), ID (Instruction Decode), EX (EXecute), MEM (MEMory access), and WB (Write Back). This pipeline structure provides for a high execution throughput in RISC processors.

In the pipeline structure shown in Figure 1, there is a delay of two cycles between the ID and WB stages. This means that the result generated by instruction I0 will not be written back to the register file in time for instructions I1 and I2 to read when they enter the ID stage. To alleviate the possible read-after-write (RAW) hazard caused by this delay, RISC processors use a hardware technique known as *bypassing* [12, 10]. At the end of the EX stage, the result of each instruction is fed back to the input of the execution

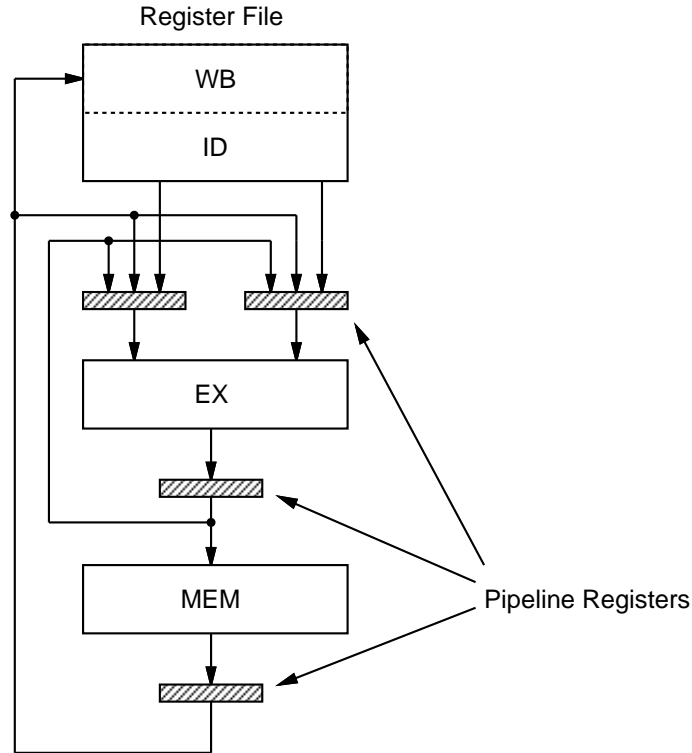


Figure 2: Block Diagram of Bypassing Hardware

unit corresponding to the EX stage. The bypassing hardware compares the source register addresses of the instruction about to enter the EX stage to the destination register address of previous instructions. If there is a match, the operand read from the register file is discarded, and instead, the result of the previous EX stage is used. In the pipeline structure shown in Figure 1, because the delay between ID and WB stages is two cycles, two levels of bypassing are needed. This means that source register addresses of I2 are compared to the destination register addresses of I1 and I0. A block diagram of the bypassing mechanism is shown in Figure 2. An important element in this scheme is that register file write operations take place during the first half of each cycle, and register file read operations take place in the second half. Thus, the result of I0 is back into the register file in time for I3 to read, and there is no need for a third level of bypassing.

### 1.1.1 Delayed Loads

In most RISC processors, RAW hazards that arise because of a dependency on the result of a load operation cannot be resolved by the bypassing hardware. RISC processors typically use the *Displacement* addressing mode for load/store operations. In this addressing mode, the effective address is computed by adding an immediate offset to the content of a register. The addition is done in the EX stage and the data cache is accessed in the MEM stage. The result of the instruction is not ready until the end of the MEM stage; thus, it cannot be bypassed to the next instruction. If the instruction after a load uses the result of the load instruction, there is a RAW hazard that cannot be resolved by bypassing. Instead of stalling the processor, the instruction after a load instruction is always executed. It

is the responsibility of the compiler to schedule an instruction in the *delay slot* of the load instruction that does not result in a RAW hazard. If the compiler cannot find such an instruction, it schedules a NOP (no operation) instruction in the delay slot of a load instruction.

## 1.2 VLIW Architectures

VLIW architectures are considered to be one of the promising methods of increasing performance beyond standard RISC architectures. While RISC architectures only take advantage of temporal parallelism (by using pipelining), VLIW architectures can also take advantage of spatial parallelism by using multiple functional units to execute several operations concurrently. Some of the key features of a VLIW processor are [13]:

1. Multiple functional units connected through a global shared register file.
2. A central controller that issues a long instruction word every cycle.
3. Each instruction consists of multiple independent parallel operations.
4. Each operation requires a statically known number of cycles to complete.

Instructions in a VLIW architecture are very long (hence the name VLIW) and may contain hundreds of bits. Each instruction contains a number of operations that are executed in parallel. Operations in VLIW instructions are scheduled by the compiler. VLIW processors rely on advanced compilation techniques such as Percolation Scheduling that expose instruction level parallelism beyond the limits of basic blocks. The micro-architecture of a VLIW processor is completely exposed to the compiler, and the compiler has full knowledge of operation latencies and resource constraints of the processor implementation.

In recent years, there have been several efforts to design and develop VLIW architectures. Multiflow's Trace was one of the pioneer architectures in this field; its design was expandable to support 1024-bit instructions by concatenating 256-bit processor boards [13]. VLIW ideas have also surfaced in the designs of Cydrome's Cydra-5 [14], iWARP [15], and LIFE [16].

Superscalar processors are similar to VLIW processors in that they also improve performance by executing multiple instructions in each cycle. Superscalar processors detect parallelism at run-time. This is done by analyzing the stream of instructions that are being fetched [17]. Superscalar processors demand more hardware support in order to manage synchronization among concurrent operations. The control paths of superscalar processors are often very complicated. VLIW machines schedule operations at compile-time. This greatly simplifies the control paths of VLIW processors because they do not have to detect dependencies at run-time. Also, compile-time scheduling allows VLIW processors to take advantage of *global* optimizations that can be performed by sophisticated compilation techniques. Superscalar processor can only analyze a limited window of instructions at any given time. The advantage of superscalar processors is that they can be binary compatible with a previous architecture.

## 1.3 Summary

The approach taken in the architectural design of VIPER was to consider design issues at hardware *and* software levels. An integral objective of the project was to develop the

code generation module of the PS compiler for the target architecture. Hardware/software trade-offs were analyzed at various points during the design process. This constitutes a comprehensive approach to architectural design where the design process takes into consideration and correlates design issues involving implementation constraints, organizational techniques, and code generation strategies. Design issues were evaluated by quantitative analysis. Extensive simulations were performed to verify design decisions.

## 2 The Compiler System

Exploiting fine-grain parallelism is an important part of exploiting all of the parallelism available in a program. Although it had been believed for years that there was no significant amount of parallelism at the fine-grain level [18], this belief was based on experiments that were looking for parallelism only within basic block limits. However, within these limits, the search for parallelism is restricted by the average number of operations within a basic block which is on the order of 4 to 5 [18]. Percolation Scheduling tries to extend the potential parallelism by compacting across basic block boundaries while still preserving program correctness. This section presents an overview of the Percolation Scheduling compiler system that has been developed at UC Irvine [5].

### 2.1 The Execution Model

An input program is represented by a Control/Data Flow Graph (CDFG), as shown in Figure 3. The vertices (nodes) of the graph correspond to instructions executed in each cycle. Each node contains a set of operations that are executed in parallel. The edges represent flow of control from one node to its successor. Initially, all nodes contain a single operation corresponding to a machine instruction in the original sequential code. If this operation is *not* a conditional branch operation, then the node has only one outgoing edge representing the flow of control from this node to its *only* successor. If, on the other hand, this operation is a conditional branch, then the node has two successors for the true and false branches. Making a program “more parallel” involves compaction of several operations into one node while preserving the semantics of the original program. The presence of conditional branch operations can limit the compaction process unless the target architecture has explicit support for multiway branch operations. VIPER allows the presence of multiple conditional branch, as well as other operations, in each node. In the machine model assumed by the compiler, a node represents a large instruction word containing several operations (all of which are executed in parallel) and a tree-like structure of conditional branch operations. A single execution path is selected from the entry point of a node down to a unique successor. The path to the next node is selected according to the condition codes in the tree. For example, if we assumed that condition *A* is true and condition *B* is false in Figure 4, then *op1*, *op2*, *op3*, *op5*, and *op6* will be executed, and the successor of this node is  $L_1$ .

The execution of a node involves three basic steps:

1. All operands and condition codes are read.
2. All operations are executed, condition codes are evaluated, and a path to a unique successor instruction is chosen.

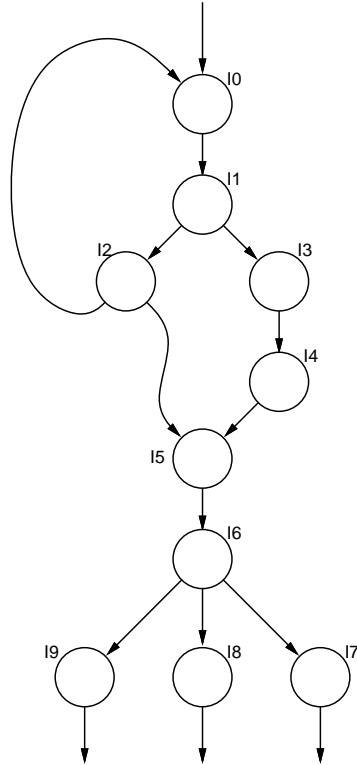


Figure 3: A Control/Data Flow Graph

3. The results of the operations on the selected path are written back to the register file or memory.

## 2.2 The Core Transformations

PS is a system of semantics-preserving transformations that convert an original program graph into a more parallel one. The core of PS consists of four transformations: *Move-op*, *Move-cj*, *Unify*, and *Delete*. These transformations are defined in terms of adjacent nodes in a program graph. They are combined with a variety of guidance rules (heuristics) to direct the optimization process. In Reference [19] it was shown that the core transformations are *complete* with respect to the set of all possible local, dependency-preserving transformations on programs. Thus, for all practical purposes, no alternate system of transformations based on the same principles (e.g., locality of application, dependency-preservation) can do better at exposing parallelism at the fine-grain level. A complete description of these transformations can be found in [20].

## 2.3 Hierarchical Approach

The application of the core transformations of PS to a given program is directed by a set of higher level transformations. These higher level transformations are needed when compacting a complete program that includes several basic blocks, loops, etc. Two of these higher level transformations are described next: *Maxcomp* and *Perfect Pipelining*.

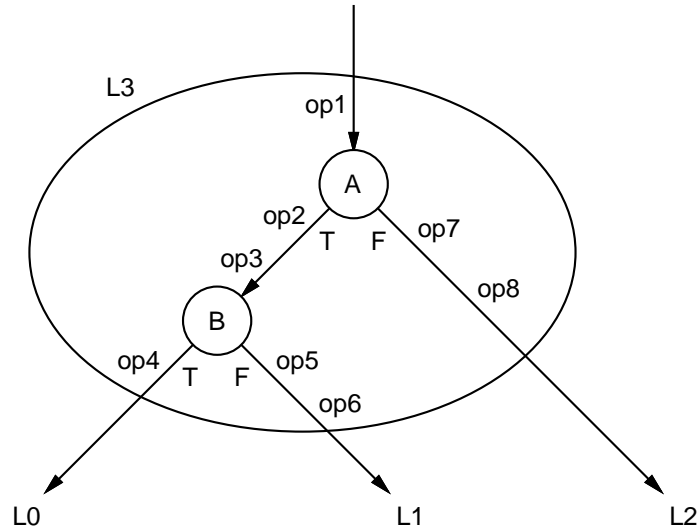


Figure 4: An Instruction Node

### 2.3.1 Maxcomp

Maxcomp is an algorithm for maximal compaction of a program. It tries to move operations as high as possible in the program graph while maintaining the original semantics of the program.

### 2.3.2 Perfect Pipelining

Programs tend to spend most of their time executing loops; therefore, compaction of loops has a major effect on overall performance. Loop Pipelining techniques are used to reduce the execution time of loops. The basic idea is to reorganize a loop so that successive iterations of the loop are executed in an overlapped fashion. Thus, a given iteration of the loop may begin before the completion of previous iterations. The PS compiler uses Perfect Pipelining [21], which is an algorithm for performing loop pipelining for general loops, including loops with conditional jumps inside the loop body.

## 2.4 Resource Constrained Scheduling

The ultimate goal of the compiler is to map the input program onto a given architecture. The mapping should take into account the hardware resources available in the target machine. This mapping process is called Resource Constrained Scheduling (RCS) and is known to be NP-hard in practice. This suggests that it should be solved by heuristics. The compiler first finds the unlimited-resources schedule (assuming that there are no resource limitations) and then applies a set of heuristics to map this schedule onto the given architecture. This strategy allows the core transformations to extract as much parallelism as possible without being limited by resource constraints.

## 2.5 The Scheduling Process

The scheduling process is performed as follows: The input program (which is a C source) is first transformed into an intermediate representation (three-address code) by the front-end of the compiler. The CDFG of the program is derived from the intermediate code. At this point, each node in the CDFG contains one operation. During the process of compaction, the objective is to move all operations in the program graph as high as possible while maintaining program correctness. First, Perfect Pipelining is applied to all innermost loops of the program. Maxcomp is applied next in order to compact operations outside the innermost loops as much as possible. Up to this point, only data and control flow dependencies restrict the process of compaction, i.e., a node may include more operation than the target machine can execute in each cycle. During code compaction, only *true* data dependencies are taken into account. A true data dependency occurs when an instruction uses a value produced by a previous instruction. In order to eliminate *false* data dependencies that are created by reusing registers, *register renaming* is used during the compaction process.

After having the unlimited-resources schedule, the RCS procedure is applied. The RCS algorithm scans all nodes of the program graph. For each node, the set of existing operations is evaluated to see if there are any resource constraint violations. If there are no resource constraint violations, the algorithm proceeds to the successor nodes of the current node. However, if there is a resource constraint violation, some operations must be deferred until the node does not violate any of the resource constraints of the target machine. In order to defer an operation, a new successor node is created. This new node will accommodate all the operations that are going to be deferred from the current node. This process continues until there are no resource constraint violations in the current node. Eventually, none of the nodes in the program graph violate any of the resource constraints. In order to fill the nodes that were created when operations were being deferred, the compiler tries to compact the code further, but this time, the process of compaction is not allowed to result in any resource constraint violations.

## 2.6 Simulations

The PS compiler includes a simulator that was utilized to evaluate architectural design decisions and code generation strategies. In order to analyze the effect of pipeline stalls caused by RAW hazards, the simulator was modified to maintain an internal representation of the execution pipeline. The state of the execution pipeline is monitored on a cycle-by-cycle basis during simulation.

To determine average performance for a set of programs, the harmonic mean of all speed-up factors is used. The harmonic mean assigns a larger weight to programs with smaller speed-up factors. This reflects the real effect of speed-up factors on the *total* execution time for *all* benchmarks.

In addition to the speed-up factor, the simulator also measures the following statistical information:

1. Frequency of individual operations.
2. Frequency of data hazards and pipeline stalls.
3. Dynamic count of NOP operations.

<i>Benchmark</i>	<i>Description</i>
binsearch	binary search algorithm
bubble	bubble sort algorithm
chain	finds the optimal sequence for chained matrix multiplication
factorial	computes the factorial of several numbers
fibonacci	computes a sequence of Fibonacci numbers
floyd	Floyd's algorithm to find shortest paths in a graph
matrix	matrix multiplication program
merge	sorting by merging algorithm
quicksort	Hoare's quicksort algorithm
dijkstra	Dijkstra's shortest path algorithm

Table 1: Benchmark Programs

### 2.6.1 Benchmark Programs

To evaluate the performance of the processor, a set of benchmark programs were written. These benchmarks include various integer processing programs that implement a variety of elementary algorithms. Table 1 contains a description of the benchmark programs.

## 3 Architectural Design of VIPER

The architectural design and analysis of VIPER are presented in this section. Two key aspects are stressed: an efficient instruction execution pipeline designed to reduce the frequency of pipeline stalls caused by data hazards in a VLIW processor with pipelined functional units, and architectural support for multiway branch operations. The architecture has been designed to reflect the execution model assumed by the PS compiler. Various organizational strategies are analyzed to evaluate the efficiency of their hardware implementations. Architectural decisions are made with full consideration of the constraints imposed by VLSI technology. Design problems are addressed with a combination of hardware and software techniques after evaluating hardware/software trade-offs. Simulation results are used to verify design decisions. This approach has been demonstrated to be very effective for VLSI processor design by RISC research efforts [7, 8].

### 3.1 Processor Configuration

Operations executed by a processor can be divided into three types. Each type of operation is executed by a corresponding type of hardware execution unit:

1. Control Transfer (CT) operations
2. Load/Store (LS) operations
3. Arithmetic/Logic (AL) operations

Based on this classification, the set of hardware resources available to a processor can be expressed in terms of the following parameters:

1. Maximum number of control transfer operations in each instruction ( $c$ )

2. Maximum number of load/store operations in each instruction ( $l$ )
3. Maximum number of arithmetic/logic operations in each instruction ( $a$ )

These parameters are determined by the number of execution units of each type available in the processor hardware.

A key element for a processor that maintains a CPI factor below one (by executing more than one operation per cycle) is the ability to fetch multiple instructions. The instruction fetch bandwidth available to the processor places an upper bound on the maximum performance the processor can attain. It is also important that the processor have sufficient hardware resources to execute all of the operations that are fetched in each cycle. In a VLIW processor, the instruction bus is very wide and requires many pins on the chip package. Beyond a certain point, it is more desirable (in terms of both cost and feasibility) to increase the on-chip hardware resources rather than use more pins on the chip package. This is due to the fact that chip packaging technology has not enjoyed the exponential growth that semiconductor processing and circuit densities have.

Allowing  $f$  to denote the maximum number of operations that can be fetched in each instruction, the hardware resources of a given processor configuration  $C$  can be specified as a 4-tuple:

$$C = (c, l, a, f)$$

For example, a processor organization capable of fetching (and executing) four operations in each cycle is shown in Figure 5. The processor has four execution units: one CTU (Control Transfer Unit), one LSU (Load/Store Unit), and two ALU's (Arithmetic/Logic Unit). This corresponds to  $C = (1, 1, 2, 4)$ . In each cycle, the processor can execute one control transfer operation, one load/store operation, and two arithmetic/logic operations, all in parallel. The CTU's interact with the Program Counter (PC), and the LSU's are connected to the data cache subsystem. A single CTU will allow the execution of regular two-way branch operations, i.e., branches with a single condition. The instruction format for this configuration is shown in Figure 6. For each hardware execution unit, there is an operation field in the instruction. This will allow the processor to fetch as many operations as the hardware resources of the processor can handle in each cycle. This configuration is similar to those of LIFE [16] and the Multiflow TRACE [13].

One disadvantage of the organization shown in Figure 5 and the associated instruction format is that instructions that do not have control transfer or load/store operations will result in empty slots in the long instruction word. This effectively results in wasted instruction fetch bandwidth.

To achieve a higher level of performance, we could add more functional units and fetch more operations in each long instruction word. For example, the organization shown in Figure 7 can fetch and execute eight operations in each cycle. For this configuration  $C = (2, 2, 4, 8)$ . Two CTU's will allow the execution of three-way branch operations, i.e., branches with two conditions (see Section 3.4). The performance improvement is due to the increase in the fetch bandwidth and the existence of additional execution units. However, this organization still suffers from the problem that the first one did. To keep the machine completely busy, each instruction must have two CT operations, two LS operations, and four AL operations. Instructions that do not have CT or LS operations result in wasted instruction fetch bandwidth. Another problem with this new configuration is that the register file must have twice as many ports as before. This will slow down the register file and will lengthen the processor cycle time.

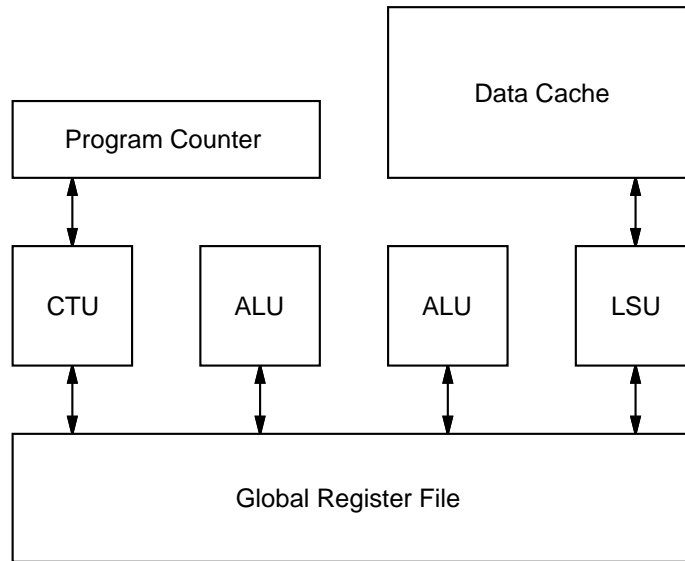


Figure 5: Typical VLIW Processor Organization



CT : Control Transfer operation  
 LS : Load/Store operation  
 AL : Arithmetic/Logic operation

Figure 6: Instruction Format for the Organization in Figure 3.1

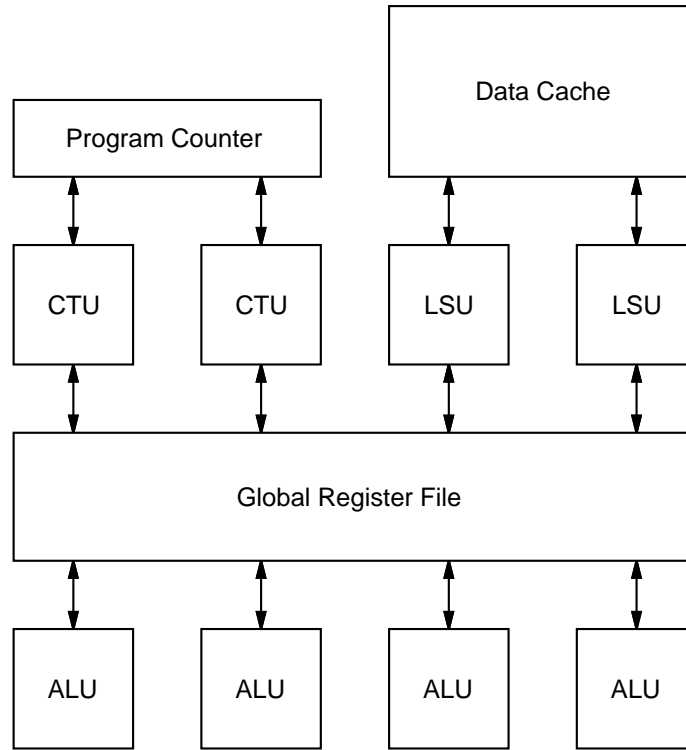
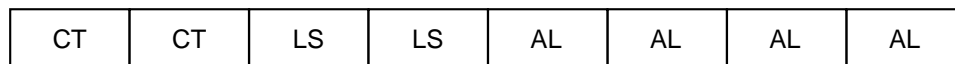


Figure 7: VLIW Processor with Increased Hardware Resources



CT : Control Transfer operation  
 LS : Load/Store operation  
 AL : Arithmetic/Logic operation

Figure 8: Instruction Format for the Organization in Figure 3.3

An alternative approach to increase performance is presented in Figure 9. Different types of execution units are combined into groups. Each group corresponds to an operation field in a long instruction word. We shall call a group of execution units a *functional unit*. The advantage of this strategy is that each operation field in an instruction is not restricted to a specific type. A long instruction word can have various combinations of CT, LS, and AL operations. This will result in better utilization of the instruction fetch bandwidth because to keep the machine busy we are not required to have a specific combination of operations in each instruction. In terms of the resource parameters of the processor, this corresponds to:

$$f < c + l + a$$

This strategy can improve the performance of the processor because it allows the processor to execute instructions that have, for example, four AL operations whereas the configuration shown in Figure 5 will require that the instruction be broken into two instructions during the Resource Constrained Scheduling phase of compilation. This will increase the path length of the program and result in more execution cycles. An important aspect of this organizational strategy is that performance gain is achieved without increasing the required instruction fetch bandwidth or the number of register file ports. Since each register file port is now connected to more than one execution unit, there is a greater load on each port; however, this is a problem of large *fan-out* and can be effectively solved by properly buffering the output ports of the register file. The delay of large fan-out circuits can be made to increase only logarithmically as the load capacitance increases [22]. On the other hand, adding extra ports to the register file (which is the case with the organization in Figure 7) presents a large *fan-in* problem and cannot be solved as easily as a large fan-out problem. Solving large fan-in problems involves trading noise margin for speed (which can only be taken to a certain extent) and requires circuit design techniques with reduced voltage swings on bus lines and sense amplifiers. These solutions increase the complexity of the design and only mitigate the delay penalties imposed by the extra ports to the register file.

In order to quantify the performance/efficiency characteristics of these organizational strategies, a series of simulations were performed. The objective was to determine which approach would result in a higher level of performance with better utilization of hardware resources and the available instruction fetch bandwidth. The results of these simulations are outlined in Table 2. The efficiency of a given processor configuration in utilizing the instruction fetch bandwidth that it requires with respect to the speed-up ( $S$ ) that it offers can be quantified by the following factor:

$$E = \frac{S}{f}$$

This factor can be found in the last column of Table 2. The data in this table shows that it is indeed better to combine execution units into functional units. A higher level of performance with better utilization of the instruction fetch bandwidth can be achieved using this approach. While the configuration with  $C = (2, 2, 4, 8)$  achieves a somewhat higher performance than the configuration with  $C = (2, 2, 4, 4)$ , it achieves this performance at the expense of a register file with twice as many ports and an instruction bus twice as wide.

An important consideration in determining the configuration of a processor is the degree of utilization of hardware resources. The law of diminishing returns is at work here:

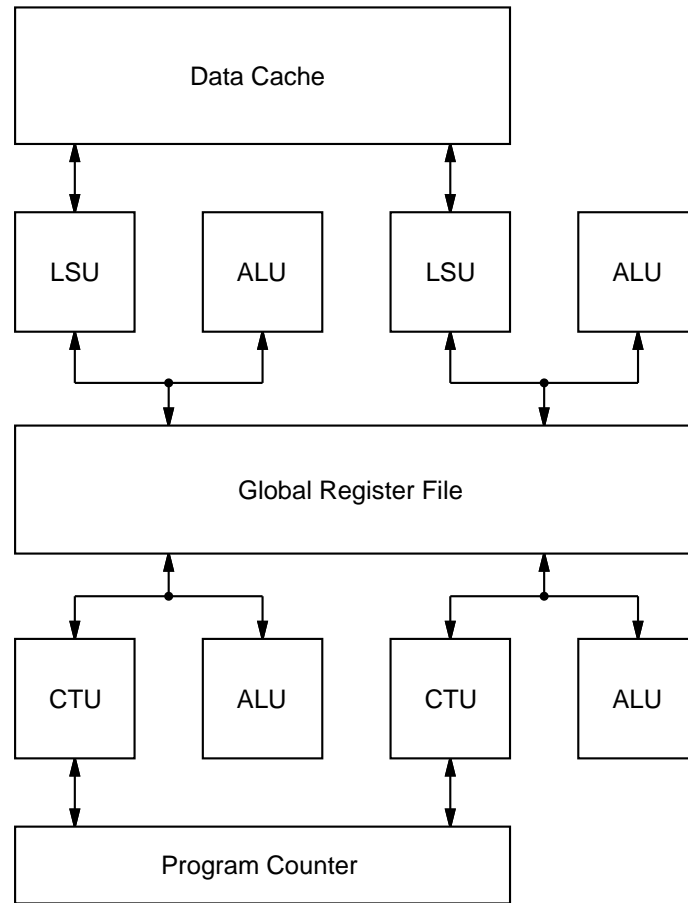


Figure 9: Alternative Processor Organization with a Greater Degree of Efficiency

$c$	$l$	$a$	$f$	$S$	$E$
1	1	2	4	1.74	0.435
2	2	4	4	2.29	0.573
2	2	4	8	2.66	0.333

Table 2: Performance of Various Organizational Strategies

each additional hardware resource will result in less performance improvement than the previous one did. The use of additional hardware resources should be justified based on the performance improvement that can be achieved by that extra hardware resource. Beyond a certain point, extra hardware resources will not contribute to performance and will be wasted.

Another important goal is to keep the machine organization as “clean” as possible in terms of its resource limitations. This approach will simplify code generation. A machine with many restrictions and idiosyncrasies is difficult to generate good code for. Ideally, each functional unit should be able to execute all types of operations supported in the instruction set. This would considerably simplify code generation because the compiler would not have to worry about assignment of operations to functional units. However, this may not be very practical from an implementation point of view. Enabling all functional units to execute all types of operations requires additional hardware resources (silicon area, power dissipation, etc.) and could also lengthen the cycle time of the processor. The approach taken in the design of VIPER was to allow all functional units to be able to execute AL operations. This corresponds to  $a = f$ . The reason for this decision is that AL operations are the most common (AL operations also include register move and comparison operations), and we would like the code generator to be able to assign at least AL operations to any of the functional units.

The configuration of the processor was determined based on a series of simulations that evaluated the performance of various possibilities in the design space. The objective was to observe the performance improvement that can be achieved by gradually adding hardware resources to the processor. The results of these simulations are outlined in Table 3 and Figure 10. The speed-up values that are reported in the table are the harmonic mean of the speed-up factors for all benchmarks. For these simulations, it was assumed that all control transfer operations have a delay of one cycle (all control transfer operations of VIPER have a delay of one cycle).

Three sets of results are presented: one for a machine with  $a = f = 4$ , one for a machine with  $a = f = 6$ , and one for a machine with  $a = f = 8$ . This corresponds to three distinct curves in Figure 10. For each data set, the starting point is  $c = l = 1$ . The  $c$  and  $l$  parameters are gradually increased. Initially, performance keeps increasing as more hardware execution units are added to the processor. Beyond configuration 4 ( $c = 2, l = 2$ ) there is very little improvement in performance. Hardware resources added beyond this point do not contribute to performance, but they do contribute to the complexity and the cost of the processor. Based on these results, it was decided that each VIPER instruction could have a maximum of two control transfer operations and a maximum of two load/store operations ( $c = 2, f = 2$ ). Notice that for  $a = f = 6$  performance actually drops slightly when  $c = 3$ . This is due to the way Resource Constrained Scheduling works. During RCS, it is assumed that branch operations have the highest priority; thus, if all branch operations in a given instruction node can be executed by the processor, no branch operations will be deferred. Going from  $c = 2$  to  $c = 3$  will allow instructions with three CT operations; however, since  $f$  is still equal to 6, the scheduler has to defer an AL or LS operation. In some cases, this results in a program with a longer path length and more execution cycles.

To determine the number of ALU’s in the processor, another set of simulations were performed. These simulations assumed that the processor can execute a maximum of two load/store and two branch operations in each cycle ( $c = 2, l = 2$ ). The number of AL operations were varied to measure the performance of different configurations. The results

<i>Configuration</i>	<i>c</i>	<i>l</i>	<i>a</i>	<i>f</i>	<i>S</i>	<i>% improvement</i>
1	1	1	4	4	2.06	N/A
2	1	2	4	4	2.13	3.4
3	2	1	4	4	2.22	4.2
4	2	2	4	4	2.29	3.2
5	2	3	4	4	2.29	0.0
6	3	2	4	4	2.29	0.0
7	3	3	4	4	2.29	0.0
<hr/>						
1	1	1	6	6	2.28	N/A
2	1	2	6	6	2.42	6.1
3	2	1	6	6	2.61	7.9
4	2	2	6	6	2.81	7.7
5	2	3	6	6	2.83	0.7
6	3	2	6	6	2.80	-1.1
7	3	3	6	6	2.82	0.7
<hr/>						
1	1	1	8	8	2.30	N/A
2	1	2	8	8	2.45	6.5
3	2	1	8	8	2.67	9.0
4	2	2	8	8	2.89	8.2
5	2	3	8	8	2.91	0.7
6	3	2	8	8	2.94	1.0
7	3	3	8	8	2.96	0.7

Table 3: Performance of Various Processor Configurations

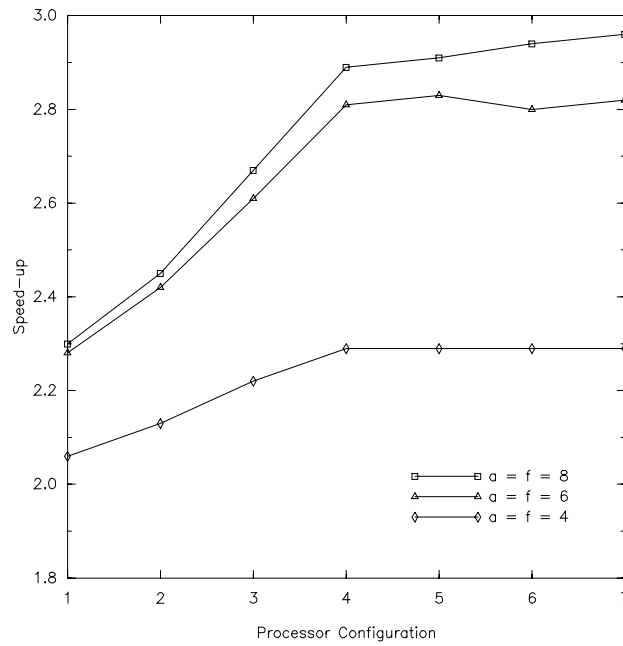


Figure 10: Performance of Various Processor Configurations

$a = f$	$S$	% improvement
4	2.29	N/A
5	2.63	14.8
6	2.81	6.8
7	2.84	1.1
8	2.89	1.8
9	2.89	0.0

Table 4: Performance Effect of Adding ALU's

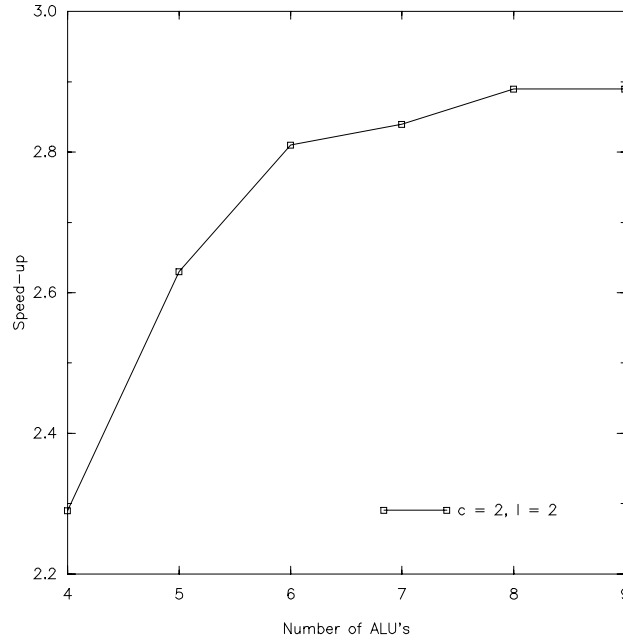


Figure 11: Performance Effect of Adding ALU's

are shown in Table 4 and Figure 11.

Increasing  $a$  from 4 to 5 and then to 6 results in improved performance. For  $a > 6$ , however, only slight performance improvements can be expected. The final decision was to allow  $a = f = 4$ . This was motivated by the desire to have a reasonable number of pins on the chip package. Assuming 32 bits per operation, a configuration with  $a = f = 4$  requires an instruction bus that is 128 bits wide and will take 128 pins on the chip package (assuming that no multiplexing is done because multiplexing will impose a major limitation on the instruction fetch rate).

Thus, the hardware resources of VIPER can be summarized as follows:

1. Each instruction contains four operations.
2. Each instruction can have a maximum of two control transfer operations.
3. Each instruction can have a maximum of two load/store operations.

4. Each instruction can have a maximum of four arithmetic/logic operations.

### 3.2 General Organization of VIPER

Figure 12 shows the block diagram of VIPER. The processor is pipelined and contains four integer functional units that are connected through a shared multiport register file that has thirty two 32-bit registers [23]. In each cycle, the register file can perform eight read and four write transactions (two reads and one write per functional unit). Register R0 is hardwired to contain zero at all times. Writing to R0 is allowed but has no effect on its content. The initial design goal was to have 64 registers, but it was finally decided to have 32 registers. One reason for this decision was that the silicon area required for 64 registers proved to be quite large after preliminary layout efforts. Another reason was that the extra number of bits required to address 64 registers presented a severe problem for the goal of having a 32-bit format for each operation. Also, more registers could result in a slower register file. Having 32 registers has not presented a performance penalty for the current benchmarks so far; however, for larger benchmarks, a larger number of registers might be a better choice.

VIPER has some of the typical attributes of a RISC processor. It has a simple operation set that is designed with efficient pipelining and decoding in mind. All operations follow the register-to-register execution model. Data memory is accessed with explicit load/store operations. All instructions have a fixed size, and there are only a few instruction formats. Arithmetic, logic, and shift operations are executed by all functional units. Load/Store operations are executed by FU2 and FU3, which have access to the data cache subsystem through their LSU's. Load/Store operations use the register indirect addressing mode instead of the more typical displacement addressing mode. This will be explained in Section 3.3. Control Transfer operations are executed by FU0 and FU1, which interact with the Program Counter through their CTU's. VIPER has hardware support for the execution of three-way branch operations. Also, to increase execution throughput, operations following a branch are executed conditionally: they are all issued and executed, but they are allowed to complete and write to the register file depending on the outcome of the branch. This will be explained in Section 3.4.

Figure 12 is drawn with the floorplan of the processor in mind. The organization of the processor was developed to facilitate an efficient VLSI layout. A key aspect was an emphasis on locality of communication between different hardware blocks of the processor. The floorplan of the processor is shown Figure 13. The relative sizes of various hardware blocks were estimated from preliminary layout efforts. As of this writing, a generic functional unit has actually been designed and fabricated in the form of a stand-alone RISC microprocessor [24]. The data path of the processor comprises a big part of the floorplan. The floorplan of the chip is centered around the register file. Shaded areas in Figure 13 highlight the  $i$ -th bit-slice of different data path blocks. A register file bit-slice is twice as wide as a functional unit bit-slice. At the top and bottom of the register file are routing areas that are used to connect the register file bit-slices to the functional unit bit-slices. Notice that in this figure, the CTU's of FU0 and FU1 and the PC unit are all combined into a single hardware block and are collectively called the Control Transfer Unit.

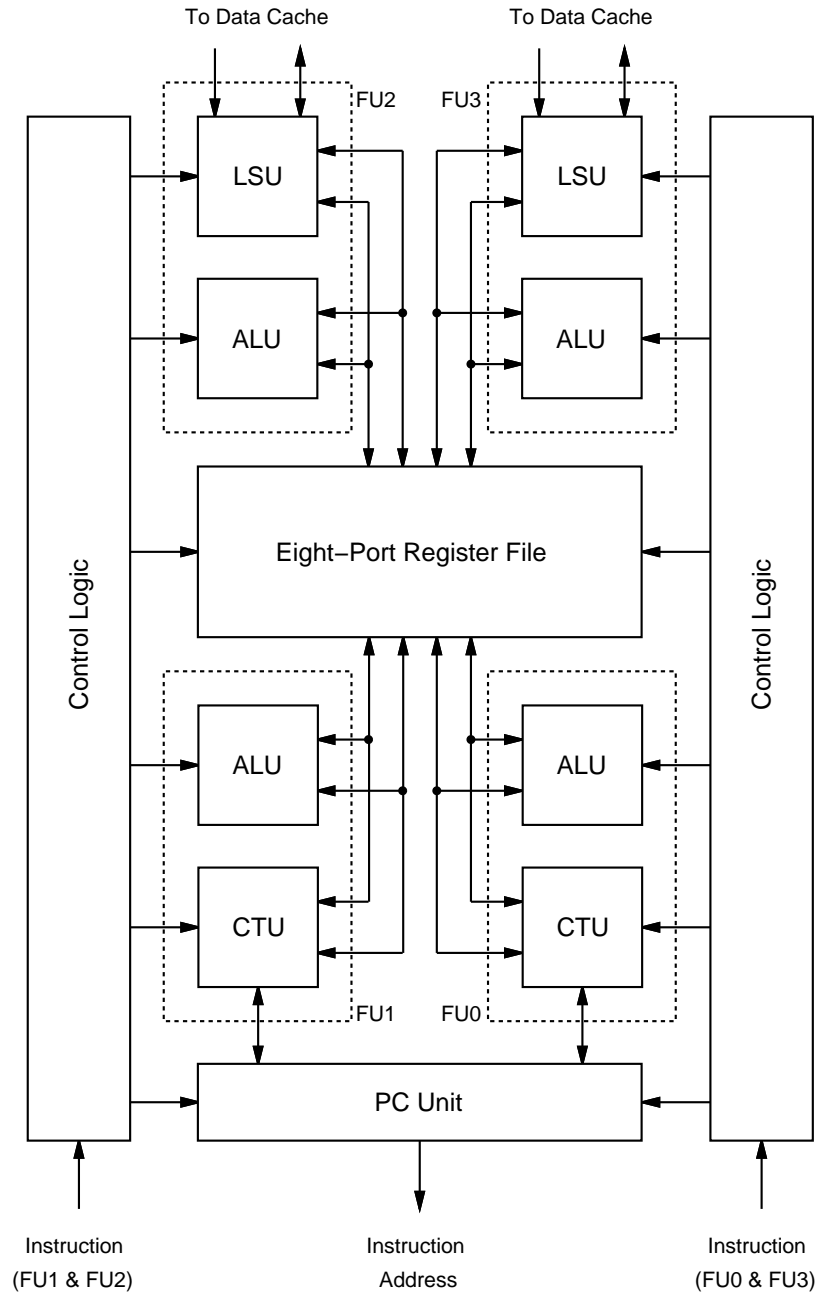


Figure 12: Block Diagram of VIPER

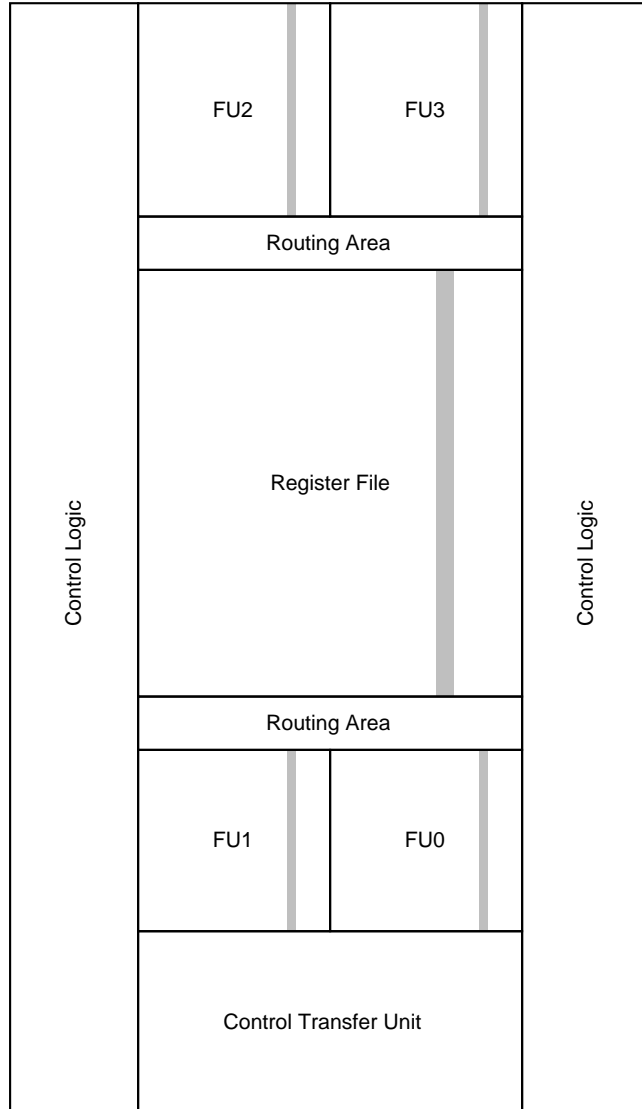


Figure 13: Floorplan of VIPER

### 3.3 Pipeline Structure and Bypassing

Figure 15 shows the pipeline structure typically used in RISC processors. As explained in Section 1.1, RISC processors use bypassing in order to resolve possible data hazards caused by the delay between the ID and WB stages. The cost of bypassing in a RISC processor is minor compared to the number of cycles that it saves. It requires  $2d$  comparators (where  $d$  is the number of pipeline stages between ID and WB) and the necessary pathways from the pipeline registers to the inputs of the execution unit corresponding to the EX stage. If the bypassing circuitry is carefully designed, its cycle time overhead can be relatively small in a RISC processor. However, in a VLIW processor with  $n$  functional units, bypassing becomes a costly function to perform. There are two factors contributing to this cost. One is the number of comparators that are required. In a VLIW processor with  $n$  functional units, the number of required comparators is  $2dn^2$  ( $d$  is the number of pipeline stages between ID and WB). The bypassing comparators are usually not in the critical path of the execution cycle because comparison of register addresses can start right at the beginning of the ID stage and progress in parallel with the register file write and read operations, which are slower than the comparison required for bypassing. The comparators present an area penalty, but given the circuit densities available in current 1.0 micron technologies, they might not present a severe constraint. On the other hand, the buses required to bypass operands not only present global layout problems, but they are also likely to be on the critical path for two reasons:

1. Bypassing of operands cannot start until these operands are available. This means that the bypassing hardware has to wait until the EX and ID stages have generated their results. Then, given the outcome of the bypassing comparators, the output of the EX stage can be bypassed back to its input.
2. In a processor with a single functional unit (e.g., a RISC processor), the bypassing pathways are within the data path of the processor and are distributed among the bit-slices of the data path; in other words, they are local to each bit-slice and do not present a major capacitive load (see Figure 14). However, in a processor with multiple function units (e.g., a VLIW processor), the bypassing pathways are global buses connecting multiple data paths and present a heavy capacitive load.

The frequency of RAW hazards can be reduced by decreasing the delay between the ID and WB stages. If instead of using the displacement addressing mode for load/store operations, the register indirect addressing mode is used, then memory access can take place during the EX stage because the effective memory address is available by the end of the ID stage. Thus, the MEM stage can be removed, and the delay between the ID and WB stages is reduced to a single cycle. The resulting pipeline is shown in Figure 16. This scheme has the following advantages:

1. The frequency of RAW hazards will decrease. This can improve the performance of the processor.
2. The number of comparators is reduced by 50 percent. This will substantially reduce the area taken by the address comparison circuitry. Even if the processor does not use complete global bypassing but detects RAW hazards and resolves them by stalling the execution pipeline, the savings in the number of required comparators is significant.

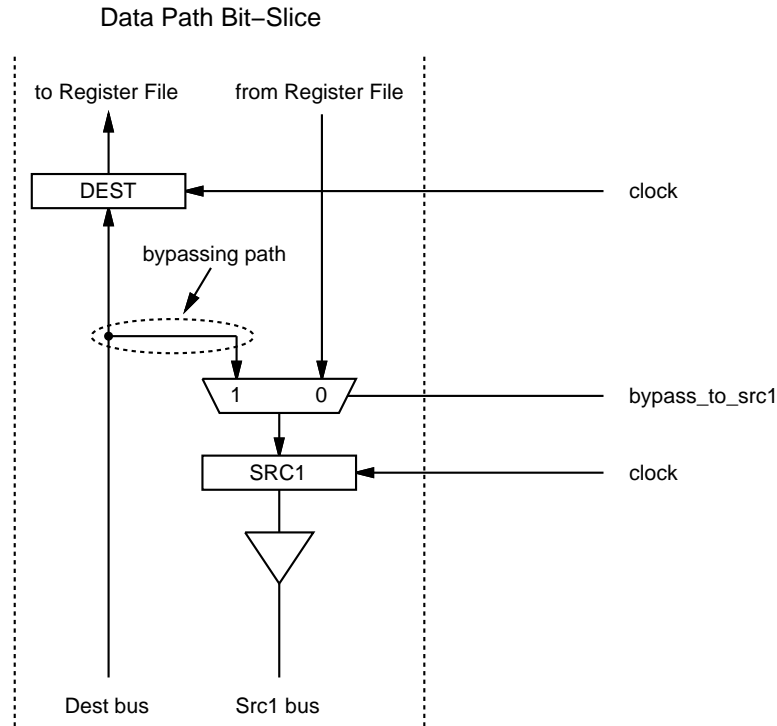


Figure 14: The Bypassing Path for a Single Functional Unit

3. The number of bypassing buses that are needed to connect different functional units decreases by half. This will somewhat relax the layout difficulties imposed by a complete global bypassing network. It can also make inter-functional unit bypassing more feasible.

Changing the addressing mode from displacement to the less powerful register indirect mode can have a negative effect on performance. With the register indirect addressing mode, the compiler will have to schedule an extra add operation before load/store operations in order to compute memory addresses. This can increase the path length of the program and reduce performance. There are, however, several mitigating factors:

1. Not all load/store operations need an add operation. In Reference [12], the percentage of load/store operations with zero displacement for two standard integer benchmarks, GCC and TeX, for a RISC style machine is reported to be 27% and 17%, respectively. In Reference [25], Gross et al. report the average percentage of load/store operations with zero displacement to be around 29% for a variety of small and large integer benchmarks in C and Pascal.
2. The extra add operation could be “absorbed” into an empty operation slot in a long word instruction during the compaction process without increasing the path length of the program.
3. With the register indirect addressing mode, the new pipeline structure does not suffer from load delays. An operation using the result of a load operation in the previous instruction can execute without delay (assuming that there is a bypass path from the

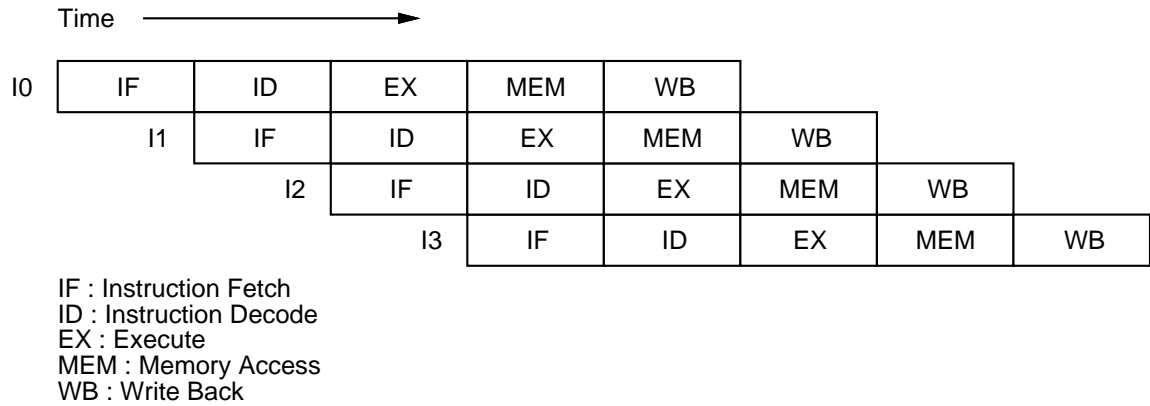


Figure 15: Pipeline Structure of Typical RISC Processors

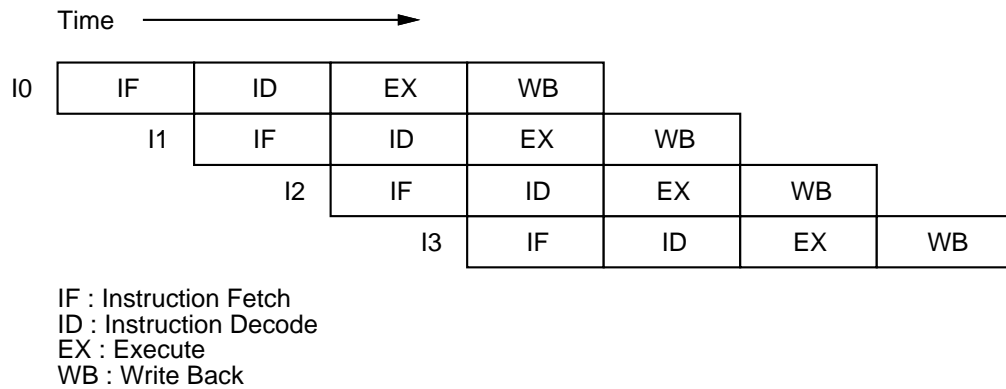


Figure 16: Pipeline Structure of VIPER

functional unit executing the load operation to the functional unit that uses the result of the load).

In order to find out which pipeline structure would result in higher performance, a series of simulations were performed. In these simulations, the objective was to compare the speed-up achieved by two different versions of VIPER, one with a 5-stage pipeline using the displacement addressing mode, and one with a 4-stage pipeline using the register indirect addressing mode. The speed-up factors for these two processors were computed by comparing them to a processor with a single functional unit using the 5-stage pipeline. The reason is that if we were to design a pipelined processor with a single functional unit, we would use the 5-stage pipeline because it would result in better performance. In these simulations, it was assumed that bypass paths existed from each functional unit to itself only, i.e., no global bypassing among different functional units. This means that a data hazard will result in a stall only if a functional unit uses the result produced by a different functional unit in the previous cycle. The results of these simulations are presented in Table 5.

The results in Table 5 show that a 4-stage pipeline can result in higher performance for a VLIW processor. On the average, the speed-up achieved by the processor with the 4-stage pipeline is 8.4 percent greater than that of the processor with the 5-stage pipeline. The percentage of pipeline stalls caused by RAW hazards is much lower for the 4-stage pipeline. This accounts for the performance advantage of the 4-stage pipeline even though it incurs an extra addition for a large fraction of load/store operations.

Given the performance advantage of the 4-stage pipeline and the fact that the hardware implementation of the 4-stage pipeline with register indirect addressing is simpler than that of the 5-stage pipeline with displacement addressing, it was decided that VIPER would use the 4-stage pipeline. RAW data hazards that cannot be resolved by the available bypassing network are detected at run-time by bypassing comparators and are resolved by stalling the execution pipeline by one cycle. It is possible to allow the code generator to eliminate RAW hazards by scheduling instructions with NOP operations at appropriate points in the program; however, this approach has two drawbacks:

1. Stalling the processor by scheduling instructions with NOP operations will increase the code size and will effectively waste some of the instruction fetch bandwidth.
2. After a branch delay slot, depending on the outcome of the branch, it might or might not be necessary to stall the pipeline. The code generator has to assume the worst case and schedule a stall cycle. If a branch operation takes the path that does not really require a stall cycle, a cycle is lost.

The frequency of pipeline stalls caused by RAW hazards can be further reduced by software scheduling. This is done in a final pass by the code generator during which it attempts to modify the assignment of operations to functional units so that RAW hazards can be resolved by the bypassing hardware of the processor instead of resulting in pipeline stalls. This will be explained in Section 4.5.

### 3.3.1 Bypassing Interconnection Network

One can think of the bypassing hardware as an interconnection network that connects different functional units together. Increasing the connectivity of the bypassing network results

<i>benchmark</i>	<i>5-stage pipeline</i>		<i>4-stage pipeline</i>	
	<i>speed-up</i>	<i>% stalls</i>	<i>speed-up</i>	<i>% stalls</i>
binsearch	1.77	31.2	1.75	27.8
bubble	1.38	17.6	1.47	12.1
chain	1.62	27.9	1.83	18.4
factorial	1.69	28.9	1.92	18.6
fibonacci	1.81	33.9	2.03	7.6
floyd	1.86	14.9	1.96	10.6
matrix	1.67	28.4	1.88	19.2
merge	1.52	27.4	1.68	17.2
quicksort	1.55	27.1	1.55	23.5
dijkstra	1.73	26.8	1.90	19.3
Average	1.66	26.4	1.80	17.4

Table 5: Comparison of the Performance of the 5-stage and 4-stage Pipelines

in two conflicting effects on the performance of the machine. A higher degree of connectivity can result in a smaller number of pipeline stalls and a higher level of performance. On the other hand, increasing the connectivity of the bypassing network will increase the capacitive load of the bypassing pathways and will lengthen the processor cycle time. In this section, the performance effects of various bypassing interconnection network topologies are analyzed. The objective is to explore the conflicting performance effects of increasing the connectivity of the bypassing interconnection network.

We can describe a given network topology by a  $4 \times 4$  matrix  $P$  that is defined as follows:

$$P_{ij} = \begin{cases} 1 & \text{if there is a path from functional unit } i \text{ to functional unit } j \\ 0 & \text{otherwise} \end{cases}$$

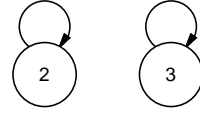
The  $P$  matrix is a simulation parameter that is used to determine whether the execution pipeline needs to be stalled in case of a data hazard.

Figure 17 shows various interconnection topologies along with their corresponding  $P$  matrices. The bypassing network for  $P = P_1$  presents a minimal degree of connectivity where destination operands are bypassed from each functional unit to itself only. The area taken by the bypassing pathways for this topology is virtually zero. They are embedded within the data paths of the functional units and are local to each bit-slice (see Figure 14). Their capacitive load is insignificant compared to the capacitive load of the destination bus that they are connected to. The cycle time penalty of this scheme is minimal; however, it can result in more pipeline stalls than the other bypassing networks.

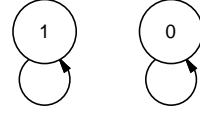
For  $P = P_2$  and  $P = P_3$ , each functional unit has access to two bypassed destination operands, one from itself, and another from a neighboring functional unit. The network topology for  $P = P_2$  includes extra buses that connect FU0 to FU1 (and vice versa) and FU2 to FU3 (and vice versa). The cycle time penalty for these extra connections is relatively modest because of the physical proximity of FU0 to FU1 and FU2 to FU3. However, since these extra connections require horizontal wires (with respect to the floorplan in Figure 13) the routing areas between the register file and the functional units have to be stretched in the vertical direction to accommodate the following horizontal buses:

1. A 32-bit bus from FU0 to FU1
2. A 32-bit bus from FU1 to FU0

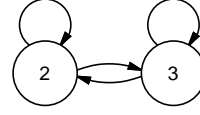
$$P = P_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



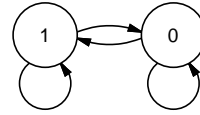
$$P = P_2 = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$



$$P = P_3 = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$



$$P = P_4 = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$



$$P = P_5 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

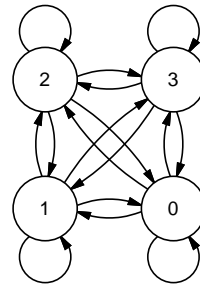
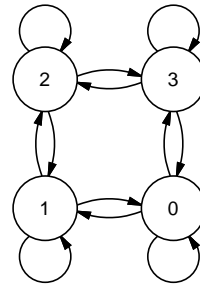
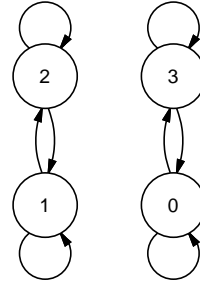


Figure 17: Various Bypassing Interconnection Network Topologies

3. A 32-bit bus from FU2 to FU3
4. A 32-bit bus from FU3 to FU2

For  $P = P_3$ , the additional bypassing connections connect FU0 to FU3 (and vice versa) and FU1 to FU2 (and vice versa). The buses required for this scheme are longer than ones required for  $P = P_2$  because they have to cross the entire height of the register file; however, since the functional unit bit-slices that are being connected by these vertical connections are at the same horizontal coordinate, the required routing tracks can be placed within the bit-slices of the register file without the need for extra routing space.

The bypassing network for  $P = P_4$  is a combination of the previous two. It offers an even higher degree of connectivity at the expense of a larger cycle time penalty. The routing area taken by this network is equal to the routing area required for  $P = P_2$ .

For  $P = P_5$ , all functional units are completely interconnected. Each functional unit has accesses to all destination operands from the previous machine cycle; however, the cycle time penalty for this configuration is the largest. The additional bypassing connections are used to connect FU0 to FU2 (and vice versa) and FU1 to FU3 (and vice versa). These connections present additional layout difficulties because they require that the routing areas between the register file and the functional units be stretched further to accommodate four more 32-bit horizontal buses.

To analyze the performance effects of these different bypassing networks, two sets of simulations were performed. In the first set of simulations, the objective was to observe the speed-up factors that can be achieved by different network topologies. These speed-up values are ideal ( $S_{ideal}$  in Table 6) in the sense that they do not include the cycle time penalty of a given network topology. They provide a measure of the number of stalls saved by a given bypassing network. For this set of simulations, the code generator was allowed to schedule operations so as to reduce the frequency of pipeline stalls caused by RAW data hazards. In the second set of simulations, the extracted layout of the bypassing path was simulated using SPICE [26]. For these simulations, the process parameters of the 1.2-micron CMOS technology offered by MOSIS were used [27]. The results of these circuit simulations provide a measure of the cycle time penalty of a given bypassing network. These results are presented as normalized cycle times ( $\tau_{normalized}$ ) in Table 6. The actual speed-up that can be achieved with a given bypassing interconnection network is computed by:

$$S_{actual} = \frac{S_{ideal}}{\tau_{normalized}}$$

The circuit simulations analyzed the path shown in Figure 18 which is the longest chain of dependencies during the ID stage. First, the output of the ALU or the shifter is driven onto the *Dest* bus. Then it goes through the bypassing network and the bypassing multiplexor, and then, it is driven onto the *Src1* bus, which goes into the branch detection logic and is used to determine the outcome of branch operations. In VIPER, the outcome of branch operations is decided by testing the least significant bit of a register operand (see Section 3.4.1). Explicit comparison operations are used to set or reset the least significant bit of a register. The outcome of a branch operation is known by the end of the ID pipeline stage. The address of the target of the branch is also computed during the ID stage. The capacitive load of the bypassing buses were estimated from preliminary layout efforts. The simulation results are also plotted in Figure 19.

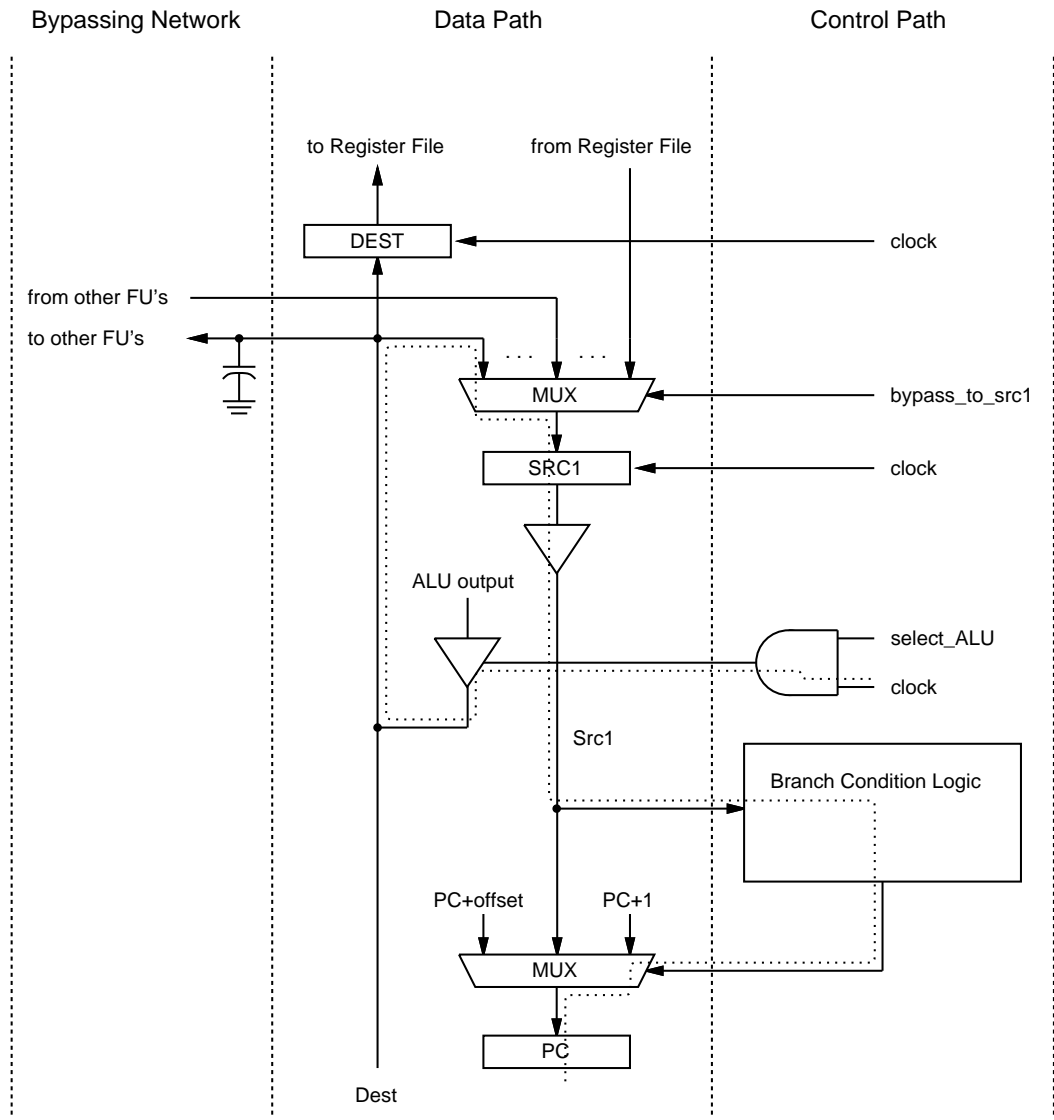


Figure 18: Critical Path Involving the Bypassing Hardware

$P$	$S_{ideal}$	$\tau_{normalized}$	$S_{actual}$
$P_1$	1.93	1.000	1.93
$P_2$	2.02	1.022	1.98
$P_3$	1.96	1.044	1.88
$P_4$	2.09	1.067	1.96
$P_5$	2.15	1.084	1.98

Table 6: Performance Effects of Various Bypassing Interconnection Networks

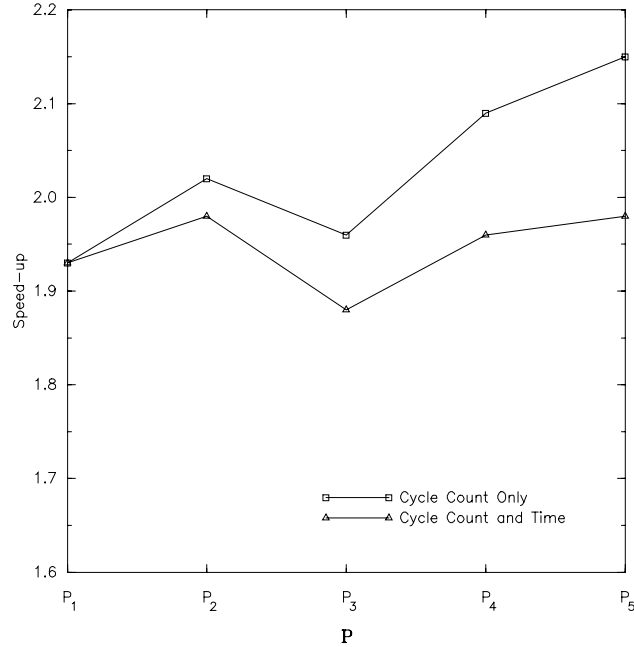


Figure 19: Performance Effects of Various Bypassing Interconnection Networks

These results show that the best performance can be achieved with  $P = P_2$ . Its performance advantage over the bypassing network for  $P = P_1$  is about 2.6 percent. Since this is not a very significant performance advantage, the simplicity of the bypassing network for  $P = P_1$  makes it a viable option even though its performance is less than optimal.

### 3.4 Branching and Conditional Execution

As mentioned earlier, one of the architectural goals of VIPER was to provide support for multiway branching and conditional execution of operations for increased throughput. This support is based on the execution model assumed by the PS compiler that was described in Section 2.1. Multiway branching and conditional execution involve instruction nodes with multiple targets; we shall call these nodes *branching nodes*. VIPER supports branching nodes with a maximum of three successors.

Since all control transfer operations have a delay of one cycle, branching nodes are allowed to have a maximum of *eight* operations; half of these operations are scheduled in the

branch delay slot. A branching node is thus allowed to have a maximum of four load/store operations and a maximum of six or seven arithmetic/logic operations depending on whether there is one or two branch operations in the node. A branching node is translated into a sequence of two instructions. The first instruction includes the branch operation(s). All operations in this instruction are said to be in the *branch slot*. Operations in the second instruction are said to be in the *branch delay slot*. Figure 20 shows a branching node with three targets. The machine instruction for this node is shown in Figure 21. Branch operations are assigned to FU0 and FU1, which are connected to the Control Transfer Unit and can execute control transfer operations. Load/store operations should be assigned to FU2 and FU3, which are connected to Load/Store Units and can execute load/store operations. Non-branch operations in the branch and branch delay slots are allowed to complete depending on the outcome of the branch. This will be explained in Section 3.4.2.

An important issue is the assignment of the operations of a branching node to branch and branch delay slots. In the execution model assumed by the PS compiler, all operations in an instruction node use values computed in a predecessor node. If some operations are to be scheduled in the branch delay slot, care must be taken to ensure that they do not use values computed by operations in the branch slot.

The first approach to solving this problem was to let the code generator schedule operations so that no operations in the branch delay slot use values computed by operations in the branch slot. This approach is quite complicated because it requires that the code generator evaluate data dependencies among the operations of a branching node. Also, there are situations where this type of scheduling fails because of resource limitations even though no resource constraints are violated in the instruction node itself. This can be demonstrated by an example. Figure 22 shows a group of operations from an instruction node with two branch operations. Notice that there are no resource constraint violations: the total number of operations is not more than eight (two branch operation plus the six shown in the figure), and the number of load/store operations is not more than four. Two of these operations should be scheduled in the branch slot and the other four in the branch delay slot. Arrows between operations indicate dependencies. For example, the arrow from *op1* to *op2* means that *op2* writes to a variable that *op1* reads; consequently, if we were to execute one of the two operations in a later cycle, it would have to be *op2*; otherwise, the semantics of the original program would not be preserved. This type of dependency is known as an *anti-dependency*. The group of operations in Figure 22 cannot be scheduled properly. One of the operations assigned to the branch slot (besides the two branch operations) has to be *op1*. The other operation that can be assigned to the branch slot is either *op2* or *op6*. Either way, the outcome is three load/store operations in the branch delay slot (a resource constraint violation).

The final solution was to disable bypassing for all operations in a branch delay slot. In Figure 23, when *op3* tries to read its source operands during the ID stage, register *a* still contains the value defined by *op1*. The value computed by *op2* is still in the pipeline. Since bypassing is disabled for the operations in the branch delay slot, the result computed by *op2* does not reach *op3*. Thus, *op3* uses the value of *a* computed by *op1*. This is in agreement with the original semantics of the program. This approach is more consistent with the execution model assumed by the compiler. The hardware needed to disable bypassing to operations in the branch delay slot is quite simple: it merely has to detect a branch operation during the ID stage of the instruction in the branch slot and disable bypassing of operands to the operations in the branch delay slot during the next cycle (the ID stage

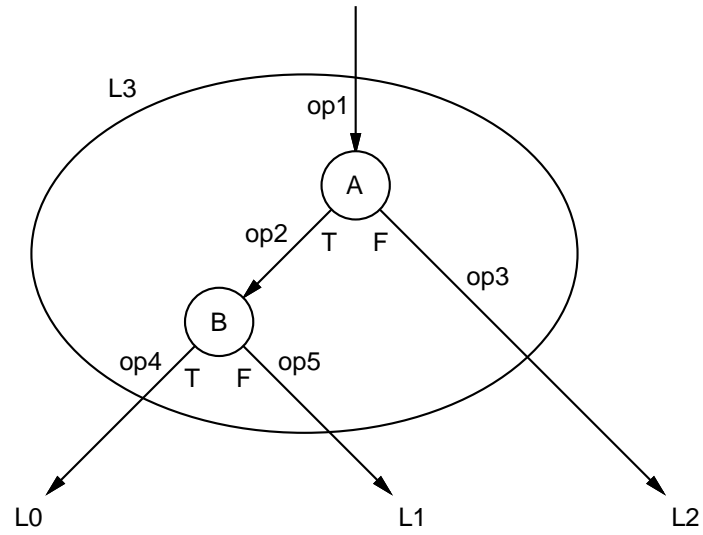


Figure 20: An Instruction Node with Three-Way Branch Operation

FU0	FU1	FU2	FU3	
branch	branch	op2	op5	branch slot
op1	op3	NOP	op4	branch delay slot

Figure 21: Processor Instructions Corresponding to a Branching Node

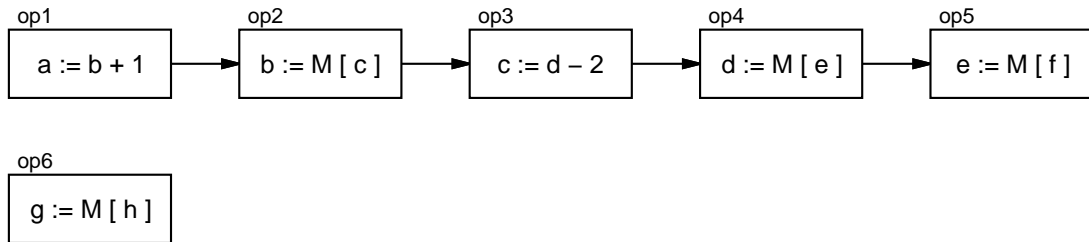


Figure 22: Example of a Group of Non-Branch Operations in an Branching Node

FU0	FU1	FU2	FU3
...	...	...	op1: a := b + c
branch	branch	...	op2: a := a + 1
...	...	...	op3: d := M [ a ]

Figure 23: Effect of Disabling Bypassing for Operations in Branch Delay Slots

of the instruction in the branch delay slot). This function is not in the critical path of an execution cycle. It requires very little hardware but greatly simplifies code generation.

### 3.4.1 Multi-Way Branching

The execution of the instruction node shown in Figure 20 will require a three-way branch operation. The unique target instruction depends on *two* condition variables,  $A$  and  $B$ . We shall call  $A$  the *parent* condition and  $B$  the *child* condition because  $A$  is the parent of  $B$  in the control tree of the node. In general, an instruction node with  $n$  targets will require an  $n$ -way branch operation, and the selection of a target instruction will depend on  $n - 1$  condition variables. Based on the results presented in Section 3.1, it was decided that VIPER would support the execution of three-way branch operations. The branching mechanism should be able to handle regular two-way branching as well as three-way branching.

Three-way branching is implemented by branch operations that have two conditions. Branch operations have the following form:

$$BRc_1c_2 \quad cc_1, cc_2, offset$$

$c_1$  and  $c_2$  represent conditions and are either  $T$  (*true*) or  $F$  (*false*).  $cc_1$  and  $cc_2$  are general purpose registers, and their values are treated as condition codes. Their values are set by conditional comparison operations. If the least significant bit of a register is 1, then it is considered true; otherwise, it is considered false.  $c_1$  corresponds to  $cc_1$ , and  $c_2$  corresponds to  $cc_2$ . A branch operation instructs the processor to load the Program Counter with  $PC + offset$  only if  $cc_1$  is  $c_1$  and  $cc_2$  is  $c_2$ . We require that  $cc_1$  be the parent condition and  $cc_2$  the child condition. This requirement facilitates conditional execution and will be explained in Section 3.4.2.

In order to perform three-way branching, a pair of branch operations, encoded according to the condition codes in the control tree of an instruction node, are issued to FU0 and FU1, which are connected to the Control Transfer Unit and can execute three-way branch operations. The instruction node in Figure 20 has three successor nodes, which are labeled  $L_0$ ,  $L_1$ , and  $L_2$ . There are three possible target addresses:

1.  $T_0$  is  $PC + offset_0$ .
2.  $T_1$  is  $PC + offset_1$ .
3.  $T_2$  is  $PC + 1$ .

The code generator will generate a one-to-one mapping from the set of target instructions,  $\{L_0, L_1, L_2\}$ , to the set of target addresses,  $\{T_0, T_1, T_2\}$ . For example, if  $L_0$ ,  $L_1$ , and  $L_2$  are

mapped onto  $T_0$ ,  $T_1$ , and  $T_2$  respectively, the following branch operations will be scheduled for execution:

BRTT	$A, B, offset_0$	executed by FU0
BRTF	$A, B, offset_1$	executed by FU1

These operations instruct the processor to branch to  $T_0$  ( $PC + offset_0$ ) if  $A$  is true and  $B$  is true, branch to  $T_1$  ( $PC + offset_1$ ) if  $A$  is true and  $B$  is false, or to fetch the instruction at  $T_2$  ( $PC + 1$ ), otherwise. Each branch operation is associated with a path through the control tree of the instruction node. If none of these two paths are chosen, then the *fall-thru* path is chosen which leads to  $PC + 1$ . Notice that in this example, because of the mapping from the set of target instructions to the set of target addresses, the paths of both branch operations go through both  $A$  and  $B$ . If  $L_0$ ,  $L_1$ , and  $L_2$  are mapped onto  $T_2$ ,  $T_0$ , and  $T_1$ , respectively, then the path to  $L_2$  will only go through condition  $A$  in the control tree. This situation is handled by using register R0 (which always contains zero) as the second condition register ( $cc_2$ ) for the branch operation associated with the path leading to  $L_2$ . This scheme works because R0 always contains zero and is considered false. The following branch operations are scheduled for execution:

BRTF	$A, B, offset_0$	executed by FU0
BRFF	$A, R0, offset_1$	executed by FU1

We require that in this situation, the branch operation associated to the path going through both condition variables be assigned to FU0. This requirement facilitates conditional execution and will be explained in Section 3.4.2.

Regular two-way branch operations are handled by using register R0 as the second condition register ( $cc_2$ ), and the associated condition ( $c_2$ ) is set to  $F$ . The following operation is issued to FU0:

BRTF	$A, R0, offset_0$	executed by FU0
------	-------------------	-----------------

This operation instructs the processor to branch to  $T_0$  ( $PC + offset_0$ ) if  $A$  is true, or to fetch the instruction at  $T_2$  ( $PC + 1$ ), otherwise. Since R0 is always *false*, The second condition of this branch operation is always satisfied and does not affect the outcome of the branch. We require that for two-way branching, the branch operation be assigned to FU0. This requirement facilitates conditional execution and will be explained in Section 3.4.2.

### 3.4.2 Conditional Execution

Operations in branch and branch delay slots are allowed to complete depending on the outcome of the branch. The general idea is to assign conditional execution tags to operations in branch and branch delay slots depending on which edge of the control tree of the instruction node they reside on. All of these operations are issued and executed, but only the ones with tags corresponding to the outcome of the branch operation are allowed to complete and write to the register file or to the data cache. While executing branch operations, the Control Transfer Unit also computes completion flags based on the outcome of the branch. These flags are used to qualify write transactions to the register file or the data cache.

The tag assigned to an operation is related to the *depth* of the operation in the control tree of the instruction node. We will define the depth of an operation as the number of

<i>No. of Conditions</i>	<i>Tags</i>
Zero	XX
One	FX, TX
Two	FF, FT, TF, TT

Table 7: All Possible Tags for Three-Way Branching

<i>No. of Conditions</i>	<i>Tag</i>	<i>Binary Code</i>
Zero	XX	000
One	FX	010
	TX	011
Two	FF	100
	FT	101
	TF	110
	TT	111

Table 8: Binary Encoding of Tags

conditions the completion of the operation depends upon. Thus, the completion of an operation at depth  $n$  depends on  $n$  conditions. In Figure 20, the depth of  $op1$  is zero because the completion of  $op1$  does not depend on any conditions. In the same way, the depths of  $op2$  and  $op3$  are one, and the depths of  $op4$  and  $op5$  are two. As mentioned earlier, in a node with a three-way branch operation, there are two condition codes:  $cc_1$  and  $cc_2$ . By definition,  $cc_1$  is the parent condition in the control tree, and  $cc_2$  is the child condition. For example, in Figure 20,  $cc_1 = A$ , and  $cc_2 = B$ . A tag assigned to an operation is a string  $t_1t_2$ , where  $t_1, t_2 \in \{X, T, F\}$ . Each symbol of the string corresponds to a condition:  $t_1$  corresponds to  $cc_1$ , and  $t_2$  corresponds to  $cc_2$ . The value of each symbol specifies a dependency on the corresponding condition.  $X$  means no dependency,  $T$  means a dependency on *True*, and  $F$  means a dependency on *False*. For example, an operation with a  $TF$  tag will be completed if  $cc_1$  is true and  $cc_2$  is false. The tag assigned to operations of all instructions that are not in branch or branch delay slots is  $XX$ .

Table 7 shows all possible tags for three-way branching. Since there are seven possible tags, a binary encoding of the tags will require at least three bits. Operation formats in VIPER have three bits for conditional execution. A possible encoding of the tags is shown in Table 8.

### 3.4.3 Computation of Completion Flags

Figure 24 shows a black-box view of the hardware unit that computes a completion signal that is used to qualify write transactions to the register file or the data cache. There is a copy of this hardware block for each functional unit. There are two sets of inputs to this hardware unit:

1.  $T_0, T_1, T_2$  are the tag bits of an operation and specify the condition required for the completion of that operation.
2.  $cc_1$  and  $cc_2$  are the condition codes that will determine which operations will be allowed to complete.

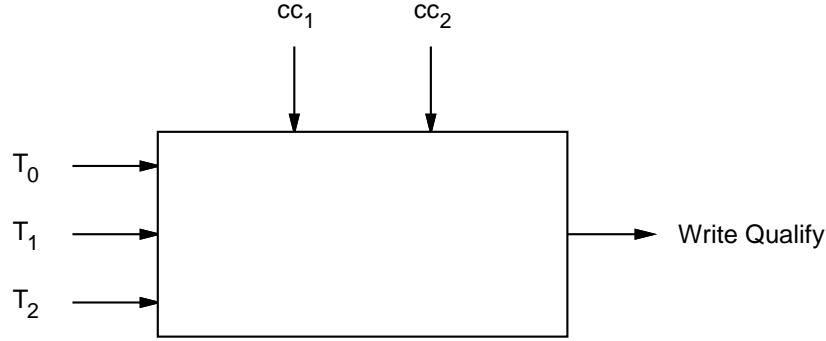


Figure 24: Computation of Execution Completion Flags

The output of this hardware unit is a signal that is used to qualify write transactions to the register file or the data cache depending on whether the operation in question is a register-to-register operation or a store operation. For all non-branch operations in branch and branch delay slots, the write qualify signal is computed during the ID pipeline stage and is used to qualify write transactions to the data cache or the register file during EX and WB pipeline stages.

The tag bits are provided in the binary encoding of an operation. In order to ensure that the processor hardware will be able to identify  $cc_1$  and  $cc_2$ , the following requirements are enforced when branch operations are scheduled:

1. For all branch operations, we require that  $cc_1$  be the parent condition and  $cc_2$  be the child condition with respect to the control tree of a branching instruction node.
2. For three-way branching, we require that branch operation issued to FU0 correspond to a path in the control tree that goes through both condition variables of the tree.
3. For regular two-way branching, we require that the branch operation be issued to FU0. We further require that  $cc_1$  be the condition variable of the control tree and  $cc_2$  be register R0.

When these requirements are enforced,  $cc_1$  is simply the LSB of the *src1* operand of FU0, and  $cc_2$  is the LSB of the *src2* operand of FU0 during the ID pipeline stage of the instruction in the branch slot. These requirements are to be met by the code generator. While requirements 2 and 3 are not strictly necessary, they significantly simplify the task of detecting the condition variables for the control hardware. These requirements can be easily met by the code generator and will result in a simple hardware implementation for conditional execution.

## 4 Code Generation

This section describes the process of code generation. The function of the code generator is to produce machine code for VIPER after all parallelizing and resource constrained scheduling optimizations are performed.

## 4.1 Overview

The input to the code generator is the program flow graph that has been compacted and gone through the Resource Constrained Scheduling process. The function of the code generator is to produce a machine program executable by the target architecture, i.e., VIPER. The final output of the code generator is an assembly language program that can be readily translated into machine code given a complete definition of the binary operation formats. The major steps involved in producing code for the target architecture are the following:

1. Instruction Address Calculation
2. Computation of Conditional Execution Tags
3. Producing Target Assembly Program
4. Scheduling Around RAW Hazards

In the following sections, each of these individual steps will be described.

## 4.2 Instruction Address Calculation

In this step, the input program flow graph (which is a directed graph) is translated into a linear array of machine instructions. Each node of the program graph is assigned the address that it will have in the final sequence of machine instructions. This is done during a depth-first search that visits every node of a given procedure and assigns an address to it.

## 4.3 Computation of Conditional Execution Tags

In this step of code generation, conditional execution tags are computed for all operations of all instructions in the program graph. As explained in Section 2.1, operations within an instruction form a tree (see Figure 26). All operations of an instruction are assigned conditional execution tags as they are visited during a recursive depth-first search. The algorithm is shown in Figure 25. Note that at the end of each path leading to a successor instruction, there is a dummy operation (not shown in the figure). These dummy operations serve as the leaf nodes of the tree of operations and signal the end of a path during the depth-first search.

At each invocation of *compute\_tag()*, the global array *Tag* contains the tag that will be assigned to the current operation *op*, and *depth* contains the depth of the current operation in the tree. For each instruction, the search starts with the operation at the top of the tree. At this point, the value of *Tag* is *XX* (i.e.,  $Tag[0] = X$  and  $Tag[1] = X$ ), and *depth* is set to zero. The search visits all operations in the tree recursively. When the search reaches a leaf node of the tree, it is time to return from the recursive call. When a conditional branch operation is encountered during the search, the value of *Tag* and *depth* are updated, and the search continues with the true (left) subtree. Upon return from searching the left subtree, the value of *Tag* is updated again, and the search continues with the false (right) subtree. In this fashion, all of the operations inside the node are visited and assigned a conditional execution tag.

```

procedure compute_tag(op)
  op.tag[0]  $\leftarrow$  Tag[0]
  op.tag[1]  $\leftarrow$  Tag[1]
  if op is a leaf operation then
    return
  else if op is a conditional branch then
    depth  $\leftarrow$  depth + 1
    Tag[depth - 1]  $\leftarrow$  "T"
    compute_tag(left successor)
    Tag[depth - 1]  $\leftarrow$  "F"
    compute_tag(right successor)
    depth  $\leftarrow$  depth - 1
  return
else
  compute_tag(successor)
return

```

Figure 25: Procedure for Computing the Conditional Execution Tags

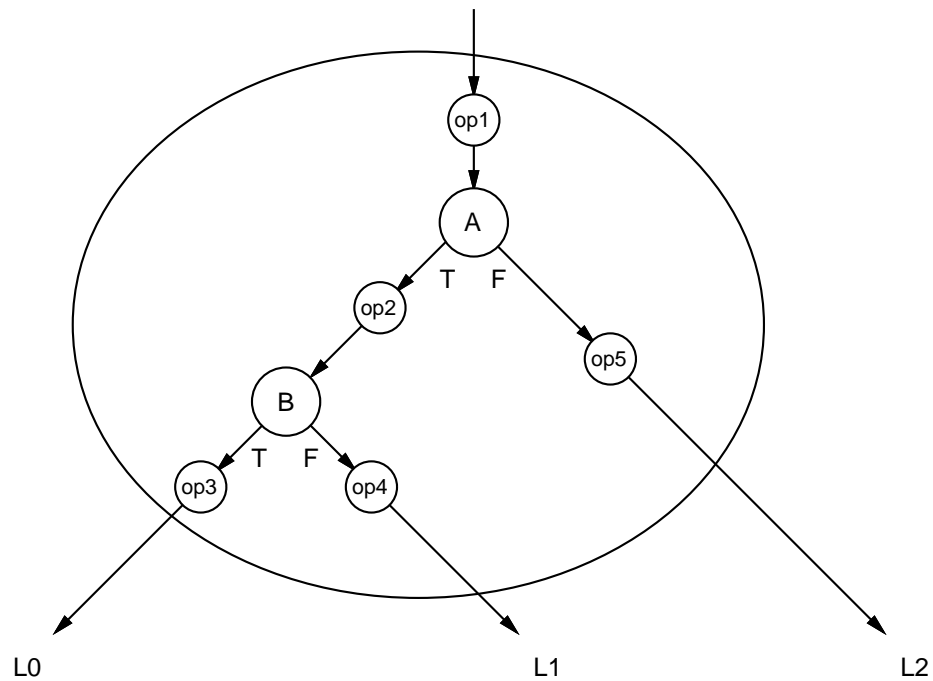


Figure 26: Tree of Operations within an Instruction

```

      .
      .
      .
11
  XX BRTF R20,R0,#10
  TX ADD  R17,R0,R12
  FX ADDI R18,#4,R18
  FX ADDI R18,#4,R11
12
  TX SLT  R0,R17,R20
  FX ADD  R19,R10,R9
  FX SUBI R18,#56,R8
  FX SUBI R18,#52,R7
13
  XX SUBI R11,#56,R5
  XX SUBI R11,#52,R1
  XX LDW  R7,R2
  XX LDW  R8,R13
14
  XX ADDI R11,#4,R3
  XX SGE  R9,R11,R4
  XX LDW  R7,R14
  XX LDW  R8,R19
      .
      .
      .

```

Figure 27: Example of VIPER Assembly Code

#### 4.4 Producing Target Assembly Program

Figure 27 shows a section of an assembly language program for VIPER. By convention, the first line in each instruction is the address of the instruction. The four subsequent lines include the operations issued to FU0, FU1, FU2, and FU3, respectively. The target assembly program is created by scanning the list of instructions starting with the instruction at address zero. Assembly instructions are created one instruction node at a time. Each operation of a given instruction node is assigned to a functional unit according to the architectural definition of VIPER and is translated into an equivalent VIPER operation. There is a one-to-one mapping from each operation of the intermediate code to a machine operation.

At this stage of code generation, the main concern is to create a schedule of instructions with a valid assignment of operations to functional units. This means that control transfer operation should be assigned to FU0 and FU1, and load/store operations should be assigned to FU2 and FU3. All other operations can be executed by all of the functional units. At a later stage, the code generator will attempt to modify this initial assignment so as to reduce the frequency of RAW hazards and the resulting pipeline stalls.

```

procedure bypass_scheduling()
   $n \leftarrow 24^w$ 
  for  $i \leftarrow 0$  to  $N - w$  do
    for  $j \leftarrow 0$  to  $w - 1$  do
      for  $k \leftarrow 0$  to 3 do
         $W[j][k] \leftarrow PROG[i+j][k]$ 
       $best \leftarrow 0$ 
       $min \leftarrow permutation\_cost()$ 
      for  $j \leftarrow 1$  to  $n - 1$  do
        for  $k \leftarrow 0$  to  $w - 1$  do
           $perm[k] \leftarrow \lfloor \frac{j}{24^k} \rfloor \bmod 24$ 
          rearrange  $W[k]$  according to  $perm[k]$ 
        if valid_permutation() then
           $cost \leftarrow permutation\_cost()$ 
          if  $cost < min$  then
             $min \leftarrow cost$ 
             $best \leftarrow j$ 
        for  $j \leftarrow 0$  to  $w - 1$  do
           $perm[j] \leftarrow \lfloor \frac{best}{24^j} \rfloor \bmod 24$ 
          rearrange  $W[j]$  according to  $perm[j]$ 
        for  $j \leftarrow 0$  to  $w - 1$  do
          for  $k \leftarrow 0$  to 3 do
             $PROG[i+j][k] \leftarrow W[j][k]$ 

```

Figure 28: Procedure Used to Schedule Operations to Reduce RAW Hazards

#### 4.5 Scheduling Around RAW Hazards

After the code generator creates an initial assembly program it attempts to reduce the frequency of pipeline RAW hazards by modifying the assignment of operations to functional units in each instruction. In other words, the code generator tries to find a new permutation of operations in each instruction that will result in less RAW hazards at run-time. Obviously, the new permutations should not violate any of the assignment rules that are required by the architectural definition of VIPER, e.g., load/store operations should not be assigned to FU0 or FU1.

The algorithm that performs the scheduling is shown in Figure 28. It is based on a sliding window that contains  $w$  contiguous instructions from *PROG* (see Figure 29). At each step of the algorithm,  $w$  contiguous instructions are copied from *PROG* into  $W[0..w-1][0..3]$ . The algorithm considers all possible permutations of operations in the instructions currently within the window and selects the one with the lowest *cost*. In this context, cost is the number of stall cycles that will arise when executing the instructions currently within the window. Next, the window is advanced by one instruction, and the process is repeated. Thus, the sliding window scans the entire program from top to bottom and at each point attempts to find the permutation of operations with the smallest number of stalls.

The size of the window,  $w$ , is one of the parameters that can be varied. Generally, the larger the size of the window, the better the schedule that can be achieved. The running time of the algorithm increases as the size of the window grows. Within an instruction there

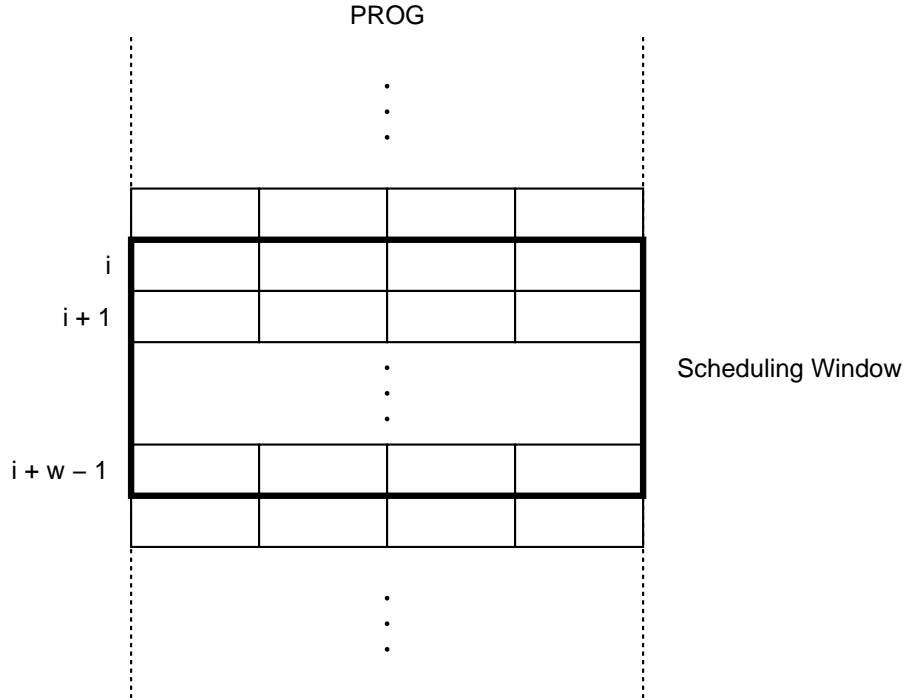


Figure 29: The Scheduling Window

are  $4!=24$  different permutations of operations. In a window of size  $w$ , there are  $w$  different instructions; thus, there are  $n = 24^w$  different assignment possibilities. For a program with  $N$  instructions, the algorithm takes a time in  $\Theta(N \cdot 24^w)$ . Thus, the time taken by the algorithm grows exponentially with  $w$ ; however, as we will see in the next section, for  $w > 1$ , the improvement that can be achieved is very small. The time taken by the algorithm for  $w = 1$  is small relative to the total compilation time. Thus, the exponential growth of the time taken by the algorithm does not present a problem.

The other input parameter of the scheduler is the  $P$  matrix that was defined in Section 3.3.1. The  $P$  matrix describes the topology of the bypassing interconnection network. It is used by the `permutation_cost()` procedure to compute the *cost* of a given permutation. It is also used during simulations to determine whether the processor pipeline needs to be stalled in a given execution cycle.

The 24 possible permutations of operations in each instruction are indexed from 0 to 23. The array `perm[0..w - 1]` contains the permutation index of the instructions in  $W$ ; `perm[i]` holds the permutation index of the instruction in  $W[i]$ . Operations in  $W[i]$  are rearranged according to `perm[i]`. Permutation number 0 corresponds to the initial permutation of operations when  $w$  instructions are read into  $W$  from *PROG*. This corresponds to `perm[i] = 0` for  $i = 0, 1, \dots, w - 1$ . The  $n = 24^w$  different assignment possibilities are indexed from 0 to  $n - 1$ . Assignment number 0 corresponds to the initial assignment of operations at the time  $w$  contiguous instructions are read into  $W$  from *PROG*. For assignment number  $j$ , `perm[k]` is computed as follows:

$$perm[k] = \left\lfloor \frac{j}{24^k} \right\rfloor \bmod 24$$

$w$	$P = P_1$		$P = P_2$		$P = P_4$	
	$S$	% improvement	$S$	% improvement	$S$	% improvement
0	1.78	N/A	1.87	N/A	2.01	N/A
1	1.93	8.4	2.02	8.0	2.09	4.0
2	1.94	0.5	2.03	0.5	2.09	0.0
3	1.94	0.0	2.03	0.0	2.09	0.0

Table 9: Effectiveness of Scheduling Around RAW Hazards

where  $k = 0, 1, \dots, w - 1$ . The initial assignment of operation in  $W$  (where  $j = 0$  and  $perm[i] = 0$  for  $i = 0, 1, \dots, w - 1$ ) is considered as the initial best assignment, and its cost is recorded as the current minimum. The algorithm then goes through the remaining  $n - 1$  possibilities. For each possible assignment, the array  $perm$  is updated, and the operations in  $W[k]$  are rearranged according to the permutation in  $perm[k]$ . The algorithm then checks to see if the assignment of operations to functional units is valid according to the architectural requirements of VIPER. If the assignment is valid, the algorithm then computes the cost of the assignment and compares it to the current minimum  $min$ . If the cost of this assignment is lower than  $min$ , then the current assignment is recorded as the best solution, its index is stored in  $best$ , and its cost is stored in  $min$ . After all possible assignments are evaluated, the operations of the instructions in  $W$  are rearranged according to the permutations dictated by  $best$ , and  $W$  is then copied into  $PROG$ . The scheduling window is then advanced by one instruction, and the process described above is repeated until the end of the program is reached.

#### 4.5.1 Simulation Results

The effectiveness of scheduling around RAW hazards was measured via a set of simulations. The results of these simulations are shown in Table 9 and Figure 30. Three sets of results are presented; each set corresponds to a different bypassing interconnection network topology. The  $P$  matrices of the three data sets are as follows:

$$P = P_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$P = P_2 = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

$$P = P_4 = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

For each interconnection network,  $w$  (the window size) was varied from 1 to 3, and the corresponding speed-up values were measured. Window size zero corresponds to no scheduling around RAW hazards. The speed-up values shown in Table 9 are the harmonic mean of the speed-up values for all benchmarks.

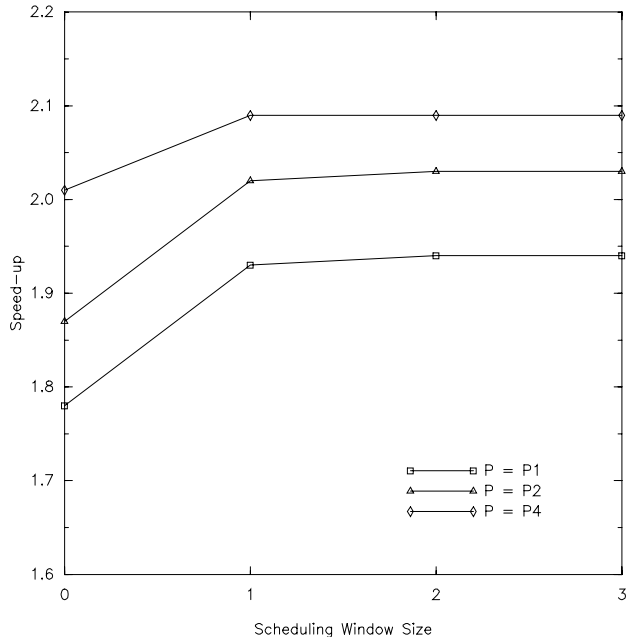


Figure 30: Effectiveness of Scheduling Around RAW Hazards

As the size of the window is increased, a better schedule of operations can be produced and a higher level of performance is observed; however, the time taken by the scheduling algorithm grows exponentially with the size of the window. The observed performance improvement was very small for  $w > 1$ . For  $w > 2$ , no improvement was observed. This suggests that  $w = 1$  is sufficient to produce a good schedule by this algorithm. For  $w = 1$ , the time taken by the algorithm is very small compared to the total compilation time.

## 5 Conclusion

In this paper, we have presented the architectural design and analysis of VIPER, a VLIW processor designed to take advantage of instruction level parallelism to increase performance beyond RISC architectures. VIPER takes advantage of the parallelizing capabilities of Percolation Scheduling. The approach taken in the design of VIPER was a comprehensive one that addressed design issues involving implementation constraints, organizational techniques, and code generation strategies. Hardware/software trade-offs were analyzed at various points during the design process. With this approach, design problems can be addressed by a combination of hardware and software techniques. This allows the architect to select the most effective solution and ultimately leads to a balanced and cost-effective design.

We presented the architectural design and analysis of VIPER. Various strategies involving machine organization were studied with respect to the efficiency of their hardware implementations. The general organization of VIPER was established via this analysis. An

important issue in the design of VIPER involved its pipeline structure. We have discussed and analyzed the subtle relationships that exist among the pipeline structure, the memory addressing mode, the bypassing hardware, and the cycle time of the processor. We proposed a pipeline structure that results in enhanced performance in a VLIW processor such as VIPER. We showed that this enhancement is due to the relationship that exists between the structure of the pipeline and the bypassing hardware. We analyzed the problem of global bypassing in a processor with multiple functional units. We explored the conflicting performance effects of increasing the connectivity of the bypassing interconnection network and arrived at an optimal point in the design space for the bypassing interconnection network.

One of the important architectural goals of VIPER was to provide support for the execution of multiway branch operations. Another goal was to provide support for conditional execution of operations following a branch. The multiway branching and conditional execution mechanism involves a fine balance of hardware and software requirements. We presented the scheme used to achieve these objectives in the design of VIPER.

An integral objective of this research was to develop the code generator for the target architecture. The overall code generation strategy was a result of the hardware/software trade-offs that were studied during the design process. We have presented the algorithms that are used by the code generator. The code generator introduces a new code scheduling technique that is devised to minimize the frequency of pipeline stalls caused by RAW data hazards.

## References

- [1] D. J. Kuck, *The Structure of Computers and Computations*, vol. 1, John Wiley and Sons, New York, 1978.
- [2] K. Hwang and F. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, 1984.
- [3] J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers*, vol. C-30, pp. 478-490, July 1981.
- [4] A. Nicolau, "Percolation Scheduling: A Parallel Compilation Technique," Technical Report 85-678, Dept. of Computer Science, Cornell University, May 1985.
- [5] R. Potasman, *Percolation-Based Compiling for Evaluation of Parallelism and Hardware Design Trade-Offs*, PhD Dissertation, University of California, Irvine, 1991 (in preparation).
- [6] J. Hennessy, "VLSI Processor Architecture," *IEEE Transactions on Computers*, vol. C-33, pp. 1221-1246, December 1984.
- [7] M. G. H. Katevenis, *Reduced Instruction Set Computer Architectures for VLSI*, MIT Press, Cambridge, Mass., 1985.
- [8] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, and T. Gross, "Design of a high performance VLSI processor," *Proc. 3rd Caltech Conf. VLSI*, California Institute of Technology, Pasadena, CA, 1983, pp. 33-54.
- [9] G. Kane, *MIPS R2000 RISC Architecture*, Prentice Hall, Englewood Cliffs, N.J., 1988.

- [10] P. Chow and M. Horowitz, "Architectural Tradeoffs in the Design of MIPS-X," *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pp. 300-308, Pittsburgh, Pennsylvania, June 1987.
- [11] D. D. Lee, S. I. Kong, M. D. Hill, G. S. Taylor, D. A. Hodges, R. H. Katz, and D. A. Patterson "A VLSI Chip Set for a Multiprocessor Workstation," *IEEE Journal of Solid State Circuits*, pp. 1688-1698, December 1989.
- [12] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Palo Alto, 1990.
- [13] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," *IEEE Transactions on Computers*, Vol. 37, 1988, pp. 967-979.
- [14] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 Departmental Supercomputer," *Computer Magazine*, Vol. 22, No. 1, 1989, pp.12-35.
- [15] R. C. Cohn, T. Gross, M. Lam, and P. S. Tseng, "A VLIW Architecture for a Trace Scheduling Compiler," *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 1989, pp. 2-14.
- [16] J. Labrousse and G. A. Slavenburg, "CREATE-LIFE: A Modular Design Approach for High Performance ASIC's," *COMPCON* 1990.
- [17] M. Johnson, *Superscalar Microprocessor Design*, Prentice Hall, Englewood Cliffs, N.J., 1991.
- [18] G. S. Tjaden and M. J. Flynn, "Detection and Parallel Execution of Independent Instructions," *IEEE Transactions on Computers*, Vol. 19, No. 10, October 1970.
- [19] A. S. Aiken, *Compaction-Based Parallelization*, Ph.D. Dissertation, Cornell University, August 1988.
- [20] A. Nicolau, "Uniform Parallelism Exploitation In Ordinary Programs," *Proceedings of the 1985 International Conference on Parallel Processing*, pp. 614-618, 1985.
- [21] A. Aiken and A. Nicolau, "Perfect Pipelining: A New Loop Parallelization Technique," In *Proceedings of the 1988 European Symposium on Programming*, Springer Verlag Lecture Notes in Computer Science no. 300, March 1988.
- [22] C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass., 1980.
- [23] A. Abnous, R. Potasman, N. Bagherzadeh, and A. Nicolau, "A Percolation Based VLIW Architecture," *Proceedings of the 1991 International Conference on Parallel Processing*, pp. 144-148, 1991.
- [24] A. Abnous, C. Christensen, J. Gray, J. Lenell, A. Naylor, and N. Bagherzadeh, "VLSI Design of the *Tiny RISC* Microprocessor," *Proceedings of the 1992 Custom Integrated Circuits Conference*, Boston, MA, May 1992.

- [25] T. R. Gross, J. L. Hennessy, S. A. Przybylski, C. Rowen, "Measurement and Evaluation of the MIPS Architecture and Processor," *ACM Transactions on Computer Systems*, August 1988, pp. 229-257.
- [26] A. Vladimirescu and S. Liu, *The Simulation of MOS Integrated Circuits Using SPICE2*, ERL Memo No. ERL M80/7, Electronics Research Laboratory, University of California, Berkeley, October, 1980.
- [27] C. Tomovich, *MOSIS User Manual*, Release 3.1, USC/Information Sciences Institute, Marina Del Rey, CA, 1988.