DESIGN AND PERFORMANCE EVALUATION OF A SUPERSCALAR DIGITAL SIGNAL PROCESSOR

by

HANI BAGNORDI

B.Sc., Laval University, 1994

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

DEPARTMENT OF ELECTRICAL ENGINEERING

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

July 1997

© Hani Bagnordi, 1997
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of **Electrical Engineering**

The University of British Columbia
Vancouver, Canada

Date **July 28, 1997**
Abstract

Multimedia applications are compute intensive applications that often contain multiple streams of operations such as audio and video running in parallel. These characteristics make them suitable for multi-stream processing through the use of parallel processing. In the first part of the thesis we evaluate the instruction and data bandwidth requirements of typical signal processing and multimedia functions. In the second part of the thesis we compare the advantages and disadvantages of programmable and algorithm specific integrated circuit digital signal processors.

In order to accommodate the high level of instruction and data bandwidth found in multimedia applications we propose a superscalar digital signal processor (DSP) that can execute multiple instructions in parallel. A second feature of the proposed DSP is scalability. Scalability is an important feature needed to accommodate future increases in the performance requirements of multimedia applications. Our proposed DSP is also compatible with existing instruction-set architectures which eliminates the need for a specialized compiler.

Next, we introduce a detailed analysis of the availability and distribution of instruction-level parallelism in ten existing multimedia applications. We also discuss the relationship between instruction-level parallelism and machine parallelism. In the following part of the thesis, we discuss machine parallelism and describe the different superscalar techniques used to fetch, decode, and execute multiple instructions per cycle. These techniques include out-of-order issue with out-of-order completion, register renaming, and branch prediction. A parameterizable superscalar simulator is used to simulate ten real-world multimedia applications on five different models of parallelism. The five models represent a wide range of machine parallelism, from a bare scalar machine to an ideal superscalar machine with unlimited parallelism. Results of the instruction-
level parallelism study show that the multimedia benchmarks simulated contain a high level of parallelism in their code which make them suitable candidates for multiple-issue machines. In addition, this high level of instruction parallelism proves to be evenly distributed throughout the program which helps in maintaining a good balance between available parallelism and on-chip resources. Furthermore, machine parallelism simulation results show that instruction fetching places the ultimate limit on speedup and is the most critical factor in determining overall performance. However, by using aggressive branch prediction mechanisms and out-of-order issue with out-of-order completion techniques we are able to obtain a two to four fold increase in performance compared to a single issue scalar machine. Overall results show that abundant instruction parallelism combined with adequate machine parallelism proves to be a real performance booster for multimedia applications.
# Table of Contents

Abstract ................................................................. ii  
List of Tables ......................................................... vii 
List of Figures ........................................................ viii 
Glossary ................................................................. x 
Acknowledgment ....................................................... xi 

Chapter 1  Introduction ................................................ 1  
1.1 Background ................................................................ 1  
1.2 Objectives ............................................................... 3  
1.3 Thesis Outline ......................................................... 5  

Chapter 2  Motivation ..................................................... 6  
2.1 Introduction ............................................................. 6  
2.2 Application Requirements ......................................... 6  
2.2.1 Instruction Bandwidth ........................................... 7  
2.2.2 Data Bandwidth ................................................... 8  
2.3 Characteristics of Digital Signal Processors ..................... 10  
2.3.1 Memory Architecture .......................................... 10  
2.3.2 External Memory Access ....................................... 13  
2.3.3 Pipelining ......................................................... 14  
2.4 Programmable DSPs .................................................. 15  
2.5 Application Specific Processors .................................... 16  
2.6 Multimedia Extensions .............................................. 17  
2.7 Proposed Solution .................................................... 19
Chapter 3 Architectural Description

3.1 Introduction

3.2 Architectural Organization

3.3 Instruction Set
  3.3.1 Load and Store Instructions
  3.3.2 Computational Instructions
  3.3.3 Jump and Branch Instructions

3.4 Pipeline Stages
  3.4.1 Fetch Stage
  3.4.2 Decode Stage
  3.4.3 Execute Stage
  3.4.4 Write Back Stage
  3.4.5 Result Commit Stage

3.5 Architectural Units
  3.5.1 Fetch Unit
  3.5.2 Decode Unit
  3.5.3 Execution Unit

3.6 Simulator Description

3.7 Benchmarks & Models of Parallelism

3.8 Conclusion

Chapter 4 Instruction-Level Parallelism

4.1 Introduction
List of Tables

Table 2.1 Typical bandwidth requirements for multimedia applications .................. 9
Table 2.2 Comparison of three different DSP technologies ................................. 18
Table 2.3 Comparison of three different multiple-issue microarchitectures .............. 20
Table 3.1 Hardware parallelism models .............................................................. 39
Table 3.2 List of benchmarks .............................................................................. 41
Table 4.1 Percentage of decreasing parallelism for the five models ....................... 45
Table 4.2 Instructions latency ............................................................................. 51
Table 5.1 Branch prediction buffer size and misprediction penalty for a several commercial microprocessors 60
Table 5.2 Issue policies and max instruction issue for different commercial microprocessors 62
Table 5.3 Average Instruction Fetch for different branch prediction technology ....... 69
Table 5.4 Fetch mechanism for different commercial microprocessors .................. 69
Table 5.5 Number of functional units and their distribution for several commercial microprocessors 77
List of Figures

Figure 2.1 Block diagram of a 2nd order FIR filter ......................................................... 8
Figure 2.2 Von Neumann architecture ............................................................................. 11
Figure 2.3 A Harvard architecture .................................................................................. 12
Figure 2.4 Instruction execution flow though a five stage pipeline ................................. 14
Figure 3.1 Block diagram of the Superscalar Digital Signal Processor ......................... 24
Figure 3.2 Instruction-level format ................................................................................. 25
Figure 3.3 Pipeline stages of a 2-way superscalar processor ........................................... 27
Figure 3.4 Instruction Window of an out-of-order issue processor ................................. 28
Figure 3.5 Layout of the decoder .................................................................................... 31
Figure 3.6 Layout of a decoder that contains wasted decoder slots ............................... 31
Figure 3.7 Layout of a Reorder Buffer .......................................................................... 32
Figure 3.8 Block diagram of the simulation process ....................................................... 37
Figure 4.1 Parallelism under the five models ................................................................. 44
Figure 4.2 Effects of register renaming on instruction-level parallelism ......................... 46
Figure 4.3 Instruction-run length distribution ................................................................. 48
Figure 4.3(b) Zoom in view of Figure 4.3 ..................................................................... 48
Figure 4.4 Branch prediction accuracy in function of branch prediction buffer size ........ 49
Figure 4.5 Effects of cycle width on parallelism ............................................................. 50
Figure 4.6 Instruction issue distribution ......................................................................... 54
Figure 5.1 Speedup for different Issue policies ............................................................... 58
Figure 5.2 Effects of output dependency on performance ............................................. 64
Figure 5.3 Instruction mix distribution .......................................................................... 66
Figure 5.4  Average number of instruction fetch for different decode size ..............................68
Figure 5.5  Branch Accuracy vs. Branch Target Buffer size ......................................................71
Figure 5.6  Instructions fetched for different decode width and branch prediction mechanism ..................................................72
Figure 5.7  Speedup improvement vs. Instruction Window Size ....................................................74
Figure 5.8  Speedup vs. number of ALUs .................................................................................75
Figure 5.9  Performance increase from using 2 Load and 2 Store Units .......................................76
## Glossary

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Arithmetic and Logic unit, responsible for executing all integer arithmetic and logic operations.</td>
</tr>
<tr>
<td>ASIC</td>
<td>Algorithm Specific Integrated Circuit, a custom designed processor optimized for a particular algorithm.</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor, a dedicated processor for signal processing functions and algorithms.</td>
</tr>
<tr>
<td>FU</td>
<td>Functional units (e.g. adders and multipliers).</td>
</tr>
<tr>
<td>BPB</td>
<td>Branch Prediction Buffer, used to store the previous history of branch instructions.</td>
</tr>
<tr>
<td>IW</td>
<td>Instruction Window, a buffer between the decoder and the functional units used for issuing instructions out-of-program order.</td>
</tr>
<tr>
<td>RB</td>
<td>Reorder Buffer, used to ensure that instructions are committed to the file register in program order.</td>
</tr>
<tr>
<td>Scalar</td>
<td>A machine that issues instructions sequentially one at a time.</td>
</tr>
<tr>
<td>Speedup</td>
<td>A measure of the performance of a superscalar machine and is calculated by dividing the total number of instructions executed by the total number of cycles taken to complete the program.</td>
</tr>
<tr>
<td>Superscalar</td>
<td>A machine that issues multiple instructions each cycle.</td>
</tr>
</tbody>
</table>
Acknowledgment

I would to thank my supervisor Dr. Ito for his moral and financial support during my research. I would like also to thank Dr. V. Leung and Dr. Babak Hamizadeh on my defense committee for their suggestions regarding the thesis. I am also grateful to my lab-mates for their support and help during my study. Special thanks to Kendra Cooper, Yue He, Xiaobin Lee, and last but not least Yul Chu, it's been a wonderful experience. I would like to express my special gratitude to the Lord for the blessing that HE gave me during all these years. Without HIS support I wouldn't be here today. Also, many thanks to my wife, Yoko, for her patience and understanding and all the broken promises of getting away from my studies.
Chapter 1 Introduction

1.1 Background

The performance of a program is limited by two factors: the available instruction-level parallelism (ILP) in the program and the amount of work that a microprocessor can do to extract that parallelism. While some programs do not have enough parallelism to take advantage of available machine parallelism other programs do not achieve their full performance potential because of limited machine parallelism. Therefore, to achieve a high level of performance we need to balance machine parallelism and instruction parallelism.

In order to achieve a good balance between available ILP and machine parallelism we first need to know exactly how much parallelism is present in the target applications and what the limits are for parallelism. We also need to know how evenly the parallelism is distributed within the program. Nonuniformity in the distribution of ILP often leads to an overestimation of the amount of resources needed to achieve the maximum speedup [21]. The first task in achieving a high level ILP is to understand for each instruction the dependencies among the instructions. The more dependencies that exist between instructions the less parallelism that is available for the microprocessor to extract.

The type of applications that are well suited for parallel machines are applications that contain a high level of ILP. Digital signal processing is an example of such an application. Digital signal processing programs are characterized by [14, 15]:

- highly structured operational control, containing few data dependent jump sequences
- frequent complex memory access (multiple source)
• short repetitive loops (FFT and FIR filters)

• perform multiple operations in parallel (IIR filters)

Thus, it is not a problem for signal processing applications to keep the ALU pipeline full. However, the performance of existing scalar digital signal processors (DSPs) is limited by the very low level of on-chip parallelism combined with a high interprocess communication overhead. For example, the adder cannot run independently from the multiplier without incurring overhead cycles [1]. Dedicated algorithm specific integrated circuits (ASICs) DSPs also have a number of disadvantages like high development cost and non-upgradable hardwired functions. Preliminary evaluation of programmable and ASIC DSP solutions indicates that there is a need to evaluate alternative architectures. However, for any new architecture to be accepted we need to make sure that it will meet the following requirements:

• greater than one fold increase in performance compared to existing DSP architectures

• scalability to accommodate future increases in the applications' execution requirements

• ease of integration to existing instruction-set-architectures

There are three different techniques used to improve the performance of a microprocessor [8]. Each of these techniques is associated with a microarchitecture (shown in parentheses) that optimizes a particular performance feature such as speed, number of instructions executed, or total number of cycles executed. These techniques are:

• reduce the processor cycle time by using faster, pipelined functional units (superpipelined machines)
• reduce the total number of instructions needed to execute a program by packing all the
instructions that can be done simultaneously into a single instruction word (Very Long
Instruction Word or VLIW machines)

• increase the average number of instructions executed per cycle (IPC) by issuing multi-
ples independent instructions in parallel (superscalar machines)

The first architecture, superpipelined, is the most difficult to implement because it requires
a complete redesign of the machine’s pipeline including the functional units and memory
access. The second architecture, VLIW, relies on extracting parallel instructions at compile time
by handing to the compiler the task of finding and extracting the parallelism in the code and pack-
ing all of the independent instructions into a single large machine word. However, extracting
instruction parallelism at compile time is not very efficient because compilers cannot keep track
of which system resources such as execution units, registers and bus will be busy on a given
cycle. The third architecture, superscalar, overcomes this problem by scheduling instructions as
they are fetched with full knowledge of what resources are available. Therefore, a superscalar
machine with a few execution pipes performs faster than a much larger VLIW machine.

The other advantage of the superscalar architecture is that they don’t require a special
compiler like in the case of the VLIW architecture. Therefore, code targeted to the superscalar
architecture is compatible with the large base of available, commercial applications.

1.2 Objectives

The first objective of our study is to measure how much parallelism is truly inherent in
commonly used signal processing programs, independent of any particular hardware limitation. In
this thesis we concentrate our effort in examining one particular type of signal processing application, multimedia. To achieve the first objective, we investigate the upper bound on ILP and dependencies that impact the ILP. The upper bound on ILP is determined using ten real-world multimedia applications. This upper bound is used to evaluate machine parallelism. Any increase in machine parallelism beyond that point has no effect on performance as the machine cannot make any use of the additional instruction-parallelism. Simulation are performed using a proprietary trace analysis program. This program simulates the ten multimedia benchmarks under five different hardware models. The models represent a wide range of machine parallelism from a single issue scalar machine to an omniscient machine with infinite parallelism.

Dependencies are investigated to determine what type of dependencies have the greatest impact on parallelism. The types of dependencies include data, control, and structural. We discuss techniques to eliminate these dependencies such as register renaming, branch prediction, and duplication of resources. The potential of each of these techniques is shown. To study the impact of different hardware configurations on ILP five different models of parallelism representing a wide range of machine parallelism. The models range from a single issue scalar machine to an unlimited multiple-issue machine.

The second objective of this thesis is to study the performance of a dynamically scheduled superscalar DSP. The purpose is to determine the feasibility of using a superscalar DSP to enhance the performance of compute intensive multimedia applications. A quantitative approach based on the simulation of ten real-world multimedia benchmarks is used to evaluate design trade-offs and determine the mix of resources required to achieve an optimal level of performance. Results are used to efficiently balance instruction-level parallelism and machine parallelism. Sev-
eral hardware techniques are used to improve machine parallelism: out-of-order issue with out-of-order completion, register renaming, branch prediction, and duplication of resources.

1.3 Thesis Outline

- Chapter 2 presents a comparative study of different DSP architectures and technologies such as VLIW and superscalar. We use a case study of several commercial microprocessors to evaluate the advantages and disadvantages of the architectures in terms of speed of execution, complexity, and compatibility, of each microarchitecture. We propose our architectural solution in this chapter as well.

- Chapter 3 introduces our new superscalar DSP architecture. Details about the instruction set and the DSP pipeline are presented. The architecture and function of each of the instruction window, reorder buffer, branch prediction and the different functional units are discussed. The simulation environment and trace gathering techniques are explained along with a complete description of the tracing utility that we developed.

- Chapter 4 discusses instruction-level parallelism. Simulation results that show the effects of data, control, and structural dependencies on performance are presented.

- Chapter 5 presents the simulation results for the machine parallelism. The impact of issue width, branch prediction, fetch efficiency, and resource duplication on performance for our new architecture are discussed.

- Chapter 6 presents the conclusions and future work.
Chapter 2 Motivation

2.1 Introduction

In the first part of the chapter we introduce the instruction and data bandwidth requirements of typical digital signal processing and multimedia applications. A Finite Impulse Response (FIR) filter is used as an example of the instruction bandwidth requirements for common signal processing operations. A brief description of the architectural characteristics of a digital signal processor (DSP) are described to provide a background information for the reader.

In the second part of the chapter, the advantages and disadvantages of programmable DSPs, ASIC DSPs, and the new multimedia extensions are reviewed. Next, a comparison between these three approaches in terms of performance, cost, and compatibility to existing instruction-set-architectures (ISAs) is presented. The purpose of this survey is to better understand the shortcomings of existing DSP and to draw the architectural specifications for a new DSP architecture.

In the final part of the chapter we propose a dynamically scheduled superscalar DSP capable of issuing multiple-instructions per cycle. The proposed microarchitecture is compared to superpipelined and VLIW architectures. Finally, the advantages and disadvantages of the superpipelined, VLIW, and superscalar architectures are discussed.

2.2 Application Requirements

Digital signal processing applications are compute intensive applications that require high instruction and data bandwidths to achieve satisfactory results. Multimedia applications are the most demanding in terms of computation requirements because they incorporate multiple instruction streams such as audio and video that need to be processed in parallel. Also, multimedia appli-
cations have real-time response requirements in order to keep a realistic feel to the application. For example, a video-conferencing session where image and sound are out of synchronization will lose all interest with the user. To better understand the impact that instruction and data bandwidth have on system performance the following two sections introduce computation and data bandwidth requirements of some typical signal processing operations.

2.2.1 Instruction Bandwidth

Digital signal processing functions contain a large amount of parallelism in their code that enables multiple arithmetic and data transfer operations to be performed in parallel [1, 14, 15]. Figure 2.1 shows a block diagram together with the algebraic equation of a simple 2nd order Finite Impulse Response (FIR) digital filter. The filter is comprised of two delay elements, three multiply operators, and two addition operators. Each iteration of the filter loop or filter tap operation requires:

- an instruction fetch
- two operand fetches from data memory
- a multiplication
- an addition
- shifting data in the delay elements

Modern DSPs can implement a FIR filter with one instruction cycle per tap through a combination of pipelining, parallel memory access, and multiply/accumulate hardware. However, if we need to execute several filter's tap in parallel then the complexity of the hardware increases dramatically.
Signal processing applications contain a large number of small repetitive loops like the one shown in Figure 2.1 [14, 15]. These DSP loops are characterized with a high level of "intra-loop" parallelism, or parallelism within the entity of the same loop. Limiting the search for instruction parallelism to "intra-loop" parallelism limits performance considerably because of the small size of these loops. To improve performance, a solution is to search for "inter-loop" parallelism by using either a software solution like loop unrolling or a hardware method like branch prediction.

![Block diagram of a 2nd order FIR filter](image)

\[
y(n) = a_1 \cdot x(n) + a_2 \cdot x(n - 1) + a_3 \cdot x(n - 2)
\]

Figure 2.1 Block diagram of a 2nd order FIR filter

### 2.2.2 Data Bandwidth

In the investigation of the instruction requirements for a common signal processing operation, a FIR filter, we note that half of the instructions needed to execute a single filter tap operation are data transfer operations. The operations are composed of fetching one instruction from program memory and two operands from data memory. If we execute three tap operations in parallel,
then we need to fetch 6 operands from memory concurrently which puts a heavy demand on the data bandwidth requirements of the system.

In this section we discuss the data bandwidth requirement of two typical multimedia application: 1D-graphics and real-time video playback. Table 2.1 shows the data transfer rate in terms of the number of bytes/sec needed for different color and speed settings [4, 5]. We note that as we increase the resolution for the applications, the data bandwidth increases considerably. This proves that data bandwidth can be major bottleneck in our efforts to increase system performance [6]. In the next section we investigate the different memory architectures commonly used in commercial DSPs.

<table>
<thead>
<tr>
<th>Resolution (pixels)</th>
<th>Colors (bits/pixels)</th>
<th>Refresh Rate (update/sec)</th>
<th>Bandwidth (bytes/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphics</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>640x480</td>
<td>8</td>
<td>10</td>
<td>2.9 MB</td>
</tr>
<tr>
<td>1024x768</td>
<td>16</td>
<td>10</td>
<td>15 MB</td>
</tr>
<tr>
<td>1280x1024</td>
<td>24</td>
<td>10</td>
<td>37.5 MB</td>
</tr>
<tr>
<td>Real-Time Video</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>160x120</td>
<td>8</td>
<td>15</td>
<td>288 KB</td>
</tr>
<tr>
<td>320x240</td>
<td>24</td>
<td>15</td>
<td>3.5 MB</td>
</tr>
<tr>
<td>Playback</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>640x480</td>
<td>24</td>
<td>30</td>
<td>26.3 MB</td>
</tr>
</tbody>
</table>

Table 2.1 Typical bandwidth requirements for multimedia applications
2.3 Characteristics of Digital Signal Processors

DSPs are dedicated processors for executing real-time, arithmetic-intensive signal processing applications. Hardware optimization and dedicated hardwired functions give DSPs a considerable advantage over traditional microprocessors. Some of the architectural features that characterize programmable DSPs are [1, 14, 15]:

- low latency functional units
- zero over-head looping
- multiple data and instruction path
- fast multiplier/accumulator
- on-chip RAM and/or ROM to speedup instruction and operand fetching

However, the performance of current scalar DSP chips is limited by the very low level of on-chip parallelism combined with a high interprocess communication overhead. For example, the adder on a DSP cannot run independently from the multiplier without incurring overhead cycles. Another disadvantage of a scalar DSP is that there is little support for multi-chip parallel operations. The memory architecture, external memory access, and pipelining characteristics of DSPs are discussed in this section.

2.3.1 Memory Architecture

Typical DSP algorithms require a large memory bandwidth to accommodate several accesses to memory within one instruction cycle (See section 2.1.2). Great speedup in execution time is obtained if the data are stored in on-chip memory. However, the available amount of on-chip memory in DSPs is usually limited in size which forces designers to use slower, external
memory for storing program code and data. The drawback of using external memory is that each memory access requires multiple cycles to complete a read operation. Furthermore, in a typical DSP application external memory space is usually shared by data and program memory which often compete for external bus control.

![Diagram of Von Neumann architecture](image)

Figure 2.2 Von Neumann architecture

In order to speedup memory accesses we need to multiplex data and memory reads in a way that allows simultaneous program and data accesses to be processed in parallel. It follows that a conventional Von Neumann architecture, used in the majority of general purpose microprocessors, can’t be effectively used in DSP applications. The shortcoming of the von Neumann architecture is the result of its single bus design which limits memory operations to a single access per cycle. Figure 2.2 shows a model of a von Neumann architecture.

DSPs provide a wide range of architectural alternatives that are based on the Harvard architecture [1]. The advantage of using a Harvard architecture is that the program is stored separately from the data on which it operates. This enables a complete overlap of instruction fetches and executions to be obtained. Figure 2.3 shows a basic model of a Harvard architecture. Other
“modified” Harvard architectures include multiple data and data/program memory banks, each with its own set of buses [14, 15]. In general, using separate multiple memories for instructions, data, or a combination of both facilitates multiple memory access operations to be executed in parallel. However, it should be noted that using multiple independent memory banks entails a duplicate set of buses and control logic for each additional bank, which adds to the chip size.

![Harvard Architecture Diagram](image_url)

Figure 2.3 A Harvard architecture.

Because of the prohibitive cost in terms of chip area and the design complexity associated with using multiple independent memory banks, DSP designers look at other alternatives including using fast memories and multiport memory banks. Fast memories allow multiple, sequential accesses per instruction cycle over a single bus. As a result, a combination of fast memory and a modified Harvard architecture can be used to increase memory bandwidth. The advantage of this combination is that fewer memory banks are used in comparison to the Harvard architecture. For example, a modified Harvard architecture that uses two banks of fast memory with half-cycle access time can perform up to four memory accesses in a single cycle.

The other alternative to the Harvard architecture is using multiported memories that allow multiple concurrent memory accesses over two or more independent sets of buses. Although
memory access time is improved, the disadvantage of multiport memory is that they are much more costly to implement than standard, single-ported memories. Difficulties also arise in implementing I/O access because multiple address and data buses must be externally available as I/O pins.

The high bandwidth memory architectures of programmable DSPs are complemented by rich addressing modes. Some of the techniques used are register-indirect mode, modulo-mode for implementing circular buffer, and bit-reversed addressing used in FFT algorithms.

2.3.2 External Memory Access

Due to their limited on-chip program storage capability, single chip programmable DSPs usually need access to external memory for a given application. Therefore, the speed of the external data transfers becomes important so as not to stall program execution.

In a DSP, memory parallelism is limited to on-chip memory. Off-chip memory accesses commonly share a single program/memory data bus which eliminates the advantage of using a Harvard architecture. However, some DSPs provide multiple off-chip memory ports (separate address and data buses for program and data memory) usually at the cost of a larger chip size.

Off-chip access is greatly enhanced with the use of an integrated direct memory access (DMA) controller that can access internal and external memory. The advantage of using an on-chip DMA controller is that DMA transfers generally occur in parallel with normal program instruction and data transfers which compensate for the slower external memory. However, if a bus contention occurs then wait states are inserted between instruction cycles to account for the DMA transfers. Other external access features of a DSP include memory mapped I/O ports and
serial ports used to interface to CODECs (A/D and D/A devices).

2.3.3 Pipelining

The main emphasis in the design of programmable DSPs lies in the rapid calculation of a large number of multiply/add operations. In addition to using highly parallel memory architectures (see section 2.3.1) DSPs rely on extensive pipelining to realize the level of parallelism demanded by computational intensive signal processing applications.

<table>
<thead>
<tr>
<th>cycle 1</th>
<th>Inst. Fetch</th>
<th>Inst. Decode</th>
<th>Oper Fetch</th>
<th>Inst. Execute</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst #1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cycle 2</th>
<th>Inst. Fetch</th>
<th>Inst. Decode</th>
<th>Oper Fetch</th>
<th>Inst. Execute</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst #2</td>
<td></td>
<td>Inst #1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cycle 3</th>
<th>Inst. Fetch</th>
<th>Inst. Decode</th>
<th>Oper Fetch</th>
<th>Inst. Execute</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst #3</td>
<td>Inst #2</td>
<td>Inst #1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst #4</td>
<td>Inst #3</td>
<td>Inst #2</td>
<td>Inst #1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cycle 5</th>
<th>Inst. Fetch</th>
<th>Inst. Decode</th>
<th>Oper Fetch</th>
<th>Inst. Execute</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst #5</td>
<td>Inst #4</td>
<td>Inst #3</td>
<td>Inst #2</td>
<td>Inst #1</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.4 shows the instruction flow through a five stage pipeline. In this pipeline model each instruction needs five machine cycles to execute and write the result of the operation to memory. The overall throughput of the pipeline is one instruction per cycle provided that each instruction is independent. The advantage of using a pipeline is that fetching and decoding of the next instruction is carried out concurrently with the execution of the present instruction.

DSP pipeline operations can be classified into two categories: hardwired pipelining and programmable pipelining. In hardwired pipelining, the internal mechanism of the pipeline is hidden from the programmer. The advantage of this method is that the programmer is relieved from the need to know the internal workings of the DSP pipeline. In normal pipeline operation the
result of an instruction should be available right after the instruction finishes executing. However, if an exception occurs, the control hardware delays the execution of the instruction until the exception condition is resolved.

The shortcomings of hardwired pipelining is that it is difficult to determine exactly how many cycles an instruction will consume because it depends on the availability of resources and operands. Consequently, program timing becomes more difficult to predict.

The alternative to hardwired pipelining is programmable pipelining. Programmable pipelining gives the programmer explicit control over the pipeline. In addition to providing predictable program timing, programmable pipelining allows the implementation of fast interrupts. An instruction is used to either specify all of the operations that either occur simultaneously in one machine cycle; or that need to be performed on a set of operands from memory.

The disadvantage of using programmable pipelining is that the results of the instruction may not be immediately available in the subsequent instruction. The result is that writing code for that DSP becomes more difficult because the programmers need to be knowledgeable about the target architecture for the application.

2.4 Programmable DSPs

Programmable DSPs, as their names implies, are user programmable processors that require either an external or an internal program memory to store the program’s instructions. Programmable DSPs have several advantage over ASICs. Some of these advantages are:

- flexibility in terms of features to be implement by the user
- software upgradable
• lower development cost

On the other hand, the performance of programmable DSPs is lower than that of ASIC DSPs because they incorporate a multitude of features needed by the wide range of applications that the DSP is targeted for. Note that not all of the features are used in each application. Another disadvantage of the programmable DSPs is the higher cost per system compared to ASIC DSPs. General purpose DSPs usually require additional external hardware such as A/D converters and program memory.

Programmable DSPs normally recourse to extensive pipelining in order to accommodate all or some of the features that characterize signal processing applications such as low overhead looping and multiple memory accesses on a single chip. Though extensive pipelining often translates to a higher programming complexity, DSPs remain more efficient than the easier to use, general purpose microprocessors for DSP algorithms because the code is adapted to make the best use of the processor's architecture.

### 2.5 Application Specific Processors

ASIC DSPs are hardwired implementations of specific DSP algorithms. They are usually optimized for speedup, power consumption, and area constraints. The main advantages of ASIC DSPs are:

• higher execution speed

• high level of on-chip integration, with no need for external circuitry

• low cost-per-chip for high volume applications
Despite all of these advantages, the main drawback of ASIC DSPs is the high development cost. Furthermore, if time to market is a factor then an ASIC cannot be considered as an option because ASIC projects are usually long term projects that extend over several years of development and design. The other disadvantage of using an AISC is that they are not easily upgradable. Once an ASIC has been designed and mass produced it is quite difficult to modify the product.

The recent trend in AISC design is to move toward system partitioning in a way that allows the use of ready-to-use design blocks known as cores. This shortens the design cycle and reduces the development cost. In the future, the development cost may not be a disadvantage of the ASIC DSPs.

2.6 Multimedia Extensions

A recent trend among general purpose microprocessor manufacturers is to integrate DSP functionality into their CPUs in order to speedup signal processing functions. The latest effort revolves around extending the existing ISA (instruction-set-architectures) with Single-Instruction-Multiple-Data (SIMD) like multimedia enhanced instructions such as Intel’s MMX technology and SPARC’s VSI instructions set. The main advantage of using these multimedia enhanced extensions is that they don’t require any modifications to existing ISAs. Other advantages of using multimedia extensions are:

- parallel instruction execution
- reduced program size by compacting several instructions into a single instruction that operates on multiple data in a SIMD like fashion

In order to make use of multimedia extensions the applications have to be reprogrammed
either at the assembler level or by using an optimizing compiler. Because data paths are partitioned between several data operands, multimedia extensions operations have a lower bit-resolution than normal operations that make use of the full width of the data bus to represent their operands. Despite these disadvantages we believe that the advantages of multimedia extensions out-weigh their disadvantages. Table 2.2 summarizes the advantages and disadvantages of the programmable, ASIC, and multimedia extension DSPs.

<table>
<thead>
<tr>
<th>Technology</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programmable DSPs</td>
<td>• user programmable</td>
<td>• higher cost -per-system</td>
</tr>
<tr>
<td></td>
<td>• low development cost</td>
<td>• requires external circuitry</td>
</tr>
<tr>
<td>ASIC DSPs</td>
<td>• higher speed</td>
<td>• fixed number of features</td>
</tr>
<tr>
<td></td>
<td>• higher development cost</td>
<td>• non-upgradable</td>
</tr>
<tr>
<td>Multimedia Extensions</td>
<td>• compatibility with existing ISAs</td>
<td>• compiler dependent</td>
</tr>
<tr>
<td></td>
<td>• parallel instruction execution</td>
<td>• lower resolution</td>
</tr>
</tbody>
</table>

Table 2.2 Comparison of three different DSP technologies
2.7 Proposed Solution

The characteristics of signal processing applications reviewed in section 2.2 can be summarized as follows:

- multiple instructions executed in parallel
- multiple data access in parallel per cycle
- low development cost to keep up with a highly competitive market
- flexibility and upgradability to integrate additional features
- scalability to accommodate future increases in computation requirements

Current DSP architectures lack the resources to accommodate all of these features in a single chip. Therefore, we propose a solution that is based on using a dynamically scheduled superscalar DSP capable of issuing multiple instructions in parallel as well as accessing multiple data operands from memory concurrently. Compatibility with existing ISAs is maintained by using a dynamically scheduled approach which doesn’t require any recompilation of existing software. Scalability is another feature that is guaranteed by using independent functional units that can be added or removed as the application requires. The strength of our solution is that it retains the advantages of both programmable DSPs and ASIC DSPs. Furthermore, because superscalar DSPs pipelines are architecturally adapted to handle parallel instruction issue, they are considered as a prime candidate for implementing multimedia extensions which speedup the execution of multimedia applications even further. Table 2.3 shows a comparison between the superpipelined, VLIW, and superscalar architectures.
<table>
<thead>
<tr>
<th>Architecture</th>
<th>Performance Factors</th>
<th>Advantage</th>
<th>Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>VLIW</td>
<td>Total number of instructions executed</td>
<td>. Reduce the number of instructions</td>
<td>. Heavily dependent on sophisticated compilers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>. Simpler hardware</td>
<td>. Density of instructions compaction depends on the instruction parallelism</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Superscalar</td>
<td>Instruction/Cycle (IPC)</td>
<td>. Reduce the average number of cycles per</td>
<td>. Extra hardware</td>
</tr>
<tr>
<td></td>
<td></td>
<td>instruction</td>
<td>. Performance depend on the amount of resources in the machine</td>
</tr>
<tr>
<td></td>
<td></td>
<td>. Compiler independent</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Superpipelined</td>
<td>Cycle Time (MHz)</td>
<td>. Reduce the processor cycle time</td>
<td>. Longer overall cycle time due to the extra pipeline stages</td>
</tr>
<tr>
<td></td>
<td></td>
<td>. No resources conflicts</td>
<td>. Requires high speed clocks</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.3 Comparison of three different multiple-issue microarchitectures

The main difference between a VLIW machine and a Superscalar machine is that the VLIW machine schedules instructions statically at compile time whereas the Superscalar machine schedules instructions dynamically at runtime. The limitation of static scheduling is that it requires a sophisticated compiler that analyses the program for available ILP and compacts multiple instructions into a single long instruction. However, because compilers don’t have a mechanism by which to predict the outcome of a branch therefore, parallelism extracted by the compiler will be limited to the basic blocks of the program. On the other hand, a Superscalar machine can make use of different hardware techniques to predict branches at runtime based on the previous behavior of branches.
2.8 Conclusion

In the present chapter we investigate the instruction and data bandwidth requirements of typical DSP operations and applications. The analysis of the FIR filter operation shows that instruction parallelism for DSP applications resides primarily within short loops that are highly repetitive. We also conclude that DSP applications and multimedia programs in particular require a high level of data bandwidth in order to accommodate multiple streams of data inputs such as audio and video.

In the evaluation of three different DSP technologies, we find that no single technology by itself incorporates all of the features needed to accelerate the performance of today's multimedia applications. Therefore, in the last part of the chapter we propose a solution that draws from the advantages of both the programmable DSPs and the ASIC DSPs. The proposed solution is based on a dynamically scheduled superscalar DSP. Advantages in using a dynamically scheduled superscalar microarchitecture as a base for our DSP include multiple instructions issue, binary compatibility with existing ISAs, and scalability.
Chapter 3 Architectural Description

3.1 Introduction

In this chapter we present a detailed description of our superscalar DSP along with the simulator developed to study its performance. The superscalar DSP uses aggressive hardware techniques to fetch, decode, and execute multiple-instructions per cycle. These techniques include: dynamic scheduling, out-of-order issue, and out-of-order completion. The superscalar DSP instruction set is based on a subset of the MIPS R3000 instruction set [26]. Choosing a RISC based architecture for the design of the superscalar DSP simplifies the simulation process in the sense that we don’t have to worry about complex addressing modes and decoding schemes as it is the case with a CISC architecture. Also, by basing our design on a commercially available microprocessor, the MIPS R300, we were able to use available C compilers to compile full application benchmarks and simulate their execution on a real machine. Six realworld multimedia benchmarks were simulated under five different superscalar models that represent a wide range of machine parallelism, from a single issue scalar machine to a perfect superscalar machine. These five models of parallelism are used as a basis for comparing the performance of different hardware configurations.

In the first part of this chapter we introduce the superscalar DSP architecture and the different classes of instruction format. A block diagram shows the different hardware components that form the design space of the superscalar processor. In the second part, we describe the different functional units included in the processor along with the different hardware techniques used to implement the dynamic scheduling algorithm and the out-of-order issue with out-of-order completion. In the third part, we describe the simulation environment and the simulation process. In the last part of the chapter, we present the five models of machine parallelism and the six
multimedia programs used for the simulation.

3.2 Architectural Organization

Figure 3.1 shows a block diagram of the model superscalar digital signal processor. The superscalar DSP is a multiple issue machine that schedules instructions dynamically at run time. The block diagram is sub-divided into six stages that represent the flow of data through the DSP. These stages are: instruction fetch, decode and register renaming, instruction issue, execute, result write back, and finally the result commit stage where results are written to the register file and/or data memory.

3.3 Instruction Set

The superscalar DSP has RISC type instructions that follow the instruction set of the MIPS R3000 [26]. These instructions are single 32-bit words. The three instruction formats are immediate, jump, and register (refer to Figure 3.2).

Furthermore, the instruction set is divided into three main classes: the load and store class, the computational class, and the control flow class. There is also the special instruction class that groups miscellaneous tasks like the BREAK instruction, exception handling, system call traps, and moving data to and from the special purpose registers HI and LO. Because multimedia applications are predominately integer based, only a subset of the floating point instructions were incorporated into the simulation process. However, full support for floating point instructions can be easily added if needed.
Figure 3.1 Block diagram of the Superscalar Digital Signal Processor
### Chapter 3 Architectural Description

#### Immediate

<table>
<thead>
<tr>
<th>6-bit</th>
<th>5-bit</th>
<th>5-bit</th>
<th>16-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>src</td>
<td>dest</td>
<td>immediate</td>
</tr>
</tbody>
</table>

#### Jump

<table>
<thead>
<tr>
<th>6-bit</th>
<th>26-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>target</td>
</tr>
</tbody>
</table>

#### Register

<table>
<thead>
<tr>
<th>6-bit</th>
<th>5-bit</th>
<th>5-bit</th>
<th>5-bit</th>
<th>5-bit</th>
<th>6-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>src1</td>
<td>src2</td>
<td>dest</td>
<td>shift</td>
<td>func</td>
</tr>
</tbody>
</table>

*Figure 3.2 Instruction-level format*

#### 3.3.1 Load and Store Instructions

The load and store instructions are immediate instructions that move data between memory and a set of 32 general purpose registers. The only addressing mode supported by the load and store instructions is the register + immediate mode. The load/store address is computed by adding the base address from the register and an immediate offset coded in the instruction.

#### 3.3.2 Computational Instructions

Computational instructions can be either register based or immediate. Computational instructions perform the following operations on register values: arithmetic, logical, shift, multiply, and divide. These operations fit to the following four categories of computational instructions: ALU immediate instructions, three-operand register-type instructions, shift instructions, and multiply and divide instructions.
3.3.3 Jump and Branch Instructions

Jump and branch instructions change the control flow of a program. Subroutines calls, returns, dispatches, and large cross-page jumps make use of the jump instruction in both immediate and register format. The target addresses for all branch instructions are computed by adding the address of the instruction to the 16-bit offset. Branches can be conditional or unconditional.

3.4 Pipeline Stages

The basic pipeline of a superscalar machine that implements out-of-order issue with out-of-order completion is composed of five partially independent stages. These stages are: Fetch, Decode, Execute, Write Back, and Result Commit. Figure 3.3 shows the pipeline stages of a 2-way superscalar processor. During the Fetch stage instructions are read from the code cache or the instruction prefetch buffer if available, in a group of multiple instructions. The size of fetch group depends on the machine’s fetch width. In the following stage, instructions are decoded and checked for dependencies. If an instruction has a dependency on one or more instructions that are being executed or that haven’t been issued yet, then the decoder tries to resolve the dependency by renaming the register in conflict. If a dependency cannot be resolved, then the instruction in question will be delayed until the result needed is made available. Once instructions are decoded they are placed into the Instruction Window where they await execution. During the execute stage, instructions that are cleared of dependencies are issued from the Instruction Window in any order, to the corresponding functional unit. The number of instructions that can be issued simultaneously in the same cycle depends on the number of functional units available in the machine. After an instruction finishes execution, its result is forwarded to the waiting instructions that depend on it. During the final stage, results are committed to the register file or data cache in program order.
3.4.1 Fetch Stage

In the first stage instructions are fetched from the instruction cache or the instruction prefetch buffer in blocks of N instructions, where N represents the size of the cache line. If a branch is encountered, the outcome of the branch will be predicted using a branch prediction scheme that relies on the past behavior of the branch. Next, the program counter will be updated and the fetcher will resume fetching at the new program location.

3.4.2 Decode Stage

Instructions are decoded in blocks of N instructions at a time. As instructions are decoded, they are being checked for dependencies. Each instruction that produces a result is assigned an entry in the Reorder Buffer to resolve any dependencies that it might have with previous or following instructions. Once instructions are decoded, they are written into the Instruction Window, where they await execution.

3.4.3 Execute Stage

In the execute stage, all instructions that are free of dependencies can be issued for execution provided that there are enough execution units to accommodate all of them. Instructions can be issued from the Instruction Window in two ways: once they become free of all dependen-
cies regardless of their age, or in a FIFO-like manner where the oldest instruction fetched is issued first [25]. Managing the Instruction Window like a FIFO guarantees that instructions will be issued in program order. The other alternative would be equivalent to issuing instructions out of program order. Out-of-order issue has the most performance advantage because instructions no longer have to wait until they reach the bottom of the window before they are issued. Instructions can be issued from any position in the window regardless of their age. However, out-of-order issue also increases the complexity of the issue circuitry. Instead of using a simple FIFO where instructions are issued when they reach the bottom of the window, now the Instruction Window is scanned from head to tail to find all of the instructions that are ready to be issued. Also, once the instructions are issued, instruction holes need to be eliminated from the Instruction Window. Figure 3.4 shows the effects of using an out-of-order issue in the Instruction Window.

![Figure 3.4 Instruction Window of an out-of-order issue processor](image-url)
3.4.4 Write Back Stage

In the Write Back stage, the processor identifies the completed operations in the Reorder Buffer and frees the corresponding functional unit. Completed results are validated and forwarded to the instructions that need them in the Instruction Window. Branch execution results are forwarded to the branch prediction unit where they are checked for prediction accuracy. If a branch was mispredicted, then all of the instructions following the branch in the Reorder Buffer are flushed and the branch history is updated with the new information.

3.4.5 Result Commit Stage

In the final stage of the pipeline, results are committed to the register file or data cache in program order to preserve program integrity. Register writes are processed in order from the top to the bottom of the reorder buffer. Committed instructions are then removed from the reorder buffer and results are sent to the store buffer to update the data cache and the register file.

3.5 Architectural Units

3.5.1 Fetch Unit

The fetch unit is responsible for fetching instructions from the instruction cache and predicting branches.

3.5.1.1 Fetch width

Instructions are fetched, independent of their alignment, from the cache into the decoder in an N-instruction block size. The number of instructions that can be fetched in parallel depends on the size of the cache line. A typical instruction-block size is 4 instructions wide. However, wider instructions-block sizes can be used depending on the number of pins available on the chip.
3.5.1.2 Branch Prediction

Branch prediction is a way by which the processor guesses the outcome of a branch before it is executed. To support branch prediction a branch prediction buffer (BPB) is used to record the previous behavior of branch instructions encountered during the execution of the program [16]. The first time a branch is encountered, a new location in the BPB is assigned to the new branch. After that, everytime the same branch is taken the information that corresponds to it will be updated in the BPB. The superscalar DSP uses a branch prediction scheme that relies on a 2-bit non-overflow counter to foretell the outcome of a branch instruction once it's fetched. Every time that the branch predictor correctly predicts the branch the counter is incremented until its reaches its maximum binary value of “11”. If the counter is at its maximum then it is not incremented. If the branch was mispredicted then the counter is decremented. When the counter reaches its minimum binary value of “00” it is not decremented any further. The counter binary values “10” and “11” means that the branch is taken while the values “01” and “00” means that the branch is not taken.

Each entry in the prediction buffer contains the information needed to restore the processor’s state in case the branch was incorrectly predicted. When the processor commits a block of result to the register file, branch predictor statistics are updated depending on the outcome of the branch prediction in that block. If a branch was correctly predicted then the branch history counter output will be increased by one. On the other hand, if the branch was incorrectly predicted than the counter count will be decreased by one. Entries in the prediction buffer are indexed by the lower address bits of the branch instruction.

3.5.2 Decode Unit

The decode unit is capable of decoding multiple instructions in parallel. It incorporates a
Reorder Buffer and an Instruction Window that support dynamic instructions scheduling. Figure 3.5 shows the layout of a 5-instructions decoder. Each position in the decoder that holds an instruction is referred to as a "decoder slot".

<table>
<thead>
<tr>
<th>1st decode position</th>
<th>2nd decode position</th>
<th>3rd decode position</th>
<th>4th decode position</th>
<th>5th/last decode position</th>
</tr>
</thead>
<tbody>
<tr>
<td>INST #N</td>
<td>INST #N+1</td>
<td>INST #N+2</td>
<td>INST #N+3</td>
<td>INST #N+4</td>
</tr>
</tbody>
</table>

Figure 3.5 Layout of the decoder

In a machine with perfect branch prediction (e.g. all branches are predicted correctly) all the decoder slots are filled with valid instructions. However, the presence of branches within the program code interrupts the normal flow of instructions execution and introduces wasted decoder slots (refer to Figure 3.6). Wasted decoder slots occur when:

- a branch instruction is not at the last decode position
- the target of a branch instruction is not at the first decode position

In order to fill the wasted decoder slots that occur because a branch instruction is not at the last decode position, instructions from different basic blocks can be merged together. This technique is called "Instruction Merging" [8].

Figure 3.6 Layout of a decoder that contains wasted decoder slots
Similarly, in order to get rid of the wasted decoder slots that occur because the target instruction of a branch does not fall into the first decode position, all branch target instructions are aligned to the instruction block boundary so that the target instruction falls every time into the first decode position.

3.5.2.1 Reorder Buffer

A Reorder Buffer has two main functions in a dynamically scheduled superscalar architecture: it allows the safe recovery from mispredicted branch instructions, and serves as a register renaming mechanism that helps to resolve storage dependencies [25]. The Reorder Buffer is implemented using a circular queue. The head of the queue points to the oldest instruction in the buffer, while the tail refers to the newest instruction. New entries are added to the tail of the queue. A new Reorder Buffer location is assigned for each new instruction that produces a result. An arbitrary temporary tag is created and appended to the newly assigned Reorder Buffer entry to identify the result. A Flush bit is used to invalidate instructions in case a branch is mispredicted. A Valid bit is used to indicate that the result stored at that location can be used by any other instruction(s) being decoded or waiting for execution.

<table>
<thead>
<tr>
<th>tag #</th>
<th>Register #</th>
<th>Opcode</th>
<th>Class</th>
<th>Result</th>
<th>Valid</th>
<th>Ready</th>
<th>IC</th>
<th>Flush</th>
</tr>
</thead>
<tbody>
<tr>
<td>Head</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tail</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.7 Layout of a Reorder Buffer
Figure 3.7 shows the layout of a Reorder Buffer. Each entry in the Reorder Buffer contains 8 fields:

- **Tag number**: result identification.

- **Register number**: result register number used to search for an operand in the reorder buffer and indicates which register to write to when results are committed to the register file.

- **Opcode**: instruction’s operation code.

- **Class**: instruction’s class.

- **Result**: computed result of the instruction.

- **Valid**: validity bit of the result indicates if the operation has been completed.

- **Ready**: indicates at what clock cycle the computation of the result is going to be completed.

- **IC**: the dynamic instruction count calculated when the instruction was fetched. Used to identify the most recent copy of the same register.

- **Flush**: the flush bit. Set to 1 when the instruction follows a wrong branch prediction.

### 3.5.2.2 Instruction Window

The primary function of the Instruction Window is to allow instructions to executed out-of-program order [8]. After being decoded the instructions are stored in the Instruction Window and wait for execution. Instructions that are freed from all dependencies are issued to the corresponding functional unit. The number of instructions that can be issued in a given cycle is
only limited by the number of functional units available in the microprocessor and by the interdependency that exists between the instructions themselves. The Instruction Window determines the execution order based on the availability of the corresponding execution units. When the resources become available the waiting instruction is released to the appropriate execution unit. The issue algorithm is based on the oldest first algorithm as previously explained. The alternative to using a single Instruction Window is to use a separate Instruction Window called Reservation Stations for each functional unit [23]. However, this scheme is proves to be more complex because each of these Reservation Stations requires it's own issue circuitry. Also, it was shown in [8] that use of Reservation Stations leads to larger buffer sizes than we would normally need if we just use a single Instruction Window.

3.5.3 Execution Unit

The basic processor model shown in Figure 3.1 incorporates seven types of functional units. These seven types are described below. A number of command line options are used to pass to the simulator the number of functional units of each type every simulation run.

3.5.3.1 ALU

The arithmetic and logic unit is responsible for carrying out integer operations like multiplication, division, addition, subtraction, and bitwise logical operations. Integer ALU operations, with the exception of the multiply and divide operations, execute with a 1-cycle latency and 1-cycle repeat rate.

3.5.3.2 Branch Unit

The branch unit executes one branch instruction per cycle. A branch bit is appended to each branch instruction during the instruction fetch. These bits are used to locate branch instruc-
tions in the issue pipeline and forward them to the branch unit. The branch unit also contains a program trace RAM that stores the program counter for each branch instruction in the pipeline. Execution results of the branch instruction are forwarded to the branch prediction unit to authenticate the correctness of the branch prediction.

3.5.3.3 Load & Store Units

The superscalar DSP uses a separate load and store unit that incorporates an address calculation unit to accelerate data retrieval. The load and store unit interfaces to a data buffer that serves as a temporary storage area before results are committed to the data cache. The other advantage of using a data buffer is that it allows the load address calculations to be decoupled from data accesses. This does not limit the number of loads that are sent to the cache on one clock cycle. Load operations are given priority over store operations to use the data cache interface and they are allowed to bypass store operations that are already waiting in the store buffer. In that case the data are read from the data buffer. However, store instructions are not allowed to bypass load instructions because of the risk that the store instruction might overwrite data that are needed by the load instruction.

3.5.3.4 Multiplier & Adder Units

A fast integer multiplier cascaded with an integer adder is used to speed up the execution of multiply-accumulate (MACC) operations. The multiplier and the adder can also be used separately to execute integer multiplication and addition in parallel. The latency of a single MACC operation is two cycles. However, because the MACC operation is pipelined we can obtain a throughput of one MACC operation per cycle provided that multiple MACC operations are issued back-to-back.
3.6 Simulator Description

In this project, we use a simulation method based on the execution trace\(^1\) of the critical path of the program. A trace simulation method allows us to analyze the execution of real code and calculates under a given architectural model how much parallelism could potentially be attained by that model. Traces were generated with the MIPS's pixie tracing utility on a MIPS3000 based machine [20]. The advantage of using a pre-generated program trace is that we can simulate machines with unlimited parallelism that portrays conditions at the limit of parallelism and beyond.

Next, a proprietary, parametrizable trace-analysis program reads in the generated trace and schedules instructions into a sequence of pending cycles that represents the execution of the program on one of five models of parallelism. In its default setting the program simulates a perfect machine with unlimited parallelism. However, individual features, such as the number of registers and the size of Branch Prediction Buffer, can be modified by passing the required options on the command line of the simulator. The simulator incorporates such features as register renaming, alias analysis, and branch prediction.

---

\(^1\) A program trace consists of a stream of operations representing the actual sequence of executed instructions.
Figure 3.8 Block diagram of the simulation process
Figure 3.8 shows a block diagram of the simulation process. The tracing analysis program uses as one of its inputs a Machine Configuration file that contains the: decode width, size of the branch prediction buffer, and the size of the reorder buffer. The tracing program itself consist of a number of data structures that represent the internal functions of a superscalar processor. The following list briefly describes the different parts of the simulator:

- **Instruction Table**: defines the processor’s instruction set.
- **Machine Organization Table**: describes the available resources. This table is initialized by data read from the Machine Configuration file at the start of the simulation.
- **Instruction Fetch Queue**: a FIFO queue, where elements are added and removed in a fixed block size which depends on the superscalar level of the machine.
- **Reorder Buffer**: a circular queue with a head and a tail used for register renaming.
- **Branch Target Buffer**: a table indexed by the lower bits of branch addresses, given by the address modulo of the size of the table.
- **Instruction Window**: implemented as a compressible stack that keeps instructions in order. Decoded instructions are placed at the top of the stack, but they can be issued from any position within the stack.
- **Functional Units**: a table is used to represent the busy functional units. Each element in the table describes a functional unit, its operational latency, multiplicity, and the number of functional units currently used.
- **Register File**: implemented as a table, accessed by the register number.
- **Memory**: organized as a table where a memory location is indexed by its address.
3.7 Benchmarks & Models of Parallelism

The five models of parallelism used in our instruction-level-parallelism study are described in Table 3.1. These five models represent a wide range of machine parallelism, from a single issue scalar machine model with no register renaming or branch prediction, to a perfect machine model with unlimited parallelism. Other models include the Basic model with a 128-entry BPB and 64 registers. The Average model has a larger BPB and more registers. The Good model has a 2048-entry BPB and 512 integer registers. All five models implement perfect alias analysis where all memory conflicts are resolved successfully. Loads that bypass stores and stores that bypass loads\(^2\) are both implemented and their conflicts resolved without any penalty.

<table>
<thead>
<tr>
<th>Model</th>
<th>Issue Width</th>
<th>Branch Prediction</th>
<th>Register Renaming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bare</td>
<td>single</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Basic</td>
<td>unlimited</td>
<td>128</td>
<td>64</td>
</tr>
<tr>
<td>Average</td>
<td>unlimited</td>
<td>512</td>
<td>128</td>
</tr>
<tr>
<td>Good</td>
<td>unlimited</td>
<td>2048</td>
<td>512</td>
</tr>
<tr>
<td>Perfect</td>
<td>unlimited</td>
<td>perfect</td>
<td>unlimited</td>
</tr>
</tbody>
</table>

Table 3.1 hardware parallelism models

\(^2\) Store operations that bypass load operations make use of a store buffer that temporarily hold the operand to store until all dependent load operations are resolved.
The trace-analysis program generates a series of run-time statistics (refer to Figure 3.8) that compare the performance of the five different models in terms of available parallelism in each benchmark applications. Parallelism measurements are represented by the average number of instructions executed per cycle (IPC) obtained by dividing the total number of instructions executed by the number of cycles scheduled.

Table 3.2 shows the ten multimedia programs used as benchmarks together with details about the number of instructions that each program took to execute and the distribution of the instruction mix in each of these programs. The benchmarks were compiled with the native MIPS C compiler using the -O2 option to eliminate unneeded and redundant instructions that usually increases the instruction parallelism in the program [5]. All ten benchmarks simulated to date were executed to completion totaling hundreds of simulation hours on a MIPS workstation and several SUN workstations. These benchmarks were all drawn from public domain C programs for ease of reproduction and result verification. Appendix A gives more details about the characteristics of each benchmark and the FTP site where the benchmark can be downloaded from or the email contact from which to request a copy of the benchmark. The last three columns in Table 3.2 show the distribution of instruction mix within each of the programs. The integer and the memory access instructions are the highest frequency type of instructions. Control transfer instructions composed of branch and jump instructions are less frequent than the integer and load instructions.

All of the ten benchmarks were used in the Instruction-level-parallelism study in chapter 4. However, for the machine parallelism study presented in chapter 5 we only used the first 6 benchmarks in order to keep simulation time down and allow more time for results analysis and presentation.
3.8 Conclusion

In this chapter we presented an architectural overview of the proposed superscalar DSP and showed all the different features included. We also reviewed the simulation environment and the trace generation techniques used to profile the benchmarks studied. Finally, we presented the five hardware models of parallelism and the ten multimedia benchmarks used in our study.
Chapter 4 Instruction-Level Parallelism

4.1 Introduction

The primary function of superscalar processors is to achieve high speedup by issuing multiple instructions in parallel every cycle. In theory, an N degree superscalar processor, where N indicates the maximum number of instructions that the processor is capable of issuing in parallel, is able to execute a given program N times faster than a scalar processor. Practically, there are software and hardware limitations that prevent achieving this theoretical maxima.

Software limitations are characterized by the dependencies between instructions in the code for an application. The more dependencies that exists in a program, the fewer instructions that the processor is able to extract and issue in parallel. Similarly, hardware limitations prevent applications from achieving their potential speedup because of limited hardware resources. It follows that the performance in terms of the number of instructions executed per cycle for a superscalar processor is not only limited by the maximum amount of instruction parallelism available in the application, but also by the amount of resources available on-chip. These on-chip resources represent machine parallelism.

In addition to the software and hardware limitations, the nonuniformity in the distribution of ILP also affects performance. Nonuniformity in the distribution of ILP occurs when available parallelism is concentrated in a short burst of massive parallelism. This contribute to increased inefficiency in resources utilization.

In the first part of the chapter software limitations including data, storage and control dependencies are studied. In the second part, we introduce the hardware cycle width and the
effects of operation latency on ILP. Nonuniformity in the distribution of ILP is reviewed in the last part of the chapter.

4.2 Software Limitations

Software limitations are the result of the dependencies that exist between different instructions of a program. There are two types of software dependencies: data and control dependencies [2, 8].

4.2.1 Data Dependencies

The first requirement to achieve a high level of ILP is to understand, for each instruction, which other instruction it actually is dependent upon. With every instruction that a processor issues, it must first check whether the instruction's operands interfere with the operands of any other instruction that is either being executed or is waiting to be issued. If the instruction depends upon a previous instructions then it is delayed until the instructions it depends upon finishes execution and the result is forwarded to it. Once the instruction is ready to be issued (freed from all dependencies), the processor must decide when and on which available Functional Unit (FU) to execute that instruction. Sustaining a high rate of instruction execution thus requires a high rate of operand transfer between separate instructions and the right mix of functional units.

The two types of data dependencies are true dependencies and storage dependencies. True dependency, also known as write-after-read dependency, results from an instruction using the outcome of a previous instruction as one of its operands. In that case, the execution of the second instruction must be delayed until the first instruction finishes executing and the result is produced. True data dependencies give rise to "zero-issue" cycles in which no new instructions are executed.
because the corresponding FU is still in the process of executing a long operation or waiting for the result bus to be freed. True data dependencies also give rise to "single-issue" cycles, where only one instruction is issued because all subsequent instructions depend on this single instruction [2, 3, 22]. The significance of a true dependency is that it cannot be removed even in an ideal case (a processor with unlimited resources). True dependency proves to set the upper bound on the limit of ILP in an application. In order to estimate the maximum ILP available in a program we need to consider instruction parallelism that is only limited by true data dependencies. Figure 4.1 shows the parallelism available under the five models with unlimited issue width and perfect caches. The figures obtained here are theoretical and are only intended to be used to estimate the limits of parallelism for these applications. These limits can be used later to decide when to stop increasing the machine parallelism to obtain higher performance. Any increase in machine parallelism beyond that level will decrease performance as there are more cycles in which the processor is waiting idle for instructions to execute.

Figure 4.1 Parallelism under the five models
Table 4.1 shows the percentage of decrease in performance for the five models. The MPEG and H.261 encode benchmarks show the highest amount of ILP, 700 and 620, under the perfect model. Whereas the parallelism for the Good model decreased from 5 to 10%, the highest decrease for the Basic model was approximately 85% incurred by the MPEG encode program. The performance of the smallest program, GSM uncompress, decreased by 40%. As expected, the results for the Bare model are the lowest because of the absence of any machine parallelism. Limiting the issue width of the Bare machine to one had the most damaging effect on performance because it set the upper bound on instruction-issue to one. If we increase other hardware resources on the Bare model, like number of functional units or number of registers, we would not get any performance improvements because the processor is limited to issuing a single instruction per cycle. This proves that issue width has the highest degree of impact on performance in terms of instruction issued per cycle. Increasing the issue will increased performance considerably for the Bare model as results for the other four model show.

<table>
<thead>
<tr>
<th></th>
<th>JPEG encode</th>
<th>JPEG decode</th>
<th>MPE encode</th>
<th>MPE decode</th>
<th>H.261 encode</th>
<th>H.261 decode</th>
<th>H.263 encode</th>
<th>H.263 decode</th>
<th>GSM compress</th>
<th>GSM uncomp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perfect</td>
<td>64.5 IPC</td>
<td>138.2 IPC</td>
<td>274.6 IPC</td>
<td>703.5 IPC</td>
<td>258.4 IPC</td>
<td>617.8 IPC</td>
<td>150 IPC</td>
<td>225 IPC</td>
<td>78 IPC</td>
<td>32 IPC</td>
</tr>
<tr>
<td>Good</td>
<td>-5.6%</td>
<td>-5.8%</td>
<td>-5.2%</td>
<td>-5.4%</td>
<td>-4.9%</td>
<td>-5.1%</td>
<td>-5.5%</td>
<td>-5.7%</td>
<td>-5.3%</td>
<td>-5.2%</td>
</tr>
<tr>
<td>Average</td>
<td>-25.4%</td>
<td>-25.1%</td>
<td>-34.9%</td>
<td>-55%</td>
<td>-35%</td>
<td>-52%</td>
<td>-26.3%</td>
<td>-25.5%</td>
<td>-22.1%</td>
<td>-21.7%</td>
</tr>
<tr>
<td>Basic</td>
<td>-45.3%</td>
<td>-44.9%</td>
<td>-65%</td>
<td>-85%</td>
<td>-60%</td>
<td>-83.4%</td>
<td>-46.2%</td>
<td>-45.4%</td>
<td>-41.9%</td>
<td>-40.2%</td>
</tr>
<tr>
<td>Bare</td>
<td>-98%</td>
<td>-98.9%</td>
<td>-99.4%</td>
<td>-99.7%</td>
<td>-99.4%</td>
<td>-99.6%</td>
<td>-99%</td>
<td>-98%</td>
<td>-97%</td>
<td>-98%</td>
</tr>
</tbody>
</table>

Table 4.1 Percentage of decreasing parallelism for the five models
The second type of data dependency is storage dependency. Storage dependencies also known as false dependencies, are the direct result of the compiler's register allocation technique and can be successfully removed using a software or hardware solution such as register renaming. The effectiveness of register renaming in eliminating storage dependency is a factor of the number of registers available for renaming. Figure 4.2 shows the effect of the number of registers available in the system on performance under the perfect model. Reducing the number of available registers from infinite to 512 has little effect on parallelism for all benchmarks. Reducing the number of registers further to 128 has little effect on the performance of the smaller programs, JPEG and GSM. The drop in parallelism from reducing the number of registers to 64 ranges from 20% for the small programs GSM and JPEG up to 40% for the larger programs. The highest decrease in parallelism for all programs is observed when the number of registers are limited to 32. Under this condition, parallelism of the large MPEG program is decreased by 80%. Large programs such as MPEG and H.261 profited the most when the number of registers was 512. However, for the smaller programs, JPEG and GSM, the ideal number of registers is 128.

![Figure 4.2 Effects of register renaming on instruction-level parallelism](image)


4.2.2 Control Dependencies

A control dependency arises when instructions following a branch instruction cannot be executed until the branch instruction itself is executed and its outcome is known. Without knowing the outcome of a branch before it is executed, parallelism is limited to basic blocks. If the average number of instructions per basic block (instruction-run length) is large enough then control dependencies are not much of a problem. However, the distribution of the instruction-run length within the multimedia benchmarks displayed in Figure 4.3, show that 50% of the total number of instruction-runs are less than 8 instructions long. If we were to limit our search for parallelism to basic blocks then the maximum ILP that we are able to extract from these applications is a factor of the average instruction-run length. If we need to increase the ILP of an application, then we need to expand the scope of the search for instructions to execute beyond the boundaries of basic blocks. One way to accomplish this expanded search is by using "branch prediction". The advantage of branch prediction is that it allows the exploitation of inter-blocks parallelism and therefore contributes to increasing performance. Without branch prediction parallelism is severely limited even for a Perfect model.

Previous work [3, 13, 16, 24] shows that a simple branch prediction mechanism like the one used in our study can achieve a 90% accuracy. Results displayed in Figure 4.4 show a level of accuracy in excess of 90% for our six benchmarks simulated under the Good model. However, branch prediction accuracy seems to level off when the size of the BPB reaches 2048 entries. This size of the BPB is twice the size of the buffer needed to accomplish the same level of accuracy for the set of general purpose UNIX programs used in [8, 10, 24]. Increasing the size of the BPB beyond 2K increased performance only by an additional 2 to 3%. For most programs this improvement is insignificant considering that we quadruple the size of the buffer to achieve it.
The reason that we need such a large buffer for branch prediction is a direct result of the characteristics of signal processing applications explained in section 2.1. Digital signal processing operations are composed of a large number of small repeating loops. Contrary to UNIX programs, the size of loops in multimedia programs are smaller and more repetitive [5], which explains the larger size of the BPB.
Figure 4.4 Branch prediction accuracy in function of branch prediction buffer size

4.3 Hardware Limitations

Hardware limitations result when two or more instructions try to access the same resource at the same time, which creates a resource conflict. The two types of resource conflicts that are reviewed in this section are cycle width and operation latency. Cycle width refers to the maximum number of instructions that can be scheduled in a given cycle. The other factor that generates resource conflict is operation latency. If the latency of an operation increases, then all of the following instructions that depend on that particular instruction will have to wait longer before they can access it’s result.

4.3.1 Cycle Width

Figure 4.5 shows the effects of limiting the cycle width on the parallelism of the Perfect model. Limiting the cycle width to 64 instructions decreases parallelism to approximately 60 IPC
for most benchmarks. The smaller JPEG and GSM benchmark do not suffer as much from the decrease in cycle width because of their limited ILP. Doubling the cycle width to 128 improves the parallelism of the larger program only because of the larger ILP contained in them. Increasing the cycle width from 128 to 256 improves parallelism of the large programs even further. Parallelism of the smaller programs do not improve significantly beyond an issue width of 128. Each time cycle width is increased, the parallelism in the benchmarks increases proportionally. Because of the variation in the ILP for each of the benchmarks, the optimal cycle width is different from one benchmark to another. MPEG encode is the only benchmark that profits considerably from increasing the cycle width from 512 to 1024, because of its unusually high ILP. Increasing the cycle width beyond 1024 does not enhance the parallelism of any of the programs.

Figure 4.5 Effects of cycle width on parallelism
4.3.2 Operation Latency

Operation latency is of interest because of the effects it has on true data dependency. Increasing the operation latency increases the effects of zero-issue and single-issue cycles because subsequent dependent instructions must wait longer before they can be issued. If the number of true-data dependencies in the program are significant then increasing the instruction execution time introduces considerable delay in the pipeline. On the other hand, if the number of true-data dependencies is small, then most of the delay originates from individual instructions.

In all our previous simulations we assumed single cycle latency operations. However, in real machine instructions latency can vary from a single cycle to as high as 20 cycles for some types of multiplication and division operations. Floating point operations have even higher operation latency. Table 4.2 compares the latency of several commercial microprocessors.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Integer</th>
<th>Integer (mul/div)</th>
<th>Float</th>
<th>Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIPS R10000</td>
<td>1</td>
<td>5 - 11</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>PowerPC 620</td>
<td>1</td>
<td>3 - 20</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Alpha 21164</td>
<td>1</td>
<td>8 - 16</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>PentiumPRO</td>
<td>1</td>
<td>NA</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>HP-8500</td>
<td>1</td>
<td>NA</td>
<td>2</td>
<td>NA</td>
</tr>
</tbody>
</table>

Table 4.2 Instructions latency
4.4 Instruction-Level Parallelism Distribution

A major concern in the design of a microprocessor is to balance the number of resources on the chip and the instruction parallelism available in the application in order to create an "efficient architecture" or an architecture with a high utilization of resources. However, if the ILP in the application is concentrated in short bursts of massive parallelism separated by long periods of low parallelism then the machine will need many more functional units than the average to achieve the theoretical parallelism limits. Furthermore, the utilization of these functional units is poor during the long periods of low parallelism. In order to better understand the effects of the distribution of ILP in the benchmarks simulated we need to know if the ILP is "uniformly distributed" within these benchmarks. A uniform distribution of ILP means that parallelism is spread equally among the majority of instruction cycles of the program. On the other hand, a nonuniform distribution of ILP corresponds to the case described above where parallelism is concentrated in short bursts of massive parallelism separated by long sequential periods.

In [9, 12, 21] several nonuniformities in the distribution of instruction-level and machine parallelism are studied. Results show a degradation in performance of almost 30% due to ILP not being evenly distributed within the program code. The decrease in performance seems to be more of a direct consequence to limiting the search for parallelism to intra-blocks rather than a result of the nonuniformity in the distribution of ILP. On the other hand, work presented in [21] shows that 50% of the ILP is evenly distributed within the set of general purpose UNIX programs that they have simulated. In the present section we study the nonuniformity in the distribution of ILP within multimedia applications.

In order to measure the nonuniformity in the distribution of ILP we classify scheduled
program cycles according to the number of instructions issued in each cycle. Figure 4.6 shows the results obtained for the ten benchmarks. Whereas 60% of the total cycles executed are evenly spread among the 1- to 12-instruction cycles, the remaining 40% of the cycles were dispersed without any distinguishable pattern between the benchmarks. These preliminary results seem to agree with the findings in [7] that conclude that 50% of the ILP are evenly distributed within the SPEC benchmarks simulated. These findings suggest a high degree of resource utilization which enhances the balance between software and hardware.

It is interesting to note that while ILP is evenly distributed within the benchmarks it is highly nonuniform between benchmarks. Inter-benchmark parallelism fluctuates between a low of 32 IPC for the GSM uncompress program and a high of 703 IPC for the MPEG encode program. Therefore, a machine optimized for one might not work efficiently for the other.
Chapter 4 Instruction-Level Parallelism

4.5 Conclusion

In this chapter, we have presented a study of the instruction-level-parallelism in multimedia applications. We have concentrated our analysis on exposing the maximum potential parallelism inherent in this special class of applications. Simulation results have shown a high level of instruction parallelism, that varied between 32 to 700 instructions per cycle under the ideal model. The ideal number of registers needed to effectively eliminate storage dependencies for the large MPEG, H.261, and H.263 programs has been found to be around 512 compared to 128 for the smaller JPEG and GSM programs. However, using only 128 registers for the large programs
Chapter 4 Instruction-Level Parallelism

decreases parallelism considerably.

Furthermore, parallelism suffers the most when no branch prediction is used. Using a simple branch predictor based on a 2-bit counter shows a high level of prediction accuracy in excess of 90% for all benchmarks simulated. Probing for different sizes of Branch Prediction Buffer have shown that performance levels off after the 2K mark for most programs. Increasing the buffer beyond 2K has an insignificant impact on performance considering that we have to quadruple the size of the buffer to get a mere 3% increase in parallelism.

The ideal cycle width varies from one benchmark to another depending on the amount of instruction-level parallelism available in that particular benchmark. For the small benchmarks parallelism levels off after a cycle width of 128 instructions-per-cycle. Only MPEG and H.261 encode benefit from an increase in the cycle width beyond 128 because of the high ILP contained in their code.

In the last section of this paper we have studied the nonuniformity in the distribution of instruction-level-parallelism. Our simulation results reveal that 60% of the total number of instructions for all of the benchmarks were evenly distributed. In 60% of the total number of program cycles, 1 to 12 instructions are issued. The concentration of cycles are around the 4 to 5 instructions cycles. These findings suggest a high level of uniformity in the distribution of ILP for multimedia applications. A uniform distribution in ILP suggests a high level in resource utilization which in turn leads to high efficiency.
Chapter 5  Machine-Level Parallelism

5.1 Introduction

Machine parallelism is a measure of the ability of the processor to take advantage of the instruction-level parallelism available in an application. The level of machine parallelism can be determined by the number of instructions that can be fetched and executed at the same time and by the speed and sophistication of the mechanism that the processor uses to find independent instructions [22].

Because the amount of ILP varies from one application to another, it is difficult to design a superscalar machine with the appropriate amount of machine parallelism to satisfy a wide range of applications. For example, previous work shows that the ILP of general purpose user applications is limited to 60 instructions-per-cycle [5, 21]. On the other hand, the multimedia class of application that we are investigating in this work have a higher ILP (refer to section 3.1). Therefore, a superscalar machine that is tailored for multimedia applications may not perform well running general purpose applications because of wasted cycles in which the processor is idle waiting for instructions to execute. Similarly, a superscalar machine that performs well for general purpose applications may perform very poorly when running multimedia applications because there is not enough machine parallelism to exploit all the ILP available in the application.

There are three techniques that can be used to improve machine parallelism [8]. These techniques are:

- out-of-order issue
- register renaming
- branch prediction
In this chapter we evaluate the effects of each of these techniques on the performance of our proposed superscalar processor. In the first section, we examine issue policies and compare three different issue policies. In the second section, we look at register renaming and determine how effective register renaming is in removing storage dependencies. Finally, we investigate issues in resource duplication and the link between ILP and machine parallelism.

5.2 Issue Policy

An instruction issue policy is the mechanism used to determine the order in which instructions are allowed to execute and complete. There are three instruction issue schemes [25]:

- in-order issue with in-order completion
- in-order issue with out-of-order completion
- out-of-order issue with out-of-order completion

Out-of-order issue and out-of-order completion are the two most widely used techniques in dynamically scheduled processors because of the considerable advantage they have over the other two other issue policies. Figure 5.1 compares the speedup in terms of the number of instructions executed per cycle for the three issue policies.

Out-of-order completion permits short latency operations to execute and partially complete ahead of longer latency operations without compromising the integrity of instruction execution order. This prevents unnecessary stalls that result from long latency operations blocking the pipeline. A Completion Buffer, also referred to as a Reorder Buffer (RB), is used to hold instructions emerging from the execution pipeline. When an instruction with a short operation finishes execution before other long operation instructions then the result of the short operation will be temporarily stored in the RB until all preceding instructions have completed and their results are
written back to the file register in program order. Then the instruction in question can be flushed from the RB and its result committed to the file register. The buffer holds a record for all of the completed instructions that are waiting for their predecessors to finish. Once an instruction is committed to the file register or stored in memory, then the entry associated with that instruction in the Reorder Buffer can be freed and its memory location used for a subsequent instruction. Managing the RB as a circular queue (see section 3.5) guarantees that instructions are committed to the file register in program order.

Figure 5.1 Speedup for different Issue policies
Out-of-order issue minimizes the effects of branches in the application code and eliminates pipeline stalls that result from dependencies between instructions in the program. Each time a branch instruction is encountered the processor has to stop fetching new instructions and wait for the branch to be resolved. If the average size of instructions runs (size of basic blocks) in the application is quite small, then the processor will be stalled more often and there won’t be enough ILP for the processor to exploit. Instead of waiting for branches to be resolved the processor can guess the outcome of a branch, fetch instructions far ahead in the execution stream, and execute them before they are even needed. This technique is called “speculative execution”. In the case where the processor guesses incorrectly, the instructions in the completion buffer are flushed, their registers stripped of their temporary names, and instructions from other branch path are fetched. If the processor guesses correctly the registers are renamed, recording the results of the instructions, and the instructions are flushed from the buffer.

Because the penalty associated with branch misprediction is quite high, usually there is a limit on the permissible level of branch prediction permitted in a processor [8, 16]. Table 5.1 compares the branch prediction mechanisms that are used by some of the commercially available microprocessors. A Branch Prediction Buffer (BPB) is used to store the past history of each branch using a 2-bit, non-saturated, up-down counter. When a branch is fetched, the BPB is searched for a match using the lower address bits of the branch instruction. On a BPB hit, the current value of the history counter is checked to determine the outcome of the branch. If that value was found to be “11” taken or “10” likely-taken\(^1\) then the branch is taken, otherwise the branch is not taken. The counter value is decremented by one whenever a branch is mispredicted, and is incremented otherwise. In order to accommodate multi-level branch prediction, each entry in the

---

\(^1\) the other two values used by the branch prediction mechanism are “00” not-taken and “01” likely-not-taken.
BPB is divided into several subentries that hold the history for each level of branch prediction.

<table>
<thead>
<tr>
<th>Vendor</th>
<th>RISC micro</th>
<th>Word Size (bits)</th>
<th>Branch Prediction Buffer Size</th>
<th>Misprediction Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC</td>
<td>21164</td>
<td>64</td>
<td>256-entry</td>
<td>4</td>
</tr>
<tr>
<td>Intel Pentium</td>
<td>32</td>
<td>256-entry</td>
<td></td>
<td>3-4</td>
</tr>
<tr>
<td>Intel Pentium PRO</td>
<td>32</td>
<td>512-entry</td>
<td></td>
<td>5-15</td>
</tr>
<tr>
<td>HP PA-8000</td>
<td>64</td>
<td>256-entry</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>IBM/Motorola</td>
<td>604</td>
<td>32</td>
<td>512-entry</td>
<td>2</td>
</tr>
<tr>
<td>IBM/Motorola</td>
<td>620</td>
<td></td>
<td>2048-entry</td>
<td>4</td>
</tr>
<tr>
<td>MIPS R8000</td>
<td>32</td>
<td>1K cache</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>MIPS R10000</td>
<td>64</td>
<td>512-entry</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>SUN UltraSPARC</td>
<td>64</td>
<td>cache prediction</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>SUN SPARC64</td>
<td>64</td>
<td>1024-entry</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.1 Branch prediction buffer size and misprediction penalty for several commercial microprocessors

Out-of-order issue makes use of a buffer, called an “Instruction Window” between the instruction decoder and the FUs. The instruction issue logic examines the instructions placed in the Instruction Window by the decoder to select the appropriate instructions for issue to the FU. There are two ways to implement the Instruction Window:

- centralized window which consist of a single buffer shared by all functional units
- reservation stations or small size buffers distributed among the different functional units

The advantage of using reservation stations over a centralized window is that the reservation stations need only to be filled at the average instruction execution rate, rather than the peak rate. However, if the Instruction Window is partitioned among the functional units, the total size of the window must be larger to support the same amount of lookahead. In this project, in order to
keep simulation time and complexity to a minimum, we use a centralized window to implement the Instruction Window. The advantage of using an Instruction Window over Reservation Stations is that the size of the Instruction Window is smaller than the total size of all Reservation Station needed to give a comparable performance [8].

Another aspect of out-of-order issue is how instructions are issued from the Instruction Window to the corresponding FUs. There are two different issue protocols:

- oldest first where the instruction window is managed like a FIFO
- fixed priority where instructions are issued on the basis on a predetermined fixed priority and not their age

A fixed priority protocol will requires extensive decode and issue circuitry to the extent that the cost associated with adding the extra hardware outdiminishes the potential performance improvements [8]. Therefore, for the present project we use the simpler, easier to implement “oldest first” protocol. Table 5.2 shows the issue policies that are used by some commercial microprocessors. The last column of Table 5.2 indicates if a particular processor supports dependent integer operations issuing or not. Dependent issue is a mechanism used to allow dependent instructions of the same class to be issued to the same functional unit. The main advantage of having dependent issue is that it reduces the effect of true-dependency by allowing successive dependent operations to be issued in parallel [3]. In all of the microprocessors that we surveyed none of them implement dependent integer issue (refer to Table 5.2). However, we found out that the only three commercial microprocessors that implement dependent issue operations are POWER2, the SuperSPARC, and the 68060. The only limitation on dependent operations issue is that the distance between the two dependent instructions in the program need to be short or otherwise the hardware will not detect it.
Table 5.2 Issue policies and max instruction issue for different commercial microprocessors

<table>
<thead>
<tr>
<th>Vendor</th>
<th>RISC micro</th>
<th>Word Size (bits)</th>
<th>Max Issue</th>
<th>Out-of-order Issue</th>
<th>Dependent Integer Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC</td>
<td>21164</td>
<td>64</td>
<td>4</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Intel</td>
<td>Pentium</td>
<td>32</td>
<td>2</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td>Pentium PRO</td>
<td>32</td>
<td>3</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>HP</td>
<td>PA-8000</td>
<td>64</td>
<td>4</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>IBM/Motorola</td>
<td>604</td>
<td>32</td>
<td>4</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td>620</td>
<td>64</td>
<td>4</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>MIPS</td>
<td>R8000</td>
<td>32</td>
<td>4</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td>R10000</td>
<td>64</td>
<td>5</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>SUN</td>
<td>UltraSPARC</td>
<td>64</td>
<td>4</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td>SPARC64</td>
<td>64</td>
<td>4</td>
<td>YES</td>
<td>NO</td>
</tr>
</tbody>
</table>

5.3 Register Renaming

Storage conflicts can severely limit the amount of exploitable ILP within an application (refer to section 3.1). Techniques such as register renaming can successfully be used to remove anti- and output dependencies and allow more instruction parallelism. Anti-dependencies also known as write-after-read dependencies occur when an instruction tries to write to the same register that is read from by a previous instruction. Executing the second instruction before the first one will cause the first instruction to read the wrong data off from the register. Output-dependencies also known as write-after-write dependencies occur when two successive instructions try to write to the same. Executing the earlier instruction after the later will leave the wrong data in the register.

In a dynamically scheduled superscalar processor, the instruction scheduler allocates a new entry dynamically in the Reorder Buffer (RB) for every new value produced. A hardware generated tag is associated with every new entry in the RB. When a subsequent instruction has a
storage dependency with an incompletely completed instruction in the RB, the decoder reads the source operand’s tag from the RB and stores it in the result field of the “Instruction Window” entry associated with that instruction. Once the result value becomes available it is forwarded directly to the corresponding entry(s) in the RB and the Instruction Window concurrently and the instruction can be issued.

Although register renaming is a useful mechanism it cannot remove all of the storage conflicts within an application because data dependences also occur between memory locations [2]. Dependencies between memory locations are called “aliases” and they occur when two instructions access the same memory location. To prevent all memory “aliasing” problems, store operations are normally not allowed to bypass load operations. However, bypassing load operations is considered a desirable feature because it eliminates instruction stalls due to a true memory dependency where an instruction is waiting for an operand to be fetched from memory. Load operations that bypass store operations need to search the store buffer to probe for true dependency on previous stores before accessing the memory. If the target data was found in the store buffer, then it will be read directly from there. Otherwise the data will have to be fetched from memory. Figure 5.2 shows the effects of output dependency on speedup.
Figure 5.2 Effects of output dependency on performance

5.4 Duplication of Resources

Functional Units and their interconnectivity, the size of the Instruction Window and Reorder Buffer, and the width of the Write and Result bus are some of the resources that compose the design space of a superscalar architecture. The number and types of resources are implementation specific and vary widely from one application to another depending on the amount of ILP available in the application [3, 6, 18]. The other limitation is that it is almost impossible to determine the requirements of an application in terms of number and types of resources necessary because we can’t tell in advance how much ILP an application has. Therefore, we need to simulate each application separately and measure the amount of machine parallelism needed to take full advantage of the available ILP in that application.

In section 3.1 we show the maximum theoretical limit of ILP for each of the benchmarks
under an omniscient superscalar machine model (with unlimited parallelism). However, these maxima are impossible to achieve in real life because of the constraint on the amount of parallelism that can be incorporated in a processor. In order to measure the real limit on performance, we need to evaluate the performance of the same benchmarks under more realistic constraints like a limited number of FUs and a non-perfect branch prediction.

An important requirement in the design of a superscalar processor is to efficiently balance the number of resources with the ILP available in the program’s code. For example, there should be a close match between the number and nature of FUs in the system and the dynamic instructions mix of the applications running on the system. The FUs on a machine must reflect the instruction frequency if we are to achieve efficient utilization of FUs. Figure 3.2 shows the instruction mix distribution for each benchmark. We notice that the highest frequency instructions are the LOAD/STORE instructions, followed by the ALU, SHIFT, and the BRANCH instructions.

An example of the trade-off that is made between system resources and speedup is most obvious in the case of the LOAD/STORE unit. Because LOAD/STORE instructions represent a large percentage of the benchmarks’ total instructions mix adding a second LOAD/STORE unit may significantly increase performance. However, the addition of a second LOAD/STORE unit requires the use of a dual port data cache, one of the most expensive system components and therefore proves to be impractical.
5.4.1 Instruction Unit

The Instruction Unit is responsible for fetching instructions from the code cache and forwarding them to the decode unit. The fetcher must keep a continuous flow of instructions going into the decoder otherwise the decoder stalls and performance decreases. The average number of instructions fetched per cycle depends on two things:

- the size of the fetch block (i.e. the length of the instruction cache line)
- the accuracy of the branch prediction mechanism

In this section we investigate the performance of the Instruction Unit in terms of fetch efficiency and branch prediction accuracy.
5.4.1.1 Instruction Fetch Efficiency

Instruction Fetch Efficiency (IFE) is a measure of the ability of the processor to effectively fill all available "decoder slots" (refer to section 3.5.2) in a given cycle and is represented by the average number of instructions fetched. Fetch Efficiency is measured by dividing the total number of valid instructions executed by the total number of cycles taken to complete the program. Valid instructions are instructions that are executed and their result committed to the file register. Instructions that were executed and later flushed from the pipeline because of a mispredicted branch instruction as well as no-operation instructions are not counted as valid instructions. The Average Instructions Fetched (AIF) mark is used in this study to represent the average number of valid instructions fetched.

If the fetcher is able to fill all the available decoder slots then it has a perfect fetch efficiency (i.e. 100%). However, due to instruction misalignment there is some wasted decoder slots and consequently fetch efficiency is less than perfect. Instruction misalignment occurs because a control transfer instruction (Branch or Jump) is not at the last decode position or the destination of a branch is not at the first decode position (refer to section 3.5.2). Figure 5.4 shows the Average Instruction Fetch for a 2 instruction decoder, a 4 instruction decoder, and an 8 instruction decoder. The AIF count places an upper bound on the speedup of a program, since the processor cannot execute more instructions than it actually decodes.
Results shows that increasing the fetch bandwidth increase the fetch efficiency. As expected the AIF for a decode size of 4 is significantly larger than a decode size of 2. On the other hand, the AIF slightly increases relative to the decode size when the decode size is incremented from 4 to 8. The relative increase in AIF is due to the higher number of wasted decoder slots resulting from instruction misalignment. Therefore, increasing the fetch bandwidth has a marginal return to speedup unless more aggressive branch prediction techniques and instruction aligning and merging are used to increase instruction bandwidth. Table 5.3 shows the increase in the AIF that results from using different level of branch prediction mechanism. Refer to section 5.4.1.2 for details on the branch prediction models used in Table 5.3. Results indicate that the AIF increases as the performance of the branch prediction increases. Performance reached its highest level for an 8-instruction decoder.
Table 5.3 Average Instruction Fetch for different branch prediction technology

Table 5.4 shows the fetch mechanism used by some of the commercial microprocessors. Only three of the microprocessors shown in table 5.4 use fetch alignment due to the added cost in hardware and complexity related to implementing alignment. All of the microprocessors have a fetch and decode width of 4. Increasing the fetch width requires adding expensive I/O pins which consequently increases system cost.

Table 5.4 Fetch mechanism for different commercial microprocessors
5.4.1.2 Branch Prediction

Branch prediction affects the performance of the dynamically scheduled processor primarily by its impact on fetch efficiency[13]. The better the branch predictor, the lower the probability that instructions are incorrectly fetched and the higher the fetch utility. Furthermore, branches or control transfer operations in general cause instructions to be misaligned in the decoder. There are techniques that can be used to align instructions to the first decode position and also to merge instructions from different instruction runs in order to fill the wasted decoder slots that result from the presence of branches in the execution trace of a program. However, these techniques have an extra hardware cost associated with them which decreases the advantage incurred from implementing them. For example, if we want to implement instruction merging we need to fetch two blocks of instructions concurrently which requires a duplicate set of the fetch unit and the width instruction cache line must be doubled to fetch twice the number of instructions from the cache.

The accuracy of the branch prediction mechanism used in this work is approximately 95% using a 2048 instruction Branch Prediction Buffer. Figure 5.5 shows the branch prediction accuracy for different sizes of the Branch Prediction Buffer. Results clearly show that a 2048 instructions BTB has the highest branch accuracy of all lower size buffers. Increasing the size of the BTB beyond that has a marginal effect on performance as accuracy levels off at the 4096 instruction deep buffer. The size of the BTB needed for multimedia applications is almost twice the size of the BTB that is used for simulating UNIX applications in [3, 8, 13, 16]. That's because large multimedia applications have a higher number of small loops that constitute the core of the signal processing operations implemented by the multimedia algorithms. For the smaller JPEG decode and encode programs speedup levels off after the BTB reaches 1K.
Increasing the level of branch prediction beyond a certain point, has a reverse effect on fetch efficiency because there will be more instructions fetched which have a lower probability of being on the true path. Also, as more branches are predicted, the Instruction Window fills up more often causing performance to flatten out after a certain level. It follows that a primary concern when considering the performance of a branch prediction mechanism is whether the Instruction Window is becoming saturated. When this happens, improving fetch efficiency may not make much difference in performance, but instead it would increase the pressure on the Instruction Window. This implies that a simple branch prediction scheme, such as the one used here, might be sufficient to achieve a high level of performance. Figure 5.6 shows the number of instructions fetched for different decode sizes and branch prediction mechanism. The perfect model where all branches are predicted correctly has a prediction accuracy of 100%. The ideal model predicts all
branches correctly similar to the perfect model plus it does instruction aligning and merging perfectly. The difference in performance between the perfect model and the ideal model is due to instruction misalignment which introduces empty instruction slot. The ideal model does not reach the maximum fetch width because of the presence of NOPs in the program code which are not counted in the total run of instructions executed.

Figure 5.6 Instructions fetched for different decode width and branch prediction mechanism
5.4.2 Scheduling Unit

The scheduling unit is responsible for issuing the majority of the ready-to-run instructions and to prevent the Instruction Window from filling up. If the Instruction Window is not large enough then there are fewer instructions for the scheduling unit to issue in parallel. In that case performance is severely limited by the Instruction Window size. Also, if the Instruction Window is filled then the decoder stall and the instruction bandwidth decreases. The Instruction Window size plays an important role in keeping all the FUs busy and preventing the decoder from stalling. Another factor that affects instruction issue is write result-bus conflicts where several FUs are trying to write back operation results to the Reorder Buffer. These stalled results are often needed by other instructions waiting in the Instruction Window. Therefore, delaying write-backs delays all of the instructions that are dependent on them. This increase the effects of dependency even further on performance.

Figure 5.7 shows the speedup for a 4 instruction wide superscalar machine with different Instruction Window sizes. As the Instruction Window size is increased speedup increases. However, when the Instruction Window is increased beyond 16 instructions deep the speedup flattens. The reason is that there are not enough instructions decoded each cycle to fill all the available execution slots. In that case it might be appropriate to increase the decode width of the superscalar processor. It follows that the ideal size for the Instruction Window of a 4 instruction decoder is 16 instructions deep. Using a larger Instruction Window will has no effect on performance and can actually reduce speedup slightly. The loss in speedup is from a decrease in branch prediction accuracy. If a branch is mispredicted the content of the Instruction Window must be flushed. As the number of instructions increases the penalty of recovering from a mispredicted branch is greater and consequently the branch accuracy decreases.
5.4.3 Execution Unit

A primary requirement in designing a superscalar microprocessor is that the number and nature of functional units (FUs) in the system should closely match the dynamic instruction mix of the intended workload. The FUs on a machine must reflect the instruction frequency if we are to achieve efficient utilization of FUs [12]. For example, it wouldn’t make much sense to use 4 ALUs for an application that has on average 2 instructions that can be executed in parallel. On the other hand, if the CPU’s resources are not adequate to keep up with the high level of instruction parallelism in the application, then the application will never reach its potential speedup and there is no advantage in using a superscalar microprocessor.
5.4.3.3 Arithmetic And Logic Unit

In Figure 5.3 it is shown that the most frequent instructions are ALU operations. Therefore, adding an additional ALU would logically increase performance. Figure 5.8 shows the speedup vs. the number of ALU units for a 4 instructions wide superscalar microprocessor. The highest speedup is attained when using 3 ALUs. Increasing the number of ALUs beyond 3 has no effect on performance because of the presence of instruction dependencies.

As we increase the number of ALUs we need to increase the width of the write result bus accordingly otherwise any benefit incurred from increasing the number of ALUs will be lost to result write conflicts. An alternative to increasing the write bus width is using an arbitration scheme to alleviate result write conflicts by assigning priority levels to each FUs. Priorities can either be fixed or dynamically assigned to FUs depending on the dependencies between instructions [18].

![Figure 5.8 Speedup vs. number of ALUs](image-url)
5.4.3.4 Shift Unit

Because of the low frequency of the shift instructions there is no advantage in increasing the number of shift units. The average speedup improvement for all the benchmarks resulting from using two shifters is approximately 2% which does not justify the added cost in hardware.

5.4.3.5 Load/Store Unit

Load and Store operations are the second most frequent operations in the ten multimedia benchmarks studied. Also, they are one of the most expensive resources to duplicate because of the high cost of adding I/O ports. Adding an additional load/store unit increases performance by 10% to 15%. Figure 5.9 compares the speedup of using two load and store units versus one load and store unit.

Figure 5.9 Performance increase from using 2 Load and 2 Store Units
Table 5.5 compares the number and distribution of functional units for several commercial microprocessors. Also shown is the maximum instruction issue rate for each microprocessor.

<table>
<thead>
<tr>
<th>Vendor</th>
<th>RISC micro</th>
<th>Word Size (bits)</th>
<th>Functional Units</th>
<th>Integer ALUs</th>
<th>Max Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC</td>
<td>21164</td>
<td>64</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Intel</td>
<td>Pentium</td>
<td>32</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Pentium PRO</td>
<td>32</td>
<td>10</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>HP</td>
<td>PA-8000</td>
<td>64</td>
<td>10</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>IBM/Motorola</td>
<td>604</td>
<td>32</td>
<td>6</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>620</td>
<td>64</td>
<td>6</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>MIPS</td>
<td>R8000</td>
<td>32</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>R10000</td>
<td>64</td>
<td>9</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>SUN</td>
<td>UltraSPARC</td>
<td>64</td>
<td>9</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>SPARC64</td>
<td>64</td>
<td>10</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 5.5 Number of functional units and their distribution for several commercial microprocessors
5.5 Conclusion

A detailed account of different aspects of machine parallelism is presented. We evaluate three different issue policies and find that the out-of-order issue with out-of-order completion has the highest level of performance of all three policies. The drawback of the out-of-order issue with out-of-order completion policy is that it requires additional support hardware and more complex control logic. However, there is a significant performance advantage obtained from using these policies. Next, we evaluate the amount of hardware needed to support out-of-order issue and out-of-order completion. Results show that the performance of the superscalar DSP is dependent on the processor fetch efficiency. Contrary to what might be believed, increasing the fetch bandwidth has a marginal return on the average number of instructions fetched. As the decode width increases, instruction misalignment increases and consequently fetch efficiency decreases. On the other hand, we found out that the longer the instruction run, the less misalignment and better fetch efficiency. Also, fetch efficiency increases as the level of branch prediction increases because the processor spends less time waiting for branches to be resolved.

Furthermore, if the Instruction Window is not large enough then there are fewer instructions for the scheduling unit to issue in parallel and the performance is severely limited by the Instruction Window size. Also, if the Instruction Window is filled then the decoder stalls and the instruction bandwidth decreases. The size of the Instruction Window plays an important role in keeping all the FUs busy and preventing the decoder from stalling.

Next, results show that as we increase the number of ALUs we also need to increase the width of the write result bus accordingly otherwise any benefit incurred from increasing the number of ALUs is lost to result write conflicts.
Chapter 6  Conclusion

In the first part of this thesis, we present a study on the availability of instruction-level-parallelism in multimedia applications. We have concentrated our analysis on exposing the maximum parallelism inherent in this special class of programs. Ten different multimedia benchmarks are simulated under five different hardware models that represent a wide range of machine parallelism, from a single issue scalar machine to an unlimited multiple-issue superscalar machine. Simulation results show a high level of instruction parallelism that varied between 32 IPC for the small GSM program up to 700 IPC for the large MPEG encode program [4, 5]. These figures represent the upper bound on instruction-level parallelism. They serve as a yardstick to tell us when to stop adding resources in order to increase performance.

The large MPEG, H.261, and H.263 programs profit the most under the Good model with 512 registers and a 2048-entry branch prediction buffer. On the other hand, the Average model has the most cost/effective solution for the smaller JPEG and GSM programs. It seems that parallelism suffer the most when issue width is limited. Limiting the issue width reduces the scope of search for ILP. The ideal cycle width varies from one benchmark to another depending on the amount of instruction-level parallelism available in that particular benchmark. For the small benchmarks parallelism levels off after a cycle width of 128 instructions-per-cycle. Only MPEG and H.261 encode benefit from an increased cycle width because of the high ILP contained in their code.

The next most damaging effect on ILP is when no branch prediction is used. Using a simple branch predictor based on a 2-bit counter shows a high level of prediction accuracy that reached 95% for all benchmarks. However, performance levels off after the branch prediction
buffer is increased beyond 2048 entries. Increasing the buffer beyond 2K has an insignificant impact on performance considering that we need to quadruple the size of the buffer to get a mere 3% increase in parallelism.

In the last part of the ILP study we examine the nonuniformity in the distribution of instruction-level-parallelism. Simulation results reveal that 60% of the total number of instructions for all benchmarks are evenly distributed between 50% of the total number of program cycles. These findings suggest a high level of uniformity in the distribution of ILP for multimedia applications which results in a high resource utilization.

In the second part of the thesis a detailed account of different aspects of machine parallelism is presented. We evaluate three different issue policies and find that the out-of-order issue with out-of-order completion has the highest level of performance of all three policies. The drawback of the out-of-order issue with out-of-order completion policy is that it requires additional support hardware and more complex control logic. However, the performance advantage that is obtained from it outweights any other disadvantages.

Next, we evaluate the amount of hardware needed to support out-of-order issue and out-of-order completion. Results show that the performance of the superscalar DSP is dependent on the processor fetch efficiency. Contrary to what it might be believed, increasing the fetch bandwidth is a marginal return on the average number of instructions fetched. As the decode width increases, instruction misalignment increases and consequently fetch efficiency decreases. On the other hand, we find out that the longer the instruction run the better the fetch efficiency. Also, fetch efficiency increases as the level of branch prediction increases because the processor spends less time waiting for branches to be resolved.
Furthermore, if the Instruction Window is not large enough then there are fewer instructions for the scheduling unit to issue in parallel and the performance is severely limited by the Instruction Window size. Also, if the Instruction Window is filled then the decoder stalls and the instruction bandwidth decreases. The Instruction Window size plays an important role in keeping all the FUs busy and preventing the decoder from stalling.

Next, results show that as we increase the number of ALUs we need to increase the width of the write result bus accordingly. Otherwise any benefits that are incurred from increasing the number of ALUs are lost to result write conflicts.

**Future Work**

An interesting extension to this thesis is to simulate additional benchmarks such as MPEG-2 and 3D-Graphics. However, the simulation time is a major hurdle in simulating large programs. Therefore, it would profitable to evaluate alternatives to the instruction-trace simulation technique used in this work that reduces simulation time. The following is a list of topics that hold interests for additional research:

- a wider selection of applications and benchmarks
- multithreaded superscalar DSP
- cache performance
- multimedia extensions
- a cost/performance in terms of chip area, power consumption, and analysis
Bibliography


Appendix A. Simulation Techniques

This appendix introduces the options that are used to configure the simulator. Also, presented are the characteristics of each benchmark together with the FTP repository address for each benchmark, the files needed to download, and the command line passed to each benchmark.

In order to insure that the results that we obtained were sound we need to test our simulator. The methodology used to test the correctness of the results obtained for our simulator is simulate the benchmarks (e.g. grep and compress) that were used in previous studies [8, 24] and compare the results obtained with our simulator with the results obtained in these studies. Results for these two benchmarks, grep and compress, matched the results the results obtained in [8] and [24].

The simulator accepts a series of command line options that are used to configure the Superscalar DSP for different issue policies, width size, number of resources, etc. These options are:

\texttt{ssim <obj_file> [options...] -prog <obj_file.pixie>}

Options are:

-\texttt{align} \hspace{1cm} \text{align instructions in decoder}
-\texttt{-cN} \hspace{1cm} \text{centralized window of N locations}
-\texttt{-compl_in_order} \hspace{1cm} \text{issue and complete instructions in sequential order (default no)}
-\texttt{-d[a,o]} \hspace{1cm} \text{check for \{a\}:anti-dependencies} \hspace{1cm} \text{[o]:output dependencies}

(true dependencies always checked)
Appendix A. Simulation Techniques

-decode\text{N} \quad \text{decode N instructions per cycle (default 2)}

-fifo\_wind \quad \text{manage central window as fifo}

-flsh\_mispred \quad \text{flush result buffer on misprediction}

-issue\_in\_order \quad \text{issue instructions in sequential order (default no)}

-ldbrm\text{N} \quad \text{limit decode to N branches per cycle}

-ldmem\text{N} \quad \text{limit decode to N loads/stores per cycle}

-li\text{N} \quad \text{limit issue to N instructions per cycle}

-merge \quad \text{merge instruction runs if predicted OK}

-nobyp\_load \quad \text{loads do not bypass stores}

-nofwd\_store \quad \text{do not forward store data from buffer}

-perf \quad \text{perfect branch prediction}

-pred \quad \text{predict branches with BTB (or icache if enabled)}

-pred\_take \quad \text{predict all branches take}

-pred\_rtns \quad \text{predict branches and procedure returns}

-rb\_int\text{N} \quad \text{use integer result buffer of N entries (overrides machine configuration)}

-regs\text{N} \quad \text{use N register read ports (default 2*decode width)}

-resbus\_int\text{N} \quad \text{use N integer result buses}

-s\text{N} \quad \text{stop at first opportunity after N instructions}

-strbuf\text{N} \quad \text{limit store buffer to N locations}
The following describes the characteristics of each benchmark. Details about the functions implemented are briefly described. For more details please read the individual README file included in the original package. The FTP site shows the location where to find the benchmark and under which directory to find it. The Source file category shows the files needed to download in order to extract the benchmark. Most of these benchmarks come in C-source code format. In order to produce an executable for a particular machine the user needs to compile the program using the Makefile provided with each application. The “command used” line shows the command that are used to run each program together with the input files and the different options passed to the application.

**JPEG** This is a public domain image encoder and decoder is based on the JPEG Committee Draft. It supports all of the baseline for encoding and decoding. The JPEG encoder is flexible in the variety of output possible. It also supports lossless coding. The JPEG CODEC is published by the The Portable Video Research Group at Stanford.

**FTP site:** havefun.stanford.edu in /pub/jpeg

**Source files:** JPEGv1.2.tar.Z

**Command used:**

- Decoder command: `jpeg -d -s nonint.jpg`
- Encoder command: `jpeg -iw 128 -ih 128 -n -hf 2 nonint.jpg.1 nonint.jpg.2 nonint.jpg.3 -s nonint2.jpg`

**MPEG** This public domain video encoder and decoder was generated according to the Santa Clara August 1991 format. It has been tested successfully with decoders using the Paris December 1991 format. The codec is capable of encoding all MPEG types of frames. The algorithms for rate control, buffer-constrained encoding, and quantization decisions are similar, but not identical, to that of the (simulation model 1-3) MPEG document. The rate control used is a simple proportional Q-steps/Buffer loop that works though not very well -better rate-control is the essence for good quality buffer-constrained MPEG encoding. Verification of the buffering is possible so as to provide
streams for real-time decoders. The MPEG CODEC is published by the Portable Video Research Group at Stanford.

FTP site: havefun.stanford.edu in /pub/mpeg
Source files: MPEGv1.2.1.tar.Z
Command used: mpeg -d -s short.mpg shortx  
encoder command
mpeg -a 0 -b 6 shortx -s short2.mpg  
encoder command

H.261 This public domain video encoder and decoder is based on the CCITT H.261 specification. Some encoding algorithms are based on the RM 8 encoder. The CODEC supports all the encoding and decoding modes, and has provisions for buffer-constrained encoding, so it can produce streams for real-time decoders. The H.261 CODEC is published by the Portable Video Research Group at Stanford.

FTP site: havefun.stanford.edu in /pub/p64
Source files: P64v1.2.tar.Z
Command used: p64 -d -s short.p64 shortx  
encoder command
p64 -a 0 -b 6 shortx -s short2.p64  
encoder command

Telenor's H.263 encoder and decoder

Coding modes implemented: unrestricted Motion Vector mode, Syntax-based Arithmetic Coding mode, the Advanced Prediction mode, PB-frames mode.

FTP site: bonde.nta.no in /pub/tmn/software.
Source files: tmn-2.0.tar.gz  
encoder source code
tmndec-2.0.tar.gz  
decoder source code
tmnbits-1.6b.tar.gz  
sample bitstreams
Command used: tmn -i short.p64  
encoder command
tmndec stream.263  
decoder command

An implementation of the European GSM 06.10 provisional standard for full-rate speech transcoding, prl-ETS 300 036, which uses RPE/LTP (residual pulse excitation/long term prediction) coding at 13 kbit/s published by the Communications and Operating Systems Research Group (KBS) at the Technische Universitaet Berlin.
GSM 06.10 compresses frames of 160 13-bit samples (8 kHz sampling rate, i.e. a frame rate of 50 Hz) into 260 bits; for compatibility with typical UNIX applications, the current implementation turns frames of 160 16-bit linear samples into 33-byte frames (1650 Bytes/s).

FTP site: \texttt{ftp.cs.tu-berlin.de.in/pub/local/kbs/tubmik.}
Source files: \texttt{gsm-1.0.10.tar.gz}
Command used: \texttt{toast STlaser.au compress untoast STlaser.gsm uncompress}