1-1-2006

Maximizing resource utilization by slicing of superscalar architecture

Shruti Ravikant Patil
University of Nevada, Las Vegas

MAXIMIZING RESOURCE UTILIZATION BY SLICING OF SUPERSCALAR ARCHITECTURE

by

Shruti Ravikant Patil

Bachelor of Engineering
Veermata Jijabai Technological Institute
University of Mumbai
2004

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science Degree in Engineering
Department of Electrical Engineering
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
August 2006

UNLV
UNIVERSITY OF NEVADA LAS VEGAS

Thesis Approval

The Graduate College
University of Nevada, Las Vegas

June 19, 2006

The Thesis prepared by

Shruti Patil

Entitled

"Maximizing Resource Utilization by Slicing of Superscalar Architecture"

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering

Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Examination Committee Member

Graduate College Faculty Representative

1017-53

ABSTRACT

Maximizing Resource Utilization By Slicing Of Superscalar Architecture

by

Shruti Ravikant Patil

Dr. Venkatesan Muthukumar, Examination Committee Chair
Professor of Electrical and Computer Engineering
University of Nevada, Las Vegas

Superscalar architectural techniques increase instruction throughput from one instruction per cycle to more than one instruction per cycle. Modern processors make use of several processing resources to achieve this kind of throughput. Control units perform various functions to minimize stalls and to ensure a continuous feed of instructions to execution units. It is vital to ensure that instructions ready for execution do not encounter a bottleneck in the execution stage.

This thesis work proposes a dynamic scheme to increase efficiency of execution stage by a methodology called block slicing. Implementing this concept in a wide, superscalar pipelined architecture introduces minimal additional hardware and delay in the pipeline. The hardware required for the implementation of the proposed scheme is designed and assessed in terms of cost and delay. Performance measures of speed-up, throughput and efficiency have been evaluated for the resulting pipeline and analyzed.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Graduate school has been a journey with classic experiences that have brought about a great change in my research and my life. I am thankful for the time I spent at UNLV.

I feel extremely fortunate to have had Dr. "Venki" Muthukumar as my advisor. He is the man who has mentored me endlessly and helped me define and refine my research ideas with an incredible amount of support and patience. As is apparent in the first few interactions, his knowledge in a wide range of arenas brings him an all-round personality topped by a witty sense of humor. The support and dedication he has for his students is more than what any student can ask for. I am extremely grateful for the research tools and the constant guidance that he has given me.

I am thankful for the wonderful laboratory environment sparked by my lab-mates, Ashwini Raina, Naveen Chinthalcheruvu, Gopinath Balakrishnan and Shankar Neelakrishnan. It is only in a conducive environment that out-of-box thinking is cultured. Friendliness and the ability for hard work that has become a mark of all Lab B348 inmates was rubbed off on me when I joined the research team. I am grateful for my interactions with my friends, Amruta Tilaye, Rathna Ramaswamy, Pradeep Nambisan and Navin Veermisti who have helped me at various points professionally and personally.

Lastly, I would like to thank my parents, sister and my grandmother for being supportive of my academic journey. They have been extremely understanding and a constant source of encouragement and joy.

CHAPTER 1

INTRODUCTION

As we move to the GSI (Giga-Scale Integration) era, the challenges presented to a computer architect increase in constraints and complexity. Current demands of technological advances have spurred an exceptional development in the way computers are designed. Advances in computer architecture span across the concepts of out-of-order superscalar architectures, aggressive speculative techniques, high bandwidth caches, etc. to distributed processor architectures.

The next section briefly traces the development of architectures and research on modern architectures with enhanced performance and capabilities. Among the factors that are sought to be continuously improved in a machine are clock speeds and instruction throughput. This research work proposes a dynamic way to increase instruction throughput, by concentrating on the processing elements of an architecture and adding flexibility so that the processing bottleneck is addressed.

1.1 History of Computing

During the late forties, computers were mostly developed as a machine performing logic and arithmetic operations using vacuum tubes. There was a need for suitable electronic hardware architecture which was much more efficient than the existing electro-mechanical devices (ENIAC). This led John Von Neumann to formulate a machine controlled by a sequential program stored in electronic memory along with the data, and it came to be known as EDVAC (Electronic Discrete Variable Automatic Computer). With Neumann's architecture as the base, new architectural concepts were conceived by integrating software with the hardware. With the invention of

programming languages and operating systems, the computing power has increased from few hundreds to several billion instructions per second. Program sizes increased from few thousands to several millions of code lines. Computers came to be seen as both special purpose machines executing a particular program and as universal machines capable of simulating any special purpose machine.

The Manchester university computer science group developed the idea of *indexed modification of addresses and the memory hierarchy* in 1949. The index registers permitted the execution of loops without modifying the instruction addresses and the memory hierarchy idea led to the development of caches and virtual machines concept. In 1951, Wilkes proposed the *microprogrammed control,* as a systematic way of controlling the operation of computers. *Stack architecture* was proposed by Barton in 1958 as a tool for compiling and executing expressions. This resulted in the machine architecture reflecting the organization of the programming language. The late fifties saw the development of *multiprocessors* with separate *I/O processors* and *arithmetic processors.*

Vector processors provided efficient machine operations involving data structures. Cray-I developed in 1973 is an example of vector supercomputer. With the advent of vector processors, *pipelined architectures* came into existence, which have since become the backbone of subsequent architectures. The pipelined architecture obtains faster operations by decomposing each operation into steps to be executed by cascaded sub-units. Systolic array architecture evolved from pipelined architecture characterized by identical processing elements connected in a linear or a multi-dimensional array where in each processing element is connected only to its adjacent elements only.

In 1972, the increasing density of the component on a chip using VLSI techniques and the corresponding lower costs resulted in implementation of a complete processor on a single chip, known as *microprocessor.* Further increase in component density has led to the evolution of microprocessors with complex instruction set (CISC) and more functionality on hardware. But this also resulted in slow down of the processor speed.

Generation	Technology and Architecture	Software and Operating System	Example Systems
First 1946-1956	Vacuum Tubes and Relay Memory, Single-bit CPU	Machine/assembly language, programs without subroutines	ENIAC, IBM 701, Princeton IAS
Second 1956-1967	Discrete Transistors, core memory, floating-point accelerator, I/O channels	Algol and Fortran with compilers, batch processing OS	IBM 7030, CDC1604, Univac LARC
Third 1967-1978	Integrated Circuits, Pipelined CPU, microprogrammed CU	C language multiprogramming, timesharing OS	PDP-11, IBM360/370, CDC6600
Fourth 1978-1989	VLSI microprocessors, multiprocessors, vector supercomputers	Symmetric multiprocessing, parallelizing compilers, message-passing libraries	IBM PC, VAX 9000, Cray X/MP
Fifth 1990-present	ULSI circuits, scalable parallel computers, workstation clusters, internet	Java, microkernels, multithreading, distributed OS, WWW	IBM SP2, SGI Origin 2000 Digital TruCluster

Figure 1. Computer Generations in 50 years [22]

To increase the speed of processing and reduce the number of instructions, architectures with reduced instruction set (RISC) were implemented with simple circuitry. The invention of CISC and RISC architectures formed the baseline for the burst of several new architectures which led to the birth of new generation known as *superscalar architectures*. Superscalar processors have the ability to process several instructions in the same instruction cycle based on whether an instruction is an independent instruction or dependent on another.

Figure 1 shows the five generations of computers [22] concisely, which depict five distinct development phases in the computer industry.

1.2 Architectures & Classifications

The parallel processing ability of the superscalar architectures resulted in many different architectures and it is imperative that we classify the architectures into various categories based on their features.

1.2.1 Classification based on Instruction Set complexity

- Complex Instruction Set Computer (CISC)
- Reduced Instruction Set Computer (RISC)
- Minimal Instruction Set Computer (MISC)
- High Level Instruction Set Computer (HISC)
- Writable instruction Set Computer (WISC)
- Zero Instruction Set Computer (ZISC)
- Very Long Instruction Word (VLIW)

1.2.2 Flynn's Taxonomy based on parallelism in instruction and data streams

- Single Instruction Single Data stream (SISD)
- Single Instruction Multiple Data stream (SIMD)
- Multiple Instruction Single Data stream (MISD)
- Multiple Instruction Multiple Data stream (MIMD)
- Centralized Shared Memory
- Distributed Memory

1.2.3 Classification based on internal storage of operands

- Stack Architecture
- Accumulator Architecture
- Load-Store Architecture
- Register-Memory Architecture
- Memory-Memory Architecture
- Extended Accumulator / Special Purpose Register Architecture

1.2.4 Classification based on application

- General Purpose Architectures

- Application Specific Architectures

1.2.5 Recent classification based on ability to exploit Instruction Level Parallelism

- Scalar, Non-pipelined
- Scalar, Pipelined
- Superscalar, Non-pipelined
- Superscalar, Pipelined
- Superscalar, Superpipelined

1.3 Design & Evaluation of architecture

The essential elements of a processor are datapaths, instruction set and control unit. A datapath is either designed with general processing elements that process all incoming tasks or it is designed to handle specific tasks using specialized components. The datapath controls the processing abilities in the architecture. An instruction set is then required to be designed for the processor. An instruction generally consists of a field to specify operations to be performed and one or more fields to specify data to perform the operations on. The instructions may also be designed to provide control information to the processor to execute the operations in an efficient manner. In this case, there is another field called the control filed that contains pre-determined control bits. The control unit generates control signals that allow a concurrent functioning of different modules in datapaths and enable the processor to output results timely and correctly.

Various designs for architectures have been developed over the years. Most are described in the classifications listed above. A new architectural design is generally required for enhancing current performance, to impart newer capabilities to an existing architecture or to exploit the latest circuit-level technology.

The impact that a newly designed architecture will have on tasks and programs needs to be evaluated in order to successfully put it to use in practical applications. Benchmark suites containing programs that represent a variety of application tasks

have been developed to assess the performance of architectures under different environments. Certain desirable properties have been identified for performance metrics to evaluate architectures. These are linearity, reliability, repeatability, ease of measurement, consistency and measurement as described in [2]. The performance metrics listed below have been extensively used since decades for reflecting performance of new architectures:

- Clock frequency
- Millions of instructions executed per second (MIPS),
- Millions of floating point instructions executed per second (MFLOPS)
- Execution Time
- Speed-up with respect to other systems

Once an architecture has been designed, it is analyzed for cost (in terms of gate equivalents) and maximum clock frequency. Benchmark programs are run on the machine and their execution time gives an indication of the quality of the architecture for the area of applications represented by specific benchmark programs. These metrics aid in comparing different architectures and facilitate the choice of an architecture for an application at hand.

1.4 Motivation for the Research work

With advancements in VLSI design tools and fabrication techniques, the chip area available to implement complex computer architecture has increased exponentially. This increase in area can be used either to accommodate more number of modules, modules of increased complexity and functionality or a combination of both. While acknowledging the available latitude of chip area, this thesis explores ways of increasing the efficiency of modules on the chip by introducing additional functionalities to existing modules.

Most modern-day processors have a data width of 64-bits. It is possible to efficiently use the processing elements to operate on data of smaller word sizes. A scheme called

as *block slicing* is proposed in this work to increase instruction throughput when such data is encountered. The scheme is applied to functional units to increase execution parallelism in wide superscalar, pipelined architectures. This technique will be more effective in general purpose machines and will lead to a higher processing rate, without increasing processing units.

The scheme of block slicing, its design, implementation and evaluation have been elucidated in this thesis. This document is organized as follows. Chapter 2 presents a literature review of computer architectures and their applications. Chapter 3 describes the superscalar architectural technique for exploiting Instruction Level Parallelism (ILP) and the DLX architecture designed for academic purposes and described in [1]. Chapter 4 describes the concepts introduced by this thesis work and their design and implementation as pipelined units. Chapter 5 evaluates the concepts and presents the results of simulations. Chapter 6 presents conclusions from this work and proposes future work that remains to be performed and evaluated.

CHAPTER 2

PRIOR WORK

Computer architectures have advanced from the ENIAC (Electronic Numerical Integrator And Computer) to present day multi-core processors. Classification of computer architectures is not only necessary to determine an optimal design for a system, but also to systematize the sample space for architectural exploration and progression. There are several categories under which architecture can be classified. Some of the conventional classification methods are based upon the complexity of the instruction set, operand storage, application and instruction processing scheme. Other criteria like cost, capacity, performance and component density have also been used in the past to provide a basis for classification. Apart from these categories, lately new classification methodologies based upon number of storage hierarchy levels, number of addressable fields, fault tolerance of the system and reconfigurability are being used to compare performance of upcoming architectures.

This chapter discusses conventional computer architecture classifications followed by a description of different types of superscalar processors. Fourth generation processors are also briefly explained in this section. A survey of existing literature is presented at the end of the chapter.

2.1 Classification of Computer Architectures

Architectures can be categorized based on a number of broad classification criteria. Most commercial processors fall into a number of these criteria. It is possible to make a narrower classification for advanced processors like parallel processors, distributed processors and network processors.

2.1.1 Classification based on Instruction Set Complexity

1. Complex Instruction Set Computer (CISC):

The CISC instruction set comprises of several RISC operations in a single instruction. This reduces the lines of code for a program and gives the designer the ability to optimize multiple instructions in a single step. A reduction in instructions leads to lower memory requirements and fewer memory accesses. However, the functions of the instruction decoder stage intensify to a large extent. Typically, the number of instructions in a CISC machine is 80-150. The main features of a CISC system include register to memory and memory to register instructions, multiple addressing modes for memory, two operand format, variable length instructions and many clock cycles per instruction. The CISC architecture is characterized by a complex instruction decode logic, a small number of general purpose registers and several special purpose registers.

2. Reduced Instruction Set Computer (RISC):

The RISC architecture supports simple basic instructions that can be combined to achieve complex tasks and capable of running faster than CISC instructions. The instruction decoder design is simplified due to the nature of instructions, and hence control path design process is uncomplicated. The RISC architecture enables a computer architect to exploit instruction parallelism and out-of-order execution. RISC processors have complex memory hierarchy in order to work at full speed and allow for uninterrupted pipeline flow.

These processors are often classified based on various measures like the datapath width, pipeline width, word size, cache structure, bus structure, type of buffers and types of register files.

3. Minimal Instruction Set Computer (MISC) [4]:

The MISC architecture is made to exploit simplicity by assuming only 32 instructions. As the speed of the RISC processors increases, a bottleneck is created between the processor and the memory. A cache memory is necessary to buffer

instruction and data streams in order to increase the memory access speed. Cache memory complicates the system design and makes the system more expensive. RISC processor is also very inefficient in handling subroutine calls and returns. A large register window big enough to handle input, output and local parameters is used to assist in subroutine calls. This large register window wastes the most valuable resource in the RISC processor and slows the system during context switching. MISC is implemented with only four instruction groups: transfer instructions, memory instructions, arithmetic instructions and register instructions.

4. High Level Instruction Set Computer (HISC) [5]:

HISC is 64 bit architecture. It involves simple instructions of fixed length, entries of operand descriptors and application oriented data types. The operands of an instruction are described by Operand Descriptors which are records and consist of virtual addresses, data types, operand sizes, vector information, operand access codes and design and system dependent information for the operand. The data types of the operands include integer, floating-point number, BCD, character and string. The vector information includes number of elements in the vector and the element spacing for vector operands.

HISC reduces the demand for conditional branching as in RISC by eliminating the looping count for operands of variable lengths and large size, as well as vectors. On the other hand, HISC will operate super-scalar on a higher level. The interdependency of operands will be much less while it is likely to operate super-scalar for two or more function units. HISC also keeps the vector information so that vector operations are done by hardware.

This is a general purpose architecture targeted on high performance, implementation flexibility, expandability, better access control and system dependent features. HISC processor provides better encapsulation and is better suited for multimedia applications.

5. Writable Instruction Set Computer (WISC):

Writable Instruction Set Computer is a stack based architecture whose design is based on VLSI design methodology. These stack machines offer processor complexity that is much lower than that of CISC machines and overall system complexity that is lower than that of either RISC or CISC machines.

Earlier, stacks were placed in program memory chips. WISC maintains separate memory chips or even on-chip memory for the stacks. This configuration provides extremely fast subroutine calling capability and superior performance for interrupt handling and task switching. WISC combines stack machine design with opportunities offered by VLSI fabrication technology. This combination produces simplicity and efficiency. Multiple stacks with hardware stack buffers, zero-operand stack oriented instruction sets and the capability of fast procedure calls lead to features like high performance without pipelining, simple logic and low system complexity, small program size, fast program execution and low interrupt response and a low cost for context switching. A successful application area for WISC is real time embedded control environments.

6. Zero Instruction Set Computer (ZISC) [6]:

ZISC is a neural network based integrated circuit which is designed for applications using super computers. ZISC uses accumulated knowledge to recognize and classify objects and take decisions. It learns by examples from samples of data. The built-in learning mechanism accumulates knowledge during the training when examples and their solutions are entered. ZISC has generalization capability which gives the capability to react to objects which were not part of the learning examples.

ZISC's learning capability is not limited in time and volume. Its chips can be cascaded to create a larger system which ensures that the system architecture caters to the increase in technology density. Several chips can be linked together to build a wider network, without adding logic. These features make ZISC very easy to use and capable of solving problems which are not clearly defined.

ZISC has a high performance, capable of operating in real time and can be used in pattern recognition and classification. It also has the ability to separate noise from signal and this makes it perfect platform for signal processing.

7. Very Long Instruction Word (VLIW):

Scheduling the instructions is the core problem in a modern processor design. VLIW design provides an alternative by letting the software do all the scheduling. The compiler examines the program, finds all the instructions with no dependencies, strings them together in a very long batch and executes them concurrently on an equally big array of function units such that all the function units are used efficiently.

Very long instructions are typically between 256 and 1024 bits wide. These instructions contain many smaller fields, each of which directly encodes an operation for a particular function unit. The hardware involved is very simple, consisting of a collection of function units which include adders, multipliers, and branch units etc, connected by a bus, plus some registers and caches. More silicon is used in actual processing and hence VLIW processor runs fast as the only limit is the latency of the function unit. Due to its ability for scientific number crunching, VLIW machines are highly used in scientific array processing and signal processing.

2.1.2 Flynn's Taxonomy

Flynn categorized all systems based on parallelism in the instruction and data streams which are simultaneously active at the bottleneck component of the multiprocessor system. All computers are placed into four different categories:

1. Single Instruction Single Data (SISD) Stream:

This is the class of conventional, sequential Von Neumann machines, in which only one instruction consuming a restricted amount of data is allowed to execute at a time. All state changes due to the instruction must be completed before the execution of next instruction begins. This category is the uniprocessor category.

2. Single Instruction Multiple Data (SIMD) Stream:

Multiple processors execute the same instruction using different data streams. Each processor had its own data memory but there is a single instruction memory and control processor, which fetches and dispatches instructions. Here, only one instruction can be executed at a time but the state changes induced by the instruction may be large. Parallelism is exploited by performing the same operation concurrently on many pieces of data. Vector architectures belong to this class of computers.

3. Multiple Instruction Single Data (MISD) Stream:

No commercial multiprocessor systems of this type exist to date. Some special purpose stream processors use this architecture as there is only a single data stream to be operated on by functional units.

4. Multiple Instruction Multiple Data (MIMD) Stream:

This class includes all parallel machines which contain multiple processors each with its own program counter. Each processor fetches its own instructions and operates on its own data. Different operations may be performed concurrently on many pieces of data. The processors in the multiprocessor system are often taken off-the-shelf.

Due to its flexibility and cost performance factors, MIMD type of architecture has clearly emerged as the most preferred architectures for general purpose multiprocessor systems. MIMD multiprocessors are divided into two different classes based on the number of processors used, the organization of the memory and the interconnection strategy:

5. Centralized Shared Memory Architectures:

In this type of architectures, typical processor count would be few dozens. A single centralized memory is connected to the processors using a single bus when the processor count is less. By replacing the single bus with multiple buses, the centralized memory can be scaled to handle more number of processors. These

multiprocessors are called *Symmetric Multiprocessors* (SMP) because of its single memory and its symmetric relationship to all the processors and the uniform access time from any processor. This style of architecture is also known as *Uniform Memory Access* (UMA).

6. Distributed Memory Multiprocessor Architecture [1]:

When the number of processors involved is large, a centralized memory system would not be able to support the bandwidth demands of processors without incurring excessively long access latency. Hence, memory must be distributed among the processors rather than being centralized. There are two major benefits of having a distributed memory system. First, this model reduces the latency for accesses to the local memory and the second, proves to be a cost effective way of scaling the memory bandwidth when most of the accesses are to the local memory.

2.1.3 Classification based on storage of operands [1]

The type of internal storage of the operands in a processor is the most basic differentiation used for classifying the architectures. These are explained below:

1. Stack Architecture: All operands accessed by this type of architecture are stored in a stack. An operation is performed by taking operands from the top of the stack.

2. Accumulator Architecture: This architecture implicitly accepts an operand stored in a special register called as an accumulator, and the second operand is stored into a register. The result of an operation is also stored implicitly in the accumulator. The advantage of this scheme is that the address of only one operand needs to be specified while performing an operation.

3. Load-Store Architecture: In this class of computers, memory access is only possible with load and store instructions.

4. Register-Memory Architecture: Here, memory access is possible as part of any instruction.

5. Memory-Memory Architecture: This is a third class of architecture, not found commercially. All operands are stored and accessed from the memory itself.

6. Extended Accumulator/ Special Purpose Register Architecture: There are more registers present in this architecture than a single accumulator but restrictions are placed on the use of these special registers. Such architecture is known as extended accumulator or special purpose register architecture.

2.1.4 Classification based on application

Architectures can also fall into two categories based upon the application they can process: general-purpose, application-specific and parallel processors [7].

1. General Purpose Architectures

These kinds of architectures can perform a variety of tasks, and are the basis for most Intel processors in a desktop machine. This is achieved by breaking down the tasks to a generic instruction set which is supported by the architecture.

2. Application Specific Architectures

These architectures are targeted towards a specific application, or a family of applications. Some application-specific architectures have been built for digital signal processing, image processing and mixed signal processing. Every module in such architecture is implemented to perform at maximum efficiency and least redundancy. The instruction set is also customized for the application.

2.1.5 Classification based on Instruction Level Parallelism

Instruction Level Parallelism (ILP) denotes a processor's ability to run many instructions at the same time. Exploiting ILP has led to the evolution of superscalar pipelined processors from a basic scalar processor. Today, ILP has become a major factor around which processors are designed. The *Amdahl's Law* is generally used to quantify a processor's performance based on ILP. Superscalar processors, when subjected to Amdahl's Law increased performance by a great magnitude.

The types of architectures based on their ability to exploit ILP are:

- Scalar, Non-Pipelined
- Scalar, Pipelined
- Superscalar, Non-pipelined

- Superscalar, Pipelined
- Superscalar, Superpipelined

1. Scalar, Non-Pipelined Architecture:

Architectures with a throughput of 1 Instruction per Clock Cycle (IPC) are termed as scalar architectures and these represent the simplest class of computers. Architectures like Intel8085™ through Intel386™ were scalar and non-pipelined architectures, with least clock speeds among all other categories in this classification.

2. Scalar, Pipelined Architecture

By interconnecting the different phases that an instruction undergoes during the time it arrives into the processor till the time it leaves it, it is possible to gain processing speed. The phases are scheduled so that each phase proceeds in a lock-step fashion, much like the assembly line processing in an automobile factory. Such architecture is called a pipelined architecture. Pipelining increases clock speeds by a factor of the number of stages that are included in it. A typical pipeline, shown in Figure 1 consists of six stages: fetch, decode, read registers, execute, writeback, write to memory.

3. Superscalar, Non-Pipelined architecture

An architecture that is capable of processing more than one instruction per cycle is called superscalar. Such an IPC is obtained by having more than one copy of processing elements. There is no machine that is superscalar and non-pipelined. This category only exists for the sake of completeness.

Figure 2 shows the time periods within which superscalar architectures were designed.

4. Superscalar, Pipelined Architecture

Superscalar architectures were built with a view to extract parallelism from data and instructions. Multiple instructions and data are fetched simultaneously and out-of-order execution is enabled to reduce stalls. Additional hardware and stages,

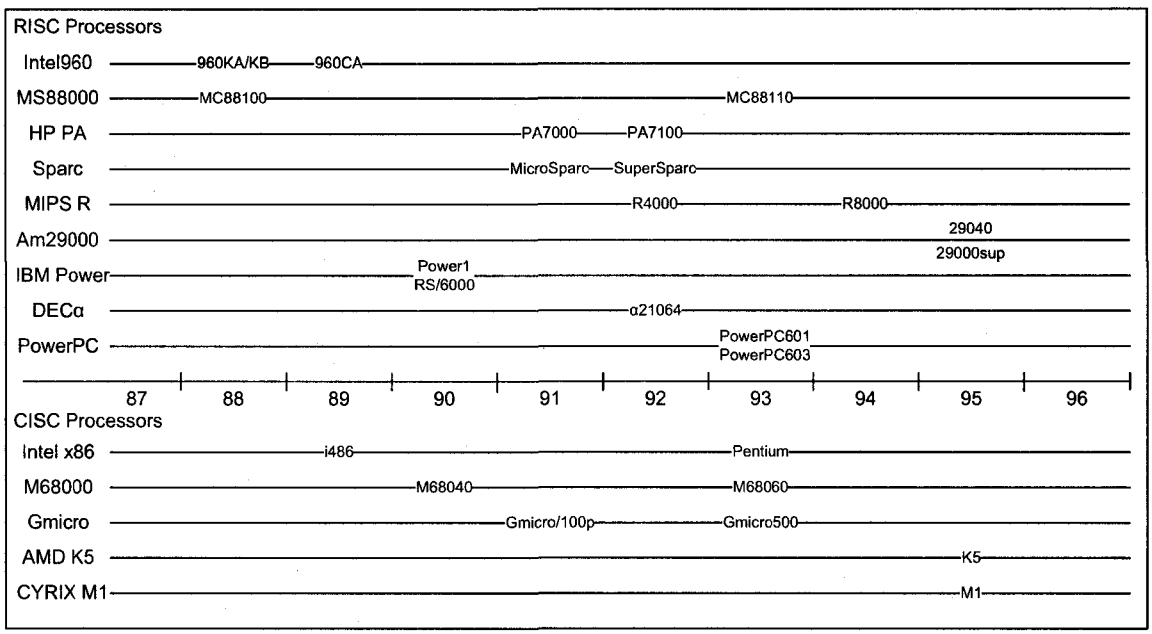

Figure 2. The appearance of superscalar processors on a timeline [23]

like reservation units and reorder buffer are necessary to process an instruction in a superscalar pipeline.

5. Superscalar, Superpipelined Architectures

If the internal stages in a pipeline are themselves pipelined, the architecture is called superpipelined. This facilitates the use of faster clocks and mechanisms to avoid a kind of WAR stall.

Figure 3 shows the evolution of commercial superscalar processors [23].

2.1.6 Fourth Generation Processors:

Improvements in processor performance are achieved by two means:

- Advances in semiconductor technology
- Advances in processor microarchitecture.

To sustain the historic rate of increase in computing power, it is important for improvements to occur in both ways mentioned. It is certain that clock frequencies will continue to increase. The main architectural challenge is to issue many instructions per

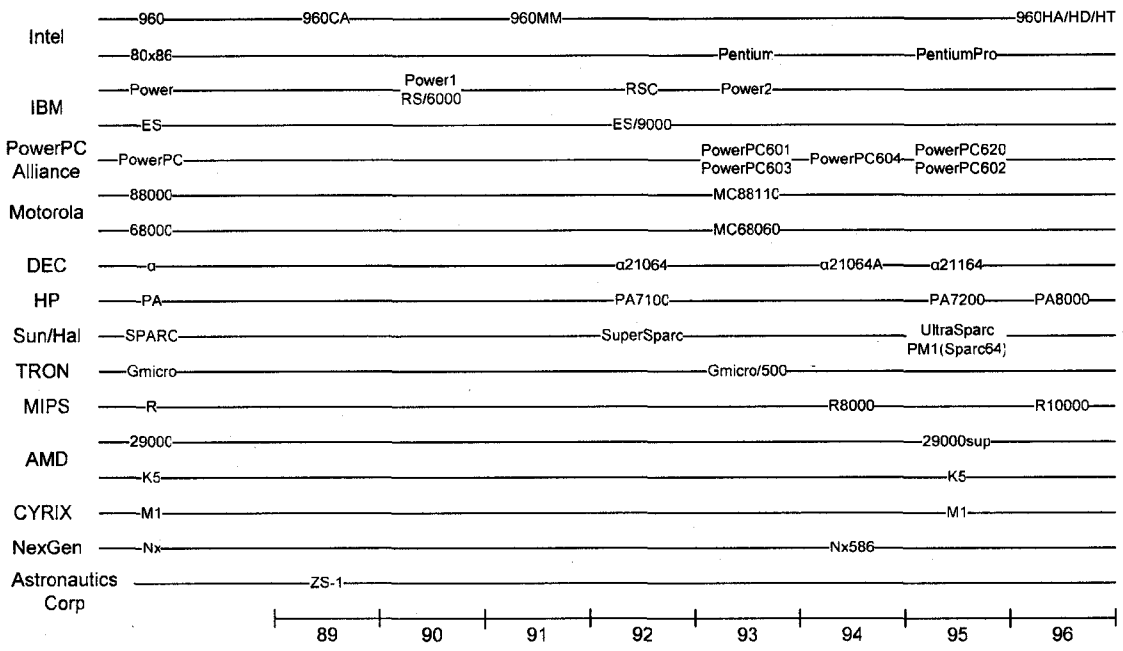

Figure 3. Evolution of Commercial Superscalar Processors

cycle and to do so efficiently. Five next generation processors are described in this section, which exploit the ILP in a system together with speculation.

1. Superspeculative Processors [8]:

These are wide issue super scalar processors, that can issue up to 32 instructions per cycle. The inability to go beyond the data flow limit restricts the complete exploitation of Instruction Level Parallelism. Superspeculative processors overcome the dataflow limit problem by aggressively speculating on past true dependencies and exploring additional instruction parallelism.

The core basis for the superspeculative processors is that the producer instructions generate highly predictable data in real programs. By successfully speculating on the source operand values, the consumer instructions can start execution without waiting for the result of the producer instructions. Thus, a superspeculative processor removes the serialization constraints between the producer and consumer instructions, there by thrusting its performance to go beyond the classical data flow limit without sacrificing the code compatibility.

2. Trace Processors [9]:

'Traces' are dynamic instruction sequences constructed and cached by hardware. Traces are built as program executes and are stored in a cache. Trace processor systems work by breaking down the system into several processing elements (PE) and the program into several traces so that the current trace is executed on one PE while the future traces are speculatively executed on other PEs.

Each processing element has enough instruction buffer space to hold an entire trace, multiple dedicated functional units, a dedicated local register file for holding the local values and a copy of the global register file. Instruction fetch hardware segments the program into traces, each of which may have 8 to 32 instructions as well as embedded predicted conditional branches. The traces are placed in a trace cache and a trace fetch unit subsequently reads the traces from the trace cache and sends them out to the parallel processing elements. Hence, the trace becomes the basic execution unit through out the processor. Two major advantages of the trace processors are:

- The physical registers are divided into local and global registers. This hierarchical organization allows for smaller register files which have fast access times and fewer ports per file.
- Successful value prediction of the trace's data allows the trace to be executed immediately and in parallel with other traces.

3. Multiscalar Processors [10]:

Multiscalar processors divide a program into different tasks that are distributed to a number of parallel processing elements (PEs) which are controlled by a single hardware sequencer.

A program is divided into a collection of tasks by using software and hardware. These tasks are then distributed to the parallel processing elements. Each PE fetches and executes the instructions assigned to it. The appearance of a single local register file is maintained with a copy in each PE. Compiler generated masks enable

the dynamic routing of the register results to different processing units. Memory accesses occur speculatively and the addresses are decoded dynamically. The only wait involved in this system is caused by the true data dependencies.

4. Datascalar Processors [12]:

The Datascalar model of execution runs the same sequential program redundantly across multiple processors. The data set is distributed across physical memories that are tightly coupled to their distinct processors. Each processor broadcasts operands that it loads from its local memory to all other processors. Instead of explicitly accessing a remote memory, processors wait until the requested value is broadcasted. Stores are completed only by the processor that owns the operand, and are dropped by the others.

This architecture exploits the fact that all memory is local to some processor in a multiprocessor system. Thus each read operand can be fetched by some processor and each memory update can be achieved by means of a write by some processor. A major advantage of the datascalar architecture will be its ability to exploit parallelism in codes that were not traditionally thought of as eligible for parallel processing. Datascalar model is way of optimizing the memory and is not intended to be substitute for parallel processing.

5. Advanced Superscalar Processors:

These are wide issue superscalar processors that can issue up to 32 instructions per cycle.

An important feature of this architecture is its large trace cache and a large number of reservation stations to accommodate 2000 instructions. There are 24 to 48 highly optimized, functional units. Aggressive speculation is performed to predict the branches.

2.2 Prior research on special architectures

2.2.1. Billion Transistor Architectures [12,13]:

Doug Burger and Goodman speculate and explore the future trends in computer architecture. They extrapolate the scope of having one billion transistors on a single chip. Important trends that would take place over the course of next 10 years are discusses in this paper.

A one billion transistor chip would require hardware manipulation but the physical limits like on chip signaling, wire delays, global clock would be serious constraints. They expect a quantum leap in compiler's ability to extract parallelism thereby shifting some of the parallelism from the hardware to the software. Considering the growing costs involved in design, verification and testing, the authors conclude that architectures that simplify the interaction among on-chip components and/or reduce the number of interacting components will have greater advantage over architectures that do not.

2.2.2. One Billion Transistors, One Uniprocessor, One Chip [14]:

Patt et.al propose that when systems with one billion transistors are available, computing systems with highest performance will have a single processor on each processor chip. They identify architecture that will have highest performance by utilizing the maximum available instruction bandwidth. The hardware will consist of the following components:

- A large trace cache
- A large number of reservation stations
- A large number of pipelined functional units
- Sufficient on-chip data cache
- Sufficient resolution and forwarding logic

These components are necessary for aggressive speculation using aggressive branch predictor and for very wide issue superscalar processing.

The highest performance computing system will be a multiprocessor consisting of powerful single chip uniprocessors. These will issue and execute 16 or 32 instructions per cycle with nearly 100 percent branch prediction accuracy.

2.2.3. Dynamic Instruction Set Computer [15]

Michael Wirthlin and Brad Hutchings describe a new computer architecture that can support dynamic modification of its instruction set based on the demand of the incoming instruction. They present an implementation of a DISC architecture based on three techniques:

- Partial FPGA reconfiguration – Partial reconfiguration provides the ability to reconfigure a sub section of an FPGA while remaining logic operates unaffected. Instructions occupy FPGAs only when needed while FPGA resources can be reused to implement an arbitrary number of performance enhancing application specific instructions.
- Relocatable hardware – Relocatable hardware gives the flexibility to relocate or make placement decisions of partial configurations at run time. This feature is used in DISC to enhance run time hardware utilization. Relocating hardware works on a strictly defined global context. Every instruction module is configured on to FPGA in such a way that each module is as close as possible to the other in order to avoid wasted hardware between modules. A global context provides physical placement positions and a communication network necessary for these modules to operate correctly.
- Linear Hardware Model – DISC implements relocatable hardware in the form of a linear hardware model. The two dimensional grid of configurable logic cells are organized as an array of rows. Each module's location is specified by the vertical and horizontal location while the size of the module is given by the module height.

DISC is an example of application specific processor with large instruction sets that can be implemented on partially reconfigurable FPGAs.

2.2.4. VISA: A Variable Instruction Set Architecture [16]

The author of this paper describes an instruction coding technique that reduces the width of the instructions using dynamic instruction coding managed by the compiler. A RISC processor is constrained by the instruction width to keep it within the limits imposed by silicon. In this case, the compiler defines the set of instructions required in order to execute a given program and selects the hardware function that can be activated during the same machine cycle by using an instruction.

Compiler divides and determines the instruction set based on two factors:

- Functions to be activated
- Number of bits needed by such functions.

The author presents a new VISA based microprocessor named VISP which delivers high performance using the variable instruction set architecture for general as well as floating point calculations. The result of the new architecture is more compact code and notable increase in optimization capabilities of the compiler.

2.2.5. Application Specific Instruction Set Processor (ASIP) [17]

Chandra Shekhar et al. compare software based general purpose architectures to dedicated hardware architectures and identify how the benefits of both are realized through ASIP architectures. Dedicated hardware architectures can be combinational, sequential, pipelined, and parallel or can be a mix of any of these but a change in functional specification necessitates a change in the architecture. These are closely tied to the logic specification of the specific application and hence are very inflexible in their functionality.

On the other side are the general purpose architectures which can implement any logical function without requiring any chance in the hardware. This flexibility comes from the use of a rich instruction set. The CPU hardware is designed only to execute any instruction from the instruction set loaded into its instruction register and then proceed to load and execute the next instruction from the memory into instruction register of the CPU. Whenever there is a change in specification, only the sequence of

instructions stored in the memory changes, which is why it is called software based architecture.

The inclusion of complex instructions in the instruction set of the processor in addition to the necessary general purpose instructions makes the instruction set and the processor application specific.

Authors propose that the ASIP hardware architecture would contain a number of application specific functional blocks and the necessary bussing to move the data. This reduces the memory accesses and the data transfers among the hardware blocks. A reduced number of busses in the CPU reduces the amount of bus interface logic in the functional blocks in the CPU and control logic in the control part of the processor. ASIP processors will run multiple overlapped executions of operations in different functional units to achieve maximum possible concurrency. Pipelining occurs at the functional block level. Application specific instruction sets and processors are suitable for embedded applications as they permit an alteration of hardware-software boundary to meet the speed and energy constraints of a specific application.

2.2.6. Application Specific Architectures [18]

Chris Weaver et al. proposed that the potential of the application specific architectures can be harnessed by specializing a design to a small domain of important applications. The benefits of this approach would be improved performance, greater power efficiency and reduced costs. Key differences between general purpose architecture versus application specific architecture are discussed in this paper.

Producing a dedicated hardware for an algorithm improves the performance drastically and reducing the silicon area costs. This is obtained by eliminating all aspects of the design that are not necessary for the algorithm. On the other hand, the main drawbacks of an application specific architecture are increased marginal design costs due to lesser production volumes and reduced design flexibility as the hardware implementation cannot be changed after it has been manufactured. The main barriers of entry for application specific architectures are:

1. Identifying the scope of their application domains. This requires analysis of performance, power efficiency and economies of scale.
2. Reduce the design problems inherent for application specific architectures.

In support of their arguments, the authors present a detailed case analysis of 'The CryptoManiac Processor'. The architecture and its application specific optimizations are explained.

2.2.7. An FPGA based Forth Microprocessor [19]

Applications which use application specific FPGA along with a microprocessor have two distinct advantages:

1. Reduce the power consumption
2. Reduce the system costs by incorporating the microprocessor in the FPGA.

In this paper, a 16 bit FPGA based microprocessor called MSL16 is described which executes the 'Forth' programming language. This is based on stack architecture with each instruction occupying 4 bits leading to small instruction set, simple datapath and control and high code density.

MSL 16 consists of a 16 deep data stack for temporary variables and subroutine parameters and a T register holds the top element of the stack so that the top two elements of the stack are available to the ALU simultaneously. It also contains a 16 deep return stack to store subroutine return addresses, a instruction register which holds the 4-bit instructions to be executed, a PC and an IR which store the address of the next instruction and finally an ALU which takes operands from T and the top element of either DS or RS and returns the result to T.

Forth machines are suitable for embedding in FPGA applications because of good code density, easy customization, easy to handle development tools, high performance and small area.

2.2.8. Flexible Instruction Processors [20]

The authors introduce a Flexible Instruction Processor (FIP) for systematic customization of instruction processor design and implementation. General purpose

processors lose performance when dealing with custom operations and non-standard data. Customizing the processor is required in such cases. This can be done either by augmenting the processor with programmable logic for implementing custom instructions or by implementing the instructions using FPGAs. Application specific instruction processors provide another method of producing custom processors.

The unique features of FIP include:

- A modular framework based on processor templates that capture various instruction processor styles, such as stack-based or register-based styles.
- Enhancements of this framework to improve functionality and performance, such as hybrid processor templates and superscalar operation
- Compilation strategies involving standard compilers and FIP specific compilers, and the associated design flow
- Technology independent and technology specific optimizations such as techniques for efficient resource sharing in FPGA implementations

FIPs are assembled from a processor template with modules connected together by communicating channels. The template can used to produce different styles of processors such as stack-based and register-based. The parameters of a template are selected to transform a skeletal processor into a processor suited for its task. Possible parameterizations include addition of custom instructions, removal of unnecessary resources, customization of data and instruction widths, optimization of op-code assignments, and varying the degree of pipelining.

When a FIP is assembled, required instructions are included from a library that contains implementations of these instructions in various styles. Depending on which instructions are included, resources such as stacks, different decode units are instantiated. Channels provide a mechanism for dependencies between instructions and resources to be mitigated. This FIP framework has been implemented in Handel-C.

2.2.9. Power efficient flexible processor architectures for embedded applications [21]

A novel processor architecture is proposed in this paper which provides the flexibility needed in practice at a reduced power and performance cost. A novel protocol which combines an efficient, customized component with a flexible processor into hybrid architecture is proposed.

Based on the required flexibility, target technology and processor architecture are selected independent of their reuse considerations. Components benefiting from a custom hardware implementation are still implemented in their optimal architecture. Flexibility is added to the system as a separate programmable component, which can take over control in those cycles where functionality needs to change. This novel protocol allows for fine grain control which is needed since it is not known in advance which execution cycles of the hardware realization will have to be substituted by a new functionality on the flexible platform. The fine grain control is realized with a control flow inspection mechanism and an interrupt mechanism. The customized memory architecture is shared with the flexible component, solving the data transfer and storage bottleneck for multimedia applications.

All processor described above have static datapaths. The hardware is incapable of adapting to input tasks at run-time. Hardware is usually designed with sufficient resources for all possible types of applications expected to run on it. However, all tasks that require minimal resources and the tasks that require maximum resources pass through the same datapath, which reduces the overall utilization of hardware. This issue has been addressed in this thesis. Chapter 3 explains the superscalar pipelining concepts for which the problem can be defined clearly.

CHAPTER 3

SUPERSCALAR PIPELINED ARCHITECTURE

The scheme proposed in this thesis work is evaluated on the DLX architecture designed by Hennessey and Patterson as a representative architecture of most commercial processors. This chapter explains the architectural design of the DLX machine. It also presents the design concepts of a pipelined, superscalar architecture.

3.1 DLX Architecture

The DLX is a simple load-store architecture described in [1]. It is developed purely for academic interests, with an architecture similar to most commercial computers like AMD 29K, DECstation 3100, HP850, IBM801, Intel i860, etc.[1]. The DLX architecture consists of thirty-two 32-bit general purpose registers called R0, R1, R2, ... R31 where R0 always holds the value zero. The word size for the DLX is 32-bit. Integer data and floating point data of single precision is thus 32-bit, while double precision floating point data is 64-bit. The DLX uses the immediate and displacement addressing modes for data, which are stored in a 16-bit field. Main memory is accessed using a 32-bit address and it is byte addressable. The operations supported by the DLX are classified into four major types: ALU, branch, load-store and floating point operations. Table 1 lists the opcodes of these operations. The control instructions are jumps and branches, where branches are conditional which need to be evaluated before the branch is resolved. The floating point unit of DLX handles all floating point operations as well as integer operations of multiply and divide.

Figure 4 shows the scalar, pipelined implementation of DLX. It consists of five stages: Instruction fetch, instruction decode and register fetch, execute and effective address calculation, memory access and write-back stage.

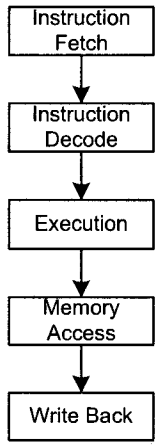

Figure 4: Pipeline stages in a DLX architecture

1. Instruction Fetch

The fetch stage is responsible for fetching the instruction to be executed. It fetches a new instruction at every clock cycle unless the pipeline is stalled. The DLX uses a special register called the Program Counter (PC) to store the address of the next instruction to be fetched. The PC is incremented by 4 to point to a sequential instruction stored in the next memory word. The fetched instruction is stored in a special register called the Instruction Register (IR), while a special register called the Next Program Counter (NPC) stores the address of the next instruction to be fetched.

2. Instruction Decode

This stage decodes the instruction stored in IR and accesses the register file to read registers that contain data. Since the DLX is a load-store architecture, operands are first loaded into registers using the load instruction and then operations are performed on them. These operands are read into two temporary registers (A and B). If immediate addressing mode is used, then it is sign-extended

Table 1 DLX instructions [1]

Instruction type/opcode	Instruction meaning
Data transfers	**Move data between registers and memory, or between the integer and FP or special register; only memory address mode is 16-bit displacement + contents of a GPR**
LB, LBU, SB	Load byte, load byte unsigned, store byte
LH, LHU, SH	Load halfword, load halfword unsigned, store halfword
LW, SW	Load word, store word (to/from integer registers)
LF, LD, SF, SD	Load SP float, load DP float, store SP float, store DP float (SP - single precision, DP - double precision)
MOVI2S, MOVS2I	Move from/to GPR to/from a special register
MOVF, MOVD	Copy one floating-point register or a DP pair to another register or pair
MOVFP2I, MOVI2FP	Move 32 bits from/to FP tegister to/from integer registers
Arithmetic / Logical	**Operations on integer or logical data in GPRs; signed arithmetics trap on overflow**
ADD, ADDI, ADDU, ADDUI	Add, add immediate (all immediates are 16-bits); signed and unsigned
SUB, SUBI, SUBU, SUBUI	Subtract, subtract immediate; signed and unsigned
MULT, MULTU, DIV, DIVU	Multiply and divide, signed and unsigned; operands must be floating-point registers; all operations take and yield 32-bit values
AND, ANDI	And, and immediate
OR, ORI, XOP, XOPI	Or, or immediate, exclusive or, exclusive or immediate
LHI	Load high immediate - loads upper half of register with immediate
SLL, SRL, SRA, SLLI, SRLI,	Shifts: both immediate(S__I) and variable form(S__); shifts are shift left logical, right logical, right arithmetic
S__, S__I	Set conditional: "__" may be LT, GT, LE, GE, EQ, NE
Control	**Conditional branches and jumps; PC-relative or through register**
BEQZ, BNEZ	Branch GPR equal/not equal to zero; 16-bit offset from PC
BFPT, BFPF	Test comparison bit in the FP status register and branch; 16-bit offset from PC
J, JR	Jumps: 26-bit offset from PC(J) or target in register(JR)
JAL, JALR	Jump and link: save PC+4 to R31, target is PC-relative(JAL) ot a register(JALR)

TRAP	Transfer to operating system at a vectored address
RFE	Return to user code from an exception; restore user code
Floating point	**Floating-point operations on DP and SP formats**
ADDD, ADDF	Add DP, SP numbers
SUBD, SUBF	Subtract DP, SP numbers
MULTD, MULTF	Multiply DP, SP floating point
DIVD, DIVF	Divide DP, SP floating point

before being stored in a register. Instruction decoding and accessing of register file is done concurrently due to fixed-width instruction format.

3. Execution and Effective Address Calculation

The instruction is issued to execution unit which performs the desired arithmetic, compare, logical or shifting operation. If it is a load or store instruction, then this stage performs effective address calculation for generating memory address from which or at which data is to be loaded or stored.

4. Memory Access

For load instructions, data is fetched from the memory address generated in the previous stage and loaded into load memory register, while for store instruction, data is written from specified register into memory. For branch instructions, the condition for branching is evaluated in the previous stage, and the PC is replaced or incremented based the result produced. ALU instructions are completed in this stage by writing the result of ALU operations into the desired register file location. This is commonly referred to as the MEM stage.

5. Write-back

Register file is updated with the data from Load Memory Register, and the load instruction is completed.

The scalar DLX pipeline can be extended to a superscalar pipelined version using superscalar concepts described in the next section. The number of pipeline stages and their functions remain similar. Section 3.3 describes the VHDL implementation of a superscalar and pipelined DLX architecture.

3.2 Generic Superscalar Pipeline

A superscalar pipeline is characterized by concurrent instruction processing and out-of-order execution. A superscalar pipeline parallelizes instruction execution by duplicating processing elements. Figure 5 shows the block diagram of a generic superscalar pipeline of width *s*. The pipeline consists of six main stages: instruction fetch, instruction decode, dispatch, execute, reorder and retirement. These stages perform tasks similar to those performed by the five-stage pipeline described for the DLX architecture. The fetch, decode and dispatch stages perform an in-order execution of instructions, the execute stage processes instructions in an out-of-order manner. The reorder stage forces the instructions to retire in an orderly fashion.

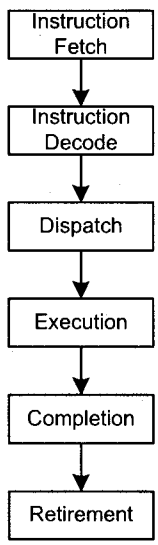

Figure 5. Generic Superscalar Pipeline Stages

3.2.1 Instruction Fetch

The objective of the instruction fetch stage is to fetch *s* instructions in every clock cycle. The IF stage employs mechanisms to maximize the input bandwidth of the pipeline to achieve this goal. A high input bandwidth is essential to achieve high instructions per cycle throughput. The fetch bandwidth is affected due to:

1. misaligned instructions

2. control instructions that alter the sequential flow of a program

Misalignment has been addressed dynamically in the IBM RS/6000 architecture by use of hardware logic that releases run-time control signals to the instruction cache to fetch misaligned instructions in a single memory access. Other techniques to reduce misalignment include static mechanisms employed at compiler time. Control instructions like jump and branch instructions change the program flow. These are dealt using branch prediction schemes that can accurately predict the next instruction to be fetched and attempt to keep the instruction fetch buffer filled with instructions.

3.2.2 Instruction Decode

The instruction decode stage deals with generating the control signals necessary for other modules to correctly execute an instruction. This includes separating individual instructions, establishing the instruction operation and location of operands and determining inter-instruction dependencies. For machines with a fixed instruction length, the task of separating instructions is trivial. The number of addressing modes and instruction types add to the complexity of the decoder. The decoder identifies dependencies between instructions and extracts parallelism between them. It employs a large number of comparators for determining dependencies. The decoder in CISC machines requires a highly intricate design. If the instruction set consists of instructions with variable lengths, then it is not possible to decode instructions in parallel. To reduce the time taken for the decoding, some commercial processors like the AMD K5™[24] use pre-decoders. The pre-decoders decode an instruction partially and communicate control bits along with the instruction to the instruction decoder.

3.2.3 Dispatch

At the dispatch stage, the following tasks take place:

- register renaming
- allocation of reservation units
- allocation of reorder buffer entries
- forwarding of instructions to the next stage

There are several types of execution units present in a superscalar pipeline, for processing different types of instructions. For example, integer operations are handled by integer units, while floating point operation are handled by floating point units. The dispatch stage is required to route an instruction to the appropriate execution unit. Instructions that have been decoded, but await one or more operands are placed in reservation units. Reservation units are multi-entry instruction buffers that are specific for each execution unit if implemented as distributed reservation units, or a single global multi-entry buffer if implemented as a centralized reservation unit. They keep track of instructions ready to execute and forward them to the execution unit to be executed once the required execution unit becomes available. Intel Pentium Pro [25] uses a centralized reservation unit, while PowerPC 620 [26] uses a distributed reservation unit.

3.2.4 Execute

The execute stage in a superscalar pipeline consists of one or more functional units or different types. Functional units are specialized and numerous in order to be able to execute instructions in parallel and in an efficient manner. The functional units that are generally present in most superscalar implementations are load-store units, integer units, branch units and floating point units. The number of these functional units is decided by the mix of instruction types expected to run on the machine. As the number of functional units is increased, there is an increase in hardware complexity due to increase in forwarding paths and interconnections required for routing operands to the appropriate execution unit.

3.2.5 Complete

In this stage results of executed instructions are written into desired registers. It is also responsible for completing instructions in sequential order. This is necessary to maintain the sequential nature of program execution. For this purpose, it uses a buffer called Reorder Buffer. The reorder buffer maintains a circular queue which enables an in-order retirement of instructions.

3.2.6 Retirement

Memory updates generally require more latency. An instruction that involves writing to memory is not complete until the memory operation is performed. The retirement unit performs this action and completes such instructions.

3.3 Superscalar, Pipelined DLX implementation in VHDL

The VHDL implementation of superscalar version of DLX is a two-width five-stage pipelined 32-bit architecture. It is capable of executing integer arithmetic and logical operations, compare, shift, jump and branch instructions. It does not contain a floating point unit. The architecture uses an Instruction Cache to store instructions loaded from memory. Figure 6 shows the pipeline stages in this implementation.

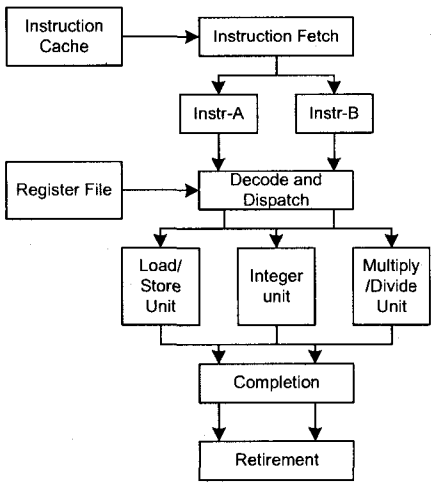

Figure 6. Superscalar, Pipelined DLX Implementation in VHDL

Each stage can process two instructions simultaneously. Figure 7 shows the block diagram of the integer unit. It is implemented as a 32-bit functional unit.

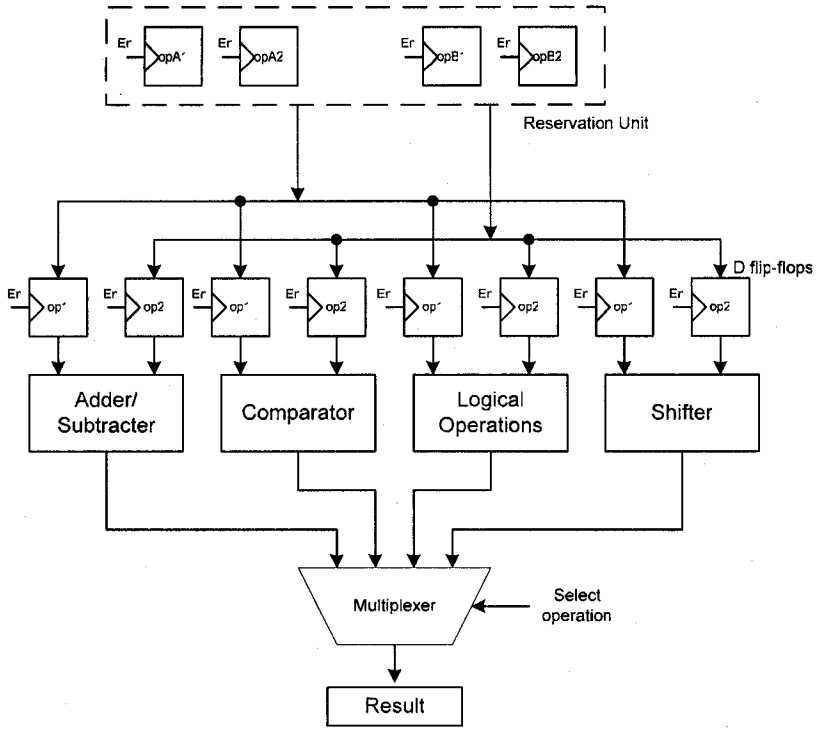

Figure 7. Block diagram of integer unit in VHDL implementation of superscalar DLX

The VHDL program takes a text file containing machine codes as input. It can be simulated using Active-HDL 7.1. Benchmark programs are usually present as assembly-level programs. Such benchmark programs for DLX cannot be directly used as input to the VHDL program. Figure 8 shows the data flow diagram while using the VHDL DLX processor emulator code. Benchmark programs with extension *.asm* are first converted to a text file with extension *.out* using a DLX assembler program called *dlxasm*[27] available freely. The *dlxasm* assembler converts DLX instructions into respective DLX machine codes. Each machine code is indexed by a 32-bit memory address in which the instruction is expected to be stored in a true hardware system. Format of the *.asm* and converted *.out* file is given in the Appendix. The *.out* file is used as input to the simulator engine that contains the VHDL code.

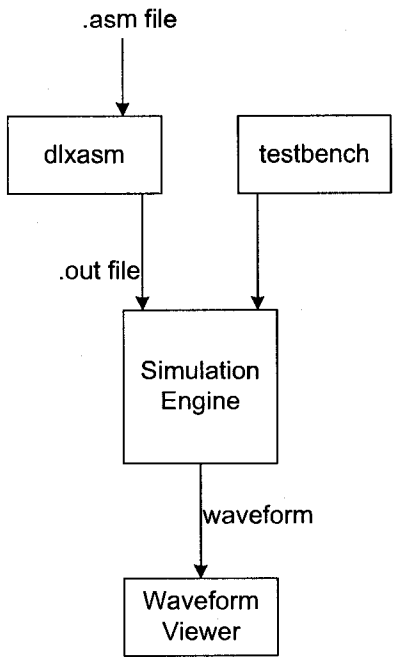

Figure 8. Dataflow diagram during simulation of VHDL implementation of proposed concepts

The simulation engine produces waveforms for signals that propagate in the processor. These are in the form of a Value Change Dump (.VCD) file and can be easily viewed using a waveform viewer.

Table 2[1] lists the average of MIPS dynamic instruction mix present in five SPECint2000 programs: gap, gcc, gzip, mcf, perl, and that present in five SPECfp2000 programs: applu, art, equake, lucas, swim.

Table 2 Average of MIPS dynamic instruction mix in SPECint2000 and SPECfp2000 benchmark suite

Instruction types	Average % of integer operations in integer benchmarks	Average % of integer operations in floating point benchmarks
Load-store	38%	22%
Add,sub,compare,shift,and,or,xor	45%	31%
Branch, conditional move, jump, call, return	16%	4%

Majority of the instruction mix consists of integer operations of add, subtract, compare, shift, and, or and exclusive-or. From these statistics, we can conclude that if there exists only one integer unit, then more often than not, a centralized reservation unit will be filled with waiting integer ALU instructions.

To cater to the high percentage of ALU instructions, it is necessary to include more than one ALU units. The number of ALU instructions can vary wildly from one benchmark suite to another. Therefore, unchecked addition of more ALU units can result in idle units or idle other functional units in the execution stage. Hence, a flexible scheme is proposed in this thesis that takes into account the observation that value of operands of ALU operations is not always as large as that accommodated by the word length of the machine. This scheme and all concepts associated with it are explained in the following chapter.

CHAPTER 4

CONCEPTS AND IMPLEMENTATION

A pipelined processor is designed so as to impart maximum throughput. The design decisions include determining the width of the pipeline (superscalar width), number of functional resources, and nature and degree of depth of the pipeline. This thesis concentrates on the execution stage of the pipeline. The execution stage consists of one or more execution units of different types and reservation stations in a centralized or distributed architecture. While designing the execution stage of a processor, it is extremely important to determine the optimum number of execution units of each type. This decision is typically based on applications served by the processor and the type of tasks that are expected to run on it. Execution units are provided to service all types of instructions present in the instruction set that need a computation unit. A generic instruction set consists of four types of instructions: ALU, Branch, Load and Store. The number of units allotted for each type of instruction has to be determined on the basis of example programs that will run on the machine and the performance expected. The number thus decided upon affects the space requirements, power usage and additional logic necessary for smooth functioning of these units in parallel.

There are several reasons for the design proposed in this thesis. The usage of an integer ALU unit was studied by running several benchmarks on a VHDL implementation of the DLX superscalar processor. Table 3 shows the results obtained.

Table 3 Usage of ALU units in benchmarks

Benchmark Program	# of instns	Time of simulation	# of ALU instns	# of ALU instns with 8-bit	# of ALU instns with 16-bit	# of ALU instns with 24-bit	# of ALU instns with 32-bit
ALUinstructionsPart1	22	93.5	22	10	4	8	0
ALUinstructionsPart2	33	137.5	33	14	10	4	5
BranchJump	85.5	85.5	10	0	0	3	7
BubbleSort	6477	12191.5	2741	658	1	708	1374
Dlx	18	61.5	9	0	0	5	4
LoadStore	30	119.5	5	1	1	2	1
MDUinstructions	39	198.5	27	8	9	7	4
PrimeNumber	1321	6595.5	718	1	22	360	335

On analysis, it can be seen that less than 50% of the ALU instructions use the entire data width of the ALU. Thus the usage of ALU is less than 100%. In any design, if additional ALU units are added to cater to larger number of input ALU instructions, then by projecting a similar usage statistic curve to these additional units, the overall ALU usage will only decrease.

The use of reservation stations encourages parallelism among ready instructions, waiting for resources. In an ALU intensive task, the number of such waiting instructions justifies the use of high number of resources, while in non-ALU intensive tasks, the usage of ALU units is minimal.

Also, in the older machines, floating point operations were performed using integer units. As the use of floating point operands increased, dedicated floating point units were introduced in the execution stage. In these machines, while executing a floating-point intensive task, the integer units are idle for most period of execution time. If the integer units had the capability to perform floating point operations on ready and waiting FP instructions, then the throughput would increase.

It is clear that the usage of execution units would increase if there was a technique to cater to different types of incoming instruction traffic. This thesis adds run-time flexibility to hardware modules for the purpose of accommodating as many instructions as possible in the execution unit. The exact extra hardware and logic required to do this

is designed, implemented and evaluated in Section 4.3, while the general concepts associated with the addition of flexibility are described below. These can be applied in any form to any application.

4.1 Block Slicing

'Block slicing' refers to the process of splitting a block into multiple modules. The concept of block slicing can be explained as follows:

A functional unit which is capable of performing an operation Ψ on two N-bit operands usually consists of N interconnected copies of units that can perform the operation Ψ on two 1-bit operands. Let a logic circuit capable of performing a certain operation on 1-bit operands be called a *unit*. When N units are interconnected so that they can concurrently perform the operation on N-bit operands, they form an N-bit *module*. In all implementations, N is known or is pre-set. Thus, the interconnection network, Γ between units that form the modules is static in nature. When operands of varying lengths are encountered, the value of N is required to be dynamic. In order to allow N to be a dynamic value determined at run-time, it is necessary to make the interconnection network flexible.

The network can be built to be completely flexible, but it is impractical to reprogram it before execution of every instruction. Instead, a degree of flexibility is allotted to it. For this, *m* units are connected together statically to form m-bit functional units. Let each m-bit unit be referred to as a *slice*. Each slice is capable of operating on m-bit operands. In a contemporary processor, if N-bit functional modules are present, then there will be N/m slices in a sliced architecture. For example, a processor containing one 64-bit ALU will now have four 16-bit slices (N=64 and m=16).

The interconnection network $\Gamma' \subseteq \Gamma$ between slices is now completely flexible, so that each slice can operate independently, or connect itself to more slices and operate concurrently with them. When two m-bit slices operate independently, they are capable

of executing two instructions simultaneously, provided the operands are m-bit. When two slices connect together, they form a 2m-bit functional module and can operate on one instruction with 2m-bit operands. Since there are N/m slices, when all slices are connected to each other, they can operate on N-bit operands as before.

When a module is thus separated into smaller parts, it is said to be 'sliced'. If a module is sliced into enough m-bit slices in the execution stage of a processor, all ready instructions requiring m-bit operands can be executed in parallel.

Based on the ready instructions encountered, slices are first allocated to each instruction. Once slice-allocation is decided, there are two functions associated with the process of allocation before the instructions are ready to be executed:

1. Directing the operands into the correct operand register slices, and
2. Directing the result correctly into an N-bit output register.

These functions can be performed by using decoders at the input and output of the execution unit. A truth table for the decoder can be easily developed and implemented as the internal circuitry for the decoder. Different execution units need different decoding functions as can be seen from the architecture explained in the next section.

4.2 Sliced ALU Implementation

When a sliced ALU is used, the stages in which an instruction undergoes processing are shown in Figure 9. The Resource Mapping is done by a unit called the 'Resource Mapper'. It is the only additional stage that gets added to the pipeline, but its latency is equivalent to a few logic gates, and hence it need not be pipelined as a separate unit, rather as a part of the dispatch pipeline stage. It is explained in detail in the next section.

4.2.1 Resource Mapper

This unit determines the number of slices required by an incoming instruction and allocates slices for all incoming instructions. For determining the number of slices required by an instruction, the resource mapper performs a function called 'zero-

checking'. This function determines the length of significant bits in both operands and returns the maximum of these two lengths as the number of slices required by the instruction. This can be achieved simply by using AND gates. The zero-checking function is slightly different for the shift operation, for which not only the number of significant bits of first operand are required, but also the value of the second operand. Using these values and a simple logic circuit, the number of units required by a shift instruction can be determined.

With each reservation unit is associated a register called the Resource Allocation Vector (RAV). The Resource Allocation Vector keeps track of slices allotted to the instruction stored in a reservation unit. In addition, the Resource Mapper uses a global register called the Resource Vector (RV). If there are m slices in the execution unit, then

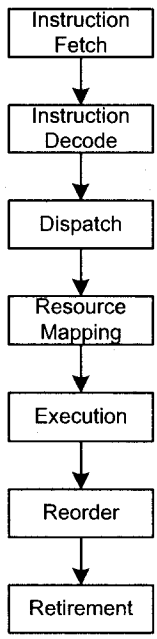

Figure 9. Processing stages for using a sliced ALU implementation

the RAV and RV are m-bit. Each bit in the RAV and RV indicate a status for slices of the execution unit as allocated/not-allocated. When a slice is allotted to an instruction, the bit in the respective location of the slice is set to 1. When an instruction finishes using

the slice, the bit is reset to 0. In absence of slicing, a similar process is followed by a reservation unit to issue instructions to an execution unit. The reservation unit checks the busy/available bit of a functional unit and issues an instruction to it if it is free. The Resource Mapper also issues an instruction to one or more slices of functional units and sets one or more bits at a time in the RAV of the instruction and global RV respectively.

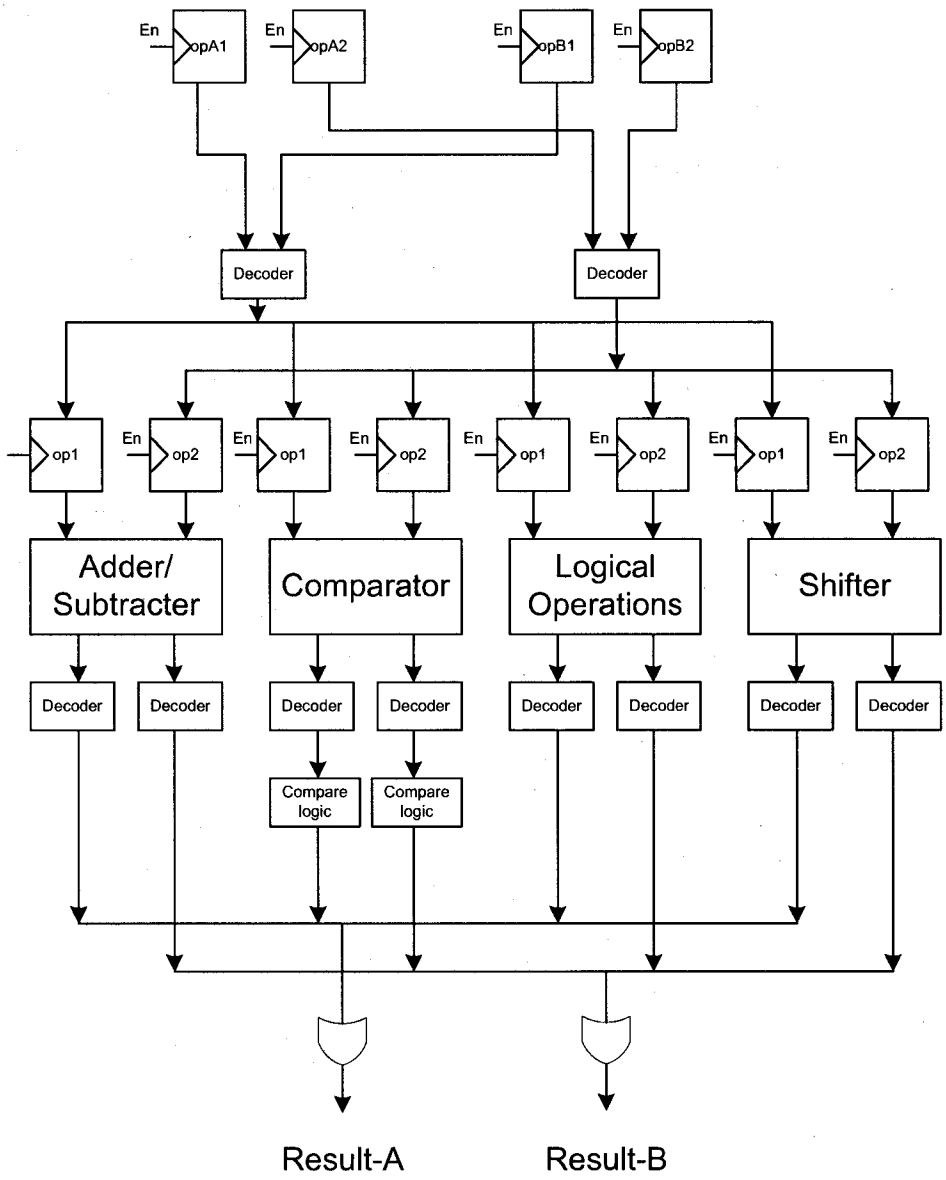

Figure 10. Block diagram of a sliced ALU

If the execution units are all known to finish the execution of an instruction in one clock cycle, then a global Resource Vector can be assumed to be an all-zero number at the beginning of every clock cycle, and is redundant. In this case, allocation is done by examining all ready instructions waiting for a resource and determining the number of slices required by each. In the situation where the ready instructions need more slices than available, the instructions can be prioritized based on instruction count and other instructions can be stalled. De-allocation is not necessary here. The Resource Vector will only be needed if some instructions take longer than a clock cycle to finish. Though unused in this thesis work, the use of Resource Vector has been proposed in view of future work, one instance of which is when integer slices are rearranged into a floating point pipeline, with a latency of more than one clock cycle.

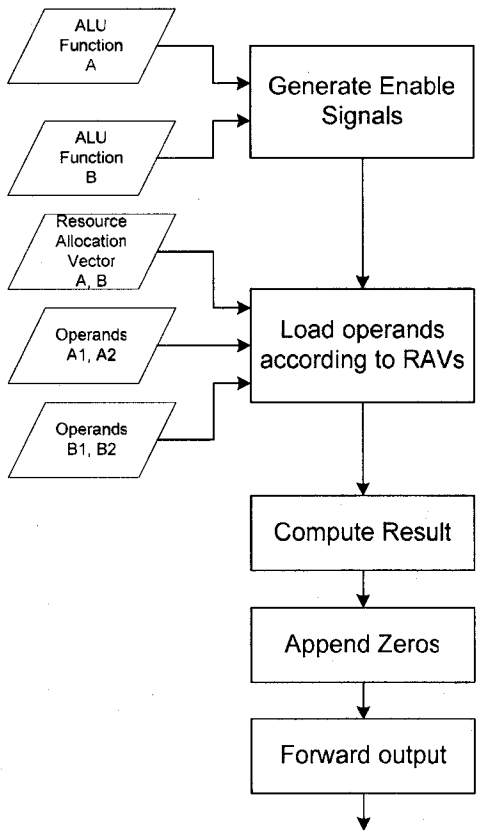

Figure 11. Steps of operation of a sliced ALU

Figure 10 shows the block diagram of a sliced ALU, while the flowchart in Figure 11 shows the steps in which a sliced ALU functions. The Enable signals in Figure 10 are fed to D-flip-flops so that only the appropriate part of the ALU functions, while the other parts retain their values. This leads to lower power consumption.

4.3 Architecture of integer execution units

The architecture of a sliced integer units that are used to execute different types of integer instructions is proposed below. The integer unit comprises of an adder/subtracter unit, a shifter, a logical unit and a comparison unit.

4.3.1 Adder/Subtracter Unit

Figure 12 shows the design of an adder/subtracter module, built using two slices of 4-bit adder/subtracter. The inputs required for the 4-bit adder subtracter are two 4-bit operands and a 1-bit operation *add/sub* ('0' for addition and '1' for subtraction). The adder/subtracter module is designed by interconnecting signals between the two slices and using multiplexers to enable it to operate at variable data length. It is capable of performing an addition and/or subtraction operation on the set of operands {X3...X0} and {X7...X4}. Once the operands are loaded into these registers, control signals *add/sub1* and *add/sub2* are given to the module. The control input *sel* indicates whether the two slices are to perform independently or concurrently. Multiplexer MUX1 determines the propagation of add/sub signal to the second slice, while Multiplexer MUX2 controls the cascading of carry out signal from the unit U3 to unit U4. Multiplexer-3 generates the overflow exception bits v1 and v2. After the output is produced, it is sign-extended in order to be passed on to the result register and subsequently stored.

When only one slice (say, slice-0) is to be used, the signals {S4..S7}, v2 and Cout7 are ignored, and vice-versa. When both slices are used for one operation, the appropriate signals are routed to the output.

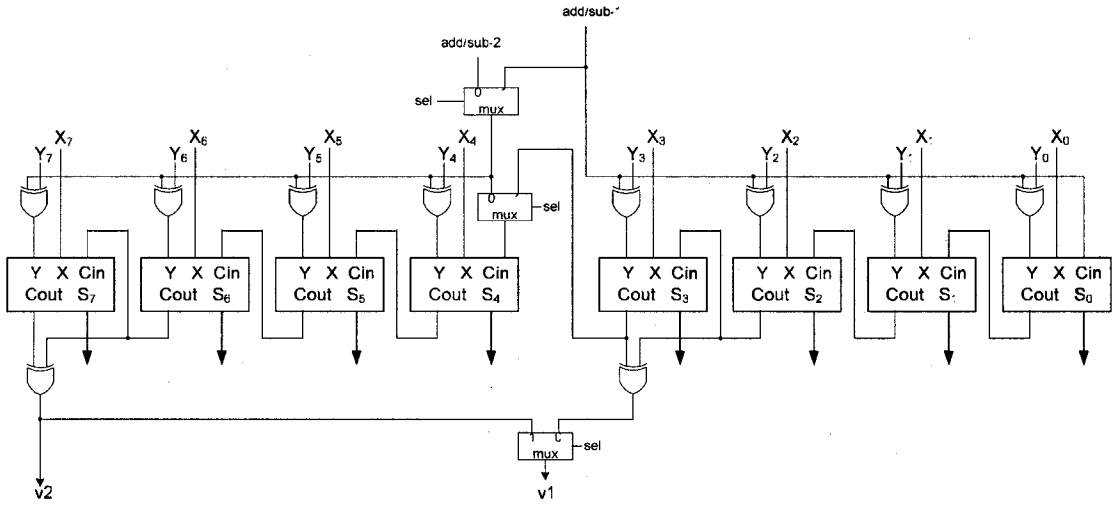

Figure 12. Two interconnected 4-bit adder/subtracter units forming one 2-slice adder/subtracter

This design can be extended to include numerous slices of the adder/subtracter unit. Figure 13 shows the interconnections of four adder/subtracter slices, each capable of operating on two 8-bit operands, resulting in a 32-bit sliced ALU. The block diagram of this flexible adder/subtracter unit is shown in Figure 14.

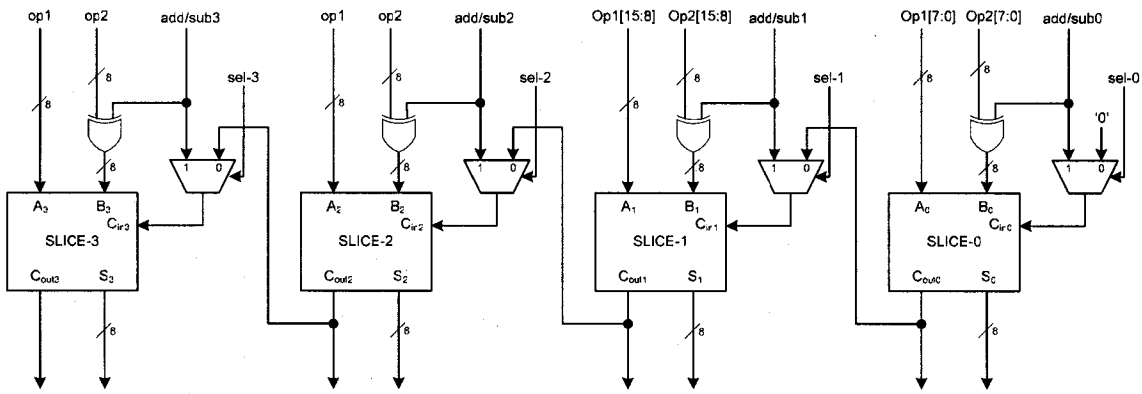

Figure 13. Architecture of flexible adder/subtracter unit

As explained before, there are two functions associated with slice-allocation:

1. Directing the operands into the correct operand register slices, and
2. Directing the result correctly into an N-bit output register.

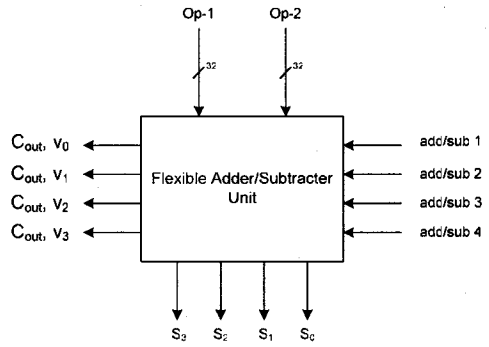

Figure 14. Block diagram of a flexible adder/subtracter unit

The input operands are initially present in N-bit operand registers. Let an ALU instruction with two input operand registers containing 8-bit values be ready for execution and be allotted *Slice-1* in Figure X. The operands have to be loaded at locations [15:8] of registers *op1* and *op2*. Similarly, the 8-bit result generated by *Slice-1* has to be directed to locations [7:0] of output register.

Table 4 Truth Table for decoder that directs the output of four slices into result register

RAV-3	RAV-2	RAV-1	RAV-0	Res-3	Res-2	Res-1	Res-0
0	0	0	0	0	0	0	0
0	0	0	1	MSB-0	MSB-0	MSB-0	S0
0	0	1	0	MSB-1	MSB-1	MSB-1	S1
0	0	1	1	MSB-1	MSB-1	S1	S0
0	1	0	0	MSB-2	MSB-2	MSB-2	S2
0	1	0	1	X	X	X	X
0	1	1	0	MSB-2	MSB-2	S2	S1
0	1	1	1	MSB-2	S2	S1	S0
1	0	0	0	MSB-3	MSB-3	MSB-3	S3
1	0	0	1	X	X	X	X
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	MSB-3	MSB-3	S3	S2
1	1	0	1	X	X	X	X
1	1	1	0	MSB-3	S3	S2	S1
1	1	1	1	S3	S2	S1	S0

The direction of input to appropriate input registers and of the output to a result register is done by the use of decoders. The truth table for a decoder that performs this

function is shown in Table 4. The RAV is the individual Resource Allocation Vector set for an instruction. There are two decoders, one for each instruction, which are input the RAVs for two instructions and outputs respective sign-extended result.

The adder/subtracter units along with input and output decoders constitute the complete flexible adder/subtracter. Area analysis for this module is made in section 4.4.

4.3.2 Compare Unit

The compare operation is required to be performed on both singed and unsigned operands, and requires a slightly different treatment for each. Figure 15 shows the block diagram of a comparator that can perform signed comparison or unsigned comparison based on a 1-bit control signal (0 for unsigned, 1 for signed).

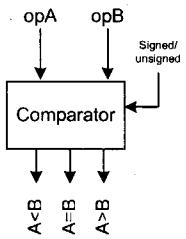

Figure 15. Block diagram of a comparator

This comparator can be designed as a minimal-delay circuitry, or it can be designed with minimal area constraint, depending upon the constraints imposed by the system. Figure 16 shows the use of such comparison units in a sliced comparator design.

Once sliced comparison is performed, the final result of compare operation is determined by a separate logic circuitry that takes into account the respective outputs of each compare slice. Therefore, the control signals for compare operations listed in Table 1 are made available to this unit. The decoder generates an output for the compare instruction. The logic equations that serve some of these functions are shown below.

Aeq ⇐ Aeq0 and Aeq1 and Aeq2 and Aeq3

Aneq ⇐ not Aeq

Agt ⇐ Agt3 or (Aeq3 and Agt2) or (Aeq3 and Aeq2 and Agt1) or (Aeq3 and Aeq2 and Aeq1 and Agt0)

Alt ⇐ Aeq nor Agt

Alteq ⇐ not Agt

Agteq ⇐ Aeq or Agt

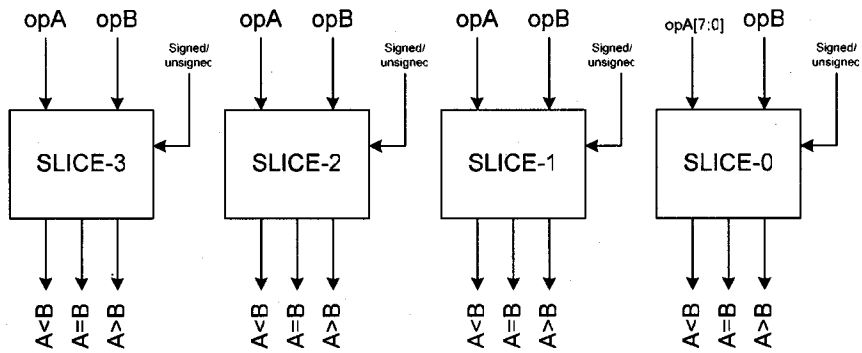

Figure 16. Four 8-bit compare slices for signed or unsigned comparison

Resource allocation vector for each instruction is also input to this unit, and equality is tested based on the allocation. For example, if the RAV for instruction A is 0011, the values returned by Aeq3 and Aeq2 are '1', while the values for Agt3 and Agt2 are '0'. Thus, the decoding functions that steer the output of sliced comparators into correct registers are different for equality and greater-than operation. The final bit output of the compare unit is determined by the equation for the function enabled by instruction decoder for that instruction. This is then concatenated with (N-1) leading zeros and returned as an output of the comparator unit.

4.4 Area analysis

This sliced ALU design requires additional hardware for decoders, multiplexers and added signals. For the implementation of 2:1 multiplexers used extensively in the design, transmission gates (pass transistor logic) can be used. These are designed using an NMOS and a PMOS transistor in a configuration that result in no static power consumption. Figure 17 shows a multiplexer implemented using pass transistor logic.

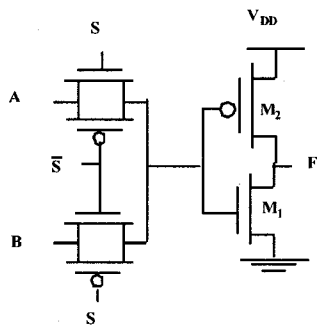

Figure 17. Multiplexer implemented using pass-transistor logic

The pass transistors add three NMOS and three PMOS gates to the hardware. To estimate the hardware used for decoders that perform direction of input and output signals into correct register slices, the average cost of decoders was computed in terms of logic gate equivalents. Table 5 lists the additional hardware used by various units in a sliced ALU.

On the whole, additional hardware introduced for implementation of slicing is minimal.

On performing a delay analysis, the maximum delay path of decoders is found to be equivalent to three gate propagation delays. Thus each decoder adds minimal delay to the execution datapath.

Table 5 Additional hardware used for slicing of ALU

	2:1 MUX	4:1 MUX	Logic Gates
Adder	4		
Adder Result Decoder			23
Comparator			
Comparator Result Decoder	8		44
Shifter	14		
Logical Operations			
Shifter and Logical Result Decode			23
RAV Decoder			16
Load Operand Decoder		16	
Total	**26**	**16**	**106**

4.5 Implementation of DLX Sliced Processor using VHDL

In order to evaluate the block slicing concept in a processor, it was implemented in a DLX pipeline using VHDL (VHSIC Hardware Description Language).

The first two stages of the DLX pipeline, the fetch and decode stage, are similar to those described in Section 3.2. The dispatch and execute stages differ from the original implementation, while the reorder and retirement units stay the same. The DLX processor has been implemented as a pipelined, out-of-order, superscalar processor of width two. Thus, there are at most two instructions in each stage of the pipeline at any given time, except the reorder unit.

Once valid operands are fetched in the dispatch stage and an instruction is ready to begin execution, the number of units required for the instruction is computed from the value of the operands. This is done by the simple zero-checking unit described in Section 4.2.1, which checks the number of leading zeros of operands. It gives the length of the significant digits of operands and hence the number of units required. For shifting operation, the number of units is computed by also considering the value of the second operand.

Once the zero checking is done, the Resource Mapper allocates execution unit slices to an instruction. In addition, the resource mapper also sets the control signals that slice an execution unit appropriately. The resource vector is a bit vector that indicates

the slices allocated to an instruction. For example, if instruction A is allocated slice number 1, then its 4-bit resource vector will be 0001. For instruction B with allocated slice numbers 2 and 3, the resource vector is 0110. Thus, the global resource vector during that clock cycle is 0111, indicating that only three slices of the execution units will operate, and the fourth slice will consume idle power.

Data is loaded into the operand registers at the rising edge of the clock. Due to block slicing, the resource mapping control signals slice the execution unit and the ALU gives at most two outputs (ALU Output A and ALU Output B) by simultaneous execution of two instructions. These results are stored into their respective reorder buffer entries, and forwarded if necessary for the next clock cycle.

The fetch stage is set so as to fetch the next set of instructions when an instruction is issued to an execution unit. Thus, when instruction-level parallelism exists in a program, the fetch stage is also speeded up and the total time of execution of a program decreases. In the absence of any additional instruction-level parallelism, the time of execution of the program remains the same as that in a non-sliced processor.

Chapter 5 presents the simulation results of benchmark programs on the VHDL implementations of the DLX machines with non-sliced and sliced ALU unit respectively.

CHAPTER 5

RESULTS

The objective of slicing is to increase the utilization and number of functional units dynamically. Increasing number of functional units leads to an increase in parallelism of execution which contributes to speed-up.

The performance criteria used for evaluating the concept of slicing are speed-up, throughput, utilization and power. These criteria are widely used for comparison of different architectures. To evaluate the performance of the block slicing concept with respect to these factors, a hardware code for the DLX processor was developed using VHDL and tested with benchmark programs. Benchmark programs were obtained from various sources from internet resources. These were assembly level programs written for the DLX machine. .asm files containing benchmark programs were converted to .out files using the package *dlxasm* [27] and then run on the VHDL code of the sliced processor. Instead of developing the code from scratch, the freely available VHDL package *dlx-vhdl*[28] was used as base code and it was suitably modified for the proposed architecture. Section 4.5 describes the VHDL processor code.

Throughput is given by number of instructions completed per unit time. It can also be related to the number "Instructions Per Cycle (IPC)", where the unit of time is a clock cycle. Considering that a new instruction is fetched every clock cycle, the number of fetch cycles indicates the input stream to the architecture and the number of instructions committed per fetch cycle indicates the output stream of the processor. The throughput is then given as:

$$IPfC = \frac{Total\ Number\ of\ instructions\ Committed}{Total\ Number\ of\ Fetch\ Cycles} \qquad (5.1)$$

The speed-up is computed with respect to the DLX architecture without the processor modifications for block slicing. Thus, speed-up is given as:

$$Speed-up = \frac{Time\ of\ execution\ on\ non-sliced\ DLX\ architecture}{Time\ of\ execution\ on\ sliced\ DLX\ architecture} \quad (5.2)$$

Resource utilization at the bit-level is given by the % of resource used during time of execution. Resource utilization can be given in terms of the ratio of number of times the resource slices were completely used to the total number of times the resource was accessed.

Power consumed during execution of two sequential operations is evaluated using the Xilinx Xpower tool that is included with Xilinx ISE. The power-delay product is then used to compare the non-sliced and the sliced architectures.

5.1 Time of Execution and Speed-Up

The above mentioned criteria were evaluated on ten benchmark programs and are presented below.

Table 6 Results of evaluation of Time of Execution and Speed-up

Benchmark Program	Time of execution (us)		Speed-up	Gain
	non-sliced	sliced		%
ALUinstrutions-1	93.5	47.5	1.968	49.198
ALUinstrutions-2	137.5	95.5	1.440	30.545
DLX	61.5	58.5	1.051	4.878
LoadStore	119.5	119.5	1.000	0.000
PrimeNumber	6595.5	6471.5	1.019	1.880
supscal	63.5	39.5	1.608	37.795
MDUinstructions	198.5	191.5	1.037	3.526
BranchJump	85.5	67.5	1.267	21.053
NtoK	76.5	62.5	1.224	18.301

Table 6 presents the speed-up obtained for the benchmark programs by listing time of execution of each benchmark on a non-sliced and sliced processor and using eqn.2.

5.2 Efficiency

Table 7 presents the efficiency of use of ALU slices. In a non-sliced implementation, each time the ALU is accessed, both potential slices are accessed. In a sliced ALU, each time two instructions are executed in parallel, they are assumed to use two slices each, resulting in entire length of ALU being used.

Let,

NS_{ALU} = Number of times ALU is accessed in non - sliced implementation

S_{ALU} = Number of times ALU is accessed in sliced implementation

$NS_{ALU-Slice}$ = Number of times potential ALU slices are accessed in non - sliced implementation

$S_{ALU-Slice}$ = Number of ALU slices accessed in sliced implementation

p = Number of times ALU instructions executed in PARALLEL in sliced implementation

n = Total Number of ALU instructions

$NS_{ALU-Slice}$ (Column5) is given as:

$$NS_{ALU-Slice} = NS_{ALU} \text{ (Column2)} \times 2$$

Also, $S_{ALU-Slice} = 2 \times S_{ALU}$

Table 7 Efficiency

Benchmark Program	# of ALU accesses		# of parallel	# of ALU slices accessed		# of ALU	Efficiency	Gain in
	NS_{ALU}	S_{ALU}	executions	$NS_{ALU-Slice}$	$S_{ALU-Slice}$	instructions	e_S	Efficiency
ALUinstrutions-1	22	13	9	44	26	22	0.846	69.23%
ALUinstrutions-2	33	31	2	66	62	33	0.532	6.45%
DLX	9	7	2	18	14	9	0.643	28.57%
LoadStore	5	5	0	10	10	5	0.500	0.00%
PrimeNumber	718	681	37	1436	1362	718	0.527	5.43%
supscal	14	8	6	28	16	14	0.875	75.00%
MDUinstructions	27	26	1	54	52	27	0.519	3.85%
BranchJump	21	14	7	42	28	21	0.750	50.00%
NtoK	16	15	1	32	30	16	0.533	6.67%

Thus, Efficiency ε is given by:

$$\varepsilon_{NS} = \frac{n}{NS_{ALU-Slice}}$$

and $$\varepsilon_{S} = \frac{n}{S_{ALU-Slice}}$$

5.3 Throughput

Table 8 shows the throughput of both implementations in terms of instructions per fetch cycle.

Table 8 Throughput in terms of Instruction Per Fetch Cycle

Benchmark Program	# of fetch cycles		# of instructions	IPfC		Gain in
	non-sliced	sliced		non-sliced	sliced	IPfC (%)
ALUinstrutions-1	23	14	22	0.957	1.571	64.286
ALUinstrutions-2	34	32	33	0.971	1.031	6.250
DLX	27	25	18	0.667	0.720	8.000
LoadStore	30	30	30	1.000	1.000	0.000
PrimeNumber	1693	1660	1321	0.780	0.796	1.988
supscal	17	11	16	0.941	1.455	54.545
MDUinstructions	71	70	39	0.549	0.557	1.429
BranchJump	34	29	28	0.824	0.966	17.241
NtoK	34	33	24	0.706	0.727	3.030

5.4 Power-Delay Product

For estimation of power consumption, the Xilinx XPower tool was used with synthesizable designs of sliced ALU and non-sliced ALU. The ALU is capable of performing addition/subtraction, shift, compare and logical operations. Every combination of two different operations was selected and simulated with worst case 16-bit operands. The operations of addition and comparison were found to consume most power. The ALU designs were then analyzed for power consumption during execution of the operations of addition and comparison of 16-bit operands sequentially on a non-sliced ALU and parallelly on a sliced ALU.

Table 9 shows the power-delay product during this analysis.

Table 9 Power Delay Product for execution of two worst-case operations for two 16-bit worst-case operands

	Power (mW)	Delay (ns)	Power-Delay Product
Non-Sliced ALU	431	20	8620
Sliced ALU	604	10	6040

Figure 18 shows a snapshot of waveforms simulated on DLX processor for the ALU integer benchmark ALUinstructions-Part1, with the fetch registers, commit signals and ALU issue signals shown. Figure 5.1(a) shows the simulation run for a DLX processor with non-sliced ALU and figure 5.1(b) shows the simulation run for the DLX processor with sliced ALU.

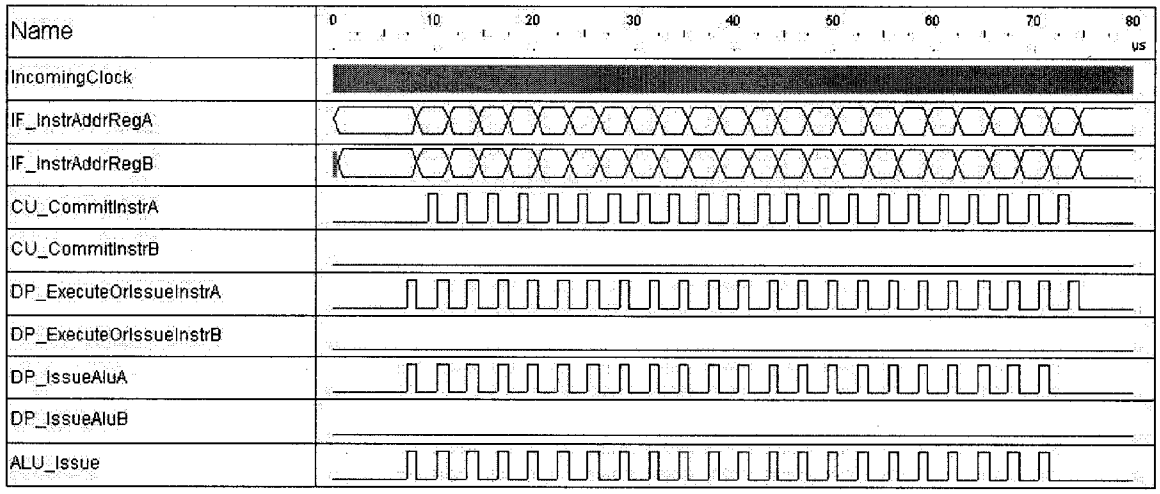

Figure 18 (a)

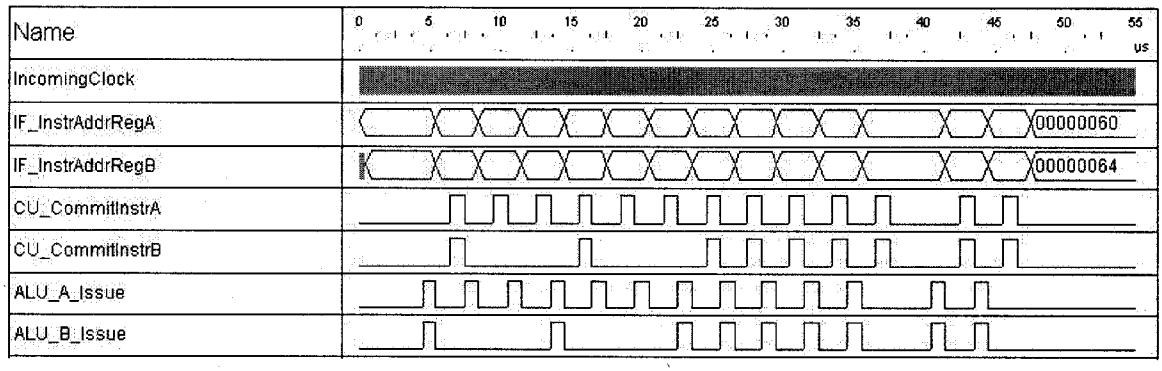

Figure 18(b)

Figure 18 Waveforms of simulations for ALUinstructions-Part1.out for DLX processor with (a) non-sliced ALU and (b) sliced ALU

IF_InstrAddrRegA and IF_InstrAddrRegB are address registers used by Instruction Fetch stage. Each contains an address of an instruction to be fetched. Whenever the contents of these registers change, the instructions present at the addresses stored by the registers are fetched by the Fetch unit. Thus, at most two instructions (Instruction A and Instruction B) are fetched at a time. If, in a previous clock cycle, only one instruction (Instruction A) is able to execute, then contents of IF_InstrAddrRegB are transferred to IF_InstrAddrRegA, and IF_InstrAddrRegB fetches a new instruction. Both instructions fetched are then decoded in the Decode Unit. The instructions are ready to execute when all their operands are fetched. Ready instructions are issued to execution units when units are available. The issue to ALU unit is signaled by ALU_Issue signal. If instruction A is to be issued, then DP_IssueAluA is high, and if instruction B is to be issued, then DP_IssueAluB is high. Both signals cannot be high simultaneously for a processor with a non-sliced ALU. However, if the nature of operands allows it, both signals will be high simultaneously for a processor with a sliced ALU. After instructions are executed, their results are stored in destination registers and the instruction is marked for retirement using the commit signals CU_CommitInstrA and CU_CommitInstrB. The two-width pipeline is capable of committing at most two instructions at a time.

The fetch address registers are loaded after an instruction is marked for issue by the dispatch unit. Thus, an instruction is fetched, decoded nd dispatched in the From Figure 18(b), it can be seen that there are nine instances when both ALU_A_Issue and ALU_B_Issue were high, which indicates that during nine cycles, the ALU instructions present in reservation units were executed simultaneously.

CU_CommitInstrA and CU_CommitInstrB committed two instructions at a time in nine instances in the sliced ALU processor, while it committed at most one instruction at a time in a non-sliced ALU processor.

The simulation waveforms show that slicing extracts instruction parallelism present in the program, and reduces instruction stalls that occur due to resource bottlenecks.

The gain percentages presented in last columns of all result tables give an indication of unresolved parallelism present in programs that was only extracted after slicing.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

The concept of resource slicing was implemented in the DLX processor using VHDL. Sliced resources process greater number of instructions without the need to add extra hardware resources. The sliced resource implementation was evaluated with respect to speed-up, throughput, power and utilization of the integer unit.

From the results thus obtained, it can be observed that by addition of one low-latency stage, the Resource Mapping and minimal hardware, it is possible to obtain a speed-up and higher efficiency of execution. The number of functional units required to be pipelined in a superscalar pipeline can also be reduced if the task running on the processor allows it. For a generic processor that runs a variety of different applications, each requiring different number of functional units, this can provide a flexible scheme for efficient execution.

The Intel MMX architecture was also developed with the purpose of parallelizing execution of instructions on data with smaller width than word size of the processor. The sliced architecture, if evaluated with MMX-type data will also perform similarly. Unlike the MMX, the sliced architecture will not require additional MMX-specific instructions and will dynamically slice itself into multiple modules to process the data.

6.2 Future Work

It is necessary to evaluate the performance enhancement obtained at varying superscalar widths on more benchmarks than used here. This will help in determining

the optimal number of slices required for different applications. This number can then be used to design sliced processors for most efficiency.

Block slicing is a general concept that can be applied in a variety of forms to modules other than functional units. It may be applied to registers and caches. It is required to design a suitable hardware to address, identify and access sliced data when stored in sliced registers and caches. A complete sliced processor will be obtained once work is performed for slicing these modules.

REFERENCES

[1] John L. Hennessy and David A. Patterson, "Computer Architecture: A Quantitative Approach", Morgan Kaufmann Publishers, Third Edition, 2002.

[2] David J. Lilja, "Measuring Computer Performance, A Practitioner's guide", Cambridge University Press, 2000.

[3] Luigi Dadda, "The Evolution of Computer Architectures", IEEE CompEuro 1991.

[4] Charles Moore and CH.TING "Minimal Instruction Set Computer", Fourth Dimensions, January 1995.

[5] Anthony Fong, "HISC: A High Level Instruction Set Computer", 7th European Simulation Symposium, 406-410, Society for Computer Simulation, October 95.

[6] J-P LeBouquin, IBM Microelectronics ZISC, "Zero Instruction Set Computer, Preliminary Information", WCNN, San Diego, CA 1994.

[7] Wayne Wolf and Jorgen Staunstrup, "Hardware/Software co-design Principles and Practice", Kluwer Academic Publishers, 1997.

[8] M.H. Lipasti and J.P. Shen, "Superspeculative Microarchitecture for Beyond AD 2000", IEEE Computer, September 1997.

[9] J.E. Smith and S. Vajapeyam, "Trace processors: Moving to Fourth-- Generation Microarchitectures", IEEE Computer, September 1997.

[10] G. S. Sohi, S. Breach, and S. Vijaykumar, "Multiscalar processors", in Proceedings of the 22nd Annual International Symposium on Computer Architecture, June 1995.

[11] S. Kaxiras, D.C. Burger, J.R. Goodman. "DataScalar: A Memory-Centric Approach to Computing", Journal of System Architecture (JSA), special issue on Microprocessor Architecture, June 1999.

[12] Burger D., Goodman J., "*Billion-transistor architectures*", Computer, September 1997.

[13] Doug Burger, James R. Goodman, "Billion-Transistor Architectures: There and Back Again", IEEE Computer, 2004.

[14] Yale N Patt, Sanjay J Patel, Marius Evers, Daniel H. Friendly, Jared Stark, "One Billion Transistors, One Uniprocessor, One Chip", IEEE Micro, pp. 51-57, September 1997.

[15] Michael J Wirthlin, Brad L. Hutchings, "A Dynamic Instruction Set Computer", IEEE Symposium on FPGAs for Custom Computing Machines, 1995.

[16] Alessandro De Gloria, "VISA: A Variable Instruction Set Architecture", ACM SIGARCH Computer Architecture News, Vol. 18, Issue 2, 1990

[17] Chandra Shekhar, Raj Singh, A.S. Mandal, S.C.Bose, Ravi Saini, Pramod Tanwar, "Application Specific Instruction Set Processors: Redefining Hardware-Software Boundary", Proceedings of the 17th International Conference on VLSI Design, IEEE.

[18] Chris Weaver, Rajeev Krishna, Lisa Wu, and Todd Austin , "Application Specific Architectures: A Recipe for Fast, Flexible and Power Efficient Designs", International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'01), November 2001.

[19] P.H.W. Leong, P.K. Tsang and T.K. Lee, "A FPGA Based Forth Microprocessor", Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa Valley, California USA, pp. 254-255, 1998

[20] Shay Ping Seng, Wayne Luk, Peter Y.K. Cheung, "Flexible Instruction Processors'" CASES 2000, November 17-19,2000, San Jose, California.

[21] Frederik Vermeulen, Francky Catthoor, Lode Nachtergaele, Diederik Verkest, Hugo De Man, "Power-Efficient Flexible Processor Architecture for Embedded Applications", IEEE Transactions on VLSI Systems, Vol 11, No.3, June 2003.

[22] Lecture slides by Minyi Guo, The University of Aizu, available online at the URL: http://www.u-aizu.ac.jp/~minyi/course/para2001.pdf

[23] Internet Resource: abrak.doc

[24] AMD® K5™ processor, at URL: http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/20092.pdf

[25] Intel® Pentium® Pro page at www.intel.com

[26] D. Levitan, T. Thomas, and P. Tu., "The powerpc 620 microprocessor: A high performance superscalar risc microprocessor", Proceedings of the 40th IEEE Computer Society International Conference, pg. 285-291, 1995.

[27] Compiler for DLX instruction set, *dlxasm* package available at URL: http://www.ashenden.com.au/designers-guide/DG-DLX-material.html

[28] VHDL-DLX package available freely at URL: http://www.rs.e-technik.tu-darmstadt.de/TUD/res/dlxdocu/SuperscalarDLX.html

VITA

Graduate College
University of Nevada, Las Vegas

Shruti Ravikant Patil

Home Address:
969 East Flamingo Road Apt#101
Las Vegas, Nevada 89119

Degree:
Bachelor of Engineering, Computer Engineering, 2004
University of Mumbai

Special Honors and Awards:
Member, Tau Beta Pi, Nevada Chapter, Initiated in Fall 2005
Recipient of the National Talent Search Scholarship(India), 1998
Rank 6, Regional Mathematics Olympiad, 1998, Mumbai
Recipient of Bombay Talent Search Award, 1997

Publications:
"HAUNT-24: Hierarchical, Application Confined Unique Naming Technique", IEEE Conference Proceedings of Fifth International Conference on Intelligent Systems Design and Applications (ISDA 2005), 8-10 Sept. 2005, Poland.

"Simultaneous Column Minimization-Encoding approach for Serial Decomposition", IEEE Proceedings of International Conference on Computational Intelligence and Multimedia Applications Conference (ICCIMA), August 2005, Las Vegas, NV, USA

"Branch Prediction by Checking Loop Terminal Conditions", Information Systems: New Generations (ISNG) Conference Proceedings, April 2005, Las Vegas, NV, USA

"Neutron Detector Characteristics in Dead Time Experiments", presented at 2005 American Nuclear Society Student Conference, April 15, 2005. Awarded the second prize as Best Student Paper Presentation.

Thesis Title: Maximizing Resource Utilization By Slicing Of Superscalar Architecture

Thesis Examination Committee:
Chairperson, Dr. Venkatesan Muthukumar, Ph. D.
Committee Member, Dr. Emma Regentova, Ph. D.
Committee Member, Dr. Shahram Latifi, Ph. D.
Graduate Faculty Representative, Dr. Ajoy Datta, Ph. D.