ΔΙΠ
1420

ΣΧΕΔΙΑΣΜΟΣ ΑΡΧΙΤΕΚΤΟΝΙΚΗΣ ΣΥΝΟΛΟΥ ΕΝΤΟΛΩΝ ΚΑΙ ΥΛΟΠΟΙΗΣΗΣ VHDL ΕΝΟΣ ΕΠΕΞΕΡΓΑΣΤΗ VLIW

Από τον

Γεώργιο Παπαδόπουλο

Διπλωματική εργασία προς μερική πλήρωση των
προϋποθέσεων για την απόκτηση του πτυχίου του

Ηλεκτρονικού Μηχανικού και Μηχανικού Υπολογιστών

Πολυτεχνείο Κρήτης

Χανιά, 2002

Επιβλέπων καθηγητής :

Διονύσιος Πνευματικάτος

Εξεταστική επιτροπή :

Διονύσιος Πνευματικάτος
Απόστολος Δόλλας
Γεώργιος Σταμούλης

Instruction Set Architecture and VHDL Implementation Design of a VLIW Processor

ABSTRACT

Historically we can notice a continuous increasing demand for more computing power, both for general and for special purpose applications. Improvements in processor performance come from two main architectural features: faster semiconductor technology coupled with higher integration of components, and exploiting parallelism. Parallelism is a key element in achieving high performance in processors, where different parts of the computation should be executed in parallel. Some methods for exploiting parallelism include pipelining and multiple processors. However, recent high performance processors have depended on Instruction Level Parallelism (ILP) to achieve high execution speed. ILP processors achieve their high performance by causing multiple operations to execute in parallel using a combination of hardware and software techniques. Two particular types of such processor styles are Superscalar and VLIW processors. VLIW and superscalar implementations have many similarities while the basic idea is the same for both. They both require an instruction stream analysis to exploit the available ILP. Their difference lies on the fact that in superscalars this analysis mostly occurs in the hardware during run-time, while in VLIW designs this is explicitly a compiler's role. Thus, a VLIW implementation achieves the same effect as a superscalar one, but the VLIW design is freed from the most complex parts of high-performance superscalar design.

This thesis goal is the study of both the Superscalar and VLIW architectures, focusing mostly on VLIW hardware and compiler issues, to define the ISA of a 4-way integer VLIW processor, and to implement it in the VHDL hardware description language. Defining the Instruction Set Architecture, in this processor design I include the basic design issues and organization techniques of the VLIW architecture. It is organized in a five-stage pipeline, and a complex bypassing network to handle resulted data hazards. In order to verify the correctness of the design, I created an assembler that translates an assembly code into bytecode, which is used as data for the instructions memory. Furthermore, I developed a software simulator of the processor that executes the same program bytecode. The VHDL code was verified using functional simulation, comparing the results with those of the software simulator.

Table of index

List of Figures and tables

Chapter 1

1 INTRODUCTION

Historically we can notice a continuous increasing demand for more computing power, both for general and for special purpose applications. Improvements in processor performance come from two main architectural features: faster semiconductor technology and exploiting parallelism. Parallelism can at first be achieved by overlapping of different parts of execution. This is achieved with pipelining, which takes advantage of only temporal parallelism. Pipelining is now universally implemented in high-performance processors. However, concurrency is the key element to achieve high performance computing and thus a higher level of parallelism is necessary to achieve this and has been issued in many ways, such as Data Level Parallelism, Instruction Level Parallelism, Algorithm Level Parallelism, Thread Level Parallelism etc.

Parallel processing on multiprocessors, multicomputers and processor clusters has traditionally involved a high degree of programming effort in mapping an algorithm to a form that can better exploit multiple processors and threads of execution. Such reorganization has often been productively applied especially for scientific programs. The general-purpose microprocessor industry on the other hand has pursued methods of automatically speeding up also for non-parallel programs without major restructuring effort. This lead to the development of Instruction Level Parallel (ILP) processors that try to speed up program execution by overlapping the execution of multiple instructions from an otherwise sequential program.

Two main approaches for instruction level parallelism are the Superscalar architecture and the VLIW architecture. They both require the same instruction stream analysis in order to exploit the available parallelism, but they differ in the way this analysis occurs. Superscalar processors perform this analysis during run-time. They can improve performance for all types of operations while they also preserve the architectural compatibility. This compatibility advantage is the key of the success for many desktop microprocessors such as the x86 architecture that dominates the desktop computer market. But the x86 is now recognized as a deficient instruction set while the recent superscalar implementations turned out to be very complicated. VLIWs seem to be a quite promising direction in microprocessor architecture. Their big advantage is that a highly parallel implementation is simpler and cheaper to build in VLIW architecture than an equivalent superscalar processor, as the scheduling to exploit the available parallelism is explicitly a software responsibility.

Because VLIW architecture transfers most of the complexity from hardware to software, better and more efficient compilers are required. Currently, some of the highest performance Digital Signal Processors are of the VLIW architecture. These types of applications are less difficult to compile for VLIW. Some VLIW DSP architectures include the StarCore SC140 from Agere Systems and Motorola, the TMS320C6x series from Texas Instruments, the Carmel Core from Infineon, etc. Finally, VLIW compiler technology has made major advances during the last decade. Thus, during the last two years there have been great efforts in building desktop microprocessors, like the Transmeta Crusoe processor [3] and the most recent Intel Itanium processor, both of which have special hardware to support x86 emulation.

In this thesis I will present and analyze the VLIW architecture and discuss some issues that characterise such processors showing both their advantages and the drawbacks. I will also show some of the design aspects of VLIW architecture building an integer VLIW processor. First, I present a thorough overview of high performance processor architectures, discussing the two main architectural approaches that exploit instructions level parallelism, superscalar and VLIW. In chapter 3, I analyse in detail some main issues about the VLIW architecture, such as compilation techniques, design aspects, object compatibility issues etc. Then, in the fourth chapter, I describe some organization issues of the VLIW architecture performing a comparison with those of the superscalar approach, and showing the distinction between VLIW and the static superscalar approach.

Finally, in chapters 5 and 6 I present the implementation of an integer VLIW processor, providing all the necessary information about its organization and architectural design, and the way it was tested. At first I designed the architecture and the instruction set, and I designed a non-pipelined single cycle implementation, describing its organization. I also wrote an assembler that translates the assembler language code to the machine's bytecode, in order to easily write programs for processor testing. Moreover, I created a software simulator in C language that simulates the instruction execution of the processor, in order to verify the processor execution results. After I validated the proper functionality of the non-pipelined implementation, I designed the final pipelined version of the processor, adding the necessary pipeline registers and implementing other required pipeline hardware. The last chapter concludes this thesis, while I present a general description of the Intel Itanium processor based on the IA-64 (EPIC) architecture, and the Transmeta Crusoe processor, a VLIW processor compatible with the x86 architecture.

Chapter 2

2 OVERVIEW OF ILP ARCHITECTURES

The most commonly used way in achieving parallelism is pipeline, where multiple instructions are overlapped in execution. With Instruction Level Parallelism, multiple instructions can be issued per cycle by a single processor. These multiple instructions can also be pipelined to achieve an even more extensive parallelism. ILP processors achieve their high performance by allowing multiple operations to execute in parallel using a combination of compiler and hardware techniques. One particular style of processor design is Very Long Instruction Word (VLIW), which tries to achieve high levels of instruction level parallelism by executing long instruction words composed of multiple operations. Another approach is superscalar processors, that issue various numbers of instructions per cycle. Parallel processors is another way used to achieve high level of parallelism, where a collection of processing elements cooperate to execute identical operations simultaneously. However, parallel processors are out of the scope of this thesis.

In this chapter I will firstly present a sort description of the pipeline implementation technique, as a global characteristic of all processor designs, which takes advantage of temporal parallelism by using pipelined functional units. Moreover, I will discuss these two processor designs and also figure out the main issues of each one of them.

2.1 Pipelined Processors

Pipelining is an implementation technique whereby multiple instructions are overlapped in execution, and it is the key used to make fast CPUs. It is sometimes compared to a manufacturing assembly line in which different parts of a product are being assembled at the same time although ultimately there may be some parts that have to be assembled before others are. Even if there is some sequential dependency, the overall process can take advantage of those operations that can proceed concurrently. In the world of processors each step in the pipeline completes a part of the instruction. Each of these steps is called a pipe stage or a pipe segment. These stages are connected one to the next to form a pipe. The throughput of the pipelining is determined by how often an instruction exits the pipeline.

Pipelining is a technique that exploits parallelism among instructions in a sequential instruction stream. It yields a reduction in the average execution time per instruction. This reduction can be obtained by decreasing the clock cycle time of the pipelined machine or by decreasing the number of clock cycles per instruction, or both. Typically, the goal of pipelining is a decreased CPI and thus provides increased throughput, which reaches one instruction per cycle when the pipeline is filled, though it can also be used to improve the clock speed too.

2.2 Superscalar Architecture

Superscalar processors are uniprocessor organizations capable of increasing machine performance by executing multiple scalar instructions in each cycle. In these machines, an instruction fetching unit can fetch more than one instruction at a time from the instruction cache. Then an instruction decoding logic decides when instructions are independent and thus executed simultaneously by sufficient execution units which are able to process several instructions at one time. These are the key to superscalar execution. The execution units may also be pipelined.

Typical superscalar architectures shift most responsibilities to the hardware. Although they are able to run unmodified object code of former sequential architectures, they certainly need compiler assistance to execute programs efficiently. Hardware complexity limits superscalars to a very small run-time scheduling window. Therefore the compiler has to schedule code in such a way that operations within the scheduling window are independent from each other as much as possible.

Instruction-level parallelism can be extracted either statically (at compile-time) or dynamically (at run-time). It is generally known that changes in control flow due to conditional branches can severely restrict ILP. Furthermore, branch prediction and speculative execution can further expose parallelism. Superscalar architectures mostly use dynamic scheduling that transfers all ILP complexity to the hardware.

2.2.1 Static Scheduling

The basis for statically scheduled superscalar processors comes from the field of horizontally-microcoded machines and Very Long Instruction Word (VLIW) architectures. Statically scheduled superscalar processors exploit instruction level parallelism with a modest amount of hardware by exposing the machine's parallel architecture in the instruction set. The compiler potentially has an infinite instruction window and uses global program knowledge, dependences, and resource constraints in constructing each instruction schedule. Thus, instructions are scheduled statically across many basic blocks and are fetched by the processor within single fetch blocks. Instructions that are selected to be independent are then issued and executed in program order. Thus, there is no overhead during run-time to schedule instructions, and the hardware is simple.

However, the effectiveness of statically scheduled superscalar processors is limited because of many unpredictable delays caused in many cases. For example, because the compiler does not know the address of some memory access operations, a static scheduler may not be able to achieve load bypassing efficiently. Moreover, it is unable to determine detailed cache behaviour at compile time. Thus, instruction issue is stalled when a functional unit conflict occurs or when an instruction has a multiple cycle latency. Finally, the effectiveness of static scheduling is also limited by the presence of conditional branches because speculative computation in compilers is too complex and not much powerful and it prevents efficient parallelization.

The Multiflow and Impact compilers are two basic examples of software schedulers that perform global code motion. Both rely on static branch prediction techniques to generate traces of execution that form the basis for global code motion between multiple basic blocks. The Multiflow compiler uses trace scheduling to perform code motion, whereas the Impact compiler includes a variant of trace scheduling called superblock scheduling. Other techniques for static global scheduling are the *Enhanced Percolation Scheduling*, the *RS/600 Global Scheduling* and the *Region Scheduling*.

An example of such a superscalar processor is the TigerSHARC from Analog Devices [2]. TigerSHARK is a DSP superscalar processor where all the scheduling is determined by the compiler and performs in-order instruction execution.

2.2.2 Dynamic Scheduling

In order to generate efficient schedules, compilers are given machine descriptions that specify the functional units, their latencies, and any issue restrictions. Even with these detailed machine descriptions, the compiler cannot get a complete picture of the dynamic behaviour of the machine. The memory system and branch architecture are two components whose behaviour cannot be fully represented by the machine description because they introduce instruction behaviour that is variable at run-time. Dynamic scheduling has the ability to address each of these issues by performing scheduling at run-time, when this information becomes known.

Decode		Execute			Write		Cycle
I1	I2	I1	I2	I2	I1	I2	1
I3	I4	I3	I4	I4	I3	I4	2
I3	I4	I3	I4	I4	I3	I4	3
	I4		I4	I4		I4	4
I5	I6	I5	I6	I6	I5	I6	5
	I6		I6	I6		I6	6
							7
							8

Figure 2. 1 : In-Order-Issue with In-Order-Completion Diagram.

Decode		Execute			Write		Cycle
I1	I2						1
I3	I4	I1	I2				2
	I4	I1		I3	I2		3
I5	I6			I4	I1	I3	4
	I6		I5		I4		5
			I6		I5		6
					I6		7

Figure 2. 2 : In-Order-Issue with Out-Of-Order-Completion Diagram.

Multiple instruction execution occurs when the hardware issues independent instructions from a window of dynamic instructions. To maintain scalar code compatibility, all instruction scheduling is done from this window by the hardware. Unlike to statically scheduled processors, in dynamically scheduled processors three primary issue policies are supported; in-order issue with in order completion, in-order issue with out-of-order completion and out-of-order issue. The first issue policy is the one implemented in the static superscalar machines and is described in **Figure 2. 1**. With out-of-order completion, as shown in **Figure 2. 2** instructions can be pipelined within each functional unit, but in order to preserve program correctness, results must be written in correct order. Thus, a reorder buffer can be used to keep track of the original instruction order and to support recovery from exceptions and speculative execution.

Decode		**Window**	**Execute**			**Write**		**Cycle**
I1	I2							1
I3	I4	*I1, I2*	I1	I2				2
I5	I6	*I3, I4*	I1		I3	I2		3
		I4, I5, I6		I6	I4	I1	I3	4
		I5		I5		I4	I6	5
						I5		6

Figure 2. 3 : Out-Of-Order-Issue with Out-Of-Order-Completion Diagram.

The most aggressive issue policy is implemented with out-of-order issue, which provides these machines the ability to simultaneously execute instructions from multiple basic blocks and not in program order. This method is described in **Figure 2. 3**. These instructions are buffered in a lookahead window and decoding may continue on subsequent instructions even if prior instructions have not been executed. However, this buffering is still performed in program order, and the issuing from that window to the functional units is the one performed out-of-order. Moreover, data dependencies must still be preserved so the processor must determine which instructions in the window can be issued to the functional units and executed. Buffering within the processor supports the conditional evaluation of instructions that are executed before previously fetched branches (speculative execution), increasing the opportunities for the hardware to find instructions to issue in parallel. The advantage of this technique is that it provides a larger number of instructions from which the processor can find independent instructions to exploit machine parallelism.

The main advantage of typical superscalar processors, where the scheduling is performed during run-time, is that it provides backward object code compatibility. Even an old compiled sequential code can be run, as the code scheduling for the specific architecture is performed by the processor itself. Although all these hardware abilities overcome some static scheduling drawbacks and make compilers much simpler, dynamic scheduling has also some shortcomings. The additional hardware, necessary to look far ahead in the instruction stream to detect and schedule independent operations out of order, is costly and complex. Finally, in contrast to the compiler capabilities, the hardware can only analyze a small window of dynamic instructions during each cycle, thus limiting the possible candidates for parallel issue.

2.3 VLIW Architecture

Out of order speculative execution comes at a significant hardware expense. The complexity and non-scalability of the hardware structures used to implement these features could significantly hinder the performance of future processors. An alternative solution to this problem is to simplify processor hardware and transfer some of the complexity of extracting ILP to the compiler and run time system the solution explored by VLIW processors.

Joseph Fisher, who coined the acronym VLIW [5] characterized such machines as architectures which issue one long instruction per cycle, where each long instruction called a MultiOp consists of many tightly coupled independent operations each of which execute in a small and statically predictable number of cycles. However, the multiple number of such operations is also fixed. Unlike its successor (superscalar architecture), in the VLIW architecture, all scheduling is static. This means that the complex task of grouping independent operations into a MultiOp and being handled as a result of ILP is done by a compiler or binary translator. Thus, the processor freed from the cumbersome task of dependence analysis has to merely execute in parallel the operations contained within a MultiOp. This leads to simpler and faster processors implementations

This is also the main advantage of VLIW. Hardware complexity is reduced greatly since the executable instructions are generated directly by the compiler that are then processed as "native code" by the execution units present in the hardware. An important thing to note is that the compiler used here is not the same as the one used in HLL (High Level Language) code compilation. This compiler is VLIW specific. It recompiles the program source code into the executable VLIW instruction code, which is then passed to the processor. Thus, a

VLIW compiler is needed as an integral part of the VLIW system. **Figure 2. 4** shows how instructions are packaged into Very Long Instruction Words and build the instructions for VLIW machines.

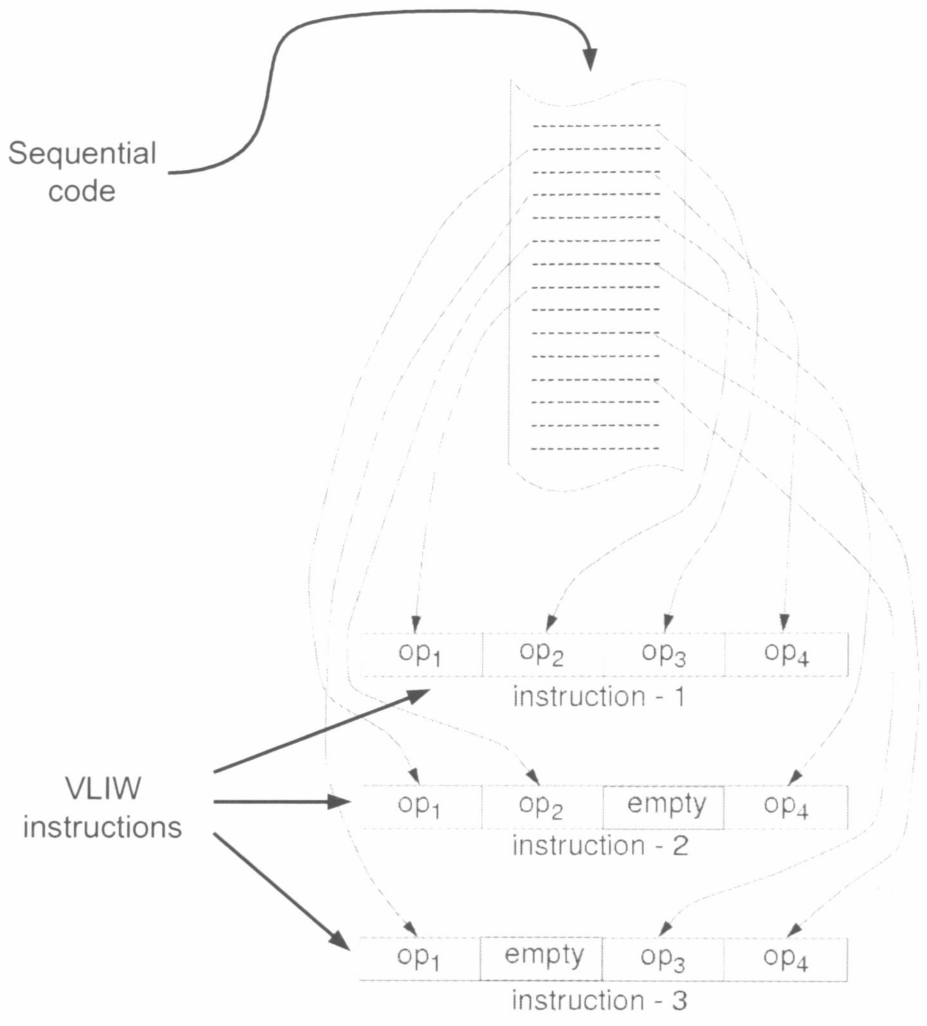


Figure 2. 4 : Detection of parallelism and packaging of operations into instructions is done, by the compiler, off-line.

VLIW machine are in some respects similar to vector machines, but they are also different. Vector machines perform the same operation on a vector of data while the VLIW machines have to have many more branch statements in the instructions, which is a key difference.

The long instruction word called a MultiOp consists of multiple arithmetic, logic and control operations each of which would probably be an individual operation on a simple Reduced Instruction Set Computing processor. Thus, VLIW is sometimes viewed as the next step beyond the RISC architecture, which also works with a limited set of relatively basic instructions and can usually execute more that one instruction at a time (a characteristic referred to as superscalar). Since the hardware complexity is moved to the software, the challenge in VLIW systems is to design a compiler or pre-processor that is intelligent enough to decide how to build the very long instruction words. If dynamic pre-processing is done as the program is run, performance may be a concern.

VLIW architecture technology had predated existing Superscalar technology that proved more useful due to greater compatibility with traditional architectures. However different, both architectures were based on the same ideology of "multiple-instruction" execution, referred to as Instruction Level Parallelism. Since superscalar technology was more compatible with traditional architectures, VLIW systems quickly became less popular as Superscalar systems started dominating the market. Two companies were founded in 1984 to build VLIW based mini supercomputers. One was Multiflow, started by Fisher and colleagues from Yale University, which delivered the Trace 200, 300 and 500 series machines. The other was Cydrome founded by Bob Rau, who was another VLIW pioneer, and his colleagues. In 1987, Cydrome delivered its first machine, the 256 bit Cydra 5, which included hardware support for software pipelining, a feature that can be found in Intel Itanium processors today. Unfortunately, the early VLIW machines failed commercially owing to which Multiflow closed in 1990 and Cydrome in 1998.

Since then recent improvements to VLIW architectures have lead to their rebirth and re-emergence into the high-performance computer systems industry. Some of the notable VLIW processors of recent years are the Crusoe processor from Transmeta, the Trimedia media processor from Philips, the TMS320C62x DSP's from Texas Instruments and the IA-64 or Itanium from Intel.

EPIC - IA-64 ISA

EPIC stands for Explicitly Parallel Instruction Computing, which comprises an evolution of traditional VLIW architectures developed by Intel and Hewlett Packard, and IA-64 is an instruction set architecture based on EPIC. In this design style, the interface between hardware and software is designed to enable the software to exploit all available compile-time information, and efficiently deliver this information to the hardware. The EPIC constructs provide powerful architectural semantics, and enable the software to make global optimizations across a large scheduling scope, thereby exposing available instruction level parallelism to the hardware. The hardware takes advantage of this enhanced ILP, and provides abundant execution resources.

Despite Intel's marketing claims, EPIC is a VLIW implementation, probably in order to ease compatibility with current x86 architecture. The easiest way to achieve this is to have a small part of the chip that can read the old complex x86 instruction and expand it into a single VLIW instruction group. This allows for an on-chip emulator to run old code, and it is easiest to do by converting one instruction into a long instruction word.

The first product based on the EPIC design technology is the Itanium processor member of the Itanium processor family. The first version was code-named "Merced" and was originally scheduled for 1997. Finally, Intel has already announced the next version of the Itanium processor family with the code-name "McKinley".

Chapter 3

3 ORGANIZATION ISSUES OF SUPERSCALAR AND VLIW PROCESSORS

VLIW and superscalar processors are both instruction-level parallel architectures that require the same instruction stream analysis in order to exploit the available ILP. Their difference lies on the way this analysis occurs. However, statically scheduled superscalar processors have many similarities to VLIW processors, but they are still different. In this chapter, I will firstly present the main organization issues of both static and dynamic superscalar architectures. Then I will describe the organization of VLIW architecture performing a comparison with typical superscalar implementation and showing the distinction between them, and also discuss the difference between VLIW and static superscalar designs. Finally, I will extend some organization issues on VLIW processors.

3.1 The Organization of a Typical Superscalar Processor

As I have already discussed in Chapter 2, most superscalar processor implementations share fundamental complexities. It is the need for the hardware to discover and exploit instruction-level parallelism. The only exception is the limited portion of those superscalar processors that are based explicitly on the compiler to do the whole scheduling task. Superscalar processors may also be static or dynamic. However, what is mostly used is the dynamic superscalar architecture, which implements all the features of a typical superscalar processor. On the other hand, static superscalars use part of the features of typical superscalar processors, simplifying the hardware. Actually the main difference between static and dynamic superscalars is focused in the way instructions are executed. Static superscalars perform in-order execution, while dynamic ones perform out-of-order execution.

Firstly, I will briefly describe the static superscalar architecture and its features. Then, I will present the dynamic superscalar describing all the features of a typical superscalar processor, where some of whose are also implemented in static superscalars too.

3.1.1 Static Superscalar (in-order)

A superscalar machine can be either statically or dynamically scheduled. Typical superscalar processors perform all the scheduling tasks dynamically, that is dependency and structural checking, branch prediction and out-of-order execution. In case scheduling is made explicitly by the compiler, superscalar hardware is freed from the complex parts of code scheduling and the execution of the instructions scheduled is performed in program order. This comprises a fully static superscalar design with a much simpler hardware than a typical dynamic superscalar design. However, in a common approach for a static superscalar processor some tasks are still performed during run-time.

Static superscalar processors generally forgo the advantages of dynamic instruction scheduling in favour of a high clock rate. A number of instructions are fetched and decoded in parallel. The hardware is responsible to identify and eliminate possible structural hazards and data dependencies among instructions. Moreover, to get more parallelism, control dependencies due to updates of the program counter, especially due to conditional branches, have to be overcome. Thus, branch and jump prediction may be performed before issuing instructions to the functional units. Following instruction fetch, and decode, instructions are inspected and arranged according to their type (the functional unit that they will use). Then, provided operand is ready, instructions are issued for execution. During this entire process, instructions are not allowed to pass one another. They are issued and complete the execution in program order.

3.1.2 Dynamic Superscalar (out-of-order)

As already mentioned, dynamic superscalar processors share all the features of typical superscalar processors. **Figure 3. 1** shows a block diagram of the hardware organization of a typical superscalar processor and **Figure 3. 2** describes the superscalar execution procedure. The major parts this implementation consists of are the execution units, which are also called as functional units (FU), the instruction buffers, decoders and dispatcher that feed the execution units with operations, and register file and reorder buffer that feed the FUs with operands. Functional units may be a collection of other units like integer ALUs, floating point ALUs, load/store units and control/transfer units.

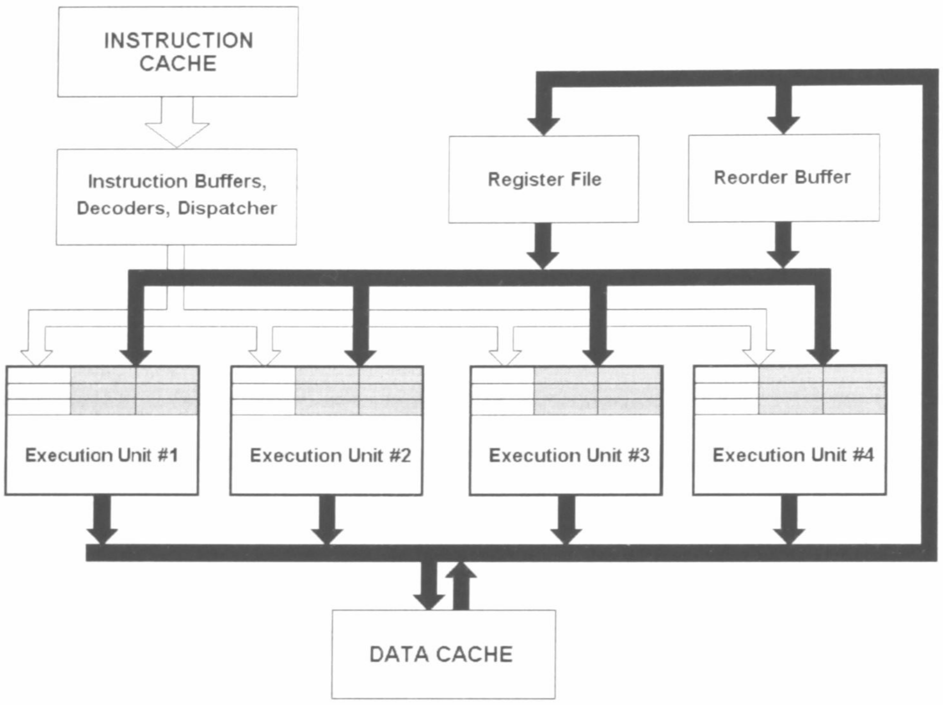


Figure 3. 1 : Block Diagram of a Typical Superscalar Implementation. It uses reservation stations to implement the instruction window, and reorder buffer to maintain the state of the register file.

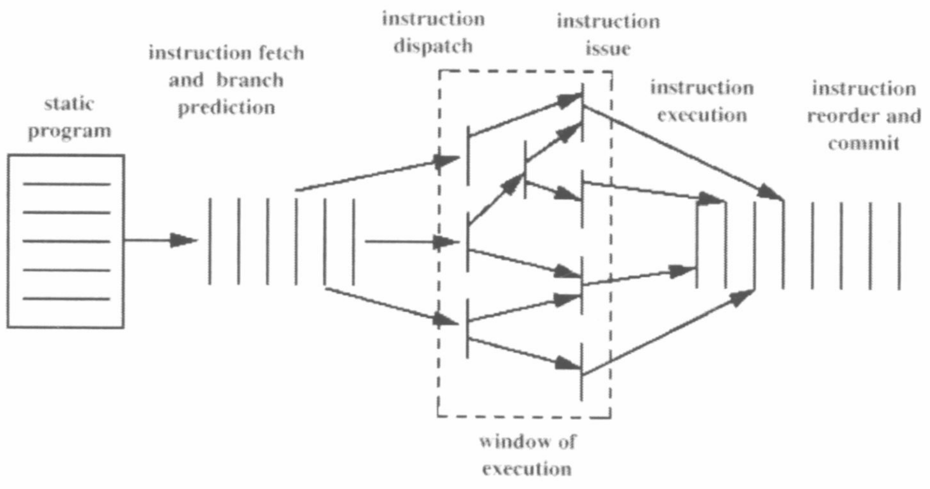


Figure 3. 2 : Superscalar Execution Diagram. It performs out-of-order execution.

To sustain the execution of multiple instructions per cycle the fetch phase must be able to fetch multiple instructions per cycle form the instruction cache. The number of instructions fetched per cycle should at least match the peak instruction decode and execution rate and in some cases it has to be higher. This extra margin of instruction fetch bandwidth allows for instruction cache misses and for situations where fewer than the maximum number of instructions can be fetched. Thus instruction buffers are used so that to build up a "stockpile" to carry the processor through periods when instructions fetching is stalled or restricted.

The job of the decode phase is to set up one or more execution *tuples* for each instruction. An execution tuple is an ordered list containing information that define an operation to be executed, the identities of storage elements where the input operands reside (or will eventually reside) and locations where the instruction's result must be placed. The instruction dispatcher examines a window of instructions contained in a buffer. The dispatcher looks at the instructions in the window and decides which ones can be dispatched to functional units. It tries to dispatch as many instructions at once as is possible; in other words it attempts to discover the maximum amount of instruction level parallelism. Higher degrees of superscalar execution with more functional units require wider windows and more sophisticated dispatcher.

As already mentioned in chapter 2, there are three main policies for issue and completion of instructions: in-order issue and completion, in-order issue with out-of-order completion, and out-of-order issue and completion. There are two main methods to implement an instruction window in order to support out-of-order execution:

- *Reservation stations:* Reservation stations were first proposed as a part of Tomasulo's algorithm. They partition the instruction window by functional units. Only instructions at each station are considered for scheduling on its assigned functional unit, which simplifies control logic.

- *Central instruction window:* A central instruction window keeps all unissued instructions in one unit, regardless of the type of the instruction. It determines which ready-to-run instructions are scheduled onto what functional unit, which thereby complicates control logic.

The default instruction fetching method is to increment the program counter by the number of instructions fetched, and to use the incremented program counter to fetch the next block of instructions. However, in case of branch instructions, the fetch mechanism must be redirected to fetch instructions from the branch target. Conceptually, the processor must wait until the branch is resolved before it can begin to look for parallelism at the target of branch. To avoid waiting for conditional branches to be resolved, high-performance superscalar implementations implement branch prediction. With branch prediction, the processor makes an early guess about the outcome of the branch and begins looking for parallelism along the predicted path. The act of dispatching and executing instructions from a predicted, but unconfirmed, path is called speculative execution.

Unfortunately, branch prediction is not always accurate. Thus, with speculative execution, it is necessary to be able to *undo* the effects of speculatively executed instructions in the case of a mispredicted branch. Some implementations simply prevent instructions along the predicted path from progressing far enough to modify any visible processor state, but to gain the most from speculative execution it is necessary to allow instructions along the predicted path to execute fully.

Moreover, out-of-order issue and completion requires that the state of the register file be maintained. In order to maintain the state of the register file and also to be able to undo the effects of full, speculative execution, a hardware structure called a reorder buffer is the most common approach employed. This structure is an adjunct to the register file that keeps track of all the results produced by instructions that have recently been executed or that have been dispatched to execution units but have not yet completed. The reorder buffer provides a place for results of speculatively executed instruction and is also used to maintain proper instruction ordering. It renames each destination register instance to a unique identifier. An associative lookup maps this identifier to the entry when results are written. After the execution completion the values are written back to the register file and the identifiers are discarded. In case of a conditional branch, when it is resolved, the results of the speculatively executed instructions can be either dropped from the reorder buffer (branch mispredicted) or written from the buffer to the register file (branch predicted correctly).

The final phase of the lifetime of an instruction is the commit state, where the effects of the instruction are allowed to modify the logical process state. The purpose of this phase is to implement the appearance of a sequential execution model even though the actual execution is very likely non-sequential, due to speculative execution and out-of-order instruction completion.

Another important characteristic of superscalar processors is that of pipelining. In an *m*-issue typical superscalar processor, the instruction decoding and execution resources are increased to form essentially *m* pipelines that operate concurrently. This pipeline structure is shown in **Figure 3. 3.** Moreover at some pipeline stages, the functional units may be shared by multiple pipelines. At first, the fetch pipeline, reads instructions from the instruction cache, decodes them, performs register renaming and dispatching to the instruction issue buffers, the reservation stations for example. Then, *m* pipelines allow overlapped instruction execution by issuing instructions to the corresponding functional units. The out-of-order execution capability of superscalar processors results in variable length pipelines for each functional unit. For example, a floating-point unit may require more time to complete execution and thus more pipeline stages.

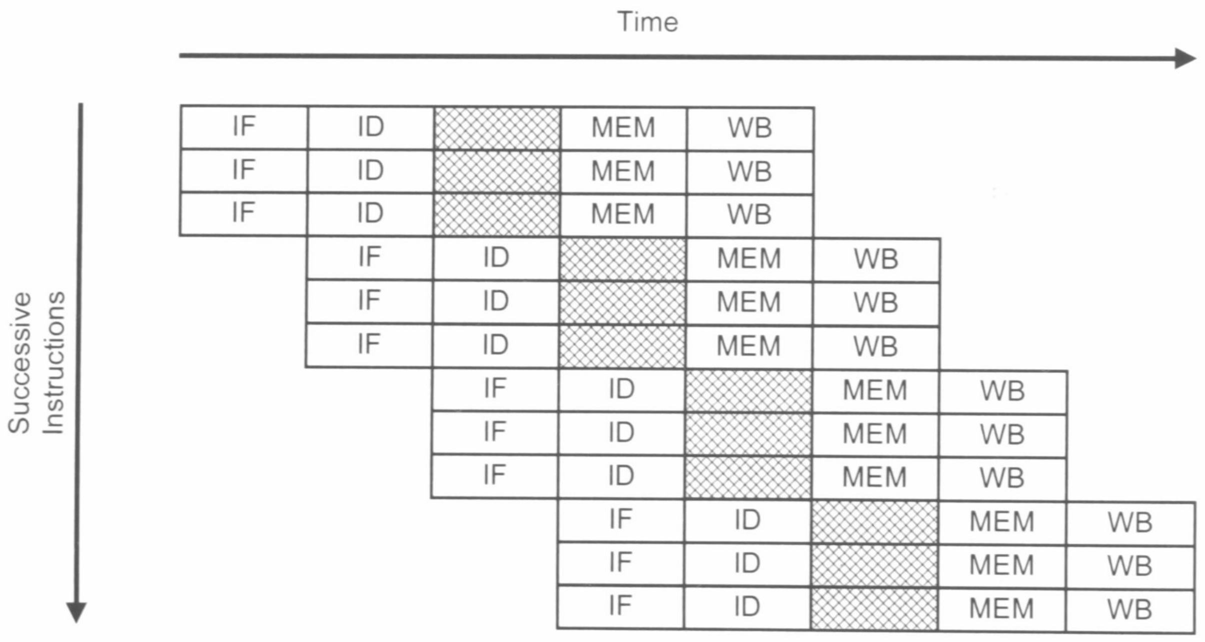


Figure 3. 3 : A typical Superscalar Pipeline Structure. Three pipelines operating concurrently. At some pipeline stages the FUs may be shared by multiple pipelines and also of variable length.

3.2 VLIW General Organization Issues - Advantages over Typical Superscalar Implementation

A VLIW implementation achieves the same effect as a superscalar implementation, but the VLIW design does so without the most complex parts of high-performance superscalar design.

VLIW instructions explicitly specify several independent operations, which means that they explicitly specify parallelism. Thus, it is not necessary to have decoding and dispatching hardware that tries to reconstruct parallelism from a serial instruction stream as in superscalar processors. Instead of having hardware attempt to discover parallelism, VLIW processors rely on the compiler that generates the VLIW code to explicitly specify parallelism.

This is also the main advantage of VLIW processors for many reasons. First, the compiler has the ability to look at much larger windows of instructions than the hardware. For a superscalar processor, a larger hardware window implies a larger amount of logic and therefore chip area. What is more, even before a simple limit on the amount of hardware is reached, complexity may adversely affect the speed of the logic, and thus the window size is constrained to avoid reducing the clock speed of the chip. Software windows can be arbitrarily large. As, a result, exploiting parallelism from a software window is likely to provide better results.

Moreover, the compiler has knowledge of the source code of the program. Source code typically contains important information about program behaviour that can be used to help exploit maximum parallelism at the instruction-set level. There are some scheduling techniques that help to produce efficient VLIW codes, such as the trace scheduling, superblock scheduling, software pipelining, loop unrolling etc, which can be employed to dramatically improve the quality of code output by the compiler. Taking advantage of these techniques the compiler may have access to some of the dynamic information that would be apparent to the hardware dispatch logic in a superscalar processor.

Furthermore, with sufficient registers, it is possible to mimic the functions of the superscalar implementation's reorder buffer. The purpose of the reorder buffer is to allow a superscalar processor to speculatively execute instructions and then be able to quickly discard the speculative results if necessary. With sufficient registers, a VLIW machine can place the results of speculatively executed instructions in temporary registers. The compiler knows how many instructions will be speculatively executed, so it simply uses the temporary registers along the speculated (predicted) path and ignores the values in those registers along the path that will be taken if the branch turn out to have been mispredicted.

Figure 3. 4 shows a generic VLIW implementation, without the complex reorder buffer and decoding and dispatching logic.

Finally, the pipeline structure for VLIW processors is quite more simple to implement. The compiler defines the operations executed in parallel explicitly and groups them into the instruction (VLIW). Thus, the pipeline structure for the VLIW architecture could be compared to that typically used in RISC processors, assuming the long instruction word as a RISC instruction. In, fact the operations that comprise a long instruction word are supposed to be RISC operations and are executed in parallel in the EX pipeline stage. As, shown in **Figure 3. 5**, in contrast to the pipeline structure of superscalar architecture, all the operations in each instruction word are fetched passed through the pipeline stages as if they are a single instruction.

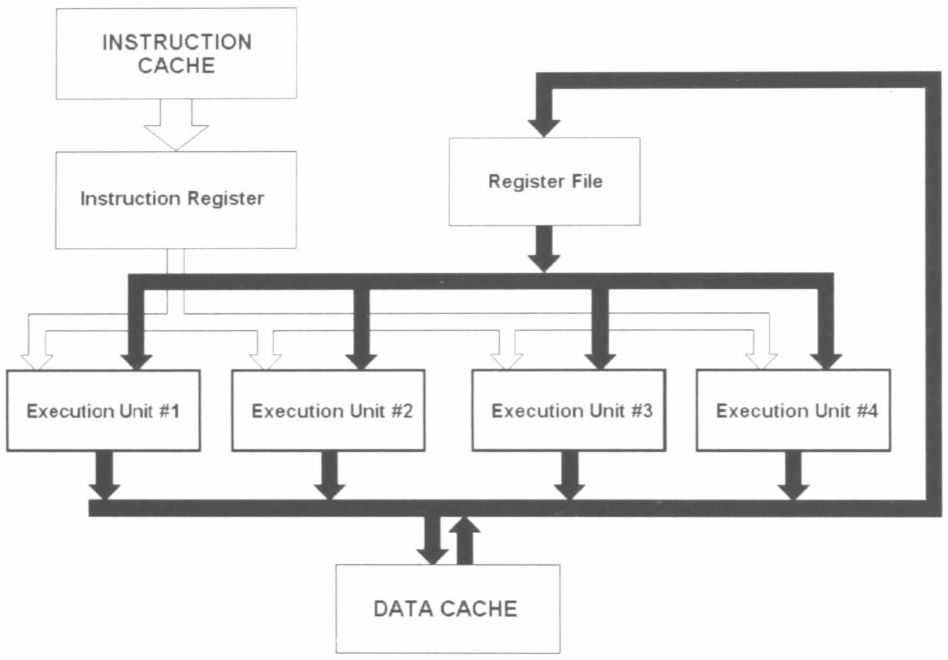


Figure 3. 4 : Block Diagram of a Generic VLIW implementation. It seems quite similar to the Superscalar one, but with most of the complex parts are not included.

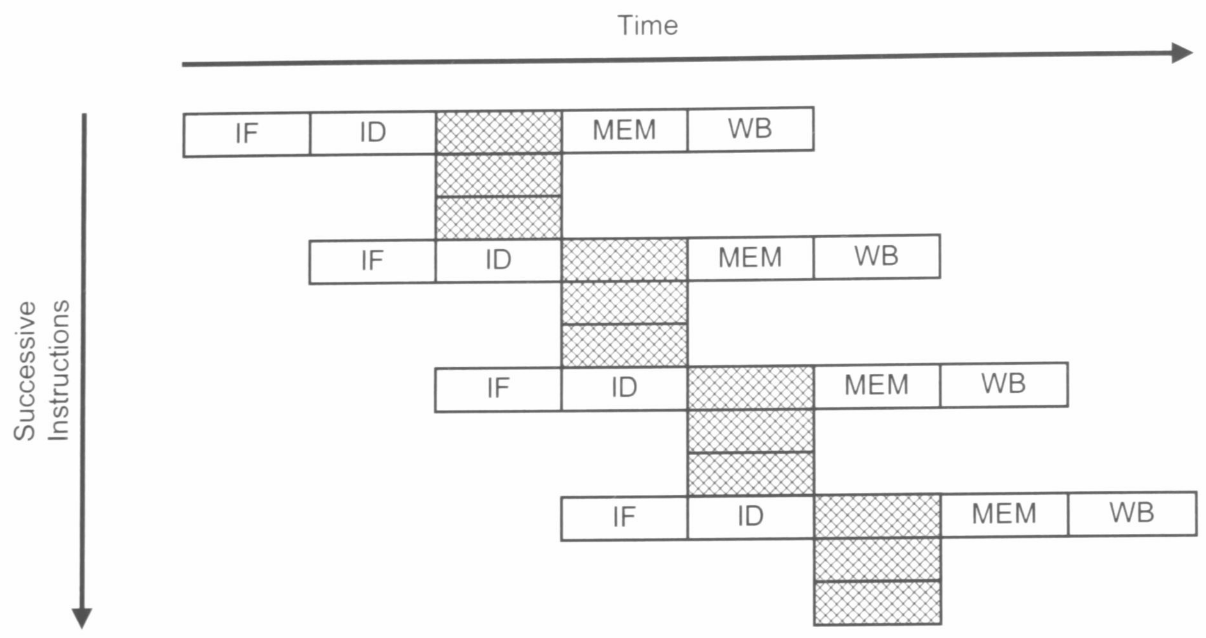


Figure 3. 5 : The pipeline structure of a typical VLIW processor with five pipeline stages. It reminds of RISCs pipeline designs. The only exception is in the EX stage where the operations in the Long Instruction are executed in parallel.

3.3 VLIW vs Static Superscalar Architecture

The basis for fully static superscalar processors comes from the field of horizontally-microcoded machines and VLIW architectures. In fact, the distinction between fully static superscalar machines and VLIWs is blurry. The basic difference comes from the terminology of the two architectures. VLIW machines refer to operations within a singly fetched instruction, while static superscalar processors refer to instructions within a single fetch block.

Both machines rely on the compiler to generate efficient schedulers to explicitly specify the instruction-level parallelism and manage the hardware resources. The scheduling techniques used for static superscalar processors are also used to perform code scheduling for VLIW processors as well. Such paradigms are the Multiflow compiler that uses *trace scheduling* and the Impact compiler that uses *superblock scheduling*. Another scheduling technique commonly used by both superscalar and VLIW machines is the *enhanced percolation scheduling*.

As far as for typical static superscalars is concerned, the difference is more obvious. Typical static superscalar processors perform main code scheduling during run-time and not at compile-time. However, all the static superscalar designs seem to be quite similar to VLIWs as far as issuing for execution is concerned. VLIW processors issue all the operations contained in the instruction word concurrently and proceed with parallel execution. Static Superscalars, issue the already scheduled instructions to the functional units for parallel execution.

3.4 Extended VLIW Organization Issues

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

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

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

1. Maximum number of control transfer operations in each instruction
2. Maximum number of load/store operations in each instruction
3. Maximum number of arithmetic/logic operations in each instruction

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

A key element for a processor that maintains a CPI factor below one (by executing more than one operation per cycle) is the ability to fetch multiple instructions. The instruction fetch bandwidth available to the processor places an upper bound on the maximum performance the processor can attain. It is also important that the processor have sufficient hardware resources to execute all of the operations that are fetched in each cycle.

A typical VLIW processor is capable of fetching and executing four operations in each cycle. As shown in **Figure 3. 6**, the processor has four execution units: one CTU (Control Transfer Unit), one LSU (Load/Store Unit), and two ALU's (Arithmetic/Logic Unit). Thus, in each cycle, the processor can execute one control transfer operation, one load/store operation, and two arithmetic/logic operations, all in parallel. The CTU interact with the Program Counter (PC), and the LSU is connected to the data memory subsystem.

Finally all the four execution units interact with a global Register File (RF). The instruction format for this configuration is shown in **Figure 3. 7**. For each hardware execution unit, there is an operation field in the instruction. This will allow the processor to fetch as many operations as the hardware resources of the processor can handle in each cycle.

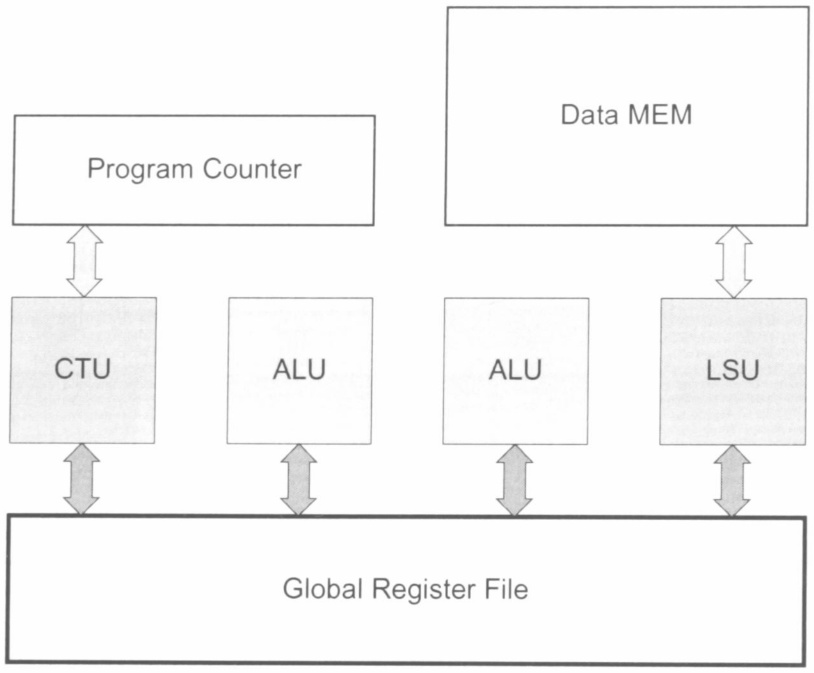


Figure 3. 6 : The General View of a Typical VLIW Processor Organization

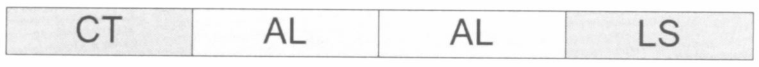


Figure 3. 7 : The Long Instruction Word Format for the above Organization (figure 3.6).

One disadvantage of the organization shown in **Figure 3. 6** and the associated instruction format is that instructions that do not have control transfer or load/store operations will result in empty slots in the long instruction word. This effectively results in wasted instruction fetch bandwidth.

To achieve a higher level of performance, we could add more execution units and fetch more operations in each long instruction word. For example, the organization shown in **Figure 3. 8** can fetch and execute eight operations in each cycle. The performance improvement is due to the increase in the fetch bandwidth and the existence of additional execution units. However, this organization still suffers from the problem that the first one did. As shown in the instruction format in **Figure 3. 9**, to keep the machine completely busy, each instruction must have two CT operations, two LS operations, and four AL operations. Instructions that do not have CT or LS operations result in wasted instruction fetch bandwidth. Another problem with this new configuration is that the register file must have twice as many ports as before. This will slow down the register file and will lengthen the processor cycle time.

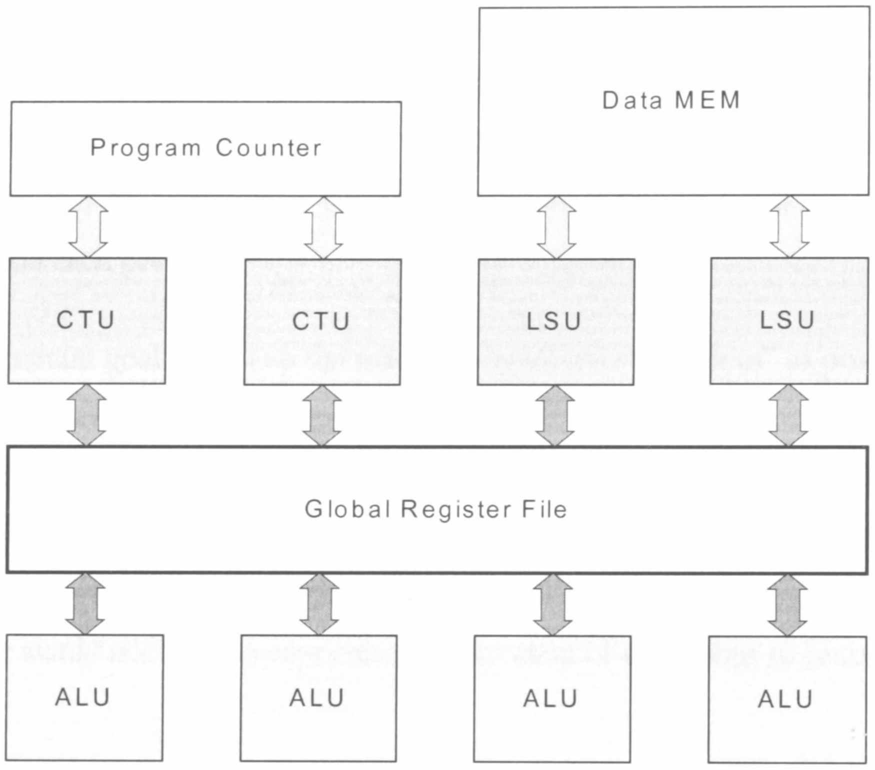


Figure 3. 8 : A VLIW Processor Organization with Increased Hardware Resources.

CT	CT	LS	LS	AL	AL	AL	AL

Figure 3. 9 : The Long Instruction Word Format for the above Organization (figure 3.8).

An alternative approach to increase performance is to combine different types of execution units into groups. Each group corresponds to an operation field in a long instruction word and is commonly known as Functional Unit. The advantage of this strategy is that each operation field in an instruction is not restricted to a specific type. A long instruction word can have various combinations of CT, LS, and AL operations. This will result in better utilization of the instruction fetch bandwidth because to keep the machine busy we are not required to have a specific combination of operations in each instruction.

This strategy can improve the performance of the processor because it allows the processor to execute instructions that have, for example, four AL operations whereas the configuration shown in **Figure 3. 6** will require that the instruction be broken into two instructions during the Resource Constrained Scheduling phase of compilation. This will increase the path length of the program and result in more execution cycles. An important aspect of this organizational strategy is that performance gain is achieved without increasing the required instruction fetch bandwidth or the number of register file ports. Since each register file port is now connected to more than one execution unit, there is a greater load on each port.

Another important goal is to keep the machine organization as "clean" as possible in terms of its resource limitations. This approach will simplify code generation. A machine with many restrictions and idiosyncrasies is difficult to generate good code for. A simple way to achieve this is to make each functional unit able to execute all types of operations supported in the instruction set. This would considerably simplify code generation because the compiler would not have to worry about assignment of operations to functional units.

However, combining all types of execution units in each functional unit has also some drawbacks. Being able to send operations of all types to all functional units increases hardware complexity and resources.

Chapter 4

4 VLIW - A DETAILED DESCRIPTION

In this chapter I perform a thorough description of many issues concerning VLIW architectures. I firstly discuss some software and then some hardware techniques used to exploit the maximum available parallelism for a VLIW processor. Finally, I present the main drawback of VLIW processors, the object compatibility problem, and discuss some approaches in order to overcome the problem.

4.1 Compilation Techniques for VLIW

Generating code for a VLIW processor is a difficult issue as the compiler is faced with the task of extracting parallelism from a sequential algorithm and scheduling independent operations concurrently. The degree of ILP that can be uncovered by the compiler is based on some compilation techniques that I summarize in this section.

4.1.1 Scheduling Algorithms

Instruction scheduling algorithms are critical to the performance of a VLIW processor. The algorithms I discuss here are also called global acyclic schedulers and each of these is able to perform code motion across branches, including speculative code motion. In this section, I describe some important scheduling algorithms, starting with the *trace scheduling* algorithm, which started off the VLIW style of architectures, approaching it more extensively. Moreover, I briefly discuss the role of *speculation* and *predicated execution*, and finally the relationship of instruction scheduling to the task of *register allocation*.

4.1.1.1 Trace Scheduling (TS)

Compilers for the first ILP processors used a 3 phase method to generate code. The passes were:

- Generate a sequential program. Analyze each basic block in the sequential program for independent operations.
- Schedule independent operations within the same block in parallel if sufficient hardware resources are available.
- Move operations between blocks when possible.

This three-phase approach fails to exploit much of the ILP available in the program for two reasons. Often times, operations in a basic block are dependent on each other. Therefore sufficient ILP may not be available within a basic block. Moreover arbitrary choices made while scheduling basic blocks make it difficult to move operations between blocks.

Trace scheduling is a profile driven method developed by Joseph Fisher at Yale in the Bulldog compiler as a part of the ELI-512 project [5] to circumvent this problem. In trace scheduling, a set of commonly executed sequence of blocks is gathered together into a trace and the whole trace is scheduled together.

To sketch the trace algorithm briefly, we start by generating a possibly unoptimized version of the program, run it on sample input and collect statistics on the probability of each conditional branch. Then, given a basic block level data dependence graph (also known as DAG for Directed Acylic Graph), we can find loop free linear sequence of basic blocks which have a high probability of execution. Such a sequence is called a trace.

Considering the trace as if it were a basic block, we build a DAG for it considering branches like all other operations. In order to prevent the scheduler from making absolutely illegal code motions between blocks, we add new, special edges to the graph. The new edges are drawn between operations that conditionally jump to where the variable is live and could be overwritten. Also edges are added to preserve the relative order of conditional branches. The edges are added to the graph and look just like all the other edges. The scheduler may now schedule the resulting DAG as if it were a basic block doing register allocation and function unit selection as each operation is scheduled.

Scheduling the trace can be done using any local scheduling algorithm. After the scheduling though, program semantics may have changed. Two cases have to be looked at:

- *Rejoins:* A first problem is that branches into the trace (rejoins) at some point cannot always branch to the same place after the trace has been scheduled. Operations that are moved below the original branch target must not be executed when that branch is taken. This means that the highest valid target in the trace is the point below the last instruction originating from before the target. But then again, some operations that were originally found after the old join point may now appear before the new join point. Since these instructions have to be executed when the branch into the trace is taken, they are duplicated into a new basic block, which is inserted before the branch. An example is given in **Figure 4. 1**.

- *Splits:* Another problem is when operations that were used to precede a conditional branch are moved below that branch. Since these operations were originally always executed, they have to be duplicated in a new basic block preceding the off-trace target of the conditional jump. Such an example is given in **Figure 4. 2**.

This recovery process of the algorithm is also called *bookkeeping*.

Finally, the new trace is linked into the old DAG. An example of the trace scheduling procedure is described in steps in **Figure 4. 3**. After scheduling the very first trace, new operations would have been added to the original DAG. We, then, pick a different frequent trace and schedule it. This is repeated untill the DAG has been covered using disjoint traces and no unscheduled operations remain.

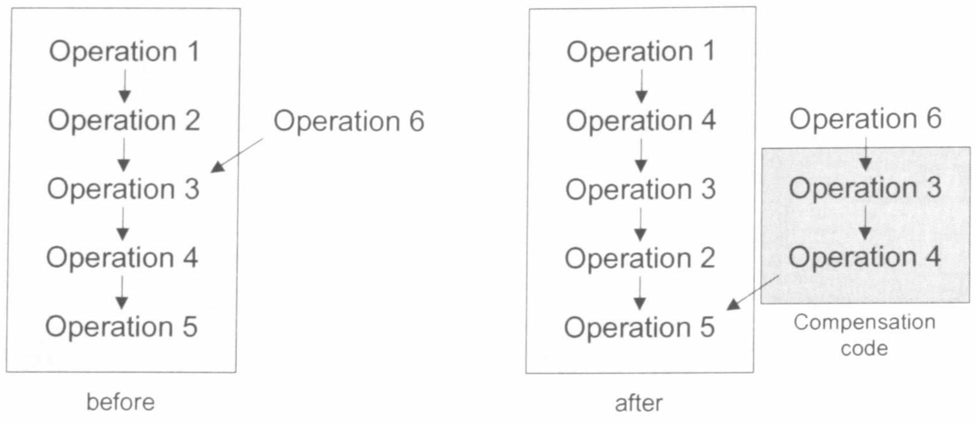


Figure 4. 1 : Compensation code at rejoin points.

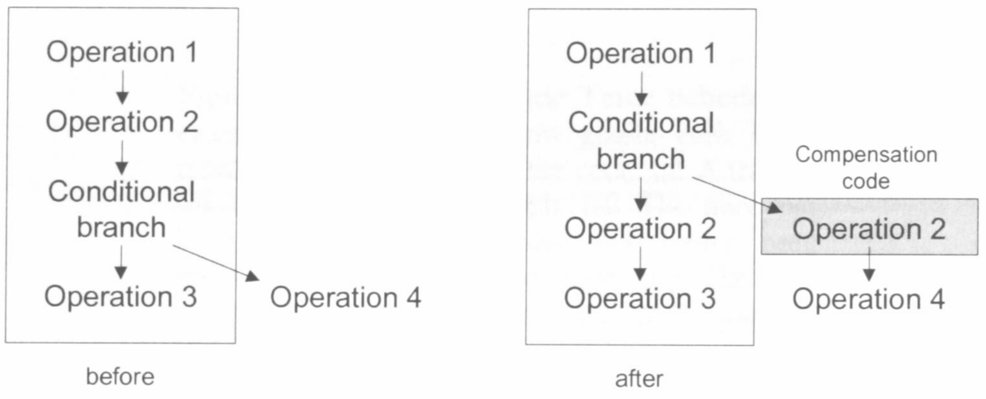


Figure 4. 2 : Compensation code at split points.

Trace scheduling provides a natural solution for loops. Hand coders use software pipelining to increase parallelism, rewriting a loop so as to do pieces of several consecutive iterations simultaneously. Trace scheduling can be trivially extended to do software pipelining on any loop. We simply unroll the loop for many iterations. The unrolled loop is a stream, and the stream gets compacted as above.

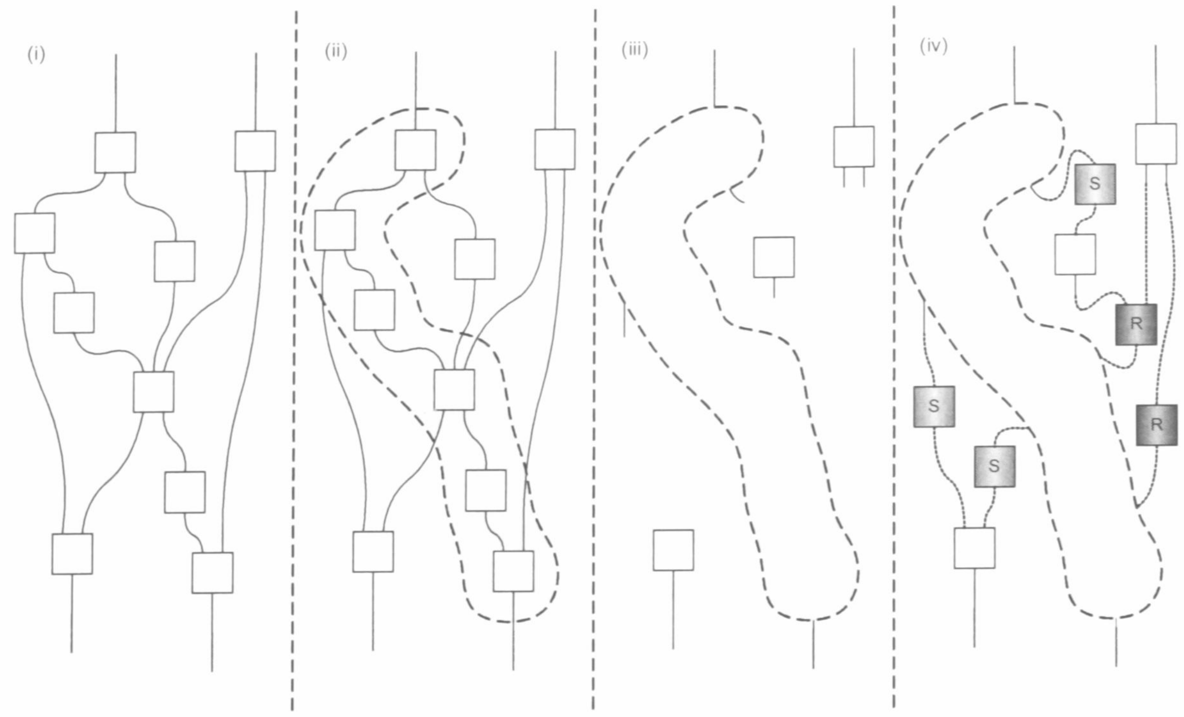


Figure 4. 3 : Loop-free code Trace Scheduling example. (i) The code flow graph; each block represents a basic block of the code. (ii) A trace is selected from the flow graph. (iii) The trace has been isolated and scheduled but hasn't been relinked to the rest of the code. (iv) The new compensation code appears at the code splits [S] and rejoins [R].

4.1.1.2 Trace Scheduling - 2

Trace scheduling - 2 [4] goes beyond trace scheduling in that it allows nonlinear code motion, i.e. it allows operations from both sides of a conditional branch to be moved above the branch. Trace scheduling usually misses code motions that are speculative or moves operations from one trace to another, because a trace contains only one direction of if-then-else structures. Trace Scheduling-2, on the other hand, enables the motion of code before a conditional jump from both directions at the same time and is more considerate of code coming from less likely paths.

Trace scheduling - 2 uses an expected value function called speculative yield to consider the cost of speculative execution and decide whether or not to move operations from one block to another. Unlike trace scheduling, which operates on a linear sequence of blocks, the newer algorithm works by picking clusters of operations where each cluster is a maximal set of operations that are connected without back edges in the flow graph of the program.

4.1.1.3 Region Scheduling

Region Scheduling is a program transformation system that operates upon the program dependence graph representation of the program. Instruction scheduling is carried out by first reordering code within a control dependence region and then by performing code motions across control dependent regions. In regions that contain too much parallelism for the underlying architecture to be exploited, parts of their instructions are moved to other regions where more parallelism can be used. Code reordering within a control dependence region is given preference because it results in less code growth and unlike speculative code motion it does not harm any program paths. Such code reordering, unlike trace scheduling, also enables instructions to be moved across loop boundaries. Much of the redundant code generated by a trace scheduler during bookkeeping is also avoided. The instruction schedule is progressively improved through code motion transformations until no more improvements in the schedule can be identified.

4.1.1.4 Superblock Scheduling

Super block scheduling is a region scheduling algorithm developed in conjunction with the Impact compiler at the University of Illinois [6]. Like trace scheduling, super block scheduling is based on the premise that to extract ILP from a sequential program, the compiler should perform code motion across multiple basic blocks. Unlike trace scheduling, super block scheduling is driven by static branch analysis, not profile data. A super block is a set of basic blocks in which control may enter only at the top, but may exit at more than one point.

Super blocks are formed by first identifying traces and then eliminating side entries into a trace by a process called tail duplication. Tail duplication works by creating a separate off-trace copy of the basic blocks in between a side entrance and the trace exit and redirecting the edge corresponding to the side entry to the copy. An example of superblock formation is shown in **Figure 4. 4**. Traces are identified using static branch analysis based on loop detection, heuristic hazard avoidance and heuristics for path selection. Loop detection identifies loops and marks loop back edges as taken and loop exits as not taken.

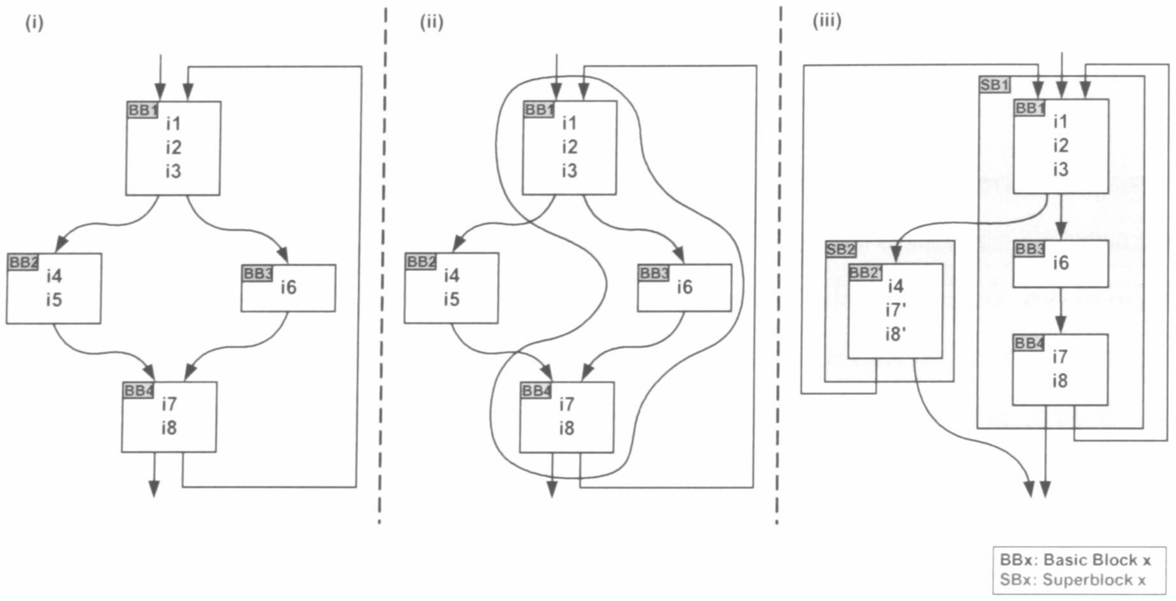


Figure 4. 4 : Superblock Scheduling Example. (i) The code control flow graph. (ii) A trace is selected from the flow graph. (iii) Superblock formation and branch expansion. Superblocks are shown in green colour.

Hazard avoidance uses a set of heuristics to detect situations like ambiguous stores and procedure calls that could a cause a compiler to use conservative optimization strategies and then predicts the branches so as to avoid having to optimize hazards. Path selection heuristics use the opcode of a branch, its operands and the contents of its successor blocks to predict its direction if no other method already predicted the direction of the branch. These are based on common programming patterns like the fact that pointers are unlikely to be NULL, floating point comparisons are unlikely to be equal etc. Once branch information is available, traces are grown and super blocks created by tail duplication followed by scheduling of the super block. Studies have shown that static analysis based super block scheduling can achieve results that are comparable to profile based methods.

4.1.1.5 Percolation Scheduling (PS)

Percolation scheduling is also a program transformation system that has certain advantages over a trace scheduler. A trace scheduler divides program transformation into two stages. The first stage reorders code along the trace while the second bookkeeping stage modifies the rest of the program to preserve program semantics. This separation of transformation process into two steps can lead to the generation of redundant code during the bookkeeping stage. By applying semantics preserving transformations in one step, percolation scheduling avoids this problem.

Percolation scheduling works on the parallel program graph, in which nodes contain one or more operations that can be executed in parallel and edges determine the execution paths of the program. A well-defined set of transformations, guided by heuristics, rearranges the code globally to exploit more parallelism. Since execution starts at the top node and goes node by node, those nodes traversed by a program must be decreased.

Nodes are generally defined as a set of operations and conditional branches that are called components. If there are no conditional branches, a continuation (successor) node is contained in the node. Several conditional jumps may be present in one node, if they form a tree. This means that paths following a jump can lead to another jump in the node (internal jump) or to a successor (outgoing jump).

The base program transformations used to achieve code percolation are the following:

- *Deletion:* Nodes that become empty due to previous transformations can be removed from the parallel program graph.

- *Move-up:* Operation components can be moved from node *n* to node *m* if no dependencies exist between node *m* and the components being moved. Paths passing through *n*, but not through *m*, must preserve their semantics. An operation move-up example is shown in **Figure 4. 5.**

- *Move conditional jumps:* Conditional jump components can be moved up if dependencies allow it. If necessary, some nodes and components have to be duplicated to preserve program semantics. A branch move-up example is shown in **Figure 4. 6.**

- *Unification:* Under certain conditions, different instances of identical operations can be replaced by one instance in the common predecessor of these nodes.

Backward code motions are not allowed since they could endanger the guaranteed termination of the PS algorithm.

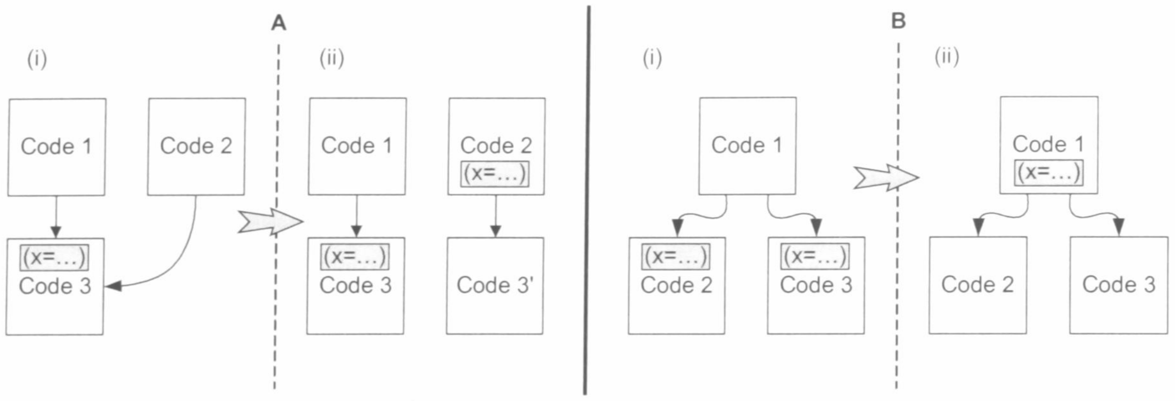


Figure 4. 5 : Move-up instruction transformation example. A: Move-up of a single assignment. B: Move-up of multiple assignment.

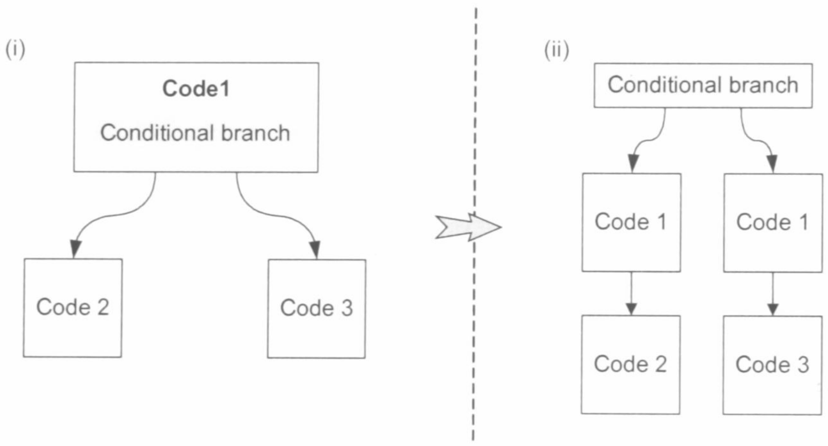


Figure 4. 6 : A conditional branch move-up transformation example.

4.1.1.6 Critical Path Reduction

In this technique the program region being scheduled is a multiple entry multiple exit acyclic region. The region exits are classified into two categories, frequently taken and infrequently taken. Even if the paths leading to frequently taken exits do not include any delay slots, an attempt is made to reduce the schedule length by pushing statements off these paths and to the infrequently taken exits. This transformation is sometimes possible because a path to a frequently taken exit may contain statements that are dead along these paths even though they may be live along other paths. Essentially, this technique integrates the partial dead code elimination optimization into the instruction scheduler.

4.1.1.7 Interaction with register allocation

The interaction of instruction scheduling and register allocation is an important issue for VLIW architectures that exploit significant degrees of ILP. Register allocation and instruction scheduling have somewhat conflicting goals. In order to keep the functional units busy, an instruction scheduler exploits ILP and thus requires that a large number of operand values be available in registers. On the other hand, a register allocator attempts to keep the register demand low by holding fewer values in registers so as to minimize the need for generating spill code.

If register allocation is performed first, it limits the amount of ILP available by introducing additional dependences between the instructions based on the temporal sharing of registers. If instruction scheduling is performed first, it can create a schedule demanding more registers than are available, causing more work for the register allocator. In addition, the spill code that is generated must be incorporated in the schedule by another scheduling pass, degrading the performance of the schedule. Thus, an effective solution should integrate register allocation and instruction scheduling. The significance of integration is greatly increased in programs where the register demands are high since the likelihood of spill code generation is high for such programs. Also, when compiling programs for wide issue machines, the need for integrating register allocation and instruction scheduling is the greatest.

4.1.2 Software Pipelining

The instruction window from which ILP is extracted by acyclic schedulers consists of paths that cannot extend across loop iterations. Thus, acyclic schedulers cannot exploit ILP present across code blocks from different loop iterations. One approach to uncover such parallelism is to unroll the loops to transform parallelism across loop iterations into parallelism that exists within a single loop iteration of the transformed loop. An acyclic scheduler can then schedule the transformed loop. A loop-unrolling example is shown in **Figure 4. 7**. However, this approach can result in substantial code growth.

```
for (i=6; i<=100; i=i+1)
{
        Y[i]=Y[i-5]+Y[i]
}
```

- Each iteration *i* depends on the value of the iteration *i-5*. Iteration I, I+1, I+2, I+3, I+4 are independent! (I+5 is dependent on I) so we can unroll the loop:

```
for (i=6; i<=96; i=i+5)
{
        Y[i]=Y[i-5]+Y[i]
        Y[i+1]=Y[i-4]+Y[i+1]
        Y[i+2]=Y[i-3]+Y[i+2]
        Y[i+3]=Y[i-2]+Y[i+3]
        Y[i+4]=Y[i-1]+Y[i+4]
}
```

⟸ Larger Basic Block
With extended parallelism

Figure 4. 7 : A Loop Unrolling example.

For loops where some iteration depends on some previous iteration, which appears to be a common case in real programs, executing the iterations in pipeline fashion is an attractive way to achieve speedup [9]. In the context of microprogrammable architectures, this technique is called software pipeline. Software pipelining exploits ILP across loop iterations without causing considerable code growth.

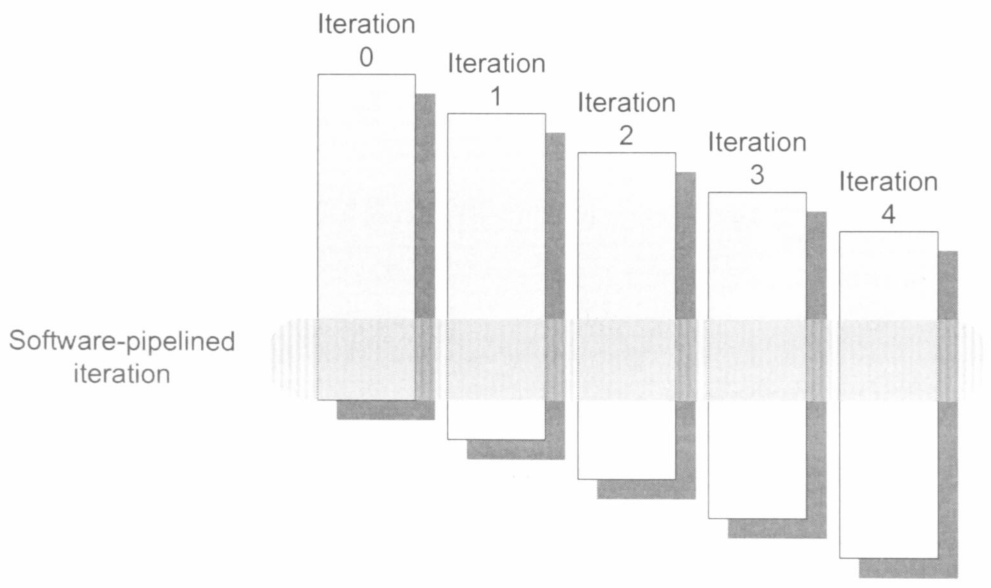


Figure 4. 8 : A software-pipelined loop chooses instructions from different loop iterations, thus separating the dependent instructions within one iteration of the original loop.

Software pipelining is a technique that overlaps execution of operations from different loop iterations and thus exploits ILP across loop iteration boundaries. This execution overlapping of operations is described in **Figure 4. 8**. After the reorganization of the loops some additional code is required. That is a start-up code in order to execute code left out from the first original iteration, and a finish code in order to execute code left out from the last original loop iteration. **Figure 4. 9** shows the way loops are software pipelined, and **Figure 4. 10** shows the transformed code and the effectiveness of software pipeline in VLIW processors in the way this code is executed. The objective of software pipelining is to generate a schedule, which minimizes the interval at which iterations are initiated, that is, the initiation interval. A software pipelining algorithm must take into account the instruction latencies and resource availability while scheduling the operations from the loop. In addition, any increase in register demands must be met to avoid generation of spill code inside loops. Generally the techniques for software pipelining assume that the loop body of the pipelined loop contains no branches.

Finally, loop unrolling may be applied before software pipelining in order to provide better performance.

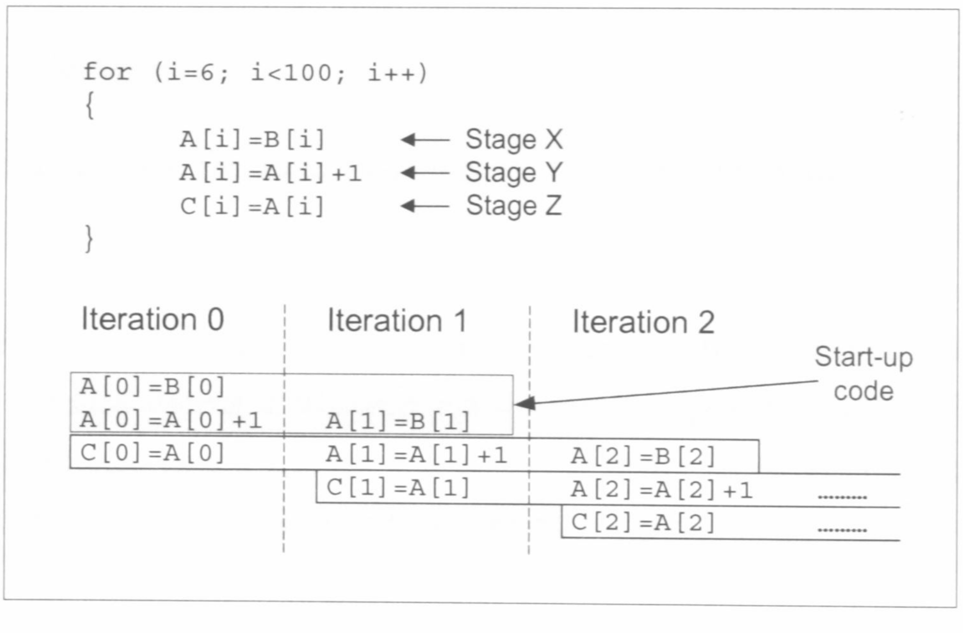
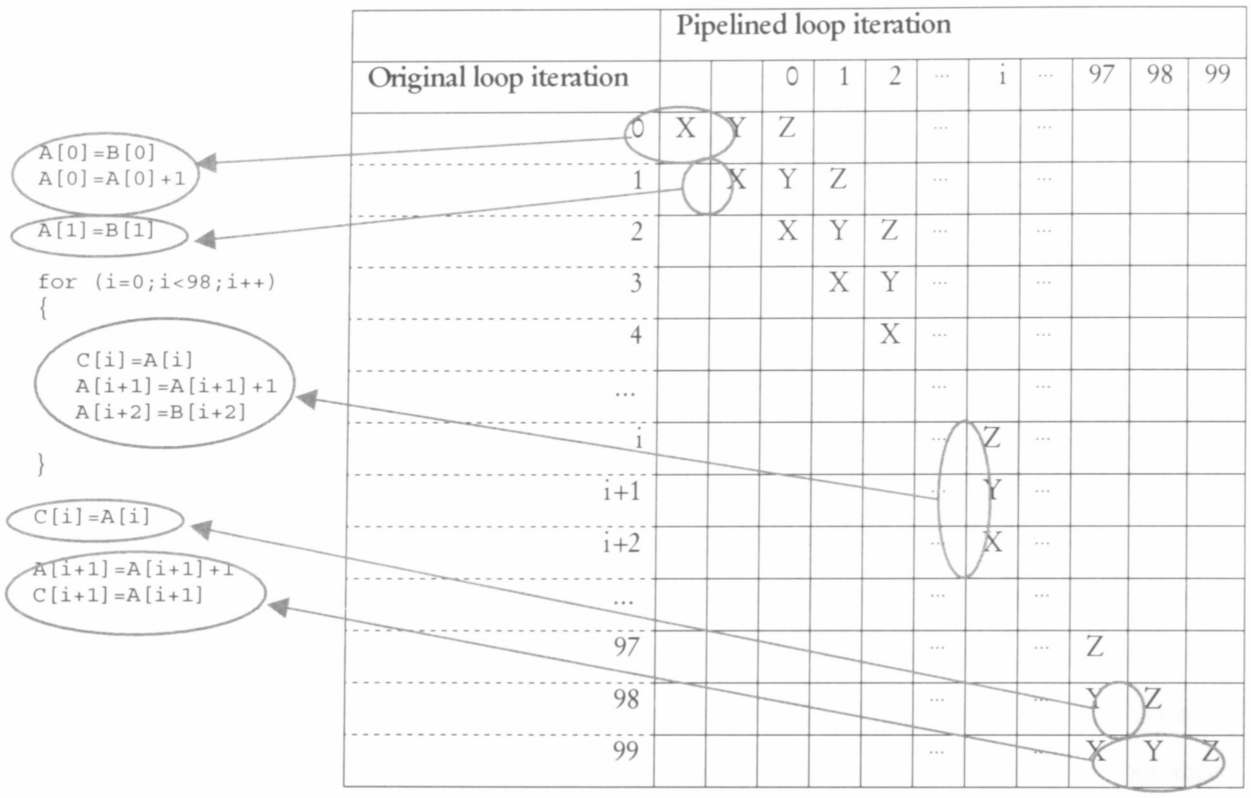


Figure 4. 9 : Software Pipelining Example.



```
A[0]=B[0]
A[0]=A[0]+1

A[1]=B[1]

for (i=0;i<98;i++)
{
   C[i]=A[i]
   A[i+1]=A[i+1]+1
   A[i+2]=B[i+2]
}

C[i]=A[i]

A[i+1]=A[i+1]+1
C[i+1]=A[i+1]
```

Figure 4. 10 : The previous code example after software pipeline. Instructions in the loop are independent. These code groups can usually fit in a VLIW (instruction).

Modulo Scheduling

Modulo Scheduling is the most commonly used algorithm for implementing software scheduling. In this approach a lower bound on the initiation interval is established based upon the data dependences in the loop and the resource demands of the loop. The modulo scheduler then searches for a schedule with the minimum initiation interval. If the search fails, the initiation interval is increased and the search is performed again. The above process is repeated until a schedule can be found. In the above manner the modulo scheduler finds a schedule with the minimum initiation interval.

4.1.3 Value Prediction for VLIW

The performance of VLIW architectures is dependent on the capability of the compiler to achieve effective scheduling to extract instruction level parallelism. Instructions are reordered to reduce the length of the code schedule and minimize the cycle count for execution. Code reordering, which is necessary to achieve this goal, is limited by both control and data dependencies. These dependencies can prevent the compiler to fill the instruction word with operations. One approach used to reduce the instruction length is control speculation by moving code above branches. Although control speculation is effective in removing control dependencies, true data dependencies are frequent and long dependency chains become a bottleneck to the scheduler. VLIW machines, where conservatively computed data dependencies sequentialise the order of the operations for execution, suffer from this problem.

Value prediction is found in many hardware based value predictor designs. However, VLIW machines are scheduled statically and hence value prediction can be applied to operations selected at compile-time. Executing operations with predicated values, value speculation, results in significant speedups for VLIW machines [11]. A predicated value is eventually verified by executing the original operation that was predicated, and comparing the correct value with the predicated one. In case the prediction is found to be correct, code is executed as before. However, if the value was mispredicted, all operations that were value-speculated using the incorrect value are re-executed with the correct value.

4.2 Some Design Issues for VLIW Architectures

Very Long Instruction Word Architectures can exploit the instruction level parallelism typically found in sequential-natured program code. Global compaction algorithms in parallelizing compilers for VLIW architectures generally produce wide instructions packed with multiple tests along with data operations. Depending on the results of parallel tests a *multiway branch* mechanism selects the next instruction from many targets.

Another proposal for alternative execution models for VLIW architectures, as an architectural improvement, is the *speculative execution* of operations where an operation may be issued before it is known that its execution is required. Finally, *predicated execution* provides hardware support for exploiting more instruction-level parallelism by allowing conditional execution of operations.

4.2.1 Multiway Branches

In a “traditional” computer, branch instructions can specify a “condition” and a “target address”, and the program counter is set to the target address if the condition is true. VLIW instruction sets, however, often contain a means for specifying a number of conditions and targets, such that the program counter value is selected from among the targets depending on a combination of conditions. In general, it is faster to execute a single multiway branch instruction than an equivalent series of two-way conditional branches, especially in non-numerical integer code, where conditional branches are more frequent.

VLIW processors execute highly optimized code. Compilation techniques for optimization include global scheduling and software pipelining. After the compiler schedules data instructions, conditional branches tend to cluster together. Since sequential execution of these branches becomes a bottleneck in increasing performance, multiple branches are also scheduled in a VLIW for parallel execution, possibly with other independent data instructions.

```
if b1 then
      if b2 then I0
      else I1
else
      if b3 then I2
      else I3
end
```

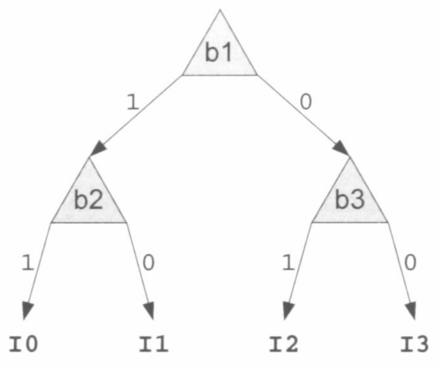


Figure 4. 11 : An example program segment and its condition tree.

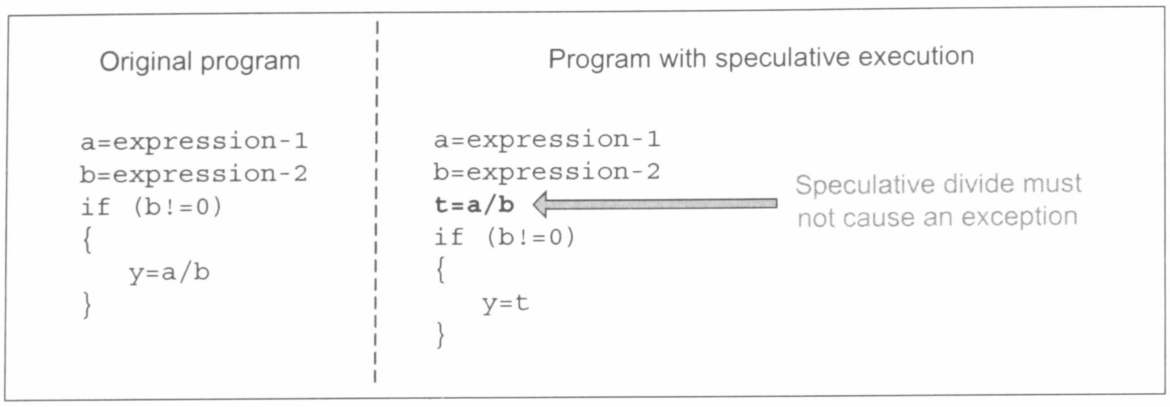
Multiway branch mechanisms fall into two categories: those that implement a fixed branching structure and those that implement a variable branching structure. All multiway branch mechanisms are designed to solve the same problem. The CPU provides a set of condition bits. The control structure programmed in a VLIW specifies a subset of these condition bits, a set of target identifiers and a mapping from condition bit values to target identifiers. This mapping takes the form of a condition tree (**Figure 4. 11**). A condition tree (also known as decision tree) is a binary tree whose interior nodes select condition bit values and whole leaf nodes are target identifiers. Selecting a branch-target involves traversing the condition tree based on the condition bit values until a leaf node (target identifier) is reached. A primary design objective for any multiway branch hardware is the ability to perform this traversal in a single clock.

A multiway branch mechanism provides a *target selection unit*, which is a hardware device that maps condition bit values to target identifiers. Mapping the condition bits onto the condition bit values is done outside the target selection unit. Given a series of condition bit values, the target selection unit provides a combinational mapping into the target identifiers. Although target identifiers can be used in a number of different ways, the ways are used does not affect the nature of the target selection mechanism provided by the hardware. For example, the target identifier might be used to select one of the address registers or immediate address operands. Alternatively, the target identifier might be concatenated with another quantity as a prefix or a suffix to form an address, so that all target VLIW instructions can be prefetched at the beginning of the cycle, while the target identifier of the tree path that is being taken can act as a late-select to obtain a faster instruction cache access path.

4.2.2 Speculative Execution

Speculative execution refers to the issuing of an operation before it is known that its execution is required. Speculative execution occurs when the scheduler places an operation above a preceding conditional branch, and thus it is also called control speculation. Operations that are data ready but not control ready (that is, operations for which the input operands have been computed but to which control flow has not yet reached) are candidates for speculative execution. It is desirable for the compiler to make use of speculative operations to reduce time required for a computation. However, operations that cause exceptions or side effects cannot be executed speculatively. An example of control speculation is shown in **Figure 4. 12.**

In order to perform speculation execution, the processor must permit it. This can be achieved by setting a mode bit in the processor that turns off exception processing for all operations marked as speculative. However, this significantly changes a program's error behavior because operations that would have caused errors will cause no errors after optimization. Another approach is to provide non-trapping versions of those operations that can cause exceptions, which then can be used in speculative execution. With this approach the program's error reporting behavior is improved because errors caused by non-speculative operations will halt execution when they occur. With additional architectural support, a program can precisely report errors even in presence of speculative execution.



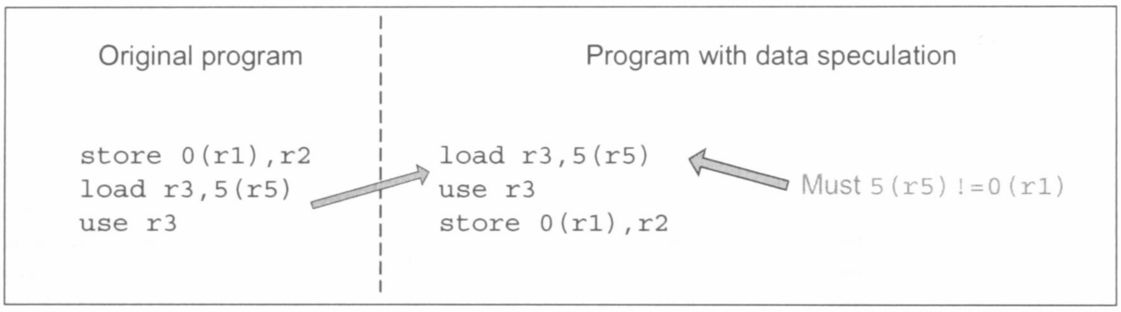
Original program
Program with speculative execution
Speculative divide must not cause an exception

```
a=expression-1
b=expression-2
if (b!=0)
{
    y=a/b
}
```

```
a=expression-1
b=expression-2
t=a/b
if (b!=0)
{
    y=t
}
```

Figure 4. 12 : Example of speculative execution.

Another kind of speculative execution is that of data speculation. As shown in the example in **Figure 4. 13**, it might be useful to move the load operation before the store one in order to fill slots between the definition of register R1 and the store. However, this kind of speculative execution may violate correct execution semantic. For example, if the store and the load addresses are the same (0(r1) = 5(r5)), it results in read-after-write hazard. Thus, the load operation can only be moved before if it is sure that there is no address aliasing. Finally, some architectures, like IA-64, perform a later check for a RAW hazard and make explicit correction to what has been done.



```
store 0(r1),r2
load r3,5(r5)
use r3
```

```
load r3,5(r5)
use r3
store 0(r1),r2
```

Figure 4. 13 : Example of data speculation.

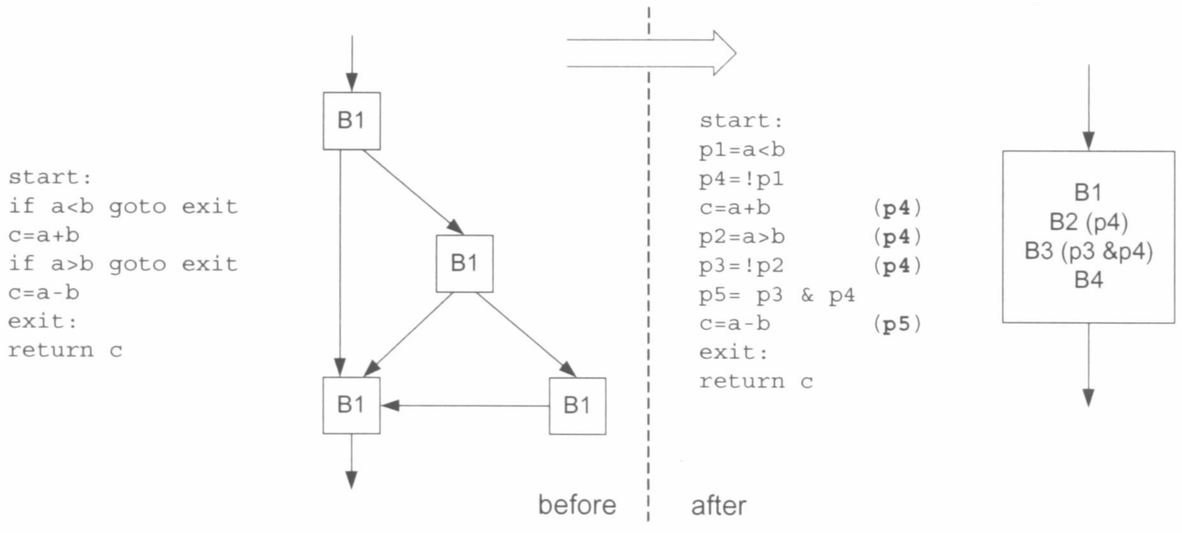
4.2.3 Predicated Execution

Predicated execution is an architectural model in which each operation is guarded by a Boolean operand whose value determines whether the operation is executed or nulled. This makes it possible to execute an operation conditionally depending upon the value of its predicated original acyclic code. The control flow through an acyclic code fragment that is implemented through branches can be entirely eliminated by predicating the instructions.

The resulting code may be viewed to simultaneously execute all the paths through the original acyclic code.

Predication may be used to eliminate unpredictable branches and consequently reduce the harmful effects of such branches on the execution. Instead of jumping around an operand that should not be executed, the hardware can just ignore the effects of the operation. The Boolean result of a condition testing is recorded in a (one-bit) ***predicate register*** *p*. An operation is executed if its predicate register *p* is true. If *p* is false, either the operation is ignored (treated as no-op) or is executed but the result register is not changed. The latter option allows the operation to be executed in the same instruction as the predicate is computed. Since the results of predicate evaluation are known before the target register is written, such overlap is possible. Such hardware support may eliminate a physical jump.

Often, an entire acyclic control flow region can be converted into a single, branch free block of predicated code. This process of converting code into predicate code is termed ***if-conversion***. Conditional branches are removed and control dependencies become data dependencies, as conditionally executed operations are data dependent on the operation that generates the predicate on which they depend. **Figure 4. 14** describes an if-conversion procedure.



```
start:
if a<b goto exit
c=a+b
if a>b goto exit
c=a-b
exit:
return c
```

```
start:
p1=a<b
p4=!p1
c=a+b          (p4)
p2=a>b         (p4)
p3=!p2         (p4)
p5= p3 & p4
c=a-b          (p5)
exit:
return c
```

Figure 4. 14 : An if-conversion example.

There are some more situations in which the use of predication can be quite useful. For example if a speculatively issued load frequently causes a cache miss along some program path then by predicating the load it may be possible to avoid the speculative issue along that path and hence minimize the cache misses.

While in the above situations the benefits of predication are clear, its aggressive application is a far more challenging task. If the lengths of the paths through an acyclic code vary greatly, predication would extend the execution times of shorter paths. Moreover the demand for register resources may be increased to a point that predication no longer yields superior instruction schedules. Finally the data flow analysis techniques used to analyze programs, for such basic tasks as uncovering data dependences and performing optimizations, is based upon explicit control flow. A further analysis must be done in order to accurately handle the implicit control flow expressed in predicated code.

4.3 Object Compatibility Issues

VLIW processors are viewed as an attractive way of achieving instruction level parallelism because of their ability to issue multiple operations per cycle with relatively simple control logic. However, the lack of object compatibility in VLIW architecture is a severe limit to their adoption as a general-purpose computing paradigm. Two classes of solutions have been pursued to solve the VLIW object-code compatibility problem: software-based and hardware-based techniques. The hardware approaches include *split-issue* proposed by B. R. Rau, and the *fill-unit* proposed by S. Melvin, M. Shebanow and Y. Patt, and finally the software approach *dynamic rescheduling* proposed by M. Conte and W. Sathaye. These three techniques are briefly discussed in this section.

4.3.1 The hardware Approach

A hardware-based solution for VLIW processors was proposed bye B. R. Rau in [12], wherein the concept of delayed split-issue together with dynamic scheduling hardware is introduced. These characteristics allow using the interlocking and score-boarding techniques known for dynamic scheduling in superscalar processors. Many studies on this approach have shown that dynamic scheduling is as viable for VLIW processors as with more conventional ones. As a result, the object-code generated for one implementation of a VLIW processor family can be executed on an implementation with fewer functional units and/or different latencies for the operations.

Another related class of hardware-based solutions consists of using sequential code as the compatible representation and performing dynamic parallelizing. For example, the fill-unit approach extracts VLIWs from the sequential execution of a program, so that successive executions of the same instructions are performed using the parallel rather than the original sequential version. However, this type of approaches is limited due to the small window of instructions analyzed, and the complexity associated to run-time parallelizing.

The Split-Issue Approach

Split-Issue was presented by Rau [13] as a technique for dynamic scheduling in VLIW processors. It provides hardware capable of splitting each Op into an Op-pair: (*read_and_execute*; *destination_write-back*). Read_and_execute uses an anonymous (i.e. a non-architected) register as its destination, whereas destination_write-back copies the destination of read_and_execute to the destination specified in the original Op. Read_and_execute operation is issued in the next available cycle, provided there are no dependence or resource constraints. The destination_write-back operation is scheduled to be issued in the latest cycle after (*issue_cycle (read_and_execute) + original_operation_latency - 1*). To ensure that the destination_write-back operation is not issued before the read_and_execute completes, support in the form of hardware flags is provided. The splitting of operations and issuing them in the correct time order preserves the program semantics, and correct program execution is guaranteed.

The Fill-Unit Approach

The Fill-Unit approach combines the advantage of code compatibility of as in superscalars and the absence of complex dependency checking logic form the decoder as in VLIW. The objective is to provide hardware that monitors the instruction stream and groups multiple instructions into VLIW-type instructions, which are then stored in a structure, called a shadow cache, within processor itself. When a shadow cache line contains the instructions requested by the fetch unit, the scalar instruction stream is preempted and all operations in the shadow cache line are simultaneusly issued and executed. The mechanism that compacts instructions is called a *fill-unit*.

Fill-unit was first proposed [15] for dynamically compacting microoperations generated from sequentially-fetched instructions into large executable units, by Melvin, Shebanow and Patt in 1988. Some years later, in 1994, M. Franklin and M. Smotherman [14] extended this approach to directly handle data dependencies, delayed branches and speculative execution. This method would allow a sequential instruction stream to be fed to the processor, with execution of parts of the code accelerated by wide issue whenever possible. At the same time it would preserve code compatibility.

4.3.2 The Software Approach - Dynamic Rescheduling

The principle of hardware techniques used to support object-code compatibility is shown in **Figure 4. 15**(i). It reminds something of superscalar architectures in that they perform run-time scheduling in hardware. A limitation of hardware approaches is that the scope of scheduling is limited to the window of ops seen at run-time, hence available ILP is relatively less than what can be exploited by a compiler.

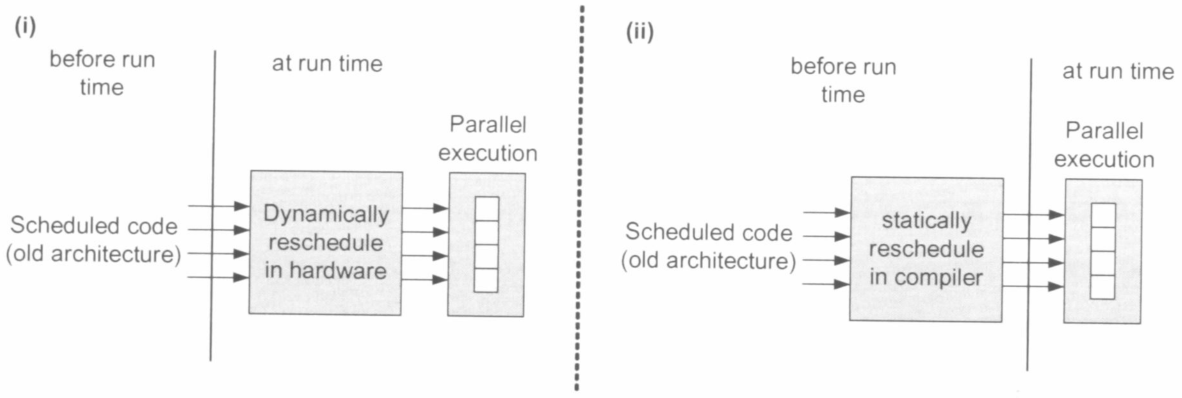

(i)
before run time
at run time
Parallel execution
Scheduled code (old architecture)
Dynamically reschedule in hardware
(ii)
before run time
at run time
Parallel execution
Scheduled code (old architecture)
statically reschedule in compiler

Figure 4. 15 : (i) The hardware approach to compatibility, (ii) The off-line (static) rescheduling of the program for compatibility.

Static recompilation is the most obvious software technique and is illustrated in **Figure 4. 15**(ii). It recompiles the entire program off-line, and hence can take advantage of sophisticated compiler optimizations to achieve desirable performance. However, the fact that an extra step is required in order to achieve code compatibility is a considerable drawback. It complicates both the development process and the user installation process.

Other software-based approaches, excluding source recompiling due to its inherent limitations, rely on binary-to-binary (object-code) translation, either with or without hardware support. Some of these actually correspond to translating binary code among different architectures, but they can be applied to different implementations of the same architecture, such as VLIW-based ones. Of particular relevance is the *Dynamic Rescheduling* software scheme, which applies a limited version of software scheduling during first-time page faults, requiring no additional hardware support. To make this practical, requires support from the compiler, the ISA, the operating system, and a fast algorithm for rescheduling.

Dynamic Rescheduling is illustrated in **Figure 4. 16.** When a program is executed on a machine generation other than what it was scheduled for, the dynamic rescheduler is invoked. The exact sequence of events is as follows: The OS loader reads the program binary header and detects the generation mismatch. After the first page of the program is loaded for execution, the page fault handler invokes the dynamic rescheduler module. The rescheduler reschedules the page for execution on the current host. This process is repeated each time a new page fault occurs. Translated pages are saved to swap space on replacement. Only the pages that are executed during the life-span of the program are rescheduled. The knowledge of architectural details of the executable's VLIW generation is necessary for the dynamic rescheduler to operate, and is retained in the executable image.

Dynamic Rescheduling is a promising general technique, capable of coping with any implementation constraint. Its limitations are the overhead introduced at page-fault time and the added complexity required to manage rescheduled and non-rescheduled pages. Finally, this scheme assumes that the object-code is already an explicit representation of ILP, and is able to exploit that representation accordingly.

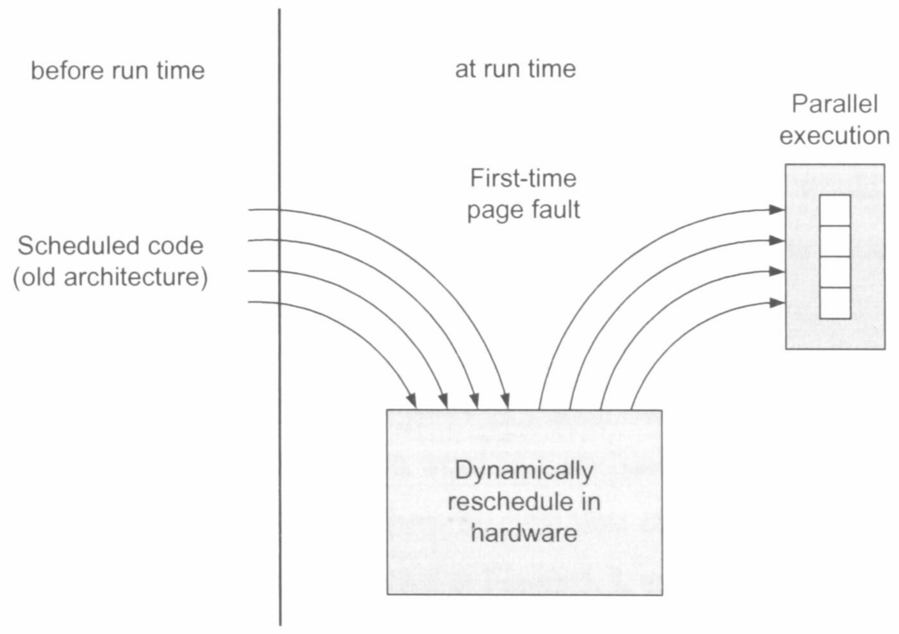

before run time
at run time
Parallel execution
First-time page fault
Scheduled code (old architecture)
Dynamically reschedule in hardware

Figure 4. 16 : Dynamic Rescheduling.

Chapter 5

5 DESIGN OF A VLIW INTEGER PROCESSOR

In this chapter I describe the architectural design, analysis and implementation of a very long instruction word (VLIW) processor. The object of this design implementation is to put into practice the basic design issues and organization techniques of the VLIW architecture and to come up against some of the implementation features and difficulties such architectures have. The implementation is divided into two parts. The first and also the most basic part of the processor is a non-pipelined version but with a five-stage pipeline in mind. The second part constitutes a more integrated version with a five-stage instruction execution pipeline added.

5.1 Processor organization

As we have already seen in Chapter 3, one way to increase performance is to combine different types of execution units into groups called Functional Units (FU). A simple way to keep a VLIW machine as clean as possible is to make each functional unit able to execute all types of operations supported in the instruction set. This would considerably simplify code generation because the compiler would not have to worry about assignment of operations to functional units.

However, this may not be very practical from an implementation point of view. Enabling sending every operation in the instruction word to every functional unit requires additional hardware resources and could also lengthen the cycle time of the processor. To avoid this, the approach I took in my design, as shown in **Figure 5. 1**, was to allow all functional units to be able to execute all types of operations, but also to limit the maximum number of control transfer and load/store operations in each instruction to one and two respectively.

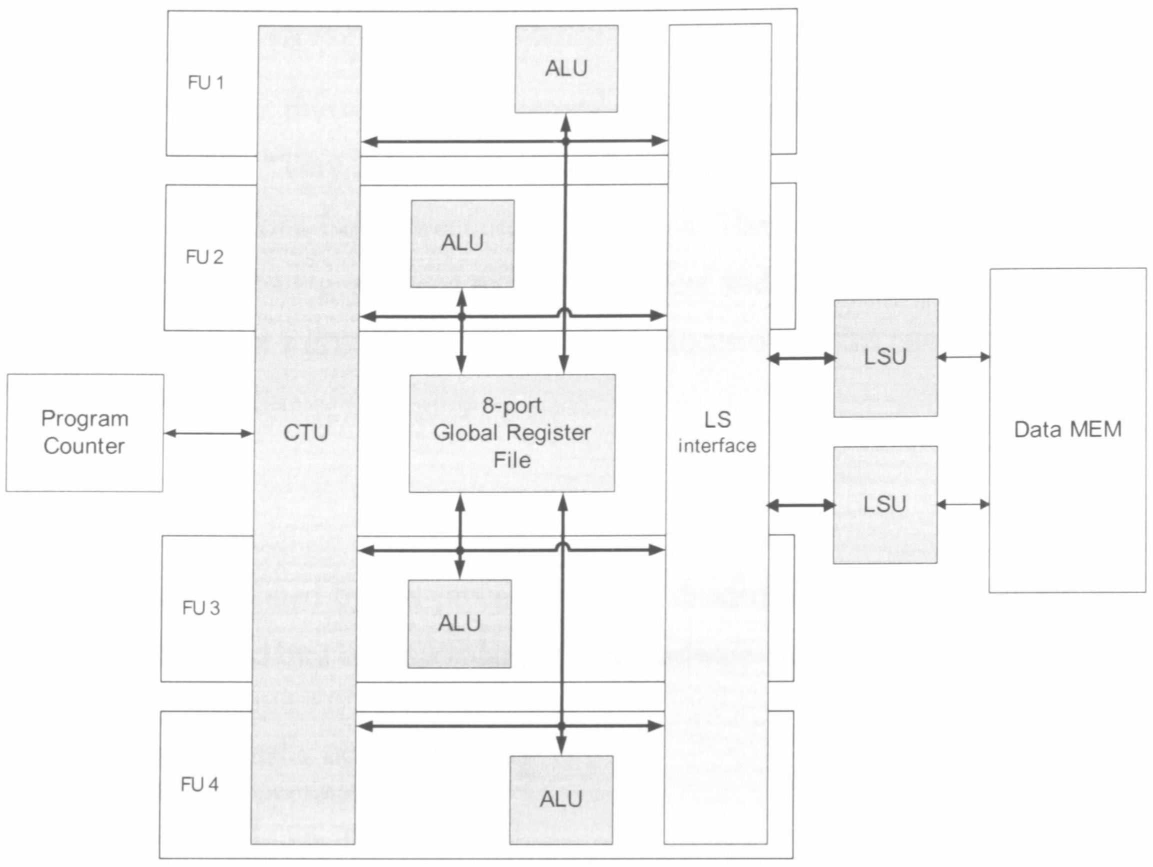


Figure 5. 1 : The general non-pipelined view of the organization of my VLIW processor design.

As shown in the figure, the processor contains four integer functional units that are connected through a shared eight-port register file. There are four ALUs, one for each functional unit, that allow the execution of instructions that have four AL operations. There is one Control Transfer Unit (CTU), which interacts with all the FUs, which means that there can only be one CT operation in each instruction word. Finally, there are two Load Store Units (LSUs) that also interact with all the FUs through a load-store interface. Therefore an instruction may contain at most two LS operations.

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

1. Each instruction contains four operations that are executed by four Functional Units.
2. There is one Control Transfer Unit that is shared to all the Functional Units, but only one FU can use it at a time.
3. There are two Load/Store Units that are shared to all the Functional Units, but at most two of the FUs can make parallel use of the two LSUs.
4. Each Functional Unit is able to execute all types of operations.

5.2 Processor Architecture and Instruction Set

The architecture has thirty-two 32-bit general-purpose registers where the value of R0 is always 0. All the "Very Long Instructions" are 128 bits and they consist of four independent operations (subinstructions) 32 bit each. These four operations in each 128-bit instruction word are guaranteed to be independent and are executed in parallel. In this section I will present a general description of the processor architecture and the format of the instruction set.

5.2.1 Operations

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

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

Each instruction can have a maximum of four operations, which are analyzed in:

1. <u>One Control Transfer</u> operation, which can be any of the four operations but cannot be followed by any other operation in the instruction.
2. <u>Two Load/Store</u> operations, which can also be any of the four operations.
3. <u>Four Arithmetic/Logic</u> operations.

Any of the general-purpose registers may both be loaded or stored, except the register R0. Register R0 is hardwired to contain zero at all times. Therefore, writing to R0 is allowed but has no effect on its contents.

It has some of the typical attributes of a RISC processor. It has a simple operation set that is designed with efficient pipelining and decoding in mind. All operations follow the register-to-register execution model. I will analyze the types of the operations in the following subsections.

5.2.1.1 Control Transfer Operations

Control is handled through a set of jumps and a set of branches. There are four jump instructions: *jump (J), jump-register (JR), jump-and-link (JAL) and jump-and-link-register (JALR)*. Using the instructions *J* and *JAL* the PC is set to the value defined by the 26-bit unsigned displacement. The two other jump instructions specify a register that contains the destination address. Moreover, *JAL* and *JALR* save the current PC value into register r31 enabling returning to that position. Finally, there are two branch instructions: *branch-equal (BEQ) and branch-not-equal (BNEQ)*. These instructions are conditional. They compare two registers and if the condition is true the new value of the PC is specified by the sum of the PC and the 16-bit sign-extended offset.

5.2.1.2 Load / Store Operations

Data memory is accessed with explicit Load/Store operations. These operations are the *load-word (LW)* and the *store-word (SW)*. They use the displacement addressing mode, base register + 16-bit signed offset, to load/store 32-bit data to/from a register from/to a data memory cell.

5.2.1.3 Arithmetic / Logic Operations

The Arithmetic/Logic operations that make use of the ALU are: *ADD, subtract (SUB), AND, OR, XOR, invert (NOT) and logical shifts (SLL and SRL)*. All these operations are also provided in immediate forms except for the *NOT* one. The operation *LHI* (load high immediate) loads the 16-bit offset to the top half of a register, while keeping the lower half intact. This allows a full 32-bit constant to be built in two instructions. There also conditional operations, which compare two registers. In these operations the destination register is set if the condition is true, otherwise value 0 is placed to it. These are: *set-less-than (SLT), set-equal (SEQ), set-not-equal (SNE) and set-less-equal (SLE)*. There are also provided immediate forms of these instructions, where a register is compared to an unsigned 16-bit immediate. All these AL operations perform at least one *read* from the register file and one *write* to it.

All the above operations, including their meaning, are summarized in the following table (**Table 5. 1**).

Instruction type / opcode	Instruction meaning
Control	***Conditional branches and jumps***
BEQ, BNEQ	Branch GPR equal / not equal
J, JR	Displacement jump, Register jump
JAL, JALR	Displacement jump and link, Register jump and link
Data transfers	***Move data between registers and memory***
LW, SW	Load word, store word (to / from integer registers)
Arithmetic / Logical	***Operations on integer or logical data in GPRS***
ADD, ADDI	Add, add immediate
SUB, SUBI	Sub, sub immediate
AND, ANDI	And, and immediate
OR, ORI, XOR, XORI, NOT	Or, or immediate, exclusive or, exclusive or immediate, invert
LHI	Load high immediate
SLL, SRL, SLLI, SRLI	Shift: left logical, right logical (variable and immediate form)
S__, S__I	Set conditional: "__" may be LT, EQ, NE, LE

Table 5. 1 : Complete list of the Long Instruction Word operations.

5.2.2 Instruction Format

As I have already mentioned in subsection 5.2.1, all of the four independent operations in an instruction are 32-bit long and they have a 6-bit opcode field. The register fields are 5-bit long, necessary to address the thirty-two registers of the Register File. The format of the operations is categorized into three types (R-type, I-type and J-type) according to the way they are encoded. A more detailed description of this operation layout and also the grouping of them is given in **Figure 5. 2**. Furthermore, in **Table 5. 2** I describe the encoding of these subinstructions.

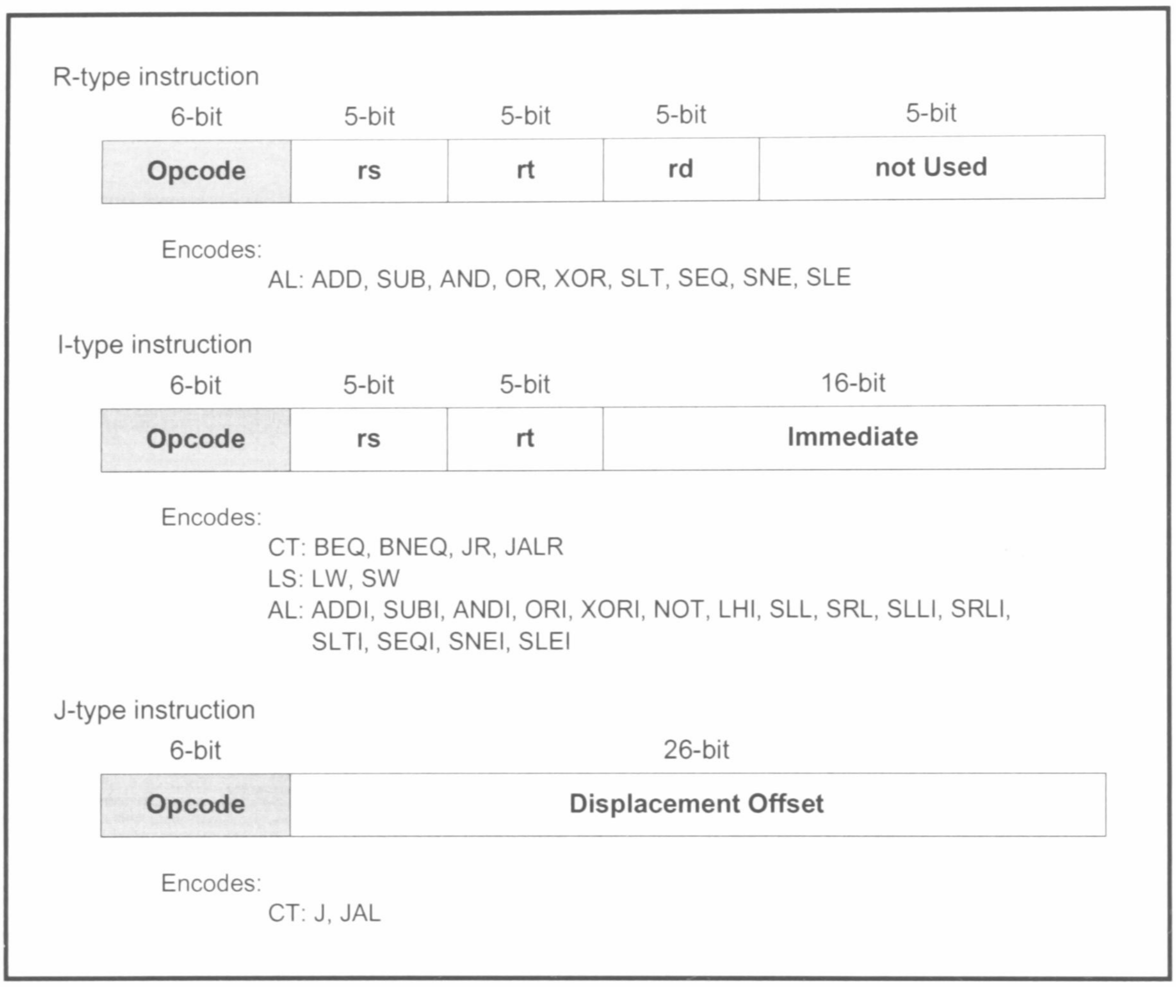


Figure 5. 2 : Layout of the operations of the instruction word. Each one is encoded in one of the three types.

Opcode	Instruction	Meaning
000001	ADD	RF[rd] ← RF[rs] + RF[rt]
000010	SUB	RF[rd] ← RF[rs] - RF[rt]
000011	AND	RF[rd] ← RF[rs] & RF[rt]
000100	OR	RF[rd] ← RF[rs] | RF[rt]
000101	XOR	RF[rd] ← RF[rs] ⊗ RF[rt]

000110	SLT	if (RF[rs] < RF[rt]) RF[rd] ← 1
		else RF[rd] ← 0

000111	SEQ	if (RF[rs] = RF[rt]) RF[rd] ← 1
		else RF[rd] ← 0

001000	SNE	if (RF[rs] != RF[rt]) RF[rd] ← 1
		else RF[rd] ← 0

001001	SLE	if (RF[rs] ≤ RF[rt]) RF[rd] ← 1
		else RF[rd] ← 0

001011	BEQ	if (RF[rs] = RF[rt])
		PC ← PC + 1 + *SignExtend*(Imm)
		else PC ← PC + 1

001100	BNEQ	if (RF[rs] != RF[rt])
		PC ← PC + 1 + *SignExtend*(Imm)
		else PC ← PC + 1

001110	JR	PC ← RF[rt] (rt = 0)
010000	JALR	RF[r31] ← PC + 1, PC ← RF[rt] (rs = 0)
010101	LW	RF[rt] ← MEM[RF[rs] + *SignExtend*(Imm)]
010110	SW	MEM[RF[rs] + *SignExtend*(Imm)] ← RF[rt]
100001	ADDI	RF[rt] ← RF[rs] + *SignExtend*(Imm)
100010	SUBI	RF[rt] ← RF[rs] - *SignExtend*(Imm)
100011	ANDI	RF[rt] ← RF[rs] & *ZeroFill*(Imm)
100100	ORI	RF[rt] ← RF[rs] | *ZeroFill*(Imm)
100101	XORI	RF[rt] ← RF[rs] ⊗ *ZeroFill*(Imm)
010100	LHI	RF[rt] ← Imm << 16
010010	SLL	RF[rt] ← RF[rs] << 1
010011	SRL	RF[rt] ← RF[rs] >> 1
110010	SLLI	RF[rt] ← RF[rs] << *ZeroFill*(Imm)
110011	SRLI	RF[rt] ← RF[rs] >> *ZeroFill*(Imm)

100110	SLTI	if (RF[rs] < *ZeroFill*(Imm)) RF[rt] ← 1
		else RF[rt] ← 0

100111	SEQI	if (RF[rs] = *ZeroFill*(Imm)) RF[rt] ← 1
		else RF[rt] ← 0

101000	SNEI	if (RF[rs] != *ZeroFill*(Imm)) RF[rt] ← 1
		else RF[rt] ← 0

101001	SLEI	if (RF[rs] ≤ *ZeroFill*(Imm)) RF[rt] ← 1
		else RF[rt] ← 0

010001	NOT	RF[rt] ← ! RF[rs]
001101	J	PC ← displacement
001111	JAL	RF[R31] ← PC + 1, PC ← displacement

Table 5. 2 : Encoding of the operations of the instruction word.

5.3 Processor Core Design

In this section the basic components of the processor and explain their characteristics and their function.

5.3.1 Instruction Decoder

Each instruction word loaded form the instruction cache consists of four independent operations. The instruction decoder has the role of separating the instruction into the four subinstructions and then to perform a classic decoding for each one of them. The fields to which each operation is decoded are:

- 6-bit for the **opcode**
- 5-bit for the **rs**
- 5-bit for the **rt**
- 5-bit for the **rd**
- 26-bit for the **offset**, from which is also extracted the 16-bit **immediate**.

Then each field of each operation is grouped together with the corresponding fields of the other operations building arrays of same fields. The block diagram of the Instruction Decoder, in the **Figure 5. 3**, provides a graphical representation of this decoding.

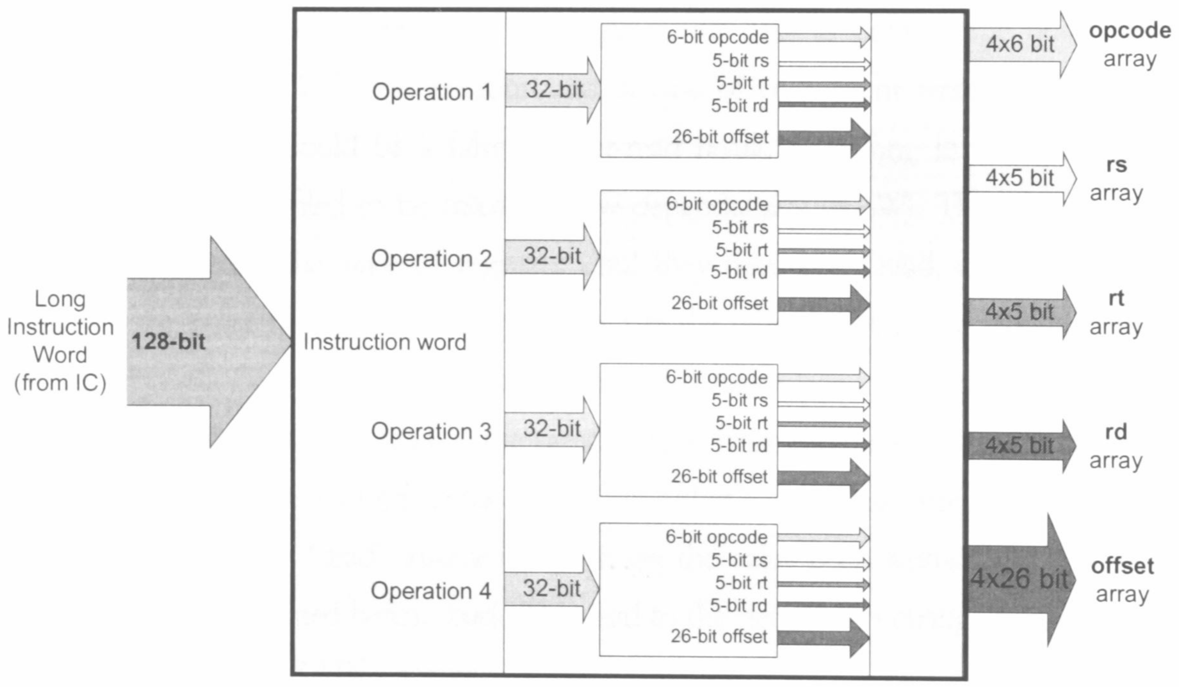


Figure 5. 3 : Instruction Decoder Block Diagram.

After the decoding of the instruction, the *opcode* array is driven to the Control Unit and to the ALU's. As defined by the ISA of the processor the fields of rs and rt provide the source registers, thus the arrays of *rs* and *rt* are driven to the read-address ports of the Register File. Moreover, the destination register is given by either the *rd* or *rt* depending on the type of the operation. Therefore, a multiplexer will have to choose the field of the destination register, for each operation, according to its type. Finally, each *26-bit offset* is either zero-filled or is being used to extract the 16-bit immediate (sign-extended or not).

5.3.2 Register File

As I have already discussed in section 5.2 at the heart of the processor core is a multi-port register file that can hold thirty-one of the thirty-two 32-bit general purpose registers as the register R0 is hardwired to contain zero at all times. All the four operations of the instruction word may be ALU operations which means that the register file must be able to allow eight read and four write concurrent accesses, two reads and one write for each operation. Thus, it has four write data ports and eight read data ports with the corresponding 5-bit address ports. In each clock cycle it can perform eight read and four write accesses. Reading the register R0 always gives zero while writing to R0 is allowed but has no effect on its contents.

There are two different designs for the register file. That is because the non-pipelined design is a single-cycle implementation and in case of concurrent write and read of the same register there could be a false register-read result. In a long instruction word, all operations are scheduled to be freed of flow dependencies (RAW). Thus, register writes should not change the state of registers until they have been read, and in that case I followed the write-after-read (WAR) scheme. On the other hand, in the pipelined version of the processor, register writes and reads in a LIW take place in separate pipeline stages, but a concurrent write and read of the same register from different LIWs may happen. In that case, when an instruction writes on a register that a successor instruction reads in the same clock cycle, the "read" instruction must get the value been written and thus register *writes* must be performed before *reads*. This lead to the decision to change the RF function to a read-after-write (RAW) scheme.

In **Figure 5. 4** I present a detailed description of the register file implementation. The register file consists of thirty-one separate registers that are handled through special decoders and multiplexers, in order to perform the register-write function, and also eight multiplexers that represent the eight read data ports.

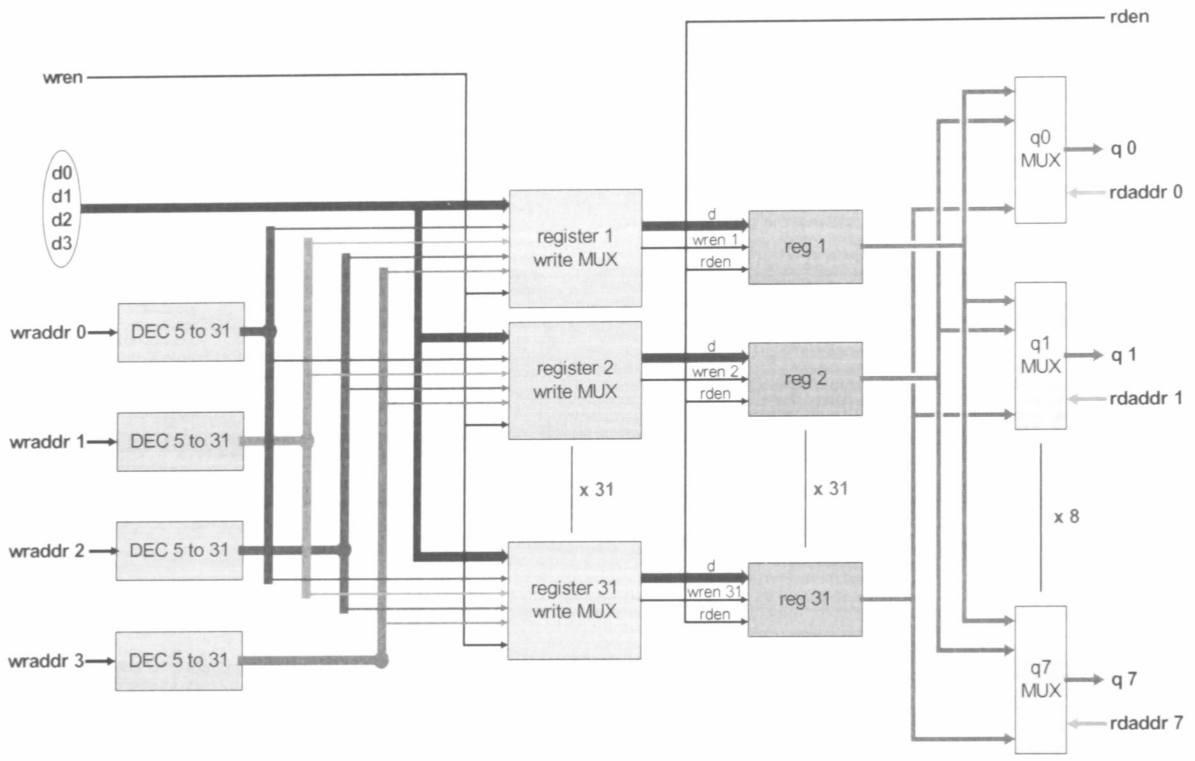


Figure 5. 4 : Register File Block Diagram.

The values that are inputs to the read-address ports come directly from the Instruction Decoder, and the output data are lead to the ALU and to some other components that construct an interface between the Datapaths and the Register File. On the other hand, the write addresses can be defined in proportion to the type of each operation, where in some operations the destination register is defined by the *rd* field and in some others is defined by the *rt* field, while in *JAL* and *JALR* operations the destination register is the *r31*. Therefore, a write-address multiplexer is added to select the proper destination register. Finally, the data to be written to the Register File may come from the ALU, the Data Cache or the Program Counter, so a destination-data multiplexer is placed at the Register File data input ports.

5.3.3 Functional Units

Generally, in VLIW literature, Functional Units are groups in which different types of execution units are combined. All the FUs in the processor are identical as they are all able to execute all types of operations. A general view of the Functional Units has also been given in **Figure 5. 1**.

The processor's data path consists of four 32-bit identical data paths, which are capable of executing all of the arithmetic and logic functions in the instruction set. Each of these data paths consist of two major blocks:

- An operand unit, which provides an interface between the data path and the register file.
- An Arithmetic/Logic Unit, which is capable of integer addition and subtraction, numerical comparison functions, logical and arithmetic shift operations, and logic function computation.

Thus every functional unit contains one **ALU**, and as a result the processor has four ALUs. The data of the register defined by the rs field is always the one of the two operands in any ALU function and is the first input to the ALU. The other source operand comes either from the register rt or from the 26-bit offset or from the 16-bit, sign-extended or not, immediate. Thus, one multiplexer is placed at the second ALU input in order to select between a register, an offset or an immediate.

The processor organization contains two Load/Store Units, but all the functional units must be able to have access to these LSUs as they can execute all types of operations. To implement this, I include a special interface circuitry in the two load/store units to interact with the data memory. This is reflected in **Figure 5. 5**, which contains the block diagram of this design.

The store-data come from the register file and the interface provides multiplexers to define which of the four operations uses the first LSU and which uses the second. There are also similar multiplexers that define which operation gives the write/read address for memory access and which operation reads data from memory through the first or the second LSU.

The processor also contains one Control Transfer Unit which is also usable by all the functional units, as the operation that performs the control transfer can be anywhere in the instruction. Similarly to the LSUs, the PC must also comply with this flexibility feature. The organization of the program counter is shown in the block diagram in **Figure 5. 6**, where additional multiplexers are used to define which operation is the one that performs the control transfer.

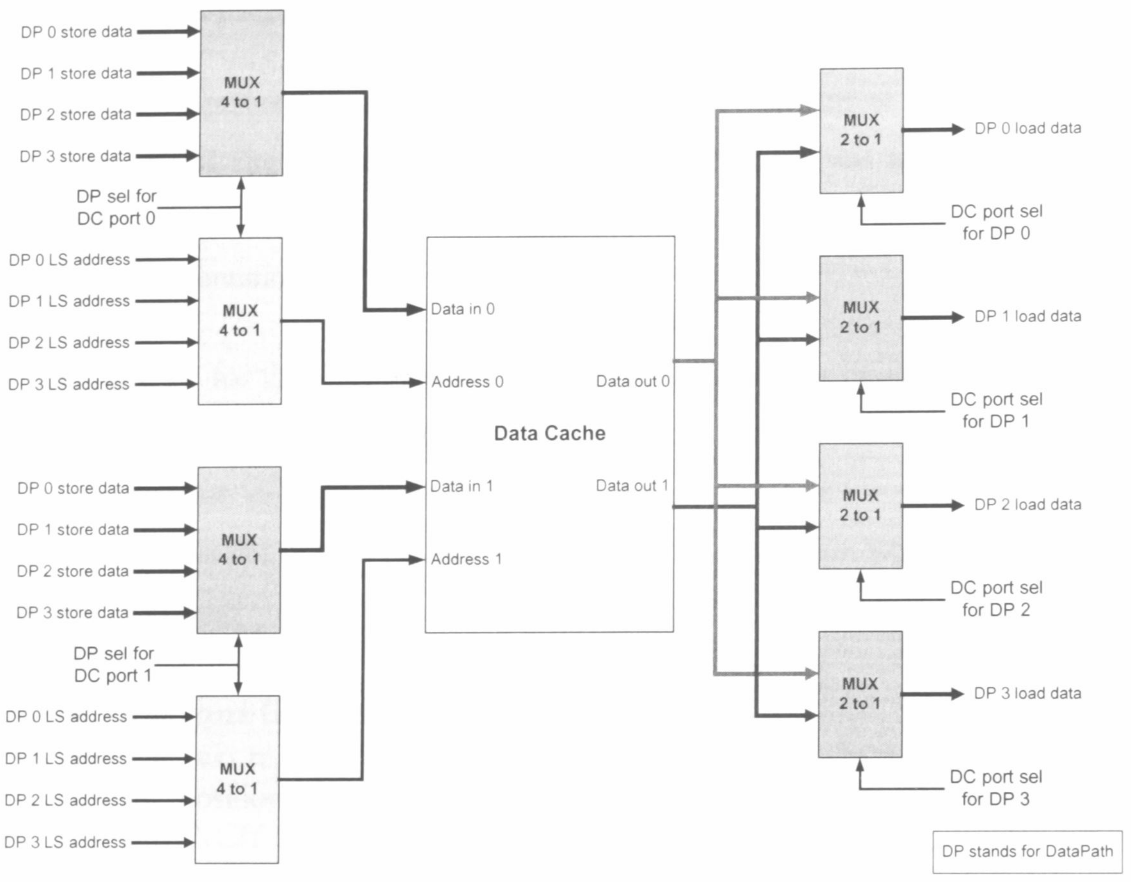


Figure 5. 5 : Block Diagram of the Load / Store Interface Circuitry.

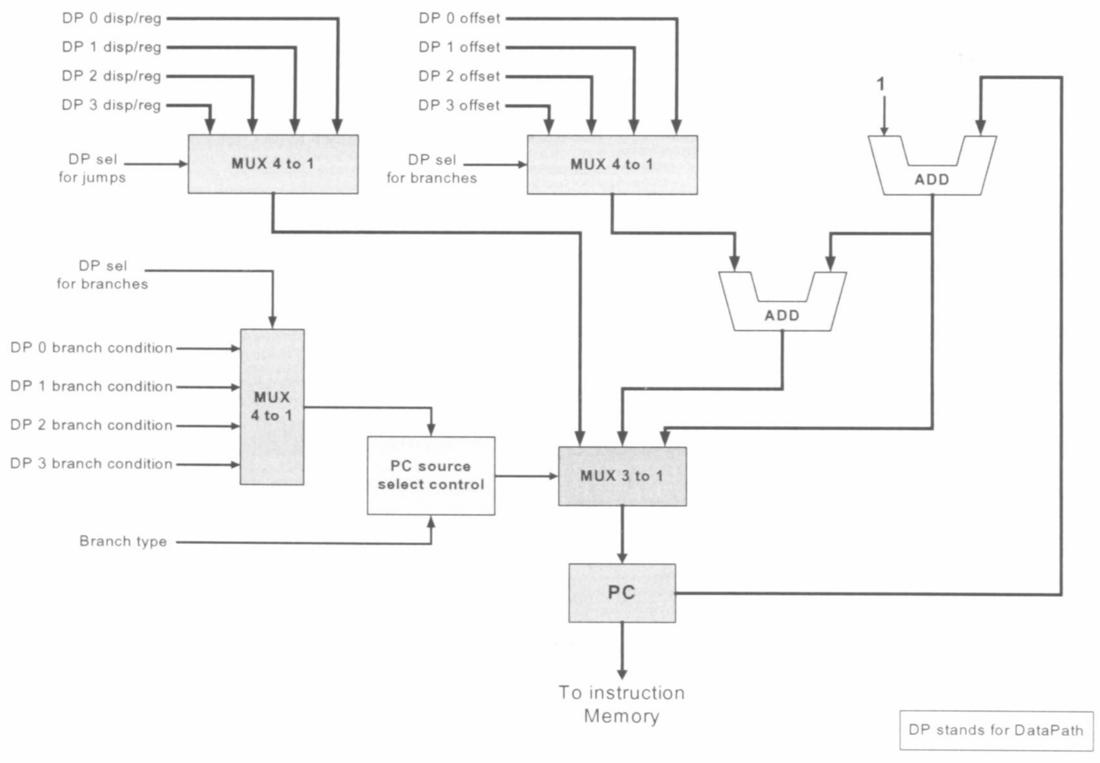


Figure 5. 6 : Block Diagram of the Program Counter Unit.

5.3.4 Control Design for the Non-Pipelined Implementation

As I have mentioned at the beginning of this chapter, the first part of the implementation is a non-pipelined single cycle implementation. The objective was to verify the proper functionality of the processor and thus simplify the extension to an efficient five-stage pipelined implementation.

The control logic for this implementation is distributed into four blocks, each of which is tightly coupled to the functional unit with which is associated. As the four functional units of the processor are identical, these four blocks of the control are identical too. The major observations about the format of each operation in the instruction word that we will rely on are the following:

- The *opcode* field is always contained in bits 31-26.
- The two registers to be read are always specified by the *rs* and the *rt* fields, at bit positions 25-21 and 20-16. This is true for the *R-type* instructions, *BRANCH* and for *STORE*.
- The base register for *LOAD* and *STORE* instructions is always in bits 25-21 (*rs*).
- The 16-bit immediate for *BRANCH*, *LOAD* and *STORE* instructions is always in bit positions 15-0.
- The 26-bit offset for *JUMP* instructions is in positions 25-0.
- The destination register is in either defined by the *rt* or the *rd* field. For *LOAD*, *NOT* and for all *AL* instructions with immediate it is in position 20-16 (*rt*), while for an *R-type* instruction it is in bits 15-11 (*rd*).

The bit-mapping of the operation format is also shown in **Figure 5. 7**. As I have discussed in section 5.3.1, instruction decoding following this scheme is hardwired and the only input the Control Unit needs is the opcode field. Moreover, the ALUs use this opcode field for *ALUop* and as a result there is no need for any signal to control ALU, except from those which control the four multiplexors needed to define whether the input of the four ALUs comes from the register file or the 16-bit immediate.

There are many control signals that must be set. For the Register File, the Control Unit controls:

- The write-enable signal.
- The multiplexor that defines the source of the data to be written.
- The multiplexor, which defines the field of the destination register.

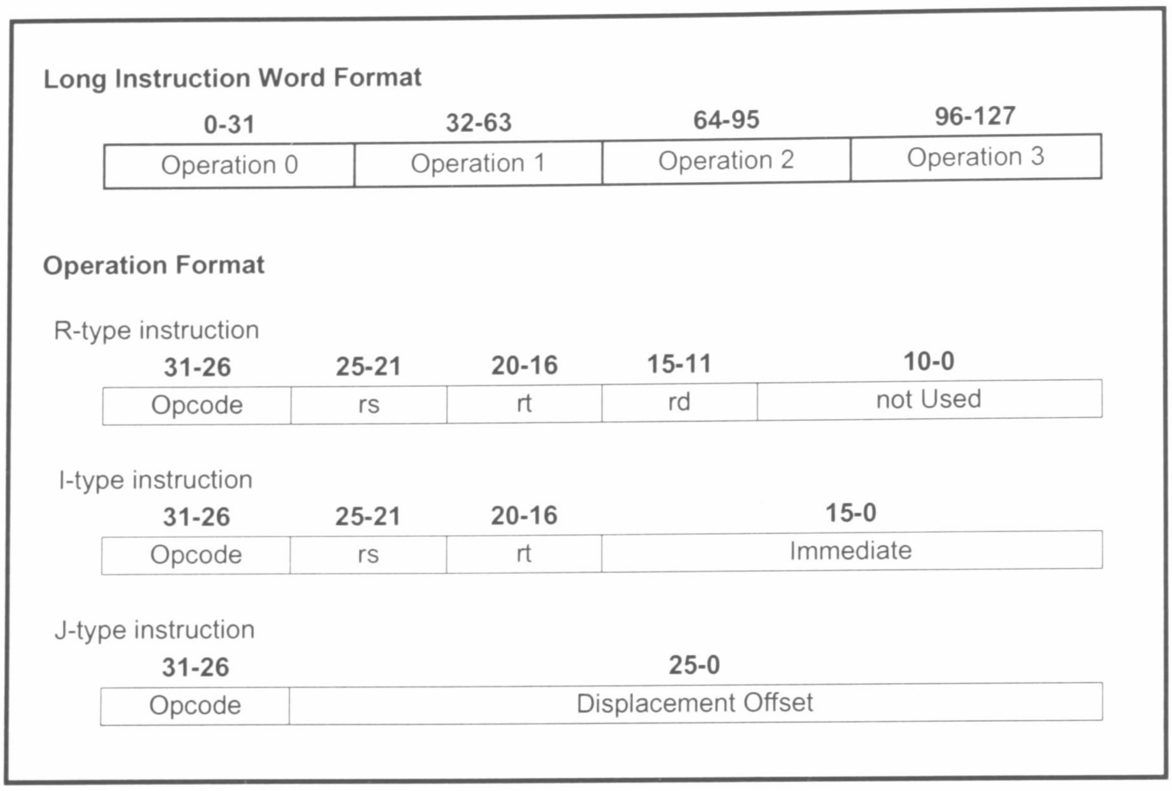


Figure 5. 7 : Bit-Mapping of the instruction Word and the operations.

Moreover, the Control Unit is the one that manages the two ports of the Data Cache, where the first LS operation in the instruction uses the first DC port whereas the other port is used by the next LS operation in the same instruction. Thus, for the Load / Store Units and Interface the control values:

- The read-enable and write-enable signals of the Data Cache.
- The multiplexors in the LS interface (**Figure 5. 5**) that define which operation writes to each of the two write ports of the DC.
- The multiplexors in the LS interface that define which operation reads from each of the two ports of the DC.

The control also defines some control signal for the CTU. These are:

- Defines whether a register or an offset is going to be used as the PC displacement, in case of a *JUMP* operation.
- Defines the source of the new value of the Program Counter. This can be a value given by a *JUMP* operation, a branch target value, or the next value of the counter.
- Defines the operation that causes the Control Transfer.

Finally, the 16-bit immediate is passed through a sign-extender, where the control also decides whether to sign-extend the immediate value or just fill it with zeros to produce a 32-bit value.

The Instruction Execution Steps

Having discussed the organization of the parts of the processor and defined the signals of the Control Unit, I am able to describe the steps of the execution for each of the four operations of the Instruction Word.

1. An 128-bit long Instruction Word is fetched from the Instruction Cache, passed to the Instruction Decoder and the Program Counter is incremented.
2. The ID decodes the four operations contained in the instruction and the two registers specified by the *rs* and *rt* fields are read from the Register File. The Control Unit also reads the four opcodes.
3. Depending on the type of the operation there are four cases:
    - In case of AL operation, the ALU operates on the data read from the register file if it is an R-type operation, or on the value of register *rs* and the immediate if it is I-type, using the instruction opcode to generate the ALU function.
    - In case of branch operation, the corresponding data path is selected, and then the ALU is used to compare the data read from the register file in order to define the branch condition. The branch target is also calculated.
    - In case of jump operation, the Control firstly selects the value of register *rs*, if it is for a jump-register operation, or the 26-bit offset if it is for a jump-offset one, and then selects this selected value given by a *JUMP* operation as the new value for the PC.
    - In case of a load/store, the ALU calculates the sum of the value of register *rs* and the sign-extended immediate, in order to produce the memory address.
4. Depending on the type of the operation there are five cases:
    - In case of R-type operation, the result from the ALU is written into the register file selecting register *rd* as the destination register.
    - In case of I-type AL operation the result from the ALU is written into the register file selecting register *rt* as the destination register.
    - In case of branch operation, the control unit selects the previously calculated branch target value as the new PC value, if the condition is true, or else the incremented PC value if the condition is false.
    - In case of jump-and-link operation, the old value of the PC is written into the register **r31**.
    - In case of load/store operation, the memory is enabled for read or write respectively, using the address previously calculated by the ALU.

5. If it is a load operation, the data read from memory is written into the register file using register *rt* as the destination register. For a store one, the value of register *rt* is stored into the memory.

This is the general description of how the instructions operate in steps. But, there are distinct features as far as load/store operations are concerned. When there is only one LS operation in the instruction word, then the port-set 0 (Data in 0, Data out 0 and Address 0) is selected for memory access, while the corresponding data path is also selected for this set of ports. In case there are two LS operations in the instruction word, then the port-set 0 is selected by the first LS instruction in the instruction word, selecting its data path for port-set 0, and the port-set 1 is selected by the second LS instruction in the instruction word, selecting its data path for port-set 1.

5.3.5 Instruction Cache

The basic characteristic of VLIW architectures is that of the large instruction word, which as a result requires an instruction cache quite different of that of standard processors. Specifically, in my processor the instruction word is 128-bit wide. Thus the processor instruction memory subsystem must have the capability of fetching the 128-bit instruction word in a single cycle. In order to fetch such an instruction in a single cycle, the read bandwidth of the cache must be four times greater than that of a single processor machine.

The instruction address width the processor supports is 32 bit. However, the instruction cache I implemented can hold 256 long instructions, thus its address bandwidth is 8 bits, and it is able to provide the output in a single cycle.

As the most important goal of this processor implementation is to present the basic design and organization issues of a VLIW architecture, in order to keep the implementation simple enough but also between these goal boundaries, I chose to use this cache as the basic memory for instructions. This means that the processor does not mind for cache misses or hits, but assumes that the whole program is put in this memory.

5.3.6 Data Cache

The data memory subsystem of the processor must be designed to meet several requirements unique to a VLIW architecture. Since it has two parallel data ports both, for load or store, the data memory subsystem must be able to service two requests per cycle. The two data ports are independent, so both ports are able to load or store data regardless of the operation of the other port. Moreover, the data ports' width is 32 bits and so is the width of the address ports.

Thus, I implemented a dual-port memory scheme in order to service both data ports of the processor independently. Unfortunately, using the tools provided by the design application I wasn't able to implement the memory scheme I intended to. Therefore, I employed a dual-port memory with two read ports and one write port. As a result this scheme supports two independent loads, or a load and a store simultaneously, without having the ability of two simultaneous stores. Moreover, storing and loading from the same memory address is allowed but since it follows the WAR scheme it loads the same value being stored at the same time.

The memory size is 256 kilobyte with 16-bit wide address ports, and can hold 65536 data words. For the same reason I mentioned above, in the instruction cache section, the processor assumes that this is the only memory for data access, and the processor does not concern about cache manners.

The block diagram for the non-pipelined implementation of the processor is shown in the following figure (**Figure 5. 8**).

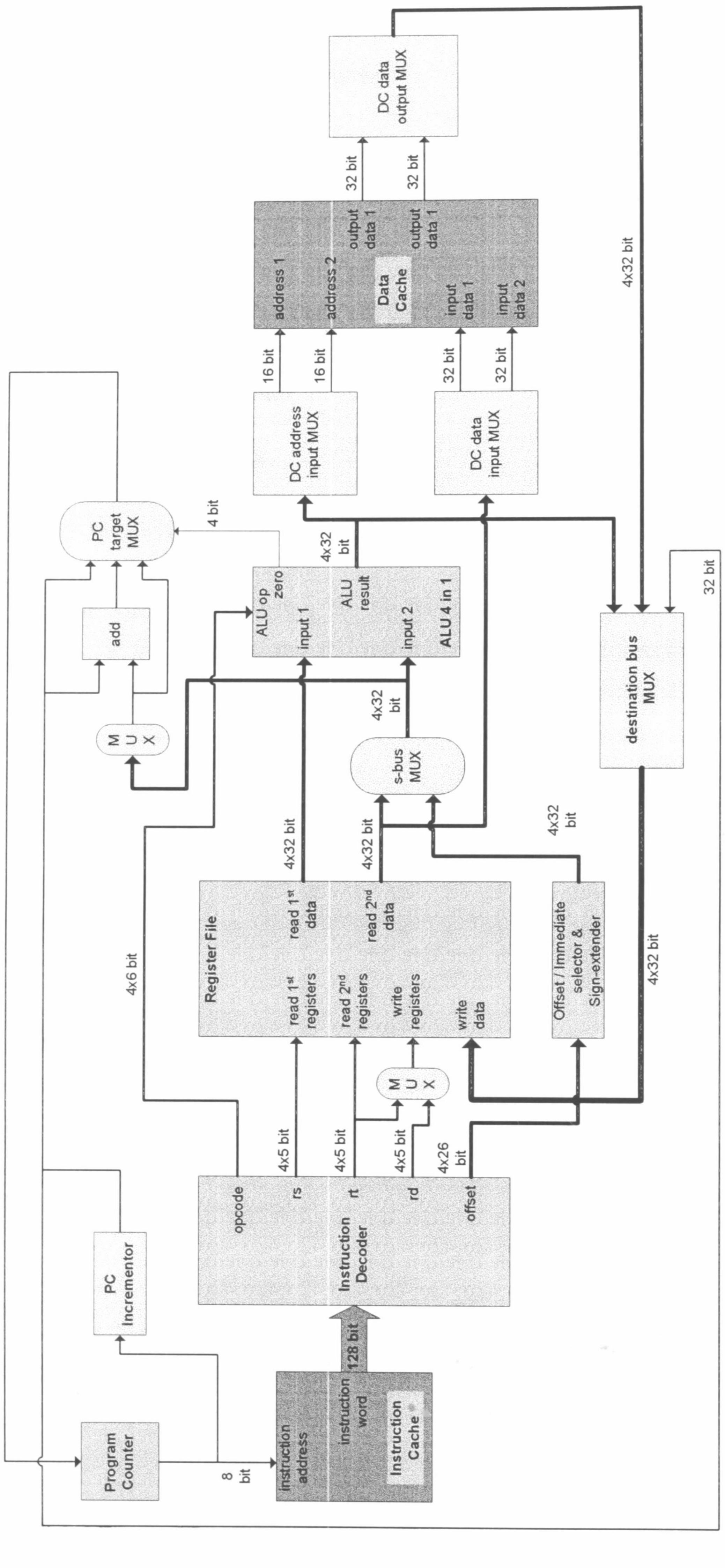


Figure 5. 8 : The Complete Datapath for the Single Cycle Implementation of the VLIW Processor.

5.4 Pipelining the Processor

As I said at the beginning of this chapter, the processor implementation is divided into two parts. In the previous sections I presented and analyzed the non-pipelined version, which constitutes the most basic part of the processor. The second part that I will describe in this section constitutes a more integrated version with a five-stage instruction execution pipeline added. I will discuss some issues about the pipeline structure, hazards and bypassing. Moreover, I will provide some changes that may be required in order to implement the pipeline design.

5.4.1 Pipeline Structure

One of the most important goals of RISC processors is efficient pipeline. Because of the simplicity of the instruction set, processors are also simple and this makes the pipeline hardware quite simple to design. The pipeline structure typically used in RISC processors consists of five basic stages: IF (Instruction Fetch), ID (Instruction Decode), EX (EXecute), MEM (MEMory access), WB (Write Back), and it is shown in **Figure 5. 9**.

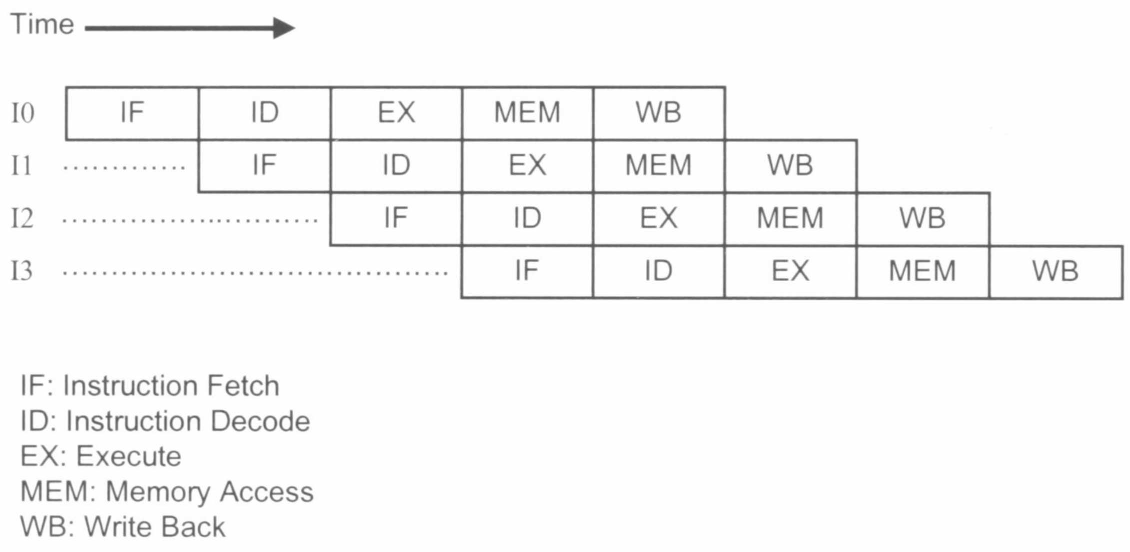


Figure 5. 9 : Pipeline Structure of Typical RISC Processors.

Because the VLIW processor I designed has some of the typical attributes of a RISC processor, having an operation set that follows the idea of the RISC instruction set, I decided to illustrate its pipeline according to the typical pipeline structure used in RISC processors, as described above. So, in the single-cycle implementation described in previous sections, I divided the operations into five stages, which means a five-stage pipeline, which in turn means that five instruction words will be in execution during any single cycle. The steps are the same with the ones of typical RISC processors and the general structure is shown in **Figure 5. 10**, where we can also see the parallel execution of the four operations contained in an instruction.

My processor has four operations in each instruction word, which means that the performance depends not only by the possible stalls happening but also the number of NOP's in an instruction. Thus, an efficient scheduling during compile-time is necessary in order to achieve efficient pipeline for a VLIW processor.

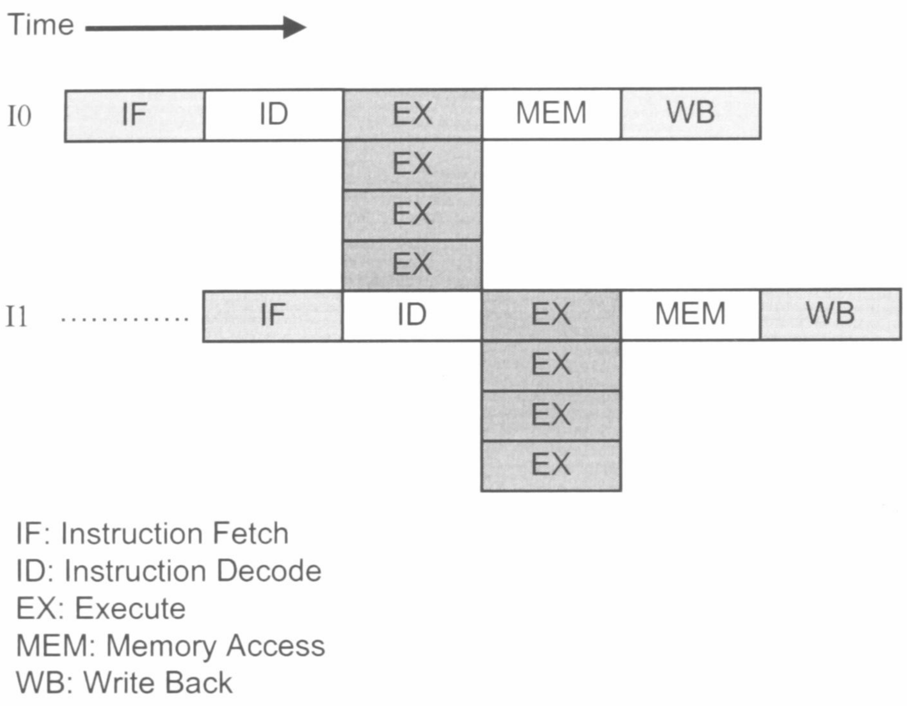


Figure 5. 10 : Pipeline Structure of my VLIW processor.

In order to describe how this pipeline works and also the five stages, I used the classic five-stage pipeline datapath representation as described in [ref. 1], and I separated the single cycle implementation datapath in **Figure 5. 8** into five parts that denote these five pipeline stages, as shown in **Figure 5. 11**.

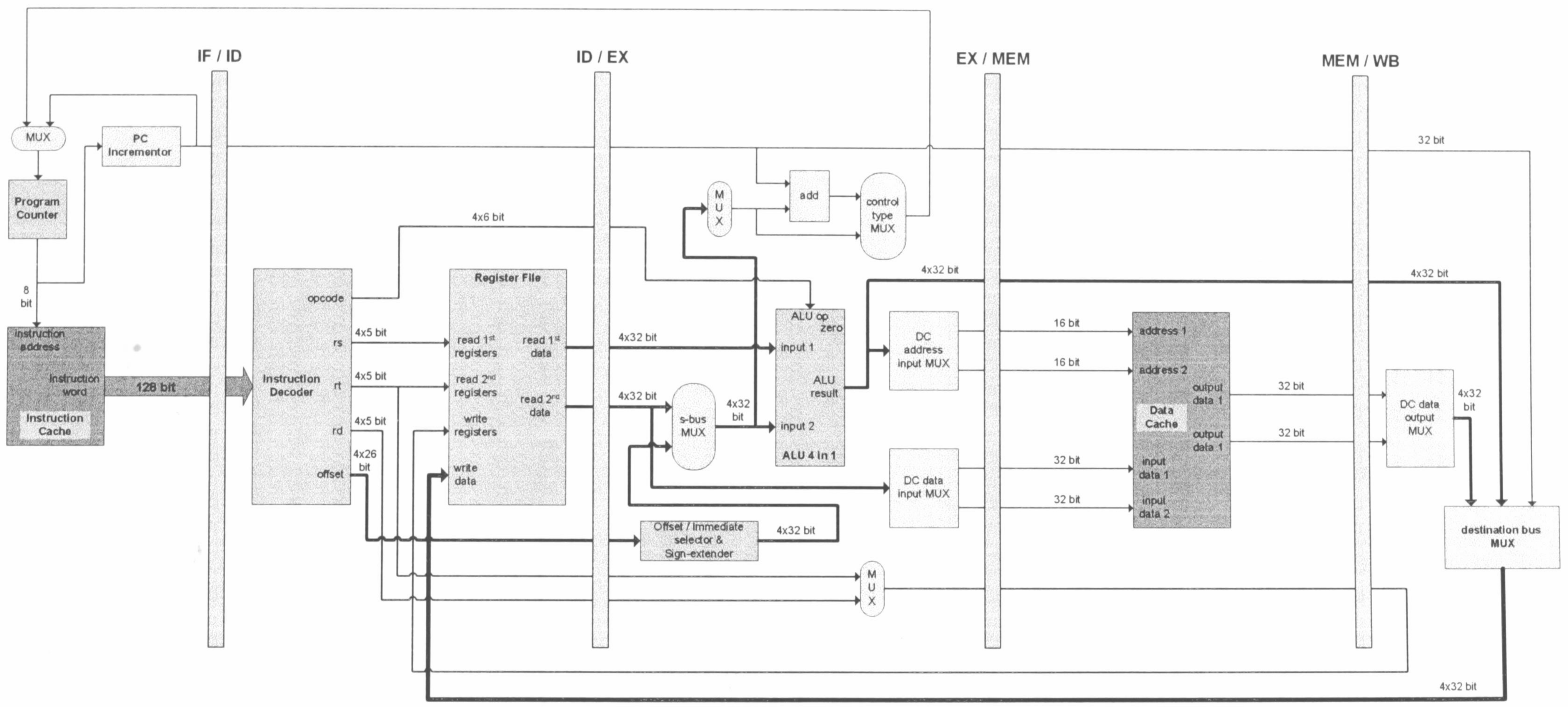


Figure 5. 11 : The Pipelined Version of the Processor. The Datapath is separated in the five stages. The pipeline registers, in orange colour, separate each pipeline stage and are labeled by the stages they separate.

Instructions and data move generally from left to right through the five stages as they complete execution. There are however two exceptions to this left-to-right flow of instructions: The selection of the next value of the program counter which happens in the EX stage, and the register-write action that takes place during the write-back stage. Moreover, between each of the five components of the datapath division, there are the pipeline registers that carry the needed signals from the one pipeline stage to the other and they are represented by the elongated orange rectangles.

A brief description of the five pipeline stages and which parts of the execution take part in each of them, as shown in **Figure 5. 11** is given below:

1. *Instruction Fetch (IF):* The program counter addresses the instruction cache and is also incremented to the next value, which is passed to a multiplexer that selects the PC target value to be ready for the next clock cycle. The new incremented PC value and the new instruction read from the cache are placed in the IF/ID pipeline register.

2. *Instruction Decode and Register Read (ID):* The instruction decoder reads the instruction from the IF/ID pipeline register, and supplies the register numbers to read the eight registers (two for each operation). The decoder also supplies the four opcodes and the 26-bit offsets that are transferred through the ID/EX pipeline register to the next stage. Moreover, the register numbers in the *rd* and *rt* fields are also stored in the ID/EX pipeline register as they will be used to define the register write addresses. Furthermore, the data read from the register file are stored into the ID/EX pipeline register too for later use. Finally, the incremented PC value is passed from the IF/ID pipeline register to the ID/EX one.

3. *Execution (EX):* The opcodes and the values that represent the first operand are driven from the ID/EX pipeline register directly to the ALUs. The offsets also read from the register are either zero filled to 32 bits or, selecting the lower 16 bits, sign-extended to 32 bits and then sent to the s-bus multiplexer. The values read from the second registers are passed from the ID/EX pipeline register to the DC input multiplexer and also to the s-bus multiplexer, which selects the second operand for each operation and directs them to the ALUs. The DC input multiplexer selects the two data to be written to the memory and sends them to the next pipeline register, while the ALUs results are used by the DC address MUX to

define the two addresses for the memory access that are also placed to the EX/MEM pipeline register. In addition to that, the ALU results are stored into the EX/MEM pipeline register too. Moreover, the register-write-address multiplexer takes as input the *rd* and *rt* register numbers from the ID/EX pipeline register and the selected write-registers are stored to the EX/MEM pipeline register. In case of a *branch* operation the four sign-extended immediate's are driven to a multiplexer that selects the immediate of the appropriate operation, which is then added to the PC value read from the ID/EX pipeline register and sent to the PC target MUX. Moreover, in case of a *jump* operation the 32-bit extended offset is also sent to the PC target MUX, which selects the appropriate target value for the program counter. This new PC value is loaded to the PC if it is a jump operation or if it is a taken branch, and the loading takes action in the same clock cycle providing the new address for the IC in the next cycle. Finally, the PC value read from the ID/EX pipeline register is transferred again through the EX/MEM one to the next stage.

4. *Memory Access:* The EX/MEM pipeline register provides the two addresses for the memory access and also the two values to be written to each of the two memory ports are driven to the data cache input ports. Moreover, the data loaded from the memory are stored to the last pipeline register (MEM/WB). Finally, the ALUs results, the program counter value, and the write-addresses for the register file are transferred from the EX/MEM pipeline register to the MEM/WB one.

5. *Write Back:* The write data are transferred from the MEM/WB pipeline register directly to the write ports of the register file. The destination addresses are defined by the destination bus multiplexer that takes as input the data loaded from the memory, the program counter and the results of the ALUs, that are all given by the MEM/WB pipeline register.

5.4.2 Pipelined Control

Just as we added control to the simple non-pipelined datapath in subsection 5.3.4, I will now define control for the pipelined implementation. The control logic for this implementation is similar to that in the non-pipelined version and the only input needed to define the control signals is the opcode for each one of the four operations. The control unit takes the opcodes as soon as they are exported from the instruction decoder thus it takes action in the EX stage of the pipeline too. It controls all the processor components in all stages and each control line is associated with a component active in only a single pipeline stage. Thus, I will divide the control lines into five groups according to the pipeline stage:

1. *Instruction Fetch:* The control has nothing to do in this stage as the control signal needed for the instruction memory read and the PC write are always asserted.

2. *Instruction Decode and Register Read:* Similarly to the previous stage there is nothing to control in this stage too.

3. *Execute:* The signal that have to be set are:

    - Four signals, one for each operation, to control the *offset/immediate selector and sign-extender* component in order to select between the 32-bit zero-extended 26-bit offset, the zero-extension of the 16-bit unsigned immediate and the sign-extension of the 16-bit sign immediate.
    - Four signals to select either a register or an immediate for the ALU, or to select either a register or an offset in case of a *jump* operation according to its type.
    - Four signals to select the result registers
    - One signal to define which operation performs a control transfer and also one signal to select between a *branch* or a *jump*.
    - Four signals to select which *load/store* operation uses the first memory port and which one uses the second memory port. These signals are used for both the address and data port multiplexers.

4. *Memory Access:* The only signals that must be set in this stage are the memory write-enable signal in case of store operations.

5. *Write Back:* Two control signals are needed to define which operation performs a memory load using the first memory port and which using the second one. Moreover, four control signals are set to control the destination bus multiplexer which decides which value to send to the each one of the four write ports of the register file. Finally, it sets the register-write-enable signals.

I should also notice that all the signals are transferred to the stages where are being used through the ID/EX, EX/MEM and MEM/WB pipeline registers.

5.4.3 Bypassing and Stalls

5.4.3.1 Data Hazards

As I have described the register file is separated into two halves and each one takes action in different pipeline stages. The registers read happens during ID and registers write during WB. Thus, some operation may read a register that is to be written by another operation, which is in EX or in MEM stage. Meanwhile, there is no problem if the operation that is about to write the required register lies at the WB stage, as the register file follows the RAW scheme and the operation in ID reads the new value written in the WB stage at the same clock cycle. Such dependencies, also called data hazards, seem to be a quite complicated issue for a VLIW processor and as a result for this processor too.

For each one of the four operations there must be performed data dependency tests between both the two read registers at the ID stage and the write registers of all the four operations both at the EX and the MEM stage of the pipeline. In a typical RISC processor pipeline there are required *2d* comparators (where *d* is the number of pipeline stages between ID and MEM) to discover the possible data dependencies between different stages, whereas in a VLIW pipeline the number of required comparators is $2dn^2$ where n is the number of the functional units. Thus, for a four-operation architecture with five-stage pipeline, like this one, there are needed 64 comparators to construct the data hazard detection unit.

In order to detect the data dependencies I designed a data hazard detector unit, which performs the necessary comparisons. I will define the comparisons describing separately the detection performed for the EX stage and for the MEM stage:

1. EX hazard (32 comparisons):

```
for i in 1 to 4 loop
   for j in 1 to 8 loop
      EX(reg_we(i)) and (EX(write_register(i)) = ID(read_register(j))
   end loop
end loop
```

2. MEM hazard (32 comparisons):

```
for i in 1 to 4 loop
   for j in 1 to 8 loop
      MEM(reg_we(i)) and (MEM(write_register(i)) = ID(read_register(j))
   end loop
end loop
```

The simplest way to resolving data hazards in hardware is to stall the instructions in the pipeline until the hazard is resolved. This is achieved by holding the instruction, where the hazard appeared, out of execution until the WB stage of the instruction that performs the register-write. But stalling the pipeline has very bad effect in the processor performance. The hazard may only occur in only one of the four operations of the instruction word but stalling goes for the whole instruction word. Thus I implemented a complex bypassing hardware in order to provide the correct inputs to the ALU directly from the EX/MEM and MEM/WB registers.

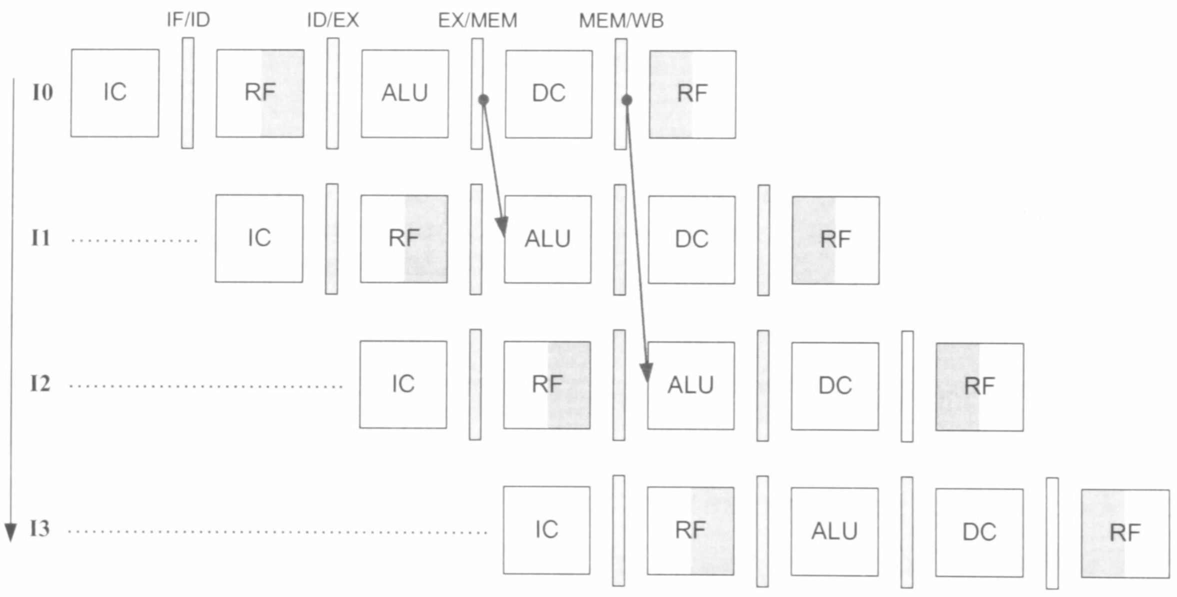


Figure 5. 12 : Bypassing description. a. Values from the EX/MEM pipeline registers are bypassed to the ALU input. b. Values from the MEM/WB pipeline registers are bypassed to the ALU input.

This bypassing hardware is like an interconnection network that connects different functional units together. If the hazard is between ID and EX stages then the required values are forwarded from the EX/MEM pipeline register to the ALU input. Moreover, if the hazard occurs between ID and MEM stages then the required values are forwarded from the MEM/WB pipeline register to the ALU input. These bypassing steps are demonstrated in figure **Figure 5. 12**, and the diagram in figure **Figure 5. 13** demonstrates the bypassing network for each case.

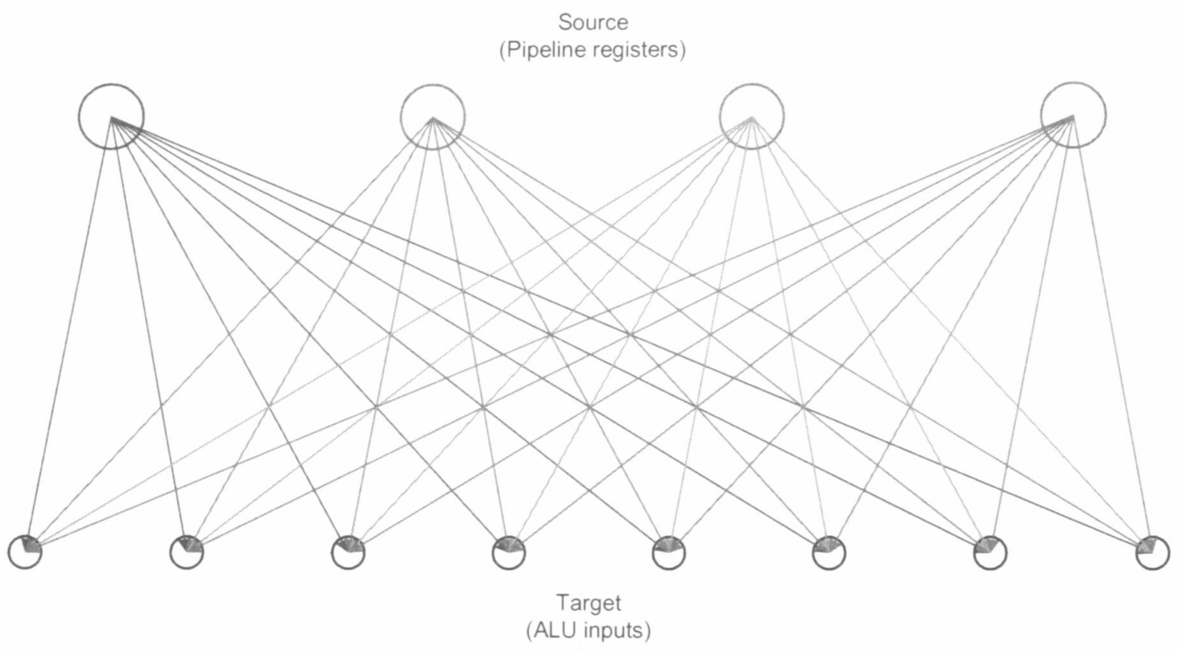


Figure 5. 13 : Bypassing network diagram. Each one of the four output values in the pipeline registers may be bypassed to each one of the eight inputs of the ALU.

However, when an operation tries to read a register following a load operation that writes the same register, bypassing cannot provide correct execution. This is also illustrated in figure **Figure 5. 14**. The data is still being read from memory during the same clock cycle the following operation lays in the EX stage. In this case, the pipeline must still be stalled in order to ensure correct execution. This is achieved by flushing the instruction in the ID stage. To flush this instruction, the data hazard detection unit zeros the control signals in order to prevent the instruction from changing the state of the execution.

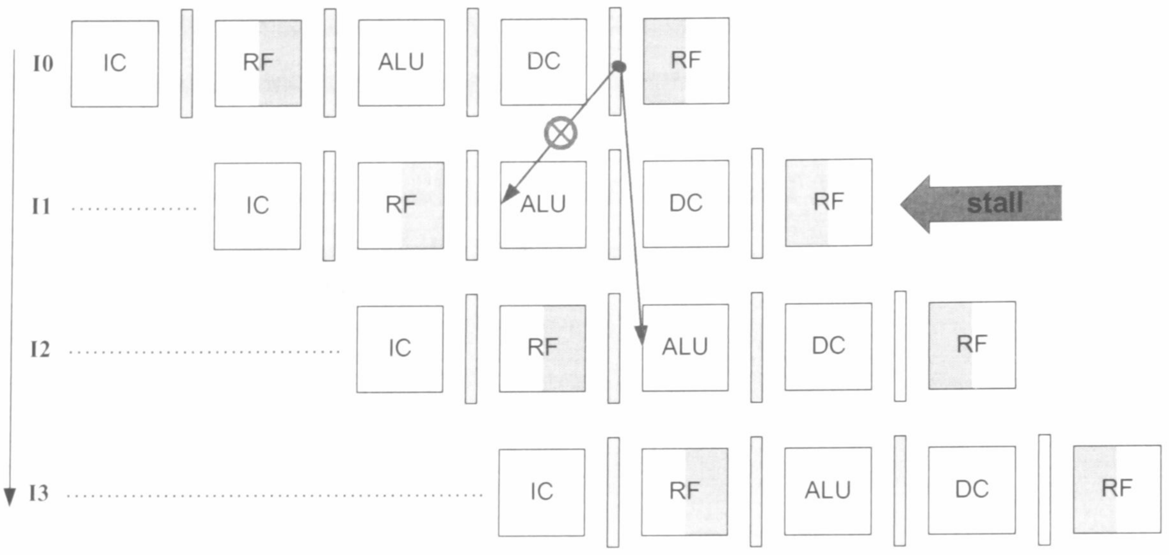


Figure 5. 14 : Pipeline stall example. Unlike I2 to which loaded data can be bypassed, data required for execution of I1 are not yet loaded from the data cache and cannot be bypassed.

5.4.3.2 Control Hazards

Another kind of pipeline hazard involves branches. When there is a branch operation in an instruction word then the instructions that must be fetched at the following clock cycles depend on the result of the branch condition. But this cannot be known until the EX pipeline stage, which means that the decision about whether to branch doesn't occur until that stage. This delay in determining the proper instruction to fetch is called branch hazard.

One solution is to stall the pipeline until the branch is complete. But this would encounter a penalty of two clock cycles for each branch in order to be able to define the new address in the instruction memory. Thus, the approach I took to improve the pipeline behaviour during branches was to allow a conditional execution assuming the branch as not taken. This constitutes a common improvement over stalling, simply by allowing the hardware to continue as if the branch were not executed. If the branch is taken, the instructions that are being fetched and decoded are discarded, and the execution continues at the branch target.

To discard the instructions at IF and ID stages I need to flush these instructions in a way similar to the load data hazard. The only difference is that the instruction in the IF stage must be discarded too. This is achieved using another control line that zeros the IF/ID pipeline register.

Unfortunately, stalling cannot be avoided in case of jump operations. In the ID stage the hardware knows that a *jump* is going to happen and thus it does not allow other instructions to follow the execution by flushing each instruction fetched in the IF stage. The new address is being stored during the EX stage, and then the pipeline restarts the instruction fetch after the inevitable two cycle stalling.

Chapter 6

6 TESTING - TOOLS AND SIMULATION METHODS

In this chapter I will describe the testing environment of the processor. In order to be able to test the processor behaviour in running complete programs, I created an assembly language and also an assembler to produce the machine language code, which is used as data for the instruction memory. Then, the verification of the processor simulation results is achieved by the use of a software simulator also written for this reason.

6.1 The Assembly Language

Creating the assembly language allows easy program writing in contrast to programming directly into the machine language. The language syntax is quite simple and similar to the assembly syntax of DLX. The main difference is that each line corresponds to a long instruction word that consists of at most four separate subinstructions (operations). These subinstructions that are contained in the same instruction are written in the same line and are separated by semi-colons. A sample of a program written in assembly language is given in **Figure 6. 1.**

```
...
       lw r4,r10,12; addi r5,r0,226; ori r6,r0,1
labelX:
       add r7,r4,r5; addi r4,r4,4;   addi r5,r5,-2;    sub r8,r4,r5
       slt r9,r7,r8; sle r20,r4,r5;  bneq r4.r5,labelX
       sw r4,r10,12
...
```

Figure 6. 1 : A sample of an assembly code for the VLIW processor.

In order to define the operation syntax, I will classify them in three types, according to the types into which they are encoded as shown in **Figure 5. 2**. Thus, the syntax of the operations is the following:

- R-type operations

  The operations that belong to this type are:

  AL: ADD, SUB, AND, OR, XOR, SLT, SEQ, SNE, SLE

  and their syntax is:

  OPER Rd, Rs, Rt

  where Rd is the destination register and Rs, Rt are the source registers.

- I-type operations

  The syntax for this type varies according to the operations. One category uses the syntax:

  OPER Rt, Rs, immediate

  and the operations that use this are:

  AL: ADDI, SUBI, ANDI, ORI, XORI, SLTI, SEQI, SNEI, SLEI, SLLI, SRLI.
  LS: LW, SW

  For AL operations Rt represents the destinations register while Rs and the immediate are the source operands. As far as the LS operations is concerned, the memory address is the sum of [Rs + immediate] while the register Rt is either the one to be loaded or stored according to the memory access type. The immediate may be in decimal or even hexadecimal form (ex. 0x404).

  The AL operations NOT, SLL and SRL follow a similar form but with no immediate, where the source register is Rs and the destination is Rd. This syntax is given bellow:

OPER Rt, Rs

Another set of I-type operations is:

CT: BEQ, BNEQ, JR, JALR

The branch operations use the syntax **OPER Rt, Rs, *label*.** The comparison is performed between the two registers and the label defines the branch target. The way labels are written in the assembly is shown in **Figure 6. 1**. Moreover, the syntax of the above jump operations is **OPER Rt,** where Rt contains the program counter target value.

Finally, the way the operation *LHI* is written is shown in the following example:

LHI Rt, immediate,

where Rt is the register of which the upper 16 bits are replaced by the immediate.

- J-type operations

  This type contains the *J* and *JAL* operations, which syntax is:

  J *label* and **JAL *label*.**

  The label defines the program counter target point.

6.2 Bytecode Generation Using the Assembler

In order to translate the assembly language into the machines language code, an assembler was necessary. Thus, I built a program in *PERL* language, which translates a file of assembly language statements into a text file of processor byte code. This file contains the representation of each instruction in '0' and '1', and each instruction word is in a separate line. This output file will have the same name as the input assembly file with the extension "out" added. Assuming that the assembly code routine shown in **Figure 6. 1** is contained in file *paradigmas* the file produced by the assembler will be named *paradigmas.out* and looks like the one shown in **Figure 6. 2**.

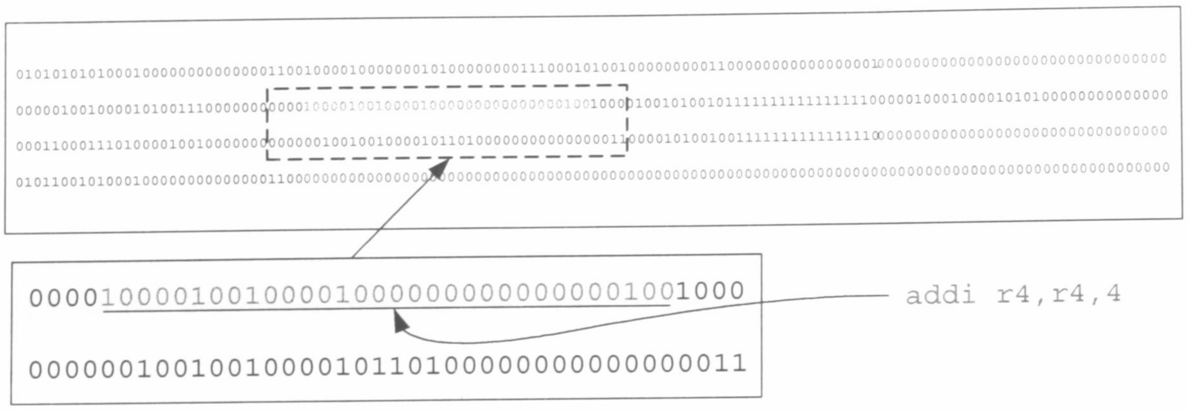


Figure 6. 2 : The output file with the bytecode of the code in figure 6. 1, generated by the assembler. Parts of code in blue colour are NOPs.

The translation process has two major parts. The first step is to scan the assembly language file for labels and to define the memory locations, so the relationship between symbolic names and addresses is known when instructions are translated. During the second step, the assembly file is scanned from the start, this time ignoring the labels, and the subroutine *translate* is called for each instruction (that is for each new line). This routine performs the translation of each assembly statement by combining the numeric equivalents of opcodes, register specifiers and labels into a legal operation. These numeric equivalents are placed in the file *assignments*, which is opened by the assembler to perform these combinations. In case there are less than four operations in a instruction word then for each operation "missing", the assembler adds 32 zeros that correspond to a NOP.

The assembler also performs some error checking such as illegal assembly syntax providing the line of the invalid token and also the invalid token. Moreover, it checks the length of the immediate not to exceed the 16 bits. Finally, the assembly language file may contain comments starting with '# ' that are ignored by the assembler.

6.3 The Simulator

In order to test the processor I had to give some instructions as an input and then watch the processor state after the execution. Using the assembler I can easily write the required instructions in the assembly code, or even a whole program, and then initialising the instruction memory with the byte code generated. However, it is pretty difficult to verify that the post execution state of the processor is correct, especially when running more complicated routines. Thus, I created a program that simulates the processor and provides a better environment for programming and testing.

The software simulator I built is written in C language and runs programs written for the processor. It is named *gpsim* and can read the byte code generated by the assembler and execute the instructions just like the processor. Using the byte code as input for the simulator rather than the assembly language code ensures that possible differences in the results means processor bug and not assembler problem, assuming that the simulator is completely debugged. Moreover, it is simpler to use the byte code in order to perform control transfers, as the assembler has already performed the label handling.

The simulator does not work exactly like the actual processor. It doesn't emulate the processor pipeline and thus it doesn't report anything about stalls or bypassing. The objective of the s/w simulator is to be able to know the processor state after the execution, so I chose to avoid this programming complexity. It works like the single cycle implementation of the CPU. At first it opens the assembler output file that contains the byte code and a file that contains the data of the data memory (memfile.txt), which can be edited before in order to contain the required data. Then, it reads the code line by line executing the operations contained in the instructions word. An array that represents the register file is changed after completing the execution of all the operations of each instruction word, while the array that emulates the data memory is refreshed after the execution of the *store* operation even in the middle of the instruction word.

At the end of the whole program execution the simulator provides three output files. One is the *memfile.txt* that represents the data memory of the CPU and gives the memory final status. The second file is *regfile.txt* that provides the values of the registers of the processor after the program ending. Finally, a log file is also created that describes the execution providing a sequence of the execution steps and the number of non-NOPs for each instruction word allowing watching the intermediate changes in the memory and the register file. Samples of these output files are given in **Figure 6. 3** that correspond to the program shown in **Figure 6. 1**

```
...
r7= r4 + r5 = 326
r4= r4 +i 4 = 104
r5= r5 +i 65534 = 224
r8= r4 - r5 = -126
 instructions: 4

r9= ( r7 < r8 ) = 0
r20= ( r4 <= r5 ) = 1
 instructions: 3

r7= r4 + r5 = 328
r4= r4 +i 4 = 108
r5= r5 +i 65534 = 222
r8= r4 - r5 = -120
 instructions: 4

r9= ( r7 < r8 ) = 0
r20= ( r4 <= r5 ) = 1
 instructions: 3

r7= r4 + r5 = 330
r4= r4 +i 4 = 112
r5= r5 +i 65534 = 220
r8= r4 - r5 = -114
 instructions: 4

r9= ( r7 < r8 ) = 0
r20= ( r4 <= r5 ) = 1
 instructions: 3

r7= r4 + r5 = 332
r4= r4 +i 4 = 116
r5= r5 +i 65534 = 218
r8= r4 - r5 = -108
 instructions: 4
...
```

a. logfile sample

```
Index Value
[0] 5
[1] 120
[2] 0
[3] 0
[4] 0
[5] 0
[6] 0
[7] 0
[8] 0
[9] 0
[10] 0
[11] 0
[12] 100
[13] 0
[14] 0
[15] 0
[16] 0
[17] 0
[18] 0
[19] 0
[20] 226
[21] 0
[22] 0
[23] 0
[24] 0
[25] 0
[26] 0
[27] 0
[28] 0
[29] 0
[30] 0
[31] 0
...
```

b. mem_file sample

```
Index Value
[0] 0
[1] 0
[2] 0
[3] 0
[4] 184
[5] 184
[6] 1
[7] 366
[8] -6
[9] 0
[10] 0
[11] 0
[12] 0
[13] 0
[14] 0
[15] 0
[16] 0
[17] 0
[18] 0
[19] 0
[20] 1
[21] 0
[22] 0
[23] 0
[24] 0
[25] 0
[26] 0
[27] 0
[28] 0
[29] 0
[30] 0
[31] 0
```

c. register file

Figure 6. 3 : Sample of the output files generated by the software simulator, after simulating the code in figure 6. 1. (a) part of the log_file, (b) the first 32 memory cells, (c) the register file.

6.4 Testing

The proper functionality of the processor was tested not only after the completion of the design but also during the implementation. I wrote several small programs in order to ensure the correct processor behaviour around some specific cases. For example I used instructions having data dependencies in order to verify the correctness of the bypassing, I used branch and jump operations to see the program counter change and also the pipeline flushing and stalling capability.

The first step is to write the assembly code and then translate it into byte code using the assembler. Then using the output file that has been generated by the assembler I create a Memory Initialization File (.mif) in order to initialize the instruction memory. A sample of that .mif file is shown in **Figure 6. 4**. Having the program loaded into the instruction memory of the processor I can start the simulation and watch the steps of the execution. Moreover, I use the same output file with the software simulator in order to figure out possible processor errors.

```
-- Instruction memory initialization file

WIDTH = 128;
DEPTH = 256;

ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;

CONTENT BEGIN
        0       :       01010101010001000000000000000011001000010000001...;
        1       :       00000100100001010011100000000000010000100100001...;
        2       :       00011000111010000100100000000000000100100100001...;
        3       :       01011001010001000000000000000011000000000000000...;
END;
```

Figure 6. 4 : A sample of the Memory Initialization File, generated for loading of the code in Figure 6. 1

The processor simulation is performed using the Altera Max+plus II functional simulator. The instruction and data memory can easily be read and changed. For that reason I created two *mif* files that I can modify using a text editor in order to load a program or even to preload the some values in the data memory.

The whole testing procedure is demonstrated in the following subsection where I use an assembly program that illustrates a recursive factorial algorithm.

6.4.1 A Complete Routine Example

In order to demonstrate the processor evaluation procedure I wrote a recursive factorial algorithm in VLIW code. This program is shown in **Figure 6. 5**.

It is a simple program but it covers many special conditions in execution that can denote the proper functionality of the processor, such as jump, jump and link, branches and conditional execution, memory access, all of which may cause the processor to stalling and data bypassing.

```
lw r1,r0,0x100; ori r30,r0,100; ori r2,r0,1;    jal start
sw r3,r0,41;    j done
start:
                slt r29,r2,r1;  addi r2,r2,1
factorial:
                sw r31,r30,0;   slt r29,r2,r1;  beq r29,r0,isz
                sw r1,r30,1;    addi r30,r30,2; subi r1,r1,1;   jal factorial
                lw r4,r30,-1;   ori r5,r0,0;    addi r11,r0,16; jal multiply
end:
                lw r31,r30,-2;  subi r30,r30,2
                jr r31
isz:
                sw r1,r30,1;    addi r30,r30,2; ori r3,r0,1;    j end
multiply:
                andi r10,r4,1;  subi r11,r11,1
                beq r10,r0,skip
                add r5,r3,r5;   sll r3,r3;      srl r4,r4;      bneq r11,r0,multiply
                or r3,r5,r0;    jr r31
skip:
                sll r3,r3;      srl r4,r4;      bneq r11,r0,multiply
                or r3,r5,r0;    jr r31
done:
```

Figure 6. 5 : The example program that computes the factorial of a number recursively, in VLIW code.

The first step is bytecode generation using the assembler. Then I use the generated bytecode as input for memory initialization in the Compiler and Simulator of Max+Plus II. Moreover, I used the same generated bytecode file as input for the software simulator to evaluate the execution results of the functional simulation of the processor. I run the program for several factorial computations and the results prove the proper execution of the processor, as the final state of the register file and the data memory were the same and are given in **Figure 6. 6.**

Finally, both the software and the functional simulator provided useful information about the execution of this program, such as the total number of the long instruction words and the operations executed and also the clock cycles required for the computation of the factorial of number 12. Moreover, I wrote and simulated the same program in simple sequential mode and compared the simulation results with those of the VLIW code. These results are provided in **Table 6. 1.**

```
Index Value    Index Value
[1] 0          [111] 7
[2] 0          [112] 5
.              [113] 6
.              [114] 5
.              [115] 5
[38] 0         [116] 5
[39] 0         [117] 4
[40] 12        [118] 5
[41]           [119] 3
479001600      [120] 5
[42] 0         [121] 2
[43] 0         [122] 5
.              [123] 1
.              [124] 5
.              [125] 1
[98] 0         [126] 0
[99] 0         [127] 0
[100] 1        .
[101] 12       .
[102] 5        .
[103] 11       [254] 0
[104] 5        [255] 0
[105] 10       [256] 12
[106] 5        [257] 0
[107] 9        [258] 0
[108] 5        .
[109] 8        .
[110] 5        .
```

b. final memory state

```
Index Value    Index Value

[0] 0          [16] 0
[1] 0          [17] 0
[2] 0          [18] 0
[3] 0          [19] 0
[4] 184        [20] 1
[5] 184        [21] 0
[6] 1          [22] 0
[7] 366        [23] 0
[8] -6         [24] 0
[9] 0          [25] 0
[10] 0         [26] 0
[11] 0         [27] 0
[12] 0         [28] 0
[13] 0         [29] 0
[14] 0         [30] 0
[15] 0         [31] 0
```

c. final register file status

Figure 6. 6 : The post execution state of the register file and the memory. The values given by the software simulator are identical to the ones derived from the functional simulation of the processor.

	Sequential Code	VLIW code	Relative difference
Clock	1972	1304	33.9% less
Long	▨	601	▨
Instructions	1269	1271	Almost same
CPI	1.554	1.026	34% better

Table 6. 1 : Execution results of both the sequential and VLIW code, and a comparison between them.

As it is described in the above table, the processor needs 33.9% less clock cycles to execute the VLIW form of the program than the simple sequential form, although the total instructions executed are almost the same. This results to a better CPI of 34% for the VLIW code, which is almost 1 but may be improved even more if the code was properly scheduled for VLIW.

Chapter 7

7 CONCLUSIONS

Concurrency is the key element to achieve high performance computing and a way to achieve this is through Instruction Level Parallelism. In this thesis I firstly performed a complete presentation of the two basic approaches that have been taken in order to achieve Instruction Level Parallelism in CPUs; The Superscalar and the VLIW processor architecture. I also described the difference between these two architectures figuring out the main advantages and disadvantages of both the two approaches. However, I analyzed more extensively most of the organization and architectural issues on VLIW architecture.

After having discussed many issues concerning VLIWs, I proceeded in building a simple VLIW processor in VHDL, following basic principles that characterise this processor type. I designed the Instruction Set Architecture for the processor and also implemented the hardware logic design based on this ISA. Furthermore, I implemented an assembly language translator necessary to easily produce byte-code used to load the instruction memory, and a software simulator that simulates the functional output of the CPU, in order to evaluate the proper processor program outputs.

7.1 Thesis Conclusions

VLIW microprocessors and superscalar implementations of traditional instruction sets share some characteristics. They both have multiple execution units and thus the ability to execute multiple operations simultaneously. The techniques used to achieve high performance, however, are very different because the parallelism is explicit in VLIW instructions, but must be discovered by the hardware at run time by superscalar processors.

In VLIW processors, with software being responsible for analyzing dependencies and creating execution schedule, the size of the instruction window that can be examined for parallelism is much larger than what a superscalar processor can do in hardware. Thus, a VLIW can expose more parallelism. Moreover, since the control logic in a VLIW processor does not have to do any dependency checking, VLIW hardware can be much simpler to implement than superscalar hardware that is quite complex.

This lack of complexity in a VLIW processor is also ascertained on my processor design. I can safely say that the control logic was one of the simplest parts to implement, while the four-functional-unit datapath was quite more complex. Actually, the main thing that made this implementation being a quite hard task was the size of the design with a large number of components, which made the project hard to be compiled and simulated.

On the other hand, VLIW processors suffer from some other issues that superscalars overpass. One issue is the poor code compatibility between different instances of the same architecture, such as the number and type of the functional units and the instruction issue width. Moreover, too many NOP insertions in the instructions results in large code size. Therefore, I could imagine that the key in designing high performance processors lays somewhere between these two architectures. Probably a superscalar like architecture with improved support for static parallelization, or even VLIW architecture but with improved support for dynamic parallelization.

7.2 The Future of VLIW Processors

In the 1980s, a few small companies attempted to commercialize VLIW architectures in the general-purpose computer market, such as Multiflow and Cydrome. Unfortunately, they were ultimately unsuccessful. VLIW compiler technology has made major advances during the last decade. However, most of the compiler techniques developed for VLIW are equally applicable to super scalar processors as well. Stream and media processing applications are typically very regular with predictable branch behavior and large amounts of ILP. They lend themselves easily to VLIW style execution. The increasing demand for multimedia applications will continue to encourage development of VLIW technology. However, in the short term, super scalar processors will probably dominate in the role of general-purpose processors. Increasing wire delays in deep sub micron processes will ultimately force super scalar processors to use simpler and more scalable control structures and seek more help from software. It is reasonable to assume that in the long run, much of the VLIW technology and design philosophy will be adopted into general-purpose computer architectures

In recent years, some VLIW and VLIW-based processors have enjoyed moderate commercial success. The most commercially known are the Intel Itanium and the Transmeta Crusoe processor, which I describe in the next paragraphs.

The Intel Itanium Processor

The Itanium processor is Intel's first implementation of the IA-64 ISA. IA-64 is an ISA for the EPIC (Explicitly Parallel Instruction Computing) style of VLIW developed jointly by Intel and HP. It is a 64 bit, six-issue VLIW processor with four integer units, four multimedia units, two load/store units, two extended precision floating point units and two single precision floating point units. This processor running at 800 MHz on a 0.18 micron process has a 10 stage deep pipeline.

Itanium has 128 general-purpose registers and the same number of floating point registers. An operation encoding hence uses 7 bits each to specify the two source operands and another 7 bits to specify the destination operand. The type of operation itself is encoded using 14 bits. Many operations also use a predicate argument that takes up another 6 bits since there are 64 predicate registers. The predicate registers store a bit, depending on whose truth value the processor decides to either execute the concerned operation or skip its execution (execute a nop). This accounts for a total of 41 bits to specify an operation and its encoding is shown in **Figure 7. 1 (b)**.

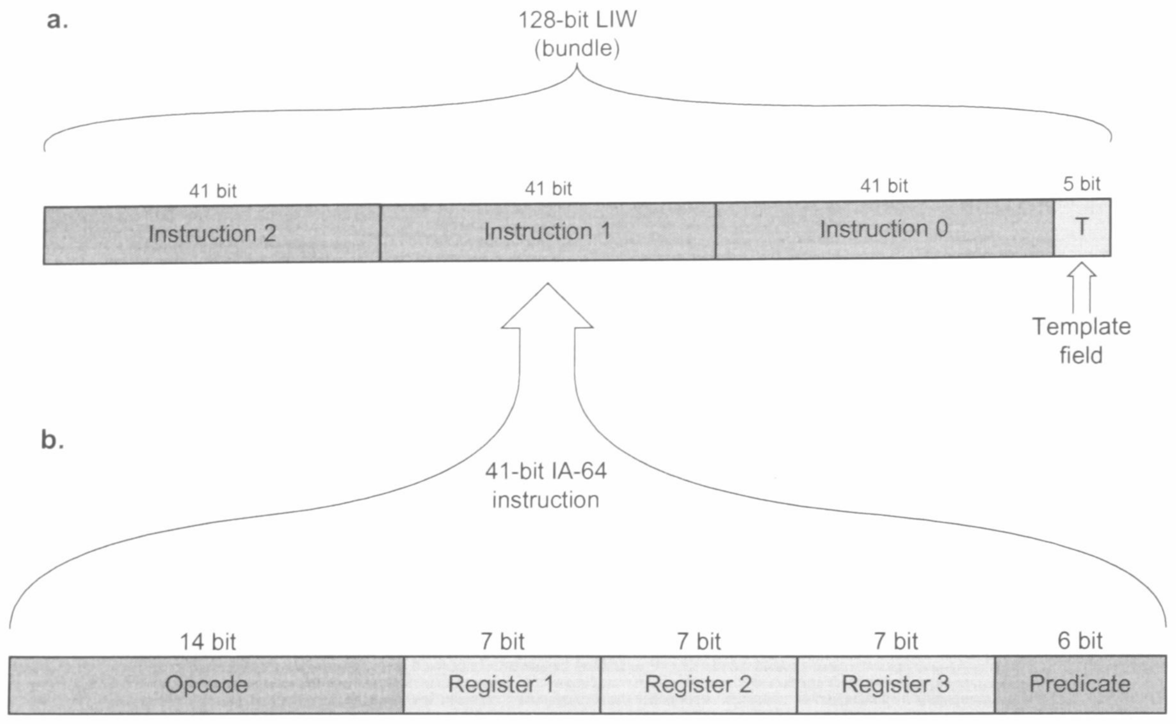


Figure 7. 1 : IA-64 instruction format. (a) IA-64 VLIW instruction encoding, (b) IA-64 operation encoding.

Each VLIW instruction is called *bundle* and consists of 128 bits. This means that each VLIW instruction can accommodate three 41-bit operations. The five bits left are used for the template, which assists in decoding and routing the instructions and also the location of stops that mark the end of a group of instructions that can execute in parallel. The 128-bit bundle encoding is shown in **Figure 7. 1 (a)**.

The main architectural features supported by the Itanium processor are:

- *Predicated execution:* It reduces the branch penalty by eliminating branches using predicated execution via the compiler technique known as *if-conversion*. Predicated execution conditionally executes operations based on a Boolean-valued input (predicate) associated with the basic block containing the operation.

- *Control speculation:* Another feature to increase operation mobility across branches. To do control speculation, the compiler moves an operation before its conditional branch. The operation then carries a flag that indicates that it needs speculative operation code.

- *Data speculation:* If the compiler cannot disambiguate between the addresses of a store and a later load, it can issue an *advance load* ahead of the store, which is also called *speculative load.* Moreover, it schedules a *data-verifying load* called *speculative check*, after potentially aliasing stores and uses hardware to detect whether an unlikely alias has occurred.

- *Software pipelining:* The Itanium register mechanism is somehow complex in order to implement software pipelining support. The general purpose registers 0 to 31 are fixed, but registers 32 through 127 can be renamed under program control to support a register stack or to do modulo scheduling for loops. In case this is used for software pipelining support it is called *register rotation.* Like general purpose registers, predicated registers 0 through 15 are fixed and 16 through 63 can be made to rotate in unison with general purpose ones. Finally, floating point registers also support register rotation.

- *IA-32 compatibility:* The Intel Itanium processor supports 32-bit binary compatibility in hardware.

Itanium was built to improve performance, thus it includes several features that are not found in traditional VLIW architecture. The Intel Itanium processor is probably the most complex VLIW ever designed.

The Transmeta Crusoe Processor

Traditionally, VLIW processors have been designed to maximize both ILP and performance. The designers of the Crusoe on the other hand designed a VLIW processor with moderate performance compared to current processors, but with low power consumption. This would enable the use of these processors in mobile systems like laptops allowing many hours of operation times between recharges. Moreover, it is able to efficiently emulate the ISA of other processors, particularly the 80x86 even though the architecture of Crusoe nowhere resembles that of an 80x86 processor.

Crusoe is a simple VLIW architecture. The long instructions are either 64 or 128 bits long. A 128-bit instruction word is called a *molecule* in Transmeta parlance and encodes four operations called *atoms*. The molecule format directly determines how operations get routed to functional units. It has two integer units, a floating-point unit, a load/store unit and a branch unit. It has 64 general purpose registers and supports strictly in-order issue. Instead of using predication, Crusoe uses condition flags, which are identical to those of the x86 architecture for ease of emulation.

To achieve binary compatibility with an x86 programs, the Crusoe processor relies on a software technique called dynamic binary translation. An x86 program is dynamically translated to execute on the Crusoe VLIW processor. This translation scheme has been dubbed *code morphing* by Transmeta. In this scheme, the software, also called the virtual machine manager since it presents an x86 virtual machine to an x86 program, uses a combination of interpretation and translation to speed up program execution.

REFERENCES

1. David A. Patterson, John L. Hennesy, *Computer Organization and Design: the hardware/software interface*, Morgan Kaufmann Publishers, 1994.

2. S. Hacker, *Static Superscalar Design: A new architecture for the TigerSHARK DSP Processor*, Analog Devices GmbH.

3. A. Klaiber, *The technology behind the Crusoe processor*, Transmeta Corporation White Paper, 2000.

4. Joseph A. Fisher, *Global code generation for instruction-level parallelism: Trace Scheduling-2*, Tech. Rep. HPL-93-43, Hewlett-Packard Laboratories, June 1993.

5. Joseph A. Fischer, *Very Long Instruction Word Architectures and the ELI-512*, Proceedings of the 10'th Symposium on Computer Architectures, pp. 140-150, IEEE, June, 1983.

6. P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter and W. W. Hwu, *Impact: an architectural framework for multiple-instruction-issue processors.* Proc. 18th Annual International Symposium on Computer Architecture (Toronto, Canada, May 1991), pages 266-275.

7. D. E. Hudak and S. G. Abraham, *Compiling Parallel Loops for High Performance Computers -- Partitioning, Data Assignment and Remapping*, Kluwer Academic Pub., Boston, MA, 1993.

8. R. Gupta, S. Pande, K. Psarris, and V. Sarkar, *Compilation Techniques for Parallel Systems*, Parallel Computing journal, North Holland, Vol. 25, No. 13-14, pages 1741-1783, December 1999.

9. K. Ebcioglu, *Some Design Ideas for a VLIW Architecture for Sequential-Natured Software*, in Parallel Processing (Proc. IFIP WG 10.3 Working Conference on Parallel Processing, Pisa, Italy), pages 3--21. North Holland, 1988.

10. W-m. Hwu, *Technology Outlook: Introduction to Predicated Execution*, IEEE Computer, Vol. 31, No. 1, pages 49-50, January 1998.

11. C.Fu, M. Jennings, S. Y. Larin, T. M. Conte, *Value Speculation Scheduling for High Performance Processors*, International Conference on Architectural Support for Programming Languages and Operating Systems, October 1998.

12. B.R. Rau, *Dynamic scheduling techniques for VLIW processors*, Technical Report HPL-93-52, Computer Research Center, Hewlett-Packard Company, June 1993.

13. B. R. Rau, *Dynamically scheduled VLIW processors*, in Proc. 26th Ann. International Symposium on Microarchitecture, (Austin, TX), pp. 80--90, Dec. 1993.

14. M. Franklin and M. Smotherman, *A fill-unit approach to multiple-instruction issue*, in Proceedings of 27th Annual International Symposium on Microarchitecture (MICRO-27), pp. 162-171, December 1994.

15. S. Melvin, M. Shebanow, and Y. Patt, *Hardware support for large atomic units in dynamically scheduled machines*, in Proceedings of 21st Annual International Symposium on Microarchitecture (MICRO-21), pp. 60-66, December 1988.

16. S. W. Keckler and W. J. Dally, *Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism*, in Proc. 19th Ann. Int'l Symp. Computer Architecture, (Gold Coast, Australia), May 1992.

17. G. Prasadh and C. Wu, *A Benchmark Evaluation of a Multithreaded RISC Processor Architecture*, in Proc. of Int'l Conf. on Parallel Processing, pp. I84--I91, Aug. 1991.

18. A. Wolfe and J. P. Shen, *A Variable Instruction Stream Extension to the VLIW Architecture*, in Proc. 4th Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems, ACM Press, pp. 2--14, Apr. 1991.