UTDSP: A VLIW Programmable DSP Processor

by

Sean Hsien-en Peng

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

© Copyright by Sean Hsien-en Peng 1999

UTDSP: A VLIW Programmable DSP Processor

Sean Hsien-en Peng

Master of Applied Science, 1999

Graduate Department of Electrical and Computer Engineering

University of Toronto

Abstract

VLIW architectures are well-suited for implementing application-specific programmable processors because of their great scalability and modularity. VLIW architectures take advantage of not only temporal parallelism found in RISC architectures but also spatial parallelism by using multiple functional units. However, the large instruction storage and bandwidth requirements have prevented VLIW architectures from being used in cost-sensitive systems.

This thesis describes a VLIW DSP processor called UTDSP, which incorporates a novel and flexible instruction packing and fetching mechanism to reduce the code size and bandwidth problems plaguing other VLIW architectures. With this scheme it is possible to actually achieve some code compression while attaining significant performance speedup over a traditional architecture. The UTDSP is flexible in that additional functional units with application-specific instructions can be easily added when required for performance with little impact on its compiler.

The VLSI design and implementation of the UTDSP is presented. This implementation, consisting of five pipeline stages, is capable of executing seven instructions per cycle and provides zero-overhead hardware loops that are nestable and interruptable. A GUI-based assembly debugger and architecture simulator were implemented. The UTDSP adopts a synthesis-based design methodology and a novel hierarchical CAD flow that can significantly reduce its area.

To Hsiao-ching, Hsien-yuan, and Yu-liang

Acknowledgements

This thesis could not have been completed without the help of many individuals and organizations, to whom I am extremely grateful.

I want to thank my supervisor, Professor Paul Chow, for his precious suggestions, friendly advice, and considerable amount of time that was spent to reading the many drafts of my thesis. Most of all, however, for his patience and encouragement.

I would like to thank Natural Sciences and Engineering Research Council for their funding in the past two years. I could not have had the chance to carry out my research at U of T without the financial support from NSERC.

I offer thanks to Louis Zhang, Jianghong Hu, Brent Beacham, Nirmal Sohi, Deshanand Singh, Warren Gross, Andy Ye, and Tor Aamodt for all the good times and numerous technical discussions from which I have learned much. Thanks to Dr. Mazen Saghir for defining a clear framework in this research project and for his technical help. Thanks also to An Wei, Jin Heng, Jianhui, Michael, Franklin, Edward, and Shuo for all the fun times in the lab.

I am extremely grateful to my family for all their support and ever-lasting love. Last, but certainly not least, I would like to thank my wife Hsien-yuan for her love, understanding, and sacrifice of countless weekends and holidays in the past years. This thesis involved a lot of hard work and months struggling with CAD tools. I could not have done it without her support.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Digital signal processors (DSPs) are specialized microprocessors designed to execute the computationally-intensive operations commonly found in the inner loops of digital signal processing algorithms. Having been used extensively in embedded systems, DSPs are required to offer high performance while reducing cost. To fulfill this goal, traditional DSPs use tightly-encoded instruction sets to reduce instruction memory requirements, and hence cost. Using tightly-encoded instruction sets reduces not only storage requirements but also instruction memory bandwidth, which is a major concern when off-chip instruction memory needs to be used.

However, tightly-encoded instruction-set architectures (ISAs) are not well-suited for high-level languages (HLL) compilers to exploit parallelism because most of the instructions are accumulator based, limiting the number of registers that can be specified in an operation. As a result, DSP compilers generate relatively poor code compared with their counterparts for general-purpose microprocessors. Therefore, more compiler-friendly DSP architectures that combine high performance with low cost are definitely needed. One alternative to the tightly-encoded instruction architectures is the very long instruction word (VLIW) architecture.

VLIW architectures offer high performance by using multiple, independent functional units, enabling multiple instruction issue while reducing cost by eliminating dynamic scheduling logic. Unlike superscalar processors, where data hazards are handled using dynamic scheduling, VLIW architectures rely on compilers to create a package of instructions that can be simultaneously issued. VLIW architectures are very well-suited for exploiting a high level of parallelism because they are easy targets for HLL compilers to generate efficient code.

However, VLIW architectures have several limitations that are not favorable in cost-sensitive DSP processors. First, instruction-memory size is increased substantially due to the unused encoding slots in long instructions and the extra instructions created using loop unrolling to exploit parallelism. Second, fetching long instructions from off-chip instruction memory requires a high bandwidth, which can be a severe problem when pin-count and packaging options are major constraints.

1.2 Objective and Contributions

The objective of this thesis is to design and implement a VLIW programmable DSP processor — UTDSP. The UTDSP eliminates the limitations mentioned above by incorporating a two-level instruction fetching and packing mechanism. The VLSI implementation of the UTDSP, along with associated software development tools, is presented in this thesis. The following details three major contributions of this thesis.

1.2.1 Design and Implementation of a Long-instruction Packer

The UTDSP instruction packer was implemented based on a two-level instruction fetching mechanism proposed by Mazen Saghir [2]. The UTDSP packer not only packs long instructions to reduce storage requirements but also serves as an assembler. Benchmark results indicate that the UTDSP instruction packer outperforms the new TI VelociTI packing algorithm, while solving the fetching bandwidth problems mentioned previously.

1.2.2 Design and Implementation of Architecture Simulator and GUI-based Assembly Debugger

The UTDSP architecture was designed using an application-driven design methodology where architectures are designed according to the performance and cost requirements of their target applications. Being written in a high-level language, the architecture simulator, which also serves as a behavioural model of the UTDSP, can be easily modified to experiment with design trade-offs, enabling the application-driven design methodology. An GUI-based assembly debugger was also implemented to allow programmers to perform interactive debugging features such as memory probing and breakpoint tracing.

1.2.3 Design and VLSI Implementation of the UTDSP

The UTDSP, which has five pipeline stages, was implemented using a synthesis-based design methodology. The UTDSP provides not only a set of highly orthogonal, RISC-like instructions but also DSP-specific features such as zero-overhead hardware loops. The zero-overhead hardware loops can be nested up to five levels. Also, interrupts and branches are allowed in the inner loop. A novel hierarchical CAD flow that significantly reduces the resulting area and interconnect delay of the UTDSP was defined in this thesis.

1.3 Thesis Organization

This thesis is divided into six chapters. Chapter 2 provides the reader with background information on the UTDSP, focusing on a VLIW model architecture and its compiler system. Chapter 3 introduces the design and implementation of the UTDSP packer. Benchmark comparison between the UTDSP packer and TI's VelociTI memory packer will be analyzed in this chapter. Chapter 4 describes the design and implementation of the architecture simulator and GUI-based assembly debugger. Chapter 5 presents the design and VLSI implementation of the UTDSP. Related CAD issues will be illustrated in this chapter. Chapter 6 concludes this thesis and offers recommendations for future work.

Chapter 2

Background

The rapid growth in the consumer electronics market has increased the demand for high-performance, low-cost processors for use in embedded systems. Although off-the-shelf DSP processors can be used to meet these demands, application-specific programmable processors (ASPPs) — processors that are designed for specific applications — are more desirable for use in cost-sensitive systems because their architectures and instruction set can be tuned for their specific performance and cost requirements. The UTDSP processor, an ASPP aimed at embedded DSP applications, incorporates an application-driven design methodology, where architectures are designed according to the requirements of the target applications.

This chapter provides an overview to the UTDSP project and describes background information upon which this thesis is built. Section 2.1 explains the application-driven design methodology and how it is used to generate architectures that are easy targets for high-level language (HLL) compilers. Section 2.2 introduces a flexible model architecture that can be easily modified according to the performance and cost requirements of the target applications. Section 2.3 describes an optimizing C compiler and its role in tuning the model architecture. Section 2.4 describes two commercially available DSP processors that have VLIW architectures. Section 2.5 summarizes this chapter.

2.1 Application-Driven Design Methodology

Conventional embedded DSP processors are designed without fully appreciating the features and limitations of their HLL compilers. Moreover, many DSP processors are developed using a methodology where compiler construction starts after functional silicon is obtained [3][4][5]. This usually results in a design that is a difficult target for HLL compilers; therefore, the compilers nei-

ther take maximum advantage of the hardware resources in the architecture nor generate efficient assembly code compared with hand-crafted versions.

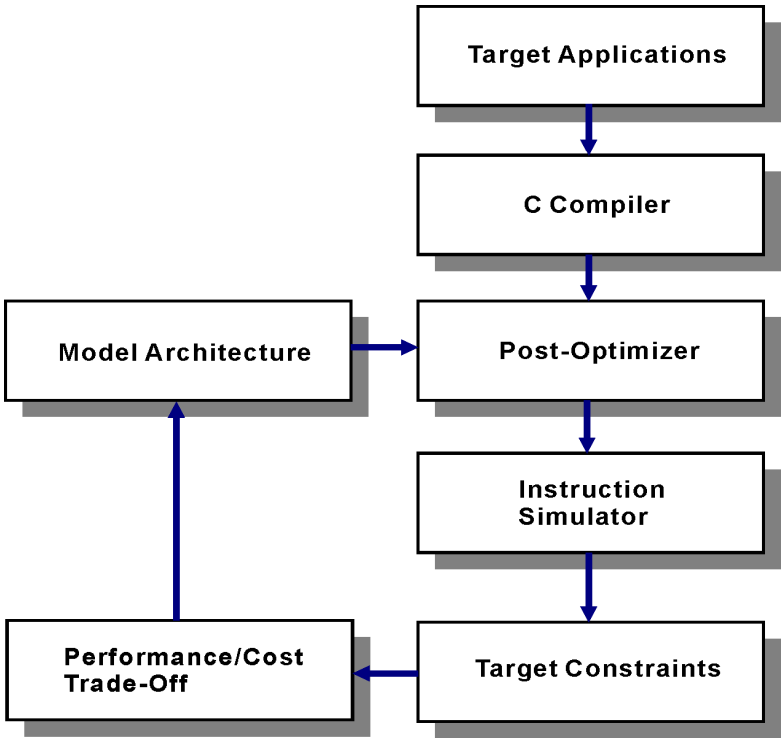


Figure 2.1: Application-driven design methodology

In contrast, the UTDSP project uses an application-driven design methodology [1] where architectures are designed according to the performance and cost requirements of their target applications. Figure 2.1 shows the flow used in the application-driven design methodology. The design starts with a flexible model architecture as a template; a suite of benchmarks is used to evaluate the performance of the model architecture. The model architecture is iteratively modified until it meets the performance and cost requirements of the target applications. The key component in this design methodology is the flexible model architecture, which is easy to configure and is able to exploit parallelism. A flexible, RISC-like instruction set is also provided to make the model architecture an easy target for HLL compilers.

As the complexity of DSP applications increases, writing a DSP application entirely in assembly language is no longer feasible although kernels and inner loop code are still often hand-optimized to achieve a better throughput. Therefore, the benchmark suite was developed in the C

programming language and a C compiler is used to translate the target applications into the machine operations that can be executed by the model architecture. The C compiler generates sequential code and performs register allocation based on the instruction set and the number of registers defined for the model architecture. A post-optimizer is then used to exploit the DSP-specific features of the model architecture. The post-optimizer also exploits parallelism in the sequential code and creates executable code that runs on the model architecture. The executable code is then simulated using an instruction-set simulator to obtain the cost and performance information of the model architecture.

When the performance requirements are not fulfilled, the architecture, compiler, and post-optimizer can be modified to exploit more parallelism. Similarly, when the cost requirements are not fulfilled, the hardware components that are under-utilized can be removed to reduce system cost. This process is repeated until the model architecture meets both the cost and the performance requirements of the target applications.

The benchmark suite used in this study consists of six kernels and ten applications. Table 2.1 shows the kernel benchmarks, which consists of simple algorithms commonly used in DSP applications. The kernels usually constitute the inner loop of DSP applications; therefore the effectiveness of exploiting parallelism in kernels dominates the overall performance. In other words, the compiler must generate efficient code for kernels to maximize the utilization of the hardware resources in the model architecture. Table 2.2 shows the DSP application benchmarks, which are commonly used in embedded systems. Using the suite of benchmarks with the application-driven design methodology thus makes the resulting model architecture an ideal design for embedded DSP processors.

	Kernels	Description
k1 k2	fft_1024 fft_256	Radix-2, in-place, decimation-in-time fast Fourier transform
k3 k4	fir_256_64 fir_32_1	Finite impulse response (FIR) filter
k5 k6	iir_4_64 iir_1_1	Infinite impulse response (IIR) filter

Table 2.1: DSP kernel benchmarks

	Kernels	Description
k7 k8	latnrm_32_64 latnrm_8_1	Normalized lattice filter
k9 k10	lmsfir_32_64 lmsfir_8_1	Least-mean-squared (LMS) adaptive FIR filter
k11 k12	mult_10_10 mult_4_4	Matrix Multiplication

Table 2.1: DSP kernel benchmarks

	Applications	Description
a1 a2	G721_A G721_B	Two implementations of the ITU G.721 ADPCM speech encoder
a3	V32.modem	V.32 modem encoder/decoder
a4	adpcm	Adaptive differential pulse-coded modulation speech encoder
a5	compress	Image compression using discrete cosine transform (DCT)
a6	edge_detect	Edge detection using 2D convolution and Sobel operators
a7	histogram	Image enhancement using histogram equalization
a8	lpc	Linear predictive coding speech encoder
a9	spectral	Spectral analysis using periodogram averaging
a10	trellis	Trellis decoder

Table 2.2: DSP application benchmarks

2.2 The Model Architecture

Flexibility and compiler programmability are two important requirements for the model architecture used in the application-driven design methodology. Flexibility enables the model architecture to be easily configured to meet the performance and cost constraints of an application, while compiler programmability requires the architecture to be an easy target for HLL compilers. A model architecture that is based on a very long instruction word (VLIW) architecture was chosen to meet both requirements [2].

A VLIW architecture consists of multiple functional units each of which can execute independent instructions simultaneously. Unlike CISC instructions, which are vertically encoded, VLIW

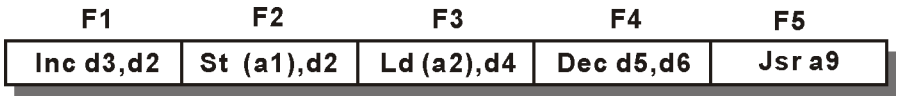


Figure 2.2: Example long instruction for a VLIW with 5 functional units

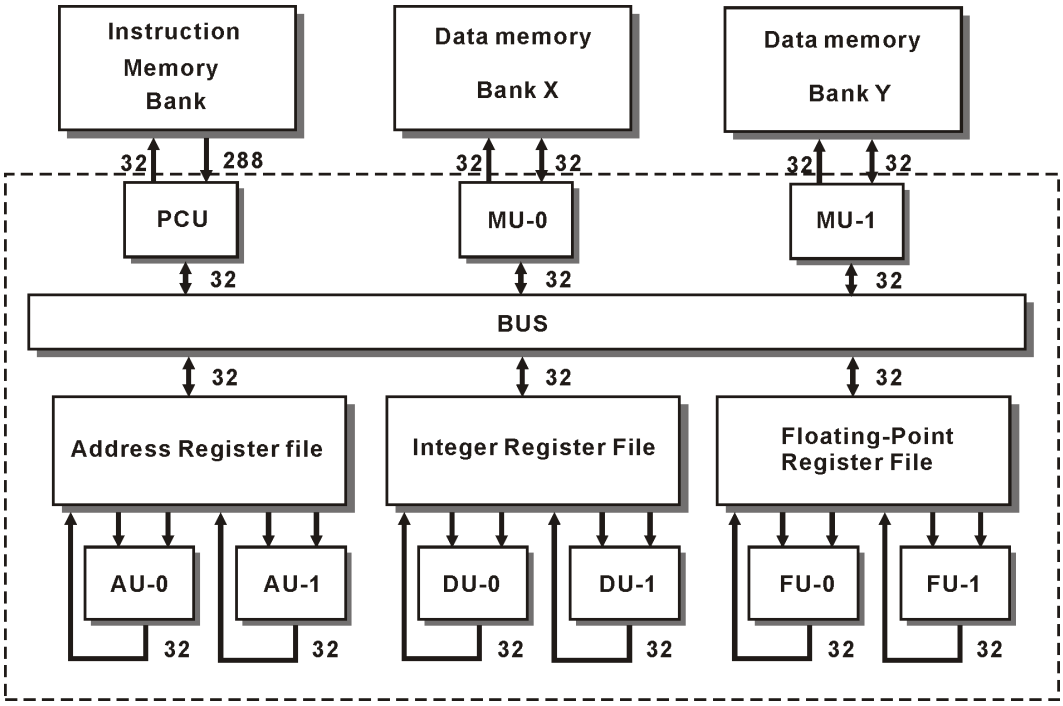


Figure 2.3: VLIW Model Architecture

long instructions are horizontally encoded. Each long instruction contains N fields, where N is the number of operations that can be executed concurrently; each field controls a corresponding functional unit. Figure 2.2 shows an example VLIW long instruction. An optimizing compiler is used to exploit parallelism and schedule parallel operations into the fields of a long instruction. Moreover, using a highly orthogonal, RISC-like instruction set helps the compiler generate efficient code for the target architecture. The VLIW architecture is flexible in that additional functional units can be easily added when required for performance with little impact on the compiler.

Figure 2.3 shows the VLIW model architecture used for the UTDSP. The model contains nine functional units: two memory units (MU0 and MU1), two address units (AU0 and AU1), two integer units (DU0 and DU1), two floating-point units (FU0 and FU1), and one control unit

(PCU). MU0 and MU1 execute memory operations. Each memory unit connects to a single-ported, data-memory bank. AU0 and AU1 execute address operations. DU0 and DU1 execute integer operations. FU0 and FU1 execute floating-point operations. PCU executes control operations. Because the model has nine functional units, up to nine parallel operations can be specified in a long instruction and execute concurrently. Unlike superscalar architectures, where instruction scheduling is handled dynamically in hardware, VLIWs adopt static scheduling, which requires compilers to resolve data hazards. Eliminating the dynamic scheduling logic gives VLIWs a faster execution speed and a smaller silicon area. Because the long instructions consisting of RISC-like operations can still fit in a pipeline scheme, VLIWs can exploit not only spatial parallelism using multiple functional units, but also temporal parallelism by introducing the pipeline scheme.

A Harvard memory architecture, where instruction memory is separated from data memory, is used to increase memory bandwidth and enable the concurrent fetching of instructions and data. Because the model architecture is a load-store design — all operands must be first loaded from data memory to register files through the two MUs, dual data-memory banks are introduced to reduce the possibility of starving for operands in DSP kernels. To take advantage of the dual data-memory banks, the compiler must distribute program data among them. More details about exploiting dual data-memory banks are given in [2][42].

The model has three register files to store address, integer, and floating-point operands, respectively. Specifically, the integer functional units only operate on registers in the integer register file; the address units only operate on registers in the address register file. Similarly, the floating-point units can only access the registers in the floating-point register file. All register files are connected to the memory units so that data can be loaded from data-memory banks to any one of the register files. The program-control unit is also connected to all register files to allow data transfers between them.

However, one major drawback that prevents VLIWs from being used in cost-sensitive systems is their high instruction bandwidth. As shown in Figure 2.3, the model architecture needs a 288-bit bus for instruction fetching. Because a long instruction must be fetched from memory on every clock cycle, the performance will be severely degraded when off-chip instruction memory is used and the number of available pins is not enough for implementing a full fetch bus. Another drawback is the large instruction storage when compilers cannot exploit enough parallelism to sched-

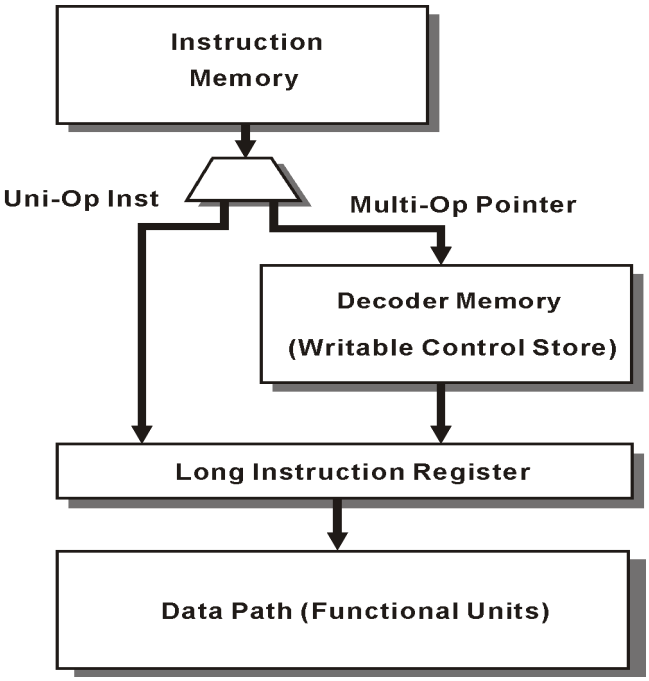


Figure 2.4: Basic Structure of Instruction Decoder

ule operations into long instructions. Storing the long instructions that have many unused fields in their original format is therefore very wasteful, and results in increased system costs.

To solve the bandwidth and storage problems mentioned above, Mazen Saghir proposed a long instruction fetching and packing mechanism [2] based on writable control stores [6][7]. The writable control stores can be found in a microprogrammed computer where an instruction contains a pointer to horizontal microcode stored in the control store. Similarly, long instructions can be stored in the control store and their pointers are stored in the instruction memory. When a pointer is fetched from the instruction memory, it is used to fetch its associated long instruction from the control store (decoder memory). Figure 2.4 shows a block diagram of the instruction and decoder memory. The instruction memory stores single operations or pointers to long instructions. The operations stored in the instruction memory are called *uni-op* instructions — long instructions that contain only one operation. In contrast, the pointers stored in the instruction memory are called *multi-op* pointers, which point to the actual long instructions stored in the decoder memory. The instruction fetching and packing mechanism used in the UTDSP processor was implemented

based on this two-level fetching model. More details will be given in Chapter 3 where the design and implementation of the UTDSP Packer is explained.

2.3 C Compiler and Post-Optimizer

A C compiler translates a program written in C into a functionally equivalent program taking the form of the machine language of the target architecture; it uses machine-independent optimizations to increase the run-time performance of the resulting machine code. These optimizations usually include loop unrolling, common sub-expression elimination, strength reduction, and constant propagation [8]. Furthermore, the compiler can perform specific machine-dependent optimizations to take maximum advantage of the hardware resources in the target architecture. For instance, the machine-dependent optimizations include instruction scheduling, software pipelining, register renaming, data prefetching, and branch prediction when the target architecture is a general-purpose RISC processor.

Originally, the C compiler for the model architecture was based on the GNU C compiler (Gcc) because it is public-domain software; it uses a good suite of scalar optimizations; and it is easy to retarget to different architectures. However, the intermediate form Gcc uses provides too little information about the source program to implement machine-dependent optimizations. To implement DSP-specific optimizations without modifying Gcc, a post-optimizing pass was developed to perform the machine-dependent optimizations for the model architecture. Figure 2.5 shows the resulting two-phase compilation process. In the first phase, the C compiler translates C programs into sequential assembly language operations that can be executed on the model architecture. In the second phase, the post-optimizer back-end performs the architecture-specific optimization. The initial work on Gcc and the post-optimizer was done by Vijaya Singh [9]. The post-optimizer was later augmented by Mazen Saghir [2][43] and Mark Stoodley [10]. The Gcc front-end was later again replaced by the SUIF compiler [11] to enable the use of a more natural coding style for applications.The SUIF compiler was ported to the model architecture by Sanjay Pujare [12].

The post-optimizer optimizes the execution performance of a program by taking maximum advantage of the underlying hardware resources in the model architecture; it applies five optimi-

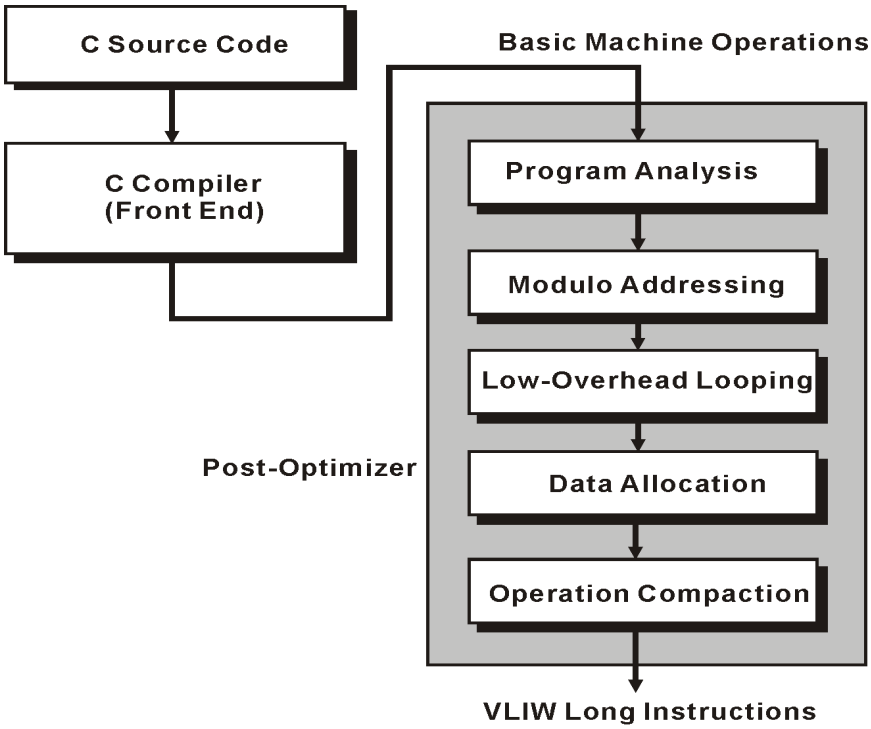


Figure 2.5: Two-phase compilation: C Compiler and Post-Optimizer

zation passes to the sequential machine operations generated by the front-end C compiler and creates long instructions that can execute on the model architecture. The five passes include the program analysis pass, the modulo addressing pass, the low-overhead looping pass, the data allocation pass, and the operation compaction pass, as shown in Figure 2.5.

First, the program analysis pass constructs a control-flow graph for the code generated by the front-end compiler. It then extracts information about its data-flow, control-flow, and aliasing characteristics, which are needed in the other phases. Second, the modulo addressing pass converts all arrays in the original code to circular buffers so that the elements in the arrays are accessed in a modulo manner. Third, the low-overhead looping pass tries to replace conditional branch instructions in a loop with a single low-overhead looping operation that specifies the iteration count and the addresses of the first and last instruction in the loop body. Fourth, the data allocation pass takes advantage of the dual data-memory banks of the model architecture and exploits parallelism by distributing program data among the banks. Finally, the operation compaction pass packs machine operations into long instructions using a list scheduling algorithm [13].

2.4 TI VelociTI and Philips R.E.A.L DSP Architectures

This section describes VLIW architectures that are used in two commercially available DSP processors — TI TMS320C62xx and Philips R.E.A.L DSP.

2.4.1 TI VelociTI Architecture

The VelociTI architecture is used in the TMS320C62xx, which is the latest in the TMS320 family of DSPs. The VelociTI architecture is an advanced VLIW design that has a long-instruction packing scheme to reduce storage requirements. The VelociTI uses a deep pipeline to eliminate traditional pipeline bottlenecks including memory access and multiply-accumulate operations. Figure 2.6 shows the block diagram of the VelociTI architecture.

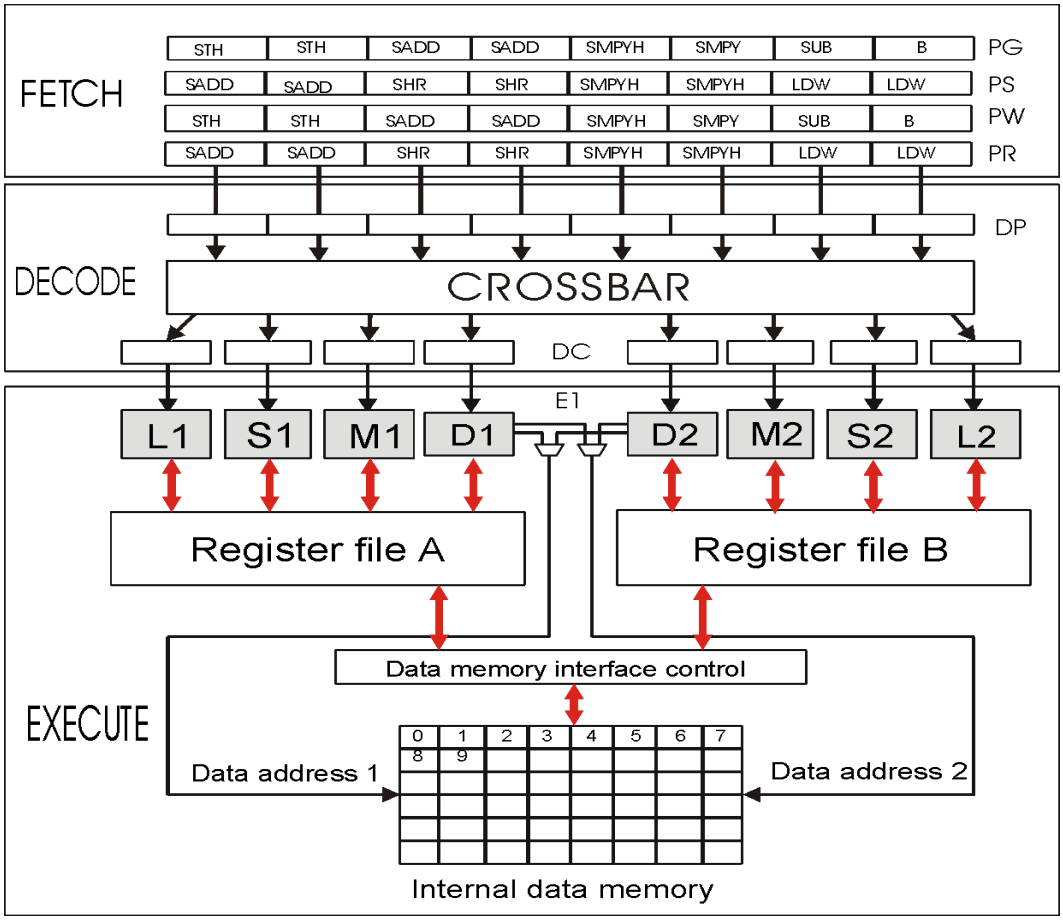


Figure 2.6: The TI VelociTI Architecture

There are eight functional units in the VelociTI architecture. They are divided into two data paths. Each data path has four functional units and a register file. The data in the two register files can be exchanged via a crosspath. Each functional unit can execute only a subset of the VelociTI instruction set. The VelociTI has a highly orthogonal, RISC-like instruction set and a load-store architecture — memory accesses are performed using explicit load or store instructions.

As shown in Figure 2.6, the VelociTI has three pipeline phases: Fetch, Decode, and Execute. The Fetch phase consists of four pipeline stages. In the Fetch phase, a program address is generated and used to fetch a long instruction (Fetch Packet) from instruction memory. In the Decode phase, which consists of two pipeline stages, the operations in the fetched long instruction are dispatched to their corresponding functional units via a crossbar. The crossbar is used because the VelociTI has a long-instruction packing scheme to reduce storage requirements. The details of this packing scheme will be discussed in Section 3.4.1.

In the Execute phase, the dispatched operations are executed in their corresponding functional units. The Execute phase is divided into 5 pipeline stages because each instruction uses different number of pipeline stages. Most of the instructions use one pipeline stage, whereas some instructions such as branch require 5 pipeline stages to execute. More details can be found in [14][26].

2.4.2 Philips R.E.A.L. DSP Architecture

The R.E.A.L. DSP (Reconfigurable Embedded DSP Architecture at Low-power and Low-cost) is designed as a flexible embedded core to enable an application-specific tuning and fast turn-around time. The R.E.A.L. DSP is a VLIW design with a dual Harvard architecture and 3 pipeline stages. The R.E.A.L. uses a look-up table to store its VLIW instructions. Figure 2.7 shows the block diagram of the R.E.A.L. DSP core.

When a 16-bit word is fetched from program memory and if its first 8 bits equal to the specified enabling mode, the lower byte of the word will be used as an address to fetch its corresponding long instruction stored in the ASI look-up table. The VLIW instructions stored in the ASI look-up table are 96-bit long and can specify many operations to control not only functional units but also application specific execution units (AXU). The AXUs are designed to execute a special set of instructions that are tuned for target applications. Most importantly, they can be placed anywhere in the datapath or the address functional units.

The ASI look-up table can contain only 256 long instructions and it does not use any instruction packing scheme. Although it stores the duplicates of a long instruction in the same table entry to reduce its storage requirement, we believe that the storage requirement is unlikely to be reduced because the possibility of having exactly the same long instructions is rare. More details can be found in [3][45].

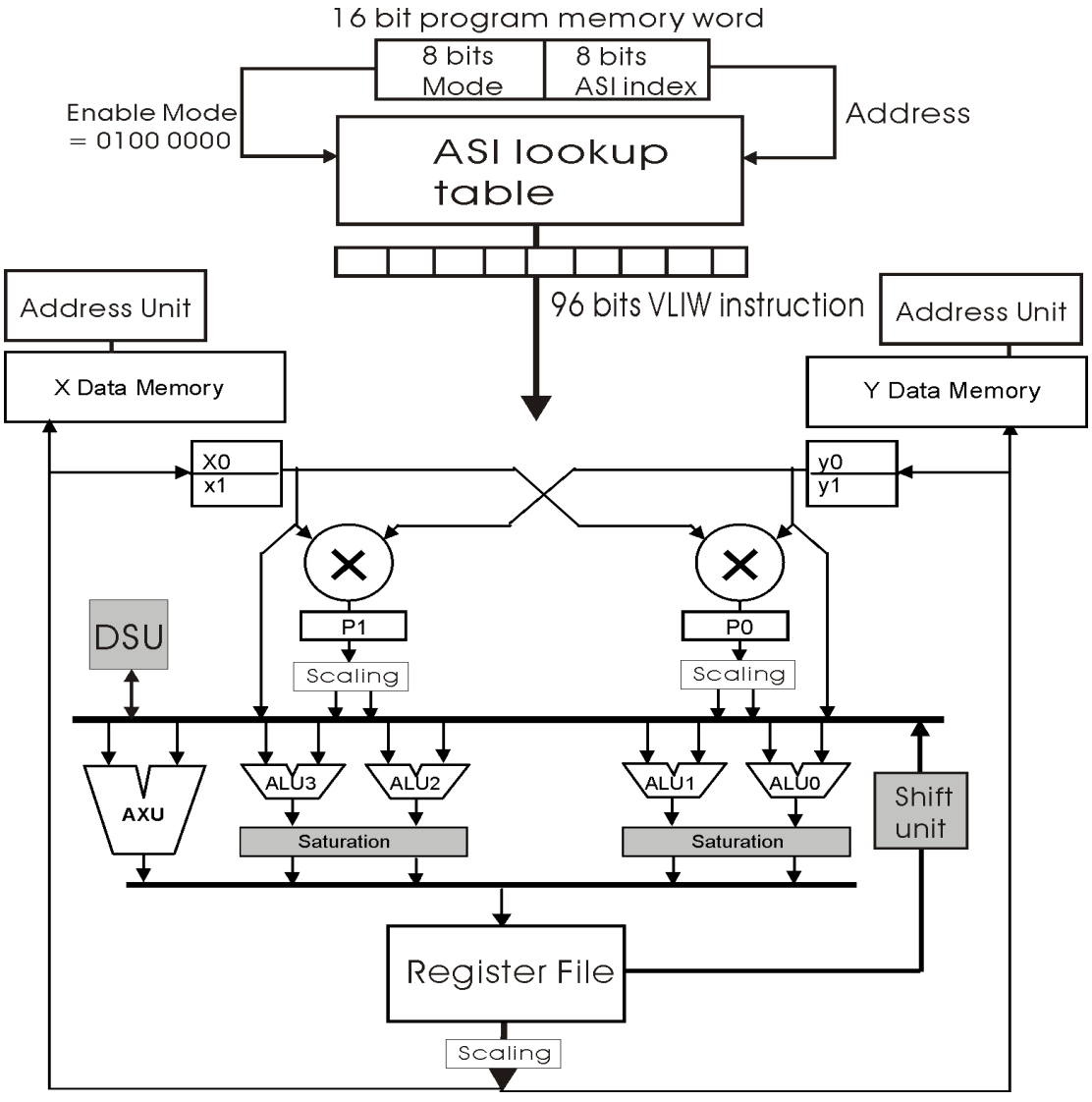


Figure 2.7: The Philips R.E.A.L DSP architecture

2.5 Summary

This chapter introduced the compiler system and the model architecture developed for an application-driven design methodology. In the application-driven design framework, the compiler-generated code for a set of target applications is used to measure the performance of the flexible model architecture. The measurements are compared with the application-specific constraints and the model architecture is iteratively modified until all the constraints are met.

The model architecture is based on a VLIW model because it is very flexible and can easily be configured to meet the target constraints. Although the long-instruction scheme in VLIWs is an easy target for HLL compilers, its high bandwidth and storage requirements prevent VLIWs from being used in cost-sensitive embedded systems. A long-instruction packing and fetching scheme that is based on a control-store mechanism was proposed by Mazen Saghir to overcome these problems [2].

Compiling an application into the long-instruction format involves a two-phase process. The first phase is to translate the source code into basic machine operations using the GNU C compiler; the second phase performs architecture-specific optimizations using a post-optimizer and generates long instructions that can run on the model architecture.

Having covered the work done by previous researchers and two commercially available DSP architectures in this chapter, the following chapters will focus on the design and VLSI implementation of the UTDSP processor and associate development tools. The UTDSP processor is based on the model architecture and incorporates the long-instruction encoding mechanism to solve the instruction bandwidth and storage problems.

Chapter 3

Long-Instruction Packing and Fetching for the UTDSP

In the last chapter, the model architecture and the optimizing compiler systems were discussed. Although the optimizing compiler could exploit enough parallelism and generate efficient code with the flexible model architecture, the storage requirements of long instructions and the high instruction bandwidth required represent major obstacles in developing a feasible system. This chapter describes the UTDSP instruction fetching and packing mechanism, which solves the problems mentioned above.

The UTDSP instruction fetching and packing mechanism was designed based on the two-level instruction fetching scheme discussed in the last chapter. Section 3.1 describes the basic architecture of this mechanism and a simple packing algorithm that reduces the instruction storage requirements. Section 3.2 presents a two-cluster packing algorithm that achieves a denser packing result by dividing a long instruction into two sub-words and sharing memory locations.

Section 3.3 describes the software implementation of the UTDSP packer using the two-cluster packing algorithm. It also shows that data structures constructed using template techniques not only ease the implementation of the UTDSP packer and assembly tools, but also shorten the design time used to explore various packing algorithms. Section 3.4 examines the impact of the packing algorithms on the storage requirements and compares the benchmark results of the UTDSP packer with that of Texas Instruments' VelociTI packing [14]. Section 3.5 summarizes this chapter.

3.1 Storing and Decoding Long Instructions

Fig 3.1 shows the two-level instruction memory system used in the UTDSP to store the long-instructions. This architecture was proposed by Mazen Saghir [2]. The instruction memory sys-

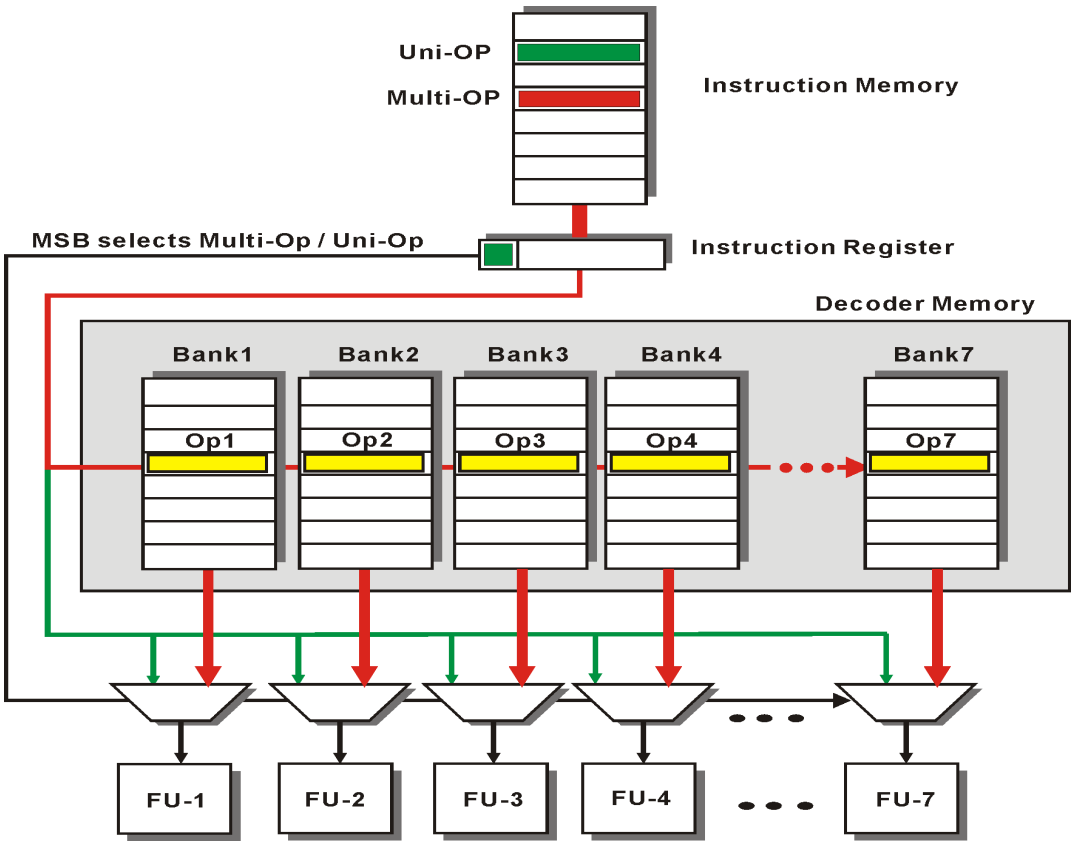


Figure 3.1: Block diagram of the UTDSP memory system

tem consists of two blocks: the instruction memory and the decoder memory. Long instructions that contain only one operation — uni-op operations — are stored in the instruction memory, while the other long instructions are stored in the decoder memory and their addresses — multi-op pointers — are stored in the instruction memory instead. The decoder memory contains seven banks, each of which is associated with a functional unit; therefore, the operations stored in a decoder-memory bank will be executed in its associated functional unit. Although the seven operations in a long-instruction word are distributed in the different banks of the decoder memory, they are stored in the memory locations that have the same physical address so that they can be fetched using their corresponding multi-op pointer stored in the instruction memory.

When a word is fetched from the instruction memory, its most-significant bit is examined to determine if the word is a uni-op operation or a multi-op pointer. If it is a uni-op operation, it is directly issued to an appropriate functional unit where it can be executed. In contrast, if a multi-op pointer is fetched, it is used to access the memory locations in the decoder-memory banks and the

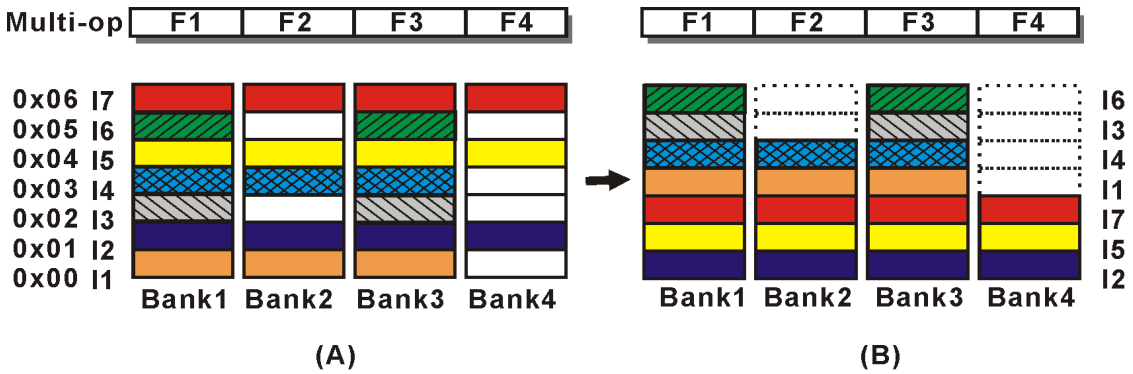


Figure 3.2: (A) Multi-op instructions stored in an arbitrary manner (B) Multi-op instructions stored according to field priorities

seven operations stored in the locations addressed by the multi-op pointer will be dispatched to their corresponding functional units.

3.1.1 Reducing the Size of Decoder Memory

Storing long-instructions in their original formats illustrated in Figure 3.1 can be very wasteful because of NOPs — the no operation fields in the long instructions. One way to reduce the overall size of the decoder memory is to store instructions in a way that allows some of the memory for NOPs to be omitted. Let the priority of a field in a multi-op instruction be the number of NOPs stored in the decoder-memory bank associated with the field. Multi-op instructions with operations in the higher-priority fields are stored first, starting from address zero, while instructions with operations in the lower-priority fields are stored last. Figure 3.2 shows the impact of storing instructions according to field priorities on the decoder memory.

In Figure 3.2 (A) multi-op instructions are stored in an arbitrary manner. Observe that bank4 suffers the most from the poor ordering of the multi-op instructions because bank4 contains the most NOPs (white boxes). Therefore, field F4, associated with bank4, is assigned the highest priority and the multi-op instructions that have operations in field F4 are stored first. Figure 3.2 (B) shows that multi-op instructions are stored according to field priorities. In this case the storage requirement for bank4 can be reduced to three words by chopping off the four consecutive empty words.

Moreover, the decoder-memory size can be further reduced by storing multi-op instructions with mutually exclusive operation fields into the same long-instruction word. Figure 3.3 (A)

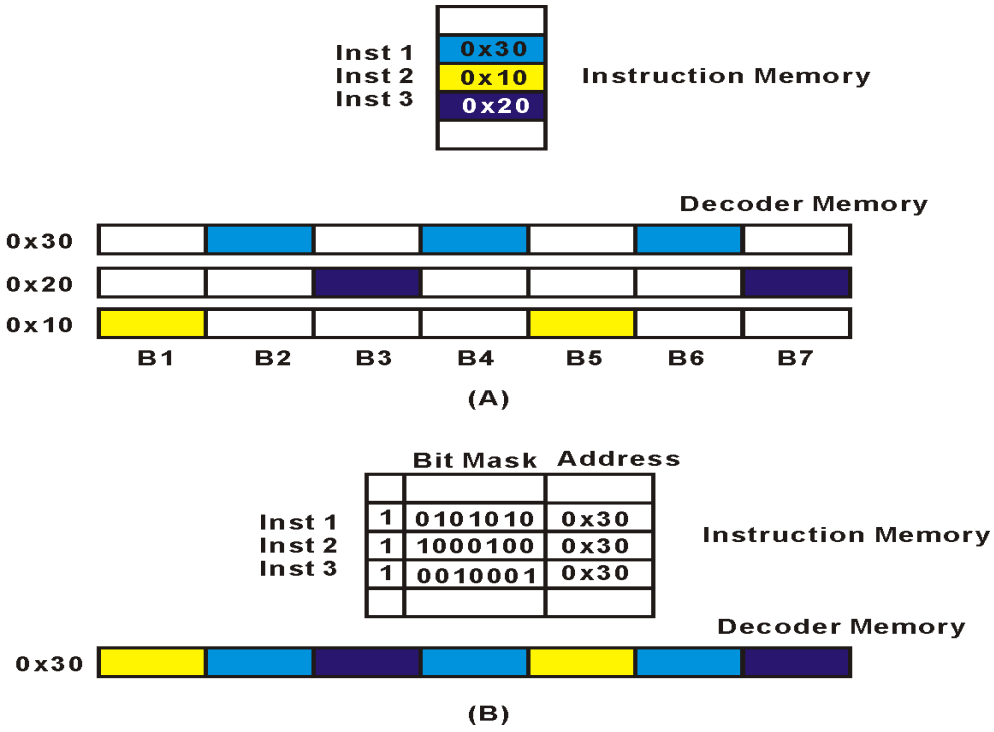


Figure 3.3: (A) Storing multi-op instructions in their original format (B) Storing multi-op instructions in a packed format

shows that three multi-op instructions are stored as three, separate, long-instruction words. When stored in this format, the instructions occupy 21 words, of which 14, or approximately 67% are NOPs. In contrast, Figure 3.3 (B) shows that the three instructions are packed into a single long-instruction word because their operations use mutually exclusive fields. In this case, storing the long instructions needs only seven memory words. Therefore, packing multi-op instructions with mutually exclusive fields into a single long instruction reduces the size of the decoder memory significantly.

To fetch the original operations of a multi-op instruction from the packed long-instruction word, the multi-op pointer should store not only the address of the packed word, but also a bit mask that selects specific memory banks in the decoder memory. Figure 3.3 (B) shows the bit mask fields in the multi-op pointers stored in the instruction memory. Note that the number of bits in the bit mask is the same as the number of memory banks; each bit in the mask is used to select its corre-

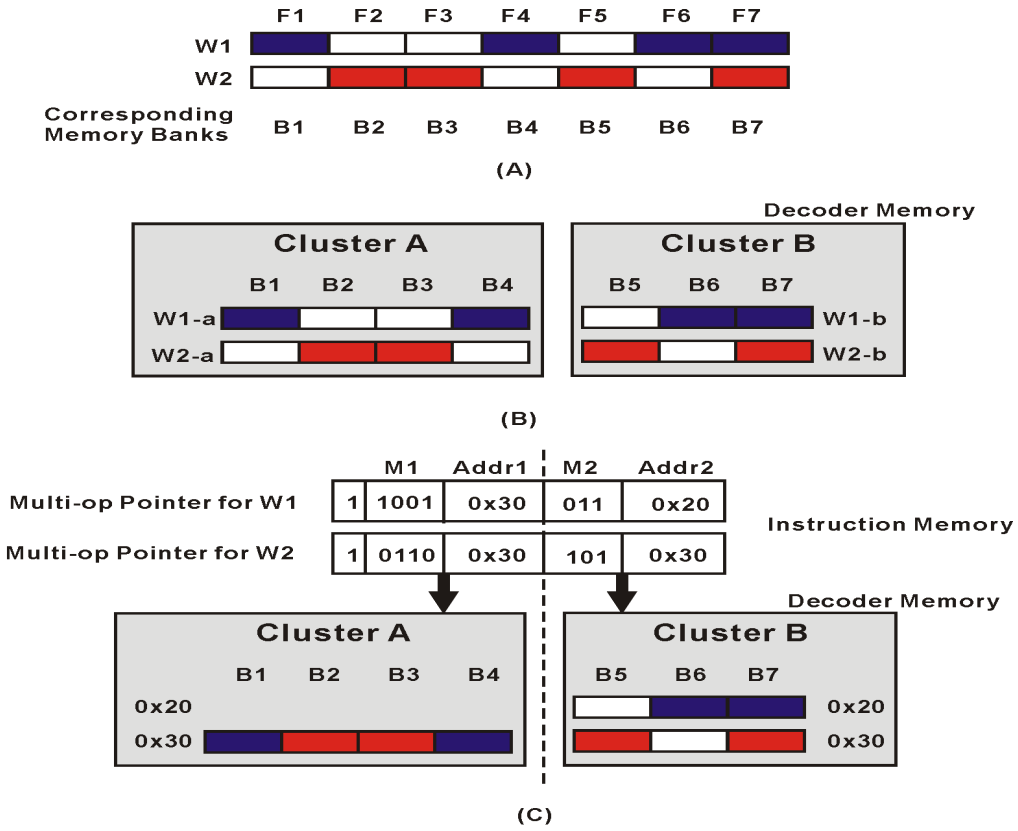


Figure 3.4: Field clustering packing method

sponding memory bank. The decoder-memory banks that are not selected by the bit mask will return NOPs.

3.2 Achieving Denser Instruction Packing

The major drawback of the packing method described in Section 3.1 is that two instructions cannot be packed into a single decoder-memory word even if they share only one field. This section describes two methods that can be used to achieve a denser instruction packing result. The final implementation of the UTDSP packer incorporates these two methods.

3.2.1 Field Clustering Method

Figure 3.4 (a) illustrates the problem where two instructions cannot be packed into a single word because both instructions use field F7. Mazen Saghir proposed a field clustering method to solve this problem [2]. Observe that the remaining fields in both instructions are mutually exclu-

sive, and that they could have been packed into the same memory word if field F7 was ignored. This suggests that dividing the memory banks into clusters can achieve a denser packing result. Figure 3.4 (B) shows that the memory banks are grouped into two clusters: cluster A consisting of memory banks B1 to B4, and cluster B consisting of banks B5 to B7.

As a result, in this example, multi-op instruction word W1 is divided into subwords W1-a and W1-b; W2 is divided into subwords W2-a and W2-b. The packing method is then applied to each cluster separately. Applying the packing method to cluster A will pack W1-a and W1-b into one subword in the decoder memory because the operations in W1-a and W1-b use mutually exclusive fields. The subwords W2-a and W2-b remain the same because both instructions use field F7. The Figure 3.4 (c) shows the final packing result using this two-cluster grouping method. Note that another set of bit masks and address fields must be added into the multi-op pointers to extract operations from the two different clusters. Figure 3.4 (c) also shows the contents of the multi-op pointers, which are used to extract the original multi-op instructions W1 and W2 from the two clusters.

3.2.2 Sharing Packing Slots

Clustering enables the exploitation of redundancy at the sub-instruction level, meaning that only one copy of the identical sub-instructions need to be stored in the decoder memory; however, the ideal packing result is still not achievable unless a seven-cluster configuration is used [2]. A further improvement to clustering can be achieved by making the following observation:

- **Observation**:

  With the help of the bit mask fields in multi-op pointers, two instructions can be packed into the same decoder-memory word as long as one instruction is a ***subset*** of the other one.

- The **definition** of ***subset*** in a long instruction is described as follows:

  A long instruction **X** is said to be a ***subset*** of the long instruction **Y** if and only if every operation in **X** can be found in the corresponding field in **Y** ,or the corresponding field in **Y** is empty.

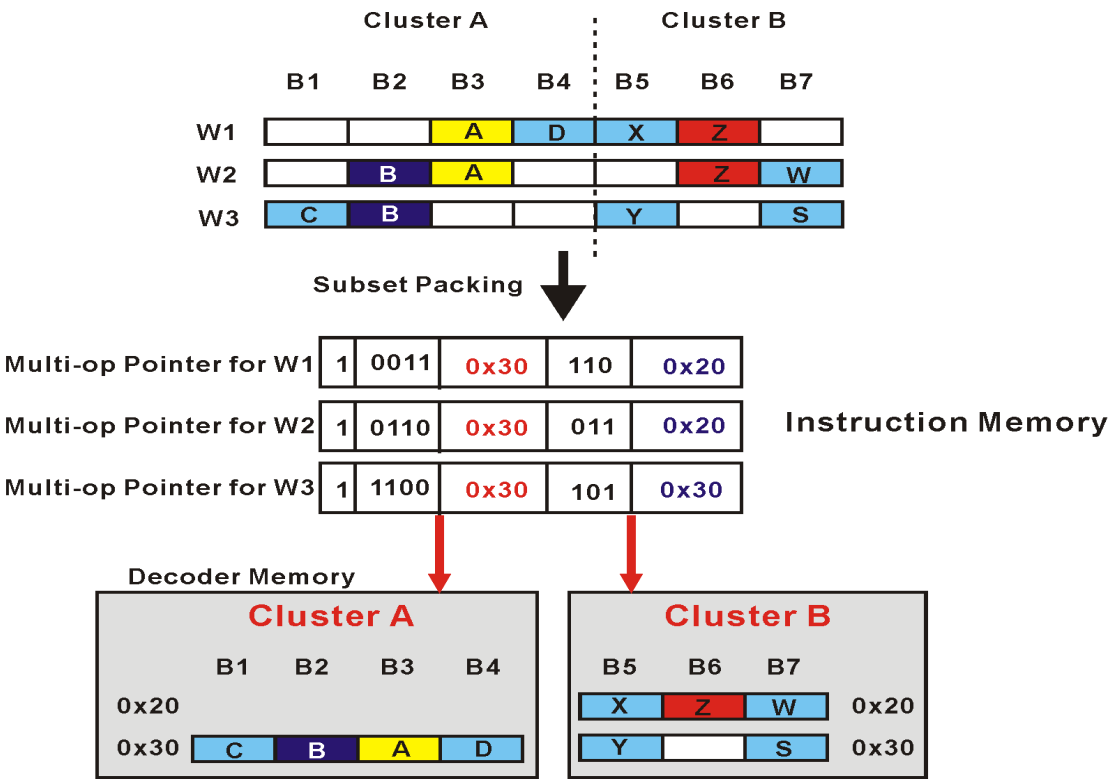


Figure 3.5: Combining the two-cluster packing and slot-sharing methods

Using the two-cluster packing method described in Section 3.2.1 increases the possibility of finding subset instructions to share the same decoder-memory word. Figure 3.5 shows that a denser packing result can be achieved by combining this subset sharing method with the two-cluster packing. In this example, the three subwords of long instructions W1, W2, and W3 in Cluster A can be packed into one subword in the decoder memory. The advantage of using the subset packing is that this mechanism needs no extra hardware.

Table 3.1 shows the impact of using the subset packing method on the decoder-memory requirements of the kernel and application benchmarks described in Chapter 2. The average decoder-memory requirements in the table are normalized to the ideal packing case, where ideal means that there are no NOPs in the decoder memory. Using the subset packing method reduces the decoder-memory requirements of the two-cluster packing by 10.16% and 10.25% on average in the kernel and application benchmarks respectively, achieving a packing result that is better than

the ideal packing case. It is even possible that with more clever register allocation, this result could be further improved.

Kernel Benchmarks	Decoder-memory Requirement (Ideal Packing Case)	Decoder-memory Requirement (Two-cluster Packing)	Decoder-memory Requirement (Two-cluster & Subset Packing)	Saving on Decoder-memory using Subset Packing
FFT_1024	1	1.10	1.10	0%
FFT_256	1	1.10	1.10	0%
FIR_256_64	1	1.16	1.16	0%
FIR_32_1	1	1.13	0.87	23.53%
IIR_4_64	1	1.11	1.08	2.5%
IIR_1_1	1	1.13	1.13	0%
latnrm_32_64	1	1.25	1.13	10%
latnrm_8_1	1	1.12	1	10.53%
lmsfir_32_64	1	1.10	0.86	21.74%
lmsfir_8_1	1	1.08	0.8	25.58%
mult_10_10	1	1.13	1.04	7.41%
mult_4_4	1	1.13	1.04	7.41%
all_kernels	1	1.16	0.89	23.33%
Average	1	1.13	1.02	10.16%

Application Benchmarks	Decoder-memory Requirement (Ideal Packing Case)	Decoder-memory Requirement (Two-cluster Packing)	Decoder-memory Requirement (Two-cluster & Subset Packing)	Saving on Decoder-memory using Subset Packing
G721a	1	1.08	0.95	11.85%
G721b	1	1.13	1.07	5.77%
V32.modem	1	1.08	0.89	17.80%
adpcm	1	1.12	1.09	2.44%
compress	1	1.08	0.98	8.80%
edge_detect	1	1.12	0.95	14.86%
histogram	1	1.06	0.96	9.30%
lpc	1	1.10	1.05	4.88%
spectral	1	1.07	1	6.95%
trellis	1	1.06	0.92	12.63%
all_applications	1	1.08	0.89	17.43%
Average	1	1.09	0.98	10.25%

Table 3.1: Impact of subset packing on decoder-memory requirements

3.2.3 Choosing the Number of Clusters in the UTDSP

In theory, there can be as many clusters as there are decoder-memory banks to achieve a denser packing result. In practice, however, the number of clusters that can be used is limited by the bandwidth of the instruction memory. Because one of the major reasons for using this two-level memory hierarchy is to reduce the off-chip instruction memory bandwidth, design decisions must be made based on a fixed instruction memory bandwidth.

The UTDSP model uses a 32-bit instruction word; therefore, the multi-op pointers should also be 32-bits long, so that both uni-op instructions and multi-op pointers can be stored in the instruction memory without wasting any bits. The bit mask fields in the multi-op pointer need seven bits in total to control the seven memory banks in the decoder memory. Also, the most-significant bit is used to identify itself as a uni-op or a multi-op pointer. As a result, 24 bits are left for encoding one or more address fields in a multi-op pointer.

Choosing the number of clusters to use in the decoder memory is a trade-off between the reduced cost achieved by a denser packing result, and the degraded performance resulting from a smaller size of decoder memory. Table 3.2 shows the maximum addressable decoder-memory space of different clustering configurations. On the basis of the data shown in the table, the two-cluster configuration was chosen for the UTDSP processor because the two-cluster case has a denser packing result than the ideal one (Table 3.1), while providing a much larger addressable space than the three-cluster configuration. For the three-cluster configuration, the total addressable decoder-memory space is only 2.2 K words, which is too small for most of the DSP applications.

Number of Clusters	Total address bits	Number of bits in a cluster address pointer	Addressable Memory space in each bank	Total addressable decoder memory space
1	24	24	16 M words	112 M words
2	24	12	4 K words	28 K words
3	24	8	256 words	2.2k words
4	24	6	64 words	448 words

Table 3.2: Maximum addressable space of different clustering configurations

To reduce the possibility of having NOPs in subwords, decoder-memory banks B1 to B4 were grouped into Cluster A and banks B5 to B7 were grouped into Cluster B. This configuration was chosen because the UTDSP compiler often schedules two memory load/store operations (associated with banks B1 and B2) and two address pointer operations (associated with banks B3 and B4) into one long instruction. Although further optimization might be achieved by using different grouping configurations, it was not attempted because using the subset packing method has already produced a result that is better than the ideal case according to Section 3.2.2.

3.2.4 The UTDSP Packing Algorithm

The original pseudo-code for the UTDSP packing algorithm was proposed by Mazen Saghir [2] and was then modified by the author to incorporate the subset packing method. The modified UTDSP packing algorithm, which is based on the two-cluster configuration, is described in Figure 3.6. This algorithm starts with a given list of unpacked instructions and two empty lists that are used to store the packed long-instruction words for the two clusters. The main loop is executed as many times as there are active fields. An active field is a member operation that has not been packed into its corresponding decoder-memory bank. At the beginning of each loop iteration, the number of active fields for each decoder-memory bank is calculated to decide its rank and the target bank in the current iteration. The highest rank (rank 7) is assigned to the decoder-memory bank that has the least number of active fields, while the lowest rank (rank 1) is assigned to the one that has the most number of active fields. The target bank is the one that has the highest rank (rank 7). Recall the situation described in Figure 3.2, where the memory banks with fewer operations suffer the most from the poor instruction ordering. Therefore, the long instructions that have operations to be stored in the target bank should be processed first.

Once the target bank has been determined, the unpacked instructions are divided into two lists: a candidate list and a reserve list. The candidate list contains instructions that have member operations associated with the target bank; the reserve list contains the remaining instructions. The instruction with the highest cost — the sum of the ranks of the memory banks associated with the operations contained — is then removed from the candidate list. The removed instruction will be added into the decoder-memory list provided it is not a *subset* of any existing word in the decoder memory list. If it is a *subset* of an existing decoder-memory word, it will be packed into the existing word as opposed to being added into the decoder-memory list as a new entry.

Algorithm: Two-cluster packing and slot-sharing method for Decoder Memory
Input: A list of multi-op instructions
Output: (1) A list of instruction memory words
(multi-op pointers and uni-op operations)
(2) Two lists of long instructions (Cluster A and Cluster B)

Assumptions and Definitions:
(1) Let **S** be the list of multi-op instructions
(2) Let **X** be the list of long instructions (decoder memory slots) in **cluster A**.
(3) Let **Y** be the list of long instructions (decoder memory slots) in **cluster B**.
(4) An **active field** is an operation in a multi-op instruction that is to be stored in its corresponding bank.
(5) A **target bank** is the memory bank that has the lowest number of active fields.
(6) **Candidate list** is a list of instructions that has an active field; **reserve list** is a list that has no active field.

1. While (there is an active field in cluster A banks) do
    2. Count the number of active fields for each bank in cluster A)
    3. Choose the **target bank**.
    4. Divide **S** into **candidate list** and **reserve list** according to the target bank.
    5. While (there is an instruction in **candidate list**) do
        6. **Inst** <= remove the instruction with **highest cost** from candidate list.
            (Please refer to Section 3.2.4 for the definition of **cost.**)
        7. Search existing slots in **X** for a sharable slot.
        8 . If (sharable slot is found) then
            9. Pack **Inst** into the sharable slot found in **X**.
        Else
            10. **Fit_list** <= find instructions that are a subset of **Inst** from **reserve list**.
            11. While (there is an instruction in **Fit_list**) do
                13. **Best_fit** <= remove the instruction with highest cost
                    from **Fit_list**.
                14. If (**Best_fit** is a subset of **Inst**) then
                    Inst <= Pack **Best_fit** into **Inst**.
            15. Add **Inst** to **X**.
16 Repeat step **1** to **15**, replace all **X**s by **Y**s and cluster **A** by cluster **B**.

Figure 3.6: The UTDSP packing algorithm

To use the empty fields in the newly added decoder-memory word, the reserve instructions are next searched for the instruction with the highest cost that can be packed into the memory word. The search is continued until no more instructions in the reserve list can be packed into the current decoder-memory word. After all instructions in the candidate list are processed, a new iteration starts again until all the active fields are stored. Because the decoder memory has two clusters, this packing process, described above, will be repeated for the second cluster.

3.3 Implementation of the UTDSP Packing Software System

The UTDSP long-instruction packer, which was implemented based on the algorithm described in the previous section, not only performs the two-cluster packing, but also parses assembly code and generates associated files for instruction simulation, assembly debugging, and VHDL simulation. Figure 3.7 illustrates the software modules in the UTDSP packer and the data flow between the modules and other development tools.

The UTDSP packer consists of a front-end assembly parser, the two-cluster packing kernel, an assembler, a TI VelociTI packing simulator, an output file generator, and a command center. The command center takes the hardware configuration of the UTDSP as input and sets up corresponding packing constraints for the packing kernel. The front-end parser parses VLIW assembly code and generates appropriate error messages when a syntax error is found. The packing kernel performs the UTDSP packing algorithm and passes the results to the assembler, which calculates addresses and encodes instructions. Finally, the output file generation module converts the packing results into various formats for simulation purposes. The VelociTI packing module uses the algorithm adopted by the Texas Instruments' new VelociTI architecture to pack the long instructions; the packing result of this VelociTI packer are compared with that of the UTDSP packer for benchmarking purposes. Section 3.4 details the comparison results.

3.3.1 Implementing the UTDSP Packer Using C++ Template Technology

As shown in Figure 3.7, the UTDSP packer needs not only a good packing algorithm, but also a powerful ability to handle the list-intensive processing in the software modules. Moreover, to ease the development and exploration of different packing algorithms, robust underlying data structures are needed to process the required computations. The most important data structure required

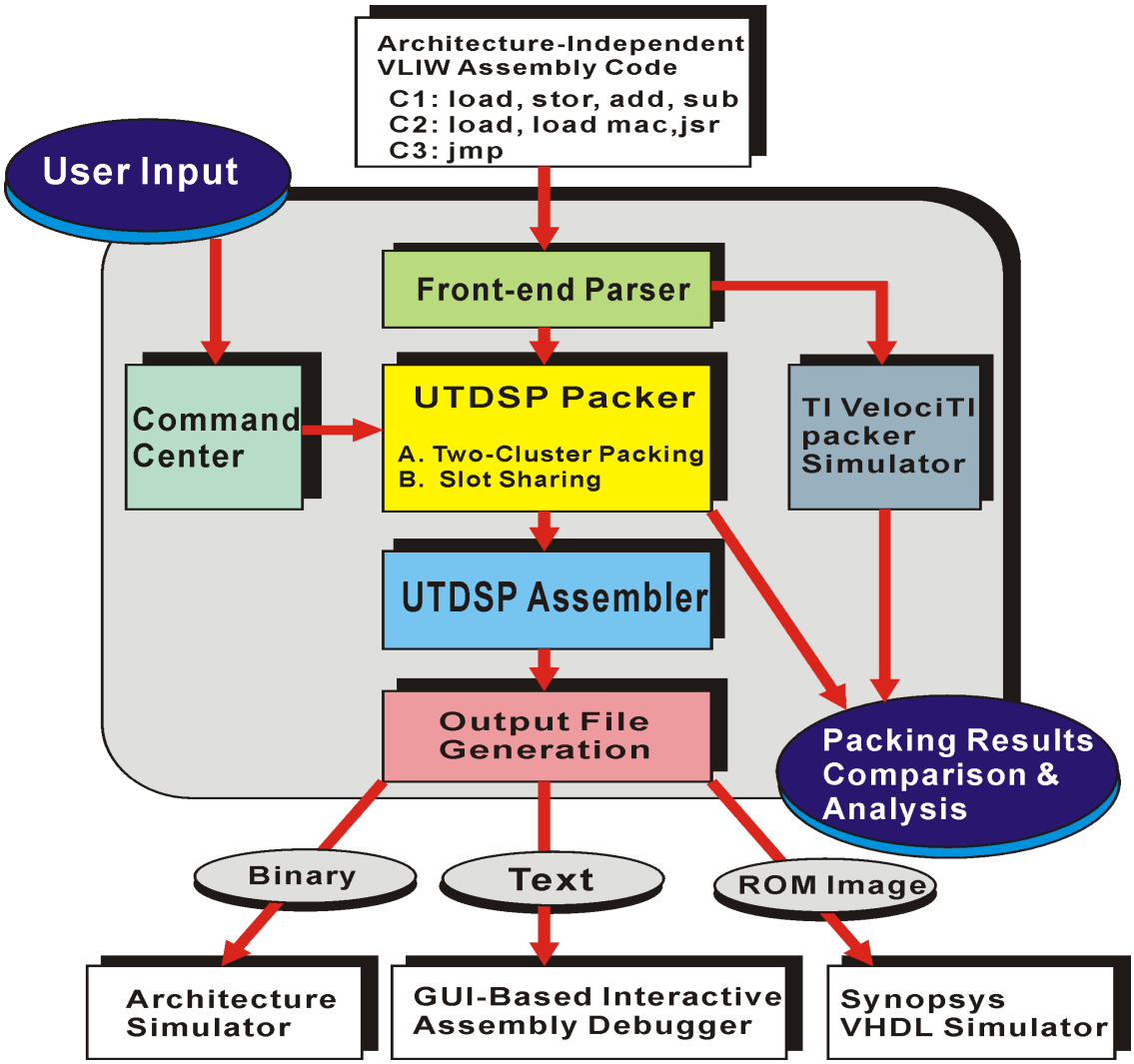


Figure 3.7: The UTDSP packer and software system

is a linked list that operates on many different data types — from both built-in and user-defined records — with associated functions such as insertion, deletion, merging, sorting, and binary search.

One way to build such a linked list is to use generic data structures. Figure 3.8 shows an internal representation of the generic linked list in C language. The list will have the same structure regardless of whether it stores strings, integers, floats, or user-defined data types. Note that the data items are not stored in the nodes of the list. Instead, each node contains a pointer to its data item. The fact that the data items don't reside in the nodes themselves leads to several drawbacks.

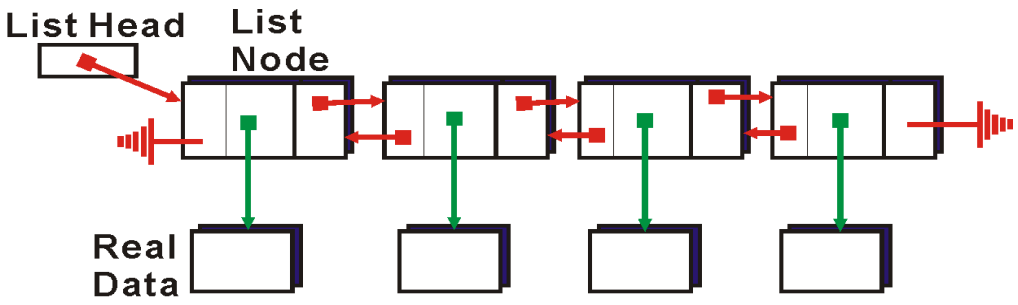


Figure 3.8: Generic linked list in C

First, pointer dereferencing is needed to get the actual value of a data item, rather then looking directly in the node. Second, a function pointer to the comparison function specific to its associated data type must be passed into procedures such as sorting and merging. This means that comparison functions must be explicitly constructed even for built-in data types such as integer and float.

C++ templates solve the problems mentioned above and provide several advantages over the generic data structures in C. First, neither constructing explicit comparison functions nor passing pointers to the sorting procedures is needed. By overloading the comparison operators such as ">", "<", and "=", comparisons of user-defined data types can be stated in the same format as that of built-in data types (ex. A > B). Second, a robust garbage collection mechanism can be encapsulated inside object destructors to completely eliminate memory leak problems, which often occur in the generic data structure implementations in C. For these advantages, the C++ template technique is used for the implementation of the UTDSP packer.

The most important part in the implementation of the UTDSP packer and assembler systems is List<T> — a C++ container template class that can store data items of any data type and perform various operations, such as insertion, deletion, sorting, and search, on its data items. Figure 3.9 shows the internal representation of List<T>. The argument class T represents all data types that are to be stored in the list container. Various user-defined classes that can be stored in List<T> were implemented to ease the development of the UTDSP packer. The user-defined classes include Token, DecoderMemory, and DataMemory, which are used to store parsed tokens, decoder-memory words, and data-memory words, respectively.

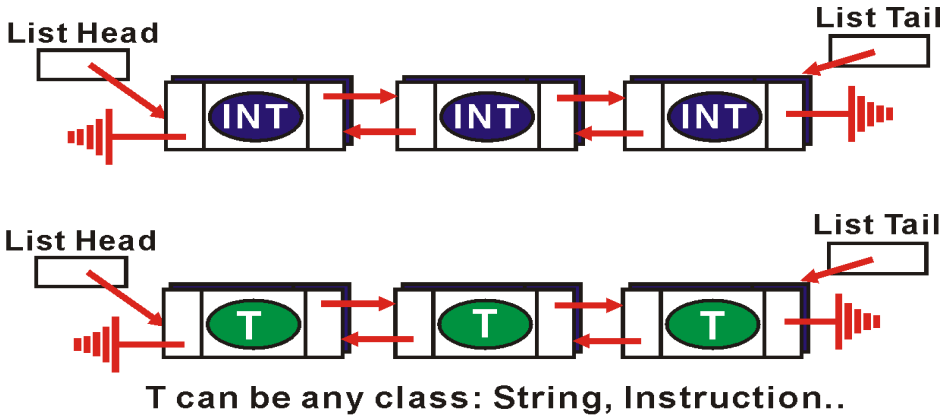


Figure 3.9: Container template class List<T> in C++

By using the robust underlying data structures designed in template techniques, the complexity and code size of the top-level application design are dramatically reduced; therefore, various packing algorithms can be easily explored to find the optimized one for the UTDSP packer.

3.4 Results and Analysis

Having covered the design and implementation of the UTDSP packer, this section examines the impact of using the two-cluster instruction packing algorithm on the storage requirements. For comparison purposes, TI's packing method was also implemented. Section 3.4.1 introduces the packing methods used in TI's VelociTI DSP architecture [14]. Section 3.4.2 analyzes the benchmark results of the TI VelociTI packing algorithm and the UTDSP packer. Section 3.4.3 discusses the impact of the two-cluster packing and fetching on execution performance.

3.4.1 Packing Mechanism used for TI VelociTI architecture

The VelociTI architecture used in the TI TMS320C62xx family of DSPs is a VLIW design that has a long-instruction packing mechanism. Figure 3.10 shows the packing mechanism used in the VelociTI architecture. In VelociTI there are eight instruction slots in one long-instruction word, which form a fetch packet (FP). Each instruction fetch moves an FP from the instruction memory to the instruction register. All the instructions that are executed in the same cycle are packed into one execute packet (EP); therefore, the number of instructions that can be contained in an EP

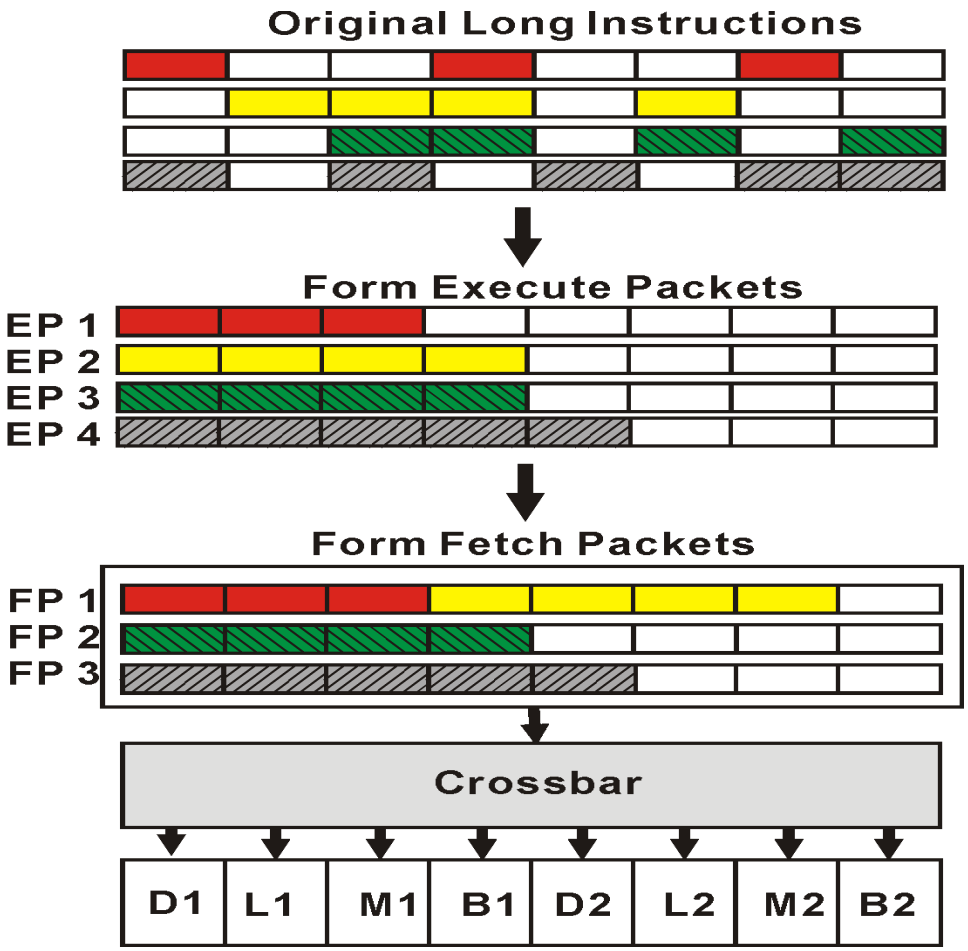


Figure 3.10: Packing mechanism in the TI VelociTI architecture

ranges from one to eight. The EPs are then packed into a block of consecutive FPs in an adjacent manner. The only restriction is that no EP can be split across two FPs.

In the example shown in Figure 3.10, execution packets are formed by compressing out the NOP operations in the original long instructions. The execution packets are then packed to form fetch packets; execute packets EP1 and EP 2 are packed into FP 1. In contrast, EP 3 and EP 4 cannot be packed because EP 4 cannot be stored separately in two FPs. During program execution, when FP1 is fetched, EP1 is first executed and then EP2. Then FP2 is fetched and EP3 is executed, followed by the fetch of FP3 and the execution of EP4. In this packing mechanism, an extra decode phase and a crossbar are needed to decode the instructions in an EP and assign them to appropriate data path and functional units.

3.4.2 Benchmark Results for the UTDSP Packer and the VelociTI Algorithm

To evaluate the VelociTI packing algorithm, a packing kernel that uses the VelociTI algorithm was implemented and incorporated into the UTDSP packer. The UTDSP packing kernel was also modified to collect statistical data in each packing phase. Figure 3.11 shows the methods used to calculate the storage requirements for the different packing schemes. First, the original format represents the storage requirements for the upper bound in which all instructions are stored in their original, long-instruction format in the instruction memory without using the second-level decoder memory. Second, the uni-op case represents a lower bound, where only the valid operations are stored in the instruction memory. Third, the UTDSP packing method A uses the two-cluster and subset packing algorithm with an assumption that the size of each decoder-memory bank can be configured independently, while packing method B assumes that the banks in a cluster must have the same size. Although the assumption made in method A seems too optimistic, method A is closer to reality because the decoder-memory configuration can be selected based on the average usage of memory banks in the target application domain.

The benchmarks used in this analysis are obtained from the UTDSP benchmark suite described in Chapter 2. Figure 3.12 shows the storage requirements of the benchmarks for each of the packing methods mentioned above. In each case, the storage size is normalized to the upper bound. The results indicate that the storage requirements of packing method A are 39% - 65% (average 51%) of those for the upper bound, which is slightly better than the packing rate of the VelociTI packing method (average 53%). The results also show that the storage requirements of packing method B are 49% - 77% (average 61%) of those for the upper bound. Table 3.3 summarizes these results. Because the UTDSP adopts an application-driven design methodology, where hardware configurations are chosen according to the target applications, the statistical results generated from packing method A can be used to select the configurations for the decoder-memory banks to minimize the storage requirements.

Moreover, the UTDSP packing mechanism provides a solution to the memory bandwidth problems that the VelociTI packing method cannot solve. Specifically, the VelociTI packing method requires that its instruction memory stay on-chip to maintain the original throughput. When the

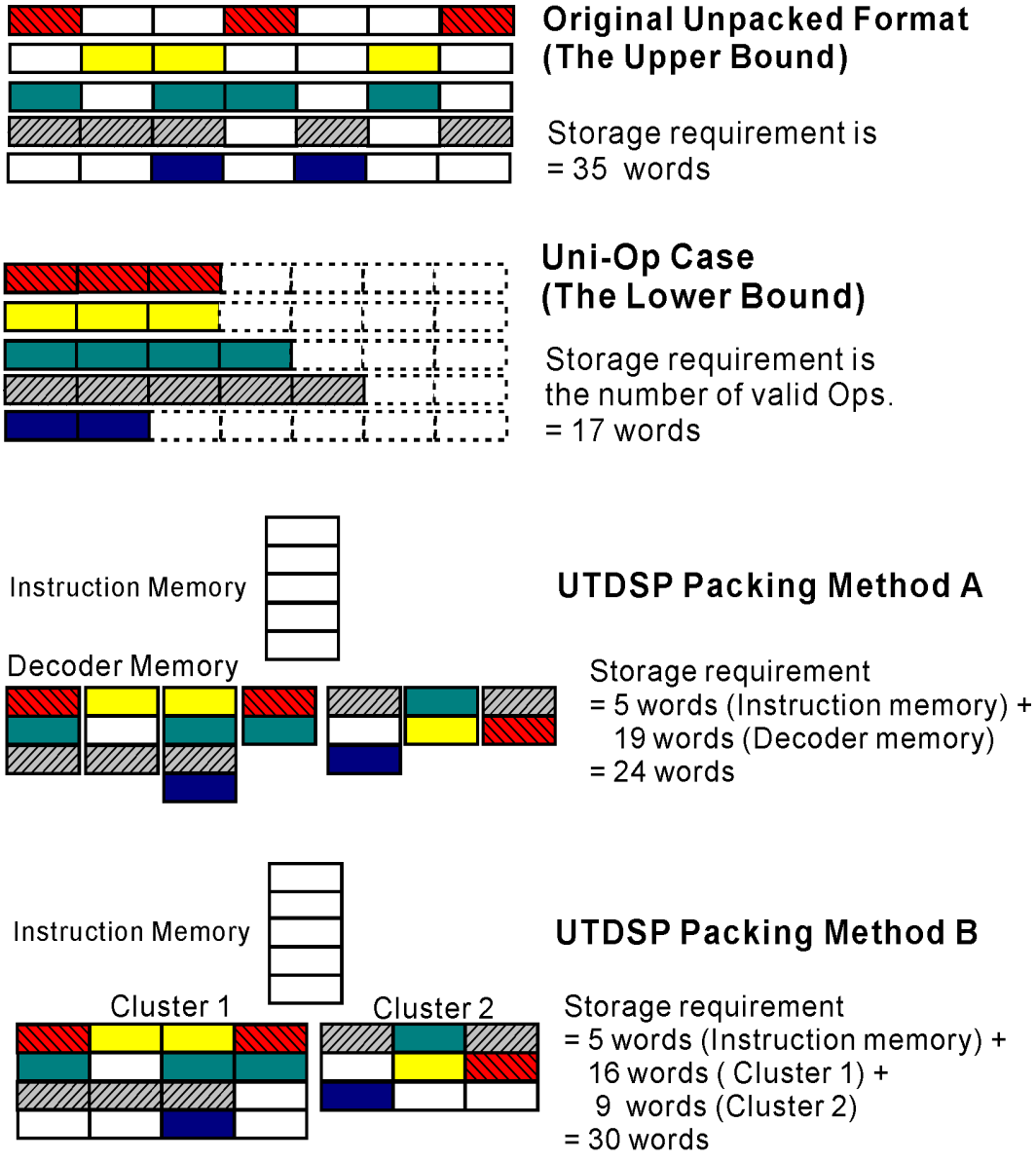


Figure 3.11: The storage requirements of different packing methods

size of long instructions exceeds the capacity of the on-chip instruction memory and the off-chip instruction memory has to be used, fetching long instructions from the off-chip memory through a 32-bit bus will significantly degrade the expected throughput.

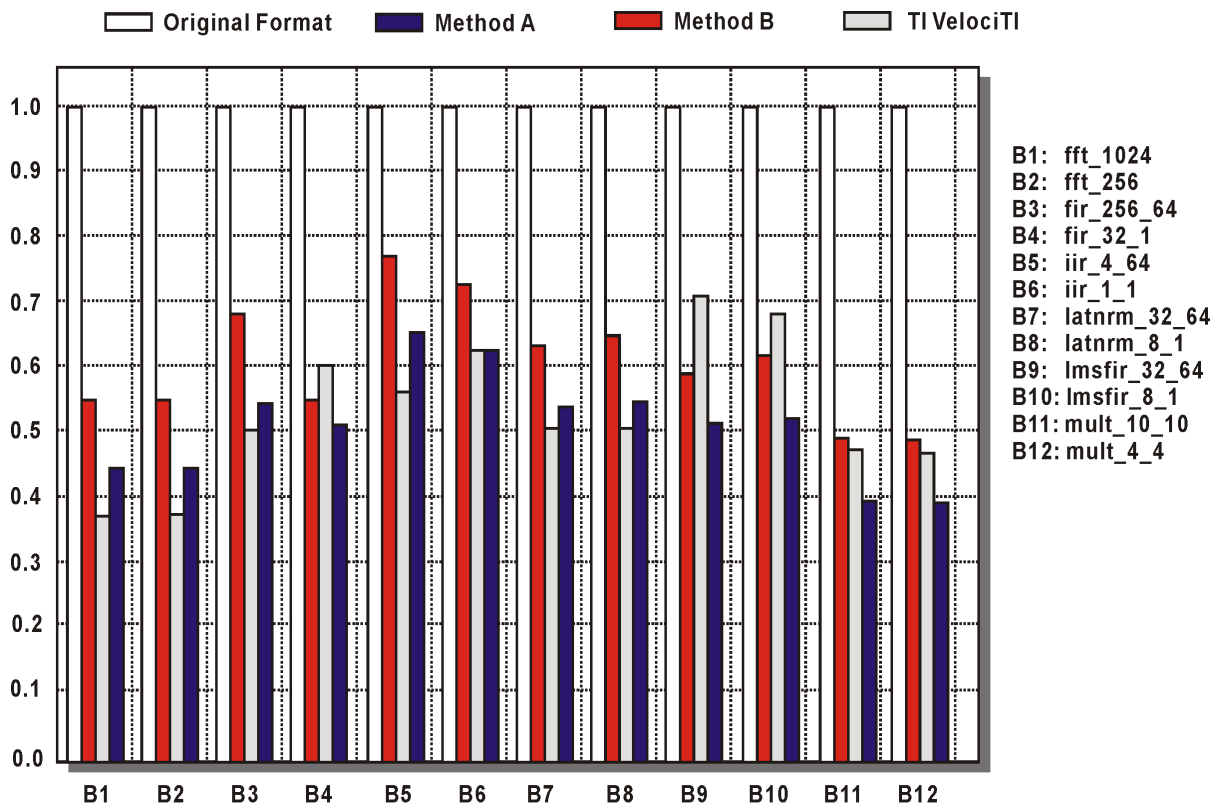


Figure 3.12: The storage requirements of the UTDSP kernel benchmarks for different packing methods

Packing Method	Average Storage Requirement
Original unpacked format	1
UTDSP packing method B	0.61
TI VelociTI packing	0.53
UTDSP packing method A	0.51

Table 3.3: Average storage requirements of the results in Figure 3.12

In contrast, the UTDSP solves these bandwidth problems and reduces to a minimum the impact of insufficient on-chip memory on performance in the following two ways: First, by storing only uni-op operations and multi-op pointers in the instruction memory, the instruction-memory bandwidth is reduced to 32 bits, which enables the use of off-chip instruction memory without affecting the performance. Second, since 90% of a program's execution time is spent on 10% of its code, only the kernel parts of DSP applications need to be stored on-chip when the on-chip mem-

ory is not large enough to accommodate all the long instructions. The remaining long instructions can be serialized and stored in the off-chip instruction memory without severely degrading the original performance.

Also, the UTDSP instruction fetching mechanism needs neither the cross bar nor the extra decoding logic that is used in the TI VelociTI architecture to route the operations in a long instruction to their corresponding functional units. Therefore, the UTDSP scales better than the TI VelociTI architecture when extra functional units are added for performance.

3.4.3 Impact of the Two-Cluster Packing and Fetching on Execution Performance

The major drawbacks that prevent VLIW architectures from being used in cost-sensitive systems are their high instruction bandwidth and storage requirements. However, with the two-cluster packing and fetching mechanism, the UTDSP can achieve significant performance speedup over a traditional architecture using its VLIW design, while maintaining the same instruction-memory bandwidth and having a modest increase in storage requirement. Table 3.4 summarizes the average execution performance and storage requirements of the two-cluster packing scheme and the uni-op case. In the uni-op case (lower bound), all the operations in long instructions are serialized and stored in the instruction memory as uni-op instructions; therefore, there is no fine-grain parallelism to exploit in the uni-op case.

	Average Storage Requirements	Average Execution Performance
Uni-op Case	1	1
Two-cluster Packing	1.27	2.6
TI VelociTI Packing	1.30	N/A

Table 3.4: Trade-off between storage requirements and execution performance

The data shown in the table are normalized to the uni-op case. These results show that the area overhead introduced by the two-cluster packing mechanism is 27% of the total storage area in the uni-op case. However, the execution performance of the two-cluster scheme is 2.6 times faster than that of the uni-op case. Moreover, both cases have the same instruction-memory bandwidth (32 bits). Using the two-cluster mechanism not only significantly increases the UTDSP performance but also reduces the area overhead to a minimum. For comparison purposes, the average storage requirement of the TI VelociTI packing is also shown in the table. The execution perfor-

mance of the VelociTI architecture is unknown because its architecture simulator is not available. However, it can be expected that the UTDSP achieves an execution performance comparable to the VelociTI because they have similar VLIW architectures, pipeline stages, and instruction execution methods.

3.5 Summary

This chapter described the packing and fetching mechanism used in the UTDSP. This long-instruction packing scheme reduces the storage requirements while eliminating the memory bandwidth problems that plague other VLIW architectures. The implementation of the long-instruction packer incorporates a template design approach that dramatically reduces design efforts and complexity. Benchmark results indicate that the UTDSP packing scheme achieves a packing rate comparable to its commercial counterpart and provides a solution to the storage problems that the VelociTI cannot solve.

Chapter 4

Development Tools

This chapter describes the design and implementation of two important development tools for the UTDSP system: an architecture simulator and a GUI-based assembly debugger. First, the architecture simulator collects run-time data and helps to evaluate various design decisions such as decoder-memory size, instruction-set modification, and register-file port sharing. Being written in a high-level language, the architecture simulator, which also serves as a behavioural model of the UTDSP, can be easily modified to experiment with design trade-offs. Once the experimentation is finished, register transfer level (RTL) modelling can be constructed according to the optimized hardware configurations. Second, the assembly debugger provides a true graphical windowing system with interactive debugging features, such as single-step tracing, memory probing, and breakpoint debugging, to ease the development of assembly programs.

Section 4.1 describes a hardware modelling technique that is used to construct the architecture simulator in a high-level language. This modelling technique, which is based on the object-oriented (OO) method, eliminates the design gap between the behavioural and RTL models of the UTDSP, so that the behavioural model can be easily converted into its corresponding RTL model. Moreover, using this OO model, the GUI-based assembly debugger can be implemented by just adding self-displaying and event-listening capabilities to the hardware objects in the architecture simulator. Section 4.2 shows the design and implementation of the architecture simulator. Section 4.3 describes the implementation of the GUI-based assembly debugger. Section 4.4 summarizes this chapter.

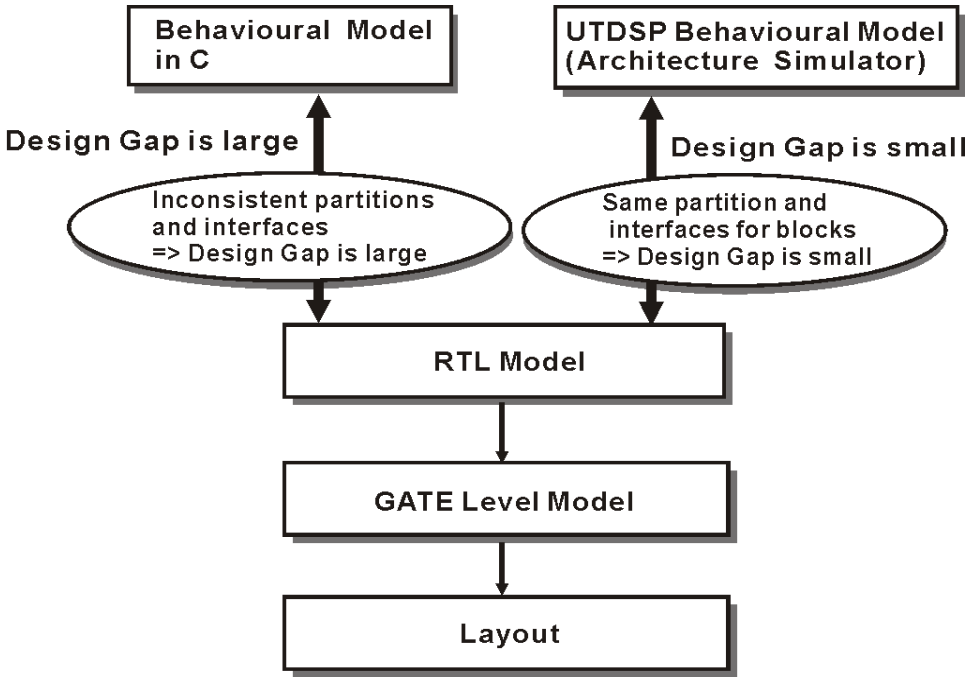


Figure 4.1: Design Gap between behavioural and RTL models

4.1 Behavioural Modelling Methods for the UTDSP

In the application-driven design methodology described in Chapter 2, the UTDSP needs to be modified according to the target applications; therefore, it is highly desirable that the UTDSP provide a model in a high-level language to shorten the turnaround time for the design exploration. Once the exploration is finished and the hardware configuration is decided, the model can be translated into an RTL model in Verilog or VHDL. This rewriting step is the so called design gap [17][22], which is shown in Figure 4.1. Compared with the other design phases shown in the figure, which are automated by CAD tools, the rewriting step requires that designers manually create two different versions of the design; therefore, the design gap duplicates design effort and becomes a bottleneck in the design flow. To minimize the design gap, the architecture simulator has to be constructed in a level of abstraction that is high enough to allow a short turnaround, but is able to ease the translation to its corresponding RTL model.

Moreover, when the UTDSP is used in a core-based design and needs to be modified for system integration, the architecture simulator will play an important role in providing a fast, easy-to-

modify simulation model. With the size and complexity of today's hardware design, modular, composable design based on a system-on-a-chip (SOC) methodology is the only approach that works [15]. Under the SOC methodology, providing behavioural models with a high enough level of productivity for system blocks early in the design cycle is a major issue.

Conventionally, hardware designers tend to use procedural languages like C to construct the behavioural model of their design and collect the statistics needed for performance optimization. This method is particularly common for CPU development [17]. However, C models suffer from the design gap problem most because C is unable to provide enough language constructs for digital hardware modelling. There is no real connection from C models to hardware design [17][22].

Therefore, the UTDSP architecture simulator should be built not only to perform the design exploration but also to minimize the design gap to ease the development of its corresponding RTL model. Also, the architecture simulator must be easily converted into a GUI-based assembly debugger with little or no extra effort. This capability is especially important in a core-base design because designers are ensured to have a GUI-based debugger that is functionally consistent with their modified behavioural model.

4.1.1 Choosing the Correct Modelling Language

Choosing a language model that best describes the characteristics of digital hardware is important to fulfill the goal mentioned above. Current high-level programming languages can be categorized into four models: procedural model, logic model, functional model, and object model [18]. First, in the procedural model, a program consists of a sequence of instructions that access named memory locations; problems are solved by executing the instructions that change the values of the memory locations. Languages using the procedural model include Pascal, C, and Fortran. Second, in the logic model, a program consists of a series of queries that are used to find solutions, and a set of databases. When queried, the databases answer the question by inferring new information using their existing knowledge and implication rules. Prolog is one of the languages using the logic model. Third, in the functional model, a program consists of a series of functions that might be composed of other functions; problems are solved by applying the functions to input data to generate output data. Lisp, APL, and Scheme are languages using the functional model. Finally, in the object model, problems are solved by creating objects that model real-world entities and sending messages to them. The objects react to the messages according to

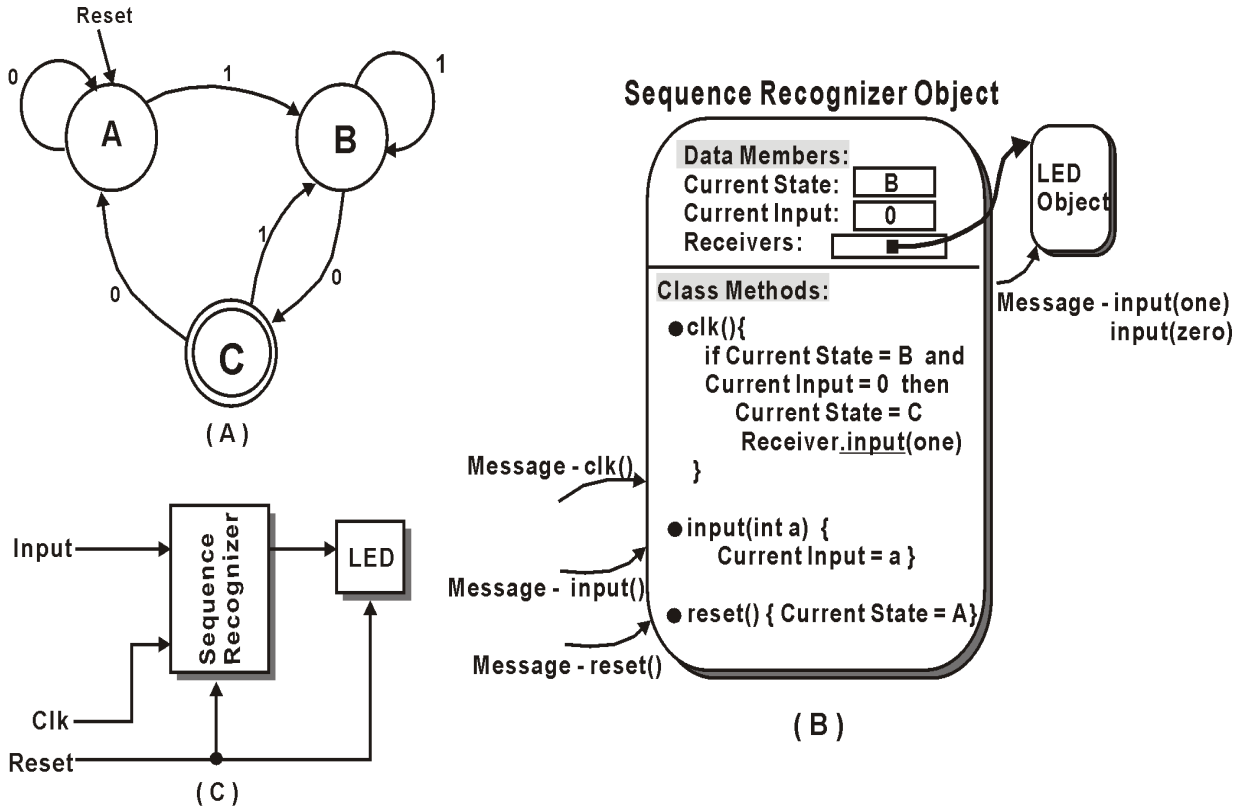


Figure 4.2: (A) An FSM that accepts input sequence “10”. (B) The FSM’s equivalent object model. (C) The resulting digital system blocks converted directly from the object model.

their own behavioural patterns. Languages that use the object model include Smalltalk, C++, and Java.

4.1.2 Digital Systems Modelling Techniques for the UTDSP

We believe that the object model is most suitable for digital system modelling, and should be chosen for implementing the UTDSP simulator and debugger. We established several digital hardware modelling methods using the object model so that the implementation of the simulator and debugger can follow these rules. First, an object itself can describe a complete finite state machine (FSM) without using any auxiliary functions or data structures. In fact, an object can model any class of automata in the Chomsky hierarchy [19] by using its data members as state memory, and its member operations as state transitions. Figure 4.2 (A) shows a simple FSM that recognizes the input sequence “10”. Figure 4.2 (B) shows its corresponding object representation.

The sequence recognizer object has three data members: CurrentState, CurrentInput, and Receiver. CurrentState stores the current state of the object itself; CurrentInput stores the current value of its input signal. Receiver stores the references of the objects to which the recognizer sends its output signal. In this case, the recognizer sends its output signal to the LED object, which will light up when it receives the *input(one)* message. The recognizer object responds to three different messages: *clk()*, *input()*, and *reset()*. When receiving the *input()* message, the recognizer invokes its class method *input()* and updates CurrentInput. When receiving the *reset()* message, the recognizer invokes its class method *reset()* and initializes CurrentState to state A. When receiving the *clk()* message, the recognizer determines its new state according to CurrentState and CurrentInput. Note that if the new state is the accepting state (state C), the recognizer will send message *input(one)* to the LED object.

Second, the message sending mechanism used in the communication between objects is used to model the signals sent between digital components. Figure 4.2 (C) shows the resulting digital system that is constructed by directly translating the message interfaces and partition from its object representation described in Figure 4.2 (B). Using this modelling method maintains an identical system partition and block interfaces between a behavioural model and its digital hardware.

Third, the relationship between a class and its objects in the object model is exactly the same as that between a component type and its instances in digital systems. Specifically, in digital systems, an arbitrary number of instances (objects) can be instantiated from a specific component type (class). Each of the instances has the same behavioural pattern (class methods) defined in their component type, but each one has its own current state (object's own data members). For example, in a shift register that consists of D flip-flops, each of the D flip-flops keeps its own current state — the data latched — although they have the same behaviour — the truth table of D flip-flops.

4.2 Design and Implementation of the UTDSP Architecture Simulator

The UTDSP architecture simulator is designed with the following three goals: First, it should be easily constructed and be able to collect statistics required for performance analysis. Second, it should bridge the design gap so that the RTL model of the UTDSP can be constructed according to the system partition and block interfaces of the UTDSP architecture simulator. Third, it must be

designed in a way that an interactive GUI can be incorporated into the model, so that core-based designers can always have a functionally equivalent GUI-based assembly debugger no matter how they modify the behavioural model.

Java was chosen to implement the UTDSP architecture simulator because it fulfills the goals mentioned above. Java helps the implementation of the simulator and debugger in the following ways: First, Java is a pure OO language, which provides the object model needed for modelling digital systems. Second, Java provides an abstract window toolkit (AWT) so that the GUI-based assembly debugger can be implemented by adding window-displaying and event-listening capabilities to the UTDSP architecture simulator. Third, unlike the window systems of the other programming languages, Java AWT achieves a high degree of portability, which makes the GUI-based assembly language debugger run on different computer systems without modifying its source code. Finally, the speed disadvantages of Java versus C or C++ are becoming less of an issue as Just-In-Time (JIT) and native-code compilers are becoming available [20].

4.2.1 Creating the Object Model of the UTDSP

Figure 4.3 shows the object model of the UTDSP and the message sending mechanism between the objects. Each object represents a hardware component in the UTDSP; the messages that it receives represent the input signals of the component, while the messages it sends represent the output signals. The **PC** object generates addresses that are used to fetch uni-op operations or multi-op pointers from the instruction memory object (**Inst**).

The **PC** object contains two **Stack** objects, which are used to handle the address operations for branching and zero-overhead looping instructions. Upon receiving the *fetch()* message, the **Inst** object will use the address embedded in the message to retrieve the corresponding instruction from its data members and send the instruction to the two **Cluster** objects. After receiving the *fetch()* message sent from **Inst**, the two **Cluster** objects retrieve the corresponding multi-op operations and broadcast them with the *exec()* message to the associated functional unit objects (**EU 1** - **EU 7**), which handle the logical and arithmetic operations.

Similarly, when the **EU** objects need operands, they will send requests to the register file objects (**REG**) and **REG**s will return the data requested. On the other hand, when **EU**s execute load or store operations, they will send requests to the data memory objects (**DataMem**) and the **Data-**

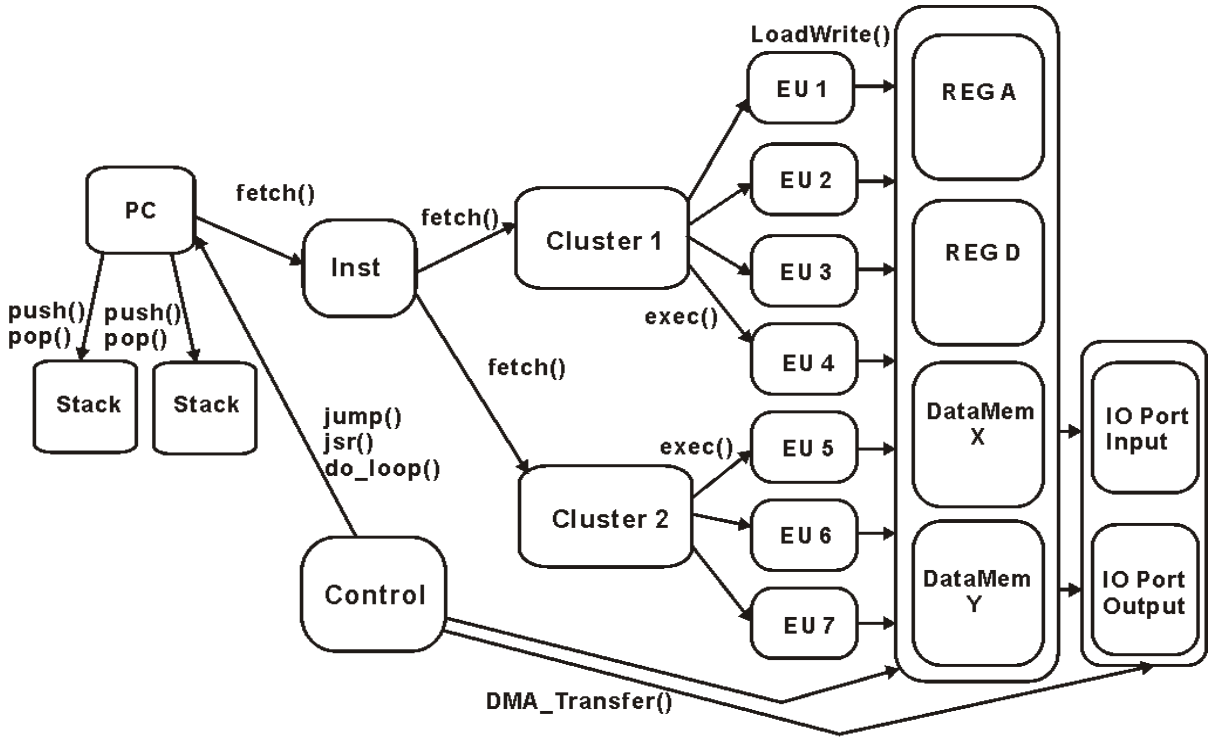


Figure 4.3: The object model of the UTDSP

Mems will perform the memory accesses requested. Finally, the **Control** object handles the DMA requests that transfer data between the **IO** objects and **DataMem**s.

4.2.2 Simulating the UTDSP Object Model

Simulating the UTDSP object model is simple. Every object is responsible for keeping its own properties (data structures) and remembering its own behaviour (algorithms); therefore, the top-level simulator driver no longer needs the complex data structures and algorithms that are usually seen in the simulators built using the procedural model. To simulate the object model of the UTDSP, the objects must be connected according to their signal flow so that they know where to send messages at run time. Consequently, each object has data members that are used to store the object references of its message receivers.

When an object needs to broadcast messages to the receiver objects that are connected to one of its output ports, it will fetch the references stored in the data member associated with that port and

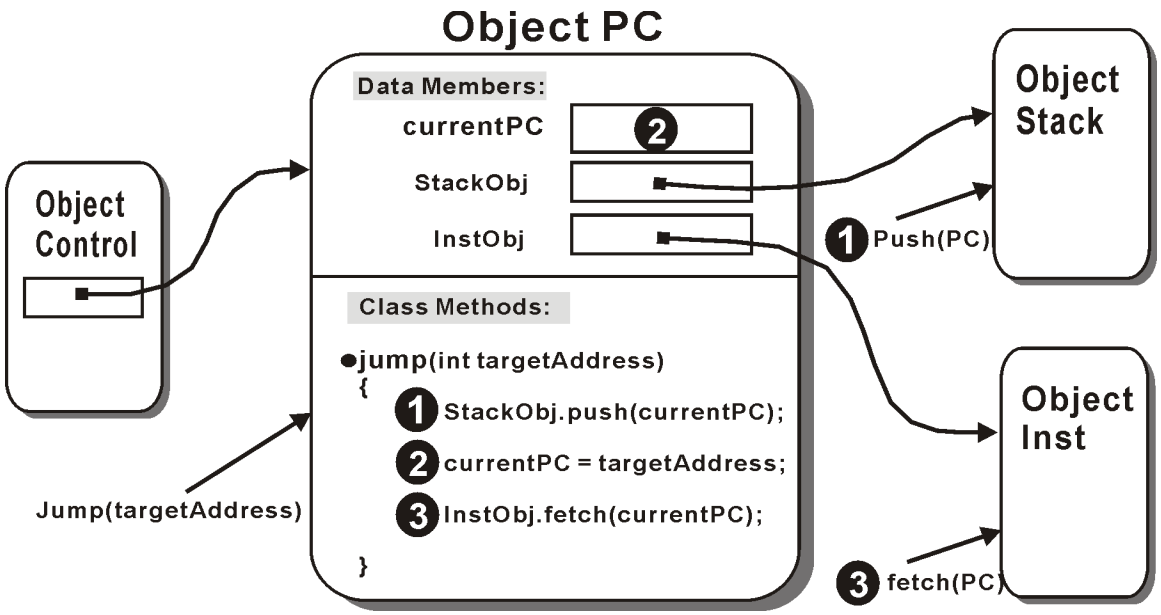


Figure 4.4: Connecting and simulating the objects in the UTDSP model

send the messages to each of the references. Figure 4.4 shows an example of how the objects are connected according to their signal flow and how the simulation is performed by message sending.

In this example, the **Control** object will send the *jump(targetAddress)* message to the **PC** when a jump instruction is executed. Upon receiving the *jump()* message, the **PC** invokes method *jump()* and executes the following three steps: First, the **PC** sends the *push(currentPC)* message to the **Stack** object where the current program counter (currentPC) is pushed onto its internal stack. Second, the program counter is updated with *targetAddress*. Third, the **PC** sends the *fetch(currentPC)* message to the **Inst** object where instruction fetching is performed.

This example shows that the hardware modelling methods we use not only reduce the complexity of the software construction, but also provide a well-defined system partition and block interfaces that subsequent RTL modelling can adopt. Being constructed in the object model, the UTDSP architecture simulator can easily collect run-time data for each hardware object and help to evaluate various design decisions early in the design phase. This capability is extremely important because it would otherwise be too expensive to wait until the design has been modeled at the RTL level.

4.3 The GUI-Based Assembly Language Debugger

An assembly language debugger is a front-end program that provides the user interface and the functionality of an instruction set simulator. The assembly language debugger provides programmers a tool to optimize DSP kernels. Because the major portion of the execution time for a DSP application is usually spent in its inner loop code, optimizing the inner-loop assembly code directly increases the overall performance of the application.

4.3.1 The Features of the Assembly Language Debugger

The UTDSP assembly language debugger provides a true graphical user interface with a set of powerful debugging features. Figure 4.5 shows the UTDSP debugger. The features provided include the following: First, the debugger can perform single-step or multi-step execution by specifying the number of steps to execute. Second, the debugger can highlight the hardware resources — instructions, memory locations, functional units, and IO ports — that are currently accessed. This feature is especially useful when programmers debug their assembly programs — they can use single-step execution to trace and observe the memory locations and hardware components that are highlighted in different colors to verify if there is any memory location or component that is incorrectly accessed.

Third, the debugger allows programmers to set up breakpoints by simply clicking on the memory locations, register file entries, or instructions that are to be monitored. Simulation will halt when a breakpoint is reached.

4.3.2 Adding Self-Displaying and Event-Listening Abilities

The debugger was implemented by adding a self-displaying and event-listening capabilities to the hardware objects of the UTDSP architecture simulator. The inconsistency problem between the debugger and the architecture simulator never exists because they use the same set of hardware objects. In other words, the debugger need not be modified when there is a change made to the architecture simulator. This capability is especially useful for core-based designers because they will have a functionally equivalent debugger ready for use after having modified the UTDSP simulator. Being implemented in Java, the self-displaying and event-listening features can be easily incorporated into the UTDSP objects using the Java Abstract Window Toolkit (AWT) [21]. The problem of displaying objects demonstrates the extensibility and maintenance advantages of

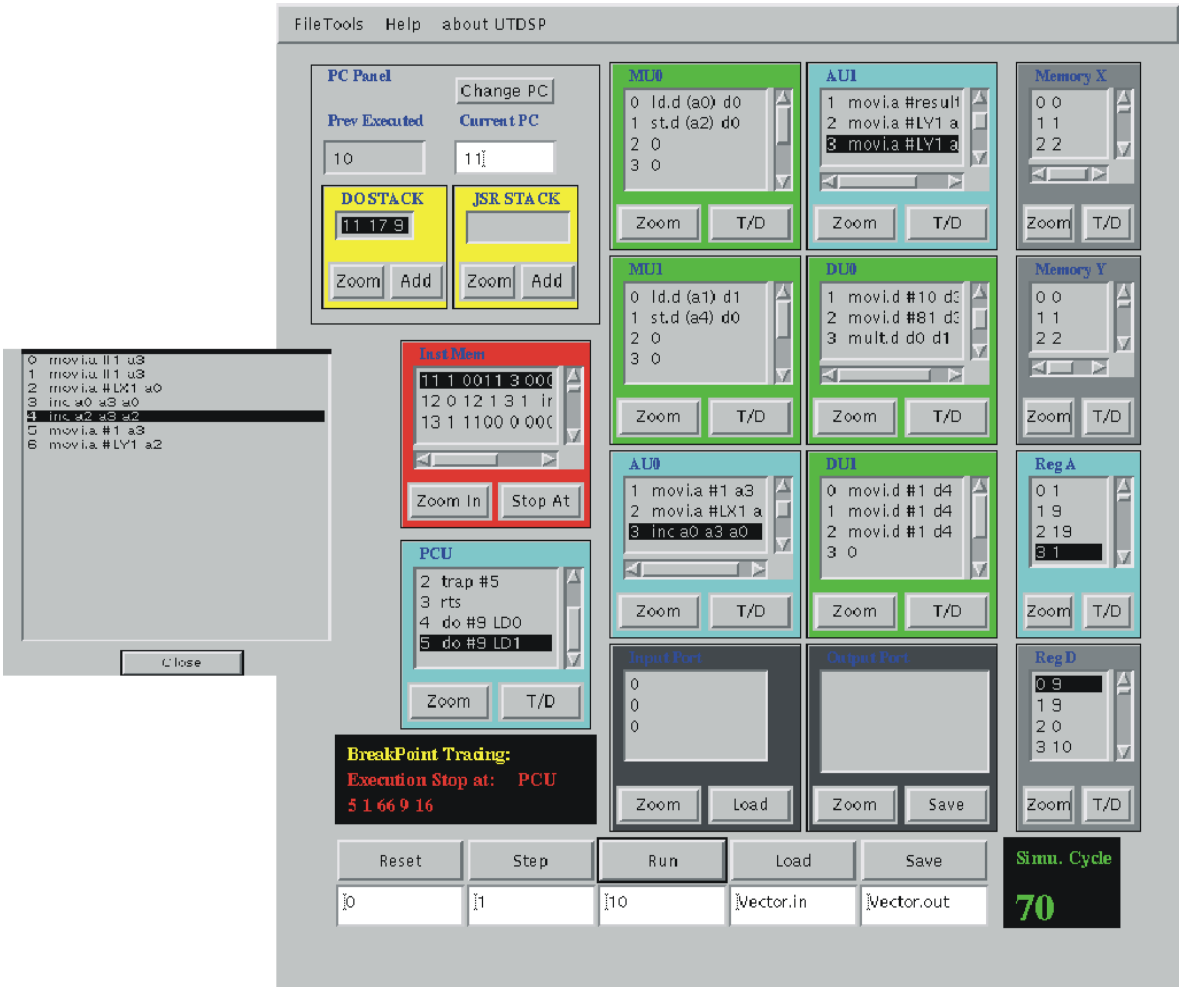


Figure 4.5: The UTDSP assembly language debugger

the Java object model. Java AWT has an OO solution (not surprisingly) to this problem: it requires that every displayable object respond to the paint() message where the object describes how to draw itself. Similarly, responding to the event-listening messages makes an object be able to handle the mouse events. Figure 4.6 shows an example of how to equip a UTDSP object with the self-displaying and event-listening abilities. This example shows the modifications made to the **Inst** object so that it can display its memory contents and allow users to set up breakpoints by clicking on the memory locations that are to be monitored.

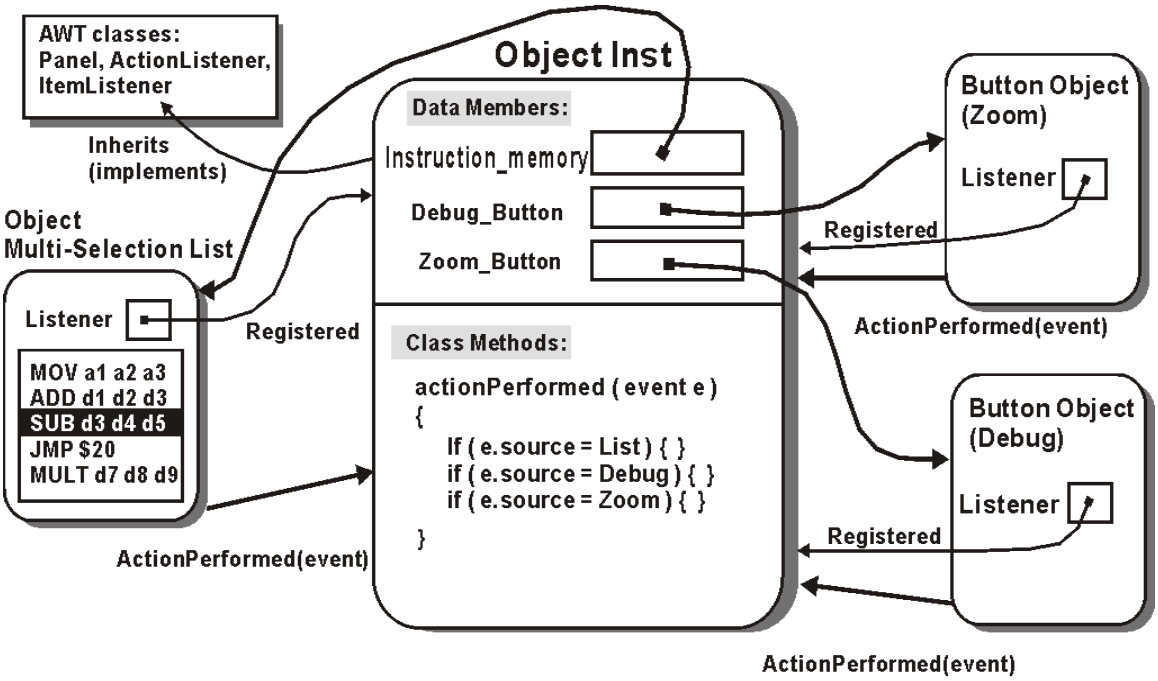


Figure 4.6: The Inst object with self-displaying and event-listening abilities

The new **Inst** class inherits from (implements) three AWT classes: the **Panel** class, the **ActionListener** class, and the **ItemListener** class. Inheriting from the **Panel** class entitles the **Inst** class to override the *paint()* method in which the steps of how to draw the **Inst** object are described. Moreover, being derived from the **Panel** class, the **Inst** object becomes a container that can contain the objects instantiated from any AWT class, such as the **Button** class and the multiple-selection **List** class. As shown in Figure 4.6, two **Button**s and one multiple-selection **List** are added into the **Inst** object. The multiple-selection **List** is used to store the instructions to make them clickable, so that a breakpoint attribute can be added to a instruction when it is clicked.

Similarly, Inheriting from the **ActionListener** class entitles the **Inst** object to override the event-handling methods in which the events generated from the **Button**s are handled. Clicking on a **Button** will generate an event object that contains information about its source. The **Button** then finds all the objects that are registered in its Listener and sends the *actionPerformed(event)* message to those registered objects. In the case shown in Figure 4.6, the **Inst** object is registered in the Listener of the **Debug Button** object; therefore, when clicked, the **Debug** button gener-

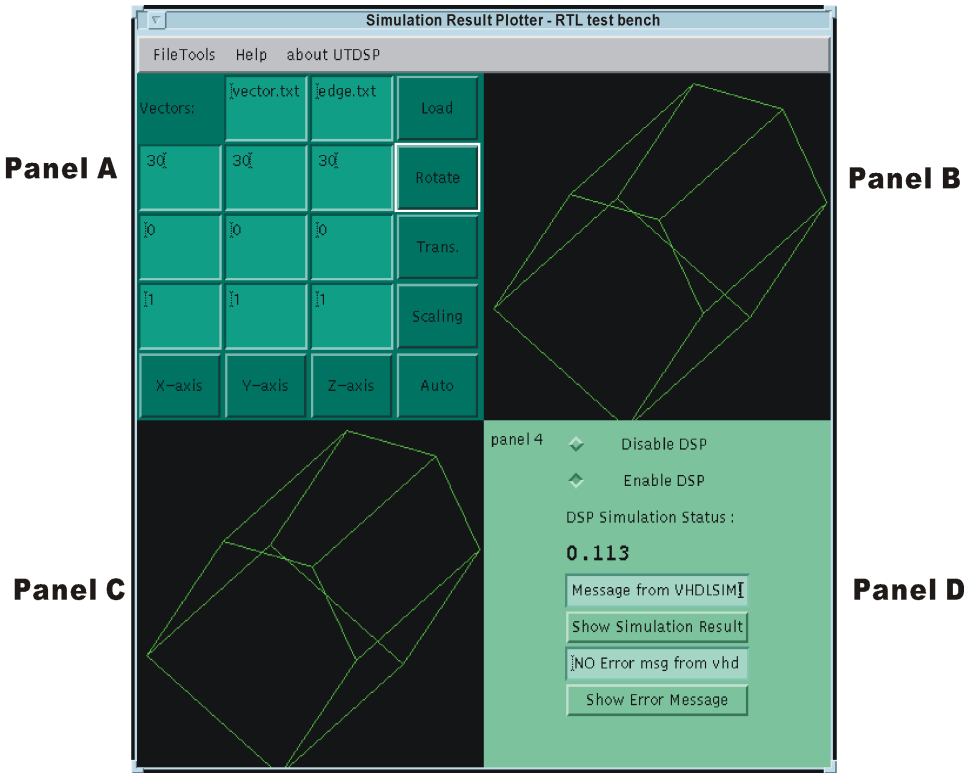


Figure 4.7: A simulation result plotter and a test bench for RTL model

ates an event and sends the *actionPerformed(event)* message to **Inst**. The **Inst** object then handles the event using its corresponding methods.

Sometimes it is necessary to visualize the simulation output from the debugger or architecture simulator. Using Java AWT eases the implementation of a simulation output plotter that can be incorporated into the assembly debugger or architecture simulator. Figure 4.7 shows a simulation output plotter that was constructed to visualize the output of a 3D graphics application. This plotter, which consists of four replaceable module panels, serves not only as a simulation result displayer but also a test bench that automates the tedious steps in the functional verification phase.

In the testbench mode, users specify input test vectors from Panel A and the RTL simulation is then launched using the Synopsys VSS simulator. Panel D shows the RTL simulation status and error messages, if any, generated from the Synopsys simulator. The output vectors from the RTL simulation will be sent to Panel B where the output vectors are plotted according to the algorithms defined in the panel. Panel C displays the results calculated using a functionally equivalent, floating-point Java program for the DSP application under test. Like commercially available virtual

instrumentation products [23][16], this platform is flexible in that each module panel is independent and can be replaced by other panels that are designed for different DSP applications.

4.4 Summary

This chapter describes the design and implementation of the UTDSP architecture simulator and the GUI-based assembly language debugger. The Object-Oriented method used not only bridges the design gap between the behaviour and RTL models of the UTDSP but also eases the development of the assembly debugger. Being implemented in Java, the assembly debugger can be easily constructed by adding self-displaying and event-listening capabilities to the objects in the architecture simulator. The next chapter will focus on the VLSI implementation of the UTDSP. The CAD methodology used and benchmark results will be also illustrated.

Chapter 5

System Design and VLSI Implementation of the UTDSP

This chapter focuses on the hardware design and VLSI implementation of the UTDSP. There are four major parts in this chapter: First, Sections 5.1 - 5.3 show the architecture of the UTDSP by giving an overview of its hardware resources, instruction set, and pipeline architecture. Second, Sections 5.4 - 5.6 discuss the hardware design of some important blocks in the UTDSP. Third, Section 5.7 describes a novel CAD methodology and design flow that is used to realize the chip. Fourth, Section 5.8 shows benchmark results and compares the UTDSP with two commercially available DSP processors that also use VLIW architectures.

5.1 The UTDSP Hardware Architecture

Figure 5.1 shows the hardware blocks of the UTDSP. The UTDSP has a RISC-like architecture with five pipeline stages. They are: IF-1, Instruction fetch stage 1; IF-2, Instruction fetch stage 2; ID, Instruction decode and register fetch; EX, Execution of the ALU and memory access instructions; and WB, Write the result into the destination registers.

The UTDSP hardware can be divided into six major sections: the PC Unit, the Instruction Memory, the Decoder Memory, the Register Files, the Execute Units, and the Controller Unit. The PC Unit generates instruction addresses, computes the destinations of branches, and handles the necessary stack operations in loop instructions. During the IF-1 stage, the address generated from the PC Unit is used to fetch the uni-op instructions or multi-op pointers stored in the Instruction Memory. During the IF-2 stage, a multi-op pointer fetched in the IF-1 stage is used to fetch the actual long instructions from the Decoder Memory, which consists of two clusters (Cluster A and Cluster B). In contrast, uni-op instructions fetched in the IF-1 stage are directly passed to the appropriate function units.

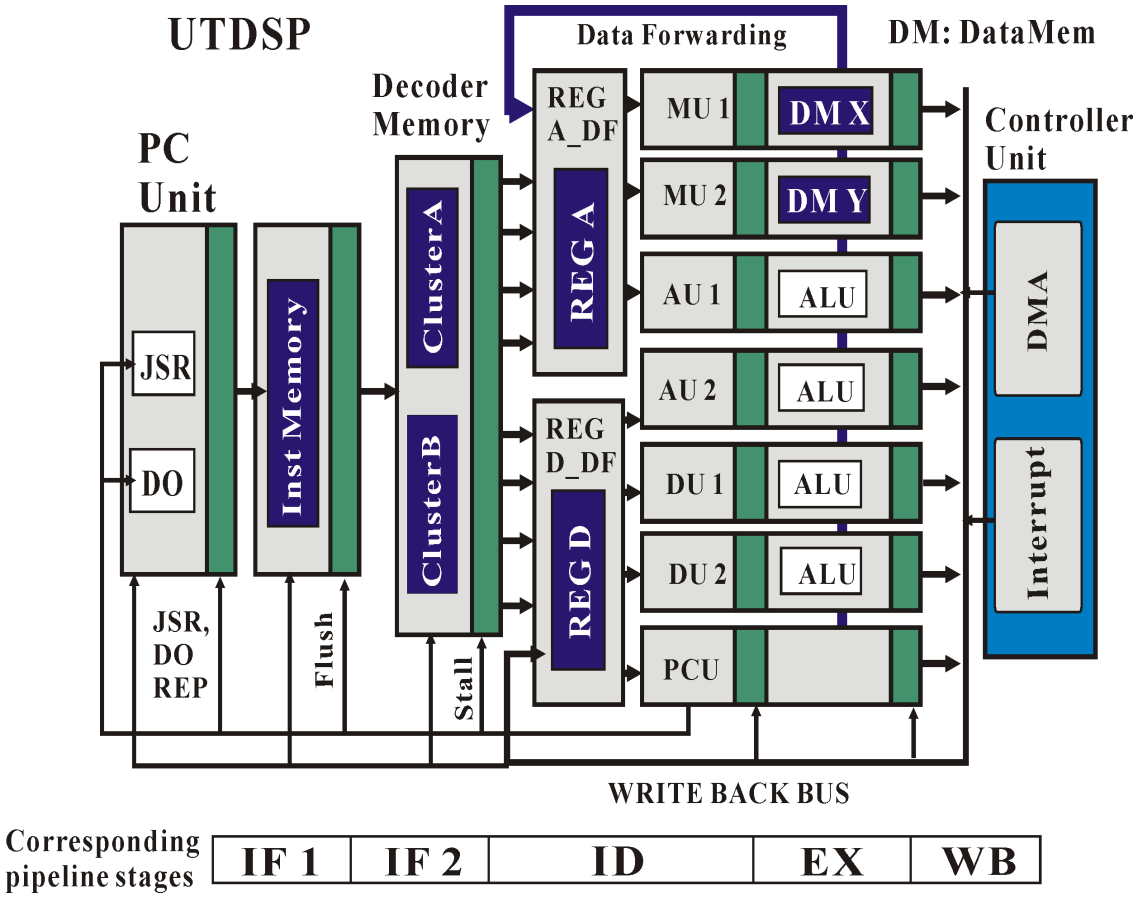


Figure 5.1: The UTDSP hardware blocks

In the ID stage, the operations in a long instruction are decoded in their corresponding execution units. There are seven execution units (functional units) in the UTDSP: MU1 and MU2 execute load and store instructions; AU1 and AU2 execute address instructions; DU1 and DU2 execute integer ALU instructions; and PCU executes branch and control instructions. The required operands for an instruction are loaded into the ID/EX pipeline registers from the two register files (REG A and REG D) during the ID stage. REG A has 16 registers used for address operations, while REG D has 16 registers for integer calculations.

In the EX stage, AUs and DUs execute instructions in their ALUs, while MUs perform load and store operations to data memory banks. There are two data memory banks (DataMem X and DataMem Y) in the UTDSP, each of which has its own independent address and data buses. MU1 is associated with DataMem X, while MU2 is associated with DataMem Y. The results obtained in

the EX stage will be written back to the Register Files during the WB stage. Finally, DMA and interrupt requests are handled by the Controller Unit, which can control the operations of the pipeline registers in the UTDSP.

5.2 Instruction Set

The UTDSP instruction set was originally proposed by Mazen Saghir [2] and was then modified by the author to reduce the number of ports on the Register Files. Several additional instructions were introduced to ease fixed-point calculations and to improve machine performance. There are 69 instructions in total, which are shown in Appendix A. Each execution unit can execute only a subset of the UTDSP instructions. The instructions can be categorized into four groups according to their associated execution units:

- Memory instructions: The UTDSP is a load-store architecture, meaning the only memory instructions are explicit loads and stores. Memory instructions can be executed only in memory units MU1 and MU2. MU1 is associated with DataMem X, while MU2 is associated with DataMem Y.

- Addressing Instructions: The addressing instructions operate on address registers, and include special instructions that use modulo- and bit-reversed addressing. The addressing instructions are executed in address units AU1 and AU2.

- Integer instructions: In addition to a common set of arithmetic and logical instructions, the integer instructions include multiply-accumulate (MAC) instructions, which are heavily used in DSP algorithms. Moreover, multiplication and MAC instructions that use the 1.15 fixed-point format were also introduced to minimize the number of shift instructions required in fixed-point calculations. The 1.15 fixed-point format will be described in Section 5.5.1, where datapaths are discussed. All integer instructions operate on integer registers, and are executed in integer units DU1 and DU2.

- Control instructions: In addition to a set of instructions that change control flow, control instructions include zero-overhead looping instructions that cause a single instruction or a block of operations to be repeatedly executed for a specified number of iterations with no branch penal-

ties. Moreover, the UTDSP hardware is designed to be able to handle nested looping instructions with up to five levels. Control instructions are also responsible for moving data between the integer register file and the address register file. DMA operations are also included in the control instructions.

5.3 The Pipeline Architecture

As shown in Figure 5.1, the UTDSP has a five-stage pipeline architecture, which is slightly different from that of the standard RISC processor described by Patterson & Hennessey [24]. The UTDSP eliminates the MEM stage used in the RISC architecture. Instead, memory accesses are performed in the two memory units (MU1 and MU2) during the EX stage. To not increase the pipeline latency, MUs support only the register-indirect addressing mode for load and store instructions. In other words, MUs can initiate a memory access from the very beginning of the EX stage without having to calculate the target address in the EX stage. Figure 5.2 shows the difference between a typical RISC and the UTDSP pipelines.

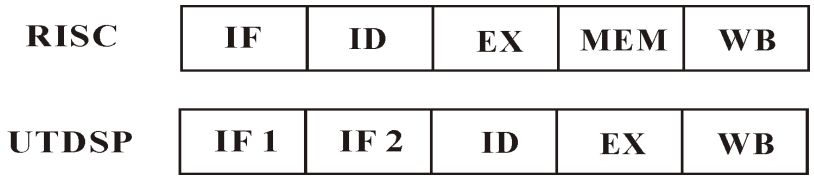


Figure 5.2: The pipeline architectures of RISC and the UTDSP

There are three type of hazards that can occur as a result of pipelining: structural hazards, data hazards, and control hazards. Structural hazards occur when there are resource conflicts. Data hazards occur when there is a data dependence between instructions. Control hazards occur as a result of branch instructions. The UTDSP eliminates the data hazards by using bypassing hardware and minimizes the control hazards by introducing zero-overhead looping instructions.

5.3.1 Data Hazards and Bypassing

The data hazards that can occur in the UTDSP pipeline are the read after write (RAW) data hazards. Consider the instruction sequence shown in Figure 5.3. The load instruction will not update

register D1 until the WB stage in cycle 5, while the add instruction reads register D1 during the ID stage in cycle 4. As a result, the add instruction will read an incorrect operand value from D1, causing a RAW hazard. To solve this problem without stalling the pipeline, a bypassing path is added to forward the correct result for D1 from pipeline register EX/WB to the EX stage. Figure 5.3 also shows the bypassing path.

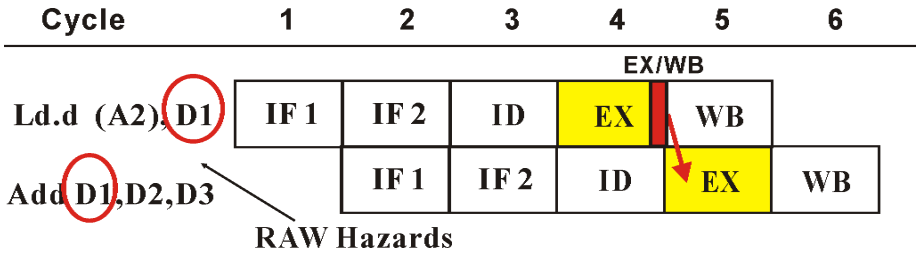


Figure 5.3: RAW hazards and bypassing path

In a VLIW processor with *n* functional units, bypassing becomes a costly function to implement because the area complexity of the comparators and bypassing buses required is $O(dn^2)$, where *d* is the number of pipeline stages between ID and WB [25]. The UTDSP pipeline eliminates the MEM stage by using register indirect mode as the only addressing mode for load and store instructions. This pipeline design provides several advantages in VLIW architectures: First, the frequency of RAW hazards will decrease and all RAW hazards can be solved by using the bypassing technique. Second, the number of comparators and bypassing buses required is reduced by 50%. The trade-off is that the UTDSP compiler will have to schedule an extra add operation before load/store operations if the displacement addressing mode is required; however, the effect on execution time is very small because this extra instruction can be usually scheduled as part of a previous long instruction.

5.3.2 Control Hazards and Zero-Overhead Looping Instructions

The UTDSP pipeline has a branch penalty of two cycles. In a branch instruction, the result of its comparison is not known until the end of the ID stage. By this time, two instructions have been fetched before the result of the branch can take effect. The UTDSP statically predicts that the branch will not be taken. If the branch is taken, the two instructions in the IF 1 and IF 2 stages

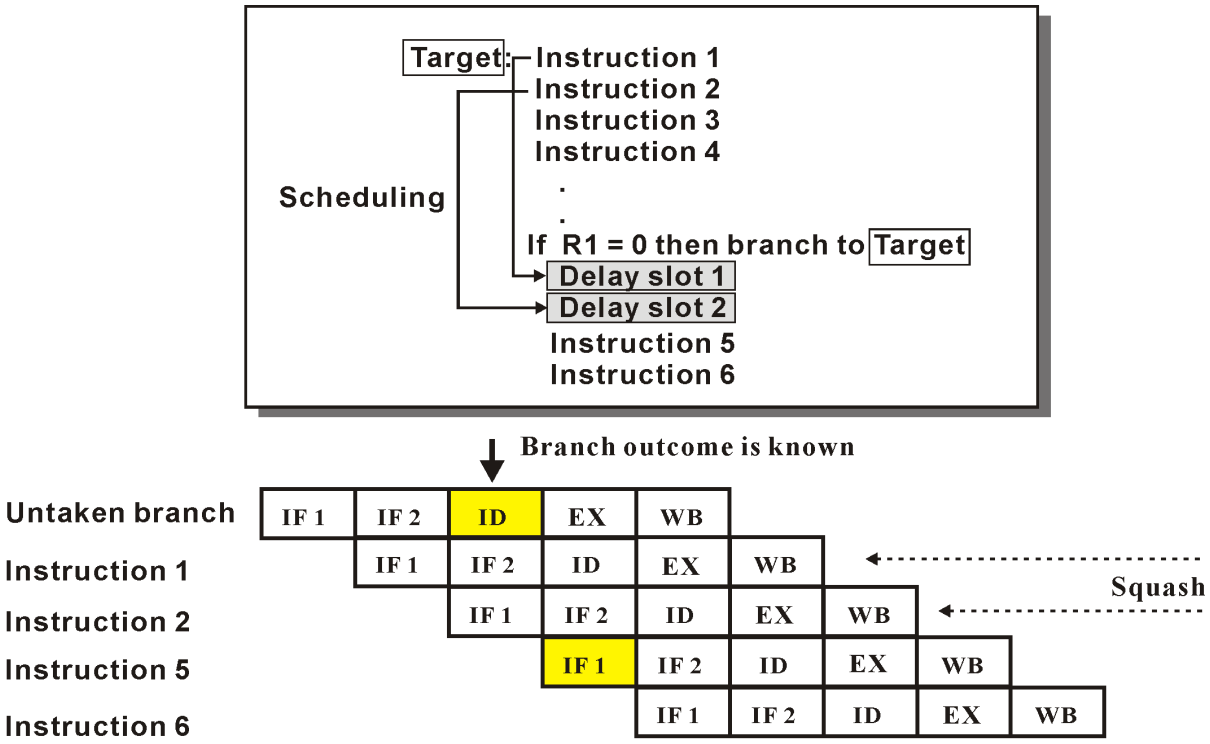


Figure 5.4: The UTDSP instruction pipeline when predict-taken scheme is used

(two branch-delay slots) are squashed or cancelled. An alternative scheme is to predict every branch as taken. In this scheme, the UTDSP compiler has to schedule the two branch-delay slots from the target of the branch. If the branch is not taken, the two instructions in the delay slots are squashed. Figure 5.4 shows the predict-taken scheduling scheme and the behaviour of an untaken branch in the pipeline. Although statistics show that 67% of the conditional branches are taken on average [38], the UTDSP uses the predict-untaken scheme because the current UTDSP compiler is unable to schedule the delay slots.

DSP algorithms usually consist of loop-intensive kernels such as FIR, IIR, FFT, and matrix multiplication. Using jump and branch instructions for the looping control will impose a great branch penalty and thus significantly degrade performance. The UTDSP provides two types of zero-overhead hardware loops that solve this problem: single-instruction loops and multi-instruction loops. The loops can be nested up to five levels. Figure 5.5 shows an example of the UTDSP

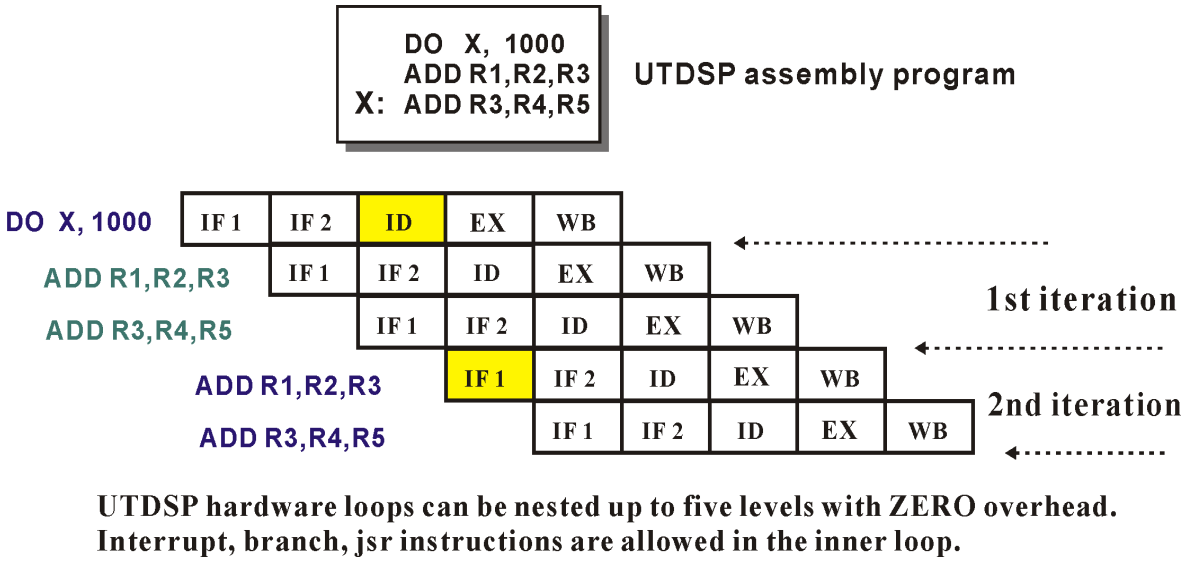


Figure 5.5: The UTDSP zero-overhead hardware loop

hardware loops and its corresponding instruction sequences in the pipeline. Observe that in this example there is no branch overhead between the first and second loop iterations. Handling interrupts, branch, and jump to subroutine (JSR) instructions in nested loops presents a challenge in the design of the PC unit. The details will be described in Section 5.4, where the PC unit is discussed.

5.3.3 Interrupt Effects

The UTDSP provides three user-defined interrupt vectors. When an interrupt occurs, the UTDSP allows instructions in the pipeline to finish executing. The processor then begins fetching from the interrupt vector associated with the interrupt. Each interrupt vector may start with a JSR instruction that transfers the control flow to its interrupt service routine provided by programmers. Figure 5.6 shows the effect of an interrupt on the pipeline operations.

When an interrupt occurs in cycle three, the UTDSP will fetch the first instruction (JSR V1) in the corresponding interrupt vector in cycle 4. The first instruction of the interrupt service routine (V1) associated with the interrupt will be fetched in cycle 7. This mechanism results in a loss of two instruction cycles. The instruction that semantically follows instruction I3 will be fetched on return from the interrupt service routine. The UTDSP also provides a fast interrupt if the size of a user-defined interrupt service routine is small enough to be fitted into an interrupt vector (10 long

instruction words). In the fast interrupt mode, programmers replace instruction JSR V1 with instruction V1 — the first instruction in the interrupt service routine. This method eliminates the two cycles of overhead.

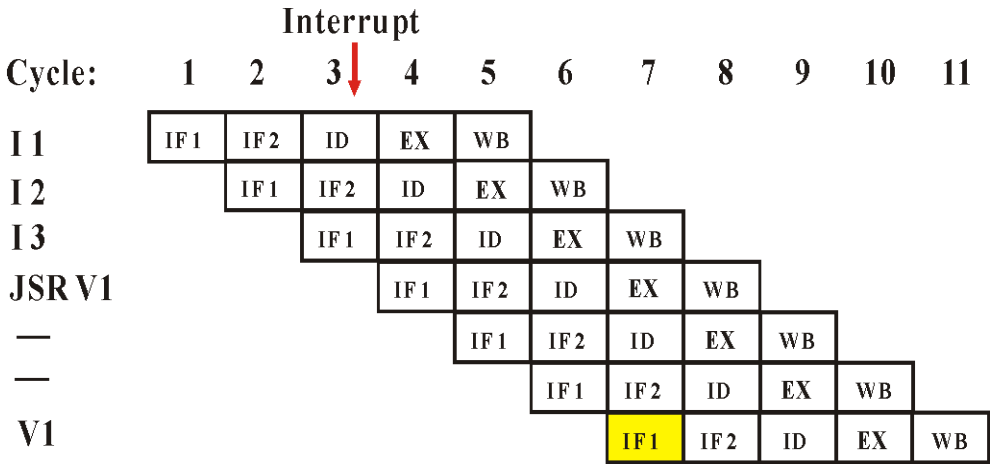


Figure 5.6: The UTDSP instruction pipeline when handling an interrupt

5.4 The PC Unit

The PC Unit handles the addressing of instructions and saving instruction addresses so that the machine can be restarted after an interrupt. The PC Unit presents one of the major challenges in the UTDSP design because it not only handles the traditional tasks that are usually seen in RISC processors but also provides the zero-overhead hardware loops that are completely nestable and interruptable. Figure 5.7 shows the block diagram of the PC Unit. For simplicity, the diagram shows only some important signals.

There are six major blocks in the PC Unit: Incrementer, PC Register, PC Controller, DO Stack, JSR Stack, and Counter. The Incrementer computes the value of the PC bus incremented by one. The PC Register stores the current PC and can flush or stall the PC bus according to the flush/stall signals. The DO Stack stores the loop beginning and end addresses and repetition count. The JSR Stack stores the return addresses for JSR instructions. The PC Controller receives the decoding results of the instructions in the ID stage, determines the next PC value, and controls the operations of the two stacks. The Counter keeps track of the current iteration count for hardware loops.

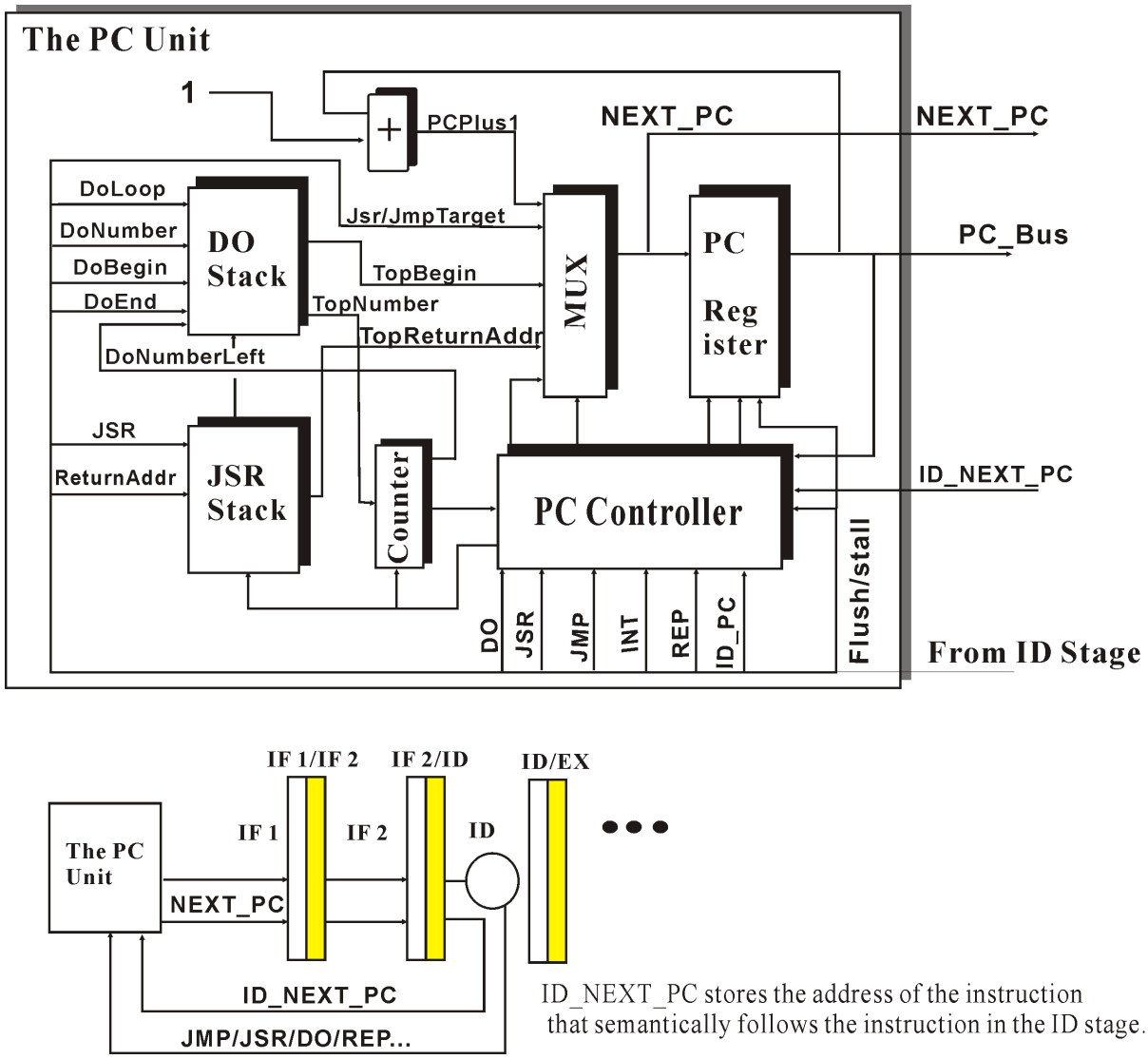


Figure 5.7: The block diagram of the PC Unit

The PC Controller plays an important role in handling the nestable, zero-overhead hardware loops. Table 5.1 describes the logic in the PC Controller that determines the NEXT_PC value for instruction fetching in the next cycle.

As shown in Table 5.1, if the instruction in the ID stage is JMP or JSR, the NEXT_PC will be set to its branch destination address (Jmp/JsrTarget). If the instruction is a subroutine return call (RTS), the top entry in the JSR Stack (TopReturnAddr) is assigned to the NEXT_PC. Similarly, when the single-instruction loop (REP) is encountered, the NEXT_PC will be changed to the address of the instruction that semantically follows that loop instruction (ID_NEXT_PC). The

nestable, zero-overhead DO loops are handled by popping the top entry (TopBegin) in the DO Stack and assigning that entry to the NEXT_PC when the end address (TopEnd) of the current loop body is encountered. Finally, when an interrupt occurs, the NEXT_PC will be assigned to its corresponding interrupt vector.

Conditions	NEXT_PC
(JMP = '1') or (JSR = '1')	Jmp/JsrTarget
RTS = '1'	TopReturnAddr
REP = '1'	ID_NEXT_PC
NeedToRepeat	PC
(DO = '1') and (PC = DoEnd)	ID_NEXT_PC
(PC = TopEnd) and (DoNumberLeft <> 0) and (DoStack is NOT empty)	TopBegin
INT = '1'	Interrupt_Vector
None of above is true	PCPlus1

Table 5.1: The value of NEXT_PC and its associated conditions

5.5 The Register Files

Like other VLIW architectures, the UTDSP suffers from the large number of ports on its register files. The original instruction set of the UTDSP as specified in [2] requires 11 read and 5 write ports on the address register file (REG A), while requiring 9 read and 5 write ports on the integer register file (REG D). We proposed a method to implement these multi-ported register files using dual-ported SRAM macros. This method incorporates the ideas used in two VLIW processors: CYDRA-5 [40] and LIFE [39].

In the CYDRA-5 processor, multiple identical register files are used to reduce the number of read ports on its register file. In contrast, the LIFE processor reduces the number of write ports on its register file by using time-multiplexing, allowing only one functional unit to write into the register file at a given time. By combining these two methods, we proposed an architecture that uses dual-ported SRAM macros to construct the multi-ported register files because the dual-ported macros are usually available in memory compilers. Figure 5.8 shows the proposed architecture.

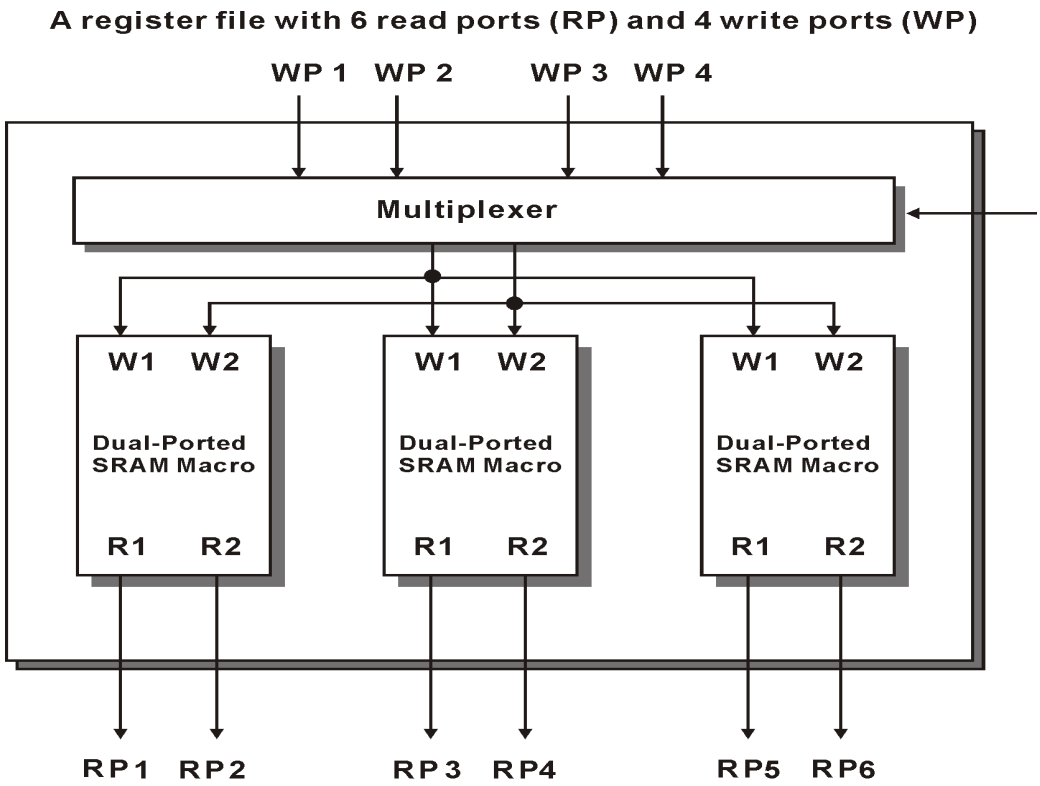


Figure 5.8: Constructing a register file with 6 read and 4 write ports using dual-ported SRAM macros

However, this architecture requires that the number of read and write ports be a multiple of two. To fulfill this requirement, we used the UTDSP architecture simulator to evaluate various modification decisions to reduce the impact on performance. Finally, several modifications were made to reduce the number of ports and fulfill the requirement, while imposing a zero penalty on the performance of the UTDSP kernel benchmarks.

First, being seldom used, the memory instructions that load to or store from REG A were removed from the UTDSP instruction set, reducing the number of read ports on REG A by two (2 address functional units). Table 5.2 lists the removed instructions. Second, the format of the modulo address instructions was changed, reducing the number of read ports on REG D by two. More details about the modulo addressing mode will be discussed in Section 5.6.2. Third, let the PCU share its read and write ports with MU 2, reducing one read and one write port on both REG A and REG D. This port sharing method was used because of the following two reasons:

Instructions	Syntax	Functional Description
Load address register	ld.a (ai), aj	aj = Memory[ai]
Store address register	st.a (ai), aj	Memory[ai] = aj

Table 5.2: The two instructions that are removed from the original UTDSP instruction set

First, most of the loop structures in the UTDSP kernel benchmarks take advantage of the zero-overhead *do* loop instructions with known loop iterations at compile time, eliminating the necessity of reading the loop iterations from the register files. Second, even in some of the benchmarks (like FFT) where the number of inner loop iterations has to be changed at run time, occupying a read port by the *do* instruction will affect neither the existing scheduling nor the execution time of the inner loop code because the *do* instruction is not part of its following inner loop body that is to be repeated.

As a result, REG A now has only 6 read and 4 write ports, while REG D has 8 read and 4 write ports. Moreover, the reduction in the number of ports on REG A and REG D will not affect the performance of the UTDSP kernel benchmarks.

5.6 The Datapath Components

The UTDSP adopts a synthesis-based design methodology where all datapath components are described in VHDL and thus are fully parameterizable. The current implementation has a 16-bit, fixed-point datapath. There are two integer functional units (DU1 and DU2) in the UTDSP, each of which has an integer ALU and an accumulator. Each integer ALU has a 16-bit multiplier, a shifter, and an adder/subtractor. Similarly, the two address functional units have their own address ALUs and each address ALU has a shifter and a modulo address generator. Section 5.6.1 will briefly describe the fixed-point formats that are used in the multipliers. Section 5.6.2 shows the modulo address generator used in the UTDSP.

5.6.1 The 1.15 Fixed-Point Format

The UTDSP provides two types of multiplication and multiply-accumulate instructions: the integer type and the 1.15 fixed-point type. Figure 5.9 distinguishes the differences between these two types. When multiplying two fractional values in the 1.15 format, the result needs to be

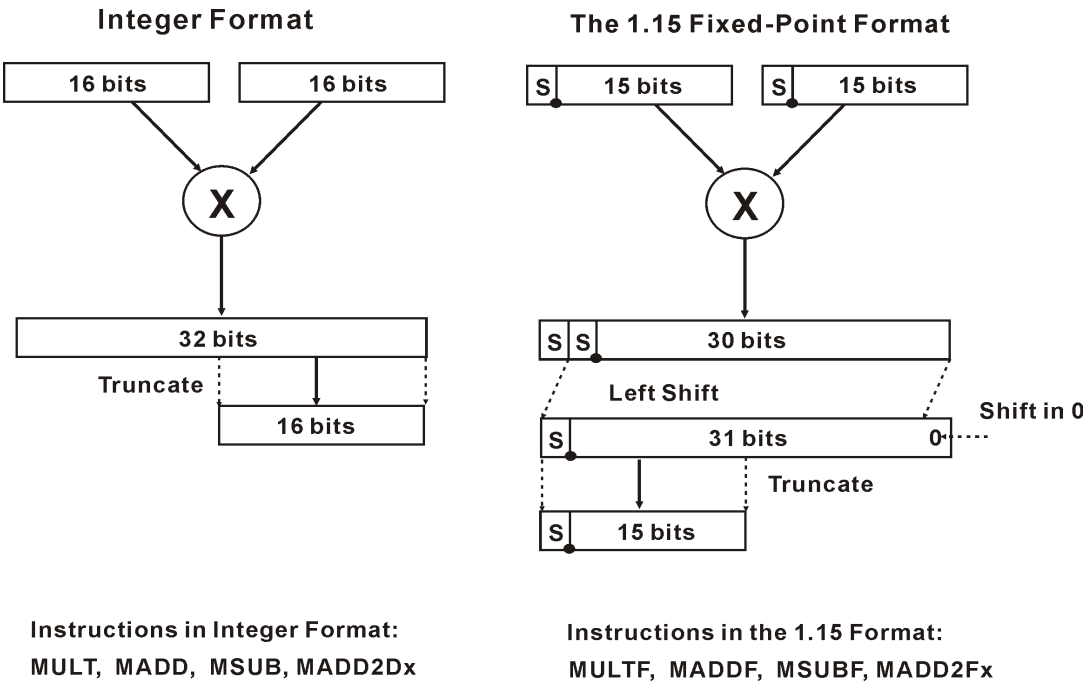


Figure 5.9: The data formats that are used in the UTDSP instructions

upshifted one bit to obtain the normalized data, which means that one extra shift operation will be needed for every multiplication instruction. To solve this problem, the UTDSP provides a set of instructions that are optimized for the 1.15 format, eliminating the need for extra shift instructions. Figure 5.9 also lists the instructions of the two formats.

5.6.2 The Modulo Address Generator

Many DSPs have a modulo addressing mode, which eliminates the need for checking the array pointer in a buffer to see if it has reached the boundary of the buffer and automatically circulates the pointer back to its valid starting position when it is out of bounds. Typically there are two methods for implementing the modulo addressing mode. In the first method, the modulo address generator uses a modifier register that contains only the length of the buffer and calculates the starting address of the circular buffer using hardware. However, this implementation method has to restrict the starting address and the size of the buffer [26][27].

In the second method, the modulo address generator uses *start* and *end* registers to hold the start and end addresses for each circular buffer. This method enables a true modulo addressing by allowing arbitrary buffer sizes and displacements. Research shows that this method is desirable

for next-generation DSPs [28]. Processors using the approach include the Lucent DSP16xx [29] and the TI TMS320C5x [30]. The UTDSP adopts the second method for two reasons: First, the second method simplifies the hardware design and significantly reduces the resulting area. Second, this method enables a modulo instruction that uses fewer read ports of the address register file, which is an important concern in VLIW architectures. Figure 5.10 shows the resulting instruction formats for the two different methods. Each address functional unit has a circular buffer whose *start* and *end* registers can be changed using the set instruction as shown in Figure 5.10.

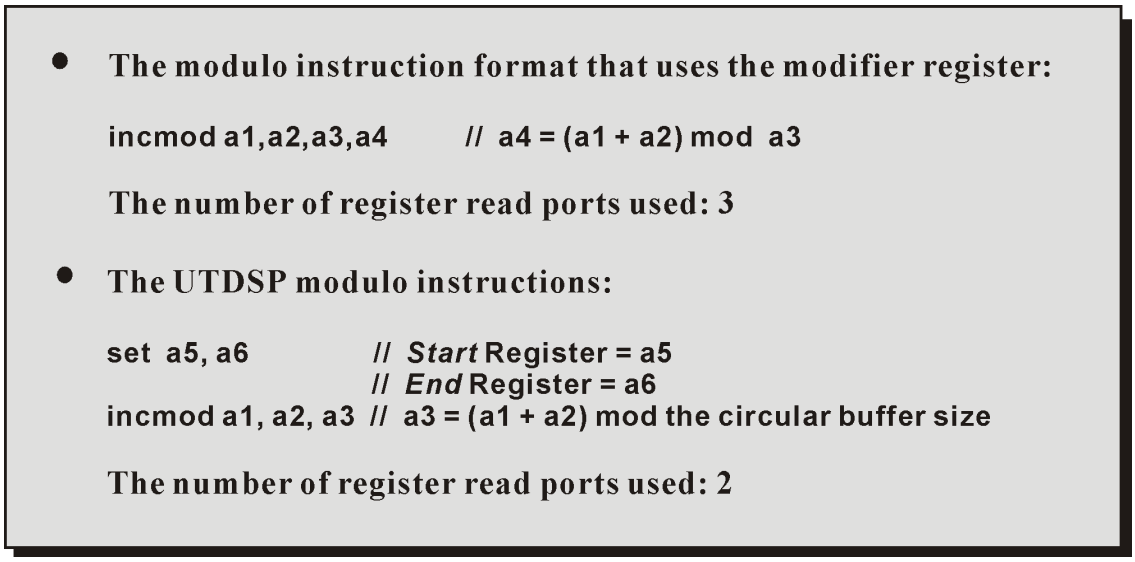


Figure 5.10: The UTDSP modulo instruction format vs. the typical format

5.7 VLSI Implementation Issues

The VLSI implementation of the UTDSP presents a major challenge in both design capture and CAD methodology. It needs not only great efforts in VHDL modelling but also a state-of-the-art CAD methodology that can provide the required features for back-end flow. Research has shown that the interconnect delays in deep-sub-micron (DSM) designs will actually decrease at the 50,000-gate module level as feature sizes shrink [31]. However, partitioning a huge chip into 50,000-gate blocks is not a trivial task. To solve this new partitioning problem, a flexible, hierar-

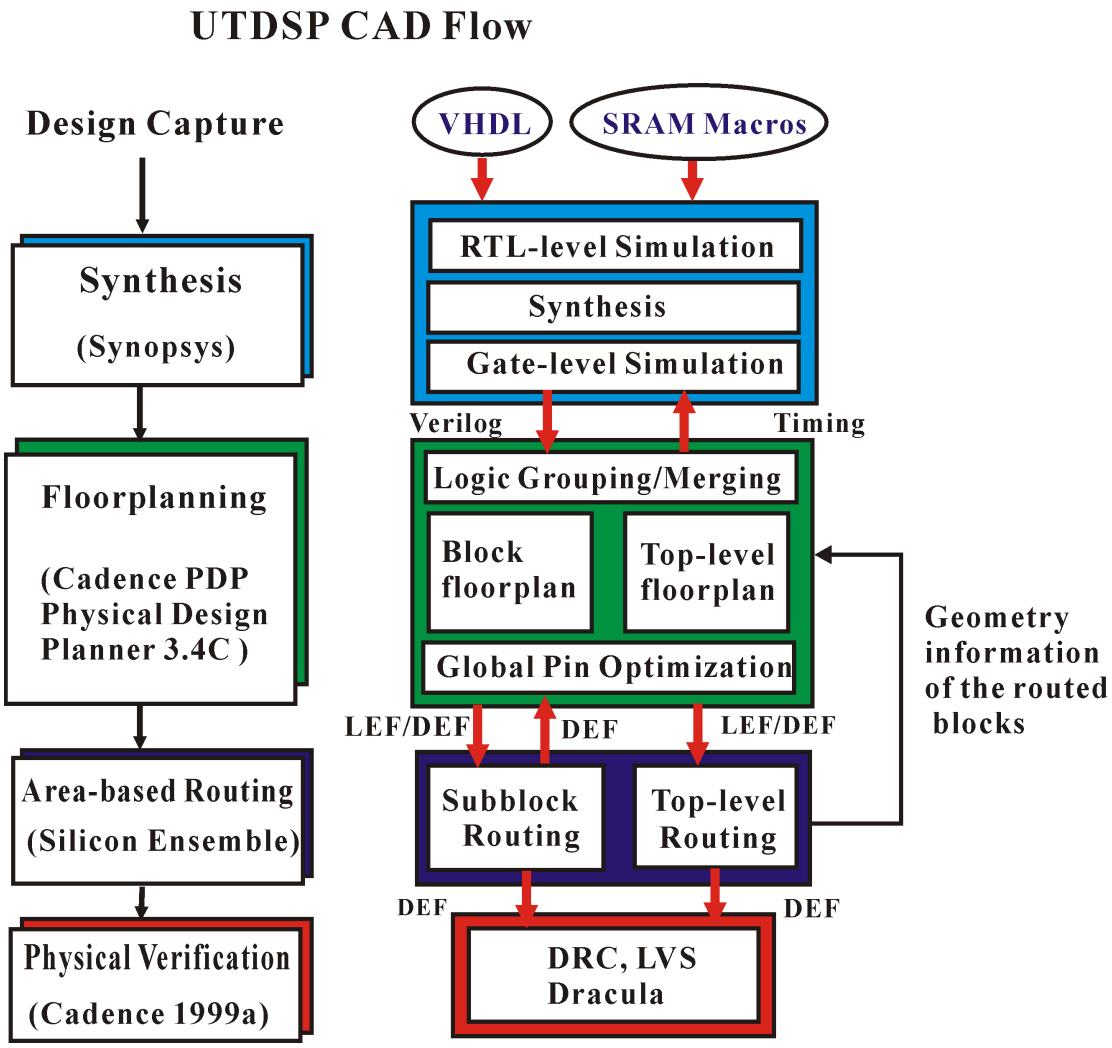


Figure 5.11: The UTDSP CAD methodology

chical CAD flow was defined that not only creates the blocks of the desired size but also significantly reduces the delay and area of the resulting UTDSP chip. Figure 5.11 shows this hierarchical design flow. The following sections will discuss each phase of this flow.

5.7.1 Design Capture and Synthesis

The technology used for the UTDSP is a 0.35 µm CMOS with three metal layers from TSMC [41]. The library cells were also from TSMC. The single-ported SRAM macros that were developed by CMC [32] were used to implement the memory blocks in the UTDSP. The Instruction Memory block consists of one wsramsp256x32 macro — an SRAM macro that has 256 words

with 32 bits per word. Each Decoder Memory bank also consists of one wsramsp256x32 macro. There are two Data Memory blocks in the UTDSP, each of which consists of four wsramsp256x8 macros — 1 Kbyte in each Data Memory Bank.

The remaining parts of the UTDSP are described using register-transfer-level (RTL) VHDL code. Because CMC was unable to release the dual-ported SRAM macros or an SRAM compiler with a multi-port capability, the register files in the UTDSP were synthesized, increasing their areas significantly. The area penalty of using synthesized register files will be discussed later. The multiplier and adder/subtractor were implemented using the DesignWare library provided by Synopsys [33]. The total VHDL code including testbench has 10,100 lines. RTL simulation was performed to verify the functional correctness of the RTL model.

In the logic synthesis phase, the RTL model was translated into a functionally-equivalent gate-level netlist under user-specified constraints such as timing and area. The critical path components, such as ALUs, were synthesized using strict timing constraints to reduce the critical path delay. In contrast, the other components of the UTDSP were synthesized using area constraints to reduce their areas. The final synthesis results show that the critical path has a 15.8 ns delay, enabling a maximum clock rate of 63 MHz under best-case operating conditions without taking interconnect delay into account.

Table 5.3 summarizes the delay of the first three critical path groups and the components that account for the major delay in each group. Observe that the multiply-accumulate (MAC) unit accounts for 11 ns of the critical path delay in the ALU and bypassing path group. Due to the limitations of our current Synopsys licenses, the fastest implementations for the synthesized multipliers and adders are CSA (Carry Save Array) and CLA (Carry Look-Ahead), respectively. Therefore, if a faster MAC unit with a delay less than or equal to 8.5 ns can be obtained, the critical path delay can be reduced to 13.3 ns, enabling a maximum clock rate of 75 MHz. Further reduction of the critical path delay can be achieved by using faster on-chip memory macros. The delays of worst-case operating conditions are also summarized in Table 5.3. Under the worst-case conditions, the UTDSP can achieve a maximum clock rate of 29 MHz. However, it is our understanding that a significant yield is achievable at the best case timing [44].

The gate-level model was then verified using the same testbench used for the RTL simulation. Sometimes the gate-level simulation fails because of setup and hold violations. In these cases,

redoing the synthesis with more accurate timing constraints will fix the problems. To avoid setup and hold violations downstream in the post-layout phase, the clock source was defined with a slightly larger clock skew tolerance. After passing the gate-level simulation, the netlist is ready for the back-end hierarchical flow.

Critical Path Group	Maximum delay in the group		Major component and its delay		
	Best Case	Worst Case	Component	Best Case	Worst Case
ALU and bypassing paths	15.8 ns	34.7 ns	MAC Unit	11 ns	25 ns
Data Memory and bypassing paths	13.3 ns	20 ns	On-chip Data Memory	9 ns	11 ns
Decoder Memory paths	9.3 ns	11.7 ns	On-chip Decoder Memory	9 ns	11 ns

Table 5.3: The delay of the first three critical path groups

Table 5.4 summarizes the gate counts and actual areas of the major components in the UTDSP. The areas are measured from the routed layout of each component. The areas shown contain about 40% routing area on average. This utilization rate could be improved if more metal layers were available for routing. Observe that a very large portion of the core area is occupied by the two synthesized register files. To estimate the area of the chip if the synthesized register files can be replaced by ones made using full-custom design methods, we use the area model of on-chip multi-port memory proposed by Michael Flynn et al. [34] to estimate the area of the register files. Table 5.4 shows that using register files made with full-custom design methods will reduce the core area of the UTDSP from 12 mm^2 to 8.48 mm^2.

Component	Gate Count	Area after routing (mm^2)	If multi-port SRAM compiler is used
PC Unit	6078	1.13	
Memory Units (x2)	1961	0.40	
Address Units (x2)	7843	1.21	
Integer Units (x2)	17647	2.81	
PCU	3333	0.62	

Table 5.4: Gate counts and areas of the components in the UTDSP

Component	Gate Count	Area after routing (mm^2)	If multi-port SRAM compiler is used
Controller (DMA + Interrupt)	4118	0.61	
Glue logic	7843	1.34	
Register Files (x2)	23725	**3.88**	**0.36**
UTDSP Core (Sum of above)	**72548**	**12**	**8.48**
Instruction and Decoder Memory Blocks (2Kx32 bits)	57843	6.05	
Data Memory Blocks (2Kx8)	11765	2.75	
Top-level routing channels	0	2.2	
UTDSP Core + On-chip Memory blocks (no pads)	142156	23	19.48
The UTDSP chip (with pads)		**33**	**29.48**

Table 5.4: Gate counts and areas of the components in the UTDSP

Recall from the previous chapters that the UTDSP is designed to meet the increasing demand for high performance, low-cost processors for use in cost-sensitive embedded systems. The UTDSP provides a unique feature that makes itself an ideal application-specific programmable processor. Being designed using the application-driven design methodology described in Chapter 2, the UTDSP has a flexible architecture and instruction set that can be easily modified to meet the performance and cost requirements of target applications.

We argue that the synthesis-based implementation method is the most suitable for the UTDSP because being implemented in VHDL, the ALUs and instruction set of the UTDSP can be easily modified, enabling the application-driven design methodology and a short time-to-market. If a higher operating speed is required, the MAC unit can be implemented using a full-custom design method to reduce the critical path delay with little impact on the flexibility of the UTDSP architecture.

5.7.2 Floorplanning

In conventional back-end flows, floorplanning tools usually perform two tasks: The placement of blocks and the placement of logic cells in the blocks. The floorplan tools take a gate-level netlist and create corresponding blocks according to the logical hierarchy in the netlist. Therefore, conventional floorplanning only tries to minimize area or interconnect delay by arranging the

locations and aspect ratios of these blocks. However, the logical hierarchy in the original design does not necessarily mean a good partition for minimizing the interconnect delay and area of a chip.

As mentioned before, research has shown that the interconnect delays in deep-sub-micron (DSM) designs will actually decrease at the 50,000-gate module level as feature sizes shrink [31], Therefore, partitioning a chip into 50,000-gate blocks becomes a new floorplanning task that needs to be solved in this design phase. The UTDSP CAD flow takes advantage of the interconnect analysis tool and logic merging features provided in the Cadence Physical Design Planner (PDP 3.4C). The interconnect analysis tool can choose the blocks to be merged to minimize the number of external nets — the nets that connect blocks, while the logic merging features can merge selected blocks into a new block. However, it is difficult to perform merging and floorplanning without the information about the physical layout of underlying blocks. Therefore, some of the blocks will be pre-routed and their geometry information can be fed back to the floorplan tool. When the final version of floorplan is decided, global-pin-optimization will be performed to optimize the pin locations of these blocks so that the top-level interconnection can be reduced to a minimum.

Figure 5.12 shows the physical groups before and after logic merging. The 16 physical groups in the original design were merged into five groups to minimize the number of external nets between the blocks, while keeping the size of each block in the range where its interconnect delay can be reduced. Figure 5.13 shows the block interconnection analysis and the final floorplan for the UTDSP. Observe that the space between blocks was kept very small so that the area of the chip can be reduced. However, to successfully route the top-level wires in such a small area, the locations of the pins in the blocks have to be arranged in a way where most of the wires can travel in their shortest distances. This step is called global-pin-optimization [36]. After the pin locations of these blocks are optimized, the placement and routing in each block can be performed according to its pin locations. Figure 5.14 and Figure 5.15 show that the global-pin-optimization significantly reduces the interconnect between blocks. Sometimes the placement and routing in blocks fail because the areas of these blocks are insufficient. Reshaping the sizes of these blocks and redoing the floorplanning and global-pin-optimization are needed to solve this situation.

Although this floorplanning method is iterative and very time consuming, the final layout of the UTDSP shows a tremendous reduction in both area and interconnect delay. Figure 5.16 shows the result of the final top-level routing and Figure 5.17 shows one previous version of the UTDSP, where logic merging and floorplanning were performed in a different configuration. This poor floorplanning resulted in an area of 7.2 mm by 7.2 mm, which is much larger than the current size (5.5 mm by 6 mm). Therefore, applying this hierarchical floorplanning technique resulted in an area reduction of 36% in this case.

To obtain the postlayout timing information of the UTDSP, a full-chip R/C extraction was performed after routing to calculate the interconnect delay. The results show that the maximum interconnect delay in the first critical path group is 92 ps, resulting in a total delay of 16 ns in the critical path. The maximum interconnect delay in the second critical path group is only 45 ps. Therefore, the UTDSP can achieve a maximum clock rate about 63 MHz. This result shows that a good floorplan reduces not only the area of the chip but also its interconnect delay.

5.8 Kernel Benchmarks

The UTDSP not only has an architecture that is an easy target for HLL compilers but also provides nestable, zero-overhead hardware loops that are ideal for loop-intensive DSP kernels. Table 5.5 summarizes the loop performance of the UTDSP kernel benchmarks, which were created by compiling their functionally-equivalent C code using the UTDSP compiler.

UTDSP Kernels	Cycles
N-tap FIR with M points	M(N+4) +2
N cascaded Biquad IIR with M points	M(5N+3)
N-section Normalized Lattice filter with M points	M(6N+3)
N-tap LMS adaptive filter with M pointers	M(4N+6)
N radix-2 FFT butterfly	4N
NxN matrix multiplication	$N(N^2+3N+1)$

Table 5.5: UTDSP benchmark results for compiler-generated kernel code

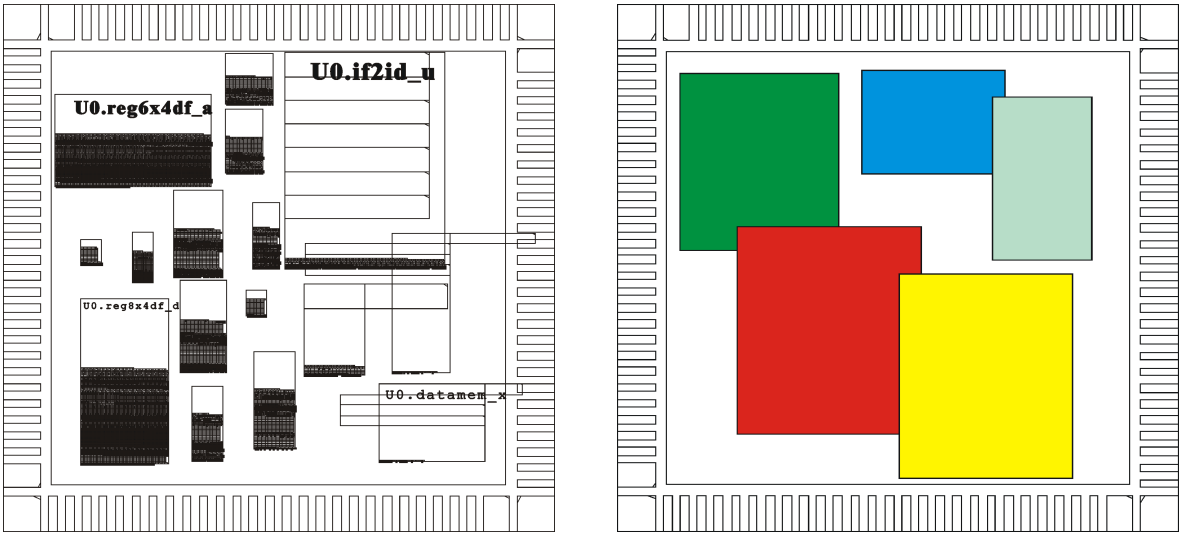


Figure 5.12: Before and after logic merging

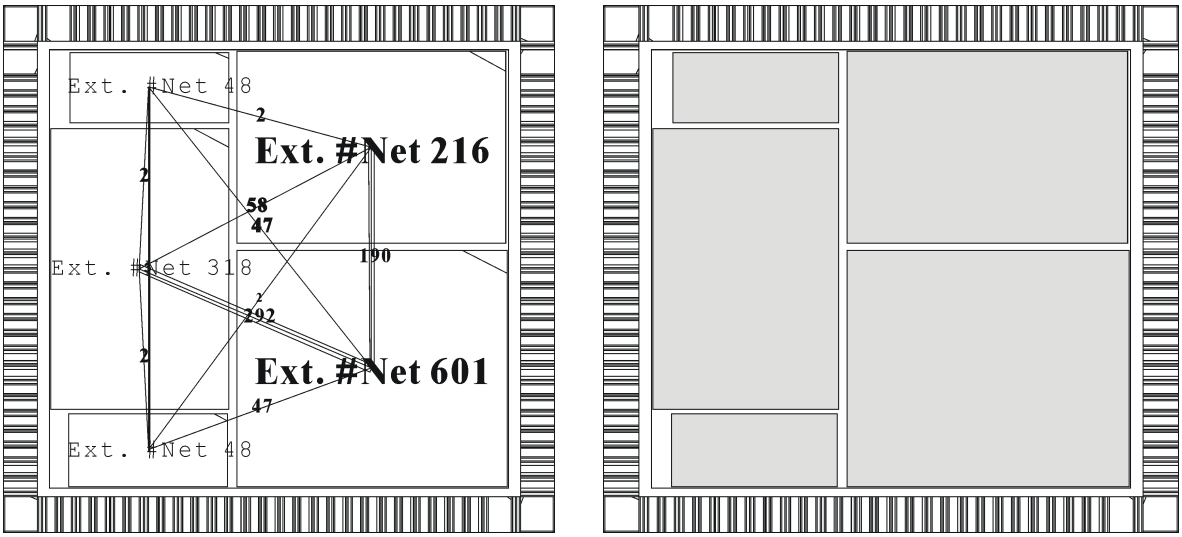


Figure 5.13: Group connectivity analysis and the final floorplan

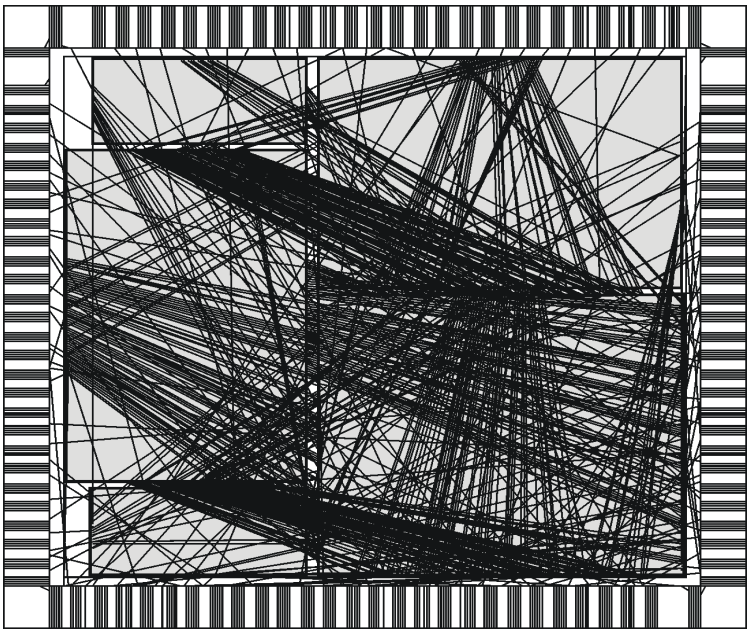


Figure 5.14: Block interconnect without using global-pin-optimization

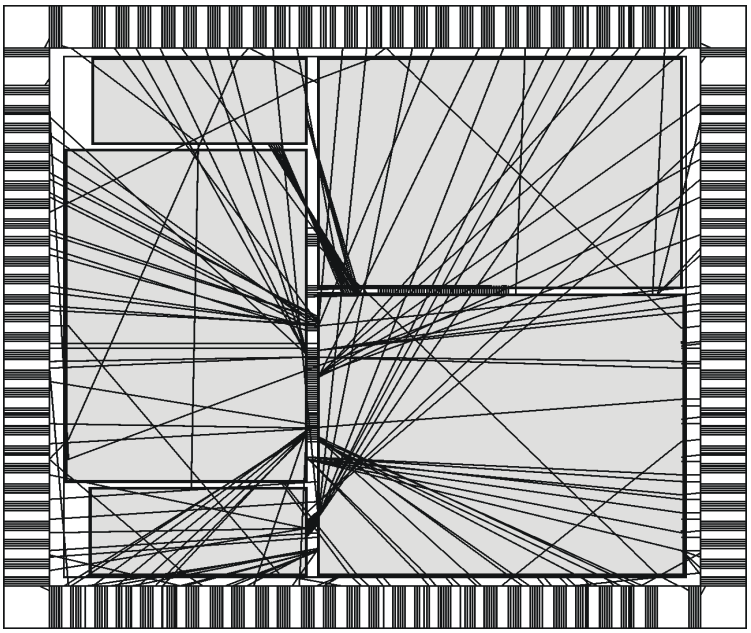


Figure 5.15: Block interconnect after global-pin-optimization

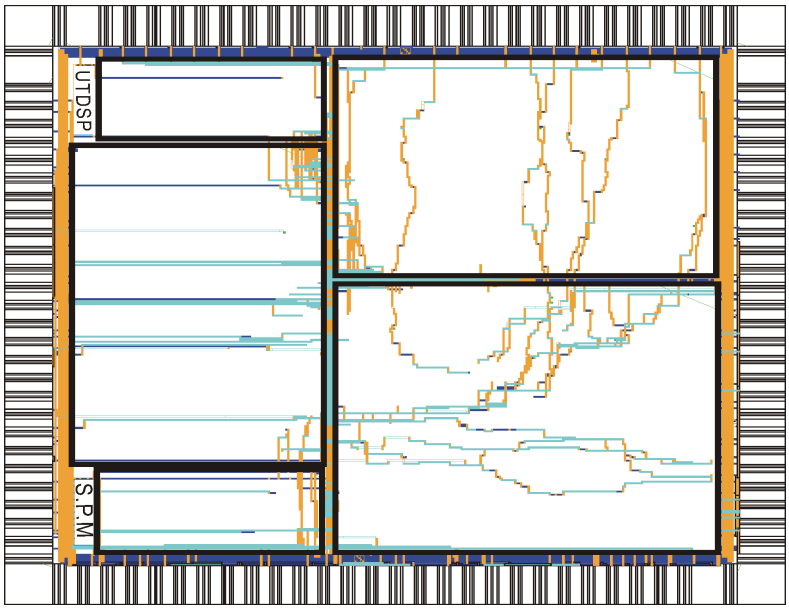
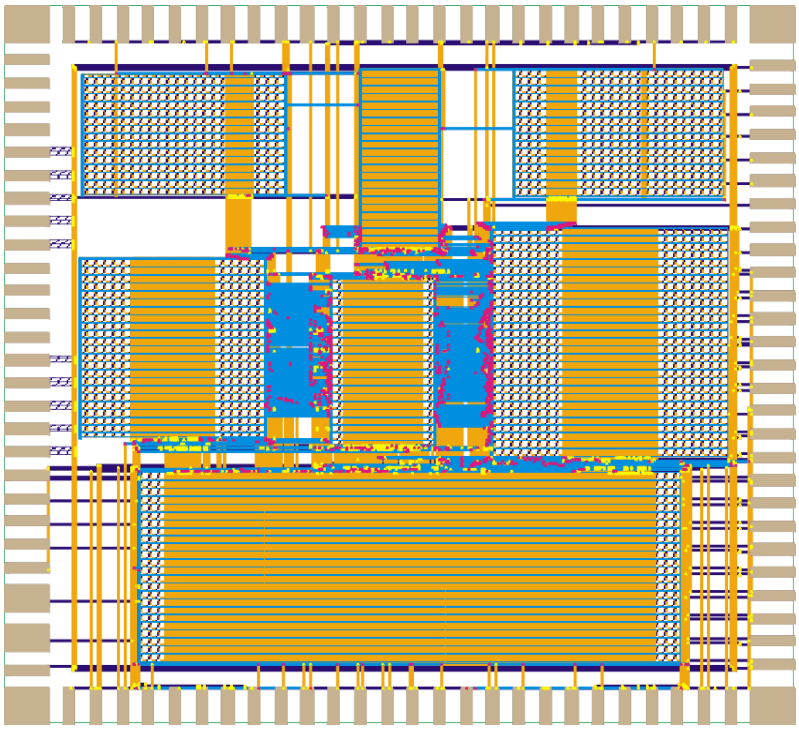


Figure 5.16: The final top-level routing (5.5 mm x 6.0 mm)



.

Figure 5.17: One of the previous top-level routing with a poor floorplan (7.2 mm x 7.2 mm)

To compare the UTDSP with the Philips R.E.A.L DSP [3] and the TI TMS320C62xx [26], which have similar VLIW architectures, an FIR benchmark used in the publications [3][37] of both processors was hand-translated into the UTDSP assembly program. To make a fair comparison, the assembly code was hand-optimized by the author because the assembly code for the other two processors were also hand-crafted by their best assembly programmers. Table 5.6 summarizes the features of these processors and the benchmark results. It shows that the UTDSP can achieve a higher performance in terms of the cycle count in this FIR benchmark.

This benchmark result demonstrates how the application-driven design methodology works with the flexible architecture of the UTDSP. To take advantage of the dual multiplier structure in the FIR computation, we used a block processing method, which is also used in the FIR benchmark [3] of the R.E.A.L DSP. In the block processing method, two output samples have to be calculated concurrently to reduce the cycle count by half; therefore, in each cycle two consecutive input samples and one coefficient must be loaded from the data memory. However, in each cycle the UTDSP can load only two new operands — one input sample and one coefficient, making the dual data-memory banks a bottleneck in the FIR benchmark.

	UTDSP	Philips R.E.A.L DSP	TI TMS320C6201
Design Methodology	Synthesis-based	Synthesis-based	Full-custom
Deliverable Form	1. 108-pin PGA 2. Synthesizable core	Synthesizeable core	352-pin BGA
Process technology	0.35 µm CMOS	0.25 µm CMOS	0.25 µm CMOS
Max. Clock Frequency	63 MHz	85 MHz	167- 200 MHz
Number of Functional Units	7	10	8
Number of Multipliers	2	2	2
Cycle Count for FIR with M outputs and N taps.	**M(N+6)/2 + 7**	**M(N+9)/2 + 8**	**M(N+8)/2 + 6**

Table 5.6: Comparison between UTDSP and two VLIW DSPs

To solve this problem, we introduced a new MAC instruction that not only performs the multiply-accumulate operation but also moves the operands. Table 5.7 shows this instruction and its corresponding operations. Note that this instruction does not need an extra write port on the regis-

ter file because the original multiply-accumulate operation stores its result to the accumulator (Acc), making its write port available during the WB stage for the shuffling operation. By using this MAC instruction, the inner loop body, which calculates two output samples simultaneously, can be scheduled into one long instruction. Figure 5.18 shows the inner loop code of the FIR benchmark. Observe that the inner loop contains only one long instruction (INST 2), which consists of six parallel operations. Without this MAC instruction, one extra long instruction has to be scheduled into the inner loop body, doubling the current cycle count.

Instruction	Syntax	Operations
Multiply-accumulate (for block processing)	madd2m di, dj, dk	Acc = Acc + (di * dj) dk = di

Table 5.7: The MAC instruction for block processing

```
INST 1: rep N                 // repeat the following instruction N times. N = number of coefficients

INST 2: ld.d (a1), d1         // load next input sample
        ld.d (a2), d2         // load next coefficient
        inc a1, a15, a1       // increment input sample array pointer a1
        inc a2, a15, a2       // increment coefficient array pointer a2
        madd2m d1, d2, d3     // calculate the term and move the input sample d1 to d3 for subsequent output
        madd2m d3, d2, null   // calculate the term for subsequent output sample
```

Figure 5.18: The UTDSP assembly code for the inner loop of FIR benchmark

5.9 Summary

The chapter discusses the hardware design of the UTDSP and various VLSI implementation issues. The design of the nestable, zero-overhead hardware loops is highlighted. A novel hierarchical CAD flow was used to minimize both the interconnect delay and chip area of the UTDSP. Applying the hierarchical CAD flow resulted in an excellent floorplan that could reduce the area by 36% in one of the examples shown. Finally, the kernel benchmark results are shown and a comparison with two other VLIW processors is provided.

Chapter 6

Conclusions and Future Work

6.1 Conclusion

This thesis discusses the design and implementation of the UTDSP system, which consists of three parts: the long-instruction packing scheme, the development software, and the VLSI implementation of the UTDSP processor. Each part contributes to the UTDSP system in its own way and makes the UTDSP a complete system that is comparable to commercially-available DSP products.

First, the UTDSP packing scheme reduces storage requirements, while eliminating the memory bandwidth problems that plague other VLIW architectures. The UTDSP packing algorithm incorporates a two-cluster packing and slot sharing methods to minimize storage requirements. Benchmark results indicate that the UTDSP packing scheme achieves a performance comparable to its commercial counterpart.

Second, the development software, consisting of an architecture simulator and an assembly debugger, not only enables an application-driven design methodology but also provides programmers an interactive GUI-based debugging tool. The architecture simulator was designed in a novel method where the design gap between the behavioural and RTL models of the UTDSP can be minimized. The GUI-based assembly debugger was implemented by adding a self-displaying and event-listening capabilities to the architecture simulator. This means that designers will always have a functionally equivalent debugger ready for use after having modified the architecture simulator. This capability is especially useful in a core-based design.

Third, designed with a goal to provide high performance and low cost, the UTDSP not only has a flexible architecture that is an easy target for HLL compilers, but also provides features such as

zero-overhead hardware loops to optimize the performance in loop-intensive computations. The VLSI implementation of the UTDSP adopts a synthesis-based design methodology and a novel hierarchical floorplanning technique that can significantly reduce the resulting area and interconnect delay of the UTDSP.

6.2 Future Work

Following are some suggestions for future work that can be used to improve the UTDSP system.

- Building the register files of the UTDSP using SRAM compilers with multi-port capabilities.
  Due to the unavailability of SRAM compilers with multi-port capabilities, the register files in the UTDSP were synthesized directly from VHDL code, which significantly increased the area of the UTDSP. It would be desirable if the register files could be generated using SRAM compilers.

- Implementing the UTDSP in a CMOS technology with more metal layers.
  The UTDSP was implemented in a 0.35 µm CMOS technology with only three metal layers. The layout of library cells uses the first metal layer exclusively, leaving only two metal layers for routing. Therefore, congestion situations often occurred and failed the routing of blocks. The size of the blocks had to be increased to successfully route the design, resulting in a low utilization rate. The area of the UTDSP can be further reduced if more metal layers are available for routing.

- Further minimizing the design gap
  Although the behavioural modelling method described in Chapter 4 can bridge the design gap between the behavioural and RTL model of the UTDSP, it still requires that designers manually create the RTL model. We have already started investigating the possibility of describing digital hardware using a set of specialized Java classes and constructing a compiler that converts the Java-based hardware description into synthesizable RTL code, completely eliminating the design gap.

Appendix A

UTDSP Instruction Set

Operation	Syntax	Functional Description
Load integer register	ld.d (ai), dj	dj = Memory[ai]
Store integer register	st.d (ai), dj	Memory[ai] = dj

Table A.1: Memory Instructions

Operation	Syntax	Functional Description
Address decrement	dec ai, aj, ak	ak = ai - aj
Modulo address decrement	decmod ai, aj, ak	ak = (ai - aj) mod Buffer Size
Bit-reversed address decrement	decfft ai, aj, ak	ak = ai - aj (bit-reversed)
Address increment	inc, ai, aj, ak	ak = ai + aj
Modulo address increment	incmod ai, aj, ak	ak = (ai + aj) mod Buffer Size
Bit-reversed address increment	incfft ai, aj, ak	ak = ai + aj (bit-reversed)
Bit-wise AND	and.a ai, aj, ak	ak = ai & aj
Arithmetic shift left	asl.a ai, aj, ak	ak = ai << aj (arithmetic)
Arithmetic shift right	asr.a ai, aj, ak	ak = ai >> aj (arithmetic)
Bit-wise inclusive OR	ior.a ai, aj, ak	ak = ai | aj
Logical shift left	lsl.a ai, aj, ak	ak = ai << aj (logical)
Logical shift right	lsr.a ai, aj, ak	ak = ai >> aj (logical)
Bit-wise exclusive OR	xor.a ai, aj, ak	ak = ai ^ aj
Set equal	seq.a ai, aj, ak	ak = (ai == aj)
Bit-wise NOT	not.a ai, ak	ak = ~ ai
Move register	mov.a ai, ak	ak = ai

Table A.2: Addressing Instructions

Operation	Syntax	Functional Description
Set up the first circular buffer	set1 ai, aj	begin register = ai end register = aj
Set up the second circular buffer	set2 ai, aj	begin register = ai end register = aj
Move immediate	movi.a #X, ak	ak = #X

Table A.2: Addressing Instructions

Operation	Syntax	Functional Description
Absolute value	abs.d. di, dk	dk = | di |
Bit-wise NOT	not.d di, dk	dk = ~ di
Move register	mov.d di, dk	dk = di
Add	add.d di, dj, dk	dk = di + dj
Bit-wise AND	and.d di, dj, dk	dk = di & dj
Arithmetic shift left	asl.d di, dj, dk	dk = di << dj (arithmetic)
Arithmetic shift right	asr.d di, dj, dk	dk = di >> dj (arithmetic)
Bit-wise inclusive OR	ior.d di, dj, dk	dk = di | dj
Logical shift left	lsl.d di, dj, dk	dk = di << dj (logical)
Logical shift right	lsr.d di, dj, dk	dk = di >> dj (logical)
Subtract	sub.d di, dj, dk	dk = di - dj
Bit-wise exclusive OR	xor.d di, dj, dk	dk = di ^ dj
Set equal	seq.d di, dj, dk	dk = (di == dj)
Set not equal	sne.d di, dj, dk	dk = (di != dj)
Set greater than	sgt.d di, dj, dk	dk = (di > dj)
Set less than	slt.d di, dj, dk	dk = (di < dj)
Multiply (1.15 format)	multf.d di, dj, dk	dk = di * dj
Multiply	mult.d di, dj, dk	dk = di * dj
Move immediate	movi.d #X, dk	dk = #X
Multiply-accumulate	madd.d di, dj, dk, dl	dl = dk + (di * dj)
Multiply-accumulate (1.15)	maddf.d di, dj, dk, dl	dl = dk + (di * dj)
Multiply-subtract	msub.d di, dj, dk, dl	dl = dk + (di * dj)
Multiply-subtract (1.15)	msubf.d di, dj, dk, dl	dl = dk + (di * dj)
Setup Accumulator 0 (Acc0)	setacc0 di	Acc0 = di
Setup Accumulator 0 (Acc1)	setacc1 di	Acc1 = di
Multiply-accumulate (Acc0)	madd2d0 di, dj, dk	dk = Acc0 + (di * dj)

Table A.3: Integer Instructions

Operation	Syntax	Functional Description
Multiply-accumulate (Acc1)	madd2d1 di, dj, dk	dk = Acc1 + (di * dj)
Multiply-accumulate (Acc0:1.15format)	madd2f0 di, dj, dk	dk = Acc0 + (di * dj)
Multiply-accumulate (Acc1: 1.15 format)	madd2f1 di, dj, dk	dk = Acc1 + (di * dj)
Multiply-accumulate (for block processing)	madd2m di, dj, dk	Acc = Acc + (di * dj) dk = di
Multiply-accumulate (1.15) (for block processing)	madd2fm di, dj, dk	Acc = Acc + (di * dj) dk = di

Table A.3: Integer Instructions

Operation	Syntax	Functional Description
Address to integer	mova2d ai, dk	dk = ai
Integer to address	movd2a di, ak	ak = di
Single-instruction repeat	rep #X	Repeat following instruction #X times
Instruction block repeat	do #X, label	Repeat instruction block #X times
Instruction block repeat	do.a ai, label	Repeat instruction block (ai) times
Instruction block repeat	do.d di, label	Repeat instruction block (di) times
Branch if address register equal to zero	beqz.a ai, label	if (ai == 0) PC = label
Branch if integer register equal to zero	beqz.d di, label	if (di == 0) PC = label
Branch if address register not equal to zero	bnez.a ai, label	if (ai != 0) PC = label
Branch if integer register not equal to zero	bnez.d di, label	if (di != 0) PC = label
Jump indirect	jmp.a (ai)	PC = (ai)
Jump direct	jmp label	PC = label
Jump subroutine	jsr label	Push current PC PC = label
Return from subroutine	rts	Restore PC from the stack
Trap	trap #6/60 trap #5/50	DMA read from IO to X/Y bank DMA write to IO from X/Y bank
Wait until interrupt	wait	Processor go to idle status
Halt	halt	System halt

Table A.4: Control Instructions

Bibliography

[1] Mazen A.R. Saghir, Paul Chow, and Corinna G. Lee, “Application-Driven Design of DSP Architectures and Compilers” Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, pp. II-437-440, IEEE, 1994.

[2] Mazen A.R. Saghir, Application-Specific Instruction-Set Architectures for Embedded DSP Applications, Ph.D. Thesis, University of Toronto, 1998.

[3] C.M. Moerman, R. Woudsma, P. Kievits, Philips Semiconductors ASIC Service Group, “Embedded DSP Technologies in Consumer Applications: REAL DSP,” Class Notes, DSP World Workshops, Toronto, September, 1998.

[4] E. Horst, W. Kloosterhuis, J. Heyden, “A C Compiler for the Embedded R.E.A.L. DSP Architecture,” The International Conference on Signal Processing Applications & Technology, Toronto, 1998.

[5] Motorola, DSP56002 Digital Signal Processor User’s Manual, 1990.

[6] E.W. Reigel, U. Faber, and D.A. Fisher, “The Interpreter — A Microprogrammable Building Block System,” Proceedings of the AFIPS Spring Joint Computer Conference, Vol. 40, pp. 705-723, 1972.

[7] Robert F. Rosin, Gideon Frieder, and Richard H. Eckhouse, Jr., “An Environment for Research in Microprogramming and Emulation,” Communications of the ACM, Vol 15, No. 8, pp. 748-760, ACM, 1972.

[8] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley Publishing Company, 1986.

[9] Vijaya K. Singh, An Optimizing C Compiler for a General Purpose DSP Architecture, M.A.Sc. Thesis, University of Toronto, 1992.

[10] Mark G. Stoodley, Scheduling Loops with Conditional Statements, M.A.Sc. Thesis, University of Toronto, 1995.

[11] Robert P. Wilson, et al., “SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers”, http://suif.stanford.edusuif/suif1/suif-overview/suif.html, 1994.

[12] Sanjay M. Pujare, Machine-Independent Compiler Optimizations for the U of T DSP Architecture, M.Eng. Thesis, University of Toronto, 1995.

[13] David Landskov, Scott Davidson, Bruce Shriver, and Patrick W. Mallett, “Local Microcode Compaction Techniques”, ACM Computing Surveys, Vol. 12, No. 3, pp. 216-294, ACM, September, 1980.

[14] Dillon, T. J. Jr, The VelociTI Architecture of the TMS320C6x, ICSPAT, 1997.

[15] John Hennessy, “The Future of Systems Research,” pp.31, IEEE Computer, August, 1999.

[16] Evan Cone, Heather Edwards, Developing an OPC Client Application Using Visual Basic, Application Note 139, National Instruments.

[17] Prem Jain, “Analyzing and optimizing embedded system-on-a-chip performance,” Computer Design, pp.42, October, 1998.

[18] Bruce J. MacLennan, Principles of Programming Languages: Design, Evaluation, and Implementation, Second Edition, HOLT, RINEHART AND WINSTON, 1987.

[19] Peter Linz, An Introduction to Formal Languages and Automata, Second Edition, D. C. Heath and Company, 1996.

[20] Sun Microsystems, http://www.java.sun.com/

[21] H. M. Deitel, Java: How to program, Second Edition, Prentice Hall, 1998.

[22] CynApps Technology, Inc., Bridging the Design Gap, http://www.CynApps.com/, 1998.

[23] Rahman Jamal, Herbert Pichlik, LabVIEW Applications and Solutions, Prentice Hall, 1999.

[24] John Hennessy, David Patterson, Computer Organization and Design: The Hardware/ Software Interface, Second Edition, Morgan kaufmann Publishers, Inc. 1998.

[25] Arthur Abnous, Nader Bagherzadeh, “Pipelining and Bypassing in a VLIW Processor,” IEEE Transactions on Parallel and Distributed Systems, Vol. 5, No. 6, pp. 658-663, June, 1994.

[26] Texas Instruments, TMS320C62xx Reference Guide, 1997.

[27] Motorola, DSP56302 User's Manual, 1997.

[28] R. J. Higgins, "Digital Signal Processing in VLSI," pp.341, Prentice-Hall, 1990.

[29] Lucent Technologies, DSP16xx Programmer's Reference Guide, 1998.

[30] Texas Instruments, TMS320C5x Reference Guide, 1998.

[31] Dennis Sylvester, Kurt Keutzer, "Getting to the Bottom of Deep Submicron," International Conference on Computer-Aided Design (ICCAD-98), 1998.

[32] Canadian Microelectronics Corporation, http://www.cmc.ca/.

[33] Synopsys Inc, http://www.synopsys.com/.

[34] Johannes M. Mulder, Nhon T. Quach, Michael J. Flynn, "An Area Model for On-Chip Memories and its Application," IEEE Journal of Solid-State Circuits, Vol. 26, No. 2, February 1991.

[35] Motorola FastSRAM Products, http://www.motorola.com/SPS/FastSRAM/.

[36] Cadence, Cadence Design Planner Training Manual, Version 3.4A, March, 1998.

[37] Texas Instruments, http://www.ti.com/sc/docs/products/dsp/c6000/62bench.htm#filters

[38] John Hennessy, David Patterson, Computer Architecture: A Quantitative Approach, Second Edition, Morgan kaufmann Publishers, Inc. 1998.

[39] J. Labrousse, G. A. Slavenburg, "A 50 MHz Microprocessor with a Very Long Instruction Word Architecture," ISSCC'90, February, 1990.

[40] R. B. Rau, D. W. Yen, W. Yen, R.A. Towle, "The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions and Trade-offs," IEEE Computer, pp. 12-35, January, 1989.

[41] TSMC — Taiwan Semiconductor Manufacturing Co., Ltd, http://www.tsmc.com.tw/e-html/index-e.htm.

[42] Mazen Saghir, Paul Chow, Corinna Lee, "Exploiting Dual Memory Banks in Digital Signal Processors," SIGARCH Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOSVII), pp. 234-243, Boston, MA, October, 1996.

[43] Mazen A. R. Saghir, Architectural and Compiler Support for DSP Applications, M.A.Sc. Thesis, University of Toronto, 1993.

[44] Macron Cheng, System Design Manager, Syntek Semiconductor Co., LTD, Taiwan, Pri-

vate Communication, 1999.

[45] P. Kievits, E. Lambers, C. Moerman, R. Woudsm, “R.E.A.L. DSP Technology for Telecom Baseband Processing,” The International Conference on Signal Processing Applications & Technology, Toronto, 1998.