Submitted to the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-31)

A Comparison of VLIW and Traditional DSP Architectures for Compiled Code

Mazen A. R. Saghir, Paul Chow, and Corinna G. Lee
Department of Electrical and Computer Engineering
University of Toronto

Abstract

Although programmable digital signal processors comprise a significant fraction of the processors sold in the world, their basic architectures have changed little since they were originally developed. The evolution and implementation of these processors has been based more on commonly held beliefs than quantitative data. In this paper, we show that by changing to a VLIW model with more registers, orthogonal instructions, and better flexibility for instruction-level parallelism, it is possible to achieve at least a factor of 1.3–2 in performance gain over the traditional DSP architectures on a suite of DSP benchmarks. When accounting for the effect of restrictive register use in traditional DSP architectures, we argue that the actual performance gain is at least a factor of 1.8–2.8. To counter an argument about extra chip area, we show that the cost of adding more registers is minimal when the overall area of the processor and the performance benefits are considered. Although a VLIW architecture has a much lower instruction density, we also show that the average number of instructions is actually reduced because there are fewer memory operations. A significant contribution to the better performance of the VLIW architecture is the ability to express more instances of parallelism than the restricted parallelism of the more traditional architectures. However, efficient techniques for encoding long instructions are required to make the higher flexibility and better performance of VLIW architectures feasible.

1. Introduction

Digital signal processors (DSPs) are specialized microprocessors optimized to execute the computationally-intensive operations commonly found in the inner loops of digital signal processing algorithms. DSPs are typically designed to achieve high performance and low cost, and are therefore used extensively in embedded systems. Some of the architectural features commonly found in DSPs include fast multiply-accumulate hardware, separate address units, multiple data-memory banks, specialized addressing modes, and support for low-overhead looping. Another common feature of DSP architectures is the use of tightly-encoded instruction sets. For example, using 16- to 48-bit instruction words, a common DSP instruction can specify up to five parallel operations: multiply-accumulate, two pointer updates, and two data moves.

Tightly-encoded instructions that could specify five operations in a single instruction were initially used to improve code density with the belief that this also reduced instruction memory requirements, and hence cost. Tightly-encoded instructions were also used to reduce instruction memory bandwidth, which was of particular concern when packaging was not very advanced. As a result of this encoding style, DSP instruction-set architectures (ISAs) are not very well suited for automatic code generation by modern, high-level language (HLL) compilers. For example, most operations are accumulator based, and very few registers can be specified in a DSP instruction. Although this reduces the number of bits required for encoding instructions, it also makes it more difficult for the compiler to generate efficient code. Furthermore, DSP instructions can only specify limited instances of parallelism that are commonly used in the inner loops of DSP algorithms. As a result, DSPs cannot exploit parallelism beyond what is supported by the ISA, and this can degrade performance significantly.

For the early generation of programmable DSPs, tightly-encoded instruction sets were an acceptable solution. At that time, DSPs were mainly used to implement simple algorithms that were relatively easy to code and optimize in assembly language. Furthermore, the compiler technology of that time was not advanced enough to generate code with the same efficiency and compactness as that of a human programmer. However, as DSP and multimedia applications become larger and more sophisticated, and as design and development cycles are required to be shorter, it is no longer feasible to program DSPs in assembly language. Furthermore, current compiler technology has advanced to the point where it can generate very efficient and compact code. For these reasons, DSP manufacturers are currently seeking alternative instruction set architectures. One such alternative, used in new media processors and DSPs such as the Texas Instruments TMS320C6x [1], is the VLIW architecture.

In this paper, we compare the performance and cost of a VLIW architecture to those achieved with more traditional DSP architectures. In Section 2, we examine several architectural styles that may be used to implement embedded DSPs, and we compare their advantages and disadvantages to VLIW architectures. In Section 3, we describe the methodology used to model two commercial DSPs, and we compare their performance and cost to those of our model VLIW architecture. In Section 4, we present the results of this comparison. Finally, in Section 5, we summarize the main points of this paper and present our conclusions.

2. Architectural Choices

Among the features that are desirable to have in an embedded DSP architecture is the ability to support the exploitation of parallelism. This is especially important with DSP applications since they exhibit high levels of parallelism. Another desirable feature is that the architecture be an easy target for HLL compilers. This enables the processor to be programmed in a HLL, which reduces code development time, increases its portability, and improves its maintainability. It also makes it easier for a modern compiler to optimize the generated code and improve execution performance. Finally, to meet the low cost requirements of embedded DSP systems, an architecture must occupy a small die area, and must use functional and control units having low complexity. In this section, we describe some of the architectural alternatives that can be used to implement embedded DSPs, and we discuss their advantages and disadvantages.

2.1. Superscalar Architectures

Superscalar architectures are the current style used to implement general-purpose processors. They exploit parallelism in the stream of instructions fetched from memory by issuing multiple instructions per cycle to the datapath. As instructions are fetched from memory, they are temporarily stored in an instruction buffer where they are examined by a complex control unit. The control unit determines the interdependencies between these instructions, as well as the dependencies with instructions already executing. Once these dependencies are resolved, and hardware resources become available, the control unit issues as many instructions in parallel as possible for execution on the datapath.

In older superscalar architectures, instructions could only be issued in parallel if they happened to be in the correct static order inside the instruction buffer. Since instructions could only be issued in the same order they were fetched, these architectures were said to exploit *static* parallelism. On the other hand, more recent superscalar architectures are able to exploit parallelism among any group of instructions in the buffer. As such, they are said to exploit *dynamic* parallelism. This is achieved, in part, by using such techniques as dynamic register renaming or branch prediction, which eliminate false dependencies between instructions and increase the ability to issue instructions in parallel [2]. Instructions in the buffer can therefore be issued out of the order in which they were fetched, and the control unit is responsible for ensuring that the state of the processor is modified in the same order the instructions are fetched.

The compiler technology associated with superscalar architectures is also very advanced and well understood. In addition to generating efficient code by using machine-independent, scalar optimizations [3], contemporary optimizing compilers are able to exploit the underlying features of their target architectures. Typically, they applying machine-dependent optimizations such as instruction scheduling, which minimizes pipeline latencies and helps expose parallel instructions to the hardware; data prefetching, which exploits the memory hierarchy; register renaming, which attempts to statically remove false dependencies between instructions; and branch prediction, which attempts to minimize branch penalties by predicting execution control paths.

Given their ability to exploit parallelism dynamically, and their sophisticated compilers, superscalar architectures are capable of exploiting the high levels of parallelism found in DSP applications. However, as available parallelism increases, and the number of functional units and registers increase to support it, the complexity and cost of their control units also increase. Given the stringent low-cost requirements of most embedded DSP systems, superscalar architectures may not therefore be the most cost-effective architectural choice.

2.2. SIMD Architectures

With the recent growth in multimedia applications for desktop computer systems, many general-purpose processor vendors have introduced multimedia extensions to their instruction set architectures [4]. Typically, these extensions take the form of special, single-instruction, multiple data (SIMD) instructions that perform identical, parallel operations on a fixed number of packed operands. The operands are typically 8-bit, 16-bit, or 32-bit integers that are packed into 32-bit or 64-bit registers. That is why these SIMD instructions are sometimes called *packed arithmetic* instructions, or are said to exploit *subword* parallelism.

SIMD instructions enhance the execution performance of *vectorizable* loops[1] that perform homogeneous operations on arrays of data. Since such loops are very common in multimedia and DSP applications, SIMD architectures can be used for these applications. Programs that use SIMD instructions can therefore achieve higher levels of code density and require less memory to store. The amount of hardware needed to support SIMD instructions is also minimal, and consists mainly of the additional control circuitry that enables an ALU to perform multiple slices of the same operation on individual subwords of its input operands.

Currently, SIMD architectures are difficult targets for HLL compilers, and, like most commercial DSPs, must be programmed in assembly language for their full benefits to be achieved. Since high-level lan-

---

1. A loop is vectorizable if it has no real dependencies between its different iterations.

guages, such as C, cannot be used to specify sub-word parallelism, it is very difficult for compilers to extract such parallelism. Instead, compilers for SIMD architectures will have to incorporate features found in vectorizing compilers, which is currently a topic of research [5].

2.3. Multiprocessor Architectures

For applications that exhibit high levels of coarse-grained parallelism, such as video or radar signal processing, multiprocessor architectures can be used. For example, the Texas Instruments TMS320C80 [6] packs four DSP processors and a RISC controller into the same chip. The four DSPs typically execute the same code, and process four, different data sets simultaneously. Since compiler support for such multiprocessor architectures has typically been poor or non-existent, applications must be carefully coded to benefit from the multiple processors.

In general, the number and type of processors used in a multiprocessor architecture, whether tightly-encoded DSPs, VLIW architectures, superscalar architectures, or SIMD architectures, depends on the application. However, since multiprocessor architectures are more costly than single-processor architectures, they are only used when the required level of performance justifies the added cost.

2.4. VLIW Architectures vs. Other Architectural Styles

Since DSP applications typically exhibit static control flow, compilers can easily be used to schedule instructions statically and exploit the available parallelism. This makes VLIW architectures, with their simpler control units, a more cost-effective choice than superscalar architectures in embedded systems.

VLIW architectures are also better suited for exploiting parallelism in DSP applications than SIMD architectures. Although SIMD architectures are suitable for applications that exhibit homogeneous parallelism, they are not well suited for applications that exhibit more heterogeneous parallelism, such as that found in the inner loops of DSP kernels, or the control code of DSP applications. The greater flexibility in VLIW architectures also makes them easier targets for HLL compilers. However, in their most basic form, they are also more expensive to implement. For example, while SIMD architectures use a single ALU to perform four parallel operations, a VLIW architecture requires four, full-precision ALUs to perform the same operations. Moreover, while the four operations can be encoded as a single SIMD instruction, they have to be encoded as four, separate operations in a long instruction. One way to reduce the cost of a VLIW architecture is to customize it to the target application. For example, smaller-precision ALUs and datapaths can be used to match the requirements of the application. Instruction storage and bandwidth requirements can also be reduced by using efficient long-instruction encoding techniques. Even specialized SIMD operations can be used to supplement a VLIW architecture and reduce its cost. This, however, requires more sophisticated compiler technology to exploit available sub-word parallelism.

Finally, VLIW architectures are better suited for exploiting the fine-grained parallelism found in DSP applications than multiprocessor architectures. Even though some applications can benefit from the exploitation of coarse-grained parallelism, most applications cannot justify the additional cost of a multiprocessor architecture. Multiprocessor DSP architectures are also difficult targets for HLL compilers, making them very cumbersome to program.

Since VLIW architectures have the flexibility to exploit different levels of parallelism, have simple control structures, and are easy targets for current HLL compilers, they are the most suitable architecture for embedded DSP systems. However, a major impediment to using VLIW architectures in cost-sensitive embedded systems has been their high instruction bandwidth requirements. Nonetheless, several approaches have been employed in research and commercial VLIW architectures to overcome this problem [18]. The remainder of this paper shows the performance and cost benefits of using VLIW architectures compared to typical, DSP-style architectures.

3. Experimental Methodology

The previous section described different architectural styles that can be used for embedded DSP applications, and compared their advantages and disadvantages with the VLIW architecture. This section describes the methodology used to model two traditional commercial DSP architectures, and compare their performance and cost to our VLIW-based model architecture.

3.1. Model DSP Architecture

Figure 1 shows our VLIW-based model DSP architecture, the UTDSP. We chose the VLIW model because of the flexibility it offers the compiler in exploiting parallelism, and its ability to support common DSP architectural features. This enabled us to develop our compiler with little concern for architectural constraints. We realize, however, that a straightforward implementation of this design is too costly, and we are currently exploring alternative, less costly implementations that deliver comparable performance.

Our model contains nine functional units. These include two memory units that access two single-ported data-memory banks; two address units that perform address calculations; two integer units; two floating-point units; and a control unit for executing control operations. Our model also contains three separate register files for storing address, integer, and floating-point data, respectively. Each register file contains 32 registers.

In our model, long instructions consist of nine operation fields that each control the execution of one of the functional units. Moreover, we assume that each functional unit executes a machine operation in a single clock cycle. Thus, our model supports the execution of up to nine independent machine operations every clock cycle.

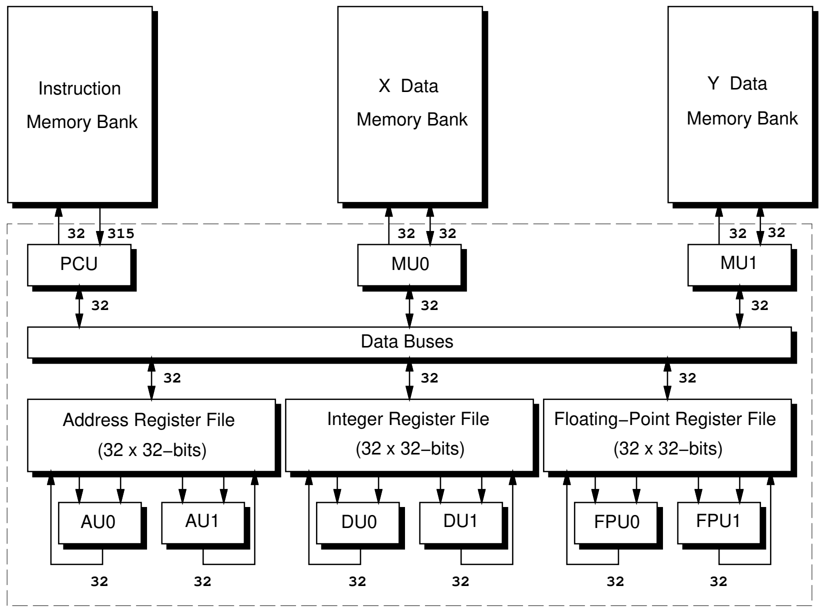


Figure 1: VLIW model DSP architecture.

We also developed an instruction set simulator for our architecture to gather performance statistics. The simulator runs the code generated by our optimizing C compiler.

3.2. Optimizing C Compiler

Since we are interested in programming DSPs using high-level languages, we developed an optimizing C compiler for our model architecture. Our compiler consists of a C-language front-end and an optimizing back-end. The front-end is based on the Stanford University SUIF compiler [7], and is mainly responsible for translating applications written in C to the basic machine operations of our model architecture. The front-end also applies traditional scalar optimizations and performs register allocation.

The back-end is responsible for optimizing the execution of the program on our model architecture and scheduling machine operations into long instructions. The code generated by the front-end is first analyzed, and information about its data-flow and control-flow characteristics is gathered for use by subsequent optimizing passes. These passes include using modulo addressing to implement circular buffers, using low-overhead looping operations, and partitioning the program's data among its dual data-memory banks to expose parallelism in load and store operations [8]. The last pass performs operation compaction where operations that can execute in parallel are identified and scheduled into long instructions.

3.3. Benchmarks

The DSP community has traditionally relied on commonly used, kernel-like algorithms to benchmark the performance of different processors. In fact, such algorithms form the basis for the design of most

Kernels		Description
k1	fft_1024	Radix-2, in-place, decimation-in-time fast Fourier transform (FFT)
k2	fft_256	
k3	fir_256_64	Finite Impulse Response (FIR) filter
k4	fir_32_1	
k5	iir_4_64	Infinite Impulse Response (IIR) filter
k6	iir_1_1	
k7	latnrm_32_64	Normalized lattice filter
k8	latnrm_8_1	
k9	lmsfir_32_64	Least-mean-squared (LMS) adaptive FIR filter
k10	lmsfir_8_1	
k11	mult_10_10	Matrix multiplication
k12	mult_4_4	

Table 1: DSP kernel benchmarks.

DSPs. While it is true that such algorithms are widely used in many applications, and while it is necessary that DSPs be capable of executing such algorithms efficiently, relying on kernel-like algorithms to assess the performance of new DSP architectures is not an accurate indicator of their performance on large applications [9]. Furthermore, as we shall show in Section 4, relying on such algorithms to guide the design of DSPs may significantly affect their execution performance on larger applications. It is therefore important to use real applications to benchmark DSPs and to guide the design of future DSP ISAs. This is especially necessary today, when applications are becoming more sophisticated, and DSPs are no longer used just to execute simple algorithms. This is also the approach used in designing contemporary, general-purpose microprocessors.

With the above in mind, we developed a suite of benchmarks consisting of 12 kernels and 10 applications to assess the performance of our architecture and compiler. Tables 1 and 2 show our benchmarks. The kernels are based on six common DSP algorithms. For each algorithm, we implemented two versions: one processing a small data set, and another processing a large data set. The applications are taken from the areas of speech processing, image processing, and data communications.

3.4. Modeling Existing Commercial DSP Architectures

To compare existing commercial DSP architectures and their compilers, the same source code should ideally be compiled to the different architectures, and the results compared. However, since DSP compilers vary in their level of sophistication and their ability to exploit the specialized features of their target architectures, it is difficult to rely on existing commercial compilers. In addition to reflecting the differences between the architectures, the code generated by these compilers would reflect the differences in the com-

Applications		Description
a1	G721_A	Two implementations of the ITU G.721 ADPCM speech transcoder
a2	G721_B	
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
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: DSP application benchmarks.

pilers. Furthermore, it is usually difficult to compile applications on different compilers without modifying the code to suite the requirements imposed by the different compilers. One way to avoid these problems was to use our own compiler, and model the resources and parallelism available in the existing DSPs. This would enable us to make a more accurate comparison between the different architectures.

Most DSPs use tightly-encoded instruction sets that support the exploitation of specific instances of parallelism. DSPs are also memory-based, and use few general-purpose registers. To study the effects of restricted parallelism and small register sets on performance and cost, we modified our compiler to generate code for two architectures modeled after commercial DSPs. In both cases, we constrained the number of functional units on our model architecture to reflect the datapaths of the DSPs under study. We also modified the compaction algorithm in the compiler back-end to only generate combinations of parallel operations that are supported by the instruction sets of both architectures. Finally, we also constrained the number of registers available to the compiler to reflect the number of registers used in each architecture.

The code generated by our modified compiler are still VLIW instructions, and they can be run on our instruction-set simulator. However, we restricted the parallelism to what is found in the processors being modeled. Each VLIW instruction corresponds, approximately, to an existing tightly-encoded instruction in the DSP being modeled. However, the actual DSPs restrict the way the registers are used in each instruction. Compiling code with these constraints is a difficult problem that is still being tackled by DSP compiler writers [10]. We did not model these restrictions on our modified compilers. Instead, all available registers could be used as the source or destination of any operation, which is one of the keys to generating efficient code using known compiler technology. Consequently, the results for the modeled processors will be overly optimistic.

To provide some insight into the difference in performance that can result from relaxing the constraints on the registers, we examined two kernels — `fir` and `iir`. We took the C code for the kernels from an EDN magazine article comparing the performance of different commercial DSP compilers [11]. Initially, we compiled the C code using our compiler, after restricting the parallelism it can exploit to what is found on the Motorola DSP56002 [12]. We chose the DSP56002 because it was one of the DSP targets used in the EDN article, and because it was easy to model on our architecture. We then compared the instructions generated by our modified compiler to the corresponding assembly code generated by the Tasking compiler for the DSP56002. We then calculated the execution time for each set of assembly code assuming an execution rate of one cycle per instruction, and accounting for loop counts. The results showed that the code generated by the Tasking compiler was 1.4–2 times slower than that generated by our modified compiler.

Since both the Tasking compiler and our modified compiler were capable of exploiting the parallelism supported by the DSP56002, and since DSPs such as the DSP56002 are specifically designed to execute kernels such as `fir` and `iir` efficiently, the results highlight the significance of restricting the use of registers on overall performance. Since our compiler does not restrict the use of registers, the effects of such restrictions should therefore be remembered when comparing the performance results of our VLIW-based architecture and the modeled DSP architectures.

The first DSP we modeled, **DSP1**, is based on the Motorola DSP56001. We chose this DSP because it is representative of popular, fixed-point DSPs including the Texas Instruments TMS320C50 [13] and the Analog Devices ADSP-2100 [14]. However, since all our benchmarks use floating-point data types and operations, we allowed the **DSP1** to execute floating-point operations subject to the same restrictions imposed by its instruction set architecture. However, this is acceptable since the available parallelism can still be modeled. The DSP56001 can execute most arithmetic and logic operations in parallel with up to two data moves and two address calculations. A data move is either a memory access or a register move. The DSP56001 also uses eight data registers and 12 address registers.

The second DSP we modeled, **DSP2**, is based on the Analog Devices ADSP-21020 [15]. We also chose this DSP because it is representative of floating-point DSPs including the Texas Instruments TMS320C30 [16] and the Motorola DSP96002 [17]. Table 3 shows the parallelism supported by the ADSP-21020's instruction set, which is more flexible than that of the DSP56001 because it allows more combinations of operations to be executed in parallel. The ADSP-21020 also uses 16 data registers and 64 address registers. However, of these 64 address registers, 16 are used to store segment base addresses. Since our model architecture uses a flat address space, these registers are not required, and the **DSP2** was modeled with only 48 address registers.

<table>
<tr><th colspan="3">Bin 1</th><th colspan="4">Bin 2</th></tr>
<tr><td>Multiply or Logical</td><td>Add</td><td>Subtract</td><td>Memory</td><td>Memory</td><td>Address</td><td>Address</td></tr>
<tr><td colspan="3" rowspan="2">Shift or Bit-Manipulation</td><td colspan="4">Register Move</td></tr>
<tr><td colspan="4">Branch</td></tr>
</table>

Table 3: Parallelism supported by the ADSP-21020. Operations in Bin 1 may be executed in parallel with operations in Bin 2. Within each bin, several operations may also be executed in parallel.

Finally, to study the effect of using a VLIW architecture with restricted parallelism and more registers on performance and cost, we increased the number of registers used in **DSP1** and **DSP2** to match the UTDSP architecture. Since **DSP2** has more address registers than the UTDSP architecture, we only increased its data registers. The resulting models are called **UTDSP1** and **UTDSP2**, and are true VLIW versions of the **DSP1** and **DSP2** architectures. This means that, even though they have the same functional units as **DSP1** and **DSP2**, they are not restricted by the instruction set architecture in exploiting available parallelism. Furthermore, since they have more registers, they make it easier for the compiler to generate more efficient code.

3.5. Measuring Performance and Cost

Execution performance is typically measured by execution time. To calculate the execution time of the benchmarks on the more traditional DSP architectures, we used basic-block execution frequencies. For every basic block, the number of instructions generated by the compiler was multiplied by the execution frequency of the block. The total execution time was then calculated as the sum, over all basic blocks, of this product. Since the compaction algorithm is limited to exploiting parallelism within a basic block, it does not affect branch operations, nor does it change a program's control-flow structure.

To compare the performance of the traditional DSP and restricted VLIW architectures with that of our full-fledged VLIW-based model architecture, we measured their *relative performance*. We define relative performance as the ratio of the execution time of the UTDSP architecture while exploiting full parallelism — we refer to this as **UTDSP_parallel** — to the execution time of the given processor. Thus, a processor with relative performance less than 1.0 has worse performance than the UTDSP architecture. To put these results in perspective, we also calculated the relative performance of the UTDSP architecture when no attempt is made to exploit parallelism. We refer to this architecture as **UTDSP_serial**, and it can be thought of as a statically-scheduled, single-issue processor executing one operation per cycle. It also represents the UTDSP architecture's worst-case performance. Table 4 shows the actual cycle counts for each of the benchmarks when executed on **UTDSP_parallel**.

To measure the impact of using different architectures on cost, we also used an area-estimation model that we developed to estimate the area of the UTDSP datapath. The area-estimation model uses technol-

Kernels		
k1	fft_1024	41834
k2	fft_256	10332
k3	fir_256_64	32995
k4	fir_32_1	94
k5	iir_4_64	50
k6	iir_1_1	1643
k7	latnrm_32_64	6346
k8	latnrm_8_1	93
k9	lmsfir_32_64	14689
k10	lmsfir_8_1	85
k11	mult_10_10	2365
k12	mult_4_4	223

Applications		
a1	G721_A	158921
a2	G721_B	357678
a3	V32.modem	1197325
a4	adpcm	90625
a5	compress	18407
a6	edge_detect	2248024
a7	histogram	215313
a8	lpc	34781
a9	spectral	84743
a10	trellis	2603

Table 4: Cycle counts for benchmarks executing on **UTDSP_parallel**.

ogy-independent estimates of the area occupied by different datapath circuits and components, and a set of architectural parameters that describe a datapath.

The technology-independent constants are estimates of the area occupied by a single bit for different datapath elements. These include integer and floating-point ALUs, integer and floating-point multiplier/divider circuits, and register and memory cells. The architectural parameters are used to describe the overall structure of the datapath. These include the size, in bits, of data and instruction words; the number of words in instruction memory and each data-memory bank; the number of address, integer, and floating-point units; the number of integer and floating-point multiplier/divider circuits; and the number of address, integer, and floating-point registers. Using the area estimates and the architectural parameters, the model can be used to obtain a first-order estimate of the area occupied by the datapath. By changing the model's parameters, different datapaths, such as **DSP1**, **DSP2**, **UTDSP1**, and **UTDSP2**, can be modeled, and their areas can be estimated.

4. Results and Analysis

In this section, we examine the impact of tightly-encoded DSP instruction sets on performance, and compare it to the performance achieved with more flexible, VLIW-based architectures based on our UTDSP architecture. We also examine the cost of using a VLIW architecture in terms of datapath area.

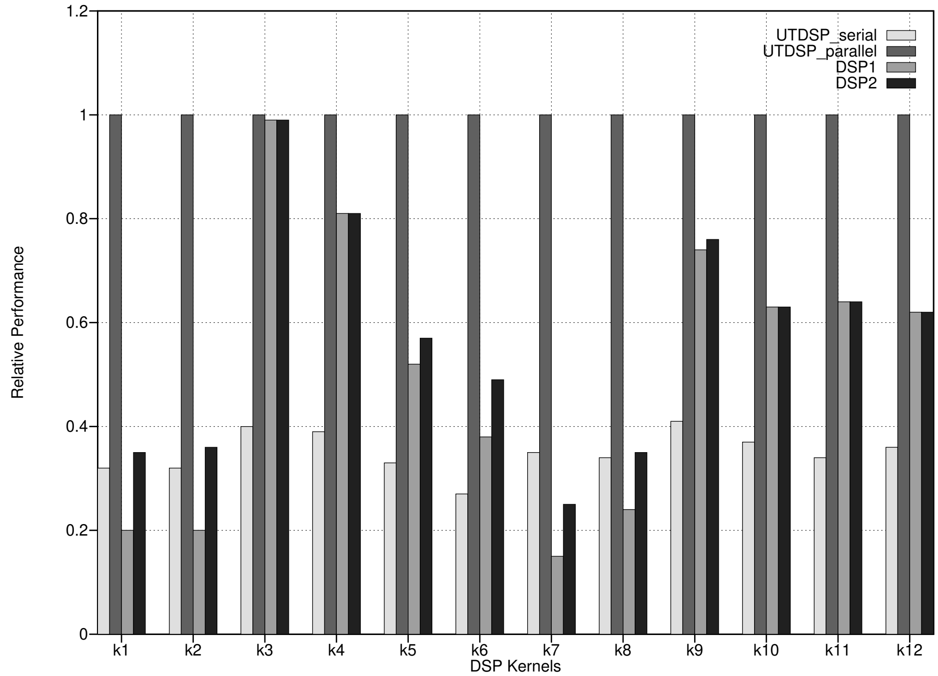
4.1. Impact on Performance

Figures 2 and 3 show the relative performance of **DSP1** and **DSP2** for the kernel and application benchmarks, respectively. Figure 2 shows that the relative performance of **DSP1** and **DSP2** varies considerably among the kernels. For example, the relative performance of **DSP1** is lower than **UTDSP_serial** for the `fft` and `latnrm` kernels (k1, k2, k7, and k8). Similarly, the relative performance of **DSP2** is lower than that of **UTDSP_serial** for the `latnrm_32_64` kernel (k7). On the other hand, the relative performance of both **DSP1** and **DSP2** is nearly identical to **UTDSP_parallel** for `fir_256_64` (k3). Overall, the relative performance of **DSP1** varies between 0.15–0.99 (0.51 on average). On the other hand, the relative performance of **DSP2** varies between 0.25–0.99 (0.57 on average).

Figure 3 shows that the relative performance of **DSP1** and **DSP2** is generally poor across the application benchmarks. On all applications except `histogram` (a7), the relative performance of **DSP1** is lower than **UTDSP_serial**. Even for `histogram`, the relative performance of **DSP1** is only slightly better than that of **UTDSP_serial**. Similarly, the relative performance of **DSP2** on `G721_A` and `edge_detect` (a1 and a6) is lower than **UTDSP_serial**. On all other applications, the relative performance of **DSP2** is either marginally better or slightly better than **UTDSP_serial**. Overall, the relative performance of **DSP1** varies between 0.16–0.65 (0.35 on average), while the relative performance of **DSP2** varies between 0.25–0.72 (0.56 on average).

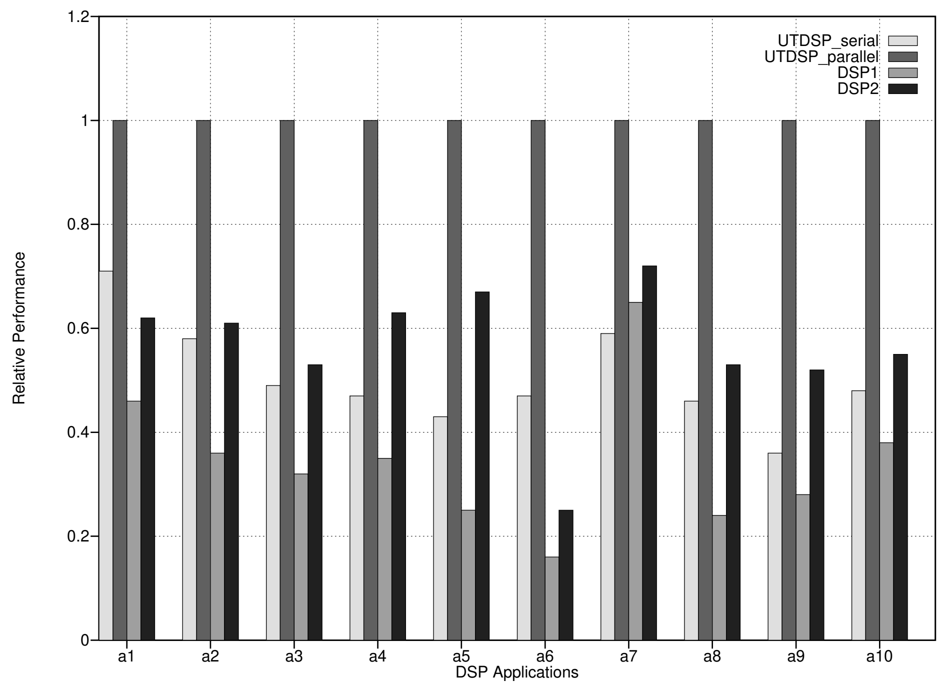
The poor performance of **DSP1** and **DSP2** compared to the UTDSP architecture is especially evident in the applications. This is due to the smaller number of data registers used in **DSP1** and **DSP2**, and to the restricted instances of parallelism that can be exploited by their instruction sets. The smaller number of data registers increases the number of memory instructions needed in a program and this increases execution time. Furthermore, the limited instances of parallelism supported by the instruction sets does not always match the instances of parallelism that occur in the applications, and this causes both DSPs to miss opportunities to exploit other instances of parallelism in the code. One reason for the better overall performance of **DSP1** and **DSP2** on the kernels is the higher incidence of parallelism that matches the parallelism supported by their instruction sets. Finally, for both the kernel and application benchmarks, the higher performance of **DSP2**, relative to **DSP1**, is due to its larger number of data registers and to its more flexible instruction set that enables it to exploit more instances of parallelism. Recall, however, that since the UTDSP compiler does not restrict the use of registers in **DSP1** and **DSP2**, these performance results are more optimistic than what would be achieved in reality.

To study the effect of using restricted, and hence less costly, VLIW architectures on performance, we measured the relative performance of **UTDSP1** and **UTDSP2**. Recall that these architectures contain a subset of the functional units used in the full-fledged UTDSP architecture, and are therefore more restricted in the amount of parallelism they can exploit. However, both architectures use the same number of registers as the UTDSP architecture, and can therefore help the compiler generate more efficient code.



k1 fft_1024
k2 fft_256
k3 fir_256_64
k4 fir_32_1
k5 iir_4_64
k6 iir_1_1
k7 latnrm_32_64
k8 latnrm_8_1
k9 lmsfir_32_64
k10 lmsfir_8_1
k11 mult_10_10
k12 mult_4_4

Figure 2: Relative performance of DSP1 and DSP2 on kernel benchmarks.



a1 G721_A
a2 G721_B
a3 V32.modem
a4 adpcm
a5 compress
a6 edge_detect
a7 histogram
a8 lpc
a9 spectral
a10 trellis

Figure 3: Relative performance of DSP1 and DSP2 on application benchmarks.

Figures 4 and 5 show the relative performance results on the kernels and applications for **UTDSP1** and **UTDSP2**.

As expected, Figure 4 shows that the greater number of registers in **UTDSP1** and **UTDSP2** significantly increases their relative performance across all the kernels. Overall, the relative performance of **UTDSP1** varies between 0.56–0.99 (0.74 on average). Similarly, the relative performance of **UTDSP2** varies between 0.58–0.99 (0.76 on average). For all kernels except fft_1024 and fft_256 (k1 and k2), **UTDSP1** and **UTDSP2** achieve identical levels of performance. This is due to both instruction-set architectures supporting similar instances of parallelism that match the ones occurring in the kernels.

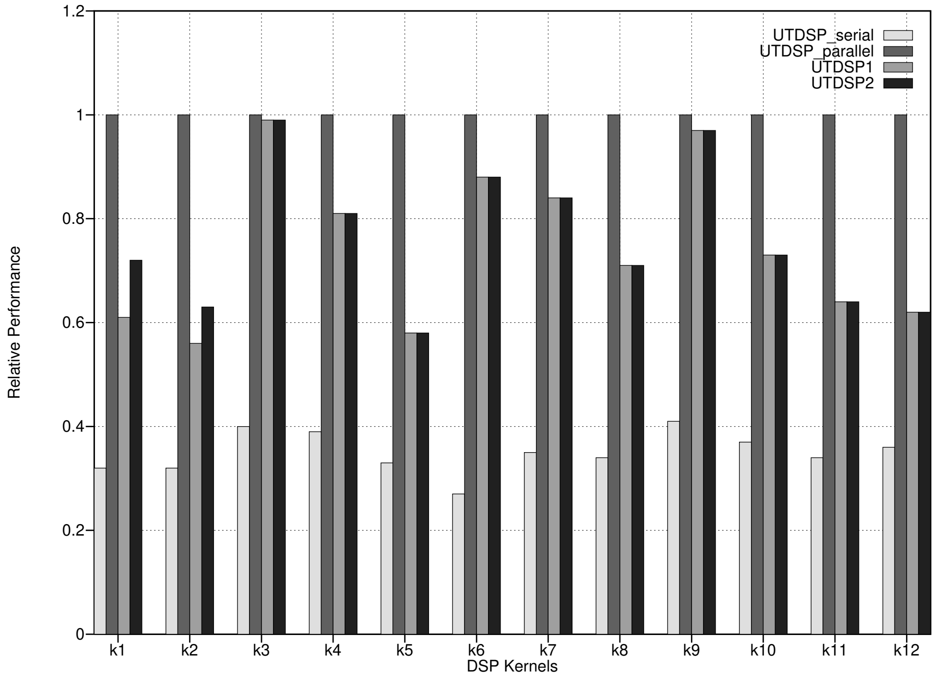
Figure 5 also shows that the greater number of registers used in **UTDSP1** and **UTDSP2** enable them to achieve significantly higher performance across all applications. This improvement is mainly due to the elimination of additional memory instructions. The relative performance of **UTDSP1** across all applications varies between 0.61–0.82 (0.70 on average). The relative performance of **UTDSP2** varies between 0.65–0.86 (0.76 on average). These results also show that the restricted parallelism remains a limiting factor to attaining higher levels of performance even after the number of registers is increased. Finally, the results show that **UTDSP2** achieves slightly higher levels of performance than **UTDSP1** for most applications, and this is due to the greater flexibility in its instruction set architecture.

These results also show that, even with restricted parallelism, the greater flexibility of the VLIW architecture and the large number of registers result in significant performance improvements over more traditional architectures. For example, the combined effect of these two factors enable the **UTDSP1** to execute an average of 1.5–2.0 times faster than the **DSP1**, and the **UTDSP2** to execute an average of 1.3 times faster than the **DSP2**. Recall, however, that since these results to not account for the limited orthogonality in the use of registers that exists in tightly-encoded instruction sets, they are much more conservative than what could be achieved with the real processors.

Finally, the results shows that, given the opportunity to exploit more parallelism than what is typically supported in traditional DSP architectures, the VLIW architecture can achieve even higher levels of performance. For example, by using its full hardware resources, the UTDSP architecture can execute an average of 1.96–2.86 times faster than DSPs with architectures similar to **DSP1**, and 1.75–1.79 times faster than DSPs with architectures similar to **DSP2**. This performance, however, is achieved at the expense of higher cost. In the next subsection we examine the impact of using a VLIW architecture on cost.

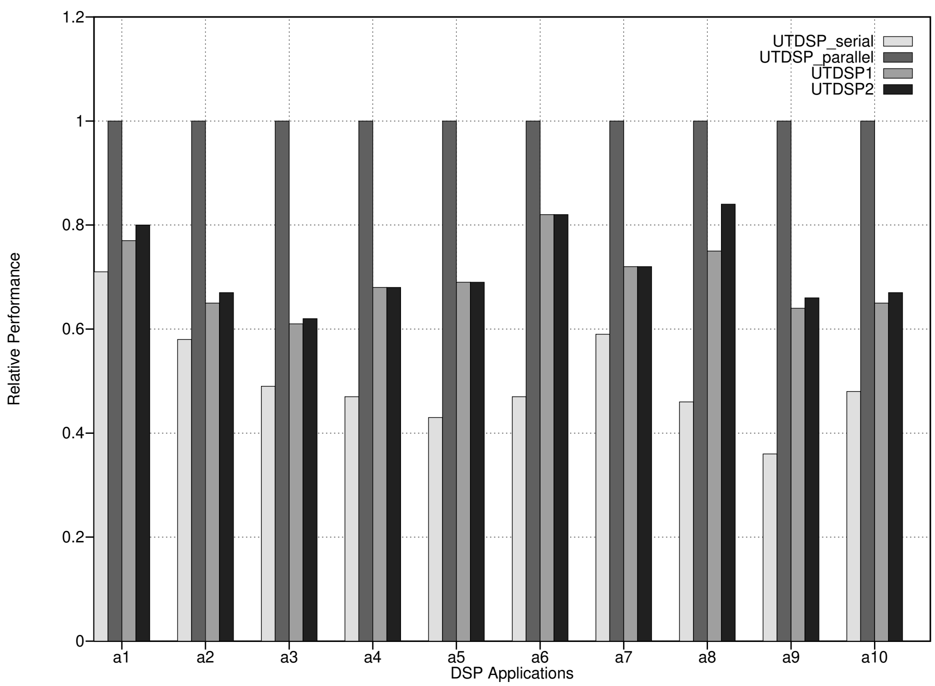
4.2. Impact on Cost

Table 5 shows the area estimate of the UTDSP model architecture, in both its restricted and unrestricted VLIW forms, and the more traditional DSPs. Recall that the datapaths of the restricted VLIW UTDSP models are identical to the traditional DSPs, except that they have more registers. The estimates are calcu-



k1 fft_1024
k2 fft_256
k3 fir_256_64
k4 fir_32_1
k5 iir_4_64
k6 iir_1_1
k7 latnrm_32_64
k8 latnrm_8_1
k9 lmsfir_32_64
k10 lmsfir_8_1
k11 mult_10_10
k12 mult_4_4

Figure 4: Relative performance of UTDSP1 and UTDSP2 on kernel benchmarks.



a1 G721_A
a2 G721_B
a3 V32.modem
a4 adpcm
a5 compress
a6 edge_detect
a7 histogram
a8 lpc
a9 spectral
a10 trellis

Figure 5: Relative performance of UTDSP1 and UTDSP2 on application benchmarks.

DSP	Total Area (mm^2)	Instruction Memory		Registers		Average Instructions Stored
		Area (mm^2)	% Total Area	Area (mm^2)	% Total Area	
UTDSP	9.82	2.86	29	0.40	4	436
DSP1	1.95	0.59	30	0.06	3	1068
UTDSP1	7.04	3.52	50	0.27	4	688
DSP2	3.02	0.63	21	0.30	10	576
UTDSP2	5.46	3.00	55	0.36	7	514

Table 5: Area estimates for $\lambda = 0.25$ µm.

lated for a feature size $\lambda = 0.25$ microns, and assuming there is no on-chip data memory in any of the models considered. Including the data memory area would only reduce the percentages, but would not change the conclusions because each processor would require the same amount of data memory. Finally, the size of instruction memory, in words, for each DSP is set to the average number of instructions generated for the application benchmarks.

The data in Table 5 shows that registers occupy a small percentage of total chip area. For the UTDSP model architecture, registers occupy only 4% of the area. Even when the hardware resources are constrained to restrict parallelism, the registers in **UTDSP1** and **UTDSP2** occupy 4% and 7% of the area, respectively. Similarly, for **DSP1** and **DSP2**, registers occupy 3% and 10% of the area, respectively. These results show that the argument presented by DSP designers, that reducing the number of registers reduces chip area and hence cost, is false. In fact, reducing the number of registers increases the number of instructions needed to move data between the registers and memory. In turn, this increases the program storage requirements and the area needed for instruction memory.

Table 5 also shows that instruction memory in **DSP2** occupies an average of 9% more area than instruction memory in **DSP1**. This is due to the longer instruction width of **DSP2** (48 bits) compared to **DSP1** (24 bits), even though **DSP2** needs to store an average of 46% fewer instructions than **DSP1**. This highlights the trade-off that must be made when designing DSP ISAs. **DSP2** uses more registers and allows more flexibility in exploiting parallelism than **DSP1**. Although this enables it to achieve an average of 1.60 times the performance of DSP1 on the application benchmarks, the higher performance is achieved at the expense of longer, less tightly-encoded instructions and larger memory area.

A similar trade-off may also be necessary when adopting a VLIW-based architecture such as the UTDSP. For example, even though the **UTDSP1** and **UTDSP2** achieve 1.3–2 times the performance of **DSP1** and **DSP2**, the memory needed to store their long instructions cause them to occupy 1.8–3.6 times the area. Similarly, the full fledged UTDSP model architecture achieves 2–3 times the performance of **DSP1** and **DSP2**, but its larger datapath and instruction memory bank cause it to occupy 3–5 times their area.

Architecture	Integer ALUs	Integer Mul	FP Adders	FP Mul	FP Div/ Sqrt	Datapath Area (mm^2)	Normalized Area
UTDSP	2	2	2	2	2	6.55	1.00
UTDSP1/ DSP1	1	1	—	—	—	1.30	0.20
UTDSP2/ DSP2	1	—	2	1	1	2.10	0.32
Customized UTDSP	1	1	1	1	1	3.39	0.52

Table 6: Datapath organizations for different architectures assuming $\lambda = 0.25\ \mu m$.

Whether the higher performance justifies the higher costs depends on the application, and must be considered by the designer. However, since cost is an important consideration in embedded systems, it is clear that implementing a VLIW architecture in its classical form is not very cost-effective.

Table 5 showed that the major factor contributing to the difference in area between the VLIW-based UTDSP and the more traditional **DSP1** and **DSP2** is instruction memory requirements. Since most fields in a long instruction are typically empty — on average, 70% of the space consumed by the long instructions generated by the UTDSP compiler is occupied by NOPs — storing long instructions in their basic format is very wasteful of memory. This suggests that a significant reduction in the storage requirements of long instructions can be achieved by using more efficient instruction encoding techniques. One approach to encoding long instructions involves storing them in compressed form, and decompressing them on-the-fly as they are fetched from memory [18]. Although this increases the complexity of the control circuitry, it is more than offset by the great reduction in memory needed to store a program. Furthermore, it does not affect the flexibility or orthogonality, and hence the performance, achieved with long instruction formats.

Another factor contributing to the difference in area is the organization of the datapath. Table 6 shows the organization and area of the full fledged UTDSP, **UTDSP1**, **DSP1**, **UTDSP2**, and **DSP2** datapaths, respectively. Though not shown in the table, each datapath uses a single program-control unit, two memory units, and two addressing units. From this table, it is clear that the greater area of the UTDSP datapath is due to its greater use of datapath components. This suggests that one way to reduce the area of the UTDSP datapath is to customize the datapath to the target application. Table 6 shows the components used and area occupied by one such customized version of the UTDSP datapath. This particular example shows that using single integer and floating-point functional units, instead of pairs, reduces the UTDSP datapath area by 48%. Naturally, the choice of datapath components depends on trading-off the performance requirements and cost constraints of the target application.

By efficiently encoding long instructions and customizing the datapath to use a minimal set of components, VLIW architectures can provide the high performance and low costs required by embedded DSP applications. When coupled with their ease of programming using HLLs, VLIW architectures become an ideal choice for implementing embedded DSP systems.

5. Summary and Conclusions

Programmable DSPs have traditionally been designed around tightly-encoded instruction sets. Since DSPs are mainly used in cost-conscious embedded systems, tightly-encoded instruction sets were used to improve code density, with the belief that this also reduced instruction memory requirements, and hence cost. Tightly-encoded instruction sets were also used to reduce instruction memory bandwidth, which was of particular concern when packaging was not very advanced. However, due to their tightly-encoded instruction sets, DSPs are difficult targets for optimizing high-level language compilers. For example, to save on the number of bits needed to encode instructions, most DSPs are accumulator-based and use very few registers. Furthermore, DSPs cannot exploit parallelism beyond that which is supported by their instruction sets. This makes it difficult for a compiler to generate efficient code and to fully exploit available parallelism. Thus, to achieve the most efficient and the most compact code, DSPs must still be programmed in assembly language. However, as DSP applications become more sophisticated, and as design and development cycles become shorter, it is becoming more difficult and more costly to develop applications using assembly languages. That is why DSP vendors are currently considering new instruction set architectures that can provide the high performance and low costs demanded by embedded DSP applications, while also being good targets for optimizing HLL compilers.

One architectural style that is particularly well suited for embedded DSP applications is the VLIW architecture. VLIW architectures provide the hardware resources and the instruction set flexibility that enable the exploitation of high levels of parallelism. They are also easy targets for optimizing HLL compilers, and make it easy for the compiler to generate efficient code.

This paper examined the performance benefits and cost trade-offs of using the VLIW-based UTDSP architecture compared to more tightly-encoded DSPs. By constraining the functional units used in the UTDSP model architecture, two commercial DSPs, based on the Motorola DSP56001 and the Analog Devices ADSP-21020, were modeled. The compiler was also modified to only generate combinations of parallel operations that are supported by the instruction sets of both architectures, and to use the number of registers available in each architecture. However, the orthogonality between available registers and machine operations was not constrained, leading to more optimistic performance results than could actually be achieved on these architectures.

To enable a fairer comparison to be made between the hardware-limited traditional DSP architectures and the UTDSP model architecture, two instances of the UTDSP architecture were modeled with the same hardware resources as the traditional DSPs. The main differences were that the VLIW-based modeled were given more registers, they were allowed more flexibility in exploiting parallelism, and they used long instructions instead of tightly-encoded ones.

The results showed that, for the kernel and application benchmarks, the restricted UTDSP architecture achieves 1.3–2 times the performance of the more traditional architectures. This is mainly due to their more flexible instruction set architecture and their larger set of data registers, which enable the compiler to exploit more parallelism and generate more efficient code. The results of an experiment measuring the effect of restricted register use in traditional DSP architectures also showed that the impact on performance is a factor of 1.4–2, suggesting that the actual performance achieved by the restricted UTDSP architecture is at least 1.8–2.8 times the performance of the more traditional architectures. However, since the experiment involved kernel code, the effect of more restrictive register use is expected to be more significant in real applications. When the unrestricted UTDSP architecture is used, it achieves 2–3 times the performance of the more traditional architectures. When considering the effect of the more restricted register use, the performance is closer to 2.8–4.2 that of the more traditional architectures.

The results also showed that the restricted UTDSP architectures occupy 1.8–3.6 times the area of the more traditional architectures, while the unrestricted UTDSP architecture occupies 3–5 times their area. This was mainly due to the large memory needed to store long instructions and, in the case of the unrestricted UTDSP architecture, to its larger datapath. Thus, in their classical form, VLIW architectures are too expensive to use in cost-conscious, embedded DSP applications. However, the cost of a VLIW architecture can be reduced by using an efficient method of encoding and storing long instructions, and by customizing the datapath to the functional and performance requirements of the target application. We are currently studying ways to reduce the cost of VLIW architectures and make them less costly to use in embedded DSP systems.

6. References

[1] Texas Instruments, *TMS320C62xx CPU and Instruction Set Reference Guide*, 1997.

[2] R. M. Keller, “Look-Ahead Processors,” *ACM Computing Surveys*, Vol. 7, pp. 66–72, ACM, 1975.

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

[4] *IEEE Micro*, special issue on multimedia extensions, Vol. 16, No. 4, IEEE, August, 1996.

[5] Gerald Cheong and Monica Lam, "An Optimizer for Multimedia Instruction Sets (A Preliminary Report)," *Proceedings of the Second SUIF Compiler Workshop*, `http://www-suif.stanford.edu/suifconf/suifconf2/`, 1997.

[6] Texas Instruments, *TMS320C80 (MVP) Parallel Processor User's Guide*, 1995.

[7] Robert P. Wilson et al., "SUIF: an Infrastructure for Research on Parallelizing and Optimizing Compilers," `http://suif.stanford.edu/suif/suif1/suif-overview/suif.html`, 1994.

[8] Reference removed for blind review.

[9] Reference removed for blind review.

[10] Ashok Sudarsanam and Sharad Malik, "Memory Bank and Register Allocation in Software Synthesis for ASIPs," *Proceedings of the International Conference on Computer-Aided Design*, pp. 388–392, IEEE/ACM, 1995.

[11] Markus Levy, "C Compilers for DSPs Flex Their Muscles," *EDN*, pp. 93–107, June 5, 1997.

[12] Motorola, *DSP56000/DSP56001 Digital Signal Processor User's Manual*, 1990.

[13] Texas Instruments, *TMS320C5x User's Guide*, 1993.

[14] Analog Devices, *ADSP-2100 Family User's Manual*, Third Edition, 1995.

[15] Analog Devices, *ADSP-21020 User's Manual*, Second Edition, 1995.

[16] Texas Instruments, *TMS320C3x User's Guide*, 1989.

[17] Motorola, *DSP96002 IEEE Floating-Point Dual-Port Processor User's Manual*, 1989.

[18] T. Conte, S. Banerjia, S. Larin, K. N. Menezes, and S. W. Sathaye, "Instruction Fetch Mechanisms for VLIW Architectures with Compressed Encodings," *Proceedings of the 29th Annual International Symposium on Microarchitectures (MICRO-29)*, pp. 201–211, IEEE/ACM, 1996.