CODEVELOPMENT OF THE TMS320C6X VelociTI ARCHITECTURE AND COMPILER

Ray Simar Jr.
Texas Instruments Incorporated
P.O. Box 1443, M/S 720
Houston, TX 77251-1443

ABSTRACT

Continuing dramatic improvements in semiconductor manufacturing processes are enabling radical new signal-processing architectures at the chip level. The development of these new architectures must be coupled with clearly defined target applications, a thorough analysis of applicable signal processing algorithms, and significant advancements in code-generation technology. The TMS320C6x development program involved the codevelopment of the VelociTI architecture, a new code-generation capability, and a large set of representative benchmarks.

1. INTRODUCTION

DSPs are designed to achieve high performance on DSP applications with minimum silicon cost. While this has been accomplished with previous DSP architectures, it has often come at the cost of architectural characteristics that overly constrain the compiler. To overcome this, the VelociTI DSP VLIW architecture was codeveloped with a new VLIW code-generation technology, and benchmarked by a set of representative DSP benchmarks. The first implementation of this architecture is the TMS320C6201 [1][2], the first general-purpose advanced VLIW DSP. Compared to a traditional DSP, VelociTI has many advantages that enable a high-level language compiler to more fully exploit the parallelism of the machine. This combination of architecture and compiler can achieve performance at or near peak for DSP applications.

As will be shown, for typical DSPs the specialization of the instruction set and the constraints of the pipeline protection generally prevent high-level language compilers from extracting optimal performance from the machine. The VelociTI DSP VLIW architecture was developed to overcome, as much as practical, many of the compiler limitations of traditional DSPs [3].

The following section introduces VelociTI and presents the first member of the VelociTI family, the TMS320C6201. Section 3 discusses the VelociTI pipeline. Section 4 explains the advantages of VelociTI for a compiler and why this architecture is a better target for a high-level language compiler than both traditional DSPs and traditional VLIWs. Section 5 describes *software pipelining*, a key compiler technique for scheduling loops on a VLIW [4] and which is especially well suited for DSP applications. Finally, Section 6 presents benchmark results that show performance of the compiler compared to hand-coded assembly on several DSP benchmarks.

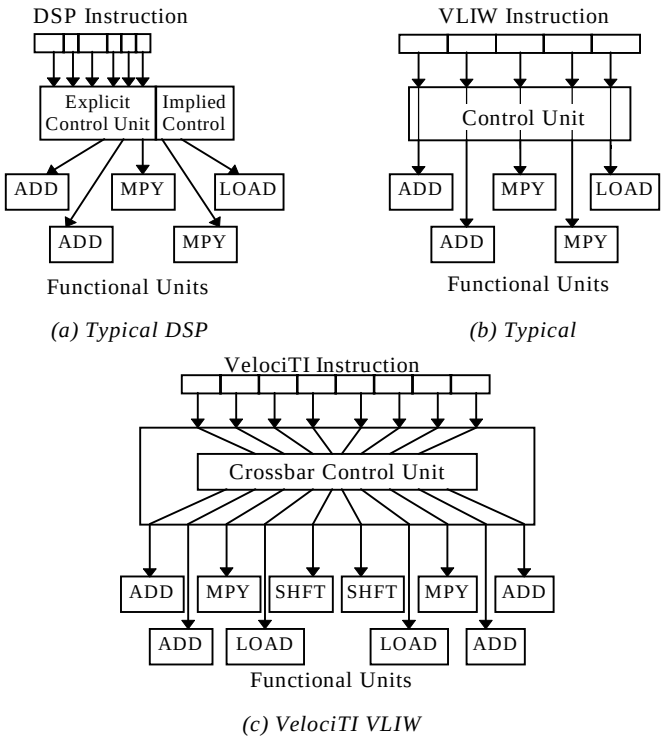


Figure 1: Architecture Comparison

2. VelociTI: ADVANCED VLIW FOR DSP

A typical DSP [5][6][7] instruction can simultaneously do one or more loads, one or more address arithmetic operations, a multiply, an add, and a decrement-and-conditional-branch. Each instruction typically uses between 16 and 32 bits to encode an opcode, one or two source operands (for memory operands this includes addressing modes, base registers, and offsets), immediate operands (constants) and a destination operand. In these few bits, obviously there is not enough room in the instruction to explicitly specify the full operation of each unit, so the instructions are often highly specialized. The operands are heavily restricted in the form of register constraints or limited addressing modes. Some operands may be missing altogether, in which case they are implied by the opcode or by the machine state (e.g. mode bits). Figure 1(a) illustrates the control structure of a typical DSP.

A VLIW [8] on the other hand, uses a wider instruction divided into fixed-length fields; each field fully specifies an operation, including opcode and operands, for one functional unit. The

```
ADD ||ADD || LOAD || MPY
SHIFT
MPY || MOVE || BRANCH
```

Fetch packet with 4 instructions in parallel, then 1 serial, then 3 parallel

```
MPY || MPY
MPY
ADD
SHIFT
ADD
STORE
BRANCH
```

Fetch packet with 2 instructions in parallel, remainder serial (8 total).

Figure 2: Flexible Serial or Parallel Execution

operations are simple, atomic, and completely independent. Figure 1(b) illustrates a typical VLIW.

The encoding scheme of the VelociTI advanced-VLIW architecture, illustrated in Figure 1(c), adds flexibility to previous VLIWs in three significant ways. First, the wide instruction (called a fetch packet) is evenly divided into fixed 32-bit atomic advanced-RISC instructions and, rather than being dedicated to a specific unit, each field contains a self-contained instruction for any unit. This is made possible by the crossbar control unit shown in Figure 1(c).

Second, rather than all the instructions in a fetch packet always executing in parallel, the parallelism within a fetch packet is programmable from fully serial to fully parallel. By allowing serial execution within the instruction packet, VelociTI reduces the code size penalty that has been a major drawback of previous VLIWs for embedded applications. Figure 2 illustrates the flexible serial/parallel execution model.

Finally, each advanced-RISC instruction in the VelociTI instruction packet can be executed based on an independent condition. The ability to make any instruction conditional increases performance by reducing pipeline delays associated with branching and allowing for speculative execution which increases the effective parallelism. This technique is also known as predication.

The TMS320C6201 is the first member of the VelociTI family. Running at an initial clock rate of 200 MHz, the 'C6201 executes up to 1,600 MIPS (Million Instructions Per Second). The 'C6201 CPU has 8 functional units and 32 registers, both evenly partitioned between two identical 32-bit data paths. Each datapath consists of sixteen 32-bit general-purpose registers and four functional units. In addition each data path has a dedicated bus that can load or store a 32-bit value to or from memory on each cycle. The data paths have the ability to access values from each other via cross-path busing. The CPU can fetch a 256-bit instruction packet on each cycle, which is divided into eight 32-bit advanced-RISC instructions that can be executed all in parallel, all serially, or in any combination (Figure 2).

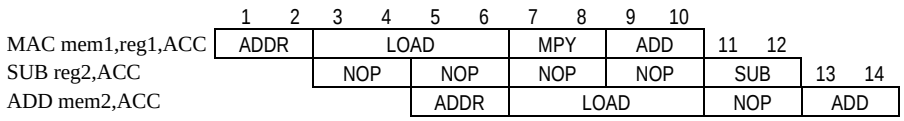
The TMS320C6201 includes 1Mbit of on-chip SRAM memory split evenly between program cache and data memory. The program cache can also be configured as a statically mapped block of program memory. An External Memory Interface (EMIF) provides for a glueless interface to various synchronous and asynchronous memory devices. Also on chip are a number of peripheral devices including a host access port, a phase-lock loop clock-generator, and multi-channel DMA and multi-channel serial ports.

3. THE VelociTI PIPELINE

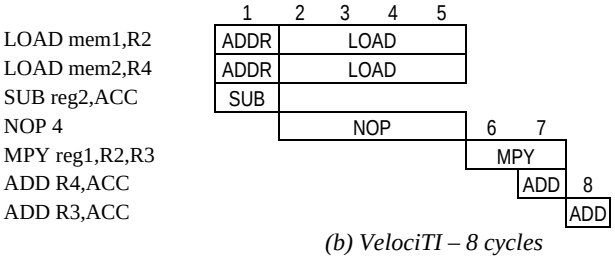
Typical DSPs have a shallow, protected pipeline that overlaps the execution of multiple instructions. The pipeline achieves single-cycle throughput by hiding memory latencies. However, the arithmetic functional units are not typically pipelined, so the cycle time of the machine is limited to the throughput of the slowest unit, often the multiplier. Most instructions can operate from memory, so even instructions that don't read memory require otherwise unnecessary memory stages.

Figure 3(a) illustrates the computation of a sample expression using the pipeline of a typical DSP. The multiply and add are performed with one MAC instruction. The pipeline length is fixed, so each stage must be as long as the longest operation (MPY), and instructions that don't load from memory (SUB) have memory stages. NOPs are inserted by the hardware.

Because of the requirement for assembly language programmability, the pipeline of a typical DSP must be protected: the hardware manages functional unit latencies and inserts delay slots where necessary rather than the programmer having to perform this complex task. Pipeline protection requires additional control logic, increasing cost, design time, and even the machine's cycle time if the control mechanism becomes part of a critical speed path.



(a) Typical DSP – 14 cycles



(b) VelociTI – 8 cycles

*Figure 3: Comparison of pipeline schedule for typical DSP and the VelociTI architecture to compute this expression: (mem1 * reg1) + ACC + mem2 – reg2*

The pipeline of VelociTI is deepened to eliminate traditional bottlenecks. The functional units themselves are pipelined so that a new operation can be issued, and a new result obtained, from each unit in each cycle. The effect is that certain operations, such as loads and multiplies, have more cycles of latency. But these deeper instructions do not create a bottleneck for simpler operations by necessitating a longer cycle time.

Figure 3(b) illustrates the computation of the same sample expression scheduled on the VelociTI pipeline. Although loads and multiplies take the same amount of time as the typical DSP, the stages are shorter. The pipeline is variable length, so instructions that do not read memory complete sooner. The compiler has complete flexibility over when operations occur in the pipeline; for example, the two loads can be scheduled together to shorten the critical path.

4. COMPILING FOR VelociTI

In order to achieve high performance on DSP code written in a high-level language such as C, a compiler must do three things. First, it must simplify the program to reduce the number and cost of the operations required to execute it. Then, it must discover parallelism from the serial code. These two tasks involve a series of steps to analyze and transform the source code [9][10]. Finally, it must be able to generate machine code for the target machine in such a way that maximizes parallelism within the limited resources of the hardware. The VelociTI compiler development focused on addressing this code generation problem.

To illustrate this, first consider the typical problem a compiler faces in generating code for a traditional DSP with significant instruction-level parallelism. The code generation task consists of three major subtasks: selecting instructions, scheduling them, and allocating registers for program variables and temporaries.

Each of these problems is significantly difficult to solve automatically. In fact, optimal instruction scheduling and register allocation have each been shown to be *NP-complete*. An *NP-complete* problem is one whose solution is believed to have exponential complexity and thus rendering the guarantee of optimal solutions impractical [11][12]. Fortunately, years of compiler research have led to practical and effective heuristic techniques.

Since the problems of instruction selection, scheduling, and register allocation are always somewhat interdependent, and since it is impractical to solve them together, a central problem in compiler design is to decide in what order to perform the steps. This is known as the *phase ordering problem* [13]. Another way to view the issues involved in the phase ordering problems is to consider the idea of "separation of concerns". The different phases have different issues or concerns associated with them and the more these concerns can be kept separate from other phases, the better the job the compiler can will do.

Since the instructions of traditional DSPs tend to be complex, tightly encoded, and functional unit specific, their selection tends to constrain scheduling and register allocation. With fewer registers, it's often impossible to allocate registers once scheduling is complete. Allocation may even affect instruction selection if different functional units (and therefore different instructions) must be chosen to operate on different types of registers.

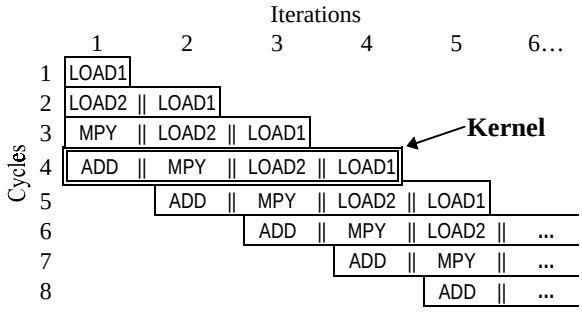


Figure 4: Software Pipelining

The VelociTI architecture helps this problem in two ways. First, it reduces architecture-imposed interdependencies between instruction selection, scheduling, and allocation. Second, it simplifies each step, increasing the likelihood of an optimal solution.

5. LOOPS AND SOFTWARE PIPELINING

Loop performance is an important factor for any application, especially in DSP which tends to be dominated by loops. Just a few examples include functions such as FIR and IIR filters, correlation, mean-square error calculation, DCTs and FFTs.

To achieve performance on loops with a VLIW architecture, it is necessary to discover as much fine-grained parallelism as possible. A common source of parallelism in loops is across the loop iterations. *Software pipelining* is a technique to exploit this parallelism by overlapping the execution of multiple iterations so that independent operations from different iterations can be scheduled in parallel [4][14]. This is accomplished by initiating each iteration before the previous one completes. Software pipelining is especially well suited to VLIW and DSP because of the degree of parallelism available and the ability to schedule functional units independently.

Software pipelining can effectively increase the throughput of a program in the same way that a hardware pipeline increases the throughput of a machine. By overlapping execution of the iterations, the program can generate a new result every cycle even if the serial execution of a given iteration takes several cycles. This is the key advantage of VelociTI's tradeoff: its deeper pipeline allows shorter cycle times which, because of software pipelining, increases overall throughput.

Figure 4 illustrates a software pipelined schedule for a loop with four operations: LOAD1, LOAD2, MPY, ADD. In each cycle, a new iteration begins with the issue of LOAD1. In cycle 4, a steady state is reached in which instructions from 4 iterations are executing in parallel. The steady state instruction packet (or packets) is called the kernel because it becomes the body of the loop. This schedule produces a new result every cycle.

Once the loop is ready to be scheduled, the compiler must first decide to what degree the loop will be overlapped, that is, how often to initiate each new iteration. This is known as the

Benchmark	Asm cycles	C cycles	Ratio C / Asm
VSELP autocorrelate matrix	994	991	1.00
Carrier Oscillator	69	65	0.94
MPEG2 Absolute Distance	376	557	1.48
VSELP Dot Product	29	32	1.10
VSELP FIR	226	260	1.15
IIR Cascaded	58	69	1.19
CELP Impulse	1367	1897	1.39
JPEG DCT	245	240	0.98
Fwd Lattice Filter Synth	40	52	1.30
LMS	69	77	1.12
VSELP MAC	49	53	1.08
Max Index	33	42	1.27
VSELP Min Err	1173	1576	1.34
MPEG DCT	247	282	1.14
VSELP Orthoganalization	50	58	1.16
Harmonic Series Oscillator	106	140	1.32
Viterbi V32 Decoder	79	113	1.43
Vector Quantizer MSE	267	269	1.01
Average			1.19

Table 1: Benchmark Results

Initiation Interval, or II. Smaller values of II result in higher throughput. The minimum possible value of II (MII) is bounded by resources and dependency [4].

After determining MII, the compiler attempts to find a schedule for the loop kernel that fits in MII cycles, while adhering to machine and dependency constraints. The VelociTI compiler uses a technique called modulo scheduling, so named because an instruction scheduled at cycle k will execute in parallel with all instructions scheduled at cycle k modulo II [13][14].

6. BENCHMARKING

For benchmarking the compiler and the VelociTI architecture, we use a suite of more than fifty DSP kernel benchmarks, including key algorithms from telecom, datacom, and wireless applications. Each benchmark consists of the C code for the algorithm, and the equivalent algorithm implemented in hand-coded assembly language.

Table 1 presents a sample of the benchmark results. The table shows the execution time in cycles for both the assembly and C versions, and the ratio of C to assembly. For this set of benchmarks, on average, the compiler came within 19% of hand-coded assembly performance. This means that, on this set of benchmarks the compiler is, on average, 84% of the efficiency of an assembly language programmer. For compiled code performance on real DSP kernels, 84% is considered remarkably good. Experience with more traditional DSPs suggests that 50%–20% efficiency is typical.

7. CONCLUSIONS

The TMS320C6x development program focused on the codevelopment of key technologies in DSP architectures and code generation benchmarked by representative DSP benchmarks. The resulting high-level language compiler technology uses advanced compilation techniques like software pipelining to far surpass the performance of existing DSPs and to achieve performance that rivals hand-coded assembly.

REFERENCES

[1] Texas Instruments *TMS320C62xx CPU and Instruction Set Reference Guide*. www.ti.com/sc/docs/dsps/products/c6x/

[2] Dillon, T. J. Jr. “The VelociTI Architecture of the TMS320C6x”. *ICSPAT*, 1997.

[3] Simar, R. “DSP Architectures, algorithms, and Code-Generation: Fission of Fusion?”. *IEEE International Conference on Innovative Systems in Silicon*, pp. 220-227.

[4] Lam, M. “Software Pipelining: An effective scheduling technique for VLIW machines”. *SIGPLAN ’88 Conference on Programming Language Design and Implementation*, June 1988, pp. 318-328.

[5] Texas Instruments. Various TMS320 DSP User’s Guides. www.ti.com/sc/docs/dsps/products.htm

[6] Motorola. Various DSP Family Manuals.

[7] Analog Devices. Various ADSP User’s Manuals.

[8] Fisher, J. A. *Very Long Instruction Word Architectures*. Yale University, 1983.

[9] Aho, A.V., Sethi, R., and Ullman, J.D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[10] Padua, D.A., and Wolfe, M.J. “Advanced compiler optimizations for supercomputers”. *Communications of the ACM*, 29(12), December 1986, pp. 1184-1201.

[11] Coffman, J.R. *Computer and Job-Shop Scheduling Theory*. John Wiley, 1976.

[12] Garey, M.R., and Johnson, D.S. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, 1979.

[13] Davis, A. L., Stotzer, E. J., Tatge, R. E., Ward, A. S. “Approaching Peak Performance with Compiled Code on a VLIW DSP”, *ICSPAT*, 1997

[14] Rau, B.R., and Glaeser, C.D. “Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing”. *Proceedings of the 14th Annual Workshop on Microprogramming*, October 1981, pp. 183-198.