Nat Seshan

High VelociTI Processing

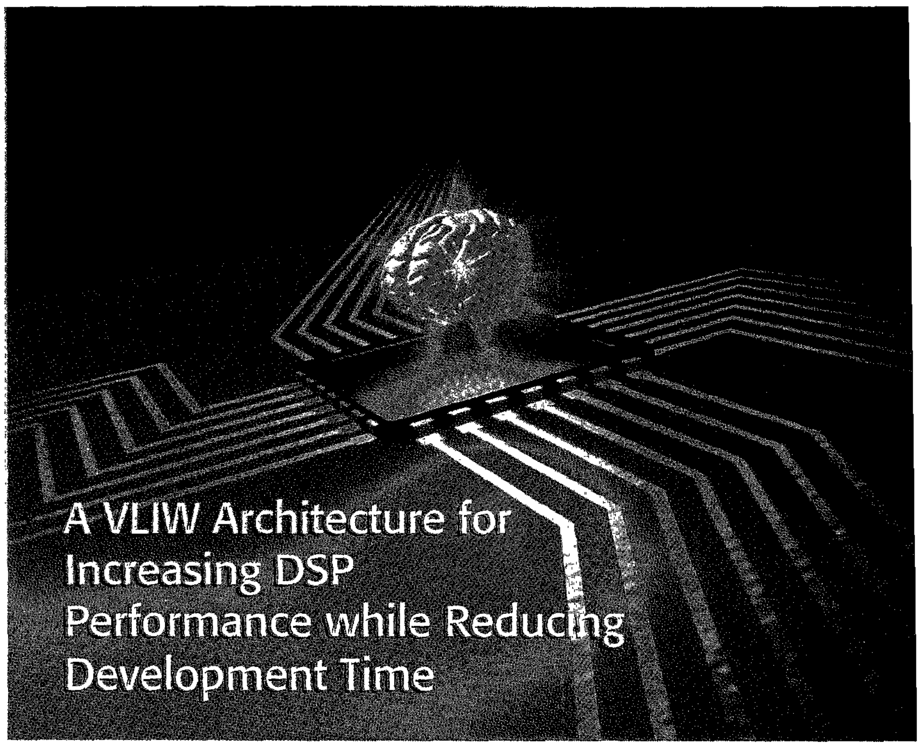
The Texas Instruments VelociTI architecture is a very long instruction word (VLIW) architecture [1]. The TMS320C6x family of digital signal processors (DSPs) is the first to employ the VelociTI architecture, with the TMS3206201 (C6201) being the first device in this family. The C6201 is based on the fixed-point TMS320C62x (C62x) CPU. The C62x has eight independent functional units running at 200 MHz for peak execution of 1600 million instructions per second (MIPS). The C6201 also contains a memory architecture and peripheral set suitable for a variety of applications including multichannel modem, multichannel vocoding for telephony and wireless, single and multichannel asymmetric digital subscriber (ADSL) modems, and imaging. The second VelociTI CPU is the floating-point 167 MHz TMS320C67x (C67x). The C67x is object-code compatible with the C62x and has six floating-point units overlaid on the existing fixed point units for a total of 1 billion floating-point operations per second (1 GigaFLOP or GFLOP) or 1333 MIPs.

This article describes the VelociTI VLIW architecture and discusses the C62x, C67x, C6201, and the VelociTI development tools. An overview of the VelociTI including architectural principles, data path, instruction set, and pipeline operation is presented, and both the C62x fixed-point CPU and the recently disclosed C67x floating-point CPU are described. A summary of C62x benchmark performance is also presented. The chip-level support outside the CPU that allows the C6201 to operate in a variety of high-performance DSP environments is also described. An overview of the C6x development environment is also given, demonstrating the breadth of the development environment and illustrating the programming methodology. We conclude with a performance analysis of the C compiler. Note that the term C6x is used in statements that apply to both the C62x and C67x CPUs.

TMS320C6x VelociTI CPU

VelociTI Principles

Eight principles underlie the VelociTI CPU architecture upon which the C6x CPU is based. These focus on increasing DSP performance while maintaining ease of programming and reducing application development time by allowing creation of a high-performance compiler.


A VLIW Architecture for Increasing DSP Performance while Reducing Development Time

©1995 Ron Lowery/The Stock Market

Parallelism

VelociTI, like other VLIWs, allows parallel fetch, decode, and execution of multiple instructions that compose the VLIW instruction word. During execution, each instruction is performed on a single functional unit. In the C6x, eight 32-bit instructions supply control for eight independent functional units.

Pipelining of Critical Speedpaths

The VelociTI architecture sets the simplest of CPU instructions to determine the cycle time for the processor.

In the C62x, the critical path is the time for a register-to-register ALU operation such as an ADD instruction. More complex instructions, such as multiply, are implemented with a one-cycle latency. To access high-performance, synchronous on-chip memories, instruction fetch and data access are performed in multiple pipeline stages. This pipelining allows the C62x CPU to operate at 200 MHz—or 1600 MIPS. Similarly, the floating-point operations on the C67x are pipelined to achieve its 167 MHz operating frequency.

Reduced Instruction Set Computer (RISC)

The VelociTI instruction set consists of simple, atomic, and completely independent instructions. DSP algorithm performance results from program compilation techniques such as software pipelining [2] and loop unrolling. The RISC architecture provides ease of CPU design, while providing flexibility for high-performance algorithms not yet conceived at the time of conception of the CPU architecture. Previously, more DSP-specific instruction sets have inhibited compilers from optimizing performance [3, 4].

Load-Store Architecture

As an extension of its RISC architecture, VelociTI is a load-store architecture. Memory operations have been decoupled from arithmetic operations. This feature also lowers the number of data fetches for a particular algorithm, and thus CPU power consumption.

Orthogonality

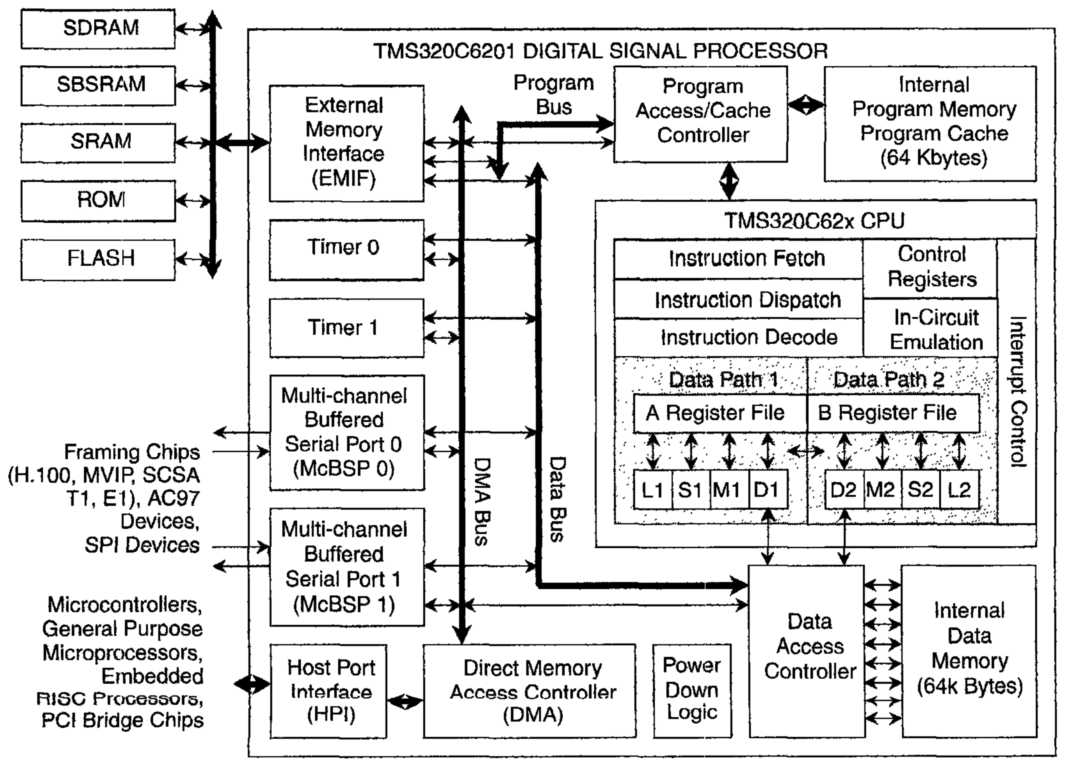
The most frequent instructions can be executed on the largest number of functional units. In the C6x, the CPU is divided into two identical data paths. Thus, every instruction can execute on at least two functional units. The most frequent instructions such as ADD and SUB can execute on six functional units. Like the instruction set, the register file is highly orthogonal. Any register can be an operand to any instruction or any type of functional unit.

Determinism

The VelociTI pipeline is unprotected and thus fully exposed to the compiler. Run-time interdependencies such as pipeline interlocks between phases used in other DSPs are difficult to predict at compile-time. The VelociTI CPU model at compiler time fully reflects the execution and completion order of instructions at run time. Lack of pipeline interlocks also reduces CPU complexity and design time while avoiding speed-limiting paths from the control mechanism.

Conditional Instructions

Every VelociTI instruction is conditional. Conditional instructions avoid branch latencies.


SDRAM
SBSRAM
SRAM
ROM
FLASH
TMS320C6201 DIGITAL SIGNAL PROCESSOR
External Memory Interface (EMIF)
Program Bus
Program Access/Cache Controller
Internal Program Memory Program Cache (64 Kbytes)
Timer 0
Timer 1
TMS320C62x CPU
Instruction Fetch
Instruction Dispatch
Instruction Decode
Control Registers
In-Circuit Emulation
Interrupt Control
Data Path 1
Data Path 2
A Register File
B Register File
L1
S1
M1
D1
D2
M2
S2
L2
Multi-channel Buffered Serial Port 0 (McBSP 0)
Multi-channel Buffered Serial Port 1 (McBSP 1)
DMA Bus
Data Bus
Framing Chips (H.100, MVIP, SCSA T1, E1), AC97 Devices, SPI Devices
Microcontrollers, General Purpose Microprocessors, Embedded RISC Processors, PCI Bridge Chips
Host Port Interface (HPI)
Direct Memory Access Controller (DMA)
Power Down Logic
Data Access Controller
Internal Data Memory (64k Bytes)

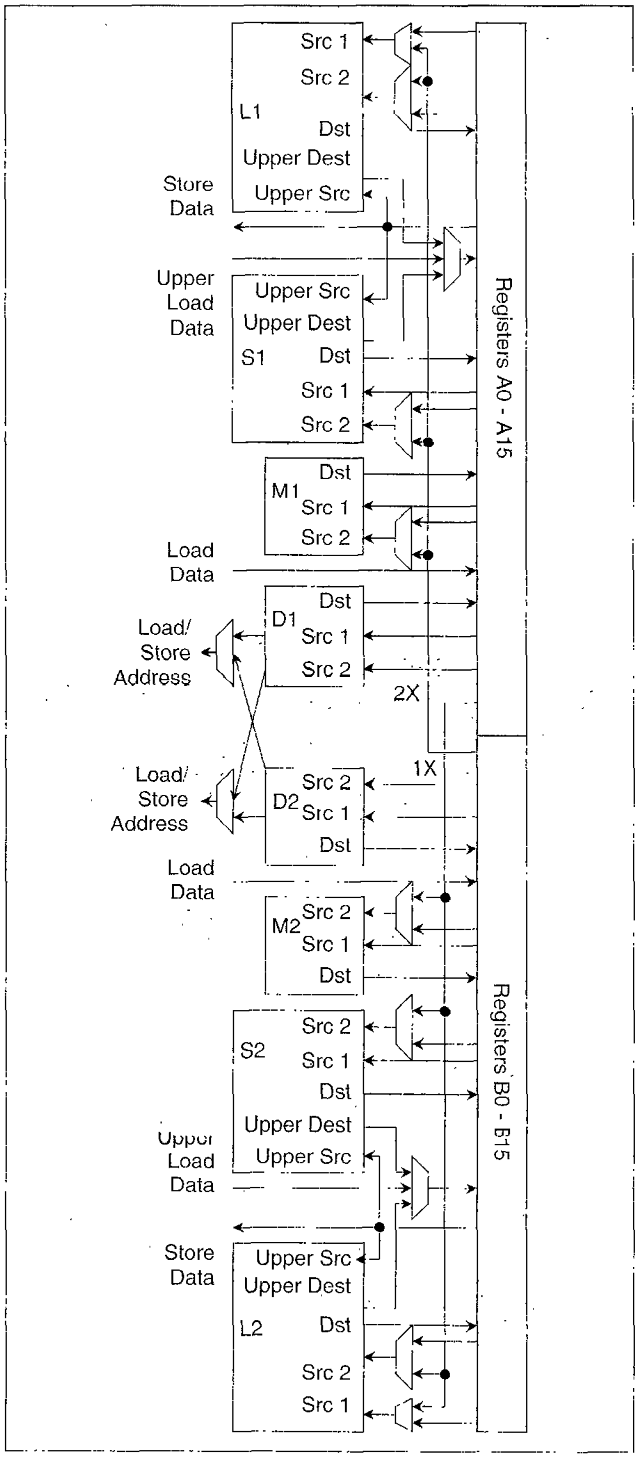
▲ *1. TMS320C6201 block diagram.*

Instruction Packing

A novel instruction packing technique allows the VelociTI architecture to achieve code size comparable with scalar RISC processors.

Data Paths

The C6x has two identical data paths with four functional units each (Fig. 2). Each data path has 16 32-bit registers. The register files have 16 ports (10 read ports and six write ports). In addition, each data path has a cross path (1X, 2X) to read operands from the other file. Note that addresses from one data path may be used to load and store values to the other data path. 40-bit integer operands as well as C67x double-precision operands use two registers for storage.



▲ 2. *C6x CPU data paths.*

Functional Units

Each functional unit can begin execution of a new instruction every cycle. The four functional units in each data path are actually a grouping of operational subunits. Subunits on the same functional unit share the same interconnect to the register file to save register porting. This port sharing reduces CPU area since register file area grows with the square of the number of ports. Also, reduced porting allows the processor to achieve its operating frequency. Tables 1 and 2 show how these subunits map to the functional units. The total number of subunits in both data paths is 24 for the C62x and 36 for the C67x.

Data Types

In general, the functional units perform 32-bit integer operations. The S-Units and L-Units also perform 40-bit integer operations to handle overflow. The integer multipliers in the M-Units generate 32-bit outputs from 16-bit inputs. The compiler types are 40bit `longs`, 32-bit `ints`, 16-bit `shorts` and 8-bit `chars`. In cases where source operands of different types are used, signed (sign extension) or unsigned (zero-filling) of operands can be specified. The C67x adds both single (32-

Table 1. C62x functional units and subunits.

L	S	D	M
Integer Adder	Integer Adder	Integer Adder	Integer Multiplier
Logical	Logical	Load-Store	
Bit Counting	Bit Manipulation		
	Shifting		
	Constant		
	Branch Control		

Table 2. Additional C67x functional units and subunits.

L	S	D	M
FP Adder	FP Compare		FP Multiplier
FP Conversion	FP Conversion		
	FP Seed Generation		

bit) and (64-bit) IEEE format floating-point computations including a variety of rounding modes as well as exception status generation.

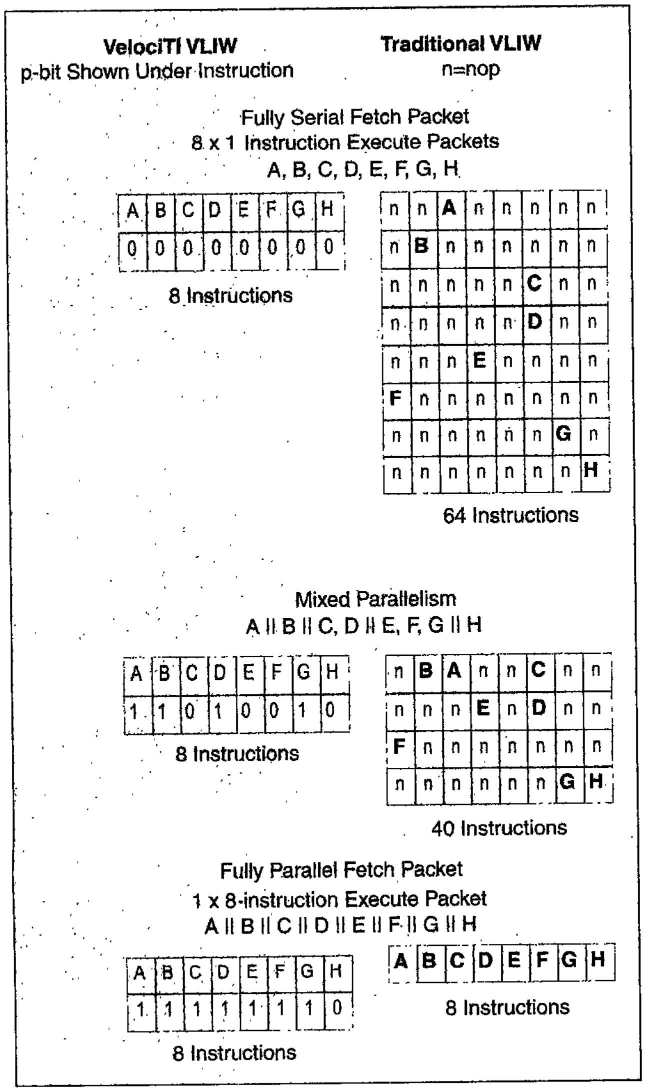
Instruction Set

Tables 3 and 4 summarize the C62x and C67x instruction sets and the mapping of these instructions to functional units. This mapping is done at code-generation time. Note that the most common integer operations can occur on four to eight units. Also, each functional unit has special-purpose features:

▲ **Saturated Operations (L-, S-, M-Units):** These provide for saturation on overflow. Such functions are used heavily in bit-exact ITU standards in the areas of voice coding for digital cellular and digital telephony.

▲ **Subtract Conditional (L-Unit):** Integer division step.

▲ **Bit Counting (L-Unit):** Bit counting allows counting of runs of bits used in image coding. Bit counting also allows normalization for scaling counts in block floating-point algorithms.


VelociTI VLIW
p-bit Shown Under Instruction
Traditional VLIW
n=nop
Fully Serial Fetch Packet
8 x 1 Instruction Execute Packets
A, B, C, D, E, F, G, H
8 Instructions
64 Instructions
Mixed Parallelism
A || B || C, D || E, F, G || H
8 Instructions
40 Instructions
Fully Parallel Fetch Packet
1 x 8-instruction Execute Packet
A || B || C || D || E || F || G || H
8 Instructions
8 Instructions

▲ *3. VelociTI instruction packing.*

Table 3. C62x instruction set.

Unit	Subunit	Mnemonic	Action
L,S,D,M	Int Add.	ZERO	Zero Register
L, S, D	Int Add.	ADD(U)	Add (Unsigned)
L, S, D	Int Add.	SUB(U)	Subtract (Unsigned)
L,S,D	Int Add.	MOV	Move
L, S	Logical	NEG	Negate
L, S	Logical	AND	Bitwise And
L, S	Logical	XOR	Bitwise Exclusive-Or
L, S	Logical	OR	Bitwise OR
L, S	Logical	NOT	Invert all Bits
L	Int Add.	SUBC	Subtract Conditional Divide Step
L	Int Add.	SADD	Add and Saturate
L	Int Add.	SSUB	Subtract and Saturate
L	Int Add.	ABS	Absolute Value and Saturate
L	Int Add.	CMPEQ	Compare for Equals
L	Int Add.	CMPGT(U)	Compare for Greater Than (Unsigned)
L	Int Add.	CMPLT(U)	Compare for Less Than (Unsigned)
L	Bit Cnt.	LMBD	Count 1's or 0's from left
L	Bit Cnt.	NORM	Count Redundant Sign Bit
S	Int Add.	ADDK	Add Dual 16-bit Constant
S	Int Add.	ADD2	Add Dual 16-Bit Values
S	Int Add.	SUB2	Subtract Dual 16-Bit Values
S	Const.	MVK	Load with 16-bit Signed Constant
S	Const.	MVKH	Load 16-MSBs w/Constant. Maintain 16-LSBs.
S	Shifter	SHL	Shift Left
S	Shifter	SSHL	Shift Left and Saturate Result
S	Shifter	SHR(U)	Shift Right (Unsigned)
S	Bit Man.	EXT(U)	Extract Field (Unsigned)
S	Bit Man.	SET	Set Field
S	Bit Man.	CLR	Clear Field
S	Br/Ctrl.	B	Branch Displacement or Register
S	Br/Ctrl	MVC	Move Control Reg. to/from Register
D	Int Add.	ADDA(B,H,W)	Add With Prescale for Data Type (Byte, Half-Word, Word)
D	Int Add. Ld/st	LD(B,H,W)(U)	Load (Byte, Half-Word, or Word) (Unsigned)
D	Int Add. Ld/st	ST(B,H,W)	Store (Byte, Half-Word, Word)
M	Int Mult.	MPY	Int 16x16 ⇒32 Can Choose Unsigned or Signed MSBs or LSBs of Either Operand
M	Int Mult.	SMPY	Multiply 2 Q15 and generate a Q31 Result. Saturate.
-	-	NOP n	Multicycle nop
-	-	IDLE	Wait for Interrupt

▲ **Integer Comparison (L-Unit):** Rather than having multiple status bits for each functional unit, the C6x employs explicit comparison instructions that generate a 1 or 0 result in a general-purpose register. Placing the result in a general-purpose register allows the value to be used directly in computation. Examples of this use include generation of cyclical redundancy codes (CRCs) as well as generation of bit-decision vectors in Viterbi Traceback algorithms.

▲ **Dual 16-Bit Pair Arithmetic (S-Unit):** The ADD2 and SUB2 instructions add two 32-bit registers while inhibiting the carry between the 15th and 16th bit positions. These instructions allow increased add/subtract throughput of 16-bit values. Examples of use include vector addition and complex addition. This feature greatly improves FFT performance.

▲ **Bit Manipulation (S-Unit):** Bit-field extraction, setting, and clearing.

Table 4. C67x instruction set additions.

Unit	Subunit	Mnemonic	Action
L	FP Adder	ZERO (SP/DP)	FP Zero Register
L	FP Adder	ADD(SP/DP)	FP Add
L	FP Adder	SUB(SP/DP)	FP Subtract
M	FP Mult.	MPY(SP/DP)	FP Multiply
M	FP Mult.	MPY24(H)	Int 24x24⇒32 L(M)SBs Multiply
M	FP Mult.	MPYI(D)	Int 32x32⇒32(64) Multiply
L	FP Conv.	DPSP	Doub. Prec. to Single Prec. Conv.
L	FP Conv.	DP (INT/TRUNC)	Doub. Prec. to Int/Truncated Int Conv.
L	FP Conv.	SP (INT/TRUNC)	Single Prec to Int/Truncated Int Conv.
L	FP Conv	INT(DP/SP)(U)	(Unsigned) Int to FP Conversion
S	FP Cmp.	CMPEQ(SP/DP)	FP Compare for Equals
S	FP Cmp.	CMPGT(SP/DP)	FP Compare for Greater Than
S	FP Cmp.	CMPLT (SP/DP)	FP Compare for Less Than
S	FP Cmp.	ABS(SP/DP)	FP Absolute Value
S	FP Seed	RCP(SP/DP)	FP Reciprocal Seed Generation
S	FP Seed	RSQR(SP/DP)	FP Reciprocal Sqrt Seed Generation
D	Int Adder	ADDAD	Add with Prescale for Double Word
D	Int Adder Ld/St	LDDW	Load Double Word (64-Bit)

DP = Double Precision, SP = Single Precision

▲ **Memory Data Types (D-Unit):** The C6x can load or store bytes, 16-bit half-words, and 32-bit words. Byte and half-word loads are sign-extended (signed) or zero-filled (unsigned). In addition, certain registers have the option of having circular arithmetic performed during address calculation (for circular buffers). The C67x can also perform 64-bit load operations and the associated address computations.

▲ **Constant Generation (S-Unit):** 32-Bit constants may be loaded with successive MVK and MVKH instructions.

▲ **Seed Generation (C67x S-Unit):** Provides single and double seeds for reciprocal and reciprocal square root algorithms.

▲ **Floating-Point Comparison (C67x S-Unit):** This provides single- and double-precision comparison, generating a 0/1 result of a general-purpose register.

▲ **Floating-Point Conversion: (C67x L-Unit, S-Unit):** Conversion to and from single and to and from double precision as well as floating-point to and from unsigned and signed integers is provided. Conversions to integers can occur with rounding or truncation.

```
if (b2) a5 += b4;
if (a0) b0 = a0 * (unsigned) (b1 >> 16);
else
b0 = 0;
a5 &= 1;
}
```

▲ *4. Conditional execution example C code (Unit indicates the units used. These unit indicators are optional but are shown for clarity. "||" indicates that a an instruction is in parallel with the one before it. ";" indicates a termination of a line—all that follows is comment.*

```
[B2] ADD          .L1X   A5, B4, A5
  [A0] MPY[illegible]SH   .M2X   A0, B1, B0
|| [!A0] ZERO     .S1    B0
|| [!A0 AND       .D1    A5, 1, A5
```

▲ *5. Conditional execution example assembly code.*

```
cnt = _lmbd(bit, val);
if (cnt < left) {
val <<= cnt;
left -= cnt;
*code++ = run+cnt;
        run = 0;
} else {
        run += left;
       val = *input++;
        left = k;
}
```

▲ *6. C code for run-length coding example.*

Table 5. Execute pipeline operation of instructions.

Instructions	Latency	Delay Slots	Functional Unit Latency
▲ All C62x instructions except LD, MPY, and SMPY, and B. ▲ C67x S-Unit SP Instructions, ADDAD	1	0	1
▲ MPY, SMPY	2	1	1
▲ C67x S-Unit DP Instructions	2	1	2
▲ C67x L-Unit and M-Unit SP Instructions, MPY24(H)	4	3	1
▲ LD(B,H,W,D)	5	4	1
▲ C67x L-Unit SP Instructions	5	4	2
▲ B	6	5	1
▲ C67x MPYI	9	8	4
▲ C67x MPYID, MPYDP	10	9	4

Instruction Packing

The following terms are used in describing VelociTI's instruction packing:

▲ Fetch Packet: A group of instructions fetched simultaneously.

▲ Execute Packet: A group of instructions beginning execution in parallel.

The C62x CPU has a 256-bit path for internal program access to fetch eight 32-bit instructions every cycle. In typical VLIW architectures, each instruction would correspond to a particular functional unit. If that functional unit were idle on any particular cycle a NOP would be placed in that functional unit's instruction slot. In contrast, VelociTI decouples fetch packets from execute packets through a novel instruction encoding system. The least-significant bit of every C62x instruction is a called the parallel or p-bit. The p-bit of a particular instruction is set if the instruction starts execution in parallel with the next instruction. Figure 3 shows three examples of VelociTI instruction packing and the resulting code-size savings. In addition, reduced code size also results in fewer program fetch accesses and thus lower power consumption.

```
LMBD.L2X  A0, B1, B0  ; cnt = _lmbd(bit,val)

        CMPLT.L1X B0, A5, A1   ; A1 = (cnt < lef
||      ADD.L2    B4, B0, B5   ; B4 = run+cnt

   [A1]  SHL.S2   B1, B0, B1 ; val <<= cnt
|| [A1]  SUB.L1X  A5, B0, A5 ; left -= cnt
|| [A1]  STW.D1   B5, *A6++  ; *code++ =run+cnt
|| [A1]  MPY.M2   B4, 0,  B4 ; run = 0
|| [!A1] ADD.L2X  B4, A5, B4 ; run += left
|| [!A1] LDW.D2  *B2++, B1   ; val = *input++
|| [!A1] MVK.S1   32, A5     ; left = 32
```

▲ *7. Assembly code of run-length coding example.*

Conditional Instructions

Every C6x instruction can be conditioned on either the zero (false) or the nonzero value (true) of one of five general-purpose registers (A1, A2, B0, B1, B2). All instructions will enter the first phase of execution regardless of the evaluation of their condition. However, if the condition is not met by the end of the first phase, the instruction will not have its results written back to the register file. In addition, a conditional load or store instruction whose condition is not met is canceled before entering the data memory portion of the execution pipeline. This prevents any undesired accesses to memory, including memory-mapped peripherals where simple accessing has undesired side-effects. This feature is an improvement over the TMS320C3x/C4x conditional load instructions where memory was accessed but not written to the register file.

Conditional instructions can be used to avoid branch latency. In control code, conditional instructions allow increased parallelism as multiple paths can be executed simultaneously. In the example, both `if` paths as well as the `else` path are executed in a single fetch packet (Figs. 4 and 5). In the inner loop of a run-length-coding algorithm, all seven statements in the `if` and `else` clauses are performed in a single instruction (Figs. 6 and 7).

Pipeline

The pipeline phases are divided into three stages (Fig. 8):

▲ Fetch, which covers four pipeline phases.

▲ Decode, which covers two pipeline phases.

▲ Execute, which covers a maximum of 10 phases. Over 90% of C62x instructions use only the first five phases. The last five phases are only used for double-precision multiplies and adds on the C67x.

Every execute packet spends one cycle in each phase (Fig. 9).There are no hardware interlocks within the CPU to stall the pipeline. Instead, during execution, each instruction has a fixed predictable delay for arrival of its results. This is discussed later in the article.

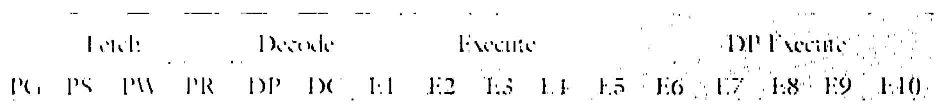
Fetch

The fetch phases of the pipeline (Fig. 10) are:

▲ **PG:** Program address generate—the next sequential fetch packet address or a branch address computed.

▲ **PS:** Program address send—program address sent to memory.

▲ **PW:** Program access ready wait. Either a memory access or tag compare is completed. In cases of cache miss or external access the CPU is stalled by the memory system in this phase. All other phases stop in lock step. A memory



▲ *8. Pipeline stages and phases.*

stall is the only case where the C6x pipeline stalls. Thus, the predictability of execution delays is maintained.

▲ **PR:** Program fetch packet receive—fetch packet sent from memories to CPU.

Decode

The decode phases of the pipeline are:

▲ **DP:** Instruction dispatch. Fetch packets are separated into execute packets. The instructions of execute packets are routed to the decode of the appropriate functional unit.

▲ **DC:** Instruction decode. Consider a fetch packet containing three execute packets: A||B||C, D||E, G||H (Fig. 11). The dashed arrows indicate instructions in the first execute packet (A||B||C) that were sent from dispatch to decode on the previous cycle. In the current cycle, the solid arrows indicate instructions in the second execute packet (E||D) being sent to decode. In the next cycle (not shown), G and H will be sent to the appropriate functional units for decode. In cases such as this where a single fetch packet contains multiple execute packets, the C6x has fetched ahead. In this case, program fetch halts until new instructions are needed (Fig. 12).

Execute

The execute portion of the pipeline is subdivided into 10 phases (E1–E10). Each instruction requires a fixed number of phases to complete execution. Table 5 details the execute operation of all C62x and C67x instructions. The following terms are used.

▲ **Result Latency:** The number of execute stages used by an instruction. For example, the MPY instruction uses E1 and E2.

▲ **Delay Slots:** Based on the latency of instructions the number of subsequent execute packets after which the results of an instruction are ready. This is the latency minus 1. For example, the results of an ADD are available for use in the next execute packet. In contrast, the results of a MPY are available for the second execute packet after the one containing the MPY. Over 90% of C62x instructions have 1 cycle result latency (no delay slots) and are available for use the next cycle. In the delay slots of an instruction, execute packets that are not dependent on that register result or using the previous value in that register may be scheduled. In cases where sufficient parallelism does not exist a multicycle NOP is provided to reduce code size.

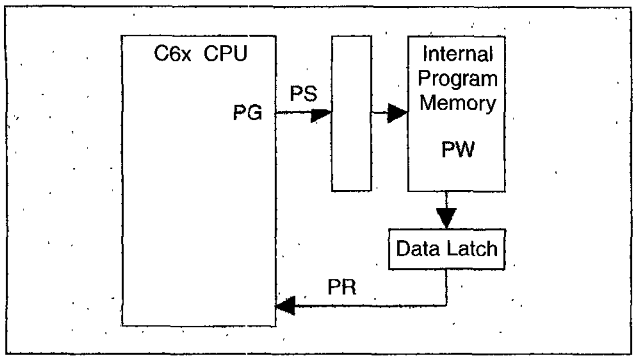
▲ **Functional Unit Latency:** The number of cycles after an instruction begins execution that another instruction can begin execution on the same functional unit. All C62x instructions and all C67x single-precision instructions have a functional unit latency of 1, and thus can be executed every cycle. Only double-precision instructions on the C67x have a functional unit latency more than 1.

Data Access Executions. Loads and stores follow the same pipeline flow (Fig. 13). Loads following a store access the memory after the store has completed. Results stored in one execute packet can be read in the next execute packet. Thus, this memory pipeline avoids the read-after-write conflicts typically found in DSPs. Two 32-bit load or store access can be executed by the C6x every cycle. In addition, the C67x loads can be 64-bits using the LDDW instruction.

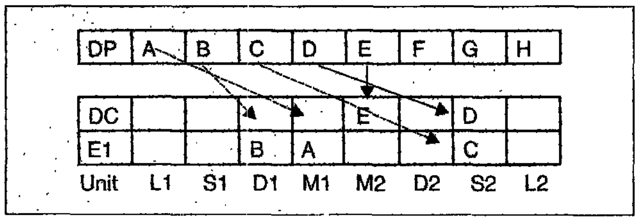
Branch Execution. Figure 14 shows a branch from execute packet N to execute packet M. Notice that there are five subsequent execute packets (five delay slots) before the branch occurs. The branch executes in the E1 phase of the pipeline and affects the execute packet currently in the PG phase. Branches can be executed every cycle, permitting single cycle loops despite the branch latency. This feature allows nesting of loops with no overhead. Two branches can be executed in parallel allowing multiple si-

Execute Packet	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
N	PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
N+1		PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7	E8	E9
N+2			PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7	E8
N+3				PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7
N+4					PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6
N+5						PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5
N+6							PG	PS	PW	PR	DP	DC	E1	E2	E3	E4
N+7								PG	PS	PW	PR	DP	DC	E1	E2	E3
N+8									PG	PS	PW	PR	DP	DC	E1	E2
N+9										PG	PS	PW	PR	DP	DC	E1
N+10											PG	PS	PW	PR	DP	DC

▲ *9. Complete pipeline flow.*



▲ *10. Fetch pipeline flow.*



▲ *11. Decode pipeline flow.*

multaneous program flows. Coupled with appropriate conditioning of instructions in the delay slots of these branches, two branch paths as well as the nonbranching program flow can execute in parallel.

Benchmark Performance

Table 6 shows TMS320C62x benchmark performance. On average, the architecture performs an application in 10-20% the execution time (5 to 10 times the performance) of contemporary DSPs. In multiply-accumulate algorithms, the dual multipliers would only give 50% the number of cycles or twice the performance. The processor operates at 2-4 times the cycle time of its 50-100 MHz contemporary DSPs, delivering 4-8 times the comparative performance. The additional improvement comes from its orthogonal architecture and additional ALUs. Unlike other DSPs these ALUs are not special-purpose functional units restricted to address calculation and looping. In general, these units add another 1.25x to the overall performance. Thus, the overall performance improvement on the C6x is 5x to 10x. Other evaluation of C62x performance is available in [6].

The TMS320C6201 Device

The TMS32C6201 consists of the C62x CPU. The memory architecture includes 64 Kbytes each of on-chip program memory and on-chip data memory. The program memory is configurable for use as mapped memory or as direct mapped cache. A direct memory access (DMA) controller provides background memory transfers to and from all peripherals and memory. An external memory interface (EMIF) connects the C6201 to external memory and peripherals. Two multichannel buffered serial ports (McBSPs) interface the C6201 to many industry-standard serial peripherals. The host port interface (HPI) allows external processors to make direct data requests of the C6201's memory space. Power-down logic allows reduced power modes while not in operation.

Memory Architecture

The C6201 memory architecture consists of four components: the EMIF, the DMA controller; 64K bytes of on-chip program memory configurable as mapped memory or as a direct mapped cache; and 64 Kbytes of interleaved data memory.

The C6x supports both little and big endian for all its memory spaces. Endianness is set by a device pin and sampled once at device reset.

External Memory Interface

The 32-bit EMIF provides:

▲ Glueless interconnect to a broad spectrum of memory devices.

▲ 800 Mbytes/second of throughput using synchronous burst SRAM (SBSRAM).

▲ 52 total Mbytes of external address reach in four separate chip-enable (CE) spaces. These CE spaces consist of three 16 Mbyte spaces and one 4 Mbyte space.

▲ Byte addressibility though four byte enables.

▲ Ability to unpack data from 8-bit and 16-bit-wide ROM devices.

▲ An external bus hold input for shared memory interfaces.

Supported Memories. The EMIF provides glueless connection to a variety of external devices including:

▲ Pipelined early write synchronous burst SRAM (SBSRAM) running at 1x or 1/2x the CPU clock rate.

▲ Synchronous DRAM (SDRAM) running at 1/2 the CPU clock rate. The EMIF provides necessary page management and refresh control.

Fetch Packet	Execute Packet	1	2	3	4	5	6	7	8	9	10
N	K	PG	PS	PW	PR	DP	DC	E1	E2	E3	E4
N	K+1						DP	DC	E1	E2	E3
N	K+2							DP	DC	E1	E2
N	K+3								DP	DC	E1
N	K+4									DP	DC
N+1	K+4		PG	PS	PW	PR	PR	PR	PR	PR	DP
N+2	K+6			PG	PS	PW	PW	PW	PW	PW	PR
N+3	K+7				PG	PS	PS	PS	PS	PS	PW
N+4	K+8					PG	PG	PG	PG	PG	PS

▲ *12. Fetch/decode pipeline operation with serial operation [5]*

▲ An asynchronous memory interface with programmable memory timing. This feature allows programmable setup, strobe, and hold timings to allow interface to a variety of asynchronous devices. These devices include standard asynchronous SRAM, ROM, flash ROM, and parallel interface-controlled external peripherals. The asynchronous interface also has an external RDY input for variable-rate external devices.

DMA Controller

The on-chip DMA controller performs background data transfer without CPU intervention to and from on-chip peripherals, internal data memory, internal program memory when not used as cache, and external memory

The DMA can perform 32-bit burst transfers at a 200 MHz rate for 800 Mbytes/second. The DMA controller consists of four programmable channels. A fifth auxiliary (AUX) channel services requests from the HPI, which has its own address-generation capability. All five channels have independently selectable priority versus the CPU. At system initialization, the DMA can be selected to boot the internal memory from an external ROM device.

Each programmable channel has a programmable source address, destination address, and transfer count. Address indexing supports fixed address, linear address striding, two-dimensional array movement, and data interleaving. Both individual and block transfers may have their read and/or write operations transfers triggered by events from internal or external peripherals. Each programmable channel can generate independent interrupt conditions to the CPU upon transfer completion or error conditions. These channels also have an auto-initialization mode that allows for continuous block data transfers without restart by the CPU. Finally, each channel may be configured in split-mode operation, which enables it to service both the transmit and receive data flow from a peripheral.

Internal Program Memory

The 64 Kbytes of on-chip program memory is configurable as either mapped memory or as direct mapped cache. In either case, the internal program contains 16K 32-bit instructions or 2K 256-bit fetch packets. The block size of the cache is a fetch packet or eight instructions.

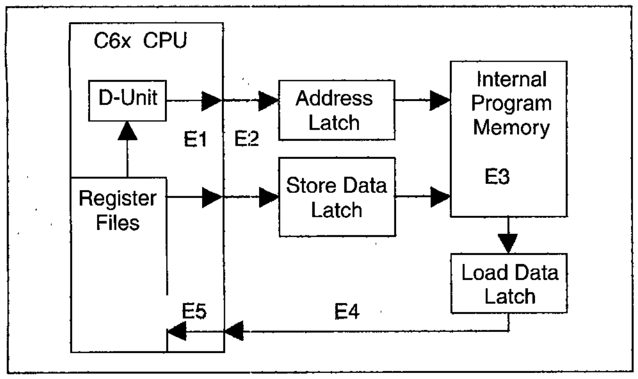
Internal Data Memory

The 64 Kbyte C6201 internal data memory is organized into eight 8 Kbyte 16-bit-wide banks of memory (Fig.

Table 6. C62x benchmark performance.*

Benchmark	Description	Perform. Equation (cycles)	Parameter Values	Cycles	Time
FIR	M Outputs N Coef.	0.5*MN	N=3 2 M=100	1613	8 us
Complex FIR	M Complex Outputs N Complex Coef and Inputs	2MN	M = 100 N = 32	6410	32 us
LMS FIR	M Coef. N Samples Non-Delayed Updates on Every Coefficient	1.125MN	M = 100 N = 32	5105	25.5 us
Lattice Analysis	N Coef.	1.5N	N=10	25	125 ns
Lattice Synthesis	N Coef.	2N	N = 10	38	190 ns
IIR	N Biquads	4N	N = 10	56	28 us
Autocorrelation	N Coef. M Outputs	0.5NM	M = 10 M = 160	81.6	4.08 us
Dot Product	Length N	0.5N	N = 100	58	290 ns
Block move	Length N	0.5N	N = 100	55	275 ns
Sum of squares	Length N	0.5N	N = 100	59	293 ns
Cx Rdx4 FFT	N Points	$1.25N\log_2(N)$	N = 1024	13228	66 us
Vector Max	Length N	0.5N	N = 100	64	320 ns
Codebook Search				30.68	15.4 us
8x8 IDCT				230	1.15 us
8x8 DCT				226	1.13 us
Gouraud Shadings	N Pixels	2N	N = 1024	2055	10.275 us
Viterbi IS54 Channel Decoder	N Points	66N	N = 89	5874	29.67 us
Viterbi V.32 PSTN Trellis Decoder				64	320 ns

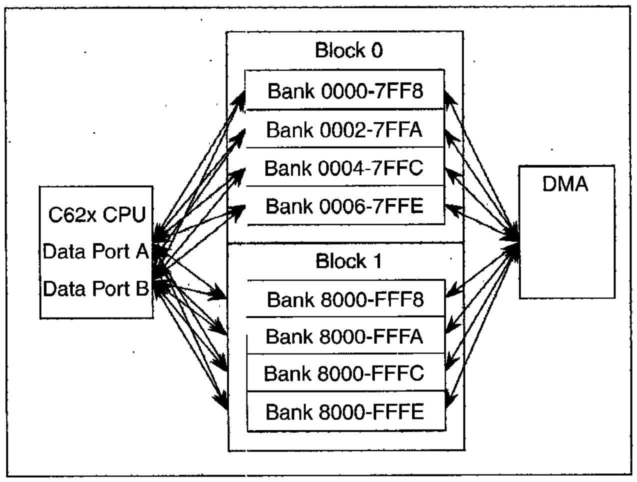
*Note that the performance equation shows the leading term only. The detailed equations may be found at http://www.ti.com/sc/docs/dsps/products/c6x/benchmk.htm



▲ *13. Execute pipeline flow for load and store.*

Execute Packet	1	2	3	4	5	6	7	8	9
N	PG	PS	PW	PR	DP	DC	E1	E2	E3
N+1		PG	PS	PW	PR	DP	DC	E1	E2
N+2			PG	PS	PW	PR	DP	DC	E1
N+3				PG	PS	PW	PR	DP	DC
N+4					PG	PS	PW	PR	DP
N+5						PG	PS	PW	PR
M							PG	PS	PW
M+1								PG	PS

▲ *14. Branch pipeline execution.*



▲ *15. Internal data memory organization.*

15). These banks are grouped into two blocks consisting of four banks each. The address of each bank in a block is interleaved. The banks are arbitrated on a 16-bit cycle-by-cycle basis. Pairs of banks are used if 32-bit CPU or DMA accesses are requested. Like the EMIF the internal data memory has byte enables to support byte and half-word accesses.

Both CPU data ports and the DMA have access to each bank. Thus, the three requesters (CPU Data Port A, CPU Data Port B, and the DMA) compete for eight banks, reducing the possibility of conflict. In addition, the two blocks are in distinct spaces allowing a ping-pong buffer approach—avoiding any DMA/CPU contention. When the DMA and CPU Data Ports do contend for the same bank, the priority versus of the requesting DMA channel determines the winner. If the DMA channel is higher priority, the CPU is held as long as it is contending with a DMA channel access. To maintain a fixed programming model, the entire CPU pipeline is frozen in this case. If the CPU has higher priority, the DMA is held until the CPU no longer is accessing the requested bank. When no conflict occurs, the DMA and CPU can cycle-steal accesses to the unused banks with neither being stalled.

Peripherals

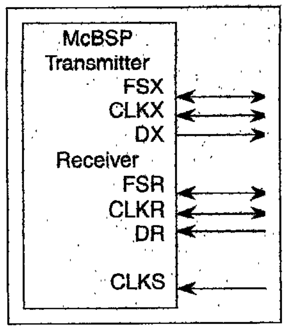
The peripherals on the C6201 include two multichannel buffered serial ports, two 32-bit timers, a 16-bit-wide HPI, and power-down logic.

Multichannel Buffered Serial Port

The McBSP is a full duplex serial port capable of running at up to 100 MHz (two 100 Mbits/second streams). The McBSP has independent frame synchronization, bit clocking, and data lines (Fig. 16). In addition, an external CLKS input allows the transmitter and/or receiver to run from a divide down of an externally provided clock. Alternatively, an internal clock running at 1/2 the CPU clock rate can drive this clock divider. For both the transmitter and receiver, each clock and frame may be configured independently to be driven from an external source (slave) or from the McBSP's sample rate generator (master). This sample-rate generator can program both the width and active period of internally generated frame synchronization.

Each serial frame consists of one or two phases of up to 128 serial words each. The McBSP supports programmable word sizes of 8-, 12-, 16-, 20-, 24-, and 32-bits. Wordsize and words per phase are independently programmable for each phase of the transmitter and receiver. Multichannel selection mode allows the McBSP to pick specific serial word slots to receive or transmit and ignore the others. This feature reduces data storage transfer to only those words the C6x processes from the serial stream. Data transfer for both the receiver and transmitter may be serviced in the background by a single DMA channel operating in split mode. The McBSP's programmability allows interoperability with a variety of standards:

▲ ST-BUS (MVIP switching compatible), which allows direct interface to T1, E1, H.100, MVIP, and SCSA framing chips.



▲ *16. McBSP.*

▲ AC97 compliant codecs.
▲ I^2S compliant devices.
▲ SPI™

Timers

Each 32-bit timer has a dedicated input and output pin. An external signal or an internal clock running at 1/4 the CPU clock rate may clock the timer. The timer generates a pulse or square wave output with a frequency determined by the period programmed into the timer. The timer can generate interrupts to the CPU on counting the required period. The timers can be used for time-slice interrupt generation for a real-time operating system (RTOS), event counting, and pulse and clock generation.

Host Port Interface

The HPI is a 16-bit wide bi-directional port that interfaces with little or no logic to a variety of industry-standard microprocessors, microcontrollers, and embedded RISC processors. This interface can operate at up to 50 MHz for 100 Mbytes/second of data throughput.

Power-Down Modes

The user may enable any one of three power-down modes to reduce power consumption while the CPU is idle:

▲ Power Down 1: The CPU clocks are halted. All other peripherals continue running as programmed. In this mode, the CPU wakes up through an external or internal interrupt.
▲ Power Down 2: All internal clocks are halted, and the C6201 wakes up through device reset. The on-chip PLL continues running.
▲ Power Down 3: Same as power down 2, but the PLL is halted.

Table 7. Compiler Version 1.00 benchmark performance.

Benchmark	Hand Asm. Cyc.	Comp Cyc.	Efficiency
CELP Impulse	1367	1897	72%
CRC	45	90	50%
DCT, Inverse MPEG	247	282	88%
DCT, JPEG	245	240	102%
FIR	715	1118	64%
FIR, LMS	69	77	90%
Gouraud Shading	2065	2571	80%
Harmonic Series Oscillator	106	140	76%
IIR	58	69	84%
Lattice Analysis Filter	27	42	64%
Lattice Synthesis	40	52	77%
Max Index	33	42	79%
Max Value	30	58	52%
Mean Square Error	273	508	54%
Mean Squared Error Vector Quantizer	267	269	99%
MPEG2 Absolute Distance	376	557	68%
Viterbi GSM Equalizer	5182	8535	61%
Viterbi V32 Decoder	79	113	70%
VSELP Autocorrelation Matrix	994	991	100
VSELP Block Move	27	30	90%
VSELP Dot Product	29	32	91%
VSELP FIR	226	260	87%
VSELP G Computation	22	32	69%
VSELP GL Computation	21	26	81%
VSELP HPF	823	989	83%
VSELP Lac MAC	32	54	59%
VSELP MAC	49	53	92%
VSELP Min Err	1173	1576	74%
VSELP Orthogonalization	50	58	86%
VSELP Vector Multiply	52	58	90%
Geometric Mean			81%
Arithmetic Mean			82%
Total Cycles	15716	21810	72%

System Examples

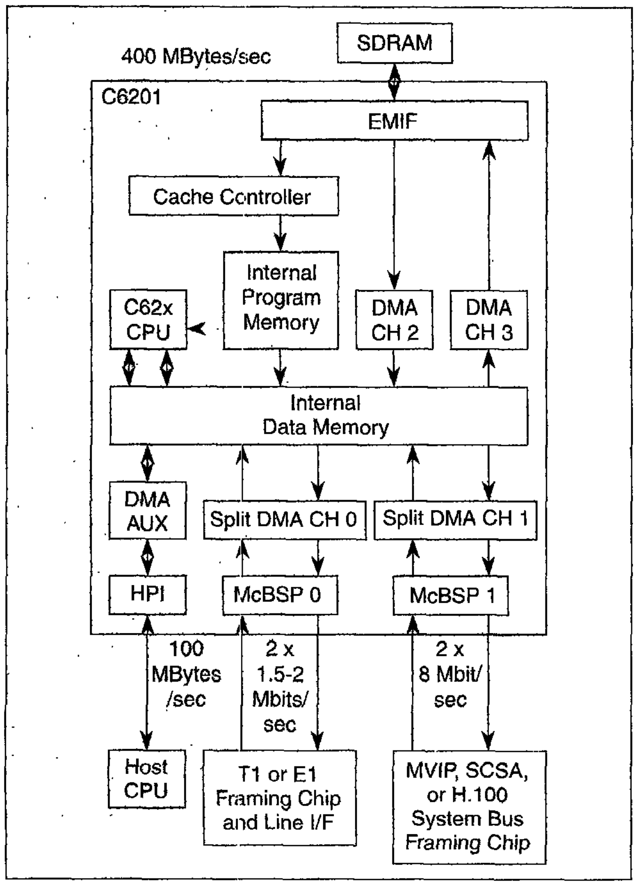
This section describe the use of the C6201 in three systems:

▲ A multichannel telecommunications or data communications system.
▲ A 2-4 channel high sample rate system.
▲ A multichannel high-fidelity audio system.

Multichannel Datacom or Telecom

The system shown in Fig. 17 is a multichannel datacom system similar to those used for pooled modem for remote access servers; multichannel vocoding for wireless, cellular, and personal communication system (PCS) basestations as well as for digital telephony; and multichannel line-echo cancellation.

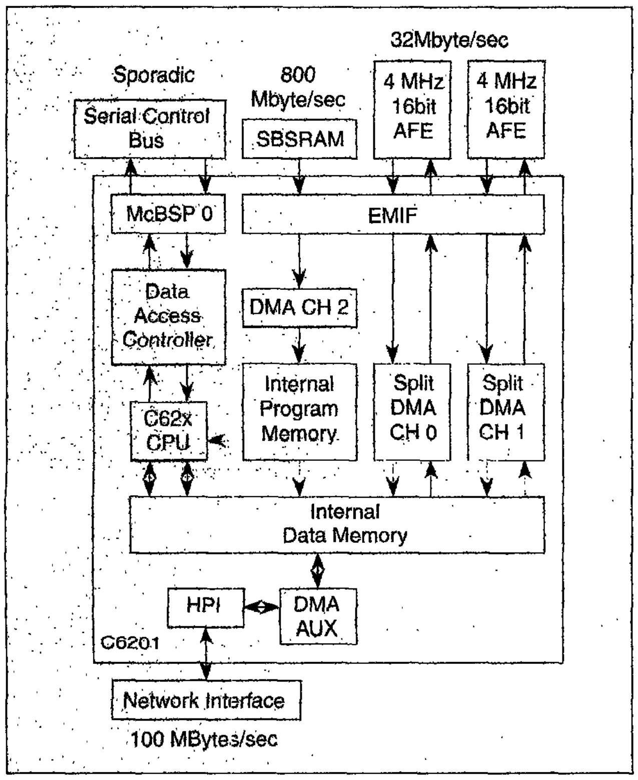
SDRAM provides high-density, low-cost-per-bit external storage. The EMIF services program fetch requests from the cache controller. External program memory is necessary to support program store for a variety of vocoding standards in multichannel vocoding as well as for all fallback modes in a x2™ or V.34 modem system. The external program is unnecessary in line-echo cancellation systems. DMA channels 2 and 3 move the context for the various channels on- and off-chip as needed. Through the HPI, a host CPU provides system control. McBSP 0 interfaces to a T1 or E1 line for connection to the communications infrastructure. McBSP 1 interfaces to a internal system bus that routes data between similar processing blocks. DMA channels 0 and 1 configured in split mode serve the full-duplex streams of the two McBSPs.



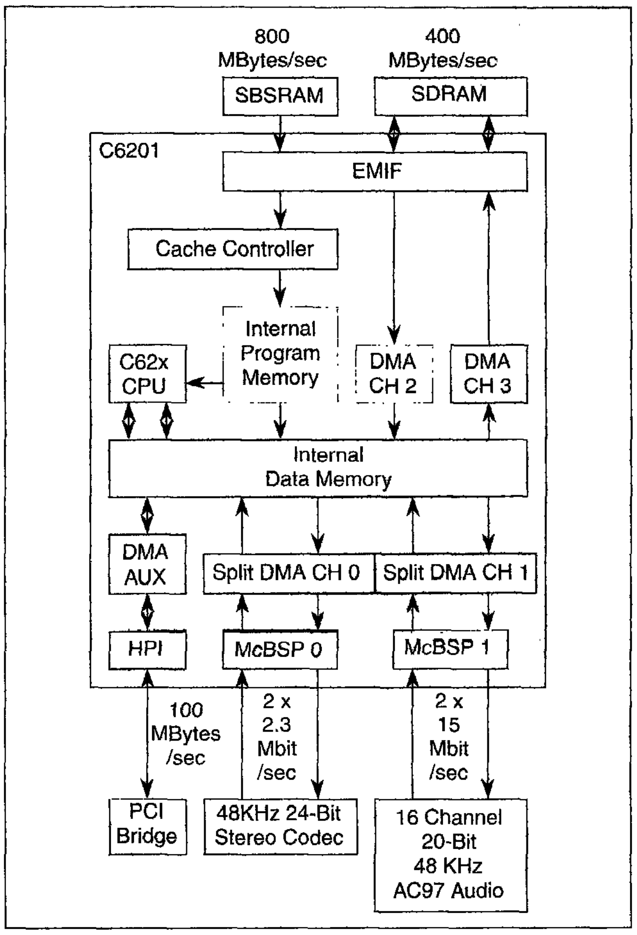
▲ *17. Multichannel datacom or telecom system.*

High Sample-Rate Processing

The system in Fig. 18 represents both ADSL modems and antenna array processing as found in digital wireless basestations. A single McBSP services sporadic communications requests from a system serial control bus. The rate of these requests is low enough for the CPU to service them directly with no performance loss. In these systems, the amount of required program and data memory is lower and can fit into high-performance SBSRAMs. In contrast to the previous example, the DMA moves program memory in the background to completely avoid any cache miss penalty. Two parallel analog front-end (AFE) chips are serviced through the asynchronous interface of the EMIF by DMA 0 and 1 configured in split mode. In an ADSL system, the HPI may serve as the interface to

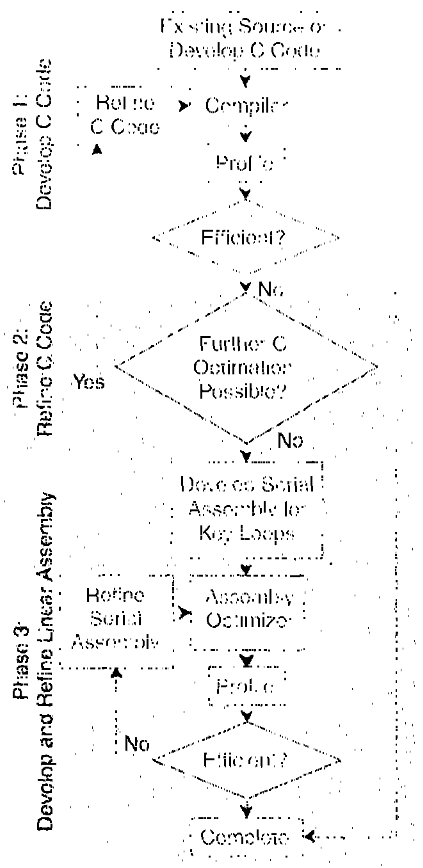


▲ *18. High sample-rate processing system.*



▲ *19. Multichannel high-fidelity audio system.*

the backbone network in the central office or to a local network in a small office or home-office environment.



▲ 20. *VelociTI programming methodology.*

```
int sadd(int a, int b)
{
  int result;
  result = a + b;
  if (((a ^ b) & 0x80000000) == 0)
    if ((result ^ a) & 0x80000000)
      result = (a < 0) ? 0x80000000 : 0x7fffffff;
return (result);
}
```

▲ 21. *Saturated add without intrinsics.*

```
          result = _sadd (a,b) ;
```

▲ 22. *Saturated add with intrinsics.*

```
LDF  0, R0                    ; zero initial product

    LDF  0, R2                ; zero initial sum

    RPTS N-1
LOOP:
    MPYF *AR0++, *AR1++, R0   ; prod[i+1] = a[i+1] * b[i+1]
||  ADDF R0, R2, R2           ; sum += prod[i]
```

▲ 23. *TMS320C3x dot product assembly.*

Multichannel High-Fidelity Audio

A multichannel high-fidelity audio system (Fig. 19) might be used in theater, audiophile home theater, as well as studio-quality digital synthesizer, mixing, and recording equipment. External SBSRAM serves as a high-rate memory system to store external programs. SDRAM provides high-volume storage for audio samples. DMA channels 2 and 3 retrieve and store samples in external memory as needed. The HPI interfaces to a PCI bus through a PCI interface chip allowing the system to reside on a personal computer. McBSP 0 interfaces to a local stereo codec, whereas McBSP 1 interfaces to an AC97 system audio stream. DMA channels 0 and 1 in split mode serve the McBSPs' data streams.

Development Environment

Like other TMS320 DSPs the VelociTI development environment includes:

▲ Code-generation tools including a C compiler, an assembler, a linker, and a ROM hex output utility.
▲ Debug tools including a standalone loader and a source-level debugger. Both the loader and debugger interfaces are available in a software simulator and in a hardware JTAG scan-based emulation interface. The loader and debugger also provide support for all C `stdio` library commands such as `printf` and `scanf`.
▲ Board-level test via JTAG boundary scan.
▲ Hardware development boards such as a PC-based evaluation module (EVM).

The remainder of this section concentrates on the programming environment for the C6x. This includes:

▲ An overview of the programming methodology.
▲ A description of optimization techniques available with the C6x compiler and assembly optimizer focusing on unique capabilities versus other DSP development environments.
▲ A brief demonstration of the capabilities of the tools to extract parallelism with low effort.
▲ A summary of available benchmark performance of the C compiler.

Programming Environment

Two new capabilities are available in the VelociTI programming environment. These capabilities allow improved optimization technology that allows C code to achieve on average 70-80% of theoretical performance, and an assembly language optimizer that automatically generates parallel assembly with all resources and registers allocated from linear symbolic assembly source.

The phases in the three-step software development flow shown in Fig. 20 are listed below. The first two steps are common to most processors. However, there are techniques in the second step not used extensively in previous DSP compilers that are described later in this section. The third

```
short *a, *b;
int c;
for(i=0; i<100; i++) c += a[i] * b[i];
```

▲ *24. Dot product C.*

```
· LDH .D1  *AA++, AI         ; load a[i]
· LDH .D2  *AB++, BI         ; load b[i]
· MPY .M1X AI, BI, AP        ; a[i] * b[i]
· ADD .L1  AP, AC, AC        ; c += a[i] * b[i]
· [BLC] SUB.S2 BLC, 1, BLC ; decrement loop counter
· [BLC] B  .S1   LOOP      ; branch to loop
```

▲ *25. List of instructions used in dot product with shown with symbolic registers.*

```
· LDW   .D1 *AA++, AI       ; load a[i] & a[i+1]
· LDW   .D2 *BA++, BI       ; load b[i] & b[i+1]
· MPY   .M1X AI, BI, AP     ; a[i] * b[i]
· MPYH  .M2X AI, BI, BP     ; a[i+1] * b[i+1]
· ADD   .L1  AP, BP, ACA    ; ca += a[i]   * b[i]
· ADD   .L2  BP, BP, BCB    ; cb += a[i+1] * b[i+1
· [B0] SUB.S2  B0, 1, B0 ; decrement loop counter
· [B0] B   .S1  LOOP      ; branch to loop
```

▲ *26. Refined list of instructions used in dot product.*

```
      B    .S1 LOOP          ; branch to loop

      B    .S1 LOOP          ; branch to loop

      B    .S1 LOOP          ; branch to loop

      B    .S1 LOOP          ; branch to loop
||    ZERO          .L1 A2   ; zero A side product
||    ZERO          .L2 B2   ; zero B side product

      B    .S1 LOOP          ; branch to loop
||    ZERO.L1 A3             ; zero A side accumulator
||    ZERO.L2 B3             ; zero B side accumulator
||    ZERO.D1 A1             ; zero A side load value
||    ZERO.D2 B1             ; zero B side load value

LOOP:LDW .D1 *A4++,    A1   ; load a[i] & a[i+1]
||    LDW .D2 *B4++,    B1   ; load b[i] & b[i+1]
||    MPY .M1X A1, B1, A2   ; a[i] * b[i]
||    MPYH.M2X A1, B1, B2   ; a[i+1] * b[i+1]
||    ADD .L1  A2, A3, A3   ; ca += a[i] * b[i]
||    ADD .L2  B2, B3, B3   ; cb += a[i+1] * b[i+1]
||[B0]SUB.S2  B0, 1, B0     ; decrement loop counter
||[B0]B   .S1  LOOP          ; branch to loop

      ADD .L1X A3, B3, A3   ; c = ca + cb
```

▲ *27. Scheduled software pipelined loop from symbolic assembly.*

step is enabled by the first production DSP assembly language optimizer, which schedules, parallelizes, and allocates resources and registers from a serial input source. A tutorial on using the C6x tools to optimize performance is available in [7].

1. Develop C Code: Once developed and compiled the code is then debugged through the source debugger. In addition to functional debug the tools provide performance debug from automated cycle counts from profiling, interactive debug cycle counts, and a library of functions used for run-time performance analysis by the program.

2. Refine C Code: C code can be refined by using procedures such as compiler options, intrinsics, statements, data type modifiers, and code transformations.

3. Develop and Refine Linear Assembly: Extract the inefficient areas from the C code and rewrite them in assembly optimizer source code.

Code-Scheduling Optimization

Optimizations specific to the C6x compiler include:

▲ Intrinsic functions that allow access to special-purpose DSP instructions from C.

▲ Software pipelining allows a more general-purpose instruction set and data path than is typically found on DSPs to still provide high performance on DSP algorithms. A brief example is provided later. A more complete tutorial is available in [8].

▲ "If" conversion/predicated execution. The compiler can turn if/else and case statements into concurrent linear code through use of conditional instructions.

▲ Memory address cloning. This allows vectorization and unrolling of memory accesses across the D and load-store units on the two data paths.

▲ Memory address-dependence elimination. The compiler eliminates the possibility of pointer aliasing and schedules code more optimally. The user can indicate that a pointer or array variable overlaps with no other pointers through use of the `const` type modifier in declaration or casting.

▲ Memory-bank disambiguation: In many cases, the compiler and assembly optimizer can detect and avoid memory-bank conflicts between the two data port accesses at compile time.

Optimizations available on all TMS320 compilers include: branch optimizations/control-flow simplification; alias disambiguation, copy propagation; common subexpression elimination; redundant assignment elimination; loop-induction variable optimizations/strength reduction; loop rotation; loop-invariant code motion; inline expansion of function calls; file-level optimizations; data-flow optimizations; expression simplification; register variables; register tracking/targeting; and cost-based register allocation. All optimization techniques are used by both the compiler as well as the assembly language optimizer.

Intrinsic Functions

Intrinsic functions map C directly to inlined C6x instructions. All instructions that are not easily expressed in C code are supported as intrinsics by the C6x compiler. The following C62x instructions are available as intrinsics: ADD2, SUB2, CLR, SET, EXT(U), LMBD, SMPY, MPY, NORM, SADD, SSUB, SAT, SSHL, SUBC. Figures 21 and 22 compare the C code for a 32-bit saturated add without and with intrinsics, respectively. For the

```
void iir(const short *coefs, const short *input,
    short *optr, short *state, int n)
{
  short x, t;
  int i;

  x = input[0];
  _nassert(n > 5);
  for (i = 0; i < n; i++){
      t = x + ((coefs[2] * state[0] +
               coefs[3] * state[1]) >> 15);
      x = t + ((coefs[0] * state[0] +
               coefs[1] * state[1]) >> 15);
      state[1] = state[0];
      state[0] = t;
      coefs += 4; /* point to next filter coefs */
      state += 2; /* point to next filter states */
  [illegible]
  *optr++ = x;
[illegible]
```

▲ 28. IIR C source with minimal refinements.

```
[illegible]
```

▲ 29. Compiler 5-cycle loop from Fig. 28.

```
void iir(const int *coefs, const short *input,
                short *optr, short *state, int n){
    short                  x, t;
    int                          i;

    x = input[0];
    _nassert(n > 5);
    for (i = 0; i < n; i++){
        t = x+((_mpy(coefs[1], state[0]) +
            mpyhl(coefs[1],state[1])) >> 15);
        x = t+((_mpy(coefs[0], state[0]) +
            mpyhl(coefs[0],state[1])) >> 15);
        state[1] = state[0];
        state[0] = t;
        coefs += 2;
        state += 2;
        }
     *optr++ = x;
}
```

▲ 30. IIR C source with intrinsics.

compiler as well as the programmer, intrinsics make DSP instructions easily accessible from C.

Software Pipelining

Rather than encode special-purpose multipipeline phase instructions, the VelociTI architecture employs high-throughput RISC-like instructions that can be combined to execute various software pipelined loops including the benchmarks shown in Table 6. This section provides an overview for software pipelining as it has not been widely deployed on DSPs. Software pipelining has its origins in vector processors built in the 1960s and to a limited extent on DSPs such as the Texas Instruments TMS320C3x where a floating-point dot product was expressed as shown in Fig. 23. Note that in the inner loop the next product is being multiplied while the previous product is accumulated.

Figure 25 shows the list of C6x instructions needed to perform a dot product shown in Fig. 24. These are listed in serial format with symbolic registers. Resource allocation is done for explanatory purposes. When using the assembly optimizer neither the side of the registers or the functional units must be specified. The assembly optimizer will perform all necessary resource and register allocation when not specified. Figure 26 shows an extension to that list with some optimizing refinements. For example, a LDW instruction is used to load a pair of 16-bit operands. Second, two iterations of the loop, i and i+1, are listed to fully use the dual multipliers. This technique is referred to as loop unrolling. Figure 27 shows a dot product after software pipelining. Note that the five branch instructions precede the loop to fill the pipeline with branches for the single cycle loop. Since, the load latency is 5, the first intended product is generated on the sixth iteration of the loop. Since the multiply latency is 2, the first intended multiply is performed on the eighth iteration of the loop. The load and product registers are zeroed to avoid spurious adds and multiplies while the first loads are completing in the first iteration of the loop. To complete both sums a final add is necessary after the loop. Both the compiler and assembly optimizer use software pipelining in achieving their performance.

Demonstration of Compiler and Assembly Optimizer

This section briefly illustrates the optimizations possible with the C6x code-generation tools. This example is covered more thoroughly in [7]. Figure 28 shows C code for an IIR filter with minimal refinements. The pointers are declared to `const` to indicate no overlap of arrays.

The `_nassert()` statement sets the minimum amount of times the loop will iterate. This results in an inner loop of five cycles (Fig. 29). Figure 30 shows code re-written to access the coefficients as 16-bit pairs and performs multiplies using intrinsics. This results in a four-cycle inner loop (Fig. 31). Figure 32 shows the assembly optimizer source for the IIR. This results in a three-cycle inner loop (Fig. 33).

```
         ADD    .L2  B7,B8,B7    ;
||       ADD    .L1  A0,A3,A0    ;
||       MV     .S2  B6,B9       ;@
||       STH    .D1  A5,*A4(6)   ;@
||       LDW    .D2  *B5++(8),B8 ;@@

         SHR    .S2  B7.15,B7    ;
||       EXT    .S1  A0.16,16,A0 ;
|| [B0]  SUB    .L2  B0.1,B0     ;@
||       MPY    .M2X B8.A5,B8    ;@
||       ADD    .L1X B6,A3,A3    ;@
||       LDH    .D2  *+B4(14),B6 ;@@@

         ADD    .L1X A0,B7,A6    ;
||       MPYHL  .M2  B8,B9,B7    ;@
||       SHR    .S1  A3,15,A3    ;@
|| [B0]  B      .S2  L3          ;@
||       LDW    .D2  *+B5(4),B7  ;@@@
||       LDH    .D1  *+A4(12),A5 ;@@@

         ADD    .L2  4,B4,B4     ;
||       STH    .D1  A0,*A4++(4) ;
||       EXT    .S1  A6,16,16,A0 ;
||       MPYHL  .M2  B7,B6,B6    ;@@
||       MPY    .M1X B7,A5,A3    ;@@
```

▲ 31. Compiler 4-cycle loop from Fig. 30.

```
iir    .cproc cptr0,sptr0
  .reg cptr1, s01, s10, s23, c10, c32, s10_s, s10_t
  .reg p0, p1, p2, p3, s23_s, s1, t, x, mask, sptr1, s10p, ctr
  MV        cptr0,cptr1
  MV        sptr0,sptr1
  MVK      .50,ctr               ; setup loop counter
LOOP:      .trip 50
  LDW.D1T1 *cptr0,c32            ; CoefAddr[3] & CoefAddr[2]
  LDW.D2T2 *cptr1,c10            ; CoefAddr[1] & CoefAddr[0]
  LDW.D1T2 *sptr0,s10            ; StateAddr[1] & StateAddr[0]
  MV        s10,s10p             ; save StateAddr[1] & StateAddr[0]
  MPY.M1   c32,s10,p2            ; CoefAddr[2] * StateAddr[0]
  MPYH     c32,s10,p3            ; CoefAddr[3] * StateAddr[1]
  ADD      p2,p3,s23             ; CA[2] * SA[0] + CA[3] * SA[1]
  SHR      s23,15,s23_s          ; (CA[2] * SA[0] + CA[3] * SA[1]) >> 15
  ADD.2    s23_s,x,t             ; t = x+((CA[2]*SA[0]+CA[3]*SA[1])>>15)
  AND      t,mask,t              ; clear upper 16 bits
  MPY      c10,s10,p0            ; CoefAddr[0] * StateAddr[0]
  MPYH     c10,s10,p1            ; CoefAddr[1] * StateAddr[1]
  ADD      p0,p1,s10_t           ; CA[0] * SA[0] + CA[1] * SA[1]
  SHR      s10_t,15,s10_s        ; (CA[0] * SA[0] + CA[1] * SA[1]) >> 15
  ADD      s10_s,t,x             ; x = t+((CA[0]*SA[0]+CA[1]*SA[1])>>15)
  SHL      s10p,16,s1            ; StateAddr[1] = StateAddr[0]
  OR       t,s1,s01              ; StateAddr[0] = t
  STW.D1   s01,*sptr1            ; store StateAddr[1] & StateAddr[0]
 [ctr] ADD -1,ctr,ctr            ; dec outer lp cntr
 [ctr] B    LOOP                 ; Branch outer loop
         .endproc
```

▲ 32. IIR assembly optimizer source.

```
    AND   .L2  B3,B7,B0       ; clear upper 16 bits
||  ADD   .S2  B0,B8,B8       ;@ CA[0] * SA[0] + CA[1] * SA[1]
|| [A1] B  S1       L3        ;@ Branch outer loop
||  ADD   .L1  A4,A5,A4       ;@ CA[2] * SA[0] + CA[3] * SA[1]
||  MPYH  .M2  B2,B1,B8       ;@@ CoefAddr[1] * StateAddr[1]
||  MPY   .M1X A0,B1,A4       ;@@ CoefAddr[2] * StateAddr[0]
||  LDW   .D2  *B6,B2         ;@@@@ CoefAddr[1] & CoefAddr[0]
||  LDW   .D1  *A3,A0         ;@@@@ CoefAddr[3] & CoefAddr[2]

    ADD   .D2  B4,B0,B9       ; x = t+((CA[0]*SA[0]+CA[1]*SA[1])>>15)
||  OR    .L2  B0,B9,B0       ; StateAddr[0] = t
||  SHR   .S2  B8,0xf,B4      ;@ (CA[0] * SA[0] + CA[1] * SA[1]) >> 15
||  SHR   .S1  A4,0xf,A5      ;@ (CA[2] * SA[0] + CA[3] * SA[1]) >> 15
||  MPY   .M2  B2,B1,B0       ;@@ CoefAddr[0] * StateAddr[0]
||  MPYH  .M1X A0,B1,A5       ;@@ CoefAddr[3] * StateAddr[1]
||  LDW   .D1  *A6,B1         ;@@@@ StateAddr[1] & StateAddr[0]

    STW   .D1   B0,*A7        ; store StateAddr[1] & StateAddr[0]
||  SHL   .S2   B5,0x10,B9    ;@ StateAddr[1] = StateAddr[0]
||  ADD   .L2X  A5,B9,B3      ;@ t = x+((CA[2]*SA[0]+CA[3]*SA[1])>>15)
|| [A1] ADD.L1  -1,A1,A1      ;@@ dec outer lp cntr
||  MV    .D2   B1,B5         ;@@ save StateAddr[1] & StateAddr[0]
```

▲ 33. Assembly optimizer 3-cycle loop from Fig. 31.

Compiler Performance

Table 7 shows C62x C compiler benchmark performance. This performance reflects refined C code without any use of the assembly optimizer. Depending on the metric used from the table, the C compiler achieves 72-82% across this set of algorithms. In larger loops, where the assembly writer has more difficulty managing the parallelism, the compiler provides comparable performance with greatly reduced development time. For more analysis of the C6x compiler performance see [9] and [10].

Summary

The TMS320C62x and TMS32067x are the first CPUs based on the VelociTI architecture. The VelociTI architectural principles allow performance of 1600 MIPS and 1 GFLOP, respectively. In addition, these principles allow creation of an optimizing tool set that reduces system development time. The TMS3206201 is the first device based on the VelociTI architecture. Advanced release samples have been shipping to customers since March 1997. Devices with the full production peripheral set have been available since October 1997. This peripheral set is suited for a wide variety of multichannel and high performance systems in communications and multimedia. The C6201 will migrate to a .18 micro process and 1.8 V internal supply voltage in mid 1998 for even lower power consumption.

Nat Seshan is an architect, product applications manger, and member of the Group Technical Staff for the Texas Instruments Semiconductor TMS320C62x DSP Group in Houston, Texas.

References

Web Sites

For a product overview, access to the literature listed below, and more detail on the TMS320C6201 as well as the source of the assembly benchmarks see: http://www.ti.com/sc/docs/dsps/products/c6x/index.htm

TMS320C6x Literature

The following books provide more detail on the TMS320C62x processor. Documentation and further details on the C67x will be available in the first quarter of 1998.

TMS320C62x Technical Brief: SPRU197

TMS320C62x CPU and Instruction Set Reference Guide: SPRU189B

TMS320C62x Peripherals Reference Guide: SPRU190.

TMS320C62x Programmer's Guide: SPRU198

TMS320C6201 Digital Signal Processor Data Sheet: SPRS051

(continued on page 117)

takes care of the register allocation, operation packing, and flow analysis.

Applications

TM-1000 has been designed into many multimedia applications such as video conferencing, multimedia accelerators in personal computers, DVD players, and high-definition television.

Summary

The TM-1000 is the first programmable multimedia processor from the Trimedia division of Philips Semiconductors. This article mainly discussed the VLIW CPU core in TM-1000. The VLIW CPU core is powerful enough to implement the MPEG2 video/audio decoder, video-conferencing applications, and the MPEG-1 encoder applications. Peripheral units such as an image coprocessor, video-in/video-out, audio-in/audio-out, PCI interface, and modem interface make up a complete multimedia system on a chip.

Selliah Rathnam is a Trimedia system architect and *Gert Slavenburg* is the Trimedia Chief Technology Officer with Philips Semiconductors in Sunnyvale, California.

References

1. J. Labrousse and G.A Slavenburg, "A 50 MHz Microprocessor with a VLIW Architecture." ISSCC, 1990.
2. J. Labrousse and G.A. Slavenburg, "CREATE-LIFE: A Design System for High Performances VLSI Circuits" ICCD-88. 1988.
3. J. Labrousse and G.A. Slavenburg, "CREATE-LIFE: A Modular Design Approach for High Performances ASICs." Compcon Conference, 1990.
4. Brian Case, "Philips Hopes to Displace DSPs with VLIW" *Microprocessor Report*, December 5, 1994.
5. Brian Case, "First Trimedia Chip Boards PCI Bus." *Microprocessor Report*, November 1995.
6. Gert Slavenburg, "The Trimedia VLIW-Based PCI Multimedia Processor" *Microprocessor Forum*, October 1995.
7. A.S. Huang, G. Slavenburg, and J.P. Shen, "Speculative Disambiguation: A Compilation Technique for Dynamic Memory Disambiguation". In *21st Annual International Symposium on Computer Architecture*, April 1994.
8. R.P. Colwell, R.P. Nix, J.J O'Donnell, D.B. Papworth, and P.K. Rodman, " A VLIW Architecture for a Trace Scheduling Compiler." *Proc. of ASPLOS II*. October 1987.
9. J.A. Fisher. "Trace Scheduling: A Technique for Global Microcode Compaction." *IEEE Trans. on Computers*, July 1981.
10. P.Y.T. Hsu and E.S. Davidson. "Highly Concurrent Scalar Processing." *Proc. of the 13th Symposium on Computer Architecture*, 1986
11. Selliah Rathnam and Gert Slavenburg. "An Architectural Overview of the Programmable Multimedia Processor, TM-1, 1996.

High VelociTI Processing

(continued from page 101)

Trademarks

TMS320 and VelociTI are trademarks of Texas Instruments. SPI is a Motorola trademark

Publications Referenced in this Article

1. J.A. Fisher, *Very Long Instruction Word Architectures*. 253, Yale University, 1983.
2. M. Lam, Software Pipelining: An effective scheduling technique for VLIW machines. *SIGPLAN '88 Conference on Programming Language Design and Implementation*, June 1988, pp. 318-328.
3. A. Davis, E. Stotzer, R. Tatge, and A. Ward, *Approaching Peak Performance with Compiled Code on a VLIW DSP*, July 1997, Texas Instruments.
4. D.A. Patterson and D.R. Ditzel, The case for the reduced instruction set computer. *Computer Architecture News* 8(6), October 1980, pp. 25-33.
5. T.J. Dillon, The VelociTI™ Architecture of the TMS320C6x, *Proceedings of the International Conference on Signal Processing & Technology*, San Diego, Sept. 1997.
6. P. Lapsley, J. Bier, A. Shohan, and E.A. Lee, *DSP Processor Fundamentals - Architectures and Features*, Berkeley Design Technology, Inc., 1996.
7. R. Scales, *Approaching Optimal Performance with the C62x Assembly Optimizer*, October 1997, Texas Instruments.
8. T.J. Dillon, The Use of Software Pipelining in Developing DSP Algorithms for the TMS320C6x, *Proceedings of the International Conference on Signal Processing & Technology*, Sept. 1997.
9. Loughborough Sound Images, PLC, *Evaluation of the Performance of the C6201 Processor & Compiler*, White Paper Version 1.1, 1997.
10. M. Levy, C Compilers for DSPs Flex Their Muscles, *EDN*, pp 93-107, June 5, 1997.