UNTIL RECENTLY, DESKTOP VIDEOCONFERENCING, MULTICHANNEL MODEMS, AND REAL-TIME VIDEO PROCESSING REQUIRED COMPLEX, EXPENSIVE SYSTEMS CONTAINING SEVERAL DIGITAL SIGNAL PROCESSORS (DSPS) AND A CONTROL PROCESSOR. LOWER COST SYSTEMS BECAME POSSIBLE IN 1994, WHEN TEXAS INSTRUMENTS INTRODUCED DSPS THAT INTEGRATE RISC, FLOATING-POINT, MULTIPLE HIGH-PERFORMANCE FIXED-POINT DSPS, ON-CHIP MEMORY CONTROLLER, AND STATIC RAM TECHNOLOGY ONTO A SINGLE CHIP.1,2 THE TMS320C8x ARCHITECTURE EXTENDS THE CORE FUNCTIONALITY OF DSPS FOR AUDIO AND COMMUNICATIONS PROCESSING WITH FEATURES THAT ENHANCE IMAGE PROCESSING AND VIDEO COMPRESSION PERFORMANCE.

The TMS320C8x is the first derivative of the C8x series. It maintains the core features of the original C80 but uses two DSPs instead of four to address users' cost and performance requirements. The C82 specifically supports PC-based multimedia applications such as desktop videoconferencing and embedded systems such as videophones.

The C82 enables a low system cost because it supports all of the digital processing for video, audio, system control, and modem on one chip. This reduces not only the processor chip count but also the amount of glue logic, memory, and other peripherals. Users can develop C82-based systems using a single DRAM-based memory subsystem that does not require external SRAM.

This article describes the C82 architecture and an implementation of the H.324 international standard for videoconferencing over analog phone lines.

C82 ARCHITECTURE

The single-chip C82 combines a RISC master processor, two advanced DSPs referred to as parallel processors, and an intelligent DMA engine and memory controller known as the transfer controller. The on-chip memory breaks down into eleven 4-Kbyte SRAMs that are connected to the processors through a crossbar switching network. Any of the on-chip processors can access the shared data and parameter RAMs in one cycle. Each processor executes its own independent instruction stream to allow different algorithms to run in parallel. The shared-memory architecture allows efficient communication and data sharing between processors (Figure 1, next page).

MASTER PROCESSOR. The 32-bit RISC master processor shown in Figure 2 targets efficient execution of high-level languages such as C. This 32-bit instruction word master processor contains an IEEE-compatible floating-point unit (FPU). Special vector floating-point operations allow a load or store working in parallel with the FPU to support single-cycle multiply-accumulates.

Like most RISC processors, the master processor carries out orthogonal three-operando data path operations, load or store memory operations, and one-cycle delayed branches with optional annul capability. Thirty-one 32-bit general-purpose registers and a zero register are common to both the floating-point and integer units. scoreboard logic keeps track of registers that receive results of loads or FPU operations and automatically prevents use of these registers until they have been updated.

A lightweight, event-driven multitasking executive kernel (shown in Figure 3) allows the various threads of execution typically required by multimedia applications.6 The kernel supports signals and messages between tasks, while an event manager decodes interrupts and informs tasks when interrupt-related events occur. A timer-manager task can use the master processor's hardware timer to maintain both periodic
and one-shot timers and notify tasks when timer events occur.

Figure 3 illustrates the role of the master processor in a PC-based environment. It controls on-chip parallel processing and supports communication between the C82 and other devices such as a host processor. Most high-level processing such as control protocols and floating-point-intensive algorithms execute directly on the master processor. For embedded systems, the master processor can also execute the application layer and/or user interface.

For repetitive, computationally intensive integer process-

and range of algorithms, including frequency domain transforms (like the discrete cosine transform or DCT), correlation, filtering, and bitstream parsing and encoding. As advanced DSPs, the parallel processors combine efficient multiply-accumulate processing with an enhanced ability to process pixels and manipulate bits to satisfy the requirements of image processing, video compression, and graphics.

As Figure 4 shows, each parallel processor has four major functional units: a data unit, two address units, and a program flow control unit. These parallel execution units can perform multiple operations in each cycle including a multiply (16×16

Figure 3. Master processor’s system role, showing parallel processors 0 and 1.
or two 8x8), three-input ALU operation (32-bit, two 16-bit, or four 8-bit), barrel rotation, mask generation, and two independent memory accesses with address updates. To specify all the parallel operations supported by the hardware, the parallel processors use a long instruction word of 64 bits.

Each parallel processor has 44 registers including eight data registers (d0-d7), 14 address registers, six index registers, two status registers, four program counter-related registers, interrupt enable and flag registers, cache tags, and numerous registers for hardware loop control.

**Data unit.** This unit (see Figure 5) has a novel 32-bit, three-input ALU and a flexible data path with several hardware units including a barrel rotator and a mask generator. Many operations that require multiple passes (and cycles) through most other ALUs execute in a single cycle.

The three-input ALU supports

![Figure 4. Parallel processor block diagram.](image)

![Figure 5. Parallel processor data unit.](image)
both Boolean and arithmetic operations. For Boolean operations, the ALU performs a sum-of-products, as

<table>
<thead>
<tr>
<th>Table 1. Base set of three-input arithmetic functions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &amp; indicates bitwise AND, ~ is negate (one’s complement), and</td>
</tr>
<tr>
<td>ALU function</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>A + B</td>
</tr>
<tr>
<td>A - B</td>
</tr>
<tr>
<td>A + C</td>
</tr>
<tr>
<td>A - C</td>
</tr>
<tr>
<td>A + (B&amp;C)</td>
</tr>
<tr>
<td>A - (B&amp;C)</td>
</tr>
<tr>
<td>A + (B~C)</td>
</tr>
<tr>
<td>A - (B~C)</td>
</tr>
<tr>
<td>A + (B&amp;~C)</td>
</tr>
<tr>
<td>A - (B&amp;~C)</td>
</tr>
<tr>
<td>A + b&gt;&gt;5n</td>
</tr>
<tr>
<td>A - b&gt;&gt;5n</td>
</tr>
<tr>
<td>(A&amp;C) + (B&amp;C)</td>
</tr>
<tr>
<td>A +</td>
</tr>
<tr>
<td>A -</td>
</tr>
<tr>
<td>A + (B&amp;C)</td>
</tr>
<tr>
<td>A - (B&amp;C)</td>
</tr>
</tbody>
</table>

* See example in Data path hardware box.
** See example in SAD box.

Boolean = F0 & (~A&~B&~C) | F1 & (A&~B&~C) |
| F2 & (~A&B&~C) | F3 & (A&B&~C) |
| F4 & (~A&~B&C) | F5 & (A&~B&C) |
| F6 & (~A&~C) | F7 & (A&B&C) |

where bits F0-F7 specified in the instruction select the appropriate Boolean minterms such as NOT A, NOT B, NOT C. The ALU can support all 256 possible Boolean (bitwise-logical) combinations of one, two, or three inputs.

For arithmetic operations, the ALU allows programmers to combine inputs using Boolean operators followed by an add or subtract in the same pass through the ALU. Programmers can specify a base set of the three-input arithmetic operations listed in Table 1 in a parallel processor's 64-bit instruction word.

Extended ALU operations use data register d0 together with the instruction word to increase a parallel processor's opcode to 96 bits. Extended ALUs support all 256 arithmetic operations allowed by

Arithmetic = A&B/(B,C) + F/(B,C) [+1]

where F/(B,C) is any Boolean combination of B and C. The square brackets indicate the optional addition of a 1 by forcing a carry into the ALU. This capability supports functions like A + B + 1, which are extremely useful for interpolation with rounding.

We can split a parallel processor's ALU to perform parallel arithmetic operations on 4 bytes (8 bit) or 2 halfwords (16 bit), as shown in Figure 6. Multiple arithmetic increases the performance of this ALU by four times for multiple byte operations or two times for multiple halfword operations.

Figure 6. Split-ALU arithmetic. A 32-bit ALU splits into four 8 bytes by not propagating a carry between the splits.
Using the data path hardware with the three-input ALU

The parallel processor ALU data path hardware, together with the three-input ALU, performs many powerful operations. The philosophy behind the architecture is to expose the hardware necessary to perform important functions like shifts and color expands as building blocks for the programmer. This allows the parallel processor to support a much wider range of operations than most other processors and is a key to its added efficiency at bitstream manipulation and pixel processing. The simple examples that follow show some interesting and useful operations performed efficiently using the flexible ALU data path.

The barrel rotator, mask generator, and three-input ALU together support all combinations of shift and add operations in a single cycle. For example, the parallel processor performs a shift-left-and-add using the following elemental functions:

- The barrel rotator (B) rotates an input to the left by the number of bits (n) to be shifted left. Bits that rotate out of the MSBs on the left wrap around into the LSBs on the right. The rotator output feeds the ALU’s B port. To perform a true shift left, the ALU must strip off the bits that wrap around.
- The shift amount (n) feeds to the mask generator to form a mask with the number of right-justified 1s equal to the shift amount. Note that the same amount is input to the barrel rotator control input and the mask generator input. The mask generator output feeds to the ALU’s C port.
- The ALU then combines the outputs of the barrel rotator and mask generator as part of the function A + B & C. The output of the barrel rotator goes to the B port and is ANDed with the inverse of the output of the mask generator fed to the C port. This strips off the bits that wrapped around in the rotation. Boolean operators have precedence in the ALU over arithmetic operators, so the AND occurs, producing the desired shift result prior to being added to the A port input.

We express shift-left-and-add in the parallel processor’s algebraic language using either shift symbol << or the explicit function based on barrel rotation \( \backslash \), mask generation \%, and the appropriate ALU function.

\[
dst = A + b << n = A + (b \backslash n \& \% n)
\]

Here, dst is the destination register for the shift-left-and-add result; A, the register input to the A port; b, the register input to the barrel rotator; and n, either a 5-bit constant or the five LSBS contained in a register.

The barrel rotator, mask generator, and ALU can combine to perform field extraction in a single cycle. They rotate the input left to right-justify the desired field and mask the rotated result according to the field size to strip off bits outside the field. Unlike shifts, the rotation amount is based on the starting bit position of the desired field, and the mask input is based on the field size, which is not necessarily the same.

The following expression defines the operation performed by the ALU data path:

\[
\text{FieldExtract} = \text{value} \backslash (32 - \text{StartingBit}) \& \% \text{FieldSize}
\]

The \% operator uses the mask generator to form a mask that has the number of right-justified 1s equal to the field size. If desired, the ALU could perform an add on the extracted field in the same cycle.

Add-with-saturate operations are often necessary in graphics and image processing to prevent pixel processing results that overflow the output precision from rolling over from the highest number back to a low number. Figure A uses the MF expander and multiple-byte split-ALU arithmetic to perform an add-with-saturate on four pairs of 8-bit pixels in two instructions. The first instruction performs a multiple-byte add saving the carry-outs from each split-ALU segment into the MF register. Each carry bit set to 1 indicates that the corresponding result was too big for an 8-bit value and needs to be saturated or clipped to a maximum value. The second instruction selects either the result from the addition or the saturation level of 0xff for each pixel based on the expanded MF.

<table>
<thead>
<tr>
<th>First instruction</th>
<th>operand (OP1)</th>
<th>OP2</th>
<th>result (OP1 + OP2)</th>
<th>carry Outs</th>
<th>MF result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>(A+B)</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Second instruction</th>
<th>expanded MF (@mf)</th>
<th>result (RESULT @ MF)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C</td>
<td>(A+B)</td>
</tr>
</tbody>
</table>

Figure A. Add-with-saturate on four 8-bit numbers.

Since image processing and video compression typically deal with byte-sized data, almost all core algorithms can use multiple arithmetic.

For multiple arithmetic operations, we can save status from the split-ALU operations based on sign, carry, zero result, or extended-precision data. A parallel processor's multiple flags (MF) register stores this status information with 1 bit for each ALU partition (4 bits set for byte, 2 bits for halfword, and 1...
**SAD: Summing the absolute value of differences**

Video compression algorithms use temporal redundancy (the similarity between one frame and the next) to obtain a high degree of compression. Full exploitation of temporal redundancy in moving sequences requires that motion estimation identify where objects in the current frame were located in a previously transmitted or decoded frame—a reference frame. The correlation measure used most frequently is the sum of the absolute value of differences (SAD). SAD is computed between the current frame block and blocks in the reference frame at various displacements.

For a full search over a plus or minus 7 displacement range at QCIF resolution (176x144 pixels) and 10 frames/s, the SAD computation would have to be performed over 57 million times each second. Although intelligent search algorithms decrease this number considerably, SAD or a similar correlation measure must be performed efficiently to obtain the full quality supported by compression standards such as H.263.

The code in Figure B uses multiple-byte arithmetic, the MF register, and the MF expander to perform the SAD computation on eight pairs of pixels in four instructions (effectively 0.5 cycles/pixel).

In the first instruction, the data unit operation performs a multiple-byte subtract between 4 pixels in the current frame and 4 pixels in the reference frame. The MF register stores the split-ALU carry-outs to record whether the result for each split of the ALU was positive (carry = 1) or negative (carry = 0).

In the second instruction, the data unit performs the ALU function A + (B&C | (-B&C)). The expander extends the saved carry to create a mask based on the sign of each of the previous 8-bit differences. This mask is routed to the ALU's C port to cause addition of positive differences and subtraction of negative differences to the running sum of the absolute value of differences.

The first two instructions effectively accumulate four byte-sized absolute differences to four byte-sized error sums. However, there is one additional complexity because the SAD can overflow 8 bits after each accumulate. To allow overflow tracking, the MF register stores the split-ALU carry-outs from the A + (B&C | (-B&C)) operation.

The third data unit operation rotates the carry bits in the MF register left by four so that they will not be overwritten. It also performs a multiple-byte subtract between four more current and reference pixels. The fourth data unit operation performs another SAD computation. At this point, there are eight carry flags on the MF register. Each carry flag with a value of 1 indicates an overflow equivalent to an absolute difference of 256.

In the next pass through the loop, parallel operations in the global address unit move the carry flags in the MF register to an index register and perform a table lookup to identify the number of overflows that occurred. An address unit adder then adds this value to a running sum of the overflow count. After the final pass through the inner loop, the four-byte absolute difference sums are combined. The total number of overflows times 256 plus the combined difference sum produces the correct total SAD between the current and reference blocks.

---

**Figure B.** SAD inner loop.

bit for word).

The expander can take 1, 2, or 4 bits in the MF register and replicate them 32, 16, or 8 times according to the multiple-data size. This will fill out a 32-bit word that is routed to the ALU's C input port. For example, using multiple byte arithmetic with 1010 in the four LSBs (least significant bits) of the
Figure 7. Address unit data path (PP-rel: parallel processor-relative addressing).

MF: the output of the MF expander (symbolized by $\delta m$) is 0xff00ff00. The MF expander can use status information from a previous operation to form a mask that controls the operation performed by the ALU. In the SAD box, we use this feature to sum the absolute value of the difference (SAD) between two data sets at a rate of 0.5 cycles per pixel.

Shifts are a fundamental element of DSP algorithms. While many architectures provide a shift unit, a parallel processor supports shifts by allowing the user access to a barrel rotator and a mask generator. This key feature permits rotations, which are very useful in bit manipulation.

The barrel rotator can rotate an incoming 32-bit quantity by 0 to 31 bits. The mask generator produces 0- to 32-bit-length masks that are used with the barrel rotator to generate signed or unsigned right or left shift operations. The mask generator operation (symbolized by $\%$) takes a 5-bit input $n$ and produces a mask with $n$ right-justified 1s with zero extension to fill a 32-bit word. We express this mathematically as $\% = 2^n - 1$. The Data path hardware box demonstrates how to use the barrel rotator and mask generator combination to perform shifts as well as other useful operations like field extraction.

The bit detector returns either the bit number of the most significant nonzero bit in a register (leftmost one), the least significant nonzero bit in a register (rightmost one), the most significant bit (MSB) that is not the same as bit 31 (leftmost bit change), or the LSB that is not the same as bit 0 (rightmost bit change). These functions are useful in Huffman encoding and decoding.

The parallel processor’s core 16-bitx16-bit multiplier produces a 32-bit result. In addition, the multiplier has two unique features: split and rounded multiply. Split multiply performs two 8-bitx8-bit multiplies to produce two 16-bit results. Rounded multiply improves precision with fixed-point arithmetic for functions such as the inverse DCT (IDCT). Then as an option, results can be shifted left to adjust the integer portion of the result to a desired bit position.

Address units: Each parallel processor has a global address unit and a local address unit. This allows two independent memory accesses in parallel with the data unit in a single instruction. The two units differ only in the registers they operate on. Both units can access any of the on-chip shared RAMs in a single cycle, provided there is no contention.

As shown in Figure 7, each address unit has five general-purpose address registers, three index registers, and its own data path including a 32-bit adder for computing addresses. Single stack pointer SP is mapped into both register files, and each unit has a zero register that always reads as the constant 0 for absolute addressing.

The address units support the 12 indexed addressing modes listed in Table 2 (next page). The modes include adding or subtracting either the value in an index register or an immediate offset to/from the base address specified in an address register. The index is optionally scaled by the data size before combining it with the address register value. Either the address register before modification (postindexing)
or the result of combining the address and scaled index (preindexing) serves as the address for the memory access. We can update the address register with the adder/subtractor result to facilitate stepping through memory.

Both units can load or store a byte (8 bits), halfword (16 bits), or word (32 bits) in a single cycle. An instruction can use data loaded by the previous instruction. For loads, byte and halfword data are always aligned right and then either zero-extended for unsigned data or sign-extended for signed data to fill 32 bits.

Special parallel processor-relative addressing allows local data RAM0 (dba) or the local parameter RAM (pba) base address to serve as the base address for the address computation. This addressing lets programmers develop object code that can run on either parallel processor.

The global address unit supports a wide variety of register-to-register moves. The basic move takes the 32 bits in the source register and writes them to the destination register. A field-extract move takes any byte or halfword in the source register, sign- or zero-extends the data to 32 bits, and writes the result to the destination register. A field-replicate move duplicates the rightmost byte or halfword in the source register to fill 32 bits and writes the result to the destination register.

The address unit adders can perform general-purpose additions or subtractions. This feature can off-load some of the data processing from the data unit for algorithms that are addition or subtraction bound.

**Program flow control unit.** The PFC unit shown in the lower section of Figure 4 controls the parallel processor instruction pipeline, fetches and decodes instructions, manages any necessary handshaking with the transfer controller, and handles interrupt response and prioritization. The major hardware elements of the PFC are the instruction decode control, the program counter registers, the instruction cache controller, and three hardware loop controllers. These allow the PFC to support conditional branches, interrupts, zero-overflow looping, and sequential program execution.

A parallel processor has a three-stage instruction pipeline consisting of instruction fetch, address generation, and operation execution. Since at any given time three instructions are in the pipe, the net throughput is one cycle per instruction. All these instructions execute in one cycle.

The program counter (PC) register is mapped into the register file so software can write to it. A data unit operation, move, or load performs branches and calls with the PC register as the destination. Branches do not affect the PC until the end of the third pipeline stage, and thus, two delay slot instructions that are always executed follow each branch. Since video-coding algorithms often deal with data-dependent operations, each parallel processor supports conditional operation of all data unit operations, moves, and loads from memory including branches. Being able to perform conditionally reduces the need for conditional branching.

Each parallel processor also includes three hardware loop controllers that support up to three levels of nested loops. Because this hardware eliminates execution time spent on loop counting and branching, we refer to them as zero-overflow loops. Overhead is minimized so that for most algorithms the parallel processor achieves essentially the theoretical performance based on the number of core operations such as multiplications or additions.

**Instruction set.** Each parallel processor uses a 64-bit instruction opcode to specify independent parallel operations performed by the data unit and both address units. When programming a parallel processor, it is useful to think of an instruction as consisting of multiple subinstructions referred to as operations for the multiplier, ALU data path, global address unit, and local address unit. A sample instruction is

\[
d1 = d2 \cdot d3 ; \quad \text{Multiply performed by data unit multiplier}
\]

\[
d4 = d5 + d6, d0 ; \quad \text{Shift and add performed by three-input ALU}
\]

\[
d8++ = d7 ; \quad \text{Postincrement store performed by global address unit}
\]

\[
d0 = *(a0 + [x1]) ; \quad \text{Scaled indexed load performed by local address unit}
\]

The combination of the three-input ALU and the different data path routings into the ALU allows literally thousands of unique operations. Also, the address units provide 12 addressing modes with several options, such as index scaling and parallel processor-relative addressing. Rather than naming these thousands of operations, an algebraic assembler accepts assignment statements that define the operation. Since numerous equivalent ways exist to specify the same Boolean operation (for example, (A & B) | (¬(A & B)) is equivalent to A ⊕ B), the assembler includes an expression evaluator. Then, the programmer does not have to use a specific form for a given operation.
While we describe the instruction set primarily as an assembly language, programmers also need a general knowledge of the opcode formats to understand which operations they can specify in parallel. As shown in Figure 8, the opcodes for most parallel processor instructions are partitioned into three fields that specify operations for the data unit, local address unit (referred to as a local transfer), and global address unit (referred to as a global transfer). This allows each unit to specify independent operations. The data unit field defines the multiplier and ALU data path operations.

Some operations require bits from the opcode fields that are normally used for one or both of the address units. To specify a long immediate offset for one address unit requires the opcode bits that would have specified the other address unit's operation. Data unit operations with a 32-bit immediate operand require the opcode bits used to specify both the global and local transfer operations. Thus, no parallel address unit operations can be specified. Conditional data unit or global transfer operations require the local transfer portion of the opcode. Using non-D register (any register other than do or d7) operands in the data unit requires the global transfer portion of the opcode.

Performance. The parallel processor's advanced features enhance performance across the wide range of algorithms required by video compression standards such as H.263. Table 3 lists some examples of the parallel processor performance. The C82 will be offered at 50- and 60-MHz clock speeds.

Transfer controller. The controller shown in Figure 9 (next page) prioritizes, schedules, and transfers data between on- and off-chip memories. The memory interface supports a wide range of external memory systems including DRAM, synchronous DRAM, video RAM, and SRAM. The transfer controller generates all the necessary control signals.

As shown in Figure 10, the architecture supports multidimensional data transfers referred to as packet transfers. The transferred data can be a simple contiguous linear sequence or up to a 3D region. Data items along the first dimension are always 1 byte apart. The P pitch in the packet transfer parameters defines the distance between entries in the second dimension. Likewise, the G pitch defines the distance between entries in the third dimension.

The architecture specifies source and destination transfers independently to maximize flexibility. For example, in Figure 10 the parallel processor defines the source data as a set of three 8x8 blocks within an image in external memory and the destination as a sequential 1D array in the on-chip RAM. Once the parallel processor sets up the structure of parameters that define the desired transfer, it issues a packet transfer request to the transfer controller. The controller then performs the requested transfer in parallel with operations performed by the parallel processor. The transfer controller's support of these independent data transfers allows the master and parallel processors to focus on processing data rather than on collecting or distributing it.

Development tools, programming environment

A full suite of development tools, including C compilers and assemblers for both master and parallel processors, supports the C8x line. Code debugging takes place in real time using on-chip emulation logic and running parallel high-level language debuggers for the master processor and each parallel processor.
Since programmers write most parallel processor code in assembly language, several tools make the programming easier. An algebraic assembler allows programmers to write code using a C-like language with additional operators added for unique features like the mask generator and the MF expander. The similarity of the parallel processor with C makes it easy to read. There’s no need to memorize a long list of mnemonics for the many different supported operations.

The register allocator lets programmers write assembly code using variable names. This allocator identifies live and free registers on an instruction-by-instruction basis and assigns specific registers to each variable. It eases code writing and maintenance since there’s no need to worry about overwriting live registers.

The code compactor lets programmers write straight-line assembly code without specifying parallel operations. The compactor creates parallel code using the rules about which operations can process together in a single instruction. Programmers typically write the most time-critical parallel code by hand because they can further optimize performance by reworking the algorithm to best exploit the parallel processor’s rich feature set. However, the code compactor allows quick development of setup, termination, and other noncritical code, allowing programmers to focus on the critical loops.

**H.324 standard**

The ITU (International Telecommunications Union) developed the H.324 standard to support videoconferencing over analog phone lines. Currently, these systems allow data rates of up to 28.8 Kbps using a V.34 modem. As shown in Figure 11, H.324 encompasses several standards, including:

- H.245—defines the control protocol for H.324 systems including the capability exchange procedure;
- H.223—defines the frame structure, field formats, and packet multiplexing protocol;
- H.263—defines the videocoding syntax for low bit-rate video compression; and
- G.723—defines a speech coder for transmitting at 5.3 and 6.3 Kbps.

**H.324 on the C82**

H.324 requires massive amounts of mathematically intensive processing including video, audio, and system processing. The C82's flexible multiple-instruction, multiple-data (MIMD) architecture permits implementation of all these code modules on a single chip.

**Task partitioning.** The first step in implementing a large application on the C82 is to determine the task partitioning. H.324 breaks down naturally into its component standards. Some of these standards, like H.263, are then partitioned further into independent tasks for encoding and decoding to make the implementation as flexible as possible.

Software design begins by coding the various modules in C. This keeps algorithms fluid during the planning stages. System control tasks like H.245 and H.223 can remain mostly...
in C and execute directly on the RISC master processor. Programmers typically rewrite the more computationally intensive portions of code—audio and video processing—in optimized assembly, but the high-level code is useful to create a verification mechanism.

Figure 12 illustrates how the H.324 modules and the V.34 modem map onto the C82. The master processor performs all high-level processing, including H.245 system control, H.223 packet multiplexing and demultiplexing, video and audio drivers, and modem system control. The master processor also has video and audio tasks that interact with the system layer and send commands to the parallel processors, which perform the bulk of the video and audio processing. Parallel processor 0 processes video, while parallel processor 1 processes audio, including G.723 and acoustic echo cancellation (AEC) and the V.34 modem data pump.

**Multitasking model.** The master processor’s multitasking executive kernel sends messages and signals between the various H.324 tasks. Figure 13 (next page) shows the various master processor tasks for H.324 and how they interact. In a PC-based videoconferencing system, the host processor off-loads the H.324 processing to the C82. An internode message manager running on the host converts function calls from the host application to request messages.

The manager transmits these requests to the master processor, which handles all communications with the client processes on the host. The master processor also has an inter node message manager. It takes the messages from the host and routes the requests to the appropriate server tasks running on the master processor.

**Video processing (H.263).** The master processor has tasks for H.263 encoding and decoding that manage buffering of image and bitstream data, and prepare commands to parallel processor 0. This processor performs the core H.263 processing as well as pre- and postprocessing.

Baseline H.263 operations, including half-pel motion esti-
mation for QCIF (176x144) resolution at 10 frames per second, consume less than 15 parallel processor MIPS. This means parallel processor 0 has about 45 spare MIPS at 60 MHz to perform encoding options and intelligent quantization control algorithms to enhance video quality. This performance headroom is critical because video quality is the biggest barrier to widespread use of videophones and videoconferencing over analog phone lines. Activity in the ITU on new extensions to H.263 promises to provide ongoing enhancements to video quality at the cost of more MIPS.

The master processor’s H.263 decoder task starts each time the H.223 receiver has extracted a minimum number of video bitstream bits. The master processor task then issues a decode command to parallel processor 0. The core functions of H.263 decoding are Huffman or arithmetic decoding, inverse DCT, overlapped block motion compensation, and pixel reconstruction (add with saturate).

A parallel processor notifies the master processor each time it decodes an entire frame. The master processor then issues a postprocessing command to the parallel processor to reduce typical artifacts of DCT-based compression. These include “blockiness” resulting from the 8x8 block processing and mosquito noise that is a side effect of quantization noticeable around edges.

The video capture driver that runs on the master processor tracks the number of received frames and initiates the H.263 encoder task periodically based on the target frame rate. For example, if the target frame rate is 10 frames per second, it signals the encoder every third captured frame.

Video encoding occurs in three steps that are issued as commands to the parallel processor. The first step is to preprocess the input frame using a temporal filter to reduce flicker and camera noise. This is a critical step since noise introduces undesired high-frequency components that are costly to encode.

After preprocessing, the master processor sends a command to parallel processor 0 to estimate motion on a set of macroblocks. This step creates a set of macroblock parameters including the selected coding mode, motion vectors, and target bit allocation.

H.263 allows many motion estimation options including half-pel motion estimation and overlapped block motion compensation. These options significantly enhance the video quality produced by H.263. The C82 fully supports these features, though many other implementations must cut corners and thus cannot provide the full video quality allowed by H.263.

The final step is H.263 encoding. The critical elements are the DCT (which converts spatial image data into the frequency domain), quantization, zigzag scanning for run-length encoding of nonzero quantized DCT coefficients, and Huffman encoding. It is also necessary to essentially mimic the far-end decoder to create a reference frame for use by the motion estimation step. This requires inverse quantization (inverse DCT) and pixel reconstruction.

Figure 13. C82 H.324 tasking model.
The H.263 encoder must adjust the quantization level to stay within the bit-rate constraints of the communication channel (typically about 20 Kbps for video using a V.34 modem at 28.8 Kbps). A flexible architecture like the C82 can adapt encoding based on sequence characteristics and minimize the amount of bits spent on noncritical information. Slight background lighting changes can be ignored to focus on more important image features like facial expressions.

**Audio processing.** The C82 implements G.723 speech compression and decompression, and acoustic echo cancellation on parallel processor 1. G.723 encodes speech in 30-ms frames using linear predictive coding. The G.723 recommendation, defined in terms of bit-exact, fixed-point mathematical operations, is very computationally intensive. The most crucial element is a multiply-accumulate using special saturation handling found in some audio DSPs and the C82's parallel processors. Even with single-cycle support for the saturated multiply-accumulate, G.723 encoding requires about 22 parallel processor MIPS, and the decoder requires about 3 parallel processor MIPS.

Although not embodied in the H.324 standard, acoustic echo cancellation (AEC) is a requirement for videoconferencing. The current AEC implementation on the C82 uses an LMS (least mean squared) adaptive filtering algorithm that can cancel up to a 64-ms echo tail using approximately 10 parallel processor MIPS.

**Modem.** While early prototypes of H.324 on the C82 use an external modem, the C82 reserves sufficient performance to run the V.34 modem. We will partition modem processing across the master processor and parallel processor 1. The modem system controller will be implemented on the master processor, and the modem data pump, on the parallel processor. Rough initial estimates indicate that the data pump will require less than 20 parallel processor MIPS.

The C82 multimedia digital signal processor supports desktop videoconferencing and videophones. Its performance lets system developers take full advantage of options within standards like H.324 and add differentiating features. Programmers can continue to improve software as better algorithms develop. Also, the C82 supports proprietary algorithms such as lower bit-rate speech coders and upcoming standards like MPEG-4, which may rely more heavily on non-DCT-based techniques such as object-based coding and wavelets.

The C82 also supports other computationally intensive applications: 3D virtual reality, H.320 videoconferencing, real-time MPEG-1 decompression, video editing, high-quality audio processing, and document image processing. [1]

**References**

2. TMS320C80 System-Level Synopsis, Texas Instruments, at 4389750@mci.com, 1995.

**Jeremiah Golston** is the TMS320C8x video applications manager and a member of the group technical staff at Texas Instruments in Houston. He leads a team implementing H.324 on the C82 and developing algorithmic enhancements to both the H.261 and H.263 C8x implementations. He was one of the original C80 applications engineers. He helped define algorithm-driven features in the C8x architecture, implemented H.261 on the C80, and was the project leader for TI's H.320 software library for videoconferencing over ISDN. His interests include algorithm and architecture development for multimedia applications.

Golston received BSEE and MSEE degrees from the University of Missouri-Rolla. He is a member of Eta Kappa Nu.

Direct questions concerning this article to the Texas Instruments Product Information Center, (214) 644-5580; 4389750@mci.com; WWW at http://www.ti.com.

**Reader Interest Survey**

Indicate your interest in this article by circling the appropriate number on the Reader Service Card.

<table>
<thead>
<tr>
<th>Level</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>153</td>
</tr>
<tr>
<td>Medium</td>
<td>154</td>
</tr>
<tr>
<td>High</td>
<td>155</td>
</tr>
</tbody>
</table>

August 1996  33