

---

# BROADCOM MEDIADSP: A PLATFORM FOR BUILDING PROGRAMMABLE MULTICORE VIDEO PROCESSORS

---

BROADCOM'S MEDIADSP TECHNOLOGY IS A MODULAR AND SCALABLE MULTIPROCESSOR PLATFORM THAT PROVIDES THE FRAMEWORK FOR BUILDING CUSTOMIZED AND PROGRAMMABLE MULTICORE PROCESSORS FOR THE VIDEO AND IMAGE PROCESSING DOMAIN. THE MEDIADSP PLATFORM EXPLOITS TASK-LEVEL PARALLELISM AND INCLUDES ARCHITECTURAL ELEMENTS THAT SUPPORT A TASK-BASED PROGRAMMING MODEL. BROADCOM'S BCM35421 PROCESSOR LEVERAGES MEDIADSP TECHNOLOGY TO REALIZE A SINGLE-CHIP, FULL HIGH DEFINITION (FHD) FRAME RATE CONVERTER FOR TODAY'S 120 HZ LCD TV SETS.

**Richard Selvaggi**  
**Larry Pearlstein**  
Broadcom

.....Designers of digital circuitry for real-time processing of video data in consumer electronics applications face the conflicting requirements of high computational capability and low hardware cost. To address these requirements, many commercial chips for consumer video processing use fixed-function logic designed to perform a set of specific tasks (for example, video decoding, scaling, and noise reduction). However, several factors motivate the development of a more flexible approach:

- Some applications must support many video-coding standards, including some that are in a state of flux.
- Several important video-processing problems are ill posed, admitting no ultimate solution.
- Certain product requirements have diverse mutually exclusive use cases.

- New business opportunities must be addressed without the cost and time of developing a new chip.

Real-time video processing involves producing data at high rates—from 27 MHz for standard-definition TV up to 450 MHz for full high-definition TV. Depending on algorithm complexity, the required processing rates can exceed 100 billion operations per second. A programmable approach should therefore be scalable to very high performance levels.

The mediaDSP technology is a scalable platform for rapid development of cost-effective solutions to a variety of video-processing problems. The platform uses a heterogeneous multicore approach, with a task-based programming model and a uniform approach for managing tasks executing on different types of programmable and

fixed-function processing elements. The modular mediaDSP multicore architecture allows easy customization for specific workloads, and the platform's programmability enables tuning and algorithmic enhancement after silicon is frozen.

Thus far, two diverse chip product lines have used the mediaDSP platform. The first application of mediaDSP was in a family of MPEG audio/video encoder chips, which also performed scaling and noise reduction and processed standard-definition video. Most recently, the mediaDSP platform served as the basis for developing the video-processing subsystem in the Broadcom BCM35421 chip. The BCM35421 hosts complex algorithms for performing frame-rate conversion on full high-definition video. Whereas the initial application of mediaDSP led to a seven-core architecture, the most recent result is a 44-core architecture.

### Classes of video processing

The Broadcom mediaDSP platform addresses problems in the general domain of video and image processing. A typical problem within this domain requires the implementation of an algorithm for compression, enhancement, analysis, or synthesis of arrays of pixel data. Although many video- and image-processing algorithms readily admit an extraordinary degree of parallel processing, they might include portions in which parallelization is intractable. In creating the mediaDSP platform, the Broadcom team considered a variety of potential applications and identified a set of four key processing classes.

The first class consists of *highly parallelizable operations*. Pixel data is the most common data type for these operations, but in many cases operations on motion vectors, picture statistics, and other types of data can be parallelized as well. This data is almost exclusively integer and fixed-point data, as opposed to floating-point data. A processor with a high degree of data-level parallelism, such as a programmable fixed-point single-instruction multiple-data (SIMD) processor or a specialized datapath-intensive engine, is a good fit for this class of operations.

The second class of processing is *ad hoc computation and decision making*. These processes often operate on smaller sets of data

produced by the parallelizable processes, and a general-purpose processor is well suited to handle this class of operations. These jobs typically exhibit an order of magnitude less computational complexity than the parallelizable operations.

The next class of processing is *data movement and formatting*, which requires address-processing intensive engines such as direct memory access (DMA) engines. For many video applications, multiple pixel data planes (such as luminance and chrominance data) are processed at one time, so a DMA engine capable of addressing multidimensional pixel maps is appropriate.

The fourth class of operations is *bit serial processing*, which appears in the entropy encode and decode portions of video codec applications. These operations can easily overwhelm reduced-instruction-set-computing (RISC) processors, so a programmable bit-stream processor that can handle multiple codec standards is a good example of an accelerator for these operations.

Realizing that video algorithms generally include these multiple classes of operations, we concluded that an architecture that accommodates multiple cores of various characteristics is both cost effective and power efficient. Furthermore, if these cores are developed using a common set of guidelines (that is, using a platform-based approach), the architecture is easily extendable to support a range of applications.

### Task-based programming model

A central concept of mediaDSP is the exploitation of task-level parallelism (whereas data-level parallelism is also exploited within the system's individual processors). To address this concept, we partition an algorithm into a set of parallelizable tasks and then efficiently map it to the massively parallel architecture. A task is a well-defined unit of work that must be done. It has a definite initiation time and runs until completion with no interruption and no further synchronization with other tasks.

Because video- and image-processing algorithm developers often use task flow graphs to describe algorithms, Broadcom adopted a task-based programming model<sup>1-5</sup> for mapping sequential algorithms into

parallel realizations. The mediaDSP task flow model differs from the classical Kahn model<sup>5</sup> (which uses a first-in, first-out (FIFO) buffer in the receiver) in that it explicitly identifies buffers residing in a shared memory space for passing data from one task to another.

The first step in programming a mediaDSP processor is to develop a task flow graph that conveys how the algorithm is partitioned into appropriately sized units for eventual mapping to the hardware resources. The task flow graph also exposes the serial and parallel portions of the algorithm.

Elements in the task flow graph include the task instances, the arcs representing a task's input and output, and the buffers that hold data produced or consumed by the connected tasks.

These elements are mapped to architectural resources in the mediaDSP platform such as processors, interconnect technologies, and memory, respectively. The elements' properties dictate the architectural resources' requirements. For example, the memory resources must be sufficient to hold all of the buffers in the task flow graph, and the interconnect must have enough bandwidth to meet the tasks' I/O demands. These decisions drive the architectural configuration required to meet a specific application's needs.

Tasks are synchronized in mediaDSP via a pair of semaphores that convey a buffer's state, specifically, *ready for writing*, *being written*, *ready for reading*, and *being read*.

In this scheme, one semaphore is located near the producing processor and the other is near the consuming processor. This lets all polling operations remain local to the polling processor, and the only communication between processors is a single write to modify the remote semaphore. Specifically, before writing, the producer task does local polling to implement a test-and-set operation on the *buffer busy* semaphore. After writing to the buffer, the producer task sends a message to set the *data available* semaphore in the consumer task. The consumer task does local polling to implement a test-and-clear operation on the *data available* semaphore. After reading the buffer, it simply clears the *buffer busy* semaphore in the producer.

## Platform-based architecture

The mediaDSP platform provides the framework to build customized architectural embodiments targeting specific applications. It includes the following architectural elements to support the task-based programming model (see Figure 1):

- task-oriented engine (TOE),
- task control unit (TCU),
- control engine,
- shared memory, and
- communication fabric.

A TOE is a hardware processing engine—either programmable or fixed function—that executes a task and then halts, waiting for its next task. The platform is heterogeneous because TOEs can be of multiple varieties, and it's extensible because new types of TOEs can be added for new applications. The use of application-specific TOEs optimizes power and performance per area.

Multicore architectures aim to maximize simultaneous utilization of all the cores. The mediaDSP TCU facilitates this process. The TCU is a specialized control unit that is closely coupled with the TOE. It maintains a queue of tasks to be issued to the TOE, and synchronizes with other TCUs and control engines in the system.

The control engine is a general-purpose RISC processor whose primary job is to orchestrate tasks for maximum parallelism. However, it can also function as a virtual TOE for the task flow graph's ad hoc tasks. Orchestrating tasks involves loading the task descriptors and commands for the task flow graph's control flow into the TCUs of the TOEs. We can include multiple control engines if a single engine doesn't have enough processing power to perform the required duties.

The mediaDSP platform's remaining structural elements include shared memory and the communication fabric. The shared memory is wide on-chip memory accessible by all TOEs and other processor elements. Its primary role is to hold intertask data buffers. The communication fabric is the interconnect network among the cores.

To better illustrate these capabilities, Figure 1 shows how the architectural elements are assembled in a generic mediaDSP embodiment. At a high level, this diagram

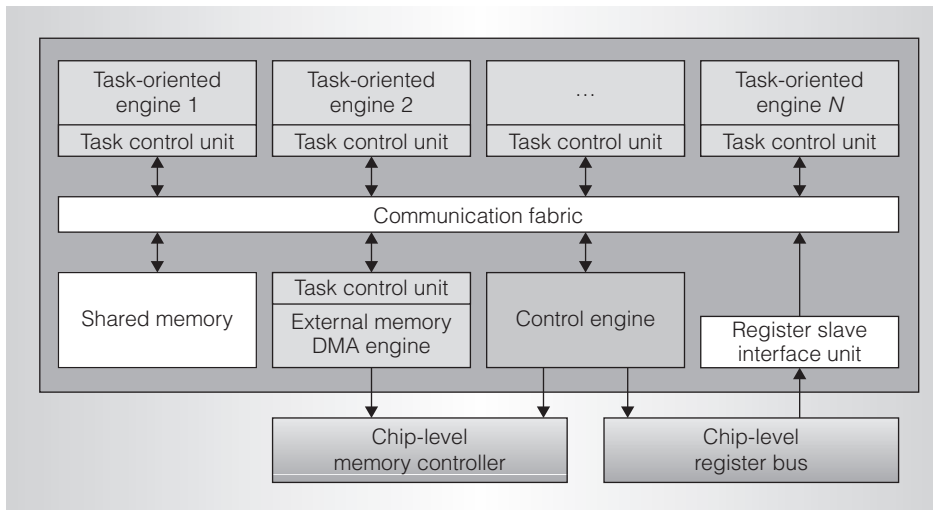


Figure 1. The mediaDSP generic topology. The basic configuration is a shared-memory multiprocessor with an array of application-specific task-oriented engines (TOEs), an external direct memory access (DMA) engine, and one or more reduced-instruction-set-computing- (RISC-) based control engines.

represents a shared-memory multiprocessor, with all engines connected via the communication fabric. A scalable array of TOEs extends along the top of the figure. A TCU resides between each TOE and the communication fabric.

The external memory DMA engine is an example of a TOE that has an interface to external memory. Any mediaDSP embodiment has at least one control engine, but the mediaDSP platform supports multiple control engines. Each control engine has direct connections to the chip-level register bus and the chip-level memory controller. As a result, the memory accesses for program execution don't consume any bandwidth on the communication fabric. Finally, all mediaDSP state is accessible from the top level of the chip via the register slave interface unit (RSIU). Multiple mediaDSPs can be instantiated within a single chip.

### Memory hierarchy

We designed the mediaDSP's memory hierarchy to reduce processor stalls, reduce bandwidth to off-chip memory, and achieve deterministic execution time.

As Figure 2 shows, the mediaDSP technology uses a multilevel memory hierarchy with DMA engines that move data through-out the levels.

The lowest level in the memory hierarchy is typically implemented in an off-chip dynamic RAM. In a system-on-chip (SoC) environment, this memory is highly contended for, and latency per access can range widely.

The shared memory is the next level in the hierarchy; it's typically implemented in one or more static RAM arrays. One use of shared memory is as a staging area for data that's transferred to or from off-chip memory. This staging area lets the programmer schedule DMA transfers between these levels early enough so the processors consuming or producing the data don't stall because of the high latency of the access to DRAM. Shared memory also holds buffers for inter-TOE communications, thereby reducing off-chip bandwidth when TOEs read data more than once.

Shared memory is located on chip and has low access latency, but multiple TOEs contend for this common resource. Accordingly, the TOEs generally have local memory at the highest level of the memory hierarchy, to which they have sole, unimpeded access on every cycle.

The control engines have a typical embedded RISC processor memory hierarchy, in which separate L1 caches hold instructions and data, and an internal register file holds frequently accessed variables.

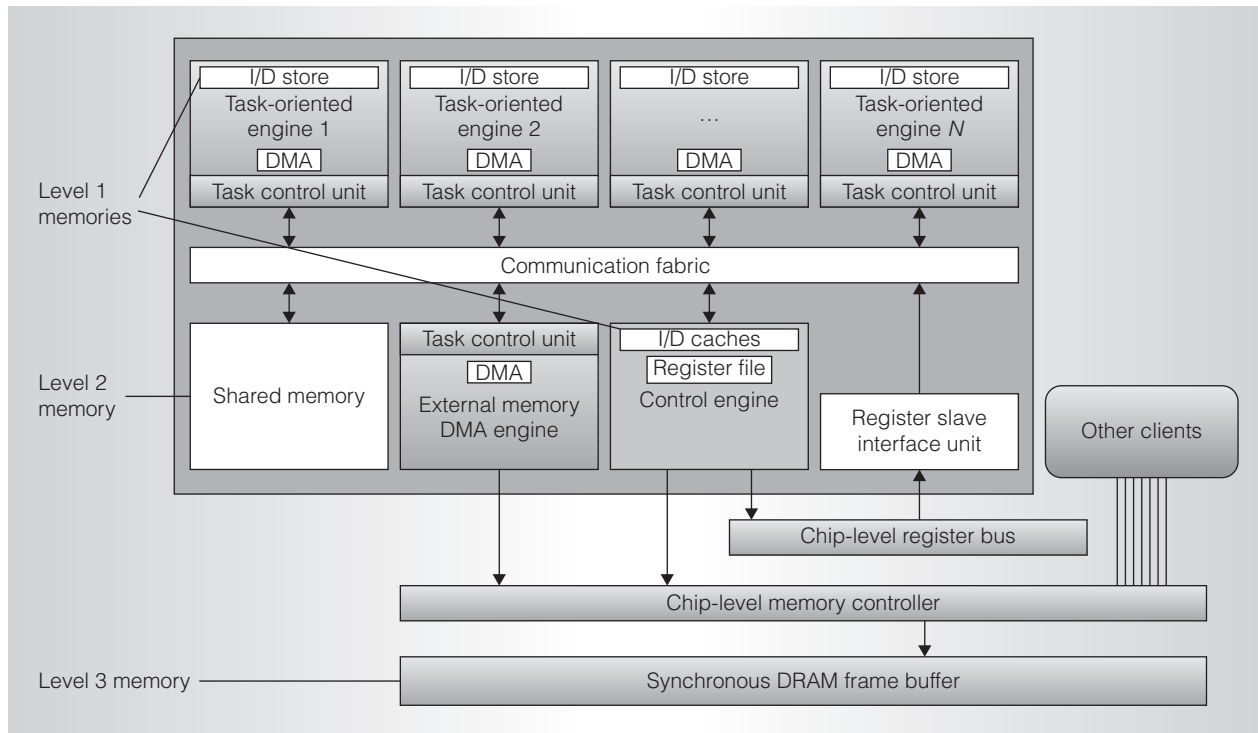


Figure 2. Memory hierarchy of mediaDSP. To reduce memory latency effects in a system on chip (SoC), mediaDSP has a multilevel memory hierarchy with DMA engines to move data throughout the levels.

All TOEs use software-managed DMA rather than caches for their local storage. Unlike caches, which are demand based, DMAs are prefetch based. Cache misses initiate a fetch to memory when the processor requires the data, causing the processor to stall until the data can be fetched. Many techniques exist to reduce this stall time (such as prefetching, early load scheduling, speculative execution, and so on), or to do useful work during the stall (such as multithreading), but these techniques are generally costly in terms of chip area and power. Prefetch-based DMA avoids these costs.

Because many video-centric applications have predictable data access patterns, the DMA engines in the mediaDSP architecture support a streaming or double-buffering approach that lets an asynchronous DMA process act in parallel with a data-processing engine. Streaming is preferred over double buffering where possible, because it lowers the latency of obtaining a first result and reduces the working set size for data storage.

Whereas image data is inherently 2D, video data can be thought of as being 3D,

where the index of a frame within a sequence forms the third dimension. We've modeled data sets as having as many as six dimensions, where the additional dimensions might represent the component index (for example, red, green, and blue) and the chunking of blocks into smaller subblocks. The DMA engine most commonly used in the mediaDSP supports the transfer of 5D data. An additional feature of mediaDSP DMA is address wrapping at array boundaries, permitting multidimensional extensions of ring buffers.

Deterministic execution time is fundamentally important for real-time applications such as video processing. A programmer must be able to determine how long the execution of a task or group of tasks will take, to guarantee that the execution time repeatedly falls within some predetermined range. The mediaDSP technology addresses this concern by avoiding the use of caches and leveraging software-managed DMAs.

### Address spaces

From an external host's viewpoint, all of the mediaDSP's shared memory, TOE local

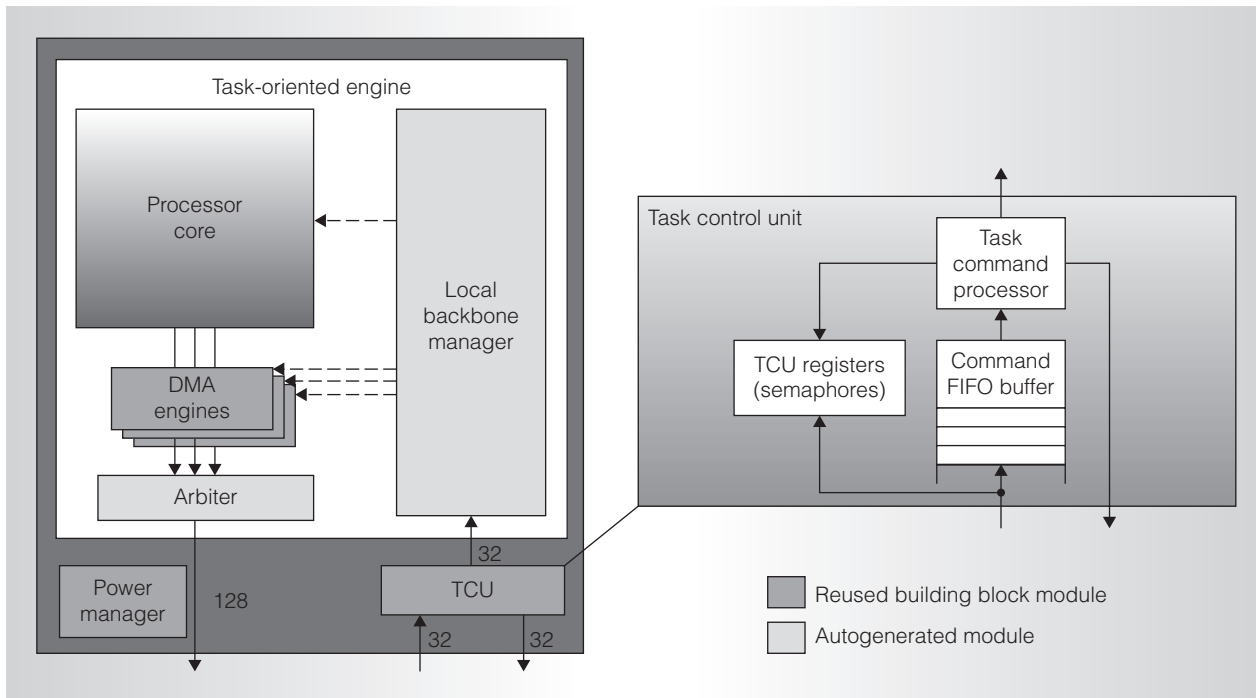


Figure 3. Template for a task-oriented engine. Each TOE reuses the DMA engine, task control unit (TCU), and power management that is part of the mediaDSP platform design approach, and uses autogenerated bus decoders and arbiters. The TCU contains a command FIFO buffer, a task command processor (TCP), and local semaphore registers.

memories, and registers are mapped into a single linear mediaDSP address space. The size of this address space is configurable, depending on the combination of TOEs that are instantiated. Typically, this address space is mapped into the register space of the SoC that integrates the mediaDSP processor. The platform also has a provision for instantiating multiple mediaDSP processors into a single chip's address map by wiring a unique ID to a mediaDSP instance.

A control engine in the mediaDSP architecture "sees" a 2-Gbyte address map that lets it access the mediaDSP address space, the external DRAM, and any chip-level registers in the SoC.

A DMA engine residing in a TOE moves data between the mediaDSP address space and the address space of any memory that is local to that TOE. A TOE's DMA engine typically moves data between shared memory and the TOE's local memory, but we can achieve peer-to-peer DMA from one TOE to another if an application requires it.

The external memory DMA engine (at the top level of the mediaDSP) moves

data between the mediaDSP address space and the memory space of the SoC that integrates the mediaDSP processor.

#### Template for a task-oriented engine

Figure 3 shows a TOE's basic architecture. We applied two principles when defining the TOE template: reuse building block elements, and use autogenerated modules.

The presence of one or more identical DMA engines, as well as the TCU and power-management blocks, demonstrates the reuse of building block elements. Reuse of these blocks goes beyond the obvious benefits of easing hardware development; it also fosters uniformity among the various TOE application programming interfaces. Thus, heterogeneous TOEs can have similar APIs, which in turn provides a unified development platform that eases the burden on the programmer.

Proprietary tools generate the local backbone manager and arbiter automatically. These modules are a simple subset of the communication fabric used at the top level of the mediaDSP architecture, so the

generator tools used for the communication fabric also generate these two modules.

This platform-based design approach increases engineering productivity, because reuse and automated code generation let the designer focus on the TOE processor core, where the main value is added. One design team provides the reusable infrastructure modules to the other teams developing the application-specific TOEs.

The processor core architecture isn't burdened by logic to handle interprocessor synchronization and messaging because the TCU handles this. Likewise, the DMA engines provide the I/O functions required by the core.

Additionally, because of the TOEs' uniformity, a common power-management design can be instanced into all TOEs whereby the power manager turns off clocks to each of the TOE subblocks when they're idle.

With this template-based approach, the processor core implementation can evolve to satisfy changing requirements as a product line matures, without requiring changes to that TOE's API. For example, a first-generation product might have a highly programmable core that can be replaced by a fixed-function core in the second generation as a cost savings.

#### Task control unit

A TCU is coupled with every TOE. The TCU maintains a queue of tasks that are executed by its associated TOE. It synchronizes these tasks with other TCUs or control engines in the system via hardware semaphore registers.

Figure 3 shows a TCU's three main components: the command FIFO buffer, TCU registers, and the task command processor (TCP).

The command FIFO buffer has 256 entries, each capable of holding a register write command destined for the TOE. These commands set up and trigger a TOE task, and can thus be viewed as *task descriptors*. A task descriptor has similar semantics to a remote procedure call in that the caller identifies which procedure (task) is to be run in the TOE and what parameters are to be passed to that procedure (task). The source of these register writes is generally a control engine.

The FIFO buffer decouples the TOE's activity from the control engine. Thus, the control engine can run ahead of the TOE, continuing to set up tasks in other TOEs while the first TOE tasks wait. The command FIFO buffer also holds commands intended for the TCP.

The TCU registers include semaphore resources that let the attached TOE synchronize the execution of its tasks with the other processors. The semaphores are accessible by any processor, including the local TCP. Each TCU has eight semaphores that the programmer can flexibly assign. Each semaphore supports the following actions through register read/write commands: set, clear, test-and-set, test-and-clear, read, and write. The control engine similarly has a set of 16 general-purpose semaphores with the same set of capabilities.

The TCP offloads low-level tracking of the TOE from the main control engine. Because the TCP operates from commands that have been queued in a FIFO buffer, it's decoupled from the current thread running on the control engine and can be viewed as running a lightweight thread on the control engine's behalf. As a result, the TCP executes the following operations:

- Pass task descriptors from the command FIFO buffer to the TOE.
- Monitor the local TOE's status.
- Wait for (and set or clear) a local semaphore.
- Set or clear a remote semaphore.

The TCP can also be programmed to perform general data movement and polling operations.

The mediaDSP's task-management method allows quick task synchronization and enables high levels of concurrency among a large set of TOEs.

#### Task programming example

Programmers develop four types of computer program code in the mediaDSP platform:

- system-management and communication code,
- task-management code,
- TCP commands, and
- task-execution code.

```

queue_toe_task()
{
1  set_dma_pars(this_toe_id, rdma_id, rdma_pars);           // Set parameters for one Read DMA task
2  set_dma_pars(this_toe_id, wdma_id, wdma_pars);         // Set parameters for one Write DMA task
3  set_toe_pars(this_toe_id, toe_pars);                   // Set parameters for one TOE task

4  tcp_wait_data_avail (this_toe_id, data_avail_sem_id);   // TCP READUNTIL test-and-clear succeeds
5  tcp_wait_buf_notbusy(this_toe_id, buf_busy_sem_id);     // TCP READUNTIL test-and-set succeeds

6  trig_dma( toe_id, rdma_id, rdma_addr );                // Trigger Read DMA
7  trig_toe( toe_id, toe_cmd );                            // Trigger TOE operation
8  trig_dma( toe_id, wdma_id, wdma_addr );                // Trigger Write DMA

9  tcp_wait_until_wdma_clean ( toe_id );                  // READUNTIL Write DMA data has been retired

10 tcp_set_data_avail (dnstrm_toe_id, dnstrm_data_avail_sem_id ); // Set data avail for downstream task
11 tcp_clr_buf_busy (upstrm_toe_id, upstrm_buf_busy_sem_id ); // Clear buf busy for upstream task
}

```

Figure 4. Task-control programming. Example C code snippet for dispatching a task to a TOE and managing synchronization with upstream and downstream tasks.

The first two types of code, which are written in standard C language, execute on the control engines. System-management code performs all hardware initialization, interrupt handling, cooperative multitasking, and communication with a host processor, if the system uses one. Task-management code generates task descriptors and issues them along with TCP commands into various TCU queues. TCP commands perform low-latency status monitoring and task control via hardware semaphores.

Tasks that are carried out on programmable processors require task-execution code. When one of the control engines is the executing processor, the task-execution code is written in C. All programmable TOEs used in mediaDSP are programmed in assembly language, and task-management code manages the TOE's code store.

Figure 4 shows a snippet of C code that illustrates how a programmer manages an example task. The variable `this_toe_id` represents the TOE performing the example task. The task involves a read DMA operation for fetching input data, a data-processing task, and a write DMA operation for storing result data.

Although the code in Figure 4 is executed on a control engine, each of the 11 lines of code results in one or more writes to the TOE's TCU queue. Thus, the procedure `queue_toe_task()` is nonblocking and would typically complete in a few dozen CPU cycles, whereas the actual task computation and data transfer typically require hundreds or thousands of cycles for completion.

Lines 1-3 in Figure 4 generate write operations destined for hardware registers within the TOE's read DMA engine, processor core, and write DMA engine, respectively. The write addresses and data pass through the TCU queue, and stall only if the TCP is executing a `readuntil` command associated with a prior task. These three lines of code typically result in approximately eight register writes.

Each of the two calls in lines 4 and 5 cause a single TCP command of type `readuntil` to be written into the TCU queue. The first `readuntil` command will cause the TCP to poll the local semaphore associated with `data_avail_sem_id` repeatedly until the test-and-clear operation succeeds. The upstream task causes the semaphore to become set after the associated

upstream data, which is an input to the current task, has been written to shared memory. If the semaphore had been set prior to the TCP's execution of this command, the `readuntil` command would read the semaphore only once, and the TCP would immediately proceed to the next item in its queue. In a similar vein, the second `readuntil` command causes the TCP to poll repeatedly, reading a different local semaphore that has been allocated for signaling the state of the buffer to which the current task will write.

Lines 6-8 each cause a single register write to be queued in the TCU. These three writes trigger the read DMA, the TOE processor core, and the write DMA, respectively.

Line 9 causes a single TCP `readuntil` command to be written to the TCU queue. The TCP will poll the write DMA `clean` status bit repeatedly until it succeeds. Because typical task execution requires hundreds of cycles, this command will likely cause the TCP to stall while the TOE is busy executing the current task. Once all of the data to be written by the current write DMA task has been retired to its destination, the `readuntil` will succeed and will let the TCP continue processing the commands in its queue.

The call in line 10 causes a TCP `write` command to be queued. This command sets a semaphore for a downstream task, indicating that the current task has completed writing its output data. The call in line 11 queues a TCP `write` command that clears an upstream semaphore, indicating that the current task has completed reading its input data. At this point, the current task is complete.

As this example illustrates, `mediaDSP` doesn't require an extensive suite of proprietary tools. The tools used for `mediaDSP` software development include a C compiler for the control engine, assemblers for programmable TOEs, a set of macros for TCP commands, and an API for task management.

### Profiling

The ability to profile specific applications while they are running on the physical chip is an important consideration when developing real-time software on a multiprocessing system. The `mediaDSP`'s hardware-supported

profiling capabilities let programmers visualize and analyze the level of parallelism being achieved. This feedback is essential for tuning applications to attain maximum parallelism. Utilities gather trace data from the chip and convert it into waveform files for visualization and analysis.

### Communication fabric

The communication fabric consists of one or more backbone managers that mediate access from several bus master devices to several bus slave devices. This includes separate request and response backbones. A backbone is a set of broadcast data signals and associated control signals. The request backbone is broadcast to all slaves, and the response backbone is broadcast to all masters. All masters arbitrate to gain access to the request backbone, and all slaves arbitrate to gain access to the response backbone. The backbone manager is a pipelined design, so a master must support multiple outstanding transactions (in a split-transaction fashion) to achieve 100 percent throughput.

If the communication fabric has more than one backbone manager, a full crossbar bridge connects them. Backbone managers can be 32 or 128 bits. The bridge manages resizing of transactions that go between backbone managers of different sizes.

Each backbone has a throughput of one transaction per clock cycle. Thus, a 128-bit backbone has a peak capacity of 6.4 Gbytes per second, and a 32-bit backbone has a peak capacity of 1.6 GBps. A master accessing shared memory that is connected to the same backbone manager experiences a latency of about nine clock cycles, whereas access across a bridge to a remote shared memory has a latency of about 25 cycles.

Analyzing the backbone bandwidth load produced by the tasks is part of mapping a video application to a specific `mediaDSP` embodiment. Based on the frame-rate conversion algorithm's traffic analysis, the BCM35421's `mediaDSP` architecture includes four 128-bit backbones and one 32-bit backbone.

Proprietary tools automatically generate both the chip-level backbone managers and the local backbone managers residing in the TOEs. These tools take a text-based specification file and generate validated Verilog

code for synthesis. Automating this effort helps avoid tedious register-transfer-level verification work, creates RTL code that is correct by construction, and allows rapid changes to the interconnect topology.

### Crunch engine

The crunch engine is a digital signal processing (DSP) engine with a 128-bit data path and an instruction set tailored to video and image processing. Its arithmetic instructions include both SIMD and reduction types of operations. Every mediaDSP embodiment to date has used this engine.

We maximize performance per area by avoiding complex pipeline control logic. Because the hardware pipeline is exposed, the programmer can schedule instructions for maximum throughput. We use fixed instruction and data storage instead of caches, thereby avoiding unpredictable penalties for cache misses. Consequently, the programmer can straightforwardly estimate the execution time by counting the number of instructions to be run.

Figure 5 shows a block diagram of the crunch engine, which has a dedicated 8-Kbyte instruction RAM holding as many as 2,048 instructions. The control engine loads tasks into the instruction RAM using DMA transfers or writes them directly. Typically, the control engine loads a library of tasks into the crunch engine at application initialization time and so doesn't need to load code dynamically during application execution.

The crunch engine contains two local data RAMs that the programmer uses to pass data between other processors and the crunch engine. It uses one RAM (local RAM A, 4 Kbytes) to pass data from an external processor to the crunch engine and hold intermediate results. It uses the other RAM (local RAM C, 64 bytes) to pass data from the crunch engine to an external processor. Two DMA units in the crunch engine transfer data between these RAMs and the main mediaDSP communication fabric.

Generally, crunch engine data processing instructions fetch operands from local RAM A and write results to either local RAM A or local RAM C. The crunch engine includes nine sets of registers for operand addressing, with each describing the geometry of a data

structure contained in local RAM. Seven of these registers are for general-purpose use, the eighth describes FIFO inputs, and the ninth describes FIFO outputs.

The crunch engine has a 32-bit instruction format that specifies three operands per instruction, whereby each instruction can fetch operands from two separate data structures and write the results back to a third data structure. Each of the three operand address registers can be auto-incremented independently, and each of the three operand data structures can have a different geometry. The address generators can update the address registers in parallel with the crunch engine's data operations, thereby maximizing the processor's efficiency. Because most machine cycles are spent processing data, few cycles in this instance are spent manipulating address information.

Each RAM includes a section that can be configured as a FIFO buffer to achieve streaming, whereby the streaming decouples memory access from the data processing, thus avoiding processor stalls due to memory access latency. The address generators and DMA engines interact to accomplish flow control for the input and output streams.

Prior SIMD processors use a register file to hold local data. Such a register file is accessed on fixed boundaries where the granularity is related to the machine's word size, requiring the processor to spend cycles on data-alignment and data-packing instructions to format the data. Conversely, the crunch engine includes a novel feature whereby operands can be fetched on any byte boundary and results can be written on any byte boundary.

The crunch engine processor core doesn't have load or store operations; therefore, we can view the DMA engines as executing separate threads that accomplish load and store functionality. Because crunch engine programming requires partitioning a program into multiple threads (such as read memory access, write memory access, and computational operations), the threads run concurrently and in concert. Early graphics chip designs and decoupled architectures influenced some of these concepts.<sup>6,7</sup>

The crunch engine supports zero-overhead looping, with nesting capabilities, and predicated versions of many instructions.

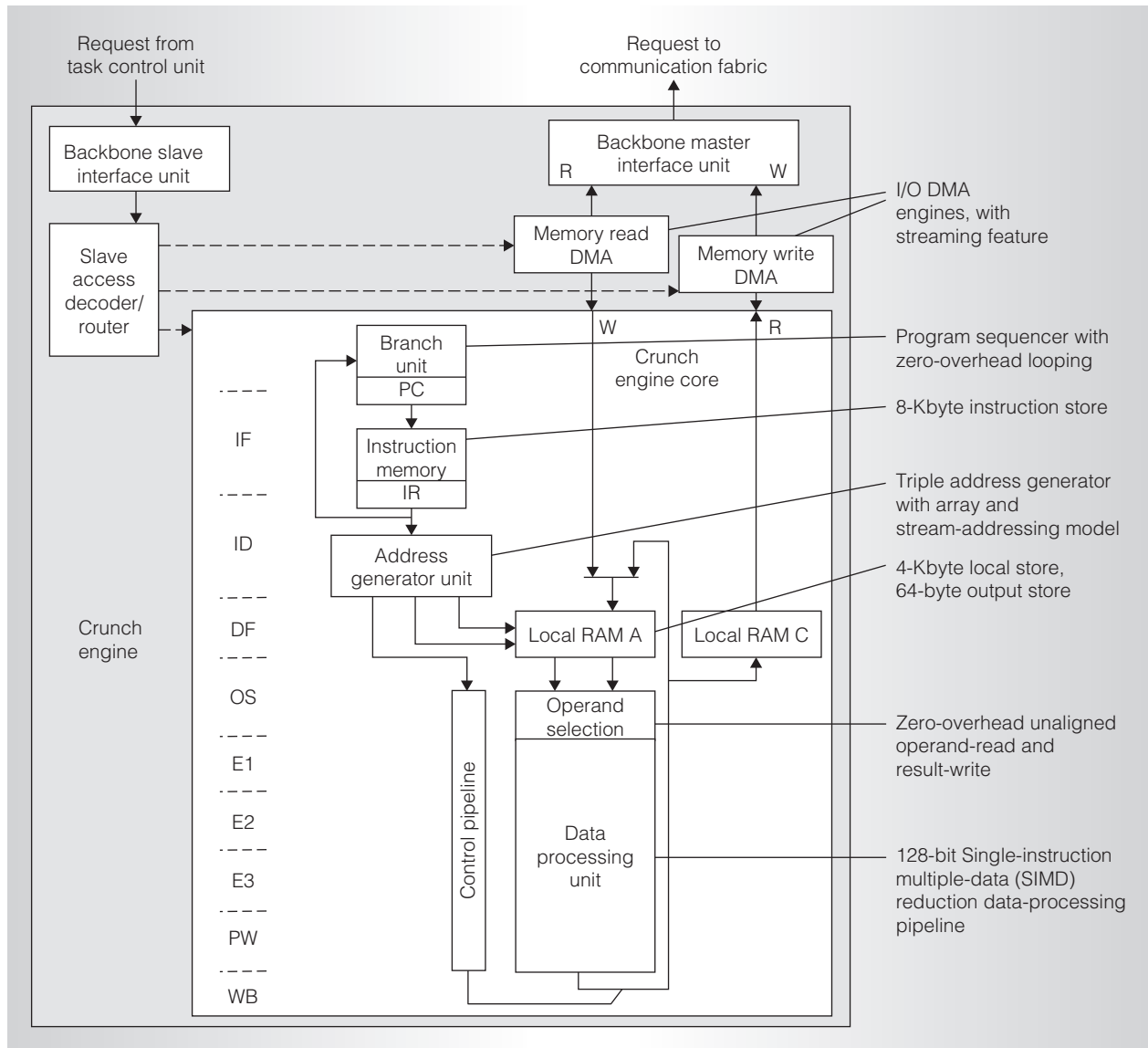


Figure 5. Block diagram of the crunch engine. This engine has a nine-stage pipeline with simple control and keeps complexity and area low by not using caches or supporting interrupts. The DMA engines can operate in concert with the processor core to achieve a streaming computation model.

With predicated versions, the processor can maintain a high degree of data-level parallelism, even when each of the SIMD data lanes requires conditional processing.

The crunch engine's instruction set consists of more than 300 different opcodes, and some of its instructions represent a large number of primitive arithmetic operations. All arithmetic opcodes use fixed-point or integer data types, because video processing doesn't require floating-point operations. Many arithmetic and move

operations that reduce result precision have the rounding mode selectable from a set of six modes—truncate, truncate the magnitude, round away from zero, round toward zero, round up, and round down.

SIMD instructions are either 4-way, 8-way, or 16-way—for 32-bit, 16-bit, and 8-bit data types, respectively. (An in-depth instruction set discussion is beyond this article's scope.)

The "Related Work in Video Processing" sidebar discusses some other processors that use programmable approaches.

---

## Related Work in Video Processing

The computer architecture community has developed various parallel processing techniques to address video-processing problems. Many general-purpose processor families adopted single-instruction, multiple-data (SIMD) instruction set architecture (ISA) extensions to accelerate video-processing tasks, and multiple instruction issue (superscalar) methods to increase instructions per clock.<sup>1-4</sup> These devices generally aimed at the desktop or server computing space, and weren't cost effective for consumer electronics products. They also used on-demand caches, which target optimal long-term statistical performance and are less appropriate for systems requiring guaranteed real-time performance.

Custom programmable media processors also used SIMD as well as multiple instruction issue methods using very long instruction word (VLIW) architectures.<sup>5-7</sup> These architectures rely on sophisticated compiler technology to detect instruction-level parallelism and create a static schedule of parallel instructions. Unlike general-purpose processors, these architectures are appropriate for the lower-cost embedded computing space, but are still basically uniprocessors and don't readily scale to high levels of task parallelism.

As applications became more demanding and uniprocessor clock rates stopped increasing, architects developed multicore architectures to increase computational capability. Multicore architectures are coarsely classified as *homogeneous* or *heterogeneous*.

Intel's 80-tile prototype chip<sup>8</sup> and Larrabee architecture<sup>9</sup> are homogeneous multicores, whereas the Cell processor<sup>10</sup> is heterogeneous. The first embodiment of Cell uses two types of cores: a power processor element and a synergistic processor element. The mediaDSP processor and Cell both use software-managed direct memory access engines to manage data locality, rather than relying on caches. Execution times are subsequently more predictable, which helps in real-time programming. Cell implements shared-memory coherence mechanisms in hardware, in contrast to mediaDSP's simpler approach of managing coherence in software.

Furthermore, the job of managing coherence is simplified by mediaDSP's task-based programming model, which uses hardware semaphores to synchronize among many task-oriented engines. The semaphore interlocks guarantee that the data produced by one task-oriented engine is available in shared memory before the consumer attempts to read it.

Because mediaDSP doesn't require a sophisticated operating system, it doesn't need virtual memory, and likewise avoids the complex logic that Cell uses to handle distributed virtual memory. Also, unlike Larrabee and Cell, mediaDSP saves die area by capitalizing on the fact that video-processing applications don't require floating-point precision.

Texas Instruments' Da Vinci products (<http://www.ti.com/corp/docs/landing/davinci/index.html>) combine reduced-instruction-set-computing processors with digital signal processors in a single die, and are aimed at video codec applications, which can be an order of magnitude less computationally complex than frame-rate conversion.

The mediaDSP platform is closest in spirit to other templates for building specialized embedded media processors<sup>11</sup> (also see <http://www.arc.com/subsystems/vraptor.html>), but is distinguished by the

scale of parallelism that is embodied in a commercial device such as Broadcom's BCM35421. Another difference between mediaDSP and other published multicore architectures is the programmer community's size. The technology's main goal is to build cost-effective custom embodiments, each for a certain limited class of applications, and isn't intended to be an open general-purpose platform.

Finally, it should be noted that the Broadcom mediaDSP technology presented in this article isn't related to other published work that uses the same name.<sup>12,13</sup>

---

## References

1. R.B. Lee, "Subword Parallelism with MAX-2," *IEEE Micro*, vol. 16, no. 4, July/Aug. 1996, pp. 51-59.
2. A. Peleg and U. Weisner, "MMX Technology Extension to the Intel Architecture," *IEEE Micro*, vol. 16, no. 4, July/Aug. 1996, pp. 42-50.
3. M. Tremblay et al., "VIS Speeds New Media Processing," *IEEE Micro*, vol. 16, no. 4, July/Aug. 1996, pp. 10-20.
4. K. Diefendorff et al., "AltiVec Extension to PowerPC Accelerates Media Processing," *IEEE Micro*, vol. 20, no. 2, Mar./Apr. 2000, pp. 85-95.
5. J. van de Waerdet et al., "The TM3270 Media-Processor," *Proc. 38th Ann. IEEE/ACM Int'l Symp. Microarchitecture (Micro 05)*, IEEE CS Press, 2005, pp. 331-342.
6. C. Basoglu et al., "The Equator MAP-CA DSP: An End-to-End Broadband Signal Processor VLIW," *IEEE Trans. Circuits and Systems for Video Technology*, vol. 12, no. 8, Aug. 2002, pp. 646-659.
7. J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufman, 2003.
8. S. Vangal et al., "An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS," *IEEE J. Solid-State Circuits*, vol. 43, no. 1, Jan. 2008, pp. 29-41.
9. L. Seiler et al., "Larrabee: A Many-Core x86 Architecture for Visual Computing," *ACM Trans. Graphics*, vol. 27, no. 3, Aug. 2008, pp. 18:1-18:15.
10. M. Gschwind et al., "Synergistic Processing in Cell's Multicore Architecture," *IEEE Micro*, vol. 26, no. 2, Mar./Apr. 2006, pp. 10-24.
11. M.J. Rutten et al., "A Heterogeneous Multiprocessor Architecture for Flexible Media Processing," *IEEE Design & Test of Computers*, vol. 19, no. 4, July/Aug. 2002, pp. 39-50.
12. D. Wu et al., "MediaDSP: An Application Specific Heterogeneous Multiprocessor SoC," *Proc. Swedish System-on-Chip Conf. (SSoCC)*, 2006, <http://www.da.isy.liu.se/pubs/divwu/divwu-ssocc2006.pdf>.
13. Z. Teng et al., "Physical Design of Dual-Core System-On-Chip," *Proc. IEEE Int'l Workshop VLSI Design & Video Technology*, IEEE Press, 2005, pp. 36-39.

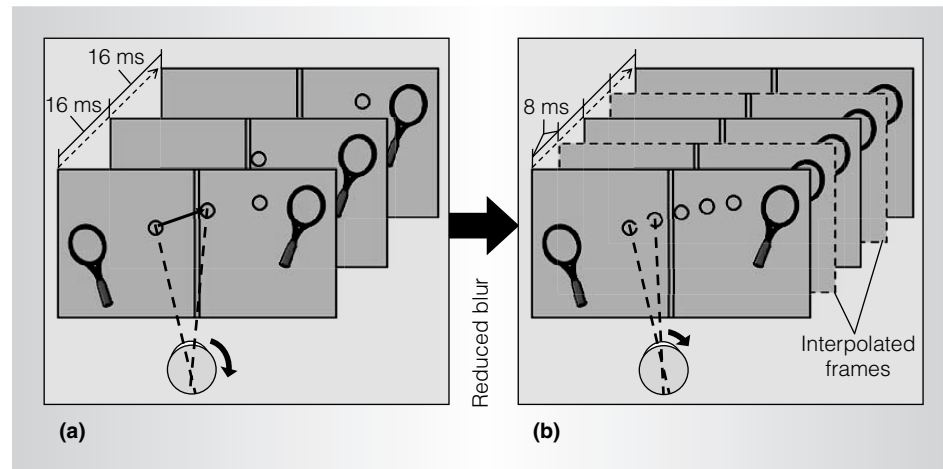


Figure 6. Perception of motion blur at 60 frames per second (a) and at 120 fps (b).

### Application to frame-rate conversion

The number of large-screen LCD TV sets sold has increased every year since the arrival of HDTV content. In fact, LCDs overtook CRTs in unit shipments in 2007, becoming the leading display technology.<sup>8</sup>

Advances in LCD display technology have yielded dramatic improvements in the viewer's visual experience. Examples include wider viewing angles, higher contrast ratios, wider color gamuts, and higher resolution. One of the most significant remaining challenges for LCD display technology is the phenomenon of fast-moving objects viewed on an LCD display appearing blurry when compared with conventional CRT TVs. This phenomenon is often referred to as *LCD motion blur*.

The cause of LCD motion blur is often misunderstood because it results from an interesting interaction between the human visual system and the LCD panel. The blur isn't on the panel; it's actually on the viewer's retina. If the viewer's eye didn't move, there would be no LCD motion blur, as Figure 6 illustrates.

An LCD panel is a hold-type display, which means that each frame of video is presented on the screen with a constant brightness throughout the entire frame time. This is in contrast to CRTs, which are raster-scanned, impulsive-type displays that illuminate phosphors on the inner surface of the screen for a short time at high brightness. An instantaneous snapshot of a CRT display would reveal an image that has a bright

horizontal area where the electron beam is currently scanning, and dimmer areas where the brightness has decayed since the time of the scan. This stroboscopic approach is one reason why CRT displays appear to flicker at low refresh rates. An instantaneous snapshot of an LCD display, on the other hand, would show a normal-looking image.

When motion occurs in a video sequence, the human viewer's eyes reflexively track the moving object over a series of video frames. Invariably, the viewer's eyes will move while each picture from a motion sequence is held on the LCD panel, causing each image displayed on the panel to be smeared across the viewer's retina so it appears blurry. Displaying a motion picture sequence at a frame rate double that of the received video reduces the degree of this perceived LCD motion blur. Although the frame rate for conventional video signals is limited to 60 Hz, recent generations of mainstream LCD panels support image refresh rates up to 120 Hz. Because these newer systems hold each displayed image for half the time (compared with conventional video), they cut the amount of smearing on the retina roughly in half. For this scheme to work, the sequence of pictures at double rate must resemble the sequence that would have been obtained had the scene been originally captured at the higher rate.

We create the picture sequence at double rate using an advanced video algorithm for motion-compensated frame-rate conversion

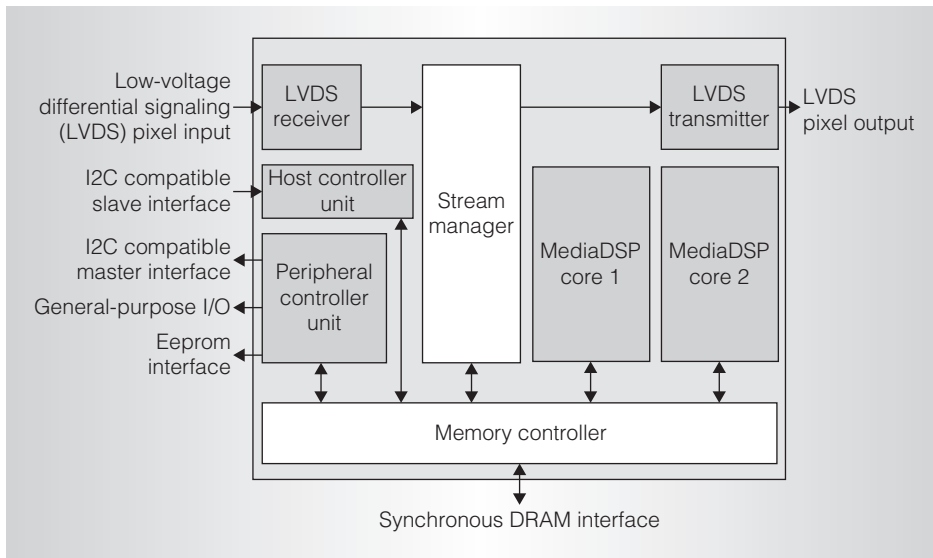


Figure 7. Block diagram of the Broadcom BCM35421 chip. The chip contains two mediaDSP cores, low-voltage differential signaling (LVDS) interfaces—dual 5 + 1 for input, and quad 5 + 1 for output (75-MHz nominal clock), a stream manager to control capture and presentation of video frames, and an efficient memory controller supporting a single 32-bit-wide GDDR3 synchronous DRAM device with bandwidth up to 1,500 Mbps per pin.

(MC-FRC). This algorithm analyzes a sequence of frames to determine how each object in the scene is moving, and then creates the intermediate frame in which each object has moved half the distance. (More background on frame-rate conversion techniques is available in other publications.<sup>9-12</sup>)

Whereas motion blur is a problem for LCD panels, *film judder* is a problem for all types of display technologies, and its reduction has become an important feature for high-end HDTV sets. Film judder is perceived as motion jerkiness when 24-Hz film-based content is formatted for viewing on a 60-Hz display. Film-based content is often displayed with a 3:2 cadence to adapt from 24 Hz to 60 Hz. This means that one frame is displayed for three frame times; the next frame is displayed for two frame times, and so forth. The same MC-FRC techniques used to deblur can serve to eliminate judder by generating intermediate frames and balancing out the uneven cadence. This technique is known as *dejuddering*.

In a mathematical sense, MC-FRC presents an ill-posed problem, meaning that no single optimal solution exists. There is no standard for FRC, and there is no

published algorithm that achieves optimal picture quality across all input sequences. Measurement of picture quality is inherently subjective, and different TV manufacturers use various metrics for evaluating FRC solutions. Given this paradigm, a programmable solution would let TV manufacturers customize FRC to their preferences. Furthermore, full-HD input rates (1,920 × 1,080p at 60 Hz), would require a large amount of computational resources. Given these two requirements, a programmable multicore solution is attractive.

The Broadcom BCM35421 chip aims to address this need, leveraging mediaDSP technology to implement the sophisticated video-processing algorithms for MC-FRC. In addition to the video processor, the chip includes low-voltage differential signaling (LVDS) interfaces and other support modules, as Figure 7 shows.

The BCM35421 processor includes two identical instances of mediaDSPs, with each one operating on one half of the screen (left and right). Figure 8 shows the specific configuration of one of these mediaDSPs.

Each mediaDSP processor includes six unique TOEs that execute tasks specific to the frame-rate conversion algorithm,

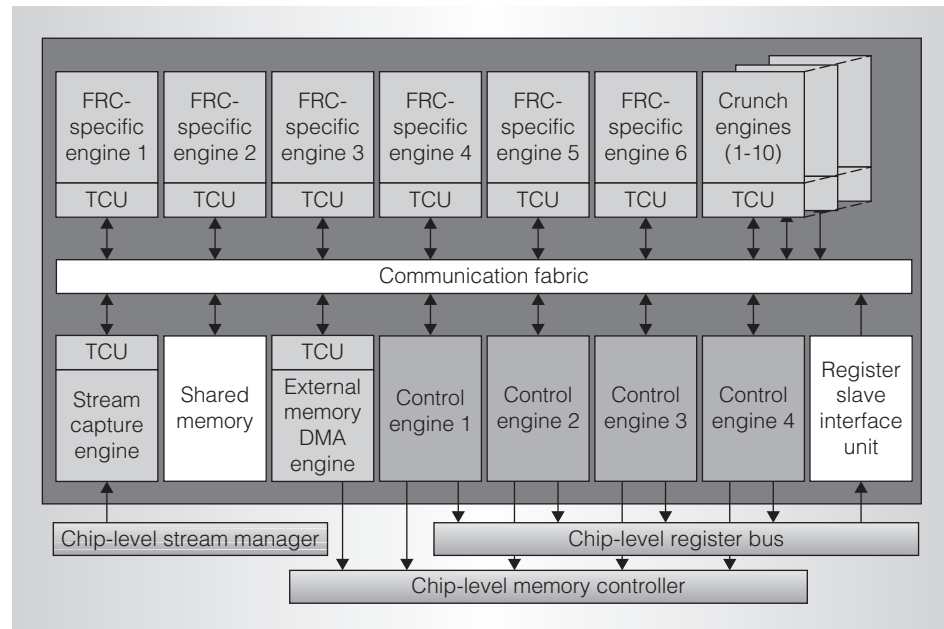


Figure 8. Block diagram of mediaDSP in Broadcom BCM35421 chip. The BCM35421 mediaDSP has four RISC-based control engines, six task-oriented engines specifically for subtasks of the frame-rate conversion application, and 10 128-bit DSP crunch engines.

10 instances of the crunch engine, four control engines, and a stream capture engine that facilitates capturing a stream of video pixels into the mediaDSP's shared memory. Broadcom manufactures the BCM35421 chip using a 65-nm CMOS process, and the mediaDSP portion runs at 400 MHz. Other physical characteristics of a single mediaDSP core include:

- typical power of 1.3 watts at 1V,
- 106 million transistors,
- 22 independent processing elements, and
- peak integer performance of 819 giga-operations per second programmable and 1,052 Gops total.

With the two cores taken together, the entire chip has greater than two teraops of peak integer performance.

The Broadcom BCM35421 chip is fully functional and commercially available. We implemented numerous algorithmic improvements in the video processor chip after receipt of first silicon by exploiting the mediaDSP's programmability and re-configurability. In many cases, implementing proposed video-processing algorithms

in real-time firmware running on the BCM35421 mediaDSP has facilitated algorithm exploration. We can evaluate an algorithmic change by viewing the results on a product prototype in real time, without having to run lengthy simulations. Due to postsilicon algorithm improvements, the BCM35421 picture quality exceeds that of the leading competing solution.

Although we designed the BCM35421 specifically for the frame-rate conversion application, this chip could help solve other video-processing problems. One potential application involves the use of a motion-compensated super-resolution algorithm<sup>13</sup> to increase effective picture resolution.

The BCM35421 represents the second product line to use the mediaDSP platform, with the first supporting MPEG audio/visual encoder chips. Although these two products perform different types of video processing, with computational requirements that differ by an order of magnitude, this wide range of application characteristics demonstrates the mediaDSP platform's scalability and extensibility. Future work on the mediaDSP platform will include firmware development tool chain enhancements and hardware cost reductions. MICRO

## Acknowledgments

We thank the exceptionally talented team members at Broadcom for their efforts in developing mediaDSP technology and the BCM35421 chip. The team includes algorithm, hardware and firmware architects, firmware developers, RTL developers, verification engineers, and physical design engineers. We could not wish for a finer group of colleagues. Special acknowledgments go to Kathy Burns, Samir Hulyalkar, and Daniel Doswald for their lead roles in the work described in this article. Broadcom, the pulse logo, Connecting everything, mediaDSP, and the Connecting everything logo are among the trademarks of Broadcom Corporation and/or its affiliates in the United States, certain other countries, and/or the EU. Any other trademarks or trade names mentioned are the property of their respective owners.

## References

1. C.A.R. Hoare, "Communicating Sequential Processes," *Comm. ACM*, vol. 21, no. 8, Aug 1978, pp. 666-677.
2. I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, Addison Wesley, 1995.
3. T. Stefanov et al., "System Design Using Kahn Process Networks: The Compaan/Laura Approach," *Proc. Design Automation and Test in Europe Conf. and Exposition (DATE 04)*, IEEE CS Press, 2004, pp. 340-345.
4. A. Davere et al., *A Platform-based Design Flow for Kahn Process Networks*, tech. report UCB/EECS-2006-30, Univ. of Calif., Berkeley, 2006.
5. G. Kahn, "The Semantics of a Simple Language for Parallel Programming," *Information Processing 74: Proc. IFIP Congress 74*, North-Holland, 1974, pp. 471-475.
6. "ET6300 Graphics and Multimedia Engine Data Book," Tseng Labs, 1997.
7. A.R. Pleszkun et al., "Features of the Structured Memory Access (SMA) Architecture," *Proc. 3rd IEEE Computer Society Int'l Conf.*, IEEE CS Press, 1986, pp. 259-263.
8. "Display Search *Quarterly Global TV Shipment and Forecast Report*," Aug. 22, 2008, <http://www.displaysearch.com>.
9. H. Pan, X.F. Feng, and S. Daly, "LCD Motion Blur Modeling and Analysis," *Proc. Int'l Conf. Image Processing (ICIP 05)*, IEEE Press, 2005, pp. II-21-24.
10. G.A. Thomas, "Television Motion Measurement for DATV and Other Applications," *BBC Research Dept. Report*, BBC RD 1987/11, 1987.
11. A. Pelagotti and G. de Haan, "High Quality Picture Rate Up-Conversion for Video on TV and PC," *Proc. Philips Conf. Digital Signal Processing*, Philips Research Laboratories, 1999, paper 4.1.
12. N. Beucher et al., "Motion Compensated Frame-Rate Conversion Using a Specialized Instruction Set Processor," *Proc. IEEE Workshop Signal Processing Systems Design and Implementation (SIPS 06)*, IEEE Press, 2006, p. 130-135.
13. Y. Jia et al., "Video Processing in HDTV Receivers for Recovery of Missing Picture Information: De-interlacing, Frame-Rate Conversion, and Super-Resolution," *Information Display*, vol. 23, no. 11, Nov. 2007.

**Richard Selvaggi** is an associate technical director at Broadcom, working in the digital television group. His research interests include multiprocessor architecture and programming models, ISA design, configurable processors, platform-based design, and system modeling. Selvaggi has an MS in electrical and computer engineering from Carnegie Mellon University. He is a member of the IEEE and the IEEE Computer Society.

**Larry Pearlstein** is a technical director of video algorithm development at Broadcom. His research interests include scalable video computing architectures and the development of practical algorithms for intelligent video processing. Pearlstein has a PhD in electrical engineering from Princeton University.

Direct questions and comments about this article to Richard Selvaggi or Larry Pearlstein, Broadcom Corp., 770 Township Line Rd., Ste. 200, Yardley, PA 19067; [rjs@broadcom.com](mailto:rjs@broadcom.com) or [lpearlst@broadcom.com](mailto:lpearlst@broadcom.com).

For more information on this or any other computing topic, please visit our Digital Library at <http://computer.org/csdl>.