A Heterogeneous Multiprocessor Architecture for Flexible Media Processing

Martijn J. Rutten, Jos T.J. van Eijndhoven, Egbert G.T. Jaspers, Pieter van der Wolf, Om Prakash Gangwal, and Adwin Timmer
Philips Research Laboratories

Evert-Jan D. Pol
Philips Semiconductors

Eclipse is a scalable architecture template for designing data-dependent stream-processing subsystems of media-processing SoCs. It combines application configuration flexibility with the efficiency of function-specific coprocessors that concurrently execute the tasks of one or more applications.

NEW MEDIA APPLICATIONS such as high-definition digital television, set-top boxes with time-shift functionality, 3D games, video conferencing, and MPEG-4 interactivity have generated a demand for increasingly flexible consumer electronics products. These products are evolving into multifunctional devices that combine a set of media applications. The required set of applications and their format vary per product, per country, and over time as standards evolve.

Managing the complexity, design cost, and time to market of these programmable, resource-constrained appliances requires a generic, scalable media-processing platform that is deployable in a wide range of products. Several vendors are entering the market with platforms that address this need to some extent.[1,2] Philips Electronics has developed a platform concept embodied in the Viper system on a chip.[1] Such SoCs typically consist of a heterogeneous mix of fully programmable processors (such as the MIPS, ARM, and TriMedia processors) and coarse-grained application-specific subsystems (such as MPEG decoders and video filters) optimized for high performance with minimal power consumption and silicon area. Currently, these subsystems are dedicated to a single application, and the hardware cannot be reused for other applications within the same domain. Therefore, although the generic platform with its interconnect structure can be reused over various generations, a change in application requirements necessitates a redesign of the application-specific subsystems. These subsystems incur a large part of the total design effort and silicon cost.

Eclipse is an architecture template for the design of versatile media-processing SoC subsystems. A *template* is a generic architecture blueprint from which designers can create different hardware instances by fixing design parameters such as number and type of coprocessors, bus widths, and memory sizes. *Instances* of the Eclipse template are heterogeneous subsystems containing a mix of programmable and hardwired functions. The

0740-7475/02/$17.00 © 2002 IEEE

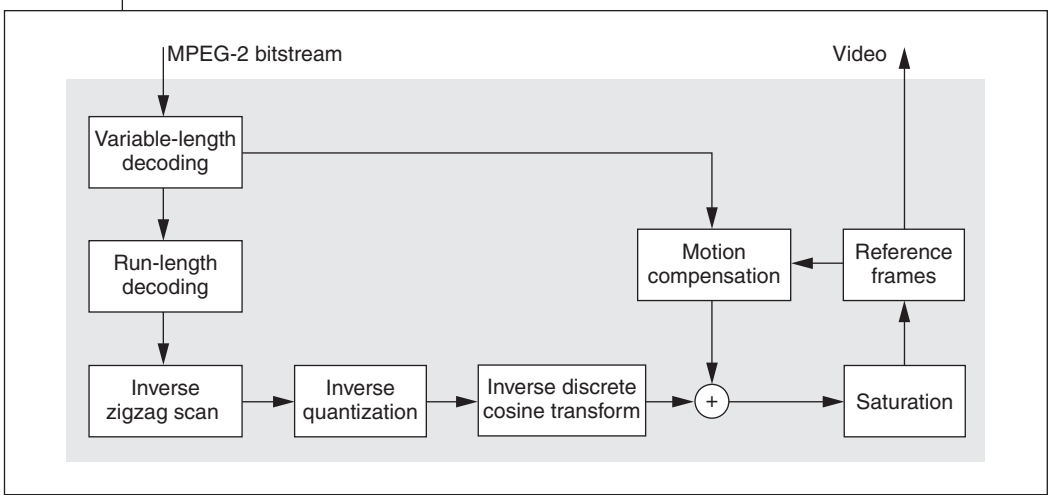


Figure 1. Kahn process network for an MPEG-2 decoder.

targeted media applications combine real-time and dynamic behavior. Eclipse instances combine the performance density of function-specific hardwired modules, or *coprocessors*, with the flexibility of one or more programmable cores. When Eclipse is programmed for a particular application, it links these coprocessors and cores into a network that resembles the application structure.

Design aspects of media-processing architectures

In the consumer-electronics domain, media-processing applications execute on resource-constrained systems. The performance, flexibility, and cost-effectiveness of such systems are highly interrelated. Architectural tradeoffs influence these parameters and underlie the Eclipse template's design choices.

Parallelism and flexibility

In consumer electronics, design constraints such as cost, power consumption, performance, and flexibility are weighted differently than in the PC market. Consumer media processors require an order-of-magnitude lower cost combined with significantly higher performance. The key challenge is to design a media-processing architecture that provides high performance with low power consumption and silicon cost.

Media-processing applications typically exhibit parallelism at various levels of granularity. For example, a time-shift recorder function consists of encoding and decoding functions that may execute in parallel. Moreover, within the decoding function, many medium-grained tasks can execute in parallel—for example, the discrete-cosine transform (DCT) and quantization tasks. Within the DCT task, many operations can execute in parallel. A designer can make the parallelism explicit by specifying a media-processing application as a Kahn process network—a set of concurrently executing tasks that exchange information solely through unidirectional data streams.[3,4] A directed graph with a node for each task and an edge for each data stream represents the application's structure. Figure 1 shows an example graph for MPEG-2 video decoding. The Kahn model defines the Eclipse architecture's model of computation.

The data streams in Kahn networks are buffered. Each buffer is a FIFO buffer, with one producer and one or more consumers. Reading from a stream with insufficient data available causes a consuming task to stall. Kahn formally proved that such a system has a well-defined, unique behavior. In particular, the functional behavior—observed as the sequence of data items that traverse the edges—is independent of the order in which the tasks execute.

One of the Kahn model's nice features is its inherent building-block nature. After defining a set of basic functions as tasks, the designer can configure a multitude of applications by instantiating tasks and connecting them in a graph structure. Describing an application in these generic tasks is followed by a mapping phase in which the designer decides which architecture modules execute which tasks. For this mapping to make sense, the Kahn tasks' granularity must match the modules' granularity. The resulting solution is highly flexible if this match is complemented by a generic infrastructure that

allows runtime instantiation of task graphs.

For example, complex media-processing SoCs exploit the performance density of sophisticated application-specific subsystems to implement critical parts of targeted media applications. At the same time, SoCs give the system flexibility by embedding programmable cores. The subsystems execute concurrently to exploit task-level parallelism at a coarse granularity. A heterogeneous mix of hardware and software is essential for a cost-effective yet flexible architecture. The subsystems implement the SoC's functions, and the SoC infrastructure handles data communication between subsystems. The infrastructure's routing flexibility plays an important role in the SoC's application configuration flexibility.

For competitive media-processing architectures in consumer products, designers maximize performance by exploiting parallelism wherever possible, and they minimize cost by introducing flexibility only where necessary. The coarse granularity of hardwired SoC subsystems renders them inflexible. For example, an MEPG-2 decoder block cannot be reused for an MPEG-2 encoding application, even though an MPEG-2 encoder contains a large portion of an MPEG-2 decoder. For configuration flexibility, the subsystem must support the mapping of medium-grained application tasks on internal function modules of corresponding granularity. Therefore, Eclipse introduces a programmable infrastructure that supports concurrent execution of medium-grained functions in a heterogeneous mix of software and reusable coprocessors.

Timing coupling

In Eclipse, coprocessors communicate data through communication buffers that implement the theoretical FIFO buffers of Kahn networks. These buffers' size determines the extent to which the producer and consumer are coupled in execution timing. The factors that determine timing coupling are the application's processing regularity and the admissible amount of stalling behavior that sacrifices parallelism. For instance, regular tasks, such as those in linear video filtering where the worst-case communication requirements equal the average case, allow tight coupling with minimal buffering.[5] Irregular tasks demand looser coupling to allow individual task progress, leading to larger buffer requirements. A typical irregular, data-dependent task is variable-length decoding in MPEG decoding, in which the quantity of input and output data can vary wildly from stream to stream or even within a single stream. A less obvious example is the DCT function; the function itself is regular, but the number of DCT-coded blocks to be processed varies per MPEG frame. Eclipse targets the video-encoding and -decoding application domain, which is highly data dependent and thus highly irregular. In practice, the ratio of worst-case versus average load can be as high as a factor of 10. Consequently, we designed Eclipse as a relatively loosely coupled system.

Communication requires both data transport and synchronization. *Data transport* is the movement of data into and out of communication buffers. Producers and consumers exchange information about the amount of produced or consumed data in the buffer available for consumption or production. We refer to this information exchange as *synchronization*. Because of the FIFO buffering, the producer and consumer need not mutually synchronize individual read and write actions on the stream. Thus, the designer can choose synchronization granularity independently. For instance, each individual data access can be synchronized at the data transport's granularity. Alternatively, synchronization can be related to the logical unit of input and output data on which a task operates—for example, the granularity of a picture in an MPEG-decoding task.

Decreasing the application functions' granularity (for example, from an MPEG-decoder function to a DCT function) enhances parallelism and reuse. However, as the number of streams passing through the communication network increases, both communication buffering and bandwidth requirements increase. Eclipse reduces communication buffer requirements by changing the synchronization granularity to a finer level (for example, from picture level to macroblock level in MPEG). The resulting small communication buffers can remain on chip, allowing deployment of a dedicated communication network to cope with the high bandwidth requirements.

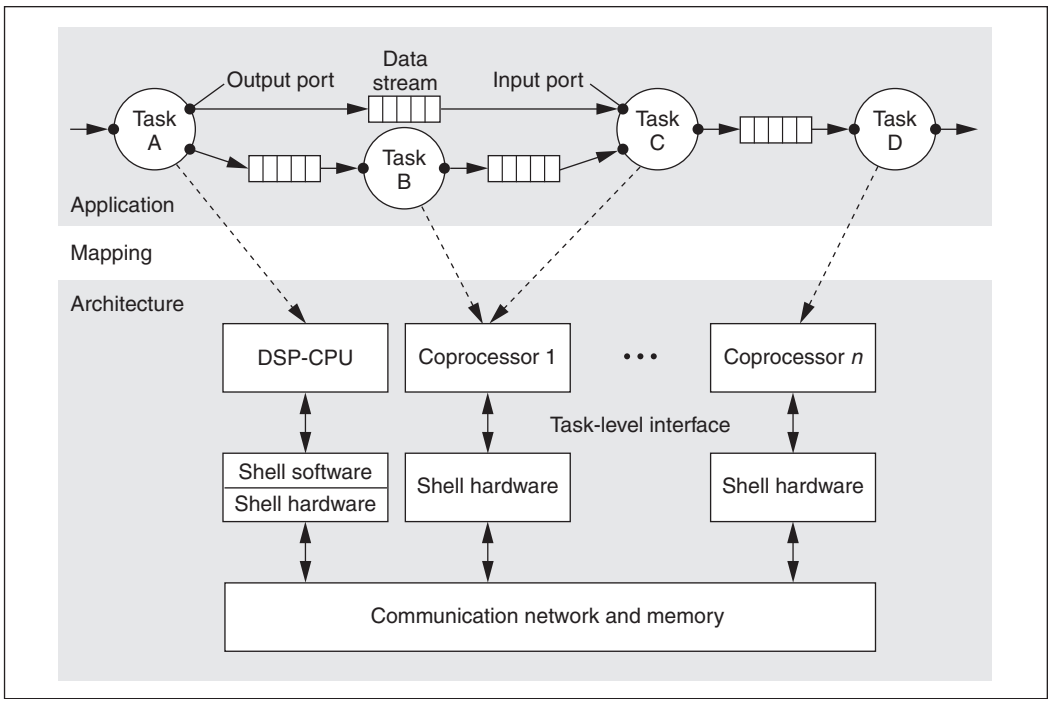


Figure 2. Application mapping to the Eclipse architecture template.

Architecture template

Eclipse provides an architecture template at the subsystem level. Although subsystem-level templates are relatively unknown, they are essential in supporting reuse of design effort by providing a set of parameterized rules for subsystem composition. Template parameters include memory size, number and type of coprocessors, and others.

The Eclipse template exploits application-level parallelism by concurrently executing medium-grained functions in function-specific coprocessors and/or software executing on a media processor. Functions eligible for coprocessor implementation are those common to media applications, such as the DCT transform used by decoders and encoders for JPEG, MPEG, and digital video applications. Typically, the functions eligible for software implementation are specific to one application only—such as still-texture decoding in MPEG-4—or are likely to change as standards evolve. At runtime, the Eclipse infrastructure links these medium-grained functions into a Kahn-style application graph, using on-chip communication and data buffering.

Figure 2 shows how application tasks and stream buffers map to the Eclipse coprocessors and FIFO buffers allocated in on-chip memory. Unlike fully hardwired SoC subsystems, the Eclipse infrastructure is programmable and provides the flexibility to configure a given Eclipse instance for different application graphs.

Eclipse coprocessors exploit the performance density of dedicated hardware function units and are only weakly programmable. The Eclipse template ensures scalability by avoiding complex centralized modules that control large parts of the architecture. Thus, all Eclipse coprocessors execute in parallel and autonomously, without requiring CPU support for task scheduling or synchronization.

New silicon technologies allow efficient coprocessors with sufficient computation speed for time-shared use. Such multitasking coprocessors are essential for configuring a range of applications and reapplying the same coprocessors at different places in an application task graph. Each coprocessor can execute multiple tasks from a single Kahn network or from multiple, possibly different, networks in a time-shared fashion. Thus, application complexity is not restricted to the number of coprocessors in the architecture. Moreover, the programmable media processor can perform

part of the application functionality when an application requires functionality beyond the implemented set of coprocessors.

The strong flexibility requirements led us to design the Eclipse infrastructure with a centralized memory module in which communication buffers can be allocated at runtime. This flexible connection of medium-grained functions requires significant communication bandwidth from the system. For media processing, the streaming nature of the application functions generates a high locality of reference—that is, consecutive references to the memory address of neighboring data. Eclipse instances exploit this characteristic to provide high data throughput (Gbytes per second) through deployment of a shared wide (for example, 128 bits) bus combined with communication buffers in a wide on-chip memory. A dedicated hardware implementation in the coprocessor shell supports the high synchronization rate.

Coprocessor shell

The infrastructure that transports and synchronizes data between coprocessors can vary among different Eclipse instances to match communication bandwidth requirements. The coprocessor shell shown in Figure 2 facilitates reuse of a coprocessor design among different instances with different communication network characteristics by introducing a stable interface that separates the coprocessor design from the infrastructure, and vice versa.[6] Moreover, the shell simplifies coprocessor design by absorbing many system-level issues, such as multitasking, stream synchronization, and data transport. Thus, coprocessor designers can concentrate on application functionality.

The architecture template deploys distributed shells, such that each shell can be close to the coprocessor it serves. The coprocessor-shell interface defines a uniform set of primitives with coprocessor-specific parameters, such as the data path's width. We designed the shell's internal architecture as a parameterized template to facilitate reuse in an Eclipse instance. A product engineer derives shell instances with coprocessor-specific parameter settings from this generic template. Examples of such parameters are the data width of the read and write interface between the coprocessor and shell and the size of data caches in the shell. Although this article focuses on the coprocessor shell, the described concepts also apply to the shell of the media processor (the DSP-CPU in Figure 2). The media processor shell can implement part of its functionality in software, as Figure 2 shows, to increase flexibility and reduce hardware cost.

Task-level interface

Each coprocessor interacts with its shell through five generic interface primitives.[6] Eclipse implements these primitives in hardware through a master-slave handshake with corresponding argument and result passing or in software through a function-call mechanism with equivalent parameters and results. Figure 2 represents the primitives as the task-level interface between coprocessors and their shells.

For multitasking, the coprocessor issues the following primitive:

```
int GetTask( int *task_info )
```

The coprocessor calls this primitive whenever it allows a task switch to another task mapped on the coprocessor. The return value is the identifier of the next task (task_id) to execute on the coprocessor. The task_info value provides parameter values for the selected task's function—for example, one bit to select whether a forward or inverse DCT is to be performed.

The primitives for accessing data in the stream buffer are

```
void Read(int task_id, int
  port_id, int offset,
     int n_bytes, Bytes
       *bytevector )
```

for reading a number of bytes from the data stream connected to an input port, and

```
void Write(int task_id, int
  port_id, int offset,
     int n_bytes, const Bytes
       *bytevector )
```

for writing a number of bytes to a data stream

```
while(true) {
  // Perform a single processing step
  task_id = GetTask(&task_info);

  // Is there data/room for reading/writing?
  blocked = !GetSpace(task_id, IN, INSIZE)
    || !GetSpace(task_id, OUT, OUTSIZE);
  if (blocked) continue; // No useful work to do

  Read(task_id, IN, 0, INSIZE, &in_data);
  PutSpace(task_id, IN, INSIZE); // Commit room

  Compute(task_info, in_data, &out_data);

  Write(task_id OUT, 0, OUTSIZE, out_data);
  PutSpace(task_id, OUT, OUTSIZE); // Commit data
}
```

Figure 3. Example of a top-level coprocessor control loop.

connected to an output port.

The Eclipse design philosophy advocates the separation of data I/O and synchronization, which is instrumental in designing cost-effective hardware tasks—for example, it allows the reduction of buffer requirements. The primitive for synchronizing access to data in the stream buffer is

```
bool GetSpace(int task_id, int
  port_id, int n_bytes)
```

to inquire whether *n* bytes of valid data are available in the stream buffer for reading or *n* bytes of room are available for writing into the stream buffer. After a **Read** or a **Write**, the **PutSpace** primitive commits the number of consumed or produced bytes:

```
void PutSpace(int task_id, int
  port_id, int n_bytes)
```

The n_bytes argument of the **GetSpace** and **PutSpace** calls allows the coprocessor to synchronize streams at a granularity and rate different from those of the individual **Read** and **Write** calls.

All task ports map to one physical coprocessor-shell interface that handles **Read**, **Write**, **GetSpace/PutSpace**, and **GetTask** requests in parallel. To discriminate among different streams, the coprocessor passes an identifier of the active task port to the shell through the port_id argument of the primitives. The shell internally combines the task_id and port_id arguments into an identifier of the associated stream. Although **GetSpace** and **PutSpace** do not distinguish between input and output ports, a **GetSpace** on an input port inquires about available data for reading, whereas a **GetSpace** on an output port inquires about available room for writing. Likewise, a **PutSpace** call on an input port commits empty room available for writing, whereas a **PutSpace** call on an output port commits valid data written in the stream buffer.

Computation architecture

In the design of Eclipse coprocessors, Kahn application models are gradually refined into task-level code that uses the task-level interface.[6] This code subsequently forms the starting point for the low-level coprocessor design. Eclipse coprocessors explicitly decide on the times at which they can switch the running task, thereby avoiding the hardware costs required for state saving at arbitrary times. Multitasking on Eclipse coprocessors is a shared responsibility of the shell and the coprocessor. The shell handles task scheduling, and the coprocessor provides task switch points and saves and restores the task state (if any) upon a task switch.

The coprocessor can continue to the point where it has minimal or no state. At that point, the coprocessor asks its shell which task it should perform next by calling the **GetTask** primitive. We refer to the intervals between **GetTask** inquiries as *processing steps*. The coprocessor executes an infinite loop of processing steps. The simplified code in Figure 3 shows such a top-level coprocessor control loop as an example of multitasking-coprocessor design using the five Eclipse primitives. This example illustrates the separation of coprocessor functionality—implemented by the Compute function—from coprocessor control to handle multitasking, synchronization, and data communication.

Before the task starts a **Read** or **Write**, it uses **GetSpace** to test whether sufficient data is available for reading and sufficient room is available for writing. After a **Read** or **Write**, the amount of generated data or room is committed to the shell via **PutSpace** calls.

Data access synchronization

From a coprocessor task port's viewpoint, a data stream looks like an infinite data tape with a current access point, as Figure 4 shows. With the **GetSpace** call, the coprocessor asks the shell permission for access to a certain data space ahead of the current access point. Data space signifies available data for reading from an input data stream, or available room for writing data to an output stream. If the shell grants permission, the coprocessor can perform **Read** or **Write** actions inside this requested space using variable-length data (through the n_bytes argument), and at random access positions (through the offset argument). When sufficient data or room is not available, the shell denies permission by returning false on the **GetSpace** call.

The coprocessor is responsible for functionally correct behavior when using the interface primitives. For example, the coprocessor must adhere to denied **GetSpace** requests and must not attempt to read or write data outside the window of granted space. Thus, when **GetSpace** fails, the coprocessor task cannot proceed, and either the coprocessor can switch tasks or the task can keep trying to proceed by repeatedly issuing **GetSpace** requests.

After one or more **GetSpace** calls—and optionally several **Read**/**Write** actions—the coprocessor can decide it is finished with processing the data and issue a **PutSpace** call. This call advances the access point a specified number of bytes ahead, in a size constrained by the previously granted space.

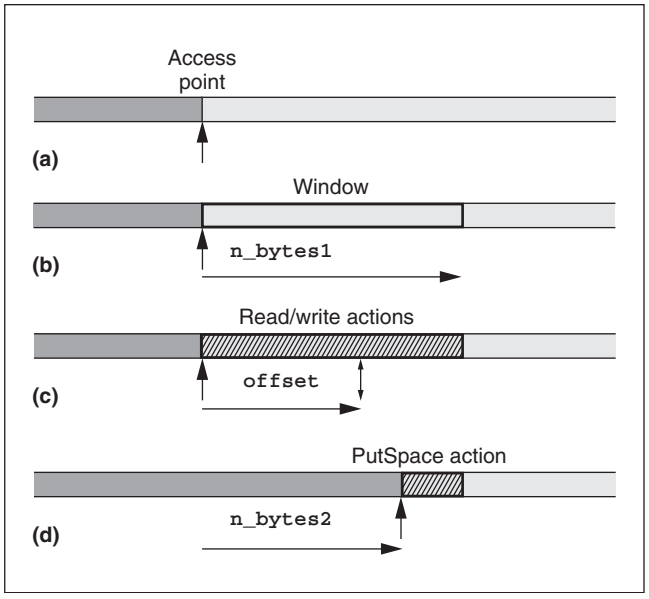


Figure 4. Synchronization and data I/O through a single port: initial "data tape" situation with current access point (a); GetSpace action provides window on requested space (b); Read/Write actions on contents (c); PutSpace action moves access point ahead (d).

Task switching

Eclipse coprocessors operate on a logical unit of data—such as an 8 × 8 block of DCT coefficients—encapsulated in a data packet. The coprocessors can have different packet consumption and creation patterns. If consumption at the input is synchronized with packet creation at the output ports, the coprocessor can switch tasks at moments when the data state is void. Typically, coprocessor state is minimal after the processing of a complete packet. For instance, a DCT coprocessor is virtually stateless after processing a block of DCT coefficients. To avoid context switch overhead, Eclipse coprocessors usually process an integer number of packets in a single processing step.

However, at the start of a processing step, the coprocessors cannot always determine the required amount of space for completing the processing step. This is the case if the coprocessor has a data-dependent condition upon which it must read more data from a second input port. In such situations, the coprocessor must inquire for additional space during a processing step and may be unable to continue executing the current task. The coprocessor designer can decide to let the coprocessor wait for the space to arrive, effectively blocking the coprocessor. Alternatively, the coprocessor can call **GetTask**, giving the shell the opportunity to provide a new task. In the latter case, the coprocessor may have to save and restore state information upon switching tasks.

Generic infrastructure

Eclipse provides a generic infrastructure used by all coprocessors. The shells maintain the application graph structure and implement a large part of the generic functionality, including stream synchronization, data transport, task scheduling, and performance measurement support.

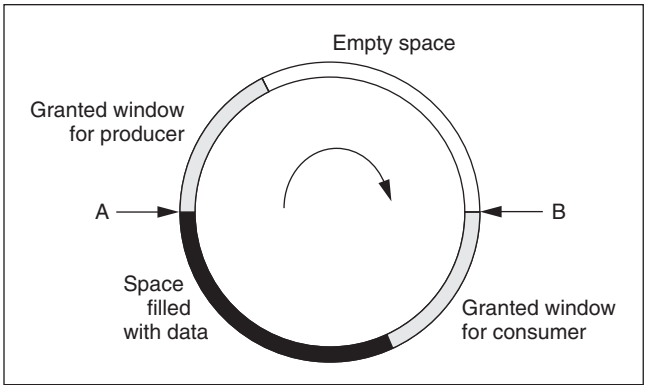


Figure 5. Basic stream mapped to a finite FIFO buffer.

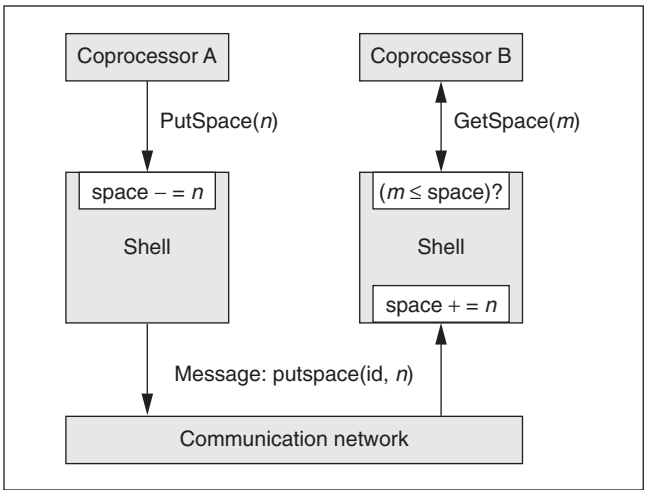


Figure 6. Updating local space values and sending "putspace" messages.

Stream synchronization

Communicating a stream of data requires a FIFO buffer with a finite and constant size preallocated in shared on-chip memory. The shell applies a cyclic addressing mechanism for proper FIFO behavior in the linear memory address range. For this cyclic addressing mechanism, the shell uses the buffer size and current access point, as maintained in the shell, and the n_bytes and offset arguments of the **`Read`**/**`Write`** requests.

Figure 5 depicts the fixed-size cyclic memory space used as a FIFO buffer. The rotation arrow in the center shows the direction in which **`GetSpace`** calls confirm the granted window for **`Read`**/**`Write`**, which is the same direction in which **`PutSpace`** calls move the access points ahead. The small arrows denote the current access points of tasks A and B. In this example, A is a producer and hence leaves proper data behind, whereas B is a consumer and leaves empty space (already consumed data) behind. The shaded region ahead of each access point represents the access window acquired through **`GetSpace`**.

Each shell locally contains the configuration data for streams incident to tasks mapped on its coprocessor and locally implements all the control logic necessary to properly handle this configuration data. The shells implement a local stream table, which contains a row of fields for each stream, or more precisely, for each access point. To handle the setup shown in Figure 5, each coprocessor shell of tasks A and B contains one row, holding the following fields:

- A space field containing a (maybe pessimistic) distance from its own access point to the other access point in this buffer. The space value corresponds to the amount of available data for reading from an input port or the available room for writing to an output port.
- A stream identifier denoting the remote shell with the task and port of the other access point in this buffer.

As Figure 6 shows, coprocessor B's shell can answer a **`GetSpace`** request immediately by comparing the requested size *m* with the locally stored space value. When coprocessor A's shell receives a **`PutSpace`** request, it locally decrements its space field with the indicated amount *n* and sends a "putspace" message to coprocessor B's shell. This remote shell holds the other access point and increments its space field upon receiving the "putspace" message.

Data transport

Coprocessors transport all media data to and from their shells through the **`Read`** and **`Write`** primitives. The shells subsequently access corresponding locations in the shared stream buffers in on-chip memory. The shells provide the **`Read`** and **`Write`** primitives to hide aspects such as the width of system data paths; data alignment in memory and cyclic buffer addressing; and data stream caching, including coherency and prefetching control. Shells incor-

porate separate read and write caches that play an important role in decoupling the coprocessor from the global communication network.

The **GetSpace/PutSpace** synchronization mechanism explicitly controls cache coherency transparently to the coprocessor. Using local **GetSpace** and **PutSpace** events for explicit cache coherency control results in a simpler and more efficient implementation than generic coherency mechanisms such as bus snooping. Apart from cache coherency, the shell also initiates stream prefetches upon local **GetSpace** and **Read** requests to reduce cache miss penalties.

Task scheduling

The task scheduler decides which task a coprocessor must execute and the times it must execute the task to attain proper application progress. The target granularity for processing steps in the Eclipse architecture ranges from 10 to 1,000 clock cycles. Typically, a processing step's duration is data dependent and can vary within this range. The number of processing steps needed to complete an application milestone (such as an MPEG frame), as well as the number of produced and consumed data items per processing step, may also be data dependent. Consequently, Eclipse must perform task scheduling at runtime to manage the highly data-dependent workloads cost-effectively. The task-switching rate is too high (10 to 100 kHz) for runtime scheduling in software. Therefore, Eclipse implements task scheduling and synchronization in dedicated hardware in the coprocessor shell. Moreover, task scheduling is distributed to contribute to the coprocessors' autonomy, thereby increasing scalability and cost-effectiveness. Thus, the task scheduler in each shell runs independently from task schedulers in other shells.

We chose a weighted round-robin scheme to select tasks because it can be efficiently implemented in the generic shell hardware. To improve the system's robustness, we endowed each task with its own weight, or *budget*—a guaranteed minimum number of cycles that a task may continuously execute, irrespective of the other tasks' resource requirements.[7] Budgets typically range from 1,000 to 10,000 clock cycles (10 to 100 processing steps). The tasks mapped to the coprocessor are configured in the shell's task table, which contains among other things the resource budget per task.

The task scheduler cannot determine in advance whether a task can complete a processing step. Therefore, the scheduler performs a best guess by considering the available data and room in the stream buffers, as well as previously denied data access.[7] This information is locally available in the shell. Task scheduling with the best-guess strategy is effective; it selects the right tasks in most cases and recovers with a limited penalty otherwise.

Performance measurement support

Eclipse supports performance measurement (profiling) through hardware in the shells. Measurements include buffer filling, coprocessor utilization, and data access latency. Product engineers can use these measurements to optimize application behavior, and a quality-of-service resource manager can use them to provide runtime control[8] in the final product. The shells accumulate measurement data in the stream and task tables. The shell collects measurement results per stream and per task rather than per coprocessor port and per coprocessor. Therefore, measurement results are available at the application level—that is, for each task and stream rather than for each coprocessor.

All shell tables are memory mapped and accessible to the main CPU via a control (peripheral interface) bus. Thus, the main CPU can collect measurement data at regular time intervals—for example, once per MPEG frame. However, accumulating measurements every cycle for a complete MPEG frame requires a significant amount of memory in the shell. To reduce the hardware costs of measurement support, a separate process in the shell takes measurement samples at regular intervals. Because the storage space for measurements is limited, the main CPU must balance the duration of these intervals with the duration of the total measurement.

An Eclipse instance

Figure 7 (next page) shows an Eclipse instance, an MPEG subsystem to be deployed

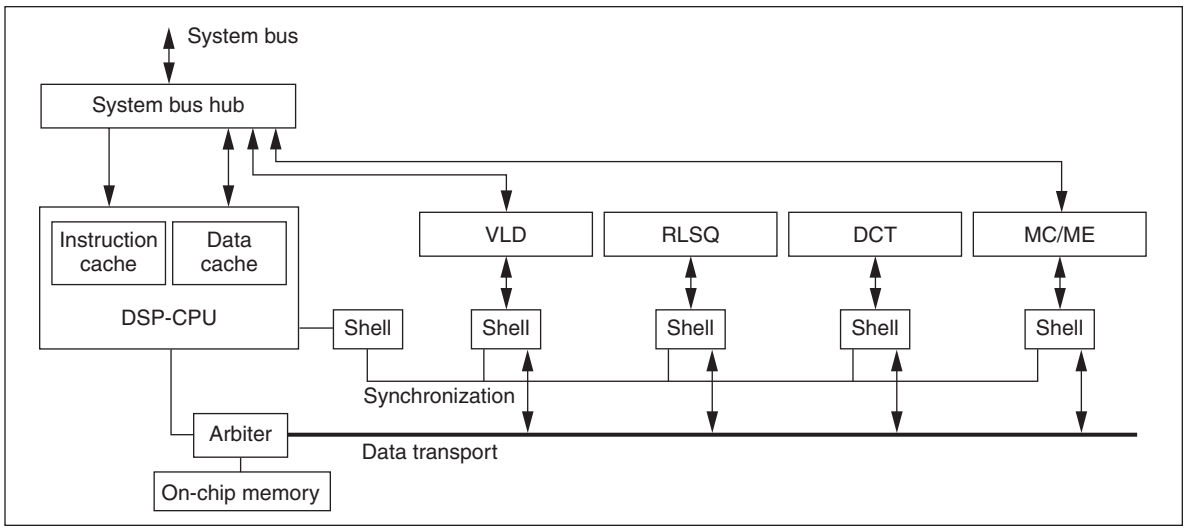


Figure 7. An Eclipse instance for video coding.

in SoC platforms for high-definition television functions, such as the Philips Nexperia digital video chips.[1] This subsystem decodes two high-definition (HD) MPEG-2 streams simultaneously or encodes standard definition (SD) MPEG-2 in parallel with decoding multiple SD MPEG-2 streams. Various combinations are possible, such as decoding one HD stream and decoding two SD streams in parallel or transcoding for time-shift functionality. The CPU configures these applications at runtime by programming the stream and task tables in the shells through the control bus (not shown in Figure 7).

Figure 7 shows the dedicated hardware units for MPEG processing. The coprocessors are multitasking and weakly programmable. Thus, the DCT coprocessor can time-share both the forward and inverse DCT functions of one or more MPEG encoding applications and the inverse DCT of one or more decoding applications. Equivalently, the run-length scan and quantization (RLSQ) coprocessor performs the run-length decoding, inverse scan, and inverse quantization of the MPEG-2 decoding network shown in Figure 1, as well as its encoding variant: quantization, zigzag scan, and run-length encoding. The motion compensation/motion estimation (MC/ME) coprocessor has a dedicated connection to the system bus to access MPEG reference frames in off-chip memory. Similarly, the variable-length decoding (VLD) coprocessor fetches the incoming compressed bit streams from off-chip memory. The DSP-CPU executes audio decoding, variable-length encoding, and demultiplexing in software.

The targeted applications allow the use of a single on-chip memory (SRAM) for communication buffering with a wide data path (128 bits) to provide the necessary bandwidth. For Eclipse instances demanding a higher communication bandwidth, the architect must balance the flexibility of allocating buffers with configurable sizes in a centralized memory against the scalability and performance of a distributed-memory implementation.

The Eclipse instance presented here is capable of decoding two HD MPEG streams, which requires a computational performance of roughly 36 GOPS on mostly 16-bit data items. Our initial estimates indicate that the instance takes less than 7 mm^2 of silicon area in 0.18-micron CMOS technology. This includes 1.7 mm^2 for a 32-Kbyte on-chip memory and 2.0 mm^2 for a programmable VLD coprocessor, but excludes the DSP-CPU. All coprocessors will be synthesized for operation at 150 MHz. The on-chip SRAM operates at 300 MHz to support separate read and write data buses, each running at 150 MHz. We estimate total power consumption at less than 240 mW for simultaneous decoding of two HD MPEG streams.

Simulation environment

Detailed analysis and hardware design of the

presented Eclipse instance is in progress. At present, Eclipse exists only as a simulator model, supporting application execution and tuning for particular instances. The full architecture is modeled in a flexible, cycle-accurate simulator.

The simulator is a design tool that supports a system design trajectory with gradual refinement of Kahn application models[9] into cycle-accurate Eclipse coprocessor models. Functionally correct simulation models provide quantitative feedback, letting us explore the Eclipse architecture's design space before diving into gate-level design through directed experiments. These experiments address caching strategies in the shell, bus latency and width, and so on. To this end, the simulation models of the Eclipse hardware are parameterized and can be configured at runtime. Moreover, the simulator collects measurement data such as communication buffer filling and coprocessor execution time. Accompanying graphical tools configure application graphs and visualize numerical measurement results.

In designing Eclipse, we strictly separated application functionality from the generic interconnect structure. The resulting uniform interface separates computation hardware, or coprocessors, from generic support for multitasking, synchronization, and data transport. This interface not only keeps coprocessor design simple but also facilitates coprocessor reuse over a set of media applications. The interface copes with irregular and unpredictable application loads by separating data transport from data access synchronization. The combined requirements of scalability and cost-effectiveness led us to a novel approach: distributed scheduling and distributed synchronization with high task-switching and synchronization rates, supported by a generic hardware implementation dedicated to each coprocessor.

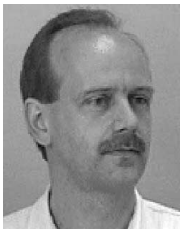
Another aspect of our design philosophy was to guide our design trajectory by simulation results. The simulator implemented for this purpose provided quantitative feedback on Eclipse's behavior early in the design phase. We have explored the Eclipse architecture in an instantiation for simultaneous MPEG-2 encoding and decoding of multiple streams at various resolutions. Currently, we are studying extensions of the MPEG-2 encoding and decoding subsystem presented here. These extensions will enable a single Eclipse subsystem to efficiently support a programmable mix of MPEG-2 and MPEG-4 encoding and decoding applications.[10] ■

References

1. S. Dutta, R. Jensen, and A. Rieckmann, "Viper: A Multiprocessor SOC for Advanced Set-Top Box and Digital TV Systems," *IEEE Design & Test of Computers*, vol. 16, no. 5, Sept.-Oct. 2001, pp. 21-31.
2. W. Lee and C. Basoglu, "MPEG-2 Decoder Implementation on MAP-CA Media Processor Using the C Language," *Proc. SPIE: Media Processors 2000*, vol. 3970, Int'l Soc. for Optical Eng., Bellingham, Wash., 2000, pp. 27-36.
3. G. Kahn, "The Semantics of a Simple Language for Parallel Programming," *Proc. Information Processing 74*, North-Holland, Amsterdam, 1974, pp. 471-475.
4. G. Kahn and D.B. MacQueen, "Coroutines and Networks of Parallel Programming," *Proc. Information Processing 77*, North-Holland, Amsterdam, 1977, pp. 993-998.
5. E.G.T. Jaspers and P.H.N. de With, "Architecture of Embedded Video Processing in a Multimedia Chip-Set," *Proc. IEEE Int'l Conf. Image Processing* (ICIP 99), vol. 2, IEEE Press, Piscataway, N.J., 1999, pp. 787-791.
6. M.J. Rutten, J.T.J. van Eijndhoven, and E.J.D. Pol, "Design of Multitasking Coprocessor Control for Eclipse," *Proc. 10th Int'l Symp. Hardware/Software Codesign* (CODES 02), ACM Press, New York, 2002, pp. 139-144.
7. M.J. Rutten, J.T.J. van Eijndhoven, and E.J.D. Pol, "Robust Media Processing in a Flexible and Cost-Effective Network of Multitasking Coprocessors," *Proc. 14th Euromicro Conf. Real-Time Systems*, IEEE CS Press, Los Alamitos, Calif., 2002.
8. R.J. Bril et al., "Multimedia QoS in Consumer Terminals," *Proc. IEEE Workshop on Signal Processing Systems* (SIPS 01), IEEE Press, Piscataway, N.J., 2001, pp. 332-344.
9. E.A. de Kock et al., "YAPI: Application Modeling for Signal Processing Systems," *Proc. 37th Design Automation Conf.* (DAC 2000), ACM Press, New York, 2000, pp. 402-405.
10. E.B. van der Tol and E.G.T. Jaspers, "Mapping of MPEG-4 Decoding on a Flexible Architecture Plat-

form," *Proc. Media Processors 2002*, vol. 4674, Int'l Soc. for Optical Eng., Bellingham, Wash., 2002, pp. 1-13.

Martijn J. Rutten is a research scientist at Philips Research Laboratories, Eindhoven, the Netherlands. His research interests include embedded-system architecture with an emphasis on media-processing applications. Rutten has an MSc in electrical engineering from Eindhoven University of Technology and is pursuing a PhD in computer science at the University of Amsterdam.



Jos T.J. van Eijndhoven is a principal research scientist at Philips Research Laboratories. His research interests include the architectural design of programmable multimedia hardware and the associated mapping of media-processing applications. Eijndhoven has a PhD in electrical engineering from Eindhoven University of Technology.

Egbert G.T. Jaspers is a senior research scientist at Philips Research Laboratories. His research interests include video-processing algorithms for digital television applications and their implementation on embedded processors. Jaspers has an MSc in electrical engineering from Eindhoven University of Technology. He is a member of the IEEE.

Pieter van der Wolf is a principal research scientist at Philips Research Laboratories. His research interests include system-level design methodologies, embedded-system architectures, and embedded processors. Van der Wolf has a PhD in electrical engineering from Delft University of Technology, the Netherlands. He is a member of the IEEE.

Om Prakash Gangwal is a research scientist at Philips Research Laboratories. His research interests include system-level design and embedded-system architectures. Gangwal has a BE in electronics and communication engineering from Malvia Regional Engineering College, Jaipur, India, and an MTech. in VLSI design tools and technology from the Indian Institute of Technology, Delhi. He is a member of the IEEE.

Adwin Timmer is an independent technology consultant in Eindhoven. His research interests include embedded-system architectures, multiprocessor designs, hardware-software codesign, and Java server architectures. He previously worked at Philips Research Laboratories, where he contributed to the work described in this article. Timmer has a PhD in electrical engineering from Eindhoven University of Technology.

Evert-Jan D. Pol is a senior system architect in the Embedded Processors Department of Philips Semiconductors, Eindhoven. His work includes the architecture of subsystems and their embedded programmable cores. He has a PhD in computer science from the University of Leiden, the Netherlands.

■ Direct questions and comments about this article to Martijn J. Rutten, Philips Research Laboratories, Prof. Holstlaan 4, 5656 AA Eindhoven, Netherlands; martijn.rutten@philips.com.