Pseudorandom verification and emulation of an MPEG-2 transport demultiplexor

by C. A. Reed
D. J. Thygesen

This paper describes two complementary approaches to performance verification for an MPEG-2 transport demultiplexor. The performance of such devices is difficult to verify during the design phase because of the many independent bus interfaces to which they may be connected and the numerous operating configurations that may be required. To address these problems, we have devised both a pseudorandom verification "environment," employing "controlled random" simulation, and a hardware-emulation platform based on field-programmable gate arrays (FPGAs). Actual hardware verification has shown the effectiveness of using these two methods together, and the overall approach can be applied to other design programs.

Introduction

Traditionally designs are verified using a suite of handwritten test cases, yet this approach is work-intensive and often limited by the resources and time available. The IBM MPEG-2 transport demultiplexor design presents a number of difficulties for traditional verification owing to its many independent bus interfaces and operating configurations. The impact of simultaneous interface requests is compounded by the asynchronous relationship among many of the bus interfaces. Considering the difficulties presented, hand-generation of encompassing tests is an intimidating task. Two complementing yet controlled verification methods help to address these problems, the first being a pseudorandom verification environment which employs random yet controlled simulation. The design is further verified in an FPGA-based emulation system. Emulation permits transport streams to be run with audio and video programs in real time to ensure operation over periods of time longer than those that can be feasibly simulated. The emulation system provides a pre-hardware vehicle for advanced software design and is an important independent check on the pseudorandom simulation methodology.

Over the past decade, verification teams around the industry have employed pseudorandom verification to address difficult challenges, specifically in processor design simulation. Significant effort has been driven by the IBM Israel Haifa Research Laboratory to create and maintain a model-based pseudorandom processor verification tool. Named Genesys, the tool has been used for numerous processor projects, across both X86 and PowerPC* architectures, and even supporting multiprocessor systems.

0018-8646/99/$5.00 © 1999 IBM

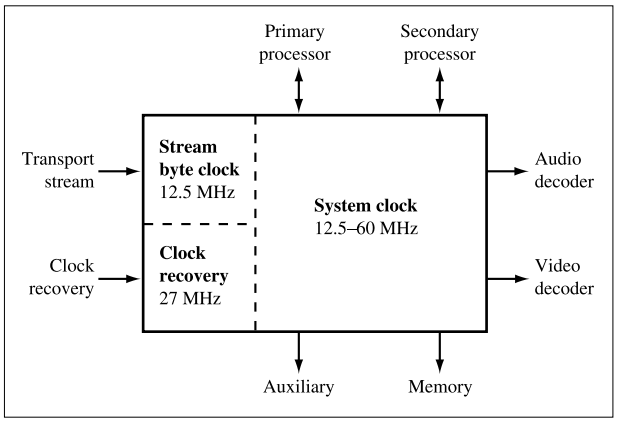

Figure 1

IBM transport macro core input and output ports, showing three different time domains and simulated frequency ranges.

This effort is described in several published papers [1–4]. A key difference from the described transport approach is that Genesys determines the final state of the design before actual simulation occurs. In the transport verification we must ensure that every byte of data is delivered correctly, regardless of all input events. By dynamically applying inputs across asynchronous bus interfaces, exact behavior cannot be predicted before simulation. Other companies have used similar methodologies for processor verification, including the Digital Equipment Company [5, 6] and Hewlett-Packard [7]. The DEC efforts most closely reflect our approach with dynamic testing; however, the overall asynchronicity of the transport bus interfaces poses a unique challenge apart from all of these works. But the applicability of these methods has not been limited to processor verification, as the ideas developed here grew from prior work verifying a serial level-2 (L2) cache. The L2 design is described in [8].

The techniques for finding implementation flaws before committing the design to silicon are described in this paper. Specifically, the IBM MPEG-2 Transport Demultiplexor is the core design referenced, and in-depth background on this architecture and on the set-top-box environment can be found in [9]. The MPEG-2 subsystem, including the transport and the video and audio decoders, is described in more detail in [10]. Results from actual hardware verification indicate that this methodology was very effective, and it is important to note that these verification techniques are applicable to many other types of design.

Pseudorandom verification

The transport demultiplexor is effectively the postal worker in a set-top-box system. The transport receives the incoming stream from a satellite receiver and separates the 188-byte packets using a synchronization byte which appears at the beginning of each packet. The packets are parsed to deliver data to the appropriate destinations, which include the video or audio decoders, the system processor, or the auxiliary port, and unrecognized packets are discarded.

- *Objectives*

The IBM MPEG-2 Transport Demultiplexor contains several independent ports across three unique time domains. A majority of the ports run off the chip system clock: the video and audio decoder ports, the system memory port, the primary and secondary processor ports, and the auxiliary output port. However, there are two other time domains within the core. The transport stream port operates off an independent transport stream clock, and the clock recovery port is based on a fixed 27-MHz clock. In addition, neither the system clock nor the transport clock has a defined frequency, and there is no fixed ratio between them. **Figure 1** illustrates the different time domains in the design and the range of frequencies to be simulated.

Asynchronous behavior results from the independent time domains. For example, the primary processor may initiate a channel change in the system clock domain; however, a packet of data from the old channel may currently be entering the core in the transport clock domain. Differing functional behavior depends on the point at which the packet-identifier (PID) filter is updated and the number of packet bytes that have entered the core. Another example is the posting of a clock recovery interrupt to the primary processor, which again is operating on the system clock. The interrupt is initiated in the clock recovery domain internal to the core, which continues to process any new clock information from incoming packets on the transport clock domain. Any difference in the ratios between the domains affects the expected behavior.

Another challenge is the internal array, which not only buffers up to ten packets but also stores the processor-initialized configuration for the 32 memory queues and the internal descrambler. Additionally, the array holds a scratchpad for queue-management working data. Hence, there is potential internal contention for access to the single SRAM.

With more than 10000 different configuration and status register bits within the IBM MPEG-2 Transport Demultiplexor, coverage of the different functional modes demands automation. It is simply too difficult a task to develop design confidence through a traditional suite of hand-generated tests.

Automation is also necessary when considering the infinite possibilities of the transport stream content. A

stream contains several multiplexed programs of audio and video data, along with the necessary system data, and each packet of data is identified by a unique tag or PID. A section of the stream could be heavily biased in a particular PID, or evenly weighted across many PIDs, or could contain a high percentage of stuffing packets, and the transport core must be stimulated with as many different stream compositions as possible.

Errors, known and unknown, are inherent in transport stream delivery and can occur at any time. The known errors are signaled either through an indicator bit within the incoming packet or by a dedicated pin from the stream driver, each resulting in predictable transport behavior. Unknown errors such as bit corruption or data loss can occur at any bit position of the stream, and may mislead the transport into unusual behavior.

Core delivery of data to the destinations occurs in a handshaking arrangement, and the receivers could introduce delay to the data throughput by stalling the delivery. The core must handle reasonable amounts of delay and backup in an appropriate manner and in any kind of operating situation.

Accommodation to design changes is important in a competitive world in which customer-driven architectural enhancements occur throughout the design cycle. Changes have the potential to introduce significant delay into traditional verification, since the accrued hand-generated tests may have to be individually rewritten.

- *Implementation*

A pseudorandom verification environment addresses all of these concerns. The idea is to have a self-contained environment in which new and unique tests are automatically generated and checked. Thus, regression capability is not constrained by manpower issues (writing and rewriting each test), but rather by material issues (availability of verification machines and simulation licenses). Each possible environment variant, be it a configuration bit, a clock ratio, a stream type, or a dynamic processor activity, is assigned a proper operating range and is randomly selected within that range for each new test case. The resulting core behavior is monitored by a software representation of the architectural specification—the golden software model expectations generator.

Figure 2 is a high-level flowchart of our pseudorandom verification environment. Several tools and significant effort are required in order to implement this pseudorandom verification environment efficiently. An important component is the mechanism which randomly creates valid test cases, e.g., a core configuration file, an input-stream-content parameter file, and a simulation-driver parameter file controlling bus-model behavior. The IBM MPEG-2 Transport Demultiplexor has over

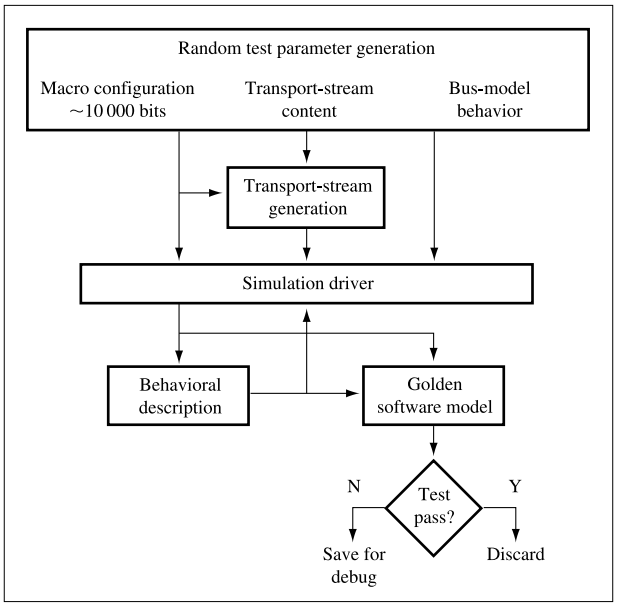

Figure 2

Test process flowchart of the pseudorandom transport simulation environment.

10 000 different configuration and status bits which must be initialized in order to create a valid operating environment. Most initialization choices are independent; however, some require knowledge of other chosen values. For example, two different PID filters cannot be initialized to the same enabled value. The transport-stream-content parameter file controls the overall flavor of the stream to be created, such as the ratios of video-to-audio-to-system packets and proportions of certain data fields. At this point the MPEG-2 transport stream can be created, providing the data stimulus to the transport. The stream generator uses the chosen core configuration along with the stream-content parameters to create a desired data stream.

The simulation-driver parameter file determines environment stimulus, such as transport control and configuration updates, decoder availability, transport-clock-to-system-clock ratio, and other environmental behavior. A simulation driver and test bench are needed to apply the test case by driving the inputs and reacting to the outputs. The driver and test bench incorporate behavioral descriptions for the transport and external interfaces, and in cases where no bus functional model exists, the necessary behavior is modeled by driver software.

A golden software model of the architectural specification that runs in real time with the logic is required to predict and then compare expected results

against those of the behavioral model. Hence the model is a key element of the verification process and can take on the additional role of being a regression monitor. The simulation driver simultaneously applies the test to both the behavioral description and the golden software model, which are independently derived. This approach ensures that both the hardware and software models are presented with the same operating conditions.

A regression tool ties the entire test flow together by automatically looping through the process to continue maximum verification per machine over a given time period. The regression vehicle calls the test-case-creation tools in the appropriate sequence, sets up the simulation environment, and kicks off the simulator. At the end of a test, the tool monitors the simulation log, and only failing tests are saved for further debug. Passing tests can be discarded because they provide little value to the designer, since new tests can be created in seconds. Once processing is cleaned up, the test-process cycle starts again.

- *Test creation*

To create effective and interesting test cases, consideration is given to all types of events that might occur, while ensuring that the test is of reasonable length to rerun if necessary for debug. Examples of possible events are channel changes and PID filter updates, descrambling key updates, other configuration changes, and soft resets, and these can occur at any point in the input stream. While the order and frequency of events should be somewhat realistic, it is also important to overemphasize the worst-case scenarios, such as a majority of the stream packets hitting the PID filter and system events occurring frequently. Techniques to bias certain events and the ability to easily disable certain features are very helpful. In regression, a two-level hierarchy of randomness is applied: one level to create the test-case parameter input files, and a second level within the transport stream generator itself. Program seeds should be recorded and saved, so that all test-case input files can be recreated or rerun in the same manner. Feedback from monitors within the golden model helps suggest changes to the test-generation programs.

An important first step in the test-case-generation process is creation of the transport core configuration file. This data is used by the simulator for initialization purposes and also by the transport stream generator in order to create appropriate streams.

The number of enabled PIDs for a given test can be anywhere from none to the maximum allowed in the architecture. A number between one and five is generally chosen, with a higher concentration of the tests enabling only one or two PIDs. The number of packets in the transport stream parameter file is a function of the number of enabled PIDs needed to produce an interesting amount of activity for each PID throughout the test. For each enabled PID filter, the following information must be chosen: PID value, auxiliary port enable, descrambling enable and key index, and the corresponding memory data queue enable. For each enabled memory data queue, the following information must be determined: the type of data to be unloaded to memory, filtering control words if appropriate, the designated address space for the data in memory, and other associated controls to further tailor management of the queue. All configuration, mask, index, and address space registers must be initialized. Also, all information necessary to perform descrambling must be created.

The core configuration generator requires no input parameter file, and the code is monitored and updated frequently to redistribute the types of tests being generated. After this first step in the test-case process, the resulting core configuration file is referenced for interdependencies as the parameter files for both the stream generator and the simulation environment are created.

- *Pseudorandom transport stream generator*

The goal of the transport stream generator is to create MPEG-2 test streams. For detailed information on the MPEG-2 transport layer protocol, see [11]. The independence of this tool allows it to be of use not only in our pseudorandom verification environment, but also in other deterministic environments such as system simulation, transport emulation, and real hardware verification. The generated streams are compliant with specification issues that are relevant to the function of the transport demultiplexor. Issues that are irrelevant to the transport function are created with the intent of rendering debug as straightforward as possible [e.g., audio and video program elementary stream (PES) data and system table section data between the table filter and the 32-bit cyclical redundancy check (CRC) tail]. The generator supports the option to incorporate real PES data.

The generator takes two files of data as input. One file includes relevant core configuration, such as enabled PIDs, memory queues, and table section filters, and the second controls the stream content. Nearly all decisions that the generator makes are random within appropriate limits; however, the user can steer the generator with event percentage control. For example, the content parameter file specifies the occurrence percentage of items in the transport stream header, of adaptation data and associated fields, of packets containing payload and associated content, and for insertion of packets that will not be processed, such as duplicates and nulls. If a user specifies that 10% of the packets should include an adaptation field, each packet has a 10% chance of doing so.

While the occurrence percentages of enabled PIDs are chosen in the parameter file, the actual order of the

multiplex is random. This is an important distinguishing point from hand-created streams. The decision process begins at the start of each packet, where first a PID is chosen for the current packet. Weighted percentages for stream content of video and audio are first considered, as are insertions of null and duplicate packets, and if a PID has still not been selected, a randomly enabled PID is chosen. The PID's history in the stream and applicable input parameters help to determine the remaining packet content. If the amount of PES data delivered is not fixed by the nature of that packet, it is always random.

The data portion of system packets, or the payload, is syntactically correct and can be split across nonconsecutive packets. The contents of the table section are dictated by the core's configuration of payload data. The stream generator creates all different combinations of applicable filters, and can easily create any variety of filter misses. A random valid section length is calculated and presented in the second and third bytes. The length may be chosen such that the section will not be fully contained within the packet (even the header could be split), or such that the section will end before the filter is complete.

If the length does allow the filter to complete, all but the final four remaining bytes of the section are filled with a random nonstuffing byte repeated throughout the section. The last four bytes are the valid 32-bit CRC for the table section, where again it is easy to insert a CRC error at any time in the test. If a section completes, either a new section header will begin or the remaining payload bytes will be stuffing. This decision is based on the existence of a payload unit start and a user control concerning multiple sections per packet.

The generator can take user-supplied scrambled datasets with corresponding keys to create an input stream which will produce known descrambled data at the output of the transport. This is to allow for verification of the descrambling function without having to incorporate the scrambling algorithm into the generator. The driver feeds the scrambled packet to the behavioral model and the descrambled packet to the software model expectations generator; thus, both the stream generator and the golden model are scrambling/descrambling-algorithm-independent.

A running program clock recovery value (PCR) is maintained and inserted on a percentage basis into packets of the PCR PID. The user can also introduce jitter into the PCR, which also can be designated to start at any nonzero value in order to test PCR wrapping.

These are just a few of the many features of the stream generator, all user-controllable via a parameter file. One of the more interesting items in the stream-generator parameter file is the number of packets that should be created per test. In order to create a substantial test that is not prohibitively long for debug, the number of packets depends upon the number of different PIDs that are enabled, the number of packets that will be ignored before synchronization is achieved, the percentages of injected errors, the maximum table length if tables will be created, and packet content if certain types will not be loaded.

- *Simulation driver*

The verification environment consists of a simulation driver that through a test bench and a traditional simulator applies the test case to both the behavioral logic and the golden software model expectations generator. All test cases minimally require a transport stream, core and driver configuration information, a test vehicle, and an analysis mechanism. Our test vehicle is a test bench with a driver, and analysis is performed by the golden model expectations generator.

Regardless of the origin of the test, the simulation driver is the moderator between the behavioral functional description of the transport and the golden software model expectations generator of the architectural specification. And at a higher level, the driver is the glue between a given test and the model being tested. Via a test bench, the driver presents the test case to both the behavioral and golden software models by performing core initialization and by methodically applying the transport stream and clocks to the transport. The test bench includes the behavioral descriptions for the transport demultiplexor and any external bus models that interface with the transport core.

Outputs of the transport are sensitized to be read for the golden model expectations generator upon any change in status. Since there is no communication between the behavioral and golden models, all coordination is controlled by the driver.

Software bus models are written into the driver. The bus-model-behavior parameter file determines which activities the driver will initiate during the test, and also how the driver and interfacing bus models will react to any given transport behavior. For example, as dictated by the bus-model parameter file, the driver can initiate various dynamic processor activities such as channel change (one or more per test), memory queue resets, stops, and filter updates, and control activities such as soft reset and disable of the transport synchronization function. Reactive processor activities include interrupt servicing and various associated activities such as external clock recovery.

The two processor ports are modeled by the driver with a request queue for each. These queues are filled and drained interactively as the test progresses, as opposed to being filled before the test is run. This allows for dynamic multiplexing as needed between the processor-initiated requests and the processor-reactive servicing requests. For example, the user does not know ahead of time if

the channel change will occur before or after a specific interrupt that will require servicing. And the secondary processor port can be enabled to drive random cycles, which may elicit a transport response.

The driver models the transport stream source, feeding stream data to the transport in the manner specified by the test case. This model provides us with a convenient mechanism for simulating channel changes with an efficient stream. If two or more video PIDs were multiplexed into the transport stream, the packets for the nonactive video PIDs would result in PID misses throughout the life of the test. We can avoid this wasted time by incorporating only one video PID into the stream and letting the driver manipulate the PID before presenting the channel-changed video packets to the transport. In a similar manner, the driver can also create duplicate packets, null packets, PID misses, invalid cycles, and data corruption as desired.

The video and audio decoder ports are also modeled, in order to simulate potential data throttling and channel-change handshaking. The decoders cannot always receive a continuous stream of data, so the capability to deactivate their respective requests for data is important. We can additionally target the deactivation to occur in the midst of a data transfer, which is the most interesting time.

The driver provides the clocks to the test environment, and the system clock frequency parameter creates various asynchronous system-to-transport-stream and system-to-clock-recovery clock ratios. This allows us to easily vary operating conditions for the core ports across the three different time domains to create all types of asynchronous activity.

These are just a few of the capabilities of the simulation driver. An important point is that any combination of these events is possible, except when events or configurations are mutually exclusive by the architectural specification. An invalid example is a system-to-transport-byte-clock ratio less than 4:1 if descrambling is to occur.

- *Golden software model expectations generator*

The expects generator is a complete software representation of the architected behavior of the transport demultiplexor. The software model receives the same initial configurations and inputs from the test bench and driver as does the behavioral model, with only a few internal logic hooks to synchronize activities. The software model determines what should be occurring on the basis of event history, and it monitors the behavioral model via the driver to ensure compliance.

Since the expectations generator runs with the event simulator, the activities that occur throughout the test do not have to be spelled out ahead of time. Another important point is that the software model predicts outputs without demanding exact cycle correlation. While some synchronization mechanisms are added for the golden model, they are limited. This allows the design to be radically changed with respect to the occurrence of events with minimal impact on the golden model.

The golden model acts as the test-case referee, since it has knowledge of all events that have occurred and what is expected to occur from architected protocol. Notable events and varying degrees of running commentary are available to a user log. At the end of a test, a summary of bus compares and miscompares is displayed; also, the final state of the memory model is read and compared against the software model's contents. If a discrepancy is noted during or at the conclusion of a test, various severities of error flags are raised.

If desired, output dump files can be created for any of the input or output ports, as well as for the final image of memory. These files are useful in other test environments, such as emulation and hardware labs. In such environments, the running expectations generator is not available, and more traditional, predetermined test cases are desired.

One of the more difficult aspects of matching the software representation to the behavioral description is due to the sequential nature of coding as opposed to the simultaneous nature of logic simulation. For example, when two monitored signals in the logic both switch state in the same cycle, the software still sees one occurring before the other. Another challenge concerns the time-domain crossings, since the expectations generator is cycle-independent wherever possible. The sequential nature of software and the time-domain crossings present difficulties, particularly in the areas of interrupt reporting and processor-initiated configuration changes. To aid in coordination, the simulation driver includes a few internal logic monitors to reflect exactly when certain events were occurring. Every attempt was made to minimize the number of internal hooks in order to maintain overall verification confidence and control.

- *Checking the pseudorandom test environment*

An important item in the control of this simulation structure is to have separate parties create and maintain the components. There are several distinct jobs to be performed: architectural definition, behavioral description, and creation of the simulation environment, which includes the golden software model expectations generator. Since both the behavioral and software descriptions are based on the architectural specification, independent coders provide protection against designing and modeling the same mistake. Care is taken to keep the golden software model and the behavioral description as independent from each other as possible; however, a great deal of coordination is necessary in order to emulate the internal order of events.

Ideally, the golden software model and the simulation environment driver should be complete and stable before the logic implementation begins. In our case, these activities occurred in parallel and facilitated quick changes to the architectural definition as it evolved. One side effect of this approach is that a failing test is not necessarily indicative of a bug in the implementation, but may be caused by an invalid test condition or by a bug in the golden model. The correction adds to the overall strength of the test environment, and in some cases to a clarification or elaboration of the architectural specification.

The hardware emulator to be described later gives an important check into the validity of our verification methodology. The pre-silicon prototype allows for the ability to view a program with synchronous audio and video on a television screen, thus providing invaluable design confidence.

A limited number of event monitors were inserted in a variety of places, including the simulation driver, the golden software model, and the regression tool, to provide a glimpse of verification coverage. However, future work in this area would certainly enhance the strength of this methodology, and could provide more feedback for biasing weights of event parameters. Test coverage is understandably an important measure of any verification procedure, yet it is a difficult measure to quantify, particularly in a pseudorandom test-generation environment. We relied on our intimate knowledge of the tools and models to maintain our confidence in this strategy. However, justification to others is more difficult, and more focus on test coverage would certainly strengthen this debate.

- *Pseudorandom verification conclusions*

In light of the challenges presented in verifying the design of the transport demultiplexor, many advantages of our pseudorandom test generation with simultaneous modeling are apparent. Windows of conflict between competing interfaces on perhaps different time domains that could not be anticipated are exercised by allowing all valid events at all pertinent clock ratios. For example, a channel change request could be made while a program filter for that channel is being parsed in the model. Scores of time-domain ratios can be tested, with the correct behavior dictated as the test unfolds. With all other inputs and configurations unchanged, two tests with different clock ratios can behave in a radically different manner.

The monotony of manual test generation and event sweeping is avoided, yet the user is able to hand-code a test if desired. If a test exposes a design bug, it can easily be tweaked to test similar scenarios. Also, the test can be saved to create a regression bucket of tests that have reached trouble spots in the design. This method is more flexible and more adaptive to major design changes than traditional verification with handwritten test cases. With large change, many traditional test cases would potentially have to be rewritten, which could take a significant amount of time. With a pseudorandom environment, the design change is reflected in the golden software model and the test-case-generation tools; the number of tests run on the new design is limited only by the number of simulation licenses and machines available. (For example, with eight machines and simulation licenses, more than a thousand new tests could be created and run in one night.) This point is also stated in [1].

Invalid transport stream conditions are easily created to verify appropriate transport response, such as random single-bit error corruption, transport-stream-error indicator activity, short or long packets, program ID filter misses, continuity counter misses, table-section filter misses, short table sections, and cyclical redundancy check errors in table sections.

All types of event monitors are easily incorporated into the golden software model, since by definition everything that happens in the logic happens in the model. These monitors aid in the immediate determination of events in a particular test case, and also can provide a collective view of the overall regression coverage.

Since the golden model expects generator runs simultaneously with the test driver, the expected outputs need not be determined before the test is run. This permits dynamic reactive test-bench processing to be interspersed appropriately among premeditated simulation events.

There is no need to save a repository of passing tests, since new test cases can be created in a short amount of time. Aside from overall regression coverage, passing test cases are not very interesting. We can automatically scan the passing logs for statistical purposes and then discard them from the database.

As mentioned earlier, pseudorandom verification has been successful for other scopes of design, namely microprocessors. It can also be effective on smaller-scale designs, such as a serial level-two (L2) cache. For example, processor cache/memory requests can be dynamically intertwined with asynchronous system "snoop" events.

Pseudorandom verification successfully prepared the transport design to enter the emulation-verification process, where design updates begin to take a larger toll in both time and cost. The emulator allows the design to be tested with longer streams containing real audio and video programs as well as the clock recovery data in order to verify the portions of our design not adequately covered by short test streams. And the pseudorandom regression can easily be continued through hardware build and test, since it requires little manual intervention or resource.

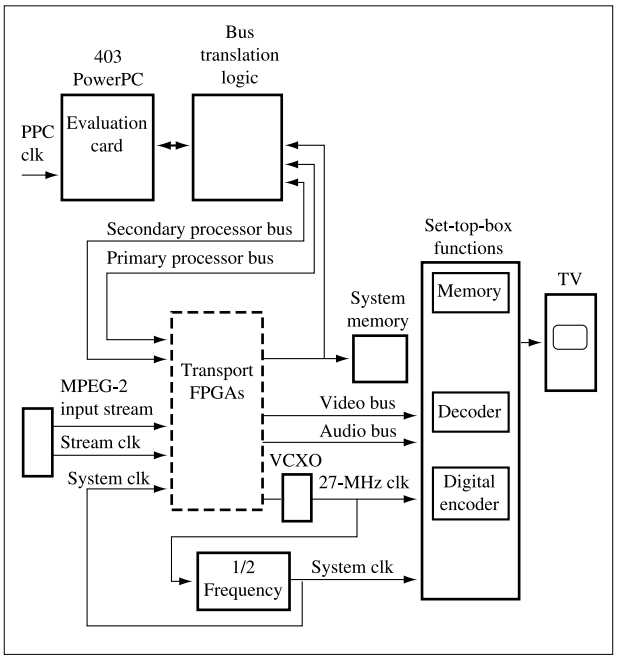

Figure 3

Diagram of transport emulator system.

Hardware emulation

A hardware-based system to emulate the core design was developed in conjunction with the pseudorandom verification system. Emulation complements the pseudorandom verification method by using actual MPEG-2 streams in a real-time set-top-box environment. Events such as system power-up, real-time host control of the transport hardware, and transport features exercised by extremely long MPEG-2 streams are ideally suited for verification by emulation.

- *Objectives*

Our goals for emulation were to integrate a functional representation of the transport core in a set-top-box subsystem. The subsystem used a host processor for control and was capable of processing MPEG-2 streams. By maintaining uniformity between the transport hardware description language (HDL) and the emulator HDL, we could provide feedback to the design team that was directly related to their logic. To be useful in the product cycle, the emulation process had to provide fast turnaround capability for design updates.

We also wanted to provide a software development platform for constructing and debugging software drivers, and a demonstration tool for customers, marketing, and development teams. An added benefit would be to use the emulation system to verify the simulation tool. This could be done by running the contrived MPEG-2 streams from simulation into the emulator and comparing the results. Consideration of these design goals forced us to address trade-offs between the technology and the methodology used to implement the emulator.

One of the first considerations in any design including an emulator is the design goals. By stating the design goals we were forced to address trade-offs between the technology and the methodology used to implement the emulator.

- *System features*

Emulation of the IBM transport demultiplexor is based on a printed-circuit board populated with field-programmable gate arrays (FPGAs). The emulation platform was created as an integrated set-top-box (STB) subsystem, with the transport core partitioned into multiple FPGAs. This produced a realistic environment for verification of the core, as well as the ability to change the design as development proceeded. With this system it is possible to connect a live unscrambled MPEG-2 stream to the input port and watch the decoded audio/video output on a television.

To emulate the transport core as it would operate in a set-top box, we designed the system board with some STB functions included. This allows the core to function in a realistic systems environment. The major system components, shown in **Figure 3**, include a PowerPC 403* evaluation card. This IBM product offers the user an IBM PowerPC 403 microcontroller set up with a memory subsystem and some ROM boot code. It interfaces with either an Ethernet port or a serial port. The card provided a programmable control interface for the emulation system. An interface was designed between the 403 processor bus and the transport bus. Using FPGAs enabled us to add function as the core design required it. We did not have to wait until the design was done to implement logic.

We added an MPEG-2 transport stream port to allow processing of real and contrived streams. To make comparisons between pseudorandom simulation results and the hardware, we provided 0.5 MB of system memory at the output of the transport core. To verify real-time operation with industry components, the output of the transport was also passed to a set-top-box back-end subsystem, allowing visual verification on a television monitor and speakers.

- *HDL design portability*

Most large designs today are written in a high-level design language (HDL) such as VHDL or Verilog. In order for the emulation system to aid in the development of the logic design, it is important to be able to periodically capture the design changes made in the HDL and update

the FPGA configurations. This process provides feedback on design functionality throughout the development cycle. In order to update the FPGAs with minimal impact to the emulation system or the core logic, it is important that the emulation team be involved early. There are several important steps for the team to perform that will aid in the design update process.

Early on, functional blocks of the design HDL should be synthesized to verify that the coding style being used by the designer provides reasonable results in the FPGA. This also flushes out any library-dependent circuits being used by the designer. These library-specific functions have to be mapped to generic HDL blocks in order to be properly synthesized into an FPGA library element. It helps to keep the HDL design style as generic as possible.

System logic blocks should be sized and partitioned among FPGA modules, allowing room for logic and I/O expansion. This makes processing design snapshots quicker and requires less rework. A rule of thumb is to initially populate any FPGA module to no more than 50–60% of its logic and I/O capacity.

- *Partitioning on the basis of density and I/O*

For an emulator to be a valuable tool in the verification of a logic system, it must be available for use before the core logic is frozen. This requires an early start in order to get boards designed and any special interfaces laid out. In the transport emulator, we began defining logic and I/O partitions as soon as there was enough HDL code written to start core simulation. At this point in the design development, the connectivity between logic blocks is established, and much of the logic function is defined.

Our first considerations in developing logic partitions are the different core functions, the size of the buses that connect them, and the time domain in which they exist. We chose logic functions that related to one another and had the most interconnections. Doing this reduced the amount of external I/O needed for each FPGA. Next, we tried to keep partitions in one time domain. This is not an FPGA timing requirement, but it helps in understanding the overall system timing. Once a partition was chosen on the basis of logic function, we needed a way to determine the I/O requirement for a possible partition; to do this, we created a software tool. The nature of the different types of connections that can exist between logic blocks in a design and the way they are coded in an HDL forms the basis for text comparison. **Figure 4** shows a potential logic partition and the kinds of interconnections that can exist between logic blocks. The only connections we want to identify are the ones that enter or leave the partition. In this diagram the FPGA would need only two inputs and two outputs. Our tool determines that signal_A did not leave the partition; therefore, it should not be counted as an I/O on the FPGA. By using the HDL to create a data

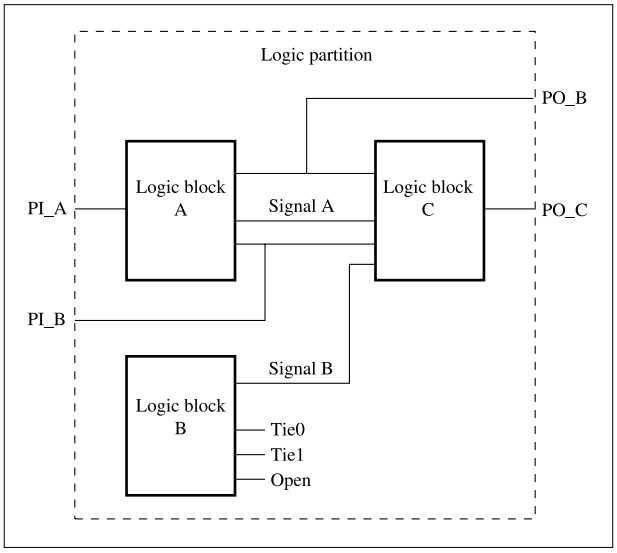

Figure 4

Logic partition example.

file of all declared inputs, outputs, and signals for the logic blocks in a potential partition from the HDL, we can perform text analysis to determine whether a signal enters or leaves the logic partition. By using this tool, we are able to create "what if" partitions of logic and quickly determine whether they exceed the FPGA I/O limit.

Once logic is partitioned from an I/O perspective, it is necessary to determine the number of FPGA gates that will be required in each partition. FPGA manufacturers define device sizes differently. Some use gates, logic cells, or logic elements. Most FPGAs consist of an array of logic blocks surrounded by programmable interconnections. Each logic block has a combinatorial function and one or two flip-flops. This does not allow a one-to-one correlation between an FPGA gate count and the IBM technology gate count for a piece of logic. The FPGA resource required for a logic partition can be estimated by determining the ratio of FPGA logic cells to IBM gates. This is done by synthesizing a cross section of logic blocks and coding styles that are used in the core design to the FPGA library and to the IBM library. Comparing the number of FPGA logic cells required by a block of logic to the number of IBM gates for that same logic block gives the ratio of FPGA cells to IBM gates. The logic-cell ratio can then be used to approximate FPGA logic capacity throughout the partitioning process. Otherwise each potential partition would have to be synthesized using the FPGA tools to determine whether the partition fits. In the transport core, we found the ratio to be six IBM gates to one Altera** logic cell. By determining the

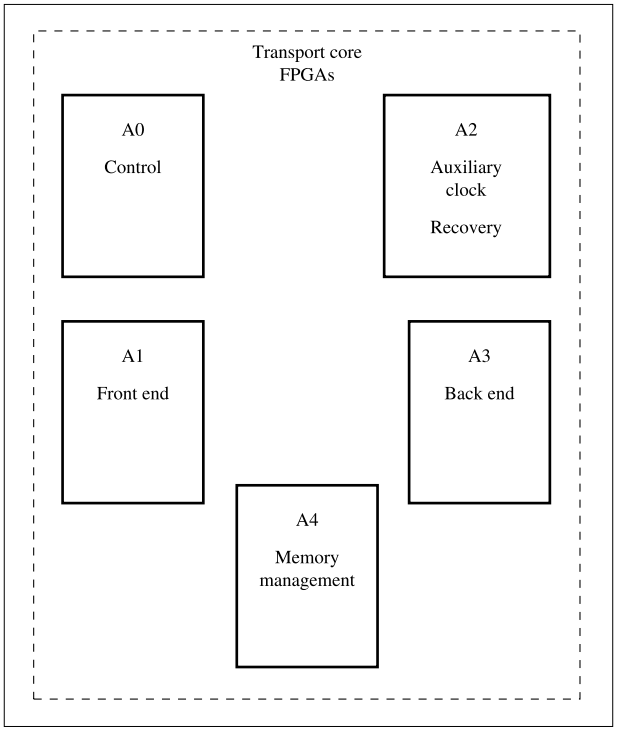

Figure 5

Transport FPGA partitions.

I/O required for a logic partition, and estimating the number of FPGA logic cells required, we can create logic partitions (**Figure 5**) that use only 50% or 60% of an FPGA's logic cell and I/O capacity. This ensures that the design can change and still place and route in the FPGA.

- *Multiple time domains*

The transport system contains multiple time domains consisting of the transport clock, the system clock, and the program clock recovery time domains. The emulator system clock must run at a frequency that allows enough time for the front-end processing of the MPEG-2 stream and keeps enough audio/video data flowing to the decoders. Through simulation, we found that maintaining this data rate requires a minimum 1:1.5 ratio between the transport clock and the system clock. For example, with an 8-MHz transport clock, we have to run the emulator system clock at 12 MHz. In a customer set-top-box environment, the ratio would be at least 1:4. The ability to run the transport so close in frequency to the transport clock and maintain data flow is a tribute to the design robustness of the transport core.

The program clock recovery (PCR) logic operates at 27 MHz. This unit processes PCR values from the MPEG-2 stream and feeds adjustments to a voltage-controlled oscillator (VCXO). This adjustment is necessary to keep the decoding set-top-box 27-MHz clock as close as possible to the 27-MHz clock that was used in generating the MPEG-2 stream on the provider end.

The following timing restrictions dictated our minimum operating clock frequencies: 8 MHz for the MPEG-2 transport clock, 12 MHz for the system clock, and 27 MHz for the program clock recovery. On the basis of these frequency requirements, parts of the design were synthesized, and a performance bottleneck was found in the PCR logic. This unit did not achieve the 27-MHz clock performance target in the FPGA because of the path lengths of some of the combinatorial logic. When using FPGAs, there is a trade-off between design speed and HDL tweaking. Obtaining maximum performance on large pieces of logic requires tailoring the HDL to the FPGA technology by running synthesis, recording logic, and running FPGA place-and-route tools. This reduces the number of logic levels in critical paths. To obtain the most control over timing and performance in some FPGAs, schematics are required. Schematics facilitate the tuning of logic placement in the FPGA. To maintain a short FPGA processing cycle and preserve the transport HDL, we decided to reduce the accuracy of the emulated PCR logic by halving the PCR clock speed. Now our target for the program clock recovery unit was the same as our system clock rate—13.5 MHz. This allowed us to share one frequency with the system clock, greatly simplifying the overall clock design. For each new design pass, the clock sharing allowed us to make runs through synthesis and to place and route the FPGAs without spending a lot of time tweaking the PCR logic.

- *System debug features*

To help in the debug process, the emulation system board was designed with connectors placed on all of the major buses. These were designed with proper spacing and orientation to mate with logic analyzers and oscilloscopes. When dealing with hundreds of I/O elements, this can save many hours of setup. Additional connectors were placed around each FPGA, with 40 FPGA I/Os connected to them. This provided a means for accessing internal logic signals during debug. In order to use the emulator early in the transport design cycle, the printed-circuit board (PCB) had to be designed to accommodate future changes resulting in additional signals between FPGAs. To allow for this, a ten-signal bus was added between each FPGA and the adjacent one in the emulator printed-circuit board. We also added extra buffers on the PCB to handle changes to the clock tree. Clock control logic was added so that clocks to certain partitions could be shut off. We put pads on the faster signals near the inputs to the FPGAs. This ensured that the net would be accessible

from the top or bottom layer of the PCB if signal tuning was necessary.

An interface to the 403 evaluation card was added from the system memory so that we could read and verify data written by the transport core. FIFOs accessible from the 403 control bus were placed on the audio and video data buses so that if necessary we could capture data coming from the transport core to the decoder.

For early system bring-up, we added a bypass multiplexor to allow the controlling processor to write audio and video data directly to the decoder FIFOs. This gave us the opportunity to verify the back-end set-top-box functions separately from the front-end transport functions.

- *Emulation results*

There were many benefits to emulating the transport core. Confidence in the design and enthusiasm from marketing grew as we progressed and saw more functions working. We were able to show potential customers progress as it was made. We also had the option of sending systems to customers or other development groups for evaluation and feedback.

The clock recovery portion of the transport core could not be verified with simulation patterns alone. The output of the clock recovery unit in the transport drives a voltage-controlled oscillator in a set-top box. Due to the analog nature of this signal it was verified in emulation.

Software development was probably the biggest beneficiary of the emulation system. Months were saved by developing code before system hardware was available and thus reducing time to market. We found several design bugs, but not of the magnitude that was expected. This can be attributed to the design/simulation team and the effectiveness of our pseudorandom simulation methodology. Using the emulation system, we were able to verify the simulation tests in a real environment by comparing output from the emulator to output from simulation. This allowed a cross-check of our pseudorandom verification process with real hardware.

Deriving a ratio of IBM gates to FPGA logic cells saved considerable time during the interactive process of partitioning. Instead of synthesizing, placing, and routing each potential partition to see whether it used too much logic for the FPGA, we knew when creating the partition how much logic it would require. When emulating a core designed with a high-level design language, emphasis should be placed on portability of the design to the FPGAs. We found that doing a few trial synthesis runs during design development and keeping to fundamental HDL constructs showed that the design was portable but still required some modification. Changes were required when designers accustomed to using a much faster technology allowed combinatorial paths to grow too long. In these cases, the logic paths were shortened in order to accommodate emulation. The time it took to update the design in the FPGAs depended on the magnitude of the changes that were made in the core. Toward the end of the project, some of the FPGAs were at 70% of logic capacity and 80% of I/O capacity; this translated into longer place-and-route times in the FPGA tools.

Summary

A unique pseudorandom verification approach was chosen to verify the transport demultiplexor core because of the asynchronous nature of possible events and the enormous collection of operating configurations. This method allowed us to successfully verify unforeseen scenarios and sequences in an automated yet meaningful manner.

Hardware emulation provided an important independent verification of the transport design, not to mention the simulation environment itself, and it gave us an early real-time vehicle on which to begin advanced software development. The calculations for logic I/O and density proved accurate. We were able to absorb major transport design changes without significant impact to the emulation system or the transport design. Trade-offs that were made in clock performance to keep from disturbing the original design and reduce processing time benefited each logic update in the emulator. Emulation systems can provide benefits to many groups in a design project, but it is important to identify the scope and limitations involved in reproducing a system in another technology.

It is impossible to measure the number of design bugs exposed by this verification environment, since the architecture, design, and environment tools were all new and were developed simultaneously. Also, it is often difficult to prove productivity and success without easily quantified measurements. However, we can use the success of the hardware as an indication of the overall validity and coverage of these methods. The types of minor issues that escaped the pseudorandom simulation were primarily in the areas of interface modeling: bits swapped on buses or protocol misunderstandings. Many of these were caught in the emulation system, and no significant problems were built in silicon.

This combined verification methodology enabled our team to release a solid design to manufacturing, and it also provided invaluable extra time for developing the necessary product software applications. The complementing verification techniques described in this paper could be applied outside the realm of transport demultiplexor design.

Acknowledgments

The authors wish to acknowledge the following for their contributions toward the development of the topics in this paper: Richard E. Anderson, Susan F. Bueti, Richard DesRoches, Eric M. Foster, Ian R. Govett, Jay G. Heaslip,

Donald S. Plosila, George W. Rohrbaugh, Bruce W. Singer, and Timothy J. Vonreyn.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Altera Corporation.

References

1. L. Fournier, Y. Arbetman, and M. Levinger, "Functional Verification Methodology for Microprocessors Using the Genesys Test-Program Generator," *DATE99 Proceedings*, Munich, Germany, March 1999, pp. 434–441.
2. A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, C. Metzger, M. Molcho, and G. Shurek, "Test Program Generation for Functional Verification of PowerPC Processors in IBM," *Proceedings of the 32nd Design Automation Conference*, San Francisco, June 1995, pp. 279–285.
3. Y. Lichtenstein, Y. Malka, and A. Aharon, "Model-Based Test Generation for Processor Design Verification," *Proceedings of the Sixth Innovative Applications of Artificial Intelligence (IAAI) Conference*, AAAI Press, Menlo Park, CA, 1994, pp. 83–94.
4. A. Aharon, A. Bar-David, B. Dorfman, E. Gofman, M. Leibowitz, and V. Schwartzburd, "Verification of the IBM RISC System/6000 by a Dynamic Biased Pseudo-Random Test Program Generator," *IBM Syst. J.* **30,** No. 4, 527–538 (1991).
5. M. Kantrowitz and L. M. Noack, "Functional Verification of a Multiple-Issue, Pipelined, Superscalar Alpha Processor—the Alpha 21164 CPU Chip," *Digital Technical Journal of Digital Equipment Corporation* **7,** No. 1, 89–99 (Winter 1995).
6. J. H. Zurawski, J. E. Murray, and P. J. Lemmon, "The Design and Verification of the AlphaStation 600 5-Series Workstation," *Digital Technical Journal of Digital Equipment Corporation* **7,** No. 1, 136–144 (Winter 1995).
7. S. T. Mangelsdorf, R. P. Gratias, R. M. Blumberg, and R. Bhatia, "Functional Verification of the HP PA 8000 Processor," *Hewlett-Packard J.* **48,** No. 4, 22–31 (August 1997).
8. G. Giacalone, R. Busch, F. Creed, A. Davidovich, S. Divakaruni, C. Drake, C. Ematrudo, J. Fifield, M. Hodges, W. Howell, P. Jenkins, M. Kozyrczak, C. Miller, T. Obremski, C. Reed, G. Rohrbaugh, M. Vincent, T. Vonreyn, and J. Zimmerman, "A 1MB, 100MHz Integrated L2 Cache Memory with 128b Interface and ECC Protection," *1996 IEEE International Solid-State Circuits Conference Digest of Technical Papers, 42nd ISSCC,* 1996, pp. 370–371, 475.
9. R. E. Anderson and E. M. Foster, "Design of an MPEG-2 Transport Demultiplexor Core," *IBM J. Res. Develop.* **43,** No. 4, 521–532 (1999, this issue).
10. R. E. Anderson, E. M. Foster, D. E. Franklin, and R. S. Svec, "Integrating the MPEG-2 Subsystem for Digital Television," *IBM J. Res. Develop.* **42,** No. 6, 795–805 (1998).
11. "Information Technology—Generic Coding of Moving Pictures and Associated Audio Information: Systems," *ISO/IEC 13818-1*, First Edition, April 1996.

Received November 18, 1998; accepted for publication February 18, 1999

Charlotte A. Reed *IBM Microelectronics Division, Burlington facility, Essex Junction, Vermont 05452 (chareed@us.ibm.com).* Ms. Reed completed a B.S. degree in electrical engineering with a dual major in mathematics from Syracuse University in 1988. She began her career in vector processors and caches for main frames with IBM, receiving an M.S. degree in computer engineering in 1992. Since joining the Microelectronics Division in 1994, she has worked in PC chipset and L2 cache design. Her past two years have been focused on verification of the IBM MPEG-2 transport demultiplexor. Ms. Reed is the coauthor of one patent.

Dana J. Thygesen *IBM Microelectronics Division, Burlington facility, Essex Junction, Vermont 05452 (dthyg@us.ibm.com).* Mr. Thygesen graduated from Vermont Technical College with an A.S. degree in electrical engineering in 1978, joining IBM that same year. His work experience includes test analysis engineering, 486/386-based in-circuit emulation, product development, FPGA development, and applications engineering. Mr. Thygesen is the coauthor of a paper presented at the fourth Canadian Workshop on Field Programmable Devices, and he is a co-inventor on an issued patent. For the past year he has worked on the development of an MPEG-2 transport emulator.