78

Rapid Design and Prototyping of DSP Systems

T. Egolf, M. Pettigrew, J. Debardelaben, R. Hezar, S. Famorzadeh, A. Kavipurapu, M. Khan, Lan-Rong Dung, K. Balemarthy, N. Desai, Yong-kyu Jung, and V. Madisetti

Georgia Institute of Technology

The Rapid Prototyping of Application-Specific Signal Processors (RASSP) [1, 2, 3] program of the U.S. Department of Defense (ARPA and Tri-Services) targets a 4X improvement in the design, prototyping, manufacturing, and support processes (relative to current practice). Based on a current practice study (1993) [4], the prototyping time from system requirements definition to production and deployment, of multiboard signal processors, is between 37 and 73 months. Out of this time, 25 to 49 months are devoted to detailed hardware/software (HW/SW) design and integration (with 10 to 24 months devoted to the latter task of integration). With the utilization of a promising top-down hardware-less codesign methodology based on VHDL models of HW/SW components at multiple abstractions, reduction in design time has been shown especially in the area of hardware/software integration [5]. The authors describe a top-down design approach in VHDL starting with the capture of system

0-8493-8572-5/98/$0.00+$.50

requirements in an executable form and through successive stages of design refinement, ending with a detailed hardware design. This hardware/software codesign process is based on the RASSP program design methodology called virtual prototyping, wherein VHDL models are used throughout the design process to capture the necessary information to describe the design as it develops through successive refinement and review. Examples are presented to illustrate the information captured at each stage in the process. Links between stages are described to clarify the flow of information from requirements to hardware.

78.1 Introduction

We describe a RASSP-based design methodology for application specific signal processing systems which supports reengineering and upgrading of legacy systems using a virtual prototyping design process. The VHSIC Hardware Description Language (VHDL) [6] is used throughout the process for the following reasons. One, it is an IEEE standard with continual updates and improvements; two, it has the ability to describe systems and circuits at multiple abstraction levels; three, it is suitable for synthesis as well as simulation; and four, it is capable of documenting systems in an executable form throughout the design process.

A *Virtual Prototype* (VP) is defined as an executable requirement or specification of an embedded system and its stimuli describing it in operation at multiple levels of abstraction. *Virtual prototyping* is defined as the top-down design process of creating a virtual prototype for hardware and software cospecification, codesign, cosimulation, and coverification of the embedded system. The proposed top-down design process stages and corresponding VHDL model abstractions are shown in Fig. 78.1. Each stage in the process serves as a starting point for subsequent stages. The testbench developed for requirements capture is used for design verification throughout the process. More refined subsystem, board, and component level testbenches are also developed in-cycle for verification of these elements of the system.

The process begins with requirements definition which includes a description of the general algorithms to be implemented by the system. An algorithm is here defined as a system's signal processing transformations required to meet the requirements of the high level paper specification. The model abstraction created at this stage, the *executable requirement,* is developed as a joint effort between contractor and customer in order to derive a top-level design guideline which captures the customer intent. The executable requirement removes the ambiguity associated with the written specification. It also provides information on the types of signal transformations, data formats, operational modes, interface timing data and control, and implementation constraints. A description of the executable requirement for an MPEG decoder is presented later. Section 78.4 addresses this subject in more detail.

Following the executable requirement, a top-level *executable specification* is developed. This is sometimes referred to as functional level VHDL design. This executable specification contains three general categories of information: (1) the system timing and performance, (2) the refined internal function, and (3) the physical constraints such as size, weight, and power. System timing and performance information include I/O timing constraints, I/O protocols, and system computational latency. Refined internal function information includes algorithm analysis in fixed/floating point, control strategies, functional breakdown, and task execution order. A functional breakdown is developed in terms of primitive signal processing elements which map to processing hardware cells or processor specific software libraries later in the design process. A description of the executable specification of the MPEG decoder is presented later. Section 78.5 investigates this subject in more detail.

The objective of data and control flow modeling is to refine the functional descriptions in the executable specification and capture concurrency information and data dependencies inherent in the algorithm. The intent of the refinement process is to generate multiple implementation independent representations of the algorithm. The implementations capture potential parallelism in the algorithm at a primitive level. The primitives are defined as the set of functions contained in a design library consisting of signal processing

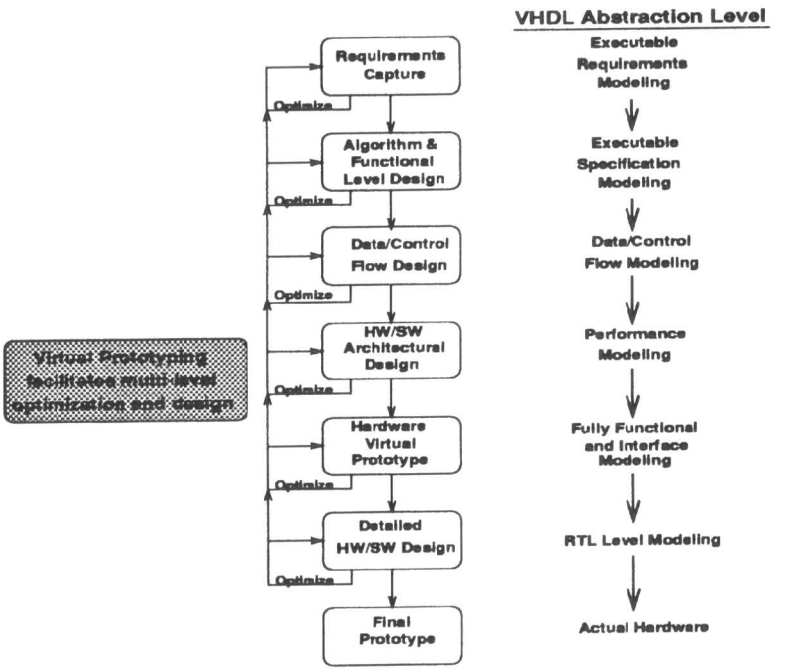

FIGURE 78.1 The VHDL top-down design process.

functions such as Fourier transforms or digital filters at course levels and of adders and multipliers at more fine-grained levels. The control flow can be represented in a number of ways ranging from finite state machines for low level hardware to run-time system controllers with multiple application data flow graphs. Section 78.6 investigates this abstraction model.

After defining the functional blocks, data flow between the blocks, and control flow schedules, hardware-software design trade-offs are explored. This requires architectural design and verification. In support of architecture verification, *performance level modeling* is used. The performance level model captures the time aspects of proposed design architectures such as system throughput, latency, and utilization. The proposed architectures are compared using cost function analysis with system performance and physical design parameter metrics as input. The output of this stage is one or few optimal or nearly optimal system architectural choice(s). In this stage, the interaction between hardware and software is modeled and analyzed. In general, models at this abstraction level are not concerned with the actual data in the system but rather the flow of data through the system. An abstract VHDL data type known as a token captures this flow of data. Examples of performance level models are shown later. Sections 78.7 and 78.8 address architecture selection and architecture verification, respectively.

Following architecture verification using performance level modeling, the structure of the system in terms of processing elements, communications protocols, and input/output requirements is established. Various elements of the defined architecture are refined to create hardware virtual prototypes. *Hardware virtual prototypes* are defined as *software simulatable* models of hardware components, boards, or systems containing sufficient accuracy to guarantee their successful realization in actual hardware. At this abstraction level, fully functional models (FFMs) are utilized. FFMs capture both internal and external (interface) functionality completely. Interface models capturing only the external pin behavior are also used for hardware virtual prototyping. Section 78.9 describes this modeling paradigm.

Application specific component designs are typically done in-cycle and use register transfer level (RTL) model descriptions as input to synthesis tools. The tool then creates gate level descriptions and final layout

information. The RTL description is the lowest level contained in the virtual prototyping process and will not be discussed in this paper because existing RTL methodologies are prevalent in the industry.

At least six different hardware/software codesign methodologies have been proposed for rapid prototyping in the past few years. Some of these describe the various process steps without providing specifics for implementation. Others focus more on implementation issues without explicitly considering methodology and process flow. In the next section, we illustrate the features and limitations of these approaches and show how they compare to the proposed approach.

Following the survey, Section 78.3 lays the groundwork necessary to define the elements of the design process. At the end of the paper, Section 78.10 describes the usefulness of this approach for life cycle support and maintenance.

78.2 Survey of Previous Research

The codesign problem has been addressed in recent studies by Thomas et al. [7], Kumar et al. [8], Gupta et al. [9], Kalavade et al. [10, 11], and Ismail et al. [12]. A detailed taxonomy of HW/SW codesign was presented by Gajski et al. [13]. In the taxonomy, the authors describe the desired features of a codesign methodology and show how existing tools and methods try to implement them. However, the authors do not propose a method for implementing their process steps. The features and limitations of the latter approaches are illustrated in Fig. 78.2 [14]. In the table, we show how these approaches compare to the approach presented in this chapter with respect to some desired attributes of a codesign methodology. Previous approaches lack automated architecture selection tools, economic cost models, and the integrated development of test benches throughout the design cycle. Very few approaches allow for true HW/SW cosimulation where application code executes on a simulated version of the target hardware platform.

DSP Codesign Features	TA93	KA93	GD93	KL93 KL94	IJ95	Proposed Method
Executable Functional Specification	✓	✓	✓	✓	✓	✓
Executable Timing Specification		✓	✓			✓
Automated Architecture Selection						✓
Automated Partitioning			✓	✓		✓
Model-Based Performance Estimation		✓	✓			✓
Economic Cost/Profit Estimation Models						✓
HW/SW Cosimulation				✓		✓
Uses IEEE Standard Languages		✓				✓
Integrated Test Bench Generation						✓

FIGURE 78.2 Features and limitations of existing codesign methodologies.

78.3 Infrastructure Criteria for the Design Flow

Four enabling factors must be addressed in the development of a VHDL model infrastructure to support the design flow mentioned in the introduction. These include model verification/validation, interoperability, fidelity, and efficiency.

Verification, as defined by IEEE/ANSI, is the process of evaluating a system or component to determine

whether the products of a given development phase satisfy the conditions imposed at the start of that phase. Validation, as defined by IEEE/ANSI, is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies the specified requirements. The proposed methodology is broken into the design phases represented in Figure 78.1 and uses black- and white-box software testing techniques to verify, via a structured simulation plan, the elements of each stage. In this methodology, the concept of a reference model, defined as the next higher model in the design hierarchy, is used to verify the subsequently more detailed designs. For example, to verify the gate level model after synthesis, the test suite applied to the RTL model is used. To verify the RTL level model, the reference model is the fully functional model. Moving test creation, test application, and test analysis to higher levels of design abstraction, the test description developed by the test engineer is more easily created and understood. The higher functional models are less complex than their gate level equivalents. For system and subsystem verification, which include the integration of multiple component models, higher level models improve the overall simulation time. It has been shown that a processor model at the fully functional level can operate over 1000 times faster than its gate level equivalent while maintaining clock cycle accuracy [5]. Verification also requires efficient techniques for test creation via automation and reuse and requirements compliance capture and test application via structured testbench development.

Interoperability addresses the ability of two models to communicate in the same simulation environment. Interoperability requirements are necessary because models usually developed by multiple design teams and from external vendors must be integrated to verify system functionality. Guidelines and potential standards for all abstraction levels within the design process must be defined when current descriptions do not exist. In the area of fully functional and RTL modeling, current practice is to use IEEE Std. 1164 – 1993 nine-valued logic packages [15]. Performance modeling standards are an ongoing effort of the RASSP program.

Fidelity addresses the problem of defining the information captured by each level of abstraction within the top-down design process. The importance of defining the correct fidelity lies in the fact that information not relevant within a model at a particular stage in the hierarchy requires unnecessary simulation time. Relevant information must be captured efficiently so simulation times improve as one moves toward the top of the design hierarchy. Figure 78.3 describes the RASSP taxonomy [16] for accomplishing this objective. The diagram illustrates how a VHDL model can be described using five resolution axes; temporal, data value, functional, structural, and programming level. Each line is continuous and discrete labels are positioned to illustrate various levels ranging from high to low resolution. A full specification of a model's fidelity requires two charts, one to describe the internal attributes of the model and the second for the external attributes. An "*X*" through a particular axis implies the model contains no information on the specific resolution. A compressed textual representation of this figure will be used throughout the remainder of the paper. The information is captured in a 5-tuple as follows,

{(Temporal Level), (Data Value), (Function), (Structure), (Programming Level)}

The temporal axis specifies the time scale of events in the model and is analogous to precision as distinguished from accuracy. At one extreme, for the case of purely functional models, no time is modeled. Examples include Fast Fourier Transform and FIR filtering procedural calls. At the other extreme, time resolutions are specified in gate propagation delays. Between the two extremes, models may be time accurate at the clock level for the case of fully functional processor models, at the instruction cycle level for the case of performance level processor models, or at the system level for the case of application graph switching. In general, higher resolution models require longer simulation times due to the increased number of event transactions.

The data value axis specifies the data resolution used by the model. For high resolution models, data is represented with bit true accuracy and is commonly found in gate level models. At the low end of the spectrum, data is represented by abstract token types where data is represented by enumerated values, for example, *blue*. Performance level modeling uses tokens as its data type. The token only captures the

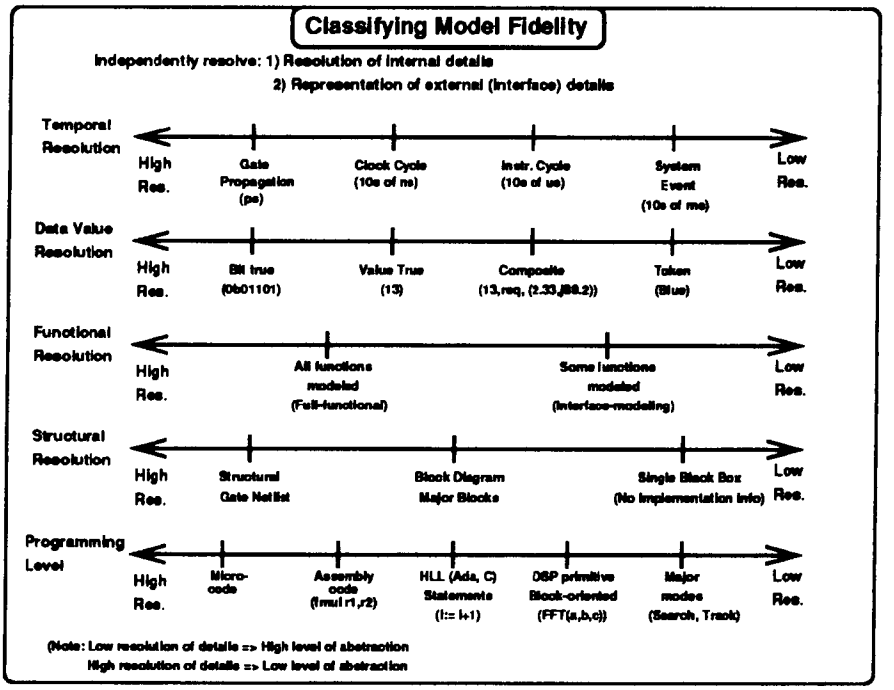

FIGURE 78.3 A model fidelity classification scheme.

control information of the system and no actual data. For the case of no data, the axis would be represented with an "*X*". At intermediate levels, data is represented with its correct value but at a higher abstraction (i.e., integer or composite types, instead of the actual bits). In general, higher resolutions require more simulation time.

Functional resolution specifies the detail of device functionality captured by the model. At one extreme, no functions are modeled and the model represents the processing functionality as a simple time delay (i.e., no actual calculations are performed). At the high end, all the functions are implemented within the model. As an example, for a processor model, a time delay is used to represent the execution of a specific software task at low resolutions while the actual code is executed on the model for high resolution simulations. As a rule of thumb, the more functions represented, the slower the model executes during simulation.

The structural axis specifies how the model is constructed from its constituent elements. At the low end, the model looks like a black box with inputs and outputs but no detail as to the internal contents. At the high end the internal structure is modeled with very fine detail, typically as a structural net list of lower level components. In the middle, the major blocks are grouped according to related functionality.

The final level of detail needed to specify a model is its programmability. This describes the granularity at which the model interprets software elements of a system. At one extreme, pure hardware is specified and the model does not interpret software, for example, a special purpose FFT processor hard wired for 1024 samples. At the other extreme, the internal micro-code is modeled at the detail of its datapath control. At this resolution, the model captures precisely how the micro-code manipulates the datapath elements. At decreasing resolutions the model has the ability to process assembly code and high level languages as input. At even lower levels, only DSP primitive blocks are modeled. In this case, programming consists of combining functional blocks to define the necessary application. Tools such as MATLAB/Simulink provide examples for this type of model granularity. Finally, models can be programmed at the level of

the major modes. In this case, a run-time system is switched between major operating modes of a system by executing alternative application graphs.

Finally, efficiency issues are addressed at each level of abstraction in the design flow. Efficiency will be discussed in coordination with the issues of fidelity where both the model details and information content are related to improving simulation speed.

78.4 The Executable Requirement

The methodology for developing signal processing systems begins with the definition of the system requirement. In the past, common practice was to develop a textual specification of the system. This approach is flawed due to the inherent ambiguity of the written description of a complex system. The new methodology places the requirements in an executable format enforcing a more rigorous description of the system. Thus, VHDL's first application in the development of a signal processing system is an *executable requirement* which may include signal transformations, data format, modes of operation, timing at data and control ports, test capabilities, and implementation constraints [17]. The executable requirement can also define the minimum required unit of development in terms of performance (e.g., SNR, throughput, latency, etc.). By capturing the requirements in an executable form, inconsistencies and missing information in the written specification can also be uncovered during development of the requirements model.

An executable requirement creates an "environment" wherein the surroundings of the signal processing system are simulated. Figure 78.4 illustrates a system model with an accompanying testbench. The testbench generates control and data signals as stimulus to the system model. In addition, the testbench receives output data from the system model. This data is used to verify the correct operation of the system model. The advantages of an executable requirement are varied. First, it serves as a mechanism to define

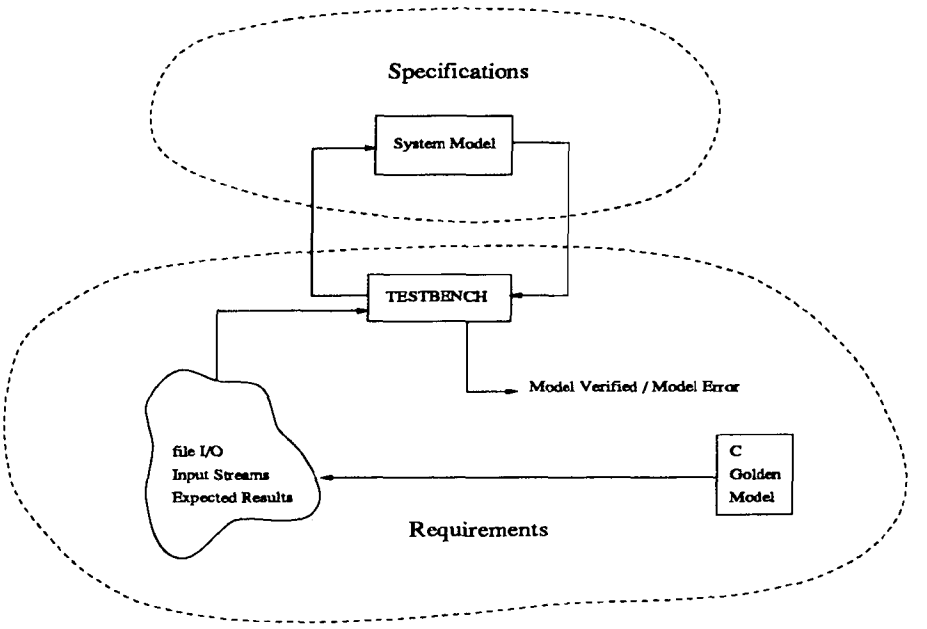

FIGURE 78.4 Illustration of the relation between executable requirements and specifications.

and refine the requirements placed on a system. Also, the VHDL source code along with supporting textual description becomes a critical part of the requirements documentation and life cycle support of the system. In addition, the testbench allows easy examination of different command sequences and data sets. The testbench can also serve as the stimulus for any number of designs. The development of different system models can be tested within a single simulation environment using the same testbench. The requirement is easily adaptable to changes that can occur in lower levels of the design process. Finally, executable requirements are formed at all levels of abstraction and create a documented history of the design process. For example, at the system level, the environment may consist of image data from a camera while at the ASIC level it may be an interface model of another component.

The RASSP program, through the efforts of MIT Lincoln Laboratory, created an executable requirement [18] for a synthetic aperture radar (SAR) algorithm and documented many of the lessons learned in implementing this stage in the top-down design process. Their high level requirements model served as the baseline for the design of two SAR systems developed by separate contractors, Lockheed Sanders and Martin Marietta Advanced Technology Labs. A test bench generation system for capturing high level requirements and automating the creation of VHDL is presented in [19]. In the following sections, we present the details of work done at Georgia Tech in creating an executable requirement and specification for an MPEG-1 decoder.

78.4.1 An Executable Requirements Example: MPEG-1 Decoder

MPEG-1 is a video compression-decompression standard developed under the International Standard Organization originally targeted at CD-ROMs with a data rate of 1.5 Mbits/sec [20]. MPEG-1 is broken into 3 layers: system, video, and audio. Table 78.1 depicts the system clock frequency requirement taken from layer 1 of the MPEG-1 document.[1] The system time is used to control when video frames are decoded and presented via decoder and presentation time stamps contained in the ISO 11172 MPEG-1 bitstream. A VHDL executable rendition of this requirement is illustrated in Table 78.5.

TABLE 78.1 MPEG-1 System Clock Frequency Requirement Example

Layer 1 - System requirement example from ISO 11172 standard	
System clock frequency	The value of the system clock frequency is measured in Hz and shall meet the following constraints: $90,000 - 4.5$ Hz $\leq$ system_clock_frequency $\leq 90,000 + 4.5$ Hz Rate of change of system_clock_frequency $\leq 250 * 10^{-6}$ Hz/s

The testbench of this system uses an MPEG-1 bitstream created from a "golden C model" to ensure correct input. A public-domain C version of an MPEG encoder created at UCal-Berkeley [21] was used as the golden C model to generate the input for the executable requirement. From the testbench, an MPEG bitstream file is read as a series of integers and transmitted to the MPEG decoder model at a constant rate of 174300 Bytes/sec along with a system clock and a control line named *mpeg_go* which activates the decoder. Only 50 lines of VHDL code are required to characterize the top level testbench. This is due to the availability of the golden C MPEG encoder and a shell script which wraps around the output of the golden C MPEG encoder bitstream with system layer information. This script is necessary because there are no *complete* MPEG software codecs in the public domain, i.e., they do not include the system information in the bitstream. Figure 78.6 depicts the process of verification using golden C models. The golden model generates the bitstream sent to the testbench. The testbench reads the bitstream as a series

[1] Our efforts at Georgia Tech have only focused on layers 1 and 2 of this standard.

```
- system_time_clk process is a clock process that counts at a rate
- of 90kHz as per MPEG-I requirement. In addition, it is updated by
- the value of the incoming SCR fields read from the ISO11172 stream.
-
system_time_clock : PROCESS(stc_strobe,sys_clk)
     VARIABLE clock_count : INTEGER := 0;
     VARIABLE SCR,system_time_var : bit33;
     CONSTANT clock_divider : INTEGER := 2;
BEGIN
     IF mpeg_go = '1' THEN
     - if stc_strobe is high then update system_time value to latest SCR
     IF (stc_strobe = '1') AND (stc_strobe'EVENT) THEN
          system_time <= system_clock_ref;
          clock_count := 0; - reset counter used for clock downsample
     ELSIF (sys_clk = '1') AND (sys_clk'EVENT) THEN
          clock_count := clock_count + 1;
          IF clock_count MOD clock_divider = 0 THEN
               system_time_var := system_time + one;
               system_time <= system_time_var;
          END IF;
     END IF;
     END IF;
END PROCESS system_time_clock;
```

FIGURE 78.5 System clock frequency requirement example translated to VHDL.

of integers. These are in turn sent as data into the VHDL MPEG decoder model driven with appropriate clock and control lines. The output of the VHDL model is compared with the output of the golden model (also available from Berkeley) to verify the correct operation of the VHDL decoder. A warning message alerts the user to the status of the model's integrity.

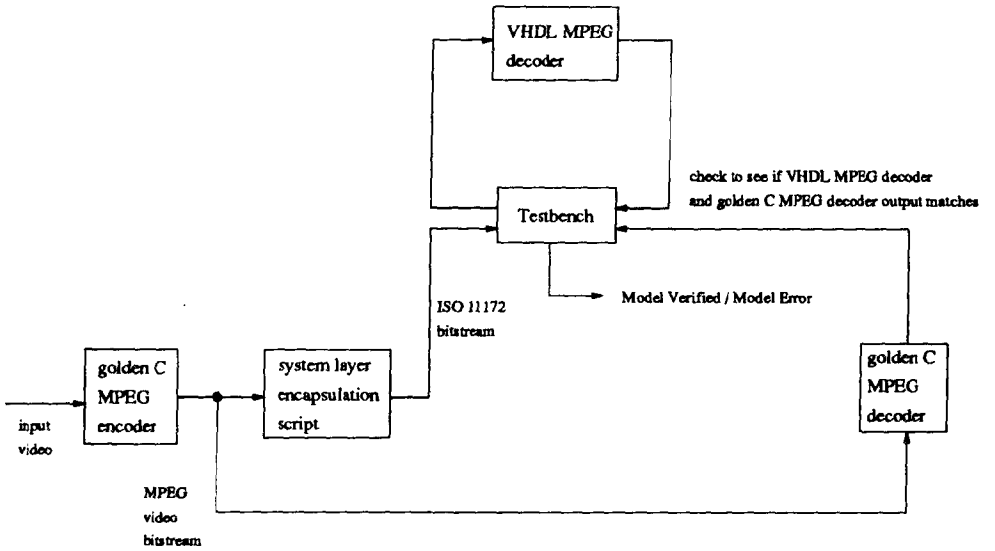

FIGURE 78.6 MPEG-1 decoder executable requirement.

The advantage of the configuration illustrated in Figure 78.6 is its reusability. An obvious example is MPEG-2 [22], another video compression-decompression standard targeted for the all-digital transmission of broadcast TV quality video at coded bit rates between 4 and 9 Mbits/sec. The same testbench structure could be used by replacing the golden C models with their MPEG-2 counterparts. While the system layer information encapsulation script would have to be changed, the testbench itself remains the same because the interface between an MPEG-1 decoder and its surrounding environment is identical to the interface for an MPEG-2 decoder. In general, this testbench configuration could be used for a wide class of video decoders. The only modifications would be the golden C models and the interface between the VHDL

decoder model and the testbench. This would involve making only minor alterations to the testbench itself.

78.5 The Executable Specification

The executable specification depicted in Fig. 78.4 processes and responds to the outside stimulus, provided by the executable requirement, through its interface. It reflects the particular function and timing of the intended design. Thus, the executable specification describes the behavior of the design and is timing accurate without consideration of the eventual implementation. This allows the user to evaluate the completeness, logical correctness, and algorithmic performance of the system through the test bench. The creation of this formal specification helps identify and correct functional errors at an early stage in the design and reduce total design time [13, 16, 23, 24].

The development of an executable specification is a complex task. Very often, the required functionality of the system is not well-understood. It is through a process of learning, understanding, and defining that a specification is crystallized. To specify system functionality, we decompose it into elements. The relationship between these elements is in terms of their execution order and the data passing between them. The executable specification captures:

- the refined internal functionality of the unit under development (some algorithm parallelism, fixed/floating point bit level accuracies required, control strategies, functional breakdown, task execution order)
- physical constraints of the unit such as size, weight, area, and power
- unit timing and performance information (I/O timing constraints, I/O protocols, computational complexity)

The purpose of VHDL at the executable specification stage is to create a formalization of the elements in a system and their relationships. It can be thought of as the high level design of the unit under development. And although we have restricted our discussion to the system level, the executable specification may describe any level of abstraction (algorithm, system, subsystem, board, device, etc.).

The allure of this approach is based on the user's ability to see what the performance "looks" like. In addition, a stable test mechanism is developed early in the design process (note the complementary relation between the executable requirement and specification). With the specification precisely defined, it becomes easier to integrate the system with other concurrently designed systems. Finally, this executable approach facilitates the re-use of system specifications for the possible redesign of the system.

In general, when considering the entire design process, executable requirements and specifications can potentially cover any of the possible resolutions in the fidelity classification chart. However, for any particular specification or requirement, only a small portion of the chart will be covered. For example, the MPEG decoder presented in this and the previous section has the fidelity information represented by the 5-tuple below,

$$\begin{aligned}\text{Internal: } &\{(\text{Clock cycle}), (\text{Bit true} \rightarrow \text{Value true}), (\text{All}), (\text{Major blocks}), (\text{X})\}\\ \text{External: } &\{(\text{Clock cycle}), (\text{Value true}), (\text{Some}), (\text{Black box}), (\text{X})\},\end{aligned}$$

where (Bit true → Value true) means all resolutions between bit true and value true inclusive.

From an internal viewpoint, the timing is at the system clock level, data is represented by bits in some cases and integers in others, the structure is at the major block level, and all the functions are modeled. From an external perspective, the timing is also at the system clock level, the data is represented by a stream of integers, the structure is seen as a single black box fed by the executable requirement and from an external perspective the function is only modeled partially because this does not represent an actual chip interface.

78.5.1 An Executable Specification Example: MPEG-1 Decoder

As an example, an MPEG-1 decoder executable specification developed at Georgia Tech will be examined in detail. Figure 78.7 illustrates how the system functionality was broken into a discrete number of elements. In this diagram each block represents a process and the lines connecting them are signals. Three major areas of functionality were identified from the written specification: memory, control, and the video decoder itself. Two memory blocks, *video_decode_memory* and *system_level_memory* are clearly labeled. The *present_frame_to_decode_file* process contains a frame reorder buffer which holds a frame until its presentation time. All other VHDL processes with the exception of *decode_video_frame_process* are control processes and pertain to the systems layer of the MPEG-1 standard. These processes take the incoming MPEG-1 bitstream and extract system layer information. This information is stored in the *system_level_memory* process where other control processes and the video decoder can access pertinent data. After removing the system layer information from the MPEG-1 bitstream, the remainder is placed in the *video_decode_memory*. This is the input buffer to the video decoder. It should be noted that although MPEG-1 is capable of up to 16 simultaneous video streams multiplexed into the MPEG-1 bitstream only one video stream was selected for simplicity.

The last process, *decode_video_frame_process*, contains all the subroutines necessary to decode the video bitstream from the video buffer (*video_decode_memory*). MPEG video frames are broken into 3 types: (I)ntra, (P)redictive, and (B)idirectional. I frames are coded using block discrete cosine transform (DCT) compression. Thus, the entire frame is broken into 8x8 blocks, transformed with a DCT and the resulting coefficients transmitted. P frames use the previous frame as a prediction of the current frame. The current frame is broken into 16 × 16 blocks. Each block is compared with a corresponding search window (e.g., 32 × 32, 48 × 48) in the previous frame. The 16 × 16 block within the search window which best matches the current frame block is determined. The motion vector identifies the matching block within the search window and is transmitted to the decoder. B frames are similar to P frames except a previous frame and a future frame are used to estimate the best matching block from either of these frames or an average of the two. It should be noted that this requires the encoder and decoder to store these 2 reference frames.

The functions contained in the *decode_video_frame_process* are shown in Fig. 78.8. In the diagram, there are three main paths representing the procedures or functions in the executable specification which process the I, P, or B frame, respectively. Each box below a path encloses all the procedures executed from within that function. Beneath each path is an estimate of the number of computations required to process each frame type. Comparing the three executable paths in this diagram, one observes the large similarity between each path. Overall, only 25 unique routines are called to process the video frame. By identifying key functions within the video decoding algorithm itself, efficient and reusable code can be created. For instance, the data transmitted from the encoder to the decoder is compressed using a Huffman scheme. The procedures *vlc*, *advance_bit*, and *extract_n_bits* perform the Huffman decode function and miscellaneous parsing of the MPEG-1 video bitstream. Thus, this set of procedures can be used in each frame type execution path. Reuse of these procedures can be applied in the development of an MPEG-2 decoder executable specification. Since MPEG-2 is structured as a super set of the syntax defined in MPEG-1, there are many procedures that can be utilized with only minor modifications. Other procedures such as *motion_compensate_forward* and *idct* can be reused in a variety of DCT-based video compression algorithms.

The executable specification also allows detailed analysis of the computational complexity on a procedural level. Table 78.2 lists the computational complexity of some of the procedures identified in Fig. 78.8. This breakdown identifies what areas of the algorithm are the most computationally intensive and the numbers were arrived at through a data flow analysis of the VHDL code. Within the MPEG-1 video decoder algorithm, the most intense computational loads occur in the inverse DCT and motion compensation procedures. Thus, such an analysis can alert the user early in the design process to potential design issues. While parallelism is a logical topic for the data and control flow modeling section, preliminary

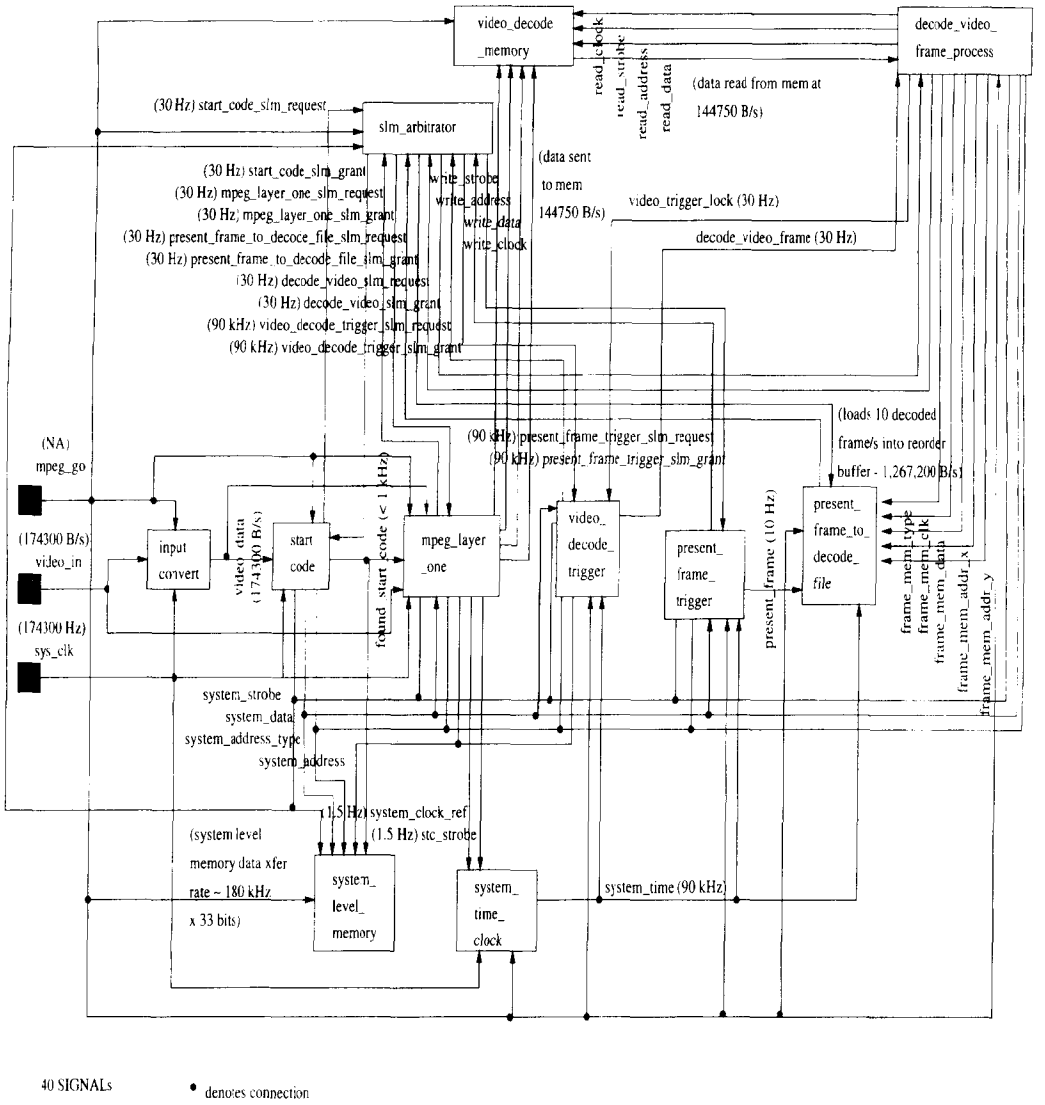

FIGURE 78.7 System functionality breakdown for MPEG-1 decoder.

investigations can be made from the executable specification itself. With the specifications captured in a language, execution order and data passing between procedures are known precisely. This knowledge facilitates the user in extracting potential parallelism from the specification. From the MPEG-1 decoder executable specification, potential parallelism can be seen in several areas. In an I frame, no data dependencies are present between each 8 × 8 block. Therefore, an inverse DCT could potentially be performed on each 8 × 8 block in parallel. In P and B frames, data dependencies occur between consecutive 16 × 16 blocks (called macroblocks) but no data dependencies occur between slices (a grouping of consecutive macroblocks). Thus, parallelism is potentially exploitable at the slice and macroblock level. This information is passed to the data/control flow modeling phase where more detailed analysis of parallelism is done.

It is also possible to delve into implementation requirement issues at the executable specification level. Fixed vs. floating point trade-offs can be examined in detail. The necessary accuracy and resolution

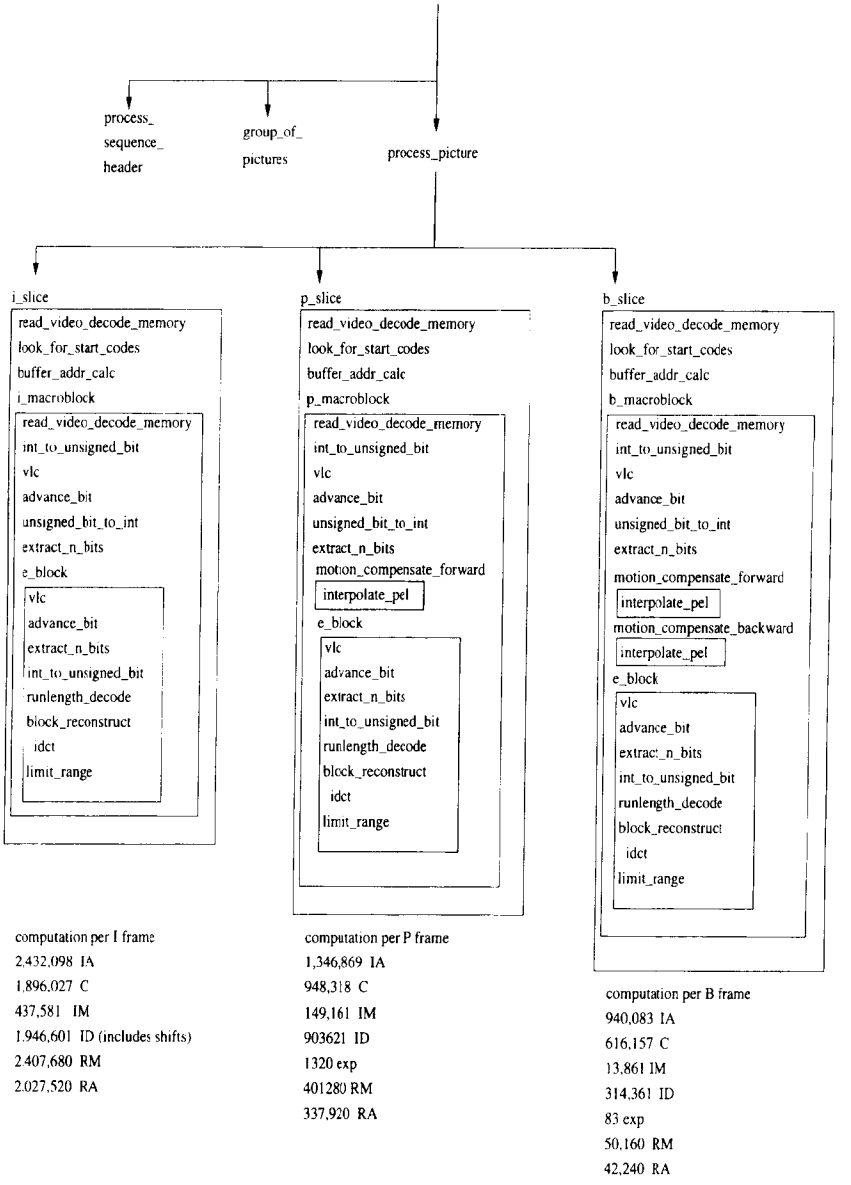

FIGURE 78.8 Description of procedural flow within MPEG-1 decoder executable specification.

required to meet system requirements can be determined through the use of floating and fixed point packages written in VHDL. At Georgia Tech, fixed point packages have been developed. These packages allow the user to experiment with the executable specification and see the effect finite bit accuracy has on the system model. In addition, packages have been developed which implement specific arithmetic architectures such as the ADSP 2100 [25]. This analysis results in additional design requirements being passed to hardware and software developers in later design phases.

Finally, the executable specification allows the explicit capture of internal timing and control flow requirements of the MPEG-1 decoding algorithm itself. The written document is imprecise about the details of how timing considerations for presentation and decoder time stamps will be handled. The control necessary to trigger present and decode video frame events is difficult to articulate in a written

TABLE 78.2 Computational Complexity of Some Specification Procedures

Procedure	Int Adds	Int Div	Comp	Int Mult	exp	Real Add	Real Mult
vlc	—	-	2	-	-	-	-
advance_bit	10	16	9	-	-	-	-
int_to_unsigned_bit	8	16	8	-	-	-	-
extract_n_bits	24	16	20	-	-	-	-
look_for_start_codes	9	16	10	-	-	-	-
runlength_decode	2	-	1	1	-	-	-
block_reconstruct	66	64	258	193	-	-	-
idct	-	-	-	-	-	1024	1216
qmotion_compensate_forward	1422	646	1549	16	-	-	-

form. The most difficult aspects of coding the executable specification for a MPEG-1 decoder were these considerations. The decoder itself hinges on developing a mechanism for robustly determining when to decode or present a frame in the buffer. Events must be triggered using a system time clock which is updated from the input bitstream itself. This task is handled by five processes (*start_code*, *mpeg_layer_one*, *video_decode_trigger*, *present_frame_trigger*, *present_frame_to_decode_file*) grouped around a common memory (*system_level_memory*). This memory was necessary to allow each concurrent process to access timing information extracted from the system layer of the input bitstream. These timing and control considerations had to fit into a larger system timing requirement. For a MPEG-1 decoder, the most critical timing constraints are initial latency and the fixed presentation rate (e.g., 30 frames/sec). All other timing considerations were driven by this requirement.

78.6 Data and Control Flow Modeling

This modeling level captures data and control flow information in the system algorithms. The objective of data flow modeling is to refine the functional descriptions in the executable specification and capture concurrency information and data dependencies inherent in the algorithm. The output of the refinement process is one or a few manually generated implementation independent representations of the algorithm. These multiple implementations capture potential algorithmic parallelism at a primitive level where primitives are defined as that set of functions contained in a design library. The primitives are signal processing functions such as Fast Fourier Transforms or filter routines at coarse-grained levels to adders and multipliers at more fine-grained levels. The breakdown of primitive elements depend on the granularity exploited by the algorithm as well as potential architectural design paradigms to which the algorithm is mapped. For example, if the design paradigm demands architectures using multiple commercial-off-the-shelf (COTS) RISC processors, the primitives consist of signal processing functional block level elements such as FFTs or FIR filters which exist as performance optimized library elements available for the specific processor. For custom computationally intense designs, the data flow of the algorithm may be dissected into lower primitive components such as adders and multipliers using bit-slice architectures. In our design flow, the fidelity captured by data/control flow models is shown below:

Internal: {(X), (Value true → Composite), (All), (X), (Major modes)}
External: {(X), (Value true → Composite), (X), (X), (X)}.

Because the models are purely functional and their major objective is to refine the internal representation of the algorithm, there is no time information captured by its internal or external representation as illustrated by the "X". The internal data processed by the model and external data loaded into the model are typically represented by standard data types such as *float* and/or *integer* and in some cases by composite data types such as records or arrays. All internal functionality is represented and is verified using the same data presented to the executable specification. No function is captured via external interfaces since data is input to the model through file input/output. The data processed by the executable specification is also processed by the data/control flow model. No internal or external structural information is captured since the model

is implementation independent. Its level of programmability is represented at the application graph level. The applications are major modes of the system under investigation and hence at a low resolution. In general, because the primitive elements can represent adders and/or multipliers, programmability for data/control flow models can resolve to higher resolutions including the microcode level.

The implementation independent representations are compared with the executable specification using the test data supplied by the requirements development phase to verify compliance with the original algorithm design. The representations are then input to the architecture selection phase and, with additional metrics, determine the final architecture of the system.

Signal processing applications inherently follow the data flow execution model. Processing Graph Methodology (PGM) [26] from Naval Research Laboratory was developed specifically to capture signal processing applications. PGM supports specification of full system data flow and its associated control. An application is first captured as a graph, where nodes of the graph represent processing and edges represent queues that hold intermediate data between nodes. The scheduling criteria for each node is based on the state of its corresponding input/output queues. Each queue in the graph can be linked to one node at a time. Associated with each queue is a control block structure containing information such as size, current amount of data, and threshold. A run-time system provides a set of procedures used by each node to check the availability of data from the upstream queue or available space in the downstream queue. Applications consist of one or more graphs, one or more I/O procedures, and a run-time system interfaced with one or more command programs. The PGM graphs serve as the implementation independent representation of the algorithm discussed earlier. An example of a 2-D FFT PGM graph is presented in the next section.

Under the support of the RASSP program, a set of tools is being developed by Management Communications and Control, Inc. (MCCI) and Lockheed Martin Advance Technology Laboratories [27, 28]. The toolset automates the translation of software architecture specifications to design implementations of application and control software for a signal processing system. Hardware/software architectures are presented to the autocoding toolset as PGM application data flow graphs along with a candidate architectures file and graph partition lists. The lists are generated by hardware/software partitioning tools. The proposed partitions are then simulated for performance and verified against the top level specification for correct functionality. The verified partition graphs are then used as inputs to detailed design level autocode tools that generate actual source code. The source code implements the partitions processing specifications using the target processor's math library. It also produces a memory map converting all queues and variables to static buffers. Finally the application graph, with its set of source files, are translated to run-time data structures that are used by the run-time system to create an executable image of the application as distributed tasks on the target processors.

Other tools provide paths from specification to hardware and are briefly mentioned. The Ptolemy [29, 30] design system from the University of California at Berkeley provides a synchronous data flow domain which can be used to perform system level simulations. Silage, another product of UC Berkeley is a data flow modeling language. Data Flow Language (DFL), a commercial version of Silage is used in Mentor Graphics' DSP Station to perform algorithm/architecture tradeoffs. It also provides a path to synthesis as a high-level design entry tool.

78.6.1 Data and Control Flow Example

An example of a small PGM application is presented in Fig. 78.9. The graph represents a two dimensional FFT program implemented in PGM. The graph captures both the functionality and the data flow aspects of the application. The source data is read from a file and represents the I/O processor that would normally provide the input data stream. The data are then distributed to a number of queues serving as inputs to the FFT primitives that perform the operations on the rows of the input stream. The output of the FFT primitives flow to another set of queues that are input to the corner turn graph. Once the data are sorted correctly, they are sent to the input queues of the column FFT primitives. The graph is then executed by the simulator where the functionality, queue sizes, and communication between nodes are examined. This

same graph is input to the hardware/software partitioning tools that generate the partition list. Given the partition list and the hardware configuration file, the autocode tool set generates the load image for the target platform.

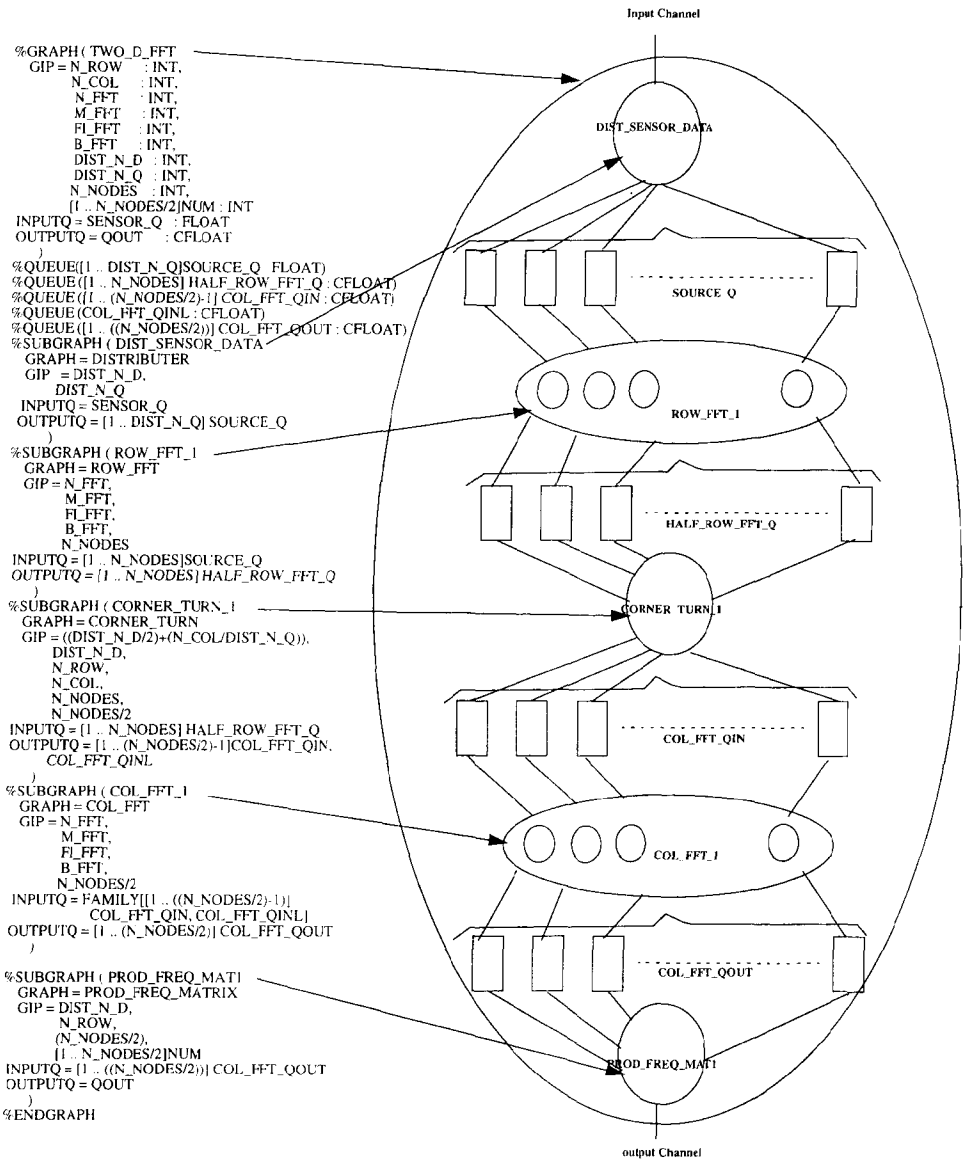

FIGURE 78.9 Example PGM application graph.

78.7 Architectural Design

Signal processing systems are characterized as having high throughput requirements as well as stringent physical constraints. However, due to economic objectives, signal processing systems must also be de-