**Center for Embedded Computer Systems**
**University of California, Irvine**

# Specification and Design of a MP3 Audio Decoder

Pramod Chandraiah, Rainer Dömer

Technical Report CECS-05-04
May 5, 2005

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

pramodc@uci.edu, doemer@uci.edu
http://www.cecs.uci.edu/

# Specification and Design of a MP3 Audio Decoder

Pramod Chandraiah, Rainer Dömer

Technical Report CECS-05-04
May 5, 2005

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

pramodc@uci.edu, doemer@uci.edu
http://www.cecs.uci.edu

**Abstract**

*In an effort to understand, experience and prove the benefits of automated SoC design, this report describes the specification modeling, design space exploration and implementation of a real world example using SpecC based System on Chip Environment (SCE). The report covers a complete description of developing the specification model of a MPEG-1 Layer 3 (MP3) audio decoder in SpecC language and the subsequent design space exploration and implementation using SCE. This report also attempts to improve the SoC design process by identifying the tasks in specification modeling that can be automated.*

# Contents

**1 Introduction** **3**
1.1 Challenges of SoC Design . . . 3
1.2 Specification Modeling and SpecC . . . 4
1.3 SoC Design Methodology . . . 5
1.3.1 Architecture Exploration and Refinement . . . 6
1.3.2 Communication Exploration and Refinement . . . 6
1.3.3 Implementation Synthesis . . . 6
1.4 Related Work . . . 7
1.4.1 Design Methodologies . . . 7
1.4.2 Specification Languages . . . 7
1.4.3 SoC Design Flow Examples . . . 8
1.4.3.1 Design Exploration and Implementation of Digital Camera . . . 8
1.4.3.2 Design Exploration and Implementation of Vocoder . . . 8
1.4.4 Our Work . . . 9

**2 Design Example** **10**
2.1 Description of MP3 Decoder . . . 10
2.1.1 Structure of an MP3 Audio Frame . . . 10
2.1.2 MP3 Decoder Operation . . . 12

**3 Specification Model** **14**
3.1 Reference C Implementation of MP3 Decoder . . . 15
3.1.1 Properties of the Source of Reference Implementation . . . 15
3.2 Initial Testbench . . . 16
3.2.1 Making C Code SpecC Compliant . . . 16
3.2.2 Building the Testbench . . . 17
3.2.3 Timing of the Testbench . . . 17
3.3 Parallelization of the Design at the Top Level of the Hierarchy . . . 20
3.4 Introducing Granularity . . . 21
3.4.1 Procedure . . . 21
3.4.2 Summary . . . 22
3.5 Elimination of Global Variables . . . 23
3.5.1 Procedure 1 . . . 23
3.5.2 Procedure 2 . . . 23
3.5.3 Summary . . . 24
3.6 Arriving at a Clean Specification Model . . . 25
3.6.1 Procedure . . . 25
3.6.2 Summary . . . 26
3.7 Introducing Concurrency in the Specification Model . . . 29
3.7.1 Conditions for Concurrency . . . 29
3.7.2 Conditions for Pipelined Concurrency . . . 30

3.7.3 Procedure for Introducing Concurrency . . . 31
3.7.4 Procedure for Introducing Pipeline Concurrency . . . 35
3.7.5 Summary . . . 39
3.8 Summary and Conclusions . . . 39

**4 Design Space Exploration and Implementation** **41**
4.1 Complete Software Solution . . . 42
4.2 Hardware-Software Solution-1 . . . 42
4.2.1 Hardware-Software Partitioning-1: Architecture Refinement . . . 42
4.2.2 Hardware-Software Partitioning-1: Communication Refinement . . . 43
4.2.3 Hardware-Software Partitioning-1 : Implementation Synthesis . . . 43
4.3 Hardware-Software Solution-2 . . . 43
4.3.1 Hardware-Software Partitioning-2: Architecture Refinement . . . 44
4.3.2 Hardware-Software Partitioning-2: Communication Refinement . . . 44
4.3.3 Hardware-Software Partitioning-2 : Implementation Synthesis . . . 44
4.4 Hardware-Software Solution-3 . . . 44
4.4.1 Hardware-Software Partitioning-3: Architecture Refinement . . . 45
4.4.2 Hardware-Software Partitioning-3: Communication Refinement . . . 45
4.4.3 Hardware-Software Partitioning-3: Implementation Synthesis . . . 45
4.5 Summary and Conclusions . . . 46

**5 Experimental Results** **53**
5.1 Functionality Verification . . . 53
5.1.1 Test Suite . . . 53
5.2 Timing Verification . . . 53

**6 Summary and Conclusions** **57**

**References** **59**

# List of Figures


1 Abstraction levels in SOC design [13] . . . 3
2 SOC design methodology [13] . . . 5
3 MPEG 1 Layer 3 frame format . . . 11
4 Block diagram of MP3 decoder [18] . . . 12
5 Call graph of major functions in the reference C implementation . . . 16
6 Top level testbench . . . 18
7 Timing of testbench . . . 19
8 Top level parallel hierarchy of the decoder . . . 20
9 Example describing conversion of a C function to a SpecC behavior . . . 22
10 Example describing conversion of unclean behavior to a clean behavior . . . 26
11 Example describing conversion of a FOR statement into FSM . . . 29
12 Hierarchy within DoLayer3 behavior in the MP3 decoder specification model . . . 32
13 Example showing the conversion of a sequential behavior into concurrent behavior 33
14 Parallelsm in the MP3 decoder specification model . . . 34
15 Relative computation complexity of the three most compute intensive behaviors of MP3 decoder specification model . . . 35
16 Pipelining in the MP3 decoder specification model . . . 37
17 Relative computation complexity of 4 most compute intensive behaviors after pipelining the synthesis filter behavior . . . 38
18 Hardware-software partitioning-1: Architecture model of MP3 decoder . . . 47
19 Hardware-software partitioning-1: Communication model of MP3 decoder . . . 48
20 Hardware-software partitioning-2: Architecture model of MP3 decoder . . . 49
21 Hardware-software partitioning-2: Communication model of MP3 decoder . . . 50
22 Relative computation complexity of the few behaviors of MP3 decoder specification model . . . 50
23 Hardware-software partitioning-3: Architecture model of MP3 decoder (before scheduling refinement) . . . 51
24 Hardware-software partitioning-3:Communication model of MP3 decoder . . . 52

## List of Acronyms

**Behavior** An entity that encapsulates and describes computation or functionality in the form of an algorithm.

**CAD** Computer Aided Design. Design of systems with the help of and assisted by computer programs, i.e. software tools.

**CE** Communication Element. A system component that is part of the communication architecture for transmission of data between PEs, e.g. a transducer, an arbiter, or an interrupt controller.

**Channel** An entity that encapsulates and describes communication between two or more partners in an abstract manner.

**DUT** Design Under Test

**FSM** Finite State Machine. A model that describes a machine as a set of states, a set of transitions between states, and a set of actions associated with each state or transition.

**FSMD** Finite State Machine with Datapath. An FSM in which each state contains a set of expressions over variables.

**GUI** Graphical User Interface. A graphical interface of a computer program that allows visual entry of commands and display of results.

**HDL** Hardware Description Language. A language for describing and modeling blocks of hardware.

**HW** Hardware. The tangible part of a computer system that is physically implemented.

**IP** Intellectual Property. An IP component is a pre-designed system component that is stored in the component database.

**OS** Operating System. A piece of software between hardware and application software that manages and controls functionality in a computer system.

**PE** Processing Element. A system component that performs computation (data processing), e.g. a software processor, a custom hardware component, or an IP.

**RTL** Register-Transfer Level. A level of abstraction at which computation is described as transfers of data between storage units (registers) where each transfer involves processing and manipulation of data.

**RTOS** Real-Time Operating System. An operating system that provides predictable timing and timing guarantees.

**SCE** SoC Environment. Tool set for automated, computer-aided design of SoC and computer systems in general.

**SLDL** System-Level Design Language. A language for describing complete computer systems consisting of both hardware and software components at high levels of abstraction.

**SoC** System-On-Chip. A complete computer system implemented on a single chip or die.

**TLM** Transaction Level Model. A model of a system in which communication is abstracted into channels and described as transactions at a level above pins and wires.

**VHDL** VHSIC Hardware Description Language. An HDL commonly used for hardware design at RTL and logic levels.

**VHSIC** Very High Speed Integrated Circuit.

## List of MP3-Specific Terms

**Alias** Mirrored signal component resulting from sub-Nyquist sampling.

**Bitrate** The rate at which the compressed bitstream is delivered from the storage medium to the input of a decoder.

**Channel** The left and right channels of a stereo signal.

**CRC** Cyclic Redundancy Code. Codes used to detect the transmission errors in the bit stream.

**Filterbank** A set of band-pass filters covering the entire audio frequency range.

**Frame** A part of the audio signal that corresponds to audio PCM samples from an Audio Access Unit.

**Granules** 576 frequency lines that carry their own side information.

**Huffman Coding** A specific method for entropy coding.

**IMDCT** Inverse Modified Discrete Cosine Transform

**Intensity stereo** A method of exploiting stereo irrelevance or redundancy in stereophonic audio programmes based on retaining at high frequencies only the energy envelope of the right and left channels.

**Joint stereo coding** Any method that exploits stereophonic irrelevance or stereophonic redundancy.

**MP3** MPEG Audio Layer-3

**MS stereo** A method of exploiting stereo irrelevance or redundancy in stereophonic audio programmes based on coding the sum and difference signal instead of the left and right channels.

**Polyphase filterbank** A set of equal bandwidth filters with special phase interrelationships, allowing for an efficient implementation of the filterbank.

**Requantization** Decoding of coded subband samples in order to recover the original quantized values.

**Scale factor band** A set of frequency lines in Layer III which are scaled by one scalefactor.

**Scale factor** Factor by which a set of values is scaled before quantization.

**Side information** Information in the bitstream necessary for controlling the decoder.

**Synthesis filter bank** Filterbank in the decoder that reconstructs a PCM audio signal from subband samples.

# Specification and Design of a MP3 Audio Decoder

**P. Chandraiah, R. Dömer**

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA

pramodc@uci.edu, doemer@uci.edu
http://www.cecs.uci.edu

## Abstract

*In an effort to understand, experience and prove the benefits of automated SoC design, this report describes the specification modeling, design space exploration and implementation of a real world example using SpecC based System on Chip Environment (SCE). The report covers a complete description of developing the specification model of a MPEG-1 Layer 3 (MP3) audio decoder in SpecC language and the subsequent design space exploration and implementation using SCE. This report also attempts to improve the SoC design process by identifying the tasks in specification modeling that can be automated.*

# 1 Introduction

In this report, we describe the system level design process adopted to design a MP3 Audio decoder. We adopted the SpecC design methodology and developed a specification model of a MP3 audio decoder in SpecC language and used the System On a Chip Environment (SCE) developed at Center for Embedded Computer Systems (CECS), to arrive at the final implementation of the design. First, we give a brief overview of SoC design challenges, followed by introduction to specification modeling and SpecC language and finally, we introduce SpecC based SoC design methodology.

## 1.1 Challenges of SoC Design

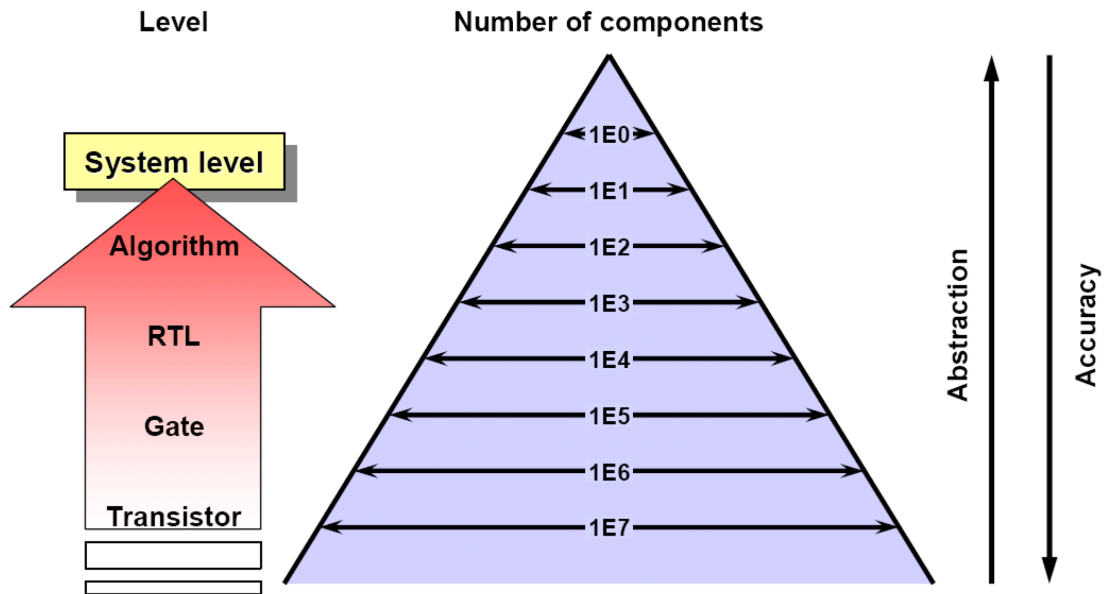



Figure 1: Abstraction levels in SOC design [13]

The system design process is elaborate and involves writing various models of the design at different levels of abstraction. Figure 1 shows the various abstraction levels. From the figure, we see an increase in the number of components and hence the complexity as we go lower in the level of abstraction. At the lowest level, an embedded system consists of millions of transistors. At Register-Transfer Level (RTL), the number of components reduces to thousands of components and finally, at the system level, the system is composed of very few components like general purpose processors, specialized hardware processors, memories and busses. The complexity of the system at the system level is far lesser than at the lower levels. However, the increase in the level abstraction is at the cost of reduced accuracy. For an embedded system designer, it is easier to handle the design

at the higher levels of abstraction. Writing and verifying each of these models is challenging and time consuming.
The goal of the SoC design methodology is to take an abstract system level description down to its real implementation using several refinement steps. The designer will specify the design using highly abstract specification model and using automation will arrive at an accurate implementation model. In the next section, we will introduce the specification modeling using SpecC language.

## 1.2 Specification Modeling and SpecC

The SoC design process starts from a highly abstract system level model called specification model. It is a pure functional, abstract model, and is free of any implementation detail. The model runs in zero simulation time and hence has no notion of time. It forms the input to architecture exploration, the first step in the system design process and hence forms the basis for all the future synthesis and exploration.

Specification models are written in System-Level Design Languages (SLDLs) [13]. Languages used to model complex systems consisting of hardware and software components are classified as SLDLs. Though it is possible to model designs in any of the programming languages, the choice of a good SLDL is a key in reducing the effort required in writing the specification model. A good SLDL provides native support to model both hardware and software concepts found in embedded system designs. A good SLDL provides native support to model concurrency, pipelining, structural hierarchy, interrupts and synchronization primitives. They also provide native support to implement computation models like Sequential, FSM, FSMD and so on, apart from providing all the typical features provided by other programming languages.

Following languages are popular choices for writing specification model: VHDL [9], Verilog [19], HardwareC [17], SpecCharts [28], SystemC [14], and SpecC [11]. VHDL and Verilog are primarily Hardware Description Languages (HDLs) and hence are not suitable to model software components. HardwareC is an HDL with C like syntax. It supports modeling hardware concepts but, lacks native support to model pipelined concurrency, timing and not suitable for modeling software components. SpecCharts is an extension of VHDL for system design and is oriented more towards hardware design and limited in terms of supporting complex embedded software. SystemC implements system level modeling concepts in the form of C++ library. It can model both hardware and software concepts and thus is a good candidate for system level design.
SpecC is another major candidate for system design. Being a true superset of ANSI-C, it has a natural suitability to describe software components. It has added features like signals, wait, notify etc. to support hardware description. It also includes constructs to support hierarchical description of system components. It also provides native support to describe parallel and pipeline execution. With all these features, the designer has the flexibility to choose and describe the system at any desired level of abstraction.
Apart from its capability, the easy availability of SpecC compiler and simulator and the SpecC based System design tool set, System on Chip Environment (SCE) made SpecC a obvious choice for developing our specification model.

In the next section, we will describe the SoC design methodology.

## 1.3 SoC Design Methodology

The SoC design methodology is shown in Figure 2. It tries to formalize individual refinements steps and gives the designer guidelines on how to handle efficiently the immense design space. The SoC design starts with the specification model that captures the algorithmic behavior

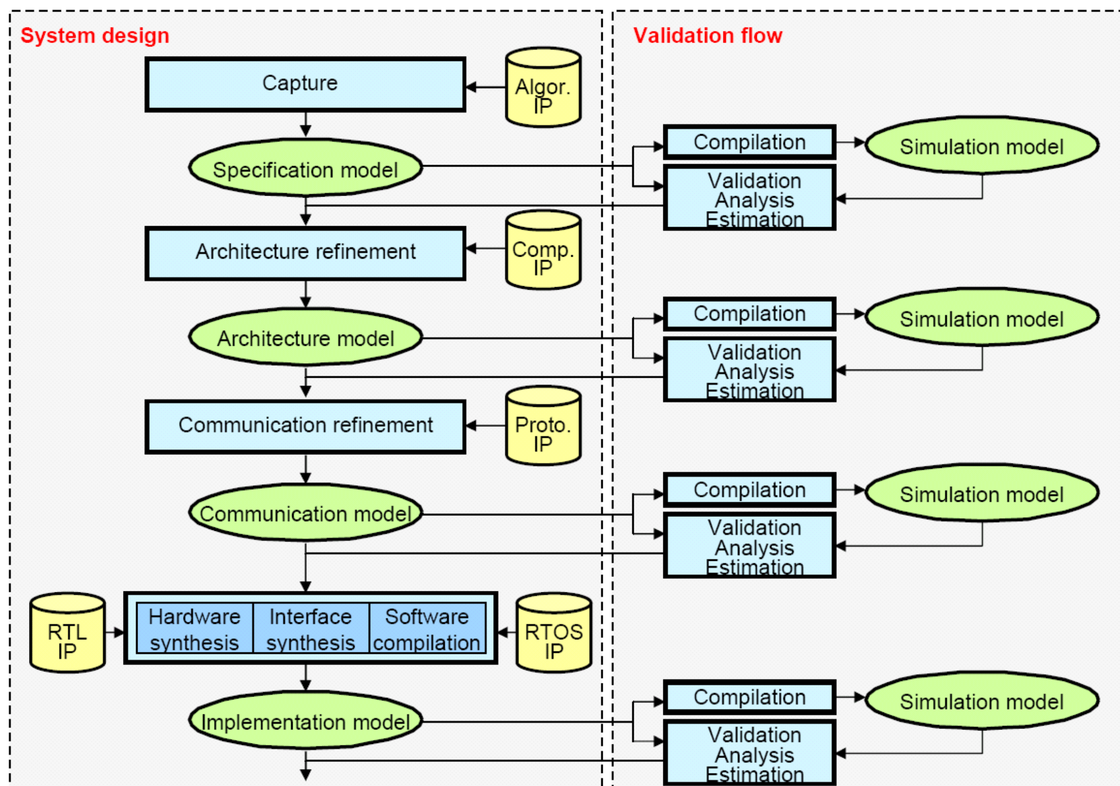



Figure 2: SOC design methodology [13]

and allows a functional validation of the description. The model is untimed, unless there are timing constraints introduced by the designer. Once the specification model is finished, it will serve as a golden model, to compare to during the design process. The specification modeling and the language used to capture the model were discussed in the previous section. In the following sections we will detail each of the refinement steps and the resulting model shown in Figure 2

### 1.3.1 Architecture Exploration and Refinement

Architecture exploration [23] determines the system architecture consisting of a set of Processing Elements (PEs). In this step, the behaviors of the specification model are mapped to the components of the system architecture. This process involves three major tasks, *Allocation*, *Partitioning* and *Scheduling*. Allocation, allocates SW, HW and memory components from the library. The decision of choosing a component is made by the designer. Partitioning divides the input system specification and maps them onto the allocated components. Also, the variables in the design are mapped onto the memory.
Scheduling, schedules the execution within hardware and software components. Partitioning and scheduling tasks are automated and require least designer interference. This process of architecture refinement results in an architecture model, in which all the computation blocks of the input specification are mapped to the system components. However, communication is still on an abstract level, and system components communicate via abstract channels.

### 1.3.2 Communication Exploration and Refinement

In communication exploration [2], abstract communication between components is refined into an actual implementation over wires and protocols of system busses. This design step can be further divided into three major tasks, *Bus allocation*, *Transducer insertion* and *Channel mapping*. During bus allocation, busses are allocated between PEs, and more often the main bus of the software processor is chosen as the system bus. Transducer insertion introduces transducer between busses of incompatible protocols (example, Parallel to Serial Protocol). During channel mapping, the abstract channels between components are mapped to allocated busses. The communication synthesis results in the bus functional model which defines the structure of the system architecture in terms of both components and connections. Just like the architecture model, bus functional model is fully executable and can be simulated and verified for proper functionality and timing.

### 1.3.3 Implementation Synthesis

Implementation synthesis takes the bus functional model as input and synthesizes the software and the hardware components. It is composed of two major independent tasks, *software synthesis* [29] and *hardware synthesis* [25]. The software synthesis task generates the machine code for the programmable processors in the architecture. As an intermediate step, the communication model is first translated to C language. Also, any concurrent tasks in the design will be dynamically scheduled by inserting a real time operating system. The resulting C code is compiled into machine codes of the processors using the cross compiler for the processor. The hardware synthesis task is performed using the classical behavior synthesis methods. This task can be divided into 3 sub-tasks, *allocation*, *binding*, and *scheduling*. *Allocation* is allocation of components like multiplexers, adders, registers. *Binding* binds the operations, data and data transfers to allocated components. *Scheduling* determines the order in which the operations are performed. The output of the hardware synthesis is a structural RTL description of the component. Implementation model is the result of both hardware and software synthesis and is the end result of the entire system level design.

## 1.4 Related Work

### 1.4.1 Design Methodologies

SoC design methodologies can be based on either top-down approach or bottom-up approach. In top-down approach, the design starts with the specification of the system at an abstract level and moves down in the level of abstraction by mapping the functionalities onto components making the implementation more accurate at each level. The design at the system level is split into small functionalities and are composed hierarchically. The required components are added and the functionalities are mapped onto the components. Once the architecture of the design is finalized, the design is synthesized to arrive at the final implementation. This approach is easier to manage and the designer gets the freedom to choose the algorithm and architecture based on the design constraints. Hardware-Software co-design environments, POLIS system [5] and COSYMA [21] use top-down design methodology.

In the bottom-up design methodology, design moves from lowest level of abstraction to the system level by putting together previously designed components such that the desired behavior is achieved at each level. The design will start by building the gates in a given technology. Basic units are built using the gates and the basic units are put together to make modules with desired functionality. Finally, the modules are assembled to arrive at an architecture. In this approach, the freedom of choosing the architecture is restricted. However, this approach has some advantages. Since each module is compiled separately, a design change in one of the modules requires re-compilation of only that module. [7] introduces high-level component-based bottom-up methodology and design environment for application-specific multi-core SoC architectures. This approach does not provide much help on automating the architecture exploration.

A mix of both top-down/bottom-up approaches to take advantage of both the approaches are also possible. Such an hybrid approach is adopted in [26] to reduce the design cycle time in FPGA design methodology by eliminating the need for complete design re-synthesis and re-layout when accommodating small functional changes.

### 1.4.2 Specification Languages

A number of system level languages (SLDLs) have been developed in the recent years with an intent to capture designs containing both hardware and software components at all levels in the design flow. Out of all the languages, two languages need mention because of their prevalent use, SystemC [14] and SpecC [11]. Both the languages are based on C language. SystemC implements system level modeling concepts extending C++ class library. SpecC, on the other hand, is a new language with a new compiler and simulator. Its an ANSI-C extension with new constructs to support system level modeling.

For our project, SpecC was chosen as the SLDL for its simplicity and completeness. The easy availability of the SpecC compiler and simulator and the SpecC based automated SoC design methodology, SCE made the decision easier.

### 1.4.3 SoC Design Flow Examples

In this section, we will discuss two related works, that apply the SoC design methodology on two real life examples.

#### 1.4.3.1 Design Exploration and Implementation of Digital Camera

A top-down design methodology with digital camera as an example is discussed in [27]. The design process of this example starts with an informal specification in the form of an English document. This specification is refined and a complete executable specification in C language is written with 5 different functional blocks. First, an implementation on an single general purpose microcontroller is considered and based on manual analysis of the computation complexity, the possibility of a competitive design solution with this partition is ruled out. Further, three more explorations based on hardware/software partitioning are discussed, to improve the design in terms performance, power and gate count. The design is manually partitioned into hardware and software partitions based on manual analysis and designer's experience.
Implementations starts at RTL. Synthesizable RTL description of the general purpose processor core is available for the project. The special purpose processors for the hardware partitions are written in synthesizable RTL description. For the software partitions, majority of the code is derived from the specification model and is modified to communicate with the hardware partitions at necessary places. The resulting software in C is compiled and linked to produce final executable. The executable is then translated into the VHDL representation of the ROM using a ROM generator. After these steps, the entire SoC is simulated using a VHDL simulator validating functionality and timing. Using commercially available synthesis tools, the VHDL RTL is synthesized into gates. From the gate level simulation, necessary data to compute power is obtained. Gate count is used to compute the area of the chip. The same process is repeated for different explorations till the implementation matching the design constraints is obtained.

In this methodology, since the implementation is manual at RTL, its time consuming to design hardware for each partition and for each exploration. The lack of design automation restricts the number of explorations and makes the design procedure not suitable for complex applications.

#### 1.4.3.2 Design Exploration and Implementation of Vocoder

A complete system level modeling and design space exploration, using top-down SpecC design methodology, of an GSM Enhanced Full-Rate speech vocoder standard is presented in [1]. This was a medium sized application and was intended to demonstrate and evaluate the effectiveness of SpecC design methodology. The complete specification model of the vocoder is captured in SpecC. SoC Environment (SCE) was used for design space exploration.
First, computational hot-spots are identified using a retargetable profiler [6] integrated in SCE. To start with, a single software partition is tried. The entire vocoder functionality is mapped to a Digital Signal Processor (DSP) available in the SCE database [12] and simulated using the SpecC simulator. Based on the simulated timing results, the single software solution was ruled out, as it could not meet the timing requirement. Next, design based on hardware software partitioning is

explored. Based on the profiler output, the hot-spot behavior in the design is mapped to special purpose hardware component with a desired operating frequency. The rest of the functionalities are mapped to a DSP. The automatic architecture refinement tool is used perform the behavior partitioning and generate the architecture model. The architecture model is simulated to verify the functionality and the timing. If the timing requirements are satisfied, busses are allocated, and channels in the design are mapped onto the busses and communication refinement is performed to produce a bus functional model. Again, the resulting model is simulated to verify functionality and timing. Finally, RTL generation tool of the SCE is used to synthesize the RTL for the hardware components and C code generation tool is used to generate the C code for the software components, to arrive at a complete implementation model.

The refinement steps proposed by the SpecC design methodology, *Architecture exploration, Communication exploration, Implementation synthesis* are automated in the SCE. Designer deals with only writing specification model and is relieved of repeated manual implementation of models at different abstraction levels. This considerably reduces the design process time. Designer can devote all the attention towards writing a good specification model of the application. Designer gets accurate feedback on timing by simulating each refined model. Considerable time is saved by running the simulation of the abstract models and getting the early feedback.

### 1.4.4 Our Work

In our work, we applied the SpecC design methodology on an industry size design example. We implemented a MP3 audio decoder using SCE. We implemented a complete specification model of MP3 audio decoder in SpecC SLDL and used the SCE to perform the design space exploration. As a result of automation provided by SCE, we explored different architectures in relatively shorter time. The report focuses on the major design effort of writing a good specification model and at relevant point discusses the possibility and techniques to automate the process of writing specification model. A preliminary implementation of this design example is discussed in [24]. The specification model in [24], was not complete and barely facilitated sufficient design space exploration. Some of the deficiencies included

- The specification model did not have enough granularity. There were very few leaf behaviors thus, restricting the extent of design space exploration.
- The specification model did not expose true parallelism in the application.
- The concurrency exposed in the specification model was not truly concurrent as the two computation units composed in parallel communicated in a Remote Procedure Call (RPC) style thus making them sequential.

In this work, the specification model was re-modeled starting from C implementation to have sufficient granularity, concurrency, and computational load balance across behaviors We were able to perform design space exploration with interesting partitions, to arrive at a suitable architecture for the MP3 audio decoder.

# 2 Design Example

In this section, we will describe the chosen design example, a MP3 Audio decoder. This section also gives an overview of the compression algorithm.

Digital compression of audio data is important due to the bandwidth and storage limitations inherent in networks and computers. The most common compression algorithm is the ubiquitous MP3 along with the other contenders like, Windows Media Audio (WMA), Ogg, Advanced Audio Coding (AAC) and Dolby digital (AC-3). A brief description of these formats is available in [4]. All of these use a variety of clever tricks to compress music files by 90% or more. Even though, standards like AAC and MP3PRO promise better quality at lower bitrates, at this stage, MP3 is an undisputed leader because of its wide spread use.

MP3 [16] provides significant compression through lossy compression, applying the perceptual science of psycho acoustics. Psycho acoustic model implemented by MP3 algorithm takes advantage of the fact that the exact input signal does not need to be retained. Since the human ear can only distinguish a certain amount of detail, it is sufficient that the output signal sounds identical to the human ears. In the following section, the generic structure of an MP3 decoder is presented.

## 2.1 Description of MP3 Decoder

The MP3 decoder for our design will use a complete MP3 stream as input. Before presenting more details about the actual decoding process, a short overview of the MP3 bit stream is given.

### 2.1.1 Structure of an MP3 Audio Frame

The MP3 stream is organized in frames of bits. Each frame contains 1152 encoded PCM samples. The frame length depends on the bit rate ( quality) of the encoded data. Since the bit rate may vary in variable rate encoded streams, the frame size may also vary within a single stream. Therefore the frame header contains information for the frame detection. Each encoded frame is divided into logical sections and these can be viewed hierarchically as shown in Figure 3.

The various fields in a frame of audio data are discussed below.

**Header** is 4 bytes long and contains sync word to indicate the start of frame. Header contains Layer information (MPEG Layer I, II or III), bitrate information, sampling frequency and mode information to indicate if the stream is mono or stereo.

**Error Check** This fields contains a 16 bit parity check word for optional error detection with in the encoded stream.

**Side information** Contains information to decode *Main data*. Some of the fields in *side information* are listed below

- It contains scale factor selection information, that indicate the number of scalefactors transferred per each subband and each channel. Scalefactors indicate the amount by which an audio sample needs to be scaled. Since, human ear response is different for

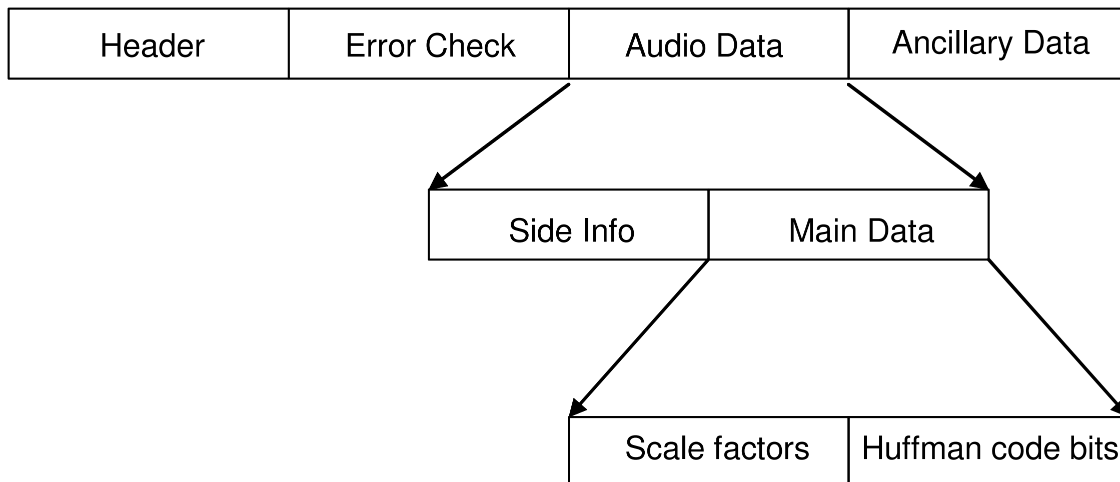



Figure 3: MPEG 1 Layer 3 frame format

signals at different frequencies, the entire audio spectrum is divided into subbands. The samples in the more sensitive bands are scaled more than the samples in the lesser sensitive region of the spectrum.

- It contains global gain which needs to be applied to all the samples in the frame.
- Information regarding the number of bits used to encode the scalefactors. To achieve compression, even the scalefactors are encoded to save the bits. This information in the sideinfo will indicate the number of bits to encode a particular scalefactor.
- Information regarding the huffman table to be selected to decode a set of samples. This information specifies one of the 32 huffman tables used for huffman decoding.

**Main data** The main data contains the coded scale factors and the Huffman coded bits.

- Scalefactors are used in the decoder to get division factors for a group of values. These groups are called scalefactor bands and the group stretches over several frequency lines. The groups are selected based on the non-uniform response of human ear for various frequencies.
- The quantized values are encoded using huffman codes. The huffman encoding is used to code the most likely values with lesser number of bits and rarely occurring values with larger number of bits. The huffman codes are decoded to get the quantized values using the table select information in the sideinfo section of the frame.

**Ancillary data** This field is the private data and the encoder can send extra information like ID3 tag containing artist information and name of the song.

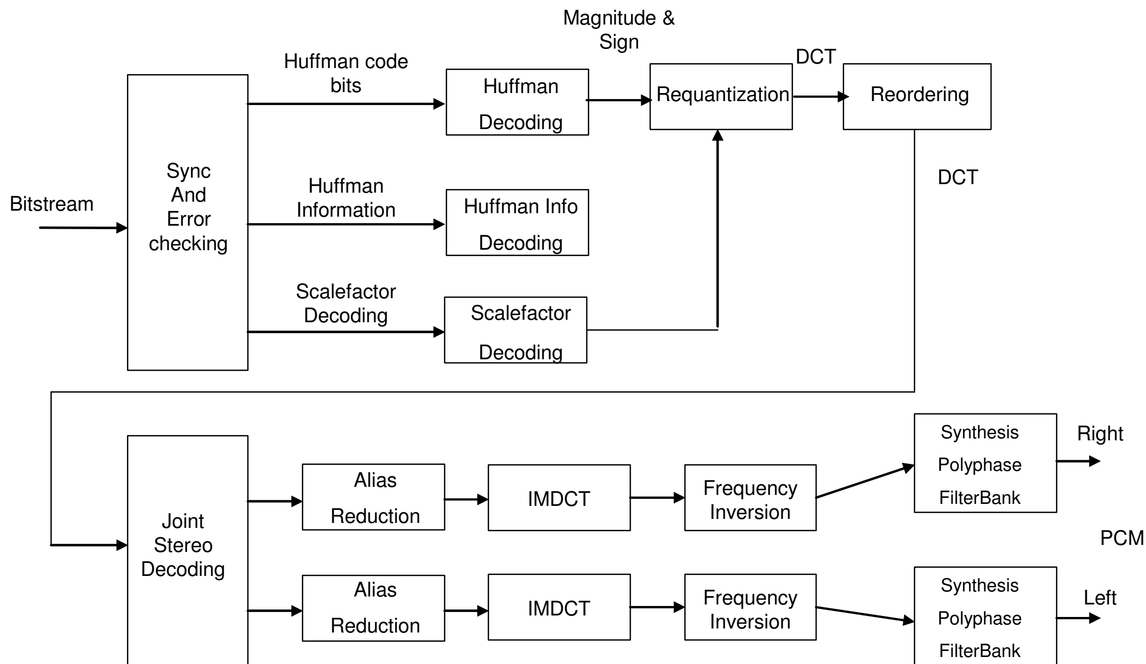



Figure 4: Block diagram of MP3 decoder [18]

### 2.1.2 MP3 Decoder Operation

The block diagram in Figure 4 shows the data flow within the MP3 decoder. The incoming data stream is first split up into individual frames and the correctness of those frames is checked using Cyclic Redundancy Code (CRC) in the *sync and the error checking* block shown in Figure 4. Further, using the scale factor selection information in the side information, scale factors are decoded in the *Scalefactor decoding* block. Scale factors are used to scale up the re-quantized samples of a subband. Subband is a segment of the frequency spectrum. Subbands are introduced in the encoder to selectively compress the signals at different frequencies. These subbands are chosen to match the response of human ear. The main data of the frame is encoded as a Huffman codes The quantized samples are derived from the huffman codes in the *Huffman decoding* block. The necessary side information needed for huffman decoding is obtained from *Huffman Info decoding* block. Since the huffman codes are variable length codes, the huffman encoding of the quantized samples results in a variable frame size. In order to optimize the space usage in a frame, the data from the adjacent frames are packed together. So, the *Huffman Decoding* stage refers to the previous frames data for its decoding. The next step after Huffman decoding, is the re-quantization. The re-quantizer, re-quantizes the huffman decoder output using the scalefactors and the global gain factors. The re-quantized data is reordered for the scalefactor bands. The re-quantized output is fed to the stereo decoder, which supports both MS stereo as well as Intensity stereo formats. The alias reduction block is used to reduce the unavoidable aliasing effects of the encoding polyphase filter

bank. The IMDCT block converts the frequency domain samples to frequency subband samples. The frequency subbands were introduced by the encoder. This allows treating samples in each subband differently according to the different abilities of the human ear over different frequencies. This technique allows a higher compression ratio. Finally, the polyphase filter bank transforms the data from the individual frequency subbands into PCM samples. The PCM samples can now be fed to a loudspeaker or any other output device through appropriate interface.
A comprehensive literature about the MP3 audio compression standard is available in [15] [22] [18].

# 3 Specification Model

Specification model is the starting point in the system design process and forms the input to the architecture exploration tool. Specification model is the result of capturing the functionality of the design in System Level Description Language (SLDL). It is a pure functional, abstract model, and is free of any implementation detail. Since the specification model forms the basis for the synthesis and exploration, it is important to write ”good” specification model. A good specification model has the following important features:

**Separation of computation and communication:** Specification model should clearly separate the communication blocks from the computation blocks. This enables rapid exploration by facilitating easy plug-n-play of modules. Abstraction of communication and synchronization functionality is a key for efficient synthesis and rapid design space exploration. In SpecC SLDL, computation units can be modeled using behaviors and communication elements using channels.

**Modularity:** Modularity is required in the form of structural and behavioral hierarchy allowing hierarchical decomposition of the system. The hierarchy of behaviors in the specification model solely, reflects the system functionality without implying anything about the system architecture to be implemented.

**Granularity:** The size of the leaf behaviors determines the granularity of the design space exploration. More the number of leaf behaviors greater are the number of the possible explorations. Granularity depends on the user and the problem size. There is a wide range of possibilities: On one extreme, every instruction can be a behavior and on the other extreme, entire design could be in one behavior. The former means complex design space exploration because of too many components, so it is not practical. The later results in reduced design space exploration. Granularity at subroutine level is usually better, as the number of components are manageable.

**Implementation details:** Specification model should not have any implicit or explicit implementation detail. Having implementation details would restrict the design space exploration. For example, describing the functionality of a behavior at RTL would result in an inefficient solution, at a later stage, if the behavior is implemented in software.

**Concurrency:** Any parallel functionality in the algorithm must be made into concurrent modules. This would enable exploration of faster architectures.

Specification model of the design could be written from scratch, which requires extensive knowledge of the algorithm being implemented. In this case, user can decide the granularity, hierarchy and concurrency of the design based on the knowledge of the algorithm. This approach might be time consuming as one is starting from scratch and the resulting specification model requires considerable amount of verification before considering it for rest of the design process. More than often, in the embedded system development, specification model needs to be developed from an existing reference C code which implements the algorithm. This approach is faster than the former as

| Properties of the reference C implementation | |
|---|---|
| Total number of source files | 66 |
| Total number of lines of code | 12K |
| Number of source files in the core MP3 algorithm implementation | 10 |
| Number of lines of code in the core MP3 algorithm implementation | 3K |
| Number of functions in the core MP3 algorithm implementation | 30 |

Table 1: Properties of the reference implementation of MP3 decoder.

the significant amount of effort has already been invested in making the reference code. Moreover, since the SpecC SLDL is just a superset of C language it would require lesser effort to convert the C reference code into SpecC specification model than writing the specification model from scratch. The rest of this section will describe the development of the specification model starting from reference C-code of a MP3 Audio decoder.

## 3.1 Reference C Implementation of MP3 Decoder

To develop the specification model we referred to the C implementation of the MP3 decoder available from MPG123 [20]. MPG123 is a real time MPEG Audio Player for Layers 1,2 and 3. The player provides, both, the core decoding functionality and interactive Graphical User Interface (GUI). This reference decoder is designed for and tested to work on Linux, FreeBSD, SunOS4.1.3, Solaris 2.5, HPUX 9.x and SGI Irix machines. It requires AMD/486 machines running at at least 120MHz or faster machine to decode stereo MP3 streams in real time.

### 3.1.1 Properties of the Source of Reference Implementation

The properties of the reference implementation are given in Table 1. The table lists some of the physical properties of the C code implementation of MP3123. The source archive contained floating point implementation of the MP3 Audio decoder. The implementation contained 66 source files, which included the actual decoding algorithm as well as supporting user interface code, contributing to 12K lines of code. For developing our specification model we only focused on the core decoding algorithm with a simple I/O spread over 10 source files, and comprising 3K lines of code. The source was split into 30 functions. A call graph of the major functions is shown in Figure-5.

Since this reference C implementation was not meant to be a SOC description, it had typical coding issues, that need to be remodeled. Some of these are listed below:

- The implementation majorly composed of pointer operations. Since pointers are not supported by the hardware synthesis tools, the presence of pointers in the section of code that would get mapped to a hardware PE is discouraged.

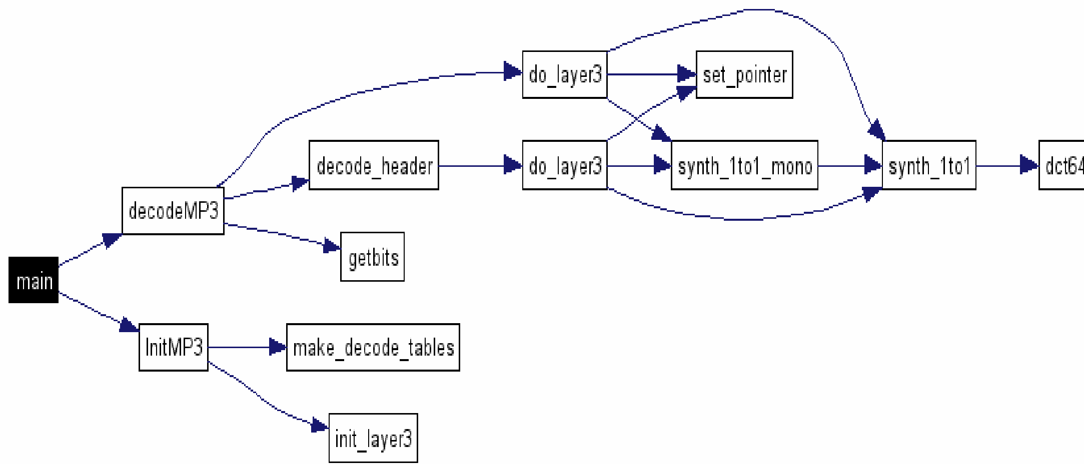



Figure 5: Call graph of major functions in the reference C implementation

- Usage of data structures with pointer members.
- Lack of behavioral hierarchy.
- Lack of separation of computation and communication blocks.
- Excessive usage of global variables.
- Absence of a distinct testbench and algorithm implementation.

To address these, a step by step approach was adopted to arrive at the final clean specification model. These are manual steps and are described in the subsequent sections, and wherever possible, we discuss the possibility of automation.

## 3.2 Initial Testbench

In this design step, we separated the core functionality of the MP3 decoder from the rest of the code in the reference implementation and built a testbench around it. The testbench remains unchanged through out the design process and provides the testing environment for our Design Under Test (DUT). This step involves few smaller tasks which are discussed in the following sections.

### 3.2.1 Making C Code SpecC Compliant

As a first step, the entire *main* function of the decoder was wrapped in one behavior, *Main*. In SpecC, the root behavior is identified by *Main* behavior and is the starting point of execution of

a SpecC program. The model was compiled using the SpecC compiler. Since the reference implementation was not ANSI-C compliant and due to some limitations in the SpecC compiler, there were compilation issues which required changes in the C code to make it SpecC compliant. Some of the issues encountered are listed below.

- In SpecC, Initialization of variables at the time of declaration is restricted only to constants. The C reference implementation had variable initialization with non-constants such as, previously declared variables or address of variables. Such variable definitions were manually changed to separate the definitions from initializations.
- Certain variable names in the C implementation like, *in, out* are keywords in SpecC. Such variables were renamed to some non-interfering names.
- One of the files in the standard library, *huge_val.h* was not ANSI-C compliant, this was changed without hampering the normal functionality.

After the above changes, we were able to compile and simulate the reference C code using SpecC compiler and simulator.

### 3.2.2 Building the Testbench

The core decoding functionality of the decoder was separated from the rest of code and was wrapped in a behavior *mp3decoder*. This new behavior is the DUT. Two leaf behaviors, *stimulus* and *monitor* were introduced to implement the functionality of the testbench. The three behaviors were instantiated in the *Main* behavior. The communication between these three behaviors was established using the queue channels, *x* and *y*. Read only information like, buffer size and type of stream being processed were shared across the behaviors using variables. The structure and the connectivity of the testbench is shown in Figure 6. The *stimulus* reads the input MP3 stream from the binary MP3 files (*.mp3) and sends it to *mp3decoder* in chunks. *mp3decoder* behavior decodes the input data and sends it to *monitor*. The *monitor* behavior receives the incoming data and writes it into an output file (*.pcm). It also compares the received data with reference output generated by the reference implementation.

### 3.2.3 Timing of the Testbench

In this section, we describe the timing of the stimulus and monitor behaviors to transmit and receive data respectively, at a correct rate. We also look at design of the buffer capacity in the testbench.
The stimulus is designed to feed the data into the mp3decoder in chunks of 256 bytes. In order to send the data at a correct rate, stimulus waits for *waittime* before every transfer. For a given bitrate, stream type (mono or stereo), and with the transfer size of 256 bytes, *waittime* for stimulus was computed as below.

$$number\ of\ chunks\ per\ second = (bitrate * stereomode/8)/(256)$$

$$waittime = (1/number\ of\ chunks\ per\ second) * 1000000000\ ns.$$

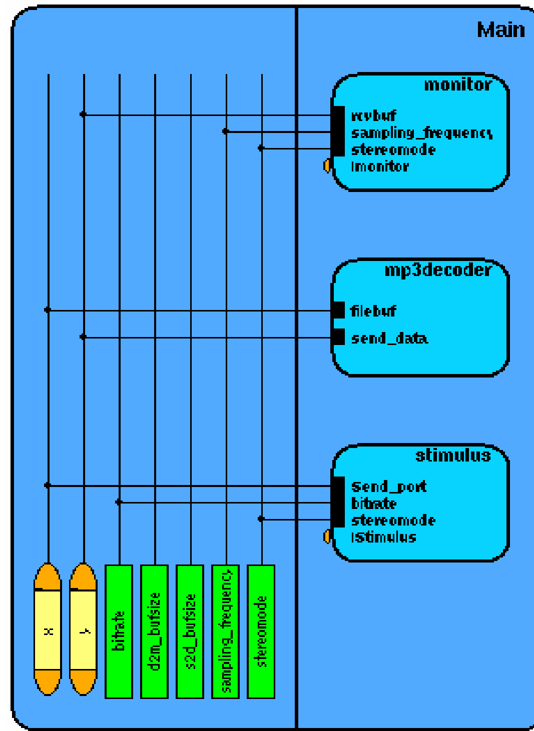



Figure 6: Top level testbench

Since we are calculating the wait period after every transfer of 256 bytes, we first compute *number of chunks per second* using the *bitrate* and *stereo mode* parameters. The inverse of the *number of chunks per second* gives the *waittime*. The above calculation gives the *waittime* in nano-seconds. The above timing detail is shown in Figure 7(a). The *x-axis* is the time line and *y-axis* indicates activity in bursts. The figure shows that there is a data transfer from stimulus to monitor in bursts of 256 bytes every *waittime ns*.

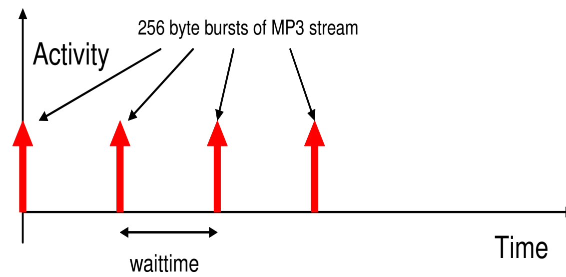
The monitor checks if the samples from the decoder are received in a stipulated time. Monitor computes this stipulated time or deadline using *sampling frequency* and *stereo mode* (This parameter is 1 for mono and 2 for stereo encoding) information. At the start of every frame, monitor checks if the frame was received within the stipulated time. This check will determine if the decoder is too slow to meet the necessary timing requirement. The *deadline* per frame of samples is computed as

$$deadline\ per\ sample\ =\ (1/(stereo\ mode * sampling\ frequency)\ *\ 1000000000.0)\ ns$$

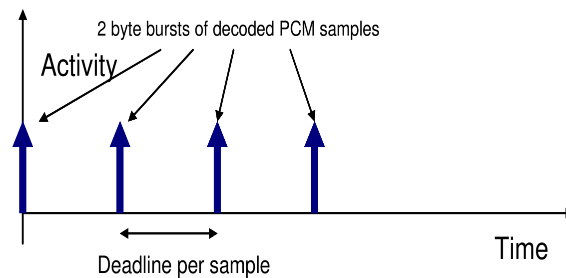
$$deadline\ =\ deadline\ per\ sample\ *\ samples\ per\ frame$$

where

$$samples\ per\ frame\ =\ 1152 * stereo\ mode$$



(a) Stimulus to Decoder data transfer activity



(b) Decoder to Monitor data transfer activity

Figure 7: Timing of testbench

In the above calculations, *deadline* is the time in nano-seconds to decode a frame of audio data. We first compute the *deadline per sample* using the *stereo mode* and *sampling frequency* parameters in terms nano-seconds. Using, number of *samples per frame*, we arrive at the deadline for the entire frame. The above timing detail is shown in Figure 7(b). The figure shows that there is a data transfer in bursts of 2 bytes every *deadline per sample ns*.

Now, we will look at the computation of the buffer capacity for the two channels in the testbench. The *stimulus* to *mp3decoder* queue must be designed to accommodate data worth at least one worst case frame size. The worst case frame size is computed as below:

$$Maximum\,Average\,Frame\,Size = samples\;per\;frame * Max\;possible\;bitrate/sampling\;frequency$$

$$= (1152 * 320Kbits/sec)/48KHz = 7680\;bits$$

$$= 960\;Bytes$$

To meet this requirement, a queue size of 1024 bytes was chosen.
Since the output from the decoder is written to the monitor one sample(2 Bytes) at a time, the mp3decoder to monitor queue could be of 2 bytes size.

### 3.3 Parallelization of the Design at the Top Level of the Hierarchy

In our specification model, there was no concurrency at the very top level of the decoder. So, the interface of the decoder with the monitor was sequential. As the monitor was designed to accept the data at a specific rate the whole decoder would be stalled till the decoded data was delivered to the monitor. This obviously was not desired as it meant wastage of resources in the decoder waiting for the data transfer to complete. Another issue with the model was that, the output data transfer rate was controlled by the monitor which required that in the real environment the output device be knowledgeable about the sampling frequency and stereo mode which is not practical when the output device is a simple speaker system. So, we moved this rate control intelligence into the decoder.
To meet the above requirements we modified the top level of the design to separate the decoder core functionality from the data transfer logic. The resultant hierarchy is shown in Figure 8.

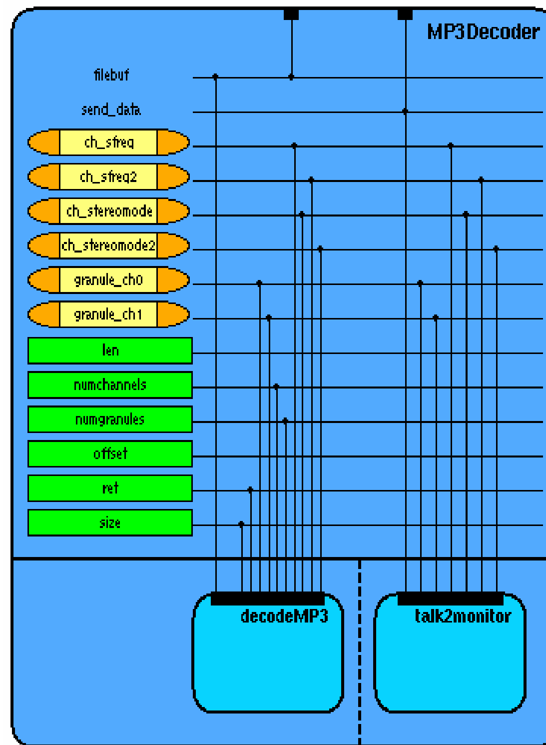



Figure 8: Top level parallel hierarchy of the decoder

The newly introduced behavior *Talk2Monitor* runs concurrently with the core behavior *decodeMP3* and consists of 3 child behaviors, *Listen2Decoder, ComputeTime* and *DataTransfer*, composed in FSM. *DataTransfer* is responsible for getting the decoded data from the *decodeMP3* in chunks of 32 samples and write it to *monitor* at a proper rate. This rate is calculated from the sampling frequency and stereo mode information by *ComputeTime* behavior. *Listen2Decoder* is respon-

sible for receiving this information from *decodeMP3* using double handshake channels *ch_sfreq, ch_stereomode*. Queue channels, *granule_ch0* and *granule_ch1* are used for communicating decoded samples from *decodeMP3* to *DataTransfer* behavior. The various channels used for communication are shown in the Figure 8.

## 3.4 Introducing Granularity

SpecC behaviors form the basic units of granularity for design space exploration. The leaf behaviors, behaviors at the bottom of the hierarchy, contain the algorithm implementation in the form of C code. So far, our Design Under Test (DUT) has just one behavior providing no scope for design space exploration. We need to introduce more behaviors into this design to make sufficient design space exploration. One easy way to do this is to convert all the major C functions in the design into behaviors. Based on the preliminary profile result obtained from GNU profiler, *gprof* and based on code observation a list of functions that needs to be converted into behaviors were identified. The behaviors were introduced based on the conventions listed below. The Figure-9 is used to explain this procedure.

### 3.4.1 Procedure

We will now describe the procedure used to convert functions to behaviors. Figure 9 shows an example for converting a function into behavior.In the figure, the code box on the left shows behavior *B1* encapsulating the function *f1*. The function returns the result which is written to the *out* port, *result*. The code box on the right shows the function *f1* encapsulated in a new behavior *B1_f1*.

1. Functions are converted to behaviors one by one using top-down approach following the functional hierarchy. This means that a function is never converted to behavior, unless its calling function (parent function) is converted to behavior.

2. The function to be converted is encapsulated in a behavior body and the function body is either inlined into the *main* of the new behavior or the function is made a member of this new behavior with the *main* of this new behavior containing a single call to this function. This second scenario is shown in the Figure-9 and the name of this new behavior is *B1_f1*.

3. The new behavior is instantiated in its parent behavior For example, in the figure the new behavior *B1_f1* is instantiated in *B1*. The ports of this new behavior are determined by the original function parameters. The necessary function arguments are made the members of the parent behavior. For example, in the figure, *i1, i2* and *s1* are made members of the behaviors *B1*. During later refinement stages, these ports might change or new ports might be introduced.

4. If any of the function parameters are pointers then they are replaced with an actual location or an array, depending on its type (notice that *i2* is mapped to the second port of *B1_f1*). The type of the port(*in, out, inout*) is determined based on how that parameter is used within the new behavior. If the function parameters are members of a composite structure(including arrays),

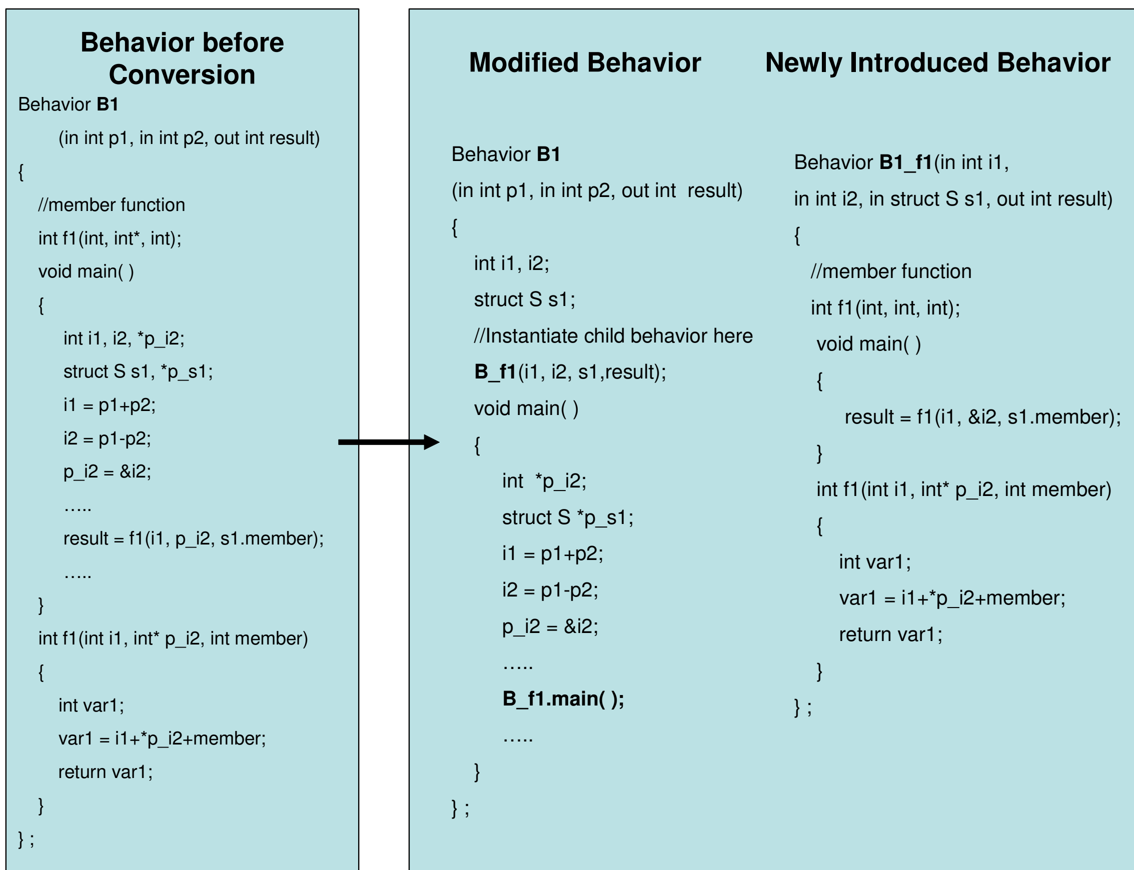



Figure 9: Example describing conversion of a C function to a SpecC behavior

then it has to be replaced with the entire structure. This is the case with the variable *s1* in the Figure-9.

5. The return result of the function is assigned to an out port. In the example, notice that there is one more port(out port result) for the new behavior, than the number of parameters to the original function.

### 3.4.2 Summary

All of the above steps except pointer conversion are pure mechanical and hence can be automated. However, the decision of choosing the function to be converted into behavior has to be made by the designer. Determining the type for each port of the newly introduced behavior, requires manual analysis. Each function parameter has to be analysed, to find if its read-only, write-only, or read-write parameter.

Using above steps, most of the major functions were converted to behaviors. After this major step we arrived at a SpecC implementation of the MP3 decoder with 18 behaviors and 31 behavior

instances. So far, we had converted only few functions to behaviors but most of the C code between function calls still exists between the behaviors. So, we now have behaviors interleaved with lots of C code. But in a ”good” specification model, C code is allowed only in the leaf behaviors ( behaviors which contain no other behaviors). For writing good specification model, which can be understood by the SCE tool, it is required, that at any level of hierarchy, all the behaviors are composed either sequentially, or in Finite Statement Machine style, or concurrently, or in a pipelined fashion. This can be achieved first by eliminating the stray code between the behavior calls. Apart from this issue, there was one more issue to be solved, the global variables. Since each behavior represents potentially an independent processing element, a behavior has to communicate with the external world only through its ports. So it is important to eliminate the global variable communication of the behaviors. We first addressed the problem of global variable, before taking up the problem of eliminating the stray code, as the former will influence the later procedure.

## 3.5 Elimination of Global Variables

Global variables hide the communication between functions, in a program because they don't appear as function parameters or return results of functions. Since they become globally available to all the functions in the program, programmers use this feature for convenience to declare variables used across many functions as globals. However, a good specification model requires the communication to be separated from the computation. So, the hidden communication through global variables must be exposed. Depending on the scenario, the communication through global variables can be removed using one of the procedures given below.

### 3.5.1 Procedure 1

If the usage of the global variable is restricted to only one behavior then the following procedure is used.

1. Global variables whose usage(read and writes) is restricted to only one behavior can be moved into that behavior making it a member of that behavior. In the Listing 1(a), the usage of global variable *g1* is restricted to behavior *b1* alone and hence has been moved all the way into *b1* as shown in Listing 1(b).

### 3.5.2 Procedure 2

If the usage of the global variable is spread across multiple behaviors then the following procedure is used.

1. Global variables whose usage is spread across more than one behavior are moved into the innermost behavior in the hierarchy which encompasses all the behaviors accessing that global variable. In the Listing 1(a), the global variable *g2* is used across two behaviors *b1* and *b2*. As shown in Listing 1(b), *g2* is moved into the *Main* behavior as *Main* is the inner most behavior encompassing both *b1* and *b2*.

```
int g1, g2;

behavior Main()
{
  int var1, var2, var3;

  b1 B1(var1, var2);
  b2 B2(var2, var3);

  int main(void)
  {
    B1.main();
    B2.main();
  }
};

behavior b1(in int i1, out int o1)
{
  void main(void)
  {
    g1 = g1+i1;
    g2 = i1++;
    o1 = i1;
  }
};

behavior b2(in int i1, out int o1)
{
  void main(void)
  {
    g2 = g2++;
    o1 = i1;
  }
};
```

(a) Specification model with global variables

```

behavior Main()
{
  int var1, var2, var3;
  int g2;

  b1 B1(var1, var2, g2);
  b2 B2(var2, var3, g2);

  int main(void)
  {
    B1.main();
    B2.main();
  }
};

behavior b1(in int i1, out int o1, out int g2)
{
  int g1;
  void main(void)
  {
    g1 = g1+i1;
    g2 = i1++;
    o1 = i1;
  }
};

behavior b2(in int i1, out int o1, inout int g2)
{
  void main(void)
  {
    g2 = g2++;
    o1 = i1;
  }
};
```

(b) Specification model without global variables

Listing 1: Eliminating global variables.

2. Moving the global variables into the innermost behavior will introduce new ports in the behaviors accessing the global variable and the type of the port is determined by the nature of the access of the variables. In Listing 1(b) there are new ports for the behaviors *b1* and *b2*. *b1* which only writes *g2* gets an extra *out* port and *b2* which both reads and writes *g2* gets an *inout* port.

### 3.5.3 Summary

The above mentioned refinement steps are mechanical and can be automated. The the necessary information regarding the usage of the variables are available in the compiler data structure and can be used to determine where the variable is defined and where all it is being used. However, determining the port types of the new ports, introduced due to motion of global variables, requires

manual analysis if these global variables are accessed using pointers within the behaviors.

## 3.6 Arriving at a Clean Specification Model

As described earlier, a clean specification model is one in which only the leaf behaviors contain the C code and all the child behaviors are composed either in *parallel* (using *par* statement), or in *pipeline* (using *pipe* statement), or in Finite State Machine(FSM) style(using *fsm* statement), or sequentially. But, at this stage, our specification model is composed of behavior calls interleaved with C code. The SpecC language reference manual [8] describes each of this composition styles in detail.
In this section we describe the procedure adopted to clean the specification model.

### 3.6.1 Procedure

The interleaving C code between behaviors can be wrapped into separate behaviors and these behaviors can be composed in either of the 4 ways mentioned above to get a clean specification model. The possibility of concurrent composition using *par* and *pipe* statements are considered later, as they are complex and require dependency analysis across behaviors to check if there exist any parallelism between them. At this stage, we look at composing the behaviors either in pure sequential fashion or in FSM style. Behaviors composed sequentially execute one after the other in the order specified. Similar to pure sequential composition, in FSM composition, the behaviors are executed one after the other sequentially. However, in addition, FSM composition facilitates conditional execution of behaviors. The conditions are specified next to the behavior and are evaluated after executing the behavior. The next behavior to be executed depends on the result of the condition evaluation. In the absence of any condition, the execution flow will fall through.

In our case, since some of the stray C code between behavior calls were conditional statements influencing the execution of behaviors, it was conducive to compose these behaviors in FSM style by having the conditional C code converted into conditional statements of the FSM. Straight line stray C code were wrapped into separate behaviors. Whenever possible, instead of introducing new behaviors, we pushed these instructions into the already existing neighboring behaviors. This later operation requires that the existing behavior's ports be changed to accommodate new changes.

The above described general methodology is adopted in the examples shown in Figure 10 and Figure 11. Figure 10 depicts the way to convert an *if* statement into an FSM. In this example, a new behavior *newB* is introduced encompassing the straight line C instructions *a = 1; var = 2;*. The conditional execution of behaviors *B1* and *B2* is made possible by absorbing the *if* condition into the FSM. These conditional C instructions appear in a different form next to the behavior call *newB*. Figure 11 shows a way to convert a *for* loop into a FSM. In case of for loops, the stray instructions include the loop initialization statements, loop condition and loop parameter update statements. The new behavior, *LOOP_CTRL* is introduced to update the loop parameter *i* with an increment instruction. The loop parameter initialization is moved to the preceding behavior, *START* and the loop condition evaluation is absorbed into the FSM body next to the *LOOP_CTRL* behavior call. The unconditional *goto* in the FSM body, next to the behavior call *B3*, forms a loop of *LOOP_CTRL, B1, B2, B3*. This loop is terminated when the conditional statements in the FSM body next to the

behavior call *LOOP_CTRL* evaluates to false. A similar strategy of code cleaning is discussed in [3].

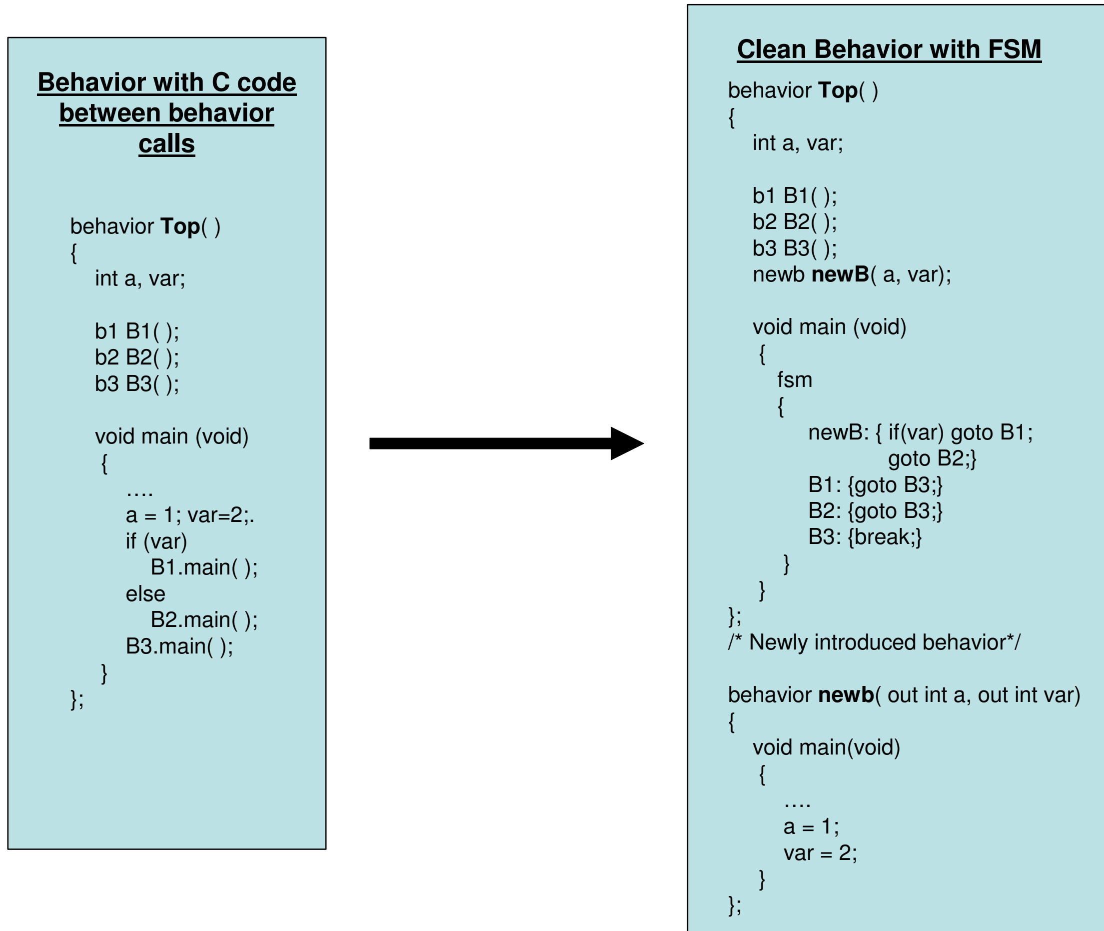



Figure 10: Example describing conversion of unclean behavior to a clean behavior

### 3.6.2 Summary

The above mentioned general procedure was used to clean up the MP3 decoder specification model. As an example, the entire granule processing unit of the MP3 decoder is shown in the Listing 2. If you notice, this section of the specification model has lots of C code in between the behavior calls. A clean FSM version is shown in Listing 3.

The general procedure adopted to clean the specification model involves purely mechanical steps and can be automated. With limited user inputs about the type of composition desired, wrapping of the C instructions in behaviors and converting conditional statements and loops into FSM can be achieved through automation.

```
behavior dogranule(/*list of ports*/)
{
  /*Instantiation of child behaviors and data structures*/
  void main()
5   {
      dolayer3_1.main();

      if (fr.lsf)
        sideinfo2.main();
10    else
        sideinfo1.main();

      setptr.main();

15    if(ret==MP3_ERR)
        return;

      for (gr=0;gr<granules;gr++)
      {
20      {

          setparam1.main();
          if(fr.lsf)
          {
25          scalefac2.main();
          }

          else {
            scalefac1.main();
30        }
          Dequant.main();
          if(dequant_ret) return;
        }
        if(stereo == 2) {
35        setparam2.main();
          if(fr.lsf)
          {
            scalefac2.main();
          }
40        else {
            scalefac1.main();
          }
          Dequant.main();
          if(dequant_ret) return;
45        msstereo.main();

          iStereo.main();

          dolayer3_2.main();
50      }

        for(ch=0;ch<stereo1;ch++) {
          antialias.main();
          Hybrid.main();
55      }
        sfilter.main();
      }
      return;
    }//main
60 };
```

Listing 2: Section of MP3 decoder specification model before clean up.

```
behavior dogranule(/* List of ports */)
{
    /* Instantiation of child behaviors and data structures */
    void main()
5   {

      fsm {
            dolayer3_1 : { if (fr.lsf) goto sideinfo2;
                           goto sideinfo1;}
10
            sideinfo1: {goto setptr;}
            sideinfo2: {goto setptr;}
            setptr: { if (ret == MP3_ERR) break;
                      goto setparam1;
15                   }
            setparam1: { if (fr.lsf) goto scalefac2;
                         goto scalefac1;
                       }
            scalefac1: {goto Dequant;}
20          scalefac2: {goto Dequant;}
            Dequant   : {if (dequant_ret) break;
                         if (stereo == 2 && dequant_ch == 0) goto setparam2;
                         if (stereo == 2 && dequant_ch == 1) goto msstereo;
                         goto antialias;
25                     }
            setparam2: { if (fr.lsf) goto scalefac2;
                         goto scalefac1;
                       }
            msstereo: {goto iStereo;}
30          iStereo: {goto dolayer3_2;}
            dolayer3_2: {goto antialias;}
            antialias : {goto Hybrid;}
            Hybrid: { if (ch<stereo1) goto antialias; //increment ch
                      goto sfilter;
35                  }
            sfilter: {if (gr<granules) goto setparam1; //increment gr
                      break;
                     }
        }//fsm
40  }//main

};
```

Listing 3: Section of MP3 decoder specification model after clean up.

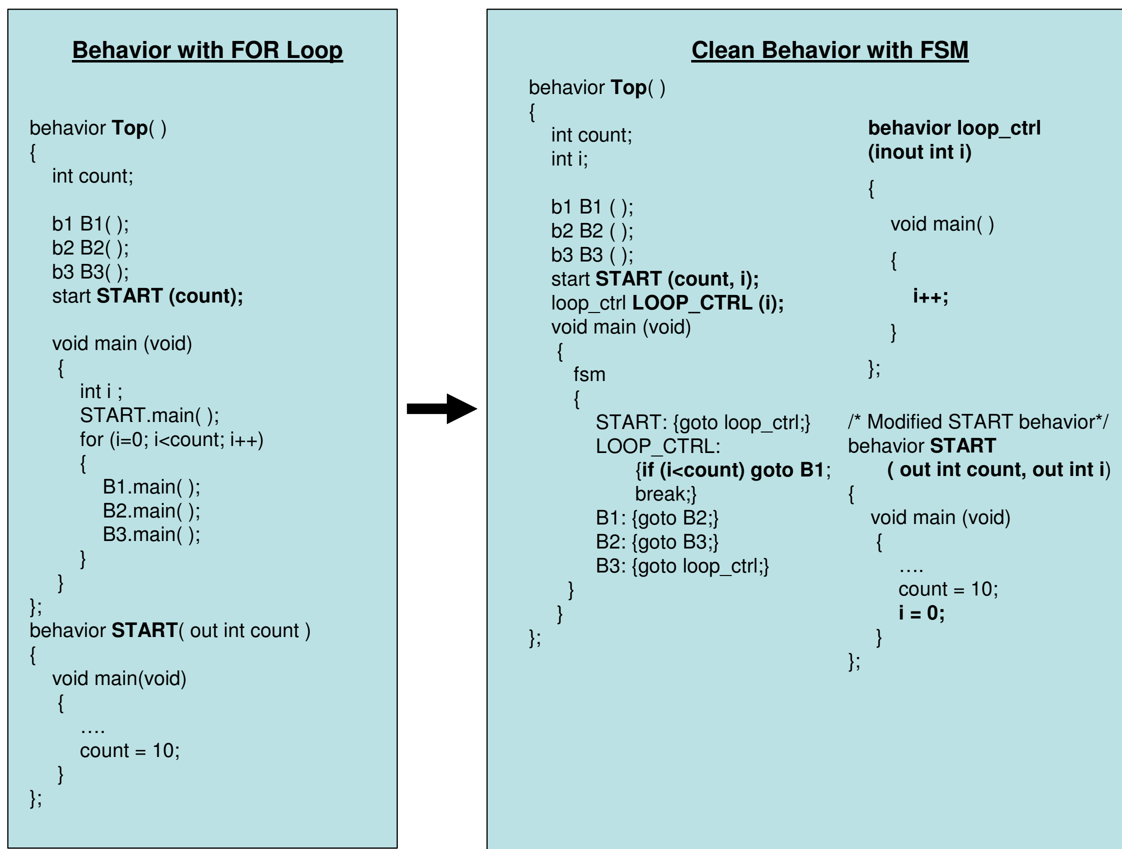



Figure 11: Example describing conversion of a FOR statement into FSM

### 3.7 Introducing Concurrency in the Specification Model

After all the above steps, our specification model was clean from global variables, it had a clearly separated design and testbench, the C code was restricted only to the leaf behaviors and at every level of hierarchy all the behaviors were composed either sequentially or in FSM style. The next step was to expose concurrency in the model. Any parallelism in the design has to be explicitly exposed in the specification model, so that it can be exploited later during design space exploration. In this section, we first talk about the various conditions to be satisfied to have parallelism between behaviors and discuss with examples the actual steps taken to introduce concurrency in the design.

#### 3.7.1 Conditions for Concurrency

In SpecC, two types of concurrent execution between behaviors can be exposed, parallel execution and the pipelined execution. The former is explicitly exposed using *par* statements and the pipelined concurrency is exposed using the *pipe* statements.

The following conditions must be satisfied for two behaviors to be composed in parallel.

1. The behaviors must be at the same level of hierarchy.

2. The behaviors must not write to the same variable.

3. The behaviors must not have access to the same variable, if at least, one of those behaviors can write to it.

In SpecC paradigm, the above conditions can be restated as, "Behaviors at the same level of hierarchy can be composed in parallel, without synchronization overhead, if the behaviors don't have their ports mapped to the same variable. If they are mapped to a common variable, then the ports of all the behaviors mapped to that common variable must be *in* ports."
The task of checking these conditions is purely mechanical and hence can be automated to determine if two behaviors can be composed in parallel. However, under some circumstances, complete automation is not possible. If the common variables across behaviors are composite variables, like arrays and structures, then, depending on just above conditions would result in conservative result, because, having a composite variable in common across behaviors doesn't necessarily mean the behaviors are accessing the same field of the composite structure. In such cases, further analysis within the behaviors needs to be done to check if the behaviors are interfering with each other by writing to the same field of the composite variable. Again, this requires manual attention and cannot be automated completely. The other possibility is to introduce another refinement step to break the composite variables into normal variables, for example, unwinding the array to individual elements, splitting the structure into individual elements. But this requires modification of the C code to change all the access to the composite variables to simple variables. Though this is possible under certain circumstances, it is not possible, when pointers and indices are used to access the composite variable as their values are not known at static time.
More than often, the parallelism between behaviors might be hidden, and relying on just the above conditions will not detect that parallelism. Detecting such a parallelism requires designer's intervention.

### 3.7.2 Conditions for Pipelined Concurrency

Now, we will look at the conditions to be satisfied to compose behaviors in pipelined fashion. Pipelined execution is possible in applications which perform a series of operations on the input data set to produce output data set with the output of each stage serving as the input to the next stage of operation. If the specification model has all those operations captured in separate behaviors then pipelining is possible, if following additional conditions are met.

1. The set of behaviors to be composed in pipelined fashion must be composed pure sequentially.

2. The input of each behavior is from the preceding behavior's output and so on. Basically, the data must flow only in one direction from the head of the pipeline to tail.

3. Two behaviors should not write to the same variable. That is, there can be only be one behavior writing to a common variable.

To get the full benefit of pipelining, the pipeline should run continuously. For efficient utilization of the pipeline, there must be continuous input at the mouth of the pipeline. For example, there is no real benefit in having a pipeline that gets flushed after every run. Also, pipelining is useful when the computation load is balanced across all the behaviors in the pipeline, otherwise, the pipeline will be as fast as the slowest behavior (the most compute intensive behavior). Because of these requirements, choosing the behaviors to be pipelined will have to be a result of manual analysis. So, only the mechanical tasks listed above can be automated and the decision making has to be taken care by the designer.

### 3.7.3 Procedure for Introducing Concurrency

Lets look at the hierarchy starting from the behavior *DoLayer3*. The hierarchy captured using the SCE tool is shown in the Figure 12. We first explored the possibility of parallelizing the two granules, *granule1, granule2* in Figure 12, but due to data dependency, it was necessary that *granule2* operations are performed after *granule1*. So we focused our attention to parallelize operations within each granule. The function of operation in a granule are captured in behavior, *DoGranule*. *DoGranule* shown in Figure 12 is an FSM of many behaviors. Of these behaviors, we narrowed our focus to 3 behaviors *alias_reduction*, *imdct* and *sfilter*. We first choose to parallelize the less complex behavior, *alias_reduction*. This behavior did sequential processing on independent input data set belonging to two audio channels. The behavior implemented alias reduction algorithm for the two audio channels. Analysis of the code revealed that the function *III_antialias ( )* implemented the alias reduction algorithm for a channel of audio data and was called twice for processing each channel data. Each call operated on independent data set and hence there were no data dependency between each channel processing. The code box on the left of Figure 13 shows the implementation of *AliasReduction* behavior. The behavior calls *III_antialias ( )* function in a for loop which loops as many times as the number of channels. This computation on each channel data was wrapped into a new behavior and this is shown in the code box on the top-right side in Figure 13. Two instances of this new behavior, *antialias_ch0, antialias_ch1* were instantiated in the parent behavior, *AliasReduction*, and the *for* loop was removed and the behavior calls were encapsulated in the *par* construct to make the parent behavior a clean parallel behavior as shown in the bottom-right code box in Figure 13. The new behavior instances get the information regarding the channel number to index into the right data set and the number of active channels which acts as an enable for the second instance, *antialias_ch1*.

After parallelizing *AliasReduction*, we focused our attention towards more complex behaviors *imdct, sfilter*. In these behaviors, we identified data independence between two audio channels just like the *AliasReduction* behavior, and hence, using similar approach we introduced concurrency at channel level processing into these behaviors. These concurrencies are shown in Figure 14.

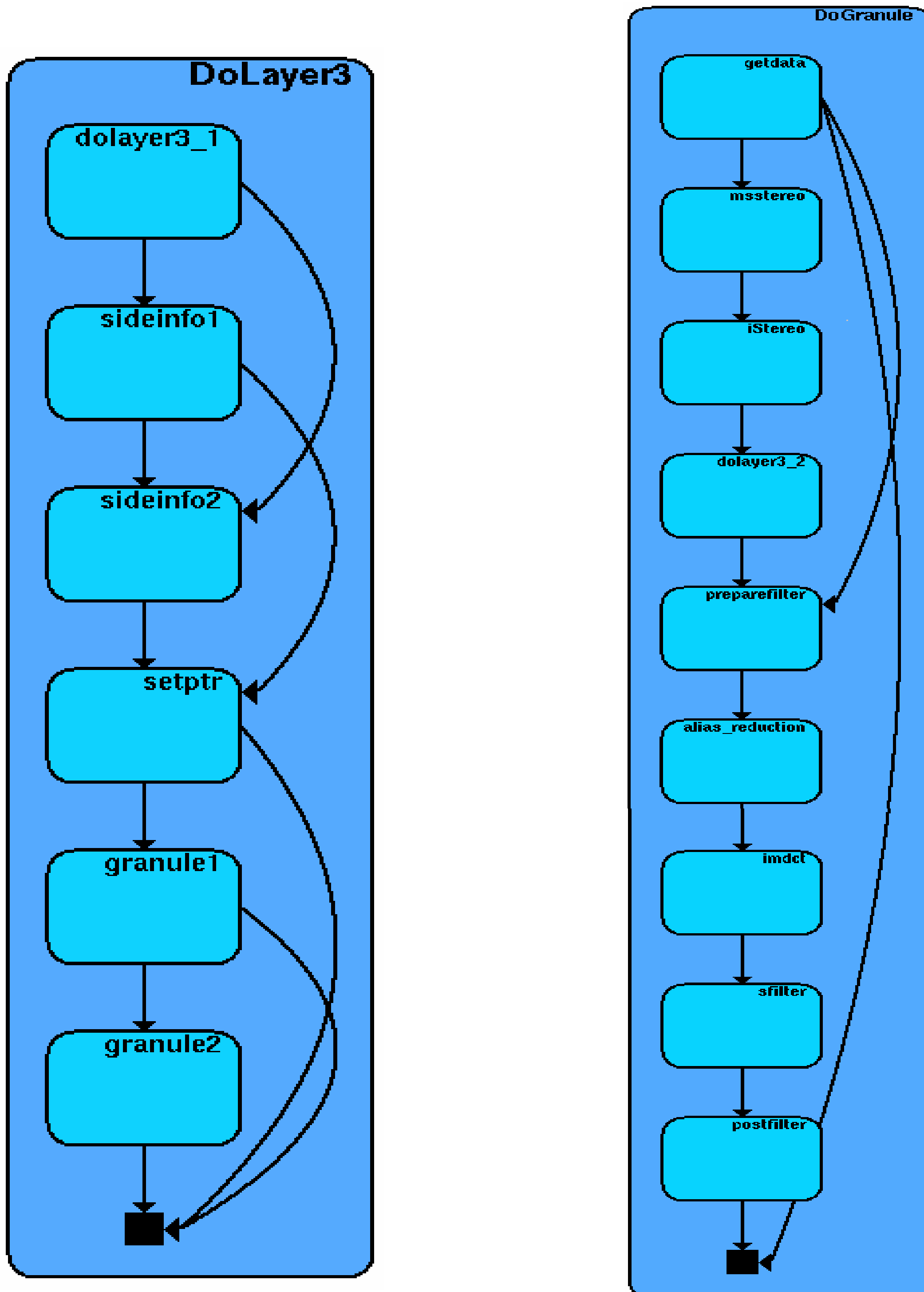



Figure 12: Hierarchy within DoLayer3 behavior in the MP3 decoder specification model

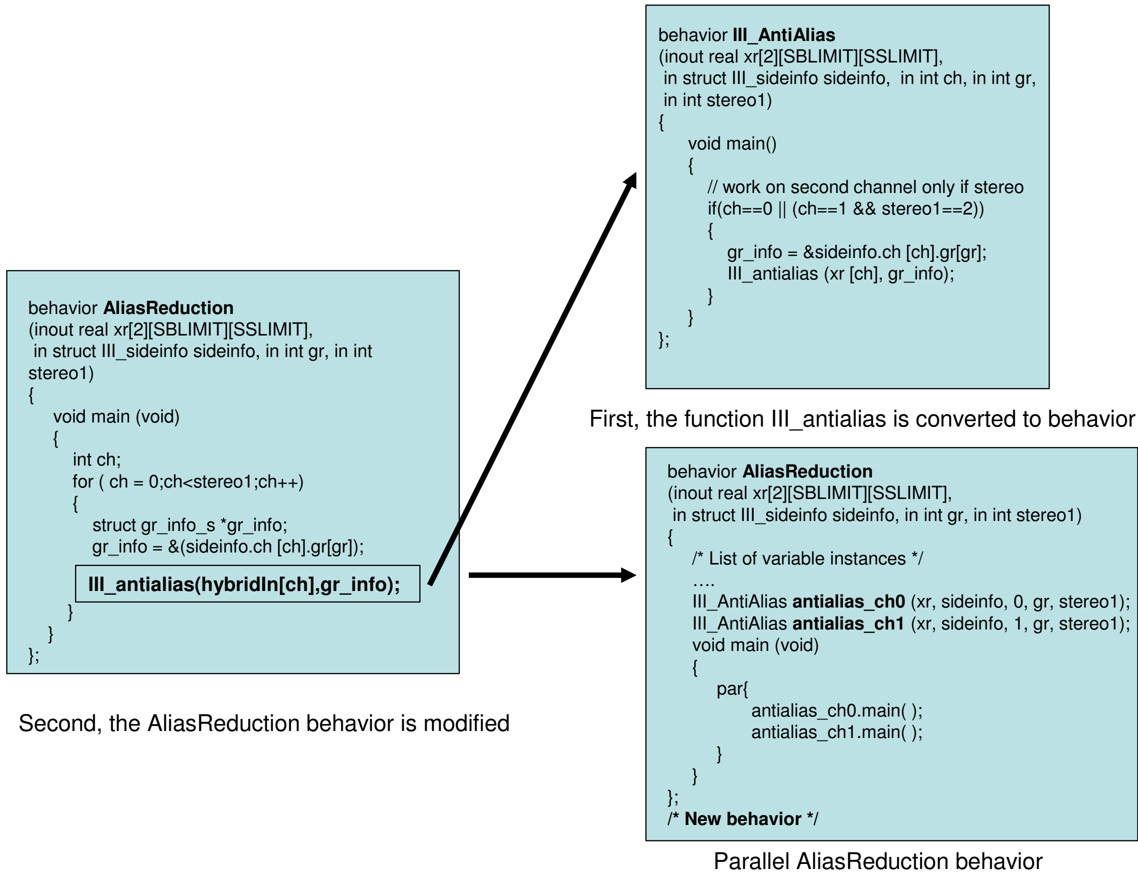



Figure 13: Example showing the conversion of a sequential behavior into concurrent behavior

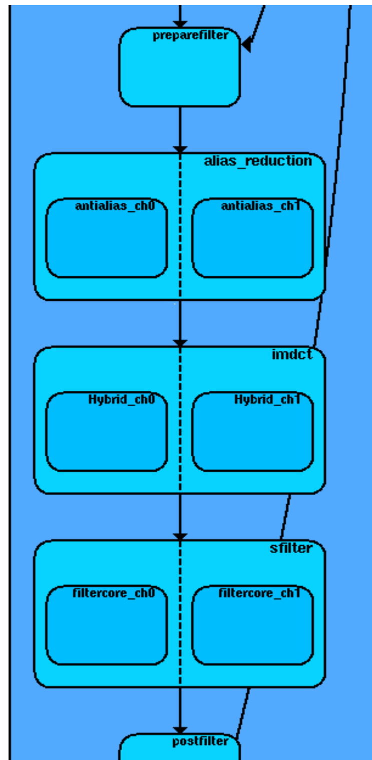


Figure 14: Parallelsm in the MP3 decoder specification model

### 3.7.4 Procedure for Introducing Pipeline Concurrency

At this stage, the specification model had enough granularity, it had 39 behaviors and 122 behavior instances. Out of the 39 behaviors, 31 were leaf behaviors providing good scope for exploration. The parallelism was explicitly exposed, opening the possibility of exploring faster architectures. With an intent to check the computation load distribution across various behaviors, we profiled all the behaviors using SCE. Considering only the most compute intensive behaviors, we narrowed our focus to three most compute intensive behaviors. The graph in Figure 15 shows the relative computation complexity of behaviors, *alias reduction*, *imdct*, and *sfilter*. From the graph, its clear that *sfilter* behavior is the single most computationally intensive behavior. It is 70-75% more expensive than the other behaviors. Since, unbalanced computation load will not result in good partitioning we decided to break the *sfilter* behavior further.

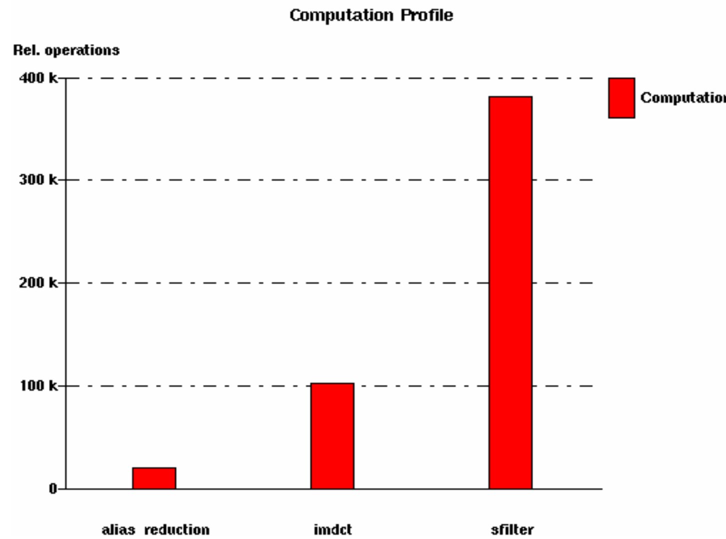



Figure 15: Relative computation complexity of the three most compute intensive behaviors of MP3 decoder specification model

We identified two logical partitions in the synthesis filter. A first stage was the computation of the 64 point DCT and the second stage was the extensive window overlap-add operation. These operations were performed in a loop running for 18 times for each audio channel. We first separated the model into two behaviors, *DCT64* and *WindowOp*, using the techniques discussed in Section 3.4. Further, we introduced two more behaviors, *setbuffer1* and *DCTOutStage* which act as helper stages by reordering data for *DCT64* and *WindowOp*. This resulted in 4 behaviors, *setbuffer1, DCT64, DCTOutStage, WindowOp* in a *for* loop executing 18 times in that order. Each behavior received its input from the preceding behavior's output and all the variables were at most written by one behavior satisfying all the conditions for pipelining discussed in Section 3.7.2. The 4 behaviors were pipelined using the *pipe* construct. In addition, following changes were necessary to complete

the pipelining.

1. All the variables used for data transfer between the behaviors in the pipeline must be buffered. In other words, all the variables mapped to the *out* or the *inout* ports of the pipelined behaviors must be buffered. In SpecC, this can be done using automatic communication buffering feature of *piped* variables. The number of buffer stages for the variable is equal to the distance between the writer and the reader behavior.

2. Variables with one writer and more than one reader require extra attention. Such variables must be duplicated to create as many copies as the number of readers. The duplicated variables also need to be buffered using *piped* variables. Each variable must be *piped* as many times as the number of buffer stages required. The writer behavior must be modified to have extra *out* port. This port is mapped to the duplicate variable. The body of the writer behavior must be modified to write the same value to this new *out* port as the value being written to the original variable. The port of the second reader, reading this variable, must be mapped to the duplicate variable.

The result of pipelining is shown in the Figure 16. After pipelining, the computation load looked more balanced as the computation load of *sfilter* is now distributed across 4 behaviors *SetBuffe1, DCT64, DCTOutStage and windowop*. The relative comparison is shown in Figure 17. The shaded extensions in the bar graph indicate the result after scaling. Behaviors, *SetBuffer1, DCTOutStage*, are not shown in the figure as their computation is neglible compared to the others.

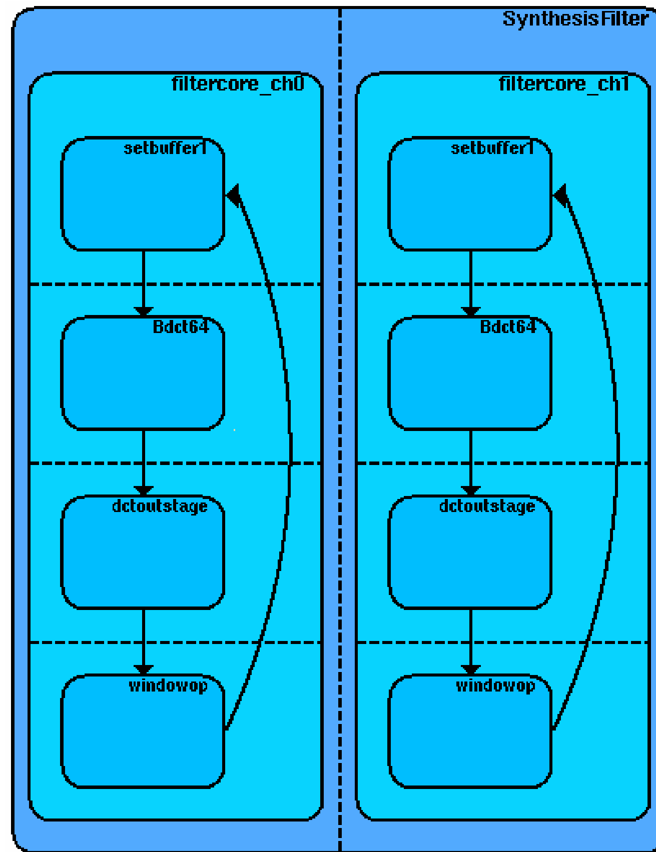



Figure 16: Pipelining in the MP3 decoder specification model

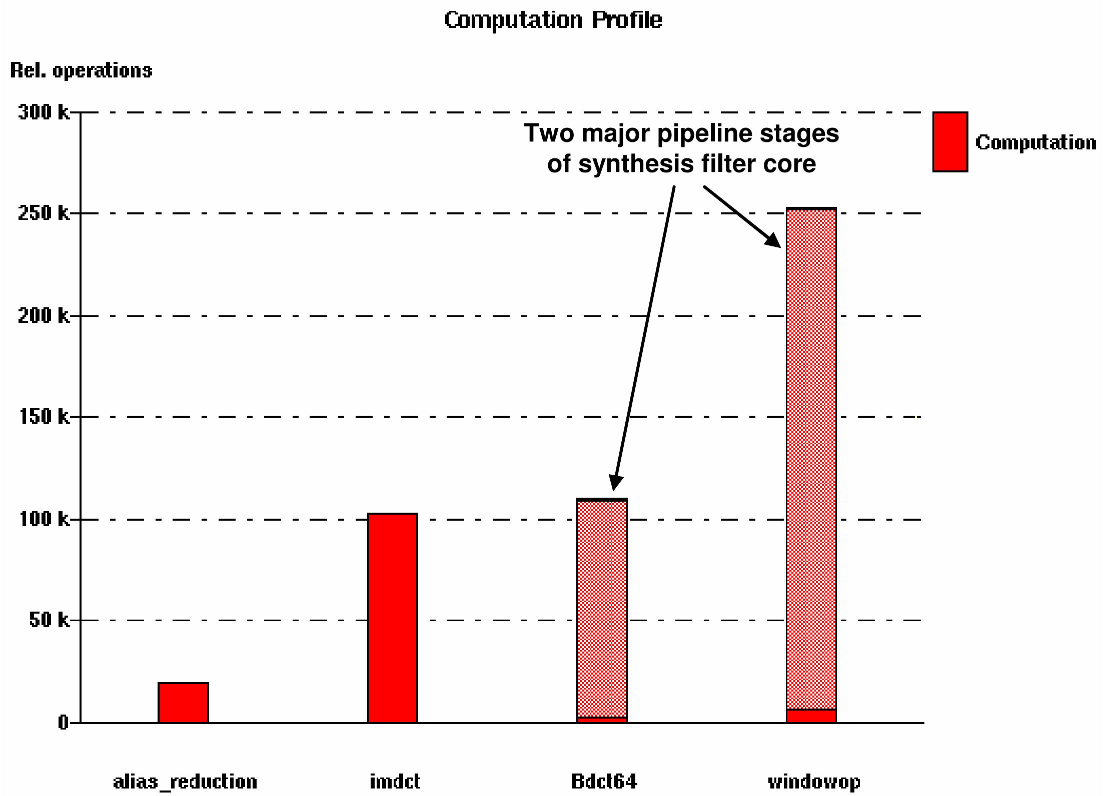



Figure 17: Relative computation complexity of 4 most compute intensive behaviors after pipelining the synthesis filter behavior

### 3.7.5 Summary

At this stage, the granularity in the specification model was satisfactory, promising wider design space exploration. There were 43 behaviors which included 33 leaf behaviors and a total of 130 behavior instances. Granularity alone does not mean good partitioning of the computation load. So, using the profiled result, we identified the computationally expensive behavior and sliced it further into smaller behaviors to get reasonable computational load balance across behaviors. The explicitly exposed parallelism and pipelining enables exploration of faster architectures. So, we decided to conclude the specification model development at this stage and move on to the design space exploration to arrive at an architecture for our design.

In this section, we discussed the procedure adopted to introduce parallelism in the specification model. We also discussed the necessary conditions to be satisfied for parallelizing and pipelining the behaviors. Some of the steps involved are mechanical and can be automated. However, identifying parallelism which is not apparent requires intelligent analysis and needs manual attention. Other than the intelligent analysis required to detect hidden parallelism, most of the code modification tasks can be automated to aid the designer.

## 3.8 Summary and Conclusions

In this section, we discussed the series of changes performed to obtain a ”good” specification model starting from a C specification. The series of steps started with the design of testbench (Section 3.2) which involved separating the design from the stimulus and monitor functionality. Interfaces between each of these behaviors was also designed during this step. In the second step, we introduced more behaviors in the design by converting the major functions into behaviors. This step was discussed in Section 3.4. In Section 3.5, we discussed the task of eliminating the global variables thus exposing the hidden communication in the design. In the fourth step (Section 3.6), we cleaned the specification model to arrive at a ”clean” specification model, in which at every level of hierarchy the behaviors are composed in either sequential, FSM, parallel or pipelined fashion, and all the C code restricted to the leaf behaviors. In the fifth step (Section 3.7), we exposed the concurrency in the design in the form of parallel and pipelined behaviors.
After these changes, we arrived at a final specification model ready to be input to the SCE tool-set for design space exploration and implementation. The Table 2 gives the statistics of the specification model in terms of number of behaviors, number of behaviors under each category (leaf, concurrent, FSM, sequential and pipelined) and number of channel instances.

The steps involved in arriving at a specification model are time consuming making the overall process of writing the specification model slow and hard. Each of these tasks and their development times are listed in the Table 3. The time includes the time for programming followed by compilation using SpecC compiler, verification by simulation and debugging. In general, compilation is not time consuming, however, making the initial C code compile using SpecC compiler takes some effort as discussed in Section 3.2. The development time shown in the table is assuming 5 days a week and 8 man hours per day. In our case, introducing granularity and cleaning of the specification model

| Properties of the specification model | |
|---|---|
| Total number of behaviors | 43 |
| Total number of leaf behaviors | 33 |
| Total number of concurrent behaviors | 4 |
| Total number of FSM behaviors | 5 |
| Total number of pipelined behaviors | 1 |
| Total number of sequential behaviors | 0 |
| Total number of behavior instances | 130 |
| Number of channel instances | 6 |

Table 2: Properties of specification model.

| **Design step** | **Development time** |
|---|---|
| Setting up of initial testbench | 1.5 Weeks |
| Introducing granularity | 5 Weeks |
| Elimination of global variables | 1.5 Weeks |
| Arriving at clean specification model | 3 Weeks |
| Introducing concurrency | 2 Weeks |
| Total | 13 Weeks |

Table 3: Development time for each design step.

took 60% of the development time.
In the process of developing the specification model, we also looked at the possibility of automating these tasks. Some of these tasks involve pure mechanical steps which can be automated to reduce the development time of the specification model. Intelligent analysis, decision making that are necessary for tasks like, handling pointers, identifying hidden parallelism and pipelining and choosing functions for converting to behaviors, determining port-types make the complete automation challenging. However, an interactive tool which automates the mechanical tasks based on the designer's decisions will be very useful.

In the next section, we will detail the next step in the system design process, the design space exploration.

# 4 Design Space Exploration and Implementation

In this section, we will look at the next step in the system level design process, the Design Space Exploration. Because of the complexity involved, arriving at the detailed implementation model from an abstract specification involves multiple exploration and synthesis design steps. Each design step results in an executable design model converting the abstract specification model of the input design into an concrete implementation model. The resulting executable model from a design step can be simulated to verify the functionality and the timing as indicated in the introduction. We used the System on Chip Environment (SCE) [1] for performing the design. The Design flow adopted by SCE can be broadly divided into three design steps, *architecture exploration*, *communication synthesis* and *implementation synthesis*. These refinement steps were discussed in the introduction section of this report and they are discussed below in the context of SCE.

**Architectural exploration and refinement** During this step, processing elements are inserted into the system and functional behaviors are mapped onto the processing elements . The processing elements can be standard components such as generic processor cores, DSPs as well as specific hardware units chosen by the designer from the SCE database. This process involves three major tasks, *Allocation*, *Partitioning* and *Scheduling*. The decision of choosing a component is made by the designer. The user attention is limited to system component allocation followed by decision making based on the simulation and profile results. All the other steps are automated in SCE. This process of architecture refinement results in an architecture model, the first timed model. It takes only computing time into account; all communication between the processing elements is still on an abstract level and system components communicate via abstract channels.

**Communication Exploration and Synthesis** In this step, abstract communication between components is refined into an actual implementation over wires and protocols of system busses. This design step involves three major tasks, *Bus allocation*, *Transducer insertion* and *Channel mapping*. In SCE, the last two steps are fully automated and the designer needs to make decision regarding the allocation and mapping of the busses. The communication synthesis results in the bus functional model, which defines the structure of the system architecture in terms of both components and connections. The bus functional can be simulated and verified for functionality and timing.

**Implementation Synthesis** Implementation synthesis takes the bus functional model as input and synthesizes the software and the hardware components. For hardware components, the RTL code will be generated after the RTL component allocation, their functional mapping and scheduling. As a result of the hardware synthesis, a cycle accurate implementation of each hardware-processing element is created. Similar activities take place during software synthesis. Here specific code for the selected RTOS is created and a target specific assembly code is generated.

For our design example, we performed the above discussed refinement steps and explored few design possibilities. Four such design explorations are described in the following sections.

## 4.1 Complete Software Solution

In this exploration, we choose to have the entire design implemented on one single general purpose processor. Such an implementation is often a good starting point for the embedded system design, since its faster to design and very likely to satisfy chip area and power requirement.
From the SCE library, we choose *Motorola Coldfire* general purpose processor. *Coldfire* is a 32-bit floating point processor with a clock frequency of 66 MHz and 64KB program memory and 128KB of data memory. The whole design was mapped onto the *coldfire* processor and using the automated architecture refinement tool, architecture model was generated. The architecture model is simulated to verify the functionality and the timing. *Coldfire* at 66MHz alone could not meet the computation complexity of the design. So, there was no point in continuing further with this exploration. However, out of curiosity to know the final implementation timing and to understand the design process, we continued further with the exploration by increasing the clock frequency of the *coldfire* to 80MHz. At this new operating frequency, the model satisfied the timing requirement. The concurrent behaviors in the model were scheduled dynamically and scheduling refinement was performed. The resulting model was compiled and simulated to verify the functionality and timing. The execution time after this refinement step increased because all the parallel behaviors were now serialized. Since there was only one component in the whole design, all the communication in the design was mapped onto the system bus of the *coldfire* processor and communication refinement was performed to generate the communication model. The communication model was simulated, and as expected, there was no change in the execution time of the design, as there was no communication overhead.
In the next step, we performed implementation synthesis by synthesizing the C code for the *coldfire* processor. The model was simulated to verify the functionality of the design. This C code can now be compiled for the *coldfire* processor using a cross-compiler.
As mentioned before, this exploration could not satisfy the performance requirement with 66MHz *coldfire* processor. It was pursued by increasing the clock frequency of the processor to 80MHz.

## 4.2 Hardware-Software Solution-1

Since the single software PE solution could not meet our timing requirement, we decided to have hardware acceleration for the time critical blocks of the design. For this exploration, we choose *coldfire* processor with a clock frequency of 66MHz and a hardware PE with a clock frequency of 66MHz.

### 4.2.1 Hardware-Software Partitioning-1: Architecture Refinement

Similar to the single software partitioning, in this exploration, the entire functionality of the decoder was mapped onto the *coldfire* processor. The *Talk2Monitor* behavior, responsible for transferring the decoded audio data to the outside world, was mapped onto hardware PE, *HW0* with a clock

frequency of 66 MHz (same as that of *coldfire* processor). This was done to isolate the decoding functionality and the data transfer logic. The model after architecture refinement is shown in Figure 18. As shown in the figure, there are only two components in this architecture with *coldfire* implementing most of the functionalities, including the compute intensive behaviors, *Synthesis Filter, AliasReduction*, and *IMDCT*. For simplicity, the architecture model omits minor details. It shows only symbolic channels between PEs and omits the PEs implementing the queue channels.
The architecture model was simulated to verify the functionality and the timing. Inspite of having a separate PE for transferring the decoded data to the output, this partition could not satify the timing requirement. We have to increase the clock frequency of the *coldfire* processor to 80MHz to meet the performance requirement. Even though this architecture required extra hardware PE and performed no better than a cheaper single software solution, we decided to pursue this exploration further, as we felt that it was a good idea to isolate the data transfer logic from the decoding functionality. This partition might perform better during later design stages when the models become more accurate in their implementation giving more accurate performance numbers than the estimated numbers given by the architectural model.

#### 4.2.2 Hardware-Software Partitioning-1: Communication Refinement

Since there are only two components in this architecture, all the communication between *coldfire* and *HW0* was mapped onto the system bus of the *coldfire* processor. The communication model for this partition is shown in Figure 19. The *coldfire* acts as the master and *HW0* is the slave of the bus. The communication model was simulated and the functionality and timing were verified.

#### 4.2.3 Hardware-Software Partitioning-1 : Implementation Synthesis

After the communication refinement, the next design step is the RTL synthesis of the hardware PEs. We considered the RTL implementation of the *Talk2Monitor* behavior which was mapped to the hardware PE, *HW0*, during architecture refinement. As discussed in the Section 3.3, the *Talk2Monitor* has 3 child behaviors, *Listen2Decoder, ComputeTime* and *DataTransfer*. To perform the RTL implementation of the *ComputeTime* behavior, we allocated one 32 bit adder unit, one divider, one multiplier unit, and a 32 bit register file of size 8. Using the RTL refinement tool of the SCE, the RTL implementation for the *ComputeTime* behavior was derived. Due to certain limitation in the RTL refinement tool, we could not synthesize the *Listen2Decoder* and *DataTransfer* behavior. Next, the software synthesis for the *coldfire* processor was performed and resultant model was simulated to verify the functionality and timing.

### 4.3 Hardware-Software Solution-2

In the previous exploration, we presented a workable implementation of our design example. In this section, we will discuss another hardware/software architecture which exploits the parallelism in the specification model and derives a different architecture for the MP3 decoder. For this exploration, we used *coldfire* processor and three hardware PEs with operating frequency of 66MHz.

### 4.3.1 Hardware-Software Partitioning-2: Architecture Refinement

In this exploration, the computational hot-spot behavior, *sfilter* was targeted for hardware accelaration. *sfilter* is a parallel composition of two instances of *FilterCore* behavior as shown in Figure 14. Each concurrent instance of *FilterCore*, *filtercore_ch0* and *filtercore_ch1* were mapped to hardware PEs, *HW0, HW1*. To make the decoding functionality independent of the data transfer functionality, the *Talk2Monitor* behavior was mapped to another hardware PE, *HW2*. The rest of the functionality was mapped to the *coldfire* processor. The architecture model generated by the architecture refinement tool is shown in Figure 20. Note that, in the figure, not all the channels in the real model are depicted. Only the user introduced channels and few important channels that represent the communication between various PEs are shown. *Coldfire* communicates with *HW0, HW1,* and *HW2* communicates with *HW0, HW1*. The architecture model in the figure is before performing the scheduling refinement. The pipelined execution in the PEs, *HW0, HW1* is sequentially scheduled and the scheduling refinement is performed using the scheduling refinement tool. The result of scheduling can be seen in Figure 20.
The architecture model was simulated to verify the functionality and timing and this exploration was able to meet our timing constraint.

### 4.3.2 Hardware-Software Partitioning-2: Communication Refinement

Similar to the previous exploration, all the communication between the hardware PEs and the *coldfire* are mapped onto the *coldfire's* main bus. Two busses based on double handshake protocol are allocated and the communication channels between *HW0-HW2* and *HW1-HW2* are mapped onto the respective busses. The communication model generated after the communication refinement is shown in Figure 21. Also, note that the execution within *HW0, HW1* is no longer pipelined as those behaviors were sequentially scheduled during scheduling refinement.

The communication model was simulated to verify the functionality and timing and this exploration also satisfied our timing requirement.

### 4.3.3 Hardware-Software Partitioning-2 : Implementation Synthesis

Due to the lack of few library components for performing the floating point operations and due to certain limitations in the RTL synthesis tool to handle ports of interface type this step could not be performed. So we have to stop at the communication model. However, the software synthesis for the *coldfire* processor was performed and resultant model was simulated to verify the functionality and timing.

## 4.4 Hardware-Software Solution-3

In this section, we will discuss yet another exploration based on hardware-software partitioning. In this partitioning, the parallelism and the pipelining exposed in the specification model are utilized to derive a different, interesting architecture for the MP3 decoder. For this exploration, we used *coldfire* processor and 5 hardware PEs with operating frequency of 66MHz.

### 4.4.1 Hardware-Software Partitioning-3: Architecture Refinement

The computational hot-spots in the design were identified by running the profiler. The profile results are shown in the Figure 22 for few critical behaviors. The four behaviors *setbuffer1, Bdct64, dctoutstage, windowop* were the pipeline stages of the *Synthesis Filter* behavior. Collectively, *Synthesis Filter* was single most compute intensive behavior. We decided to map each pipeline stage of *Synthesis Filter* behavior onto independent hardware units. Since, the computation in the two stages, *setbuffer1, DCTOutStage* was very less compared to *Bdct64*,and *windowop* stages, we decided to map *setbuffer1, Bdct64* onto one PE and *dctoutstage, windowop* onto another hardware PE. The partitioning of the input design and the mapping of each partitions onto the system components is shown in the architecture model in Figure 23. In this partition, general purpose processor, Motorola *Coldfire* is assigned only a partial part of the decoding algorithm and the most compute intensive part which was represented by the behavior *Synthesis Filter* is distributed to 4 hardware PEs (HW0, HW1, HW2, HW3). HW0, HW1 process the first stereo channel and HW2, HW3 process the second stereo channel. The behavior *Talk2Monitor*, which is responsible for combining the outputs of two channels and write to the external device, is mapped to another hardware PE, HW4. Even though, *Talk2Monitor* was not computationally intensive, it was mapped onto an independent unit to separate and parallelize the decoding activity and output data transfer activity. By this partitioning, all the parallelism and pipelining that was exposed in the specification model were utilized. In this figure, to avoid cluttering and confusion, not all the channels in the real model are depicted. However, all the user defined channels (channels in the specification model) and important channels showing communication between various PEs are shown.

### 4.4.2 Hardware-Software Partitioning-3: Communication Refinement

After architecture refinement, busses were allocated. The main bus of the *coldfire* processor served as the system bus. The four hardware PEs (HW0-HW3) communicate with the *coldfire* using this system bus. 4 Busses based on double handshake protocols *HW0_2_HW1, HW1_2_HW4, HW2_2_HW3, HW3_2_HW4* were allocated for the communication between hardware PEs. All the channels in the corresponding paths were mapped onto the respective busses and communication refinement was performed. The resulting communication model is shown in Figure 24. There are totally 5 busses in the design. The *coldfire* processor which acts like a master orchestrating the entire decode operation communicates with PEs HW0 - HW3 using its main bus. HW0 and HW2 communicate the partially processed data to HW1 and HW3 using double handshake bus. HW4 which outputs the data to the external world gets the data from HW2 and HW4 and this communication is through another pair of double handshake bus.

### 4.4.3 Hardware-Software Partitioning-3: Implementation Synthesis

After the communication refinement, the next design step is the RTL synthesis of the hardware PEs. Due to the lack of few library components for performing the floating point operations, this step could not be performed. So we have to stop at the communication model. However, the software

synthesis for the *coldfire* processor was performed and resultant model was simulated to verify the functionality.

## 4.5 Summary and Conclusions

In this section, we discussed the various explorations we performed using SCE. We discussed 4 design implementations, 3 of them based on hardware/software partitioning. The key features of the 4 explorations are given in the Table 4. The table lists the number of software PEs, number of hardware PEs, operating frequency and the number of channels in each exploration. The performance of the various models in each design exploration is discussed in the next section in detail. The automation provided by SCE makes it possible to perform many explorations within a short amount of time. Early feedback about the performance of the design can be obtained by simulating the models at higher abstraction levels. Due to few limitations in the RTL synthesis tool, we could synthesize only a part of our design. In a nutshell, using SCE design environment, optimized architectures, satisfying the design constraints, can be obtained in a short time.

| **Feature** | **Complete Software Solution** | **HW-SW Solution-1** | **HW-SW Solution-2** | **HW-SW Solution-3** |
|---|---|---|---|---|
| No. of General purpose Processors | 1 *Coldfire* | 1 *Coldfire* | 1 *Coldfire* | 1 *Coldfire* |
| No. of hardware PEs | 0 | 1 | 3 | 5 |
| Clock frequency of the PEs | 66 MHz | 80 MHz | 66 MHz | 66 MHz |
| No. of busses | 1 | 1 | 3 | 5 |
| Performance requirement | Not satisfied | Satisfied | Satisfied | Satisfied |

Table 4: Key features of the different explorations.

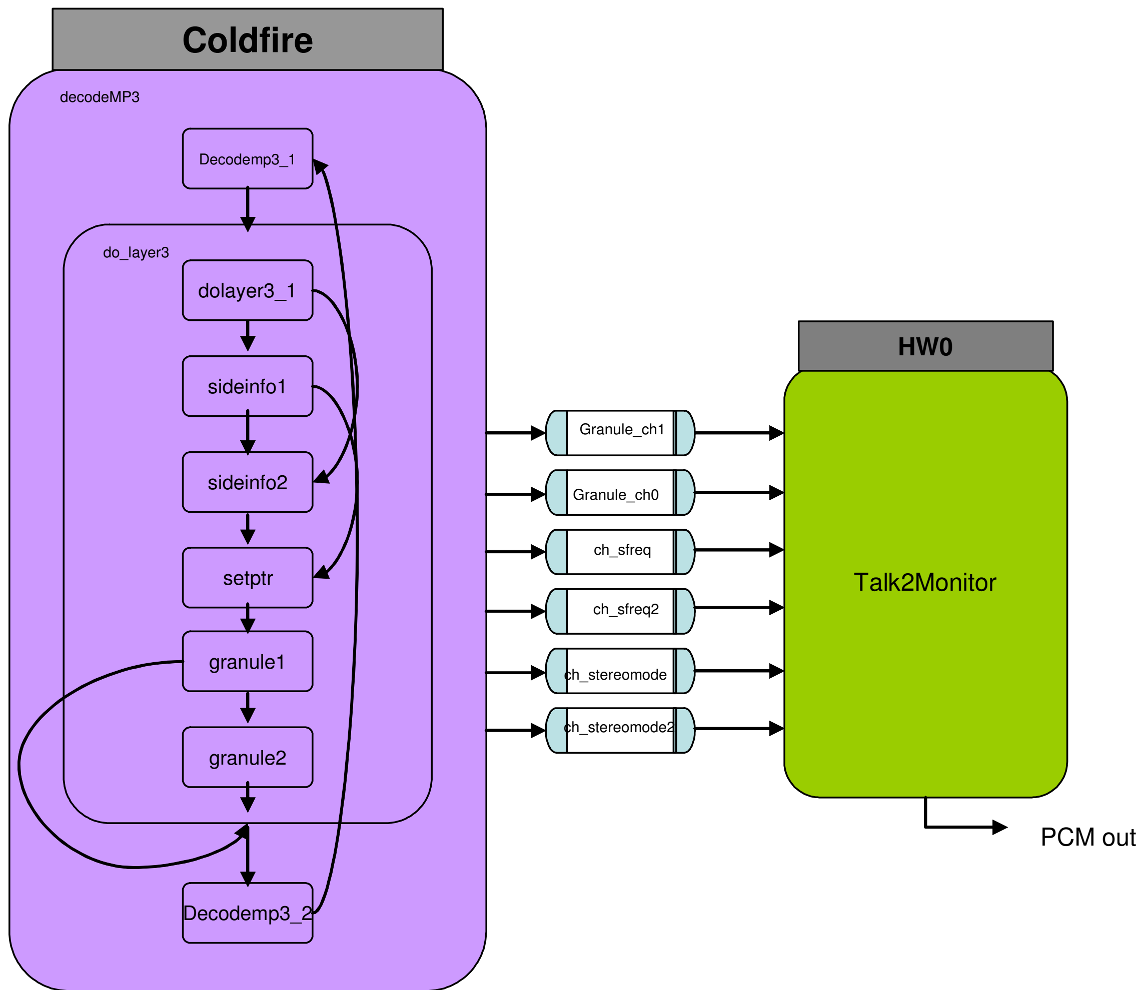



Figure 18: Hardware-software partitioning-1: Architecture model of MP3 decoder

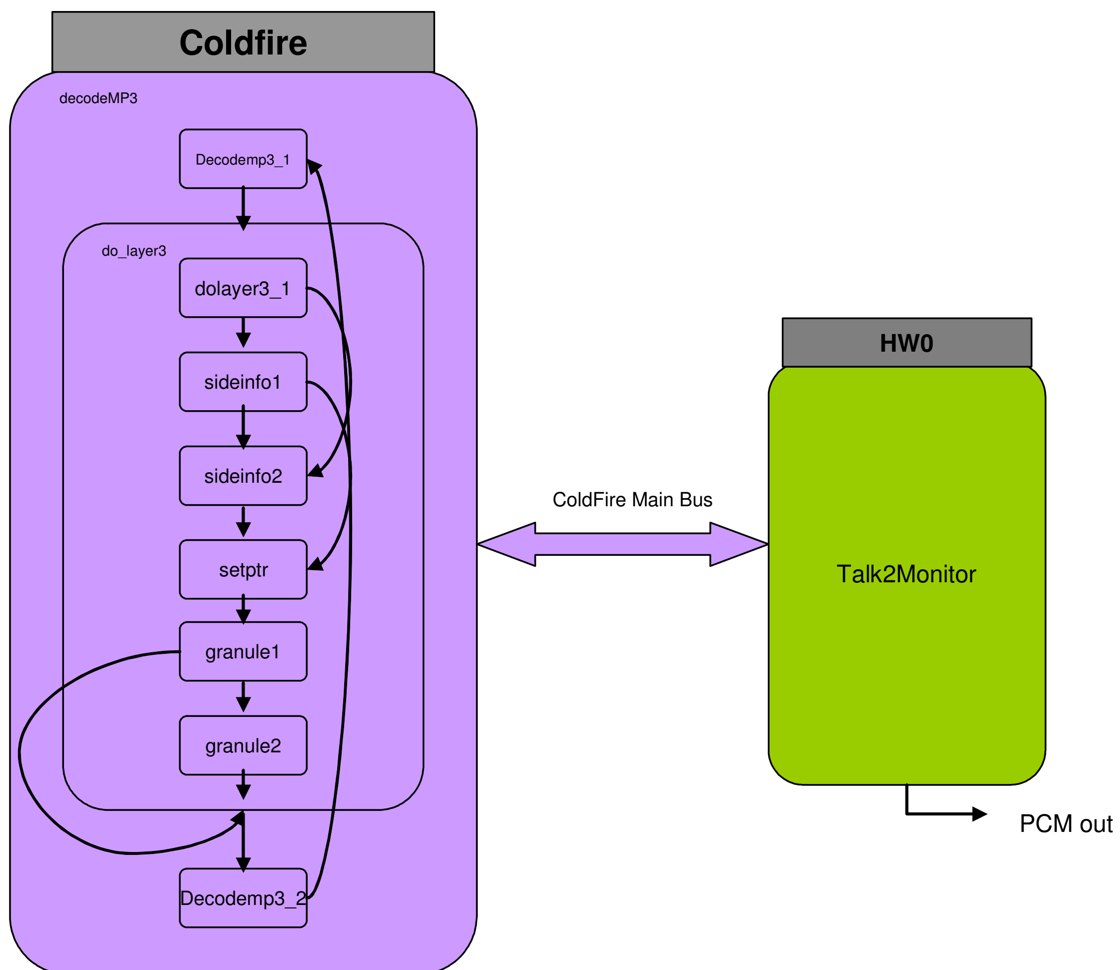



Figure 19: Hardware-software partitioning-1: Communication model of MP3 decoder

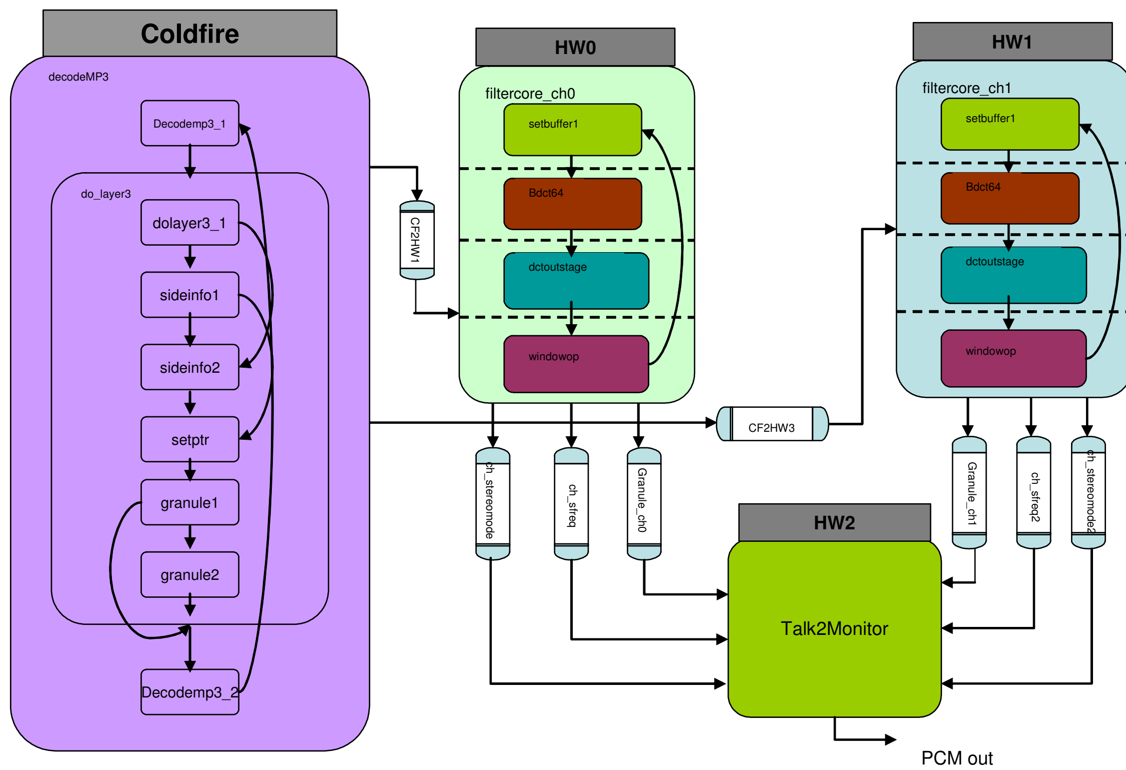



Figure 20: Hardware-software partitioning-2: Architecture model of MP3 decoder

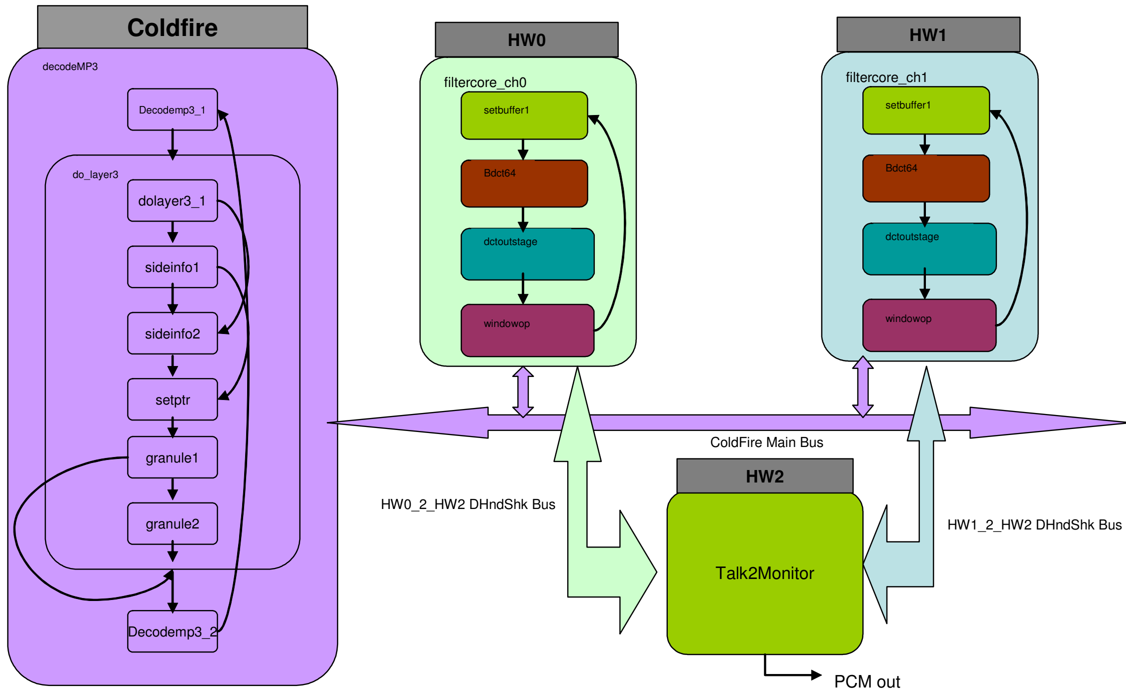



Figure 21: Hardware-software partitioning-2: Communication model of MP3 decoder

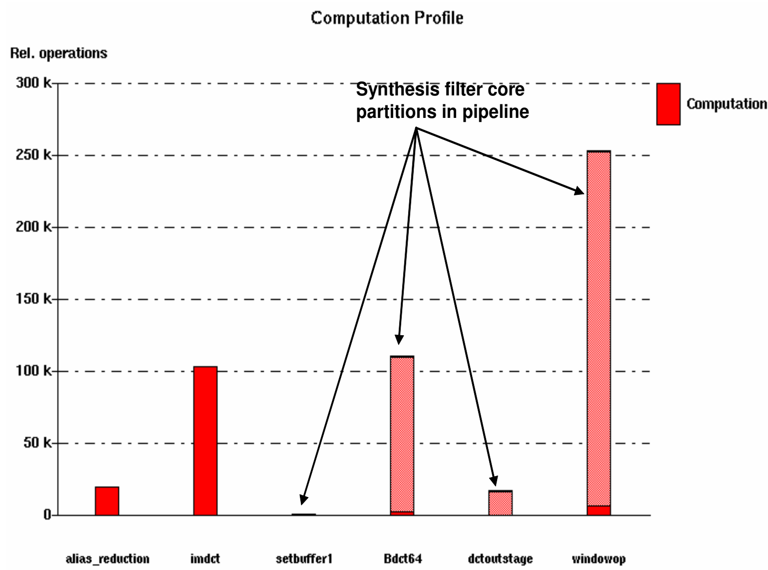



Figure 22: Relative computation complexity of the few behaviors of MP3 decoder specification model

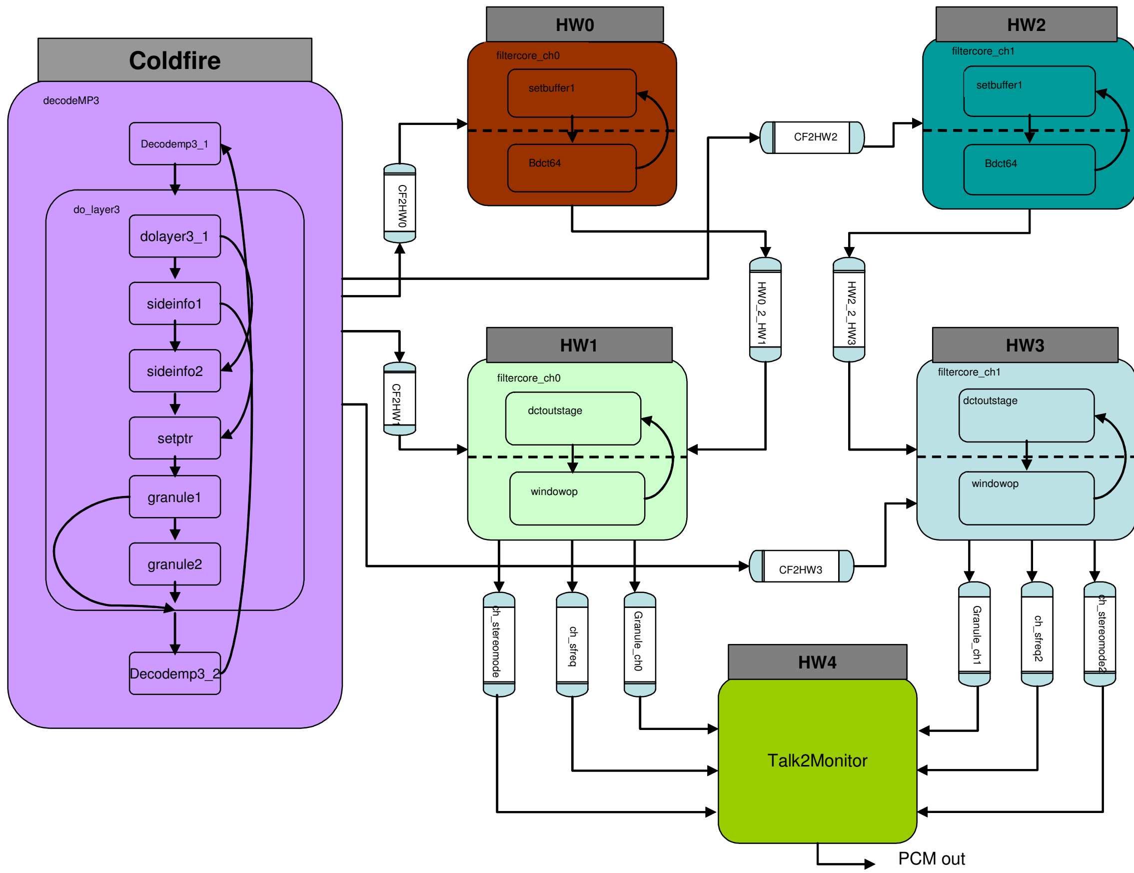



Figure 23: Hardware-software partitioning-3: Architecture model of MP3 decoder (before scheduling refinement)

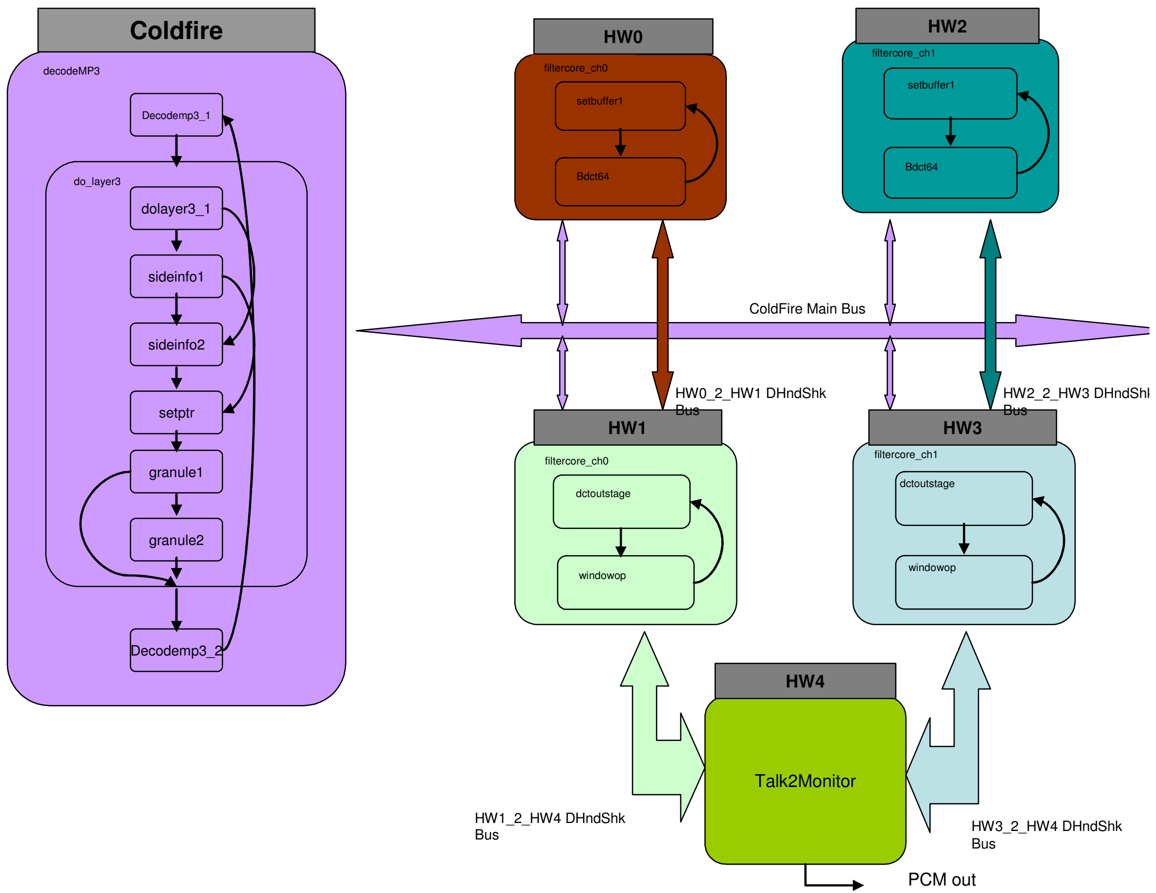



Figure 24: Hardware-software partitioning-3:Communication model of MP3 decoder

# 5 Experimental Results

This section summarizes the experiments and results. There are two aspects of the model to be tested. First, the functionality of the design, and second, the timing of the MP3 decode operation.

## 5.1 Functionality Verification

To test the functionality, we used the testbench described in the Section 3.2. The output PCM file generated by the *monitor* was compared with the one generated by the reference decoder for verifying the functionality.

### 5.1.1 Test Suite

For verifying each of the models, a set of test streams obtained from Fraunhofer Institute [10] were used. These streams and their key properties are given in the Table 5. The table lists the sampling frequency, bitrate at which the streams were encoded, real time length and the type of stream (Stereo/Mono). The sampling frequency is the frequency at which the analog signal was sampled and the bitrate indicates the extent of compression. For example, streams *classic1.mp3, classic2.mp3* have the same sampling frequency but, *classic1* is coded at a higher bitrate than *classic2*. This indicates that *classic1* is less compressed than *classic2* and hence of better quality.

| Properties of the test MP3 streams | | | | |
|---|---|---|---|---|
| **Title** | **Sampling Frequency** | **Total Bitrate** | **Real Time length** min:sec | **Stereo/Mono** |
| funky.mp3 | 44.1 KHz | 96 Kbits/Sec | 1:02 | Stereo |
| spot1.mp3 | 44.1 KHz | 96 Kbits/Sec | 0:10 | Stereo |
| spot2.mp3 | 44.1 KHz | 96 Kbits/Sec | 0:11 | Stereo |
| spot3.mp3 | 44.1 KHz | 96 Kbits/Sec | 0:11 | Stereo |
| classic1.mp3 | 22.05 KHz | 56 Kbits/Sec | 0:19 | Stereo |
| classic2.mp3 | 22.05 KHz | 48 Kbits/Sec | 0:20 | Stereo |

Table 5: Properties of test streams.

## 5.2 Timing Verification

Apart from decoding correctly to produce bit accurate results, the decoder is expected to deliver the output PCM samples at the correct bitrate. This rate depends on the sampling frequency of the input MP3 stream and puts a timing constraint on the decoder. The decoder is required to decode and output exactly at this rate. If the output rate control logic is not part of the decoder, then the decoder can generate output faster and expect the external logic to take care of the rate control. However, in our design, as this logic was part of the design the decoder was expected to deliver the decoded data exactly at this rate. The specification model is untimed and will run in zero simulation time.

Since, our design included this output rate control logic, the delivery of the decoded samples to the output device would happen at a controlled rate and hence even the specification model would take finite non-zero simulation time to run. In order to measure the actual time to decode without considering the explicit delay introduced by rate control logic, we disabled the delays in the rate control logic. This change in the model was done only for the timing measurement. As we go down the abstraction level performing each refinement steps, the decode operation takes non-zero finite time.
The average estimated decode time per frame of audio data for each partition discussed in Section 4 and for each refined model is given in the tables Table 6, tables Table 7, Table 8, Table 9 and Table 10. The results are obtained by simulating each model with one of the test streams, *spot1.mp3*. Also provided in the tables are the deadline for decoding each frame of audio data for the test stream *spot1.mp3* and the clock frequency of the PEs used in the design. The *Estimated Initial Latency* is the time it takes to decode the very first sample of the very first frame of audio data. The last column in the tables gives the ratio of the decode time to the stipulated deadline. A value of greater than 100% implies that the model could not meet the performance requirement.
For the single software solution, two tables are given, table Table 6 gives the decode times when the clock frequency of the *coldfire* processor is 66 MHz. Clearly, this single software design solution could not meet the stipulated deadline taking 27.15 msecs to complete the decode of single frame. The second table, Table 7 is obtained with *coldfire* processor at 80 MHz and it meets the stipulated deadline by taking 22.41 msecs to decode a frame. In general, the simulation time increases with each model. However, since there exists no communication overhead, the communication model in this case, does not show increase in execution time compared to the architecture model.

The Table 8 gives timings for the partition in Figure 18. This architecture meets the stipulated deadline and its estimated decoding times are same as that of the pure software exploration. The implementation model for this architecture contains synthesized software in C and RTL implementation of only a subset of the functionality mapped to hardware PE.
The timing of the third architecture (Figure 20), composed of 1 *coldfire* processor and 3 hardware PEs at 66 MHz, is given in Table 9. This architecture meets the performance requirement even at a lower clock frequency of 66 MHz because of the hardware accelaration of the critical computational blocks.
The final architecture of Figure 23 has the most complex architecture with 1 *coldfire* processor and 5 hardware PEs, each operating at 66 MHz. This architecture exploits both the parallelism and the pipelining in the application. Though, it meets the stipulated deadline, its performance is not as good as the third architecture (Figure 20).

| Timing of Various Models of Single Software partition<br>*Deadline to decode one frame of spot1.mp3* = 26.12 msec<br>*Operating clock frequency of the processor* = 66MHz | | | |
|---|---|---|---|
| **Model** | **Estimated Initial Latency** | **Estimated Time to Decode a Frame** | **Ratio of Decode Time to Deadline** |
| Specification Model | 0.0 msec | 0.0 msec | – |
| Architecture Model | 25.03 msec | 12.80 msec | 49.0% |
| Scheduled Architecture Model | 27.15 msec | 27.17 msec | 104% |
| Communication Model | 27.15 msec | 27.17 msec | 104% |
| Implementation Model | 27.15 msec | 27.17 msec | 104% |

Table 6: Timing of various models of Software partition.

| Timing of Various Models of Single Software partition<br>*Deadline to decode one frame of spot1.mp3* = 26.12 msec<br>*Operating clock frequency of the processor* = 80 MHz | | | |
|---|---|---|---|
| **Model** | **Estimated Initial Latency** | **Estimated Time to Decode a Frame** | **Ratio of Decode Time to Deadline** |
| Specification Model | 0.0 msec | 0.0 msec | – |
| Architecture Model | 20.65 msec | 10.56 msec | 40.4% |
| Scheduled Architecture Model | 22.40 msec | 22.41 msec | 85.17% |
| Communication Model | 22.40 msec | 22.41 msec | 85.17% |
| Implementation Model | 22.40 msec | 22.41 msec | 85.17% |

Table 7: Timing of various models of Software partition (Working solution).

| Timing of Various Models of Hardware-Software partition-1<br>*Deadline to decode one frame of spot1.mp3* = 26.12 msec<br>*Operating clock frequency of HW and SW PEs* = 80 MHz | | | |
|---|---|---|---|
| **Model** | **Estimated Initial Latency** | **Estimated Time to Decode a Frame** | **Ratio of Decode Time to Deadline** |
| Specification Model | 0.0 msec | 0.0 msec | – |
| Architecture Model | 20.65 msec | 10.56 msec | 40.4% |
| Scheduled Architecture Model | 22.14 msec | 22.41 msec | 85.8% |
| Communication Model | 22.14 msec | 22.45 msec | 85.8% |
| Implementation Model | 22.14 msec | 22.45 msec | 85.8% |

Table 8: Timing of various models of Hardware-Software partition-1.

| Timing of Various Models of Hardware-Software partition-2<br>*Deadline to decode one frame of spot1.mp3* = 26.12 msec<br>*Operating clock frequency of HW and SW PEs* = 66 MHz | | | |
|---|---|---|---|
| **Model** | **Estimated Initial Latency** | **Estimated Time to Decode a Frame** | **Ratio of Decode Time to Deadline** |
| Specification Model | 0.0 msec | 0.0 msec | – |
| Architecture Model | 24.78 msec | 6.37 msec | 24.4% |
| Scheduled Architecture Model | 26.36 msec | 9.87 msec | 37.8% |
| Communication Model | 26.42 msec | 10.02 msec | 38.4% |
| Implementation Model (Synthesized software only) | 26.42 msec | 10.02 msec | 38.4% |

Table 9: Timing of various models of Hardware-Software partition-2.

| Timing of Various Models of Hardware-Software partition-3<br>*Deadline to decode one frame of spot1.mp3* = 26.12 msec<br>*Operating clock frequency of HW and SW PEs* = 66 MHz | | | |
|---|---|---|---|
| **Model** | **Estimated Initial Latency** | **Estimated Time to Decode a Frame** | **Ratio of Decode Time to Deadline** |
| Specification Model | 0.0 msec | 0.0 msec | – |
| Architecture Model | 24.82 msec | 7.78 msec | 29.8% |
| Scheduled Architecture Model | 26.41 msec | 10.99 msec | 42.1% |
| Communication Model | 26.41 msec | 11.21 msec | 42.9% |
| Implementation Model (Synthesized software only) | 26.41 msec | 11.21 msec | 42.9% |

Table 10: Timing of various models of Hardware-Software partition-3.

# 6 Summary and Conclusions

In this project, we adopted the SpecC design methodology to implement a System on a Chip MP3 decoder. We used the SpecC based System on a Chip Environment (SCE) tool for performing the design exploration and implementation. We choose SpecC, as a language to implement the specification model, as it best suits for describing systems involving both hardware and software components. Being a true superset of ANSI-C, it has a natural suitability to describe software components. It has added features to support hardware description. It also includes constructs to support hierarchical description of system components. With all these features, the designer has flexibility to choose and describe the system at any desired level of abstraction. SpecC is easy to learn and a clean language. Anyone with background knowledge of C can learn SpecC quickly. The availability of SpecC based SCE for performing design space exploration and synthesis was another main reason for choosing SpecC as the specification language.

As an input to the SCE, we provided the Specification model of the MP3 decoder written in SpecC SLDL. SCE provides designer a way to deal with the complexity of the design by having the designer handle the design complexity at a higher level of abstraction. It provides complete design automation with occasional manual intervention for decision making and controllability. The user intervention is restricted to the allocation of processing elements, busses, memories and mapping of the behaviors and channels onto the allocated components. The tool allows an easy design space exploration. It enables the designer to estimate performance during the early stages of the design and additionally allows the early pruning of the design space.

With SCE tool available for doing all the exploration and refinement, the main responsibility of the designer is to write a good, clean specification model. We spent 13 man-weeks to convert an C code into the Specification model. Though the starting C specification was good enough to be a general software program to run on servers and desktop systems, it was not suitable to be a SoC specification. A noticeable effort had to be spent in writing a specification model to eliminate the issues like usage of global variables, lack of separation of communication and computation blocks, lack of behavioral hierarchy. We introduced sufficient granularity in the model to facilitate good number of explorations. We separated the computation and communication blocks by having all the computation captured in behaviors and all communication using channels. We exposed the concurrency in the design by having parallel and pipelined execution of behaviors.
In this report, we also proved the power and usefulness of automated SoC design methodology, SCE. SCE lets designer to focus on the development of the specification model by taking care of all the refinement steps through an automated tool set.
In this report, we defined a ”good” specification model and described a step by step procedure to arrive at a good, clean specification model. We identified various tasks that can be automated fully or partially automatable. An interactive tool which can perform automatic refinement based on designer decisions will be a good replacement for the manual effort. Since writing a specification model is a time consuming effort, it will be most useful to focus the future effort in the direction of automating the process of writing the specification model from C code. Having such a tool would

be next logical step towards having an end to end system design automation. Such a tool would obviate the user to learn new System Level languages like SpecC and the system specification could start with a more abstract level in C.

# References

[1] Samar Abdi, Junyu Peng, Haobo Yu, Dongwan Shin, Andreas Gerstlauer, Rainer Dömer, and Daniel Gajski. System-on-chip environment (SCE version 2.2.0 beta): Tutorial. Technical Report CECS-TR-03-41, Center for Embedded Computer Systems, University of California, Irvine, July 2003.

[2] Samar Abdi, Dongwan Shin, and Daniel D. Gajski. Automatic communication refinement for system level design. In *Proceedings of the Design Automation Conference (DAC)*, Anaheim, CA, June 2003.

[3] Rainer Dömer Andreas Gerstlauer, Kiran Ramineni and Daniel D. Gajski. System-on chip specification style guide. Technical Report CECS-TR-03-21, Center for Embedded Computer Systems, University of California, Irvine, June 2003.

[4] Competitive audio compression formats. http://www.litexmedia.com/article/audio_formats.html.

[5] Felice Balarin, Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Ellen Sentovich, Kei Suzuki, and Bassam Tabbara. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer Academic Publishers, 1997.

[6] Lucai Cai, Andreas Gerstlauer, and Daniel D. Gajski. Retargetable profiling for rapid, early system-level design space exploration. Technical Report CECS-TR-04-04, Center for Embedded Computer Systems, University of California, Irvine, March 2004.

[7] Wander O. Cesário, Damien Lyonnard, Gabriela Nicolescu, Yanick Paviot, Sungjoo Yoo, Ahmed A. Jerraya, Lovic Gauthier, and Mario Diaz-Nava. Component-based design approach for multicore socs. June 2002.

[8] Rainer Dömer, Andreas Gerstlauer, and Daniel Gajski. *SpecC Language Reference Manual, Version 2.0*. SpecC Technology Open Consortium, http://www.specc.org, December 2002.

[9] Petru Eles, Krzysztof Kuchcinski, and Zebo Peng. *System Synthesis with VHDL*. Kluwer Academic Publishers, December 1997.

[10] Fraunhofer mp3 streams. ftp://ftp.fhg.de/pub/layer3/mp3-bitstreams.tgz.

[11] Daniel D. Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.

[12] Andreas Gerstlauer, Lukai Cai, Dongwan Shin, Haobo Yu, Junyu Peng, and Rainer Dömer. *SCE Database Reference Manual, Version 2.2.0 beta*. Center for Embedded Computer Systems, University of California, Irvine, July 2003.

[13] Andreas Gerstlauer, Rainer Dömer, Junyu Peng, and Daniel D. Gajski. *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, 2001.

[14] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[15] International Organization for Standardization (ISO). *Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s - Part 3: Audio*, first edition, 1993. ISO/IEC 11172-3 Standard.

[16] K.Brandenburg and H.Popp. *An introduction to MPEG Layer-3*. Fraunhofer Instirut fur Integrierte Schaltungen (IIS), EBU Technical Review, June 2000.

[17] David Ku and Giovanni De Micheli. HardwareC - a language for hardware design, version 2.0. Technical Report CSL-TR-90-419, Computer Science Laboratory, April 1990.

[18] Krister Lagerstrom. Design and implementation of an MPEG-1 layer-3 audio decoder, Masters Thesis, May 2001.

[19] David J. Lilja and Sachin S. Sapatnekar. *Designing Digital Computer Systems with Verilog*. Cambridge University Press, December 2004.

[20] MPG123. http://www.mpg123.de/mpg123/mpg123-0.59r.tar.gz.

[21] Achim Österling, Thomas Brenner, Rolf Ernst, Dirk Herrmann, Thomas Scholz, and Wei Ye. The COSYMA system. In Jorgen Staunstrup and Wayne Wolf, editors, *Hardware/Software Co-Design: Principles and Practice*. Kluwer Academic Publishers, 1997.

[22] Davis Pan. A tutorial on mpeg/audio compression. *IEEE Multimedia*, 2(2):60–74, Summer 1995.

[23] Junyu Peng, Samar Abdi, and Daniel D. Gajski. Automatic model refinement for fast architecture exploration. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Bangalore, India, January 2002.

[24] Nirupama Srinivas Pramod Chandraiah, Hans Gunar Schirner and Rainer Dömer. System-on chip modeling and design, a case study on mp3 decoder. Technical Report CECS-TR-04-17, Center for Embedded Computer Systems, University of California, Irvine, June 2004.

[25] Dongwan Shin, Andreas Gerstlauer, Rainer Dömer, and Daniel D. Gajski. C-based interactive RTL design environment. Technical Report CECS-TR-03-42, Center for Embedded Computer Systems, University of California, Irvine, December 2003.

[26] Fpga design cycle time reduction and optimization. http://www.xilinx.com/xcell/xl29/xl29_20.pdf.

[27] Frank Vahid and Tony Givargis. Digital camera example. In *Embedded System Design: A Unified Hardware/Software Introduction*. John Wiley & Sons, Inc., 2002.

[28] Frank Vahid, Sanjiv Narayan, and Daniel D. Gajski. SpecCharts: A VHDL frontend for embedded systems. *IEEE Transactions on Computer-Aided Design of Intergrated Circuits and Systems (TCAD)*, 14(6):694–706, June 1995.

[29] Haobo Yu, Rainer Dömer, and Daniel Gajski. Embedded software generation from system level design languages. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Yokohama, Japan, January 2004.