Design of a GSM Vocoder using SpecC Methodology

Technical Report ICS-99-11
February 26, 1999

Andreas Gerstlauer, Shuqing Zhao, Daniel D. Gajski

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{gerstl, szhao, gajski}@ics.uci.edu

Arkady M. Horak

Motorola Semiconductor Products Sector
System on a Chip Design Technology
Austin, TX 78731, USA

RVKA30@email.sps.mot.com

Design of a GSM Vocoder using SpecC Methodology

Technical Report ICS-99-11
February 26, 1999

Andreas Gerstlauer, Shuqing Zhao, Daniel D. Gajski

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{gerstl, szhao, gajski}@ics.uci.edu

Arkady M. Horak

Motorola Semiconductor Products Sector
System on a Chip Design Technology
Austin, TX 78731, USA

RVKA30@email.sps.mot.com

Abstract

This report describes the design of a voice encoder/decoder (vocoder) based on the European GSM standard employing the system-level design methodology developed at UC Irvine. The project is a result of a cooperation between UCI and Motorola to demonstrate the SpecC methodology. Starting from the abstract executable specification written in SpecC different design alternatives concerning the system architecture (components and communication) are explored and the vocoder specification is gradually refined and mapped to a final HW/SW implementation such that the constraints are satisfied optimally. The final code for downloading onto the processors and the RTL hardware descriptions for synthesis of the ASICs are generated for the software and hardware parts, respectively.

Contents

List of Figures

Design of a GSM Vocoder using SpecC Methodology

A. Gerstlauer, S. Zhao, D. Gajski

Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA

A. Horak

Motorola Semiconductor Products Sector
System on a Chip Design Technology
Austin, TX 78731, USA

Abstract

This report describes the design of a voice encoder/decoder (vocoder) based on the European GSM standard employing the system-level design methodology developed at UC Irvine. The project is a result of a cooperation between UCI and Motorola to demonstrate the SpecC methodology. Starting from the abstract executable specification written in SpecC different design alternatives concerning the system architecture (components and communication) are explored and the vocoder specification is gradually refined and mapped to a final HW/SW implementation such that the constraints are satisfied optimally. The final code for downloading onto the processors and the RTL hardware descriptions for synthesis of the ASICs are generated for the software and hardware parts, respectively.

1 Introduction

For the near future, the recent predictions and roadmaps on silicon semiconductor technology all agree that the number of transistors on a chip will keep growing exponentially according to Moore's Law, pushing technology towards the System-On-Chip (SOC) era. However, we are increasingly experiencing a productivity gap where the chip complexity that can be handled by current design teams falls short of the possibilities offered by technology advances. Together with growing time-to-market pressures this drives the need for innovative measures to increase design productivity by orders of magnitude.

It is commonly agreed on that the solutions for achieving such a leap in design productivity lie in a shift of the focus of the design process to higher levels of abstraction on the one hand and in the massive reuse of predesigned, complex system components (intellectual property, IP) on the other hand. In order to be successful, both concepts eventually require the adoption of new methodologies for system design in the companies, backed-up by the availability of a corresponding set of system-level design automation tools.

At UC Irvine, evolving from a background in behavioral or high-level synthesis, research has been conducted in this area for a number of years now. An IP-centric system-level design methodology including a new system specification language called SpecC was developed. Currently, work in progress at UCI deals with the creation of algorithms and tools for automation of the steps in this design process.

UCI, in cooperation with Motorola, launched a project in June 1998 with the goal of demonstrating the SpecC methodology on a real design example. For this purpose, the design and implementation of a voice encoder/decoder (vocoder) for cellular applications was chosen as an example. The actual design of the vocoder was done at UCI whereas Motorola is responsible for the final integration, implementation, and eventually manufacturing of the chip. In this report we document the design of the vocoder chip employing the SpecC methodology as a result of the project part done at UC Irvine.

1.1 GSM Enhanced Full Rate Vocoder

The vocoder used in this project is based on the standard for speech coding and compression in the European cellular telephone network system GSM (Global System for Mobile Communications). The codec scheme was originally developed by Nokia and the University of Sherbrooke [10]. The so called *Enhanced Full Rate (EFR) speech transcoding* is now standardized by the European Telecommunication Standards Institute (ETSI) as GSM 06.60 [9]. In addition, the same codec has also been adopted as a standard for the American PCS 1900 system by the Telecommunications Industry Association (TIA) [11]. In general, this codec scheme and variations thereof are widely used

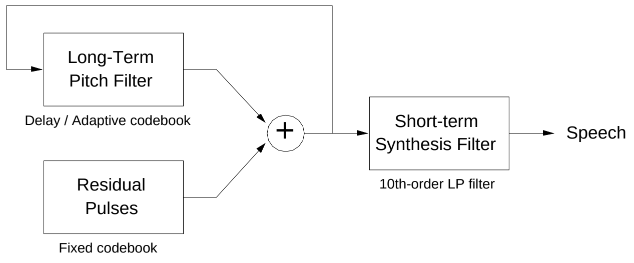


Figure 1: Speech synthesis model.

in voice compression and encoding for speech transmission (e.g. [12]).

1.1.1 Human Vocal Tract

Conceptually, the main idea of a speech synthesis vocoder is based on modeling the human vocal tract using digital signal processing (DSP) techniques in order to synthesize or recreate speech at the receiving side.

Human speech is produced when air from the lungs is forced through an opening between the two vocal folds called the glottis. Tension in the vocal chords caused by muscle contractions and forces created by the turbulence of the moving air force the glottis to open and close at a periodic rate. Depending on the physical construction of the vocal tract, these oscillations occur between 50 to 500 times per second. The oscillatory sound waves are then modified when they travel through the throat, over the tongue, through the mouth and over the teeth and lips.

1.1.2 Speech Synthesis Model

The model assumes that the speech signal is produced by a buzzer at the end of a tube. The glottis produces the buzz which is characterized by intensity (loudness) and frequency (pitch). The vocal tract (throat and mouth) is modeled by a system of connected lossless tubes.

Figure 1 shows the GSM vocoder speech synthesis model. A sequence of pulses is combined with the output of a long-term pitch filter. Together, they model the buzz produced by the glottis and they build the excitation for the final speech synthesis filter which in turn models the throat and mouth as a system of lossless tubes.

The initial sequence of so called *residual pulses* is constructed by assembling predefined pulse waveforms taken out of a given, fixed codebook. The codebook contains a selection of so called *fixed code vectors* which are basically fixed pulse sequences with varying frequency. In addition, the pulse intensities are scaled by applying a variable gain factor.

The output of the long-term pitch filter is simply a previous excitation sequence, modified by scaling it with a variable gain factor. The amount by which excitations are delayed in the pitch filter is a parameter of the speech synthesis model and can vary over time. The long-term pitch filter is also referred to as adaptive codebook since the history of all past excitations basically forms a codebook with varying contents out of which one past excitation sequence, the so called *adaptive code vector*, is chosen.

Finally, the excitation which is constructed by adding fixed and adaptive codebook vectors is passed through the short-term speech synthesis filter which simulates a system of connected lossless tubes. Technically, the short-term filter is a tenth order linear prediction filter meaning that its output is a linear combination (linear weighted sum) of ten previous inputs. The ten linear prediction coefficients are intended to model the reflections and resonances of the human vocal tract.

1.1.3 Speech Encoding and Decoding

Instead of transmitting compressed speech samples directly, the input speech samples are analyzed in order to extract the parameters of the speech synthesis model which are then transmitted to the receiving side where they are in turn used to synthesize the reconstructed speech.

On the encoding side, the input speech is analyzed to estimate the coefficients of the linear prediction filter, removing their effects and estimating the intensity and frequency. The process of removing the linear prediction effects is performed by inverse filtering of the incoming speech. The remaining signal called the *residual* is then used to estimate the pitch filter parameters. Finally, the pitch filter contribution is removed in order to find the closest matching residual pulse sequence in the fixed codebook.

At the receiver, the transmitted parameters are decoded, combining the selected fixed and adaptive code vectors to build the short-term excitation. The linear prediction coefficients are decoded and the speech is synthesized by passing the excitation through the parameterized short-term filter.

All together, this speech synthesis method has the advantages of achieving a high compression ratio since it tries to transmit only the actual information inherent in the speech signal. All the redundant relationships which are due to the way the human vocal tract is organized are captured by the filters of the speech

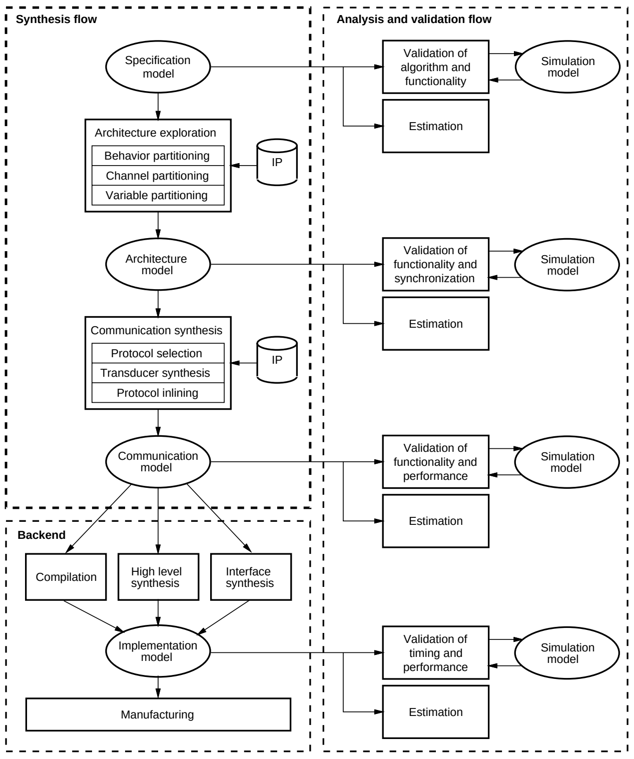


Figure 2: SpecC methodology.

synthesis model. The vocal tract model provides an accurate simulation of the real world and is quite effective in synthesizing high quality speech. In addition, encoding and decoding are relatively efficient to compute.

1.2 System-Level Design

1.2.1 SpecC Methodology

The system-level design methodology which has been developed at UC Irvine is shown in Figure 2 [3, 5]. The system methodology starts with an *executable specification*. This specification describes the functionality as well as the performance, power, cost and other constraints of the intended design. It does not make any presumptions regarding the implementation details.

As shown in Figure 2, the synthesis flow of the codesign process consists of a series of well-defined design steps which will eventually map the executable specification to the target architecture. In this methodology, we distinguish two major system level tasks, namely architecture exploration and communication synthesis.

Architecture exploration includes the design steps of allocation and partitioning of behaviors, channels and variables. *Allocation* determines the number and the types of the system components, such as processors, ASICs and busses, which will be used to implement the system behavior. Allocation includes the reuse of intellectual property (IP), when IP components are selected from the component library.

Behavior partitioning distributes the behaviors (or processes) that comprise the system functionality amongst the allocated processing elements, whereas *variable partitioning* assigns variables to memories and *channel partitioning* assigns communication channels to busses. *Scheduling* is used to determine the order of execution of the behaviors assigned to the processors.

Architecture exploration is an iterative process whose final result is the definition of the system architecture. In each iteration, estimators are used to evaluate the satisfaction of the design constraints. As long as any constraints are not met, component and connectivity reallocation is performed and a new architecture with different components, connectivity, partitions, schedules or protocols is evaluated.

After the architecture model is defined, *communication synthesis* is performed in order to obtain a design model with refined communication. The task of communication synthesis includes the selection of communication protocols, synthesis of interfaces and transducers, and inlining of protocols into synthesizable components. Thus, communication synthesis refines the abstract communications between behaviors into an implementation.

It should be noted that the design decisions in each of the tasks can be made manually by the designer, e. g. by using an interactive graphical user interface, as well as by automatic synthesis tools.

The result of the synthesis flow is handed-off to the backend tools, shown in the lower part of Figure 2. Code for the software and hardware parts is generated automtatically. The software part of the hand-off model consists of C code and the hardware part consists of behavioral VHDL or C code. The backend tools include compilers and a high-level synthesis tool. The compilers are used to compile the software C code for the processor onto which the code is mapped. The high-level synthesis tool is used to synthesize the functionality mapped to custom hardware and the interfaces needed to connect different processors, memories and IPs.

During each design step, the model is statically analyzed to estimate certain quality metrics such as performance, cost and power consumption. This design model is also used in simulation to verify the correctness of the design at the corresponding step.

For example, at the specification stage, the simulation model is used to verify the functional correctness of the intended design. After architecture exploration, the simulation model will verify the synchronization between behaviors on different processing elements (PEs). After communication synthesis, the simulation model is used to verify the performance of the system including computation and communication.

1.2.2 SpecC Language

The methodology described in the previous section is supported by a new system-level description and specification language called SpecC [1, 2, 4] which was developed at UC Irvine in realization that existing languages lack many of the features needed for system-level design [3]. At all stages of the SpecC methodology, the current state of the design is represented by a model described in the SpecC language. In this homogeneous approach transformations are made on the SpecC description in contrast to a heterogeneous approach where each step also transforms the design into a new language, ending up with a mix of design representations at different stages of the process.

SpecC is being built as a superset of ANSI-C [6] which allows easy reuse of the existing algorithmic and behavioral C descriptions that are common in todays industrial practice. SpecC contains all of the features required to support system-level design including IP integration in general and the SpecC methodology in particular:

- Structural and behavioral hierarchy.
- Concurrency.
- Communication with explicit separation of computation (*behavior*) from communication (*channel*).
- Synchronization.
- Exception handling (traps and interrupts).
- Timing.
- Explicit state transitions (FSM modeling).

All these features are explicitly specified in a clear and orthogonal manner which makes it easy to understand and analyze given SpecC descriptions both for humans and for automation tools. This is an essential requirement to enable successful design automation and synthesis of high-quality results.

SpecC descriptions are translated into a C++ model by the SpecC compiler. These C++ descriptions are then in turn compiled into a native executable for simulation and verification. This results in very high simulation speeds due to the fact that the design is compiled instead of interpreted.

1.3 Overview

The rest of the report is organized as follows: Section 2 describes the specification of the vocoder standard in SpecC, including a more detailed description of the functionality and other requirements. In Section 3 the different steps performed during architectural exploration are shown. Starting with a general overview the final vocoder architecture is developed. The process of mapping the abstract communication onto real protocols, busses, etc. is described in Section 4. Communication synthesis also shows the requirements for successful integration of intellectual property (IP). Again, a general discussion is followed by the vocoder specifics. Finally, Section 5 concentrates on the synthesis of the system's software and hardware parts in the backend. Section 6 concludes the report with a summary.

2 Specification

2.1 General

The first step in any design process is the specification of the system requirements. This includes both functionality as well as other requirements like timing, power consumption or area.

2.1.1 Formal, Executable Specification

As outlined in the introduction, in the SpecC system the specification is formally captured and written in the SpecC language. As opposed to the informal specifications (e.g. in plain English) that have been commonly used in the past, a formal specification of the system has two main advantages:

1. It is executable for simulation and verification of the desired functionality or for feasibility studies at an early stage.
2. The formal, executable specification directly serves as an input to the following synthesis and exploration stages that eventually lead to the final implementation without the need for time-consuming modifications or translations into other languages and models.

2.1.2 Modeling Guidelines

The initial SpecC specification should model the system at a very abstract level without already introducing unnecessary implementation details. In addition, a good synthesis result using automated tools also requires the user to follow certain modeling guidelines when developing the initial specification. Basically, the specification should capture the required system functionality in a natural way and in a clear and concise manner. Specifically, some of the main guidelines for developing the initial system specification are:

- Separating communication and computation. Algorithmic functionality has to be detached from communication functionality. In addition, inputs and outputs of a computation have to be explicitly specified to show data dependencies.
- Exposing parallelism inherent in the system functionality instead of artificially serializing behaviors in expectancy of a serial implementation. In essence, all parallelism should be made available to the exploration tools in order to increase room for optimizations.
- Using hierarchy to group related functionality and abstract away localized effects at higher levels. For example, local communication and local data dependencies are grouped and hidden by the hierarchical structure.
- Choosing the granularity of the basic parts for exploration such that optimization possibilities and design complexity are balanced when searching the design space. Basically, the leaf behaviors which build the smallest indivisible units for exploration should reflect the division into basic algorithmic blocks.
- Using state transitions to explicitly model the steps of the computation in terms of basic algorithms or abstracted, hierarchical blocks.

The SpecC language provides all the necessary support to efficiently describe the desired system features following these guidelines. Each of the modeling concepts like parallelism or hierarchy is reflected in the SpecC description in an explicit and clear way.

2.2 Vocoder Specification

The GSM 06.60 standard for the EFR vocoder contains a detailed description of the required vocoder functionality [9]. The standard description was translated into a formal, executable SpecC specification

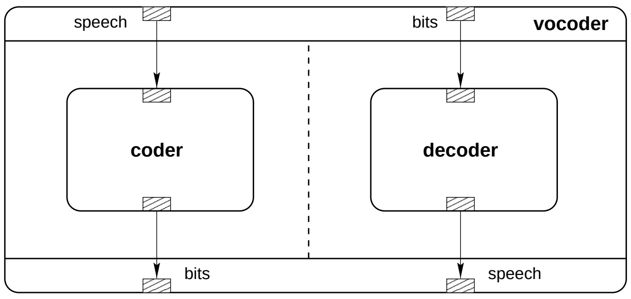


Figure 3: Vocoder.

building the basis for the following synthesis and design steps. In addition, the specification was simulated to verify the vocoder functionality.

The SpecC specification was developed following the guidelines mentioned in the previous section. In our case, part of the vocoder standard is a complete implementation of the vocoder functionality in C (see Appendix A). The C code is based on a 16-bit fixed-point implementation of the algorithms and it serves as a bit-exact reference for all implementations of the vocoder standard, i.e. the C code specifies the vocoder functionality down to the bit level.

Therefore, for the vocoder, the C reference implementation builds the basis for the specification in SpecC. However, a great amount of time had to be spent on analyzing and understanding the standard including the 13,000 lines of C code in order to extract the high-level structure and the global interdependencies. Once this was done, a mapping into a SpecC representation was straightforward. Using the guidelines of Section 2.1.2, the high-level picture of the vocoder's abstracted functionality could be directly and naturally reflected in its SpecC specification.

As will be seen in the following sections this greatly eases understanding of the vocoder basics and therefore supports quick exploration of different design alternatives at the system level in the first place. At each level, the SpecC description hides unnecessary details but explicitly depicts the major aspects, focusing the view of the user and the tools onto the important decisions at each step. For example, at the lowest level, detailed algorithmic code is hidden in the leaf behaviors whereas the relations between the behaviors are made explicit through state transitions.

In terms of the actual algorithmic behavior, SpecC being build on top of ANSI-C made it possible to directly plug the C code of each basic function in the C description into the corresponding leaf behavior of the SpecC specification.

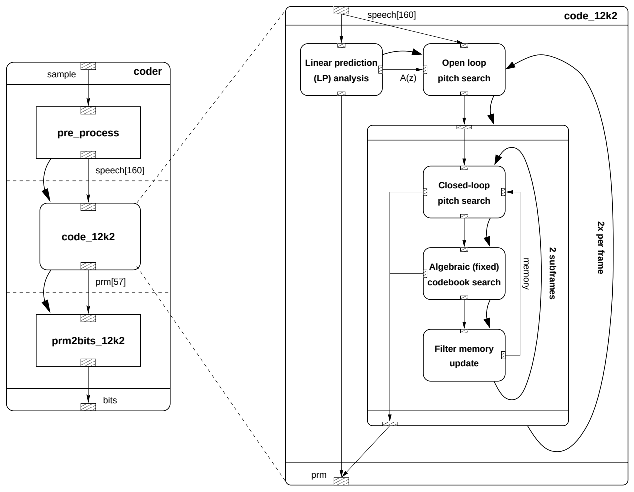


Figure 4: Coder.

2.2.1 Overview

Figure 3 shows the top level of the vocoder specification in SpecC consisting of independent coding and decoding subbehaviors running in parallel. The coder receives an input stream of 13 bit wide speech samples at a sampling rate of 8 kHz, corresponding to an input bit rate of 104 kbit/s. It produces an output bit stream of encoded parameters with a bit rate of 12.2 kbit/s. Decoding, on the other hand, is the reverse process of synthesizing a reconstructed stream of speech samples from an input parameter bit stream. The following sections will describe the encoding and decoding processes in more detail in so far as they are relevant for the following discussions about the vocoder design process. See Appendix B for an in-depth description of the vocoder functionality in SpecC.

2.2.2 Coder Functionality

Coding is based on a segmentation of the incoming speech into frames of 160 samples corresponding to 20 ms of speech. Speech parameters are extracted on a frame-by-frame basis. For each speech frame the coder produces a frame of 244 encoded bits resulting in the aforementioned output bit rate of 12.2 kbit/s.

Figure 4 shows the first two levels of the coder hierarchy. At the top level, the coder consists of three main parts which execute in a pipelined fashion:

1. **Pre-processing:** Buffering of the incoming speech stream into frames of 160 samples. Initial high-pass filtering and downscaling of the speech signal.
2. **Encoding:** The main encoding routine which extracts a set of 57 speech synthesis parameters per speech frame. Encoding will be described in more detail in the following paragraphs.
3. **Serialization:** Conversion and encoding of the parameter set into a block of 244 bits per frame. Transmission of the encoded bit stream.

Due to the pipelined nature, all three parts operate in parallel but each on a different frame, i.e. while a frame is encoded the next frame is pre-processed and buffered, and the previous frame is serialized.

In general, encoding uses an analysis-by-synthesis approach where the parameters are selected in such a way as to minimize the error between the input speech and the speech that will be synthesized on the decoding side. Therefore, the encoder has to simulate the speech synthesis process.

In order to increase reaction time of certain filters, the main encoding routine (also shown in Figure 4) further subdivides each frame into subframes of 40 samples (or 5 ms) each. Depending on their criticality, parameters are computed either once per frame, once every two subframes or once per subframe.

Encoding Encoding itself basically follows the reverse process of speech synthesis. Given the speech samples, in a first step the parameters of the LP filter are extracted. The contribution of the LP filter is then subtracted from the input speech to get the remaining LP filter excitation. The LP filter parameters are encoded in so called *Line Spectral Pairs* (LSPs) which reduce the amount of redundant information. Two sets of LP parameters are extracted per frame, taking into account the current speech frame plus one half of the previous frame. LP analysis produces a block of 5 parameters containing the two LSP sets.

Next, using the past history of excitations, all the possible delay values of the pitch filter are searched for a closest match with the required excitation. The search is divided into an open-loop and a closed-loop search. A simple open-loop calculation of delay estimates is done twice per frame. In each subframe a closed-loop, analysis-by-synthesis search is then performed around the previously obtained estimates to obtain the exact filter delay and gain values.

The long-term filter contribution is subtraced from the excitation. The remaining residual is the input to the following fixed codebook search. For each subframe an extensive search of the codebook for the closest code vector is performed. All possible code vectors

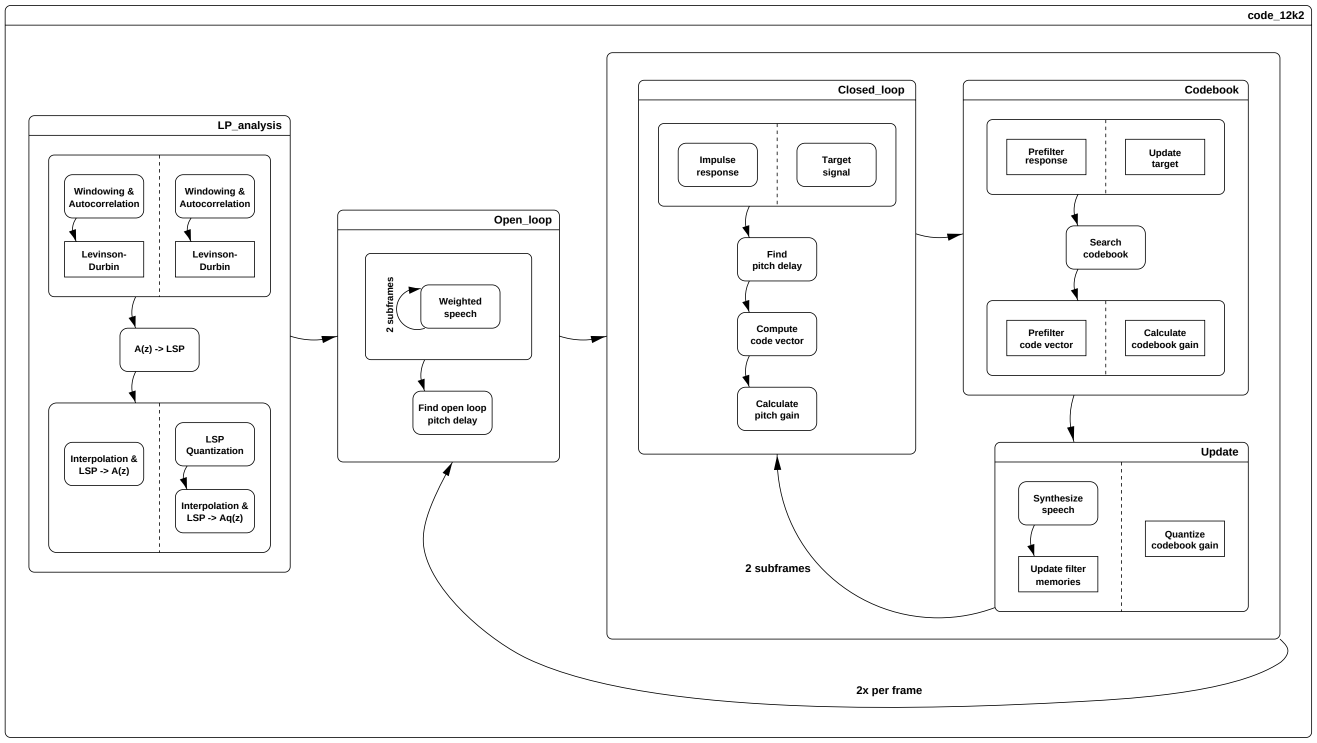


Figure 5: Encoding.

are searched such that the mean square error between code vector and residual is minimized.

For each subframe the coder produces a block of 13 parameters for transmission. Finally, using the calculated parameters the reconstructed speech is synthesized in order to update the memories of the speech synthesis filters, reproducing the conditions that will be in effect at the decoding side.

Figure 5 exposes the next level of hierarchy in the encoding part, showing more details of the encoding process. Note that for simplicity only the behavioral hierarchy and no structural information is shown, i.e. the diagram doesn't include the information about connectivity between behaviors. A complete block diagram of the coder which provides an idea about the complexity by exposing all levels of hierarchy down to the leaf behaviors can be found in Appendix B on page 66 (Figure 64).

As can be seen, at this level, the coder specification exhibits some limited explicit parallelism. However, in general, due to the inherent data dependencies both the coder and decoder parts of the system are mostly sequential in their natures.

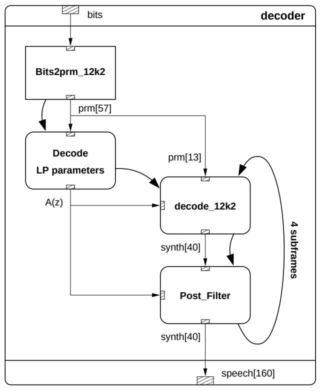


Figure 6: Decoder.

2.2.3 Decoder Functionality

Decoding (Figure 6) basically follows the speech synthesis model in a straightforward way and is more or less the reverse process of encoding. The decoder receives an encoded bit stream at a rate of 12, 2 kbits/s and reproduces a stream of synthesized speech samples at a sampling rate of 8 kHz. For each incoming frame of 244 encoded bits a frame of 160 speech samples is generated.

Incoming bit frames are received and the corresponding set of $5 + 4 * 13 = 57$ speech parameters is reconstructed. The first 5 parameters containing the Line Spectral Pairs are decoded to generate the two sets of LP filter parameters. Then, once for each subframe the following blocks of 13 parameters each are consumed, decoded and the speech subframe of 40 samples is synthesized by adding the long-term pitch filter output to the decoded fixed code vector and filtering the resulting excitation through the short-term LP filter. Finally, the synthesized speech is passed through a post filter in order to increase speech quality.

A more detailed block diagram of the decoder showing all levels of hierarchy down to the leaf behaviors can be found in Appendix B, Figure 69 on page 71. Compared to the encoding process, decoding is much simpler and computationally much cheaper.

2.2.4 Constraints

Transcoder Delay The GSM vocoder standard specifies a constraint for the total transcoder delay when operating coder and decoder in back-to-back mode. According to the standard, back-to-back mode is defined as passing the parameters produced by the encoder directly into the decoder as soon as they are produced. Note that this definition doesn't include encoding and decoding, parallel/serial conversions, or transmission times of the encoded bit stream. Back-to-back mode is not considered as the connection of the coder output with the decoder input. Instead, the 57 parameters produced by the encoder are assumed to be passed directly into the decoder inside the vocoder system.

The transcoder delay is then defined as the delay starting from the time when a complete speech frame of 160 samples is received up to the point when the last speech sample of the reconstructed, synthesized frame leaves the decoder. The GSM EFR vocoder standard specifies a maximum timing constraint of 30 ms for this transcoder delay.

Analysis and Budgeting In addition to the explicitly given transcoder delay constraint the requirements on the input and output data rates pose additional constraints on the vocoder timing. All requirements of the standard were analyzed to derive timing budgets for different parts of the vocoder, resulting in the actual constraints of the SpecC description.

Figure 7 depicts an analysis of the transcoder delay constraint. Note that the time difference between the first and the last sample of synthesized speech

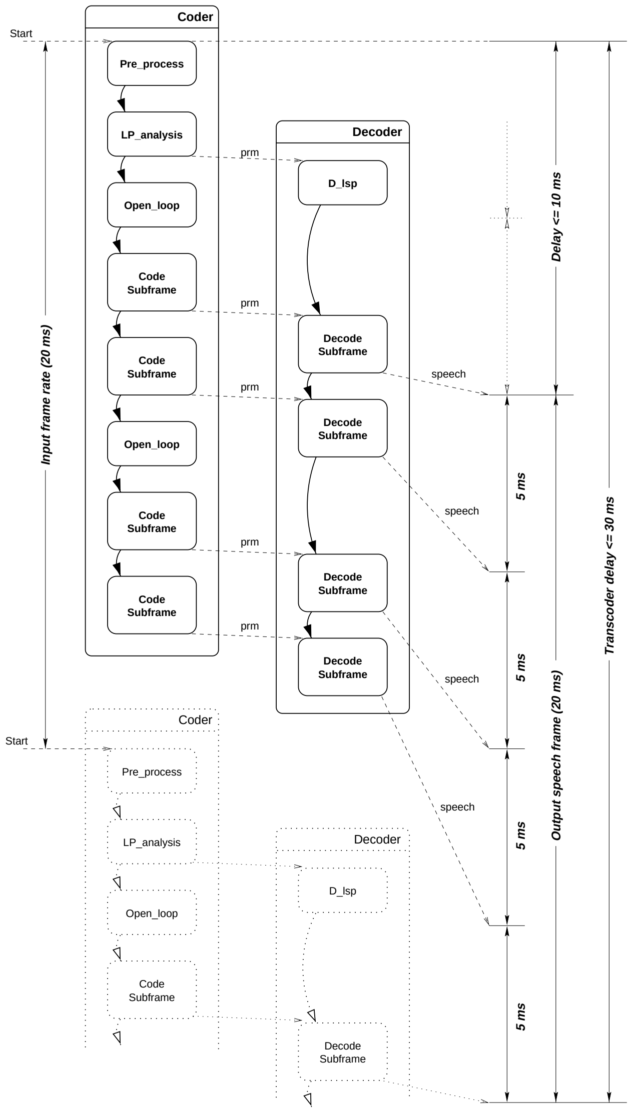


Figure 7: Timing constraints.

at the decoder output is 20 ms (with the given sampling rate). Therefore, if encoding and decoding would happen instantaneously in zero time the theoretically achievable minimum for the transcoder delay is 20 ms, too. In other words, the first sample of reconstructed speech has to leave the decoder not more than 10 ms after the input speech frame is received.

Hence, encoding and decoding of the first subframe of 40 speech samples has to happen in less than 10 ms. This includes all the information needed for the first subframe, i.e. encoding and decoding of the 5 LP filter parameters plus the set of 13 parameters for the first subframe. Then, while the speech samples are written to the decoder output at their sampling rate, the following three subframes have to be encoded into blocks of 13 parameters and decoded into reconstructed speech subframes such that the following subframes are available at intervals of at most 5 ms.

However, while encoding and decoding of the current frame take place the next frame is already received and buffered, and processing of the next frame will have to start once its last sample is received. Therefore, an additional, implicit constraint is that encoding and decoding of a complete frame of 160 samples have to be done in less than the intra-frame period of 20 ms. Hence, decoding of the last subframe will have to be done before that time or—in relation to the transcoder delay constraint—up to 10 ms before the last sample of the synthesized speech frame at the decoder output. Note that this requires a buffering of the decoded speech subframes at the decoder output.

To summarize the constraints for the vocoder, there are two basic timing constraints derived from the given time budgets:

- The encoding and decoding delay for the first subframe ($5 + 3$ parameters) has to be less than 10 ms.
- The time to encode and decode a complete frame (all 57 parameters) has to be less than 20 ms.

3 Architectural Exploration

The goal of architectural exploration is initially to quickly explore a large number of target architectures, comparing them to each other after an initial mapping of the design onto the architecture has been done and finally pruning the design space down to a few candidate architectures. These system architectures are then evaluated further, trying to improve the mapping of the design onto the architecture and possibly modifying certain aspects of the system until a final architecture is selected.

Exploration is a part where the design process can benefit to a great deal from human experience. Therefore, interactivity is an important requirement of system-level design environments. However, with the help of automated tools that quickly search large parts of the design space, provide feedback about design quality, perform tedious, time-consuming jobs automatically, etc. the designer will be able to explore a large number of promising design alternatives in a shorter amount of time. In general, exploration is an iterative process where the different steps described in the next sections are repeated for different architectures. Design automation tools significantly reduce the time needed for each iteration.

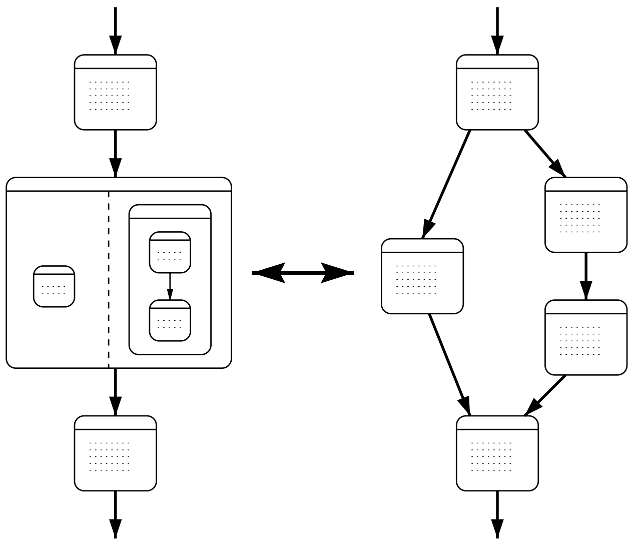


Figure 8: General specification model.

Since the corresponding tools are not yet available at this time exploration for the vocoder project had to be done mostly manually. However, manual exploration strictly followed the flow and the step-by-step procedures proposed for the implementation of the automated tools. Nevertheless, due to the lack of tools we had to restrict ourselves to a very small number of candidate architectures.

3.1 Models

3.1.1 Specification Model

The initial specification written by the designer is the basis for architectural exploration. The specification model at the input to architectural exploration is shown in Figure 8. The specification model is a *superstate finite state machine* (SFSM) with hierarchy and concurrency.

At each level of hierarchy, the superstates (SpecC behaviors) are decomposed further into either parallel or sequential substates. Superstates at the bottom of the hierarchy are called leaf states (or leaf behaviors). They finally contain actual program fragments describing the algorithmic behavior of the leaf states.

In a sequential composition of states at any level of hierarchy the states are traversed in a stepwise fashion. After flattening of the hierarchy, the model resembles a standard state machine with the additional feature of parallelism. In contrast to a classical low-level FSM or FSMD, however, superstates can take an arbitrary

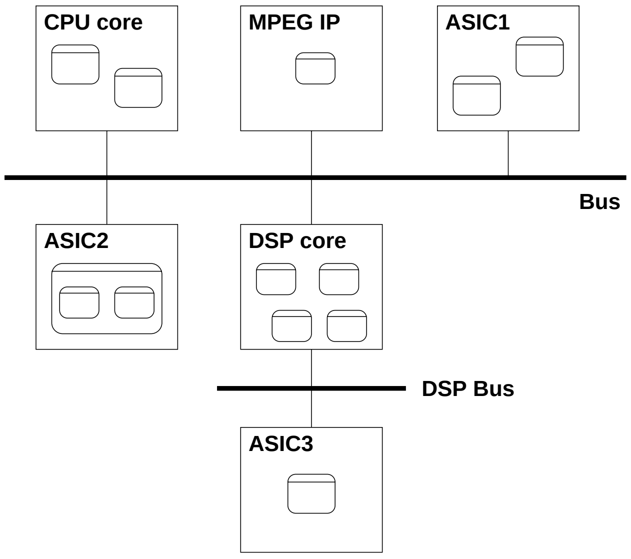


Figure 9: General model for architectural exploration.

amount of time to execute the statements or substates contained within. Much like a dataflow model, a state doesn't start to execute until all of its predecessors are finished. In addition, control flow is introduced by the possibility to augment transitions with conditions.

3.1.2 Architecture Model

Figure 9 shows the general target or output model of architectural exploration. An architecture consists of a set of *processing, memory and communication components*. Processors and memories are connected together by communication links, forming a bipartite graph. During exploration, given the initial specification, behaviors are mapped to processors and communication is mapped to the network in the architecture. The architecture can contain multiple instances of the same component.

Components are taken out of a library or database of available component types. The library contains the functional models for simulation together with information for exploration (e.g. about speed or cost).

In general, the functionality of the library components ranges from fully customizable components which can implement any behavior and any interface to fixed components with predefined behaviors and interfaces. Flexibility in the component functionality is exploited by synthesizing customized versions of the components during the design process. If part or all of the component functionality is fixed the components are also referred to as *intellectual property* (IP) components.

Processing Components In general, processing elements (PEs) can be arbitrary programmable software processor cores or non-programmable hardware PEs. Examples for software processors are digital signal processing (DSP) cores or general purpose (GP) CPU cores. Examples of hardware processors are fully synthesized application specific integrated circuit (ASIC) components or IP components with fixed, predefined functionality. Processors are considered to have a single thread of control. If a processor can execute behaviors in parallel it is split into logical single-threaded processing components for exploration. For example, an architecture with multiple hardware processors will have several custom-hardware FSMDs although all logical HW processors or FSMDs might end up on the same physical chip.

Processing components are fully synthesized or taken out of an IP library. In general, IP processing components are hardware modules with all or parts of their functionality fixed, e.g. an MPEG hardware component or a software processor which can be programmed to implement a large range of behaviors but nevertheless has a fixed, predefined interface, for example. Processing components in the library are characterized by their functionality and parameters like cost, power, speed, etc.

Memory Components In addition to the pure functionality, system-level processing components generally include some sort of local memory. Along with the behaviors, variables and storage in general is mapped onto the local memories of the processors. A special case of system components are (shared) memories. The functionality of memory components is limited to simple reading and writing of their local memories. Global variables in the specification which are shared by two behaviors mapped to two different processors can either be mapped to the local memories of the processor or to a shared memory in case such a component is included in the system architecture.

Communication Components The communication components of the interconnect network handle the communication between the processing and memory components. Therefore, they include any type of communication media used to implement system-level communication. For example, busses are commonly used for interconnection at the system level.

Similar to processing components, communication components are taken out of a library of available interconnect types characterized by their functionality (protocol) and parameters like delay, throughput, etc.

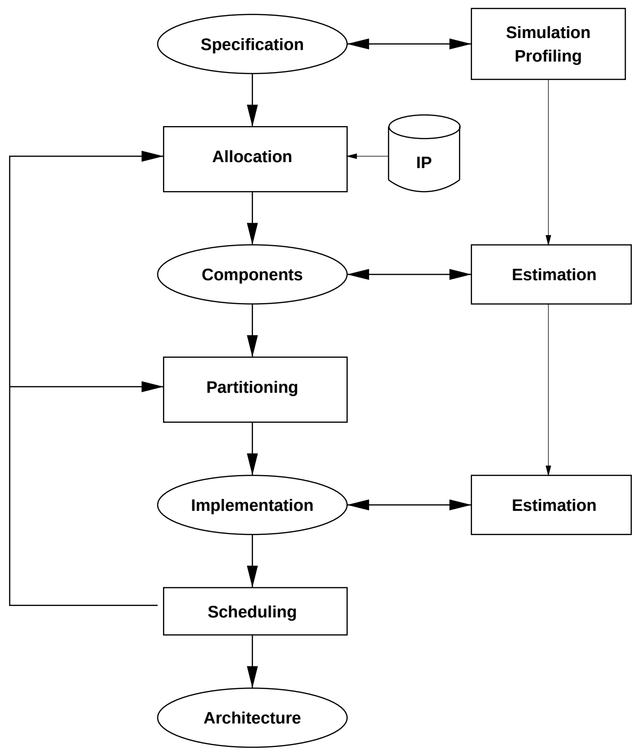


Figure 10: Architectural exploration flow.

Protocols in the library are usually at least partly predefined, implementing industry standards like PCI or VME. On the other hand, protocols are customized or full-custom protocols are synthesized during the design process.

3.2 Exploration Flow

The general flow of steps performed during architectural exploration is shown in Figure 10. With the design specification at the input, exploration creates an architecture for implementation of this design.

Initially, during simulation the specification is profiled to extract estimates about the design complexity and the dynamic behavior. Using these implementation-independent estimates a set of components is selected out of a library during allocation. Allocation is based on matching components with the computational requirements and the available parallelism of the specification.

Once a set of components has been selected, design metrics like cost, delay, etc. of implementing the parts of the specification on these components are estimated taking into account information obtained during initial profiling. In the next step, partitioning is performed to map the design onto the components based on these component-related estimates. During partitioning design trade-offs related to implementation of behaviors on different components and parallel versus sequential execution of behaviors are explored.

After the design has been mapped onto the allocated components the design space is reduced to a single implementation of the design parts on the components they are bound to. A more accurate re-estimation of this implementation is then performed. Finally, the design is scheduled with this information to derive the actual system timing. Constraints are verified and depending on the severity of the violations a new iteration of the exploration loop is started with reallocation or repartitioning until an optimal architecture has been found.

3.3 Analysis and Estimation

3.3.1 General Discussion

The basis for any exploration of the design space—including system-level architectures—is the availability of good and useful design quality metrics. On the one hand, metrics are closely related to the design constraints like performance, power or cost (area, code size, etc.). On the other hand, other metrics can provide additional useful information. Basically, these metrics are the only means of deciding how good a chosen architecture is in comparison with other possible architectures. Therefore, analysis of the specification and estimation of the design metrics for different implementations is an integral part of the design process.

Behavior Estimation During architectural exploration behaviors are mapped to the processing components of the architecture. For each type of target processor the behaviors will exhibit different metrics, i.e. different cost, different performance, etc. Therefore, by combining the properties of the specification with the abstracted properties of the target components estimates about the behavior metrics are obtained without actually implementing the behavior on the component.

However, different implementations of a behavior on the same processor resulting in different metrics have to be considered during architectural exploration, too. For example, the behavior can be optimized for cost resulting in the least-cost solution. At the other end of the spectrum, a behavior can be optimized for performance to achieve the fastest execution possible. Based on the estimated values, the exploration tools will assign budgets for cost or performance, for example, to behaviors or groups of behaviors. This information is then passed to the backend where it is used to guide the synthesis process.

Communication Estimation In general, the overhead of communicating data values between the behaviors of the specification can not be neglected when evaluating possible target architectures. For behaviors mapped to the same component, communication between those behaviors is handled according the component's communication model and is therefore included in the behavior estimation. For example, on software processor the call overhead of pushing/popping values to/from the stack is part of the software implementation of the caller and callee on the processor.

On the other hand, communication among behaviors mapped to different processing components requires transferring data over the channels in the system architecture. Estimates for the overhead of this communication are taken into account during exploration. The main communication overhead is due to the delay of transmitting values from one component to the other.

Basic estimates about the time needed for communication are obtained by evaluating the size of the data block to be transmitted divided by the channel data rate. More elaborate estimates are obtained by considering the protocol overhead of the channel including possibly the segmentation of the data block into smaller packets for transmission.

3.3.2 Initial Simulation and Profiling

In the first step an analysis of the initial specification independent of any implementation is performed by profiling the specification during initial simulations. Simulation of the initial specification is necessary in any case to verify functional correctness. Therefore, these simulations can be augmented to obtain valuable information which will be used for implementation-dependent estimations and exploration in general.

Estimates about the relative computational complexity and the computational requirements of different parts of the specification are obtained by counting basic arithmetic operations (additions, multiplications, etc.), logic operations (and, or, shift, etc.), and memory access operations (moves) during simulation. For each behavior an operation histogram (Figure 11) is created in which the complexity of an execution of the behavior is broken down into the number of occurrences of each basic operation.

Operation profiles are summed according to the specification hierarchy such that total profiles for each

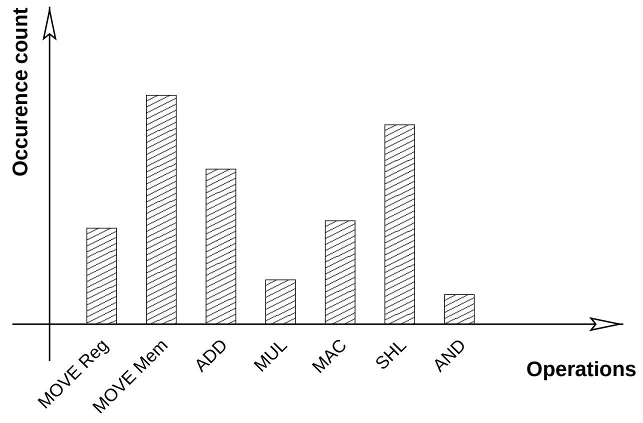


Figure 11: Sample operation profile.

constrained execution path are obtained. The computational requirements for each path can then be calculated by summing the operation counts $count_i$ and dividing the sum by the path's timing constraint T_{path}:

$$\text{MOPS} = \frac{\sum count_i}{T_{path}},$$

giving the complexity in million operations per second (MOPS). Note that these initial estimates are architecture independent and therefore don't have to be recalculated during exploration.

In addition, with dynamic profiling information about the dynamic dependencies both among the behaviors and inside the behaviors can be derived. For example, for data dependent loop bounds information about the worst case execution is obtained by counting the number of loop iterations. In general, a dynamic analysis of the possible paths through the specification and through the behaviors from inputs to outputs along with the frequency at which each path is taken is performed and the results are stored for future estimations.

3.3.3 Estimation

In contrast to the architecture independent initial simulation and profiling the actual estimation during architectural exploration is concerned with deriving the metrics for an implementation of the specification on an underlying architecture. Using the previously collected profiling data a retargetable estimator is used to obtain metrics for a wide range of HW and SW implementations. Depending on the stage in the exploration flow the estimator can work at different levels of accuracy in return for estimation speed.

Coarse Estimation For the initial exploration, a relative comparison of a large number of architectures has to be done in order to select the best candidates. Therefore, at this point absolute accuracy is not of utmost importance. The estimates should rather be obtained very quickly and provide a good relative accuracy, the so called *fidelity*. On the other hand, in order to evaluate different architectures estimation of an implementation on the currently selected target architecture has to be performed.

With the help of retargetable profilers and estimators the behaviors are analyzed statically on the basic block level. For each basic block the number of cycles required for execution of that block on each of the allocated processors are estimated. With the additional dynamic information about the relations of basic block executions and execution frequencies that were obtained during simulation and profiling of the initial specification final estimates about the execution times of the behaviors on the allocated components are computed.

Estimation is very fast this way since both unoptimized synthesis and basic block profiling can be performed quickly. Extensive, time-consuming simulation of the complete specification is not necessary. Due to the unoptimized nature of the implementations, the absolute accuracy of these results is low. However, under the assumption that optimization gains are independent of the actual target these estimates exhibit a high fidelity.

In addition, in each iteration of the exploration loop estimates have to be derived for the newly allocated components only. For previously allocated components the information about behavior execution times from quick or accurate estimates of previous iterations are retained.

Fine Estimation Later, as the design progresses, the solutions have to be evaluated in relation to the design constraints thus requiring estimates with high absolute accuracy. However, the closer the design gets to a final implementation the smaller is the number of architectures under consideration. Major decisions have already been made. For each behavior the component on which it will execute is known after partitioning. Since estimation is fixed to one implementation per behavior more time can be spent on its analysis. Therefore, estimation run times are traded off for absolute accuracy.

Basically, accurate estimation creates an actual, optimized hardware or software implementation of the behaviors. Using the backend process, software is compiled and hardware is synthesized with all optimizations enabled in the same manner as for the final design. Accurate estimated are obtained by combining a static analysis of the final hardware and software execution times with the dynamic information computed during initial simulation and profiling. Since static analysis is done only once for each basic block—the dynamic nature of multiple executions is captured through the initial analysis information—estimation is faster than extensive simulation of the complete implementation.

Only when a final architecture has been selected and pushed through the backend, an overall simulation of the final design will be done in order to verify both, the functionality of the final implementation and the satisfaction of design constraints.

3.3.4 Vocoder Analysis and Estimation

For the vocoder example, the goal was basically to come up with the least-cost solution that satisfies the given timing constraints. Therefore, the specification was analyzed to obtain estimates about the execution times and hence eventually the actual delays of the different vocoder parts.

Initial Analysis Initially, analysis of the vocoder specification was performed with the goal of obtaining estimates about the relative computational complexity of the behaviors in the specification. Computational complexity is directly related to execution times on different platforms. The higher the complexity the longer it will take to compute the result on any platform.

In case of the vocoder example, the C reference implementation of the vocoder standard (see Appendix A) already provided initial estimates through dynamic profiling by counting basic arithmetic, logic and memory access operations. The operations are counted on a frame-per-frame basis, weighted according to their estimated relative complexity and finally combined into the so called WMOPS estimate (weighted million operations per second) by dividing the sum through the time allowed for each frame (20 ms).

Figure 12 and Figure 15 show the WMOPS estimates for the coder and the decoder, respectively. The estimates are broken down into the major parts LP analysis, closed-loop search, open-loop search and codebook search for the coder, and LSP decoding, subframe decoding and post filtering for the decoder. For each part both the total per frame and the major contributing subbehaviors are shown. Each subbehavior

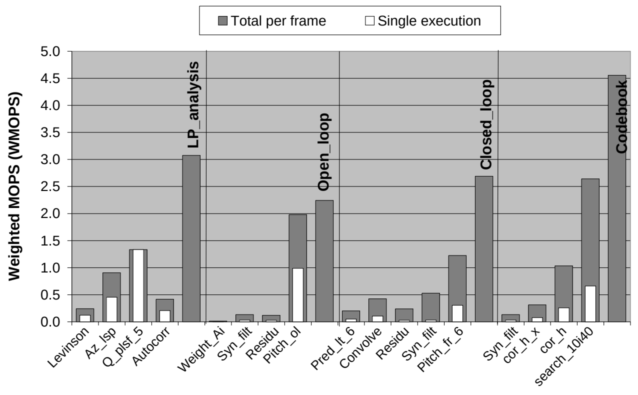


Figure 12: Estimates for computational complexity of coder parts.

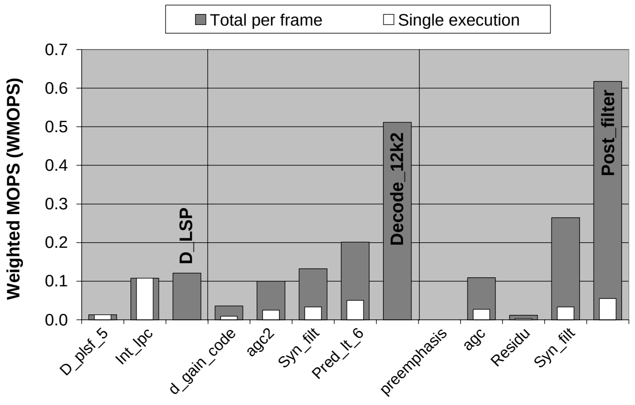


Figure 15: Estimates for computational complexity of decoder parts.

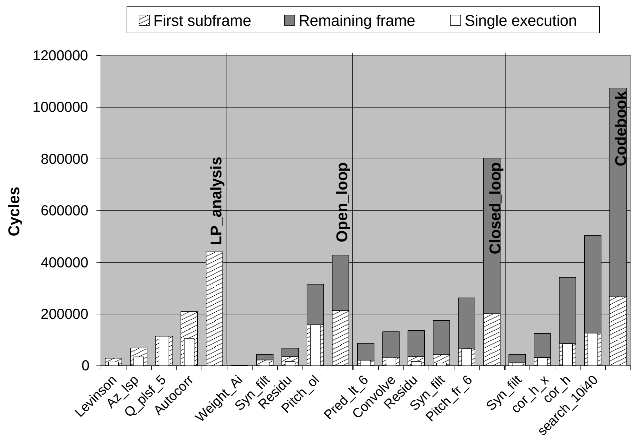


Figure 13: Breakdown of initial coder delays.

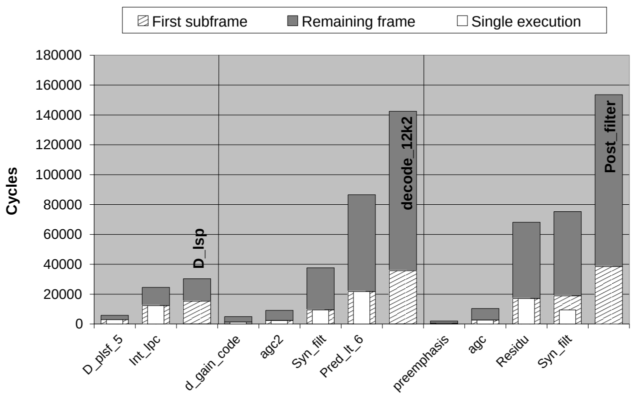


Figure 16: Breakdown of initial decoder delays.

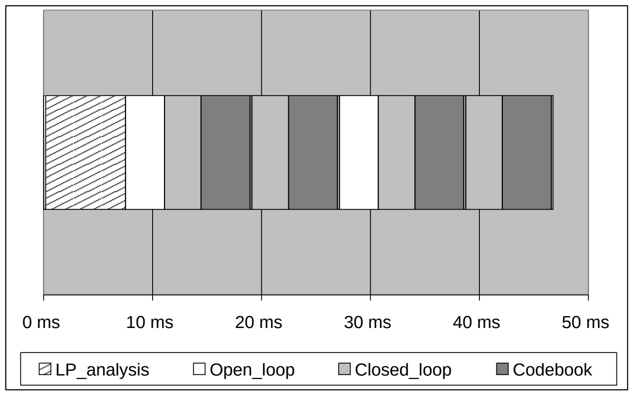


Figure 14: Initial coder delay.

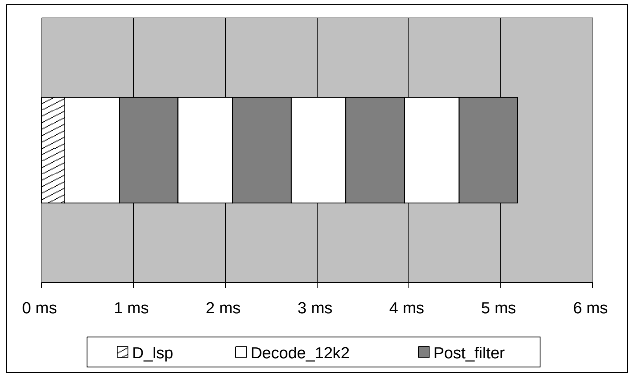


Figure 17: Initial decoder delay.

of the different vocoder parts can be executed several times per frame (e.g. in a loop or once for each subframe). Therefore, for the subbehaviors both the total complexity per frame and the complexity of a single execution, i.e. the total complexity divided by the number of executions are shown.

Accurate Estimation In order to obtain more accurate estimates about the real execution times on the target platform, a quick and straightforward compilation of the behaviors for the target processor under consideration was done next.

Due to the unavailability of the retargetable estimators and profiles, the target analysis of the vocoder was done by compiling the behavior code for the chosen target processor (Motorola DSP56600 family, see Section 3.4) using the official compiler Motorola provides for their processors. The resulting machine code was then simulated on the Motorola Instruction Set Simulator (ISS) to obtain cycle-accurate execution times.

Simulations were performed using typical input data. The results given here represent average execution times per frame or per execution. An analysis of the code reveals that the execution times at this level have only minimal dynamic data dependencies in the order of a few statements difference (at most 10% difference between worst case and average case). Some subbehaviors exhibit static execution time variations through constant parameters passed into the routines depending on the calling environment but these variations are averaged out at the higher levels of the hierarchy.

Figure 13 and Figure 16 show the execution time results (in number of cycles) for the coder and decoder, again broken down into parts and their major subbehaviors. According to the timing budgets derived from the constraints in the initial specification (see Section 2.2.4), the figures show cycles for encoding or decoding of the first subframe plus the rest of the frame totaling in the cycles per complete frame. In addition, the cycles for a single execution of each behavior are included in order to show the complexity of each behavior in comparison with the initial estimates.

Finally, Figure 14 and Figure 17 show the sequences of execution of the parts for one frame as time progresses in the coder and decoder. The given times are given based on a processor clock frequency of 60 MHz. As can be seen easily, for this initial implementation both timing constraints are severely violated. For a complete list of simulation results, see Appendix C, Section C.1.

As mentioned earlier, the results confirm that the coding part is the major contributor to the delays and the computational complexity in general. Therefore, architectural exploration should focus on this part of the system.

Inside the coder, overall, the codebook search is the most critical part since it contributes the most to the violation of the two timing constraints. Also, the largest indivisible leaf behavior in terms of execution time is the actual search routine (`search_10i40`) inside the codebook search.

Furthermore, comparing the actual execution times with the initial WMOPS estimates suggests that an implementation of the codebook search on the processor introduces a higher overhead of operations not directly related to the computation than the software implementation of the other parts. Analysis of the specification shows that most of the time of the other parts is spent inside very tight loops with loop bodies of a few statements only but a large loop count as they are typical in DSP-style applications. These loops promise very good opportunities for optimizations even in software. An exception to this is the codebook search since it is relatively irregular with large and hard to optimize code blocks.

3.4 Architecture Allocation

3.4.1 General Discussion

The first step in each iteration of the search through the architectural design space is the choice of an architecture to be evaluated further. Architecture allocation selects the types of components and the number of components of each type in the system. Along with the selection of processors, architecture allocation also defines the general connectivity between the processors by selecting the types and numbers of communication components between processing components.

Typically, processing elements (PEs) are either programmable processor cores running software or processors implementing a fixed functionality in hardware. Under the aspect of design reuse the architecture can include predefined components taken out of an intellectual property (IP) library, either from in-house sources or provided by a third-party vendor. Usually the functionality of these IP components is fixed and limited to a small range of possible behaviors. An exception are the software processors. Although they are taken out of the library of predefined IP components they are fully programmable and able to implement basically any behavior.

Possible system architectures range from pure soft-

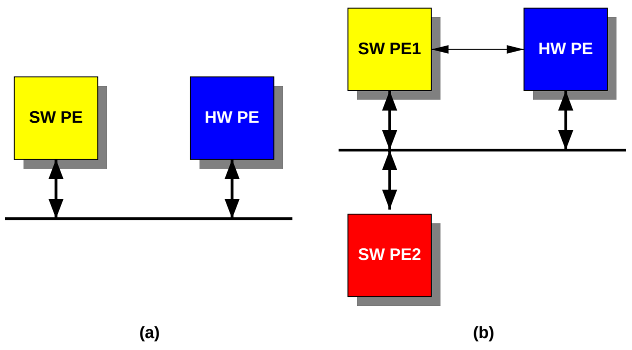


Figure 18: Examples of mixed HW/SW architectures.

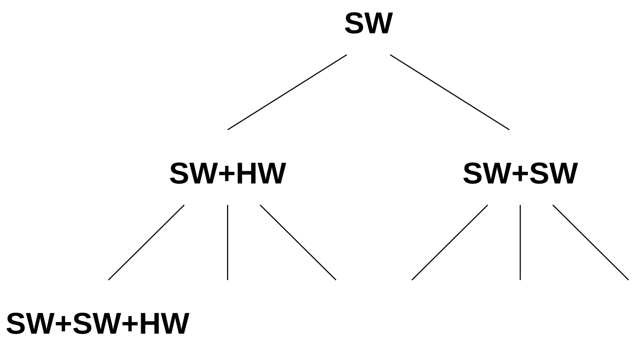


Figure 19: Allocation search tree.

ware solutions using one or more processing elements up to full-custom hardware implementations with one or more ASICs. In between is a vast range of mixed HW/SW architectures based on a combination of SW and HW processors (Figure 18).

3.4.2 Allocation Flow

Allocation successively walks through the design space guided by the information obtained during initial analysis and estimation, selecting components out of the library based on their characteristics. The goal is to select architectures which promise to satisfy timing constraints while staying within the given bounds for other constraints like cost and power.

Figure 19 shows the basic search tree for processor selection. Starting with no components allocated, in each iteration a new component is added to the architecture, possibly replacing another one. With the help of heuristics and pruning techniques like branch-and-bound the breadth-first search through the tree is directed towards promising architectures.

Component Selection Given the requirement to execute a certain number of basic operations in a certain amount of time a set of processing components has to be selected such that their computational power promises to run the operations in the given time. The goal is to satisfy the timing constraints while keeping the cost low.

The principle of component selection is based on choosing a set of processors that achieve the required MOPS rate as defined in Section 3.3. The specific properties and capabilities of the components are matched with the properties of the instruction mix of the specification in order to get more realistic estimates.

Each component in the library has an associated set of operation weights w_i that reflect the number of instruction cycles typically needed to execute each basic operation in the specification. In addition, the components are tagged with their MIPS rate (million instructions per second). Together with the operation counts c_i in the histogram obtained during initial analysis and estimation the rate at which each component will approximately execute the specification is calculated to

$$\mathrm{MOPS}_C = \frac{\sum c_i}{\sum w_i c_i} \mathrm{MIPS}_C .$$

An example of a graph is shown in Figure 20. Components are sorted by their cost. Although in general the computing power increases with increasing cost there are local maxima and minima in the operation rates. Basically, certain components match the operation profile better than others. For example, the vocoder profile includes a large number of multiplications and multiply-accumulates (MACs). A digital signal processor (DSP) with dedicated hardware multipliers and MAC units that operate in one instruction cycle can therefore achieve a higher operation throughput than a general-purpose processor with the same MIPS rate.

Given the local maxima of the operation rate graph, sets of components are then selected such that their added MOPS rates satisfy the MOPS requirement of the specification, at the same time trying to minimize the combined cost of the components.

Allocation Strategy During allocation, trade-offs between resource and timing requirements are explored. Depending on the parallelism available in the specification adding resources (processing components) increases cost but reduces the overall delay. On the other hand, reducing the number of components requires serialization of the specification and increases the delays. Orthogonal to the concept of parallelism,

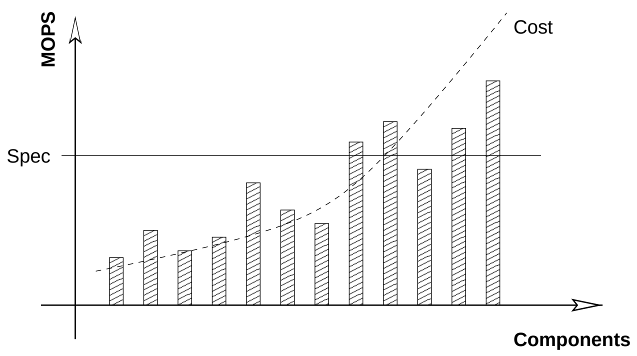


Figure 20: Component matching.

the speed of the components as described in the previous paragraphs, determines how fast the operations of the specification will run.

To combine the parallelism available in the specification with the computational requirements, the specification SFSM is divided into basic steps according to the boundaries between the leaf behaviors. In each step along a path from inputs to outputs the operation profiles of the behaviors obtained during initial estimation are combined. In case a behavior covers multiple steps its operational requirements are distributed evenly among the range of steps.

Figure 21 shows an example of the graph obtained for the given SFSM assuming unit operation requirements per behavior. The graph shows the combination of computational requirements and available parallelism in each step. Note that the duration of each step can and will vary depending on the processors on which the corresponding behaviors will be implemented.

Given the graph, parallelism in the specification is explored by dividing the operations along the graph into chunks assigned to run on different processors in parallel. The amount of operations to be executed on each processing element together with the total timing constraint determines the required MOPS rates for component selection. On the other hand, given a selection of well-matching components a fitting cut of the operation graph can be found.

In general, exploration of parallelism focuses on balancing resource utilization and keeping the allocated parallel components busy. In addition to the exploration of overall parallelism, single peaks in the operation waveform that temporarily exceed the processing rate of the selected components can be cut and flattened by allocating a small and fast coprocessor (ideally an IP component or a small custom ASIC) that will run the operations in a short time. Depending on the available parallelism other processors are either idle or can run other parallel behaviors while the coprocessor is executing.

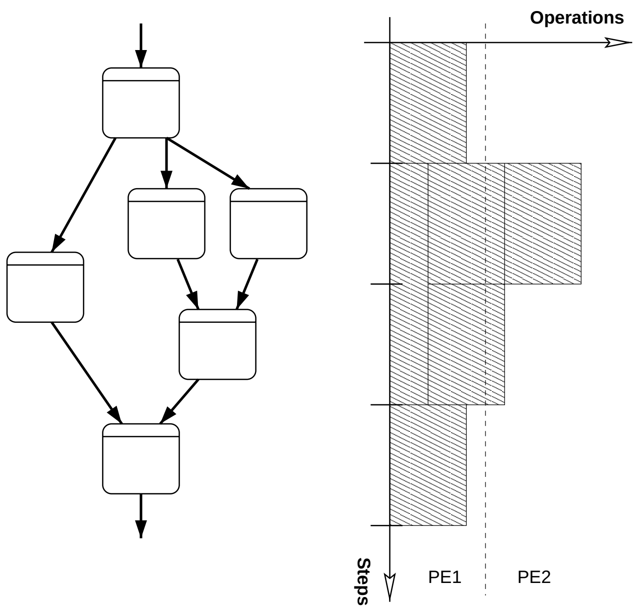


Figure 21: Computational requirements.

3.4.3 Vocoder Architecture

As mentioned previously, for the vocoder project we were restricted to evaluating one architecture. The goal is an implementation which is as cheap as possible while satisfying the constraints.

Initially, a pure software solution based on a single, cheap programmable processor core is assumed to be the cheapest possible solution. Due to the signal-processing nature of the vocoder the instruction profiles of digital signal processors (DSP) conform the best with the application's operation histograms.

Since the vocoder is based on a 16-bit fixed-point implementation out of the class of DSP components a 16-bit architecture matches best. Increasing the bit width of the DSP to 24 bits, for example, just increases cost without reducing the number of instructions required since the additional precision is not needed and therefore not used.

For the vocoder we selected the DSP56600 family [13] out of the DSPs available from Motorola as the one satisfying the above mentioned criteria.

However, the execution time estimation described in Section 3.3 showed that even under the assumption of a poor accuracy of the results the given constraints can't be satisfied by a software solution on the DSP. In the analysis it was also noted that the parallelism inherent in the vocoder is quite limited. Therefore, architectures with multiple parallel processing components will not prove to be beneficial. Since there is not much parallelism to be exploited no speedup can be expected by adding parallel components. In contrast, since the utilization of these components will be low such architectures are cost-ineffective.

Therefore, the option of adding additional processing elements is not considered further. This leaves two options for decreasing the vocoder delays, choosing a faster processor or adding an ASIC component and moving parts of the vocoder functionality into hardware. In the former case, since the DSP architecture is already optimal for the given application faster software execution can only be achieved by increasing the processor clock frequency. However, with increasing processor frequency and speed in general both power consumption and cost increase dramatically.

Under these assumptions, moving to a faster processor is not the most cost-effective next step at this point. Instead, adding a small ASIC component for sequential implementations of vocoder parts in simple hardware promises a significant speedup at a reasonable cost increase.

In conclusion and under consideration of all these points at the end of the process an architecture as shown in Figure 18(a) with one DSP (DSP56600) and one ASIC connected through one communication channel was selected for implementation of the vocoder.

3.5 Partitioning

3.5.1 General Discussion

After an architecture has been selected for evaluation the next step is to map the specification onto the architecture. This includes partitioning the behaviors onto the system processors and partitioning the behavior communication according to the system connectivity. Note that partitioning is closely related to scheduling (Section 3.6) in the sense that partitioning has to be followed immediately by scheduling in order to get the total delay and therefore feedback about the quality of the partition.

Behavior Partitioning Behavior partitioning distributes the functionality of the system onto the available system processing components. The goal is to reach a near optimal partition under consideration of design constraints like cost, timing, power, etc., e.g. by looking for a solution that satisfies the timing constraints with minimal cost in the case of the vocoder.

During partitioning several options for trading off different design aspects exist. Mapping behaviors to different types of components results in different execution times where a speedup is usually associated with an increase in cost. For example, implementing a behavior in hardware instead of software will decrease the delay. On the other hand, multiple processing components of the same or different type allow the exploration of parallelism among behaviors and among components during partitioning and scheduling.

Partitioning will also insert additional synchronization in places where it is necessary to coordinate the execution of the components operating generally in parallel. For example, in case of data dependencies successive behaviors mapped to different components have to be synchronized in order to ensure correctness and equivalence with the semantics of the original specification.

Figure 22 shows an example of a HW/SW partitioning for the encoding part of the vocoder based on the selected target architecture with one DSP and one ASIC. Although the available parallelism in the encoder is limited, the partition tries to exploit the parallel execution on the DSP and the ASIC. Mostly, however, the partition is based on moving computationally expensive parts into hardware, thereby reducing execution times and satisfying constraints even when DSP and ASIC operate in a serialized fashion.

Note that the example shown is only one partition out of thousands of possible mappings. Again, this justifies the need for automated partitioning and estimation tools in order to be able to quickly explore a large number of different architectures and partitions.

Channel and Variable Partitioning Similar to the mapping of behaviors onto processing components, after behavior partitioning the communication and synchronization of behaviors across component boundaries has to be mapped onto the available system interconnectivity.

The abstract communication between behaviors in two different processors is grouped into abstract system channels corresponding to the mapping of each communication onto the available connectivity between the processing components in the system architecture. In reference to behavior partitioning, channel partitioning selects the type of system connection to be used to implement a given abstract communication. After channel partitioning and successive scheduling of communication the grouping into abstract channels reflects the mapping of communication onto system connections.

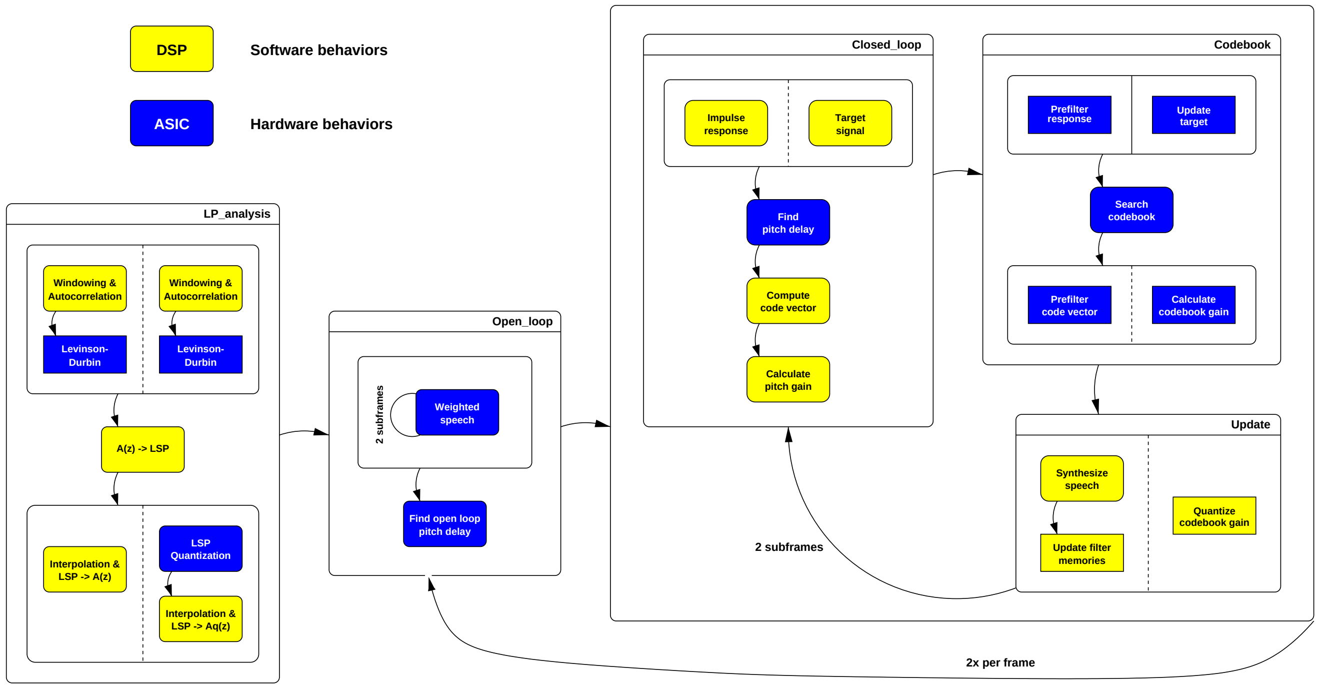


Figure 22: Example of an encoder partitioning.

In case of communication through variables or buffers in general, the task of variable partitioning maps these component-global variables into local memories or into a shared memory if a global memory has been included into the system architecture.

3.5.2 Partitioning Flow

In each iteration of the exploration loop the partitioning step refines the current partition by moving selected behaviors between components. After each change, the partition is rescheduled. This process is repeated until a partition is found that promises to satisfy the timing constraints. If no partition can be found a reallocation will become necessary.

Similar to the situation during allocation partitioning can either focus on the speed of the components trying to satisfy the timing constraints or it can try to balance and maximize resource utilization. The selection of a strategy or a combination of both strategies is done in each iteration of the exploration loop depending on the constraints the the component capabilities. Strategies are switched between iterations to explore different dimensions of the design space and to optimize for different constraints.

Timing-Driven Partitioning The basis for timing-driven partitioning is a strategy with the goal of satisfying the timing constraints while keeping implementation cost low. Behaviors on the critical paths are speeded up by moving them to faster processing components. Under the assumption that the implementation costs of all behaviors are approximately the same, a least-cost solution is achieved by making moves that exhibit the highest gains in total path delay.

The general principle for timing-driven partitioning is based on a measure of so called *criticality*. Given a current partition, the criticality tries to measure the contribution of a behavior mapped to a certain component to the violated overall timing constraints. For each critical path the delay a behavior on that path contributes to the overall path delay is given by the behavior execution time d and the number of times n the behavior is executed. The criticality of a behavior is then the sum of its relative contributions on all critical paths:

$$\text{Criticality} = \sum_{\text{paths } p} \frac{n_p \times d_p}{T_p}$$

with T_p being the total delay of path p, i.e. the sum of behavior execution times with behaviors being mapped to different components along the path. Note that by summing over all paths behaviors that contribute to multiple critical paths are favorized. As

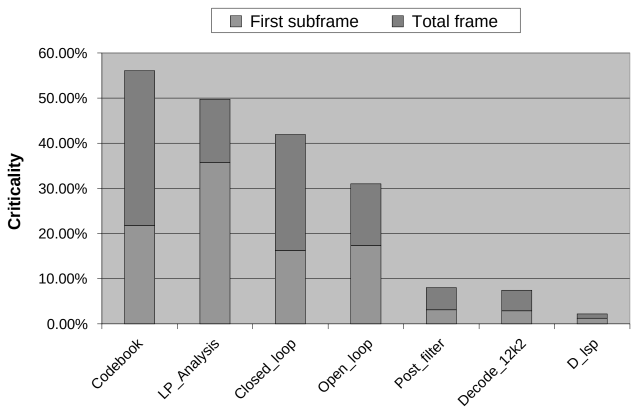


Figure 23: Criticality of vocoder behaviors.

an example, Figure 23 shows the criticality for the top-level behaviors of the vocoder.

Under consideration of increases in cost due to behavior moves, timing-driven partitioning then tries to speed up the most critical behaviors by moving them to faster components. Behaviors are selected depending on the gain in criticality that is achieved by a move. Note that moving the most critical behavior doesn't necessarily result in the highest gain when other behaviors experience higher speedups during a move. In addition, information about available parallelism among critical behaviors can be included in deciding which behavior to select. Additional speedups can be obtained by moving critical behaviors such that they can execute in parallel with other critical behaviors.

Resource-Driven Partitioning In contrast to timing-driven partitioning, resource-driven partitioning tries to speed up the design as much as possible by balancing resource usage and thereby utilizing the resource parallelism maximally. The peak computational power of the system is reached when all the components are fully utilized.

The basic idea is to assign parallel behaviors to the components in such a way that the *slack* is minimized. The slack (see Figure 24) of a parallel decomposition of behaviors is the total amount of unutilized computational power resulting from components being idle during the execution of the graph. If the slack is zero all components are fully utilized and peak performance is reached.

Mapping the behaviors to different components results in different execution times for the same behavior. Theoretically, the shortest delay that can be achieved is given by the critical path after choosing the fastest implementation of every behavior. However, parallel behaviors mapped to the same component have to be serialized and dependencies among the behaviors have to respected. In general, both iterative and constructive approaches for slack minimization and optimization of resource utilization are possible.

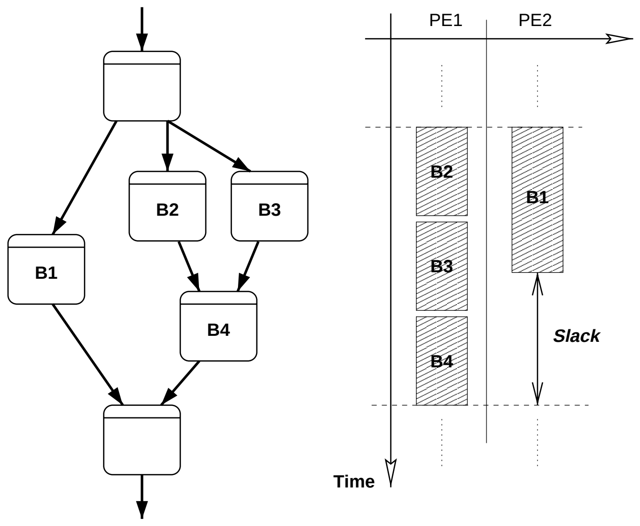


Figure 24: Balancing resource utilization.

3.5.3 Partitioning for Vocoder

In case of the vocoder, the target system architecture consists of one DSP and one ASIC connected by one system channel.

Behavior Partitioning The objective is to move parts of the vocoder functionality into the ASIC, achieving a speedup through implementation in hardware in order to satisfy the timing constraints. To keep the cost low the ASIC should be kept as small as possible while the software has to fit into the selected processor (i.e. program and data memory).

Starting from a pure software solution where all behaviors are mapped to the DSP, the basic strategy followed for partitioning of the vocoder example onto the selected architecture was to successively move the most critical behaviors in terms of violation of the timing constraints from the DSP into the ASIC until the constraints are satisfied. At the same time, on the other end, starting with the least critical behaviors the software implementation was improved by optimizing the code. Assuming that the cost of implementing each behavior in hardware is is in the same order of magnitude, moving behaviors first where the largest

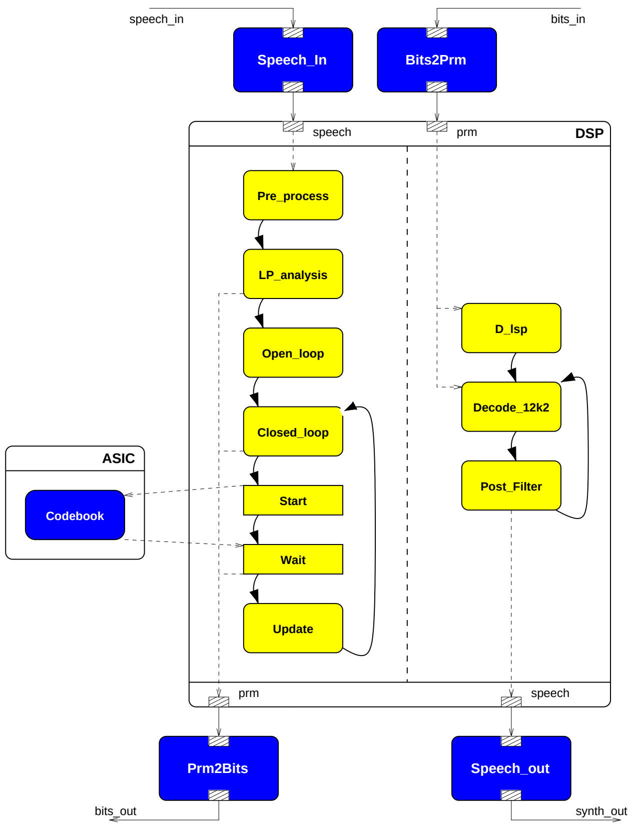


Figure 25: Final vocoder partitioning.

gain in execution time is expected will eventually result in the least-cost solution.

As the analysis in Section 3.3 has shown, the codebook search is the most critical part under consideration of both timing constraints. In addition, it has been noted that the codebook search comes with a high software overhead and is not as amenable to optimizations compared to the other parts with their tight DSP-style loops.

Therefore, it was decided initially to move the codebook search into hardware. To reduce the expected communication overhead between the DSP and the ASIC not only the actual search routine (`search_10i40`) but the complete codebook search part (`Codebook`) including the prefiltering, gain calculation, etc. was mapped onto the ASIC.

As it turned out, the implementation of the codebook search in hardware (described in Section 5.2) together with the optimization of the software code (see Section 5.1) was already sufficient to satisfy the given constraints. Hence, no more behaviors had to be moved into the ASIC.

Figure 25 shows the SpecC model after final partitioning of the vocoder including coder, decoder and all the external interfacing. The interfaces are responsible for parallel/serial conversions, initial buffering, synchronization and protocol handling at the external ports, etc. Since they include all the functionality that has to be performed in parallel to the actual encoding and decoding they are implemented as separate, independent hardware modules running in parallel.

Therefore, the final partition consists of six parallel behaviors: the central processor running the main coder and decoder behaviors, a dependent coprocessor implementing the codebook search, and interfaces for incoming and outgoing speech sample and encoded bit streams.

Parallel, independent coder and decoder task are assigned to the processor. Executions of the tasks are triggered by events from the external input interfaces. The tasks then receive the frames of input data from their corresponding input interfaces. During one execution they produce a frame of output data which is send to the corresponding output interfaces, respectively.

The speech input interface receives the constant stream of speech samples, buffering one frame of samples each. Once the frame buffer is full, the coder task in the processor is signaled and the PE copies the frame buffer to its local memory. Note that after the last sample in the buffer has been received the copying of the frame has to be started before the next speech sample arrives. As will be seen later, satisfaction of this constraint will be ensured after communication synthesis.

During execution, the coder task triggers the hardware codebook search by sending the search data to the ASIC. The coder then waits until the result is returned by the codebook search routine. Basically, the ASIC is a dependent coprocessor whose executions are triggered by the master processor.

Encoded speech parameters are sent directly to the coder's output interface as soon as they are produced during the coder run. The output interface then converts the parameters into encoded bit blocks and transmits them according to the external protocol. Once the coder has finished one execution it returns to its initial state, waiting for the next input frame (for simplicity, the outer loops are not shown in the figure).

On the other hand, the decoder's input interface performs the reverse process of receiving the encoded bit packets from the external world, decoding them into parameters sets. Each time a complete subblock of LP or subframe parameters is received it is sent to

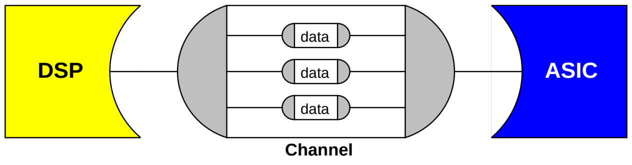


Figure 26: Channel partitioning.

the decoder task in the processor.

The decoder task in the DSP runs in an endless loop. For each frame it waits for the LP parameters, decodes them and repeatedly waits for the four blocks of subframe parameters, decoding them to produce a subframe of speech as they arrive. The synthesized speech subframes are sent to the speech output interface as they arrive. The speech interface then buffers the output speech samples and generates the speech stream at the external interface.

As can be seen, communication between the six behaviors combines data transfers with synchronization. At this step, all the communication of the six behaviors at the system level is implemented through abstract channels. The channels are automatically inserted during partitioning. In case of the vocoder, channels based on the abstract semantics of synchronous, blocking message passing are used. Note, that the channels just define the abstract communication model without any decision on their actual implementation.

Channel Partitioning In case of the vocoder, only one system channel exists which connects DSP and ASIC. Therefore, all communication between behaviors mapped to the DSP and behaviors mapped to the ASIC is grouped into one channel representing the system connectivity at an abstract level (Figure 26).

3.6 Scheduling

3.6.1 General

Partitioning is immediately followed by the task of scheduling. Both are closely related since the quality of a partition is not finally revealed until scheduling has been performed.

Wherever necessary, scheduling determines the order of execution of the behaviors in relation to each other under consideration of the given constraints. For example, parallel behaviors in the specification which are supposed to be implemented on the same sequential, single-threaded hardware or software component have to be ordered and serialized. The serialization of

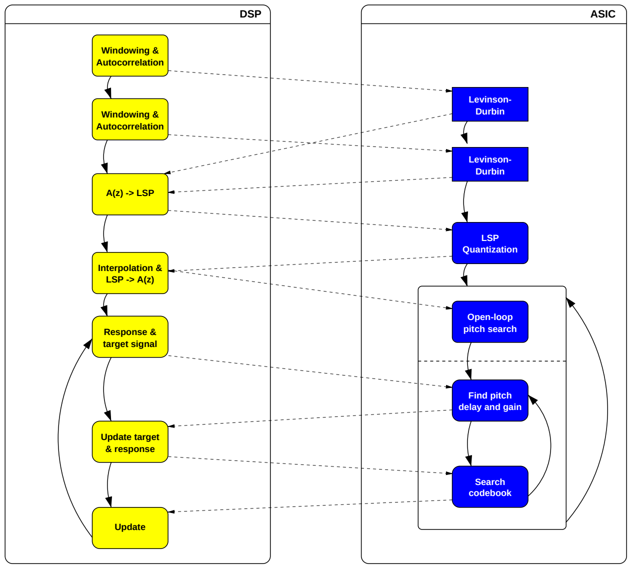


Figure 27: Sample encoder partition after scheduling.

one component is done on the system level in combination with scheduling of behaviors in other processors, maximizing resource utilization and minimizing delays due to waiting for input data.

Finally, scheduling not only determines an ordering for behavior executions but it also selects the final communication implementation. Similar to the aspects for behaviors and processors, the final mapping of communication channels to actual instances of connections in the system is determined. If necessary, this requires a serialization of communication mapped to the same communication component, for example. On the other hand, due to the serialization of behaviors certain synchronizations might have become redundant and are therefore removed during scheduling.

Scheduling may be done statically or dynamically. In *static scheduling*, each behavior is executed according to a fixed schedule. The scheduler computes the best schedule at design time and the schedule does not change at run time. On the other hand, in *dynamic scheduling*, the execution sequence of the subtasks is determined at run-time. During scheduling priorities are assigned to the different tasks running in parallel. An application-specific run-time scheduler is automatically generated. On the software side the scheduler becomes part of the embedded operation system whereas on the hardware side the control arbiter FSM will be synthesized as part of the ASIC. The runtime scheduler or arbiter maintains a pool of behaviors ready to be executed. A behavior becomes ready for

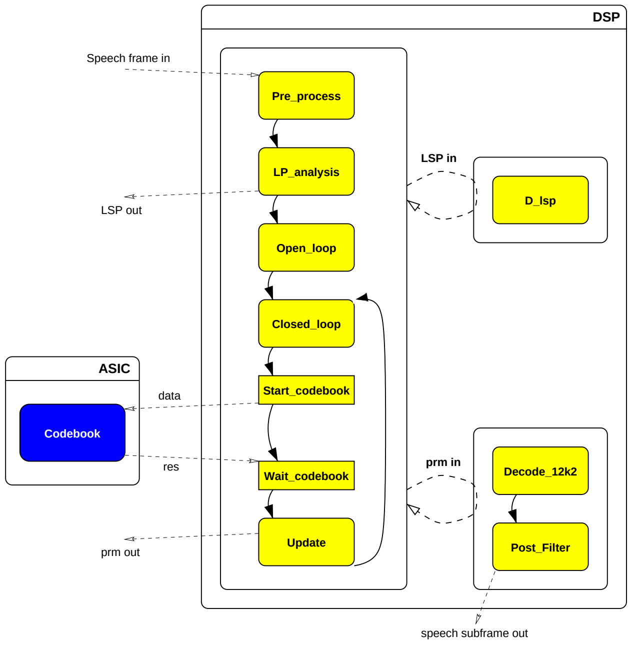


Figure 28: Final dynamic scheduling of vocoder tasks.

execution when all of its predecessor behaviors have been completed and all inputs are available. With a *non-preemptive* scheduler, a behavior is selected from the ready list as soon as the current behavior finishes, whereas for a scheduler with *preemption*, a running behavior may be interrupted in its computation when another behavior with higher priority becomes ready to execute.

Figure 27 shows an example of a schedule for the sample partition from Figure 22. Software behaviors are serialized and execution of software and hardware behaviors is overlapped where possible in order to exploit the available parallelism between ASIC and DSP.

3.6.2 Vocoder Scheduling

For the vocoder example, the behaviors mapped onto the processor have to be scheduled. At the top level, this requires serialization of the parallely executing coder and decoder tasks. The coder and decoder tasks themselves are already inherently sequential.

Due to the dynamic nature of the relation between coder and decoder—in general, the timing relationship of coder and decoder execution depends on external triggers and will only be determined at run-time—a dynamic scheduling approach is needed. A fixed schedule would possibly incur unwanted additional delays through required buffering if the external input doesn't directly conform with the fixed schedule.

The SpecC model of the dynamic scheduling approach chosen for the vocoder is depicted in Figure 28. The coder task builds the main program which executes in synchronization with the external input, i.e. a new iteration of the main loop is started as soon as a new speech input frame arrives. The coder communicates with the coprocessor for offloaded executions of the codebook search.

Apart from that, the coder task is interrupted asynchronously whenever a new piece of decoder data arrives. Depending on the type of incoming parameters either LP or subframe decoding is executed and the control flow returns to the coder at the point where it was interrupted.

This is a simple implementation of dynamic scheduling as a degenerated version of the general case. A general dynamic scheduler would require additional scheduling code which would run as the main program. The scheduler would execute at regular intervals, either interrupt-driven or by splitting the tasks into separately executed chunks. At each execution the scheduler would select which of the two task to execute in the next interval. However, due to the limited number of tasks and the simple relationship between the tasks such a general scheduler is not needed for the vocoder and this simple scheduling scheme with a low overhead is sufficient.

In terms of communication scheduling, since the processor is the master of all external communications—data transfers are synchronous and are initiated by the processor, possibly in reaction to external events—and since the processor itself is sequential, there is no need to serialize external communication on the system channels. It is ensured that at no point two data transfers can happen at the same time.

3.7 Results

The results for the vocoder after architectural exploration are summarized in Table 1. Codebook search is implemented in hardware and the software behaviors have been optimized (see Section 5).

Note that at this point, the given delays do not

	First subframe		Total frame	
	Cycles	ms	Cycles	ms
Coder	287943	4.80	511130	8.52
Decoder	29648	0.49	89596	1.49
Combined	317591	5.29	600726	10.01

Table 1: Delays after architectural exploration.

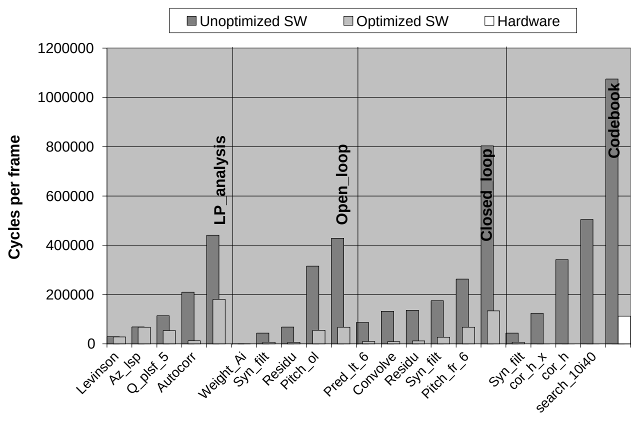


Figure 29: Breakdown of coder delays after exploration.

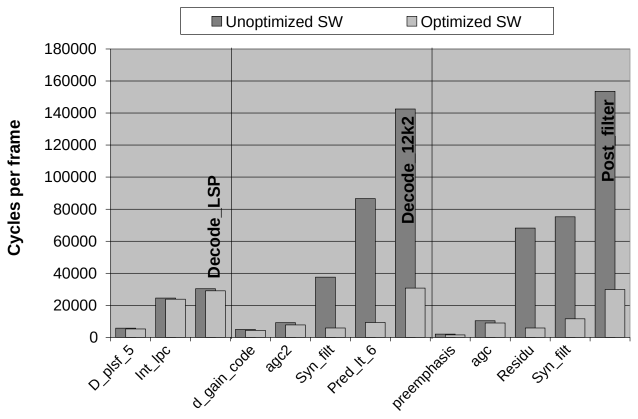


Figure 30: Breakdown of decoder delays after exploration.

yet include the overhead for communication between DSP and the hardware blocks. However, as the results show both timing constraints are satisfied with enough margin to include even very conservative estimates for the communication delays.

The results even suggest that it will be possible to slow down processor and/or ASIC, e.g. by reducing the clock frequency, resulting in a significantly reduced final power consumption.

Finally, Figure 29 and Figure 30 show the delays broken down into the major parts and major subbehaviors per part for the coder and decoder, respectively. Both initial, unoptimized delays as well as delays for hardware and optimized software are shown in comparison. The data confirms that a significant speedup can be obtained by optimizing the software. However, even a moderately simple hardware implementation results in considerably larger gains compared to optimized software.

4 Communication Synthesis

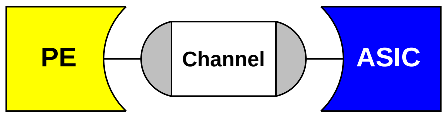


Figure 31: Architecture model.

In the SpecC architecture model obtained as a result of architectural exploration the communication between system components is still modeled on a high level through abstract channels (Figure 31). Although the channels represent the grouping according to the mapping onto underlying communication media like busses etc. they don't yet contain any information about the actual implementation of `send()` and `receive()` primitive's semantics.

Communication synthesis, therefore, has the purpose of gradually refining the channels in the system model down to an actual implementation with data transfers over wires. This comprises the steps of protocol selection, transducer synthesis and protocol inlining.

4.1 Protocol Selection

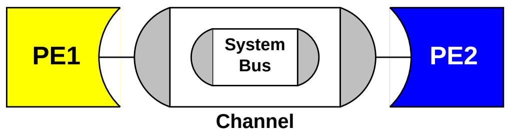


Figure 32: General model after protocol selection.

During protocol selection, for each abstract channel on the system level an actual communication protocol is selected out of the library of available protocols. The protocols in the library are described as SpecC channels and include, for example, standardized, proprietary or custom bus protocols like PCI, VME, etc.

The selected protocol is then hierarchically inserted into its system channel and the abstract communication of the system channel is transformed into an implementation based on the primitives provided by the protocol. For example, this includes assembling and disassembling system messages into protocol packets.

As an example, Figure 32 shows the SpecC model after a system bus protocol has been selected for implementation of the communication between the processing elements.

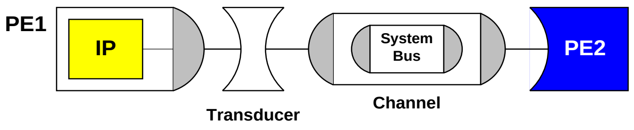


Figure 33: Sample model after transducer synthesis.

4.2 Transducer Synthesis

Some system components including in particular non-synthesizable and IP components come with a fixed protocol on their external interfaces. This also includes processing elements in general since the external processor bus is usually fixed to a protocol defined by the provider. If the protocol of such components is not compatible to the protocol selected for the channels connected to the component interfacing hardware has to be inserted which translates between the two protocols.

In these cases, the task of transducer synthesis therefore inserts an additional behavior, a so called *transducer* between the component and the channel in the SpecC model. In the system model, the behavior of the IP component is replaced with a true functional model enclosed into a *wrapper* (Figure 33). The wrapper encapsulates the proprietary component protocol and provides the abstract canonical interface for communication with the IP. The transducer performs the necessary protocol translations between the wrapper and the channel communication primitives. Note that transducers are not required for interfacing to synthesizable components since they can be designed to implement any selected system protocol.

In case of processor IP, application-specific I/O routines are synthesized and added to the embedded operating system on the software side when replacing the behavior with the processor model. The calls of the abstract channel routines in the software behaviors are replaced with system calls to these I/O routines. The routines in turn handle the interfacing to the external world which includes external data transfers, memory-mapped I/O, interrupt handling, etc.

In general, wrappers together with the clear separation of computation (behaviors) and communication (channels) are the key for IP plug'n'play. At any time during the design flow it is possible to replace a combination of a general, synthesizable component and a channel with an IP component plus wrapper in the SpecC model. As will be shown later on the vocoder example (Section 4.4, Figure 35), in such a case two general components connected through a general system channel are replaced with a general component being directly connected to an IP component using the

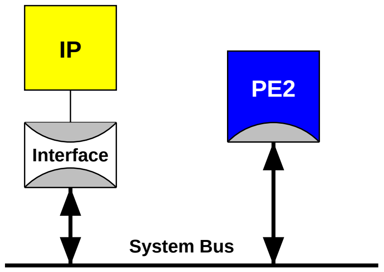


Figure 34: General communication model after inlining.

proprietary IP protocol for communication. However, since the wrapper abstracts the IP protocol onto the same canonical interface as used by the other system channels the replacement is possible without any further modifications. Only later, after protocol inlining, will the IP protocol be exposed.

Again, the key is the separation of communication and computation and the abstraction through channels, features provided by the SpecC language. A mix of behavioral functionality with communication functionality would require to separate those two before the behavior could be replaced with functionality provided by an IP component. A process that is tedious and almost impossible to do automatically.

4.3 Protocol Inlining

Protocol inlining is the final step in communication synthesis. It is the process of inlining the channel functionality into the connected behaviors, exposing the actual ports, wires, etc. of the system connectivity. After inlining the final system model consists of components connected through wires and ports (Figure 34).

Inlining moves the communication functionality into the components, adding it to the already existing behavioral functionality. Naturally, this can only be done for flexible components where the channel functionality will be synthesized into SW or HW together with the other behavior. In all other cases, a transducer has been inserted before and the channel functionality will be inlined into the transducer resulting in the final interfacing hardware.

The final communication model obtained in this step includes all information about communication and the corresponding overhead and delays. It is simulated to verify both functional and timing correctness of the design including communication details before the model is finally handed off to the backend (Section 5).

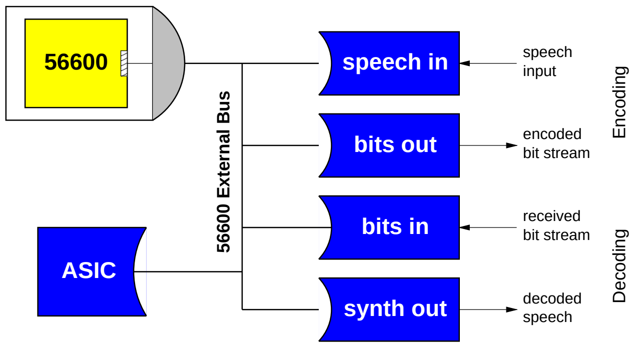


Figure 35: Vocoder model with processor bus protocol selected.

It should be noted that after inlining has been performed it is no longer possible to exchange and replace components since communication and computation are interleaved and not distinguishable any more.

4.4 Vocoder Communication Synthesis

4.4.1 Protocol Selection

In the vocoder example, the system channel simply connects the DSP with the ASIC and the interfacing hardware. Due to the limited number of components and since the hardware modules can be synthesized to implement any communication protocol there is no need to select a standard protocol out of the library for the system channel.

Instead, the proprietary processor bus protocol was selected for system communication. On the processor side, this eliminates the need for a transducer. The hardware modules, on the other hand, will be synthesized to interface with the given processor bus.

Figure 35 shows the vocoder model after protocol selection and after the DSP behavior and the system channel have been replaced with a model of the real DSP56600 processor consisting of functional component plus wrapper. As mentioned previously, at any time during the design process behavior/channel combinations can be replaced with IP components (plus wrappers) without the need for any modifications inside the behaviors or channels. Therefore, as the vocoder example shows, integration of IP is easily possible in the SpecC models.

In the vocoder example after protocol selection, the processor is the central component and all data transfers on the processor bus from and to the dependent hardware modules are initiated by the software on the processor. Apart from that, the hardware components can send asynchronous events to the processor by triggering interrupts. Based on this protocol, the abstract synchronous message-passing communication of the vocoder is implemented as follows.

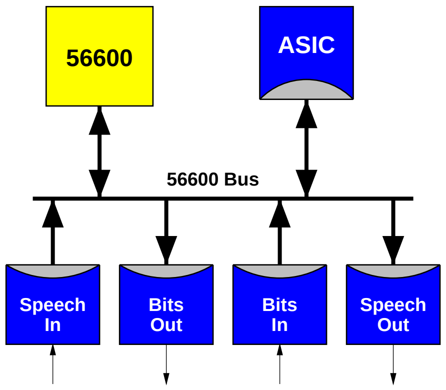


Figure 36: Vocoder communication model after inlining.

On the software side, abstract communication primitives are replaced with calls to I/O routines and interrupt handlers. Address ranges on the external bus are assigned to the different communication links. On the DSP56600, external data transfers are performed by accessing program memory locations above `$8000` (hexadecimal). Therefore, external communication is replaced by writing to or reading from the selected program memory locations.

Externally, through calls to the processor wrapper's communication routines, the hardware modules decode their assigned address ranges and map matching bus accesses to reads and writes of their local memories or registers. Events sent to the hardware modules are signaled by writes to selected memory locations, e.g. start of processing is triggered by receiving the last item of an input data block transfer.

On the other hand, hardware components send events to the software by raising interrupts through their wrapper calls. For example, the hardware signals the availability of new data (e.g. new incoming speech or parameter frames) or computation results (e.g. codebook search results) to the processor. On the software side, the interrupt handlers receive these events and transfer the data one word at a time by handshaking with the ASIC over the bus. The handlers repeatedly execute instructions that initiate read cycles on the external bus, putting the data words read from the bus into the local processor memory. After the complete data block has been read the handlers either start the corresponding behavior execution immediately (e.g. decoding in the vocoder) or they set a flag which can be tested by the program.

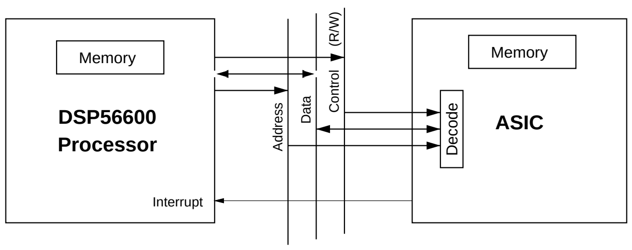


Figure 37: Vocoder hardware/software interfacing model.

4.4.2 Protocol Inlining

Finally, inlining of the wrapper functionality into the hardware components is performed (Figure 36). There the address decoding, interrupt generation, and bus protocol handling functionality is combined with the hardware behavior. Both parts will then be synthesized together to generate the final hardware components. After inlining the actual processor ports and their connections to the ports of the hardware modules are exposed and visible, resulting in the final system model as actually seen after implementation.

Figure 37 shows the implementation of the interfacing between hardware and software in case of the vocoder after final inlining. A typical flow for transfering control and data in the vococoder would look like this:

1. The processor successively writes the block of data for the hardware onto the bus one word at a time by initiating a sequence of bus write cycles with addresses corresponding to the desired hardware module.

2. The hardware modules listening on the bus decode the addresses, take the data words from the bus and write them into their local memories if the address falls into their assigned range.

3. When the last item has been written control in the hardware module is transfered from the bus decoder to the execution of the corresponding (computational) behavior.

4. The behavior reads the values from the memory, processes them and writes the result back into the memory.

5. When the behavior has finished execution it triggers the processor by raising the interrupt line.

6. The processor in the vocoder reads the block of result data over the bus one word at a time by initiating a sequence of successive bus read cycles. Again, the hardware modules interfacing to the bus decode the corresponding adresses and supply the requested values out of their local memories.

In case of the vocoder it is assumed that the synthesized hardware will be fast enough to react to transfers initiated by the master processor at the maximal bus speed (2 processor cycles per bus transfer). Otherwise, wait states would have to be added to the bursts of bus transfers on the processor side or a more elaborate handshaking scheme (e.g. DMA or interrupt-based acknowledgements of single transfers) would become necessary.

The bus decoder and interrupt generation logic are parts of the wrapper channel which after inlining into the hardware module are combined with the computational logic. Note that in general there are many different ways of implementing the transfer functionality and a choice about the final hardware design has to be made at this point. (e.g. to combine or to separate the decoder and computation state machines).

When waiting for an event without any further processing to be done, the program suspends itself by halting the processor. For example, on the vocoder the main coder program waits for the start of a new frame after processing of the previous one has finished. After an interrupt has woken up the program it will check the flag for the correct event type and will either continue waiting or it will start processing of the received data.

Table 2 summarizes the results of protocol selection for the communication between processor and the different hardware modules in the vocoder. It lists all the address and interrupt assignments for the implementation of the synchronous, blocking message-passing communication as described in the previous paragraphs. Each message is assigned an exclusive address range.

Note that for each message the processor transfers the data items sequentially over the bus. Each data word of each message has been assigned a different address on the external processor bus. By decoding the bus addresses the hardware modules can determine which item of a message is being transfered. Note that items are transfered one word at a time and that the sequences and the transfers in a sequences are initiated by the processor in a fixed order. Hence, as an implementation alternative it would be possible to assign only a single address each for communication with the hardware modules. All data transfers between a certain hardware module and the processor would be

Message		Address range	SW Trigger	HW Trigger
Speech In		`$8000`–`$809F`	Interrupt A	
Bits/Prm Out	LSP	`$8500`–`$8504`		Write to `$8504`
	Prm 1–4	`$8505`–`$8538`		Write to `$8511`, `$851E`, `$852B`, `$8538`
Bits/Prm In	SID, TAF	`$953A`–`$953B`	Interrupt B, #0	
	BFI, LSP	`$9500`–`$9505`	Interrupt B, #0	
	Prm 1–4	`$9506`–`$9539`	Interrupt B, #1–4	
Speech Out	Subframe 1–4	`$9000`–`$909F`		Write to `$9027`, `$904F`, `$9077`, `$909F`
Codebook Data (ASIC In)		`$A000`–`$A0C9`		Write to `$A0C9`
Codebook Result (ASIC Out)		`$A0CA`–`$A124`	Interrupt C	

Table 2: Vococeder interrupt and address assignment.

	Priority
Interrupt A	high (2)
Interrupt B	middle (1)
Interrupt C	high (2)

Table 3: Vocoder Interrupt priorities.

handled using the same address on the bus. However, this would require the hardware modules to keep track of the history of data transfers in order to recognize the end of the sequence, for example. Therefore, the implementation as shown in Table 2 was chosen for the vocoder example.

On the processor side, incoming messages are assigned to different interrupts. However, due to the limited number of available interrupts all incoming parameter blocks (LSP and subframe parameters) are mapped to the same interrupt. Since the parameter order is fixed and given, different blocks are distinguished by their index in the sequence of incoming messages.

Table 3 lists the priorities assigned to the different interrupts. Interrupt priorities define the ordering in which overlapping interrupts are processed. Interrupts of lower priority are disabled while a high-priority interrupt is processed in its handler.

The vocoder priorities are selected such that incoming speech frames have priority over incoming parameter blocks. It is time-critical that the hardware speech buffer is copied into the processor as soon as it becomes full. The selected priorities ensure that this data transfer can't be interrupted and therefore will be finished before the next speech sample will arrive at the buffer. Note that interrupts A and C (incoming speech and codebook done signal) can never happen simultaneously and therefore can share the same priority.

4.5 Results

A final simulation of the communication model including interrupt handling, external data transfers, etc. was done by extending the instruction set simulator of the processor to include an emulation of the hardware module functionality at the interface to the processor (Section 5.1.3). The resulting co-simulation was used to verify functional correctness of the results produced by interface synthesis and to the obtain final timing data including communication overhead.

	Cycles	ms	Constraint
First subframe	366809	6.11	10 ms
Total frame	642351	10.71	20 ms

Table 4: Worst-case delays for vocoder in back-to-back operation.

Table 4 lists the simulation results for both constraints in terms of worst-case delays for operating coder and decoder in back-to-back mode as required by the specification (see Section 2.2.4). Also, comparing the delays of Table 4 to the estimates obtained after architectural exploration (Table 1 in Section 3.7) shows the additional delays due to the communication overhead.

As these results show, both constraints are easily satisfied and there is even room for other optimizations, trading of speed for other parameters of the design space. For example, by lowering the clock frequency power consumption can be reduced at the expense of execution times and delays.

5 Backend

At the end of the SpecC design process the final communication model is handed off to the backend tools. For the software parts code is generated which will be compiled into a program that runs on the corresponding processors. For the hardware parts high-level synthesis is performed to create an RTL description which will then be further processed using traditional logic synthesis and P&R tools.

5.1 Software Synthesis

Software synthesis is the process of generating executable machine code to run on the processors in the system. Given the final SpecC model, C code is generated for behaviors mapped to processors. Using specialized or general, retargetable compilers the C code is compiled and optimized for the given processor.

Finally, using an instruction set simulator (ISS), again either specialized or retargetable, the software generated for each behavior is simulated to obtain detailed timing information. For the final verification of the communication model the different parts of the system are cosimulated at the native C level using the detailed timing information to emulate the delays of the behaviors at their interfaces. Therefore, the functionality and the timing of the communication between the behaviors can be simulated without the need for slow, cycle-accurate simulation of the hardware and software behaviors themselves.

5.1.1 Code Generation

During code generation, the SpecC model of the software behaviors mapped to the processor is translated into a C program for that processor. Due to the fact that SpecC is based on ANSI-C this translation process is straightforward.

The software behavior hierarchy is converted into a hierarchy of C functions where the functions are called in the order given by the scheduled SpecC model. The C code contained in leaf behaviors is directly used as the body of the corresponding C function.

In addition, the behavioral C code is linked with the customized operating system kernel as determined during scheduling and communication synthesis. The operating system kernel is generated using a library of templates and standard modules. The corresponding schedulers, interrupt handlers, I/O routines, etc. are customized according to the specifics of the given processor (e.g. mapping to interrupt vector addresses).

The SpecC language includes certain extended features (e.g. bit vector data types) not available in ANSI-C. Using a library, operations can be mapped to function calls of library routines implementing the desired functionality on the target processor. In general, depending on the features of the processor in connection with their support by the C compiler, specialized operations are either emulated or directly implemented using the processor's capabilities.

Since the corresponding tools for automated code generation in the SpecC environment were not yet available, for the vocoder project the following tasks were instead performed manually:

- Scheduling of the software behavior hierarchy into a sequential hierarchy of C function calls, largely based on the model of the initial C reference implementation.
- Parts of the runtime library provided with the Motorola C compiler were linked to the vocoder code. The linked assembly code (`crt` module) is responsible for initializing the C runtime environment (e.g. stack, etc.) on the DSP core. Since they are not used by the vocoder code, the C standard library routines were not linked with the program, saving memory space.
- A customized operating system kernel consisting of interrupt handlers, I/O routines for external bus accesses, process synchronization operations and the interrupt-based dynamic scheduling of coding and decoding processes was created.
- Appropriate calls of the operating system kernel routines were inserted into the C code to synchronize with incoming events and to transfer data between the processor and the external hardware.
- The 16-bit saturated fixed-point arithmetic of the algorithms in the vocoder behaviors was implemented using native assembly instructions provided by the DSP core. Therefore, the explicit saturations and fixed-point adjustments (shifting, etc.) of the basic operations in the original specification were replaced with corresponding native assembly code and the previously generated processor initialization routines were modified to switch the processor into saturated arithmetic mode. Also, complex operations like multiply-accumulate (MAC) were mapped onto equivalent machine instructions as far as they were available in the DSP instruction set.

5.1.2 Compilation

Following code generation, the sources have to be compiled into an executable program for the chosen processor. During compilation, general and processor-specific optimizations of the code have to be performed to improve code quality. This requires good, optimizing compilers for each processor in the system. In general, a retargetable compiler provides the basis for generating optimized code for a large range of typical embedded processors.

Compilation of the C code for the vocoder's software parts was accomplished using the compiler provided by Motorola for their DSP processors. Initial compilation of the software was done with all compiler optimizations enabled, including post-processing with the assembly-level optimizer.

Unfortunately, analysis of the code produced by the compiler revealed that the Motorola compiler does a poor job optimizing for the DSP56600. The Motorola compiler is based on a retargeted GCC. However, GCC is a compiler for general-purpose processor and therefore doesn't include DSP-specific optimizations. Especially for the typical tight loops, the code produced by the compiler for the loop bodies spends most of the time spilling register data to and from memory compared to performing actual computations.

For the vocoder example the objective was to produce code that is comparable to the level that could be expected as output of a good, state of the art compiler. To get results that are similar to the ones obtained once the compilers of the SpecC environment are available the following three-step procedure was employed:

1. The generated C code of the behavior hierarchy assigned to the processor was compiled into assembly code for the DSP core using the GCC-based Motorola compiler.

2. The assembly code was profiled using the instruction set simulator (ISS) for the DSP core supplied by Motorola (see also Section 5.1.3). The results were presented in the section about initial analysis and estimation of the vocoder complexity (Section 3.3, for a complete list of results see Table 8 and Table 9, Section C.1 in Appendix C).

3. The loops in the assemble code that dominated the execution times were manually optimized. Basically, register allocation was optimized to reduce the spill code and memory moves inside the loop bodies. Modifications were made to improve execution times without increasing code or data memory size.

In general, only straightforward manual modifications were made without applying any sophisticated optimization strategies even a good compiler wouldn't be capable of doing automatically. Figure 38, Figure 39 and Figure 40 show examples of the assembly code produced by the Motorola compiler and the assembly code after optimization for a simple filtering loop (part of `Syn_filt`).

As the results (presented in Section 3.7 and Section C.1) show, this simple optimization strategy already leads to significant gains in terms of execution times:

- Up to 94% improvement of the loop execution times could be achieved.

- On average, the execution time gain for the optimized software behaviors was about 82%.

The data clearly indicates the importance of compiler techniques for system design. A system design process that produces good results mandates the availability of a good optimizing compiler.

5.1.3 Simulation

After compilation, the final program code has to be simulated to verify software synthesis results in combination with the results of hardware synthesis. Similar to the compiler aspects, a simulator for every processor in the system has to be available. Again, a retargetable simulator, possibly derived from the same processor description as the retargetable compiler, will cover simulation requirements for a large number of processor architectures. In order to get cycle-accurate results the different software blocks are simulated on retargetable instruction set simulators (ISS) that emulate the cycle-true behavior of the target processors on a simulation host machine.

Once accurate timing and delay results for each hardware and software block have been obtained the whole system can be cosimulated to verify the interaction among the different parts. For fast cosimulation the different parts of the system are simulated at the native C level, i.e. the C code of the behaviors is compiled into a native program on the simulation host. Using the previously obtained timing data the C code emulates the cycle-accurate timing and the behavior of the different hardware and software parts at their interfaces without actually simulating the cycle-per-cycle behavior inside the blocks.

```
/* Do the filtering . */
for (i = 0; i < lg; i++)
{
    s = L_mult (x[i], a[0]);
    for (j = 1; j <= m; j++)
    {
        s = L_msu (s, a[j], yy[-j]);
    }
    s = L_shl (s, 3);
    *yy++ = round (s);
}
```

Figure 38: Original C source code example.

```
    ; --- for (i = 0; i < lg; i++) {
    cmp y0,b
    jge L96
    move     r4,r5
    do  y0,L97
    ; --- s = L_mult (x[i], a[0]);
    move     y:(r2),y1
    move     y:(r5)+,x1
    MPY      x1,y1,b
    ; --- for (j = 1; j <= m; j++) {
    clr  a    r2,x1
    add #1,a
    move     a1,y:(r6+(-5))
    add x1,a
    move     a1,r4
    do  #10,L95
    ; --- s = L_msu (s, a[j], yy[-j]);
    move     r1,a
    move     y:(r6+(-5)),x1
    sub x1,a      y:(r4)+,x1
    move     a1,r0
    move     y:(r0),y1
    MAC      -x1,y1,b
    ; --- }
    move     y:(r6+(-5)),r7
    move     (r7)+
    move     r7,y:(r6+(-5))
L95
    ; --- s = L_shl (s, 3);
    ASL      #3,b,b
    ADD      #0,b
    ; --- *yy++ = round (s);
    tfr  b,a
    RND      a
    ; --- }
    move     a1,y:(r1)+
L97
    nop
L96
```

Figure 39: Assembly output of Motorola compiler.

```
  ; Do the filtering
  MOVE#12,n5
  MOVE(r5)-                       ; &(yy[-1])
  MOVE y:(r6+(-82)),n0            ; lg

  DO n0,_LOOP2                    ; for(; < lg; )
  MOVE a,r4                       ; &a[]
  MOVE y:(r4)+,x0                 ; a[0]
  MOVE y:(r1)+,x1                 ; x[i]
  MPY x0,x1,b    y:(r5)-,x1       ; L_mul(), yy[-1]

  DO#10,_LOOP3                    ; for(; m; )
  MOVE y:(r4)+,x0                 ; a[j]
  MAC-x0,x1,b    y:(r5)-,x1       ; L_msu(), yy[-j]
_LOOP3
  ASL #3,b,b                      ; L_shl()
  RND b    (r5)+n5                ; round(), &yy[i]
  MOVE b,y:(r5)                   ; store in yy[i]
_LOOP2
```

Figure 40: Assembly code after optimizations.

This cosimulation strategy provides cycle-accurate results for the interactions among system parts at a high simulation speed. Behavior and timing of hardware and software parts is emulated at the full speed of the simulation host in contrast to traditional time-consuming cosimulation where a slow simulation of the hardware at the structural or gate level is combined with a slow simulation of the software processor at the instruction level.

Due to the fact that the retargetable simulators and cosimulation engines of the SpecC environment are still under development, the cosimulation of the vocoder example had to be done using a combination of specialized standard tools and manually created simulators:

- The execution of the compiled program parts on the DSP56600 processor was simulated using the instruction set simulator (ISS) made available by Motorola [14]. The cycle-accurate execution time results presented previously were obtained this way.

- For timing-accurate simulation of the interactions between processor and external hardware in the final communication model a specialized cosimulator was developed based on the source code of the Motorola ISS for the DSP core.

As described above, for cosimulation of hardware and software an emulation of the hardware modules' behavior at the C level was added to the source code of the instruction set simulator. The ISS sources were modified such that in each cycle the conditions at the processor interface as seen by the software running on the simulated processor reflect the actual expected hardware behavior. Finally, the resulting cosimulator source code was compiled into a program to run on the simulation host.

Basically, after each simulated cycle the cosimulator program checks for accesses of the software to the processor bus, catching and handling them appropriately. In case of write cycles the values written by the simulated software are passed into a call of the C function that emulates the corresponding hardware module. On the other hand, given the known timing of the hardware modules, interrupts in the simulator are scheduled at certain regular intervals or after certain delays. Once the simulation has reached a cycle with an interrupt condition the control in the simulated processor is transfered to the appropriate interrupt handler. Finally, at bus read cycles initiated by the software the simulator supplies the values returned by the previous call of the C function for the corresponding hardware module.

For example, once the simulator program recognizes that the DSP program has triggered a codebook search by writing the search input data to the processor bus it calls the codebook search C function with the given parameters. The simulator program will then schedule a codebook search interrupt after the given hardware delay (see Section 5.2). Once the interrupt cycle is reached the interrupt condition in the DSP is simulated. The DSP program will then try to read the search result over the DSP bus and the simulator program supplies the result calculated during the C function call to the simulated read cycles.

In terms of interfacing to the external world, in the simulator program corresponding interrupts are generated at regular intervals according to the input data rates. Again, the values read by the DSP program over the processor bus are supplied by the simulator program. On the other end, the simulator program takes results produced in the simulated DSP and stores them in a file along with their timing information.

For verification of the timing constraints, the simulator program runs the vocoder in back-to-back mode as required by the specification. Output parameter blocks generated by the coder in the DSP and written to the processor bus cause the simulator program immediately to raise an parameter input interrupt. The parameters written to the bus at the coder output are directly supplied to the bus read cycles of the decoder input.

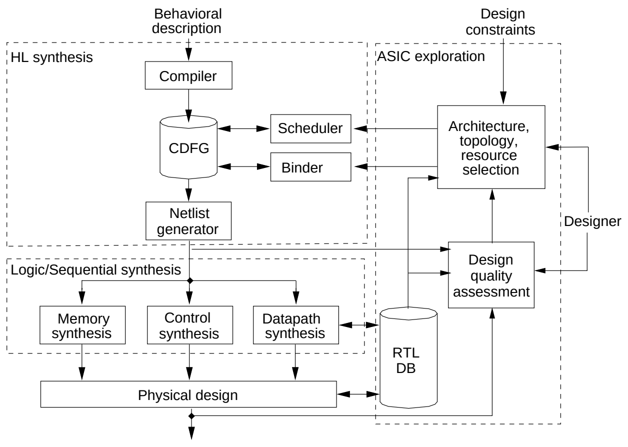


Figure 41: HLS design flow.

5.2 ASIC exploration

ASIC exploration is based on principles of *high level synthesis* (HLS) which can be defined as a translation process from a behavioral description into a register-transfer level (RTL) structural description. Usually the input to HLS tools is a behavioral description written in an HDL or a general purpose programming language. The output of a HLS tool consists of two parts: an RTL datapath structure and a description of the finite state machine (FSM) that controls the datapath. At the RTL level, a datapath is composed of three types of components: functional units (e.g. ALUs, multipliers or shifters), storage units (e.g. registers or memories), and interconnection units (e.g. busses or multiplexers). The FSM specifies a set of register transfers executed by the datapath in every control step.

A typical HLS tool design flow (Figure 41) usually starts with a pre-synthesis step in which the behavioral description is compiled into an internal representation such as control/dataflow graph (CDFG). It often includes a series of compiler optimizations such as code motion, dead code elimination, constant propagation, common subexpression elimination, loop unrolling etc. This step is followed by the core HLS process which typically contains three tasks: scheduling, resource allocation and binding. Scheduling assigns operations of the behavioral description to control steps. A control step usually corresponds to a cycle of the system clock, the basic time unit of a synchronous digital system. Resource allocation chooses functional units and storage elements from the component library. There may be several alternatives among which the synthesis tool must select the one that best matches the design design constraints and maximizes

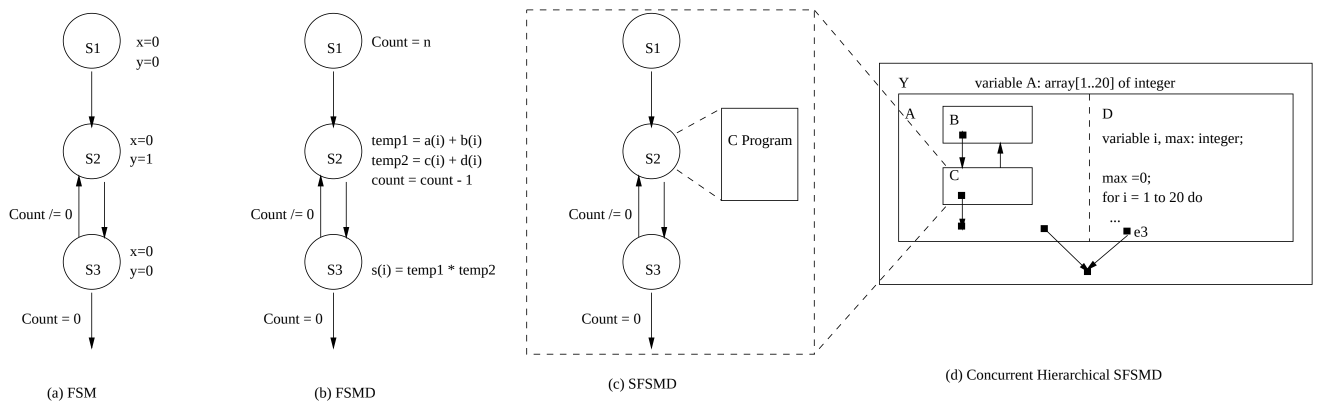


Figure 42: State-oriented models.

the optimization objective. Binding assigns operations to functional units, variables to storage elements, and data transfers to wires or busses.

Those three steps will be addressed in detail in the following sections using the hardware design of the codebook search ASIC in the vocoder project as an example.

5.2.1 Behavioral Model

The input to HLS is a behavioral description which specifies the hardware functionality. A natural language description is often ambiguous and incomplete. Therefore, a more formal model is needed. A model is a system consisting of objects and composition rules. It provides a high-level view of a system in which different details are abstracted for different applications.

Generic models Figure 42 shows several state-oriented models for describing the behavior. A *finite-state machine* (FSM) is the most popular control model. It consists of a set of states, a set of transitions between states, and a set of assignments to boolean control variables associated with these states. Traditionally, every state is associated with one control step or one clock cycle. However, for a computationally intensive system a FSM model may suffer from a state explosion problem. For example, a 16-bit integer data value represents 65536 different FSM states. To solve this problem the FSMD (*FSM with datapath*) model is introduced where non-boolean variables and complex data structures can be used in state assignments in order to reduce the numbers of states. However, neither FSM nor FSMD models are suitable for specifying complex systems since neither one supports concurrency and hierarchy which are the two essential characteristics exhibited by the real world system. Therefore, *superstate FSMD* (SFSMD) and *concurrent hierarchical SFSMD* (CHSFSMD) are introduced. A SFSMD extends a FSMD by adding the concept of a superstate which associates a behavior, an algorithm or a program with each state. In this case, each superstate is assumed to execute in more than one control step. Finally, a CHSFSMD adds hierarchy and concurrency to the SFSMD model. The CHFSMD model basically consists of a hierarchy of program states in which each program state represents a computational functions or procedures. At any given time only a subset of program states will be active. Within its hierarchy the model consists of composite and leaf program states. A composite program state is a state that can be further decomposed into either concurrent or sequential program substates. If they are concurrent all the program substates will be active whenever the program state is active. If they are sequential the program substates are actived one at a time while the program state is active [8]. SpecC supports this CHFSMD model.

Model for vocoder hardware The vocoder is specified by a simplified CHFSMD model in which there are only TOC (transition on completion) arcs but no TI (transition immediately) arcs (Figure 22). The TOC arcs not only specify the control transitions but also imply that the data will be ready for the next superstate behavior. After hardware/software partitioning some TOC arcs are crossing the boundary between the processor and the custom hardware as shown in Figure 43. To maintain the semantics regarding the availability of data a separate FSMD has to be added which implements the data transfer for each TOC arc between hardware and software. These FSMDs are supposed to be introduced during communi-

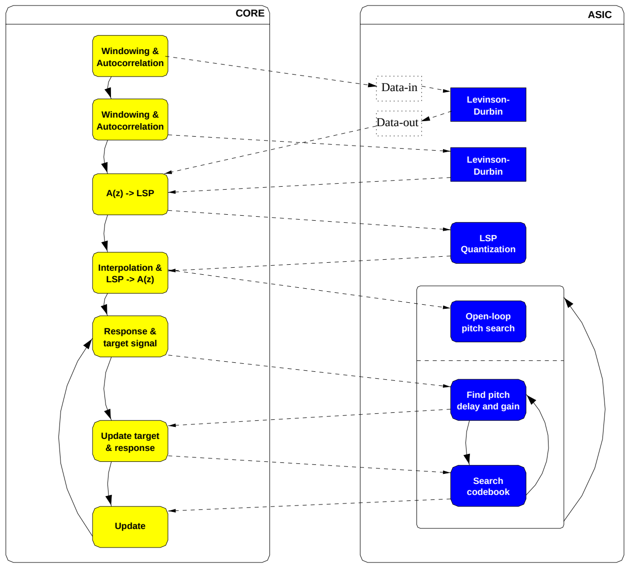


Figure 43: The sample encoder partition.

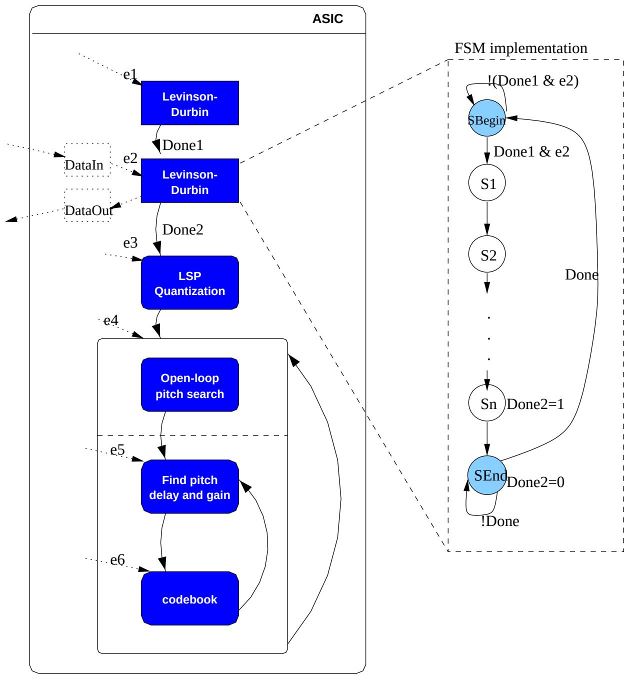


Figure 44: Scheduled encoder ASIC partition *(Note: DataIn and DataOut FSMD for behaviors other than the 2nd Levinson-Durbin are omitted.)*

cation synthesis through channel insertion and protocol selection, etc. For example, a simple handshaking protocol could be selected to synchronize the hardware and software execution by exchanging *Start* and *Done* signals between the processor and the ASIC. A full-fledged CHFSMD model with TI arcs complicates the hardware model after partitioning significantly. This issue is still an ongoing research topic which will not be discussed in this report.

The CHSFSMD model needs to be described with a language which can support both hierarchy and concurrency. In this project, SpecC was used to specify the CHSFSMD. For comparison, the VHDL model for the codebook search has also been included in Appendix F. At this level it is not much different from the SpecC code except for some syntax variations and the fact that the pointers in the SpecC code have been converted to array accesses. However, expressing concurrency in VHDL at this level would be more difficult than in SpecC because of the different semantics of signals and variables in VHDL. In VHDL a process can be used to model concurrency of leaf program states such as blocks `prefilter` or `pitch_contr` shown in Figure 45 and Figure 46. However, only signals can be used to communicate data between processes. Unfortunately, signals are not efficient in modeling algorithms because of their delta delay property. Generally speaking, algorithms always assign values to temporary storage and use them immediately in the following computation. If signals are used instead, one needs to insert many *"wait 0 ns;"* statements in order to ensure correct data values. This makes it unnatural for the designer to describe algorithms using VHDL at the behavioral level.

Usually, each hardware module has a single thread of control because the behaviors in the module typically share the same datapath. Therefore, the CHSFSMD model has to be scheduled at the behavioral level such that all concurrent behaviors are serialized. Conceptually, additional TOC arcs between those concurrent behaviors will be inserted as shown inFigure 27. The execution of a behavior is triggered when its incoming TOC arc is traversed. To implement the semantics of this model each behavior functionality is adjusted as shown in Figure 44. One initial state will be inserted before the behavior in which the incoming TOC arc conditions will be checked to determine whether the behavior should be executed. In the last state of each behavior instance i, a completion signal $Done(i)$ signal is asserted. In addition, an end state will be added after the behavior in which the $Done(i)$ signal is de-asserted and the completion signal *Done* of the top level behavior will be polled to determine whether to advance back to the initial state. In this way the ordering and the mutual exclusiveness among the behaviors will be preserved. Moreover, this provides additional flexibility in implementing the con-

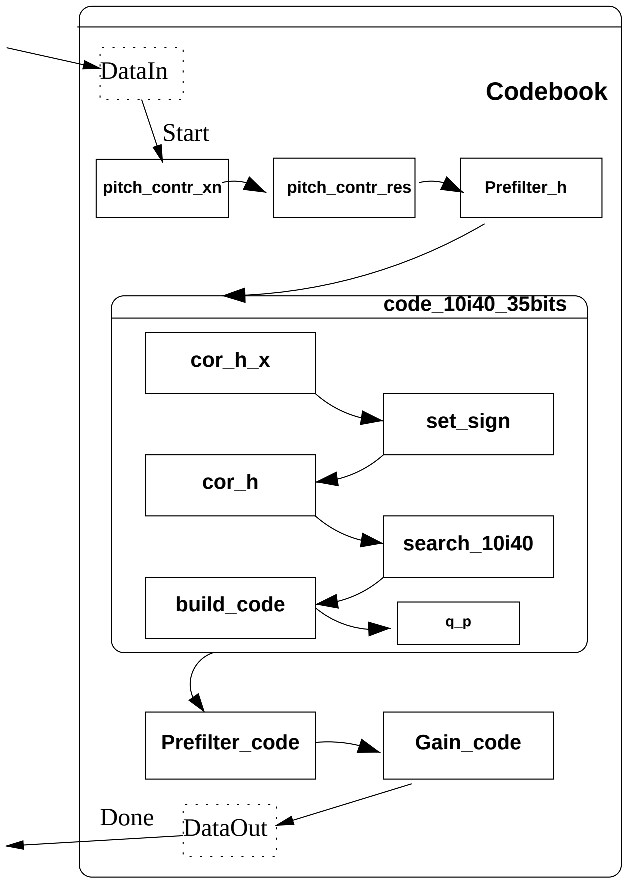


Figure 45: The scheduled codebook search CHSFSMD model.

troller. The controller can be implemented as one combined FSM or several separate FSMs. In the former case only the the initial and end states of each behavior have to be removed. All behavior states will be chained to form the controller FSM. For the latter case the controller will be decomposed so that each behavior has its own control FSM and all the behavior FSMs are coordinated with each other through *Done(i)* and *Done* signals. The independent FSMs' control outputs are combined in the primary output logic (e.g. OR gates) to form the datapath control word. Figure 45 shows the scheduled CHSFSMD model of the codebook search algorithm which has been selected for the final hardware implementation. The following discussions will be based on this model.

So far we have discussed the hardware system from a control-flow view only. A data-flow view (Figure 46) shows the exact I/O relationships between the partitioned hardware and software parts. It is obtained by data-flow analysis which can determine the input data, i.e. variables that are alive at the point of entrance, and the output data, i.e. variables that are defined/redefined in the hardware portion and need to be alive across the exit point. This data-flow view actually implies the memory hierarchy of the implementation. Despite registers and register files in the design, a larger memory is also needed as a buffer that holds the input/output data and big temporary array variables such as *rr[40][40]* inside *code_10i40_35bits* in Figure 46.

5.2.2 Architecture Exploration

Up To this point, we have focussed on how the system should be described and modeled. An architecture is has to be dervied in order to specify how it will actually be implemented. HLS is a process of turning the model into an architecture under given constraints.

Architecture selection The architecture model for exploration always consists of a control unit and a datapath. A generic implementation is shown in Figure 47. The control unit is usually described with a FSM. It contains a set of state registers and two combinatorial blocks computing the next-state and output functions, respectively. The FSM in this project is state-based (a so-called Moore machine), i.e. the FSM output depends only on the current state. As a result some extra states will be introduced as opposed to a transition-based FSM. On the other hand, the critical path length will be reduced. The datapath consists of functional units, storage units and interconnection units as shown in Figure 47. The exploration of the datapath consists of selecting the functional unit types and numbers, their connectivity and pipelining stages.

Scheduling and Allocation Scheduling and resource allocation are two major tasks of HLS architecture exploration.

Resource allocation determines the number and types of RT components to be used in the design. Components are taken out of a library which may contain multiple types of functional units, each with different characteristics (e.g. functionality, size, delay and power dissipation).

Scheduling is the key to determining whether architecture exploration will satisfy the timing constraints. It assigns operations in the behavioral description to control steps which correspond to clock cycles. The number of clock cycles along with the clock period thus determine the execution time of the hardware. Scheduling also affects resource allocation. Remember that within a control step a separate functional

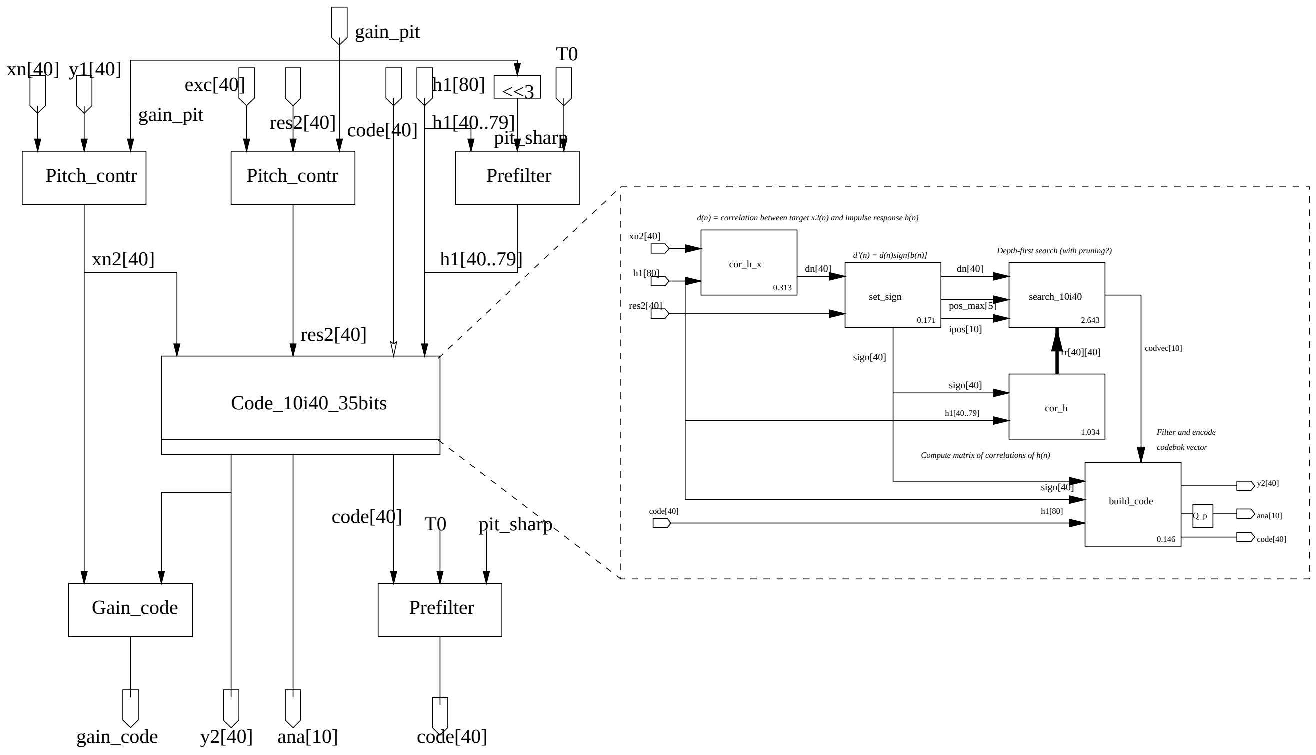


Figure 46: Data-flow view of codebook search behavioral model.

unit is required to execute each operation assigned to that step. Hence, the total number of functional units required in a control step directly corresponds to the number of operations scheduled into it. If more operations are scheduled into each control step more functional units are necessary which results in fewer control steps for the design implementation. On the other hand, if fewer operations are scheduled into each control step fewer functional units are needed but more control steps are required. Therefore, scheduling is the most important factor in determining the tradeoff between design cost and performance.

Scheduling and allocation are closely interdependent. For example, an optimal schedule of operations to control steps without explicit information about performance and cost of allocated components is impossible. Similarly, an optimal allocation of components cannot be performed without exact information about their computation profiling data. Furthermore, performance/cost tradeoffs have to be considered when performing scheduling and allocation. For example, the most area-efficient design consists of the minimum number of the slowest components that requires the largest number of control steps. On the other hand, allocating more components allows to exploit parallelism resulting in higher performance at the expense of area cost. Hence, a design space can be constrained by adjusting parameters such as resource limits, timing requirements or both. In a HW/SW codesign environment it is almost always the case that the hardware part is intended to achieve some speedup over a software solution. Therefore, timing constraints are usually the dominant factor while resource constraints play a secondary role.

For these reasons, a timing-constrained approach for combining scheduling with resource allocation was used in this project. The major steps are shown in Figure 48. It has two phases: in the first phase, we tried to find a feasible and reasonable solution that can satisfy the timing constraints by exploiting the parallelism in the specification and minimizing the number of resources needed to satisfy the timing constraints. In the second phase the allocated resources were adjusted to reduce the implementation cost while still satisfying the timing constraints.

The first phase starts with an initial set of resources, i.e. storage elements for every variable, functional units for every operator and connections for every data transfer. We initially chose the fastest resource or the one with the maximum number of stages

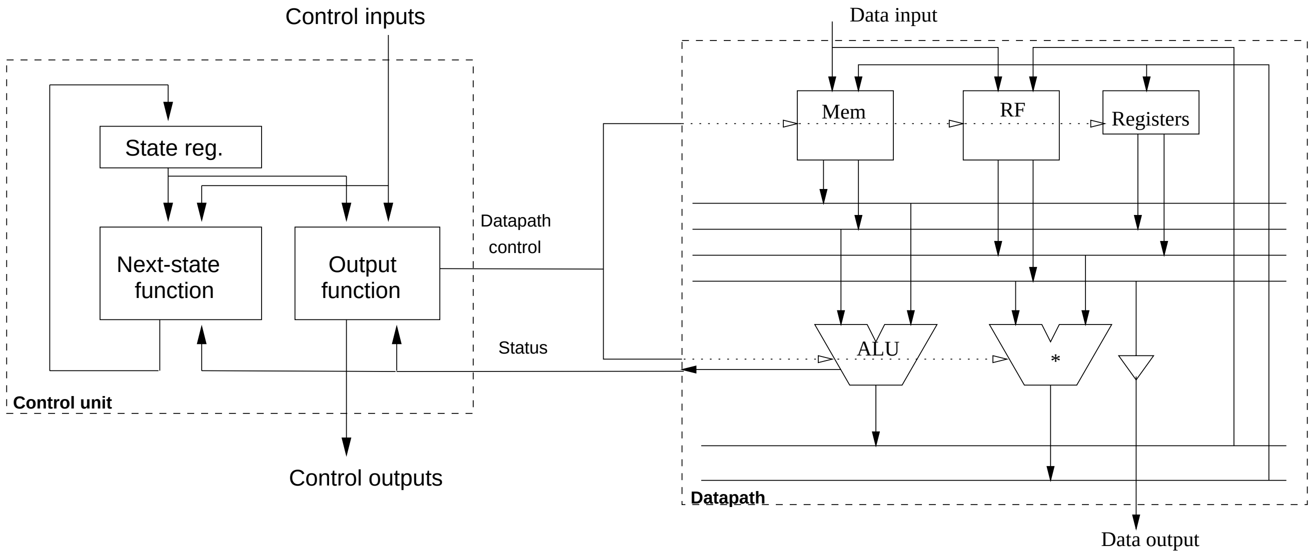


Figure 47: A generic control unit/datapath implementation.

in case of a pipelined resource to get the best performance.

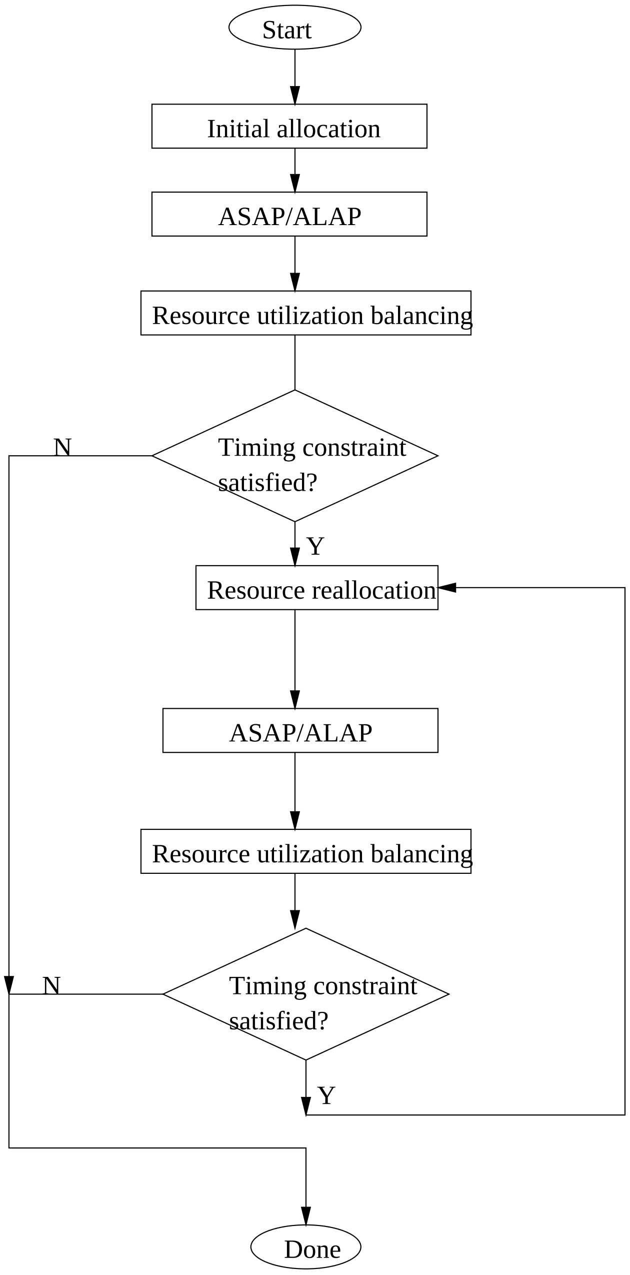
Next, the feasibility analysis phase was started by gradually exploiting the parallelism in the specification. Simple ASAP (as soon as possible) and ALAP (as late as possible) scheduling algorithms were performed to expose the parallelism based on the data dependencies. The goal was then to find a schedule which utilizes the components maximally therefore requiring a minimal number of components. We achieved this objective by uniformly distributing operations of the same type into all available control steps. A uniform distribution ensures that resources allocated to operations in one control step are used efficiently in other control steps, leading to a high resource utilization rate. The expected operator cost (EOC) for any operation type in each control step is given by the product of the resource cost and the sum of the probabilities that this operation will be scheduled into this control step. Finally, the goal is to balance the EOC value for each operation type. See [7] for further details. The schedule is checked to find whether the timing constraints can be satisfied. If the timing constraint is satisfied a feasible schedule has been found, otherwise the process fails.

The first phase of the algorithm finds the fastest solution which guarantees to satisfy the performance constraints. However, the cost of such a solution may be excessive due to the large number of resources allocated. Therefore, the objective of the second phase is to minimize the number of the resources types, therefore minimizing the overall cost of the hardware implementation while still satisfying the performance constraints.

First, we decided to choose a strategy to relax the stringent condition for the components by using slower components, less pipelined components, multifunctional units or multiport memories. Alternatively, the number of operation types can be reduced by eliminating the least utilized resource. For example, the utilization profile of the operations in the codebook search algorithm is illustrated in Figure 49. Operation *div_s* can be chosen to be eliminated, replacing it by an algorithm performing *div_s* using sub and shift operators. Rescheduling is then required in order to ensure that the timing constraints are still satisfied. This process is repeated until the timing constraint is violated or the cost cannot be reduced any further.

RTL behavioral model The strategy above will generate a clock accurate schedule that can be described by an RTL behavioral model which is depicted as a FSMD in Figure 50. This scheduled RTL model can be used for synthesis by RTL synthesis tools that have the capability to perform binding. In addition, it can be used to verify the functional correctness of the schedule by simulation.

With the scheduled result the final resource allocation—resource types and quantities–can be obtained fairly straightforward. The results for the codebook search are summarized in Table 5 and Figure 51.



Start
Initial allocation
ASAP/ALAP
Resource utilization balancing
Timing constraint satisfied?
N
Y
Resource reallocation
ASAP/ALAP
Resource utilization balancing
Timing constraint satisfied?
N
Y
Done

Figure 48: Hardware exploration.

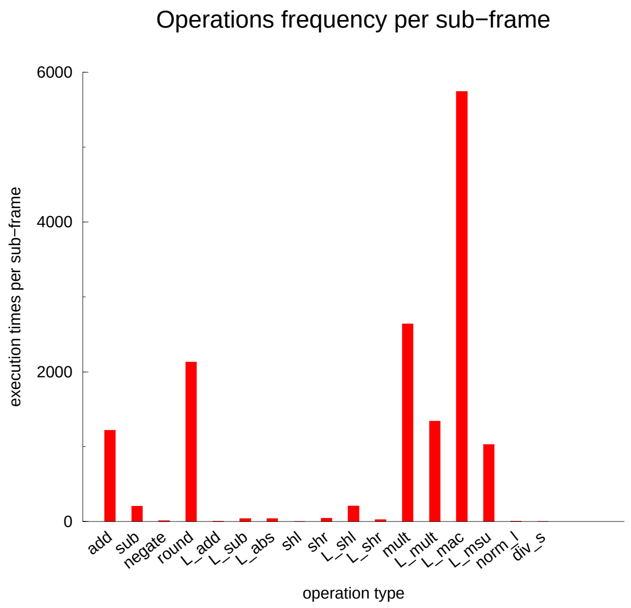


Figure 49: Operation profile for one sub-frame.

FU	Operations	#	Delay(ns)	Area(μm^2)
ALU	add,sub,negate,round, L_add,L_sub,L_abs	1	3.02	99531.25
Shifter	shl,shr,L_shl,L_shr	1	3.00	128171.87
Multiplier	mult,L_mult	1	4.09	271212.50
MAC	L_mac,L_msu	1	4.79	479598.43
NORM	norm_l	1	3.00	8429.68
MEM	storage access	1	2.6	1550156
REG32	temp. storage access	5	.75	27442.18
REG16	temp. storage access	4	.73	11578.12
COUNTERS	array index generation	4	.40	6987.50

Table 5: Functional Unit Selection Result.

Binding Binding is the process of mapping the variables and operations in the scheduled RTL model onto functional, storage and interconnection units while ensuring that the design functions correctly on the selected set of components. For every operation in the RTL behavioral model a specific functional unit that is capable of executing the operation is needed. For every variable that is used across several control steps in the scheduled RTL model a storage unit to hold the data values during the variable's lifetime has to be selected. Finally, for every data transfer we need a set of interconnection units that will handle the communication.

Functional Unit Binding Having selected a set of units, functional unit binding in this project was straightforward. Each operation in the behavioral description can be mapped onto one of the selected functional units only.

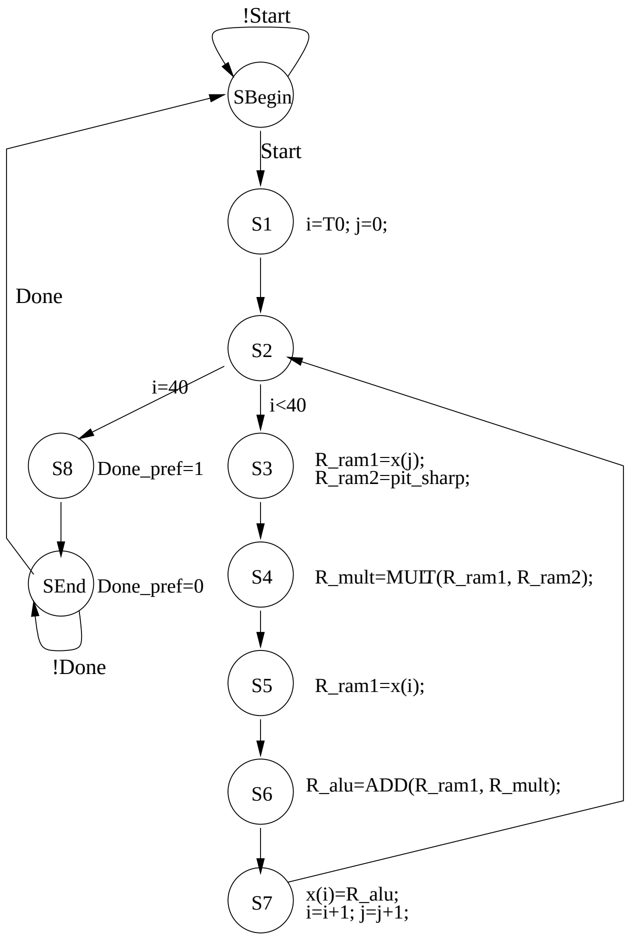


Figure 50: Behavior `prefilter` FSMD.

Storage Binding Storage binding maps data carriers (e.g. constants, variables, and data structures like arrays) in the behavioral description to storage elements (e.g. ROMs, registers and memory units) in the datapath. Constants, such as coefficients in a DSP algorithm are usually stored in a ROM. If their number is small they can be hardwired to VCC or ground like the 16 constants in the codebook search algorithm. Variables are stored in registers or memories. As discussed previously, variables whose lifetime intervals do not overlap with each other may share the same register or memory location. Actually the last step of register allocation in the scheduling algorithm has already determined the binding of the variables, i.e. selecting whether a variable will live in a register or in the memory. The memory addresses of each variable are listed in Table 6.

Address	Variables
0	xn[40]/y32[40]
40	y1[40]
80	xn2[40]
120	exc[40]/dn[40]
160	res2[40]
200	h1[80]
280	sign[40]
320	rr[40][40]
1920	h2[40]/en[40]/rrv[40]/scal_y2[40]
1960	code[40]
2000	y2[40]
2040	codevec[10],ana[10],_sign[10],ipos[10]
2080	posmax[5]

Table 6: Memory Addresses.

Interconnection Binding Every data transfer (i.e. a read or a write) needs an interconnection path from its source to its sink. Two data transfers can share all or part of the interconnection path if they do not take place simultaneously. The objective of interconnection binding is to maximize the sharing of interconnection units. Therefore, interconnection cost is minimized while conflict-free data transfers required by the register-transfer description are still ensured.

RTL structural Model The unit binding generates the final RTL structural model which consists of a control unit and a datapath as mentioned earlier.

The control unit can be obtained by synthesizing a single central FSM in which the chained behavior FSM are merged together. Alternatively, when the combined FSM becomes too large each behavior FSM can be synthesized into a separate control unit and

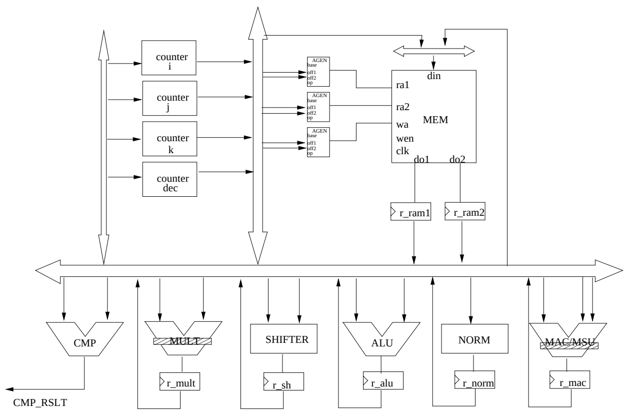


Figure 51: Datapath diagram.

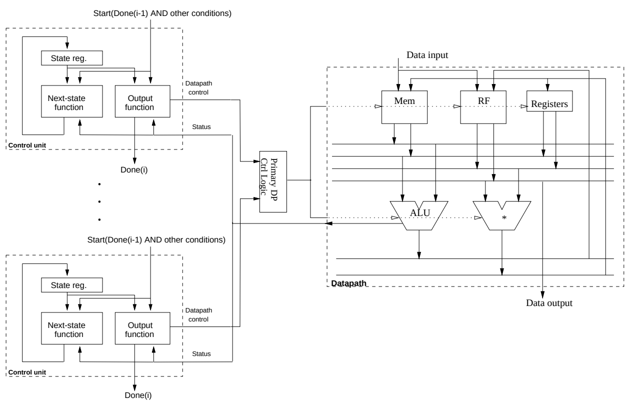


Figure 52: A FSMD implementation with a decomposed-CU

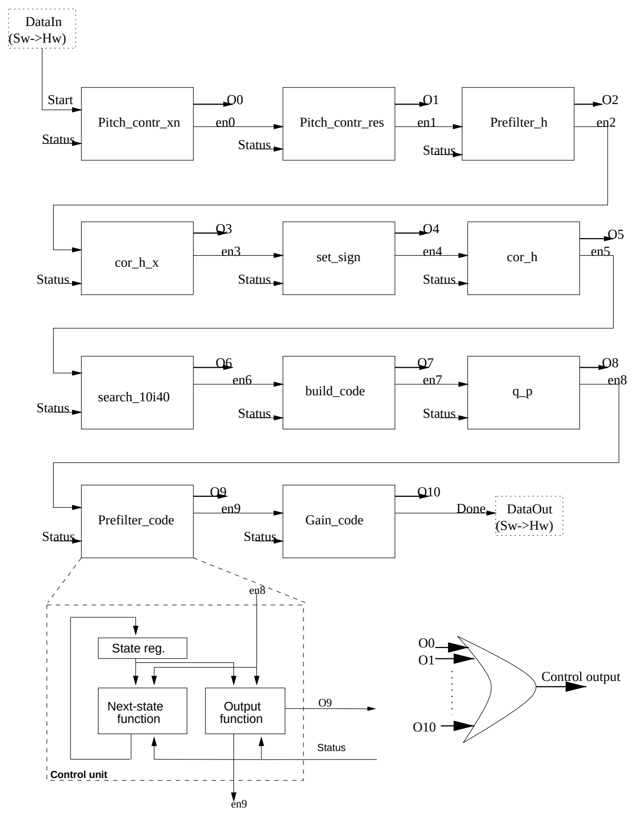


Figure 53: Control unit decomposition

connected as specified in Figure 44 The hardware architecture of the decomposed design in shown in Figure 52. One motivation for the decomposed controller is that the heuristics algorithms for state assignment and logic optimization used in logic synthesis tools such as Synopsys $DesignCompiler^{TM}$ provide superior results for smaller designs. A reasonable decomposition may lead to a more economical realization in terms of area. The performance of a decomposed design may also be better due to a smaller critical path delay achieved by logic optimization. Due to these reasons, in this project the latter alternative has been chosen. The control unit FSM decomposition is shown in Figure 53. Every sub-FSM of the control unit can be modeled with a HDL case statement as accepted by logic synthesis tool like Synopsys $DesignCompiler^{TM}$. An example is shown in Figure 54.

The datapath has been designed with Synopsys SGE^{TM} as a schematic from which a structural VHDL netlist of components can be generated automatically. The schematic is shown in Appendix D.

5.2.3 Performance analysis

The two most important quality metrics are the cost and performance. The most common cost metrics is the design area which is a measure of the silicon area required by the implementation.

Area For a given FSMD design the area cost includes the area needed for the control unit, the datapath and the wiring area required to connect these components.

Datapath The datapath consists of three kinds of RT components: storage units (memories and registers), functional units (ALUs) and interconnect units (busses and multiplexers). The total area of the datapath as the sum of the three kinds of component areas is a total of $3,847,342\,\mu\text{m}^2$. Wiring also contributes to the overall area. Unfortunately, estimating the wiring area requires knowledge about the placement and the physical layout of the units. Fast floor planners have been used by engineers to obtain this placement information. Alternatively, statistical wiring models have been used. In this project we arbitrarily assumed that wiring requires 10% of the components area. Hence, the total area of the datapath is approximately $4,232,076\,\mu\text{m}^2$.

Control Unit The control unit of the hardware uses the FSM decomposition approach and is decomposed into 11 sub-FSMs(Figure 53). The area of the control unit can be simply calculated by adding the are for the 11 FSMs area and the area for the big OR gate, totalling $1,094,255\,\mu\text{m}^2$.

```
...
case State is
        ------------------------------------------
        --      State PF_S0
        ------------------------------------------
        when PF_S0 =>

                ...

                --r_ram1 <= gain_pit;
                R1_BASE <= A_gain_pit;
                R1_OFF1_SEL <= dont_care(2 downto 0);
                R1_OFF2_SEL <= dont_care(1 downto 0);
                R1_OP <= "00";
                REN1 <= '1';
                RR1_LD <= '1';

                --r_ram2 <= CONV_STD_LOGIC_VECTOR(T0, 16);
                R2_BASE <= A_T0;
                R2_OFF1_SEL <= dont_care(1 downto 0);
                R2_OP <= "00";
                REN2 <= '1';
                RR2_LD <= '1';

                ...

                Next_State <= PF_S1;

        ------------------------------------------
        --      State PF_S1
        ------------------------------------------
        when PF_S1 =>

                ...
```

Figure 54: sub-FSM in VHDL

Hence the total estimated area of the design is

$area = area(CU) + area(DP) = 5,326,331\,\mu\mathrm{m}^2 \approx 5\,\mathrm{mm}^2$

Performance Performance metrics can be classified into three categories: clock cycles, control steps and execution times. Execution time is the final measure and the other two metrics contribute to its calculation.

We define the execution time as the time interval needed for process the complete input data set and generating the complete output data set. This will cover both a pipelined and a non-pipelined design. If the number of clock cycles of the interval is *num_cycles* and the clock cycle delay is *clock_cycle* the execution time can be computed as follows:

$execution\_time = num\_cycles \times clock\_cycle$

Clock cycle Given the FSMD design shown in Figure 56 the clock cycle can be determined as the maximum of the critical path candidates as follows:

(a) Delay of path $p1$, computing the next state of the FSM:

$\Delta(p1) = delay(SR) + delay(CL) + delay(CMP) + delay(NL) + setup(SR) \approx 8.9\,\mathrm{ns}$

(b) Delay of path $p2$, reading data from the memory:

$\Delta(p2) = delay(SR) + delay(CL) + delay(AGEN) + delay(MRD) + setup(MR) \approx 8.5\,\mathrm{ns}$

Units	Delay(ns)
AGEN	1.94
MEM	2.6
MULT	4.09/2
SHIFTER	3.0
ALU	3.02
NORM	1.25
MAC	4.79/2
CMP	1.22
REG32	.75
REG32(setup)	.59
REG16	.73
REG16(setup)	.59
CU	3.85

Table 7: Unit delays.

(c) Delay of path $p3$, performing the arithmetic operation:

$\Delta(p3) = delay(SR) + delay(CL) + delay(AU) + setup(RR) \approx 7.0\,\mathrm{ns}$

where $delay(SR)$ is the delay of reading the state registers, $delay(CL)$ is the delay of the control logic, $delay(CMP)$ is the delay of the comparator, $delay(NL)$ is the delay of next state logic, $setup(SR)$ is the setup time of the state registers, $delay(AGEN)$ is the delay of the address generation unit, $delay(MRD)$ is the delay of reading the memory, $setup(MR)$ is the setup time of the register connected to the memory read port, $delay(MR)$ is the delay of reading the register connected to the memory read port, $delay(AU)$ is the delay of the ALU, $setup(RR)$ is the setup time of the registers storing the functional unit results. Table 7 lists the delays of the functional units.

Hence, the minimum clock cycle is

$clock\_cycle = Max(\Delta(p1), \Delta(p2), \Delta(p3)) \approx 9\,\mathrm{ns}$

Even when adding a 10% engineering margin the hardware part can still run at 100 MHz, i.e. with a clock cycle of 10 ns.

Number of execution cycles For a data-flow dominant design like this project the number of cycles needed for execution can simply be obtained by profiling the RTL model. A profiling result is shown as Figure 55. The number of cycles for one sub-frame of voice data samples is around 33,000. Therefore, the total number of cycles for one frame is 132,000.

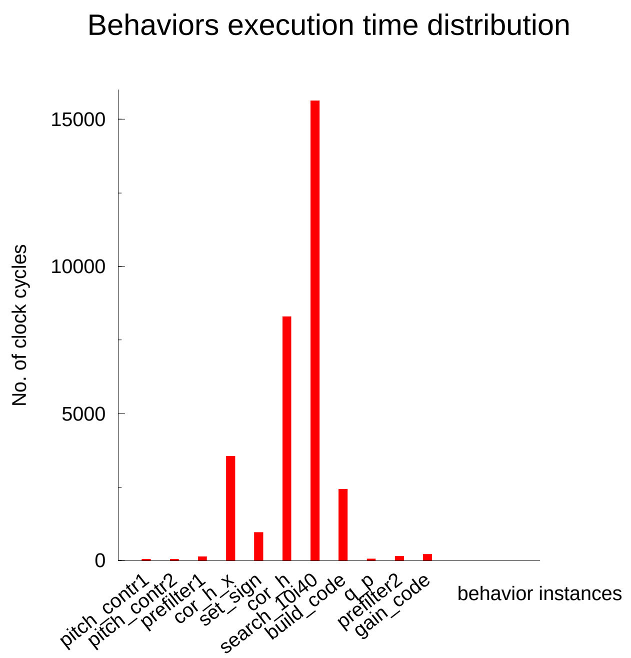


Figure 55: Execution time distribution.

The performance of the hardware part, i.e. the total execution time for one frame of voice data samples is

$$execution\_time = 132,000 \times 10\,\mathrm{ns} \approx 1.3\,\mathrm{ms}$$

6 Conclusions

In this report we presented the SpecC system-level design methodology applied to the example of designing and implementing a GSM EFR vocoder. We have shown the various steps in the SpecC methodology that gradually refine the initial specification down to an actual implementation model. The well-defined nature of the models and transformations provides the basis for design automation tools and in general enables application of formal methods, e.g. for verification or equivalence checking.

Starting with the executable SpecC specification, architectural exploration—supported by estimators and analysis tools—creates an architectural model of the design through the steps of allocation, partitioning and scheduling. We demonstrated how a large part of the design space can be quickly explored to select the best architecture. Communication synthesis then transforms the abstract communication of the architectural model into an implementation. After protocol selection, transducer synthesis and protocol inlining the final communication model is obtained.

At any point the design is represented by a model in SpecC. We perform equivalence checking and simulation on each model to validate the transformations. The SpecC language explicitly supports all the features necessary for system-level design including hierarchy, timing, concurrency, communication and synchronization, exceptions, state transitions, etc. On the other hand, the fact that SpecC is a superset of C allows to draw from the large body of existing algorithms. The clear separation of communication and computation in SpecC facilitates reuse of system components and enables easy integration of IP.

After finishing the design on the system level the communication model is handed off to the backend for synthesis of the software and hardware parts. For the software parts C code including a custom real-time operating system kernel for scheduling, task synchronization and I/O is generated, compiled, and optimized for the chosen processor. For the hardware parts a behavioral description is generated and synthesized into a custom RTL implementation using behavioral or high-level synthesis tools. The structural RTL design is then further transformed down to a gate- or transistor-level netlist using traditional logic synthesis tools.

In case of the vocoder, the initial GSM standard including the C code was analyzed and a SpecC specification was developed. The 14,000 lines of the specification were partitioned into 12,000 lines of code for a software part running on a Motorola DSP56600 core at 60 MHz and 2,000 lines of code for a custom hardware part implementing the codebook search. The final implementation of the vocoder consists of 70,500 lines of compiled assembly code and 45,000 lines of synthesized RTL code. The transcoder delay of the final implementation is 26 ms and the time for encoding and decoding a speech frame is 11 ms, easily meeting the constraints of 30 ms and 20 ms, respectively.

The design of the vocoder has been done by two people working at the project part-time over the course of six months. The schedule of the different tasks in the vocoder project is shown in Figure 57. Most of the time was actually spent on initial understanding of the standard including its complex, unstructured C code specification, and on tedious, manual software and hardware synthesis in the backend. Simply following the well-defined steps of the SpecC methodology helped to reduce the design effort significantly.

With the availability of automated tools that will cover a large part of the tedious and error-prone synthesis tasks performed mostly manually in the vocoder project the time-to-silicon will be reduced even fur-

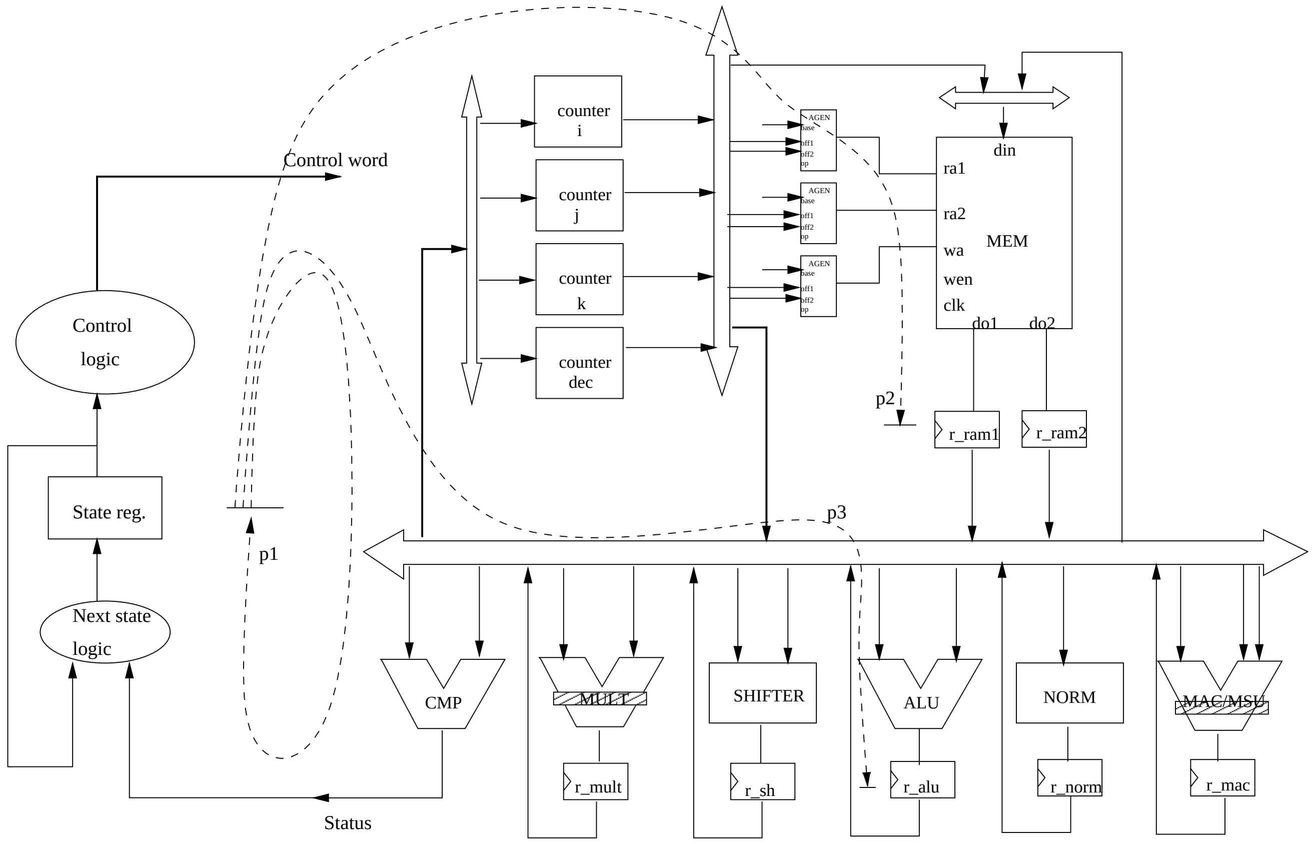


Figure 56: Critical path candidates.

ther. The time spent on the actual design tasks of the vocoder project was about 12 weeks only.

All in all, the project has shown that the SpecC methodology will result in significant productivity gains. A simplified design process based on well-defined, clear and structured models at each exploration step enables quick exploration and synthesis. In addition, a well-defined IP model allows easy integration and reuse of IP components. In general, communication among designers and customers is minimized, allowing for design and manufacturing globalization and Internet-based design strategies.

Furthermore, due to the formal nature of the design process and the models, product evolution and product customization is greatly simplified. Redesign, integration of new features and incorporation of customer feedback (e.g. in case of changing requirements) as well as upgrades to new technologies are all easily achieved. In addition, the high abstraction levels of the specification models allow easy reuse of existing models by adding or changing features as necessary or by customization of product templates for a product-on-demand business model.

Finally, focussed design concepts and design processes together with a uniform and formal methodology based on automated tools significantly reduce the amount of resources and the man power required to complete a System-On-Chip design. A steep learning curve and the low designer expertise needed reduce the training overhead and limit the demand for highly qualified designers.

Acknowledgments

The authors would like to thank Motorola for supporting this project. Also we would like to thank Lukai Cai, Hongxing Li from UCI and Justin Denison, Mike Olivarez from Motorola for help in synthesis of the codebook search.

References

[1] D. Gajski, J. Zhu, R. Dömer, *The SpecC+ Language*, University of California, Irvine, Technical Report ICS-TR-97-15, April 15, 1997.

[2] J. Zhu, R. Dömer, D. Gajski, *Syntax and Semantics of the SpecC+ Language*, University of California, Irvine, Technical Report ICS-TR-97-16, April 1997.

[3] D. Gajski, J. Zhu, R. Dömer, *Essential Issues in Codesign*, University of California, Irvine, Technical Report ICS-TR-97-26, June 1997.

[4] J. Zhu, R. Dömer, D. Gajski, "Syntax and Semantics of the SpecC Language," *Proceedings of the Synthesis and System Integration of Mixed Technologies 1997*, Osaka, Japan, December 1997.

[5] D. Gajski, G. Aggarwal, E.-S. Chang, R. Dömer, T. Ishii, J. Kleinsmith, J. Zhu, *Methodology for Design of Embedded Systems*, University of California, Irvine, Technical Report ICS-TR-98-07, March 1998.

[6] R. Dömer, J. Zhu, D. Gajski, *The SpecC Language Reference Manual*, University of California, Irvine, Technical Report ICS-TR-98-13, March 1998.

[7] D. Gajski, N. Dutt, C.H. Wu, Y.L. Lin, *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, Boston, Massachusetts, 1991

[8] D. Gajski, F. Vahid, S. Narayan, J. Gong, *Specification and Design of Embedded Systems*, Prentice Hall, Englewood Cliffs, New Jersey, 1994

[9] European Telecommunication Standards Institute (ETSI), *Digital cellular telecommunications system; Enhanced Full Rate (EFR) speech transcoding (GSM 06.60)*, Final Draft, November 1996.

[10] K. Järvinen *et. al.*, "GSM Enhanced Full Rate Speech Codec," *Proceedings ICASSP '97*, pp. 771-774, 1997.

[11] Telecommunications Industry Association (TIA), TR-46, *PCS1900 Enhanced Full Rate Codec US1 (SP-3612)*, Ballot Version, August 1995.

[12] R. Salambi *et. al.*, "Design and Description of CS-ACELP: A Toll Quality 8 kb/s Speech Coder," *IEEE Transactions on Speech and Audio Processing*, Vol. 6, No. 2, pp. 116-130, March 1998.

[13] Motorola, Inc., Semiconductor Products Sector, DSP Division, *DSP56600 16-bit Digital Signal Processor Family Manual*, DSP56600FM/AD, 1996.

[14] Motorola, Inc., Semiconductors Products Sector, DSP Division, *Motorola DSP Simulator Reference Manual*, 1995.

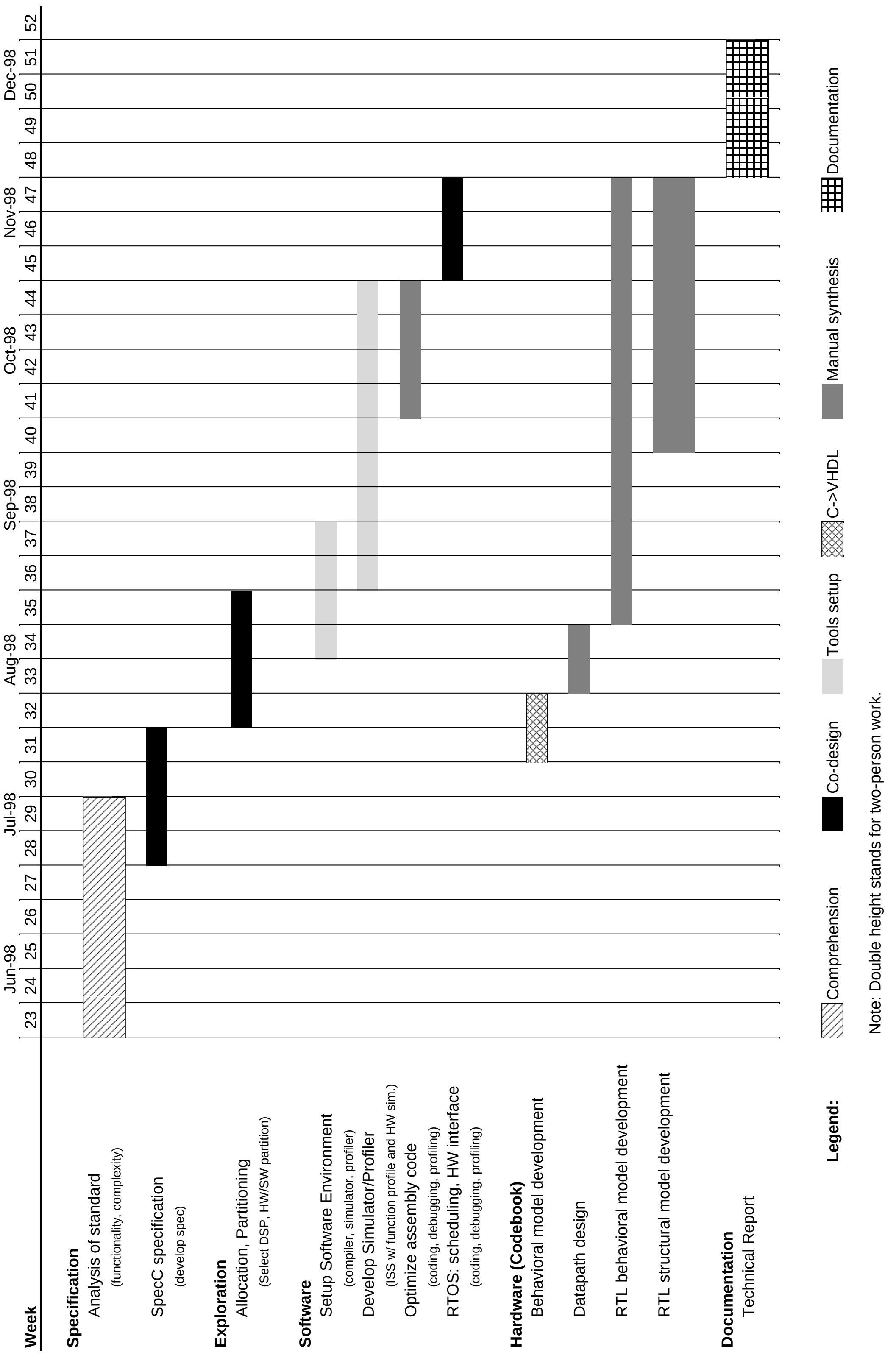
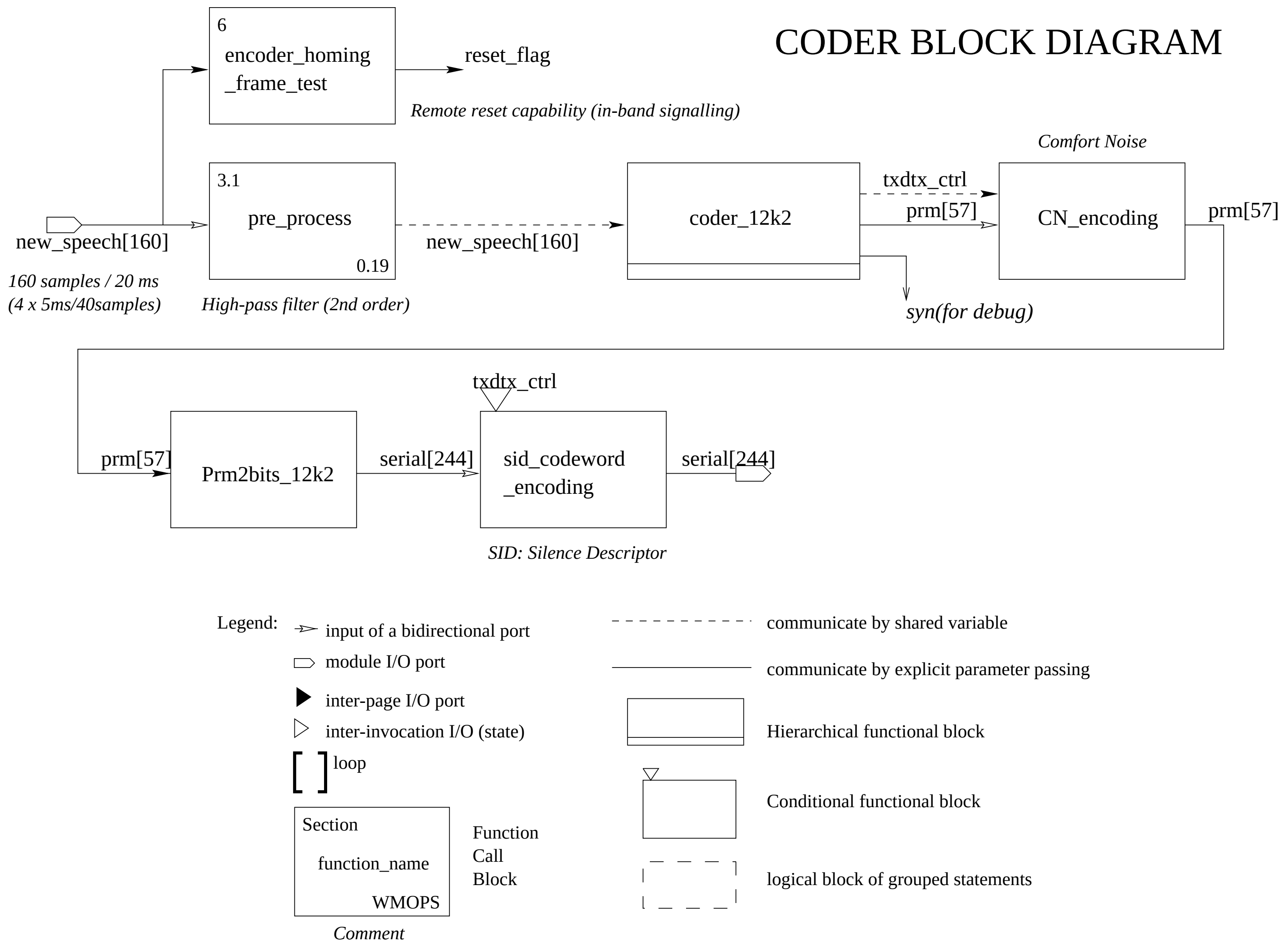


Figure 57: Vocoder project tasks schedule.

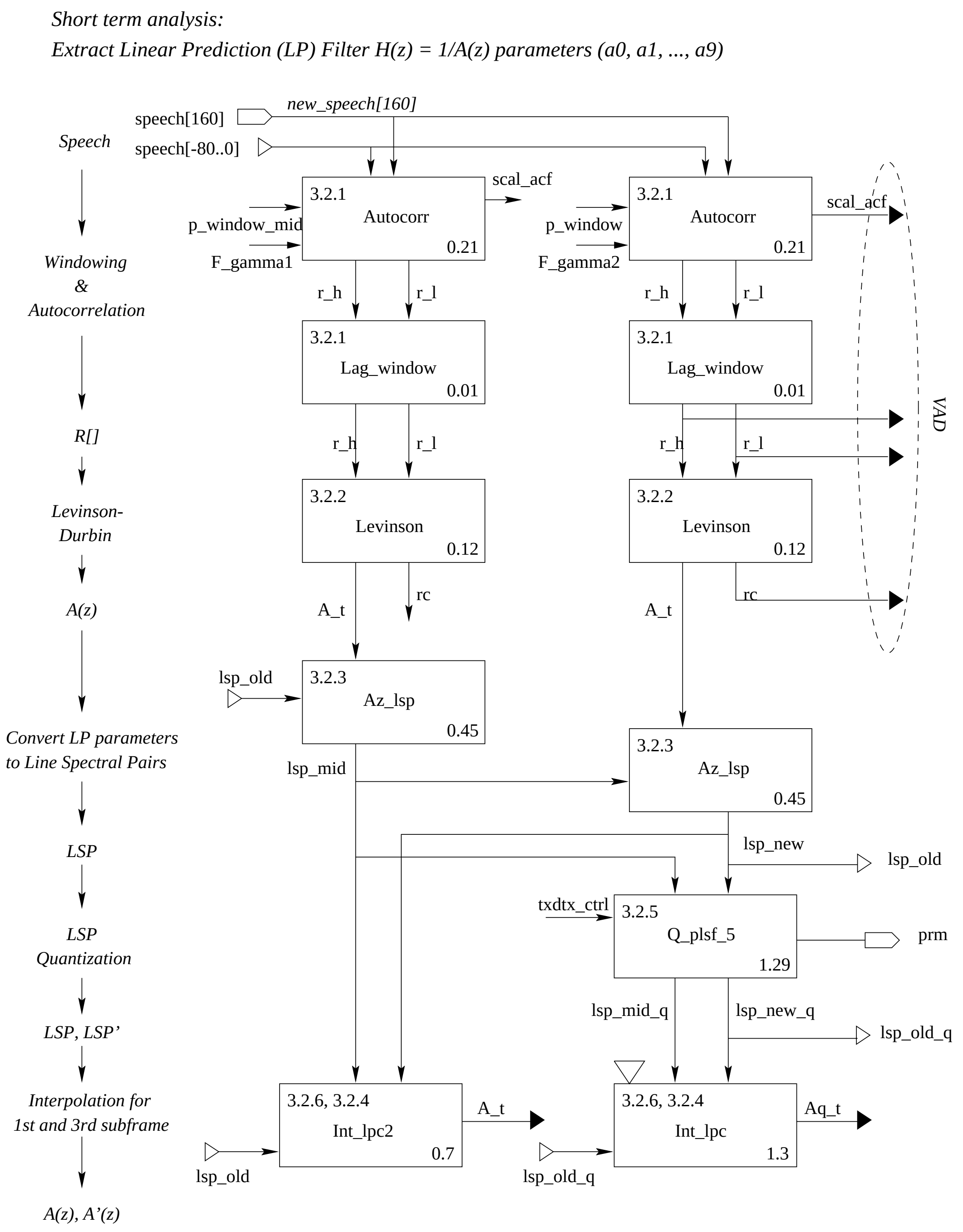
A C Reference Implementation Block Diagrams

This appendix contains the detailed block diagrams describing the architecture of the ANSI-C reference implementation of the GSM vocoder. The C implementation is supplied with the ETSI Enhanced Full Rate (EFR) speech transcoding standard GSM 06.60 [9] and it represents the bit-exact reference for any implementation of the standard.

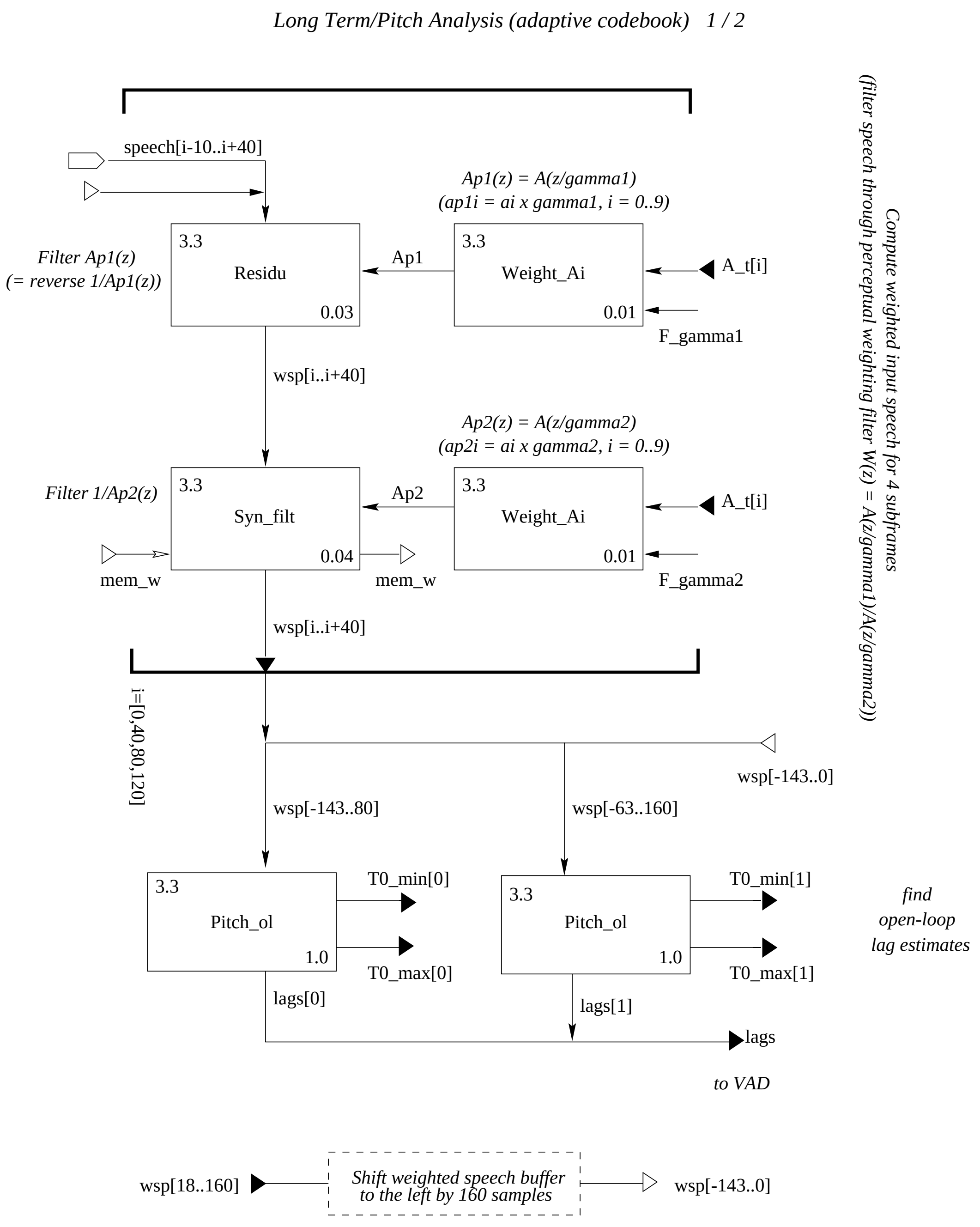

CODER BLOCK DIAGRAM
6
encoder_homing
_frame_test
reset_flag
Remote reset capability (in-band signalling)
3.1
pre_process
0.19
new_speech[160]
160 samples / 20 ms
(4 x 5ms/40samples)
High-pass filter (2nd order)
new_speech[160]
coder_12k2
txdtx_ctrl
prm[57]
syn(for debug)
Comfort Noise
CN_encoding
prm[57]
prm[57]
Prm2bits_12k2
serial[244]
txdtx_ctrl
sid_codeword
_encoding
serial[244]
SID: Silence Descriptor
Legend:
input of a bidirectional port
module I/O port
inter-page I/O port
inter-invocation I/O (state)
loop
Section
function_name
WMOPS
Comment
Function
Call
Block
communicate by shared variable
communicate by explicit parameter passing
Hierarchical functional block
Conditional functional block
logical block of grouped statements

A.1.1 Encoding: `coder_12k2`

A.1.1.1 Linear prediction analysis

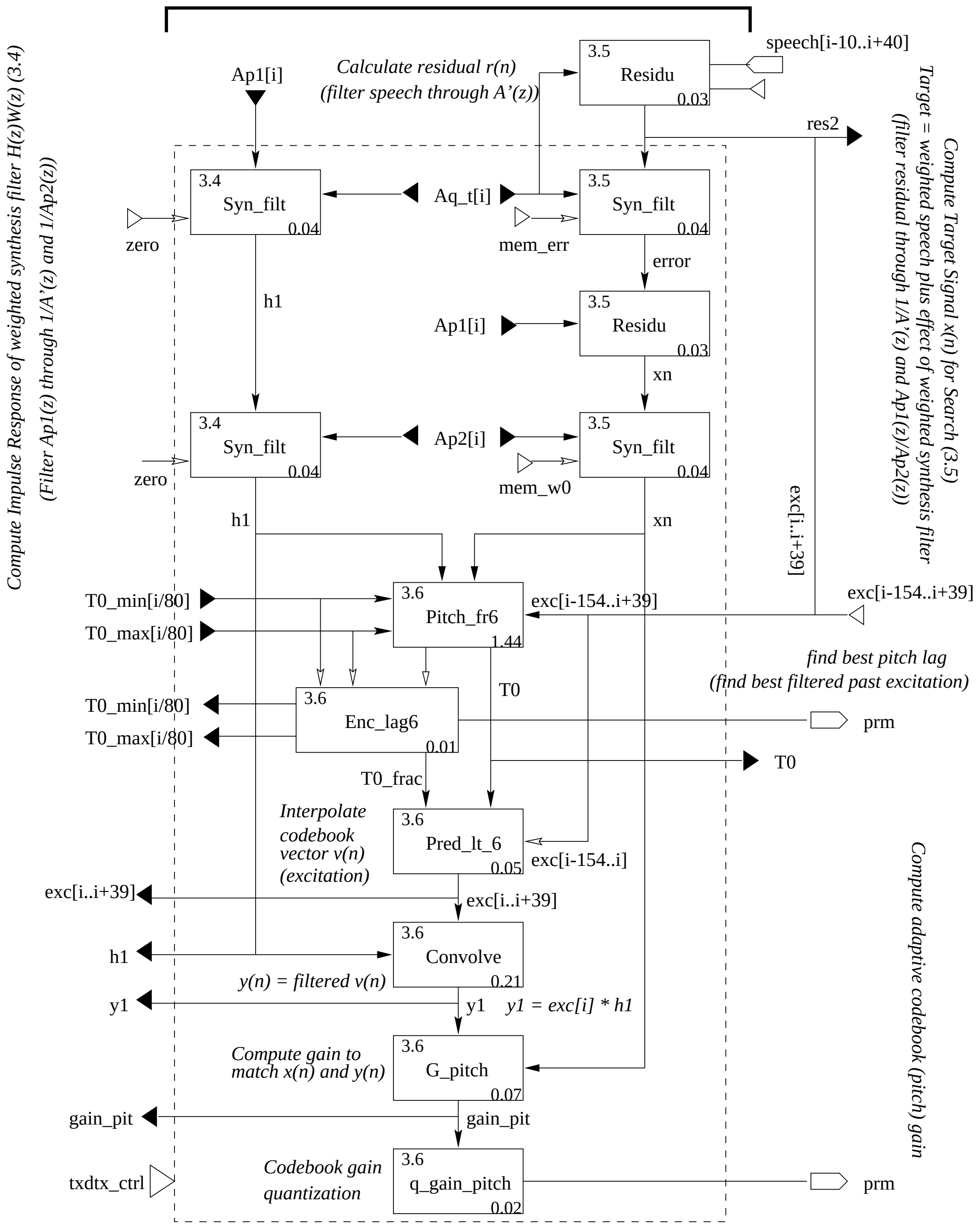


A.1.1.2 Open-loop pitch analysis

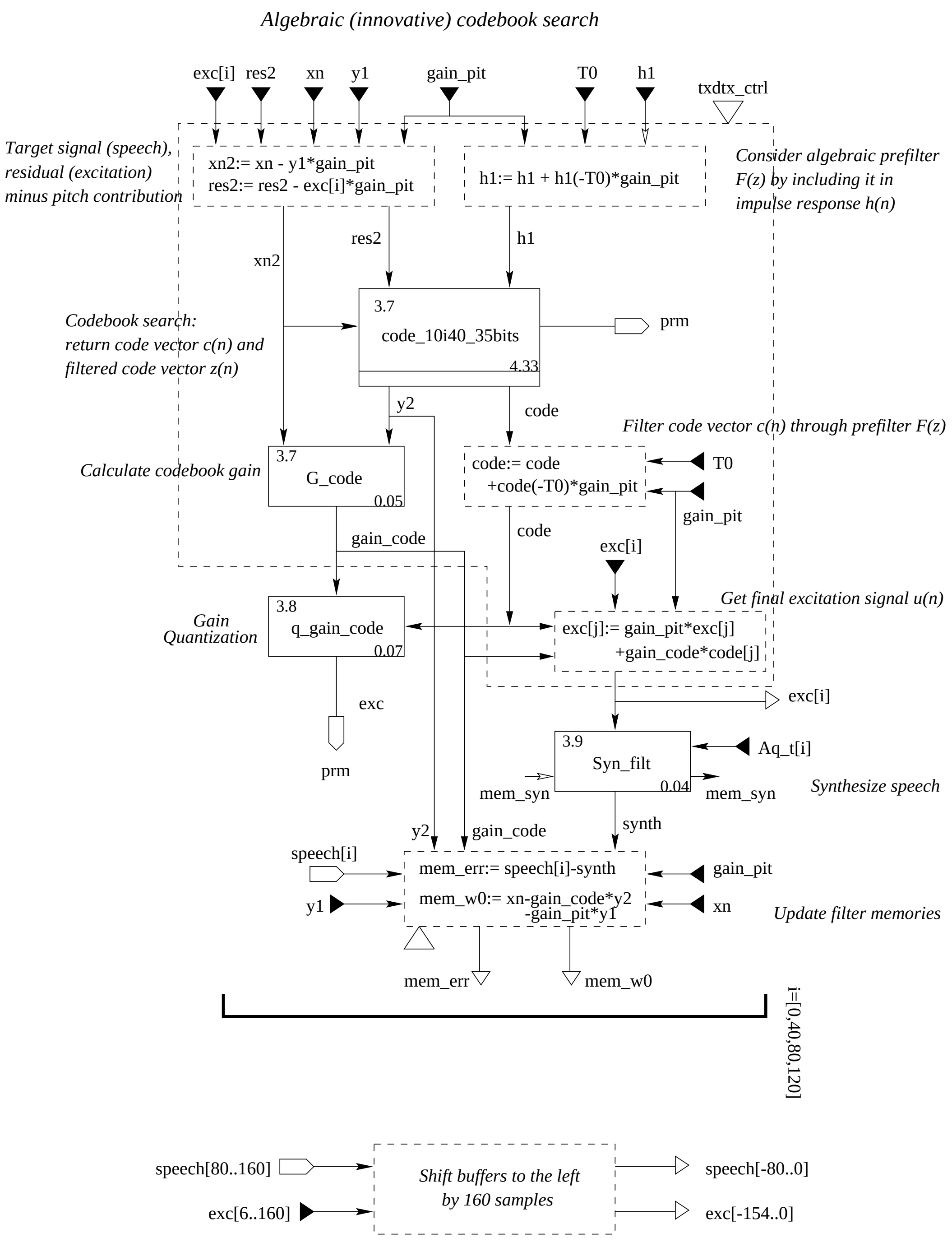


A.1.1.3 Closed loop pitch analysis

Long Term/Pitch Analysis (adaptive codebook) 2 / 2

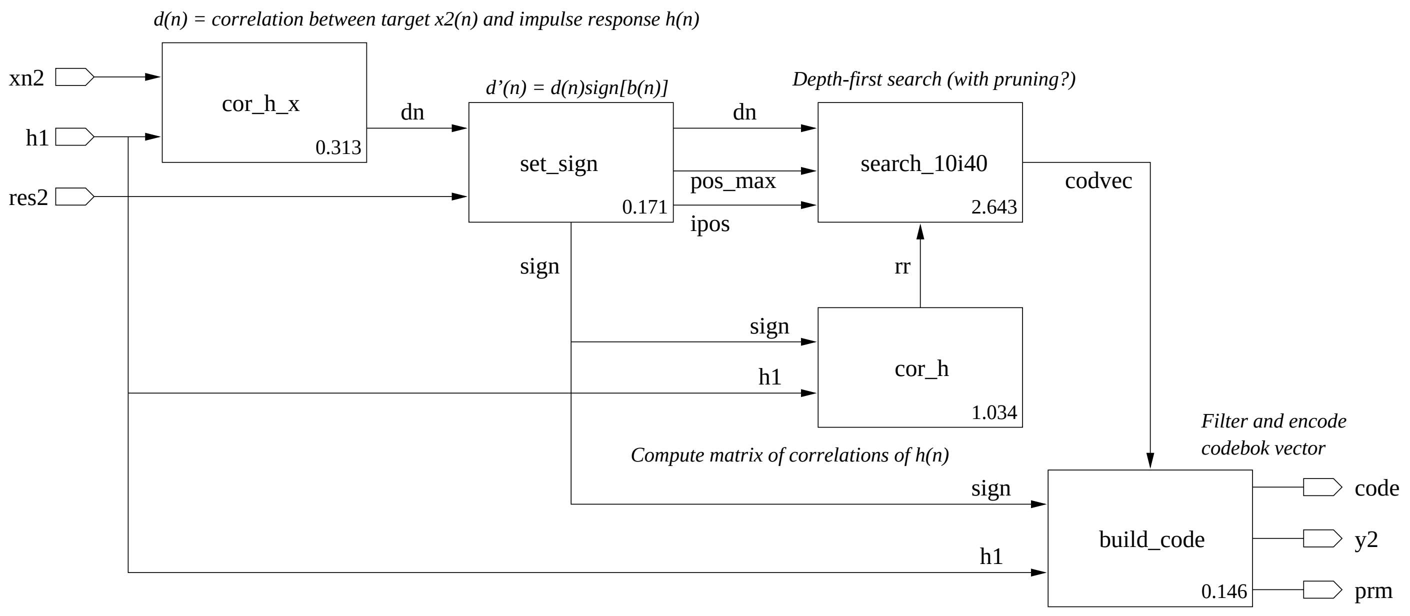


A.1.1.4 Algebraic codebook analysis and filter updates

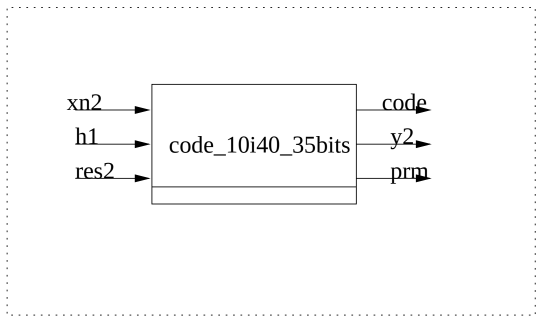


Algebraic (innovative) codebook search
exc[i]
res2
xn
y1
gain_pit
T0
h1
txdtx_ctrl
Target signal (speech), residual (excitation) minus pitch contribution
xn2:= xn - y1*gain_pit
res2:= res2 - exc[i]*gain_pit
h1:= h1 + h1(-T0)*gain_pit
Consider algebraic prefilter F(z) by including it in impulse response h(n)
xn2
res2
h1
Codebook search: return code vector c(n) and filtered code vector z(n)
3.7
code_10i40_35bits
4.33
prm
y2
code
Filter code vector c(n) through prefilter F(z)
Calculate codebook gain
3.7
G_code
0.05
code:= code +code(-T0)*gain_pit
T0
gain_pit
gain_code
code
exc[i]
Get final excitation signal u(n)
Gain Quantization
3.8
q_gain_code
0.07
exc[j]:= gain_pit*exc[j] +gain_code*code[j]
exc[i]
exc
prm
3.9
Syn_filt
0.04
Aq_t[i]
mem_syn
mem_syn
Synthesize speech
y2
gain_code
synth
speech[i]
mem_err:= speech[i]-synth
gain_pit
y1
mem_w0:= xn-gain_code*y2 -gain_pit*y1
xn
Update filter memories
mem_err
mem_w0
i=[0,40,80,120]
speech[80..160]
Shift buffers to the left by 160 samples
speech[-80..0]
exc[6..160]
exc[-154..0]

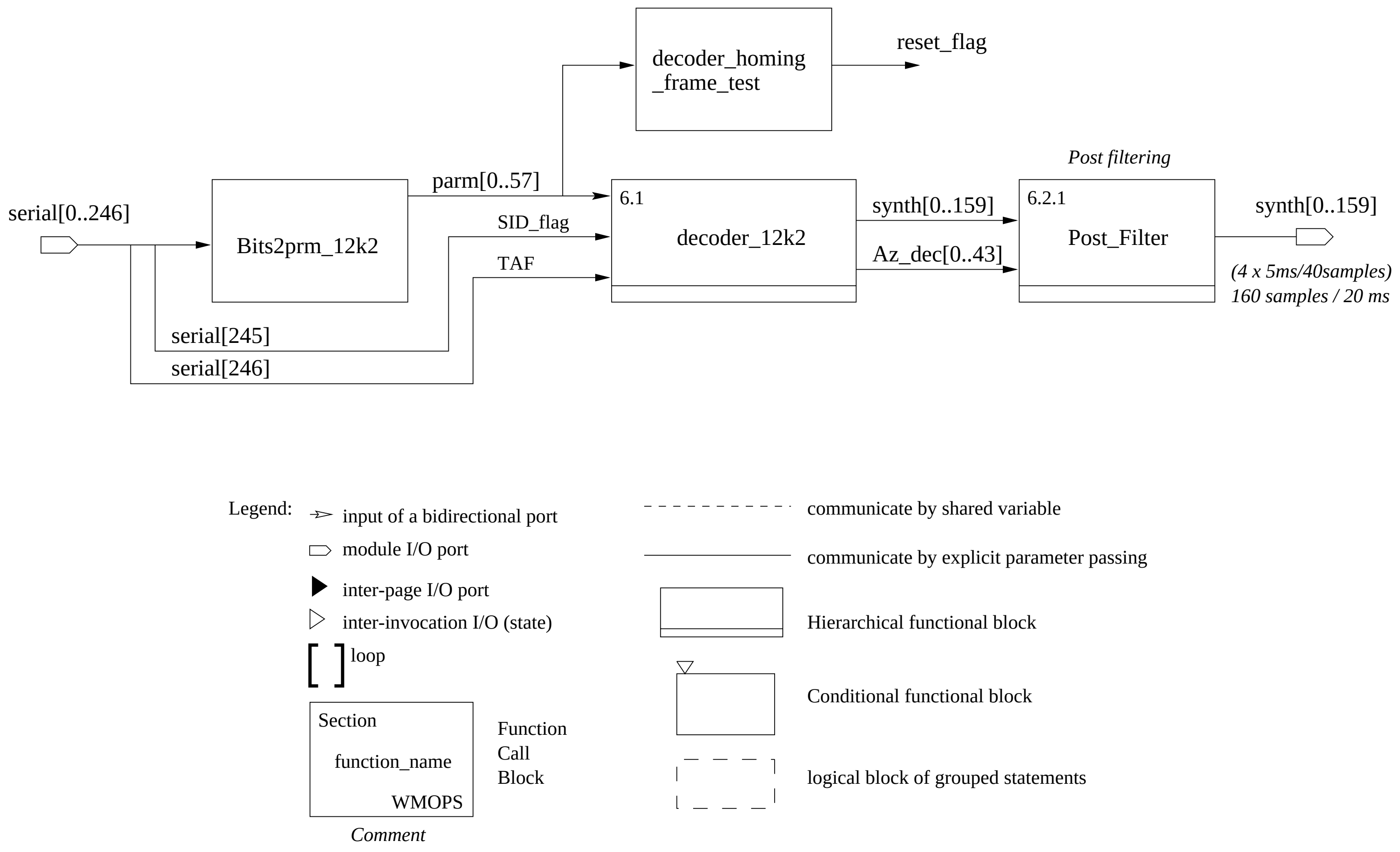
A.1.1.5 Codebook search: `code_10i40_35bits`



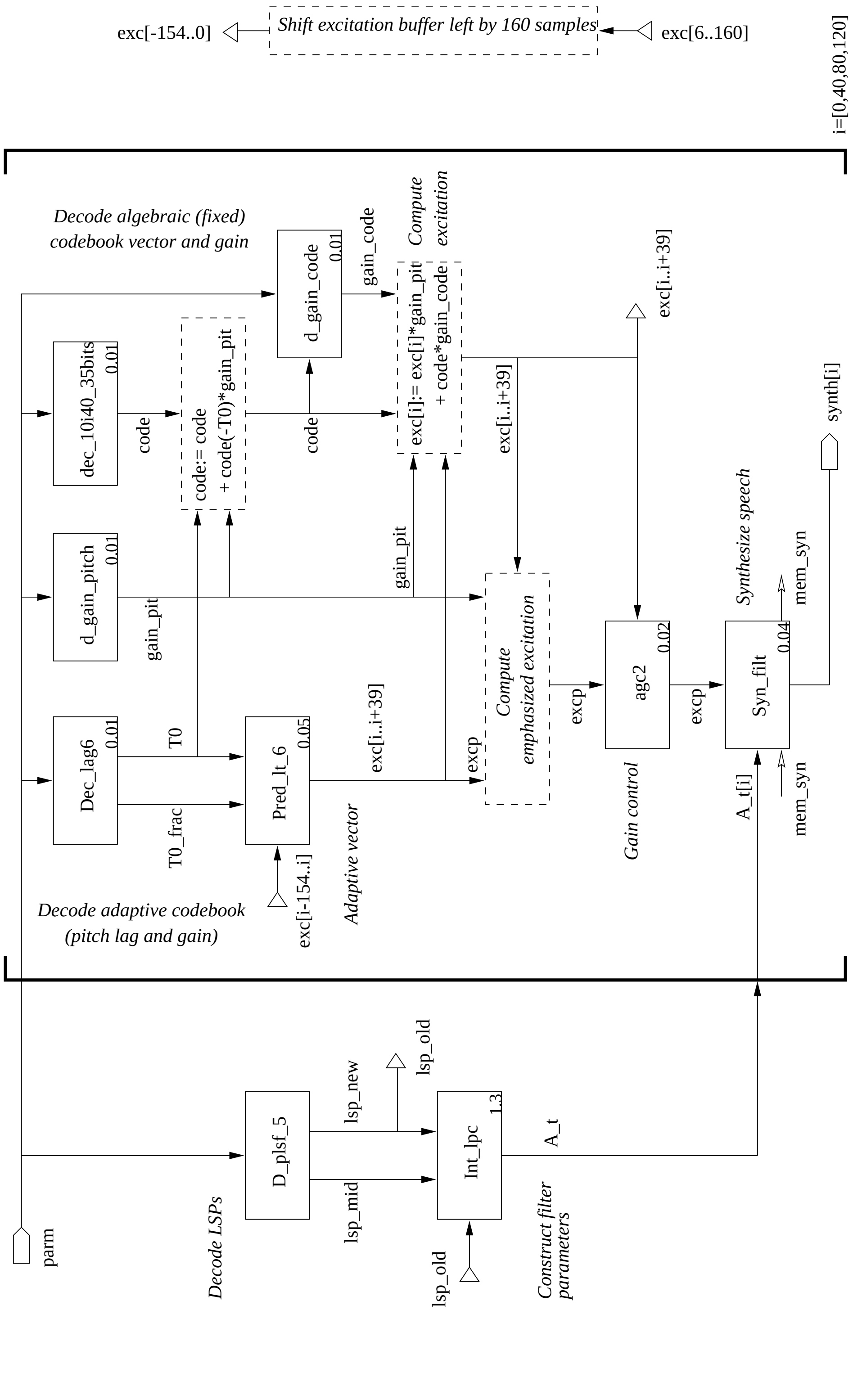
d(n) = correlation between target x2(n) and impulse response h(n)
xn2
h1
res2
cor_h_x
0.313
dn
d'(n) = d(n)sign[b(n)]
set_sign
0.171
dn
pos_max
ipos
Depth-first search (with pruning?)
search_10i40
2.643
codvec
sign
rr
sign
h1
cor_h
1.034
Compute matrix of correlations of h(n)
Filter and encode codebok vector
sign
h1
build_code
0.146
code
y2
prm



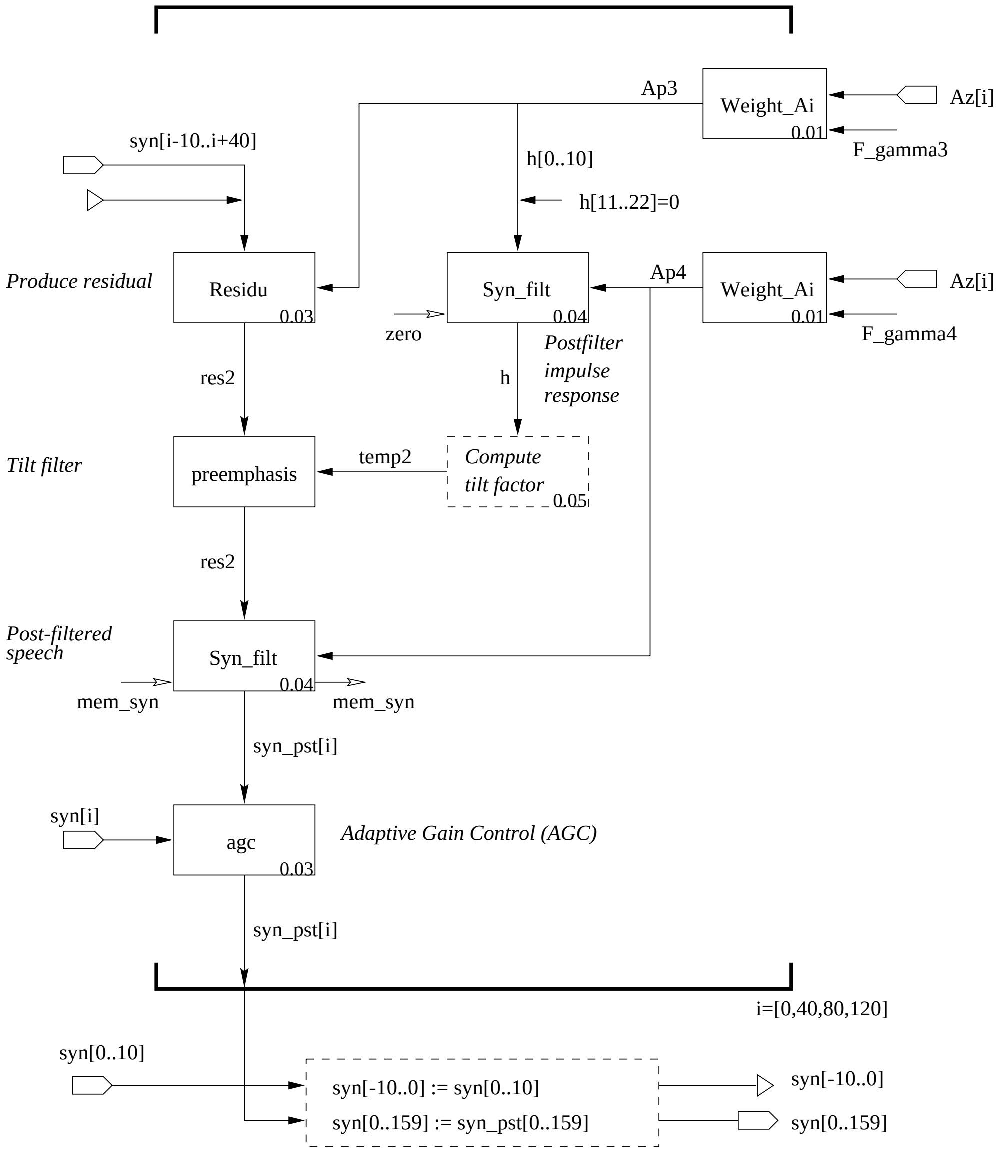
DECODER BLOCK DIAGRAM

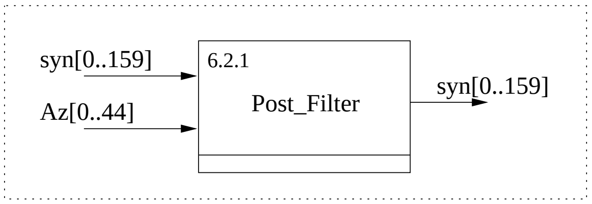


A.2.1 Decoding: `decoder_12k2`



A.2.2 Post-processing: `Post_Filter`





B Vocoder Specification

This appendix describes the overall SpecC [1, 2, 4, 6] specification of the GSM Enhanced Full Rate Vocoder [9]. The SpecC blocks are directly derived from the blocks of C reference implementation (see Appendix A).

B.1 General (shared) behaviors

`Syn_filt` Implement the LP filter (synthesis filter) $1/A(z)$.

Given an input sequence $x(n)$, $n = 0 \ldots 39$, the LP filter coefficients $a(k)$, $k = 1 \ldots 10$ and the filter memory $y(m-10) = mem(m)$, $m = 0 \ldots 9$, the output sequence is

$$y(n) = x(n) - \sum_{k=1}^{10} a(k)y(n-k), \quad n = 0 \ldots 39.$$

In addition, the filter memory can be updated, too:

$$mem(m) = y(30+m), \quad m = 0 \ldots 9.$$

`Residu` Implement the LP inverse filter $A(z)$ (to get the residual).

Given an input sequence $x(n)$, $n = -10 \ldots 39$ and the LP filter coefficients $a(k)$, $k = 1 \ldots 10$ the output sequence is

$$y(n) = x(n) + \sum_{k=1}^{10} a(k)x(n-k), \quad n = 0 \ldots 39.$$

B.2 Coder

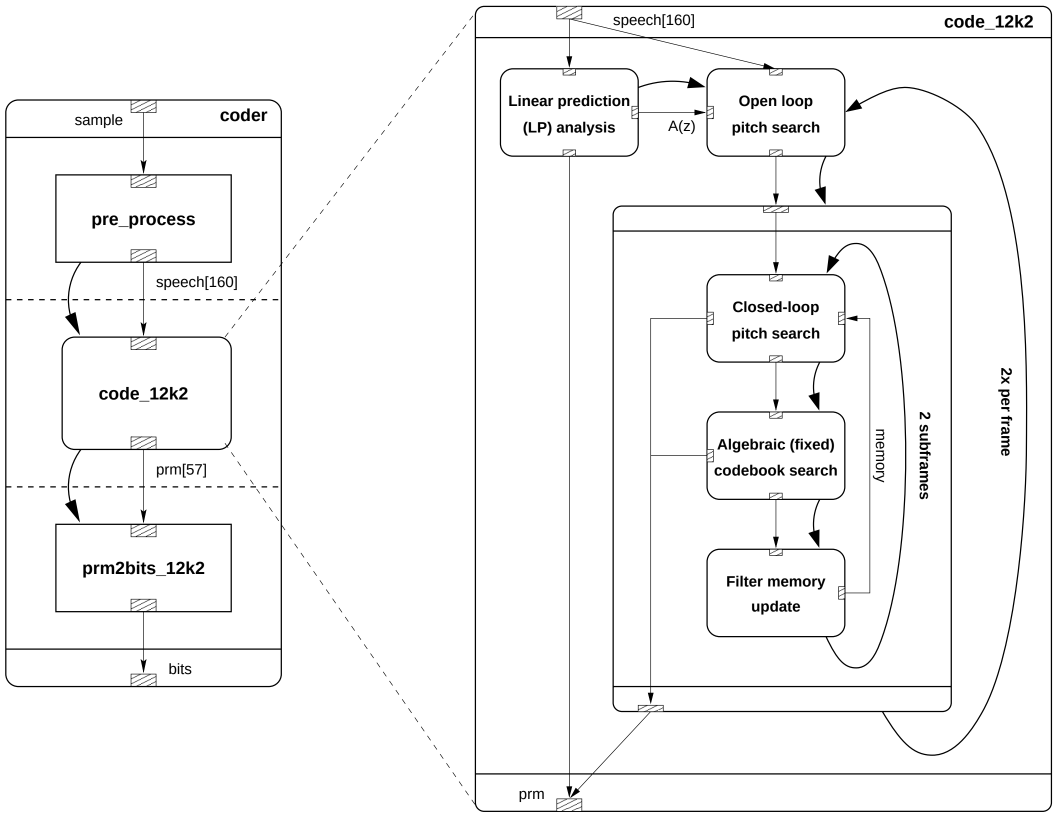


Figure 58: Coder

Encoding is based on finding the parameters for the speech synthesis model at the receiving side which will then be transmitted to the decoder over the medium.

The speech synthesis model is code-excited linear predictive (CELP) model: to synthesize speech in the decoder a 10th order linear predictive (LP) synthesis filter $H(z) = 1/A(z)$ (responsible for the short term effects) is excited with a signal constructed by adding two vectors from two codebooks:

- The so-called adaptive codebook is based on a pitch synthesis filter which is responsible for covering long term effect. The output of the pitch filter is simply a previous excitation signal delayed by a certain amount (lag) and scaled with a certain gain. Since the delay/lag of the pitch filter can be fractional the delayed excitation has to be interpolated (using a FIR filter) between the two adjacent (delayed by an integer lag) excitation values.

- The so-called fixed or algebraic codebook covers any remaining pulse excitation sequence left after removing the short-term and long- term contributions. The fixed codebook contains 5 tracks with 8 possible positions each. For each track two positions are chosen (10 pulses all together) and transmitted.

In general, the parameters for the two codebooks are chosen such that the error between the synthesized speech (at the output of the LP synthesis filter!) and the original speech is minimized. However, for the codebook searches the original speech is weighted (by a weighting filter $W(z)$) in order to account for the special properties of human acoustic perception.

B.2.1 Preprocessing: `pre_process`

The pre-processing consists of high-pass filtering and signal down-scaling (dividing the signal by two to reduce the possibility of overflows) of the input speech samples. The high-pass filter $H_{h1}(z)$ is as given in the specification.

Given the input signal $x(n)$ and the filter coefficients a and b the output sequence $y(n)$ of the pre-processing step is

$$y(n) = \frac{b_0}{2}x(n) + \frac{b_1}{2}x(n-1) + \frac{b_2}{2}x(n-2) + a_1 y(n-1) + a_2 y(n-2).$$

B.2.2 Linear prediction analysis and quantization

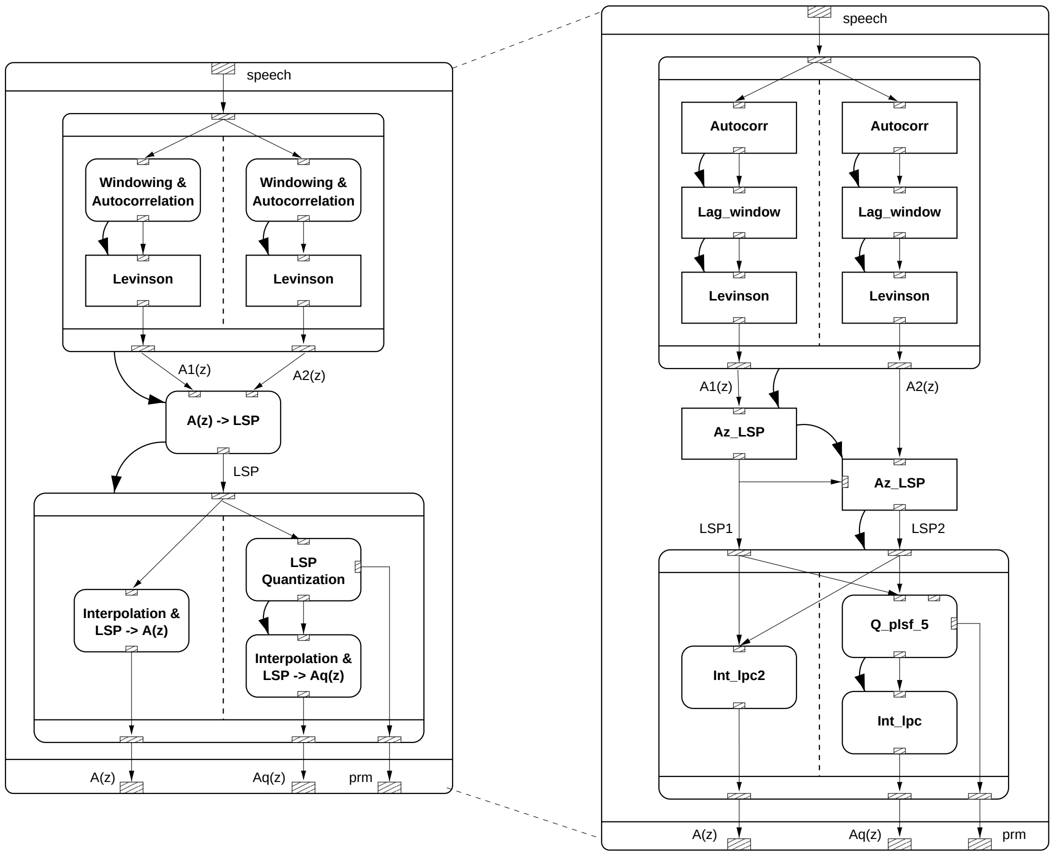


Figure 59: LP Analysis

Determine the parameters (coefficients) of the LP (short term) synthesis filter twice per frame, encode them for transmission and recompute the coefficients for every subframe (e.g. to include the effects due to encoding losses):

`Autocorr` Windowing and autocorrelation computation.

Two fixed windows (see specification) $w(n)$ are applied to 80 samples from the past speech frame plus the 160 samples from the current frame to get the windowed speech

$$s'(n) = s(n)w(n), \quad n = 0 \ldots 239.$$

The autocorrelation $r(k)$ of the windowed speech is then computed as

$$r(i) = \sum_{n=i}^{239} s'(n)s'(n-i), \quad i = 0 \ldots 10.$$

In addition, the C reference implementation normalizes $r(i)$ and checks for overflows.

`Lag_window` Lag windowing of the autocorrelations.

$$r(k) = r(k)w_{lag}(k), \quad k = 1\ldots10$$

where the fixed window $w_{lag}(k)$ is as given in the specification.

`Levinson` Levinson-Durbin algorithm to recursively compute the LP (linear prediction) filter coefficients $a(k)$, $k = 1\ldots10$.

`Az_lsp` Convert LP filter coefficients to line spectral pairs (LSPs) $q(k)$, $k = 1\ldots10$.

`Q_plsf_5` Quantization of the two set of line spectral pairs (LSPs) per frame.

`Int_lpc`, `Int_lpc2` Interpolation of the quantized and unquantized LSPs for intermediate subframes and re-conversion of the LSPs to LP filter coefficients.

The two sets of LSPs, $q_2(k)$ and $q_4(k)$ previously computed are used directly for the 2nd and 4th subframes. For the 1st and 3th subframe two sets of LSPs are calculated by linearly interpolating the LSPs from adjacent subframes

$$\begin{aligned} q_1(k) &= 0.54q_{4,old}(k) + 0.5q_2(k), \\ q_3(k) &= 0.5q_2(k) + 0.54q_4(k), \qquad k = 1\ldots10. \end{aligned}$$

For the unquantized case of the 2nd and 4th subframes the LP filter coefficients are already directly available. For all other cases (quantized LSPs and unquantized but interpolated LSPs for 1st and 3rd subframes):

`Lsp_Az` Convert the LSPs back to LP filter coefficients.

Once the LP filter parameters are found in the next steps the signals at the input of the LP filter, i.e. the two codebook conttributions have to be found.

B.2.3 Open-loop pitch analysis

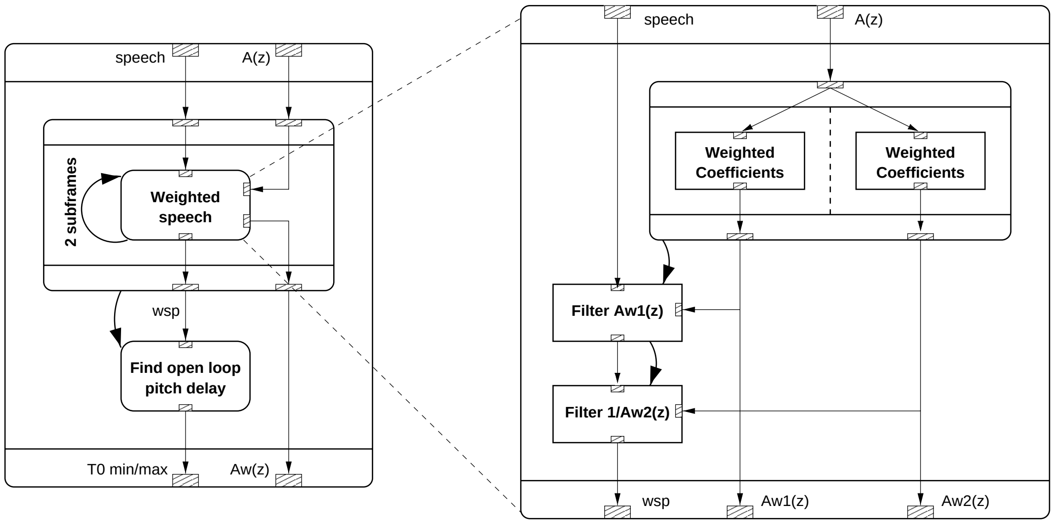


Figure 60: Open-loop pitch analysis

Open-loop pitch analysis determines delay/lag estimates for the (closed-loop) calculation of the pitch filter parameters in order to narrow the actual adaptive codebook search.

`Weight_Ai` Calculate the two sets of weighted filter coefficients for the implementation of the weighting filter:

$$a_i(k) = a(k)\gamma_i(k), \quad k = 1 \ldots 10, \quad i = 1, 2$$

where the spectral expansion factors $\gamma_i(k)$ are given as in the specification.

`Residu`, `Syn_filt`, **weighted coefficients** For each of the 4 subframes filter the speech signal through the weighting filter $W(z) = \frac{A(z/\gamma_1)}{A(z/\gamma_2)}$ to obtain the weighted speech signal.

`Pitch_ol` Perform open-loop pitch analysis based on the weighted speech twice per frame to obtain two pitch lag estimates $T_{op}(s)$, $s = 0, 1$.

Open-loop pitch analysis is done by finding maximal correlations in the weighted speech signal.

Finally, using the two open-loop pitch lag (pitch delay) estimates the closed-loop pitch analysis search ranges $[T0_{min}, T0_{max}]$ for the 1st and 3rd subframes are preset to

$$T0_{min}(s) = T_{op}(s) - 3, \quad T0_{min}(s) \geq 18,$$
$$T0_{max}(s) = T_{op}(s) + 3, \quad T0_{max}(s) \leq 143, \quad s = 0, 1.$$

B.2.4 Closed loop pitch analysis

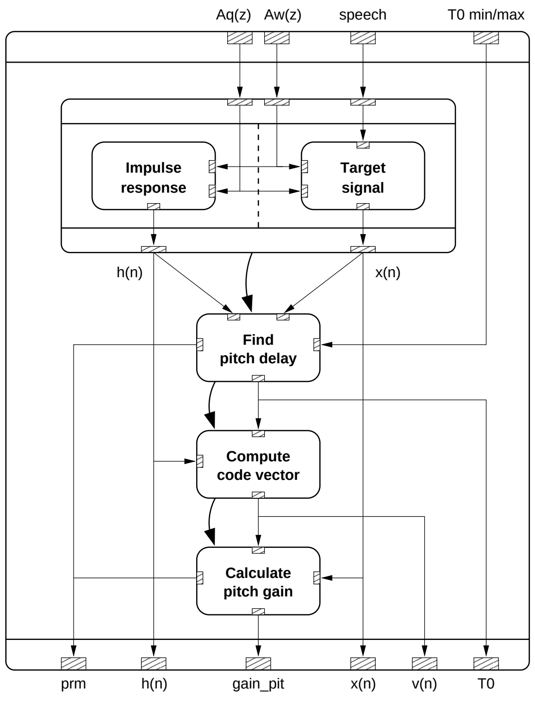


Figure 61: Closed loop pitch search.

Based on the open-loop lag estimates search for the pitch filter parameters in the given intervals.

B.2.4.1 Impulse response computation

For each subframe the impulse response $h(n)$ of the weighted synthesis filter $H(z)W(z) = \frac{A(z/\gamma_1)}{\hat{A}(z)A(z/\gamma_2)}$ is computed by

`Syn_filt`, **quantized coefficients** $\hat{a}(k)$, **zero-filled memory** Filtering of the weighted filter coefficients $a_1(k) = a(k)\gamma_1(k)$ through the quantized synthesis filter $1/\hat{A}(z)$.

`Syn_filt`, **weighted coefficients** $a_2(k) = a(k)\gamma_2(k)$, **zero memory** Filtering of the intermediate result through the weighted filter $1/A(z/\gamma_2)$.

The impulse response is needed in the codebook searches to model the effects a certain excitation vector will have on the error at the *output* of the LP synthesis filter.

B.2.4.2 Residual computation For each subframe the residual signal $res_{LP}(n)$ is calculated by

`Residu`, **quantized coefficients** $\hat{a}(k)$ Filter the speech signal $s(n)$ through the inverse quantized synthesis filter $\hat{A}(z)$.

The residual is basically the signal needed at the input of the LP filter in order to get the original speech back at its output.

B.2.4.3 Target signal computation For each subframe calculate the target signal $x(n)$ (weighted speech minus effect of weighted synthesis filter $H(z)W(z)$) for the adaptive codebook search:

`Syn_filt`, **quantized coefficients** $\hat{a}(k)$ Filter the residual $res_{LP}(n)$ through the quantized synthesis filter $1/\hat{A}(z)$.

`Residu`, `Syn_filt`, **weighted coefficients** Filter the result through the weighting filter $W(z) = \frac{A(z/\gamma_1)}{A(z/\gamma_2)}$.

The filters for the target signal computation use a special memory which is updated separately using additional filters (see B.2.6).

B.2.4.4 Adaptive codebook search For each subframe compute the adaptive codebook parameters (delay/lag and gain of the pitch filter).

First, the pitch delay/lag is found and encoded for transmission:

`Pitch_fr6` Closed-loop search to find the best pitch delay/lag $T0$ (integer and fractional parts) such that the error between the target signal $x(n)$ (original speech) and the past LP filtered excitation (past excitation convolved with impulse response $h(n)$) at delay $T0$ is minimized.

First, the integer part of the lag is found. Then, the fractional part (with resolution 1/6) is found by interpolating the error between the adjacent integer boundaries.

Since the past excitation for delays inside the current subframe is not known yet the excitation buffer for the current subframe is initialized with the residual $res_{LP}(n)$.

`Enc_lag6` Encoding of the pitch lag $T0$ into 9 bits (1st and 3rd subframes) or 6 bits (2nd and 4th subframes). If necessary, adjusts the fractional part of $T0$ for the following gain calculations.

For the 1st and 3rd subframes the lag is encoded with a resolution of 1/6 in the range $[17\frac{3}{6}, 94\frac{3}{6}]$ and integers only in the range $[95, 143]$. In the latter case the fractional part of the pitch lag is set to zero.

For the 2nd and 4th subframes the lag is encoded with a resolution of 1/6 in an interval around the lag in the previous (1st or 3rd) frame.

In addition, in the 1st and 3rd subframes the search ranges $[T0_{min}, T0_{max}]$ are updated for upcoming searches in the 2nd and 4th subframes, respectively.

$$\begin{aligned} T0_{min}(s) &= [T0] - 5, \quad T0_{min}(s) \geq 18, \\ T0_{max}(s) &= [T0] + 4, \quad T0_{max}(s) \leq 143, \quad s = 0, 1. \end{aligned}$$

In the 2nd and 4th subframe the search ranges are based on the integer pitch lag parts $[T0]$ found in the 1st and 3rd subframe.

Then, the actual adaptive codevector is calculated in order to compute the adaptive codebook gain:

Pred_lt_6 Compute the adaptive codebook vector $v(n)$ by interpolating the past excitation at $T0$ using two FIR filters.

Convolve Filter the adaptive codebook vector $v(n)$ through the weighted synthesis filter $H(z)W(z)$ by convolving it with the impulse response $h(n)$: $y(n) = v(n) * h(n)$, i.e.

$$y(n) = \sum_{i=0}^{n} x(i)h(n-i), \quad n = 0 \ldots 39.$$

G_pitch Calculate the adaptive codebook gain

$$g_p = \frac{\sum_{n=0}^{39} x(n)y(n)}{\sqrt{\sum_{n=0}^{39} y(n)y(n)}}, \quad 0 \leq g_p \leq 1.2$$

q_gain_pitch Quantize the adaptive codebook gain for transmission.

B.2.5 Algebraic (fixed) codebook analysis

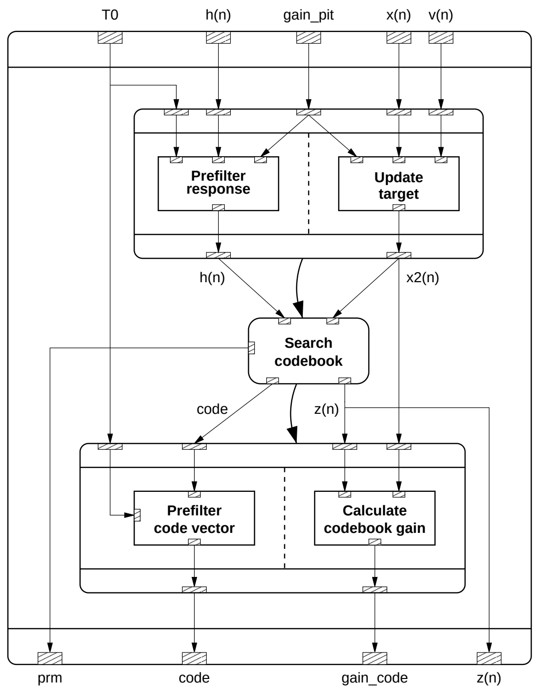


Figure 62: Algebraic (fixed) codebook search

B.2.5.1 Update target signal and residual The remaining target signal $x_2(n)$ and residual $res_{LTP}(n)$ (after removing long-term prediction contributions) are computed for the fixed codebook search by subtracting the adaptive codebook effects:

$$x_2(n) = x(n) - \hat{g}_p y(n), \quad res_{LTP}(n) = res_{LP}(n) - v(n), \quad n = 0 \ldots 39.$$

where $x(n)$ and $res_{LP}(n)$ are the target signal and the residual for the adaptive codebook search (see B.2.4.3) and B.2.4.2), $\hat{g}_p$ is the quantized adaptive codebook gain and $v(n)$ is the adaptive codebook vector.

B.2.5.2 Update impulse response The impulse response $h(n)$ is updated for the fixed codebook search by including a prefilter $F_E(z) = \frac{1}{1-\hat{g}_p z^{-[T0]}}$ (where $\hat{g}_p$, $\hat{g}_p \leq 1.0$ is the quantized pitch gain and $[T0]$ is the integer part of the pitch lag) which enhances spectral components to improve quality:

$$h_E(n) = h(n) + \frac{\hat{g}_p}{8} h(n - [T0]), \quad n = [T0] \ldots 39.$$

B.2.5.3 Codebook search: `code_10i40_35bits`

`cor_h_x` Compute the correlation between the target $x_2(n)$ and the impulse response $h_E(n)$:

$$d(n) = \sum_{i=n}^{39} x_2(n) h_E(i-n), \quad n = 0 \ldots 39.$$

The vector $d(n)$ corresponds to the backward filtered target signal.

The C reference implementation adds some normalization of $d(n)$ such that the sum of the maxima of $d(n)$ for each of the 5 tracks will not saturate.

`set_sign` Calculate the pulse sign information

$$sign(n) = sign[en(n)], \quad n = 0 \ldots 39$$

with $en(n)$ being the sum of the normalized long term residual and the normalized correlation vector $d(n)$:

$$en(n) = \frac{res_{LTP}(n)}{\sqrt{\sum_{i=0}^{39} res_{LTP}^2(n)}} + \frac{d(n)}{\sqrt{\sum_{i=0}^{39} d^2(n)}}, \quad n = 0 \ldots 39.$$

The sign information is then included into $d(n)$:

$$d(n) = d(n) sign(n), \quad n = 0 \ldots 39.$$

Also, the position with maximum correlation in each of the 5 tracks is computed:

$$pos_{max}(t) = p \text{ s.t. } en(p)sign(p) = \max_{j=t,t+5,\ldots,39} en(j)sign(j), \quad t = 0 \ldots 4.$$

Finally, the starting positions of each pulse are calculated:

$$ipos(0) = ipos(5) = t \text{ s.t. } pos_{max}(t) = \max_{j=0\ldots4} pos_{max}(j),$$
$$ipos(i) = ipos(i+5) = (ipos(0) + i) \bmod 5, \quad i = 1 \ldots 4.$$

`cor_h` Compute the matrix of correlations of the impulse response $h_E(n)$ and include the sign information in it:

$$rr(i,j) = \left(\sum_{n=i}^{39} h_E(n-i) h_E(n-j) \right) sign(i)sign(j), \quad i \geq j, \quad i,j = 0 \ldots 39.$$

`search_10i40` Search the algebraic (fixed) codebook to find the optimal pulse positions m_j:

```
/* Fix position of first pulse to global maximum position */
$i_0 = pos_{max}(ipos(0))$;
/* Four iterations over local maxima in other tracks */
for each track $t = 1 \ldots 4$ do
    $i_1 = pos_{max}(ipos(1))$; /* max. pos. in track */
```

```
        /* Successively add pulses in pairs */
        for each pair $(a, b) = (2, 3), (4, 5), (5, 7), (8, 9)$ do
            /* Search pair positions to maximize mean square error $A$ */
            for $i_a = ipos(a) \dots 39$, step 5 do
                for $i_b = ipos(b) \dots 39$, step 5 do
                    $C = \sum_{j=0}^{b} d(i_j)$;
                    $E_D = \frac{1}{16} \sum_{j=0}^{b} rr(i_j, i_j) + \frac{1}{8} \sum_{j=0}^{b} \sum_{k=0}^{j-1} rr(i_k, i_j)$;
                    if $\frac{C^2}{E_D} > \frac{C_{max}^2}{E_{D,max}}$ then
                        $C_{max} = C$; $E_{D,max} = E_D$;
                        $ia_{max} = i_a$; $ib_{max} = i_b$;
                    end if
                end for
            end for
            /* Set pair positions to maximizer */
            $i_a = ia_{max}$; $i_b = ib_{max}$;
        end for
        /* All pulse positions assigned, is it global maximum? */
        if $\frac{C_{max}^2}{E_{D,max}} > A_{max}$ then
            $A_{max} = \frac{C_{max}^2}{E_{D,max}}$;
            /* Remember pulse positions */
            for $j = 0 \dots 9$ do $m_j = i_j$ end for
        end if
        /* Cyclically shift starting positions for next iteration */
        $ipos(1 \dots 9) = ipos(2 \dots 9, 1)$;
    end for
```

`build_code` Given the positions and signs of the 10 pulses build the fixed codebook vector $c(n)$ and encode it for transmission.

In addition, the fixed codebook vector is filtered by convolving with the impulse response $h_E(n)$:

$$z(n) = \sum_{i=0}^{n} c(i) h_E(n-i), \quad n = 0 \dots 39.$$

B.2.5.4 Codebook gain

`G_code` Calculate the fixed codebook gain

$$g_c = \frac{\sum_{n=0}^{39} x_2(n) z(n)}{\sum_{n=0}^{39} z(n) z(n)}$$

B.2.5.5 Quantization of fixed codebook gain In a preprocessing step, the fixed codebook vector $c(n)$ is filtered through the prefilter $F_E(z)$:

$$c_E(n) = c(n) + \frac{\hat{g}_p}{8} c(n - [T0]), \quad n = [T0] \dots 39.$$

(see B.2.5.2) followed by:

`q_gain_code` Quantize the fixed codebook gain for transmission.

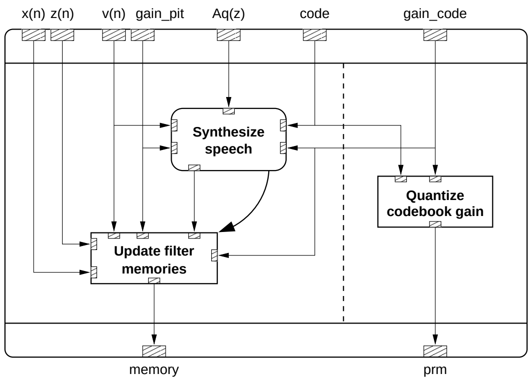


Figure 63: Filter memory update

B.2.6 Filter memory updates

In this final step, the memories of the synthesis and weighting filters for the calculation of the pitch analysis target signals (B.2.4.3) are updated for the next subframe.

The excitation signal $u(n)$ in the present subframe is calculated:

$$u(n) = \hat{g}_p v(n) + \hat{g}_c c_E(n), \quad n = 0 \ldots 39$$

where $\hat{g}_p$ and $\hat{g}_c$ are the quantized gains, $v(n)$ is the adaptive codebook vector and $c_E(n)$ is the filtered fixed codebook vector. The excitation signal is also copied to the past excitation buffer for the pitch synthesis filter.

`Syn_filt`, quantized coefficients $\hat{a}(k)$ Synthesized the speech $\hat{s}(n)$ locally by filtering the excitation signal $u(n)$ through the LP filter $1/\hat{A}(z)$.

The memories of the synthesis and weighting filters are then updated to

$$e(n) = s(n) - \hat{s}(n), \quad n = 30 \ldots 39$$

and

$$e_w(n) = x(n) - \hat{g}_p y(n) - \hat{g}_c z(n), \quad n = 30 \ldots 39,$$

respectively.

B.2.7 Serialization: `Prm2bits_12k2`

Conversion of the set of parameters obtained by the encoder for a complete frame into a serial stream of 244 bits corresponding to a transfer rate of 12.2 kbit/s.

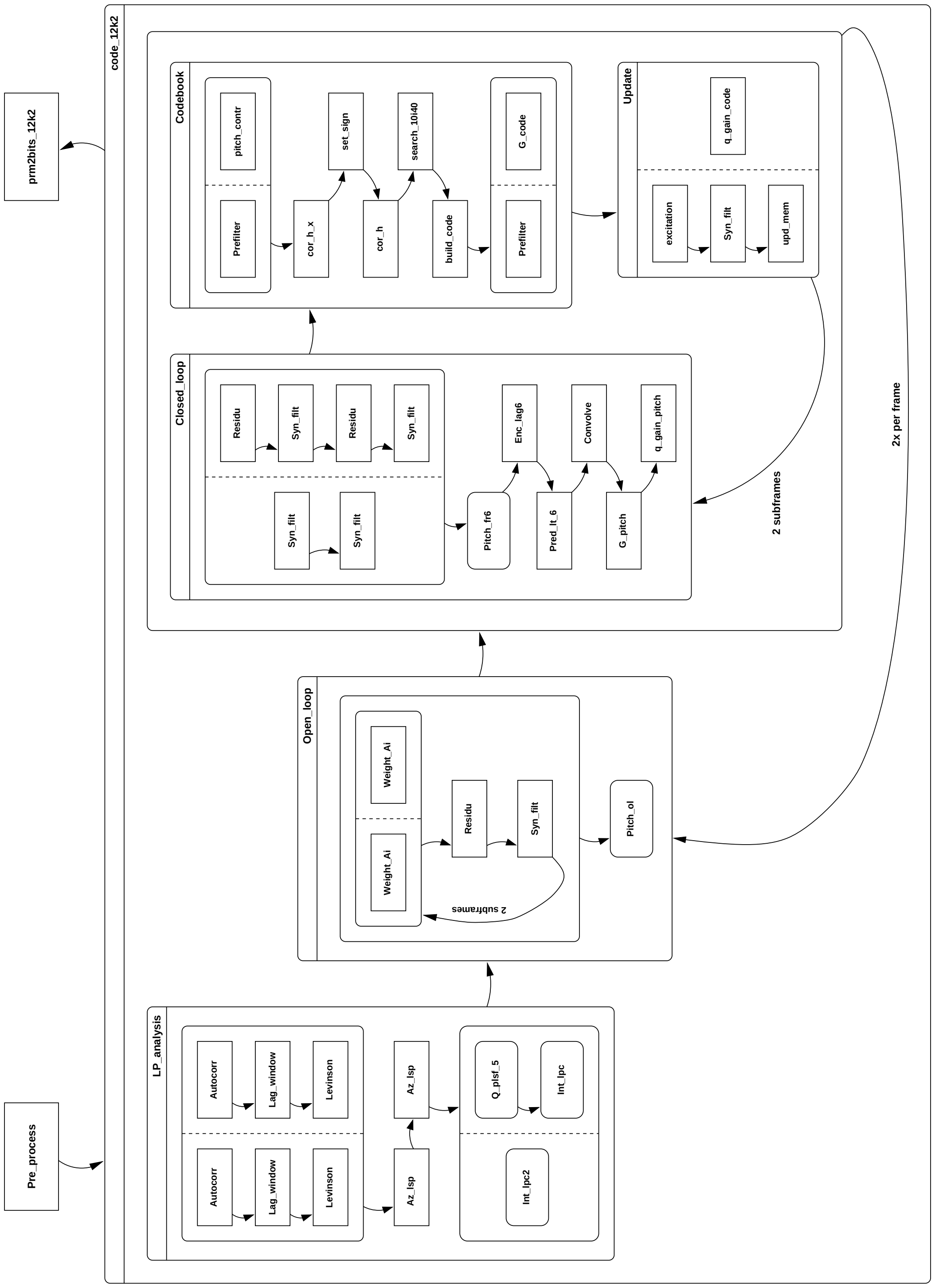


Figure 64: Coder block diagram.

B.3 Decoder

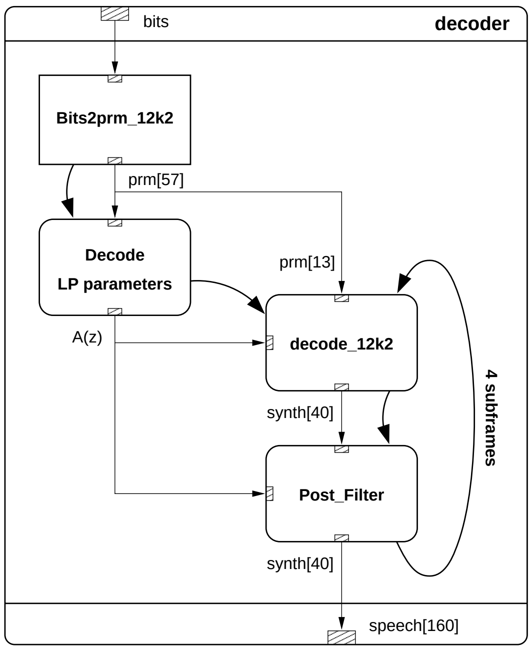


Figure 65: Decoder

Decoding is basically the reverse process of encoding in the sense that simply the synthesis model described in B.2 is implemented. Therefore, the steps are very similar to the routines described in the encoding part and the reader is referred to the first part for details.

B.3.1 Parameter extraction: `Bits2prm_12k2`

Extract the decoder parameter set from the serial stream of 244 bits for a complete frame.

B.3.2 Decoding of LP filter parameters

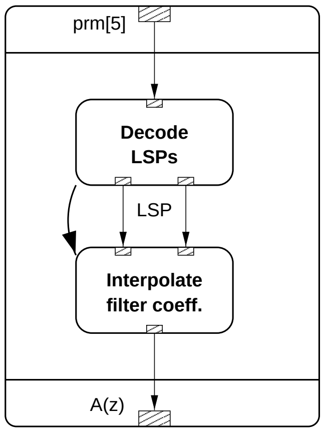


Figure 66: LSP decoding

For each complete frame:

`D_plsf_5` The received LSP indices are used to reconstruct the two LSPs for the 2nd and 4th subframes.

`Int_lpc` Interpolation of the LSPs for the 1st and 3rd subframes and conversion of the LSPs to LP filter coefficients $a(k)$, $k = 1 \ldots 10$ for all 4 subframes.

B.3.3 Decoding subframe and synthesizing speech

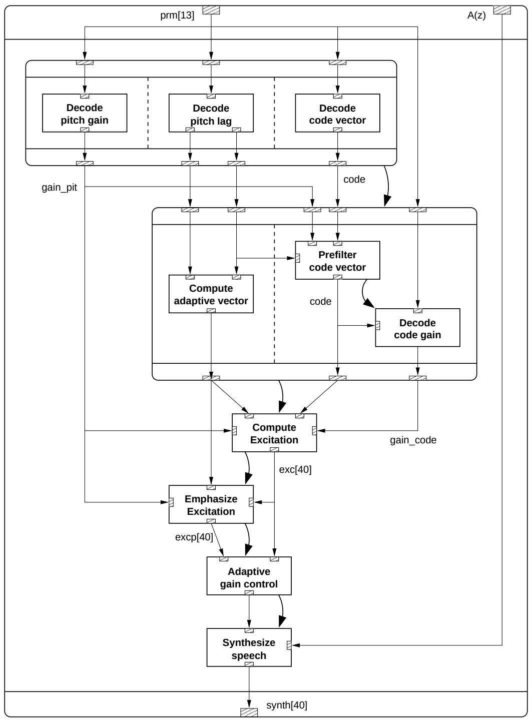


Figure 67: Subframe decoding

B.3.3.1 Decoding of the adaptive codebook vector For each subframe, the received pitch lag is decoded and used to construct the adaptive codebook vector $v(n)$ from the past excitation buffer.

`Dec_lag6` Decode the received pitch index to construct the integer and fractional parts of the pitch lag $T0$.

`Pred_lt_6` Compute the adaptive codebook vector $v(n)$ by interpolating the past excitation at $T0$.

B.3.3.2 Decoding of the adaptive codebook gain For each subframe:

`d_gain_pitch` Decode the received gain index to construct the adaptive codebook gain $\hat{g}_p$.

B.3.3.3 Decoding of the algebraic codebook vector For each subframe:

`dec_10i40_35bits` The received fixed codebook index is used to reconstruct the signs and positions of the 10 pulses which then give the fixed codebook vector $c(n)$.

After decoding, the prefilter $F_E(z)$ (see B.2.5.2) is applied to the fixed codebook vector:

$$c_E(n) = c(n) + \frac{\hat{g}_p}{8} c(n - [T0]), \quad n = [T0] \ldots 39$$

where $\hat{g}_p$ and $[T0]$ are the previously decoded pitch gain and integer part of the pitch lag.

B.3.3.4 Decoding of the algebraic codebook gain

For each subframe:

`d_gain_code` Given the codebook vector $c_E(n)$ and the received gain index the fixed codebook gain $\hat{g}_c$ is calculated.

B.3.3.5 Computing the reconstructed speech

In each subframe, the basic excitation at the input of the LP synthesis filter is

$$u(n) = \hat{g}_p v(n) + \hat{g}_c c_E(n), \quad n = 0 \ldots 39$$

given the previously decoded codebook vectors and gains.

If $\hat{g}_p > 0.5$ then the excitation is modified to emphasize the contribution of the adaptive codebook vector:

$$\hat{u}(n) = u(n) + \frac{\hat{g}_p}{16} \hat{g}_p v(n), \quad n = 0 \ldots 39$$

is calculated and the excitation $u(n)$ is updated by

`agc2` Adaptive gain control to compensate for the difference between $u(n)$ and $\hat{u}(n)$.

The gain scaling factor is

$$\eta = \sqrt{\frac{\sum_{n=0}^{39} u^2(n)}{\sum_{n=0}^{39} \hat{u}^2(n)}}$$

and the final excitation signal is then calculated to

$$u(n) = \eta \hat{u}(n), \quad n = 0 \ldots 39.$$

In other cases ($\hat{g}_p \leq 0.5$) the excitation signal $u(n)$ is not modified.

Finally, the speech is synthesized by passing the excitation through the synthesis filter $1/\hat{A}(z)$:

`Syn_Filt`, coefficients $a(k)$, $k = 1 \ldots 10$ Filter the excitation signal $u(n)$ through the LP synthesis filter to get the reconstructed speech signal $\hat{s}(n)$.

B.3.4 Post-filtering: `Post_Filter`

In each of the 4 subframes a post-filtering of the synthesized speech is performed.

`Weight_Ai` Calculate the weighted filter coefficients

$$\hat{a}_n(k) = \hat{a}(k)\gamma_n(k),$$
$$\hat{a}_d(k) = \hat{a}(k)\gamma_d(k), \quad k = 1 \ldots 10.$$

for the formant postfilter $H_f(z) = /frac\hat{A}(z/\gamma_n)\hat{A}(z/\gamma_d)$.

B.3.4.1 Produce residual signal

`Residu`, coefficients $\hat{a}_n$ Filter the speech $\hat{s}(n)$ through the LP inverse filter $\hat{A}(z/\gamma_n)$ to get the residual $\hat{r}(n)$.

B.3.4.2 Tilt compensation filter

First, calculate the truncated impulse response $h_f(n)$ of the formant postfilter:

`Syn_filt`, coefficients $\hat{a}_d$**, zero memory** Filter the coefficients $\hat{a}_n(k)$ through the LP filter $1/\hat{A}(z/\gamma_d)$.

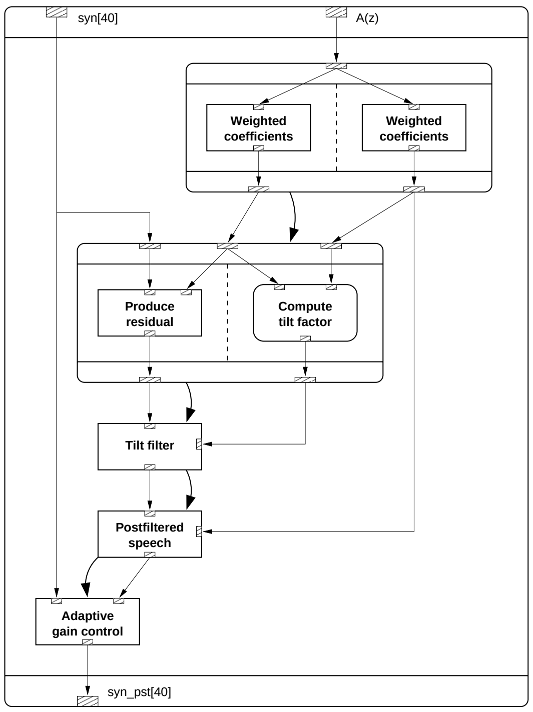


Figure 68: Post filtering

Next, the correlations of the impulse response are calculated:

$$r_h(i) = \sum_{j=0}^{21-i} h_f(j)h_f(j+i), \quad i = 0, 1.$$

The tilt factor is then given by

$$\mu = \begin{cases} 0.8\frac{r_h(1)}{r_h(0)}, & r_h(1) > 0 \\ 0 & \text{otherwise} \end{cases}$$

Finally, the residual is passed through the tilt compensation filter $H_t(z) = 1 - \mu z^{-1}$:

`Preemphasis` Filter the residual $\hat{r}(n)$ through the tilt compensation filter:

$$\hat{r}(n) = \hat{r}(n) - \mu\hat{r}(n-1), \quad n = 39 \ldots 0.$$

B.3.4.3 Post-filtered speech

`Syn_filt`, **coefficients** $\hat{a}_d$ Filter the compensated residual $\hat{r}(n)$ through the LP filter $1/\hat{A}(z/\gamma_d)$ to get the post-filtered speech signal $\hat{s}_f(n)$.

`agc` Given the synthesized speech signal $\hat{s}(n)$ and the post-filtered speech signal $\hat{s}_f(n)$ perform adaptive gain control to compensate for the gain difference.

Computes the final gain-scaled post-filtered speech signal $\hat{s}'(n)$.

B.3.5 Up-scaling

Reverse the signal down-scaling done in the encoder by multiplying the post-filtered speech signal $\hat{s}'(n)$ by 2. Finally, the signal is truncated to 13 bits according to the output format.

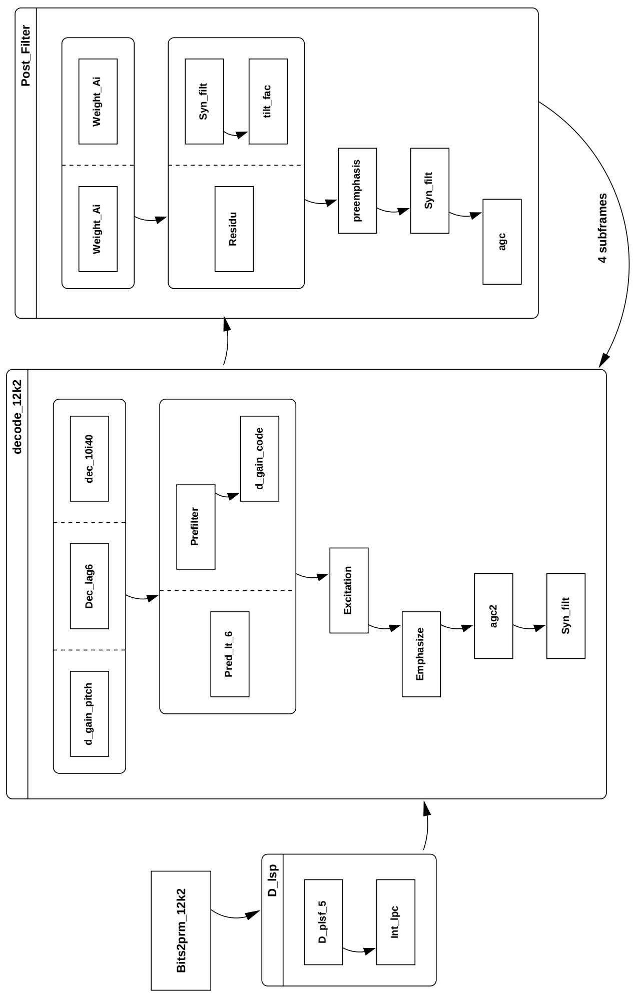


Figure 69: Decoder block diagram.

C Simulation Results

C.1 Software

The following tables list the cycle-accurate results obtained for simulating the coder and decoder on the ISS for the Motorola DSP5600. Tables are given for unoptimized, initial code and for the software after optimizations.

Routine		calls	cycles			instructions		
			max	total	avg.	max	total	avg
FAutocorr	18ae	6	104958	629748	104958	53938	323628	53938
FAz_lsp	16df	6	41258	246714	41119	24798	148315	24719
FChebps	1822	628	322	202216	322	200	125600	200
FCoder_12k2	11cc	3	2715917	8142878	2714292	1561164	4680860	1560286
FConvolve	19a0	24	29702	712848	29702	16181	388344	16181
FEnc_lag6	2852	12	99	928	77	52	503	41
FG_code	2bd2	12	937	11244	937	842	10104	842
FG_pitch	28ae	12	2826	33818	2818	1625	19449	1620
FGet_lsp_pol	36c4	36	1530	55080	1530	963	34668	963
FInt_lpc	30e4	3	14879	44629	14876	9471	28409	9469
FInt_lpc2	3140	3	7618	22840	7613	4866	14591	4863
FInterpol_6	37e6	70	270	18340	262	134	9180	131
FInv_sqrt	3788	252	135	30946	122	77	17668	70
FLag_max	25b0	18	86950	916735	50929	60707	641995	35666
FLag_window	2fd2	6	443	2658	443	318	1908	318
FLevinson	2cb7	6	18650	111788	18631	10373	62168	10361
FLsf_lsp	358b	6	404	2421	403	239	1434	239
FLsf_wt	34af	6	419	2490	415	293	1758	293
FLsp_Az	363b	18	3628	65195	3621	2300	41347	2297
FLsp_lsf	35bc	6	1960	11616	1936	945	5606	934
FNorm_Corr	271b	12	64233	733169	61097	39419	448217	37351
FPitch_fr6	2669	12	66638	757923	63160	40623	460564	38380
FPitch_ol	248b	6	158656	946007	157667	111212	664359	110726
FPre_Process	383c	3	15248	45744	15248	10268	30804	10268
FPred_lt_6	2c34	12	21995	263828	21985	11516	138150	11512
FPrm2bits_12k2	3013	3	9599	28592	9530	4731	14152	4717
FQ_plsf_5	318f	3	116512	348864	116288	75269	225521	75173
FReorder_lsf	361c	6	207	1239	206	126	756	126
FResidu	29d0	36	12528	451008	12528	8063	290268	8063
FSet_zero	34f7	9	1299	3119	346	1289	3029	336
FSyn_filt	306f	72	11124	792192	11002	5885	418248	5809
FVq_subvec	3331	12	22498	184014	15334	14900	122334	10194
FVq_subvec_s	33be	3	48033	143853	47951	30655	91860	30620
FWeight_Ai	2a20	48	132	6336	132	110	5280	110
Fbuild_code	2363	12	5730	68159	5679	3190	37976	3164
Fcode_10i40_35bits	19f7	12	246805	2957399	246449	132574	1589235	132436
Fcor_h	1c25	12	80454	965331	80444	44758	537033	44752
Fcor_h_x	1a74	12	31074	371601	30966	17096	204580	17048
Fencoder_reset	47a3	1	3553	3553	3553	3322	3322	3322
Fq_gain_code	2a7d	12	2471	28688	2390	1415	16661	1388
Fq_gain_pitch	2a40	12	584	6450	537	299	3402	283
Fq_p	1bf7	120	60	6360	53	31	3120	26
Fsearch_10i40	1d21	12	124153	1485073	123756	64517	772537	64378
Fset_sign	1b17	12	4633	55127	4593	2673	31697	2641

Table 8: Initial delays for coder behaviors.

Routine		calls	cycles			instructions		
			max	total	avg.	max	total	avg
FBin2int	195c	1425	160	119720	84	97	70635	49
FBits2prm_12k2	1920	25	6533	162495	6499	3676	91485	3659
FD_plsf_5	1b14	25	2888	72194	2887	1801	44968	1798
FDec_lag6	1516	100	140	12475	124	72	6453	64
FDecoder_12k2	116c	25	192021	4730059	189202	105919	2607675	104307
FGet_lsp_pol	1a06	200	1278	255600	1278	809	161800	809
FInit_Decoder_12k2	112c	1	775	775	775	719	719	719
FInit_Post_Filter	13b6	1	270	270	270	235	235	235
FInt_lpc	15b4	25	12295	306844	12273	8009	200022	8000
FInv_sqrt	1ab6	191	142	27098	141	76	14504	75
FLog2	1fe8	200	118	23600	118	64	12800	64
FLsf_lsp	206f	50	470	23475	469	246	12300	246
FLsp_Az	1993	100	2982	297244	2972	1934	193072	1930
FPost_Filter	13cc	25	153541	3838501	153540	85951	2148763	85950
FPow2	2035	100	84	8308	83	42	4154	41
FPred_lt_6	14a8	100	21645	2163702	21637	11321	1131815	11318
FReorder_lsf	1973	50	261	13025	260	144	7200	144
FResidu	1848	100	17044	1704400	17044	9300	930000	9300
FSet_zero	1806	104	635	7334	70	625	6192	59
FSyn_filt	189c	300	11078	2821100	9403	5882	1497600	4992
FWeight_Ai	1653	200	132	26400	132	110	22000	110
Fagc	1673	100	2585	258476	2584	1778	177788	1777
Fagc2	1747	91	2291	208481	2291	1607	146237	1607
Fd_gain_code	1d81	100	1242	123217	1232	742	73929	739
Fd_gain_pitch	1d12	100	109	10550	105	67	6560	65
Fdec_10i40_35bits	1f7e	100	845	82905	829	492	48455	484
Fdecoder_homing_frame_test	2753	24	47	1128	47	24	576	24
Fdecoder_reset	2774	1	1373	1373	1373	1222	1222	1222
Fpreemphasis	1815	100	490	49000	490	384	38400	384
Freset_dec	27f7	1	2454	2454	2454	1776	1776	1776
Freset_rx_dtx	21fe	1	1064	1064	1064	545	545	545
Frx_dtx	22e8	25	127	3175	127	48	1200	48
Fupdate_gain_code_history_rx	2550	100	29	2609	26	11	1100	11
Fupdate_lsf_history	2470	25	715	17875	715	548	13700	548

Table 9: Initial delays for decoder behaviors.

Routine		calls	cycles			instructions		
			max	total	avg.	max	total	avg
FAutocorr	1888	12	6228	74736	6228	5806	69672	5806
FAz_lsp	16cc	12	34209	405440	33786	18818	223085	18590
FChebps	1820	1244	251	312244	251	142	176648	142
FCoder_12k2	11c9	6	1437055	8610119	1435019	812501	4866760	811126
FConvolve	4077	48	2297	110256	2297	1926	92448	1926
FEnc_lag6	2680	24	98	1860	77	51	1007	41
FG_code	2963	24	1231	29544	1231	805	19320	805
FG_pitch	26da	24	2448	58585	2441	1291	30903	1287
FGet_lsp_pol	41cf	72	1276	91872	1276	807	58104	807
FInt_lpc	2cc5	6	11911	71466	11911	7746	46476	7746
FInt_lpc2	2d15	6	6096	36576	6096	3969	23814	3969
FInterpol_6	30ca	154	259	38478	249	124	18568	120
FInv_sqrt	306c	504	142	68625	136	76	36765	72
FLag_max	4033	36	13620	290418	8067	12517	265881	7385
FLag_window	2c3a	12	226	2712	226	187	2244	187
FLevinson	29d5	12	14092	168872	14072	7549	90443	7536
FLsf_lsp	2fb2	12	470	5634	469	246	2952	246
FLsf_wt	2ee4	12	379	4470	372	231	2746	228
FLsp_Az	4166	36	2898	104286	2896	1882	67752	1882
FLsp_lsf	2fe0	12	2572	30364	2530	1216	14372	1197
FNorm_Corr	25ad	24	15928	352299	14679	10859	241341	10055
FPitch_fr6	2505	24	18283	403716	16821	11979	265867	11077
FPitch_ol	23f5	12	28097	328942	27411	24770	291140	24261
FPre_Process	311d	6	10766	64596	10766	7226	43356	7226
FPred_lt_6	408a	24	2319	55617	2317	1938	46486	1936
FQ_plsf_5	2d58	6	53627	321478	53579	37723	226206	37701
FReorder_lsf	304c	12	261	3126	260	144	1728	144
FResidu	401f	72	1457	104904	1457	1126	81072	1126
FSet_zero	2f20	9	339	908	100	328	809	89
FSyn_filt	3fec	144	1692	239712	1664	1263	179664	1247
FVq_subvec	40ae	24	9020	149286	6220	6694	110736	4614
FVq_subvec_s	40f5	6	20551	123306	20551	15150	90900	15150
FWeight_Ai	27db	96	102	9792	102	80	7680	80
Fbuild_code	22ea	24	5326	126266	5261	2809	66806	2783
Fcode_10i40_35bits	18eb	24	248694	5958414	248267	128118	3070280	127928
Fcor_h	1af7	24	82634	1983120	82630	42753	1026024	42751
Fcor_h_x	1965	24	28544	682536	28439	14511	347144	14464
Fencoder_homing_frame_test	3f46	6	25	150	25	12	72	12
Fencoder_reset	3f59	1	1242	1242	1242	999	999	999
Fq_gain_code	282f	24	2194	50136	2089	1097	25578	1065
Fq_gain_pitch	27f8	24	566	12840	535	267	6160	256
Fq_p	1adc	240	41	8400	35	18	3480	14
Fsearch_10i40	1bdc	24	126468	3023685	125986	65225	1560428	65017
Fset_sign	1a00	24	5199	122983	5124	2640	61886	2578

Table 10: Coder delays after software optimization.

Routine		calls	cycles			instructions		
			max	total	avg.	max	total	avg
FBin2int	17da	1425	143	109285	76	78	57010	40
FBits2prm_12k2	179e	25	6099	152060	6082	3131	77860	3114
FD_plsf_5	186c	25	2588	64694	2587	1501	37468	1498
FDec_lag6	148a	100	140	12475	124	72	6453	64
FDecoder_12k2	1168	25	72030	1736836	69473	44869	1087593	43503
FGet_lsp_pol	26f6	200	1276	255200	1276	807	161400	807
FInit_Decoder_12k2	1128	1	283	283	283	225	225	225
FInit_Post_Filter	139e	1	120	120	120	83	83	83
FInt_lpc	1528	25	11911	297775	11911	7746	193650	7746
FInv_sqrt	180e	191	142	27104	141	76	14507	75
FLog2	1cf2	200	118	23600	118	64	12800	64
FLsf_lsp	1d79	50	470	23475	469	246	12300	246
FLsp_Az	268d	100	2898	289675	2896	1882	188200	1882
FPost_Filter	13b4	25	29825	745607	29824	21315	532866	21314
FPow2	1d3f	100	84	8308	83	42	4154	41
FPred_lt_6	25b1	100	2319	231729	2317	1938	193686	1936
FReorder_lsf	17ee	50	261	13025	260	144	7200	144
FResidu	2546	100	1457	145700	1457	1126	112600	1126
FSet_zero	1753	104	173	3392	32	162	2146	20
FSyn_filt	2513	300	1692	433400	1444	1263	322600	1075
FWeight_Ai	15bb	200	102	20400	102	80	16000	80
Fagc	15d8	100	2228	222782	2227	1421	142091	1420
Fagc2	16a0	91	1934	175994	1934	1250	113750	1250
Fd_gain_code	1aac	100	1097	108717	1087	597	59429	594
Fd_gain_pitch	1a40	100	97	9350	93	55	5360	53
Fdec_10i40_35bits	1c8e	100	711	69505	695	357	34955	349
Fdecoder_homing_frame_test	2421	25	47	1175	47	24	600	24
Fdecoder_reset	2442	1	646	646	646	489	489	489
Fpreemphasis	175e	100	372	37200	372	266	26600	266
Freset_dec	24b9	1	1629	1629	1629	938	938	938
Freset_rx_dtx	1eff	1	966	966	966	440	440	440
Frx_dtx	1fe3	25	127	3175	127	48	1200	48
Fupdate_gain_code_history_rx	222f	100	29	2609	26	11	1100	11
Fupdate_lsf_history	215b	25	505	12625	505	338	8450	338

Table 11: Decoder delays after software optimization.