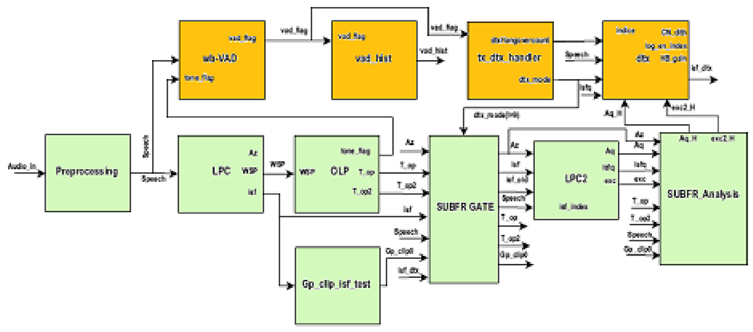
Technical report, IDE1009 , June 1, 2010

Design and Implementation of an Audio Codec (AMR-WB) using Data Flow Programming Language CAL in the OpenDF Environment

Master's Thesis in Embedded and Intelligent Systems

Hazem Ismail Abdel Aziz Ali
Mohammad Nazrul Ishlam Patoary
{hazali08, mohpat08}@student.hh.se



School of Information Science, Computer and Electrical Engineering
Halmstad University

Design and Implementation of an Audio Codec (AMR-WB) using Data Flow Programming Language CAL in the OpenDF Environment

Hazem Ismail Abdel Aziz Ali
Mohammad Nazrul Ishlam Patoary
{hazali08, mohpat08}@student.hh.se

Halmstad University
Project Report IDE1009

Master's thesis in Embedded and Intelligent Systems, 30 ECTS credits

Supervisors

Halmstad University
Prof. Bertil Svensson   PhD. Jerker Bengtsson

Ericsson AB
PhD. Johan Eker   PhD. Harald Gustafsson

Examiner
Prof. Tony Larsson
June 1, 2010

School of Information Science, Computer and Electrical Engineering
Halmstad University

Preface

This project is the concluding part of Master's in Embedded and Intelligent Systems with specialization in Embedded Systems from Halmstad University, Sweden. The whole project has been held at Ericsson Research premises in Lund, since it is a part of an European FP7 project called "ACTORS".
Firstly, we would like to thank our supervisors from Halmstad University especially Professor Bertil Svensson for his continuous encouragement and support and excellent guidance throughout our study.
Also, we are very thankful to PhD. Jerker Bengtsson for his nice comments and feedback. Also, we would like to express our gratitude to our supervisors from Ericsson Research in Lund PhD. Johan Eker and PhD. Harald Gustafsson for providing concrete ideas and cooperating both in terms of time and frequency and providing great guidance and supervision throughout the entire project.
Finally, we would like to thank all our families for their support and faith in us throughout our whole life.

Hazem Ismail Abdel Aziz Ali
Mohammad Nazrul Ishlam Patoary
Halmstad University.

Abstract

Over the last three decades, computer architects have been able to achieve an increase in performance for single processors by, e.g., increasing clock speed, introducing cache memories and using instruction level parallelism. However, because of power consumption and heat dissipation constraints, this trend is going to cease. In recent times, hardware engineers have instead moved to new chip architectures with multiple processor cores on a single chip. With multi-core processors, applications can complete more total work than with one core alone. To take advantage of multi-core processors, we have to develop parallel applications that assign tasks to different cores. On each core, pipeline, data and task parallelization can be used to achieve higher performance. Dataflow programming languages are attractive for achieving parallelism because of their high-level, machine-independent, implicitly parallel notation and because of their fine-grain parallelism. These features are essential for obtaining effective, scalable utilization of multi-core processors.

In this thesis work we have parallelized an existing audio codec - Adaptive Multi-Rate Wide Band (AMR-WB) - written in the C language for single core processor. The target platform is a multi-core AMR11 MP developer board. The final result of the efforts is a working AMR-WB encoder implemented in CAL and running in the OpenDF simulator. The C specification of the AMR-WB encoder was analysed with respect to dataflow and parallelism. The final implementation was developed in the CAL Actor Language, with the goal of exposing available parallelism - different dataflows - as well as removing unwanted data dependencies. Our thesis work discusses mapping techniques and guidelines that we followed and which can be used in any future work regarding mapping C based applications to CAL. We also propose solutions for some specific dependencies that were revealed in the AMR-WB encoder analysis and suggest further investigation of possible modifications to the encoder to enable more efficient implementation on a multi-core target system.

Contents

Chapter 1

Introduction

Multi-core architectures integrate several processors on a single chip, and they are quickly becoming widespread. Being affordable, it is now possible for every PC user, or even a small embedded system such as a mobile phone, to incorporate a truly parallel computer. This massive computational power makes parallel programming a concern for more software developers than before. Since each field of application requires its own way of parallelization e.g., numerical applications mostly parallelized using task level parallelism, server applications parallelized using multithreading [21]. From that perspective, parallel programming is considered difficult since many fundamental questions must be answered before starting to develop parallel applications for multi-cores, such as: what programming language is useful? Which parallelization approach suits the application? And how can existing sequential applications be reengineered for parallelism? [20].

In our thesis work, we conducted a case study of parallelizing an audio codec application for multi-core processors using CAL dataflow programming language under OpenDF environment. Ericsson AB selected the sequential Adaptive Multi-Rate Wide Band (AMR-WB) audio codec program for the study, because it is widely used in communication systems, and relevant application in everyday life. Its source code specification is available in C language, and it is well documented.

The selection of dataflow programming language for parallelizing AMR-WB codec came from the fact that the application lies in the digital signal processing field (DSP). The DSP field is characterized by continuous data streams that flow across the application, where different application tasks operate on the flow. By this, dataflow programming language achieves both data level parallelism(DLP) and task level parallelism (TLP), which will increase the performance gained from multi-core processors.

The selection of CAL language as an example of dataflow programming languages for design and implementation came from Ericsson AB. Ericsson is leading a European Fp-7 project called "ACTORS", developing tools and theories for dataflow programs on multi-core systems. As part of this, a CAL to C compiler and a runtime system are being developed for an ARM11 multi-core platform. CAL is a dataflow language based on an actor model that provides the proper foundation for implementation of efficient, component based, and adaptive algorithms for both multimedia applications in consumer electronics and industrial control systems and signal processing applications [1].

1.1 Related Work

A case study is performed in [14], where the MPEG-4 Simple Profile (MPEG-4 SP) decoder was specified in CAL, according to the MPEG Reconfigurable Video Coding (RVC) formalism. The MPEG RVC framework is a new ISO standard, aiming to design a decoder at a higher level of abstraction than the one provided by current generic monolithic C based specifications. Instead of low level C/C++ code, an abstract model based on modular components taken from the standard Video Tool Library (VTL) is the reference specification [9]. The MPEG-4 Simple Profile decoder has been implemented on an FPGA using a CAL-to-RTL code generator called Cal2HDL. A similar implementation has also been done directly in VHDL. In the same case study a comparison has been made. It was found that code generated from CAL needs less development effort and less memory space compared to the hand-written reference in VHDL. Thus, the work on the MPEG-4 decoder confirms the potential of the dataflow approach.

In another case study [11], the H.264 encoder reference C code was converted to an extended Synchronous Dataflow (SDF) model, using HW/SW codesign environment PeaCE [4] that supports automatic C code generation from dataflow specification. The authors presented a systematic procedure to convert a sequential C code to a dataflow specification and successfully applied it to the H.264 encoder algorithm and obtained the dataflow specification. Lastly, they compared the synthesized code from dataflow specification with the reference code in terms of encoding time, code size and data size. As they did not make optimization on their dataflow specification, the synthesized code showed worse performance and code size, but it showed a better result on the data size.

1.2 Thesis Contribution

The goal of this thesis work is to convert an existing single processor AMR-WB audio codec to multi-core platform (ARM11 MPs). The approach is to first find out the independent dataflow from the codec specification by using a dataflow approach and then implement the codec in actor programming language CAL. As the specification of AMR-WB audio codec was in C code, we have proposed a systematic approach, in Chapter 6, to convert reference C code to CAL Actor Language.

In the implementation of the AMR-WB encoder, we got two major flows of speech data which are considered as data-parallel for two cores. However, in reality, according to our specification of the AMR-WB encoder, those two flows of speech data have some dependencies to each other. In our work, we specified the cause of dependencies and we have proposed some solutions to overcome those dependencies in Chapter 8.

1.3 Thesis Organization

In this thesis, we have analyzed the dataflow in AMR-WB audio codec to find the hidden parallelism. Chapter 2 will introduce the dataflow and some graphical model of computations. It will also introduce the OpenDF environment that is used as a tool in our thesis work.

We implemented our encoder by using the CAL programming language. Chapter 3 will introduce the CAL programming language and will show how a hierarchical level of programming can be achieved from the actor network design concept in CAL. In Chapter 4, we shall explain the working principle of AMR-WB audio codec in brief, aided with various plots of our speech frame in different computational phases.

The high level dataflow of the AMR-WB encoder is described in Chapter 5. To find out a way to map from C to CAL language, Chapter 6 provides some guidelines that help in that transformation process.

In Chapter 7 we will discuss in detail the implementation of the AMR-WB encoder and data dependency problems throughout the design. Chapter 8 discusses the proposed solutions for the data dependency problems mentioned in Chapter 7 and solutions for enhancing the entire performance of the AMR-WB encoder. Finally, our thesis ends with Chapter 9 that concludes the whole work.

Chapter 2

Dataflow Programming

There are many applications like audio and video coding, which apply a series of transformations to a data stream. Dataflow programming is an efficient strategy for implementation of such kinds of applications for multi-core processors. The dataflow programming emphasizes only the flow of data and does not represent the flow of control explicitly. Applications implemented in dataflow programming language consists of a set of modules that interconnect, forming a new module or network. The modules are self-contained computational entities that perform a specific operation. Thus, a module is a computational unit while a network is an operational unit. Inter-module communication is done by passing tokens through unidirectional input and output ports. The dataflow models offer a naturally parallel representation that can effectively support the tasks of parallelization [16] and thus providing a practical means of supporting multiprocessor systems and utilizing vector instructions.

Dataflow programming language implements dataflow principles and architecture, and models a program as a directed graph of data flowing between modules. Such kinds of programming languages were originally developed in order to make parallel programming easier. The Dataflow Process Network [18] model of computation provides a framework within which a language is defined, called "CAL Actor Language". Dataflow programming structures the applications as networks of "black box" elements that exchange data across predefined connections by token passing.

In the case of conventional programming languages, a program is modeled as a series of operations and the flow of data is effectively invisible, for example: C, Java etc, whereas dataflow programming models the flow of data through the network of computational elements, for example, the CAL Actor Language. CAL is a domain specific language that provides useful abstraction for dataflow programming with actors.

The first section of this chapter will explain how dataflow can facilitate parallel computing for the multi-core environment, then the next section will define some graphical models of computations, and then concludes by introducing the Open Data Flow (OpenDF) environment. The OpenDF [8] is an environment for building and executing dataflow models, including support for the CAL Actor Language.

2.1 Dataflow Model of Computation

The basis of dataflow programming models is explicitly specified by a directed graph, the nodes are considered as computational units and the connection between the nodes, i.e arcs, as channels of data. To be able to establish the reasons for the behaviour of the computations performed, the dataflow graph has to be put in the context of a computation model, which defines the semantics of the communication between the nodes. The most common Models of Computation (MoC) for DSP are Kahn Process Networks, Dataflow Process Network and Synchronous Dataflow Network.

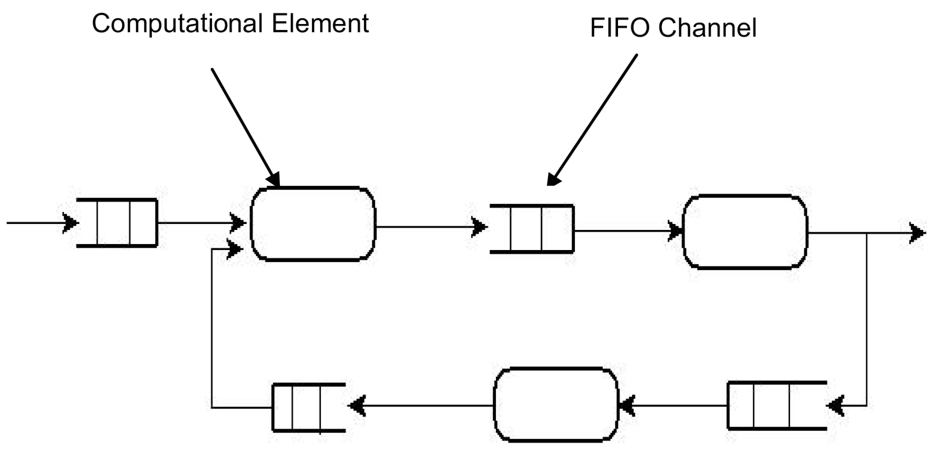


Figure 2.1: Dataflow model of computation.

2.1.1 Kahn Process Network

Process Networks is a MoC that was originally developed for modeling distributed systems and this model is also used for modeling signal processing systems. Process Networks are also called "Kahn Process Networks", (KPN), since G. Kahn first introduced this model in his thesis work in 1974 [15]. It is a natural model for describing signal processing systems where infinite streams of data are incrementally transformed by processes executing in sequence or parallel.

In this model, concurrent determinant processes communicate only through one-way FIFO channels with unbounded capacity and the resultant process must be determinant. Each channel carries a possibly infinite number of atomic data objects, or tokens. Writes to the channels are non-blocking, but reads are blocking. If a process tries to read from an empty input, it is suspended until it has enough input data and the execution context is switched to another process.

In 1995, Lee and Parks pointed out [18] that a model of computation does not require multitasking or parallelism and, in reality, infinite queues in communication channel is impractical. It is usually more efficient than comparable methods in functional languages. Process Networks have found many applications in modeling of embedded systems as it is typical for embedded systems to be designed to operate infinitely, with limited resources.

Research software, like Khoros [19] from the University of New Mexico, and Ptolemy [12] from the University of California at Berkeley, are all based on variants of the Process Network model. Departing from the original Process Networks by Kahn, a number of more specific models have been derived.

There are many applications of KPNs model in modelling embedded and high-performance computing systems, such as, for example, the Ambric Am2045 massively parallel processor array, in which 336 32-bit processors are interconnected by a programmable interconnect of dedicated FIFO and the channels are strictly bounded with blocking writes.

2.1.2 Dataflow Process Network

Dataflow Process Networks is a MoC very closely related to Kahn Process Networks. In this model, arcs represent FIFO queues as arcs in Kahn Process Networks, but now the nodes of the graph, instead of representing processes, represent actors. Instead of responding to the blocking-read semantics of Process Networks, actors use firing rules that specify how many tokens must be available on every input for the actor to fire. When an actor fires, it consumes a finite number of tokens and produces also a finite number of output tokens, i.e. channel capacity is not infinite.

An actor may have more than one firing rule. The evaluation of the firing rules is sequential in the sense that rules are sequentially evaluated until at least one of them is satisfied. Thus, an actor can only fire if one or more than one of its firing rules are satisfied.

In Dataflow Process Networks, each process consists of repeated "firings" of a dataflow "actor". An actor defines a (often functional) quantum of computation.

- Dataflow Actor: this maps input token to output token.
- Firing: this consumes input tokens and produces output tokens.
- Firing rules: these determine when an actor can fire.

In Dataflow Networks, instead of suspending a process on blocking read or non-blocking write, processes are freely interleaved by a scheduler that determines the sequence of actor firings. The biggest advantage is that the cost of process suspension and resumption is avoided [18].

2.1.3 Synchronous Dataflow (SDF)

Synchronous Dataflow (SDF) is a special case of dataflow. An actor is said to be synchronous if the number of input tokens that are consumed on each input and the number of output tokens that are produced on each output can be specified a priori. A SDF graph is a network of synchronous nodes. The same behavior repeats in a particular actor every time it is fired.

In SDF, channels can have initial tokens (delay). Every initial token represents an offset between the token produced and the token consumed at the other end. If a channel has an initial token, then the receiving actor will read that initial token before it will read the token output by the sending actor. Initial tokens are indicated on an arc by a number followed by a D (for delay). Figure 2.2 demonstrates a simple SDF model that contains an initial token.

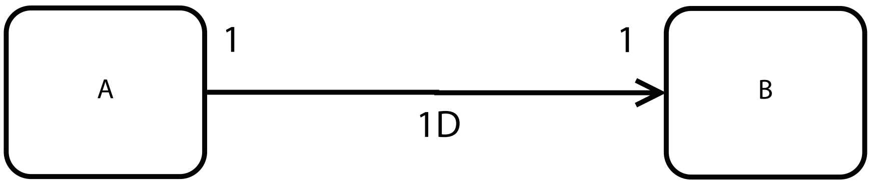


Figure 2.2: SDF with one initial token.

Here the a single delay 1D between actor A and B is specified. When actor B first fires, it consumes the initial token from the delayed channel, and actor B consumes A's first output token at his second firing time. Thus, delays on a channel can affect the precedence relationship between actors. In

Figure 2.2, actor B can fire before actor A. However, in order to fire a second time, actor A will have to fire at least once. An SDF network can contain a feedback loop. At that time, the feedback loop must contain initial tokens so that there will be no deadlock. Figure 2.3 shows an SDF network with a feedback loop.

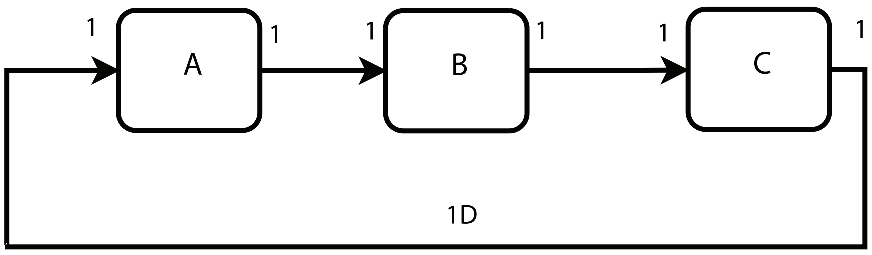


Figure 2.3: SDF with feedback & initial token.

Note that actor A will fire first due to the initial token on the arc between actor C. Then actor B will fire by consuming the output token from actor A. In this way, actor C will fire and, in turn, actor A. If there is no initial token between actor C and A, then there is no actor which has sufficient tokens on its inputs to fire and a deadlock situation will arise.

To implement an SDF model, there must be a scheduler to order the firing of actors in the model. The firing rules of an SDF model include that an actor have sufficient tokens on all of its inputs before the actor can be fired. The SDF schedule must fire the actors in an order that ensures each actor has sufficient tokens in its input ports.

2.2 Multi-core Platform and Dataflow Programming

Dataflow model represents a parallel processing model, which combines Task Level Parallelism (TLP) and pipeline parallelization to achieve high performance and good scalability from a multi-core platform. If we want to deploy a stream based application on multi-core platform we have to, at first, investigate different independent dataflows. Then we partition those independent flows to perform task parallel processing. Finally, we organize and execute them as several stages pipeline to exploit more parallelism.

In dataflow programming, the independent dataflows can be modeled as parallel computation. Each independent flow of data passes through a set of sequential computational elements (actors). These set of sequential computational elements represent a pipeline with several stages. Thus, dataflow programming is an efficient and excellent way to design and implement stream based applications for multi-core platform.

2.3 OpenDF

There are a number of tools available for specifying and modeling the stream based application in dataflow semantics. One such tool is the OpenDF to model and design stream based applications like audio or video codecs. The OpenDF environment is structured as a sequence of transformations on Extensible Markup Language (XML) documents which describe the functionality of a dataflow system. The transformation, from source code to XML and then XML to Hardware Description Language(HDL) or simulation model, or other compiler output, uses XML documents that may be useful representations for integration with other tool flows or for export to other compiler back-ends.

The typical user interface for the Open Dataflow tool is the Eclipse IDE platform. In this platform there are various plugins have been developed and deployed, which provide access to the various development, simulation and compilation features. All the tools of the OpenDF are implemented as a combination of Java source code and Extensible Stylesheet Language Transformation (XSLT). The Java source provides the interfaces to logical groupings of the XSLT transforms.

The OpenDF supports the CAL language and generates HDL (VHDL / Verilog), C for integration with the SystemC tool chain, and embedded C [8].

Chapter 3

Programming in CAL

In this era of multi-core processing, it is a big challenge for the software designers to model stream based applications in parallel computing. Programmers have to replace the conventional sequential programming paradigm for parallel programming. As Edward Lee pointed out [17], using threads as a model of computation makes this parallelisation process a tedious task. In this chapter we shall introduce the CAL Actor Language (CAL), created as a part of the Ptolemy II project at UC Berkeley. For more information about CAL Actor Language (CAL) you can refer to the *CAL Language Report* [10] and the *A Gentle Introduction to CAL* tutorial written by J. Janneck [13].

3.1 The Basic Structure of an Actor

In actor oriented programming with CAL, an actor is a basic computational entity with input and output ports, parameters, states and actions as shown in Figure 3.1. It communicates with other actors by sending and receiving atomic data, called token, along unidirectional FIFO channels. When actors are interconnected to each other, then a model is developed called "actor network".

CAL is a small, domain-specific language to specify the functionality of actors. The functionality of an actor is defined by a set of actions and their associated firing rules. The firing rules are conditions on the presence of tokens on the input ports and possibly also on their values. The execution of an actor is said to be a "firing". During a firing, tokens on the input ports are consumed and tokens on the output ports are produced. The selection order and the firing conditions for actions form the core of the design of an actor. CAL provides a number of constructs for describing action selection, which include guards (conditions on the values of input tokens and/or the

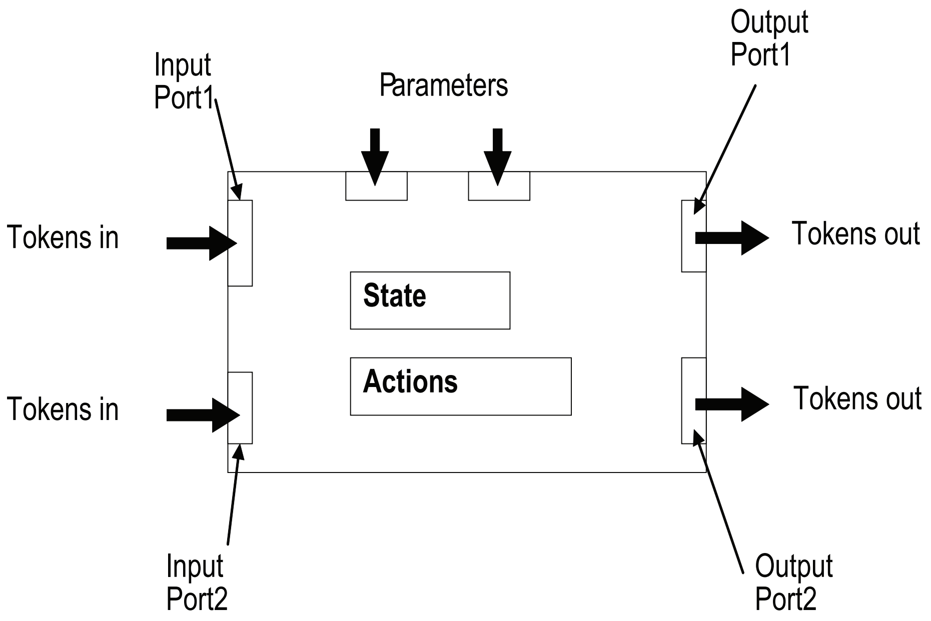


Figure 3.1: Actor basic structure.

values of actor state variables), a finite state machine and priorities.

3.2 Basic Syntax of an Actor in CAL

The basic syntax to implement an actor in CAL language is shown in Figure 3.3.

The first step to declare an actor is to use the `actor` keyword, followed by a list of parameters, input and output ports . The input ports are those in front of the `==>` symbol e.g. `InputPort1`, `InputPort2`, the output ports are those after that symbol e.g. `OutputPort1`, `OutputPort2`.

Another keyword `action` defines an action with its label `Action1`, `Action2`. In general, an actor may have any number of actions. The input syntax (before the `==>` symbol in action) `InputPort1:[a]` says that one token with the name "a" is consumed from input port `InputPort1` while the actor is fired. It has to be remembered that an action will fire only when it gets sufficient number of tokens on its input ports. The output syntax (after the `==>` symbol in action) `OutputPort1:[a]` says that one token "a" is produced whenever the action is fired.

The order of action firing can be controlled by making a schedule using the keyword schedule `fsm`. In this schedule, at the `initialState` only `Action1`

```
actor ActorName(parameter1, parameter2 .... )
InputPort1, InputPort2, ... ==> OutputPort1, OutputPort2, ... :

   DataType state1;
   DataType state2;
   ⋮
   Action1 : action InputPort1: [a], ... ==> OutputPort1:[a]
      Statements ;
      ⋮
   end

   Action2 : action InputPort2: [a], ... ==> OutputPort2:[a]
      Statements ;
      ⋮
   end

   schedule fsm initialState:
      initialState  (Action1) --> state2;
      state2 (Action2)  --> state3;
      ⋮
   end
end
```

Figure 3.2: Basic syntax of an actor.

will fire and at the next state state2 the action `Action2` will fire and so on. The `Action2` in `state2` can not fire before `Action1` in `initialState` has been fired.

3.3 Some Simple Examples of Actors

Our first example, *Add*, is an actor that has two input ports (`Input1` and `Input2`) and one output port (`Output`). It has a single action that reads one token from each of the input ports. The single output token produced by this action is the sum of the two input tokens:

```
actor Add() Input1, Input2 ==> Output:
   action Input1: [a], Input2: [b] ==> Output: [a + b]
   end
end
```

Figure 3.3: Simple Add actor.

In a dataflow programming language, an actor is considered as an operator on a flow of data tokens. This operator acts as consumer, while token entering on its input ports, and producer, while a flow of tokens leaving on its output ports. In our thesis work, the stream of data is the samples of speech frame.

3.3.1 Nondeterministic Actors

As we already mentioned, actors can have any number of actions. A nondeterministic actor is one that, for the same input sequences per port, allows more than one possible output dependent on, e.g. timings of token reception. Nondeterminism can be very powerful when used appropriately, but it can also be a very troublesome source of errors. An example of a nondeterministic actor is given in Figure 3.4.

Here, there are three actions in this actor named *calculator*. Any action can be fired whenever they fulfill the necessary condition, i.e when one token arrives in `Input1` port `Square` action will fire and will produce an output token. Again, if a token arrives at `Input2` port, and then one in `Input1` port, any action can be fired. So it indicates that, although the same input sequence is inputted to both input ports, the produced output will not be the same. Thus, this actor is nondeterministic.

```
actor calculator () Input1, Input2 ==> Output:
   Add      : action Input1: [x], Input2: [y] ==> [x+y] end
   Subtract : action Input1: [x], Input2: [x] ==> [x-y] end
   Square   : action Input1: [x]              ==> [x*x] end
end
```

Figure 3.4: Nondeterministic actor.

3.3.2 Building Deterministic Actors: Guard Condition

We can control the firing of actions by using guard condition. Guard condition specifies additional criteria (condition) that need to be satisfied for an action to fire. The condition might be the values of the input tokens or state of the actor. Such type of condition is specified by using guards as, for example, the same calculator actor:

```
actor calculator() Input1, Input2 ==> Output:
   int selector = 1;
   Add : action Input1: [x], Input2: [y] ==> Output: [x+y]
   guard selector = 1
   do
      selector := selector + 1;
   end
   Subtract : action Input1 : [x], Input2: [y] ==> Output: [x-y]
   guard selector = 2
   do
      selector := selector + 1;
   end
   Square : action Input1 : [x] ==> Output: [x*x]
   guard selector = 3
   do
      selector := 1;
   end
end
```

Figure 3.5: Deterministic actor using guard condition.

In the example shown in Figure 3.5, `selector` is a state with initial value 1 and we use it as our guard of each action. When tokens in both ports have arrived and the selector value is 1, then action Add will be fired. Thus, we can determine our firing order of actions. In this example, at first `Add` action will fire, then `Sub` action and then `Square` action will fire. Now, for any sequence of input tokens on both input ports, this actor will produce the same sequence of output. So this actor is a deterministic actor.

3.3.3 Building Deterministic Actors: Schedule

CAL language provides a special syntax to control the order of firing of actions that is `schedule`. The new version of `calculator`, with the `schedule` shown in Figure 3.6, has the same order of firing of the actions, first `Add` then `Subtract` and then `Square`.

```
actor calculator () Input1, Input2 ==> Output:
   Add : action Input1:[x], Input2:[y] ==> Output:[x+y] end
   Subtract : action Input1:[x], Input2:[y] ==> Output:[x-y] end
   Square   : action Input1:[x] ==> Output:[x*x] end
   schedule fsm initialState :
      initialState (Add) --> state1;
      state1 (Subtract) --> state2;
      state2 (Square) --> initialState;
   end
end
```

Figure 3.6: Deterministic actor using schedule.

3.3.4 Firing more Tokens: using Repeat

All examples discussed till now were consuming and producing only one token at a time. However, in stream based applications, sometimes it is required to manage more than one token at time. In CAL, actors can consume and produce more than one token at a time by using the `repeat` keyword, e.g. Figure 3.7 shows an actor named *twice*, which has an action called `mul2`, consumes 5 tokens and each token is multiplied by 2, and then 5 output tokens are fired. It is not essential to keep equal the repeat value in input port and output port i.e. any number of tokens can be consumed and any number of tokens can be fired at a time.

```
actor twice () Input ==> Output:
   mul2: action Input:[x] repeat 5  ==> Output:[y] repeat 5
   var List(int , size=5) y
   do
        y := [x[i]*2 : for i in 0..4 ];
   end
end
```

Figure 3.7: Actor receiving and firing multiple tokens using repeat.

3.4 Compositions of Actors

Modeling of stream based applications can not be represented by one actor only; it requires several actors connected to each other through channels to maintain data flow. In this section, we will show how a network of actors can be built in CAL.

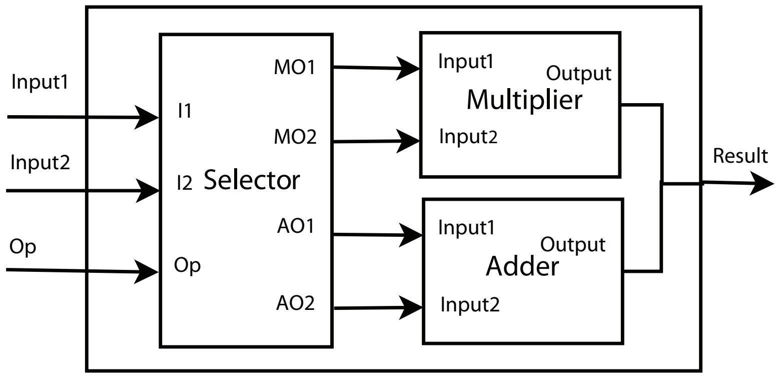


Figure 3.8: The graphical representation of *Calculator* application.

To build an actor network, the first step is to instantiate the building components of the network which are the actors; second is to create connections among them. In addition, a network of actors might have input and output ports who are connected to the ports of actors inside the network. As an example, we will discuss a small *Calculator* application, which is shown in Figure 3.8.

Consider we have two actors, `Multiplier` and `Adder`, shown in Figure 3.9. Both actors have the same input and output ports `Input1`, `Input2` and

`Output`; each actor consumes 2 tokens, `a` and `b`, and fires the result `a+b` for `Adder` actor and `a*b` for *Multiplier* actor. `Selector` has three input ports i.e., `I1`, `I2`, `Op` and 4 output ports i.e., `MO1`, `MO2`, `AO1`, `AO2`. If the `Op` input token `m` is equal to 1, the input tokens `a` and `b` on input ports `I1` and `I2` respectively will be directed to output ports `MO1` and `MO2`, which will result in firing of actor `Multiply`. Otherwise if `Op` value is 2, the same input tokens will be directed to output ports `AO1` and `AO2`, which will result in firing of actor `Add`.

```
actor Adder() Input1, Input2  ==> Output:
   Add: action Input1:[a], Input2:[b] ==> Output:[a+b]
   end
end

actor Multiplier() Input1, Input2 ==> Output:
   Mult: action Input1:[a], Input2:[b] ==> Output:[a*b]
   end
end

actor Selector() I1,I2,Op ==> MO1,MO2,AO1,AO2:
   int flag:=0;
   Decision : action  Op[m] ==> guard flag = 0 do
      if m = 1 then flag := 1; end
      if m = 2 then flag := 2; end
   end
   Mult : action I1:[a], I2:[b] ==> MO1:[a], MO2:[b]
   guard flag =1 do
      flag := 0;
   end
   Add : action I1:[a], I2:[b] ==> AO1:[a],AO2:[b]
   guard flag =2 do
      flag := 0;
   end
end
```

Figure 3.9: *Calculator* application basic actors .

To build the *Calculator* application network, we have to create instances of all actors, that will be used. This is done by using the keyword `entities` followed by instantiation of different actors participating in this network. After the `entities` section comes the `structure` section which describes the different connections between actors. Figure 3.10, shows a complete network file syntax for *Calculator* Application. While Figure 3.8 shows the *Calculator* network file graphical representation. This *Calculator* application network is not deterministic.

```
network Calculator() Input1, Input2, Op  ==> Result :
   entities
      M = Multiplier();
      A = Adder();
      S = Selector();
   Structure
      Input1 --> S.I1;
      Input2 --> S.I2;
      Op --> S.Op;
      S.MO1 --> M.Input1;
      S.MO2 --> M.Input2;
      S.AO1 --> A.Input1;
      S.AO2 --> A.Input2;
      M.Output --> Result;
      S.Output --> Result;
end
```

Figure 3.10: *Calculator* application network file.

Chapter 4

Adaptive Multi-Rate Wideband Speech Codec (AMR-WB)

In this chapter we will present the functional description of the AMR-WB speech codec. First, we will introduce the codec and then the working principle of encoder and decoder parts, respectively. The specification of this codec is written in C code [5].

4.1 Background

An audio or speech codec is a device or computer program capable of encoding and decoding an audio signal. The objective of an audio codec algorithm is to represent the audio signal with minimum number of bits while retaining the natural quality. Thus, an audio codec effectively reduces the storage space and the bandwidth required for transmission of the audio signal.

The Third Generation Partnership Project (3GPP) and European Telecommunication Standards Institute (ETSI) have chosen AMR-WB codec for the Universal Mobile Telecommunications System (UMTS) to get wideband speech services. At present, the audio codec used in second and third generation (3G) mobile communication systems operates with a narrow audio bandwidth limited to 200-3400 Hz. AMR-WB introduces a wide audio bandwidth of 50-7000 Hz and improved speech quality and naturalness [7]. This codec is called "Adaptive Multi-Rate" because it is capable of operating with a multitude of speech coding bit-rates from 6.6 to 23.85 Kbits/s. In device context view, AMR-WB is shown in Figure 4.1.

The AMR-WB speech codec has nine speech coding modes with bit-rates of 6.6, 8.85, 12.65, 14.25, 15.85, 18.25, 19.85, 23.05 and 23.85 kbit/s [5].

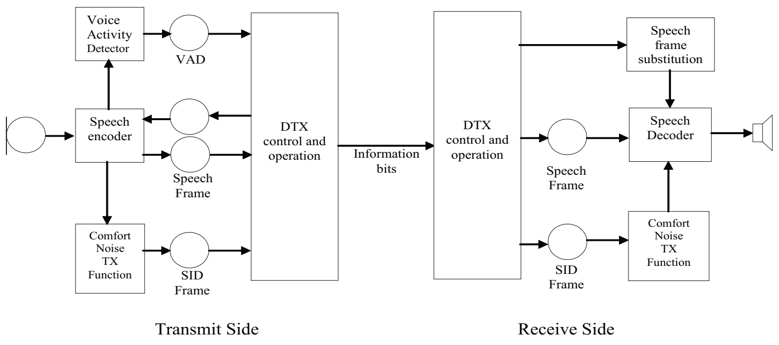


Figure 4.1: Device context view of AMR-WB codec.

AMR-WB includes also a background noise mode which is designed to be used in discontinuous transmission (DTX) operation in GSM/UMTS and as a low bit-rate source dependent mode for coding background noise in other systems. In GSM/UMTS the bit-rate of this mode is 1.75 kbit/s.

4.2 Working Principle of AMR-WB Encoder

The codec is based on the Linear Predictive Coding (LPC) model and the Code Excited Linear Predictive (CELP) coding model. At the beginning, the speech signal is sampled at a rate of 16 kHz and then it is processed for LPC analysis and CELP model.

In LPC analysis, linear prediction coefficients of an order 16 synthesis filter are generated and then those coefficients are quantized and interpolated. The LPC is performed once per 20 ms speech frame. In the CELP model, the excitation signal at the input of the LP synthesis filter is constructed by adding two excitation vectors from an adaptive and a fixed codebook. The speech is synthesized by feeding the two properly chosen vectors from those codebooks through the LP synthesis filter. The optimum excitation sequence in a codebook is chosen using an analysis by synthesis search procedure in which the error between the original and synthesized speech is minimized according to a perceptually weighted distortion measure.

At each frame, the speech signal is analysed to extract the parameters of the CELP model (LP filter coefficients, adaptive and fixed codebooks indices and gains). A high-band gain index is computed in 23.85 Kbit/s mode. Those parameters are encoded and transmitted. At the decoder, those parameters are decoded and speech is synthesized by filtering the reconstructed excitation signal through the LP synthesis filters.

4.2.1 Speech Signal Pre-processing

In this phase, the speech signal is decimated from 16 kHz to 12.8 kHz which converts a frame of 320 samples to 256 samples. After the decimation, two preprocessing functions are applied to the signal prior to the encoding process: high-pass filtering and pre-emphasizing. The high-pass filter serves as a precaution against undesired low frequency components.

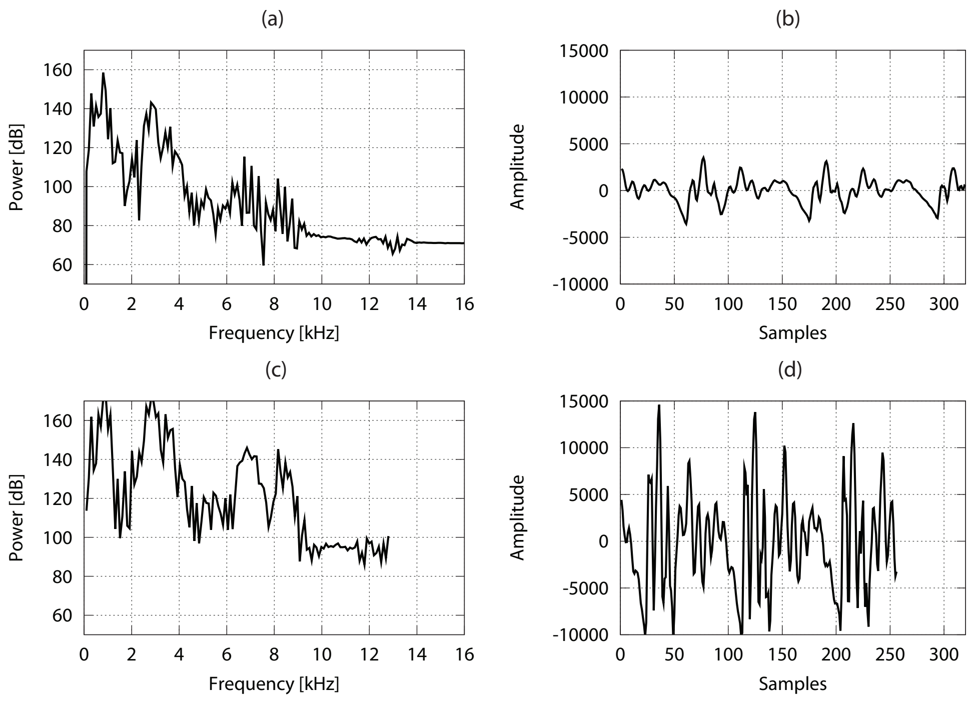


Figure 4.2: *a.* Spectrum of input speech frame, 16kHz signal *b.* Input speech frame in time domain, 16kHz *c.* Spectrum of preprocessed speech frame, 12.8 kHz signal *d.* Preprocessed speech frame time domain, 12.8 kHz

In pre-emphasis, a first order high-pass filter is used to emphasize higher frequencies to whitening the signal speech. One frame of input speech signal and corresponding frame after processing the signal is shown in Figure 4.2.

4.2.2 LPC Analysis

After pre-processing the input speech signal, the emphasized signal (256 samples/s, 12.8 kHz.), is sent for LPC analysis. LP analysis is performed once per speech frame using the autocorrelation approach. Before autocorrelation, the speech signal is windowed by a Hamming window [7]. The autocorrelation of the windowed speech signal is used to obtain LP filter coefficients by using Levinson-Durbin Algorithm [7]. For this codec, there are 16 LP coefficients computed for the order 16 linear prediction filter. The LP filter coefficients are converted to the Immetance Spectral Pairs (ISP) representation for quantization and interpolation purpose. After quantization and interpolation of ISPs, they are converted back to the LP coefficient.

A weighting filter coefficient is generated based on the LP coefficients and the pre-processed speech signal and a residual weighted speech signal is generated for open loop pitch analysis as shown in Figure 4.3.

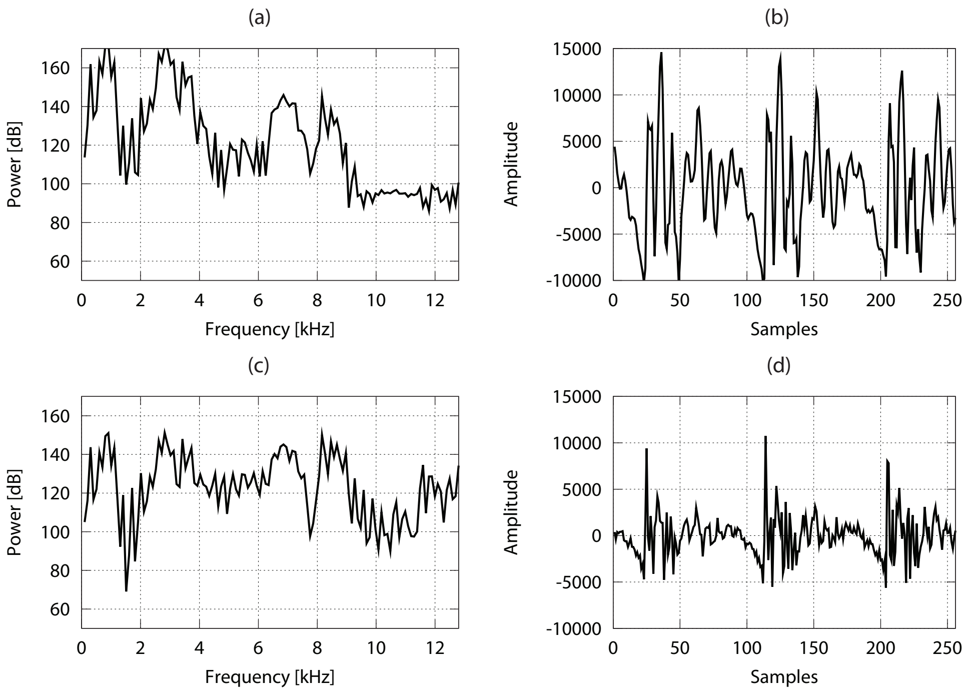


Figure 4.3: *a.* Spectrum of preprocessed speech frame *b.* Preprocessed speech frame in time domain *c.* Spectrum of residual weighted speech signal *d.* Residual weighted speech signal in time domain

4.2.3 Open Loop Pitch Analysis

On the basis of the current mode, open-loop pitch analysis is performed once per frame, only for the mode 6.6 kHz/s and twice per frame, for higher modes, to find estimates of the pitch lag in each frame. This is done in order to simplify the pitch analysis and confine the close loop pitch search to a small number of lags around the open loop estimated lags. Open loop pitch estimation is based on the weighted speech signal which is obtained by filtering the input speech signal through the weighting filter.

4.2.4 VAD Analysis

AMR-WB codec has a feature to save power in the mobile station and reduce the over all interference level over the air interface called Voice Activity Detection (VAD). In VAD analysis, a boolean VAD decision for each frame is generated. In this phase, a tone detection function is used for indicating the presence of a signaling tone, voiced speech or other strongly periodic signal and generating a tone-flag. The tone detection function uses the open loop pitch gain achieved from the open loop pitch analysis phase. If the pitch gain is higher than the threshold value, the tone is detected and the tone flag is set. When the tone flag is set and the speech level of a speech frame is greater than a minimum speech level, the VAD flag is set. Thus, the VAD decision depends on the tone-flag and the speech level of the speech frame.

4.2.5 Discontinuous Transmission (DTX) and Comfort Noise Generation

Discontinuous Transmission is a strategy in which a mobile station transmitter is to be switched off most of the time during speech pauses and a DTX comfort noise almost similar to background noise is generated in the receiver end. This comfort noise is generated on the basis of the parameters from the background noise during the initial part of the speech pause.

4.2.6 Subframe Analysis

The speech frame is divided into four subframes of 5 ms each, i.e. 64 samples in each subframe. For each subframe, the operation is briefly explained as follows:

- The target signal is computed by filtering the LP residual through the weighting filter as shown in Figure 4.4.
- The impulse response signal of the weighted synthesis filter is computed.
- Closed loop pitch analysis is then performed to find the pitch lag and gain by using target and impulse response and searching around the open-loop pitch lag.

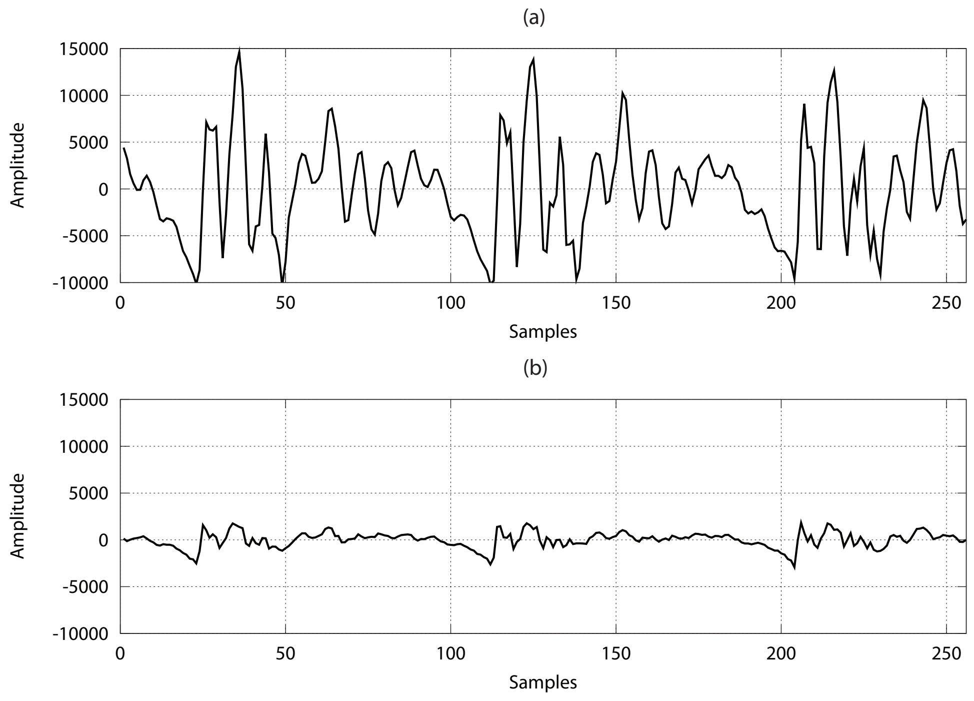


Figure 4.4: *a.* Preprocessed speech frame, 12.8 kHz signal *b.* Target signal for adaptive codebook search, 12.8 kHz signal

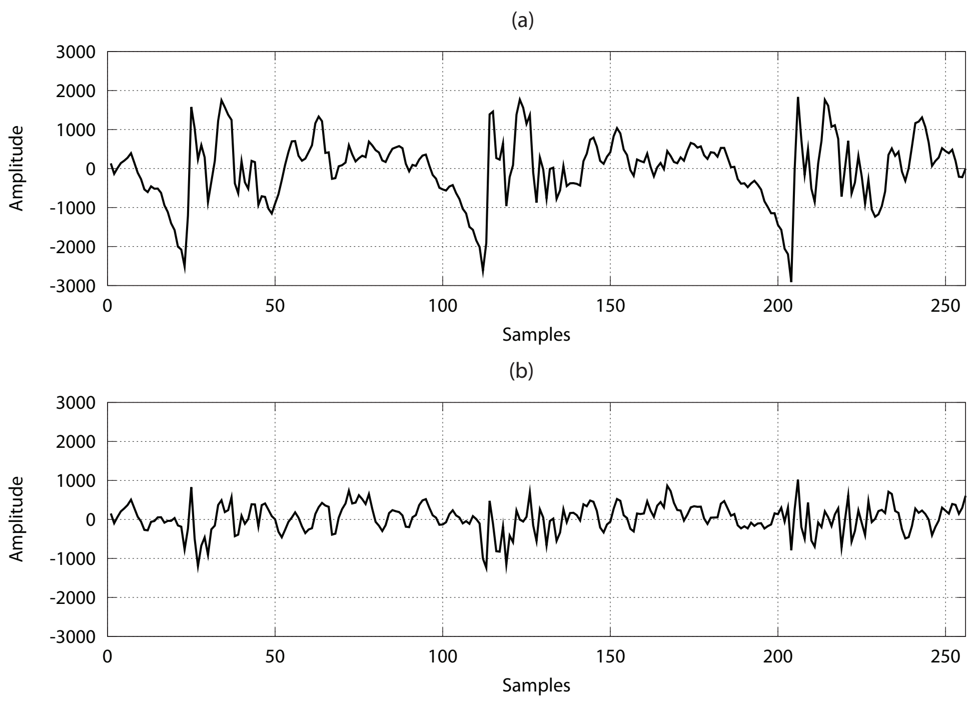


Figure 4.5: Adaptive codebook search in mode 0: *a.* Target signal for adaptive codebook search, 12.8 kHz signal *b.* Updated target signal after removing adaptive contribution

- The target signal is updated by removing the adaptive codebook contribution as shown in Figure 4.5 for lowest mode (6.6 Kbits/s) and in Figure 4.6 for highest mode (23.85 Kbits/s), and this new updated target is used in the fixed algebraic codebook search to find the optimum innovation as shown in Figure 4.7 for lowest mode (6.6 Kbits/s) and in Figure 4.8 for highest mode (23.85 Kbits/s).
- The gains of the adaptive and fixed codebook are indexed to transmit.
- Finally, the filter memories are updated for finding the target signal in the next subframe.

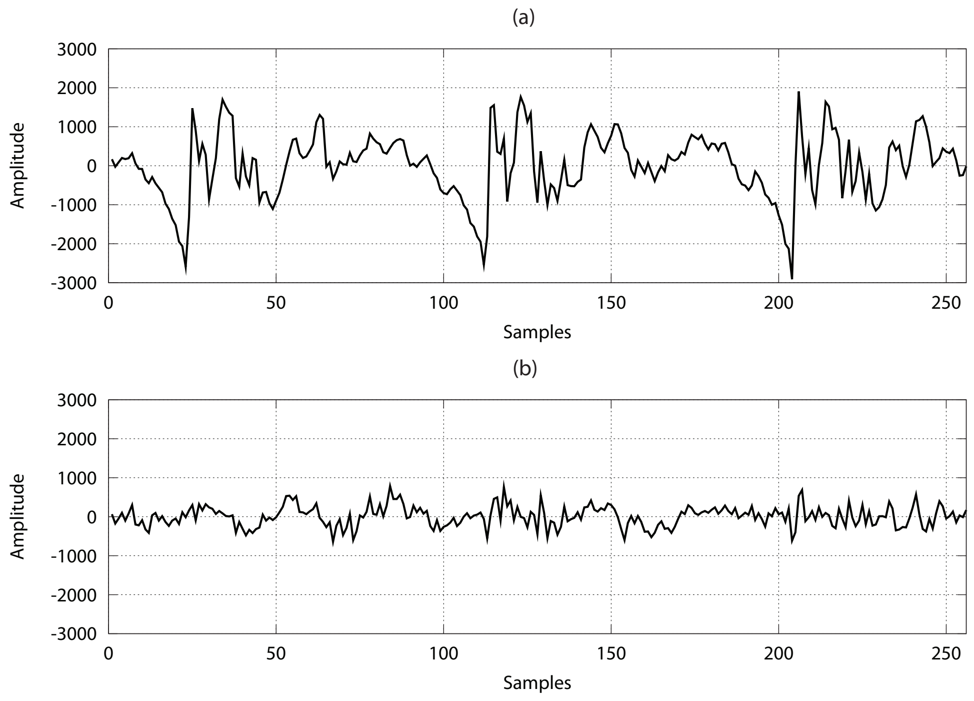


Figure 4.6: Adaptive codebook search in mode 8: *a.* Target signal for adaptive codebook search, 12.8 kHz signal updated *b.* Target signal after removing adaptive contribution

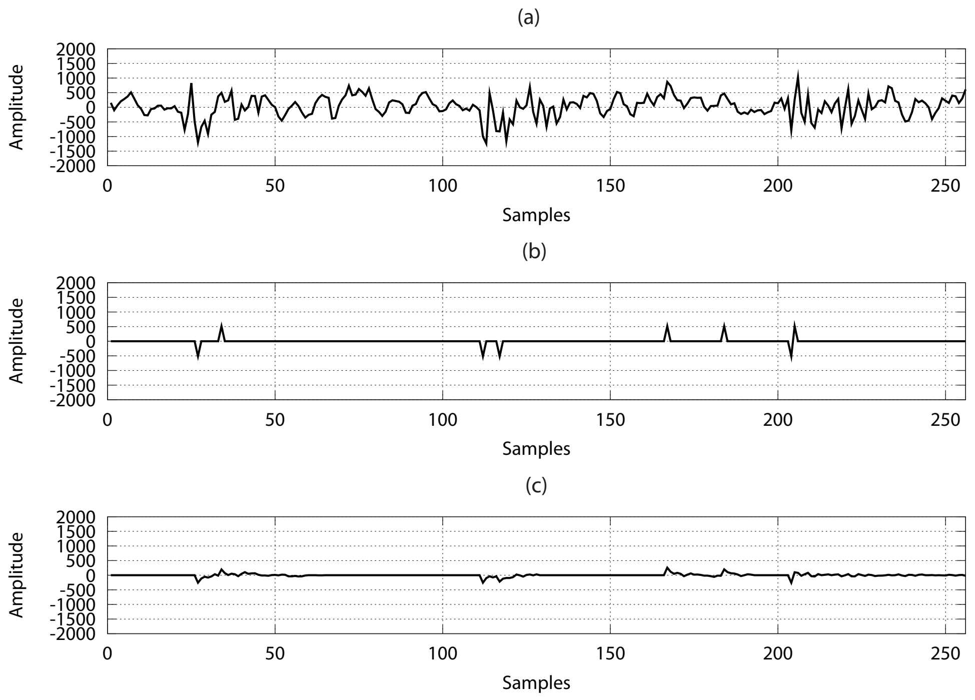


Figure 4.7: Fixed codebook search in mode 0: *a.* Updated target signal *b.* Optimum fixed codebook innovation *c.* Optimum fixed codebook innovation after filtering

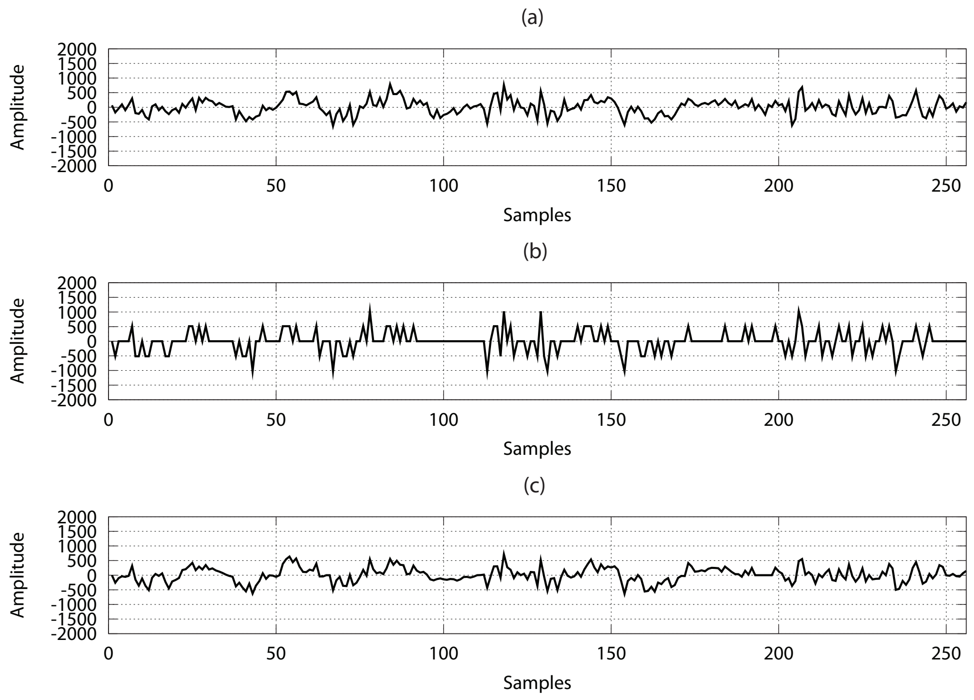


Figure 4.8: Fixed codebook search in mode 8: *a.* Updated target signal *b.* Optimum fixed codebook innovation *c.* Optimum fixed codebook innovation after filtering

4.3 Working Principle of the AMR-WB Decoder

The encoder transmits all indices after the speech signal is encoded in the encoder and at the decoder; those indices are extracted from the received bit stream. The indices are decoded to obtain the codec parameters at each transmission frame. The sequence of parameters for each frame received by the decoder is the ISP vector, the 4 fractional pitch lag, the 4 LTP filtering parameters, the 4 algebraic codebook indices, the 4 set of codebook gain. For the 23.85 kbit/s mode, the high-band gain index is decoded.

The ISP vector is converted to the Linear Prediction (LP) filter coefficients and interpolated to obtain LP filters at each subframe; this is done for the whole frame and then, at each 64-sample:

- Construct the excitation by adding the pitch lag and algebraic codebook vectors scaled by their respective gains.
- Filter the excitation by LP synthesis filter to reconstruct the 12.8 kHz speech signal.
- De-emphasize the reconstructed 12.8 kHz speech signal and then upsample to 16 kHz

Chapter 5

Dataflow Model of the AMR-WB Encoder

The flow model diagram of any design concerns first the input data stream progress and operations applied on that stream throughout the design. Figure 5.1 shows the flow model diagram of the AMR-WB encoder. In this case, the input stream is the audio signal in the form of frames that consist of 320 - 16 bit - PCM audio samples. Each frame starts by the *Preprocessing* stage in which the audio frame is decimated and preemphasized to prepare the frame for next analysis to extract speech parameters. Then, the preemphasized frame flows simultaneously in two separate paths. The 1st path is *LPC*, followed by *OLP*, *LPC2*, *SUBFR_Analysis*, and the 2nd path is *wb-VAD*, followed by *vad_hist*, *tx_dtx_handler*, *dtx*. In the next lines, we will discuss how the preemphasized audio frames - denoted by *Speech* in Figure 5.1 - progress through these two paths and how they relate to each other.

In the 1st path, the *Speech* flows through the *LPC* analysis stage where the immittance spectral pairs *isp/isf* and LPC coeffіcents *Az* parameters are extracted and the Perceptual Weighted Speech *WSP* is generated. Then, the *WSP* and *tone_flag* of the current audio frame flows through the Open Loop Pitch analysis *OLP*, where the open-loop pitch lag *T_op*, *T_op2* parameters are extracted and the *tone_flag* for the next frame computations in the 2nd flow path is generated. This *tone_flag* updating nature creates a problem we will discuss later. Then, the audio frame faces the subframe gate *SUBFR Gate* which passes audio frames depending on the control signal *dtx_mode* gnerated from the *tx_dtx_handler* block in the 2nd path. If the *dtx_mode* is equal to 9, the *SUBFR Gate* consumes the tokens of the current frame and does not allow the frame tokens to propagate through the rest of the

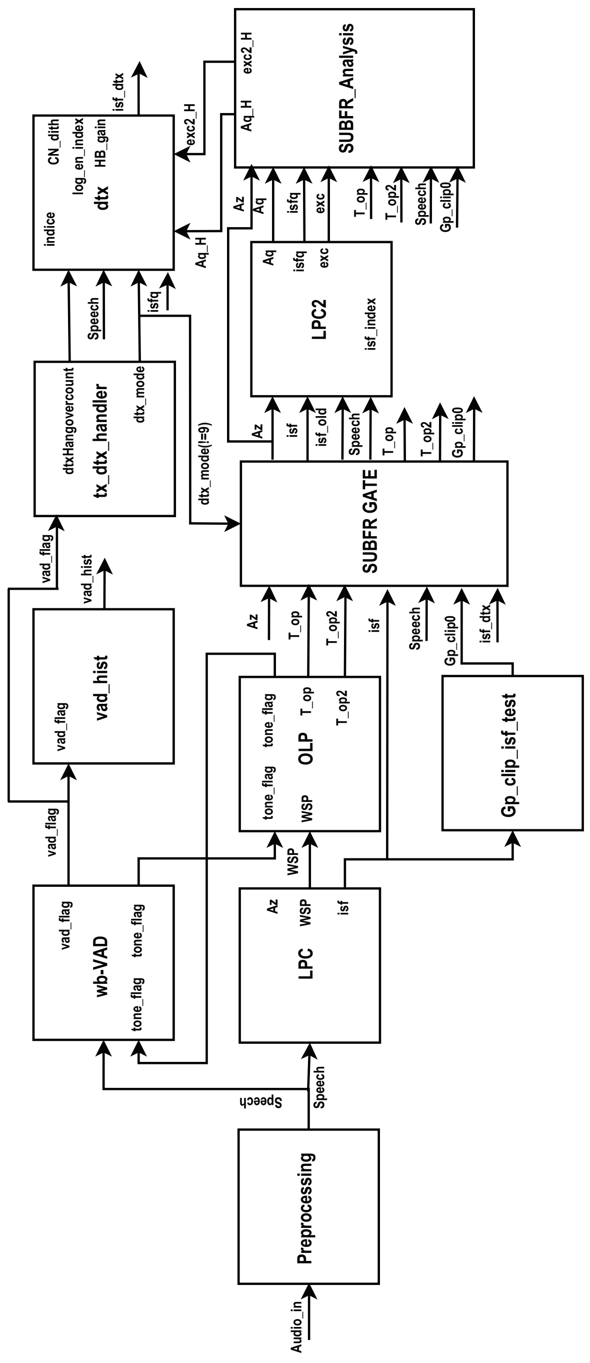


Figure 5.1: Dataflow model of AMR-WB encoder.

analysis. However, if *dtx_mode* is less than 9, that means that the *dtx_mode* value ranges from 0 to 8 - the *SUBFR Gate* allows the current frame tokens to propagate to the next stage *LPC2* and *SUBFR_Analysis*.

In the 2nd path, the *Speech* goes through the voice activity detection stage *wb-VAD* to check if the frame is a speech or a background noise and generates the voice activity detection flag *vad_flag*, then the *tx_dtx_handler* block uses the *vad_flag* to decide whether this frame is voice or background noise by generating the *dtx_mode* control signal. If *dtx_mode* is equal to 9, the discontinuous transmission *dtx* block is enabled to encode the audio frame as a background noise disregarding its original encoding mode and disabling the *SUBFR Gate* preventing the audio frame from further progressing in the 1st flow path as mentioned before. Otherwise, if the *dtx_mode* is in the range between 0 to 8, the *dtx* block is disabled, preventing extra progress on the 2nd flow path and enabling the audio frame to progress on the 1st flow path.

Back again to the 1st flow path. The audio frame, after confirming it is a speech frame, continues through the second stage of the LPC analysis (*LPC2*) or the Quantized LPC Analysis, where the quantized *ispq/isfq* parameters are internally extracted. Then, the *Speech* frame goes through the subframe analysis *SUBFR_Analysis* block where it is divided into four subframes. The closed loop pitch lag, adaptive and fixed codebook parameters, and the pitch and algebraic codebook gains are extracted in sequence for every subframe. The 23.05 kbit/s mode differs from other modes by extracting the higher band gain parameter.

Chapter 6

Mapping from C to CAL

In our case study, the AMR-WB audio codec was given in the form of sequential reference C code. Our goal was to analyse the reference C code with respect to dataflow and expose different kinds of parallelism as well as unwanted data dependencies, and then map it to CAL Actor Language.

In the related work Section 1.1 in Chapter 1, we related two case studies of implementing video codecs in CAL. The first case study [14], was implementing a standard MPEG-4 SP decoder in CAL according to the MPEG RVC formalism. The MPEG RVC framework is a new ISO standard aiming to design a decoder at a higher level of abstraction than the one provided by current generic monolithic C based specifications. Instead of low level C/C++ code, an abstract model based on modular components taken from the standard Video Tool Library (VTL) is the reference specification [9]. This makes their approach totally different from ours, since we start from a reference C code. The second case study [11], was about converting of a reference C code specification of the H.264 encoder into an extended SDF model, using HW/SW codesign environment PeaCE [4]. In that case study [11], they proposed a systematic approach for converting sequential C code to a dataflow specification. That approach deals only with identifying the dataflow model, specifying functional blocks and analyzing global variable dependencies. This is similar to what is discussed in Sections 6.2, 6.3 and 6.4 in this chapter, but it is missing the ways to handle the conversion from C language syntax to CAL language, i.e., handling pointers, loops,...etc.

In this chapter, we propose a systematic approach for mapping from reference C code to CAL Language that has been successfully applied in our AMR-WB implementation.

6.1 Overview

A C program is a set of instructions operating in sequence on a set of data. On the other hand, a CAL application is a set of functional blocks (*actors*) that consume and produce unbounded data vectors (*tokens*).

```
void vec_op (int *x,
             int *y,
             int *Sum,
             int *Mul,
             int len)
{
   for (i = 0; i < len; i++)
   {
      *Sum = add(*x , *y);
      *Mul = mult(*x , *y);
      Sum++; Mul++;
      x++; y++;
   }
}
```

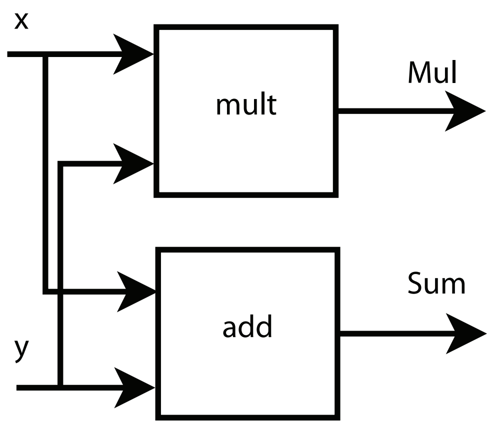


Figure 6.1: Sample C code to the left and its CAL representation to the right.

Figure 6.1 shows a sample C code that represents `add()` and `mult()` operations over two arrays, `x` and `y`, in the function `vec_op()` where the resultant `Sum` array and `Mul` array are generated by iterating in sequence through the `for` loop without explicitly telling the compiler that these two operations can run concurrently. On the other hand, the CAL representation explicitly exposes the possible parallelism which is hidden in the C function between the `add()` and `mult()` functions by representing them as two separate actors, `add` and `mult`, running concurrently, that feed on the input tokens `x` and `y` and generate the resultant output tokens `Sum` and `Mul` arrays.

From the example shown in Figure 6.1, we can notice that the CAL language exploits different kinds of hidden parallelism in conventional programming languages, such as the C language. In the next sections we will discuss a number of guidelines that will help in mapping from C language to CAL Actor language.

6.2 Drawing a Flow Diagram

The first step to your CAL program is to analyze the C program. This analysis must result in a flow model that exposes the flow of input data flowing into the application and various operations applied on that flow. It must also show if there are any other parallel flows that are independent or branched from the main flow.

```
void main (int *insig)
{
   ....
   ....
   preproc(insig, outsig);
   mode=voice_act(outsig);
   LPC_analysis(outsig);
   if (mode == NOISE)
      DTX_CNG(outsig);
   ....
   ....
}
```

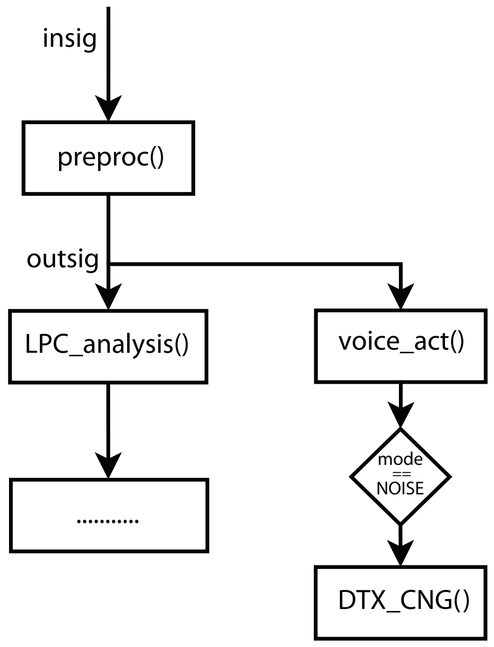


Figure 6.2: Sample C code and its flow diagram.

Figure 6.2 shows us a section of an audio DSP application. That DSP application receives an input stream of an audio signal `insig` that is pre-processed, then checked to establish if it represents a voice signal or just background noise. Then it extracts the LPC coefficients that are used in generating the encoded background noise frame, or otherwise used in the extra analysis of the voice signal. These DSP operations mentioned before are applied throughout the functions `preproc()`, `voice_act()`, `LPC_analysis()` and `DTX_CNG()`. If we track down the input signal throughout those functions we will discover that the application contains a main flow that represents the analysis of the voice signal and, branching from that flow, a secondary flow that works in parallel with the main flow to detect the background noise signal as represented in the flow diagram in Figure 6.2.

From the example shown in Figure 6.2, we notice that the flow diagram gives a very abstract high level overview on the design and what degree of parallelism can be reached from such an application.

6.3 Partitioning

After constructing the Flow Diagram Model, we start to look for a more detailed view of each coarse grain block alone, and begin to build it up from middle grain blocks which we call "partitioning". This partitioning technique helps in finding more possible parallel flows in our design. Even if it is impossible to find any parallel flows, we would still benefit from partitioning, since the partitioning technique will represent the coarse grain block as a multistage pipeline that can execute more than one operation at a time, which leads to an increase in performance. Throughout the partitioning process, we have to consider similar and repetitive operations in our application and use it in partitioning to design unified building blocks that are reusable throughout the application.

```
void LPC_analysis(int *insig)
{
    .....
    r = Autocorr(insig);
    r = Lag_window(r);
    A = Levinson(r);
    .....
}
```

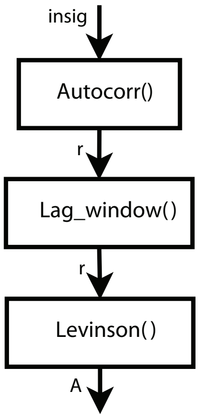


Figure 6.3: LPC_analysis partitioning.

Figure 6.3 shows the `LPC_analysis()` block mentioned in the previous Figure 6.2 in a more detailed view. We notice that it consists of several functions i.e., `Autocorr()`, `Lag_window()` and `Levinson()`. These functions cannot be parallelized since the input of each function depends on the output of the other one e.g., the `Lag_window()` input depends on the output of `Autocorr()`. However, we can still benefit from partitioning them since it will create a multistage pipeline that will increase the performance by a factor equal to the number of stages. Also partitioning will help in increasing the reusability of the design since we can use these actor blocks in other parts of the design.

6.4 Global Variables Handling

Sharing information between different parts of code is very common in various applications. This property - the sharing property - is represented in many ways, like passing parameters by value and by reference between different functions, or by defining global variables, which can be accessed by all entities of the application from different locations and on which various operations can be applied, such as write and read operations.

The use of *global variables* in C programs and in other conventional programming languages is of great benefit for many types of applications, but it also introduces many dependencies between different code pieces. In CAL applications, there are two types of variables, *Local Variables* and *State Variables*. The *Local Variables* are similar to the local variables used in C programs. The *State Variables* have a scope on the level of an actor only and can be accessed by different actions or functions inside the actor. This means that the notion of *global variables* is not supported in CAL language like in C language.

This limitation drives us to find a way to distribute the global variables in a way to ensure synchronization and low communication cost. We have to analyze the behavior of these variables to make the right decision in mapping them across the actor network and, thereafter, some guidelines that control the placement of these variables:

6.4.1 Initialization and Reset Conditions

Global variables may have initial values that are assigned to them during the initialization of a program and, under certain conditions, through the execution of the program (so called "reset conditions"). It is so easy in C or conventional applications to reset and initialize the global variable through a single function that is called whenever the conditions are achieved. In CAL, globals are not an option. Only state and local variables can be used. So, to achieve reset and initialize functionalities in CAL, we have considered two possible solutions:

1. Represent the structure that contains all the global variables in the C program in a single actor that acts as a global memory. This global

memory (actor) can be accessed by different actors in the design whenever they need to write or read a certain variable in a fashion similar to a memory cycle call.

2. Distribute the globals across the actor network in such a way that, if a group of actors share some variable, only one of them has the capability to reset that variable. The actor who is responsible for resetting is the one that is in an early position in the flow with respect to other actors sharing the same variable.

The global memory solution is not the optimal solution since it will create the conventional memory integrity problems that need to use semaphores to control access to the memory. This will lead to a decrease in the degree of parallelism which is the main benefit of using CAL in programming. However, distributing the globals across the actor network solution is much better since it eliminates race conditions that will occur from the global memory solution and also will lead to a more balanced computation load. Thus, we adopted the second solution in our design.

6.4.2 Synchronization Conditions

Shared variables must be handled with care in C and other conventional languages to prevent data hazards and ensure synchronization between different tasks running concurrently. This is done by using mutual exclusion and semaphores. In CAL, there is no such problem due to the fact that there are no global variables, just local and state variables that cannot be accessed by the other actors. This means that semaphores and mutual exclusion techniques are not needed in CAL.

That drives us to the conclusion that the updating manner of global variables - which means who is doing the updating(writing) and who is just reading the value of the variable without update - decides where to locate the variable in the design in order to maintain synchronization. The updating behavior can be classified in the following categories:

- **Read after Write/ Fetch after Updating:** Where the global variable is just updated in one block of code and read in the other parts and, in this situation, we assign the variable to the actor that represents the updating part of the code while the others just read the updates generated from the update actor as shown in Figure 6.4.

- **Write after Write/ Update after Update:** Where the global variable is updated more than one time in different parts of the code, i.e.

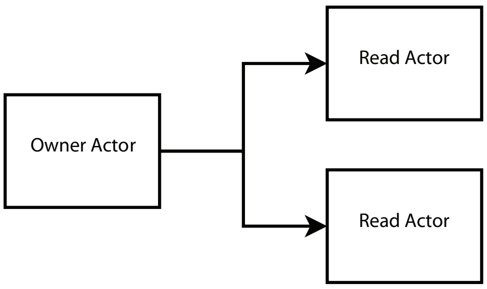


Figure 6.4: Distribution of global variables in read after write case.

in different actors. In this case, to keep correct and synchronized value between different actors, we have to supply the updated value in a feedback loop manner as shown in Figure 6.5.

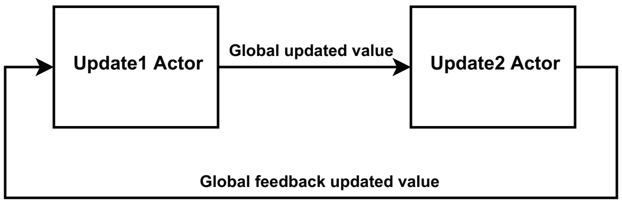


Figure 6.5: Distribution of global variables in write after write case.

6.5 Loops

Loops are used frequently in conventional applications. They are also found in the CAL language and utilize almost the same syntax. In this section, we are introducing the big blocks of code that iterate for certain number of times i.e. `for` loops.

This big loop can be realized as a parallel concurrent architecture that has a number of flows exactly equal to the number of iterations, but only if the next input data stream is not dependent on the current output, otherwise it is represented as a single actor that contains that loop inside it. Figure 6.6 shows an addition over two vectors/arrays operation using a `for` loop. This `for` loop operation can be realized by a number of parallel adders equal to the number of iterations of the `for` loop `n`.

```
int *Sum, *x, *y;
⋮
for (i=0;i<n;i++)
   Sum[i] := x[i] + y[i];
⋮
```

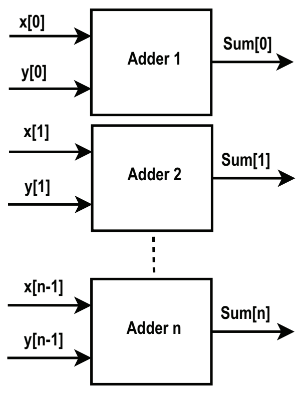


Figure 6.6: Actor representation of loop.

The case is different for conditional loops, i.e. `while` loops. It can not be represented as a parallel concurrent architecture since we can not determine the number of iterations since it only depends on the running condition.

6.6 If and Switch Statements

if and *switch* statements represent the ability to choose between different blocks of instructions according to certain conditions, which is very popular in conventional programming. In the CAL language, the *if* statement utilizes a similar syntax like conventional programming languages but, on the other hand, the *switch* statement is not represented in CAL and can be replaced by using nested *if else* statements. Representing these blocks of code separately - as separate actors - is a waste of space and increases the communication cost, since only one path will be chosen between them and they will never run in parallel.It is better to represent such kinds of statements by encapsulating them in a single actor with multiple guarded actions to select between them, according to the guard condition.

Figure 6.7 shows a sample *if else* C code that represents a simple calculator application `Calc()`, where it contains four operations, i.e. addition, subtraction, multiplication and division. These four operations can not work simultaneously, since they are controlled by the `if else` statement condition `op`, where only one operation can be enabled at a time.

If we consider rewriting the `Calc()` function in CAL, we can not represent it as four concurrent actors since only one actor will run at a time, so it is better to encapsulate it in a single actor and represent these operations as four guarded actions, as shown in Figure 6.8.

```
int Calc(int a,int b,int op){
   if (op == 1)
      return a+b;
   else if (op == 2)
      return a-b;
   else if (op == 3)
      return a*b;
   else
      return a/b;
}
```

Figure 6.7: *if else* sample C code.

```
actor Calc() int a,int b,int op ==> int res:
   int mode := 0;
   action op: [m] ==> do
      mode := m;
   end
   action a: [i1], b: [i2] ==> res: [i1+i2]
   guard mode=1 do
      mode := 0;
   end
   action a: [i1], b: [i2] ==> res: [i1-i2]
   guard mode=2 do
      mode := 0;
   end
   action a: [i1], b: [i2] ==> res: [i1*i2]
   guard mode=3 do
      mode := 0;
   end
   action a: [i1], b: [i2] ==> res: [i1/i2]
   guard mode=4 do
      mode := 0;
   end
end
```

Figure 6.8: *if else* CAL representation.

6.7 Pointers

Pointers are widely used in C applications since they simplify a lot of operations like memory manipulation or in sorting algorithms. However, in CAL, pointers are not supported, so we have to pass to the actor the tokens needed and use indexing instead. Before doing that, we have to analyze the behavior of a pointer, i.e., we have to know if the pointer is moving forward or backward, for example, to determine which tokens are needed exactly.

Chapter 7

Implementation of the AMR-WB Encoder

In this chapter, we will discuss the implementation process of the AMR-WB encoder and the phases that it has gone through to get the final output (which is the AMR-WB encoder represented in CAL). In the implementation process of the AMR-WB encoder, we used the following set of tools and specification manuals:

- OpenDF plug-in for eclipse, which is a simulator for developing and execution of CAL applications [3].
- Eclipse IDE for Java development tool, which is used as a platform for running the OpenDF plug-in and editing CAL applications [2].
- AMR-WB codec fixed point representation specification written in C language.
- AMR-WB 3GPP manuals and specifications [6].

Also, we will give a detailed explanation on the structure of the main functional blocks of the AMR-WB encoder and its operational role.

7.1 Steps of implementation

The implementation of the AMR-WB encoder has gone through four phases. The first phase was the analysis phase where we understood the reference C code and analyzed it with respect to dataflow. The second phase was the development of the libraries that represent the exact instruction set of the target processor. This was followed by designing of the main functional blocks of the AMR-WB encoder, which we considered as the third phase. Finally, the fourth phase that included testing and verification of the whole AMR-WB encoder implementation. The next subsections will give a detailed overview on each phase of the implementation process.

7.1.1 The Analysis and Initial Design

In this phase, we analyzed and acquired knowledge of the AMR-WB encoder algorithm by studying the 3GPP manuals and related literature [7], that helped us so much in tracking the main audio stream in C code application and in partitioning the C code in functional blocks related to the algorithm description. By applying the guidelines mentioned in Chapter 6, i.e. drawing a flow diagram, tracking the main stream and partitioning, we came up by an initial design shown in Figure 7.1.

It can be noticed that it is different from the final design shown in Figure 5.1 because, at the analysis phase, the global variable dependencies were not all clear at that time. In the Section 7.2, we will explain in detail the main blocks of our AMR-WB encoder flow model design described in the Chapter 5.

7.1.2 ETSI Intrinsics Library Developing

The AMR-WB encoder C code fixed point representation was implemented using the ETSI Intrinsics library, which represents the exact instructions used by the target CPU for executing any operation, e.g. addition or subtraction. Also, all fixed point operations in the C application are done using these ETSI Intrinsics. So, to ease the transformation from C representation to CAL, we decided to develop this ETSI Intrinsics library in CAL to use it when needed in our design.

7.1.3 Hierarchical Design and Integration

The implementation of the AMR-WB has been done in a hierarchical structure. Since each block consist of a several actors combined together through different levels of network files, we will describe this design in detail in Section 7.2. Also, the design methodology followed an iterative approach since each block had to be tested before adding it to the whole design.

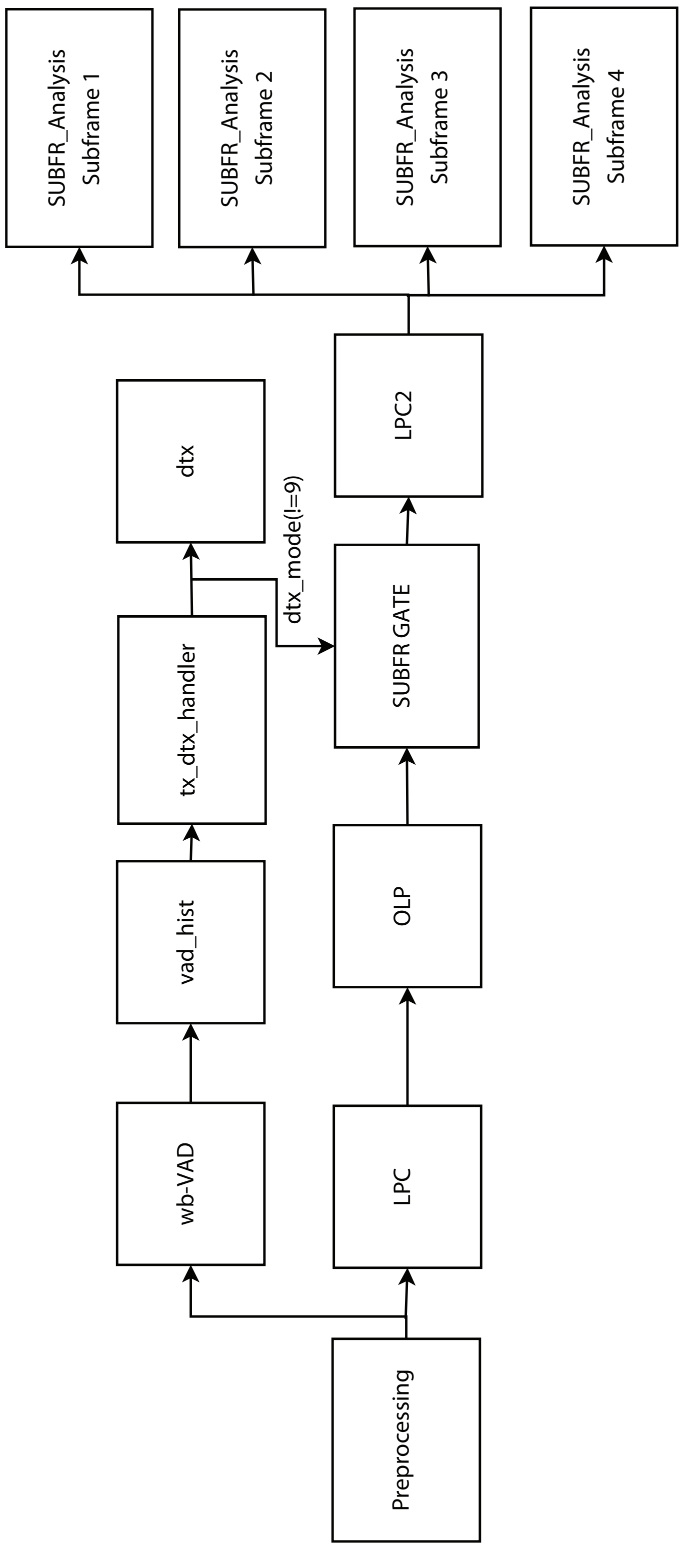

SUBFR_Analysis Subframe 1
SUBFR_Analysis Subframe 2
SUBFR_Analysis Subframe 3
SUBFR_Analysis Subframe 4
dtx
LPC2
dtx_mode(!=9)
tx_dtx_handler
SUBFR GATE
vad_hist
OLP
wb-VAD
LPC
Preprocessing

Figure 7.1: Initial Dataflow Model of AMR-WB Encoder.

7.1.4 Testing and Verification

Testing and verification is done simultaneously with the hierarchical design and integration phase since each component of the design has been tested as a single component (that is what is called "unit testing") and after integration and the output was verified with C code output.

7.2 Detailed Explanation of the AMR-WB Encoder Flow Model

7.2.1 Preprocessing

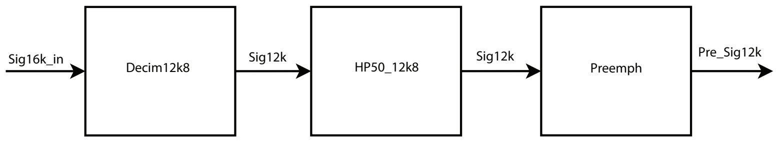


Figure 7.2: Preprocessing actor representation.

The preprocessing is the 1st functional stage in the AMR-WB encoder; it is considered as a preparation stage that prepares the speech signal for further processing throughout the entire encoder. Figure 7.2 shows the input speech signal, *Sig16K_in*, to the encoder in the form of 16KHz PCM 16 bit 320 samples speech frames. The 1st operation applied on the input speech frame is the decimation *Decim12K8*, where upsampling by 4, then by filtering the output through lowpass FIR filter that has the cut off frequency at 6.4 kHz. Then, the signal is downsampled by 5 and we get the decimated output 12.8KHz signal *Sig12K*. Next *Sig12K* is highpass filtered through *HP50_12K8* stage to remove the undesired low frequency components and, finally, the speech frame goes through a 1st order highpass filter *Preemph* to emphasize high frequency components in an operation called "Preemphasis" resulting in a pre-emphasized speech frame *Pre_Sig12K*.

7.2.2 LPC Analysis

As Shown in Figure 7.3, the LPC analysis starts by the auto-correlation *Autocorr* stage, where the pre-emphasized output speech frame *Pre_Sig12K*

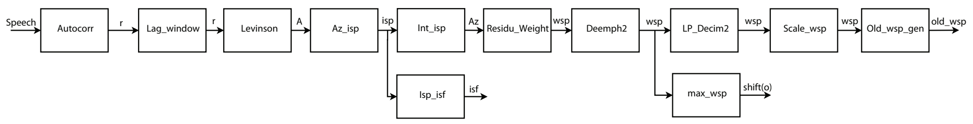


Figure 7.3: LPC Analysis actor representation.

from preprocessing is auto-correlated and lag windowed throughout *Autocorr* and *Lag_window* stages, generating the auto-correlation coefficients *r*, which are used in the Levinson Durbin algorithm *Levinson* stage to get the Linear Prediction Coefficients (LPC) *A* of size 17 word. This LPC coefficient vector *A* is used to generate the Imittance Spectral Pairs *ISP* through the *Az_isp* stage, then interpolated using *Int_isp* to get a 68 word LPC *Az* (17 word for each subframe). At the same time, the *ISP* is transformed to the frequency domain to get the *ISF*.

Finally, the LPC analysis stage ends with generating the Perceptual Weighted Speech *WSP* from the LPC *Az* through the residual weighting filter *Residu_Weight* stage followed by the deemphasis and decimation stages *Deemph2, LP_Decim2* to prepare the *WSP* for the Open Loop Pitch analysis *OLP*.

7.2.3 VAD Analysis

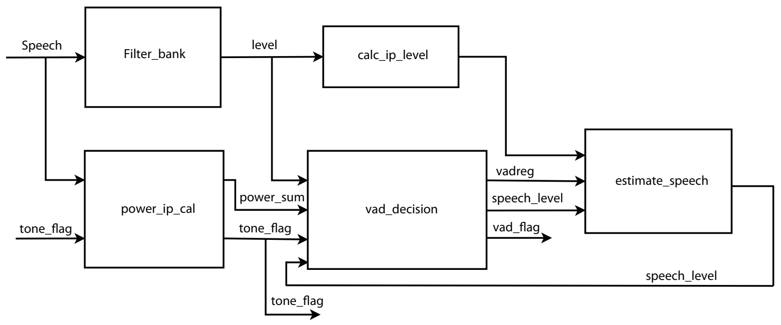


Figure 7.4: VAD Analysis actor representation.

The purpose of the Voice Activity Detection (VAD) stage shown in Figure 7.4, is to determine whether the speech frame represents a signaling tone, voice or background noise frame. This is achieved by using the parameters of the speech encoder to compute the Boolean VAD flag *VAD_flag*. The *VAD_flag* is computed by dividing the pre-emphasized input speech frame into 12 frequency sub-bands, where the signal level is calculated for each sub-band *level*.

The *tone_flag*, calculated from the normalized open-loop pitch gains, which are calculated by open-loop pitch analysis of the speech encoder, which indicates presence of a signaling tone, voiced speech, or other strongly periodic signal, is supplied to the *power_ip_cal* in addition to the pre-emphasized input speech frame to calculate its power, *power_sum*. If the *power_sum* is less than the power threshold, the *tone_flag* is cleared, otherwise we must keep the old value, then supply the new value of the *tone_flag* to the *vad_decision* and the open loop pitch *OLP* analysis stage, which uses the *tone_flag* value for the next frame computations. This passing of the *tone_flag* value between the *VAD* stage and the *OLP* stage creates a closed feedback loop between the two main flows in the design. This feedback loop will affect the parallelism between the two main flows and the pipelining in each flow, since the *OLP* stage will wait for the updated *tone_flag* value from the *VAD* stage and, in sequence, the *VAD* stage will wait for the updated value of *tone_flag* from the *OLP* stage, which means that they can not work simultaneously and bubbles will be created in the pipeline in each flow due to waiting durations.

After calculating the sub-band levels, the background noise level is estimated in each band, based on the *tone_flag*, the power of the speech frame *power_sum* and the previous frame estimated *speech_level* generating the *vad_flag* and the updated value of *speech_level*, according to the signal to noise ratio. If the *speech_level* is lower than a minimum threshold, so it is increased, otherwise keeps the old value. Finally, the *estimate_speech* calculates the *speech_level* of the current frame, according to the sum of the input sub-band levels generated from *calc_ip_level* stage. We can also notice the *speech_level* feedback loop between the *vad_decision* and *estimate_speech* stages, which will result in bubbles in the pipeline and decrease the performance.

7.2.4 Discontinuous Transmission (DTX)

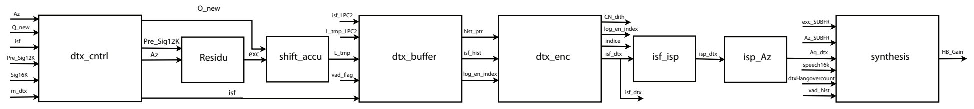


Figure 7.5: DTX actor representation.

The Discontinuous Transmission (DTX) shown in Figure 7.5 is not executed continuously; it only executes if a background noise frame is detected by the VAD stage. To represent this mode of execution, the DTX stage is controlled by a gate at the beginning called *dtx_cntrl*, where its duty is to prevent normal voice or signaling tone frames from proceeding further into the DTX and allow only background noise frames to proceed. This gate is controlled by the control signal, *m_dtx*. If its value is equal to 9, it means that the current frame is a background noise frame and the gate allows it to proceed, otherwise it consumes the frame tokens and prevents the frame from going further.

In the case of a background noise frame, the residual signal, *exc*, is calculated from the 4th subframe LPC coefficient, *Az*, and the pre-emphasized speech frame, *Pre_Sig12K*. Then, the residual signal, *exc*, is shifted and accumulated to get the *L_tmp* value, that is used in the *dtx_buffer*, in addition to, the *ISF* and the *vad_flag*, to buffer the *ISF* and the frame energy in the *isf_hist* and the *log_en_hist* parameters respectively. These two parameters, the *isf_hist* and the *log_en_hist*, are used by the *dtx_enc* to encode the background noise frame and, extract its feature parameters, which are, dithering control, *CN_dith*, frame energy index, *log_en_index* and, the *ISF* index, *indice*. Finally, the *ISF* is converted back to the *ISP* to get the quantized LPC coefficients in background noise mode *Aq_dtx*, which is used with 16 KHz speech to get the high band gain from *synthesis*.

As mentioned before, the DTX stage remains idle at non-background noise frames, but there are two stages that execute even in normal modes, which are *dtx_buffer* and *synthesis*. These two stages share some global variables that need to be updated at both background noise and normal modes and their working conditions are mutually exclusive. This means that they either serve at background noise mode or normal modes non-simultaneously. This working behavior impelled us to build them in a way that makes them work in both situations to share these variables locally, which decreases the communication overheads and space.

The *dtx_buffer* stage runs in normal modes when the *vad_flag* is equal to 0, which indicates signaling tone frames not speech frames. In this condition, the LPC2 stage sends the *isf_LPC2* and the accumulated quantized residual signal value *L_tmp_LPC2* to the *dtx_buffer* to buffer the *ISF* and frame energy. On the other hand, the *synthesis* stage runs only in normal mode at the highest quality mode (mode 8) where it computes the high band gain, *HB_gain*, that is included in the generated encoded frame at that mode.

7.2.5 LPC2 Analysis

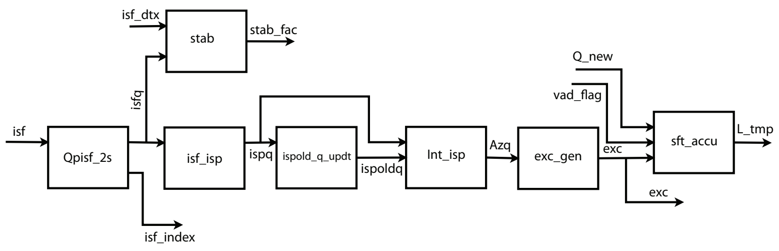


Figure 7.6: LPC2 Analysis actor representation.

The LPC2 analysis shown in Figure 7.6 represents the 2nd phase of the LPC analysis or the Quantized LPC analysis. In that stage, the unquantized *isf* pairs quantized through the *Qpisf_2s* stage generating *isfq* and extracting the *isf_index* parameter which is considered one of the components of the encoded output frame. Then, the *isfq* transformed to the Quantized Imittance Spectral pairs, *ispq*, where it is interpolated to get the Quantized LPC coefficients, *Azq*. These quantized LPC coefficients, *Azq*, are used in generating the quantized residual signal, *exc*, which is sent to the *dtx_buffer* in the DTX stage in the form of the accumulated quantized residual signal value *L_tmp* to buffer the *isfq* and energy of the signaling tone frame as discussed in Section 7.2.4. Also, the quantized residual signal, *exc*, is used to generate the target signal in subframe analysis.

7.2.6 Subframe Analysis

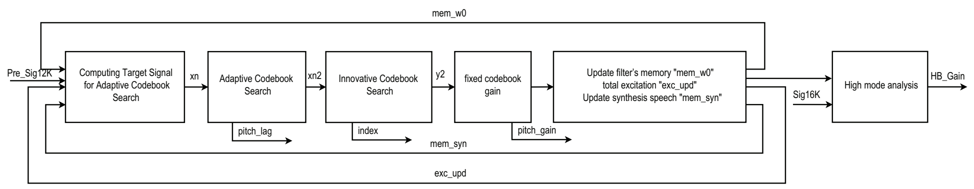

mem_w0
Pre_Sig12K
Computing Target Signal for Adaptive Codebook Search
xn
Adaptive Codebook Search
xn2
Innovative Codebook Search
y2
fixed codebook gain
Update filter's memory "mem_w0" total excitation "exc_upd" Update synthesis speech "mem_syn"
Sig16K
High mode analysis
HB_Gain
pitch_lag
index
pitch_gain
mem_syn
exc_upd

Figure 7.7: Subframe Analysis actor representation.

The Subframe analysis stage shown in Figure 7.7 starts by dividing the pre-emphasized speech frame *Pre_Sig12K* into four subframes. Each subframe goes into defined sequence of operations, starting by preparing the target signal *xn*, that proceeds through the *Adaptive Codebook Search* to extract the closed loop pitch lag *pitch_lag* parameter to be encoded. Then, the remaining updated target signal *xn2* is used to extract the fixed codebook innovation index, *index*, after applying the *Innovative Codebook Search* stage. Finally, the fixed codebook gain index *gain_pitch* parameter is extracted, followed by the updating stage that is responsible for :

- updating the filter's memory *mem_w0* responsible for finding the target vector in the next subframe.
- finding the total excitation and calculate the updated *exc_upd*.
- updating the filter's memory synthesis speech *mem_syn* used in calculating the target signal *xn* for the next subframe.

These three updated parameters *mem_w0*, *exc_upd* and *mem_syn* are fed back to the begining of the subframe analysis process to repeat and apply the same operations on the next subframe. At this point, the problem appears, the feedback loop nature prevents the subframe analysis from being pipelined (subframe to frame dependency) since the updated parameters of the 4th subframe are used for the calculation's next new frame. It also prevents parallelizing the subframe analysis stage by instantiating a stage for each subframe, as the initial design in Figure 7.1, due to each subframe being dependent on the updated values of the previous subframe (subframe to subframe dependancy).

One more operation left in the subframe analysis is the *High mode analysis*, where the highband gain index, *HB_Gain* parameter, is extracted at the highest quality mode only, *Mode 8*. This high mode analysis is not completely done in this stage; the final part of it is done in the *synthesis* stage in the DTX part, since *synthesis* stage - as we mentioned in Section 7.2.4 - shares the operation both on DTX and subframe analysis.

Chapter 8

Results

The AMR-WB Encoder flow model shown in Figure 5, and its implementation described in Chapter 7, exploited two main features - the parallelism and pipelining - to enhance the performance compared to the C code representation designed for single core processor. The parallelism is represented in the two main flows, *LPC - SUBFR_Analysis* and *wb-VAD - dtx*, that run simultaneously, while the pipelining is represented in the design through the multiple stages that each flow consists of, and which allow the encoder to handle more than one frame at a time when the pipeline is full.

The design utilizes the exposed parallelism, but the performance gain is limited due to frame and subframe dependences, as described in Chapter 7. In this chapter, we will discuss proposed solutions for some of these problems. Also, we will discuss the processor core allocation problem.

8.1 Problems and Solutions

8.1.1 tone_flag Frame Dependency

The *tone_flag*, as we mentioned in Section 7.2.3, is a parameter that indicates the presence of a signaling tone, high voiced speech or other strongly periodic signal. The *tone_flag* loops between the *OLP* stage and the *wb-VAD*, the *wb-VAD* provides the *tone_flag* value of the current frame *n* to the *OLP*, and the *OLP* provides the value of the *tone_flag* for the next frame *n+1* to the *wb-VAD*, and so on. This looping manner of the *tone_flag* will lead to a non continuous flow on both branches of the design, since each branch waits for the updated value from the other.

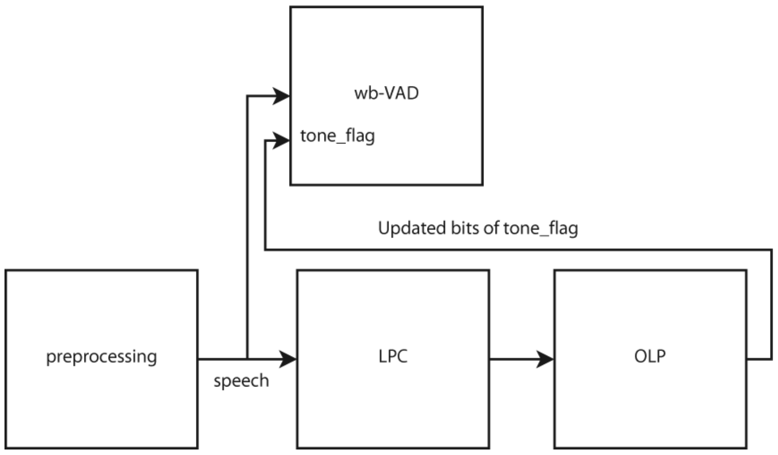


Figure 8.1: *tone_flag* frame dependency solution.

To eliminate that dependency, we re-analyzed the *wb-VAD* and *OLP* and we found that the *wb-VAD* accesses the *tone_flag* by read and write operations, but the *OLP* accesses the *tone_flag* by write operation only, which means that the *OLP* does not use the *tone_flag* variable in any computations, it just updates it if the speech frame contains high gain tones by setting few certain bits in the *tone_flag*. So, instead of providing the *tone_flag* value from the *wb-VAD* to the *OLP*, we can eliminate this dependency and just send the updated bits values to the *wb-VAD* to update the *tone_flag*. By this, we resolved the dependency of the main stream flow (*LPC* - *SUBFR_Analysis*) on the secondary flow (*wb-VAD* - *dtx*), which will enhance the performance of the overall design. Figure 8.1 shows the updated design.

8.1.2 Subframe Analysis Parallelization

The subframe analysis stage from the preliminary analysis appears to be parallelized into four stages that can run concurrently. Since the speech frame is split into four subframes, each subframe is processed to extract the speech parameters that will be encoded. However, the reality is totally the reverse, the subframe analysis stage can not be parallelized due to the subframe dependencies discussed in Section 7.2.6. These dependencies resulted from the nature of the audio signal processing and the predictive nature of the AMR-WB algorithm that is adopted from Algebraic Code Excited Linear Prediction (ACELP) that depend on the previous frames to adjust its filters for the next frames.

8.2 Core Allocation for the AMR-WB Encoder

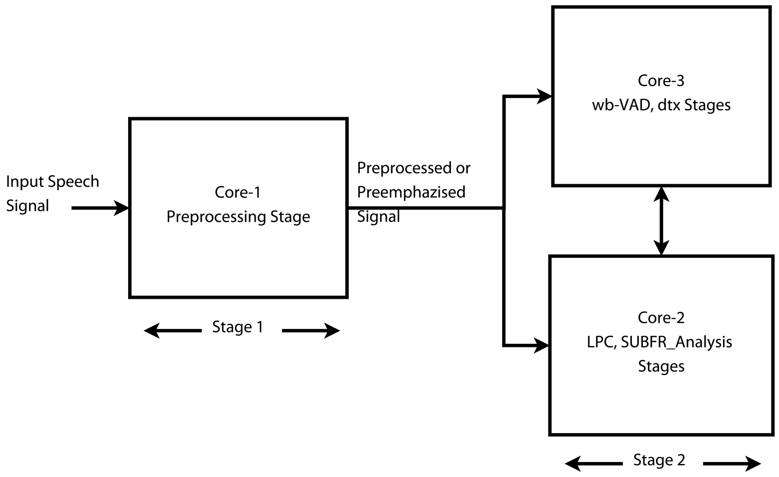


Figure 8.2: AMR-WB core allocation for target processor ARM11 MPs.

The AMR-WB encoder can be mapped on a processor with up to eight cores. Since we have seven encoder stages (*preprocessing, LPC, OLP, LPC2, SUBFR_Analysis, wb-VAD, dtx*), each one of them could be assigned its own core. The eighth core can be reserved for the decoder. Since our target processor (ARM11 MPs) is a four core processor, we propose the core allocation shown in Figure 8.2. Three cores are allocated for the encoder and the fourth is reserved to the decoder. From Figure 8.2 we notice that Core-1 is allocated exclusively for the preprocessing stage. The result is forwarded to Core-2 and Core-3. Core-2 and Core-3 process the two major parallel flows in our design. Core-2 has been allocated for*LPC, OLP, LPC2, SUBFR_Analysis* stages, and Core-3 has been allocated for *wb-VAD* and *dtx* stages. Some of these stages are running under all conditions, e.g. *LPC, OLP* and *wb-VAD*, and the remaining stages, *LPC2*, the *SUBFR_Analysis* and *dtx*, run in a switching manner, which means that, when the *LPC2, SUBFR_Analysis* stages are running, the *dtx* is not running, and vice versa. The intercommunication among those two cores (Core-2 and Core-3) and Core-1 is balanced since both Core-2 and Core-3 receive the same inputs (pre-processed signal) from Core-1. The intercommunication between Core-2 and Core-3 is very light compared to the communication between Core-1 and either Core-2 or Core-3, since this communication link only works at normal speech frames in the high quality mode (Mode 8), excluding other modes and background noise frames. The load balance between Core-2 and Core-3 is not even, since the computation complexity of Core-2 workload is larger compared to Core-3. However, if we tried to follow the balanced load on both cores, e.g., to move any of the stages on Core-2 to Core-3, the communication overhead would increase, which would have a serious negative effect on the overall performance.

Chapter 9

Conclusions

This thesis presents an example for converting a single-processor stream application to multi-core environment by using a dataflow programming approach. In our work, we started by analysing the C Language AMR-WB encoder specification. The result is a flow model that shows two main dataflows in the encoder that allow parallelism, which means less execution time for each audio frame. Also, the flow model defined a number of stages in each flow that allow pipelining. This means that more than one frame can be handled at a time when the pipeline is full. The parallelism and pipelining features that have been exposed are essential to get the best performance from any multi-core environment.

The specification of the AMR-WB codec was in C-code. So, in this thesis work, we also provided a systematic approach that we used to map from C-code to dataflow CAL Actor language. That approach concerns how to investigate different flows of data across an application, how to handle and distribute global variables among different actors in your design, how to manage pointers,... etc. Those specified strategies will be very helpful for future work in conversion from C to dataflow language CAL.

Throughout our analysis and implementation, we exposed unwanted dependences - frame and subframe dependences - due to the predictive nature of the AMR-WB algorithm. We proposed solutions for some of these problems, and the rest of the dependences may need modifications on the encoder algorithm to suit multi-core environment.

Our implemented encoder in CAL shows bit exact result with existing encoder (written in C). Since we use the parallelism and pipelining features that are missing in the single core representation we expect that the encoder CAL implementation will provide better performance in multi-core environments compared to the C implementation.

Finally, there is much work that can be done in the future for testing and enhancing the performance of AMR-WB codec for multi-core. With regard to testing, the CAL representation of AMR-WB encoder needs to be compiled and tested on the target multi-core processor (ARM11 MPs) and record all the performance results and compare them to single processor performance. Also, the decoder part of the AMR-WB audio codec needs to be implemented in CAL and tested on the target processor. With regard to enhancing performance, the AMR-WB codec needs to be reanalyzed and modified on the level of the algorithm, not the C reference specification, to eliminate further unwanted dependences discovered in the C reference specification and which can not be solved by applying conventional techniques.

Bibliography

[1] *Actors Project, Adaptivity and Control of Resources in Embedded Systems*, homepage http://www.actors-project.eu/, checked on May 2010.

[2] *Eclipse IDE for Java EE Developers*, homepage http://www.eclipse.org/, checked on May 2010.

[3] *OpenDF plug-in for Eclipse IDE for Java Development*, homepage http://opendf.sourceforge.net/eclipse , checked on May 2010.

[4] *PeaCE (Ptolemy extension as Codesign Environment)*, homepage http://peace.snu.ac.kr/research/peace/.

[5] 3rd Generation Partnership Project (3GPP), *3GPP TS 26.190 Adaptive Multi-Rate Wideband (AMR-WB) speech codec; Transcoding functions*, v8.0.0 ed., December 2008, homepage http://www.3gpp.org/ftp/Specs/html-info/26-series.htm, checked on May 2010.

[6] ——, *Adaptive Multi-Rate Wideband (AMR-WB) speech codec 3GPP Manuals*, v8.0.0 ed., December 2008, homepage http://www.3gpp.org/ftp/Specs/html-info/26-series.htm, checked on May 2010.

[7] B. Bessette, R. Salami, R. Lefebvre, M. Jelinek, J. Rotola-Pukkila, J. Vainio, H. Mikkola, and K. Jarvinen, "The adaptive multirate wideband speech codec (amr-wb)," *Speech and Audio Processing, IEEE Transactions on*, vol. 10, no. 8, pp. 620–636, Nov 2002.

[8] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. von Platen, M. Mattavelli, and M. Raulet, "Opendf: a dataflow toolset for reconfigurable hardware and multicore systems," *SIGARCH Comput. Archit. News*, vol. 36, no. 5, pp. 29–35, 2008.

[9] S. S. Bhattacharyya, J. Eker, J. Janneck, C. Lucarz, M. Mattavelli, and M. Raulet, "Overview of the MPEG Reconfigurable Video Coding Framework," *Journal of Signal Processing Systems*, 2009.

[10] J. Eker and J. Janneck, "CAL language report," Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley California, Berkeley, CA 94720, USA, Technical Memorandum, December 2003, http://embedded.eecs.berkeley.edu/caltrop/docs/LanguageReport/.

[11] H. Hwang, T. Oh, H. Jung, and S. Ha, "Conversion of reference c code to dataflow model h.264 encoder case study," in *Design Automation, 2006. Asia and South Pacific Conference on*, jan. 2006, p. 6 pp.

[12] C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, Y. Zhao, and H. Zheng, "Overview of the ptolemy project," Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley California, Berkeley, CA 94720, USA, Technical Memorandum UCB/ERL M03/25, July 2003, http://ptolemy.eecs.berkeley.edu/.

[13] J. W. Janneck, *A gentle introduction to dataflow programming*, 1st ed., Programming Solutions Group Xilinx, March 2007, http://www.opendf.org.

[14] J. Janneck, I. Miller, D. Parlour, G. Roquier, M. Wipliez, and M. Raulet, "Synthesizing hardware from dataflow programs: An mpeg-4 simple profile decoder case study," in *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, Oct. 2008, pp. 287–292.

[15] G. Kahn, "The semantics of a simple language for parallel programming," in *Information Processing '74: Proceedings of the IFIP Congress*, J. L. Rosenfeld, Ed. New York, NY: North-Holland, 1974, pp. 471–475.

[16] E. A. Lee and D. G. Messerschmitt, "Synchronous dataflow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, September 1978.

[17] E. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, May 2006.

[18] E. Lee and T. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, May 1995.

[19] P. J. Mercurio, “Visualization tools,” University of New Mexicos Khoros, Technical Memorandum, April 1992.

[20] V. Pankratius, A. Jannesari, and W. Tichy, “Parallelizing bzip2: A case study in multicore software engineering,” *Software, IEEE*, vol. 26, no. 6, pp. 70 –77, nov.-dec. 2009.

[21] A. Sodan, J. Machina, A. Deshmeh, K. Macnaughton, and B. Esbaugh, “Parallelism via multithreaded and multicore cpus,” *Computer*, vol. 43, no. 3, pp. 24 –32, march 2010.