Virtual Radio Engine: A New Programming Environment for Software Defined Radio

Riyadh Hossain

Dissertation zur Erlangung des akademischen Grades eines Doktors der Ingenieurwissenschaften (Dr.-Ing.)

Gutachter: (1) Prof. Dr. Claudia Fohry, (2) Prof. Dr. Axel Hunger

Tag der Disputation: December 11, 2008

Fachbereich Elektrotechnik / Informatik

Universität Kassel

Zusammenfassung

Software Defined Radio (SDR) bezeichnet eine neue Art, drahtlose Kommunikationsgeräte zu entwickeln. Zentral ist hierbei die Nutzung von Softwaremodulen zur Steuerung der Radio-Funktionalitäten wie Modulation und Demodulation, Signalerzeugung, Kodierung sowie Generierung der Verbindungsschicht. Im Gegensatz zur herkömmlichen Bauweise drahtloser Geräte, welche die Radio-Funktionalitäten hauptsächlich als Hardware realisiert, stützt sich SDR auf die Programmierbarkeit und Konfigurierbarkeit der zugrundeliegenden Hardwaremodule. Dies erlaubt beispielsweise die Anpassung eines bestehenden Gerätes an ein neues Kommunikationsprotokoll wie z.B. IEEE 802.11 WLAN durch einfaches Softwareupdate.

Ein Kommunikationsprotokoll umfasst typischerweise eine physikalische Übertragungsschicht und eine Kontrollschicht (media access control, MAC). Die physikalische Schicht führt dabei Signalverarbeitungsoperationen (z.B. Modulation und Demodulation) durch, um Bitströme in MAC-Pakete zu wandeln (empfängerseitig) bzw. MAC-Pakete in Bitströme (senderseitig). Die MAC-Schicht erledigt die zur Kommunikation zwischen verschiedenen Netzwerkknoten notwendigen Adressierungs- und Zugriffskontrollfunktionen.

Im Allgemeinen unterliegen Kommunikationsprotokolle bestimmten Echtzeit Bedingungen, beispielsweise muß das IEEE 802.11a WLAN-Protokoll Daten mit einer Geschwindigkeit von 54 Mbit/s verarbeiten. Zusätzlich sind die benutzten Algorithmen der physikalischen Schicht (z.B. Viterbi-Algorithmus) sehr rechenintensiv. Daher ist eine hohe Verarbeitungsgeschwindigkeit eine der Hauptanforderungen für SDR-Hardwareplattformen. Weil batteriegetriebene SDR-Hardware strengen Stromverbrauchsgrenzen unterliegt, kann diese erforderliche Verarbeitungsgeschwindigkeit nicht durch Erhöhung der internen Taktrate erreicht werden. Stattdessen setzen diese Architekturen auf Parallelisierung, wenden also mehrere Arithmetik- und Signalverarbeitungseinheiten gleichzeitig an.

Die Entwicklung von Programmen für parallele Architekturen ist schwieriger als die Entwicklung von serieller Software, da Entwickler hierbei mit zusätzlichen Aufgaben konfrontiert sind: Identifizierung paralleler Vorgänge (Tasks), Berechnung von Mapping (Zuweisung von Tasks auf die verfügbaren Verarbeitungseinheiten) und Scheduling (Erstellung der Ausführungsreihenfolge der Tasks) sowie Einbau von Synchonisierungen, um die gewünschte Ausführungsreihenfolge über die verschiedenen Verarbeitungseinheiten hinweg zu gewährleisten.

Sowohl Mapping als auch Scheduling sind hardwarespezifisch. Als Beispiel kann eine Applikation angenommen werden, die aus vier parallelisierbaren Tasks besteht.

Diese Applikation soll auf zwei unterschiedlichen Hardwareplattformen ausgeführt werden; eine Plattform erlaubt die parallele Ausführung aller vier Tasks, die andere Plattform erlaubt lediglich zwei gleichzeitige Tasks. Zusätzlich können die Funktionen der Applikation hardwarespezifisch sein. Während z.B. die SDR-Plattform SB3010 POSIX-Funktionen zur Tasksynchronisierung nutzt, kommen auf der Hardware Adelante VD3204x EVP-spezifische Funktionen zum Einsatz.

Derzeit findet die Entwicklung von SDR-Applikationen mit plattformspezifischen Toolchains statt. Jede Hardwareplattform bestimmt also ihre jeweilige Entwicklungsumgebung. SB3010 beispielsweise wird mit der Sandblaster-Entwicklungsumgebung (Sandblaster IDE) ausgeliefert. Diese nutzt POSIX-C als Applikations-Beschreibungssprache. Hinzu kommt ein optimierender POSIX-C Compiler, Debugger und Profiler. Durch Kenntnis der Zielhardware kann dieser Compiler effiziente Executables erzeugen.

Das aktuelle Vorgehen bei der Entwicklung von SDR-Applikationen ist hardwareabhängig, d.h. Entwickler beschreiben eine Applikation (ein Kommunikationsprotokoll) in einer plattformspezifischen Sprache wie z.B. POSIX-C, einschließlich des Mappings, Schedulings und der Synchonisierungen. Folglich ist die Programmierung komplex, erfordert vorherige Kenntnis der Zielhardware und kann nicht parallel mit der Hardwareentwicklung durchgeführt werden, darüberhinaus sind die Applikationsbeschreibungen nicht portabel.

Um diese Komplexität zu mindern, ist ein Programmierkonzept wünschenswert, welches die Beschreibung einer Applikation von ihrer Implementierung auf einer spezifischen Hardware trennt, also ein Konzept, das es dem Entwickler erlaubt, die Applikation hardwareunabhängig zu beschreiben und dann mithilfe eines Compilers in ausführbaren Code umzusetzen. Leider wird ein solches Programmierkonzept bislang von keiner heutigen Programmierumgebung unterstützt.

Diese Arbeit führt ein derartiges Programmierkonzept unter dem Namen Virtual Radio Engine (VRE) ein und beschreibt den Prototyp einer Programmierumgebung, die vom Autor zur Überprüfung des Konzepts implementiert wurde. Es ist zu beachten, daß der Schwerpunkt auf der Entwicklung von VRE zur Auslegung und Implementatierung der physikalischen Schichten von SDR-Applikationen liegt.

Bei VRE besteht die Entwicklung einer Applikation aus zwei Schritten. Zunächst wird die Applikation in hardwareunabhängiger Form spezifiziert. Im zweiten Schritt erfolgt die hardwarespezifische Implementierung (größtenteils) automatisch durch eine Toolchain. VRE definiert eine eigene Sprache, die die Beschreibung von Applikationen (z.B. für WLAN) in plattformunabhängiger Weise erlaubt. Eine derartige Beschreibung wird bei VRE als plattformunabhängiges Modell (PIM) bezeichnet.

Ferner definiert VRE einen Compiler, bestehend aus einem Compiler-Kern und einem Code-Generator. Der Compiler-Kern transformiert ein PIM in eine weitere Ebene der Programmbeschreibung namens PSM = plattformspezifisches Modell. Wie das PIM wird das PSM in der VRE-eigenen Sprache dargestellt. Im Unterschied zum PIM enthält das PSM aber das hardwarespezifische Mapping und Scheduling sowie die Synchonisierungen. Dies wird vom Compiler aus dem PIM sowie aus weiteren notwendigen Dateien erzeugt, welche die hardwarebezogenen Informationen enthalten.

Aus dem PSM erzeugt der Code-Generator ein plattformspezifisches Quellcode-Prog-

ramm (PSSP). Dieser Quellcode liegt in der plattformspezifischen Sprache (z.B. POSIX-C) vor, zur weiteren Verarbeitung durch den plattformspezifischen Compiler (z.B. POSIX-C-Compiler). Die hieraus hervorgehenden Executables sind nun auf der Zielhardware lauffähig.

Die VRE-Toolchain läuft halbautomatisch ab. Dies bedeutet, daß Entwickler die Executables aus einem PIM Schritt für Schritt automatisch erzeugen, den Code nach jedem Schritt aber auch manuell optimieren können.

Diese Arbeit trägt mit fünf Schwerpunkten zur Entwicklung von VRE bei: Zu allererst wurde die VRE-Sprache entwickelt mit dem Ziel, sowohl PIM als auch PSM zu genügen. In einem PIM beschreiben Entwickler keine Hardwarespezifika, sondern Einschränkungen des Schedulings, d.h. Abhängigkeiten zwischen Tasks. Produziert ein Task T1 Daten, die von einem anderen Task T2 konsumiert werden, muss T1 vor T2 ausgeführt werden. Im Detail existieren drei Abhängigkeitstypen: Nachrichtengekoppelte, speichergekoppelte und kontrollgekoppelte Abhängigkeiten. Nachrichtengekoppelte Abhängigkeiten entsprechend einem direkten Datenfluß zwischen Blocks; ein Task ist ausführbereit, wartet aber auf Input aus einem anderen Task. Speichergekoppelte Abhängigkeiten entstehen bei mehreren Tasks, die auf den gleichen Speicherbereich zugreifen, sobald mindestens ein Zugriff davon schreibend ist. In Schleifen und bei Verzweigungen entstehen kontrollgekoppelte Abhängigkeiten. Diese bestimmen das Ablaufverhalten, z.B. eine Schleife die iterative Ausführung von Tasks.

Das PSM-Format erlaubt es Entwicklern, ein Programm mit weniger Aufwand zu verstehen und zu optimieren, als dies in einer plattformspezifischen Sprache der Fall wäre.

Als zweiter Schwerpunkt wurde Mathwork's Simulink-Tool in die VRE-Toolchain integriert, um PIMs beschreiben zu können. Im Rahmen dieser Arbeit wurden verschiedene Workarounds aufgezeigt, die es erlauben, ein PIM mit Simulink abzubilden, obwohl sich das Beschreibungskonzept für Applikationen bei Simulink von dem von VRE unterscheidet. Simulink wurde gewählt, da derzeit noch keine spezifische Umgebung zur Beschreibung von VRE-Applikationen existiert. Desweiteren ist Simulink in der Kommunikationsindustrie weit verbreitet und wird daher das intuitive Verstehen der Applikationen und die Eingewöhnung für neue Entwickler erleichtern.

Dritter Schwerpunkt war die Identifizierung der hardwarespezifischen Informationen, die für den VRE Compiler-Kern notwendig sind. Dies umfaßt beispielsweise die Anzahl der Verarbeitungsmodule, die unterstützten Typen von SIMD-Befehlen etc. Es wurde zusätzlich ein geeignetes Format für diese Dateien gewählt.

Die Entwicklung des Prototyps des VRE Compiler-Kerns für die SDR-Hardwareplattform SB3010 war der vierte Schwerpunkt. Es handelt sich um einen parallelisierenden Compiler; dieser identifiziert automatisch mögliche Tasks, bereinigt bestimmte Abhängigkeiten, berechnet Mapping und Scheduling und fügt Synchonisierungen ein.

Im fünften Schwerpunkt wurde experimentell ein PIM entworfen und erfolgreich für den IEEE 802.11b-Empfänger übersetzt. Das Experiment zeigt, daß Simulink zur Beschreibung von PIMs genutzt werden kann und VRE die Programmierung von Tasks erleichtert. Beispielsweise benötigt der Entwickler weniger Zeit, eine Applikation als PIM in Simulink zu formulieren, als diese Applikation in einer plattformspezifischen

Sprache wie POSIX-C zu schreiben.

Einschließlich der Einführung umfasst diese Dissertation neun Kapitel. Kapitel 2 beschreibt das Konzept und die Vorteile von SDR, stellt einige aktuelle SDR-Hardwareplattformen vor und gibt einen Überblick über SDR-Applikationen.

Kapitel 3 diskutiert die aktuellen Entwicklungskonzepte für SDR-Applikationen und ihre Nachteile. Es werden zusätzlich Entwicklungsumgebungen für signalverarbeitende Applikationen vorgestellt, die zur Entwicklung von SDR-Applikationen in Frage kommen und ihre Beschränkungen aufgezeigt. Ferner beschreibt das Kapitel die existierenden Herangehensweisen zur Programmierung paralleler Maschinen jenseits von SDR und stellt ihre Unterschiede zu VRE dar.

Kapitel 4 führt die VRE-Sprache ein. Zunächst werden die Entwicklungsziele dargelegt und die Gründe für die Wahl einer visuellen Sprache genannt. Im Anschluss werden Syntax und Semantik der Sprache dargestellt und die Repräsentation von PIMs und PSMs erläutert.

Kapitel 5 beschreibt das Simulink-Tool und die Gründe, dieses Werkzeug in die VRE-Toolchain zu integrieren. Ferner enthält das Kapitel Richtlinien zur Repräsentation von PIMs mit Simulink.

Kapitel 6 gibt einen Überblick von IEE 802.11b WLAN und beschreibt das durchgeführte Experiment, einen IEEE 802.11b WLAN-Empfänger als PIM mit Simulink abzubilden.

Kapitel 7 stellt Inhalte und Formate der Hardware-Beschreibungsdateien vor. Es erklärt, warum diese Inhalte vom Compiler benötigt werden.

Kapitel 8 befasst sich mit dem Compiler-Kern. Vornehmlich wird an Beispielen gezeigt, wie der Compiler-Kern Tasks erkennt, Abhängigkeiten zwischen Tasks bewertet und einige von diesen auflösen kann, Mapping und Scheduling berechnet sowie benötigte Synchonisierungen einfügt.

Das abschließende Kapitel 9 zieht das Fazit der Arbeit und diskutiert mögliche zukünftige Arbeiten im Hinblick auf eine Weiterentwicklung von VRE.

Abstract

Software Defined Radio (SDR) hardware platforms use parallel architectures. Current concepts of developing applications (such as WLAN) for these platforms are complex, because developers describe an application with hardware-specifics that are relevant to parallelism such as mapping and scheduling. To reduce this complexity, we have developed a new programming approach for SDR applications, called Virtual Radio Engine (VRE). VRE defines a language for describing applications, and a tool chain that consists of a compiler kernel and other tools (such as a code generator) to generate executables. The thesis presents this concept, as well as describes the language and the compiler kernel that have been developed by the author. The language is hardware-independent, i.e., developers describe tasks and dependencies between them. The compiler kernel performs automatic parallelization, i.e., it is capable of transforming a hardware-independent program into a hardware-specific program by solving hardware-specifics, in particular mapping, scheduling and synchronizations. Thus, VRE simplifies programming tasks as developers do not solve hardware-specifics manually.

Acknowledgement

As an acknowledgment, I would like to express my heartiest gratitude towards my research supervisors, Prof. Dr. Claudia Fohry (Department of Computer Science and Electrical Engineering, University of Kassel, Germany) and Dr. Matthias Weßeling (CTO, mimoOn GmbH, Germany) for their support, advice and guidance throughout the progress of the research work. My special thanks and gratitude also go to Prof. Dr. Axel Hunger (Faculty of Engineering, University of Duisburg-Essen, Germany) for providing his valuable time as a reviewer of this dissertation. I am sincerely grateful to my family and friends, especially my parents and wife for their endless mental support, constant inspiration and deepest care during my stay away from home. Last but not least, my gratitude extends to Siemens AG/BenQ Mobile Group for giving me the opportunity to do research on the challenging topic and providing necessary funding.

Contents

Chapter 1

Introduction

1.1 Motivation

Software Defined Radio (SDR) is a recent approach for building wireless communication devices, which is characterized by the use of software modules to control radio functionalities such as modulation and demodulation, signal generation, coding, and link layer generation [26]. In contrast to the traditional way of building wireless devices, where radio functionalities are mainly implemented in hardware, it builds on the fact that the underlying hardware modules for digital radio systems are both reprogrammable and reconfigurable, i.e., a device can be easily adapted to a new communication protocol such as WLAN by simply replacing the software.

A communication protocol typically includes a physical layer and a medium access control (MAC) layer. The physical layer performs signal processing operations (such as modulation and demodulation) to convert streams of bits into MAC packets (on the receiver path) and MAC packets into streams of bits (on the transmitter path). The MAC layer performs addressing and channel access control operations that are required to communicate between network nodes. See [57] for further details.

In general, communication protocols have real-time requirements, e.g., the IEEE 802.11a WLAN protocol must process data at a rate of 54 Mbit/s. On the other hand, their physical layer algorithms (such as the Viterbi algorithm) are computationally complex. Therefore, one of the key requirements for SDR hardware platforms is high processing speed. As the battery-driven SDR hardware has a strict low power consumption budget, the required processing speed cannot be obtained through increasing the internal clock frequency. Instead, the architectures use parallelism, i.e., they deploy multiple arithmetic and signal processing units simultaneously.

Developing a program for a parallel platform is harder than developing a program for a serial platform, since developers have to deal with additional programming tasks: identification of parallel activities (tasks), computation of mapping (assigning tasks to processing modules) and scheduling (defining execution order of tasks), and insertion of synchronizations that ensure a desired execution order of tasks running on different processing modules.

Both mapping and scheduling are hardware-specific. Assume, for instance, that we

are developing an application that consists of four tasks with potential to run in parallel. Further assume that the application shall run on two different hardware platforms, one consisting of four and the other of two processing modules. Therefore, on the first platform, all four tasks can be run simultaneously, but on the second platform at most two tasks can be run at a time. Moreover, the functions themselves are hardware-specific. For example, SB3010 [48] uses POSIX functions [5, 28] to synchronize tasks, whereas Adelante VD3204x [51] uses EVP-specific functions [12].

At present, software development for SDR is performed with platform-specific tool chains, i.e., every platform defines its own application development environment. SB3010, for instance, comes with the Sandblaster Integrated Development Environment (Sandblaster IDE), which uses POSIX-C as its application description language, along with an optimizing POSIX-C compiler, debugger, simulator and profiler. From its knowledge of the target hardware, the compiler generates efficient executables.

The current programming concept for SDR applications is hardware-dependent, i.e., developers describe an application (communication protocol) in a platform-specific language such as POSIX-C including mapping, scheduling, and synchronizations. Hence, programming is complex:

- Developers need prior knowledge about the target hardware.
- Software development cannot be started in parallel with hardware development.
- Application descriptions are not portable.

To reduce this complexity, a programming concept is desirable that separates the description of an application from its implementation on a specific hardware, i.e., that allows developers to describe an application in a hardware-independent way and then deploys a compiler to produce executable code. Unfortunately, such a programming concept is not yet supported by any programming environment available today.

This thesis introduces a corresponding programming concept and describes a prototype of a programming environment that the author has implemented to verify the concept. The concept is described in the next section.

It is important to mention here that, there are concepts available that enable us to describe a program in a platform-independent way and define an interpreter to run the program on a platform. Java [49], for instance, defines a language to describe platform-independent programs and an interpreter, called Java Virtual Machine (JVM), to run Java programs on platforms for which JVM is available. However, such concepts are not quite suitable for developing real-time applications, as a program running on an interpreter is relatively slow as compared to a compiled, e.g., C program.

1.2 New Programming Concept for SDR

The research work presented in this thesis has been performed as part of the Virtual Radio Engine (VRE) project at Siemens AG in co-operation with the Department of

Electrical Engineering and Computer Science of the University of Kassel, and the Department of Computer Engineering of the University of Duisburg. The goal of the project has been development of a new and efficient programming concept (called VRE) for SDR applications. This thesis concentrates on the development of VRE for designing and implementing physical layers of SDR applications*. Some results of this thesis have already been published in [17, 18, 19, 20]. The VRE concept for MAC layers is presented in [22, 23].

Fig. 1.1 presents an overview of the VRE concept. In VRE, the development of an application is performed in two steps. First, the application is specified in a hardware-independent way and then, in a separate step, the hardware-specific implementation is done (to a great part) automatically by a tool chain.

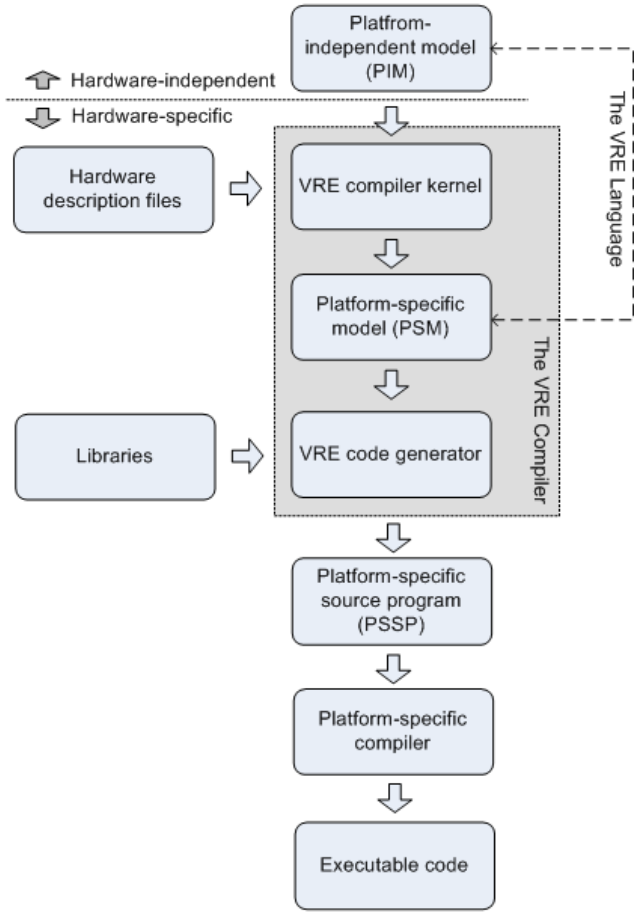


Figure 1.1: VRE tool chain.

VRE defines a language (called the VRE language) to describe an application (e.g. for WLAN) in a platform-independent way. In VRE, such a description is called Platform-Independent Model (PIM). Additionally, VRE defines a compiler (called the

*In the remaining thesis paper, the term 'application' refers to the physical layer of an SDR application unless explicitly mentioned otherwise.

VRE compiler) to generate executables. The compiler consists of a compiler kernel and a code generator. The compiler kernel transforms a PIM into another level of program description, called Platform-Specific Model (PSM). Like a PIM, a PSM is represented in the VRE language. Unlike a PIM, a PSM is hardware-specific in that it includes mapping, scheduling, and synchronizations.

Note that, in addition to the PIM, the compiler kernel takes as input hardware-description files that contain hardware-related information. It needs this information to compute mapping, scheduling, and synchronizations.

From a PSM, the code generator produces a Platform-Specific Source Program (PSSP). A PSSP is described in a platform-specific language (such as POSIX-C) and hence can be processed further by the platform-specific compiler (such as the POSIX-C compiler). Note that, in addition to the PSM, the code generator takes as input libraries of base functions that are available in the platform-specific language.

Finally, executables are generated from a PSSP by a platform-specific compiler, and can be run on the target hardware.

The VRE tool chain is semi-automatic, i.e., developers may automatically produce executables from a PIM step by step, but are additionally allowed to manually improve the performance of the code after each step.

VRE simplifies the application development process for the reasons given in Section 1.1. In particular, developers are neither responsible for mapping and scheduling, and nor for synchronizations. As will be discussed in the Chapter 4, use of the VRE language simplifies the program analysis by the compiler as compared to languages such as POSIX-C, and thus is the basis for automatizing mapping, scheduling, and synchronizations.

1.3 Scope

The thesis has made the following contributions to the development of VRE:

Development of VRE language: We decided for a visual language since it is more appropriate to SDR application development than a textual language such as C, as will be discussed in Section 4.1. The VRE language has been designed to be appropriate for both PIM and PSM descriptions. In a PIM, developers describe tasks, as well as mapping and scheduling restrictions, i.e., dependencies between tasks, e.g., as task T1 produces data for task T2, T1 has be executed before T2. In particular, they describe three types of dependencies: message-coupled, memory-coupled, and control-coupled dependencies. See Section 4.4 for further details. The PSM format allows developers to analyze and improve a program with less effort than if it would have been written in a platform-specific language.

Use of Simulink to represent PIM: Simulink is a software package developed by Mathworks [42], which enables developers to describe and simulate a signal processing chain. However, it has been developed to describe an application for a serial platform and therefore, as will be discussed in Section 5.2, its program description concept substantially differs from that of VRE. Several workarounds have

been discovered within the scope of this thesis to represent a PIM in Simulink. We integrate Simulink in the VRE tool chain, because we currently do not have any dedicated program description environment for representing a PIM. Other reasons for incorporating Simulink in the VRE tool chain will be discussed in Section 5.1.

Experiments: We have successfully designed and compiled a PIM for the IEEE 802.11b receiver.

Identification of hardware-specific information: The hardware-specific information that is relevant for the VRE compiler kernel has been identified and includes, e.g., number of processing modules, supported types of SIMD operations, etc. Additionally, an appropriate format for these files has been chosen.

Development of VRE compiler kernel: To provide an example, we have developed a compiler kernel for the SDR hardware platform SB3010. It is a parallelizing compiler, i.e., it automatically identifies tasks and dependencies between tasks, eliminates some dependencies, computes mapping and scheduling, and inserts synchronizations.

1.4 Outline

This dissertation comprises nine chapters, including this introduction.

Chapter 2 describes the conceptual model and advantages of SDR, presents some contemporary SDR hardware platforms, and gives an overview about SDR applications.

Chapter 3 discusses current concepts for developing SDR applications and their disadvantages. It also presents some signal processing application development environments that can potentially be used for developing SDR applications, and shows their limitations. Additionally, it describes existing approaches for programming parallel machines (beyond SDR) and how they differ from VRE.

Chapter 4 introduces the VRE language. First, it outlines the objectives in developing the language, as well as reasons for choosing a visual language. Then, it presents syntax and semantics of the language, and finally explains how PIM and PSM are represented.

Chapter 5 states reasons for integrating Simulink in the VRE tool chain and describes this tool. Then, it provides guidelines for representing a PIM in Simulink.

Chapter 6 gives an overview of the IEEE 802.11b WLAN and describes the experiment that has been conducted to represent the IEEE 802.11b WLAN receiver as a PIM in Simulink.

Chapter 7 presents contents and formats of hardware description files. Additionally, it explains why these contents are needed by the compiler.

Chapter 8 is devoted to the compiler kernel. In particular, it shows with examples how the compiler kernel identifies tasks, evaluates dependencies between tasks, eliminates some of them, computes a mapping and scheduling, and inserts required synchronizations.

Finally, chapter 9 provides conclusions and discusses potential future work towards further development of VRE.

Chapter 2

Software Defined Radio

This chapter describes the concept and benefits of Software Defined Radio (SDR). Additionally, it presents the hardware architectures of some contemporary SDR platforms. Considering the subject area of this dissertation, it does not provide a detailed description of the architectures, instead the architectural features that influence the development of software. Moreover, it provides a general overview of SDR applications.

2.1 Introduction to SDR

The Software Defined Radio forum [39] defines SDR technology as "radios that provide software control of a variety of modulation techniques, wide-band or narrow-band operation, communications security functions (such as hopping), and waveform requirements of current and evolving standards over a broad frequency range". In other words, SDR is a kind of radio device whose physical layer functionalities are implemented in software and the functionalities can be significantly altered through changes in software. It consists of a hardware that is both reprogrammable, i.e., the hardware can be reprogrammed to run different applications at different times, and reconfigurable, i.e., the functionality of the hardware can be customized at run time by reconfiguring the functionalities of the logic gates. In contrast to an SDR, a conventional radio consists of dedicated hardware modules to perform various radio operations such as modulation. Thus, it is usually determined primarily by hardware with minimal configurability through software. As the hardware dominates the design, upgrading a conventional radio design basically means completely abandoning the old design and thereby starting over again.

Jeffrey H. Reed presented a well accepted practical model of SDR in his book [38], which is shown in Fig. 2.1. The radio begins with a Radio Frequency (RF) front-end that consists of a smart antenna and the flexible RF hardware. The RF front-end is responsible for receiving and transmitting radio frequency signals. The flexible RF hardware performs RF amplification and analog down-conversion from RF to intermediate frequency* on the receiver path, and analog up-conversion and RF power amplification on the transmitter path.

*Intermediate Frequency (IF) is a frequency to which a carrier frequency is shifted as an intermediate step in transmission or reception.

The Analog-to-Digital Converter (ADC) and Digital-to-Analog Converter (DAC) convert the signals from analog to digital on the receiver path and digital to analog on the transmitter path, respectively. They form the interfaces between the analog and digital sections of the radio system. One of the important characteristics of SDR is that, in contrast to conventional radios, it converts the incoming signals from analog to digital on the receiver path as early as possible, and the outgoing signals from digital to analog on the transmitter path as late as possible.

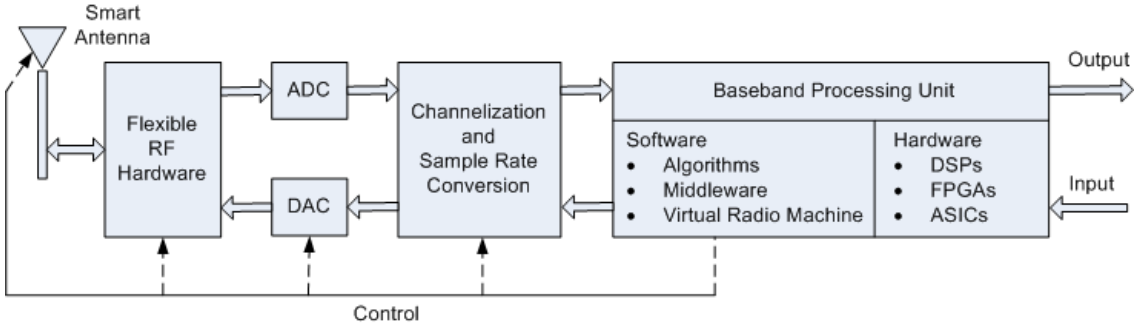


Figure 2.1: Model of SDR.

The next block in the model provides an interface between the ADC/DAC blocks and the baseband processing unit. It performs the digital filtering (channelization) and sample rate conversions. These operations are required both on the receiver and transmitter paths (see [38]).

SDR includes a reprogrammable piece of hardware, called baseband processing unit, to perform baseband processing operations. The unit consists of programmable modules such as Digital Signal Processors (DSPs) and Field Programmable Gate Arrays (FPGAs). The software running on these modules performs various signal processing algorithms such as modulation and demodulation.

Developers select an SDR platform chiefly based on the performance of the baseband processing unit. The performance is evaluated on the basis of metrics [10] such as processing speed, power consumption budget, die size, and reconfigurability.

2.2 Benefits

SDR has generated tremendous interest in the wireless communication industries for its wide-ranging benefits:

1. **Multi-functionality:** As already mentioned, SDR enables development of a mobile device that can operate different communication protocols such as both GSM and CDMA at different times, since it includes a reprogrammable and reconfigurable piece of hardware.

2. **Design flexibility:** Radio frequency components are difficult to design as their performance characteristics may vary due to unpredictable reasons and optimization in the analog domain takes long. SDR, on the other hand, digitizes the signals

as early as possible on the receiver path and transforms to analog domain as late as possible on the transmitter path, and thus simplifies product development since analyzing digital signals is easier than analog signals.

3. **Ease of upgrading and fixing:** Repairing system defects is an important issue for radio manufacturers. A small bug in the radio functionality may cause the return of thousands of mobile sets to device manufacturers. With SDR-enabled handsets, bugs can be repaired by simply upgrading the software. That means, users do not need to return their handsets, but can simply download software updates and install them.

2.3 Contemporary SDR Hardware Platforms

From programming point of view, an SDR hardware platform corresponds to the base-band processing unit shown in Fig. 2.1. There are some SDR hardware platforms already available in the market. These platforms have one point in common, i.e., their architectures use parallelism. However, they architecturally differ from one another and support different types of parallelism. For an overview, this section present four contemporary SDR platforms: Sand-Blaster 3010 (SB3010), Adelante VD3204x, MuSIC, and Reconfigurable Compute Fabric (RCF).

2.3.1 SB3010

SB3010 is developed by Sandbridge Technologies [48]. As depicted in Fig. 2.2, it consists of four architecturally identical DSP cores connected in a ring topology. It supports:

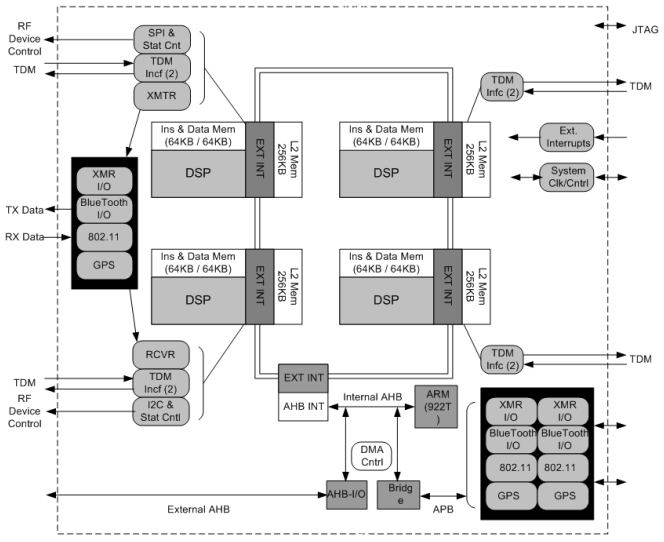


Figure 2.2: The SB3010 platform architecture.

Multithreading: Each core can run up to 8 concurrent hardware contexts (threads), and thus the four cores support up to 32 concurrent hardware threads.

Vector Processing: Each core contains a vector processing unit that performs vector operations on two vectors of four 16-bit elements each.

SB3010 defines its own programming environment, called the Sandblaster Integrated Development Environment (Sandblaster IDE), that uses POSIX-C as its application description language and includes a tool chain to support the implementation process. The tool chain consists of an optimizing POSIX-C compiler along with a linker, debugger, emulator and profiler. See [9, 14, 15, 16, 40] for more information on the hardware and software development environment.

2.3.2 Adelante VD3204x

The Adelante VD3204x [51] is an Embedded Vector Processor (EVP) DSP subsystem developed by NXP (former Philips Semiconductor) [32]. As depicted in Fig. 2.3, it consists of several functional units: A Program Control Unit (PCU), assisted by an Address Computation Unit (ACU), controls the functionality of a Scalar Data Computation Unit (SDCU) and a Vector Data Computation Unit (VDCU). SDCU and VDCU perform arithmetic and logic operations on scalar and vector data, respectively. In EVP, the length of a vector is limited to 256 bits, which may correspond to 32 elements of 8 bits, 16 elements of 16 bits, or 8 elements of 32 bits.

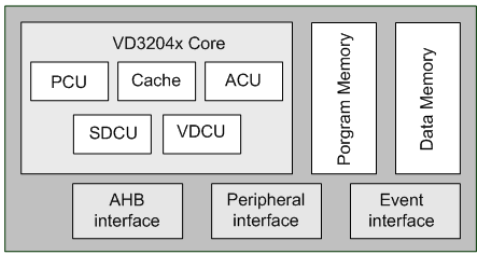


Figure 2.3: The Adelante VD3204x architecture.

Like SB3010, EVP defines its own programming environment, called the EVP Standard Development Kit (EVP-SDK), that uses EVP-C as its application description language and includes a tool chain to support the implementation process. The tool chain consists of an EVP-C compiler along with a linker, debugger, simulator, emulator, and profiler. See [12] for more information on the hardware and software development environment.

2.3.3 MuSIC

MuSIC is a product of Infineon Technologies [33]. As shown in Fig. 2.4, its architecture contains four Single Instruction Multiple Data (SIMD) cores connected by a bus. Each

core, on the other hand, consists of four Processing Elements (PEs), where each PE can perform SIMD operations on two vectors of four 16-bit elements each. Additionally, the architecture includes specialized processors (i.e., hardware accelerators) to perform FIR and Viterbi operations.

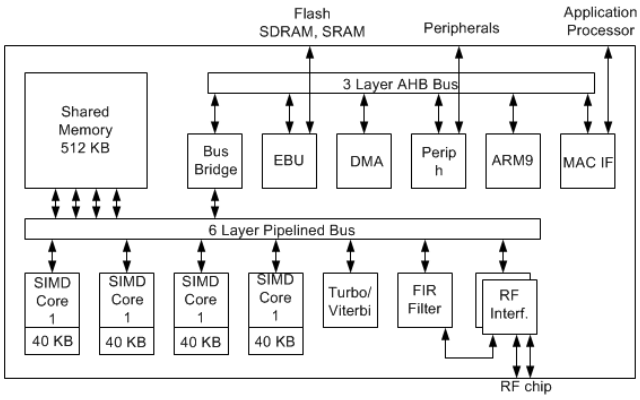


Figure 2.4: The MuSIC architecture.

Like the SB3010 and EVP platforms, MuSIC defines its own programming environment that uses the Data Parallel C Extension (DPCE) [31] as application description language and the DPCE compiler to generate executables. See [1, 3, 4, 37] for more information on the hardware and software development environment.

2.3.4 RCF

RCF is developed by Morpho Technologies [47]. Fig. 2.5 presents the architecture of RCF. In RCF, computations are mainly performed in a reconfigurable cell (RC) array that is composed of two rows of 8 RCs, where each RC is a 16-bit processing element. The RC array enables SIMD operations, i.e., 16 separate computations by 16 (8 * 2) RCs in one clock cycle. In additions, RCF comprises a hardware accelerator ("Interleaver") to perform various kinds of interleaving and deinterleaving operations.

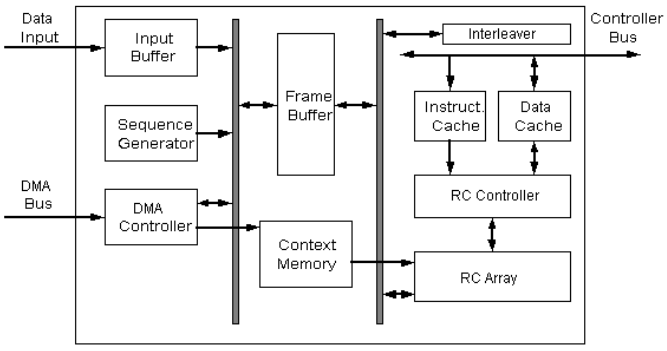


Figure 2.5: The RCF architecture.

Like the other platforms, RCF defines its own programming environment that uses RCF-C as application description language and the RCF-C compiler along with a linker to generate executables.

2.3.5 Comparative Analysis

As shown in Table 2.1, the architectures of different SDR hardware platforms differ from one another to a great extent. For instance, some architectures (e.g., MuSIC) contain hardware accelerators to perform computationally complex time-critical operations (e.g., the Viterbi operation), whereas others (e.g., SB3010) do not.

Table 2.1: Comparison among different SDR platform architectures.

No. of Cores	SB3010 Adelante VD3204x MuSIC RFC	4 1 4 1
HW-Threads/Core	SB3010 Adelante VD3204x MuSIC RFC	8 1 4 1
SIMD Length	SB3010 Adelante VD3204x MuSIC RFC	4*16 (64 bits) 8*32,16*16,32*8(256 bits) 4*16 (64 bits) 2 rows of 16*8 or 128bits
HW Accelerator	SB3010 Adelante VD3204x MuSIC RFC	-No- -No- FIR/Viterbi Interleaver
Supported Data Widths	SB3010 Adelante VD3204x MuSIC RFC	16, 32 (bit) 8, 16, 32, 40 (bit) 16, 40 (bit) 16, 32 (bit)
Instruction Type	SB3010 Adelante VD3204x MuSIC RFC	Compound (3 operations) VLIW LIW RISC

Although the architectures use parallelism, they substantially differ from one another with respect to their supported parallelism. For instance, SB3010 allows concurrent execution of hardware threads, whereas Adelante VD3204x is mainly a vector processor and does not support multithreading. Some architectures even support similar types of parallelism (e.g. SIMD operations), but do not support these in the same way. Both SB3010 and Adelante VD3204x, for instance, support SIMD operations, but SB3010 works on 64-bit vector data and Adelante VD3204x on 256-bit vector data.

2.4 Overview of SDR Applications

This section provides an overview of SDR applications and discusses briefly the issues that influence the software development of SDR applications.

In general, SDR applications convert data from one format to another, e.g., as shown in Fig. 2.6, the IEEE 802.11b WLAN transforms MAC packets into streams of bits on the transmitter path, and streams of bits into MAC packets on the receiver path. These applications include operations that often correspond to computations applied to a vector of values such as FFT and Viterbi.

To develop software for an SDR application, developers first determine which operations are to be performed by the application and in which order. Then, they write source code for the application, and later employ a compiler to produce executables.

The standard specification document of an SDR application provides a description of the application, which is similar to the block diagram shown in Fig. 2.6. The description in particular describes which operations are to be performed and in which order, but not how the operations are to be realized.

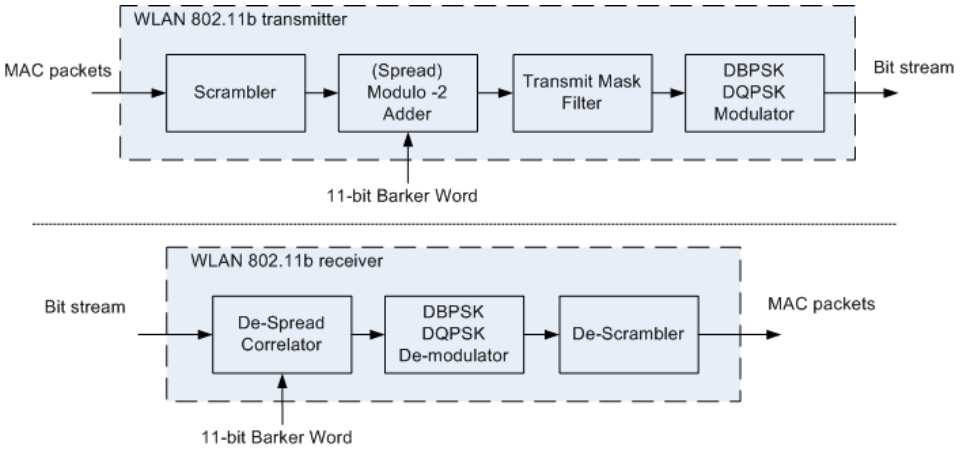


Figure 2.6: The IEEE 802.11b WLAN signal processing chain.

As a result, developers themselves do not have to invent which operations are to be performed, instead they mainly concentrate on how to perform an efficient implementation, i.e., how to deploy the program to a target hardware in such a way that the real-time requirements of the application are fulfilled.

In contrast to SDR applications, developers of other types of applications such as computer games and word processors usually spend a lot of time in determining application logics. For instance, while developing a compute game, they typically spend a lot of time to determine graphical components, user interfaces, database techniques, etc.

2.5 Summary

This chapter has given an introduction to SDR. To provide an overview of SDR hardware architectures, it has presented four contemporary SDR hardware platforms: SB3010,

Adelante VD3204x, MuSIC, and RCF. Additionally, it has discussed about SDR applications and the issues that influence the software development of SDR applications. The next chapter describes the current programming concepts for SDR applications with more details and shows their limitations.

Chapter 3

Programming SDR Application

This chapter shows that the current concept for developing SDR applications is complex as developers describe an application with hardware-specifics such as mapping and scheduling. Additionally, it describes various signal processing application development environments that can potentially be used for developing SDR applications, and discusses their limitations. Moreover, it presents existing approaches of parallel programming (beyond and including SDR) and compares them with VRE.

3.1 The Current Programming Concept for SDR

As already mentioned in Chapter 1, at present, developers use platform-specific tool chains such as Sandblaster IDE to develop software for SDR. Programming is complex, because developers describe an application in a platform-specific language such as POSIX-C with hardware-specific details: vector operations, mapping, scheduling, and synchronizations. This section describes why vector operations, mapping and scheduling, and synchronizations are hardware-specific.

Vector Operations:

Vector instructions usually take operands of length 2^n, as specified by the size of the vector registers of the corresponding platform. Therefore, original vector operations must be divided into strips of this length, which is called stripmining. For instance, as shown in Fig. 3.1, the vector operation on two vectors with 64 elements of 16-bit has to be broken down into strips of 256 bits in length in the case of EVP and 64 bits in case of MuSIC. See [46, 55] for further details about stripmining.

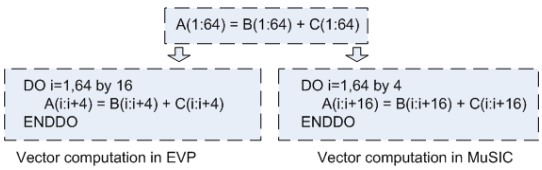


Figure 3.1: Vector operations in EVP and MuSIC.

Scheduling and Mapping:

In the context of this thesis, mapping means assignment of tasks to processing modules for execution, and scheduling means definition of tasks' execution order on processing modules. To show that both mapping and scheduling are hardware-specific, an example in depicted in Fig. 3.2. Here, the functional description shows two data paths of a signal that correspond to the real and imaginary parts, respectively. Each path includes two different filter modules with the same functionality (noise elimination) but different parameters (Coef_rx1 and Coef_rx2) to specify different coefficient lengths. Then, in case of the hardware platform A that contains a dedicated processing module to perform these noise filtering operations, all four logical filters must be mapped to that processing module, for instance using time-multiplexing. To avoid a huge number of reconfigurations of this hardware module (regarding coefficient loading and number of taps), an optimal schedule will apply the first filter of each path first, and then the second. On the other hand, the hardware platform B has a specific module for each of these noise filtering functionalities (i.e., the processing modules named "FIR A" and "FIR B" respectively), because they require high performance. For hardware B, as depicted in Fig. 3.2, the mapping and scheduling described for platform A is not possible at all, because data can not be buffered after the first filter.

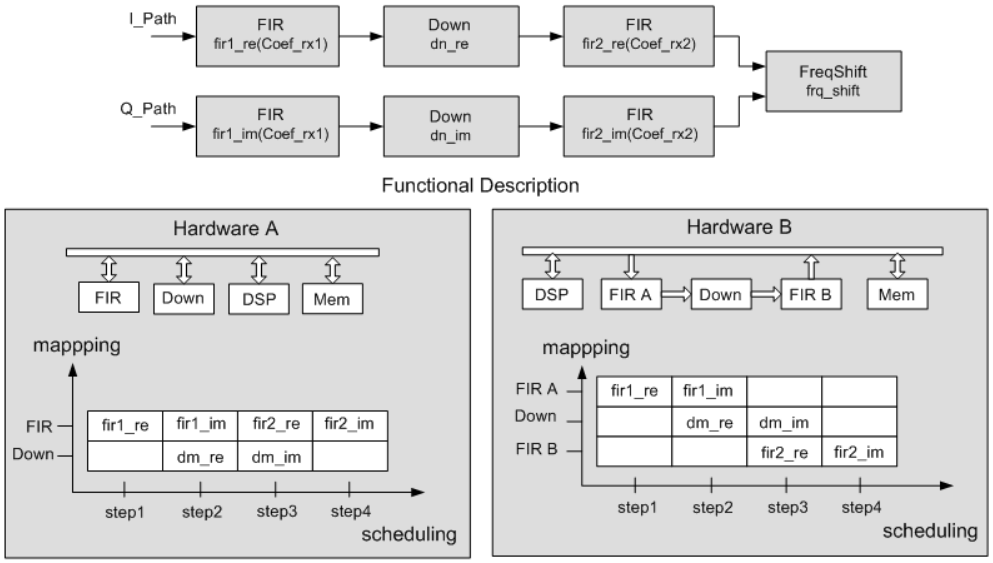


Figure 3.2: Mapping and scheduling a functional description in different platforms.

Synchronization:

Tasks are synchronized on different hardware platforms employing different synchronization mechanisms. On the SB3010, for instance, they are synchronized with POSIX functions using mutexes and condition variables (see [5, 28]). On the EVP, in contrast, they are synchronized with EVP-specific functions using semaphores [12].

3.2 Other Development Environments

In addition to the platform-specific tool chains, there are environments available in the market for modeling and simulating signal processing applications. These environments typically offer their own application description language. Some of them also include compiler and have the potential to be used for developing SDR applications, but are unable to provide an efficient solution.

An example of such an environment is Simulink [42]. It offers a graphical language to describe signal processing applications and can be used together with Real Time Workshop [56] and target language compiler to perform the implementation. The tool chain works fine as long as there is a simple DSP-like hardware architecture, i.e., application development for parallel hardware platforms is not possible. Furthermore, Simulink requires the schedule to be fixed, which restricts portability.

Van der Wolf et al. [50] describe an approach for multiprocessor-based embedded software development and implementation. Their goal is to automate the generation of source code according to guidelines provided by developers, but they do not give consideration to efficiency of the source code, e.g., the automatically generated code is not as efficient as hand written-code.

Examples of other development environments that support multi-processing include Gedae [13, 45], Ptolemy [34], MLDesigner [27], and Waveform Description Language (WDL) [54]. Gedae defines its own application description language, called Primitive and Graph language, and transforms an application description into a program on a virtual machine. Its concept of virtual machine seems to not support the high throughput and good latency that are required for high-performance SDR applications. Ptolemy is an environment for simulation and designing application for multiprocessor platforms. The industrial successor of Ptolemy is MLDesigner. The problem with Ptolemy and MLDesigner is that they require the scheduling to be fixed, i.e., the description of an application can only be realized on one particular hardware platform. Like Ptolemy and MLDesigner, WDL includes the same problem, i.e., it requires scheduling to be fixed.

Another example is SynDEx [2, 35]. It supports application development for parallel machines. The system resembles VRE, i.e., it separates the description of an application from its implementation on a specific hardware and contains a compiler that generates executables. Unfortunately, its Directed Acyclic Graph (DAG) based application description language is not quite suitable for specifying the complete behavior of an application, e.g., it is not possible to describe switch or branch operations. Moreover, in Syndex, mapping and scheduling are restricted to multiprocessor-based architectures, but some issues related to parallelism, such as vectorization, are not considered.

3.3 Existing Parallel Programming Concepts vs VRE

In general, beyond and including SDR, there are three concepts available for developing programs for parallel machines with shared memory architecture*:

*We only concentrate on shared memory architecture as SDR hardware uses that.

1. Writing a serial program and compiling it with a parallelizing compiler.

2. Writing a program placing compiler specific directives at appropriate points to describe parallelism and then compiling it with a compiler to produce a parallelized executable code.

3. Writing a program expressing parallel activities explicitly.

Option 1 is obviously the easiest for a developer as it is easier to write a serial program than it is to write a parallel program. It relies on a compiler that is capable of analyzing the source code written as a serial program and identifying the opportunities for parallelism. From its knowledge of the hardware, as discussed in [8], the compiler generates parallelized executable code. Unfortunately, there is no parallelizing compiler available in the market that is capable of generating efficient executables for parallel machines from serial programs.

Option 2 is easy for developers as well, because developers can leave many programming tasks to be solved by a compiler such as creation of threads. The programming approach that has been developed based on this option is OpenMP [21] that supports multithreaded programming in C/C++ and Fortran. OpenMP includes a set of compiler directives and an OpenMP program is just a serial program with OpenMP directives placed at appropriate points. A serial compiler will ignore the directives and produce usual serial executable code. An OpenMP-enabled compiler will recognize the directives and produce parallelized executable code. Unfortunately, OpenMP has some limitations, e.g., reliable error handling is missing, lacks fine-grained mechanisms to control thread-processor mapping, etc. On the other hand, OpenMP is not available for SDR hardware platforms. Therefore, it is unable to provide an efficient programming solution for developing SDR applications.

Option 3 corresponds to the current practice of developing SDR applications (i.e., developing an application with a platform-specific tool chain). Thus, as already discussed in Chapter 1, it is complex.

On contrary to these options, as already noted in Chapter 1, in VRE, developers first describe a program in a platform-independent way, i.e., they describe tasks and dependencies between tasks, but not hardware-specifics such as mapping and scheduling. Then, they employ a compiler to generate efficient parallelized executable code from the described program.

Moreover, as already stated in Chapter 1, VRE enables developers to produce executables from described programs semi-automatically, i.e., developers may automatically produce executables from a PIM step by step, but are additionally allowed to manually improve the performance of the code after each step. Thus, in VRE, developers are allowed to influence the implementation of an application, but not have to totally rely on the performance of the tool chain. For instance, developers employ the compiler kernel to produce a PSM from a PIM, and then other tools (the code generator and the platform-specific compiler) to perform an implementation. Additionally, they can manually modify the PSM and thus can perform a different implementation of the program. It is important to mention here that a PSM is described in a well-defined format that

enables developers to analyze and modify mapping, scheduling, and synchronizations with less effort (see Section 4.5).

For clarity, an example is presented in Fig. 3.3. It shows an application that consists of five blocks, where the block "RxBuf" reads samples and others blocks perform signal processing processing operations on the samples. Note that each block performs operations on a vector of 80 elements. Now assume that the application description corresponds to a PIM, and we employ the VRE compiler and thus produce a PSM that corresponds to the implementation A. Then, we can analyze the PSM, and thus can improve the PSM, i.e., the signal processing operations can be started immediately after receiving 20 samples instead of 80 samples. Hence, we can perform another implementation that corresponds to the implementation B, as the overall execution time of the signal processing chain in the implementation B is lower than the implementation A.

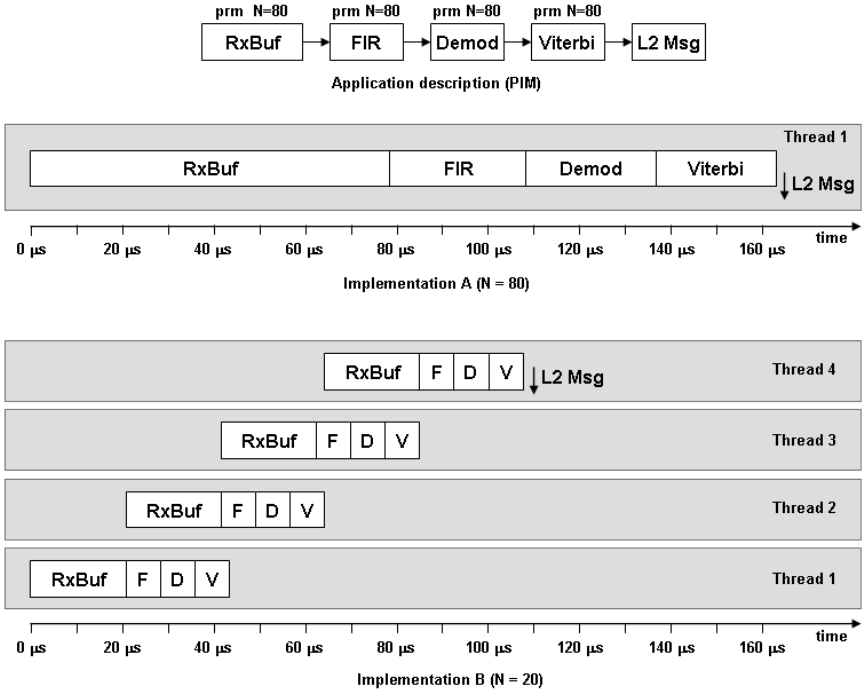


Figure 3.3: An example that shows how VRE's semi-automatic implementation process enables developers to perform an efficient implementation.

3.4 Summary

This chapter has shown that the current programming concept of developing SDR applications is hardware-specific. It has also presented some signal processing application development environments that have the potential to be used for developing SDR applications, and discussed their limitations. Additionally, it has presented existing concepts of parallel programming, compared them with VRE. The next chapter presents the VRE language.

Chapter 4

The VRE Language

The chapter is devoted to the VRE language. It discusses factors that are considered for the language development, explains reasons for choosing a visual language, describes syntax and semantics of the language, and presents PIM and PSM description concepts.

4.1 Introduction to the VRE Language

The VRE language has been developed to be appropriate for SDR applications. As discussed in Section 2.4, SDR applications correspond to signal processing chains, i.e., they consist of a sequence of signal processing operations such as FFT and Viterbi. These applications can be represented by a block diagram that is a way to represent an application using two major classes of elements: blocks and arrows. Blocks are used to represent operations and arrows to represent transfers of signals (data) between blocks, where directions of arrows denote directions of signal flows.

A block diagram in particular describes the order in which different operations of an application are to be performed, as well as inputs and outputs of these opertaions. For clarity, an example signal processing chain (as a block diagram) is presented in Fig. 4.1. It consists of four signal processing operations ("Filter", "Demodulator", "Descrambler", and "CRC") and describes in which logical sequence the operations are to be performed, i.e., first the "Filter" block then the "Demodulator" block, and so on. Note that it also describes the inputs and outputs of the blocks.

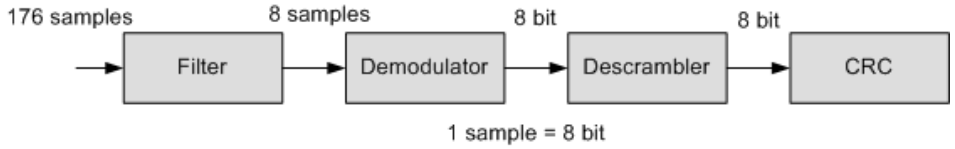


Figure 4.1: Description of a signal processing chain as a block diagram.

The concept of representing an application as a block diagram is easy, especially when required blocks are available to developers as libraries. For instance, as shown in Fig. 4.2, we can simply describe an application by adding library blocks to a graphical

editor, and then connecting them with arrows. This concept of describing an application is easier than to describe an application textually (e.g., in C), as we can avoid a lot of programming tasks, e.g., defining and initializing variables, allocating and freeing memory locations, writing code for algorithms, etc. Additionally, a program described as a block diagram is easy to analyze if appropriate tools are available. For instance, Simulink enables us to describe a program as a block diagram and simulate the program, as well as provides some special library blocks such as "Scope" to visualize outputs of blocks and thus to check whether any block produces an incorrect output (see [42]).

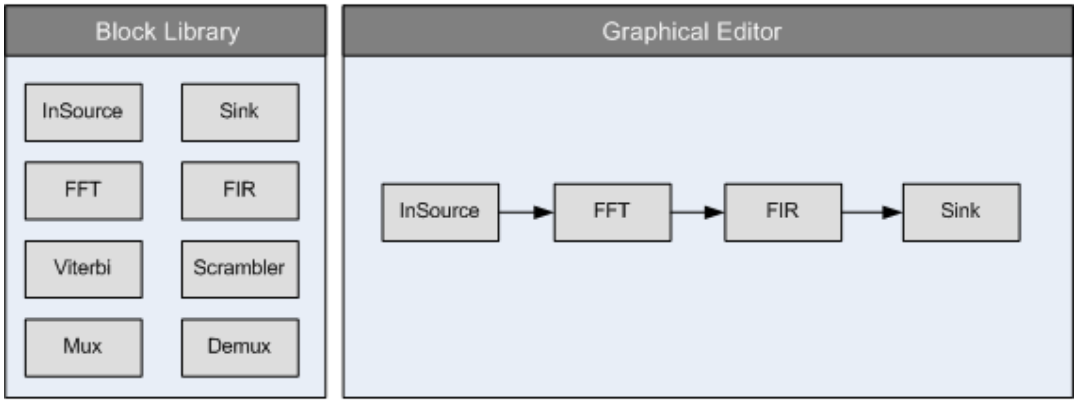


Figure 4.2: Describing a block diagram using library blocks.

Due to the advantages discussed above, the VRE language has been developed as a visual language, i.e., it defines a set of description components and enables developers to represent an application graphically similar to a block diagram using these description components. The next sections introduces these description components.

4.2 Description Components of the VRE Language

VRE defines the following description components to represent an application:

4.2.1 System

A system corresponds to an application description, i.e., either a PIM or a PSM. It is described as a block diagram, and is composed hierarchically, i.e., it consists of blocks (not yet introduced) that may internally contain other blocks.

4.2.2 Parameter

A parameter is a compile-time constant and, as shown in Fig. 4.3, has four attributes:

1. Name - name of the parameter.

2. Type - data type of the parameter such as T_Uint8 (8-bit unsigned integer). See Section 4.3 for more information about VRE-specific data types.
3. Size - size of the parameter, e.g., a parameter with size = 4 corresponds to a vector of 4 elements.
4. Value - value of the parameter.

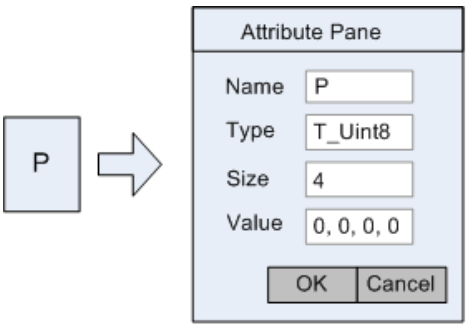


Figure 4.3: Parameter description in the VRE language.

In VRE, some description components have their own parameters, e.g., a "primitive" (not yet introduced) may have its own parameter. These parameters are discussed along with their corresponding description components.

4.2.3 Memory

A memory corresponds to a variable, i.e., its value may change at run-time. As shown in Fig. 4.4, like a parameter, a memory has four attributes.

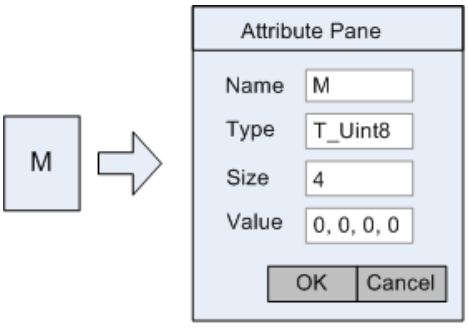


Figure 4.4: Memory description in the VRE language.

4.2.4 Flag

VRE separates boolean variables from other variables that we do not have to describe data types and sizes of boolean variables explicitly. In VRE, a flag corresponds to a

boolean variable, i.e., its value can be either 0 or 1. As shown in Fig. 4.5, a flag has two attributes: name and value.

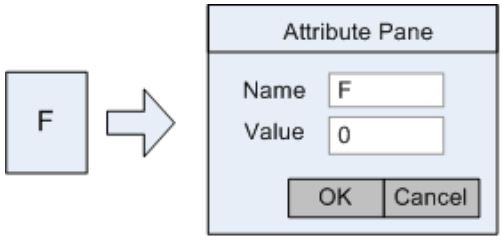


Figure 4.5: Flag description in the VRE language.

4.2.5 Primitive

In VRE, primitives are considered as software modules (such as FFT and Viterbi) and an application program is built up from them. They are available to application developers as libraries and considered as black boxes: only the interfaces, e.g., memory and flag accesses, are known to application developers. Their implementations are provided by hardware manufacturers, and may differ between platforms.

An example of a primitive is shown in Fig. 4.6. Each primitive has a name ("FIR"), and may have input and output data or control flow (trigger) interfaces. An input interface is described by an input port ("X"), and an output interface is described output port ("Z"). Every port has a unique name such as "X" , as well as data type and size (not shown in the figure).

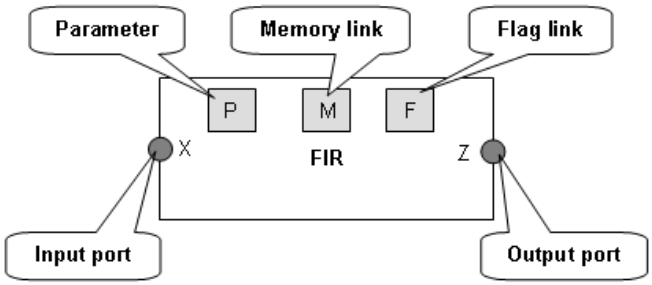


Figure 4.6: Primitive representation in the VRE language.

A primitive may have parameters to customize functionalities, e.g., the same filter primitive can be instantiated in different parts of a system to work with different filter coefficients by means of setting the filter coefficient parameter ("P") to different values.

A primitive may internally access memories and flags during execution, which are specified through memory links (i.e., pointers to memories) and flag links (i.e., pointers to flags), respectively. Both memory and flag links have the following attributes:

- Name - name of the memory/flag link.
- Link - name of the memory/flag that the memory/flag link points to.
- Type - data type of the memory/flag that the memory/flag link points to.
- Access - type of access to the memory/flag such as ReadOnly (will be discussed later).

Additionally, a memory link have another attribute, called Count, which will be discussed in the next section.

4.2.6 Block

A block is an instantiated primitive or a module (not yet introduced) in a system. In VRE, these two types of blocks are called primitive block and modular block, respectively. Each block has a unique name. Unlike modular blocks, each primitive block has an attribute (called class) that defines which primitive the block refers to. For instance, assume a block with attributes: name = "FIR1" and class = "FIR". Then, it means the block's name is "FIR1" and the block is an instance of a primitive named "FIR".

4.2.7 Signal

A signal represents an exchange of information between blocks. As shown in Fig. 4.7, it connects ports and is represented by an arrow. It describes either a data flow or a trigger (will be discussed later). Note: A signal itself does not describe the type, size or other properties of the information exchanged, instead these can be seen from the corresponding ports' properties.

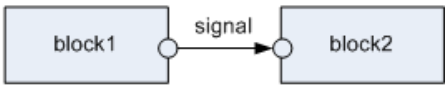


Figure 4.7: Description of signal in VRE.

4.2.8 Module

A module is a hierarchical description and composed of blocks. Internally, it describes a signal processing chain, i.e., a set of blocks exchanging information through signals, accessing memory to read or write data, etc. Externally, its interface corresponds to that of a primitive, i.e., it includes parameters, input/output ports, memory as well as flag accesses. An example is depicted in Fig. 4.8 that shows both the top level view and the internal description.

As shown in the figure, each module has a name ("HeaderModule"). It may internally contain memories ("M2") and flags ("F2") in addition to those that are visible as its

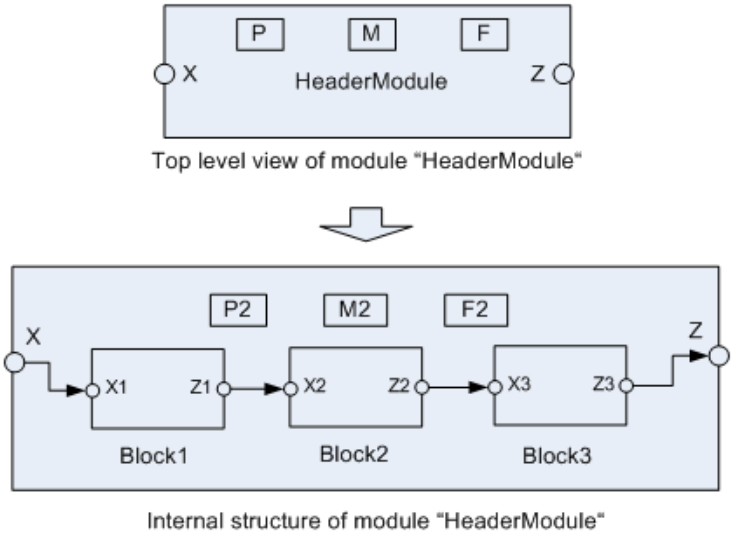


Figure 4.8: Representation of a module in the VRE language.

interfaces such as the memory link ("M") and the flag link ("F"). Likewise, it may internally contain additional parameters ("P2"). The external interfaces such as ("M") and ("F") are not explicitly described within the internal structure since they are part of the module definition and thus visible to the internal blocks such as Block1. For clarity, assume a function that is shown in Fig. 4.9 and written in C, where the argument p of the function (function1) is a part of the function definition and thus is not declared again within the body of the function. However, another variable k whose scope is local to the function is declared within the body of the function.

```
int function1 (int p)
{
    int k;
    k= 100;
    // do something with the variables here
    return k*p;
}
```

Figure 4.9: An example function written in C.

Like primitives, modules can be available to developers as libraries, i.e., developers describe modules independent of a system and store them as libraries.

4.2.9 Task

In VRE, a task corresponds to a primitive or a set of primitives that is mapped to a processing module for execution. It is a building block of a PSM and can be hierarchically composed, i.e., it may internally contain other tasks which are called *subtasks*. A task

that internally contains subtasks is called the *supertask* of these tasks. See Section 4.5 for further details.

4.3 Important Rules and Type Information

This section contains additional rules on the use of the components from Section 4.2.

Naming

Within a system, description components of the same type are distinguished from one anther by their names, i.e., two blocks within a system must have different names.

Scope

In a system, memories, flags and parameters are visible only within their scopes, i.e., if they are described at system level then they are visible throughout the application, and if they are described within a module then they are only visible to the blocks within the module. For clarity, an example is depicted in Fig. 4.10. Here, the system comprises two memories: M1 and M2. As M1 is described at the system level, it is accessible by all the blocks, i.e., B1 and B2, as well as B11, B12, B21, and B22. On the other hand, as "M2" is described within the modular blocks B1, it is visible to B11 and B12 only.

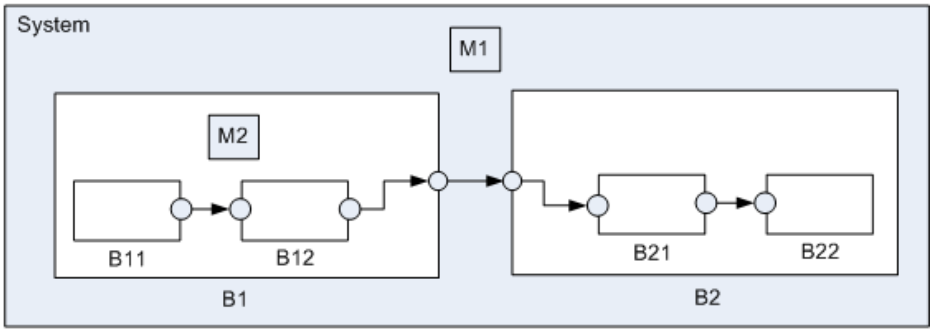


Figure 4.10: Scopes of memories in a system.

Accessing Memories

Two blocks that access the same memory to read data may run in parallel, but if at least one of the accesses is a write then the blocks must not run in parallel. Assume, for instance, that a block B1 writes data to a memory and another block B2 reads from the same memory. If the write has be done before the read then B1 has to be executed before B2, otherwise B2 would read incorrect data.

To enable the VRE compiler to evaluate whether or not blocks that access shared memories have the potential to run in parallel, it must be explicitly described in a

program whether accesses are reads or writes. Thus, VRE distinguishes three types of memory accesses:

1. ReadOnly - a block accesses a memory only to read data.
2. ReadWrite - a block accesses a memory either to write or both to read and write data.
3. Counter - a block accesses a memory to increment or decrement the value of the memory with a compile-time constant.

Note that both "ReadWrite" and "Counter" correspond to write operations. However, they carry different types of information for the VRE compiler. For clarity, an example is depicted in Fig. 4.11. Here, the signal processing chain consists of two blocks: *B1* and *B2*. Upon execution, both of these blocks first read N data elements from array M starting from offset C, where N is the value of the parameter *P1*, M is the data stored in the memory *M1*, and C is the data stored in the memory *C1*. Next, they perform two different types of interpolation operations using the read data, respectively. Then, they store the interpolation results to the memories *S1* and *S2*, respectively. Finally, they increment the value of *C1* by N and thus have a data dependency, i.e., *B1* should first increments the value of *C1* and then *B2*. However, this dependency can be eliminated as it is possible to determine at compile-time how each of these blocks increments the value of *C1* ($C = C + N$). Therefore, eliminating the dependency, it is possible to run these blocks in parallel. In VRE, such a write (e.g., the write to *C1* by the block *B1*), which creates an avoidable data dependency between blocks, is described as a "Counter" access. In contrast to this, when a block accesses a memory and then changes the value of the memory in a way that cannot be determined at compile-time, it is described as a "ReadWrite" access.

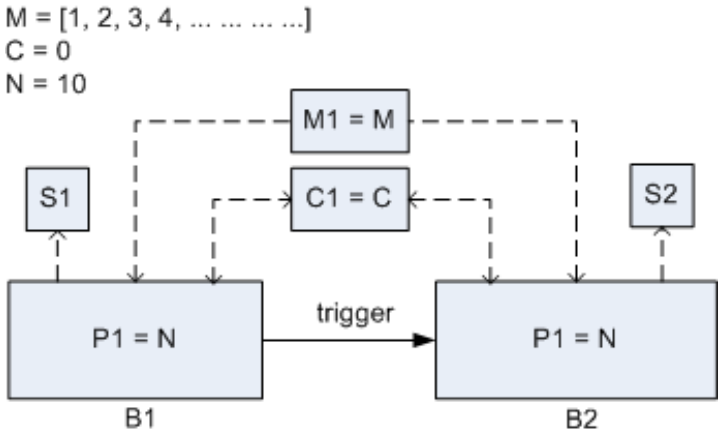


Figure 4.11: An example to show difference between 'Counter' and 'ReadWrite' accesses.

If a block has a "Counter" type access to a memory then the "Count" attribute of the corresponding memory link defines how the block increments or decrements the value

of the memory, e.g., "Count += 10" means the block increments the memory value by 10 and "Count -= 10" means the block decrements the memory value by 10.

Accessing Flags

In VRE, it is explicitly described in a program why a block accesses a flag, i.e., to toggle or check the value. The reason is analogous to the reason for distinguishing different types of memory accesses. VRE distinguishes two types of accesses:

1. ActiveAccess - an access that may change the value of a flag.
2. PassiveAccess - an access to check the value of a flag.

Representation of Controls

The VRE language supports two classes of control descriptions: loops and branches. See Section 4.4 for details.

Data Types

Table 4.2 presents the supported data types of VRE. In addition to the data types shown in the table, ports can be of type "T_Trigger". A signal that describes a connection between an input and output port of type "T_Trigger" represents a *trigger flow*, see Section 4.4.2.

Table 4.1: VRE-specific data types.

Type	Description
T_UInt8	8-bit unsigned integer.
T_UInt16	16-bit unsigned integer.
T_UInt32	32-bit unsigned integer.
T_SInt8	8-bit signed integer.
T_SInt16	16-bit signed integer.
T_SInt32	32-bit signed integer.
T_Bit	Boolean value, i.e., either 0 or 1.
T_String	Array of characters.

4.4 PIM Description Concept

As PIM is hardware independent, it does not include hardware-specific information such as mapping and scheduling. Instead, it describes restrictions of scheduling, i.e., dependencies between blocks, e.g., block "A" produces data that block "B" consumes then "A" has to be executed before "B". In particular, it describes three types of dependencies: *message-coupled*, *memory-coupled*, and *control-coupled dependencies*.

4.4.1 Message-Coupled Dependencies

Message-coupled dependencies refer to direct data flow between blocks, i.e., a block is ready to execute but waiting for input data that is produced by another block. VRE describes these dependencies by simply connecting an output port to one or several input ports. The dependencies are directed, i.e., if an output port of block "A" is connected to an input port of block "B", then the execution of block "B" is dependent on the execution of block "A". In VRE, if a message-coupled dependency exists between two blocks then the block that produces the data is called *the producer block* and the block that consumes the data is called *the consumer block*.

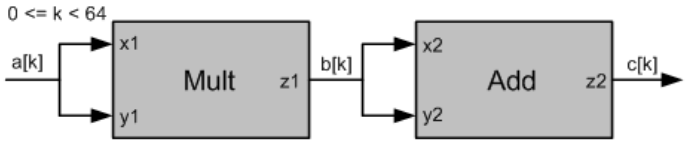


Figure 4.12: Example of message-coupled dependency.

Message-coupled dependencies describe the following scheduling restriction: the consumer and producer blocks can execute in any order as long as their executions do not violate the data dependency between them. For clarity, an example is depicted in Fig. 4.12. There are two blocks, "Mult" and "Add", which are connected through message exchange. The input and output ports of each block correspond to a data vector of 64 values. Thus, each block has to perform 64 operations: (b[k] = a[k] * a[k] and c[k] = b[k] + b[k]), for k = 0..63. Note that the description does not define how the blocks have to be executed. For instance, the blocks do not necessarily have to execute sequentially, but can run in parallel with one restriction: T(b[k] = a[k]*a[k]) < T(c[k] = b[k]+b[k]) for each k. That means, the k^{th} "Add" operation can be performed any time after the k^{th} "Mult" operation, but not necessarily after all 64 "Mult" operations.

4.4.2 Memory-Coupled Dependencies

If two blocks access the same memory, we speak of a memory-coupled dependency. In this case, the order of memory accesses define the order in which the corresponding blocks are data-dependent. To represent memory-coupled dependencies between blocks, the developer needs to introduce trigger flows (i.e., connecting input and output ports of type "T_Trigger" through signals) between the blocks to indicate the order of memory accesses. Trigger flows may connect the blocks directly or go through other blocks in-between. An example is depicted in Fig. 4.13. Here, both "Block1" and "Block2" access the memory "M", where the access in "Block1" is a "ReadWrite" and the access in "Block2" is a "ReadOnly". As the "ReadWrite" must be performed before the "ReadOnly", the trigger is directed from "Block1" to "Block2".

Blocks accessing the same memory do not necessarily describe a memory-coupled dependency. There is a memory-coupled dependency only if at least one of the accesses is a "ReadWrite" or "Counter". A memory-coupled dependency describes the following

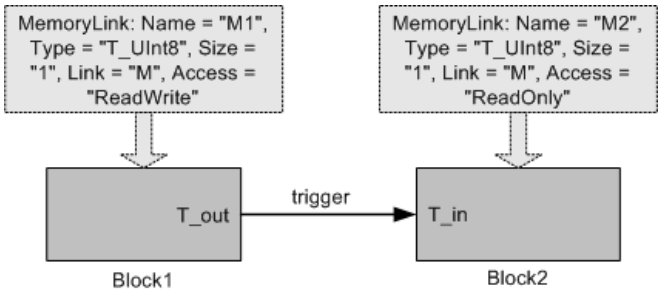


Figure 4.13: Example of memory-coupled dependency.

scheduling restriction: two blocks that have a memory-coupled dependency must not run in parallel, but sequentially in the order specified by the corresponding trigger flow (e.g., "Block1" before "Block2"). Note: the VRE compiler eliminates message-coupled dependencies that result for "Counter" type accesses, see Section 8.5.3.

4.4.3 Control-Coupled Dependencies

Control-coupled dependencies occur in loops and branches and describe predefined scheduling behaviors, e.g., a loop describes the iterative execution of blocks. VRE-specific loops and branches are discussed below.

Loops:

In general, loops can be categorized into three types depending on how their iteration limits (i.e., total number of iteration steps) are defined:

1. The iteration limit corresponds to a compile-time constant.
2. The iteration limit depends on a run-time value that is not set in the body of the loop and thus known prior to entering the loop.
3. The iteration limit depends on a run-time value that is set in the body of the loop.

For a compiler, these three types of loops carry three types of information. Type 1 enables the compiler to compute mapping and scheduling efficiently. For instance, assume that there is a loop whose iteration limit is defined by a compile-time constant and iteration steps are independent from one another, i.e., there is no data dependency between the iteration steps. Then, the compiler can decide at compile-time how many threads are needed to run the iteration steps in parallel. Type 2, in contrast, does not enable the compiler to take such a decision at compile-time. However, the compiler can insert additional code into the program, which can decide that at run-time right before executing the loop. The OpenMP compiler [6], for instance, uses this technique to parallelize some loops.

Correspondingly, VRE distinguishes three types of loops: *constant for loop*, *dynamic for loop*, and *do-while loop*. It is important to mention here that each loop in the VRE language is represented by a module.

The constant for loop is a loop whose iteration limit is known at compile-time. The VRE representation of such a loop is depicted in Fig. 4.14(a). Here, the loop is described by a composed module and the iteration limit is specified through a parameter named ("VRE_CONST_FOR"). In VRE, this parameter name is reserved to specify constant for loops, i.e., a module that has a parameter named "VRE_CONST_FOR" represents a constant for loop.

The dynamic for loop is a loop whose iteration limit is not known at compile-time but prior to entering the loop at run-time. The VRE representation of such a loop is depicted in Fig. 4.14(b). Here, the loop is described by a composed module and the iteration limit is specified through a memory link ("VRE_DYNAMIC_FOR"). Since a memory value cannot be predicted at compile-time, the iteration limit of a dynamic for loop is not known at compile-time. In VRE, the memory link name "VRE_DYNAMIC_FOR" is reserved to specify dynamic for loops, i.e., a module that has a memory link named "VRE_DYNAMIC_FOR" represents a dynamic for loop.

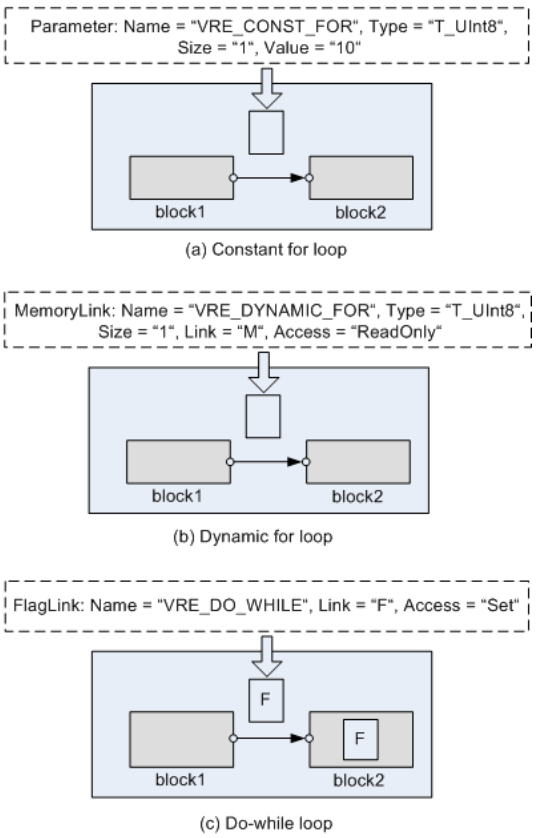


Figure 4.14: VRE representations of loops.

.

The do-while loop is a loop whose iteration limit depends on a run-time value that is produced from the body of the loop. The VRE representation of such a loop is depicted in Fig. 4.14(c). Here, block2 checks the condition by means of consuming data through its input port. If the condition becomes false, it sets the flag "VRE_DO_WHILE", and

thereby stops the loop. In VRE, the flag link name "VRE_DO_WHILE" is reserved to specify do-while loops, i.e., a module that has a primitive block which has a flag link named "VRE_DO_WHILE" represents a dynamic do-while loop.

Branches:

As shown in Fig. 4.15, branches in VRE may have three different forms: "CaseStatic", "CaseDynamicMemory", and "CaseDynamicMessage". For brevity, the figure contains only two branches, but more branches are allowed. For all types of branches, the required control (selection of the branch to be executed) is performed by a special block (called "Select"), and another special block (called "EndSelect") marks the end of the branch construct, i.e., connects the selected branch output to the input of another block if required.

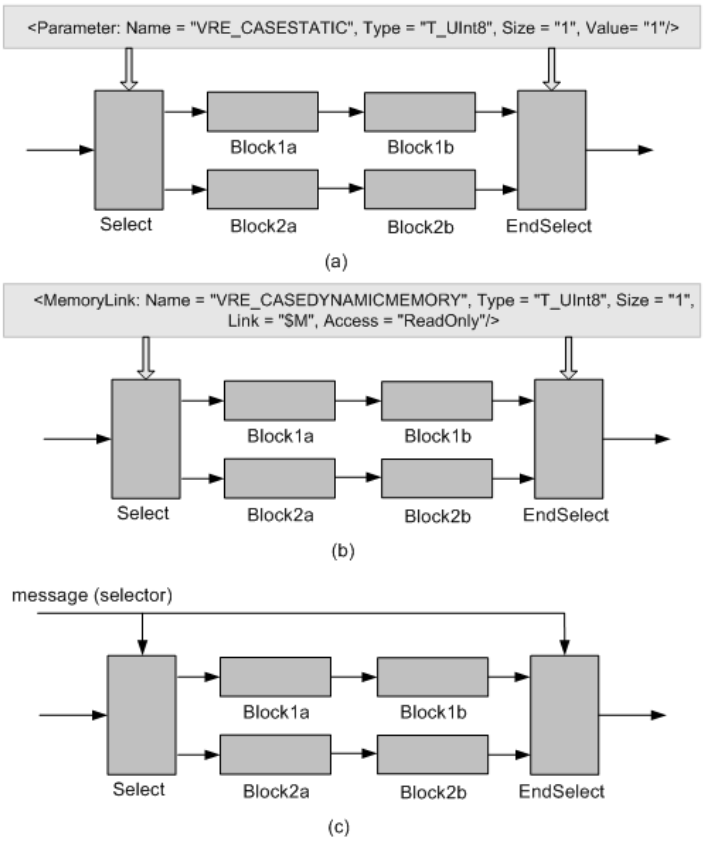


Figure 4.15: Different types of branch descriptions in VRE: (a) CaseStatic, (b) CaseDynamicMemory and (c) CaseDynamicMessage

.

For "CaseStatic", the branch is selected by a parameter that can be determined at compile-time. Thus, there is no real choice at run-time, but this case simplifies the description with more flexible blocks. During compile-time, this case is replaced with the selected branch. The execution of "CaseDynamicMemory" and "CaseDynamicMes-

sage", in contrast, depends on either a memory value or a message, which is not possible to determine at compile time.

4.5 PSM Description Concept

A PSM is a hardware-specific representation of an application. It describes scheduling, mapping and synchronizations. It builds up from tasks that may internally contain subtasks, and thus is a hierarchical description. An example is depicted in Fig. 4.16, which consists of four tasks, where "task3" internally contains two subtasks. Note that a task in a PSM looks similar to a block in a PIM, i.e., it describes interfaces such as input and output ports.

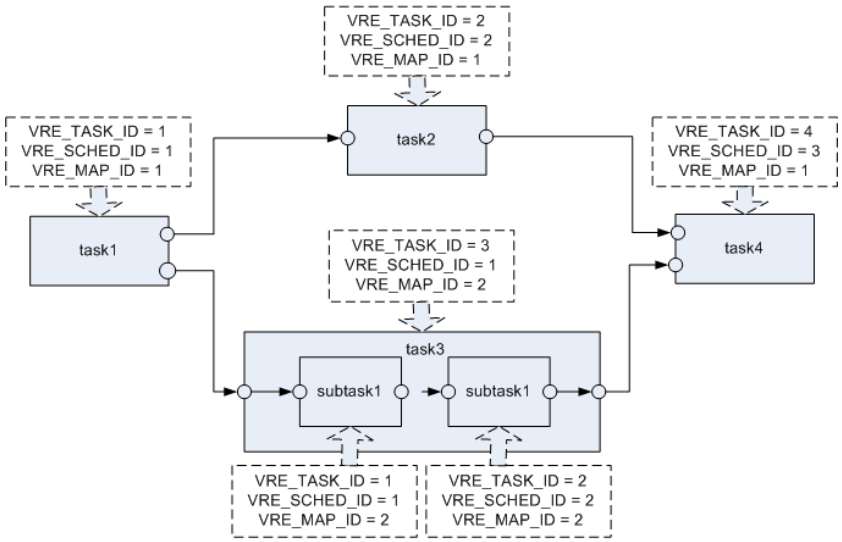


Figure 4.16: The description of scheduling and mapping of tasks in a PSM.

Every task in a PSM has three parameters: "VRE_TASK_ID", "VRE_SCHED_ID", and "VRE_MAP_ID". The parameter "VRE_TASK_ID" defines an ID. In a PSM, tasks are distinguished by their IDs, see Fig. 4.16 for instance. Note that the IDs of the subtasks ("subtask1" and "subtask2") are unique with respect to their supertask ("task3") only. The other two parameters specify scheduling and mapping. For instance, in Fig. 4.16, the values of parameter "VRE_MAP_ID" of task1, task2 and task4 are 1. This means, these tasks are mapped to a processing module with ID 1. Note that processing modules are also distinguished by their IDs, see Chapter 7. The values of parameter "VRE_SCHED_ID" of task1, task2 and task4 are 1, 2, and 3 respectively. This means, the processing module with ID 1 has to execute first task1, next task2, and then task3. Note that the PSM describes scheduling and mapping for both the tasks and subtasks.

Additionally, a task may have two more parameters named "VRE_EXEC_TIME" and "VRE_IMP_ID", which will be discussed in Chapter 7.

Loops and branches are represented both in a PSM and a PIM similarly, except they are represented in a PIM with blocks and in a PSM with tasks. For instance, a loop in a PIM is described by a modular block, and a loop in a PSM is described by a task that comprises subtasks. Another exception is that "CaseStatic" can only be described in a PIM but not in a PSM because there is no branch selection at run time, as already discussed in Section 4.4.3.

A PSM may include tasks for synchronizations, which are called synchronization tasks and inserted by the VRE compiler. For instance, assume that two tasks (T1 and T2) are mapped to two different processing modules (PM1 and PM2), but have dependency, i.e., T1 must be executed before T2. Then, some synchronization tasks need to be inserted in the program to ensure the execution of T1 before T2.

VRE defines a predefined set of primitives, called synchronization primitives, and the VRE compiler inserts synchronization tasks as instances of these primitives. It is important to restate here that each SDR platform defines its own set of functions for describing required synchronizations (see Section 3.1). Again, as already said, the dissertation concentrates on the hardware platform SB3010, i.e., a prototype version of the VRE compiler has been developed within the scope of the thesis for the SDR hardware platform SB3010. Thus, the synchronization primitives that are discussed next are specific to SB3010.

Within the scope of this thesis, four synchronization primitives have been defined and developed for SB3010:

vreInCounter: This primitive has a parameter ("P") and accesses a memory ("M"). Upon execution, it increments the value of "M" by the value of "P".

vreWaitOnCounter: This primitive has a parameter "P" and accesses a memory "M". Upon execution, it waits as long as the value of "M" does not become equal to the value of "P".

vreCondWait: This primitive has two parameters "InitialValue" and "Interval". It accesses a memory ("M"). Upon execution, it checks the following conditions: (1) m = i and (2) (m - i) MOD v = 0, where m = the value of "M", i = the value of "InitialValue", v = the value of "Interval", and, for any value x and y, x mod y = the reminder of x/y such as 5 MOD 2 = 1. The primitive returns if at least one of the conditions is true. Otherwise, it waits for a wakeup signal, where the wakeup signal is sent by a "vreCondSignal" primitive block (which is discussed next). Upon receiving a wakeup signal, it checks the conditions again. It returns if any of the conditions is true, otherwise waits again.

vreCondSignal: This primitive has two parameters, "Interval" and "Type". Upon execution, it accesses a memory ("M") to either increment or decrement the value of "M" by the value of "Interval" depending on the value of "Type", i.e., it increments if "Type" = 1 and decrements if "Type" = -1.

Additionally, each synchronization primitive accesses a flag. A common rule for synchronization primitives is that "several synchronization primitives that access the

same flag do not internally access any shared data (memory) simultaneously". For instance, if two synchronization tasks access the same flag and the same memory ("M") then they do not internally access "M" at the same time. Actually, the synchronization primitives ensure this with the help of the hardware-level supports of synchronization mechanism, see the next paragraph.

The SB3010 hardware supports two types of synchronization objects (i.e., means of performing synchronizations): *mutexes* allow multiple threads to share the same resource (such as a shared memory value) but not simultaneously, and *condition variables* enable multiple threads to wait until a particular condition occurs, see [5, 28] for details. The synchronization primitives internally call the hardware-specific functions, and perform required synchronizations using mutexes and condition variables. In particular, each synchronization primitive internally accesses a mutex and a condition variable. In VRE, this is described with the access to a flag by the synchronization primitive. In other words, a synchronization primitive accesses a flag means it internally accesses a mutex and a condition variable. Note that we exclude synchronization-specifics of the target hardware (such as mutexes and condition variables) from the primitives' definitions and thus from a PSM description.

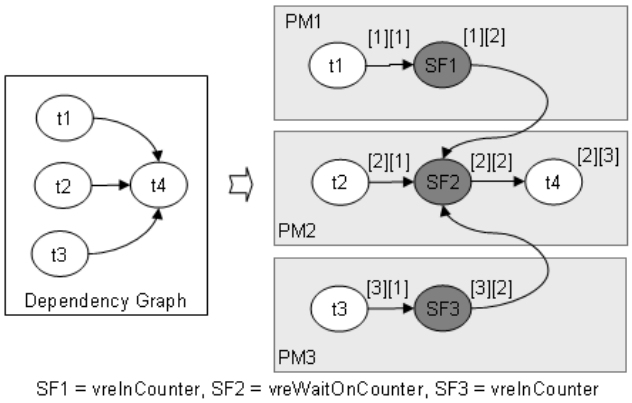


Figure 4.17: Synchronizing using the primitives vreInCounter & vreWaitOnCounter.

Fig. 4.17 presents an example that shows how the compiler ensures a desired execution order of tasks using the synchronization primitives "vreInCounter" and "vreWaitOnCounter". Here, as shown in the dependency graph, the task t4 depends on the tasks t1, t2 and t3. Since t1 and t3 are not mapped to the processing module PM2 along with t4, one cannot guarantee the execution of t4 after t1 and t3 without proper synchronization. Therefore, two synchronization tasks (SF1 and SF3) of type "vreInCounter" and another synchronization task (SF2) of type "vreWaitOnCounter" are inserted. For clarity, assume that:

- SF1, SF2, and SF13 access the same flag named "F".
- "M" is a memory whose initial value is 0.
- Both SF1 and SF3 access and thus increment the value of "M" by 1.

- SF2 checks the value of "M" and returns when the value of "M" is 2 (i.e., the value "P")

Thus, SF2 does not let PM2 to execute t4 as long as the value of "M" is not 2. Whereas, the value of "M" will only be "2" after the executions of SF1 and SF3. Thus, SF3 ensures the execution of t4 after SF1 and SF2, and hence after t1 and t3.

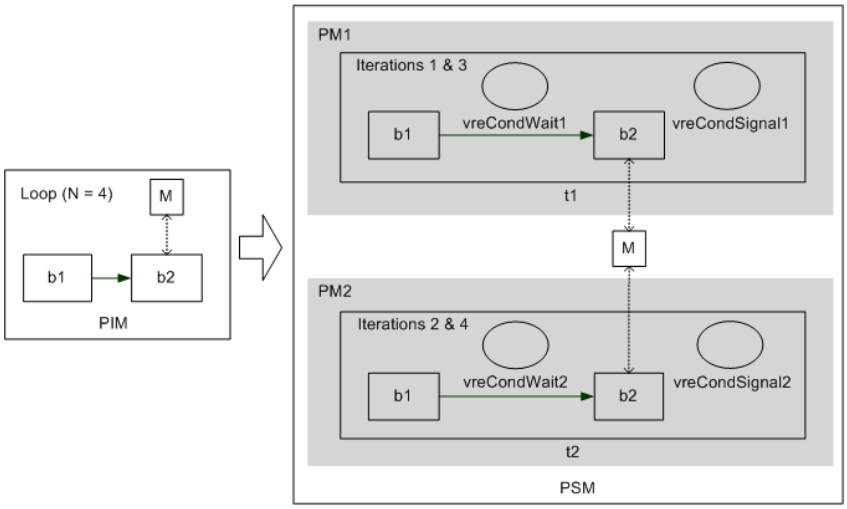


Figure 4.18: Synchronizing using the primitives vreCondWait and vreCondSignal.

Fig. 4.18 presents another example that shows how the compiler kernel ensures a desired execution order of tasks using the synchronization primitives "vreCondWait" and "vreCondSignal". Here, the loop that consists of four iterations ($N = 4$) has been broken down into two tasks*: t1 corresponds to the 1^{st} and 3^{rd}, and t2 corresponds to the 2^{nd} and 4^{th} iteration steps, where t1 and t2 are mapped to the processing modules PM1 and PM2, respectively. As b2 has "ReadWrite" access to the memory "M", it must be executed sequentially in different iteration steps. Therefore, two synchronization tasks have been inserted in t1, i.e., *vreCondWait1* before b2 and *vreCondSignal1* after b2". Likewise, two synchronization tasks (*vreCondWait2* and *vreCondSignal2*) have been inserted in t2. These synchronization tasks access a memory named "M" and a flag named "F". Additionally, they are initialized as below:

- vreCondWait1's parameters "InitalValue" and "Interval" are initialized as 0 and 2, respectively.
- vreCondSignal1 increments the value of "M" by 1.
- vreCondWait2's parameters "InitalValue" and "Interval" are initialized as 1 and 2, respectively.
- vreCondSingal2 increments the value of "M" by 1.

*See Section 8.5 to know how the VRE compiler creates tasks from loops.

Therefore, as shown in Table 4.2, PM1 and PM2 execute the instances of b2 of different steps sequentially in the desired order, i.e., PM1 executes b2 of the 1^{st} iteration step before PM2 executes b2 of the 2^{nd} iteration step, and PM1 executes b2 of the 3^{rd} iteration step before PM2 executes b2 of the 4^{th} iteration step. For further clarity, the activities of the processing modules at time stamps 1 to 6 are discussed below with more details:

Table 4.2: The executions of the tasks in different processing modules.

Time	PM1	PM2
1	b1 (iteration 1)	b1 (iteration 2)
2	vreCondWait1	vreCondWait2
3	b2 (iteration 1)	vreCondWait2
4	vreCondSignal1	vreCondWait2
5	b1 (iteration 3)	vreCondWait2
6	vreCondWait1	b2 (iteration 2)
7	vreCondWait1	vreCondSignal2
8	vreCondWait1	b1 (iteration 4)
9	b2 (iteration 3)	vreCondWait2
10	vreCondSignal1	vreCondWait2
11	-	vreCondWait2
12	-	b2 (iteration 4)
13	-	vreCondSignal2

Time 1 PM1 executes the instance of b1 of the 1^{st} iteration step and PM2 executes the instance of b1 of the 2^{nd} iteration step, respectively.

Time 2 PM1 executes "vreCondWait1" and PM2 executes "vreCondWait2". As the value of "M" is equal to "0", the 1^{st} condition is true for "vreCondWait1". Whereas, neither of the conditions is true for "vreCondWait2".

Time 3 PM1 executes the instance of b2 of the 1^{st} iteration step and PM2 still executes "vreCondWait2" that is waiting for a wakeup signal.

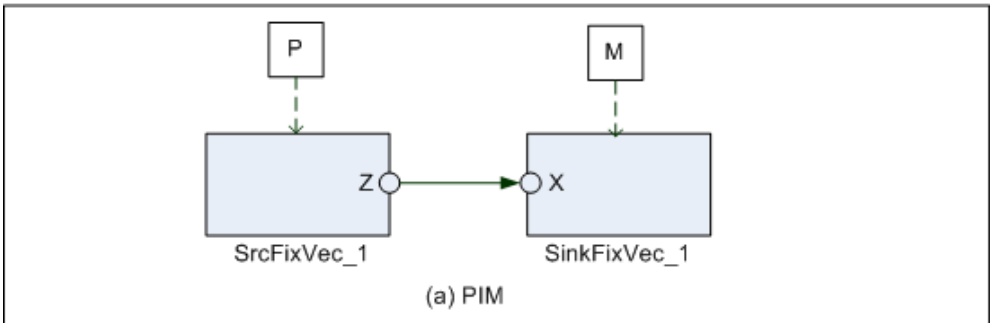
Time 4 PM1 executes "vreCondSignal1" that increments the value of "M" from "0" to "1" and sends a wake up signal. On the other hand, PM2 still executes "vreCondWait2". At this time stamp, PM1 finishes the execution of the 1^{st} iteration step.

Time 5 PM1 executes the instance of b1 of 3^{rd} iteration step. The "vreCondWait2" running on PM2 at this point receives the wakeup signal and evaluates the conditions. Here, the 1^{st} condition is true for "vreCondWait2" since the value of "M" is "1".

Time 6 PM1 executes vreCondWait1, where none of the conditions for that is true. PM2 executes the instance of b2 of the 2^{nd} iteration step.

4.6 Storing VRE Programs

VRE programs, both PIM and PSM, are textually stored in Extensible Markup Language (XML) format, because XML's tag-based program representation concept is suitable to store information that consists of elements that may consist of other elements and have attributes. VRE programs are composed of elements, such as blocks and signals. Some of these elements may consist of other elements, e.g., modular blocks consist of primitives and/or other modular blocks, and have attributes, e.g., flags have attributes called name and value. As an example, Fig. 4.19 presents a PIM in both graphical and XML format.




```
<system name= "VRE-Test">

 <parameter name = "P" type = "T_UInt8" size = "1" value = "10"/>
 <memory name = "M" type = "T_UInt8" size = "1" value = "0"/>

 <block name= "SrcFixVec_1" class = "SrcFixVec">
  <outputport name = "Z" type = "T_UInt8" size = "1" />
  <parameter name = "PrtLen_Z" type = "T_UInt8" size = "1" value = "$P"/>
 </block>

 <block name= "SinkFixVec_1" class = "SinkFixVec">
  <inputport name = "X" type = "T_UInt8" size = "1"/>
  <memorylink name = "M1" type = "T_UInt8" size = "1" link = "$M" access = "ReadOnly"/>
 </block>

 <signal name = "Signal1">
  <producer  port = "SrcFixVec_1.Z"/>
  <consumer port = "SinkFixVec_1.X"/>
 <signal/>

</system>
```

(b) XML representation of PIM

Figure 4.19: An example PIM and its corresponding XML representation.

.

Additionally, storing programs in XML format provides some benefits:

1. Information coded in XML is easy to read and understand, plus it can be processed and parsed easily by computers.

2. Information formatted in XML can be exchanged across platforms, languages, and applications, and can be used with a wide range of development tools and utilities.

3. Information expressed in XML can be visualized conveniently in browsers using the Extensible Stylesheet Language (XSL) (see [52]).

Like VRE, several signal processing application development environments store programs in XML such as MLDesigned [27]. XML is also widely used in various other fields of SDR, e.g., for the management of hardware reconfiguration data. See [11, 30, 41, 43, 44] for more information about various uses of XML in the SDR domain.

4.7 Summary

The chapter has introduced the VRE language. In particular, it has discussed reasons for choosing a visual language, described syntax and semantics of the language, and presented PIM and PSM description concepts. The next chapter provides guidelines to use Simulink for representing a PIM.

Chapter 5

VRE Program in Simulink

This chapter first discusses reasons for integrating Simulink in the VRE tool chain, then presents Simulink, and finally describes guidelines for using Simulink to represent a PIM.

5.1 Why Simulink

Within the scope of this thesis, we have developed the VRE language, but not a dedicated application description environment for VRE due to time limitation. Instead we use Mathworks' software package Simulink [42] to describe a PIM as:

- Simulink provides an interactive graphical environment to design, simulate, implement, and test signal processing applications.
- Simulink offers tight integration with the Matlab environment [24], i.e., it can either be scripted from or drive Matlab. This has some benefits that will be discussed later in this chapter.
- Simulink is widely used by the communication industries, and thus the incorporation of Simulink in the VRE tool chain will increase the intuitive understanding of applications and simplify familiarization for new developers.

5.2 Introduction to Simulink

This section presents Simulink, and discusses how applications are described in Simulink. It only includes Simulink-specifics that are required to understand guidelines for describing a PIM in Simulink, which will be discussed after this section.

Simulink offers its own application description language and includes a graphical editor to describe applications. The language differs from VRE as it is not hardware-independent, i.e., developers describe an application with scheduling. It defines two major classes of elements to describe an application:

- *Blocks* that correspond to operations (such as filter) and are available to application developers as libraries. They are considered as black boxes, i.e., their function-

alities are unknown to application developers, but interfaces such as parameters and ports are known.

- *Signals* that correspond to exchanges of information between blocks.

In Simulink, an application description is called a *model*. To describe a model, developers first add required blocks from the block libraries to the graphical editor (also called the model editor) and then connect them with signals. See Fig. 5.1 for clarity. Here, the model consists of four blocks ("Signal Generator", "Sine Wave", "Product", and "Power Spectral Density") and three signals (shown as arrows).

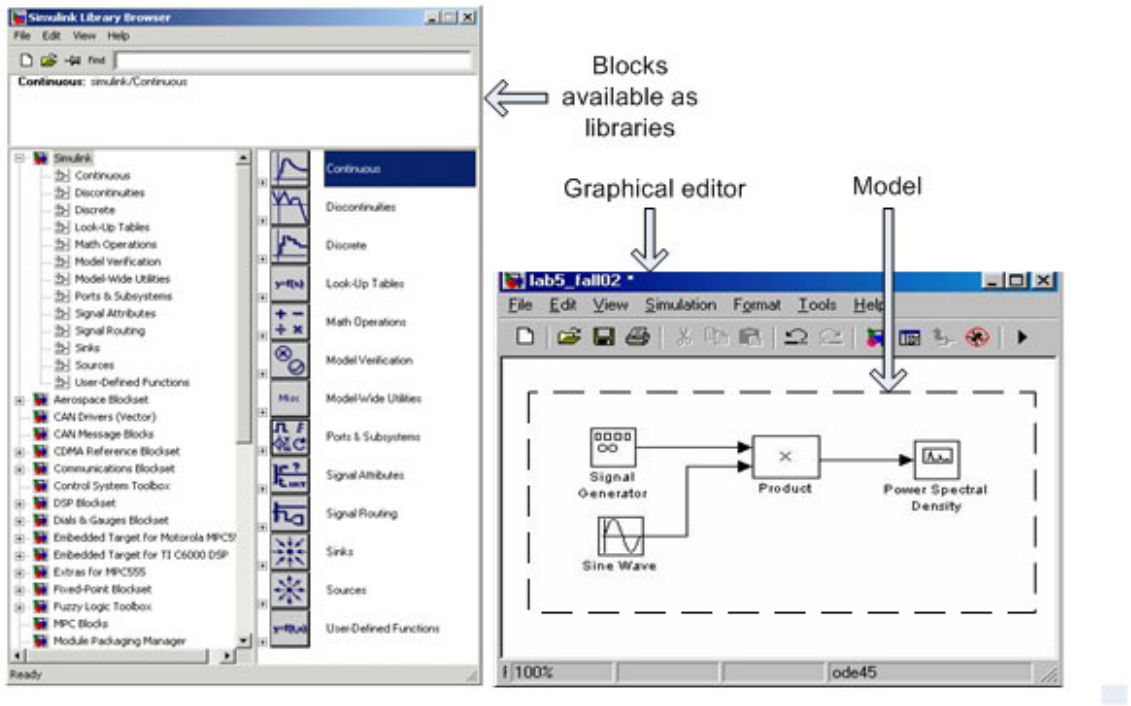


Figure 5.1: Application description in Simulink.

.

Simulink includes a simulator that executes (simulates) a model as a time-varying system. Prior to simulating a model, one has to set a start time, e.g., *T1*, and an end time, e.g., *T2*. The simulator then executes the model for the period of *T2-T1* in such a way that it executes every block of the model in each simulation time steps, i.e., *T1*, *T1+1*, ..., *T1+N* where *T1+N* <= *T2*. However, some blocks may execute iteratively or may not execute in a simulation time step, such as conditionally executed subsystem blocks, which will be discussed later.

Each block in Simulink has a parameter pane, i.e., a dialog window for defining parameters of a block. We can open the parameter pane of a block by clicking onto the block. Fig. 5.2(d), for instance, shows the parameter pane of an S-Function block that will be discussed in the next paragraph.

Simulink's block libraries include a set of built-in blocks (such as filter) that are commonly used to represent signal processing applications. Additionally, it is possible

to add a custom-defined block to a model. Simulink provides a special library block called S-Function block for this, i.e. an S-Function block is capable of running an S-Function that is a user defined function written in: C, C++, Matlab and Fortran. To add an S-Function block to a model, as shown in Fig. 5.2, we have to first describe an S-Function and next add an S-Function library block to a model. Then, to associate the S-Function to the S-Function block, we have to open the parameter pane of the S-Function block and define two parameters: the parameter "S-function name" with the name (reactor_sfcn) of the S-Function, and the parameter "S-function parameters" with the arguments (0, 1, 40) of the S-Function, e.g., if *f(int p1, int p2)* is an S-Function then *p1* and *p2* are the arguments of the S-Function.

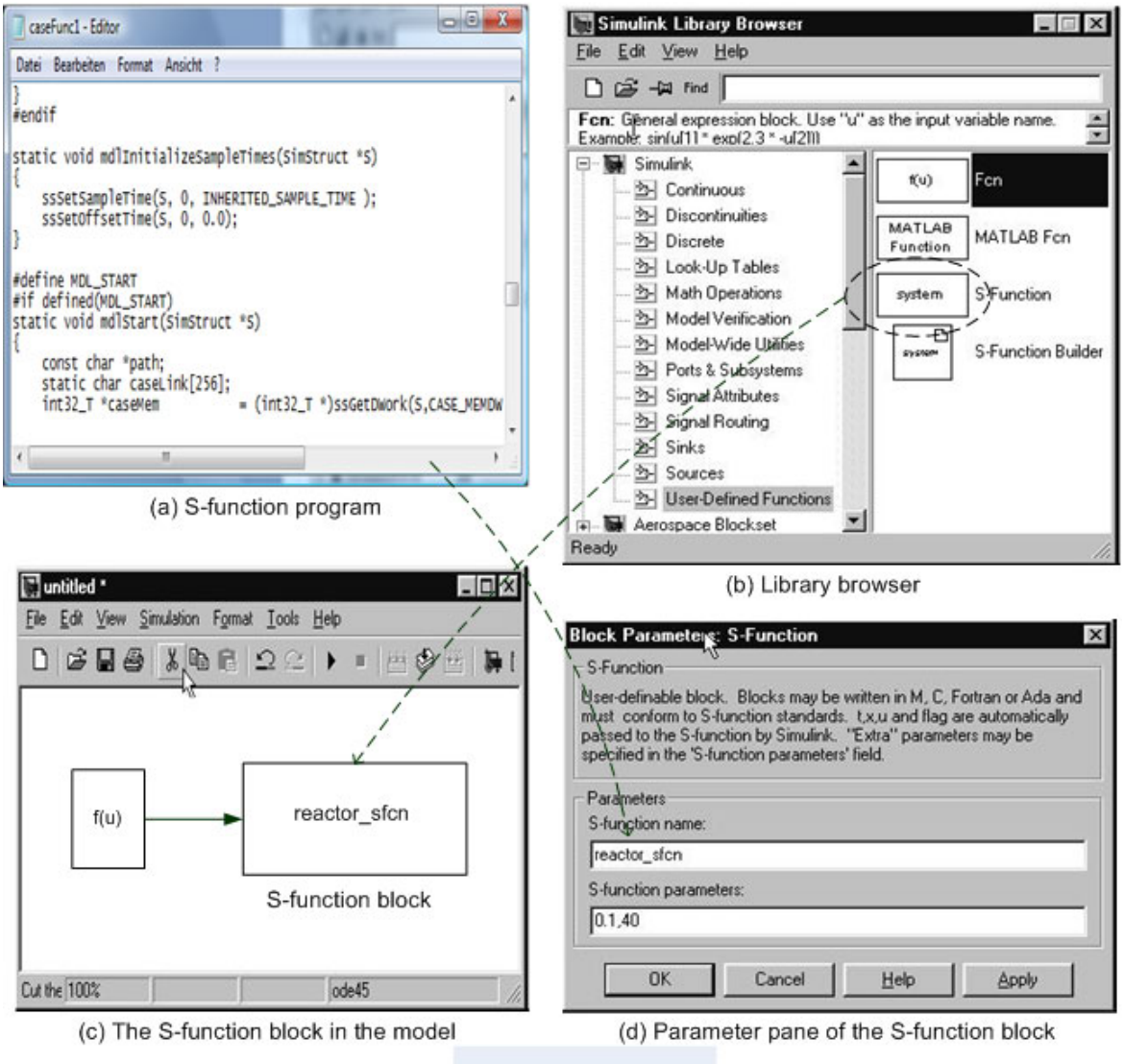


Figure 5.2: Adding S-Function to a model.

Simulink also provides a special block called subsystem to establish a hierarchical

block diagram, i.e., a subsystem block is one layer and the blocks that lie within the subsystem block are another. We insert a subsystem block to a model to establish a hierarchical block diagram as follows:

- First, we have to add a subsystem block from the block library to our model editor.
- Next, we have to click onto the subsystem, which in turn opens a different model editor window.
- Then, we have to add required blocks for the subsystem blocks to the new model editor window.
- Finally, we have to connect those blocks with signals and thus can describe the internal block diagram of the subsystem.

As shown in Fig. 5.3, Simulink provides a library block named Inport ("In") to realize an input data port and another library block named Outport ("Out") to realize an output data port of a subsystem block ("Subsystem"). Note that the "Trigger" block also realizes an input port that is not connected by signal with any other internal block of the subsystem block (see Fig. 5.3(b)). This block actually determines whether or not the other internal blocks of the subsystem block may execute in a simulation time step, as will be discussed next.

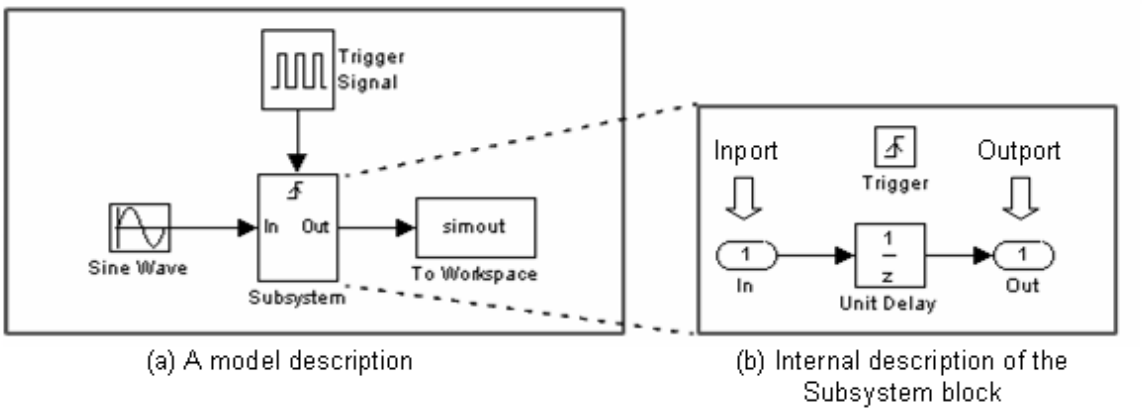


Figure 5.3: A subsystem block in a model.

In Simulink, a subsystem block can be executed unconditionally or conditionally. An unconditionally executed subsystem block always executes. A conditionally executed subsystem block internally contains a control block that evaluates a condition and, based on the condition, determines whether or not other internal blocks of the subsystem block may execute. For instance, the "Trigger" block in Fig. 5.3 is a control block, and therefore the subsystem block is a conditionally executed subsystem block. Here, in each simulation time step, the trigger block consumes an input signal, and let the other internal blocks of the subsystem block to execute if the input signal changes (either increments or decrements).

Simulink also provides library blocks to describe memories. It provides a library block named "Data Store Memory" to define a memory, and two other library blocks, named "Data Store Read" and "Data Store Write", to read from and write to the memory, respectively. See Fig. 5.4 for instance. Here, the block "Data Store Memory" represents a memory ("M1"), where the block "B1" reads from and the block "B2" writes to the memory.

It is possible in Simulink to create our own block libraries. For instance, we can design various subsystem blocks and develop our own S-Function blocks, and then can store them as libraries.

Simulink offers tight integration with Matlab, i.e., Matlab offers a command shell and defines a set of commands, where one can execute a command in the command shell to access or set the functionalities of a model. We can, for instance, execute a command to obtain the properties of a block. Section 5.4 discusses further details.

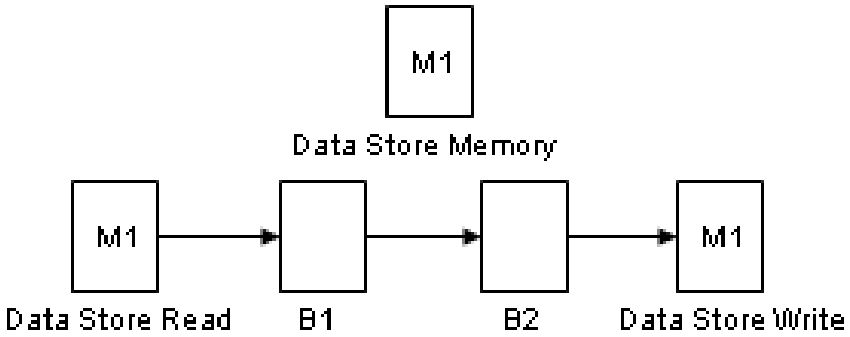


Figure 5.4: Memory blocks in Simulink.

Simulink supports the concept of callback functions. A callback function is a scrip file (or M-file) that is written in the Matlab language. Developers can associate a callback function with a model, where the model executes the callback function based upon a specific action on the model, e.g., during simulation but before any other block. A use of callback function is discussed later in this chapter.

5.3 Guidelines for Representing PIM

There is a similarity between Simulink and VRE, i.e., we describe an application as a block digram in both Simulink and VRE. To describe a PIM in Simulink, we have to consider the following:

- A model in Simulink corresponds to a system in VRE, and blocks and signals have similar meaning in both VRE and Simulink.
- A library block in Simulink, which is not a subsystem block, corresponds to a primitive in VRE.
- A library block in Simulink, which is a subsystem block, corresponds to a module in VRE, and can be said as a modular block with respect to VRE.

As already stated, we describe a PIM in terms of different types of dependencies between blocks, i.e., message-coupled, memory-coupled, and control-coupled dependencies. Thus, to represent a PIM in Simulink, we have to add required blocks to Simulink's model editor window and then describe different types of dependencies between them. We can for instance connect two blocks with a signal that corresponds to an exchange of data to describe a message-coupled dependency. Unfortunately, Simulink does not enable us to describe some VRE-specifics directly, e.g, it does not support VRE-like flags. However, some workarounds have been devised within the scope of this thesis so that we can describe a PIM in Simulink, which are discussed below.

Describing Parameters at System Level

In Simulink, there is no direct way to specify a global parameter for a model. Fortunately, it is indirectly possible using a callback function named "PreLoadFnc" that is executed by a model before its blocks (during simulation). To add parameters to a model, we have to first describe required parameters in a script file that is written in the Matlab language and then execute the following command in the Matlab command shell:

set_param('modelname','PreloadFcn', 'loadvar')

This command (set_param) ensures that a model named "modelname" will execute a script file named "loadvar" as a callback function named "PreLoadFnc" during simulation prior to any other blocks. Thus, during simulation. the model first executes the callback function and thereby loads the parameters that are global with respect of the model and can be accessed by blocks.

Specifying Memories and flags

As shown in Fig. 5.4, Simulink's concept of memory differs from that of VRE, i.e., a memory in Simulink is described by a "Data Store Memory" block and can only be accessed by a "Data Store Read" block for reading and a "Data Store Write" block for writing, whereas a memory in VRE is not described as a block, instead as a memory component that is accessed by a block through a memory link. On the other hand, Simulink does not support flags. Therefore, a new concept for realizing VRE-like memories and flags in Simulink has been developed: An S-Function named "mem" has been written that defines a run time variable for a model to be accessed by other S-Function blocks, and takes the name, data type, size and initial value of a corresponding memory as arguments. If its type argument is specified as boolean then it corresponds to a flag else a memory. A block can access such a memory or flag by calling some Simulink-specific Dynamic Library Link (DLL) functions with the memory or flag name as argument. For instance, assume we describe a memory named "M1" as an S-Function block that associates the S-Function "mem". Then, another block can access the memory by calling a DLL function with "M1" as argument. See [53] to know about these DLL functions and other details. It is important to mention here that, in Simulink, blocks that access memories and flags take the names of the memories and flags as parameters, as Simulink supports parameters for blocks but not memory and flag links. Therefore, we have to

properly mask parameter panes of blocks to distinguish parameters, memory links and flag links from one another, as will be described next.

Masking Parameter Panes

We have to mask (i.e., customizing a user interface) parameter panes of blocks as shown in Fig. 5.5, because they do not contain sufficient information for developers. For instance, they describe names of parameters but not other properties such as data types and sizes. Furthermore, as discussed earlier, they do not distinguish parameters, memory links and flag links from one another. See "Mask Editor" Section in the Simulink documentation (available at [42]) for further details about how to mask parameter panes of blocks.

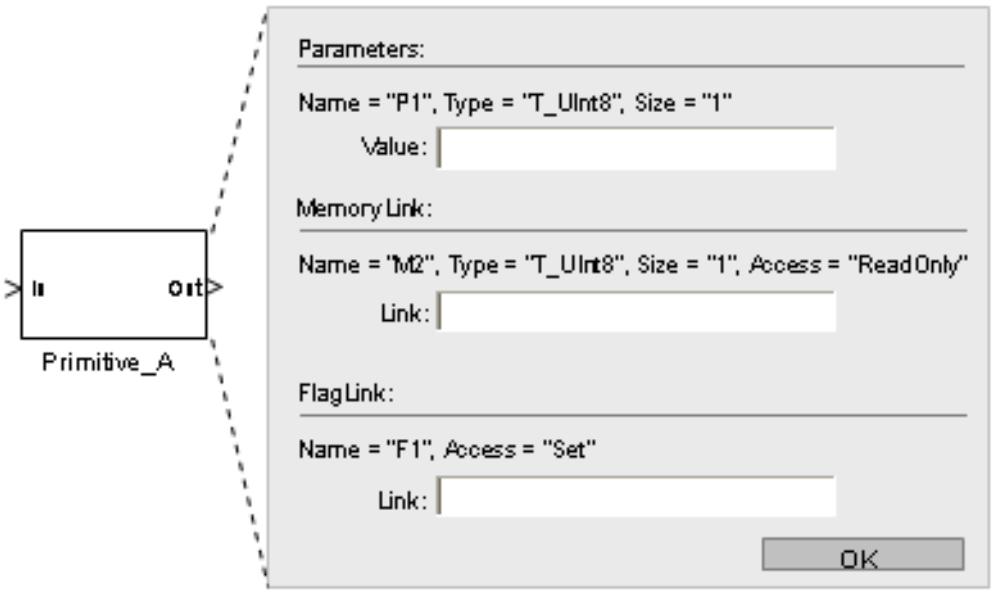


Figure 5.5: An S-Function primitive block and its parameter pane.

Realizing Trigger Flows

As already discussed in Section 4.4, in VRE, one block triggers another may describe a memory-coupled dependency. However, Simulink does not enable us to describe output trigger ports for subsystem blocks, as well as input and output trigger ports for S-Function blocks. Therefore, it is not directly possible to represent VRE-like trigger flows in Simulink. Nevertheless, we can use triggered subsystem blocks (already introduced in Section 5.2) to represent VRE-like trigger flows as shown in Fig. 5.6. Note that a Trigger block (already introduced in Section 5.2) realizes an input trigger port, and an Outport block (also introduced in Section 5.2) named "Z" realizes an output port. As the Outport block itself does not produce any output trigger, a library block named

"Constant" that generates an impulse upon execution and hence produces a trigger output is connected with the Outport block.

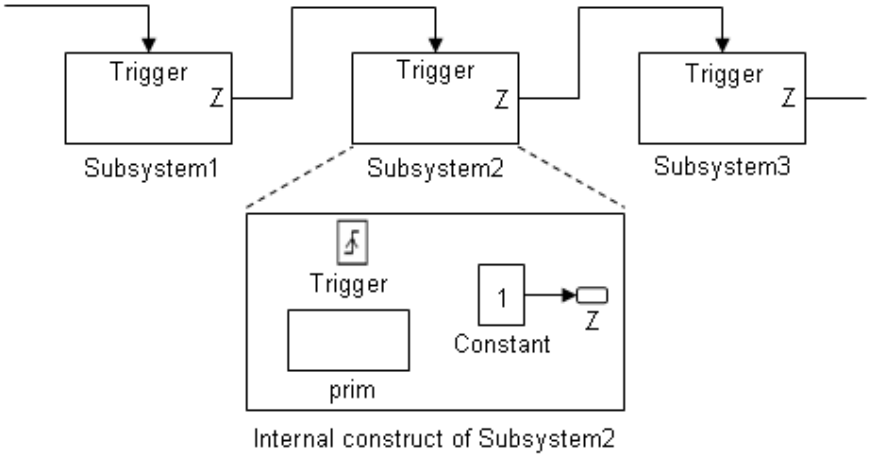


Figure 5.6: Representation of trigger flows in Simulink.

Representing VRE-Like Loops

Simulink provides two control blocks for describing loops as conditionally executed systems, i.e. a control block named "For Iterator" and another control block named "While Iterator". A subsystem block that contains a "For Iterator" block is called a *for iterator subsystem block* and a "While Iterator" block is called *a while iterator subsystem block*. In a simulation time step, both for and while iterator subsystem blocks execute their internal contents iteratively, and their iteration limits (i.e., total number of iterations) are determined by their "For Iterator" and "While Iterator" blocks, respectively.

We can use a for iterator subsystem block to describe VRE's constant and dynamic for loops. A "For Iterator" block can be parameterized to enable us to specify a corresponding iteration limit in two ways, either as a parameter value or as a data that is produced by another block and read through an input port. We use the first option to describe a VRE-like constant for loop and the second to describe a VRE-like dynamic for loop. See Fig. 5.7 for clarity. Here, in case of dynamic for loop, the block named "M_Val" reads a memory and thus produces the value of the memory as output that defines the iteration limit.

We can, on the other hand, use a while iterator subsystem block to describe a VRE-like do-while loop, where its "While Iterator" block determines its iteration limit, i.e., this block consumes an input through port in each simulation time step and may stop the loop as soon as the input corresponds to zero. See Fig. 5.7 for clarity. In VRE, as discussed in Section 4.4.3, a block that is within a do-while loop accesses a flag named "VRE_DO_WHILE" and thus stops the loop. In Fig. 5.7, it is the "condCheck" block that accesses the flag (not shown in the figure).

As shown in Fig. 5.7, descriptions of loops in Simulink comprise two levels of hierarchies: one by a triggered subsystem that is to describe input/output trigger ports, and another by either a for or a while iterator subsystem block that describes the body of a loop, as Simulink does not permit to coexist two control blocks (such as "Trigger" and "For Iterator") at a same hierarchical level.

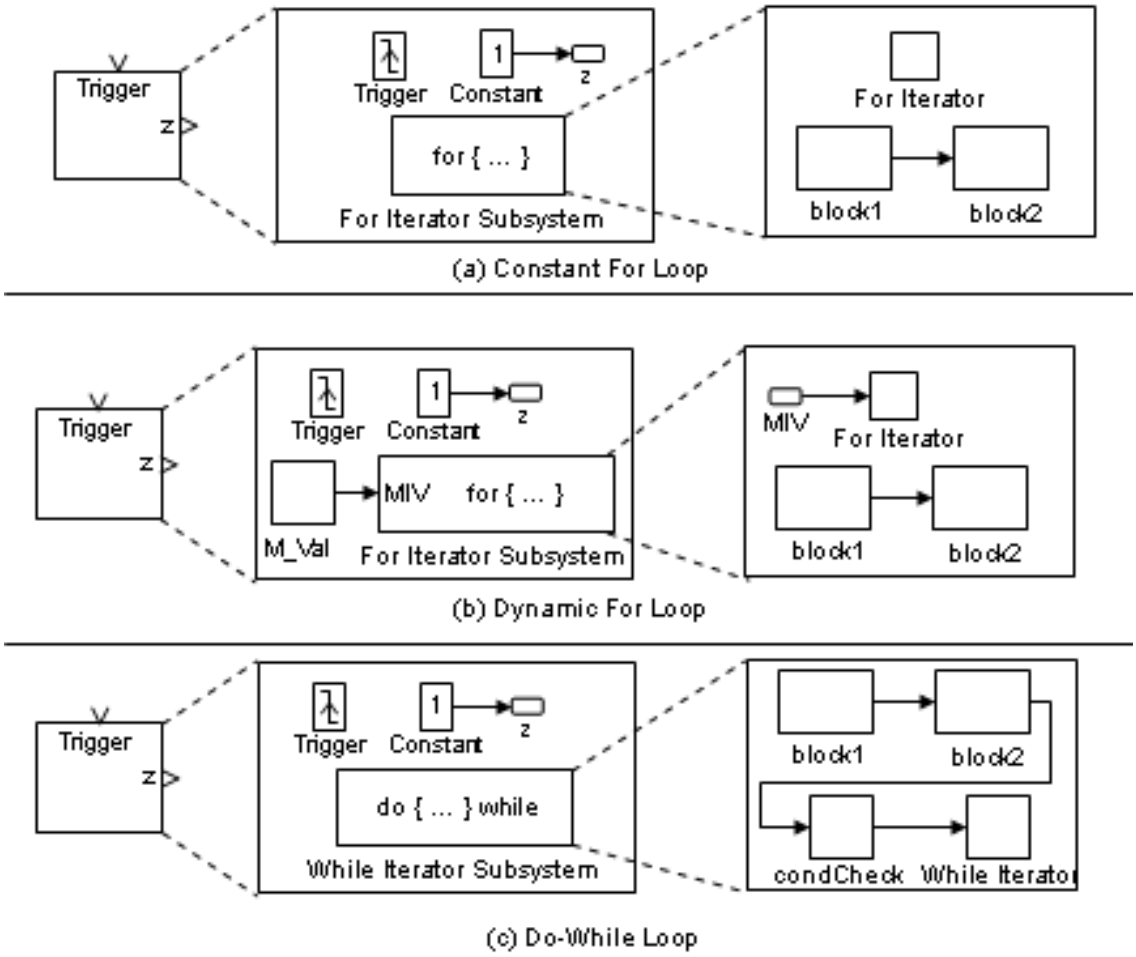


Figure 5.7: Representations of VRE-like loops in Simulink.

Representing VRE-Like Branches

To specify different types of VRE-like branches, six S-Function primitive blocks have been developed, i.e., three "Select" and three "EndSelect" primitives, where each combination of "Select" and "EndSelect" primitives is used to describe one of the three types of VRE-like branches, as already discussed in Section 4.4.3.

In Simulink, VRE-like branches are described within a triggered subsystem, where the S-Function block that performs the "Select" operation activates one of the triggered subsystems and the S-Function block that performs "EndSelect" operation determines

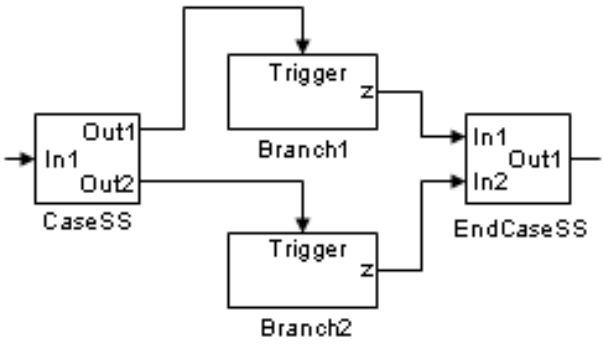


Figure 5.8: Representation of VRE-like CaseStatic branch in Simulink.

the end of the branch. For example, Fig. 5.8 shows a VRE-like "CaseStatic" branch representation in Simulink, where the blocks "CaseSS" and "EndCaseSS" represent "Select" and "EndSelect" primitive operations, respectively. Note that "Branch1" and "Branch2" are subsystem blocks, and the bodies of the branches are described internally within these subsystems.

5.4 XML Representation of PIM

As already discussed in Section 4.6, VRE programs (both PIM and PSM) are stored in XML format to be input to the VRE compiler. Within the scope of this thesis, a Matlab program has been developed that is capable of generating a PIM in XML format from a model representing the PIM. It has to be executed from the Matlab command shell and takes a model name as an argument. Upon execution, it first gathers information about the model and then generates a corresponding XML file. It is composed of Matlab-specific commands, i.e., Matlab provides a set of commands that can be used to gather information about a model. In particular, it uses two such commands: "find_system" and "get_param". The "find_system" expression returns the handle of an object (i.e., blocks, lines, etc.) of the model. The syntax of this expression is:

$$find\_system(s,'c1','c2',...,v1,v2,...)$$

This searches the model (or a subsystem) named "s" using the constraints specified by "c1", "c2", etc., and returns handles of the objects (such as blocks) having the specified parameter values "v1", "v2", etc. For example, assuming a model contains a block named "GainBlock" that has a parameter named "Gain" whose value is "1", we can obtain the handle of the "GainBlock" block using the following expression:

$$blk = find\_system(gcs,'BlockType','GainBlock','Gain','1')$$

Here, "gcs" is a predefined Matlab variable that specifies the path of the current model.

After obtaining the handle of an object, we can use the "get_param" command to gather information about the object. For instance, we can use the following expression to obtain information about the parameter pane of the "GainBlock" block:

$$rval = get\_param(blk,'DialogParameters')$$

Here, the valiable "blk" that contains the return value of the previously mentioned "find_system" expression refers to the object handle of the "GainBlock" block. The variable "rval" is a cell array that is a Matlab-specific data structure capable of preserving the information (see [24]).

Thus, the program gathers information about a model using Matlab command expressions. Then, it writes this information to an XML file that corresponds to a PIM. Matlab provides two functions called "xmlread" and "xmlwrite" to read and write an XML file. Using the "xmlwrite", the program writes the gathered information.

5.5 Summary

This chapter has presented Simulink and the guidelines for representing a PIM in Simulink. The next chapter describes an experiment that has been conducted to represent the IEEE 802.11b WLAN receiver as a PIM in Simulink.

Chapter 6

WLAN Experiment

This chapter gives an overview of the IEEE 802.11b WLAN and describes an experiment that has been performed to represent a PIM for the physical layer of the IEEE 802.11b WLAN receiver in Simulink.

6.1 Overview of WLAN Physical Layer

The IEEE 802.11 standard [7] defines three different types of Physical Layer (PHY) implementations: (1) infrared (IR) baseband, (2) Frequency Hopping Spread Spectrum (FHSS) PHY, and (3) Direct Sequence Spread Spectrum (DSSS) PHY [7]. The PHY that is discussed in this chapter is the DSSS PHY, also known as 802.11b.

PHY resides at the bottom of the OSI model. It acts as an interface between the Medium Access Control (MAC) layer and the wireless media. As shown in Fig. 6.1, it consists of a Physical Layer Convergence Procedure (PLCP) sublayer and a Physical Medium Dependent (PMD) sublayer.

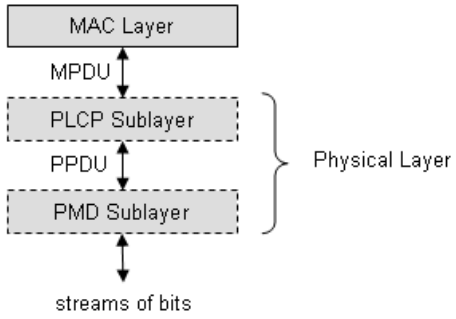


Figure 6.1: Different functional layers of WLAN protocol.

The PLCP sublayer exchanges packets with the MAC layer. On the transmitter path, it receives packets from the MAC layer in the format of MAC Protocol Data Unit (MPDU) and forwards them to the PMD sublayer in the format of PLCP Protocol Data Unit (PPDU). On the receiver path, it receives packets from the PMD sublayer in the format of PPDU and forwards them to the MAC Layer in the format of MPDU.

The PMD sublayer on the other hand transmits and receives bits over the wireless medium. On the transmitter path, it takes binary bits of information in the format of PPDU from the PLCP sublayer and transforms that into RF signals to be transmitted through the wireless media. On the receiver path, it takes the received data and converts that to PPDU to be forwarded to the PLCP layer.

Thus, PHY has the following functionalities: (1) it provides a frame exchange between the MAC and PHY under the control of the PLCP sublayer, (2) it uses signal carriers and spread spectrum modulation to transmit data frames over the media under the control of the PMD sublayer, and (3) it provides a carrier sense indication back to the MAC to verify activity on the media [29].

6.2 Overview of WLAN Receiver

The WLAN receiver transforms incoming RF signals into PPDUs. The PPDU frame format is presented in Fig. 6.2. It consists of a PLCP preamble, a PLCP header and an MPDU.

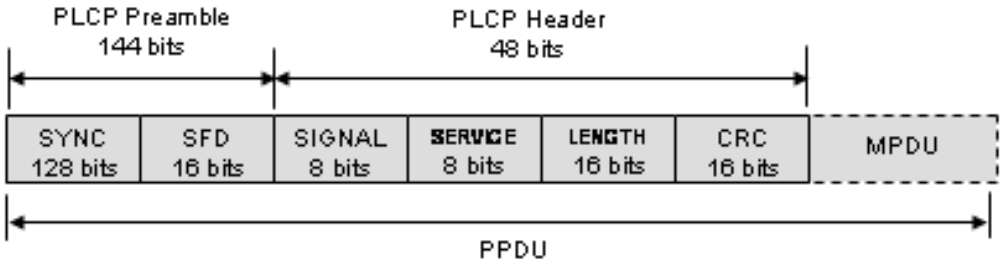


Figure 6.2: The PPDU frame format.

The PLCP preamble consists of the followings:

SYNC: This field is 128 bits (symbols) in length and contains a string of 1s which are scrambled prior to transmission. The receiver uses this field to acquire the incoming signal and synchronize the receiver's carrier tracking and timing prior to receiving the start of frame delimiter (SFD).

Start of Frame Delimiter (SFD): This field contains information marking the start of a PPDU frame. The SFD specified is common for all IEEE 802.11 DSSS radios and uses the following hexadecimal word: F3A0.

The PLCP header consists of the followings:

Signal: The signal field defines which type of modulation must be used to receive the incoming MPDU. The binary values in this field are equal to the data rate multiplied by 100kbit/s. For instance, in the June 1997 version of IEEE 802.11 [29], two rates are supported. They are: 0A hex for 1 Mbps DBPSK and 14 hex for 2 Mbps DQPSK.

Service: The service field is reserved for future use. However, the default value is 00 hex.

Length: The length field is assigned an unsigned 16-bit integer that indicates the number of microseconds necessary to transmit the MPDU. The MAC layer uses this field to determine the end of a PPDU frame.

CRC: The CRC field contains the result of a calculated frame check sequence from the sending station. The calculation is performed prior to data scrambling. The CCITT CRC- 16 error detection algorithm is used to protect the signal, service and length fields. The CRC-16 is represent by the following polynomial: $G(x) = x^{16} + x^{12} + x^5 + 1$. The receiver performs the calculation on the incoming signal, service, and length fields and compares the results against the transmitted values. If an error is detected, the receiver's MAC makes the decision if the incoming PPDU should be transmitted.

The MPDU consists of a header and a payload, which is understandable by the MAC layer. This frame format is unknown to PHY.

In the signal processing chain, the receiver first acquires samples that represent incoming signals. Then, it looks for the preamble SYNC pattern that is scrambled 1s. As soon as it detects the preamble SYNC pattern, it looks for the the start of the frame delimiter (SFD) pattern that marks the start of the PPDU frame. Once the SFD pattern is found, it decodes the header, i.e., it first parses the header bytes to obtain the data rate defined in the signal field and the packet length defined in the length field, and then determines the CRC information, and thus performs error checking. It discards the packet if it detects an error in the packet, otherwise it demodulates the incoming signal and thereby determines the payload using the known values of the data rate and length (these are given in the packet header).

In addition to the functionalities discussed above, the receiver performs channel estimation and equalization to compensate the Inter-Symbol Interferences (ISI) (see [25]). ISI means a form of distortion of a signal that causes the previously transmitted symbols to have an effect on the currently received symbol. This is usually an unwanted phenomenon as the previous symbols have a similar effect as noise, and thus making the communication less reliable. It is usually caused when signals suffer from fading, reach the receiving antenna in various paths obstructed by atmospheric ducting, ionospheric reflection and refraction, and reflection from objects like buildings. If ISI occurs then the receiver may demodulate the signal incorrectly and thus result in error.

6.3 Experiment

As an experiment, we have described a PIM for the IEEE 802.11 WLAN receiver in Simulink. The experiment has been conducted to evaluate whether it is possible to describe an SDR application in the VRE language and to use Simulink to represent a PIM, as well as to evaluate whether it is easier to describe an application in the VRE language than in platform-specific languages such as POSIX-C.

The IEEE 802.11 WLAN has been chosen for this experiment, as its signal processing algorithms are computationally complex and it has a strict real-time requirement (i.e., it has a high data rate), and thus developers experience similar level of programming complexity to describe an application for the IEEE 802.11 WLAN and other SDR applications such as GSM and Bluetooth.

In this experiment, in addition to describe the PIM, we have also simulated the PIM using Simulink's simulation environment, as well as compiled the PIM employing the VRE compiler. This section describes the experiment, except remarks about the compilation, which will be discussed in Section 8.6. In particular, it first describes the PIM representation, then presents the simulation result, and finally discusses the experiences that have been gathered from the experiment and explains why it is easier to describe an application in the VRE language than in platform-specific languages.

6.3.1 WLAN Receiver Description in Simulink

To describe the PIM for the IEEE 802.11b WLAN receiver in Simulink, we have first recognized signal processing algorithms (such as filter and FFT) required to describe the application, and next described S-Functions for these algorithms and thus developed primitives as S-Function blocks. Then, we have represented the PIM according to the guidelines discussed in the last chapter.

Fig. 6.3 sketches the PIM representation of the IEEE 802.11b WLAN receiver in Simulink. For brevity, it only shows the receiver's signal processing chain, i.e., blocks exchanging information, but not other details such as memories and flags. The receiver description consists of ten blocks. The blocks "vreSyncIn1" and "stop" are added to the receiver description for simulation purpose, i.e., the former generates an impulse and thus triggers (activates) the block "subsystem1", and the later is an instance of a Simulink library block "Stop" that terminates the simulation upon receiving a signal. The other blocks perform the signal processing operations of the receiver. In particular, they do the following:

1. The block "subsystem1" acquires the samples and looks for the SYNC bits. Upon receiving the SYNC bits, it triggers the block "subsystem2".

2. The block "subsystem2" does channel estimation and equalization, and then triggers the block "subsystem3".

3. The block "subsystem3" detects the SFD, and then triggers the block "subsystem4".

4. The block "subsystem4" processes the header, i.e., it determines the type of modulation to be used by the receiver (1Mbps DBPSK or 2Mbps DQPSK) and packet length. Then, it triggers the block "caseSS".

5. The block "caseSS" is a VRE-defined "Select" block (see 4.4.3), which reads a memory and then triggers either the block "subsystem5" or the block "subsystem6" based on the memory value that is set by the block "subsystem4", i.e.,

"subsystem4" determines the type of modulation to be used by the receiver and thus sets the memory value to 1 if the modulation type is 1Mbps DBPSK or to 2 if the modulation type if 2Mbps DQPSK.

6. The block "subsystem5" processes the PPDU payload using 1Mbps DBPSK modulation scheme. Then, it triggers the block "endCaseSS".

7. The block "subsystem6" processes the PPDU payload using 2Mbps DQPSK modulation scheme. Then, it triggers the block "endCaseSS".

8. The block "endCaseSS" is a VRE-defined "EndSelect" block (see 4.4.3), which marks the end of the branches and triggers the "Stop" block.

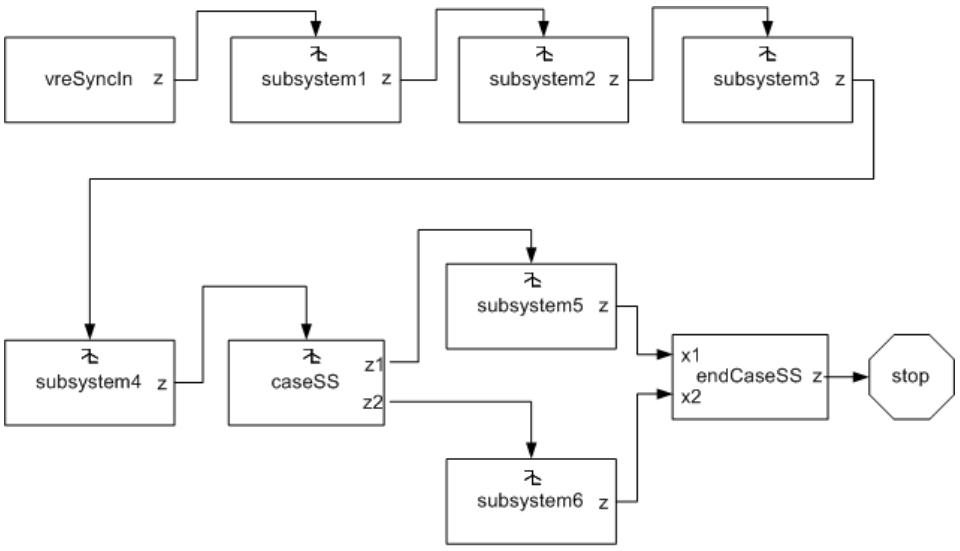


Figure 6.3: The IEEE 802.11b WLAN receiver in Simulink.

Note that Fig. 6.3 shows blocks exchange triggers and thus describes possible memory-coupled dependencies, e.g., "subsystem4" writes a value to a memory (not shown in the figure) after determining the type of modulation to be used by the receiver, where "caseSS" evaluates the value from the memory for activating one of the associated branches. Here, the blocks "caseSS" and "endCaseSS" are added to describe the Case-DynamicBranch (see 4.4.3) representation. The other blocks perform receiver operations, i.e., they process different sections of PPDU, e.g., "subsystem1" processes the first 128 bits and "subsystem3" processes the next 16 bits (the PPDU packet format is already presented in Fig. 6.2). These blocks are modular block, and thus internally contain other blocks and describe the restrictions of scheduling of their internal blocks. For example, Fig. 6.4 presents the internal construct of the block "subsystem1". It internally contains a "While Iterator Subsystem" (coarsSyncSS) block and thus describes a "do-while" loop. The block "coarsSyncSS" consists of several primitive blocks such as

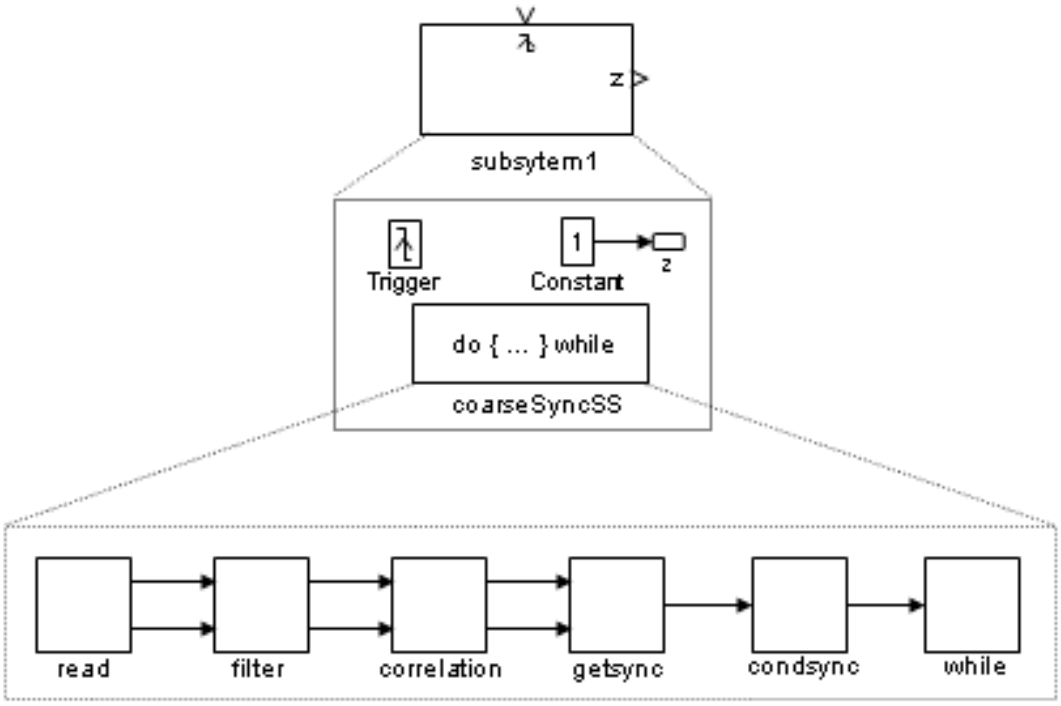


Figure 6.4: Internal construct of "subsystem1" (coarse synchronization module) block of the WLAN receiver.

"read" and "filter", and describes message-coupled dependencies between these primitive blocks, e.g., the block "read" produces data that the block "filter" consumes.

Thus, the WLAN receiver is represented in the VRE language, i.e. it describes message-coupled, memory-coupled, and control-coupled dependencies between the blocks.

6.3.2 Simulation

Before simulating the PIM discussed above, we added two more blocks ("vreSyncIn" and "readSamplesSS") to the PIM for loading input samples for the receiver, i.e., "VreSyncIn" (that generates an impulse) triggers "readSampleSS" that then loads samples from a file to the model. The receiver processes these samples during simulation and thus generates PPDU. It is important to mention here that the blocks "vreSyncIn" and "vreSyncIn1" have been parameterized to generate an impulse in the first and second simulation time steps, respectively. Thus, to simulate the PIM, we have to set the simulation start time to 0 and the simulation end time to 1. This means there are two simulation time steps: the time step 0 to execute the blocks "VreSyncIn" and "readSamplesSS", and the time step 1 to execute the receiver chain. Setting other simulation options are not important for simulating the PIM, however one is free to do that. See the Simulink documentation (available at [42]) to know about the other simulation options.

Simulink's simulation environment computes a schedule that corresponds to a serial program and thus executes a program during simulation. For instance, in case of the

PIM that does not describe a schedule, it executes first the block "vreSyncIn1", next the block "subsystem1", then the block "subsystem2", and so on.

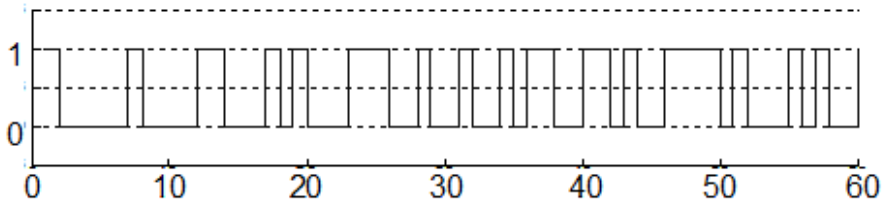


Figure 6.5: Identical transmitted and received payloads.

Fig. 6.5 depicts the transmitted signal. The result of the simulation showed that the received signal was identical to the transmitted signal. Therefore, we can consider the WLAN receiver description in Simulink as correct. Thus, it can be concluded that one can use Simulink as a graphical editor for describing a PIM.

6.3.3 Remarks

The experiences gathered from this experiment have shown that, for an application developer who has a detailed knowledge of both Simulink and the IEEE 802.11b WLAN protocol, it has taken one working week (approx.) to determine which primitives need to be developed for describing the protocol, plus four working weeks (approx.) to develop the required primitives as S-Functions, plus one working week (approx.) to describe the protocol and bug fixing. That means, it has taken altogether six weeks (approx.). Assuming that the required primitives are available to the developer as libraries, it has taken two weeks. The experiences from other environments such as Sandblaster IDE, in contrast, have shown that it has taken more than twelve working weeks for a developer to describe the protocol, assuming the similar preconditions, i.e., the developer has a detailed knowledge of Sandblaster IDE and the IEEE 802.11b WLAN protocol.

6.4 Summary

This chapter has given an overview of the IEEE 802.11b WLAN and described the experiment that has been conducted to represent the IEEE 802.11b WLAN receiver as a PIM in Simulink.

Chapter 7

Hardware Description Files

As already stated in Chapter 1, in VRE, we describe hardware-specific information that is needed by the VRE compiler in hardware description files. There are two such files, called hardware model description file and hardware implementation library file, respectively. This section presents contents and formats of these files, as well as discusses why these contents are needed by the VRE compiler.

7.1 Hardware Model Description File

The hardware model description file contains information about processing modules of the target hardware. In particular, it contains two pieces of information about each processing module:

1. The maximum number of threads that are supported.
2. The types of vector operations that are supported.

The former enables the compiler to determine how many tasks can be run in parallel. The later enables the compiler to determine strip sizes for vector data and hence to realize vector operations, as discussed in Section 3.1. As an example, a hardware model description file is shown in Fig. 7.1. Note that it is described in XML format. It shows that the hardware has four processing modules that are distinguished by an ID. These processing modules correspond to the DSP cores of the hardware and therefore are of class "DSP". It is important to mention here that an SDR hardware may consist of processing modules of different classes. MuSIC, for instance, consists of four processing modules of class SIMD, one processing module of class filter, i.e., a hardware accelerator dedicated for the filter operation, and another of class viterbi, i.e., a hardware accelerator dedicated for the viterbi operation (see Section 2.3). To determine whether or not a processing module is capable of executing a primitive, the VRE compiler needs to know what the class of the processing module is, as will be discussed in the next section.

Note that the file contains two pieces of information about each processing module, "Thread" and "VectorOp", as discussed earlier. Further note that the "VectorOp" is described with two attributes: (1) "vector_length" is equal to the size of the vector

register and (2) "element_length" is equal to the length of vector elements, both in terms of number of bits. For instance, "vector_length = 64" and "element_length = 16" mean that the corresponding processing module can perform operations on two vectors, where each vector consists of four (i.e., "vector_length/element_length" = 64/16 = 4) 16-bit (i.e., "element_length") elements.

```
<?xml version="1.0" encoding="utf-8" ?>
 <Hardware xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  Name="SB3010" xsi:noNamespaceSchemaLocation="hw.xsd">
   <ProcessingModule ID = "1" class = "DSP" >
    <Threads Value ="8" />
    <VectorOp vector_length = "64" element_length="16"/>
   </ ProcessingModule >
   < ProcessingModule  ID = "2" class="DSP" >
     < Threads Value ="8" />
     < VectorOp vector_length = "64" element_length="16"/>
   </ ProcessingModule >
   <ProcessingModule ID = "3" class = "DSP" >
     < Threads Value ="8" />
     < VectorOp vector_length = "64" element_length="16"/>
   </ ProcessingModule >
   < ProcessingModule ID = "4" class = "DSP" >
     < Threads Value ="8" />
     < VectorOp vector_length = "64" element_length="16"/>
    </ ProcessingModule >
  </Hardware>
```

Figure 7.1: Hardware model description file.

7.2 Hardware Implementation Library File

The hardware implementation library file contains hardware-specific information about primitives. In particular, it contains two pieces of information about each primitive:

1. The type of processing module that is capable of executing it.

2. The time the processing module takes to execute it.

These two pieces of information enables the VRE compiler to compute mapping and scheduling. Assume, for instance, that we describe a PIM that includes a FIR block and a CRC block, where the FIR block produces data for the CRC block and thus

has to be executed before the CRC block. Further assume that we want to perform an implementation of the PIM on a hardware platform that consists of two processing modules: a processing module named P1 that is capable of executing the FIR block in 4ms and the CRC block in 8ms, and another processing module P2 that is capable of executing the FIR in 6ms and the CRC block in 4ms, where these processing modules can exchange information via shared memory without much delay. From this information, the compiler can compute an efficient mapping and scheduling, i.e., assign FIR to P1 and CRC to P2 for execution.

An implementation library file contains these two pieces information also about synchronization primitives such as "vreCondWait" and "vreCondSignal" (already introduced in Section 4.5). The later piece of information about a synchronization primitive corresponds to the minimum execution time of the primitive, as execution times of synchronization primitives may vary depending on synchronization scenarios. For instance, in case of the synchronization primitive "vreCondWait", it corresponds to the time that requires to execute the primitive when the primitive finishes its operation without waiting for a wake up signal, as either of the primitive condition is true. Additionally, an implementation library file contains information about different possible implementations of primitives, e.g., two different implementations of one primitive, each for a processing module of a different class.

```
<?xml version="1.0" encoding="utf-8" ?>
<ImplLibrary xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 Name="SB3010" xsi:noNamespaceSchemaLocation = "ImplLib.xsd">
 <Primitive Class="filter">
   <Implementation ID = "1" >
     <Parameter Name = "VRE_EXEC_TIME", Value = "120" />
     <Parameter Name = "VRE_PM_CLASS", Value = "1" />
   </Implementation>
   <Implementation VRE_IMP_ID = "2" >
    <Parameter Name = "VRE_EXEC_TIME", Value = "80" />
    <Parameter Name = "VRE_PM_CLASS", Value = "2" />
   </Implementation>
 </Primitive>
 <Primitive Class="adder">
   <Implementation VRE_IMP_ID = "1">
     <Parameter Name = "VRE_EXEC_TIME", Value = "$pSize*16" />
     <Parameter Name = "VRE_PM_CLASS", Value = "1" />
  </Implementation>
 </Primitive>
</ImplLibrary>
```

Figure 7.2: Hardware implementation library file.

For clarity, an example hardware implementation library file is presented in Fig. 7.2. Note that it is also specified in XML. It contains information about two different

implementations of a "filter" primitive (where the implementations are distinguished by a unique ID), as well as one implementation of an "adder" primitive. It shows that both the "filter" and the "adder" primitives can be mapped to the processing module whose ID is "1" ("VRE_PM_CLASS"), and the "adder" primitive can also be mapped to the processing module whose ID is "2". Moreover, it contains information about the execution times ("VRE_EXEC_TIME") of the primitives (in microsecond) on the processing modules.

Note that the parameter "VRE_EXEC_TIME" of the "adder" primitive is described by an expression ($pSize*16), which means a DSP type processing module needs 16 times "pSize" micro seconds to execute the primitive, where "pSize" is a parameter of the "adder" primitive. For clarity, see Fig. 7.3. Here, the parameter "pSize" defines the width of the input and output ports of the primitive and thus eventually specifies how many addition operations to be performed by the primitive. For instance, if "pSize" is 16 then the "adder" primitive performs 16 addition operations, i.e., it consumes two vectors ("a" and "b") of 16 elements each and produces another vector ("c") of 16 elements. Thus, if the value of "pSize" increases or decreases then the execution time of the adder primitive also increases or decreases.

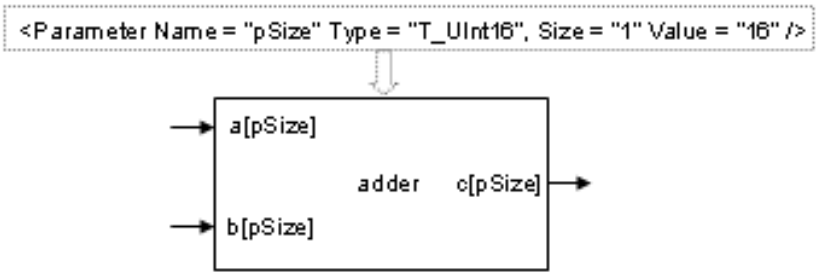


Figure 7.3: Representation of "adder" Primitive.

7.3 Summary

This section has described the contents and formats of the hardware description files files, as well as discussed why these contents are needed by the VRE compiler. The next chapter presents the VRE compiler kernel.

Chapter 8

The Compiler Kernel

This chapter first presents an overview of the VRE compiler kernel. Then, it describes in detail how the compiler kernel transforms a PIM into a PSM, i.e., how it identifies tasks, evaluates dependencies between tasks, eliminates some dependencies, and computes a mapping and scheduling as well as inserts required synchronizations. Finally, it makes some remarks based on some experiments that have been conducted to evaluate the compiler kernel.

8.1 Definition of Terms

This section describes some terms that have specific meaning in VRE and have been used in this chapter.

Task, subtask and supertask:

As already described in Section 4.2, a task is a building block of a PSM and corresponds to either a primitive or a set of primitives. A task may internally contain other tasks that are called subtasks and a task that internally contains subtasks is called the supertask of these tasks. During compilation, the VRE compiler kernel transforms blocks into tasks and thus assigns scheduling and mapping. See Fig. 8.1 for an example. Here, the "Adder" block that performs 512 addition operations is mapped to two different processing modules (PM1 and PM2): one task that performs the first 256 addition operations is mapped to PM1, and another task that performs the remaining 256 addition operations is mapped PM2. In this chapter, the term task is used to refer to a primitive or a set of primitives that are (or are to be) assigned to a processing module for execution.

Dependent, independent and partially dependent tasks:

A set of tasks that have data dependencies and therefore cannot be run in parallel is said to be dependent or non-parallel tasks, and a set of tasks that do not have data dependencies and therefore can be run in parallel is said to be independent or parallel tasks. Tasks that internally contain subtasks can also be partially dependent (partially

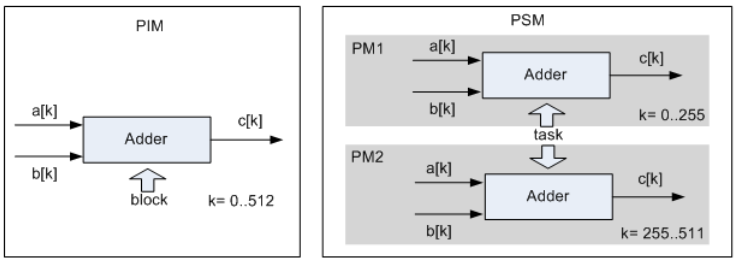


Figure 8.1: Creation of tasks from blocks.

parallel), i.e., some subtask of one task can be run in parallel with some subtask of another but not all subtasks of one task can be run in parallel with all subtasks of another. See Fig. 8.2 for an example. Here, a loop of three iterations is broken down into three different tasks, each corresponding to one iteration. These tasks are not fully parallel as data dependencies exist among them: the CRC block is writing data to a global memory (CRC_REG), and therefore the instances of this block for different iteration steps must be executed sequentially. Albeit, as shown in Fig. 8.2, it is possible to run these tasks in parallel by means of synchronizing the executions of the instances of the CRC block and hence these tasks are said to be partially dependent.

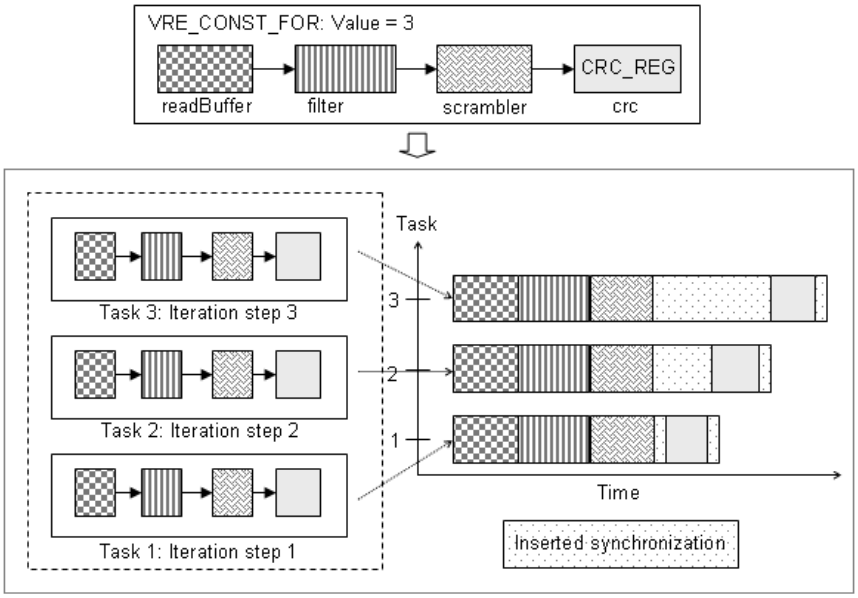


Figure 8.2: Partially dependent or partially parallel tasks.

8.2 Overview of the Compiler Kernel

The heart of the VRE compiler is the compiler kernel that takes the PIM and hardware description files as inputs, and generates a corresponding PSM as output. The compiler

kernel solves hardware-specific issues, i.e., it computes mapping and scheduling, as well as inserts synchronizations. Within the scope of this thesis, a prototype of the compiler kernel has been developed for the SDR hardware platform SB3010.

Fig. 8.3 presents an overview of the compiler kernel. It shows that the compilation process of the compiler kernel includes three phases, called preprocessing phase, program analysis phase, and transformation phase, respectively. In the preprocessing phase, the compiler kernel prepares a PIM to be transformed into a PSM. In the analysis phase, it performs program analysis, i.e., it evaluates dependencies between blocks. In the transformation phase, it computes mapping and scheduling and inserts synchronizations, and hence transforms a PIM into a PSM. The next sections describe these phases in more detail.

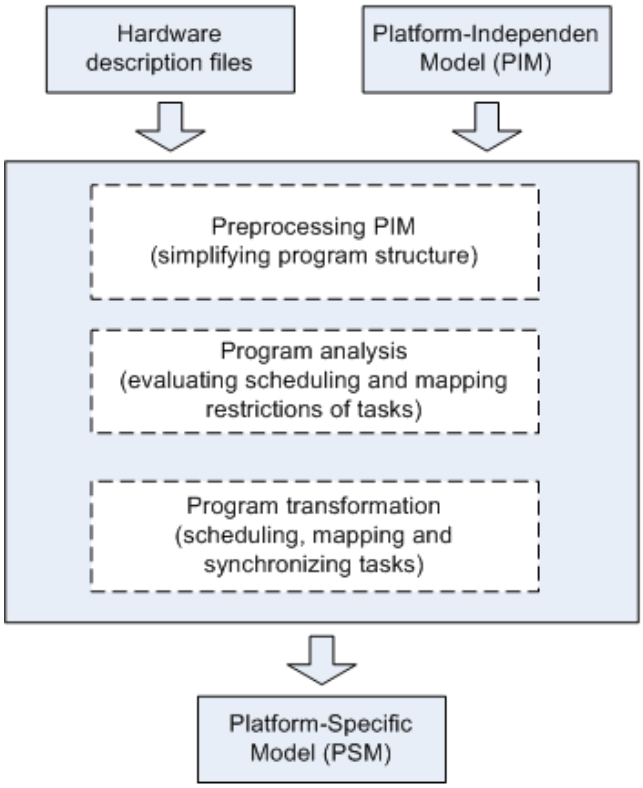


Figure 8.3: The compiler kernel.

8.3 Preprocessing Phase

In the preprocessing phase, the compiler kernel performs error checking, i.e., it stops compilation indicating an error if, for instance, the value of a parameter does not correspond to the data type of the parameter. Additionally, it resolves parameters values, memory links, and flag links, as well as eliminates irrelevant hierarchies and branches, which are discussed below.

Resolving Parameter Values:

It is often the case that the value of a parameter is not directly specified. Consider, for instance, the example depicted in Fig. 8.4, which shows a modular block that internally contains two primitive blocks. Here, the values of the parameters "P1" and "P2" are indirectly specified, i.e., "P1" refers to "P2" ("P1" = "$P2") and "P2" refers to "P3" (P1 = "$P3"). In the preprocessing phase, the compiler kernel resolves such indirect references, i.e., it substitutes the values of "P1" and "P2" by the value of "P3", which means both "P1" and "P2" contains "2" (not "$P3") after the preprocessing step.

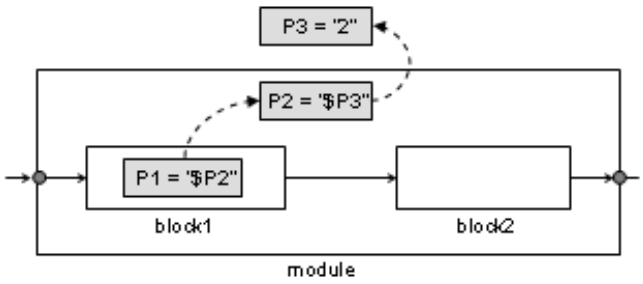


Figure 8.4: Indirect references of parameters.

Resolving Memory and Flag Links:

Like parameters, memory links may point to memories indirectly. Consider, for instance, the example shown in Fig. 8.5. Here, the memory link "MB" of the block "B" refers to "MA" that is the memory link of block "A" and refers to memory "M". Thus, "MB" actually refers to "M". In the preprocessing step, the compiler kernel resolves such indirect references, i.e., "MB" will refer to "M" instead of "MA" after the preprocessing step. Similarly, it resolves flag links.

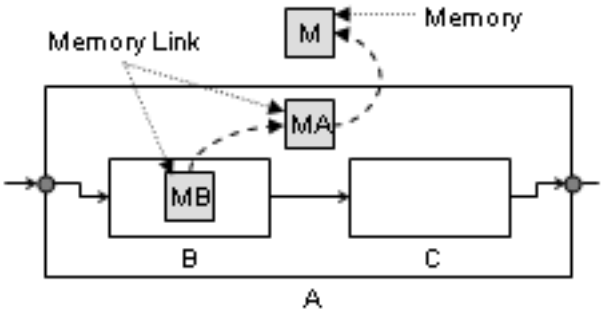


Figure 8.5: Hierarchical references of memory links.

Flattening PIM:

After resolving parameter values as well as memory and flag links, the compiler kernel flattens the representation of a PIM, i.e., it eliminates the composed modules and re-

places them with their internal contents. However, it does not eliminate the modules that represent loops in order to keep them separate from the rest of the algorithm. Thus, after the preprocessing phase, a PIM comprises no other modular blocks than loops.

Eliminating Irrelevant Branches:

For "CaseStatic" type branches, the compiler kernel recognizes the actual branch to be executed, and then eliminates the irrelevant branches from the PIM, because there is no real branch selection at runtime, as already discussed in Section 4.4.3.

8.4 Program Analysis Phase:

In this step, the compiler kernel gathers information about dependencies. It checks whether a block produces data that another consumes and hence identifies message-coupled dependencies, and it checks whether a block writes data to a memory that another reads and hence identifies memory-coupled dependencies. Control-coupled dependencies, on the other hand, are identified from representations of loops and branches. The compiler kernel preserves information about dependencies in the format of following tables:

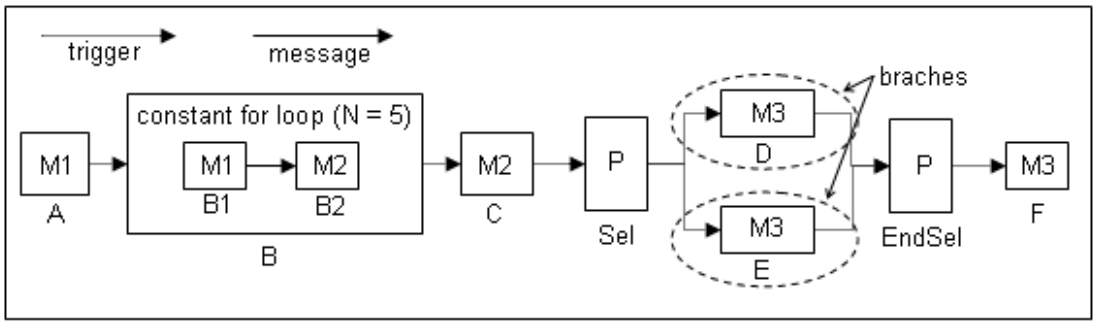
- One *system table* per PIM holds information about primitive blocks.
- One *loop table* per loop holds information about the loop.
- One *branch table* per set of branches whose executions depend on the same branch condition holds information about the branches.

Fig. 8.6 presents a PIM and a corresponding system table. This table consists of five columns and nine rows, where the first column entry of each row contains the name of a primitive block and the other column entries of that row contain information about the primitive block:

Column 2: The names of the other primitives with which the primitive has message-coupled dependency, e.g, the column 2 of row 3 contains "B1" which means the primitive "B1" produces a data that the primitive block "B2" consumes thus "B2" has a message-coupled dependency with "B1".

Column 3: The names of the other primitives with which the primitive has memory-coupled dependency, e.g., the column 3 of row 2 contains "A" which means the primitive block "B1" has a memory-coupled dependency with the primitive block "A".

Column 4: If the primitive block is part of a loop then this entry contains the name of the loop table that contains information about the loop, e.g., the column 4 of row 2 contains "B" which means the primitive block "B1" is part of a loop and the information about this loop is available in the loop table named "B".



(a) PIM

	Primitive	Message-coupled dependency	Memory-coupled dependency	Name of loop table	Name of branch table
row 1	A				
row 2	B1		A	B	
row 3	B2	B1		B	
row 4	C		B2		
row 5	Sel				Sel
row 6	D				Sel
row 7	E				Sel
row 8	EndSel				Sel
row 9	F		D. E		
	column 1	column 2	column 3	column 4	column 5

(a) System table

Figure 8.6: An example PIM and the corresponding system table.

Column 5: If the primitive block belongs to a conditional branch then this entry contains the name of the branch table that contains the information about the conditional branch, e.g., the column 5 of row 6 contains "Sel" which means the primitive block "D" belongs to a conditional branch and the branch table named "Sel" contains relevant information about the conditional branch.

As shown in Fig. 8.7(a), the loop table consists of one row and two columns, where the first column contains information about the type of the loop such as "constant for loop" and the second column contains the names of the blocks that belong to the loop. The format of a branch table is simple, see Fig. 8.7(b) for an example. The branch consists of one row and several columns. The first column contains information about the type branch representation (CaseStatic), and other columns contains information about branches: the column 2 contains the names of the blocks (D) that are part of branch 1, and the column 3 contains the names of the blocks (E) that are part of branch 2, and so on. It is important to mention here that each loop and branch table has a unique name.

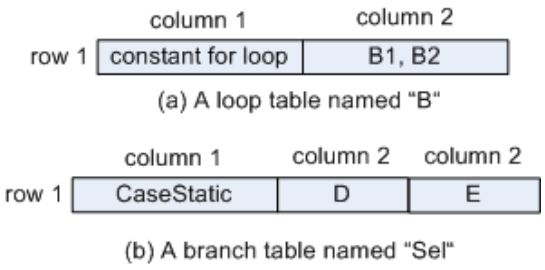


Figure 8.7: An example loop table.

8.5 Transformation Phase

In the transformation phase, the compiler kernel first identifies tasks, and then computes mapping and scheduling as well as inserts synchronizations, as discussed below.

8.5.1 Identification of Tasks

The compiler kernel creates tasks from primitive and modular blocks. To reduce the overall execution time of a program, it creates as many tasks as possible so that a maximum number of tasks can be run in parallel at a given time. It creates one task per primitive for primitives that do not consume (or produce) as input (or output) a vector of values, or part of a loop (as will be discussed). Otherwise, it may create several tasks. In this case, it first recognizes the part of the signal processing chain that comprises the primitive block and is described in terms of message-coupled dependencies. Then, it creates tasks on the basis of which particular mapping and scheduling provides a minimum execution time of that part of the signal processing chain. For clarity, an example signal processing chain is depicted in Fig. 8.8, which consists of two primitive blocks, "Mult" and "Add". These blocks are connected through message exchange. The input and output ports of each block correspond to a data vector of 512 values, and thus each block has to perform 512 operations: (b[k] = a[k] * a[k] and c[k] = b[k] + b[k]), for k = 0..512. The figure also provides the implementation-specific information about the primitives, which shows both the primitives can perform every 128 operations in 4ns. Using this information, the compiler kernel evaluates different possible implementations of the signal processing chain and thus determines which particular implementation results in a minimum execution time. As shown in the figure, the minimum execution time for this signal processing chain is 8ns, which is obtained by creating four tasks out of each primitive block and then mapping them to four different processing modules to run in parallel. It is important to mention here that the compiler kernel defines an unique ID for each task to distinguish the task from others.

The compiler kernel creates one task per loop for loops whose iteration steps must not be run in parallel, as one iteration step produces data that the next iteration step uses. It creates several tasks from a loop with independent iterations, i.e, one iteration step can be run in parallel with the others. In particular, from a loop with m independent iteration steps, it creates $m * k$ tasks, where k is the number of blocks that are to be executed per iteration steps. However, such as a loop is not often seen in SDR

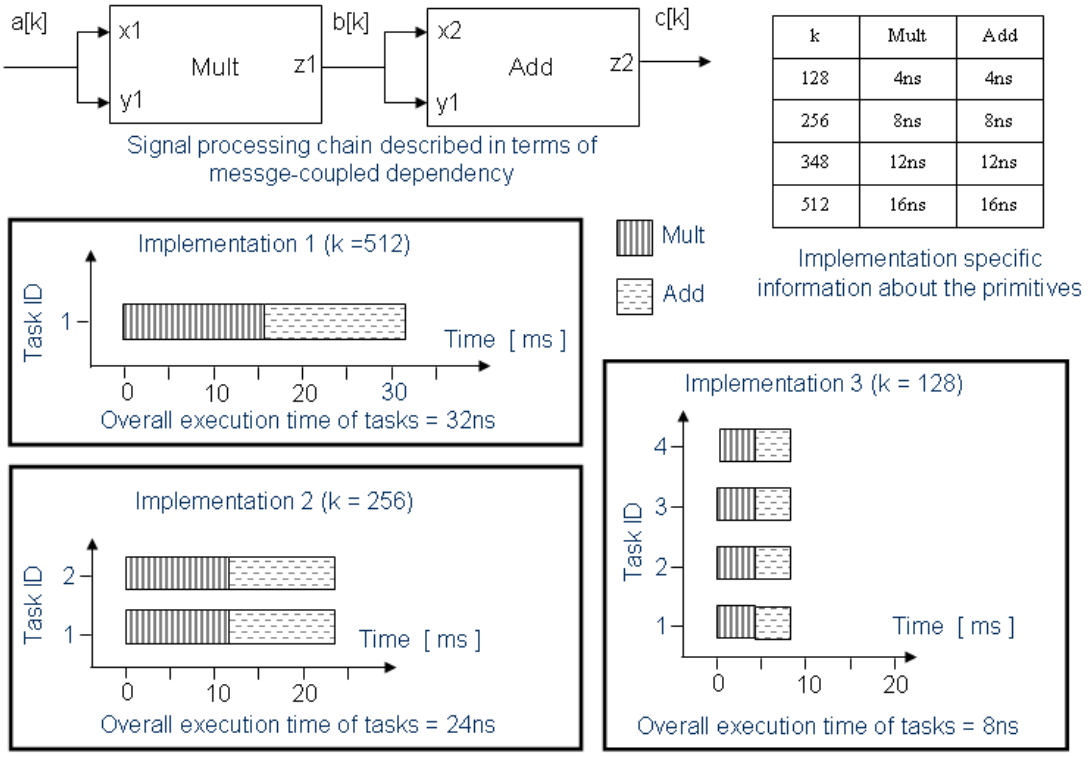


Figure 8.8: Two primitive blocks that are interconnected in terms of message-coupled dependencies and exchanging vectors or values.

applications. Most loops in SDR applications are partially parallel, i.e., some blocks of one iteration step can be run in parallel with some blocks of another, but not all blocks of one iteration step can be run in parallel with all blocks of another iteration step.

To illustrate how the compile kernel creates tasks from a partially parallel loop, let's first start with an example and then discuss the actual techniques. Fig. 8.9 depicts a partially parallel loop that is a "constant for loop" with 64 iteration steps, where each iteration includes the following primitive operations in a sequential order.

Here, the blocks "Read-Buffer", "Filter-Despreader" and "Demodulator" do not change any memory value and thus do not create any dependency between different iteration steps. Thus, their instances of one iteration step can run in parallel with their instances of another. The block "Descrambler" and "CRC" write data to the memories "Scrmbl_ Reg" and "CRC_Reg" respectively. Therefore, their instances of one iteration step must not run in parallel with their instances of another.

Albeit the dependencies exist among the iteration steps, it is possible to run one iteration step of the loop in parallel with another. Fig. 8.9 shows a possible multi-threaded mapping and scheduling. Here, at a time, four different iteration steps of the loop are run in parallel. Note that the desired execution order of the instances of both "Descrambler" and "CRC" blocks in different iteration steps are ensured through the

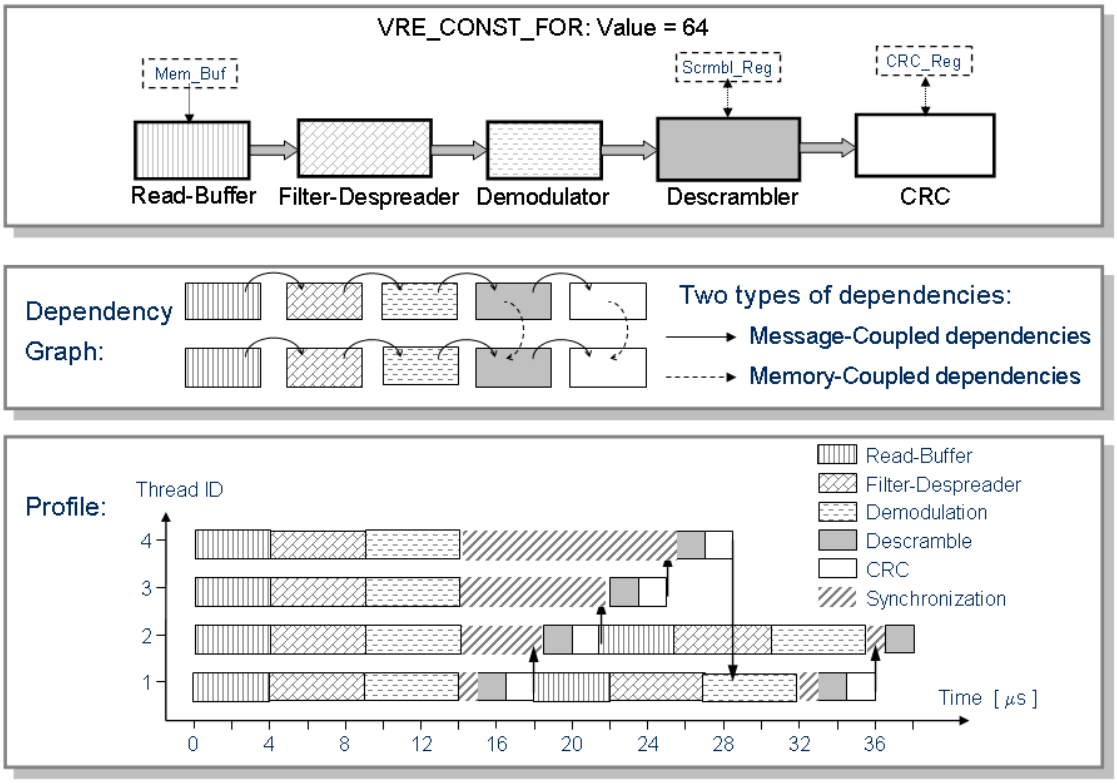


Figure 8.9: Creating tasks from a partially parallel loop.

insertion of the required synchronizations.

The compiler kernel creates several tasks out of a partially parallel loop depending on how many iteration steps of the loop can potentially be run in parallel. For instance, in the example shown in Fig. 8.9, four different iteration steps can potentially be run in parallel. This is because, the 1^{st} thread for instance is capable of executing the independent part ("Read-Buffer", "Filter-Despreader", "Demodulator") of the 5^{th} iteration step while the 4^{th} thread is still executing the dependent part ("Descrambler" and "CRC") of the 4^{th} iteration step. Thus, it can execute the dependent part of the 5^{th} iteration step without any delay just after the 4^{th} thread finishes the dependent part of the 4^{th} iteration step. In this case, running the 5^{th} iteration step in a 5^{th} thread will just result in wastage of an additional resource.

The compiler kernel creates n number of tasks from a partial parallel loops if the n^{th} iteration step is run in parallel with the 1^{st} to $(n-1)^{th}$ iteration steps and then is finished earlier than if it would have been run sequentially with the 1^{st} iteration step. Thus, the compiler kernel computes how many tasks can be created from a partially parallel loop solving the following equation:

$$i + (n * d) + s <= 2(i + d)$$

Here, i = execution time of the independent part of each iteration step,
s = synchronization overhead (will be discussed),

d = execution time of the dependent part of each iteration step,
and n = total number of tasks that can be created from a partially parallel loop.

Thus, we get

$$n <= 2 + (i - s)/d$$

From hardware-specific implementation library file, the compiler kernel gathers information about the execution times of different primitive blocks and hence computes the value of i and d. It determines which synchronization tasks need to be inserted and thus computes the synchronization overhead s that corresponds to the minimum execution time of these synchronization tasks. As discussed in Section 4.5, in each partially parallel task (like shown in Fig. 8.9), it inserts a synchronization task that corresponds to the synchronization primitive "vreCondWait" and another synchronization task that corresponds to the synchronization primitive "vreCondSignal" before and after the dependent part, respectively. Then, the value of the synchronization overhead is that of the value of the total minimum execution time of these two synchronization primitives. As already discussed in Section 7.2, the hardware-specific implementation library file also contains information about the minimum execution times of synchronization primitives.

For clarity, let's compute the value of n for the partial loop shown in Fig. 8.9 assuming that the approximate execution times of the primitive blocks are as follow: "ReadBuffer" = 4μs, "Filter-Despreader" = 5μs, "Demodulator" = 5μs, "Descrambler" = 2μs, and "CRC" = 2μs. Thus, we can compute the value of i and d, i.e., i = 14μs and d = 4μs. Further assume that the minimum execution times of the synchronization primitives "vreCondWait" and "vreCondWait" are 1μs and 3μs, respectively, and thus s = 1μs + 3μs = 4μs. Then, we get

$$n <= 2 + (i - s)/d = 2 * (14 - 4)/4 <= 4.5$$

The value of $n <= 4.5$ means we can create 4 partially parallel tasks from the loop shown in Fig. 8.9.

The compiler kernel creates several tasks from a loop in a way that any task T_i will execute the i^{th} iteration step and every n^{th} successive iteration steps starting from i^{th} iteration step. For example, task T_1 will execute iteration steps 1, (1+n), (1+2n) and so on.

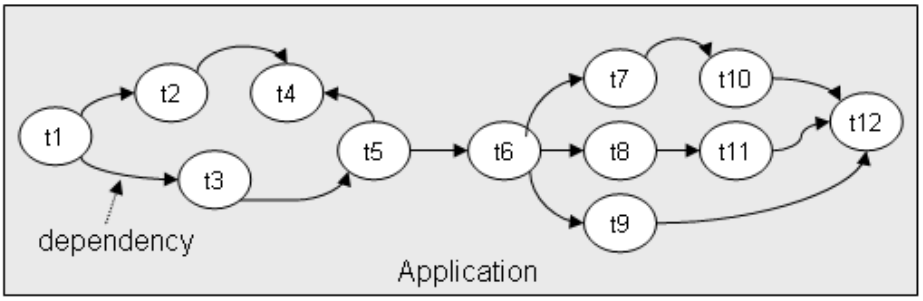
It is important to mention here that if the compiler kernel creates a task that corresponds to a loop (or one to several iteration steps of a loop) then the task internally contains subtasks (body of the loop). In VRE, like tasks, subtasks are also distinguished with unique ID.

8.5.2 Mapping and Scheduling Tasks

As already mentioned, the current version of the compiler is developed for the SB3010 hardware platform. This hardware enables simultaneous execution of parallel tasks in terms of multithreading, i.e., it supports a maximum of 8 threads per DSP core and thus up to 8*4 = 32 threads by all four DSP cores (see Section 2.3). The SB3010 hardware platform has been developed to run multiple applications simultaneously, i.e.,

one application per DSP core. Therefore, the compiler kernel performs mapping and scheduling tasks considering that the hardware consists of one DSP core that has 8 processing modules (hardware threads), where each processing module is capable of executing any task.

As already discussed in Section 8.4, the compiler kernel evaluates dependencies between primitives during program analysis phase and stores this information in various tables such as the system table. Using this information, it determines dependencies between tasks, e.g., assuming task T1 corresponds to primitive P1 and task T2 corresponds to primitive T2, T1 depends on T2 if P1 depends on P2. The compiler kernel stores this information about dependencies between tasks in a table, let's call it *TaskTable-1*, which keeps information about dependencies that exist among the tasks on the basis of one column per task, i.e., each column contains information about an individual task. For clarity, an example TaskTable-1 is depicted in Fig. 8.10. It consists of twelve columns, where the 1^{st} column contains information about the task "t1", the 2^{nd} column contains information about the task "t2", and so on. In particular, each column that corresponds to a task states which other tasks are immediate predecessor of the task. For instance, the 4^{th} column states that "t4" is to be executed after "t2", but not explicitly states that "t1" is to be executed before "t4", as the execution of "t1" before "t2" and then "t2" before "t4" guarantees the execution of "t1" before "t4".



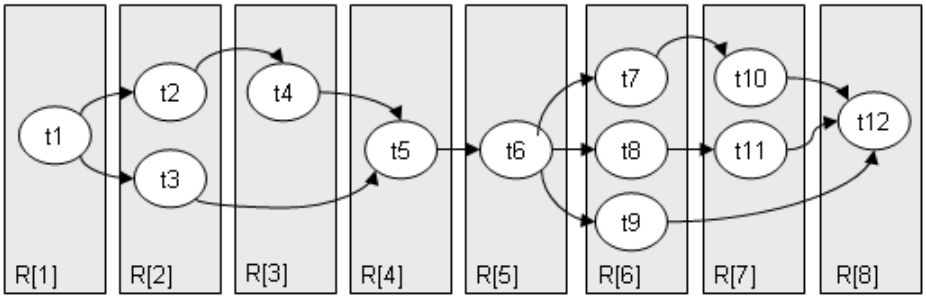
Task	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12
Dependent Tasks	-	t1	t1	t2	t3, t4	t5	t6	t6	t6	t7	t8	t11, t10, t9

TaskTable-1

Figure 8.10: Example of TaskTable-1.

Using information from TaskTable-1, the compiler kernel determines in which order tasks are to be executed. Hence, it creates another table, let's call it TaskTable-2, where any column *n* contains the names of the tasks that can be executed simultaneously and have to be executed after that of the column *(n-1)*. For clarity, an example TaskTable-2 is depicted in Fig. 8.11, which is created using the information from TaskTable-1 shown in Fig. 8.10. It describes the order in which tasks can be executed, e.g., first "t1", then

both "t2" and "t3" simultaneously, and so on.



Application (R[N] = row N of TaskTable-2)

Execution step	1	2	3	4	5	6	7	8
Parallel tasks	t1	t2, t3	t4	t5	t6	t7, t8, t9	t10, t11	t12

TaskTable-2

Figure 8.11: Example of TaskTable-2.

Using information from TaskTable-2, the compiler kernel computes a mapping and scheduling using the algorithm discussed below:

1. Let $N = 1$ and $T = 0$.

2. If $N > M$ then go to step 7. Otherwise, go to step 3. Here, M = total number of columns in TaskTable-2.

3. Assuming R_N is a set of tasks whose names are there in the N^{th} column and R_{N+1} is a set of tasks whose names are there in the $(N+1)^{th}$ column of TaskTable-2, if *(N+1)* $<= M$ then compute which particular mapping and scheduling of R_N and R_{N+1} results in a minimum value of $T_{R(N)}$, else compute which particular mapping and scheduling of R_N results in a minimum value of $T_{R(N)}$, where $T_{R(N)} = T + T_N$, T_N = overall execution time R_N.

4. Based on the particular mapping and scheduling that is determined in step 3, specify a schedule and the corresponding mapping to R_N and also insert the required synchronizations.

5. Compute new value of T, i.e., $T = T_{R(N)}$.

6. Increment the value of N, i.e., $N = N + 1$, and then go to step2.

7. End of algorithm.

The compiler kernel computes all possible mapping and scheduling of R_N and R_{N+1} (or only R_N if $(N+1) > M$), and hence determines which particular mapping and scheduling results in a minimum value of $T_{R(N)}$. It computes $T_{R(N)}$ considering synchronization overhead. For clarity, an example is depicted in Fig. 8.12, which shows both t2 and t3 can run in parallel but must be executed after t1, and two different scheduling and mapping of R_N: (1) R_N is mapped to PM1, and (2) R_N is mapped to both PM1 and PM2, i.e., t2 of R_N is mapped to PM2 and t3 of R_N to PM1. Note that the former results in a lower value of $T_{R(N)}$ than the later due to the synchronization overhead, which is required to ensure the execution of t2 after t1. Then, the compiler kernel considers a mapping and scheduling for R_N that corresponds to the former. Thus, its mapping and scheduling concept differs from other parallelizing compilers such as the OpenMP compiler [6], as it evaluates different possible mappings and schedulings and hence selects an efficient one.

The compiler kernel computes a mapping and scheduling of R_N considering the dependency relation between R_{N-1} and R_N (where $N > 1$), i.e., tasks of R_N must be executed after tasks of R_{N-1}.

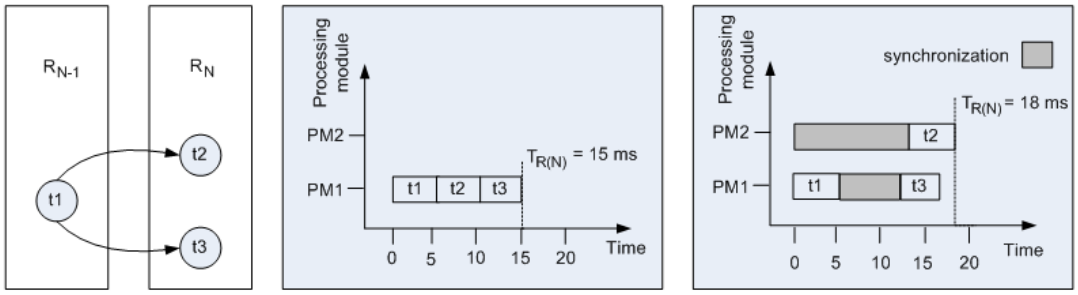


Figure 8.12: An example that shows two different scheduling and mapping.

After computing the mapping and scheduling that result in the minimum value of $T_{R(N)}$, the compiler kernel specifies the corresponding mapping and scheduling to R_N. Hence, as introduced in Section 4.5, it associates two parameters to each task of R_N: VRE_MAP_ID and VRE_SCHED_ID. The former specifies in which processing module the task is to be mapped for execution, and the later specifies at what time the task is to be executed on the corresponding processing module.

Additionally, the compiler kernel inserts required synchronizations to ensure a desired execution order of dependent tasks. It inserts synchronization tasks that correspond to the synchronization primitives "vreCondSignal" and "vreCondWait" to enable parallel execution of different iteration steps of a partial parallel loop, as shown in Fig. 8.9. In other cases, to ensure required synchronizations, it inserts synchronization tasks that correspond to the synchronization primitives "vreInCounter" and "vreWaitInCounter". Some examples are already discussed in Section 4.5.

For the application description shown in Fig. 8.11, a possible mapping and scheduling along with inserted synchronization tasks are presented in Fig. 8.13. It shows in which order tasks are to be executed in different processing modules, e.g., the processing module

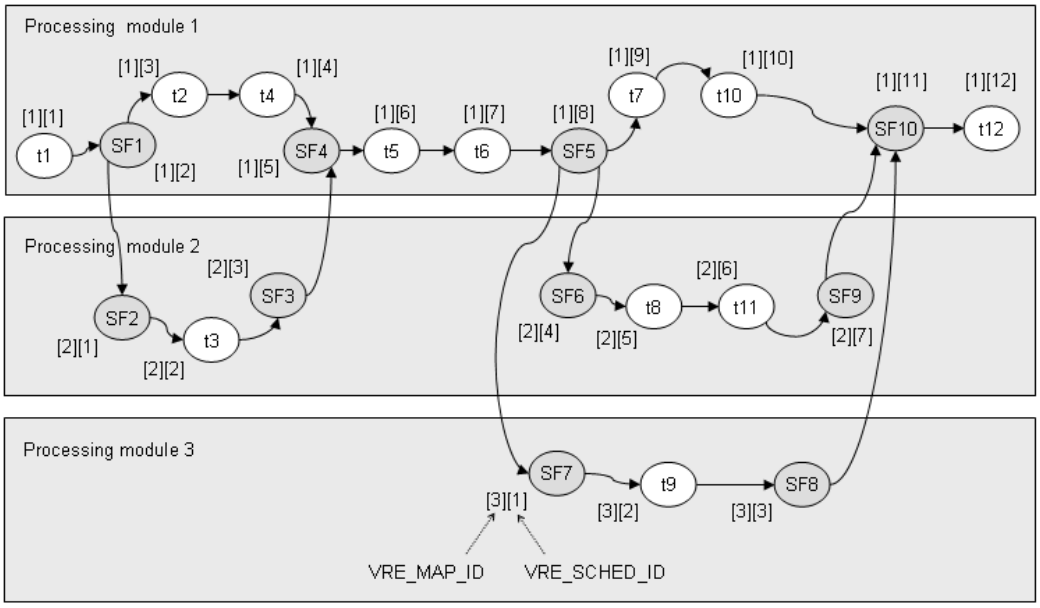


Figure 8.13: A mapping and the corresponding scheduling of tasks.

1 executes the tasks in the following order: "t1"->"SF1"->"t2"->"t4"->"SF4"->"t5"->"t6"->"SF5"->"t7"->"t10"->"SF10"->"t12", where "SF1" and "SF5" are synchronization tasks that correspond to the synchronization primitive "vreInCounter", and "SF4" and "SF10" are synchronization tasks that correspond to the synchronization primitives "vreWaitInCounter" (see Section 4.5).

It is important to mention here that the compiler kernel not only computes mapping and scheduling for tasks but also for subtasks. After computing where a task is to be mapped to and at what time is to be executed, it computes at what sequence the subtasks of the task are to be executed. For instance, a task that corresponds to a loop (or an iteration step of a loop) internally contains subtasks. While creating tasks, the compiler kernel determines and hence keeps information about which tasks internally contain which subtasks.

8.5.3 Eliminating Avoidable Dependencies

A loop expresses less parallelism when one iteration step of the loop has to wait for another to produce needed data. In some cases, it is possible to eliminate this kind of dependency. See the loop shown in Fig. 8.14 for example. Here, the data dependencies between the iteration steps are due to the presence of the variable "p" within the body of the loop. Since the value of "p" for different iteration steps can be predicted, it is possible to eliminate the variable from the loop body, and thus run the different iteration steps in parallel.

Similarly, the VRE compiler kernel eliminates avoidable dependencies that are resulted for "Counter" type accesses to the same memory by different blocks. For clarity, a signal processing chain is depicted in Fig. 8.15, which represents a loop that includes

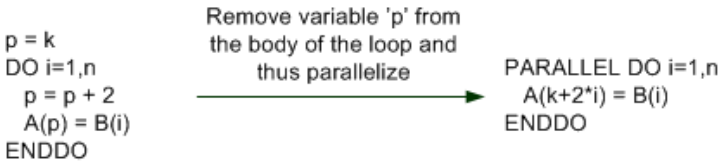


Figure 8.14: An example that shows how to eliminate avoidable dependencies between different iteration steps of a loop.

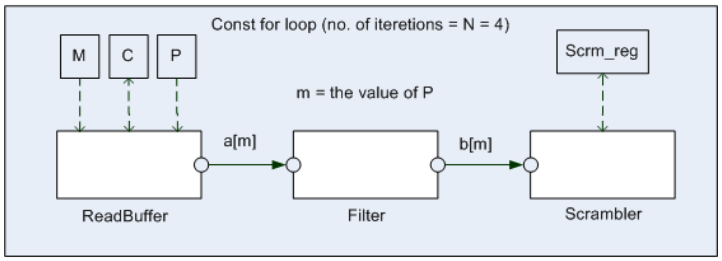


Figure 8.15: An example of avoidable dependencies in PIM.

three blocks: "ReadBuffer", "Filter", and "Scrambler". Here, the "ReadBuffer" block has a parameter ("P"), "ReadOnly" type access to the memory "M", and "Counter" type of access to the memory "C". In each iteration step, it reads m number of data elements from the memory "M" starting from the offset as specified by the value of "C", thus makes those data elements available for the "Filter" block, and then increments the value of "C" by the value of "P", i.e. $c = c + m$, $c =$ the value of "C" and $m =$ the value of "P". Thus, it creates data dependencies between different iteration steps as every time it executes it increments the value of "C". However, as the access to "C" by the "ReadBuffer" block is of type "Counter", it is possible to determine at compile time how the "ReadBuffer" block increments the value of "C" in each iteration step, i.e., $c = c + m * i$ (where $i =$ the iteration number, $0 < i <= N$). Thus, it is possible to eliminate the data dependencies.

Fig. 8.16 presents, as an example, how the compiler kernel creates parallel tasks by means of eliminating the avoidable dependencies from the PIM depicted in Fig. 8.15. Here, four tasks have been created out of that loop on the basis of one task per iteration step. Unlike PIM, where the "ReadBuffer" block accesses the global memory "C", the "ReadBuffer" block in each task accesses the memory "C1" whose scope is local within the corresponding task, e.g., "C1" in the task "1" is visible to the "ReadBuffer" block within the task "1" but not to any other blocks of any other tasks. That means, the instances of the "ReadBuffer" block in different iteration steps no more write data to any global memory upon execution. Thus, they can run in parallel. It is important to note that the value of "C1" in each task is initialized by the known value of "C" with respect to the corresponding iteration step, i.e., the value of "C1" in the 1^{st} iteration

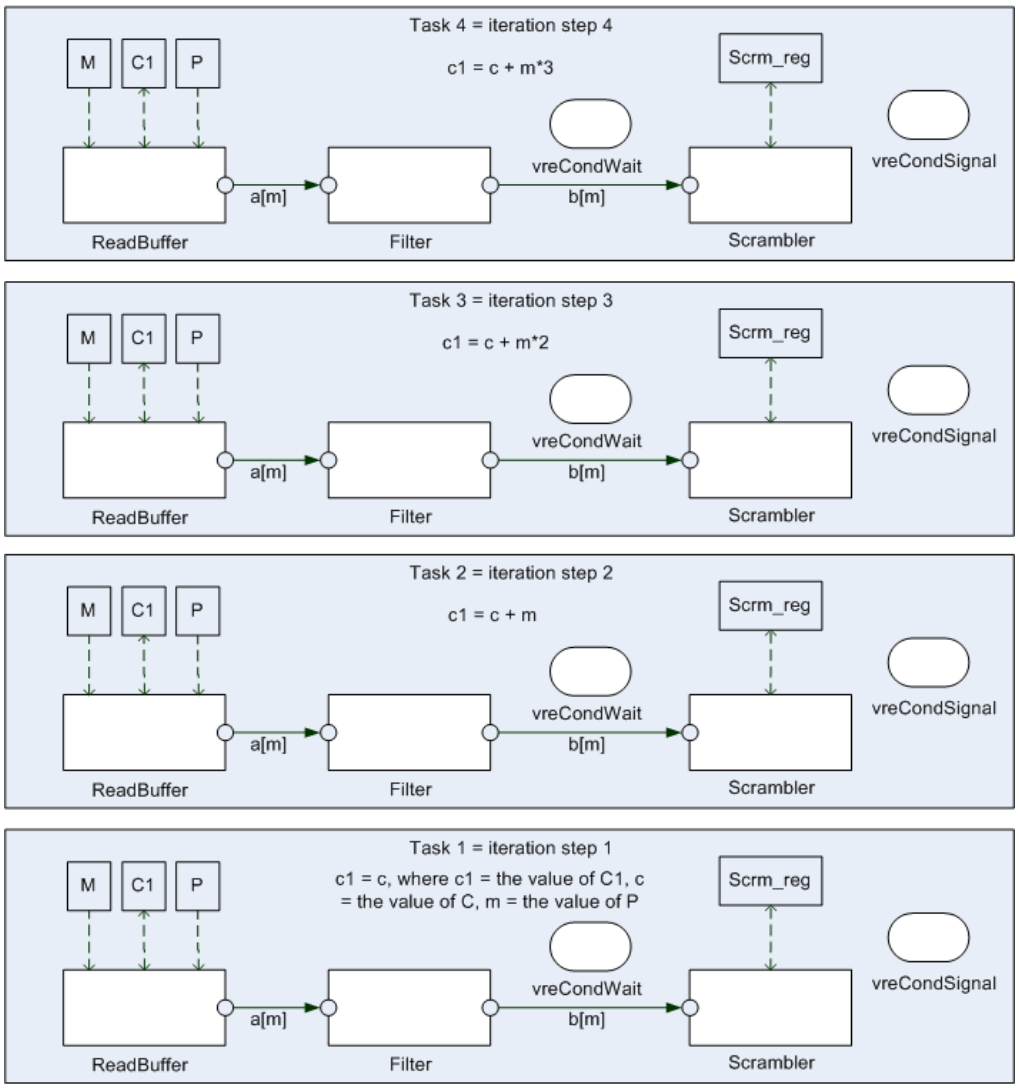


Figure 8.16: Parallelizing tasks by eliminating avoidable dependencies.

step is c, in the 2^{nd} iteration step is $c + m$, and so on. Thus, the instances of the "ReadBuffer" block in different tasks (each of which corresponds to a different iteration step) read data from the memory "M" starting from different offsets, i.e., c in case of the first iteration step (task 1), $c + m$ in case of the second iteration step (task 2), and so on.

8.5.4 Keeping Program Consistency

Creating parallel tasks as discussed in the last section may make a program inconsistent. For instance, consider the example depicted in Fig. 8.16. Here, the value of "C" is incremented by m in each iteration step, and thus by $4 * m$ (where, 4 = the total number of iteration steps of the loop) by the complete loop. On the other hand, "C"

is not incremented by the tasks that are created from the loop as shown in Fig. 8.16. Now assume that another task accesses the memory "C" after the loop. Then, that task would not get the desired value of "C". To avoid such inconsistency, the compiler kernel inserts an additional task that corresponds to the following primitive:

VreInCntParam This primitive has a parameter named "P" and a memory link named "M". Upon execution, it increments the value of a memory that is pointed by the memory link "M" by the value of "P", i.e., $M^{'} = M^{'} + P^{'}$, where $M^{'}$ is the value of a memory that is pointed by memory link "M" and $P^{'}$ is the value of the parameter "P".

The compiler kernel inserts the primitive "VreInCntParam" initializing as follows: the value of the parameter "P" $= 4 * m$ and the memory link "M" refers to "C". This makes the program consistent, because the task that corresponds to the inserted "VreInCntParam" primitive increments the content of "C" by the desired value (i.e., by $4 * m$) before another task accesses "C" after the loop.

Likewise, the compiler kernel inserts tasks that correspond to the following primitives upon creating parallel tasks from a "dynamic for loop" and a "do while loop", respectively:

VreInCntMem This primitive has a parameter named "P" and two memory links named "M1" and "M2". Upon execution, it performs the following computation: $M1^{'} = M1^{'} + (M2^{'} * P^{'})$, where $M1^{'}$ is the value of a memory that is pointed by the memory link "M1", $M2^{'}$ is the value of a memory that is pointed by the memory link "M2", and $P^{'}$ is the value of "P".

VreInCntFlag This primitive has two memory links named "M1" and "M2" and a flag link named "F". Upon execution, it performs the following computation: if $F^{'} = 1$ then $M1^{'} = M1^{'} + M2^{'}$, where $F^{'}$ is the value of a flag that is pointed by the flag link "F", $M1^{'}$ is the value of a memory that is pointed by the memory link "M1", and $M2^{'}$ is the value of a memory that is pointed the memory link "M2".

It is important to mention here that the compiler kernel inserts these primitives initializing their parameters, memory links and flag links in a way that they increment contents of desired memories by proper values.

8.6 Remarks

The current version of the VRE compiler consists of the compiler kernel that has been presented in this chapter, and a code generator that has been developed outside this thesis in [36]. Like the compiler kernel, the code generator has also been developed for the SDR hardware platform SB3010. To evaluate, we successfully employed the compiler kernel to transform several PIMs into PSMs including the IEEE 802.11b WLAN receiver presented in Section 6.3.1. Unfortunately, along with other limitations, the current version of the code generator is not capable of generating code from a PSM that includes

nested loops, and thus unable to generate platform-specific C code for the IEEE 802.11b WLAN receiver.

To evaluate the VRE compiler (both the compiler kernel and the code generator), we carried out a number of experiments. First, we described several simple PIMs that are composed of 10 to 20 primitives, and 1 to 4 modules that correspond to loops (but not nested loops) and branches. Then, we employed the compiler kernel to generate corresponding PSMs, and later the code generator to produce platform-specific C codes. The experiments showed that the compiler kernel is capable of generating a PSM from a PIM for SB3010, i.e., it is capable of mapping tasks to threads in a way so that independent tasks can be run in parallel, scheduling them according to their order of dependencies, e.g., if a task "T2" depends on another task "T1" then "T1" is to be executed before "T2", and also inserting required synchronizations. As an example, the compiler kernel is capable of computing mapping and scheduling as well as inserting required synchronization as shown in Fig. 8.16.

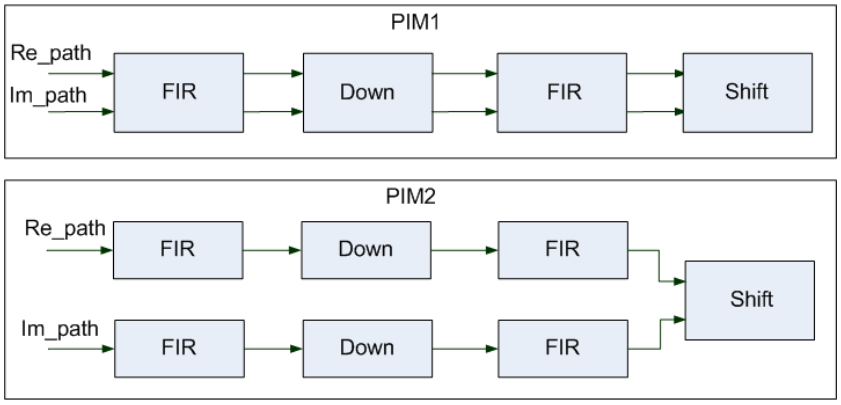


Figure 8.17: Description of potential parallelism in PIM.

The experiments also showed that the performance of the compiler kernel to a great extent depends on how a PIM is described, i.e., a PIM must express potential parallelism explicitly in order to enable the compiler kernel to generate a corresponding PSM more efficiently. For example, developers should carefully describe within a PIM which blocks have "ReadWrite" type access and which blocks have "Counter" type access. Otherwise, the compiler kernel cannot generate more parallel tasks, for instance, from a loop as discussed in the last section. Another example is shown in Fig. 8.17. Here, both the PIMs ("PIM1" and "PIM2") correspond to one application, i.e., they both include two different data paths of a signal that correspond to real ("Re_Path") and imaginary ("Im_path") parts, respectively, where in each path two filtering ("FIR") operations and one down-sampling ("Down") operation are performed. However, contrary to "PIM1", "PIM2" is represented with more blocks, i.e., it explicitly separates the operations of different data paths from one another. From this description, the compiler kernel can easily determine potential parallelism, i.e., the operations relevant to different data paths can be run in parallel. Whereas, "PIM1" does not explicitly separate the operations of the independent data paths. From this description, the compiler kernel

cannot automatically determine whether or not it is possible to run the operations of different data paths in parallel.

8.7 Summary

This chapter has described how the VRE compiler kernel transforms a PIM into a PSM, i.e., it has shown with examples how the compiler kernel identifies tasks, evaluates dependencies between tasks, eliminates some dependencies, computes mapping and scheduling, and inserts required synchronizations.

Chapter 9

Conclusions and Future Works

9.1 Conclusions

The dissertation has presented a new programming concept for SDR applications, called VRE, which has been developed with the goal to provide a complete and efficient development environment for SDR applications. The concept separates the description of an application from its implementation on a specific hardware. Thus, it allows developers to describe an application without any prior knowledge about target hardware, and reduces dependencies between software and hardware developments, i.e., application and hardware developments can take place in parallel.

The dissertation has also discussed current approaches for developing SDR applications and presented several contemporary SDR hardware platforms. It has shown that there are some points in common in these platforms, i.e., they are battery driven and thus have a low power consumption budget, and therefore use parallel architectures to achieve required performance. Then, it has described that the current programming approaches are complex, i.e, developers describe an application with hardware specific details that are relevant to parallelism such as mapping and scheduling.

Comparing VRE with other programming concepts for SDR applications, the dissertation has described how VRE differs from others and simplifies programming. For instance, unlike other approaches, VRE enables us to describe an application without hardware-specific details. Then, it has presented a detailed description of the VRE language as that has been developed within the scope of this thesis. It has described the syntax and semantics of the language, as well as explained how programs (both PIM and PSM) are represented. It has shown that VRE's concept of describing applications as a block diagram is suitable for representing SDR applications both in a platform-independent and a platform-specific ways, i.e, PIM and PSM, respectively. For instance, it has discussed why it is easier to represent an application in VRE than textual languages such as POSIX-C.

Additionally, the dissertation has shown how to incorporate Simulink into the VRE tool chain for describing PIMs, i.e., it has given guidelines to represent a PIM in Simulink. It has also discussed the advantages of incorporating Simulink to the VRE tool chain, e.g., Simulink includes a powerful simulation environment that can be used

to simulate SDR applications.

To evaluate VRE's concept of describing SDR applications, an experiment has been carried out, in which a PIM has been designed for the IEEE 802.11b WLAN receiver in Simulink using VRE-specific guidelines. The dissertation has presented the experiment as well as discussed the experiences that have been gathered from it. Thus, it has shown that it is easier to describe an application as a PIM in Simulink than to describe the application in platform-specific languages such as POSIX-C.

The dissertation has shown how VRE's concept of parallel programming differs from others as VRE defines a complete tool chain that enables developers to generate executables from described programs semi-automatically, i.e., developers may automatically produce executables from a PIM step by step, but are additionally allowed to manually improve the performance of the code after each step.

The dissertation has presented detailed description of the compiler kernel that has been developed by the author. It has shown that, in contrast to other parallelizing compilers, the compiler kernel itself does not possess any prior knowledge about the target hardware, instead it takes required hardware-specific information as input (i.e., hardware description files). Additionally, it has described how the compiler kernel generates a PSM from a PIM by solving complex programming tasks, i.e., mapping, scheduling, and synchronizations.

VRE is still under development, i.e., a complete dedicated tool chain for VRE is yet to evolve. Therefore, a reasonable amount of work still needs to be carried out in this field, which will be discussed next.

9.2 Future Works

The success of VRE to a great extent depends on the efficiency of its tool chain, e.g., the performance of a program would be low if the compiler kernel is not capable of computing an efficient mapping and scheduling. On the other hand, the present version of the VRE tool chain is still in its infancy, and therefore needs to be developed further. In particular, the VRE compiler has to be developed for various SDR hardware platforms and its algorithms for computing mapping and scheduling as well as for inserting synchronizations have to be improved.

Additionally, we have to develop a dedicated application description environment for VRE. However, one may ask why we need another environment when it is possible to use Simulink. Briefly stating, it is not possible to visualize a PSM in Simulink, and not so convenient to describe a PIM in Simulink, e.g., it is difficult to describe memory-coupled dependencies. Moreover, we can focus on developing some complementary tools for VRE, e.g. a simulator that will be capable of simulating a PSM.

List of Figures

List of Tables

Acronym

ADC Analog-to-Digital Converter

ACU Address Computation Unit

DAC Digital-to-Analog Converter

DSPs Digital Signal Processors

DPCE Data Parallel C Extension

DSSS Direct Sequence Spread Spectrum

EVP Embedded Vector Processor

EVP-SDK EVP Standard Development Kit

FPGAs Field Programmable Gate Arrays

FHSS Frequency Hopping Spread Spectrum

IF Intermediate Frequency

ISI Inter-Symbol Interferences

MPDU MAC Protocol Data Unit

PCU Program Control Unit

PEs Processing Elements

PIM Platform Independent Model

PSM Platform Specific Model

PHY Physical Layer

PLCP Physical Layer Convergence Procedure

PMD Physical Medium Dependent

PPDU PLCP Protocol Data Unit

PIM Platform-Independent Model

PSM Platform-Specific Model

PSSP Platform-Specific Source Program

RF Radio Frequency

RCF Reconfigurable Compute Fabric

SDR Software Defined Radio

SB3010 Sand-Blaster 3010

SIMD Single Instruction Multiple Data

Sandblaster IDE Sandblaster Integrated Development Environment

SFD Start of Frame Delimiter

VRE Virtual Radio Engine

VDCU Vector Data Computation Unit

XSL Extensible Stylesheet Language

XML Extensible Markup Language

Bibliography

[1] T. Arnaud. Multi-standard receiver architecture and circuits. In *European Solid-State Circuits Conference, Lisbon*, September 2003.

[2] R. Balakrisnan and K. Ranganathan. *A Textbook of Graph Theory*. Springer, 2000.

[3] H.-M. Blüthgen, C. Graßmann, and U. Ramacher. A software-programmable multiple-standard radio platform. In *IST Mobile and Wireless Communication Summit, Dresden*, June 2005.

[4] H.-M. Bluethgen, C. Grassmann, W. Raab, U. Ramacher, and J. Hausner. A programmable baseband platform for software defined radios. In *2004 Software Defined Radio Technical Conference, Arizona*, volume 3, page B155, November 2004.

[5] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[6] OpenMP compiler. http://phase.hpcc.jp/Omni/.

[7] IEEE/ANSI Std 802.11 1999 Edition, 1999-2006. http://standards.ieee.org/getieee802/802.11.html.

[8] R. Eigenmann and J. Hoeflinger, 2000. Parallelizing and Vectorizing compilers, Purdue Univ. School of ECE, High-Performance Computing Lab, ECE-HPCLab-99201, available at: www.ece.purdue.edu/ eigenman/reports/encyclo.pdf.

[9] K. Jarvinen et al. Gsm enhanced full rate speech codec. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 771–774, 1997.

[10] J. Eyre and J. Bier. The Berkeley Design Technology, Inc. available: http://www.bdti.com/articles/evolution.pdf.

[11] D3.2.2 Recommendations for API Definitions and Management of Core Software in Terminals. IST-2001-34091 SCOUT Project deliverable.

[12] Adelante Software Development Kit for VD3204x, 2005. C-Compiler User Manual.

[13] Gedae. available at: http://www.gedae.com/.

[14] J. Glossner, D. Iancu, J. Lu, E. Hokenek, and M. Moudgill. A software defined communications baseband design. *IEEE Communications Magazine*, 41(1):120–128, January 2003.

[15] J. Glossner, M. Moudgill, D. Iancu, G. Nacer, S. Jinturkar, S. Stanley, M. Samori, T. Raja, and M. J. Schulte, 2004. The Sandbridge Sandblaster Convergence Platform, pp. 1-21, 2005. Available: http://www.sandbridgetech.com/documents/.

[16] J. Glossner, T. Raja, E. Hokenek, and M. Moudgill. A multithreaded processor architecture for sdr. In *The Proceedings of the Korean Institute of Communication Sciences*, volume 19, pages 70–84, November 2002.

[17] R. Hossain, M. Wesseling, and C. Leopold. Application description concept with system level hardware abstraction. In *in Proc. IEEE Workshop on Signal Processing Systems Design and Implementation*, pages 36–41, November 2005.

[18] R. Hossain, M. Wesseling, and C. Leopold. Virtual radio engine - a programming concept for separation of application specifications and hardware architectures. In *in Proc. 14th IST Mobile and Wireless Communications Summit*, June 2005.

[19] R. Hossain, M. Wesseling, and C. Leopold. Towards automatic scheduling for software defined radio applications on parallel hardware. In *Proc. Karlsruhe Workshop on Software Radios*, pages 107–114, March 2006.

[20] R. Hossain, M. Wesseling, and C. Leopold. A new programming environment for software defined radio applications. *SEuropean Transactions on Telecommunication*, 19:61–66, 2007.

[21] The OpenMP Application Programming Interface. http://www.openmp.org/.

[22] S. Kolevatov. PhD thesis, A contribution to bridge design gap between software and hardware for complex signal processing systems in mobile communication, Faculty of Computer Engineering, University of Duisburg, to be submitted.

[23] S. Kolevatov, M. Wesseling, and A. Hunger. Generated implementation of a wlan protocol stack. In *IEEE 17th International Symposium on Personal Indoor and Mobile Radio Communications*, pages 1–5, 2006.

[24] Matlab. available at: http://www.mathworks.com/products/matlab/.

[25] H. Meyr. *Digital Communication Receivers: Digital Communication Receivers, Synchronization, Channel Estimation, and Signal Processing.* Wiley-Interscience, 1997.

[26] J. Mitola. *Software Radio Architecture: Object-Oriented Approaches to Wireless Systems Engineering.* John Wiley and Sons, 2000.

[27] MLDesigner. available at: http://www.mldesigner.com.

[28] B. Nichols, D. Buttlar, and J. P. Farell. *Pthreads Programming.* O'Reilly & Associates, 1996.

[29] B. OHára and A. Patrick. *IEEE 802.11 Handbook: A Designer's Companion.* Standards Information Network, IEEE Press, 1999.

[30] WP 4.1 Report on WP4.1 Research & Developed Technology. IST-1999-10287 CAST Project deliverable.

[31] Data parallel C extension (DPCE). http://www.crescentbaysoftware.com/dpce/.

[32] NXP Products. available: http://www.nxp.com/products/.

[33] Simultaneous Multithreading Project, 2006. available: http://www.infineon.com/.

[34] The Ptolemy Project, 1999-2006. available at: http://ptolemy.eecs.berkeley.edu/.

[35] The SynDEx Project, 1998-2006. available at: http://www-rocq.inria.fr/syndex/.

[36] A. Punsmann, 2005. Diplom Thesis, Entwicklung eines Codegenerators für eine Hochsprachen-Signalverarbeitungsbeschreibung mit Hardware-Randbedingungen, Fachhochschule Gelsenkirchen.

[37] U. Ramacher. Next generation embedded communication systems: Reconfigurability, flexibility and programmability. In *Intel Corp. Hillsboro/Oregon, On Chip Reconfigurable Computing and Communications Workshop*, May 2003.

[38] J. H. Reed. *Software Radio: A Modern Approach to Radio Engineering.* Prentice Hall PTR, 2002.

[39] Software Defined Radio Work Group Reports. available: http://www.sdrforum.org.

[40] M. J. Schulte, J. Glossner, S. Jinturkar, M. Moudgill, S. Mamidi, and S. Vassiliadis. A low-power multithreaded processor for software defined radio. *Journal of VLSI Signal Processing*, 41, 2005.

[41] H. Seidel and et. el. End-to-end reconfigurability: Towards the seamless experience. In *MIST Mobile and Wireless Summit 2005*, June 2005.

[42] Simulink. available at: http://www.mathworks.com/products/simulink/.

[43] D3.2.2 Specification and Simulation of Re-configurable Baseband Architecture. IST-1999-12070 TRUST Project deliverable.

[44] Software Communications Architecture Specification, JTRS 5000, and 2004 SCA V3.0. available at: http://jtrs.army.mil/.

[45] J. Steed, K. Barnes, and W. Lungdren. Gedae: A tool for implementing software radio on heterogeneous system. In *Software Defined Radio Technical Conference-Arizona*, pages A127–A131, Nov 2004.

[46] M. Steven. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[47] Morpho Technologies. http://www.morphotech.com/.

[48] Sandbridge Technologies. available: http://www.sandbridgetech.com/.

[49] Java Technology. http://java.sun.com/.

[50] P. v. der Wolf, E. de Kock, T. Henriksson, and Wido Kruijtzer. Design and programming of embedded multiprocessors: an interface centric approach. In *2nd IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, pages 206–217, 2004.

[51] C. H. van Berkel, F. Heinle, P. P. E. Meuwissen, K. Moerman, and M. Weiss. Vector processing as an enabler for software-defined radio handsets from 3g+ wlan onwards. In *2004 Software Defined Radio Technical Conference, Arizona*, volume 2, page B125, November 2004.

[52] W3C, XML, , Binary, Schema, DOM, and RDF). available at: www.w3.org/XML/.

[53] A. Wardhani, 2006. Thesis Report, WLAN 802.11 Hardware-independent Description in Simulink, MP PD TI 2, Siemens AG, Bocholt.

[54] E. D. Willink. The waveform description language: moving from implementation to specification. *IEEE Communication for Network-Centric Operation: Creating the Information Force*, 1:208–212, 2001.

[55] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 2004.

[56] Real-Time Workshop. available at: http://www.mathworks.com/products/rtw/.

[57] H. Zimmermann. Osi reference model - the iso model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432, 1980.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Dissertation selbständig und ohne unerlaubte Hilfe angefertigt und andere als die in der Dissertation angegebenen Hilfsmittel nicht benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder unveröffentlichten Schriften entnommen sind, habe ich als solche kenntlich gemacht. Kein Teil dieser Arbeit ist in einem anderen Promotions- oder Habilitationsverfahren verwendet worden.

Riyadh Hossain

Kassel, January 16, 2009