Pragma-Based Approach For Mapping DSP Functions On A Coarse Grained Reconfigurable Architecture

OMER MALIK

Licentiate Thesis in Electronic and Computer Systems
Stockholm, Sweden 2015
Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie licentiatexamen fredagen den 29 maj 2015, klockan 13:00 i Sal C, Forum, Isafjordsgatan 39, Kista.

© Omer Malik, May 4, 2015

Tryck: Universitetsservice US AB
Abstract

The complexity of System-on-Chips are growing exponentially. This increase in complexity demands from the algorithmic designers to improve their methodology for mapping applications (which are also becoming more advanced) on these complex hardware. A way of dealing with this complexity is to raise the level of abstraction and move towards Electronic System Level design tools but this move also poses some serious challenges (e.g., how to deal with and express parallelism, how to explore design space effectively, how to map a sequential code on these advance System-on-Chips). The work presented in this thesis makes an effort to deal with few of these challenges with the focus on DSP applications. This thesis presents a framework for mapping DSP functions on a coarse grained reconfigurable platform with the help of pragmas. These pragmas are guidance directive which are embedded in the high level code by the designer and they constrain the design implementation in terms of allocation and binding. They make the implementation general by covering a wide range of problems (rather than a single instance) and can generate multiple solutions varying in the degree of parallelism by changing the value of these pragmas from the same high level code. These pragma-annotated implementations can be used as generic library elements in a system level synthesis framework which can select the optimal functional implementation during design space exploration phase at system level. For each DSP function, there can be more than one implementation in the library where these would differ in architectural styles and the degree of parallelism resulting in components with different cost metrics (i.e. energy, latency, and area). In many of these dimensions, a pattern exists and allows a compact representation. Being able to capture such a pattern is a fundamental requirement to be able to build a library of pre-designed DSP functions. The proposed framework can efficiently capture the micro-architectural space of DSP functions for a regular coarse grained reconfigurable architecture. The methodology enables the developer to focus at higher level of abstraction and guide the synthesis process to generate specific architectural solutions bounded by pragmas. This makes the whole process more reliable, predictable, easy, and controllable from the designer’s perspective.

The focus of this thesis are these pragmas, their definition, syntax and usage with some examples to demonstrate their functionality. The thesis will also highlight how the proposed framework can efficiently capture the rich micro-architectural space of DSP functions for a regular coarse grained reconfigurable architecture and will also briefly describe a methodology to estimate the average energy of the generated solution. This framework is proposed as the basis for raising the abstraction level to automate synthesis of the entire physical layer applications.
Acknowledgements

I would like to thank my supervisor Professor Ahmed Hemani for giving me the opportunity to work in his group and for keeping his faith in me. His support and encouragement were extremely important to finish this thesis and I am more than thankful to him for helping me whenever it was required. I have learned many things from him and I am sure the experience gained in his company is going to be very helpful in future.

I would also like to thank all members of our research group for their support and friendship. Thank you Adeel Tajammul, Jamshaid Sarwar Malik, Haider Abbas, Muhammad Badawi, Jean-Michel Chabloz, Sandro Penolazzi, Nasim Farahini, Asad Hassan Jafri, Shou Li, Siavoosh, Liu Pei, Hassan Sohofi for your companionship and time. I would specially like to thank Muhammad Ali Shami for his friendship, guidance, advice and help through out these years. I have learned a lot from him while working together and sharing the office space. He was very patient in answering all the questions related to DRRA during the initial years of our research.

I am thankful to KTH in general and ES in particular for providing me an opportunity to work here. I am grateful to everyone here who helped me. I would like to extend my sincere gratitude to Professor Johnny Öberg, Professor Ingo Sander, Professor Elena Dubrova, Professor Christian Schulte, and Professor Zhonghai Lu for their friendly discussions, valuable advice and feedback during these years. I am specially thankful to Alina Munteanu for taking care of all administrative issues. She was always there whenever I required anything. KTH IT-support also took care of many things in very timely manner and I am thankful for their support.

I would also like to thank all the friends who were with me in Sweden. We spent some really quality time during all these years. It is impossible for me to mention everyone here as the list is extremely long.

My deepest gratitude goes to all my family members. I am specially thankful to my parents as they have always provided me with everything during all the stages of my life. They really missed me a lot during all these years and it was not possible to finish this work without their constant support along with prayers. I am extremely thankful to my wife Rabia. She has supported me a lot during all these years by taking care of all other “non research” related things (from shopping grocery till fixing house hold items), bearing the long working hours (including weekends) and comforting our beloved son Abdul Hadi in my absence (who always wanted to spend
more time with me).

Lastly I would like to thank Higher Education Commission of Pakistan for funding this research work.

Omer Malik,
March 2015
List of Acronyms

AGU  Address Generation Unit
ANSI  American National Standards Institute
ASIC  Application Specific Integrated Circuit
CGRA  Coarse Grained Reconfigurable Architecture
DCT  Discrete Cosine Transform
DFG  Data Flow Graph
DPU  Data Path Unit
DRRA  Dynamically Reconfigurable Resource Array
DSP  Digital Signal Processing
ESL  Electronic System Level
FFT  Fast Fourier Transform
FIMP  Functional Implementations
FIR  Finite Impulse Response
FPGA  Field Programmable Gate Array
FSM  Finite State Machine
GUI  Graphical User Interface
HDL  Hardware Description Language
HLL  High Level Language
HLS  High Level Synthesis
ILP  Integer Linear Programming
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP</td>
<td>Intellectual Property block</td>
</tr>
<tr>
<td>LSB</td>
<td>Least Significant Bit</td>
</tr>
<tr>
<td>MAC</td>
<td>Multiply And Accumulate</td>
</tr>
<tr>
<td>MPSoc</td>
<td>Multiprocessor System-on-Chip</td>
</tr>
<tr>
<td>MSB</td>
<td>Most Significant Bit</td>
</tr>
<tr>
<td>NOC</td>
<td>Network On Chip</td>
</tr>
<tr>
<td>NP</td>
<td>Nondeterministic Polynomiale</td>
</tr>
<tr>
<td>PHY</td>
<td>Physical</td>
</tr>
<tr>
<td>RFILE</td>
<td>Register File</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computer</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>SDF</td>
<td>Synchronous Data Flow</td>
</tr>
<tr>
<td>SOC</td>
<td>System-on-Chip</td>
</tr>
<tr>
<td>SLS</td>
<td>System Level Synthesis</td>
</tr>
<tr>
<td>SYLVA</td>
<td>System Level Architectural Synthesis</td>
</tr>
<tr>
<td>VESYLA</td>
<td>Vectorizing Symbolic Assembler</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td>VHSIC</td>
<td>Very High Speed Integrated Circuits</td>
</tr>
<tr>
<td>VLSI</td>
<td>Very Large Scale of Integration</td>
</tr>
<tr>
<td>YACC</td>
<td>Yet Another Compiler Compiler</td>
</tr>
</tbody>
</table>
Contents

Contents ix

List of Figures xi

List of Tables xiii

1 Introduction 1
  1.1 Library of DSP Functions ............................................. 2
  1.2 VESYLA ................................................................. 4
  1.3 Thesis Layout ......................................................... 5

2 Related Work 7
  2.1 Benefits of using pragma-based approach ......................... 10

3 DRRA Architecture & System Level Synthesis Framework 13
  3.1 DRRA Architecture .................................................... 13
    3.1.1 Datapath Unit ................................................. 14
    3.1.2 RFILE ........................................................... 15
    3.1.3 Sequencer ....................................................... 17
    3.1.4 Interconnect .................................................... 18
  3.2 DRRA System Level Synthesis Framework .......................... 18

4 VESYLA 21
  4.1 VESYLA Characteristics ............................................. 21
    4.1.1 Design Space Exploration ..................................... 21
    4.1.2 Interactive Design Tool with Automation .................... 22
    4.1.3 Library of Functions .......................................... 22
    4.1.4 Higher Level of Abstraction using MATLAB .................. 22
    4.1.5 Instant Verification .......................................... 23
  4.2 VESYLA Flow ......................................................... 23
  4.3 Pragmas: Their Types & Examples .................................. 25
    4.3.1 Constants & Range Pragmas .................................. 26
    4.3.2 Address Management Pragmas for RFILE ....................... 26
# List of Figures

1.1 Example of Library related to FIR Filter Design ................................. 4  
2.1 “An example of movable template” [12] ........................................ 11  
3.1 “DRRA Fabric Instance” [35] ...................................................... 14  
3.2 RFILE basic and extended instruction Modes ................................... 16  
3.3 “Address generation pattern with initial, middle & repetition delays” [8] 17  
3.4 “Overview of the SYLVA flow” [34] .............................................. 19  
4.1 Overview of VESYLA flow ............................................................ 23  
4.2 Example to differentiate between Scheduling and Synchronization .......... 25  
4.3 Example of RFILE allocation & binding pragmas ............................... 28  
4.4 RFILE pragmas with variation in MATLAB code ............................... 29  
4.5 An Example of VESYLA functional statement pragmas ...................... 31  
4.6 Partial parallel solution for MAC operation .................................... 32  
4.7 Fully Serial Solution for MAC operation after changing the pragmas ... 33  
4.8 Usage of “addertree” pragma ........................................................ 34  
4.9 “Multiple SIMD threads with distributed FSMs” [35] .......................... 36  
4.10 Synchronizing distributed FSMs in Sequencers ............................... 37  
4.11 Example of datapath and Sequencers state for calculating Initial Delays 39  
4.12 An example to compute the initial delay without any dependencies ....... 40  
4.13 Algorithm for computing initial delay ............................................ 41  
4.14 Data path and the algorithm for using middle delay ......................... 42  
4.15 Middle delay waveform .............................................................. 43  
4.16 Repetition Delay Example 1 ....................................................... 44  
4.17 Repetition Delay Example 2 ....................................................... 45  
5.1 Asymmetric FIR Filter ................................................................. 49  
5.2 5-tap Asymmetric FIR Filter (Fully Parallel Solution) ....................... 50  
5.3 n-tap Asymmetric FIR Filter (Partial Parallel Solution) ....................... 51  
5.4 Effects of changing dimensions and parallelism for FIR Filter ............. 52  
5.5 e(x) Taylor Series computation Example ...................................... 53  
5.6 Sin(x) Taylor Series Partially Parallel Example ............................... 54
5.7  Sin(x) Taylor Series Fully Parallel Example  . . . . . . . . . . . . . . . . 56
5.8  MATLAB/VESYLA code for FFT  . . . . . . . . . . . . . . . . . . . . . . . . . . 58
5.9  DataPath generated by VESYLA for computing FFT  . . . . . . . . . . . . . 60
5.10 MATLAB/VESYLA code for Matrix operations  . . . . . . . . . . . . . . . . . . . 61
5.11 The Energy Estimation Flow  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62

6.1  An example code introducing pragmas smartly  . . . . . . . . . . . . . . . . . . . 66
List of Tables

3.1 DPU modes of operation ........................................... 15

5.1 “Estimated and actual average energy for DSP algorithms” [9] ....... 63
Chapter 1

Introduction

As transistor size continues to decrease, System-On-Chip designs (SoC) are getting more complex and heterogeneous [1]. Depending upon the target application a multiprocessor system-on-chip (MPSoC) may have large number of microprocessors, digital signal processors (DSPs), accelerators, embedded memories, reconfigurable logic, shared memory and network-on-chip (NoC). The increase in complexity is also due to the fact that signal processing, wireless, and multimedia applications are also getting complex [2]. A mobile phone with real time video capabilities may roughly need to perform 460 billion operations per second [3]. Designers are facing the challenge to improve the performance by 1000X with only 40% increase in the power budget for future computing needs [4]. With the increase in complexity of SoC design and applications the algorithmic developers are also looking for better and innovative ways to facilitate and accommodate these complex designs but the gap between the progress of algorithm/application modeling tools and the complexity is increasing [1, 5]. This gap is an obstacle to reduce the design cycle and verification time (which in many cases exceeds the design time), to rapidly deliver the end product to the market and to efficiently explore the design space for optimally selecting the latency, area, and energy cost of the application to be mapped.

Raising the level of abstraction is an effective way to deal with this complexity and it also encourages the algorithmic developers to move from RTL synthesis to Electronic System Level (ESL) design tools which includes High Level Synthesis (HLS) and System Level Synthesis (SLS) tools. An HLS tool usually deals with a single function by accepting the specification (in a high level language e.g., c/c++, MATLAB, etc.) of the system behavior along with a set of constraints, and it generates the corresponding hardware while satisfying the set of constraints. This approach lets the designer to specify the algorithm in a high level language and the HLS tool automatically converts it into equivalent RTL code. Although it sounds simple but practically the synthesis process for generating the RTL is not as simple as the compilation of c code for software [6]. There are many obstacles which needs to tackled and some of the most challenging demands of this process is as follows:
1. It is easier to express the application (or algorithm) in sequential languages (e.g., C/C++ or MATLAB) as compared to RTL description but it is hard to extract the parallelism from the sequential description for complex applications. The quality of results is not very optimal while going from sequential model to equivalent parallel model of computation [7].

2. Another daunting task is to perform efficient design space exploration and to get not only accurate but quick feedback. Even a simplest datapath can be implemented in multiple ways (e.g., pipe-lined implementation, serial/parallel implementation, etc.) and each option has different performance, area, and energy cost metrics. It is a challenging task to quickly analyze these trade-offs while going from High Level Synthesis (or system level synthesis) to RTL[6]. These trade-offs need to be considered not only for the individual algorithm but also for the whole application. It is an extremely challenging task to perform efficient and quick design space exploration for ESL design tools.

3. High Level Synthesis (HLS) is a complex nine dimensional problem: Three types of information: time, type, and number of resources are synthesized for three types of operations: computational, storage, and interconnect. Searching such a large design space and coming up with an acceptable quality solution has been in many respects, the fundamental challenge that has plagued HLS.

4. User interaction is necessary to ensure high quality results. This way introducing the constraints and user interaction must be simple but sufficient to ensure maximum coverage without compromising the automation process.

5. The design to verification ratio is a major concern and moving at higher level of abstraction doesn’t necessarily bridge this gap.

The work presented in this thesis is an effort to overcome the above mentioned challenges for a Coarse Grained Reconfigurable Architecture (CGRA) fabric called Dynamically Reconfigurable Resource Array (DRRA) [8].

1.1 Library of DSP Functions

The design of DRRA is a step in the direction to achieve the goals described by ITRS [4]. The DRRA fabric is customized to map signal processing and high speed data parallel functions [8]. Modems and Codecs are also dominated by functions like FFT, FIRs, Viterbi etc. We call these as “standard DSP functions” because:

1. Their implementation (or architectural) space is very well understood.

2. They are largely regular and exhibit a pattern as the dimensions of the architecture changes.
These properties can be used to compactly capture the architectural space of such functions as temporal and topological patterns which can cover different architectural styles. Each style can vary in the degree of parallelism with solutions ranging from fully serial to fully parallel and anything in between. We propose developing a library of functions which are available in MATLAB toolboxes (e.g., communication toolbox, signal processing toolbox)\cite{9}. System Level Synthesis can use this library to synthesize the complete application. For each DSP function, there can be more than one implementation in the library where these would differ in architectural styles and the degree of parallelism resulting in components with different cost metrics (i.e., energy, latency, and area). The library will be populated with functions that can be parametrized and used in a context independent way. For example, there are different choices for implementing an FIR filter (e.g., Symmetric, Asymmetric, Transpose) as shown in Figure 1.1. Each implementation choice can contain multiple implementations differing in the architectural style (e.g., the number of taps in an FIR filter) and the degree of parallelism (e.g., fully serial FIR, fully parallel FIR, etc.). Each such implementation can have different energy, latency, area costs because it can be placed anywhere in the fabric. This also happens because the resource allocation and binding can be different for the same solution (for example, the coefficients and samples can be mapped to the same resource). These functions would differ widely in their complexity and richness of their micro-architectural implementation space. Some would be as simple as a convolution encoder while others far more complex e.g., convolution decoder, the Viterbi, etc. Building this library (for DRRA) is a one time effort and they are designed by an algorithmic designer with in-depth knowledge of both DRRA and DSP functions.

High Level Synthesis (HLS) is a complex nine dimensional problem and searching a huge design space for an acceptable quality solution has been the fundamental challenge that has plagued HLS. Having such a library of implementations of standard DSP functions, whose performance and energy are almost instantly and perfectly predictable will enable efficient synthesis of the whole modem and codec applications. This will be possible because a significant portion of PHY functionality is available in pre-designed form and will serve to reduce the search space and ensure good quality results.

A system level synthesis (SLS) tool to map applications (like multi-media and wireless communications) to DRRA is in the advanced stage of development \cite{10}. The goal of the DRRA system-level synthesis tool is to be able to handle complete modem and codec applications instead of handling an individual DSP function. This is completely different from typical HLS approach which usually handles a single function at a time. The DRRA system level synthesis framework accepts hierarchical model of the application and performs the system level synthesis under specified global latency and energy constraints. The DRRA SLS framework uses the (above mentioned) library of functions to synthesize the complete application.

This approach has several benefits that (we argue) enables efficient, high quality system level synthesis of complete application:
1. The PHY layer applications are dominated by the functions typified by the contents of the aforementioned MATLAB toolboxes; a fact proven by the popularity of these toolboxes in modeling. And by having pre-designed DRRA functions implementations from these toolboxes, the design space that the system level synthesis tool has to explore is significantly reduced; the building blocks for composing the PHY layer are coarse granular (different from the coarse granularity of the DRRA micro-architecture).

2. “DRRA is a regular fabric, whose elements are characterized very accurately with post layout data” [9]. The DRRA library elements contain precise information about how the mapping is done. This translates into a very accurate prediction of area, average energy, and latency cost of each possible implementation. This ability is critical for the system level synthesis optimization algorithms.

1.2 VESYLA

“VESYLA (VEctorizing SYmbolic Language Assembler)” [11] is a semi-automatic framework for implementing DSP functions on DRRA. It is a smart tool to speed-up the library development, by providing an abstract way to specify structural and temporal patterns. VESYLA is not however a synthesis tool and has no optimization capability at present. “VESYLA takes a MATLAB specification of a DSP function and generates configware for DRRA after performing timing and synchronization synthesis; an activity that is cumbersome, time consuming, and
error prone. The developer makes and explicitly expresses critical implementation decisions on how many resources are allocated and how the operators and operands are mapped to the allocated resources. These activities are very much like HLS, but in this case VESYLA user knows the targeted RTL structure and guides the tool towards it with the allocation and binding pragmas. VESYLA performs the scheduling, control-synthesis and does the code generation. Algorithmic developer guides VESYLA towards a specific architectural style by using VESYLA pragmas. A small change in these pragmas would result in a different architectural implementation and this change can also manipulate serial/parallel structure of the implementation; thus user is always in full control and can capture the architectural space effectively. These properties makes VESYLA an interactive design tool where it utilizes the human developer’s guidance and yet following the “push button” methodology to generate the RTL implementation” [11].

The focus of this thesis are these pragmas, their definition, syntax and usage with examples to demonstrate their working. The main contributions are as follows:

1. An effective tool which makes the process of design space exploration easy. The Same algorithmic level code can generate various design implementations using the same set of pragmas.

2. An efficient methodology which simplifies the process for extracting parallelism from sequential code and also bridges the gap between the sequential code and the equivalent parallel model.

3. Error prone and time consuming tasks of scheduling, synchronization, allocation and binding are handled by VESYLA as directed by the algorithmic designer [12].

4. The micro-architectural space of DSP functions can be compactly captured in a generic parametrized code using VESYLA, resulting in a readable and compact representation [11, 12].

5. It is possible to almost instantly and perfectly, predict the latency and energy consumption of DSP functions.

1.3 Thesis Layout

The rest of the thesis is organized as follows:

- Chapter 2 briefly describes some industrial and academic tools for mapping applications and algorithms on different architectures. It also describes that how our work is different from the works presented in this chapter.

- Chapter 3 briefly presents the DRRA fabric and the DRRA system level synthesis framework. The chapter describes the working of DRRA building
blocks and their functionalities. It also briefly describes the DRRA system level synthesis framework which helps in understanding the overall methodology for mapping the complete application using the library of functions.

• Chapter 4 presents the core contribution of this work VESYLA. As mentioned before at the beginning of chapter 1, there can be several architectural styles, each style in different serial-parallel implementations, storage hierarchy, etc. In many of these dimensions, a pattern exists and allows a compact representation. Capturing such a pattern is a fundamental requirement to be able to build a library of pre-designed DSP functions. This chapter answers the question of how to develop and compactly represent this multi-dimensional micro-architectural space for DRRA. It describes in detail about the pragmas, their types and usage with the help of some examples. The chapter also highlights, discusses and presents the solution of the problems which arises from converting the MATLAB code to the equivalent RTL representation for DRRA.

• Chapter 5 presents examples of DSP algorithms mapped to DRRA using VESYLA and it also presents the methodology to predict average energy and latency with fairly high degree of accuracy. The algorithm presented in this chapter includes FIR Filter design, FFT, Taylor series, and matrix operations. The focus of this chapter is mainly on the generation of solutions using pragmas.

• Lastly Chapter 6 presents the conclusion and future work.
Chapter 2

Related Work

There are many approaches and tools for mapping applications and algorithms on different types of architectures. They can be categorized as assemblers, ESL design tools, SLS tools, and HLS tools. In broader terms there are two choices for building this library of function. The algorithmic designer can either work at lower level of abstraction (manually generating these functions) or at higher level of abstraction (using a high level language). All these options have their own merits and de-merits.

Manually written codes are known for generating the best quality results as they are usually written by an algorithmic designer who knows the architecture like back of his hands and can utilize all the resources in the best possible way. This information and experience enables the programmer to generate fully optimized codes for mapping purpose. Programming at lower level of abstraction (or manual code generation) can be achieved by using a low level assembler. For example, [13] describes a programming model based on low level assembly programming language designed specifically for a 2-D mesh-based CGRA system. The reconfigurable cells (RC) of CGRA consists of 8-bit processing elements capable of performing basic arithmetic and logical functions. The expression language of this assembler is similar to the common assembly language and supports basic instructions for the RC. Although developing and maintaining an assembler is also a very straightforward, easy and simple task (as compared to automatically generating the code from a high level language) but the down side of this approach is that writing codes (at lower level of abstraction) is very time consuming and tedious task. Designer needs to plan everything like the number of resources, scheduling, allocation and binding, creating control data flow graphs, extracting the datapath, etc. All these tasks need to be done manually by the designer itself. Chance of making a mistake during all these stages/transformations is very high and debugging is also very time consuming. It is practically impossible to perform design space exploration for complex applications. Programming at lower level generates fully optimized solutions, but the complexity of applications has outpaced this manual method.

Designer can also use hardware descriptive languages like Verilog and VHDL
Programming at higher level of abstraction is easier as compared to pro-
moving at lower level of abstraction. An algorithm is specified in a high level language
and a tool generates low level details after analyzing this algorithm. Algorithmic
designer don’t need to spend time in planning various steps like scheduling, allo-
cation and binding, creating control data flow graphs, extracting the datapath etc.
These steps are automated by the tool. The designer’s primary responsibility is to
specify the algorithm correctly in the high level language and then the tool takes
care of all lower level tasks.

Significant amount (7x - 10x) of reduction was observed for a design of 1M
gates while going from lower level to higher level of abstraction and a 300k lines
of RTL code was reduced to 30k-40k lines of c/c++ code [14]. Clearly, this ap-
proach has significant advantage in terms of easiness and efficiency. The amount
of code is reduced significantly which saves time and reduces the chance of making
mistakes/errors. The process of design space exploration can be performed very
quickly and the whole process can be fully automated.

Many examples can be taken from both industry and academia, for examples,
[15] uses loop level parallelization for ADRES architecture [16] which consists of
functional units, register files, routing resources tightly coupled with a VLIW pro-
cessor. The reconfigurable array is used for mapping the loops while the VLIW
processor is used for the rest of the code. The input to the compiler is a C code
which is profiled for selecting suitable candidate (sections of code) to be mapped
on reconfigurable array to speed-up the computational extensive tasks. C code is
further optimized before passing it to the modulo scheduling algorithm to achieve
parallelism for optimal performance.

The high level synthesis like tool presented in [17] accepts DSP applications
represented in C language to target the MONTIUM architecture tiles. The CDFG
is partitioned into different clusters after initial transformations and these clusters
are mapped to the MONTIUM ALUs. Lastly the compiler performs scheduling and
allocation.

The synthesis tool presented in [5] generates RTL code from C or SystemC
description of the application for different hardware platforms (including FPGAs,
ASICs and embedded processors). The designer also need to manually identify the
sections of the code to be mapped on hardware and software.

The downside of such fully automated approach is lacking the quality of the
generated output when compared with the fully optimized hand written codes. These are closed solutions and the end user is restricted to the solution presented
by these tools. There are many scenarios, trade-offs and optimization decisions
required at various steps during this whole process and wrong decision at higher
level badly affect the quality of results as compared to design generated at lower
level of abstraction. The results must be generated quickly to reduce the time
between multiple iterations[6] and these results must be presented in such a way
that they are very well understood by the end user (in case user needs to modify
them manually for further optimizations). There is a minimum user interaction
and due to the complexity of design space it is very important to keep the designer involved in order to match the output (in terms of quality) to the manually (hand written) generated code.

User interaction can be introduced in many ways. Reference [19] is an academic tool targeting DSP applications. The tool accepts C language with set of mandatory constraints (clock period and throughput) and optional constraints (memory mapping and I/O timing diagram). Designer also specifies the target technology using a .lib file. The back-end of the tool extracts parallelism after generating a DFG and the tool synthesizes the architectural components according to design constraints. Lastly, it generates the target RTL which can also be mapped to FPGAs.

The work presented in [20] uses a restricted subset of C language for mapping applications on FPGAs. The algorithmic designer introduces a set of directives (embedded in the code) to guide the compiler for choosing different trade-offs. The model also supports parallelism but the designer needs to write multiple programs for multiple threads and needs to specify special directives/methods for enabling communication between these threads. This approach limits the options to optimize and automate the whole synthesis process. The designer needs to modify the source code significantly every time with the change in parallelism.

In [21] a C-based tool for mapping DSP algorithm targeting an RTL processor is presented. It is an interactive design tool which uses designer’s feedback at various stages of the synthesis process starting from high level C code down to RTL. Designer provides some constraints (related to clock period, allocation/binding etc.) by using a GUI and the tool generates preliminary results based on the constraints. Designer can change the constraints based on these results to improve quality and have a better optimization. This is an iterative process and the final output is the RTL for the targeted hardware.

In [22] an interactive design tool is presented which synthesizes algorithms described in C++ code. Designers specify the algorithm in an untimed C++ along with synthesis constraints using a GUI or scripts. The constraints are divided into optional constraints (e.g., I/Os, loops, storage etc.) and mandatory (e.g., target technology, desired frequency etc.). The designer can view the impact of these constraints at different stages of transformations and can change these constraints for generating desired/optimized solution. Lastly the tool generates an RTL code suitable for the targeted hardware architecture.

Reference [23] generates an RTL code staring from untimed SystemC code. The designer specifies technology file for targeted hardware architecture along with control/design constrains (for selecting target clock period, loop unrolling, etc.).

Reference [24] presents a behavioral synthesis tool for mapping DSP functions on FPGAs. The algorithms are described in MATLAB and the synthesis tool generates equivalent RTL models along with VHDL simulation testbenches. The algorithmic designer can also perform design space exploration by specifying different directives in the MATLAB code (e.g., the number of TAPs in an FIR Filter).

Reference [25] supports c/c++/SystemC description of DSP algorithms with directives and constraints to guide the HLS process. RTL code targeting FPGAs is
generated after performing optimizations (such as loop transformations, exploiting parallelism, reducing code complexity, memory optimizations).

Reference [26] introduces an architecture description language targeting CGRA templates. Designer formally describes the architecture and its properties (e.g., the number of Functional units, ports, processing elements) at higher level of abstraction using the architecture description language and also specifies the interconnection networks.

2.1 Benefits of using pragma-based approach

The work presented in this thesis uses “a pragma-based approach, where the developer, gives hints about the architecture to the high-level synthesis tool in the form of pragmas. This is done in a symbolic and parametric way and is embedded in comments. This approach enables the algorithmic designer to guide the high level synthesis process, which is a key advantage. The designer can fully utilize the resources and features available in DRRA for mapping an algorithm.” [12] The complex task of automatically finding an optimal solution for scheduling, allocation, and binding from the huge search space is simplified by shifting the responsibility to the algorithmic designer and these pragmas helps in exploring the design space efficiently by just selecting a solution from that space. They are only introduced according to the needs and this approach ensures minimum (but sufficient) user interaction to cover maximum constraints while making the whole process easy. The success of our approach lies in identifying the right balance between low level and high level of abstraction and the mechanism through which the input is going to fed to the tool. The information is given in a way that it can be easily decoded and processed by the tool without adding any additional overhead. The solution presented here not only provide a way of user interaction (without compromising the automation process) at higher level of abstraction to generate good quality results (matching manual hand written codes) but it also allows designers to specify design functionality at higher level of abstraction and let the algorithmic designer quickly experiment with various tradeoffs from a single functional specification.

All the approaches discussed above can be categorized as HLS-type techniques (despite the fact they differ in details). Although our approach has similarities to the techniques presented earlier in this section but it is different in at least four ways:

1. Finding an optimal solution for allocation/binding in HLS is an NP complete problem. The solution presented here reduces this search space (for finding the optimal solution of the problem related to allocation/binding) because the architectural guidance (presented in this work) is parametric in term of the size of the problem. The pragmas cover a wide spectrum of the problem (e.g., the number of tap in the FIR Filter or size of the FFT) rather than a specific single instance (e.g., 64-point FFT). With the huge search space it is
2.1. BENEFITS OF USING PRAGMA-BASED APPROACH

extremely difficult to perform automatic synthesis from the algorithmic level because most of the tools take micro architectural elements as the building blocks and this choice leads to a huge search space.

2. Another factor which also helps in reducing this search space is that allocation/binding pragmas (presented in this solution) are also parametric in terms of parallelism. One can generate multiple solutions varying in degree of parallelism by changing the values of these pragmas for the same dimension of the problem.

3. The resulting solution can be rotated and mapped anywhere on the CGRA fabric because the “pragmas are relative to a symbolic position in the DRRA fabric” as shown in Figure 2.1 [12].

4. Predicting performance/energy estimations for ASICs without the PHY design is not an easy task and in case of FPGA designs the error margin is up to 20% [27]. “Optimization decision taken at the algorithmic level requires very accurate estimation or look ahead functions for latency and energy that have not been available for the standard cell based ASIC or FPGA. The solution presented in this thesis comes up with very accurate energy estimations without even executing the algorithm on actual hardware” [9].

Figure 2.1. “An example of movable template” [12]
Chapter 3

DRRA Architecture & System Level Synthesis Framework

This chapter briefly describes the DRRA architecture along with system level framework for mapping applications on DRRA. This will help to understand how the standard DSP functions are mapped to this fabric and how the implementation pattern is represented.

3.1 DRRA Architecture

“Dynamically Reconfigurable Resource Array [8] is a Coarse Grained Reconfigurable Architecture and platform for multiple, complete Radio and multimedia applications. It has resources for physical layer, protocol processing layers, application, system control and run-time management” [28]. All these resources are organized as a regular, seamlessly connected fabric. A 3D integrated memory die is planned to provide a streaming, high bandwidth distributed memory to DRRA [28]. The complete DRRA architecture has not been fully implemented, though the concept has been defined in quite some details.

Figure 3.1 shows an instance of a 7 x 2 DRRA fabric and this instance is only a fragment. “On a 10mm X 10mm chip in 90nm technology, 324 DRRA cells can be accommodated”[12].

DRRA PHY resources are:

- Logic and morphable Data path Unit or DPU [29]
- REgister FIle or RFILE [30]
- Micro-coded hierarchical sequencing machine or Sequencer [31]
- a Seamless, sliding window, circuit switched interconnect fabric or Interconnects [32, 33]
3.1.1 Datapath Unit

DPUs are used for computing arithmetic and logical operations [29]. These DPUs operate on 16-bit integer units and four inputs with two outputs. The outputs and inputs are connected to the DRRA interconnect network for sending/receiving data. These DPUs can be connected with RFILEs or with other DPUs to achieve the parallelism. These DPUs can be configured in different modes using a Sequencer.

The modes of DPUs are shown in Table 3.1. “Arithmetic mode provides MAC, with internal and external accumulation which can also be used for symmetric FIR MAC, where the symmetric samples are added before doing the multiplication with the coefficients. It also provides half of radix-2 butterfly for real or imaginary number” [8, 28, 11]. Arithmetic modes can also be configured as in different add or subtract modes as shown in Table 3.1. DPUs performs all the add/subtract based operations in a single cycle and the multiply based operations in 4 cycles. An additional cycle is required to do the optional saturation. The logical part can be configured to perform the comparison operations.

DRRA DPU is also capable of performing post processing functions like saturation, truncation/rounding, overflow, underflow check. The I/Os of DRRA are 16 bits but multiplication operation results in a 32 bits number and accumulation will give 33 bits. The bit-width can be reconfigured to be represented as anything from
### 3.1. DRRA ARCHITECTURE

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
<th>DRRA Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Symmetric FIR with Internal MAC</td>
<td>( \text{Out}_0 = (\text{In}_n + \text{In}_m) \times \text{In}_1 ) ( \text{Out}_1 = (\text{In}_n + \text{In}_m) \times \text{In}_1 + \text{Acc} )</td>
</tr>
<tr>
<td>1</td>
<td>Symmetric FIR with External MAC</td>
<td>( \text{Out}_0 = (\text{In}_n + \text{In}_m) \times \text{In}_1 ) ( \text{Out}_1 = (\text{In}_n + \text{In}_m) \times \text{In}_1 + \text{In}_3 )</td>
</tr>
<tr>
<td>2</td>
<td>Asymmetric FIR with Internal MAC</td>
<td>( \text{Out}_0 = (\text{In}_n \times \text{In}_m) ) ( \text{Out}_1 = (\text{In}_n \times \text{In}_m) + \text{Acc} )</td>
</tr>
<tr>
<td>3</td>
<td>Asymmetric FIR with External MAC</td>
<td>( \text{Out}_0 = (\text{In}_n \times \text{In}_m) ) ( \text{Out}_1 = (\text{In}_n \times \text{In}_m) + \text{In}_3 )</td>
</tr>
<tr>
<td>4</td>
<td>Radix 2 FFT</td>
<td>( \text{Out}_0 = \text{In}_2 + (\text{In}_n \times \text{In}_m) ) ( \text{Out}_1 = \text{In}_3 - (\text{In}_n \times \text{In}_m) )</td>
</tr>
<tr>
<td>5</td>
<td>Radix 2 FFT</td>
<td>( \text{Out}_0 = \text{In}_2 + (\text{In}_n \times \text{In}_m) ) ( \text{Out}_1 = \text{In}_3 - (\text{In}_n \times \text{In}_m) )</td>
</tr>
<tr>
<td>7</td>
<td>Two input adder</td>
<td>( \text{Out}_0 = \text{In}_n + \text{In}_1 ) ( \text{Out}_1 = \text{In}_3 - \text{In}_3 )</td>
</tr>
<tr>
<td></td>
<td>Two input Subtraction</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Difference of Sum</td>
<td>( \text{Out}_0 = (\text{In}_n + \text{In}_m) - (\text{In}_n + \text{In}_m) )</td>
</tr>
<tr>
<td>9</td>
<td>Sum of Difference</td>
<td>( \text{Out}_0 = (\text{In}_n - \text{In}_m) + (\text{In}_n - \text{In}_m) )</td>
</tr>
<tr>
<td>10</td>
<td>Four input adder</td>
<td>( \text{Out}_0 = (\text{In}_n - \text{In}_m) + (\text{In}_n - \text{In}_m) )</td>
</tr>
</tbody>
</table>

Table 3.1. DPU modes of operation

8 to 16 bits by using these post post processing functions.

#### 3.1.2 RFIRE

DRRA Register File is 64 word 16-bit register file with dual read and write ports [8]. The I/Os of RFIRE are connected to the interconnect network for providing high speed parallel data to the DPU s. RFIRE has a DSP style AGU [30, 8] that processes the address generation pattern stored in register bank of the RFIRE. This AGU enables streaming data for DSP algorithms (like FIR, FFT etc.) to exploit parallelism and is capable of producing dedicated addressing modes, generating arbitrary addresses, repetition of instructions, synchronization of different streams, generating elastic steams, etc.

Figure 3.2 shows the basic and extended instruction modes for RFIRE. These are also programmed by their respective Sequencers.

RFIRE AGU has two address generation modes:

**Basic Instruction Mode using Linear addressing**

This mode generates the addresses linearly between the range of addresses. This mode is selected by writing '0' in the mode bit of the basic instruction as shown in Figure 3.2. The range of addresses is decided by the start/end address field
of the basic instruction mode and one can also increment/decrement between this range for generating the desired addresses. The output control bits are used to select either one or both ports of the RFILE for sending/receiving the data. The instruction complete bit determines if there is a need for an extended instruction or not.

**Basic Instruction Mode using Bit Reverse addressing**

Bit reverse addressing mode is a dedicated addressing mode for computing the FFTs. “The start and end address range defines the boundaries of the virtual buffer which depends upon the size of the FFT. The addressing depends upon the number of stages and the mode is defined in a way that it could generate addresses for any single stage (if required) or range of FFT stages (like 1 - 6 in case of 64-point FFT)” [30].

**Extended Instruction Mode**

The extended instruction repeats the basic instruction but one can also change the start/end address range in each repetition to generate a different address pattern (for example, creating circular buffer etc.) in every repetition. This is achieved by using repetition incr/dcr value. This mode works with both basic instructions modes.

DRRA AGUs also have programmable delays for creating different address patterns, achieve synchronization, support various data rates and create elastic streams [8, 30]:

---

**Figure 3.2.** RFILE basic and extended instruction Modes
3.1. DRRA ARCHITECTURE

Initial Delay

This delay is only used once at the beginning. It delays the address generation pattern by 'n' number of cycles. It is mainly used for synchronizing different ports of the same (or different) RFILeEs. For example, if a computation requires data from two parallel ports and these ports are in the same RFILE then one of them will start executing before the other. This happens because instructions are executed in a sequential manner by the Sequencer. In such a situation, initial delay can be used to delay the execution, so that both ports can generate data synchronous with each other.

Middle Delay

These delays are used when two connected streams have different data production/consumption rates depending upon the nature of the algorithm or resource dependency.

Repetition Delay

The extended instruction part allows for repetition of address generation pattern. This address generation pattern can be delayed by using repetition delays.

VESYLA performs the cumbersome task of delay synthesis. This process with examples is described in chapter 4.

![Address generation pattern with initial, middle & repetition delays](image)

Figure 3.3. “Address generation pattern with initial, middle & repetition delays” [8]

3.1.3 Sequencer

Sequencer is a simple FSM that controls a single DPU, RFILE and the switch-box of the Interconnects in its own DRRA cell. They are organized in two rows connected by the interconnection network. Sequencer configures DPUs and RFILEs in different modes and also configures the appropriate switchbox accordingly. A single sequencer can also control the neighboring sequencers within the 3 hop distance.

Sequencer program memory is built from flip-flops. It consists of 64 instructions and the length of each instruction is 36 bits. The 4 MSB bits are used as instruction code while the rest of the bits are used as instruction operands. It takes maximum of 16 cycles to configure a DRRA cell.
### 3.1.4 Interconnect

The DRRA interconnect scheme enables I/Os of DRRA resources in different or same DRRA cells to connect to each other. The DRRA resources (DPUs, RFILEs and Sequencers) are organized in two rows and seven columns connected by horizontal and vertical buses. The output of DRRA resources (DPU, RFILE) are carried by the horizontal buses (between the rows), 3 columns on each side. “The vertical buses feeds the inputs of RFILEs and DPUs. Each column can reach up to 3 columns to the right and 3 columns to the left” [32, 35]. This distance shrinks at the extreme left or extreme right of the fabric. This approach enables 8 RFILEs and 8 DPUs to connect directly to each other thus ensuring single cycle latency in a non-blocking manner [32, 33, 8].

Programmable switchboxes are used at the intersection of these horizontal and vertical buses for selecting the appropriate outputs and loading them to the desired inputs of DRRA resources. The size of these switchboxes are 14 x 12 and only one switch can access a bus at a time. A 12 x 16 bits configuration memory is also attached to these switchboxes. This memory is used for connecting the correct output (from 14 lanes) to the desired input (of 12 lanes). DRRA sequencer programs the individual switchboxes accordingly.

### 3.2 DRRA System Level Synthesis Framework

The DRRA compiler framework shown in Figure 3.4 is not the focus of this paper. A brief overview is however given here for completeness and to describe the context for the main contribution of this thesis. DRRA compiler framework is called SYLVA which stands for System Level Architectural Synthesis. This is a system level framework which maps DSP subsystem (like Modem, Codecs, etc.) to hardware including ASIC, FPGA and DRRA.

SYLVA takes three inputs; a SDF graph, FIMP (Functional Implementations) library and constraints with optimization goals [10, 34]. The output of SYLVA is the RTL description of the DSP sub-system targeted for a specific platform (ASIC, FPGA, and DRRA).

SYLVA takes three inputs; a SDF graph, FIMP (Functional Implementations) library and constraints with optimization goals [10, 34]. The output of SYLVA is the RTL description of the DSP sub-system targeted for a specific platform (ASIC, FPGA, and DRRA).

The SDF graph is generated by industry standard tools and each node in this SDF graph represents an individual DSP function (for example, FFT, and FIR). Optimization constraints (total latency, maximum sample intervals, total area) are provided to SYLVA for minimizing the energy and area. SYLVA synthesizes in terms of FIMPs which are pre-designed and verified functional implementations of reusable objects [10]. These functional implementations are organized in the form of libraries and these libraries are available for ASIC, FPGAs and DRRA. Multiple FIMPs exist for an individual function which differs in architectural styles and implementation options thus affecting the cost of in terms of latency, area and energy. For example, an ASIC FIMP library would have multiple implementations of 64-point FFT differing in micro-architectural styles (radix-2, radix-4, mix-radix) and the degree of parallelization. Each node in SDF has a potential FIMP in
3.2. DRRA SYSTEM LEVEL SYNTHESIS FRAMEWORK

Figure 3.4. “Overview of the SYLVA flow” [34]

this library and SYLVA selects the most suitable FIMP from the library which satisfies the optimization constraints. FIMPs are selected during the design space exploration phase.

The design space is explored in three dimensions which includes:

1. Selecting the number of FIMPs,
2. Selecting the type of FIMPs
3. Managing the pipeline between different FIMPs in the SDF.

The followings are also considered while selecting the number of FIMPs

- Individual FIMP cost (for example, the energy constraint)
- Functional level parallelism (functions that can execute in parallel in absence of data dependency)
- Degree of parallelism (for each function node).

A feasible schedule that satisfies all system constraints and optimization goals is generated. SYLVA automatically generates the necessary glue logic to combine different FIMPs in the SDF to get a complete working model and generate RTL descriptions for different platforms (ASIC, FPGA and DRRA) [10].

FIMPs are the building block for SYLVA synthesis and VESYLA builds the FIMPs library for DRRA. This library helps in reducing the search space thus
reducing the size of optimization problem. The system level synthesis becomes simpler by using these libraries of functions.
Chapter 4

VESYLA

VESYLA [11, 12] is a tool for mapping DSP functions on DRRA. Input to VESYLA is a function written in MATLAB and the output is a configuration file (or set of files) for executing the MATLAB code on DRRA. VESYLA takes care of all tedious, time consuming and error prone tasks (like timing and synchronization synthesis, dependency analysis) while hiding all the low level details from the designer [12, 35]. User while remaining at higher level of abstraction gets the expected results in a single iteration instead of going through all these processes. MATLAB code is embedded with user inserted pragmas. These pragmas are hint-directive to guide VESYLA towards specific architectural style. A small change in these pragmas can result in different architectural implementation including the variation in serializing or parallelizing the algorithm. VESYLA allows the designer to considerably influence the synthesized architecture. “The developer makes and explicitly expresses critical implementation decisions on how many resources are allocated and how the operators and operands are mapped to the allocated resources. These activities are very much like HLS, but here VESYLA users obviously know the targeted RTL structure and guides the tool towards it” [11]. This approach enables the algorithmic designer to fully utilize the resources and features available in DRRA in the best possible way as desired.

4.1 VESYLA Characteristics

VESYLA offers the following to the end-user when mapping algorithms to DRRA:

4.1.1 Design Space Exploration

VESYLA simplifies the process of design space exploration. The same code can be used to generate various designs varying from fully serial to fully parallel and anything in between. The pragmas (embedded in the code) can cover range of
problem dimensions rather than a specific instance. For example, one can write a single code for 'n' Taps (dimension of the problem) FIR Filter which can generate (specific instances like) 2, 3, 4, . . . 'n’ solutions as shown in Figure 1.1. “These pragmas are relative to a symbolic position in the DRRA fabric which enables the generated solution to be placed anywhere in the fabric” [9]. These implementations have different average energy, latency and area cost. VESYLA also helps in estimating these factors for every solution. The user simply needs to run the MATLAB code in order to make early estimations and trade-offs (without even executing the code on DRRA or generating the netlists).

4.1.2 Interactive Design Tool with Automation

“VESYLA is an interactive design tool which utilizes the human developer guidance and yet follows the “push button” methodology” [11]. The important criteria to consider is that the manual interaction (or guidance) never interferes with automation. The guidance comes in the form of pragmas which are embedded in the code by the developer and small change in these pragmas would result in different architectural implementation. The designer can control the whole process and can effectively capture the architectural space. This sort of interaction facilitates the purpose of automation, as it is only done at the beginning of the process.

4.1.3 Library of Functions

This library consists of commonly used DSP functions found in MATLAB’s signal processing and communication processing toolboxes [9]. “For each function, the DRRA library contains multiple implementations, differing in dimension, architectural style and degree of parallelism; each such implementation have a potentially different average energy, latency and area costs” [9]. This library is a part of DRRA system level synthesis framework (SYLVA) which uses them for optimally mapping the whole application on DRRA.

4.1.4 Higher Level of Abstraction using MATLAB

VESYLA uses MATLAB which is very well known to DSP designers and algorithm developers. Since it is widely used by the programmers which enables the library developer to model the function easily and can use VESYLA to take care of low level details. This mechanism ensures that all the time consuming and error prone tasks are automatically resolved by the back-end of the tool while the user programs at higher level of abstraction. Although c/c++ is also very popular among the developers community but it is not the natural language for mapping DSP algorithms when compared with MATLAB. Support for array operations (both on matrices and scalars) with powerful set of built-in library functions makes MATLAB an obvious choice for DSP functions.
4.1.5 Instant Verification

MATLAB code is executable in MATLAB environment and can act as a golden model. The output of the VESYLA generated configware (after executing it on DRRA) can be verified against this golden MATLAB model. One needs to execute the algorithms in both environments and compare the outputs for differences. Absence of any difference would validate that the tool is working correctly. The feature works by using the print statement inside the MATLAB code.

4.2 VESYLA Flow

This section briefly summarizes the steps involved in generating the low-level output from the high-level input (MATLAB code) using VESYLA. Figure 4.1 shows the complete VESYLA flow. The front-end of the tool accepts a code written in MATLAB. This code is instrumented with user-defined pragmas. As discussed before these pragmas are architectural guidelines and guides VESYLA towards a specific architectural choice. These pragmas sweep the architectural space in terms of degree of parallelism and also helps VESYLA to determine the number of resources and binding various operators/operands to these resources. It is assumed that user (who wrote the MATLAB code) is strongly familiar with the DRRA architecture, VESYLA’s features/limitations and usage of pragmas.

VESYLA front-end performs lexical and syntactical analysis on MATLAB algorithm. The parser is a grammar-based design and is developed using LEX & YACC. VESYLA builds an intermediate graph on this initial information. The back-end
of the tool deals with synchronization and scheduling issues. The tool traverse this intermediate graph back and forth to resolve resource/data dependencies while considering the allocation/binding pragmas and also makes the critical decision about parallelizing or sequentializing particular sections of the code. Lastly (before generating the final output) VESYLA synchronizes the operations by using the following techniques:

- Synchronization using AGU delays
- Synchronization using DPU Counters
- Synchronization using wait & stall statements
- Synchronization by adding new instructions

VESYLA may use some or all of these techniques in order to execute the algorithm and synchronizing different interacting FSMs. After performing all necessary transformations the back-end generates the configware in the following forms:

- VHDL test-bench for executing the algorithm at the RTL.
- VHDL test-bench for executing the algorithm at the gate-level
- Files for executing the algorithm on MANAS (Manual Assembler which facilitates the programmer to work at the low level).
- Details about the average energy consumption

One should not confuse synchronization and scheduling. These are two different (yet related) steps in the flow [35]. Scheduling determines the order in which operations are executed while synchronization is used to prevent unpredictable timing issues in order to ensure timely and correct execution of the function. Scheduling ensures effective usage of resources and synchronization avoids conflicting usage of DRRA resources which would eventually lead to incorrect operation.

Consider the piece of code shown in Figure 4.2. The MATLAB code consists of three functional statements which are semi dependent on each other. These functional statements are going to be mapped on DRRA using VESYLA (Since the pragmas are not discussed yet, so they are omitted from this piece of code to make it simpler for better understanding).

Apart from variable X₀ and Y₀, all the operators and operands are not sharing any resource (i.e. no resource dependency). Variable X₀ and Y₀ are sharing the same port of a RFILE. Figure 4.2 shows the scheduled output of these statements if they are executed in the order as specified in “Non-Efficient Scheduling”. It takes 11 to complete the execution. The yellow lines are pointing towards the resource dependency among the variables X₀ and Y₀ which is in the form of port re-usage [35]. Although both variables are using different address space in the same RFILE but there is a waiting time of 3 cycles before the port can be reused again. This
implies that $Y_0$ needs to wait for 3 cycles to begin its execution, since it is sharing the same port with $X_0$. One can reduce the number of cycles by scheduling “(B)” after “(C)”. This move reduces the number of cycles from 11 to 6.

Scheduling saves the number of cycles but synchronization makes sure that different FSMs (for RFILeS and Sequencers) interact with each other properly for correct execution of the algorithm. Synchronization needs to be performed irrespective of the scheduling algorithm in order to correctly execute the function.

4.3 Pragmas: Their Types & Examples

This section describes the pragmas, their syntax, semantics and demonstrate their usage with examples. The pragmas are introduced in the code by using “%$” symbol. The “%” sign in MATLAB is used for introducing comments. If it is followed by a “$” sign then VESYLA treats it as a pragma rather than an ordinary comment. The pragmas are broadly divided into the following four categories:

1. Constant & Range Pragmas
2. Address Pragmas
3. Allocation & Binding Pragmas
4. Functional statement Pragmas

4.3.1 Constants & Range Pragmas

These pragmas are identified by using “%$Range” and are used to make the implementation general and generic. These are always used in combination with other pragmas and can help in sweeping the architectural space in terms of locality (selecting a cell within DRRA) and dimension (of the algorithm) [12]. These are optional pragmas and it is up to VESYLA to determine the range in the absence of these pragmas. These pragmas are used to specify the following:

- Range of addresses to be used in RFILE:
  - User identifies the specific range of addresses within a single/multiple RFILE(s) by using this pragma and VESYLA accommodates the user requirements accordingly (An example can be seen in section 4.3.4).

- Range of required resources in DRRA:
  - User identifies the specific range of DPU(s) or RFILE(s) required in a particular implementation by using this pragma.

4.3.2 Address Management Pragmas for RFILE

Address management pragmas are designed to use the RFILE addressing mode and addressing space in a convenient and efficient way [12]. Different categories of Address Management Pragmas are as follows:

Address Mode Pragma

These are used to specify the addressing modes (linear & bit reverse) available in the AGU of RFILE [12]. “%$linear” and “%$bitreverse” are used to represent linear and bit reverse modes respectively. These are optional pragmas and if they are not used then VESYLA determines the correct addressing modes by analyzing the MATLAB code.

Address space pragma

The address space of a RFILE can be categorized as logical or physical address space. This pragma helps the user to manage the address space as desired [12]. “This an optional pragma and if not used then the address space is always used as logical address space. If user wants to use specific address space in RFILE then one should insert “%$physical” address pragma, which directs VESYLA to treat the address space as physical addresses in RFILE” [12].
Address Distribute Pragma

This pragma facilitates the designer to control the distribution of address space in multiple threads. For example, it is possible to distribute a vector of size \( N \) evenly over \( M \) threads such that each thread contains \( N/M \) vectors. If address distribution pragma, "\$ no_distribute" is used then each thread ends up containing \( N \) vectors. This is again an optional pragma and in the absence of this pragma the address space is distributed linearly among multiple threads. This scenario is explained with an example in section 4.3.4.

4.3.3 Allocation & Binding Pragmas

These pragmas are used for allocation and binding of computational and storage resources. “Topological relationship of the DRRA resources is also part of the mapping specifications in the pragmas” [11].

They are used by VESYLA to determine:

- The number of required resources (DPUs and RFILEs) implied by the user.
- “How operands and operators are mapped to the DRRA resources involved in that particular implementation” [11, 12].

This approach serves the following purposes:

1. With the help of these pragmas, user specifies a pattern of a particular implementation.

2. This pattern sweeps the implementation space in terms of the degree of parallelism from fully serial to fully parallel and the number of SIMD/MIMD threads.

3. “This is a clean, simple and efficient method which lets the designer focus at higher level of abstraction” [12].

These pragmas also begin with “\$” sign and followed by one of the following keywords; “RFILE” or “DPU” [12]. As the name suggests that RFILE and DPU are pointing towards the Register Files and datapath units of DRRA respectively. These keywords are followed by the row and column indexes which represents a single DRRA cell (in case of serial implementation) or multiple DRRA cells (in case of mapping the function in parallel). The value of row and column indexes can either be an absolute number value or a static compile time constant used in the declarative section. Allocation and binding pragmas are further sub-divided into the following:

- RFILE allocation & binding pragmas
- DPU allocation & binding pragmas
4.3.4 RFILE allocation & Binding Pragmas

These pragmas are used in combination with address management pragmas and helps in using RFILE of DRRA purposefully.

The syntax for using RFILE Pragmas is as follows:

```
%% RFILE[Row indexes, Column indexes]; (port_allocationPragma);
(Address Management Pragma)
```

The syntax is divided into three parts. The first part ([Row indexes, Column indexes]) is pointing toward the allocation/binding instances for the variables used with this pragma. The usage of the second part (address space pragma, address distribute pragma, addressing mode pragma) is already described in the previous section.

Consider the code presented in Figure 4.3 with pragmas written in comments. The statements used in the code are explained as follows:

- Variable `column_range` (Range Pragma) in statement 4 varies between the values 1 and M. This variable is used with RFILE allocation and binding pragma (in statement 6) to use column 1 till M in the DRRA fabric.

- Similarly variable `address_range` (Range Pragma) in statement 5 varies from 1 to N. This variable is used with RFILE allocation and binding pragma (in statement 6) to use the address space from 1 till N in the DRRA RFILE.

- RFILE allocation and binding pragma is used in statement 6 that defines a variable `z` that needs to be mapped on RFILE of DRRA. The location(s) of the RFILE(s) to be used is determined with variables `row` and `column_range`. 

4.3. PRAGMAS: THEIR TYPES & EXAMPLES

**Figure 4.4.** RFILE pragmas with variation in MATLAB code
Figure 4.4 shows three variants of this MATLAB code which use different set of pragmas in order to generate different type of RFILE(s) configurations for DRRA:

In Figure 4.4-a:

- The value of M is 1, which implies that variable z is mapped to row 0 and column 1 (as column_range varies from 1 to M) of DRRA.

- The variable N (= 15) is directing VESYLA to linearly distribute 15 addresses. The choice of using the physical locations of these addresses (in RFILE) is up to VESYLA (because address distribute pragma is not invoked). In this particular case VESYLA selects physical addresses (10 – 24) in the RFILE for mapping the address space (1 : 15).

In Figure 4.4-b:

- The value of M is 2, which implies that variable z is mapped to row 0 and columns 1,2 (as column_range varies from 1 to M) of DRRA. This small change in pragma generates multiple threads as shown in Figure 4.4-b.

- The variable N (= 15) is directing VESYLA to linearly distribute 15 addresses on two (since M=2) DRRA cells. The user has also specified about the physical distribution of these addresses by using “%$Physical” pragma. This directs VESYLA to use physical address (16 – 29) in respective RFILEs. Each RFILE is going to operate over N/M addresses.

In Figure 4.4-c:

- The value of M is 3, which implies that variable z is mapped to row 0 and columns 1,2,3 (as column_range varies from 1 to M) of DRRA.

- VESYLA uses linear addressing mode and each RFILE is going to operate on exactly N addresses instead of N/M addresses. This is achieved by using “%$no_distribute” pragma as shown in Figure 4.4-c. Since user has not mentioned the physical address space, therefore it is up to VESYLA to select the appropriate address space for mapping variable z on RFILEs.

**RFILE port allocation Pragma**

This is an optional pragma and it specifies which port to be used from the dual port RFILE. For example if the designer wants to use the statement 6 in Figure 4.4-c is directing VESYLA to use port B of RFILE. The designer can also leave the assignment of the ports to VESYLA by not using this pragma. In such scenario, VESYLA does the allocation accordingly and send the information (about the allocation of the port) back to the user.
4.3.5 DPU allocation & Binding Pragmas

As the name suggests, these pragmas are used for binding/allocating of variables and resources related to DPUs [12].

The general syntax for using these pragmas is:

```plaintext
%$ VariableName DPU[Row indexes, Column indexes] or %$ DPU[Row indexes, Column indexes]
```

The keyword DPU is followed by the respective row(s) & column(s) indexes which indicate the respective DRRA cells to be allocated for the DPU. The usage is similar to the RFILE allocation and binding pragmas.

4.3.6 VESYLA Functional statements & Pragmas

Functional statements are core part of the algorithm. These statements define the actual working of algorithm. These statements include operation over vectors, loops and conditional statements. There can only be a single statement or combination of various statements. VESYLA functional statements should not be mixed with MATLAB functions (e.g., sum, zeros, ones, etc.). Any statement which involves reading/writing from RFILE or can invoke a particular mode of DRRA DPU is considered as VESYLA functional statement. This section explains the usage of pragmas with simple and complete examples. Consider the MATLAB code presented in Figure 4.5.

```
1. N = 50;
2. M = 5;
3. row = 1;
4. column_range = (1:M); %$Range
5. address_range = (1:N); %$Range
6. x(address_range) = 1 * ones(1, N); %$RFILE[row, column_range]_A
7. y(address_range) = 1 * ones(1, N); %$RFILE[row, column_range]_B
8. res = 0; %$RFILE[row=1, 1]
9. %$mac_opr = Dpu[row, column_range]
10. res = sum(x .* y); %$mac_opr
```

**Figure 4.5.** An Example of VESYLA functional statement pragmas
The code is used to implement a MAC operation on DRRA. Figure 4.5 also shows the state of sequencers (Five sequencers are used since N=5). For demonstration purpose we have used only 5 threads but practically there can be several threads working in parallel. It was not possible to write all the instructions in each and every sequencer but it gives a good idea to the reader that how sequencers store the instructions. VESYLA generates all these instructions automatically after examining the MATLAB code and one can imagine the difficulty of managing the sequencers by manually writing the code at lower level of abstraction. This MAC operation can be implemented as a fully serial solution, fully parallel solution or anything in between by varying the value of N and M. For example, Figure 4.6 shows a partial parallel solution with N=50 & M=5. All timing related issues are resolved automatically by VESYLA. From VESYLA’s perspective, the statements can be categorized as follows:

- **Statements 1 – 5 are Range statements**
  - N (= 50) is the size of vectors.
  - M (= 5) is the number of threads/resources.
  - column_range (1 : M) is defining number & positions of the columns to be used.
  - address_range (1 : N) is used to operate on vector of size N
• Statements 6 – 8 are RFILE Allocation Pragmas

- x is a variable which operates on 50 vectors (address_range = 50) and is mapped to RFILE pointed by row [1] & column_range[1:5]. The column_range[1:5] also implies that x is mapped to multiple RFILEs. “A” specifies that variable x is going to use port A of RFILEs as shown in Figure 4.6.

- y is a variable which operates on 6 vectors (address_range = 50) and is mapped to RFILE pointed by row [1] & column_range[1:5]. This implies that y is mapped to multiple RFILEs. “B” specifies that variable y is going to use port B of RFILEs.

- res is a variable which will be used to store the results of the MAC operation. It is only operating on a single vector and is mapped to RFILE[1, 1] as shown in Figure 4.6. The “port pragma” is not mentioned, which implies that VESYLA is going to take care of the write-back port.

• Statement 9 is for DPU allocation.

- This statement defines a variable mac_opr which is mapped to DPUs pointed by row [1] & column_range[1:5].

• Statement 10 is a functional statement.

- The mac operation takes x & y as input and the results are stored back in res. The “%$ mac_opr” specifies that MAC operation is going to be performed on DPU defined in statement 9. VESYLA translates this statement according to the architecture shown in Figure 4.6. The computation is distributed on multiple SIMD threads. “This approach (of using pragma) helps VESYLA to exploit parallelism from the sequential MATLAB code” [12].

One can generate various architectures by changing the values of N & M. VESYLA can generate a fully serial solution from the same MATLAB Code as shown in Figure 4.7 by using M = 1.

---

**Figure 4.7.** Fully Serial Solution for MAC operation after changing the pragmas
4.3.7 Adder Tree Pragma

The example presented in the previous section used a long carry chain for generating the final sum obtained from multiplication operations. A different architecture can be generated by using the “adder_tree” pragma by making a small change in statement 10 of the code used in previous example. The rest of the code remains unchanged. This pragma forces VESYLA to use an adder_tree chain (The positions of DPUs in the fabric is inferred by VESYLA) for generating the final sum as shown in Figure 4.8. In a scenario, where the value of N = ’1’, VESYLA ignores the adder_tree statement as there is no need for adder_tree in fully serial implementation. VESYLA infers the allocation and binding for adder_tree chain.

![Figure 4.8. Usage of “addertree” pragma](image)

The question arises that how significant is the introduction of this pragma? At higher level of abstraction it is equivalent to modifying a single line, but this small change (at higher level of abstraction) has major impact. There are fewer instructions after introducing adder_tree pragma and it leads to 35% modification in the code (as compared to the implementation which uses long chain of adders) [12]. The changes includes configuration of switch-boxes, modifying the delays of RFILEs and other low level tasks which are very tedious and cumbersome to perform. Benefit of using VESYLA is obvious as keeping track of all these changes and doing them manually is a very hard, time-consuming, error-prone task as compared to a single change at higher level of abstraction and getting the same results. This was a simple example and making changes manually was a very difficult task. The
complexity increases with more complex algorithms and with variation in parallelism. Programming at lower level of abstraction is nearly impossible for mapping complex algorithms on DRRA without using VESYLA.

4.3.8 DPU Saturation Pragma

“The DPU are 16-bit integer units. The multiplication of two 16 bits number would result in a 32-bit integer and this result must be saturated back to 16-bit. The saturation pragma is used for selecting the desired 16 bits. LSBs (least significant bits) are selected by default in the absence of this pragma” [12].

The general syntax for using this pragmas is:

```plaintext
$\%$ select saturation (MSB - LSB)
```

The select saturation is a VESYLA keyword. The MSB and LSB are identifying the range of the 16 bits to be selected from the 32 bits.

4.4 Synchronization in FSMs using VESYLA

As we have already discussed in chapter 3 that DRRA is capable of executing multiple DSP sub-systems like modems & codecs. “The DRRA fabric can be dynamically partitioned to accommodate these DSP sub-systems and these partitions can be further sub-partitioned to host individual DSP algorithms in that particular sub-system. DRRA in general adopts a distributed control model [8] when implementing algorithms” [35].

“Distributed control logic requires synchronization amongst the individual controllers” [35]. DSP algorithms in DRRA are implemented in varying degrees of parallelism in terms of multiple SIMD threads and each SIMD thread has it’s own RFILE(s), DPU(s) and Sequencer(s) [35]. These SIMD threads usually interact with each other for correctly executing the algorithm. On top of that, each RFILE is capable of streaming data with programmable delays and can be viewed as a programmable FSM.

This requires VESYLA to perform synchronization at multiple levels. First, it needs to make sure that programmable FSMs within each thread are synchronized properly and the second is too make sure that these parallel SIMD thread are synchronized correctly as shown in Figure 4.9. There are set of interacting FSMs (F) and the executions of these FSMs are dependent upon the set of dependencies (D). VESYLA takes both these sets and generates synchronized interactive set of FSM (F’) after resolving the dependencies [35].

The situation complicates further as multiple variants exits for mapping a single instruction on DRRA. The operands and operators can be mapped to any DRRA cell as long as they follow the 3 - hops communication mechanism. The number of (low level) instructions and execution cycles may differ for different mapping
choices. The Sequencer executes the instructions sequentially and VESYLA needs to synchronize these sequential instructions for correct execution of algorithm.

Consider the one line code presented in Figure 4.10 which implements an add operation on DRRA with three different implementation choices (shown in red, green and yellow) [35]. The first choice (in green) uses the same DRRA cell for mapping the operators and operands while the other two cases use more than one DRRA cells. These different choices lead to different state of sequencers as shown in the Figure 4.10. The execution of the algorithm is based on the following scenario:

- X₀, X₁ and MAC operation should start executing at the same cycle.
- Write-back operation Y₀ should start executing 4 cycles after the MAC operation.

Sequencer executes the instructions sequentially and all these instruction needs to be synchronized properly by VESYLA. This synchronization is achieved by using techniques described in chapter 3 section 3.1.2. VESYLA back-end performs all these calculations and assign these delays to their respective instructions in Sequencers. The scenario presented in this example is very simple and the complexity increases with more complex algorithms which makes it practically impossible for the developer to perform these tasks manually. VESYLA makes the job easier for the developer by letting it to change few parameters at higher level of abstraction and the back-end of the tool takes care of the error prone, cumbersome and complex tasks. The quality of VESYLA generated code matches the quality of hand written (manual fully optimized) code with much ease and in less amount of time.
DRRA provides multiple ways of achieving synchronization:

1. The synchronization can be achieved by using the delays provided by the AGU of RFILE [35, 8, 30]. These delays (initial delay, middle delay, repetition delay) have already been discussed in chapter 2.

2. Delay alone are not always enough to achieve synchronization. One needs to stall cycles in order to achieve synchronization and for this purpose the wait cycle command can be used. The wait cycles insert ’n’ number of wait cycles before moving onto the next instruction.

3. Dynamic delays are used to synchronize the delays in the branches of IF-ELSE condition. These are extension to the existing AGU delays and helps in setting the delays dynamically while considering which part of the branch is taken.

4. Another way of achieving synchronization is hierarchical control. This mechanism allows a single sequencer to control its neighboring sequencers.

At the moment VESYLA is only capable of achieving synchronization using first two techniques and the rest is part of the future work. In the next section, we will briefly describe the synchronization mechanism within a basic execution block using AGUs delays with few examples.
The following are some terms which are used while explaining the synchronization algorithm:

- **ExCyR = initial delay + instruction number + writeback delay**
  - ExCyR represents the first cycle at which the read operation will start executing and the read output is available at the respective port of the RFILE.
  - The calculation (in context of CP_Blocks) depends upon instruction index of RFILE instruction in the Sequencer, initial delay (by default it is 0) and write-back delay (in case of read after write dependency).

- **CmCY = ExCy + operator_delay**
  - CmCy represents the cycle at which the output is available (after computation) at the output ports of DPU.
  - This calculation is based (in context of CP_Blocks) on the delay of the operator along with their respective ExCy.

- **ExCyW = initial delay + instruction number + readback delay**
  - ExCyW represents the first cycle at which write operation will start executing and the input is available at the respective write port of the RFILE.
  - The calculation (in context of CP_Blocks) depends upon instruction index of RFILE instruction in the Sequencer, initial delay (by default it is 0) and read-back delay (in case of write after read dependency).

- **Step = PreviousStep + operationDelay + write-back/read-back delay**
  - It is the clock step in which the operation executes.
  - The calculation (in context of CP_blocks) depends upon initial delay and write-back or read-back delay. Step is a relative term in the context of whole datapath and the value of step also depends upon the previous values.

### 4.4.1 Synchronization using initial delays

The algorithm for synchronizing the initial delays is presented in Figure 4.12 and is explained through an example in this section. Consider the datapath generated by VESYLA (from a MATLAB code) shown in Figure 4.11. VESYLA traverses the CDFG graph and identifies resource or data dependencies. It also divides the graph in atomic blocks within the basic blocks which we call as CP_blocks. Two
4.4. SYNCHRONIZATION IN FSMS USING VESYLA

CP_blocks exist in this example and there is no resource dependency among the operators or operands. Read after Write dependency exists between CP_Block_A and CP-Block_B because of operand Y0. Figure 4.11 also shows the state of Sequencers for this datapath. Multiple threads exist but only a few are shown for the sake of simplicity. These instructions are not synchronized with each other because the (default) initial delay is zero for all read/write operations. If the instructions are executed in this initial state then Read X0 will execute before Read X1 and similarly Write Y2 is going to execute before the output from the multiply operation is available. It happens because Sequencer executes these instructions in sequential order with a default delay of 1 cycle between each instruction (unless it is overridden by AGU delays or there is a jump/wait instruction in between). All these operations need to be synchronized in order to get the correct results after executing this MATLAB code [35].

Synchronization in CP_Block_A

First step in the process is to synchronize all the read operations (Read X0 & Read X1) involved in the CP_block_A. It starts with calculating the ExCyR for both read operations as shown in Figure 4.12 (CP_block_A-1). In order to synchronize this algorithm, the values of X0_ExCyR and X1_ExCyR must be equal to each other. The algorithm calculates the difference between these two (as shown in Figure 4.12 (CP_Block_A-2) and assign this difference as initial delay to read

---

**Figure 4.11.** Example of datapath and Sequencers state for calculating Initial Delays
operation $X_0$ (as shown in 4.12 (CP_Block_A-3)). The values of $X_0\_ExCyR$ and $X_1\_ExCyR$ are updates based on this new initial delay (as shown in 4.12 (CP_Block_A-5)). It can be observed that the values are equal now which indicates that both read operations are synchronized with each other.

Next step in the algorithm is to calculate the execution cycles ($CmCy$) required for the multiplication operation. This calculation is based on $ExCyR$ and the delay of the operator.

Lastly, the algorithm needs to synchronize the write-back operation with this whole process because (as in this current state) the write back operation will start executing even before the read operations are executed. This process starts with calculating the value of $ExCyW$ for write operation $Y_0$ (as shown in 4.12(CP_Block_A-7)) and the value of $Y_0\_ExCyW$ is compared with the value of $Mul\_CmCy$ (as shown in Figure 4.12 (CP_Block_A-8)). The difference in the values is allocated to the initial delay of Write operation of $Y_0$ (as shown in Figure 4.12(CP_Block_A-8)) and the value of $Y_0\_ExCyW$ is updated accordingly (as shown in Figure 4.12 (CP-Block_A-8)). After going through all these steps the initial delays of the read operations ($X_0 = 1$, $X_1 = 0$, and $Y_0 = 3$) are updated in the Sequencers:
### 4.4. Synchronization in FSMS Using Vesyla

#### Figure 4.13. Algorithm for computing initial delay

<table>
<thead>
<tr>
<th>Step</th>
<th>Formula</th>
<th>Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( X_0_ECYR = 2 )</td>
<td>( X_0_InitialDelay )</td>
</tr>
<tr>
<td>2</td>
<td>( X_0_ECYR = 2 + 0 )</td>
<td>( X_0_InitialDelay + Diff_ECYR )</td>
</tr>
<tr>
<td>3</td>
<td>( X_0_InitialDelay = 0 + 1 )</td>
<td>( X_0_InitialDelay )</td>
</tr>
<tr>
<td>4</td>
<td>( X_0_ECYR = 4 )</td>
<td>( X_0_InitialDelay )</td>
</tr>
<tr>
<td>5</td>
<td>( X_0_ECYR = 4 + 6 )</td>
<td>( X_0_InitialDelay + Diff_ECYR )</td>
</tr>
<tr>
<td>6</td>
<td>( X_0_InitialDelay = 0 + 9 )</td>
<td>( X_0_InitialDelay )</td>
</tr>
<tr>
<td>7</td>
<td>( X_0_ECYR = 10 )</td>
<td>( X_0_InitialDelay )</td>
</tr>
<tr>
<td>8</td>
<td>( X_0_InitialDelay = 1 )</td>
<td>( X_0_InitialDelay )</td>
</tr>
<tr>
<td>9</td>
<td>( X_0_ECYR = 10 + 0 + 0 )</td>
<td>( X_0_InitialDelay + Diff_ECYR )</td>
</tr>
</tbody>
</table>

#### Calculate Initial Delay Using for Read

<table>
<thead>
<tr>
<th>Step</th>
<th>Formula</th>
<th>Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( X_0_ECYR = 2 )</td>
<td>( X_0_InitialDelay )</td>
</tr>
<tr>
<td>2</td>
<td>( X_0_ECYR = 2 + 0 )</td>
<td>( X_0_InitialDelay + Diff_ECYR )</td>
</tr>
<tr>
<td>3</td>
<td>( X_0_InitialDelay = 0 + 1 )</td>
<td>( X_0_InitialDelay )</td>
</tr>
<tr>
<td>4</td>
<td>( X_0_ECYR = 4 )</td>
<td>( X_0_InitialDelay )</td>
</tr>
<tr>
<td>5</td>
<td>( X_0_ECYR = 4 + 6 )</td>
<td>( X_0_InitialDelay + Diff_ECYR )</td>
</tr>
<tr>
<td>6</td>
<td>( X_0_InitialDelay = 0 + 9 )</td>
<td>( X_0_InitialDelay )</td>
</tr>
<tr>
<td>7</td>
<td>( X_0_ECYR = 10 )</td>
<td>( X_0_InitialDelay )</td>
</tr>
<tr>
<td>8</td>
<td>( X_0_InitialDelay = 1 )</td>
<td>( X_0_InitialDelay )</td>
</tr>
<tr>
<td>9</td>
<td>( X_0_ECYR = 10 + 0 + 0 )</td>
<td>( X_0_InitialDelay + Diff_ECYR )</td>
</tr>
</tbody>
</table>

#### Mul. Delay

<table>
<thead>
<tr>
<th>Step</th>
<th>Formula</th>
<th>Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( X_0_ECYR = 2 )</td>
<td>( X_0_InitialDelay )</td>
</tr>
<tr>
<td>2</td>
<td>( X_0_ECYR = 2 + 0 )</td>
<td>( X_0_InitialDelay + Diff_ECYR )</td>
</tr>
<tr>
<td>3</td>
<td>( X_0_InitialDelay = 0 + 1 )</td>
<td>( X_0_InitialDelay )</td>
</tr>
<tr>
<td>4</td>
<td>( X_0_ECYR = 4 )</td>
<td>( X_0_InitialDelay )</td>
</tr>
<tr>
<td>5</td>
<td>( X_0_ECYR = 4 + 6 )</td>
<td>( X_0_InitialDelay + Diff_ECYR )</td>
</tr>
<tr>
<td>6</td>
<td>( X_0_InitialDelay = 0 + 9 )</td>
<td>( X_0_InitialDelay )</td>
</tr>
<tr>
<td>7</td>
<td>( X_0_ECYR = 10 )</td>
<td>( X_0_InitialDelay )</td>
</tr>
<tr>
<td>8</td>
<td>( X_0_InitialDelay = 1 )</td>
<td>( X_0_InitialDelay )</td>
</tr>
<tr>
<td>9</td>
<td>( X_0_ECYR = 10 + 0 + 0 )</td>
<td>( X_0_InitialDelay + Diff_ECYR )</td>
</tr>
</tbody>
</table>
Synchronization in CP_Block_B

After assigning these values all the operations in CP_Block_A are synchronized with each other and the algorithm continues with synchronizing the operations in CP_Block_B which has a read after write dependency due to variable Y0.

The process starts with synchronizing both read operations (Y0 & X2) by the respective ExCyR (as shown in Figure 4.13 (CP_Block_B-1). The only difference (when compared to computation performed in CP-Block-1 shown in 4.12 (CP_Block_A-1)) is that it needs to consider the write back delay of Y0 (because of read after write dependency) when computing ExCyR. The rest of the steps 2 - 9 are the same as described previously in this section.

4.4.2 Synchronization using middle delays

The initial delays are used to delay the execution of a FSM. This type of delay can only be used once and helps in synchronizing the FSMs when they are starting the process of address execution, but it would often be the case to synchronize the FSMs at different stages while they have already started the execution.

Algorithm For Calculating Middle Delay

For N number of Threads
For Each CP-block in a thread
For Each Read Node
  → Calculate StepR (for each Read FSM)
  → Compute difference between StepR
  → Compute difference between ExCyR
End For
→ Update StepR for each read node
  (w.r.t. Highest StepR & WriteBack Delays)
→ if (Marked_Rd == No) then
  → set Initial delays and Update ExCyR for read Nodes
→ Else
  → set Middle Delay w.r.t. StepR
→ Calculate CmCY delay and Step(CmCy) for operators

For Each Write Node
→ Calculate StepW (for each write FSM)
→ Calculate ExCyW
→ If(CmCY > ExCyW)
  → if (Marked_Wr == No) then
    → set Initial delays and Update ExCyW for write Nodes
  → Else
    → set Middle Delay w.r.t. StepW
Else
  GoBack to respective Nodes and Adjust Initial delays accordingly
  (repeat the process)

Figure 4.14. Data path and the algorithm for using middle delay
This happens because interacting FSMs might operating on the same/different resource with variable data consumption/production rates [8, 30, 35]. The algorithm for synchronizing the middle delays is presented in Figure 4.14 and is explained through an example in this section.

**Synchronization in CP_Block_A**

Consider the datapath showed in Figure 4.14. This example is a modified version of the example presented in section 4.4.1. The datapaths (in both examples) are very similar and the only difference is that the variable X1 is now used as a Read operation in both blocks. No other dependency is introduced apart from this slight modification. The synchronization steps for CP_Block_A are exactly the same (with results) as described in 4.4.1 and Figure 4.12.

**Synchronization in CP_Block_B**

The process starts with synchronizing the Read operation X0 with Read operation Y0 after calculating the respective ExCyR. The value of ExCyR for Y0 is 10 and the value of ExCyR for X0 is already computed CP_Block_A. This scenario presents a problem because ExCyR of Y0 is greater than ExCyR of X0. If the value of ExCyR of X0 is changed to synchronized with ExCyR of Y0 then X0 is no longer synchronize with X1 in CP_Block_A.

![Figure 4.15. Middle delay waveform](image)
This implies that $X_0$ can either be synchronized with $Y_0$ in CP_Block_B or $X_1$ in CP_Block_A. The problem is overcome by using the concept of middle delay. Read operation $X_0$ is initially synchronized with Read operation $X_1$ in CP_Block_A and then it halts its execution by using a middle delay until Read $Y_0$ starts its execution. This procedure makes sure that Read $X_0$ is synchronized with both Read $X_1$ and Read $Y_0$. The process keeps track of the resources being read or write multiple times in each block (using “marked_rd” or “marked_wr” variable in algorithm shown in Figure 4.14) and in case if it is read or write multiple times then the difference is not allocated as initial delay rather it is allocated as middle delay [35] as shown in Figure 4.15.

Rest of the steps are same as described in section 4.4.1 and the calculations are shown in Figure 4.12

4.4.3 Synchronization using repetition delays

Repetition delays are used to delay the repetition of RFILE instructions. This delay is used to re-synchronize the FSMs if a read/write operation repeats its execution. There are three different synchronizing scenarios that need repetition delay and algorithms for synchronizing the repetition delays are presented and explained through examples in this section.

The first scenario is related to the case when FSMs are operating at different rate. Consider the datapath shown in Figure 4.16 for computing MAC operation that is extracted by VESYLA from the MATLAB code. Two read operations are operating in a loop on vector with different sizes. Due to operating on vector with different sizes, the Read FSM $Y_0$ will complete its execution before Read FSM $X_0$. These operations needs to be re-synchronized again before they start repeating their execution because if they are not re-synchronized then Read $Y_0$ will start its execution before Read $X_0$. This implies that execution of Read $Y_0$ needs to be delayed before the start of next loop iteration and this is achieved by using the repetition delay. The process starts by computing $CmCyR$ for Read $Y_0$ and Read...
X₀ and the difference between the values is assigned to the Read FSM Y₀ as the value of the repetition delay which forces Read Y₀ to delay its re-execution by 2 cycles at the next iteration of loop.

The second scenario is similar to the first one but is more appropriate for synchronizing the parallel threads. Consider the datapath shown in Figure 4.17 which is generated by distributing the variables over multiple RFILEs (49 elements of the vectors distributed over 4 RFILEs). Two read operations are operating in each thread and the last thread is operating on less vector elements as compared to the previous threads. Due to operating on different number of elements, the last thread will complete its execution before other threads but all these threads needs to be re-synchronized with each other before repeating their execution for the next iteration of loop. This implies that execution of Read operations in the last thread needs to be delayed before the start of next loop iteration. The process starts by computing CmCyR for all read operations in all the threads. The difference in the values are assigned to the read operation in the last thread as the value of repetition delay.

The third scenario is related to deal with the dependencies among the variables within multiple loops. For example, consider a variable (dealing with vectors) is written in the outer loop but is read inside the inner loop. The variable in the outer loop needs to wait for the completion of inner loop before iterating again and the repetition delay of the variable in the outer loop is adjusted w.r.t. the completion of the inner loop.

![Figure 4.17. Repetition Delay Example 2](image-url)
All these scenarios related to adjusting AGU delays can be as simple as adjusting delay in one operation or as complex as adjusting all these delays for multiple atomic blocks. For example, while executing a DSP algorithm, VESYLA may need to deal with adjusting all these delays and dependencies including loops. VESYLA can cope with most of the cases but still has few limitations. For example, VESYLA is not yet capable of dealing with the delays related to IF-ELSE statements. It is part of the future work to strengthen the VESYLA back-end for dealing with all kind of scenarios and make it more generic.
Chapter 5

Mapping MATLAB codes using VESYLA

In this chapter the usage of pragmas (discussed in chapter 4) is demonstrated to implement DSP algorithms. The chapter is divided into two main sections. The first section discusses mapping DSP algorithms, written in MATLAB, on DRRA using VESYLA and the second section describes the methodology for estimating the energy consumption of the DSP algorithm.

5.1 Mapping DSP algorithms on DRRA

The section describes mapping FIR Filter, Taylor Series, Matrix Operations and FFT on DRRA using VESYLA.

5.1.1 FIR Filter

FIR filters are the core part of many signal processing applications. The section demonstrates mapping of Asymmetric FIR filter using VESYLA. This code shown in Figure 5.1 shows the MATLAB code of a generic N-tap Asymmetric FIR filter. The code can be mapped as a fully serial solution, fully parallel solution and anything in between after changing the values of pragmas accordingly. The Asymmetric FIR filter is represented by the following equation where X₀ to Xₙ represents the samples and C₀ to Cₙ represent the coefficients:

\[ Sum = \sum_{i=0}^{n} CiXi \]  

(5.1)

\[ Sum = C_0X_0 + C_1X_1 + C_2X_2 + \ldots + C_nX_n \]  

(5.2)

Description of the code is as follows:
Variables affecting the number of tap & parallelism

- Variable N (statement 1) is used to change the number of tap in the filter. For example, Figure 5.1 shows a fully serial 47-tap FIR filter by assigning value 47 to variable N. Similarly Figure 5.2 shows a 5-tap fully parallel FIR filter. One can change the value of N to implement a smaller or larger FIR filter.

- Variable M (statement 2) is used to control the parallelism. For example, Figure 5.1 shows a fully serial 47-tap FIR filter by assigning value 1 to variable M. It uses a single RFILE to store the samples and coefficients and a single DPU is used to obtain the final sum. Similarly Figure 5.2 shows a 5-tap fully parallel FIR filter and Figure 5.3 shows a 47-tap partially parallel FIR filter. One can change the value of M to vary the parallelism.
  - If M = 1 then the fully serial solution is implemented.
  - If M = N then the fully parallel solution is implemented.
  - If N > M then the partially parallel solution is implemented.

- Variable column_range is used to identify the number of columns which are going to be used for mapping the samples and coefficients. In this example, the value of M is '1' therefore column_range varies from [1 : 1]. This implies that only a single column is used.

- Variable address_range is used to identify the number of addresses which are going to be used in a RFILE. In this example the value of N is '47' therefore the value of address_range is (1 : 47).

Variables related to delay line, samples and coefficients

- Variable “x” (defined in statement 6) is used to map samples and delay line. If M = 6 then “x” is mapped to RFILE pointed by row [1] and columns [1, 2, 3, 4, 5, 6]. The ‘_A’ is directing VESYLA to use port A of the RFILEs.

- Variable “c” (defined in statement 7) is used to map coefficients on RFILE pointed by row [1] and column_range. The coefficients are sharing the same RFILE with samples “x” but using a different port (samples are using port A while coefficients are using port B). This enables coefficients and samples to execute in parallel.

- New samples are inserted through variable “new_sample”. Ideally, these new samples should come from memory and stored back into memory but memory transactions are not yet implemented in VESYLA, therefore the new samples are stored in another RFILE.
5.1. MAPPING DSP ALGORITHMS ON DRRA

Functional Section

The outer loop is used for the total number of iterations. This outer loop also helps in determining the value of the “repetition delay” field along with the “number of repetitions” in the RFILE Instructions (as shown in Figure 3.2 - extended instruction set).

The inner loop iterates over variable M (which is used to control parallelism) and divides the computation over M SIMD threads. These threads work in parallel with each other to implement the functionality of the FIR filter.

The IF-ELSE statement inside the loop is used to provide the new sample at the start of each computation. In (statement 14), 1 is representing the vector x[1] (i.e. the first sample) and j is representing the column number for the RFILE where x is mapped. This statement is only executed once (after the arrival of a new sample).

MAC operation is implemented in statement 16. There can be multiple MAC operations working in parallel (depending upon the value of M). The ‘:’ is representing the whole vector x in RFILE presented at column number ‘j’. Each DPU [mac_out(j)] is configured in mode ‘0’ and this is automatically inferred by VESYLA in each thread. For example, Figure 5.1 shows a single thread in which the

![Figure 5.1. Asymmetric FIR Filter](image-url)

```matlab
1. N = 47;
2. M = 1;
3. Par = N/M; row = 1;
4. column_range = (1:M); %$Range
5. address_range = (1:N); %$ Range
6. x(address_range) = zeros(Par,M); %$RFILE[row, column_range] A
x = reshape(x(1:Par*M), Par, M);
7. load('c.mat', 'c'); %$RFILE[row, column_range] B; address_range
   c = reshape(c(1:Par*M), Par, M);
8. load('samples.mat', 'new_sample'); %$RFILE[row+1, 1];
   address_range
9. Res = 0; %$RFILE[row+1, M+1]
10. mac_out = zeros(M, 1); %$DPU[row, column_range]
11. for i = 1 : N
12.     for j = 1 : M
13.         if j == 1
14.             x(1, j) = new_sample(i);
15.         end
16.         mac_out(i) = sum(x(:, j).*c(:, j));
17.     end
18.     end
19.     res = sum(mac_out); %$adder_tree
20. end
```
coefficients and samples are multiplied together and are internally accumulated by
the DPU for computing the MAC operation.

\textbf{Delay line} is implemented by the help of statement 18. Vector (2 : P) is
replaced by vector (1 : P) and the implementation of this statement differs with
the degree of parallelism. For example in case of fully serial implementation the
delay line is implemented within a single RFILE as shown in Figure 5.1 but in
case of multiple threads the samples are mapped to multiple RFILEs (as shown
in Figure 5.2) where delay line is mapped in both horizontal (RFILE within each
thread) and vertical (RFILE in column ‘j’ communicating with RFILE in column
‘j+1’) directions. Each implementation needs to execute the correct addressing
mode of the AGU, making sure that input(s) are received at correct clock edges
and the synchronization is perfect (not only between each thread but also between
different threads). This scenario is discussed at the beginning of section 4.4 and
is elaborated in Figure 4.9. Statements 16 & 17 are working in parallel with each
other within the same thread and then there are multiple threads like this which
are working concurrently. This requires VESYLA to perform synchronization at
multiple levels.

\textbf{Adder Tree} (in statement 19) is used to obtain the final convolution sum for
multiple threads.
Figure 5.3. n-tap Asymmetric FIR Filter (Partial Parallel Solution)

Figure 5.4 shows the impact of making changes in the pragmas for generating different solutions. Moving horizontally increases parallelism (i.e. from fully serial to fully parallel and other solutions in between) and moving vertically changes the dimension of FIR Filter (i.e. number of tap). The arrows (orange, purple) between the gray boxes represent the changes required for moving in different dimensions. For example, the machine level code is changed by 90% and 45 more instructions are required for parallelizing 33-tap filter from a fully serial to partially parallel solution. The red boxes are representing the relative complexity for synchronizing the FSMs for implementing the algorithm. The complexity also increases by changing the dimension of FIR Filter or varying in the degree of parallelism. Making all these changes at a lower level is a very difficult task. Another issue which make things extremely difficult to handle at the low level is the dimension of the problem. VESYLA deals with multiple implementations (and these implementations can reshape as shown in Figure 2.1) instead of dealing with single solution and the designer only needs to alter the values of different parameters (like N and M). “VESYLA takes care of the low level details of synchronizing the FSMs, calculating the delays, configuring the switch box instructions, configuring RFILE and DPUs
in their respective modes and generating related instructions. It is virtually impossible to generate handwritten codes (or using low level assembler) for multiple solutions while taking into consideration all these issues” [12].

5.1.2 Taylor Series

Taylor series is useful to represent any real or imaginary number in terms of its 'n' derivatives and can be used to calculate trigonometric, logarithmic, hyperbolic and exponential functions.

\[ e(x) = 1 + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots \] (5.3)

The MATLAB code presented in Figure 5.5 can map this equation on DRRA. The solution presented here is not restricted to the number of terms and can map 'n' number of terms on DRRA. It can be observed that there is a natural dependency in this algorithm which restricts parallelizing the multipliers. The dependency comes in terms of obtaining the higher orders of sample 'x'. Description of the code is as follows:

Declaration variables
5.1. MAPPING DSP ALGORITHMS ON DRRA

- Variable N (statement 1) is used to control the number of terms for e(x). The value of ’N’ is 8 in this example.

- Variable “x” (statement 4) is mapped to RFILE[1,1]. The address range used is (0 : N). The first location (x[0]) stores the value of “x” while the rest are used to save the higher orders of “x” obtained after multiplications.

- Variable fact_x (statement 5) is mapped to RFILE[1,2] and uses address locations pointed by address_range. This variable contains the pre-stored factorial values.

- const_1 (statement 6) contains the constant value.

![Algorithm Diagram](image)

**Figure 5.5.** e(x) Taylor Series computation Example

**Functional statements**

The **loop** iterates over ’N’. Statement 11 uses DPU[1,1] for obtaining the higher order values of sample “x” as shown in Figure 5.5. The results obtained by this multiplication operation are stored at address i+1 of RFILE[1,1]. VESYLA configures DPU[1,1] in multiplication mode for performing this multiplication operation. The MAC operation (statement 12) is mapped to DPU[1,2] and works in parallel with the first multiplication operation. This multiplication operation is used to calculate the terms. The results are then fed to an adder for accumulating all the terms obtained during this process. The DPU[1,1] is therefore configured in mode '2' (as per explained in Figure 3.1) by VESYLA for obtaining these results.
This section demonstrates mapping of a sin(x) Taylor series (MATLAB code) on DRRA using VESYLA. The sin(x) function can be represented as follows:

\[
\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots
\]  

(5.4)

Figure 5.6. Sin(x) Taylor Series Partially Parallel Example

Two methods are described for mapping sin(x) Taylor series on DRRA. The method described in this section is a scalable solution and it is not limited to using the number of terms in Taylor series. It can be observed that there is a natural dependency in this algorithm which restricts parallelism. The dependency comes in terms of multiplication (for obtaining the polynomial terms). The term x5 can be obtained from using x3 and similarly x7 can be obtained by using x5 and this pattern can continue for rest of the terms. There is not much benefit in terms of parallelizing the multiplication operation but one can parallelize the adders/subtractors for accumulating the results obtained from the multiplication of the terms by the factorials. VESYLA back-end needs some additional incremental works for implementing the solutions presented for sin(x).
5.1. MAPPING DSP ALGORITHMS ON DRRA

Figure 5.6 shows a MATLAB code with a corresponding datapath for computing the sin(x) Taylor series. This is a generic code which can compute sin(x) Taylor series up to ‘n’ terms. The deceleration section (statement 1 - statement 14) defines the variables with allocation and binding pragmas. The functional section (statement 15 - statement 29) defines the functionality of this algorithm. The code can easily be viewed as two sections where the first section is performing the multiplication operations and the next section is accumulating these multiplication operations by parallelizing the adder operations. Description of the code is as follows:

Variables affecting number of terms and parallelism

- Variable N (statement 1) is used to change the number of terms in the Taylor series. In this example the Taylor series is expanded to 7 terms but one can change the value of N to expand the Taylor series as desired.
- Variable M (statement 1) is used to control the parallelism for adders/subtractors.

Variables related to terms, samples, factorials

Mapping of allocation and binding pragmas are thoroughly discussed in previous sections so here we are not going discuss them in detail.

- “x” stores the sample value and fact_x stores the factorials (which are constant values and are stored in the RFILEs before the execution of MATLAB/VESYLA code).
- Term is used to store the output obtained after multiplying the sample “x” with the factorials “fact_x”.
- SinX stores the final sum for Taylor series.

Functional statements

The functional statements are (conceptually) divided into two sections (or two loops). The computation starts by calculating the value of x2 (in statement 15). This value is computed outside the first loop and is stored at Nth address position in RFILE[1,1]. This is used as a base value for obtaining x3, x5, x7 . . . and so on. The first loop do the following:

1. It computes the polynomial terms (x3, x5, x7 . . .) using statement 17. A single RFILE[1,1] and DPU[1,1] is used for this purpose. The loop index (x(i+1) in statement 17) is used as an address index which implies that the polynomial terms are stored at consecutive addresses of RFILE[1,1].

2. It also computes the complete terms by multiplying the polynomial terms with their respective factorials (x3/3!, X5/5! . . .) using statement 18. This computation is performed over RFILE[1,2] and DPU[1,2]. The result of this computation is represented by variable Term (pointing towards RFILE[1,3 - 1,4]). Parallelism is also introduced at this stage (for storing the value of
terms). The results are stored in multiple RFILEs. The variable j in the loop is pointing towards the location of RFILE while variable index is pointing towards the exact address location to be used in that RFILE. The IF-ELSE statement is not mapped to DRRA. This IF-ELSE statement helps VESYLA to distribute the results obtained from the multiplication operation to store in multiple RFILEs (used by variable Terms). These RFILEs can then provide data in parallel to the adders/subtractors for calculating the final sum of Taylor series. In this example, each RFILE contains 4 terms (because N=8, M=2, PAR=4)

```
1. N = M = 4;
2. Par = ceil(N/M); row = 1;
3. column_range = (1:M); %$Range
column = 1; %$Range
4. adr_range = (1:N); %$ Range
5. load('x.mat', 'x');
%$RFILE[row, column]; adr_range
load('fact_x.mat', 'fact_x');
%$RFILE[row, column+1]; adr_range
load('term.mat', 'Term');
%$RFILE[row+1, column_range]
6. Term = reshape(Terms, PAR, M)
sinX0; %$RFILE[row+1,M+1]; adr_range
7. %$Op_1 = DPU[row, column]
%$Op_2 = DPU[row, column+1]
8. Diff = zeros(M, 1);
%$DPU[row, column_range]
9. x(N) = x(1) * x(1) %$Op_1
10. for i = 1 : N
11.   x(i+1) = x(i) * x(N) %$Op_1
12.   Term(i, 1) = x(i) * Fact(i) %$Op_2
13. end
14. end
15. for i = 1 : 2; N
16.   Diff(i) = Term(1, i) - Term(1, i+1);
17. end
18. SinX = sum(Diff); %$adderTree
```

**Figure 5.7.** Sin(x) Taylor Series Fully Parallel Example

The nested loops (statement 23 - 28) are used to compute the sum of differences. The inner loop (indexed by j) iterates over M threads while the outer loop (indexed by i) iterates over the addresses in each RFILE within the thread. Statement 25 in the inner loop is subtracting the terms in each thread and statement 26 is used to accumulate these differences. VESYLA makes sure that all these threads work in parallel on DRRA. Lastly, an adder tree accumulates all the values obtained from parallel threads to get the final sum. This final sum is stored in variable SinX which is mapped to RFILE.

**Fully parallel solution**
The code presented in Figure 5.6 can generate partially parallel and fully serial solutions by changing the values of M and N. This code is not capable of generating fully parallel solution because in the fully parallel solution there is no need for doing internal accumulation. The code and datapath presented in Figure 5.7 shows a fully parallel solution for implementing the sin(x) Taylor series. This code is similar to the code presented earlier. The computation is still divided over multiple loops and all the statements related to the internal accumulation is omitted from this version.

5.1.3 FFT

This section describes an example of mapping FFT (radix-2 with real and imaginary butterflies) using VESYLA on DRRA. The code presented in Figure 5.8 can implement a fully serial but generic ‘n-points’ FFT where n varies from (2, 4, 8, 16, 32, 64). One just need to change the value of n in order to generate the desired FFT. This can easily be extended to 1024 and 2048 points with the help of memory support. The equations presented below are used to map the FFT on DRRA. The variables $A_0$, $A_1$, $B_0$, $B_1$ are the inputs of real and imaginary butterflies and $A_{real}$, $A_{img}$, $B_{real}$, $B_{img}$ are the outputs of the butterflies.

\[
X[k] = \sum_{n=0}^{N-1} W_n^{nk} x[n], 0 \leq k \leq N - 1 \quad (5.5)
\]

\[
A_0 = A_{real} + W_{real} B_{real} \quad (5.6)
\]

\[
A_1 = A_{real} - W_{real} B_{real} \quad (5.7)
\]

\[
A_{real} = A_0 - W_{img} B_{img} \quad (5.8)
\]

\[
B_{real} = A_1 + W_{img} B_{img} \quad (5.9)
\]

\[
B_0 = A_{img} + W_{img} B_{real} \quad (5.10)
\]

\[
B_1 = A_{img} - W_{img} B_{real} \quad (5.11)
\]

\[
A_{img} = B_0 + W_{img} B_{real} \quad (5.12)
\]

\[
B_{img} = B_1 - W_{img} B_{real} \quad (5.13)
\]
MATLAB/VESYLA code for FFT

```
1. N = 64; % Number of points in FFT
2. Max_Stages = log2(N); % Maximum Stages
3. row = 1; column = 1;
4. address_range = (0 : N-1); %$Range
5. phy_address_1 = (0 : 2 : N-2); %$Range
6. phy_address_2 = (0 : 2 : N-1); %$Range
7. % $ L0 = DPU[row, column];
8. % $ L1 = DPU[row, column+1];
9. % $ L2 = DPU[row+1, column];
10. % $ L3 = DPU[row+1, column+1];

11. load('ArBr.mat', 'ArBr'); %$ RFILE[row, column]; bitrvers ; phy_address_1
12. load('AiBi.mat', 'AiBi'); %$ bitrvers RFILE[row, column+1]; bitrvers ; phy_address_2
13. load('WrWi.mat', 'WrWi'); %$ RFILE[row+1, column]; address_range
14. BrBi_copy = zeros(1:N);
   %$ RFILE[row+1, column+1]; address_range

15. for stage = (1 : Max_Stages)
16.     k = 1;
17.     BitRvr1 = bitreverse (phy_address_1, MAX_STAGES - stage + 1, 16);
18.     BitRvr2 = bitreverse (phy_address_2, MAX_STAGES - stage + 1, 16);
19.     limit = 2^(stage - 1);
20.     index = 2^(MAX_STAGES - stage);
21.     for i = (1 : limit);
22.         j = (k : (k + index) - 1);

23.         A0(j) = ArBr(BitRvr1(j)+1)+(WrWi(i).* ArBr(BitRvr2(j)+1)); %$ L0
24.         A1(j) = ArBr(BitRvr1(j)+1)-(WrWi(i).* ArBr(BitRvr2(j)+1)); %$ L0
25.         B0(j) = AiBi(BitRvr1(j)+1) + (WrWi(i).* AiBi(BitRvr2(j)+1)); %$ L1
26.         B1(j) = AiBi(BitRvr1(j)+1) - (WrWi(i).* AiBi(BitRvr2(j)+1)); %$ L1

27.         BrBi_copy(j) = A0(j) - (WrWi(i + N/2).* AiBi(BitRvr2(j)+1)); %$ L2
28.         ArBr(BR_1(j)+1) = A1(j) + (WrWi(i + N/2).* AiBi(BitRvr2(j)+1)); %$ L2

29.         BrBi_copy(j + N/2) = B0(j) - (WrWi(i + N/2).* ArBr(BitRvr2(j)+1)); %$ L3
30.         AiBi(BitRvr1(j)+1) = B1(j) - (WrWi(i + N/2).* ArBr(BitRvr2(j)+1)); %$ L3

31.         ArBr(BitRvr2(j)+1) = BrBi_copy(j); %$ RFILE
32.         AiBi(BitRvr2(j)+1) = BrBi_copy(j + N/2); %$ RFILE

33.     k = k + index;
34.     end %$ for
35. end %$ for
```

Figure 5.8. MATLAB/VESYLA code for FFT
The code is divided into declaration and computation sections and line 1 - 14 and declarative pragmas

- N is a generic pragma which can be modified to change the number of points in FFT. For this example 64 is used for implementing a 64-point FFT.

- MAX_Stages are derived from the value of FFT points. In this example, the value of MAX_Stages = 6 since a 64-point FFT needs to be implemented.

- phy_address_1 and phy_address_2 are the address_range used for storing the real and imaginary samples.

- %L0 - %L3 are the DPUs which are going to compute the butterfly operations for real and imaginary parts. Each equation presented earlier is going to use a DPU to perform the computation.

- ArBr is pointing towards the register file which is going to store/provide the real part of the butterfly and AiBi is pointing towards the register file which is going to store/provide the imaginary part of the butterfly.

- WrWi is pointing towards the register file which contains twiddle factors for butterfly operations.

The computation is divided into two loops.

**The outer loop** iterates over the number of stages in each butterfly.

**Bit-reverse addresses** BitRvrs1 & BitRvrs2 (in statement 17 & 18) are calculated at the start of this loop. These bit-reverse addresses are going to be used in each stage for computing the butterfly operations. These bit-reverse addresses are unique for every stage and are calculated once at the beginning of each stage.

**limit** is used to control the generation of twiddle factors in each stage. The generation of twiddle factor varies for each stage and limit (statement 19) makes sure to provide these twiddle factors in the unique order to each butterfly operation in every stage.

**index** is controlling the decomposition of FFT in each stage. It makes sure to read the appropriate number of real and imaginary samples in each stage.

**The inner loop** computes the real and imaginary butterfly operations by providing the data samples and the twiddle factors. Statement 23 - 32 are VESYLA/DRRA equivalents for equation presented earlier in this section. Each equation is translated into one butterfly operation and is mapped to the respective DPU. An extra RFILE is used in this implementation to delay Breal and Bimg.

Figure 5.9 shows the implementation of a 64-point FFT using one real and one imaginary butterfly. RFILE[1,1] stores the Real samples and RFILE[1,2] stores the imaginary samples. These RFILEs are configured to generate bit reverse addressing instead of linear addressing. Twiddle factors are stored in RFILE[2, 1] and one...
more $\text{RFILE}[2, 2]$ is used to create a delay line for $\text{Breal}$ and $\text{Bimg}$. Each butterfly operation takes a single cycle to complete (after the initial pipeline delay).

VESYLA generates all the address modes, DPU modes, switch box instructions and configures the datapath accordingly.

Figure 5.9. DataPath generated by VESYLA for computing FFT
5.1.4 Matrix operations

In this section mapping matrix operations on DRRA using VESYLA is discussed.

Transpose Operation

Transpose of a matrix is computed by converting all the rows of a given matrix into columns and vice-versa. Figure 5.10 shows MATLAB (VESYLA) code for computing transpose of a given Matrix A. The value of ‘N’ is used to change the size of matrix and the value of ‘M’ is used to change the parallelism. VESYLA assumes the data of Matrix is stored as row wise in RFILE; for example, the addresses 1 - 3 are reserved for the first row, 4 - 6 are reserved for the second row and 7 - 9 are allocated to the third row for a 3 x 3 matrix to be stored in a single RFILE (using physical address range 1 - 9). VESYLA also assumes that the data from the memory is already available in RFILEs.

```
1. N = 9;
2. M = 1;
3. size = 3;
4. row = 0;
5. column_range = (1:M); %$Range
6. address_range = (1:N); %$ Range
7. load('A.mat', 'A');
   %$RFILE[row, column_range]
8. load('B.mat', 'B');
   %$RFILE[row+1, column_range]
9. for i = 1 : size
10.    for j = 1 : size
11.     B((i-1)*size + j)) = A((j* size + (i-size))
12. end
13. end
```

Figure 5.10. MATLAB/VESYLA code for Matrix operations

This is a fully parallel version and the body of loop converts rows into columns. The partially parallel versions needs more switch box configuration instructions because there are only two RFILE write/read ports and they are accessed by n RFILEs which introduces overhead. The MATLAB keyword Transpose is not supported by VESYLA. The idea is that whenever VESYLA encounters the keyword “Transpose” then it needs to change the orientation of the given matrix. User will be allowed to use all the pragmas as it uses normally. This is part of the future work.

Addition and Subtraction Operations
The storage of data in RFILE is exactly the same for Matrix addition and subtraction as described in the previous section. VESYLA assumes the data of Matrix is stored as row wise in RFILE and also assumes that the data from the memory is already available in RFILEs. The value of N is used to change the size of matrix and value of M is used to vary the parallelism.

5.2 Energy Estimation

The micro-code generated by VESYLA for the Sequencer, very precisely specifies the number, type and topological relationship of DRRA resources involved and activated for static functions like FIRs and FFTs. In other words the following information is available at compile time:

1. \(#\text{Cy}\) - the number of cycles
2. \(#\text{r.m}\) - the number of times a used DRRA resource is activated in a specific mode (\(r\) is resource and \(m\) is the mode)

For such static functions, predicting performance and energy is fairly straightforward, given that the DRRA resources are well characterized with post-layout data.

![Figure 5.11. The Energy Estimation Flow](image)

To deal with the adaptive functions where the number of cycles and activation of different modes of DRRA resources are dependent on run-time input data or some global information, the algorithms (in MATLAB) are executed for sufficiently large number (> 10K) times to find out the average number of times the individual
5.2. ENERGY ESTIMATION

basic blocks are executed [9]. This information is combined with the generated micro-code to find out the average number of times, one can expect each DRRA resource to be fired and in which mode. The values (average) of #Cy and #r.m. can also be predicted by using the profiling data. The technique to automate the instrumentation, profile and associate the count with the micro-code has been mastered in-house where this method has been developed for large SOCs and RISC processors in the software development context.

5.2.1 Estimated Average Energy

The characterization process to calculate energy consumed (UEr.m) is a one time activity. “The DRRA resources (RFILEs, DPU(s), Sequencers, Interconnects) are characterized by simulating the gate level netlist back annotated with post layout data in NC-Sim. The switching activity in a VCD file is imported in the CADENCE Encounter tool to find out the average energy per cycle for each resource in specific modes” [9]. The simulation was done by providing at least 1000 random stimuli to the resources in specific modes. The simulation process is automated with the help of scripts. These scripts generates various solutions where DRRA resources are initialized in different modes to generate stimuli for them as shown in Figure 5.11 [9].

<table>
<thead>
<tr>
<th></th>
<th>DPU</th>
<th>RFILE</th>
<th>Sequencers</th>
<th>Interconnects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial 31 Taps</td>
<td>1</td>
<td>111.6</td>
<td>2</td>
<td>307.5</td>
</tr>
<tr>
<td>Serial 41 Taps</td>
<td>1</td>
<td>1552.0</td>
<td>2</td>
<td>408</td>
</tr>
<tr>
<td>Parallel 6 Taps</td>
<td>1</td>
<td>322.0</td>
<td>5</td>
<td>144.9</td>
</tr>
<tr>
<td>Partially Parallel 6 Taps</td>
<td>3</td>
<td>2098.2</td>
<td>5</td>
<td>664.1</td>
</tr>
<tr>
<td>FFT 1 Butterfly 64 points</td>
<td>4</td>
<td>48880</td>
<td>4</td>
<td>15516</td>
</tr>
<tr>
<td>FFT 2 Butterfly 64 points</td>
<td>8</td>
<td>54745</td>
<td>7</td>
<td>16559</td>
</tr>
</tbody>
</table>

Table 5.1. “Estimated and actual average energy for DSP algorithms” [9]

Characterizing DPU(s) and Sequencer(s) is relatively an easier task as compared to characterizing RFILE(s) and Interconnect(s). The RFILEs can be configured into various addressing modes, can use loops and delays, can read/write in parallel, can distribute data on both left/right segment of the output bus, and the ports can be active or passive [9]. The complexity is very similar for characterizing the Interconnect(s), as the activity depends on the state of switches, number of active inputs/outputs through/from a particular switch, length of the wire segment,
number and positions of switch-boxes involved etc. “While clock gating was used, there is still some UEr.m for resource even when it is inactive, in other words, all resources have a mode called inactive as well. Since characterizing the UEr.m space is a one time activity and not dependent on any specific function, its implementation instance and the size of the DRRA fabric (regular layout and segmented wires ensure that the fabric size does not impact congestion and wire length), expending large time and space (look up table) to accurately represent the UEr.m space is justified” [9].

“On a quad core Pentium class Linux machine, it takes roughly 5 days to fully characterize the UEr.m space. This was done for 90 nm technology using Cadence RTL and Encounter tools”[9]. The average energy for different algorithms are presented in Table 5.1. The average energy is reported because the power consumptions is dependent upon the switching activity and combination of input data. The estimated energy is broken down in terms of different DRRA resources (number of columns are indicating the resources involved). Table 2 reports lumped value of each resource (activated in different modes) because reporting the individual energy numbers for each mode of operation for every resource will make Table 2 unreasonably large[9]. “The total estimated values are Estimated Total Average Energy compared against the actual average dynamic energy consumed. This was found by simulating the gate level netlists with VESYLA generated configware of specific implementation instances listed in Table 2. As can be seen the percentage error is very small and each of this estimation is produced more or less instantaneously” [9].
Chapter 6

Conclusion And Future Work

6.1 Conclusion

Raising the level of abstraction is the key to deal with complex SoC designs and applications. Raising abstraction enables the designer to focus on the algorithm instead of spending the time in doing the guess work for efficiently exploring the design space. In this thesis we have introduced a pragma-based approach for mapping DSP functions on DRRA along with a method to estimate the average energy of such functions (which is reported with error margin of less than 3%). The methodology presented in this thesis needs minimum designer interaction by using pragmas and it effectively captures the design implementation space of DSP functions. The end user can easily generate multiple architectural solutions with varying degree in parallelism using same set of pragmas without compromising the automation. All low level tasks (scheduling, synchronization, allocation/binding) are handled by VESYLA as guided by the designer. This enables the user to focus on the details of of algorithm (at higher level of abstraction). The MATLAB code embedded with pragma serves as golden model and the solutions generated by VESYLA can be verified immediately after execution of MATLAB code.

The benefits are quite evident from the examples being presented. Small changes in pragmas generates different solutions which would have taken much more time if it was done manually at lower level of abstraction. The library can be populated with many solutions for the same function or architecture and this would enable the system level design tool to generate better optimized solutions.

6.2 Future Work

some of the near term goals for VESYLA are described next.

- The next immediate step is to equip VESYLA to deal with conditional statements mapping on DRRA.
- VESYLA lacks the hierarchical control scheme [31] and is a near term future goal.
- A memory architecture is under development and VESYLA needs to add pragmas for supporting memory transactions.
- The pragmas introduced so far mainly identify and exploit SIMD type of parallelism. VESYLA needs to support more pragmas for loop unrolling etc.
- VESYLA should support MATLAB built-in functions (like transpose etc.) and constructs.
- Detailed and methodological characterization is required for all the DRRA resources in order to predict the estimated average energy for any DSP function mapped to DRRA.
- VESYLA should also support dynamic and run-time calculations.
- At the moment VEYSLA is not capable of dealing with non-affine loop indexing and adding support for this feature is also a near term goal.

Pragma smart grouping

The pragmas needs to be introduced smartly instead of having a single pragma for every line of code as shown in Figure 6.1. This method reduces the number of lines and make things even more simpler for the developers. All the pragmas which fall into the same category can be combined together. For example, statements 2 - 6 are covered by range pragmas. Similarly, statements 7 - 10 are covered by RFILE pragmas. If there is a need to change some parameter then it can be introduced directly. For example, statement 10 is using different column (‘X’ is a don’t care) as compared to other RFILE pragmas.

![Figure 6.1. An example code introducing pragmas smartly](image-url)
Bibliography


BIBLIOGRAPHY


