FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

VLSI design of configurable low-power Coarse-Grained Array Architectures

Diogo Alexandre Ribeiro de Sousa

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Advisor: Professor Doutor João Paulo de Castro Canas Ferreira

July 13, 2017

Abstract

Coarse Grained Reconfigurable Arrays have gained importance in the field of accelerators. Several types of architectures have been proposed in the literature mainly targeting applications in the multimedia field. This document aims to contribute to the application of CGRAs in different areas by targeting low-power architectures for biomedical signal processing. The objective is to design a low power architecture which may be placed in a small battery-operated portable device. To do so, a look is taken into the different types of power consumption in a chip giving special attention to static power consumption.

To produce a chip EDA (Electronic Design Automation) tools are used. These tools impose a considerable time overhead which delays the project. The purpose of the design flow is to ease the process of taking a CGRA architecture to tape-out in addition to save a considerable amount of time spent dealing with the aforementioned tools. The proposed design flow is capable of transforming a HDL description of a CGRA in a physical design while applying low-power methodologies such as the insertion of power domains along with power-gating capabilities which will deal with the static power consumption previously mentioned.

There is also a set of use cases which assess the efficiency of power-gating in a CGRA which is helpful for a designer who wishes to understand how the grouping of elements in power domains and how shutting them down impacts the system's power efficiency.

This strategy is applied to a proposed CGRA architecture which reveals power savings in the order of 31.6% when powering down $1/3$ of the circuit and, if the CGRA is not being used, shutting it completely off achieves savings in the order of 99.93%.

These results show that this is a promising approach which could enhance the way medical exams are made and the time it takes for a patient to be diagnosed. In addition, a step towards having portable devices performing medical exams would grant doctors extra time to deal with matters of greater importance.

Agradecimentos

Durante a realização deste trabalho foram várias as pessoas que me empurraram na direção certa e a todas essas pessoas devo os meus mais profundos agradecimentos.

Ao meu orientador, Professor João Canas Ferreira, gostaria de agradecer pela orientação e por todo o apoio tanto a nível académico como pessoal, que em muito me ajudaram a atingir os objetivos deste trabalho.

Aos meus amigos da I224 (Artur, Miguel, Pedro, Chico, JDA, Rui, Baixinho e Lopes), e ao André, gostaria de deixar um agradecimento pelo ambiente proporcionado ao longo do semestre, assim como por todo o espírito de entreajuda e companheirismo que revelaram.

Gostaria de agradecer também aos meus grande amigos José Espassandim, João Silva, Marina Castro, João Ungaro, Luís Daniel Almeida e Ricardo Almeida pela vossa amizade, que guardarei para sempre no meu coração.

Para ti, Mariana Tedim Dias, não consigo encontrar palavras suficientes para expressar a importância que tens na minha vida. És o meu maior apoio em todos os sentidos e serás, para sempre, o motivo principal do meu sorriso.

Finalmente, aos meus pais, Fernanda e Augusto, ao meu irmão, Pedro, à minha avó, Belmira, e ao meu avô, Joaquim, que infelizmente não irá partilhar esta alegria comigo, deixo um agradecimento do fundo do meu coração por me terem criado e por terem feito de mim a pessoa que sou hoje. Espero que estejam orgulhosos.

Diogo Ribeiro Sousa

“I am very fond indeed of it, and of all the dear old Shire but I think I need a holiday.”

J. R. R. Tolkien

Contents

List of Figures

List of Tables

Abbreviations

ALU	Arithmetic Logic Unit
ASIC	Application-Specific Integrated Circuit
CGRA	Coarse-Grained Reconfigurable Array
CMA	Cool Mega-Array
CPU	Central Processing Unit
CTS	Clock Tree Synthesis
DEF	Design Exchange Format
DRC	Design Rule Checking
DSP	Digital Signal Processor
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FPGA	Field-programmable Gate Array
HDL	High Description Language
ITF	Interconnect Technology File
IP	Intellectual Property
LEF	Library Exchange Format
LPM	Low Power Methodology
LPMM	Low Power Methodology Manual
LVS	Layout vs. Schematic
MMMC	Multi-Mode Multi-Corner
PAE	Processing Array Element
PD	Power Domain
PE	Processing Element
PLE	Physical Layout Estimation
RAI	Roundabout Interconnect
RC	Reconfigurable Cell
REACT	Real-Time Seizure Detecting Algorithm
RFU	Reconfigurable Functional Unit
RTL	Register-Transfer Level
SDC	Synopsys Design Constraints
SDF	Standard Delay Format
SoC	System-on-Chip
STA	Static Timing Analysis
SS	Supply Set
UPF	Unified Power Format
VCD	Value Change Dump
VLSI	Very Large Scale Integration
VLIW	Very Long Instruction Word

Chapter 1

Introduction

1.1 Context

Biological signals are of major interest in terms of health monitoring. With today's technology it is possible for an individual to carry in or on himself electronic devices capable of harvesting and even processing signals from body sensors. As detailed in [1] there are many types of bio-signals that can provide vast information about an individual, from respiration rate to blood pressure and there is even the possibility of performing electrocardiograms (ECGs) on the fly. This ability to monitor relevant signals can help doctors act in a timely manner and even prevent many health issues. However, a big interest in harvesting these data is to do it while the patient is on the move or at least not in the premises of a hospital, ideally allowing the patient to not even notice that he's being monitored.

Monitoring the aforementioned signals often gives rise to the need of local pre-processing, either to weight the relevance of the collected data or to treat the information before sending it to a remote processing unit thus possibly decreasing the data payload. This processing may consist in cleaning the signal (removing noise), calibrating the sensors or even compressing data.

With the aggressive energy consumption to which this processing is subjected and given its restrictions while having, at the same time, real-time requirements, there is a rise in the need of carefully aiming towards energy-efficiency. One way of addressing energy-efficiency is by designing application-specific circuits that aim to boost performance while reducing energy consumption. Nevertheless, the development of these application-specific circuits is in many cases very risky as there exist economic and time-to-market constraints which can render the whole development useless.

A possible solution[2] comprises the development of a general-purpose architecture combining a microcontroller or a microprocessor with a coarse-grained array architecture (CGRA) acting as an accelerator.

The CGRA is an array of processing elements (PEs) whose function and interconnections are determined by some configuration data. When carefully designed, it is possible to map all the needed algorithms for bio-signal processing in this architecture and efficiently diminishing power consumption.

1.2 Motivation

Reconfigurable architectures are much more energy-efficient than general-purpose CPUs. "It is shown that reconfigurable computing designs are capable of achieving up to 500 times speedup and 70% energy savings over microprocessor implementations for specific applications"[3]. [4] take advantage of a SYSCORE CGRA architecture to run a real-time seizure detecting algorithm (REACT). For certain features of the REACT algorithm, like for example *Fisher Information*, the proposed architecture showed 40% energy savings when compared to a DSP processor.

It is shown in [5] how energy-efficiency can be achieved by applying power-gating techniques to the CGRA. These techniques consist in switching off parts of the array, i.e. a group of PEs, when possible. This type of technique must be applied using a Low Power Methodology (LPM) and must be done by someone who has a deep knowledge of the design flow for Systems-on-Chip (SoCs).

In order to reduce the overhead of designing a CGRA, this dissertation will set its scope on having a semi-automated process to generate a physical layout from the description of the PEs with the insertion of scalable power-gating techniques (detailed in the next sections).

1.3 Objectives

The objective of this dissertation is to have a defined design flow for CGRAs that can be translated to a set of scripts which can then be used to synthesize a physical layout of a CGRA using standard cells, thus reducing the overhead of manually executing each stage of the design flow. While doing this, one major objective is the exploitation of tools and techniques to automatically insert power-gating in the design in order to reduce power consumption. A complementary objective is the analysis of the impact of the mentioned techniques in terms of power reduction and the adaptation of these tools and techniques to different designs.

Please note that throughout the document a webpage will be mentioned. This webpage may be found here: `https://paginas.fe.up.pt/~ee12173`.

Chapter 2

Literature Review

2.1 Coarse-Grained Reconfigurable Arrays

SoCs (Systems-on-Chip) have been extensively developed during the last years for applications in several industries, from automotive to consumer electronics or even military applications. Although these chips are suitable for intense applications and have relatively low power consumption, we must take into account the amount of time spent in the design of an application specific chip. With the constant changes in today's technology comes the need for a swift response to the market's needs, compelling design companies to deliver a product as soon as possible in order to be competitive. Nowadays, SoC designers face great challenges in terms of time having to deal with bottlenecks in floor-planning and chip layouts, especialy in advanced CMOS design where wiring is critical [6].

In order to face this problem, there has been a crescent interest in coarse grained reconfigurable fabrics (CGRAs) attached to a general purpose CPU (Central Processing Unit). These coarse grained reconfigurable fabrics come as a compromise between the application-specific chip and the fine-grained architectures [7] such as FPGAs (Field-programmable Gate Arrays), giving the chip manufacturer enough flexibility to reconfigure the CGRA's internal interconnections and change the system's behaviour.

2.1.1 Generic CGRA Architecture

A CGRA is composed of an array of interconnected PEs commonly containing ALUs (Arithmetic Logic Unit) and other functional blocks such as registers, bit-wise operators, multiplexers, among others. In it's simplest version, as seen in figure 2.1, all the elements are equal, making the CGRA homogeneous (equal PEs for the entire array) with nearest-neighbour connections. In this case we can see that there are 16 PEs (4 per line and 4 per column) making this, as mentioned before, a very simplistic view of a CGRA architecture.

Looking at figure 2.1 there are some aspects of the CGRA that can already be seen and will be detailed during the course of this report. One of these aspects is the homogeneity of the processing elements which is an important fact to take into account if the designer is aiming at

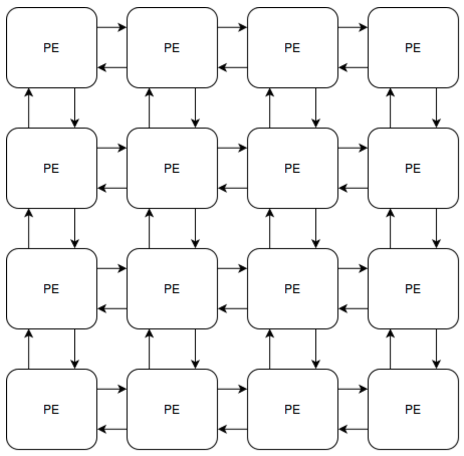


Figure 2.1: CGRA fabric

area optimality as well as functionality. Another crucial aspect, as mentioned earlier, is the way the interconnections are made, either if each PE is connected to its nearest neighbours, if it uses buses, roundabout connections or even if it has run-time reconfigurable interconnections.

2.1.1.1 Processing Element

The PEs are the computational units of a CGRA. Together with a well structured network of interconnections the PEs are able to perform and accelerate computations that are usually performed in a general-purpose CPU. Several types of PEs have been proposed in the literature. MorphoSys[7], detailed in figure 2.2, has a PE (the authors call it Reconfigurable Cell (RC)) composed of an ALU-multiplier, a shift unit, two multiplexers for input selection, one output register and other registers for configuration purposes. Other authors proposed similar architectures, like the CS2112[6] (fig-

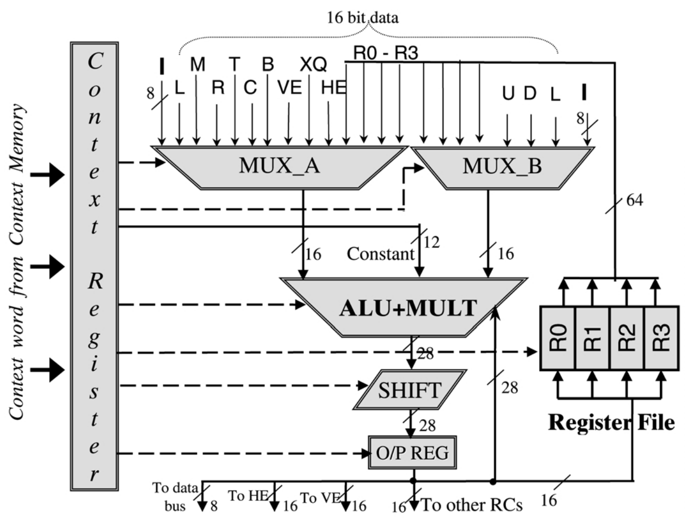


Figure 2.2: MorphoSys PE. Source:[7]

ure 2.3), which comprises an ALU, output registers and routing and shift logic for the input data. All of these elements are again connected to some kind of configuration data.

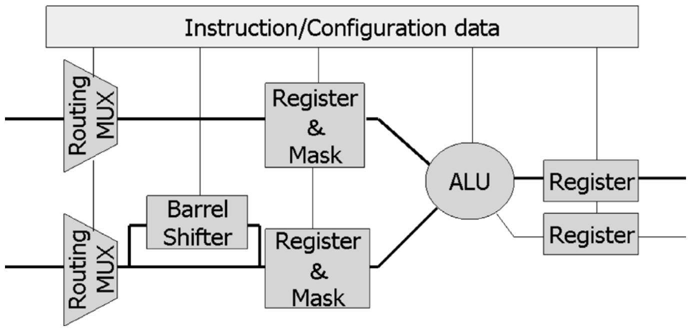


Figure 2.3: CS2112 PE. Source:[6]

An example of a complex structure is the XPP-III[8] which has three types of PEs in its core structure (fig. 2.4). The XPP-III contains ALU-PAEs (Processing Array Elements), RAM-PAEs and

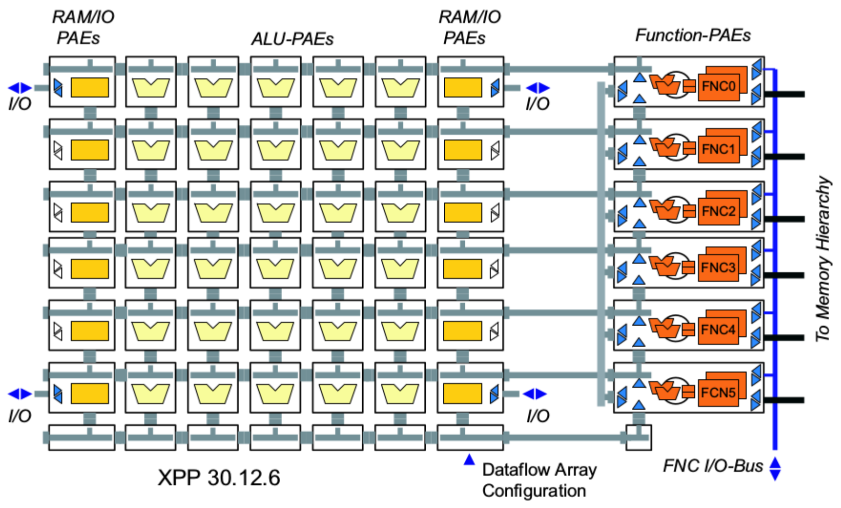


Figure 2.4: XPP-III Core Structure. Source:[8]

FNC-PAEs. The ALU-PAEs contain three ALUs while the RAM-PAEs contain only two ALUs together with a small RAM (Random Access Memory) and an I/O object. The FNC-PAE contains a complete VLIW-like sequential processor kernel[8].
Many architectures described in the literature are similar in terms of PEs, containing a standard ALU and encapsulating some other functionalities the designer finds suitable in order to target a specific algorithm. Even with very similar PEs there is a vast set of options a designer can choose from and this comes from the way they're structured in terms of homogeneity, interconnections, bit-widths, number of PEs and overall configuration. One example is the number of multipliers and their ratio to other PEs. Some of the architectures in table 2.3, that are heterogeneous, contain multiplication PEs. This is summarized in table 2.1.

Name	Multipliers	Other PEs	Ratio
CS2112 [9]	24	84	1 : 3.5
DAPDNA-2 [10]	56	168	1 : 3.3
FE-GA [11]	8	24	1 : 3

Table 2.1: Number of multiplication PEs. Source:[6]

2.1.1.2 Interconnection network

One of the issues when physically dividing an accelerator into several PEs is the way they exchange data. Besides, with an increasing number of computational blocks on a chip, a major bottleneck may be caused by the communication among these functional blocks [12].
There are several ways of interconnecting PEs. The simplest way is to connect every PE to its nearest neighbours, considering or not diagonal connections. Although nearest neighbour connections are easy to understand and to implement, they bring some drawbacks. For example, if you need to obtain the result of a computation made in a cell that's a few PEs away from the output, you will need to use those intermediate PEs simply as pass-through PEs, making them unusable thus degrading efficiency. Besides this there is also the possibility of causing deadlocks since one PE may be depending on the results of several other PEs to carry on.

To address this problem other ways of interconnecting PEs have been developed. Bus interconnections allow communication between any two PEs but have some disadvantages regarding the number of PEs that can communicate on the same bus at the same time and the area overhead needed to implement hardware capable of communicating using a bus-compatible protocol. Some designers, as seen in [7], even combine these techniques grouping PEs in quadrants and delivering two different kinds of communication infrastructure, one for intra-quadrant communication and another for inter-quadrant communication. Besides nearest-neighbour and bus connections other interconnection methods have been proposed, like torus (many times combined with nearest-neighbour), roundabout connections[4] (figure 2.5), among others (see table 2.2).

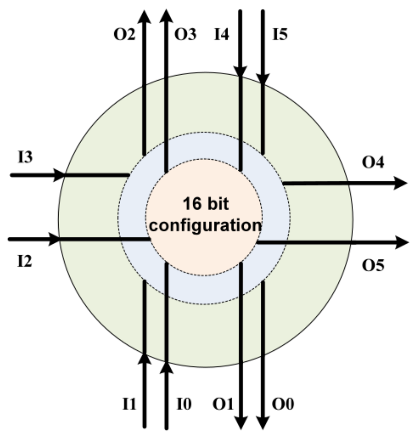


Figure 2.5: Syscore Roundabout Interconnections. Source:[4]

Name	Interconnect
CS2112 [9]	Tile base, 2D-bus
DAPDNA-2 [10]	Segment base, 2D-bus
FE-GA [11]	2D-mesh direct, Crossbar for memories
Cluster Machine [13]	3-stage switch
DRP-1 [14]	Tile base, 2D-bus
Kilocore KC256 [15]	Crossbar for row direction
ADRES [16]	2D-mesh direct with extra links
Xpp-64 [17]	2D-bus and direct
D-Fabrix [18]	Chess-board like switch connection

Table 2.2: Interconnection types. Source:[6]

It's not practical to design a CGRA where the PEs have all-to-all interconnections. It is a question of both area and power. If PEs are connected without a critical way of thinking, this will probably have a big impact on the final product in terms of area and power consumption. Increasing the number of connections increases the amount of metal and probably the number of metal levels, therefore also increasing the number of vias which has an impacting on the design's timings. Also, several wires may connect to a particular input, and will have go through a multiplexer in order to select which one of them is driving the input. This results in an area overhead that becomes intolerable when increasing the number of wires driving a single input.

An interesting alternative was published in [19], where the authors propose the use of *Omega Networks*. In other words, the authors propose using several switch boxes interconnected that route the signal reducing the number of wires. An example of these switch boxes can be seen in figure 2.6.

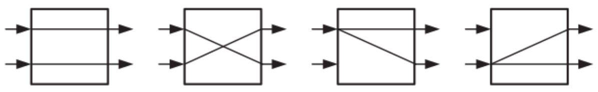


Figure 2.6: Types of Switches. Source:[19]

These switch boxes can be combined in order to drive 4 inputs (fig. 2.7(a)) and 8 inputs (2.7(b)). It can be shown that it is possible to establish a route between any input/output pair.

2.1.1.3 Reconfiguration

A coarse grained structure by itself does not guarantee the desired performance cost ratio. Making use of the fabric's reconfigurability, i.e. by using a single PE array for multiple tasks, area efficiency may be enhanced and the semiconductor area can be utilized more efficiently compared with dedicated hardware logic[6].

This dynamic reconfiguration can be achieved in a few different manners. The simplest way is

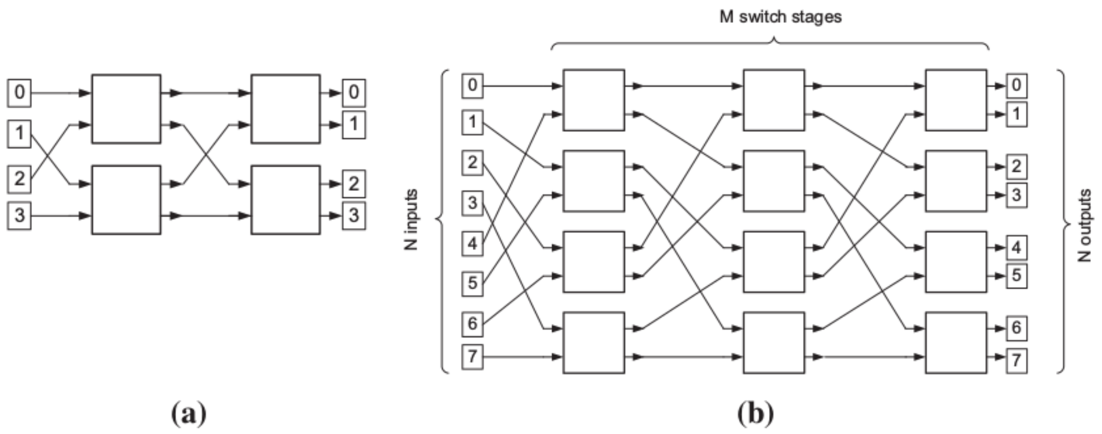


Figure 2.7: Switch interconnections. Source:[19]

by storing the configuration data in one or various on-chip memory modules [6]. These modules will then be able to deliver the memory contents to the PEs during run-time in order to change their configurations and thus switching the functionality of the PE.
Another method of dynamic reconfiguration consists in placing memory modules inside each PE that will store context data. This method allows every PE to be configured at the same time, since the context controller only feeds the PEs with the context number, allowing all PEs to reconfigure in a single clock cycle. Although this reconfiguration method is faster, it brings with it an area increase that can, in some cases, double the area of each PE [16].

In table 2.3, a number of architectures proposed in the literature are classified according to the properties explored before. The reader may notice that architectures show very different numbers of PEs. This is mainly a result of the operations that which PE is capable of executing. As a practical example: if no PE supports multiplication, there may exist the need for various PEs in a

Name	Configuration	PE array	Data bits	PEs
CS2112 [9]	Multicontext(8)	Heterogeneous	16/32	108
DAPDNA-2 [10]	Multicontext(4)	Heterogeneous	32	376
FE-GA [11]	Multicontext(4)	Heterogeneous	16	32
Cluster Machine [13]	Multicontext	Heterogeneous	16	15/c
DRP-1 [14]	Multicontext(16)	Homogeneous	8	512
Kilocore KC256 [15]	MMulticontext/ /Delivery	Homogeneous	8	256
ADRES [16]	Multicontext(32)	Homogeneous	16	64
Xpp-64 [17]	Delivery	Homogeneous	24	64
D-Fabrix [18]	Delivery	Homogeneous	4	576
S5-engine [20]	Delivery	Homogeneous	4/8	-

Table 2.3: Features of CGRA architectures. Source:[6]

pipelined fashion in order to deal with timing requirements, thus increasing the number of needed PEs. The bit-width, however, is reflected by the target application.

2.1.2 Examples of architectures proposed in the literature

2.1.2.1 MorphoSys

MorphoSys is a reconfigurable computing system that targets applications with inherent data-parallelism, high regularity and high throughput requirements, like video compression, graphics and image compression, data encryption and DSP transforms[7].

This architecture (in figure 2.8) is comprised by a reconfigurable processor array with a high bandwidth data interface both connected to a system bus. Together with these elements the designers also included one RISC processor[21] and an instruction/data cache.

This architecture is designed to operate on 8 or 16-bit data and can be dynamically reconfigured by loading context data into inactive parts of Context Memory without interrupting the PE Array operation. These context data loads are managed by the host processor.

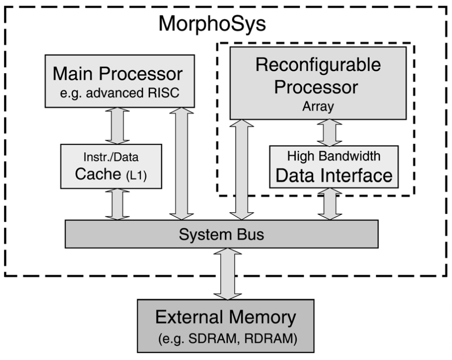


Figure 2.8: MorphoSys Architecture. Source:[7]

The context memory present in this architecture allows 32 planes of configuration to be stored and allows configuration-related data to be broadcast either to PE columns or rows.

2.1.2.2 ADRES

The ADRES architecture has two distinct parts. One of the parts contains a VLIW processor suitable for control and load/store operations while the other part contains a reconfigurable fabric that serves as an accelerator optimized for data-flow kernels[16]. This architecture makes use of orthogonal buses and each Functional Unit (FU) is fitted with two configurable ports to facilitate data input. In terms of data output, it is achieved by means of both vertical and horizontal distribution.

This architecture is provided with a data-width of 32 bits distinguishing it from regular fine grained architectures which contain mostly bitwise operations. On the VLIW part there can be up to eight FUs in a row, communicating via a horizontal data bus. As mentioned before, there is a reconfigurable fabric attached to this VLIW block, containing several rows of PEs controlled by a local context memory. In the implementation mentioned in [16], the mentioned context memory blocks are configured by loading data from an external memory at the boot phase of the device. These contexts are switched via a central pointer that selects the context for the whole fabric.

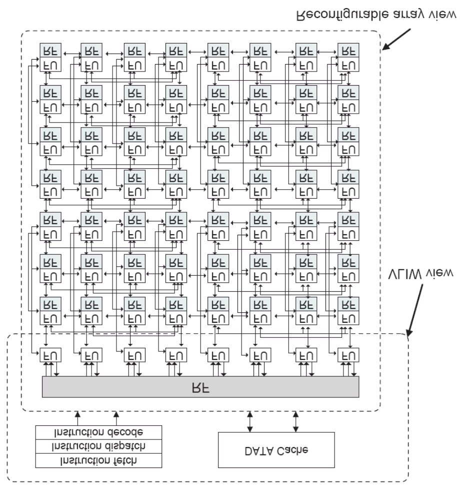


Figure 2.9: ADRES Architecture. Source:[16]

2.1.2.3 SYSCORE

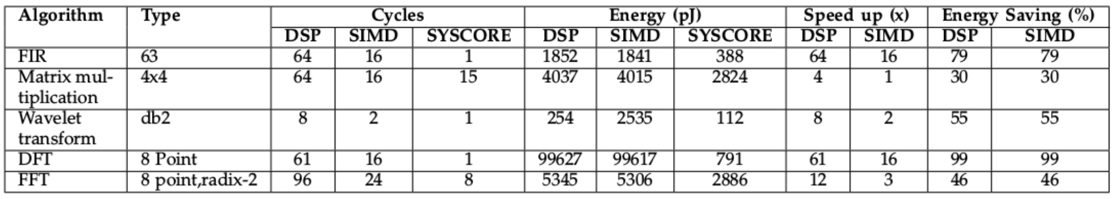
SYSCORE is a Coarse Grained Reconfigurable Array for low-power on-chip biosignal processing. This architecture surges from the need for significant savings in the energy consumption of on-chip biosignal processing and has led to research interest in low power biosignal processor platforms[22].

In [23], Patel et al. propose an 8x4 SYSCORE architecture. This architecture contains 32 Configurable Function Units (CFUs) and 8 RoundAbout Interconnect (RAI) units. Each CFU has 4 input ports and 3 output ports which feed data to a Computation Unit (CU) that suports MUL-ADD, MUL-SUB and CMP (compare) on top of the casual functions supported by a conventional ALU/MAC. This set of additional functions can be useful for systolic algorithms mapping and feature extraction [23, 24].

In terms of interconnections, this architecture is provided with nearest-neighbour connections to the East and West and cross interconnections at odd numbered columns. After the second column of CFUs a RAI is inserted to provide more interconnection options.

This architecture is prepared to operate in three distinct modes: configuration, execution and flush. The mode of operation is selected using global control signals.

To test the performance of this architecture, Patel et al. implemented an 8x8 SYSCORE array and used the RaCAMS simulator to obtain performance results[25]. This comes as a very useful information in the scope of this thesis since it also targets biosignal processing. The hardware was implemented in Verilog and the algorithms were mapped using SystemVerilog. A 90nm CMOS technology library was used and, for comparison purposes, a DSP processor was implemented, as was a SIMD processor. These performance tests resulted in the values depicted in figure 2.10 and show that not only SYSCORE is faster but it is also much more energy-efficient.



Algorithm	Type	Cycles			Energy (pJ)			Speed up (x)		Energy Saving (%)	
		DSP	SIMD	SYSCORE	DSP	SIMD	SYSCORE	DSP	SIMD	DSP	SIMD
FIR	63	64	16	1	1852	1841	388	64	16	79	79
Matrix multiplication	4x4	64	16	15	4037	4015	2824	4	1	30	30
Wavelet transform	db2	8	2	1	254	2535	112	8	2	55	55
DFT	8 Point	61	16	1	99627	99617	791	61	16	99	99
FFT	8 point,radix-2	96	24	8	5345	5306	2886	12	3	46	46

Figure 2.10: SYSCORE benchmarks. Source:[23]

2.1.2.4 Cool Mega Array

Cool Mega-Array (CMA), proposed in [26], is a reconfigurable accelerator consisting of a PE array without memory elements. This is made possible by having fully combinational PEs in opposition to the traditional PEs containing registers and memory elements as seen in many other architectures [8, 23, 27].
As this architecture targets mainly multi-media processing in battery-driven embedded systems, the main goal is to increase power efficiency. Speed is not a critical aspect to be improved since in this kind of processing there is always a certain time window to process data and there is no practical interest in reducing processing time beyond that window.

In [26] the authors take into account that the energy used is the product of power ($P = C \cdot V^2 \cdot f$, being C the capacitance of the gate, f the frequency and T the time) and suggest that since power (P) is related to the square of the supply voltage (V), there is a great interest in reducing the supply voltage as much as possible. Although this seems like a good way to increase power-efficiency, the authors also mention that the logic circuit delay increases as V approaches threshold voltage V_{th}, since the delay D is given by $D = \beta \cdot \frac{C \cdot V}{(V - V_{th})^{\alpha}}$ where $\alpha = 1.6$[26].

As mentioned before, CMA's PE array contains combinational circuits that operate with a supply voltage ranging from 0.5V to 1.2V. There are 64 PEs (8x8 array) and each PE consists of a 24-bit ALU and two input mutiplexers. This architecture, being mostly combinational, only enables its clock tree before execution, while the configured data are loaded. Also, the configuration is achieved using a multicast method "RoMultiC" [28].

2.2 Power-efficiency

Power consumption of a VLSI chip can be categorized into two different types:

- Dynamic Power Consumption;

- Static Power Consumption.

Dynamic power consumption results from the switching activity of the input signal and is proportional to its switching frequency and to the capacitance value (C_L in figure 2.11) that the logic gate is driving. In figure 2.11 the arrow represents the current flow which causes dynamic power consumption. Equation 2.1 represents the calculation of dynamic power (P) using the load

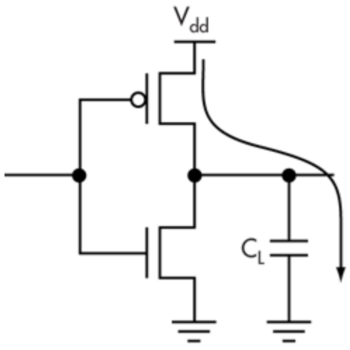


Figure 2.11: Capacitor Charging Current. Source:[29]

capacitor value (C_L) and the frequency of the input signal (f).

$$P = C_L \cdot V_{dd}^2 \cdot f \tag{2.1}$$

Static power consumption, on the other hand, results from leakage currents in the circuit. Leakage currents may be divided into four main classes (the following classes were taken from [30]):

- Sub-threshold Leakage (I_{SUB}): current flowing from the drain to the source of a transistor operating in the weak inversion region.
- Gate Leakage (I_{GATE}): current flowing from the gate through the oxide to the substrate due to gate oxide tunneling and hot carrier injection.
- Gate Induced Drain Leakage (I_{GIDL}): current flowing from the drain to the substrate induced by a high field effect in the MOSFET caused by a high V_{DG}.
- Reverse Bias Junction Leakage (I_{REV}): caused by minority carrier drift and generation of electron/hole pairs in the depletion regions.

Although these categories are due to different physical causes, they all depend on the power supply voltage.
In order to achieve power-efficiency, dynamic power can be reduced by lowering the voltage supply or the frequency. One commonly used technique in reducing dynamic power consumption is Voltage and Frequency Scaling, which consists in decreasing the supply voltage when the performance requirements are not of major concern and, since this limits the operating frequency, the

frequency will also be scaled down, lowering the speed of the circuit but saving on energy consumption.
Other well-known technique is clock gating. Clock gating consists in switching off parts of the clock tree whenever possible. As a practical example, if the input of a register is kept stable during, for the sake of the example, one million clock cycles, the clock signal can be turned off preventing the register's operation thus saving the power that otherwise would be spent saving the exact same value in the register.

However, as we move towards more advanced technology nodes, static power consumption becomes a critical issue (see figure 2.12 and table 2.4).

Node	**90nm**	**64nm**	**45nm**
Dynamic Power per cm^2	1X	1.4X	2X
Static Power per cm^2	1X	2.5X	6.5X
Total Power per cm^2	1X	2X	4X

Table 2.4: Technology Node Comparison. Source:[30]

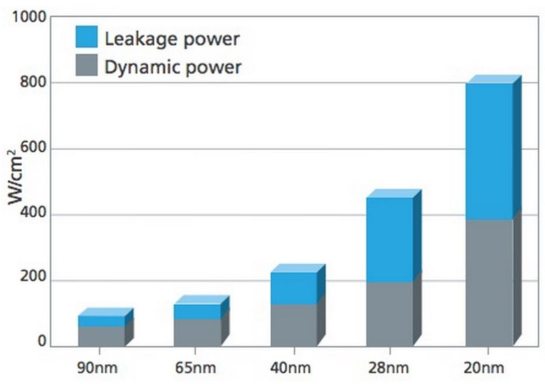


Figure 2.12: Technology Node Comparison. Source:[31]

When moving to a smaller node, the increase of static power consumption derives from the existence of a decrease of the threshold voltage (V_{th}) that causes an increase in sub-threshold leakage current [32].
Facing the problem related to static power consumption there are some common techniques like body-bias control, dual-threshold domino circuits and input vector control[32]. Another way of reducing current leakage consist in decreasing the voltage supply value thus also decreasing the leakage currents, possibly meeting a compromise between supply voltage and operating frequency. A drastic approach to this method is setting the supply voltage to zero resulting in virtually zero power consumption. This method is known as power-gating and is one of the subjects on which the scope of this dissertation is set.

Power gating (see figure 2.13) consists in the ability of switching between two power modes: an active mode and a low-power mode[30]. The active mode is when the circuit is operating in the same way as if there was no power-gating while the low-power mode sets the supply voltage to zero. This can be accomplished by the pull-up MOSFET seen in figure 2.13 which deals with driving or not the virtual V_{dd} line. Although this method reduces power consumption due to

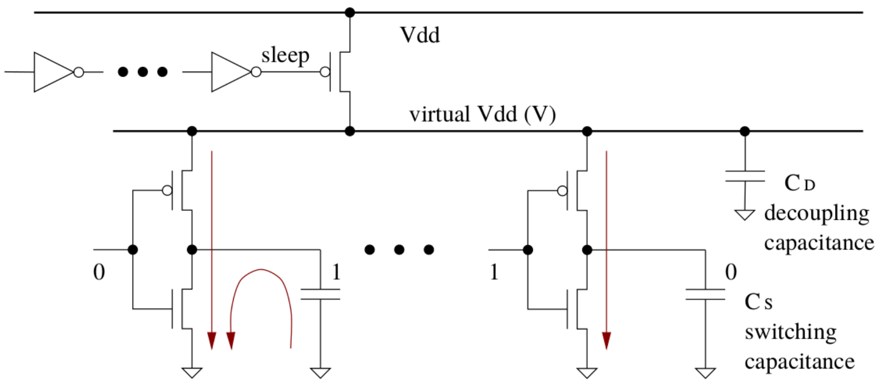


Figure 2.13: Power-Gating. Source:[32]

leakage currents to virtually zero, it shows disadvantages. Switching off a circuit, or parts of it, is only possible in the time intervals during which the circuit is idle. Another aspect that should be taken into account is the time spent in transitions between power modes. In some cases, the time and energy spent during these transitions may greatly diminish power-efficiency.

Power-gating can be implemented in a whole system, e.g. a CPU, and only be capable of switching off the entire circuit. This is known as coarse-grained power-gating. However, it is possible and in many occasions more interesting to fine-tune the circuit elements, i.e. applying a fine-grained power-gating methodology, in order to retain the functionality of main element while idle areas are power-gated[30].

Several authors have proposed methods for power-gating. For example, in [33] a method is proposed consisting in power-gating the entire processor when a long idle loop is detected [32]. Other techniques may consist of switching off the power supply from unused blocks. It has been shown that the correct use of this technique has a big impact on power-consumption [2]. Regardless of the technique there is always the need of support from EDA (Electronic Design Automation) tools which depend on well-defined design flows to process the design. These design flows will be explored during the course of this document.

Chapter 3

Problem Characterization and Proposed Solution

3.1 Biosignal Processing

Harvesting signals from body sensors requires pre-processing before sending the data to a remote node. This pre-processing may include operations such as noise removal, sensor calibration, event detection and data compression. To perform these operations there is a fair amount of algorithms that may prove to be useful such as the FFT computation, FIR filtering, matrix multiplication, correlation, among others[4, 2, 5].

Although these algorithms have been proven useful, computing them in a general purpose CPU is not a power-efficient solution and may in some cases not even be time-efficient. Therefore and taking advantage of the possible parallelization that derives from the nature of these algorithms, a CGRA can be used as an accelerator since multiple computations can be performed at the same time thus increasing performance. The decrease in computing time and increase of performance may lead to a more power-efficient platform.

To acquire knowledge on time requirements in terms of processing, there is an advantage in knowing the sampling frequency of the signals to be acquired. Table 3.1 shows the sampling frequency of a set of sensors used in biomedical signal harvesting. After a brief analysis of the

Sensor	Sampling Frequency	Description
HR	100 Hz	Heart Rate
SpO2	0.2 - 0.5 kHz	Blood Oxygen
ECG	0.2 - 1.0 kHz	Heart Elec. Act.
EEG	0.1 - 1.0 kHz	Scalp Elec. Act.
GSR	50 - 100 Hz	Skin Conductance
EMG	1 - 2 kHz	Muscle Elec. Act
RESP	50 - 100 Hz	Respiration

Table 3.1: Sampling Frequencies of Biomedical Sensors. Source:[34]

sampling frequencies, it can be concluded that there's an upper-bound of 2kHz. Using a CGRA as an accelerator to perform computations on the values acquired by these sensors requires a computation time which is expected to be immensely smaller than the sampling period. Therefore, there is no great interest in applying Frequency Scaling since the clock frequency of the CGRA can be set at a low frequency and thus not profit from a frequency reduction. Also, the supply voltage may be set to a lower value, not requiring dynamic scaling.
This being the case, power-gating reveals itself as the most promising methodology to be explored in this scenario.

3.2 Power-Gating

In this dissertation, the use of power-gating techniques will be explored and the aim is to propose a set of methodologies that automatically insert power-gating cells in a CGRA. The idea is to implement scalable power-gating. By this it is meant that the design tools (together with scripts that result from the work done during this dissertation) are able to receive an HDL (High Description Language) description of the PEs and their properties as inputs and generate a physical layout of a CGRA with power-gating and a number of PEs and interconnections specified as an input of the process. The way to include and manage power-gating in this approach is by switching off rows or columns of PEs. By doing this it is possible to obtain several performance modes by changing the number of active PEs.

This document presents a set of scripts that allow power-gating to be inserted and at the end of the design there is a chapter about results in which two use cases are presented.

3.3 Algorithms

As previously mentioned, the harvesting of biological signals requires a significant amount of pre-processing such as filtering. To filter a signal, an algorithm that makes sense is the Finite Impulse Response (FIR) filter. After having a clean signal there are many algorithms that can be applied to the gathered data, whether just to pack it and send it to a remote node or treat its information and, for example, extract features. To achieve this result more complex algorithms have been proposed and significant part of these algorithms requires, for example, matrix multiplications.

Thus, taking these two cases as an example, while reminding the reader that the objective is to have a simple architecture to assess the low-power design flow, it is possible to express the aforementioned algorithms in terms of operations that use basic operands such as addition, multiplication, difference or division.
In the case of matrix multiplication, it is known that if:

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \tag{3.1}$$

then:

$$A * B = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix} \tag{3.2}$$

which, if matrix A is of size $n * m$ and matrix B is of size $m * p$ the multiplication can be written as

$$(AB)_{ij} = \sum_{k=1}^{m} A_{ik}B_{kj} \tag{3.3}$$

which is basically achieved through two operations, multiplication and addition.

Regarding the FIR filters mentioned earlier, these can be expressed as

$$y[n] = \sum_{k=0}^{N-1} c_k x[n-k] \tag{3.4}$$

in which the N denotes the number of taps of the filter. In this case it can be seen that, once again, the operations required for the computation of the result are addition and product.

3.3.1 Parallelization

Given these two simple cases, it is noticeable that certain operations may be done concurrently. This introduces the possibility of paralellization.
To do so there is the need of establishing the order of operations and understanding dependencies between them.

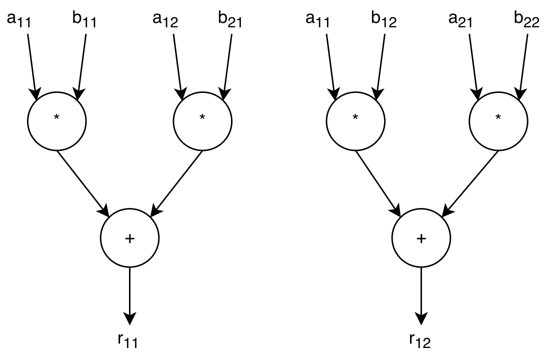


Figure 3.1: Mapping matrix multiplication

Figure 3.1 shows that the first line of the result matrix (r_{11}, r_{12}) can be calculated in parallel. In terms of hardware, if these operations were executed with combinational logic, each line would take one single clock cycle to be calculated. In terms of FIR filter, it can be seen (fig. 3.2) that, with a larger circuit (one extra addition), it is possible to calculate $y[n]$ if the filter has four taps (as many taps as the number of multiplications on the first level).

With this in mind it's possible to realize that each of these operations, or even a group of operations, can be performed in one single PE. For example, the algorithm depicted in figure 3.1

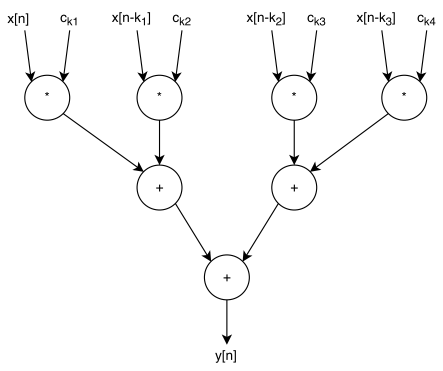


Figure 3.2: Mapping FIR filter

could be mapped to a four by two array of PEs and thus calculate the matrix multiplication using a CGRA which supports multiplication and addition. Keep in mind that if each operation has a dedicated PE in a four by two array of PEs, there will be two PEs left unused that, even not taking part in the computation of the final result, consume power. This will be addressed further as it is inserted in one of the main discussions of this dissertation.

3.4 CGRA Architecture

3.4.1 CGRA Structure

In the case of a homogeneous CGRA with four rows and four columns whose PEs support at the very least operations like multiplication and addition, besides being able to perform bypass (i.e. connect its input directly to its output), a FIR filter can easily be mapped using the previous approach. The advantage in this implementation is the fact that each PE can have a registered output which will naturally pipeline the algorithm thus allowing the frequency to be increased resulting in an increase of the throughput of the circuit comparing to the purely combinational FIR. This mapping can be achieved performing all the multiplications in the first line of the CGRA and adding all of their values two by two until the bottom of the CGRA is reached, as it can be seen in figure 3.3.

As previously mentioned there is the need of having at least multiplication and addition in order to execute the two algorithms explored in the last sections. However, including a larger set of operations would definitely result in a larger amount of options the final user could take and therefore a larger number of algorithms could be mapped.

[1]Image generated on `https://paginas.fe.up.pt/~ee12136/cgra-config/`

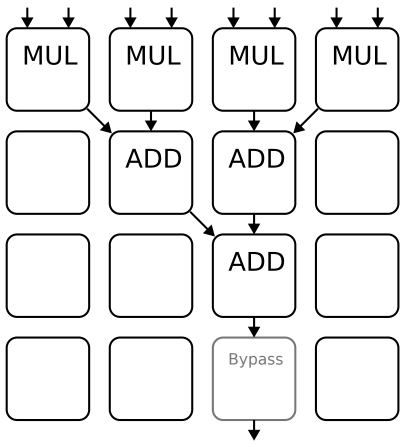


Figure 3.3: CGRA calculating FIR[1]

3.4.2 CGRA proposal for design flow evaluation

The chosen architecture of each PE consists of a set of six different operations. The chosen operations are ADD (addition), SUB (subtraction), MUL (multiplication), RSH (right-shift), LSH (left-shift) and BYP (bypass). Besides the operations, the PE has an internal configuration register which chooses from where the ALU's inputs come and which operation to perform. Each PE is able to choose, for both of its inputs independently, which signal to fetch, whether an output from one of its nearest neighbours or the value it has stored in its register which means that the PE can connect its output to any of its inputs (fig. 3.4).

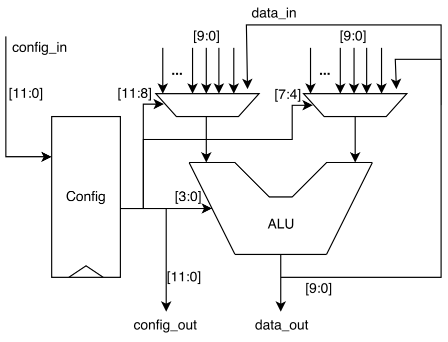


Figure 3.4: Processing Element

Each PE is connected to its eight nearest neighbours, with the exception of those that are placed at the sides that will only have five neighbours or even those that are placed at the corners that will only have three neighbours.
Since the PE code is the same for all of the elements in the array, the unused connections will be connected to ground thanks to a command given to the synthesis tool that connects all undriven

[1]Image generated on https://paginas.fe.up.pt/~ee12136/cgra-config/

signals to logic zero.

The chosen configuration method for the PEs consists of having internal registers that control the input multiplexers and choose the operation to be done by the ALU. These registers are connected in a daisy-chain fashion, having an input configuration port at one end and an output configuration port at the opposite end, as it can be seen in figure 3.5.

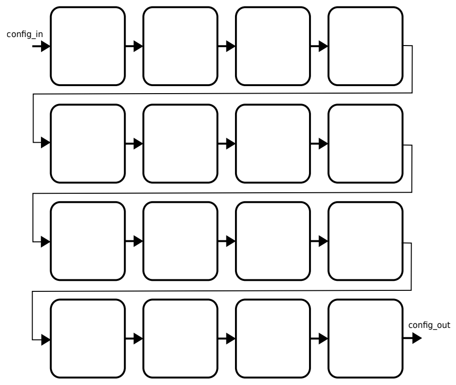


Figure 3.5: Configuration chain[2]

The proposed CGRA (code available in the webpage) consists of an homogeneous 4 by 4 array of PEs which means that once the verilog code for a PE has been written, the CGRA module only has to instance it as many times as the product of the number of columns with the number of lines.

The implemented PE has parameterized bitwidth which has a default value of 8, however, may be changed during synthesis. The array of PEs is defined inside a *generate* function which creates ALU instances and is parameterized in terms of number of columns and lines.

3.5 Validation

After establishing the architecture comes the need for validation. This can be achieved mapping a simple four-tap filter as seen in figure 3.2. In this case, the input data and filter coefficients are generated with random numbers using *Matlab* and the results are also calculated and stored in a text file so that the outputs generated on *Matlab* may be compared to the ouputs generated on the functional simulation.

After simulating and checking that the results match the expected values, it is possible to proceed to synthesis.

A running example of functional simulation is available in the webpage.

[2]Image generated on `https://paginas.fe.up.pt/~ee12136/cgra-config/`

Chapter 4

Design Flow for Low-Power CGRAs

This chapter presents a general description and proposal of a low-power design flow which will be split into two major fields. The first field is related to the front end design aimed at logical synthesis and the second field is dedicated to back end design which consists of physical synthesis. However, before tape-out, further analysis must be performed. One example is rail analysis which allows the designer to observe and study how the power rails behave. This will be interesting once the final chip is produced to perform the transient analysis of the power rails and of the power switching cells and may be done using Voltus[35]. This step, however, will not be addressed during the course of this document since this design is an IP (intellectual property) core that will be placed in a bigger design which will then be a candidate for rail analysis.
There will also be an effort to express the importance of all the steps and in what way they are connected and depend on each other.

An ASIC design starts off as a RTL description of the hardware to be implemented, which may be coded in a Hardware Description Language (HDL) such as Verilog or VHDL. After testing and validating the RTL, the code is converted to a gate-level netlist. Then, with a gate-level netlist it is possible to physically organize its elements in a design layout on which signoff analysis will be performed to evaluate functional, power and timing characteristics.

All of the mentioned steps are explained in detail in the next sections.

4.1 Low-Power Intent

The low-power intent is the specification of the low-power attributes of the design. It is possible to write a low-power intent file according to *IEEE 1801* which defines a TCL-based language that describes the low-power behaviour of each block, its power sources and connectivity and the grouping of logic into power domains. Low-power intent is present in several steps of the design flow although sometimes needs to be tuned to fulfill the tools' requirements. It is also important to know that this document addresses *IEEE 1801-2009* which is the *UPF2.0 Standard*. There is a more recent version, *UPF2.1 Standard*, which comprises the *UPF2.0 Standard*, eliminates the compatibility with *UPF1.0 Standard* and adds macros and hierarchical support. Since more information was found about the *UPF2.0 Standard* and it is compatible with *UPF2.1 Standard*, **UPF2.0 Standard** was chosen.

4.1.1 Power-intent elements

A very simplistic power intent block diagram can be seen in figure 4.1. This design, although it may be simple, contains relevant aspects like power switches, power domains and isolation cells. These elements will now be discussed in greater detail.

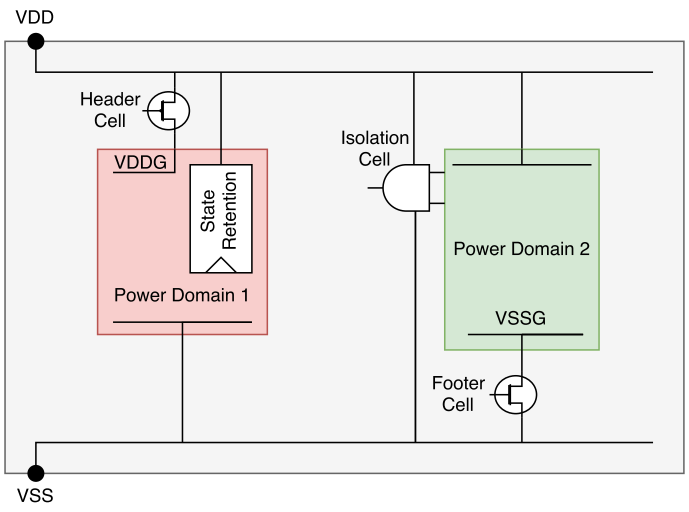


Figure 4.1: Power intent block diagram

4.1.1.1 Power switches

Power switches control the voltage in the power-gated nets, i.e., turn the nets on and off. There are two big families of power switches: header cells and footer cells. Header cells are placed as pull-up cells and connect the supply net to the power-gated supply net while footer cells, used as pull-down cells, connect the ground net to the power-gated ground net. The choice between both of these cells is up to the designer, however, [30] makes some suggestions such as choosing one of the two and never both as it increases IR drop, which is the voltage drop in the power rail due to high currents crossing a wire with finite resistance. Besides, [30] also suggests the usage of header cells whenever external power gating will be used as well as if multiple power rails and/or voltage scaling will be used on the chip so that the common net, which is the ground net, is always connected to every power domain thus providing the tools with a less error-prone power intent.

One popular method of connecting power-switching cells is called Mother/Daughter connection and it consists of having smaller switches turned on first until the rail voltage reaches 95% of its nominal value and then the bigger switches may be turned on, thus reducing IR drop since the

current spikes are less significant. To clarify, IR drop is a subject that should be addressed specially when there are peaks of current in the power nets, which would be the case when powering on a power domain.
Another way to do it is quite similar but makes use of a pin that exists in a special switch cell that tells when the power-up or power-down sequence for that same cell is stable and is used as an enable signal on the next switch, in this way gradually turning on the switch cells and reducing IR drop.
The UPF command to insert a power switch is:

```
create_power_switch SW1 \
    -domain PD1 \
    -input_supply_port { VIN1 VDD } \
    -output_supply_port { VOUT1 PD1_VDD } \
    -control_port { EN1 sleep_pd1[0] } \
    -on_state { PD1_ON VIN1 {!EN1} } \
    -off_state { PD1_OFF {EN1} }
```

where:
-domain <domain_name> specifies the power domain that will be power-gated;
-input_supply_port <alias> <port> specifies which port serves as supply input and its alias if there is the need of referring to this port later on the UPF.
-output_supply_port <alias> <port> follows the exact same logic as the previous command.
-control_port <alias> <net> specifies which net turns the switch on and off and its alias.
-on_state <alias> <input_port_alias> <expression> specifies the expression that leads to an on state.
-off_state <alias> <expression> specifies the expression that makes the switch turn to the off state.

Power switches must also be mapped to power switching cells. This is done with the following command:

```
map_power_switch SW1 \
    -domain TOP \
    -lib_cells { HEADX2 }
```

where:
-domain <domain_name> specifies the power domain in which the power switch is to be inserted.
-lib_cells <cells> specifies the technology's cells that should be used as power switches.

4.1.1.2 Isolation

Power switches, like other cells, have leakage current. This leakage current may lead to power-gated nodes that never fully discharge to ground or charge to the supply, reaching an equilibrium when the leakage current through the switches is balanced by the sub-threshold leakage of the switched cells. This may cause the outputs of the powered-down cells to float and drive corrupted values. If these outputs from powered-down cells are connected to other cells that are switched on, an even bigger loss in terms of leakage power may be provoked. To prevent this there is the need of inserting isolation cells with the only purpose of clamping the outputs to a static value, either a logic zero or a logic one, using an always-on power net to do so. In terms of isolation strategy, it is possible to isolate inputs, outputs or both. With IEEE1801 it is possible to choose if the tool should isolate only if the driver and receiver supply sets are different, if the receiver has a specific supply set or if the driver has a specific supply set. Besides these options there is also the possibility of specifying in which power domain the isolation cells are to be placed. Isolation may be inserted with the following commands on the UPF:

```
set_isolation iso_strategy1 \
    -domain PD1 \
    -isolation_signal { sleep_pd1[1] } \
    -isolation_sense high \
    -applies_to outputs \
    -clamp_value 0 \
    -isolation_supply_set { TOP_SS }
```

where:
-domain <domain_name> specifies the domain name.
-isolation_signal <signal> specifies which signal activates isolation.
-isolation_sense <high/low/posedge/negedge> specifies the sensitivity towards the isolation signal.
-applies_to <inputs|outputs|both> tells the tool where to place isolation cells.
-clamp_value <0|1|Z|latch> specifies which value should be clamped in the output. **-isolation_supply_set <supply_set_name>** specifies the supply set that powers the isolated values.

To specify the isolation cells for each isolation strategy, the following command is used:

```
map_isolation_cell iso_strategy1 \
    -domain PD1 \
    -lib_cells { ISOLANDX2 }
```

where:
-domain <domain_name> specifies in which domain the isolation cell is to be placed.
-lib_cells <cells> specifies the technology's cells that should be used for clamping.

4.1.1.3 Retention cells

One disadvantage of powering down a circuit is the fact that the information in every register is lost or corrupted. If there is an explicit need or advantage in keeping the values of the registers after powering down the circuit, the designer must include state retention cells that make use of an always-on power net to retain the stored values even when the main net is off. Retention strategies are specified in the following manner:

```
set_retention ret_strategy1 \
    -domain PD1 \
    -retention_supply_set TOP_SS \
    -restore_signal {{sleep_pd1[2]} negedge} \
    -save_signal {{sleep_pd1[2]} posedge}
```

where:

-domain <domain_name> specifies the domain which will be the target of the retention strategy.
-isolation_signal <signal> specifies the signal which controls the activation of the isolation strategy.
-isolation_sense <high/low/posedge/negedge> specifies whether to activate the control strategy when the signal is high, low or at any edge.
-isolation_supply_set <supply_set_name> specifies the supply set that will be used by the isolation cell.

And the retention cells are mapped with the command:

```
map_retention_cell ret_strategy1 -domain PD1 -lib_cells RDFFSRX1
```

where:

-domain <domain_name> specifies the domain in which to place retention cells.
-lib_cells <cells> specifies the technology's cells that should be used for retention.

4.1.1.4 Power domains and power nets

The elements mentioned previously require the existence of power domains and power nets. The power domains specify a set of low-power properties that are common to a certain group of cells. Each power domain contains the previously mentioned elements plus the supply nets and the supply sets (which are groups of supply nets). Power nets can be created and associated to power sets with the following commands:

```
create_supply_net VDD
create_supply_net VSS
create_supply_set TOP_SS \
    -function { power VDD } \
    -function { ground VSS }
```

where:

-function specifies the net and if it's a power or ground net.

And the power domain may be created with the command:

```
create_power_domain PD1 \
    -elements { {pe_array1/H[0].V[0].ALU.PE}
    -supply { primary PD1_SS }
```

where:

Power state	TOP_SS	PD1_SS	PD2_SS	PD3_SS	PD4_SS
PS1	high	high	high	high	high
PS2	high	high	high	high	off
PS3	high	high	high	off	high
PS4	high	high	high	off	off
PS5	high	high	off	high	high
PS6	high	high	off	high	off
PS7	high	high	off	off	high
PS8	high	high	off	off	off
PS9	high	off	high	high	high
PS10	high	off	high	high	off
PS11	high	off	high	off	high
PS12	high	off	high	off	off
PS13	high	off	off	high	high
PS14	high	off	off	high	off
PS15	high	off	off	off	high
PS16	high	off	off	off	off

Table 4.1: Power states definition

-elements specifies which elements of the HDL/netlist belong to the power domain.
-supply specifies the supply set of the power domain.

4.1.1.5 Power states

The *UPF2.0 Standard* requires the definition of power states. Power states define the combination of possible states of the power supplies. As an example, table 4.1 shows the possible states of the power supplies of the implemented 4 by 4 CGRA with 4 power domains. In the context of the document PD stands for Power Domain and SS stands for supply set. This means PD1_SS is the supply set associated with power domain 1. With these 16 (2^4) states it is possible to independently switch on and off every power domain. To define this table in a UPF file one must first specify the power states that each supply set may take with the following command:

```
add_power_state PD1_SS -state high { \
    -supply_expr { PD1_VDD == ‘{FULL_ON, 1.2} && VSS == ‘{FULL_ON, 0.0} } }
add_power_state PD1_SS -state off { \
    -supply_expr { PD1_VDD == ‘{OFF} && VSS == ‘{FULL_ON, 0.0} } \
    -simstate CORRUPT}
```

where:
-supply_expr <expression> specifies the status of each power net that belongs to the power set and
-state <name> specifies the name of this combination of power net values.
-simstate <state> specifies the state of the values stored in the registers that belong to the power domain. The possible states are: NORMAL, CORRUPT_ON_CHANGE, CORRUPT_STATE_ON_CHANGE,

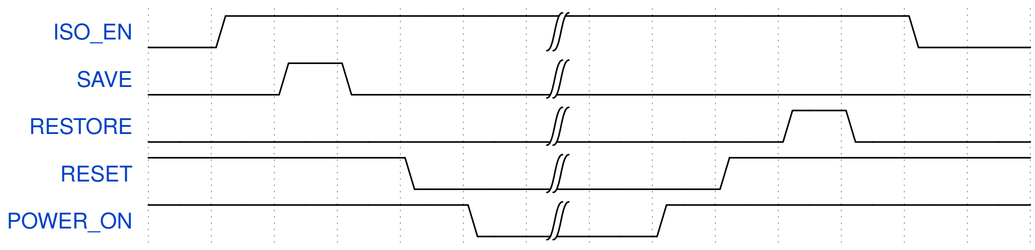


Figure 4.2: Power controller waveforms

CORRUPT_STATE_ON_ACTIVITY, CORRUPT_ON_ACTIVITY, CORRUPT and NOT_NORMAL.

Having defined the power states for each supply set it is then possible to define power states for the chip itself with the following commands:

```
add_power_state TOP -state P1 { \
    -logic_expr { TOP_SS == high && \
        PD1_SS == high && \
        PD2_SS == high && \
        PD3_SS == high && \
        PD4_SS == high } }
add_power_state TOP -state P2 { \
        -logic_expr { TOP_SS == high && \
        PD1_SS == high && \
        PD2_SS == high && \
        PD3_SS == high && \
        PD4_SS == off } }
```

where:
-supply_expr <expression> specifies the status of each power set.
-state <state> specifies the name of the power state.

4.1.2 Power controller

When dealing with power domains with isolation and state retention, there is an order of commands that should be kept. As an example, if the values are saved to state retention cells when the power is already off, the retention cells will store corrupted data.
Waveforms of a robust power controller with state retention and isolation are depicted in figure 4.2.

Figure 4.2 shows that the correct order to isolate and retain values when turning off a power domain.

1. Assertion of the isolation (ISOEN) signal.
2. After isolation has taken place, the save signal (SAVE) should be asserted and de-asserted

once the values are stored.

3. Once the retention cells have saved the relevant data, the reset (RESET) signal should be asserted so that, when the system turns back on, a clean start is achieved.

4. With the isolation and retention strategies activated, it is then possible to de-assert the power signal (POWERON) so that the power net is finally turned off.

The opposite case (turning on a power domain) is achieved by:

1. Assertion of the power signal (POWERON).
2. De-assertion of the reset (RESET) signal.
3. At this stage the data may be recovered with the RESTORE signal.
4. Finally, with the power domain turned on and the data recovered from the retention cells, the isolation signal may be de-asserted.

4.1.3 Low-power intent of a CGRA

In a CGRA there are PEs that are not used during certain algorithms. Those PEs can be turned off in order to save power. However, having power domains that only contain a single PE instance can lead to an increase in area that overshadows the savings achieved by turning off the PE.

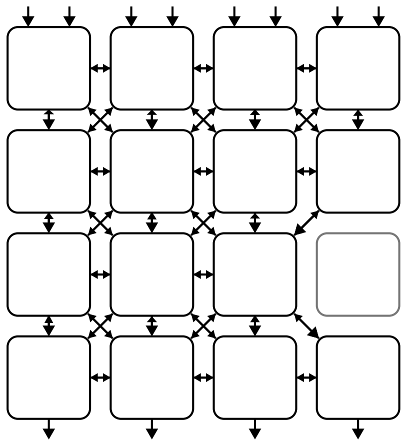


Figure 4.3: Turning off a single PE[1]

Grouping PEs into power domains, with a priori knowledge of the algorithms to be mapped on the CGRA, may lead to higher efficiency. In the design proposed in this dissertation, data are driven from the top of the CGRA and the natural processing flow is vertical downwards, meaning that turning off one PE instance either increases the data flow in its neighbours or, in the worst case, renders the whole column useless. This example is depicted in figure 4.3, where the light-grey square represents a powered-down PE.

For this reason it was decided to create power domains that group every element of every column, i.e., each columns of the CGRA represents a different power domain. Besides this, when turning off a whole column, the input to output ratio is kept, which is the ideal situation when

[1]Image generated on https://paginas.fe.up.pt/~ee12136/cgra-config/

scaling algorithms. Figure 4.4 shows the CGRA with established power domains. Each colour corresponds to a different power domain.

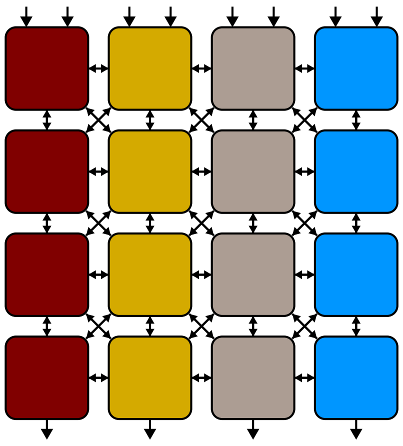


Figure 4.4: Power domains of a 4 by 4 CGRA[2]

4.2 Front end

Front end design is focused on logic functionality and timing constraints. This section describes the steps involved in front end design with its focus set on low-power.

4.2.1 Front end relevant files

A front end design starts with the design of the architecture and follows with the synthesis process during which it is possible to evaluate the design and its viability to proceed to the next design stages. To synthesize a design and produce values and files that are as close to the reality as possible, the designer must provide the synthesis tool with relevant information. The first and most obvious input that should be fed to the synthesis tool is the HDL code which, in the case of the design proposed in the scope of this document, is written in verilog.
In order to generate a netlist, the tool needs to know which cells are available for mapping and those cells are imported via a liberty (.lib) file that is usually provided by the foundry which contains information about the cells in terms of timing and power and information about the its pins and functionality.
An UPF file, explained in section 4.1, should also be loaded in order to specify the low-power intent of the design. Since synthesis tools work on optimizing a given design around a set of given constraints, it is also important to import these constraints, which come in a Synopsys Design Constraints[4.4.3] (.sdc) file.
During front end design, three major software tools will be used. To simulate the design's functionality with a testbench, the chosen tool is Incisive[36]. After having a validated design, the tool used for synthesis is Genus[37]. Finally, formal verification is achieved using Conformal[38].

Besides regular design synthesis, Genus[37] also supports PLE (Physical Layout Estimation) which takes into consideration information about the physical implementation of the design. This is used when the design has gone through the entire design flow at least once and requires inputs like a cap table, tech LEF and Standard Cell LEF libraries and/or a QRC tech file (these files will be explained in section 4.3). PLE is capable of providing the designer with results taken from the synthesis that show higher correlation to those that can be extracted after place and route.
At the end of front end design it is possible to extract timing, power and utilization reports, together with a set of files useful to the place and route tool. The latter ones may be exported from Genus[37] with the command:

```
write_design $DESIGN -basename ./outputs/$DESIGN/cgra_synthesized -innovus
```

4.2.2 Standard design flow

A standard design flow starts with reading the libraries which contain the cells to be mapped and also with reading the HDL to be synthesized. At the end of the flow a netlist file is created which can then be fed to the next stage, i.e., back end design.
Figure 4.5 shows a proposal of a low power design flow. Everything starts off with reading the

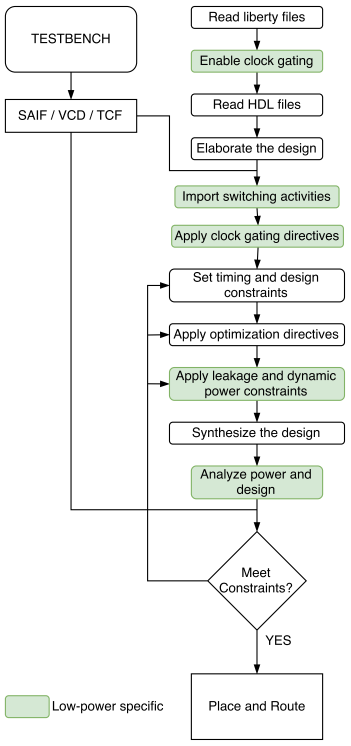


Figure 4.5: Low power design flow. Adapted:[39]

liberty files. Liberty files contain information about the power and timing characteristics of every cell under several operating conditions (like voltage and temperature) and are used to calculate

[2]Image generated on https://paginas.fe.up.pt/~ee12136/cgra-config/

both timing and power of the circuit after synthesis. After reading the liberty files it is possible to enable clock gating, which consists in reducing dynamic power by not feeding unnecessary clock cycles to the logic circuits thus reducing activity. This step, however, is not applied to this project as the main focus is to assess power saving features in the static power domain.

After importing the aforementioned files, the design may be elaborated. The elaboration stage takes the HDL and maps the design to hardware structures. High-level optimizations also take place, such as dead code removal, and registers are inferred for the design. At this stage it is also possible to override the design's parameters.
After elaboration it is good practice to check the design to search for problems like unresolved references or undriven ports, among others. After ensuring that the design has no errors, the next step is to map the HDL to the technology cells that will actually be implemented and are described in the liberty files. However, before actually doing so, the designer should include timing and design constraints, like a description of the clock waveform or even of the cells that drive the inputs of the circuit or that are driven by its outputs. These constraints are specified in a SDC file and its content is explain in section 4.4.3.
After loading the constraints it is possible to apply optimization and clock-gating directives. Then, the design may be synthesised and analysed for power and timing values. This is an iterative process, meaning that if the design does not fulfill the desired specifications and constraints, the designer must go back in the design flow in order to either change the constraints and optimization directives or, as a last resort, change the HDL architecture.

4.2.3 Equivalency

A synthesised design does not always correspond to the HDL description of the same design. This happens because optimization takes place and some slight changes may occur. The synthesis tool can add, delete or change several instances if it makes sense regarding the optimization constraints. So, in order to verify if the design maintains functionality it is possible to use a tool (in this case Cadence Conformal[38]) that checks the equivalence between the netlist and the HDL and even between the user-written UPF and the UPF obtained after synthesis.

4.2.4 Post-synthesis simulation

Once the design is synthesized and its equivalence is checked it is unlikely that functionality has been lost. However, if the designer wishes to make sure that the circuit keeps its functionality by running the testbench, it is possible to do so. The only difference to functional simulation is the fact that now the verilog files that will be simulated contain a netlist which contains instances of cells that are not defined in that same file. Thus, in order to simulate the post-synthesis netlist, there is the need of including the verilog files of the library so that they are compiled by the simulator before the netlist.

Please note that equivalency checking is further explained in the webpage and a script is also provided.

4.3 Back end

It is possible to import the gate-level netlist to a physical design tool and, together with the relevant files, produce a physical design of the final chip. In order to do so there is the need of understanding the back end design flow. The physical design flow can be divided into 6 relevant parts:

- Partitioning
- Floorplanning
- Placement
- Clock Tree Synthesis
- Signal Routing
- Parasitic Extraction

These items will be discussed in the following subsections.

4.3.1 Partitioning

Nowadays designs are extremely big to be handled by a single engineer and/or a single computer. Partitioning the design helps to reduce data size and allows parallelism in the implementation, both in hardware and human resources. In sum, a partitioned design and implementing it physically in a hierarchical fashion allows for better exploitation of resources together with tighter control of the design blocks since the designs become smaller and with better defined boundaries. The alternative to an hierarchical partitioned design is the flattened design in which the whole circuit is treated as a single instance.

4.3.2 Floorplanning

Floorplanning is the step during which the layout begins to be planned. In this stage the designer may choose to keep elements with frequent communication near each other. In some way this step resembles the floorplanning of a house in which, for example, the kitchen is kept near the dining room[41]. In a low-power design flow the process is quite similar to the ordinary one with the exception of the power domains. If a power-intent is loaded, it is possible to establish the boundaries and placement of the power domains inside the chip's core. It is possible to include physical constraints, like placement or routing blockages or even to define fence spacing. This will be mentioned further ahead when the power rings of the power domains are inserted.

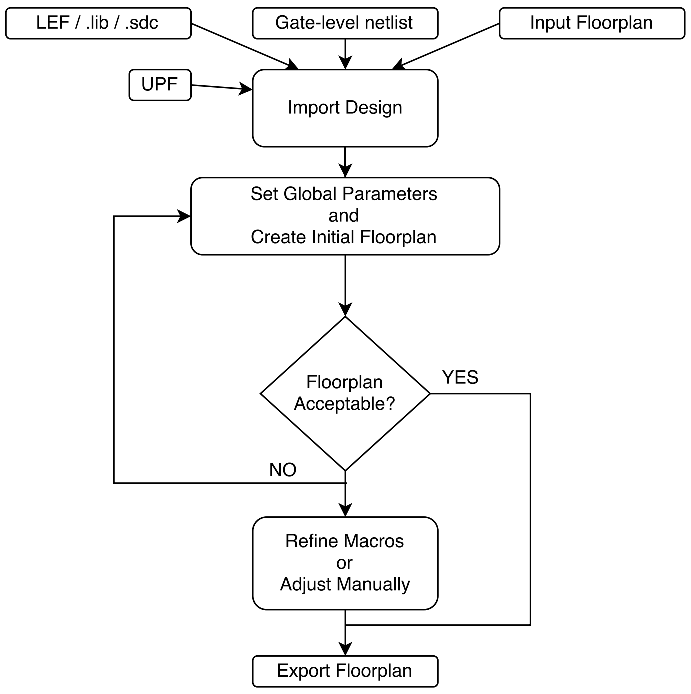


Figure 4.6: Typical floorplan flow. Adapted:[40]

4.3.3 Placement

After having a planned design in terms of soft and hard macros the cells may be placed. In this stage, the tool will place the cells. Optimization will take place with iterative placement of the cells in order to improve timing and power results.

4.3.4 Clock Tree Synthesis

The clock signal is one of the signals, and in many cases the signal, that has the highest fanout. A great part of the design, like for example every flip-flop, needs a clock signal. This makes the clock signal one of the most important signals in the whole design and also a signal that requires a great amount of power.
Clock Tree Synthesis (CTS) is thus a very important step in a physical design. The clock tree must be balanced to ensure that the clock signal is delivered to every cell with minimum skew and jitter. To accomplish this the tool places clock buffers and connects the clock cells in a tree fashion (hence the name clock tree) with the objective of delivering approximately the same clock signal to every cell in the design.

4.3.5 Signal Routing

For signal routing the tool makes use of the information available about timing arcs of the cells and estimates wire delays for every interconnection.

4.3.6 Parasitic extraction and timing closure

To perform timing analysis on a final layout it is possible to extract information about the interconnections to have a more faithful timing estimate. This extraction is made by reading the resistance and capacity values of every metal layer and, taking into account the connection's length and width, an RC value is then calculated.
Extracting data about parasitics requires a cap table which contains information about capacitance values of the circuit such as coupling capacitance. To generate a cap table, the following command is used:

generateCapTbl -ict <ict_path> -output <output_path> -lef <tech_lef> -shrinkFactor <integer>

where:
-ict <ict_path> specifies the ICT files which is either provided by the foundry or obtained from the Interconnect Technology File (ITF) file which contains information about conductor, dielectric and via layers.
-output <output_path> specifies the path for the output file.
-lef <tech_lef> specifies the path to the Technology Layout Exchange File (LEF) which contains physical information of the used technology.
-shrinkFactor <integer> allows the designer to specify a compensation factor if gate length shrinking has taken place.

After generating the cap table, it is possible to extract parasitics. To do so, the used command is:

extractRC

This will be helpful when the need of creating a Standard Delay Format (SDF) file surges, as it will be mentioned further ahead.

4.3.7 Proposal of back end flow for CGRAs

Since a description of the main steps in back end design flow has been given, the proposed back end design flow will be exposed and explained while making use of running examples of the application of the flow to a synthesized CGRA that resulted from section 4.2. As previously mentioned, the design starts as an HDL netlist. The software used for floorplanning during the scope of this dissertation is Innovus[42] and it is launched from the console by typing the command *innovus*. It can be used either with or without a Graphical User Interface (GUI). However, since this is a physical implementation and for demonstration purposes the GUI will be useful to prove points and demonstrate concepts. Innovus supports back end design from design import to the final chip's routing. With Innovus it is possible to execute the design flows depicted in figures 4.5 and 4.6.

A Tcl script is available at the dissertation's webpage, however, the steps will be explained in the course of this document. The script starts off by preparing the layout in terms of initial files

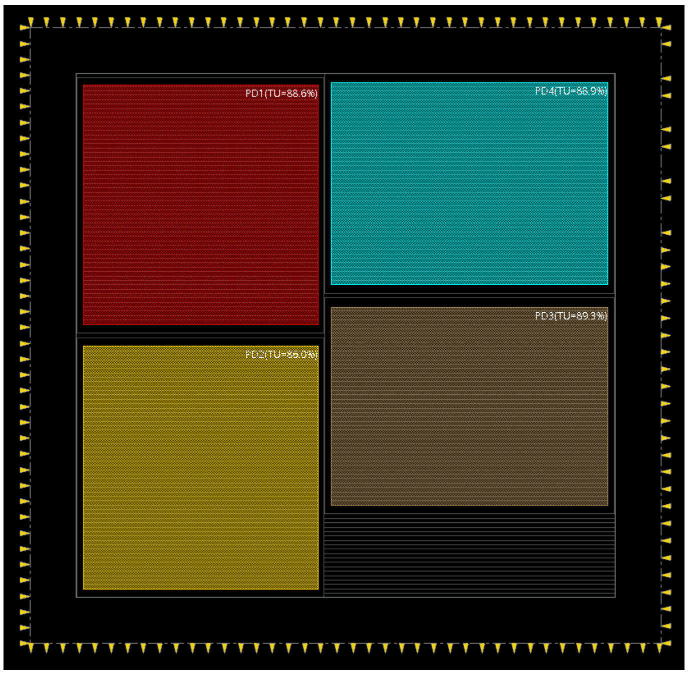


Figure 4.7: Planning the design

and configurations. The default power and ground nets are established. Liberty and LEF files should also be imported before the netlist as they contain physical information (LEF) and timing information (Liberty) about the cells mapped during synthesis.
Having imported the library related files it is time to load the netlist. Once loaded, the design's macro blocks are already imported and visible in the layout window.
The next step is to define the core and its boundaries. The objective of this step is to have a core area capable of housing all of the elements in the design and also to have a boundary reserved for power rings that should be wide and contour the entire core.

Once the floorplan is well defined and there is enough space to move things around it is possible to read and commit the power intent file. Once committed, it will be possible to see macro blocks defining the power domains and these can be manually or automatically placed in the core. In parallel it is possible to load an IO file or even to manually place the IO pins. (fig. 4.7).

In the case of this particular design, each power domain has its own power net. About those nets it is possible to choose whether to place them around the core, parallel to the top domain's power nets or to place them around each of the power domains. Since this is a four domain design, that would mean that, for the first option, the core would have a great deal of area overhead just in terms of power rings. Because of this it was chosen to modify the power domains so they'd include a reserved space between the placement of the cells and the fence of the macro block in order to have supply rings around each domain. Together with the power rings, power stripes are also be added (fig. 4.8).

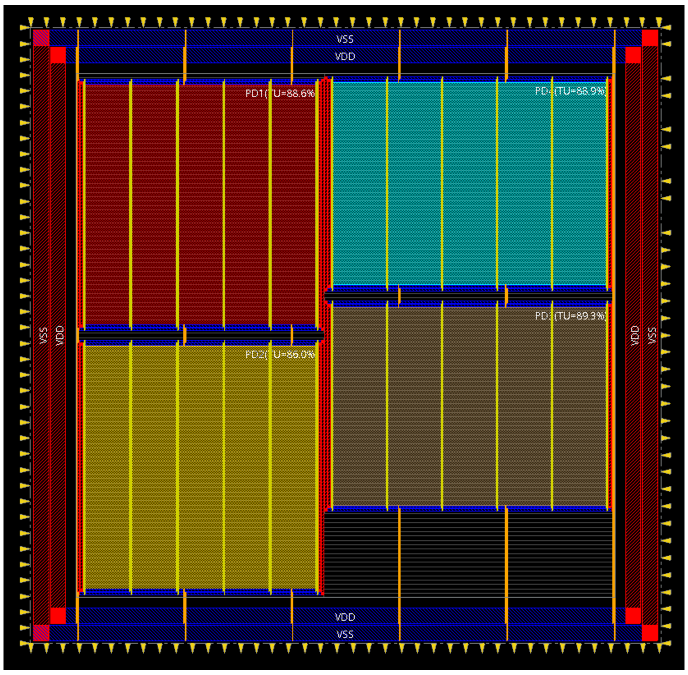


Figure 4.8: Power rings and stripes

Having the pins, the pads and the domain boundaries well established it is possible to plan the design and place all of the standard cells in their power domains.

Together with this placement, the tool already does an estimation of the interconnections so that the user can already have a perspective of what the final design will look like.

This placement of the design, in the Innovus [42] tool, already performs CTS. Now that the design is placed and CTS is performed there are still gaps between cells and power nets have not yet been routed. To fix this issue, filler cells will be placed in every domain. After placing the filler cells there is the need of specifying all of the connections between nets and pins in order to do power routing. Header cells are more laborious since each cell has three power pins and the tool only routes two of them automatically, leaving it up to the designer to route the remaining power pin as if it were a signal net. After setting up all of these connections it is possible to perform a special route which will result in a floorplanned design with power connections, missing only the interconnections between the cells. To end this stage, a command is given to route the cells which will perform several iterations and can target low-power or fast circuits, making use of both the timing arcs of the cells and the length and width of the connecting wires. This whole process results in a placed and routed chip as it can be seen in figure 4.9. The next step towards tape-out is signoff which will be addressed in the next section. However, some data should be exported from the Innovus[42] tool and imported to signoff tools for better accuracy of the results. Relevant files:

- Design Exchange Format (DEF) - these files contain information about the physical placement of the cells.
- Library Exchange Format (LEF) - these files contain physical information about individual cells and, complemented with DEF, provide a physical representation of the chip.
- Standard Delay Format (SDF) - contain information about the chip's internal delays combining the cells delays with the interconnect delays.
- Cap Table - specifies capacitance values such as coupling capacitance.

Furthermore, the cap table may be used to perform parasitic extraction which is the annotation of resistance and capacitance values of the circuit. This is important information when calculating realistic circuit delays.

4.4 Validation and Signoff

Signoff checks on the CGRA are performed using Innovus[42], Incisive[36], Conformal[38], Voltus[35] and Tempus[43].
Innovus, the place and route tool, already allows for DRC (Design Rule Check) and LVS (Layout Vs Schematic) checks.
These two checks include:

LVS:
- Open-circuit
- Short-circuit
- Different number of ports
- Connectivity error

etc.

DRC:
- Mininum spacing error
- Metal loop violation

etc.

4.4.1 Formal verification

Formal verification is achieved by comparing pre and post-route netlists and power intent files on Conformal[38]. If both netlists and power intent files match it is then possible to run the testbench with the final netlist and generate a VCD (Value Change Dump) which contains the activity of the nets. This file may be used to extract power reports for specific algorithms or behaviours. It is also possible to run the simulation with all extra information about net and cell delay if a SDF (Standard Delay Format) file is included. This file may be obtained from Innovus. To do so, a

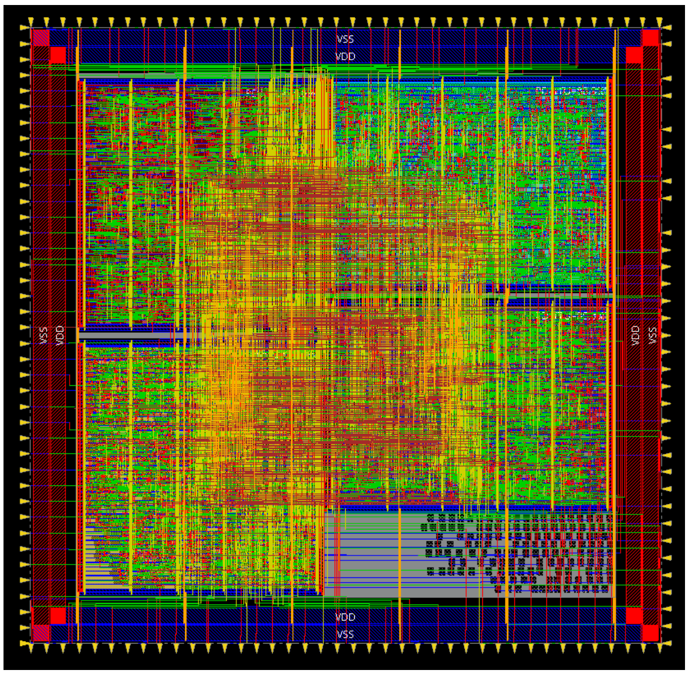


Figure 4.9: Implementation of a 4 by 4 CGRA
with 4 power domains

cap table, which contains information about coupling, fringe and area capacitance values, must be created and, after that, RC extraction should take place. Please note that this is further explained in the webpage.

4.4.2 Post-route simulation

After place and route it is possible to generate a verilog netlist of the final chip which now contains information about power supplies and also the header cells. With this netlist and the verilog files of the library cells, the designer may run the testbench in order to verify if functionality is maintained. Besides these files, there is the possibility of obtaining a more realistic simulation of the circuit if the cell and net delays are annotated. To do so, a SDF file must be imported to the simulator. This simulation is achieved with Incisive [36] and a running example may be found in the webpage.

4.4.3 Static timing analysis

Static Timing Analysis (STA) allows the timing of a circuit to be computed without requiring a simulation of the entire circuit. The STA tool computes the sum of the cell delays and the path delays in a given path of a circuit and compares the result with the timing characteristics specified in a Synopsys Design Constraints (SDC) file, which will be explained below.

One of the first things to specify in a SDC file is the clock signal which is achieved with the following command:

create_clock -name <clock_name> -period <value> -waveform <edge_list>

where:

-name <clock_name> specifies the name of the clock signal.

-period <value> specifies the clock period.

-waveform <edge_list> List of edge values.

At this stage the clock has an ideal waveform. If the designer wishes to have a better approach to the real clock signal, the following commands may be used, replacing "<*signal*>" for "*[find / -clock <clock_name>]*":

set_attribute slew_rise <value> <signal>

set_attribute slew_fall <value> <signal>

where:

<value> specifies the slew time.

<signal> specifies the targeted signal.

Another property that can be modelled in a signal is clock uncertainty which can be specified for both setup and hold with the commands:

set_clock_uncertainty -setup <value> [get_clocks <clock_name>]

set_clock_uncertainty -hold <value> [get_clocks <clock_name>]

where:

<value> specifies the uncertainty time.

<clock_name> specifies the targeted clock signal.

After modelling the clock signal it is possible to model the environment in which the chip is inserted. It is possible to constraint the data arrival time of a signal relative to the clock signal for both inputs and outputs of the circuit. The following command specifies the input signal delay:

set_input_delay -clock <name> <delay> <targets>

where:

-clock <name> specifies the clock name.

<delay> specifies the delay value.

<targets> specifies the inputs.

For the output signals, the command is *set_output_delay* and has the same attributes as *set_input_delay*.

After specifying constraints for the input and output signals, the SDC file may be complemented with environmental information. One example are the external drivers i.e. the cells that drive the input pins. These cells can be specified with the command:

set_attribute external_driver [find [find <design> -libcell <cell_name>] -libpin <pin>] <inputs>

where:

[find [find <design> -libcell <cell_name>] -libpin <pin>] locates the pin <pin> of the cell <cell_name>.

<inputs> specifies which inputs have this driving cell.

Likewise, it is possible to specify output loads with the following commands:

set output_load [get_attribute capacitance [find /libraries/saed90nm_typ/ -libpin INVX4/INP]]

set_attribute external_pin_cap ${output_load} /designs/${DESIGN}/ports_out/*

4.4.4 Multi-Mode Multi-Corner Analysis

Modern designs have typically have more that one clock signal, several supply voltages, timings constraints and libraries. For example, if a circuit has more than one voltage level, there may be an interest in decreasing or increasing the clock frequency. This leads to a different set of timing constraints. In a single-mode analysis, the chip is analysed for a single set of characteristics. Facing this limitation, multi-mode multi-corner (MMMC) analysis has been introduced and it gives the designer the advantage of defining several analysis views for the same chip with each view being defined as a set of clocks, supply voltages, timing constraints and libraries.

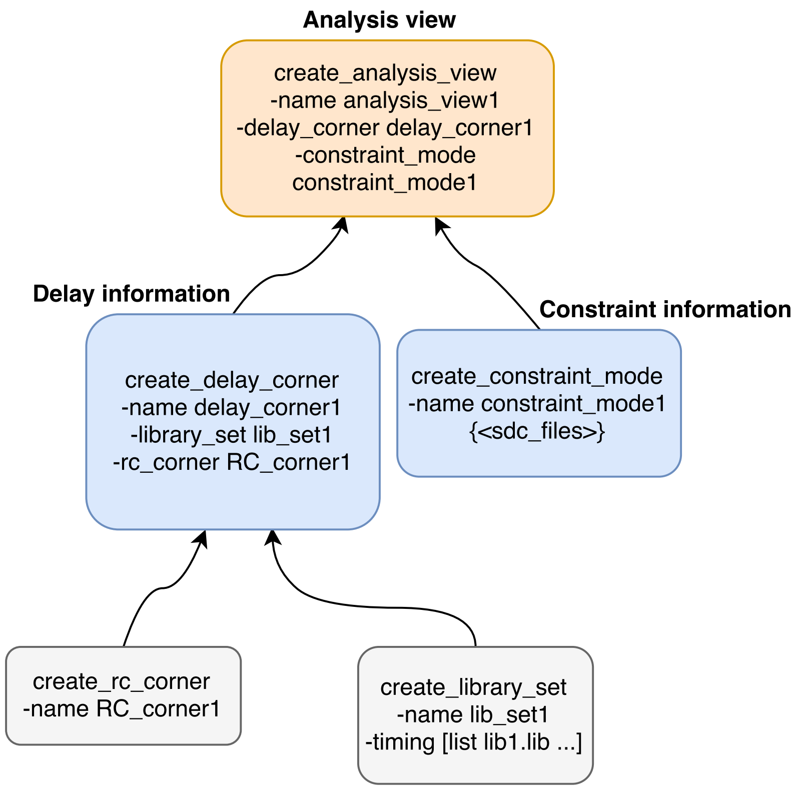

Analysis view
create_analysis_view -name analysis_view1 -delay_corner delay_corner1 -constraint_mode constraint_mode1
Delay information
Constraint information
create_delay_corner -name delay_corner1 -library_set lib_set1 -rc_corner RC_corner1
create_constraint_mode -name constraint_mode1 {<sdc_files>}
create_rc_corner -name RC_corner1
create_library_set -name lib_set1 -timing [list lib1.lib ...]

Figure 4.10: MMMC hierarchy. Adapted:[43]

As depicted in figure 4.10, to create an analysis view there is the need of previously importing the SDC files (supplying constraint information) and delay corner information. In the latter case there is also the need of previously defining RC corners and library sets.

4.4.5 Tempus typical flow

The Tempus[43] timing analysis tool supports MMMC analysis and its typical design flow is depicted in figure 4.11. There is a running example of the design flow depicted in figure 4.11 in the webpage.
The most important steps while using Tempus[43] are the ones in which the design and its features are imported. Some of the features, like SDF files which contain information about parasitics or even DEF files which contain physical placement information, are not requires for the tool to perform timing analysis. However, once these optional files have been imported, the results of the timing analysis are far more accurate and close to the real values.

4.4.6 Power consumption estimation

Power consumption may be estimated using Voltus[35]. Once the design has been imported to Voltus, it is possible to perform power analysis by specifying a toggling percentage for each net. However, if the designer wishes to assess the power consumption of a given algorithm, for example, it is possible to generate a VCD file from simulation and import it to Voltus thus annotating the real net activity.

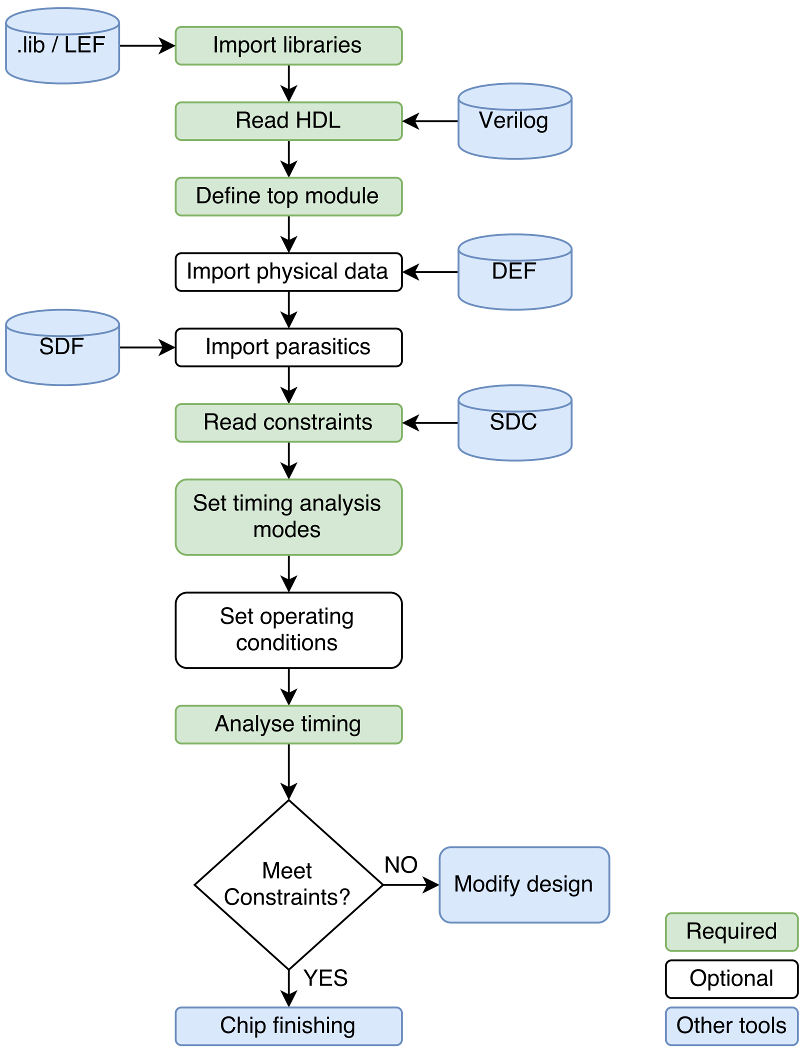


Figure 4.11: Tempus typical flow. Adapted:[43]

Chapter 5

Design Flow Scripts

Having a more pragmatic view of the design flow detailed in 4, this chapter explains the content of the scripts written during the course of this project and how they should be used during the design stages.

5.1 Frontend

To accomplish frontend design of a CGRA, Genus must be invoked from a console. After invoking the program, the frontend script should be executed by typing the command **source <script_name>.tcl**. The script, which can be found in appendix, starts by setting up the tool for low-power design processing and creating a folder in which to save the outputs:

```
set enable_ieee_1801_support 1
set systemTime [clock seconds]
set currTime [clock format $systemTime -format %a%d%B%Y_][clock format\
 $systemTime -format %H-%M]
set REPORTS ../run/reports_${currTime}
file mkdir ${REPORTS}
```

After setting up the environment and before importing the HDL there is the need of providing the tool with the paths where the libraries may be found, which is done using the following commands:

```
create_library_domain {saed90nm_typ}
set_attr library {/home/diogo/Documents/SAED-EDK90/\
SAED90_EDK/SAED_EDK90nm/Digital_Standard_cell_Library/\
synopsys/models/saed90nm_typ.lib} saed90nm_typ
#Change the paths when using different .lib files

set_attr lef_library { \
/home/diogo/Documents/SAED-EDK90/SAED90_EDK/SAED_EDK90nm/\
Digital_Standard_cell_Library/lef/saed90nm_tech.lef \
/home/diogo/Documents/SAED-EDK90/SAED90_EDK/SAED_EDK90nm/\
Digital_Standard_cell_Library/lef/saed90nm.lef \
} #Change the paths when using different LEF files
```

If the designer wishes to adapt the design flow to other libraries, this is where changes should be made. The snippet of code above shows how library domains are created, in this case creating the library *saed90nm_typ* which is only an alias. However, the path to the ".lib" file should correspond to the path where the designer placed the libraries to be used. The command **check_library** should be invoked in order to check for any errors in the libraries before moving forward. At this stage it is possible to read the HDL files together with the power intent file and elaborating the design using as seen below:

```
read_hdl {        ../src/bottom_alu.v \
../src/top_alu.v \
../src/alu.v \
../src/pe_array.v \
../src/cgra.v \
} #Change the paths when using different HDL files

set bitwidth 10
set rows 4
set columns 4
set DESIGN cgra
set DESIGN ${DESIGN}_bitwidth${bitwidth}_rows${rows}\
_columns${columns}
read_power_intent ../synth_scripts/powerIntent.upf\
 -1801 -module ${DESIGN} -version 2.0
elaborate cgra -parameters {10 4 4}
```

The commands seen above specify where the HDL and UPF files may be found and elaborate the design with specified parameters. Still regarding UPF it is relevant to mention that **a UPF-generating Perl script for CGRAs has been written and may be found in the webpage**. Having the design elaborated, it is possible to load the timing contraints and proceed to the synthesis stage. This is achieved by the following set of commands:

```
source ../synth_scripts/constraints.sdc
apply_power_intent
commit_power_intent

check_design #Check the design for errors

syn_generic  #Synthesise to generic gates
syn_map      #Synthesise to library cells
syn_opt      #Optimises the synthesised design
```

After successfully synthesising the design, reports may be generated. To move to the backend flow, the design must be exported and Genus allows output files relevant to Innovus to be generated. To do so, the following commands are invoked:

```
report_timing > ${REPORTS}/timing.rep
report_gates  > ${REPORTS}/cell.rep
report_power  > ${REPORTS}/power.rep
write_design ${DESIGN} -basename ../outputs/innovus/${DESIGN}_${currTime}/synt\
h_cgra -innovus
```

Having the outputs of the synthesised design generated, it is possible to run Conformal to perform formal verification. To do so, start the software and source a script as the one seen here:

```
#Configuring Conformal low-power
set lowpower option -native_1801
set lowpower option -golden_analysis_style PRE_SYN -revised_analysis_style
POST_SYN

#Importing libraries and both versions of the design
#To use a different library, change the paths
read library /home/diogo/Documents/SAED-EDK90/\
SAED90_EDK/SAED_EDK90nm/Digital_Standard_cell_Library/\
synopsys/models/saed90nm_typ.lib

#The following command should be changed to point to
#the Verilog files which the designer wishes to read
read design { \
../src/bottom_alu.v \
../src/top_alu.v \
../src/alu.v \
../src/pe_array.v \
../src/cgra.v \
} -golden
read design ../outputs/innovus/${DESIGN}/${DESIGN}.v -revised

#Importing both versions of the low-power intent
read power intent ../synth_scripts/powerIntent.upf -1801 -golden
read power intent ../outputs/innovus/${DESIGN}/${DESIGN}.upf -1801 -revised

#Comparing netlists and UPFs
compare power intent
report compared power intent

compare power consistency
report compared consistency
```

If the reports show no anomalies, the designer may proceed to backend design.

5.2 Backend

The backend scripts are more extensive than the scripts shown before thus will be available both as appendix and from the webpage. In this document only the most important steps are shown.

This stage of the design is achieved using Innovus. After setting up the environment and the paths for relevant files, the first step is to define the area of the floorplan which, in this case, is done with the command "**floorPlan -site unit -r 1 0.7 30 30 30 30 -dieSizeByIoHeight max**" where the height to width ratio is specified (1) together with the total utilization percentage of the core (0.7) and the space between the core and the pins where power rings will exist. It is important to clarify that the utilization percentage of the core is set to 0.7 so that the tool has enough space to place every cell and interconnections without causing placement/wiring errors. After defining the floorplan, the power intent must be imported in order to insert the header cells which are not explicit in the post-synthesis netlist. However, Innovus does not do a good job while placing header cells as it occasionally places cells on top of each other or even outside the core's boundaries. To solve this issue, the proposed method is the placement of header cells followed by the deletion of their objects from the design core so that they become known to the tool, however, are not physically placed (will be placed later together with all the cells of the design that are unplaced). A snippet of code below clarifies the process:

```
read_power_intent -1801 ${power_intent_files}
commit_power_intent
#Place the power domains
planDesign
#Add power switches
addPowerSwitch -ring -powerDomain PD1 -topSide 1
#(...) Repeat the command above for each power domain
#Unplace switches
deleteAllFPObjects
#deleteAllFPObjects deletes the power intent information
#thus it must be inserted again
read_power_intent -1801 ${power_intent_files}
commit_power_intent
#Modify the properties of the power domains in order to have
#empty space around the power domains where the power rings
#for each domain will be placed
modifyPowerDomainAttr TOP -rsExts {4 4 4 4}
modifyPowerDomainAttr PD1 -minGaps {4 4 4 4} -rsExts {4 4 4 4}
#(...) Repeat the command above for each power domain
#Place the power domains once again
setPlanDesignMode -useGuideBoundary fence -effort high\
 -incremental false -boundaryPlace true -fixPlacedMacros\
  false -noColorize false -fenceSpacing 5
planDesign
```

At this stage the power domains are placed and the designer may move or resize them. After doing so, power rings and power stripes must be inserted and uses the commands "**addRing**" and "**addStripe**". After having the power nets in place it becomes possible to place the standard cells in the design by invoking the "**placeDesign**" command. This is followed by the connection of power pins to power nets. Below there is an example of all of the power connection s for a single power domain, which must be repeated for all domains.

```
globalNetConnect VDD -type pgpin -pin VDD -powerDomain\
 TOP -override
globalNetConnect VSS -type pgpin -pin VSS -all -override
globalNetConnect PD1_VDD -type pgpin -pin VDD -powerDomain PD1
globalNetConnect PD1_VDD -type pgpin -pin VDDG -inst\
 *PD1_1_HEAD* -all -override
globalNetConnect PD1_VDD -type pgpin -pin VDDG -inst\
 *PD1_iso* -all -override
globalNetConnect PD1_VDD -type pgpin -pin VDD -inst\
 *FILLER_PD1* -all -override
```

The next step is to perform the power net routing using the "**sroute**" command. There are header cells in the design and Innovus does not recognize the power-gated pin of the header cell present in this library. Thus, to route the power connection of this pin there is the need of routing it as a signal, which can be seen in the snippet below:

```
#Route the VDDG pin of the header cells
setPGPinUseSignalRoute HEADX2:VDDG
#HEADX2 should be replaced with the name of used cell
#in case the designer is using a different technology
routePGPinUseSignalRoute -nets {VDD PD1_VDD PD2_VDD PD3_VDD\
 PD4_VDD VSS}
```

With the standard cells placed and the power nets routed it is then possible to route the design using the "**routeDesign**" command. After this step the design is almost finished, lacking only the filler cells placement which, for the power domains, is inserted with the command "**addFiller -cell SHFILL1 -prefix FILLER -doDRC**" where "SHFILL1" is the cell's name and should be replaced if the designer wishes to use a different library. The top domain's filler cells, however, are not inserted by the tool which is yet to be optimized for low-power designs. To overcome this problem, the proposed solution is using a different script which should be sourced by the tool. This script, provided in appendix, checks the entire core for gaps where filler cells may be inserted. With the filler cels placed it is possible to export the design and perform signoff analysis.

5.3 Signoff

To perform signoff analysis two small scripts were written. These scripts allow timing and power analysis which help the designer understand the impact of the low-power design and how much does power gating affect timing and power consumption results.

5.3.1 Timing analysis script

The script written for timing analysis is executed using the Tempus tool. After starting the tool, the script must be sourced. This script deals with importing the design generated after floorplan and also with importing timing libraries for the typical, best and worst timing results (which depend on the operating conditions such as supply voltage and temperature). This is achieved with the following set of commands:

```
#Restore the output obtained from Innovus
restoreDesign ${SRC_PATH}/cgra_bitwidth10_rows4_columns4
#Read the worst(max), best(min) and typical(typ) case libraries
```

```
4 read_lib -max ${LIB_PATH}/saed90nm_max.lib
5 read_lib -min ${LIB_PATH}/saed90nm_min.lib
6 read_lib ${LIB_PATH}/saed90nm_typ.lib
7 #Read verilog netlist
8 read_verilog ${SRC_PATH}/cgra_bitwidth10_rows4_columns4.v
9 set_top_module cgra_bitwidth10_rows4_columns4
```

The command "**restoreDesign**" already imports a compiled version of the netlist. However, the netlist which was simulated with Incisive and generated the VCD file was the Verilog file which is read in line 9 of the snippet above. This is done so that the tool is capable of finding the activity values of the nets by reading the VCD file. Besides these file, other optional files may be read by the tool which will allow it to perform more accurate estimations. One example may be seen below:

```
read_sdc ${SRC_PATH}/constraints.sdc
read_spef ${SRC_PATH}/cgra_bitwidth10_rows4_columns4.spef
```

At this stage it is only necessary to specify both the power sources and the delay corners (explained in 4.4.4)

```
set_dc_sources -force -power {VDD PD1_VDD PD2_VDD PD3_VDD PD4_VD\
D}
set_dc_sources -force -ground {VSS}
create_rc_corner -name rccorn1 -T {25} -qx_tech_file {${SRC_PATH\
}/tech_file.tch}
create_op_cond -name op1 -library_file {${LIB_PATH}/saed90nm_max\
.lib} -P {1.0} -V {0.7} -T {125}
create_library_set -name s1 -timing {${LIB_PATH}saed90nm_max.lib}
create_constraint_mode -name sdc1 -sdc_files {${SRC_PATH}/constr\
aints.sdc}
create_delay_corner -name delCorn1 -library_set {s1} -rc_corner \
{rccorn1}
create_analysis_view -name analysis_view1 -delay_corner delCorn1\
 -constraint_mode sdc1
```

At this stage it is possible to generate a timing report by using the command "**report_timing**" which will output the timing of design and show the critical path. An example is shown below:

```
 (...)
H[0].V[3].ALU.PE_out_reg[9]/D     <<<   DFFARX1(0)       +0     2366
H[0].V[3].ALU.PE_out_reg[9]/CLK        setup        2    +93     2459 R
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 (clock clk)          capture                                  3000 R
                      uncertainty                      -150    2850 R
-----------------------------------------------------------------------
 Cost Group    :  'clk' (path_group 'clk')
 Timing slack  :     391ps
 Start-point   :  input42[1]
End-point      :  H[0].V[3].ALU.PE_out_reg[9]/D
```

5.3.2 Power analysis script

As previously mentioned, a script to perform power analysis has been written. The purpose of this script is to assess the power consumption using the activity files obtained from simulation in Incisive. This script is executed using Voltus and starts by importing the design obtained from Innovus and the libraries necessary for power analysis. This is achieved with the following commands:

```
#Restore the output obtained from Innovus
restoreDesign ${SRC_PATH}/cgra_bitwidth10_rows4_columns4
#Read the worst(max), best(min) and typical(typ) case libraries
read_lib -max ${LIB_PATH}/saed90nm_max.lib
read_lib -min ${LIB_PATH}/saed90nm_min.lib
read_lib ${LIB_PATH}/saed90nm_typ.lib
#Read verilog netlist
read_verilog ${SRC_PATH}/cgra_bitwidth10_rows4_columns4.v
set_top_module cgra_bitwidth10_rows4_columns4
```

After importing the design, the default activity for nets which are not depicted in the VCD file is set along with the clock period. Having imported the design, it is then possible to define the activity file and report the power results, like so:

```
set_default_switching_activity -reset
set_default_switching_activity -input_activity 0.2 -period 10.0
read_activity_file -format VCD -scope tb_array/u_array -start 0\
 -end 150000 ${RES_PATH}/simvision.vcd
report_power -rail_analysis_format VS -outfile ./cgra_power.rpt
```

This script outputs information about power consumption which allows the designer to have an accurate estimate of the power consumption of given algorithms which will allow the designer to explore the capabilities of power-gating. A snippet of a typical output file may be seen below.

```
Total Power
-----------------------------------------------------------------
Total Internal Power:       0.38841375          44.3194%
Total Switching Power:      0.20892906          23.8396%
Total Leakage Power:        0.27905313          31.8410%
Total Power:                0.87639594
-----------------------------------------------------------------
```

The snippet above is the result of a power simulation in which the activity of the nets is obtained from a VCD file. This VCD file is obtained from Incisive and the way to run the program is by executing the tool with the following set of commands:

```
irun -sdf_file ../src/cgra_bitwidth10_rows4_columns5.sdf \
  ../../src/verilog/pattern_generator.v \
   /home/diogo/Documents/SAED-EDK90/SAED90_EDK/SAED_EDK90nm/Digita\
   l_Standard_cell_Library/verilog/saed90nm.v \
/home/diogo/Desktop/19June/cgra4b5/cgra_bitwidth10_rows4_columns5\
.v ../src/tb_cgra_synth5.sv -SV \
-lps_1801 ../src/powerIntent.upf -gui -access +C -64bit
```

The command above will start Incisive. The VCD may then be exported to a file. Note that, once again, this is detailed in the webpage.

Chapter 6

Case studies

During this chapter two case studies are presented. The first takes advantage of the parameters that define the size of the CGRA and its input/output bitwidth to generate several CGRAs with different number of columns. All of the generated CGRAs will execute the same algorithm so that power comparisons can be made. The entire design flow is applied to all of the generated CGRAs and the results are presented to be compared and discussed.
The second case study is the application of the design flow to a RTL of a CGRA which was developed by a student[1] working on his dissertation. The main focus of this case study is to have cooperation between the author of this dissertation and the author of the RTL so that the design flow may be assessed in terms of adaptability and functionality and, on the other side, accurate power and timing results may be obtained to assess the performance of the architecture.

6.1 Use case #1

The RTL used to obtain the results shown in this section is the same that was already mentioned in earlier sections and was used during the development of the design flow, therefore demanding no further clarification.
The testbench used to simulate the architecture and generate activity files that can provide a realistic estimate of the algorithm's power consumption maps a FIR filter with 4 taps to the CGRA. It can be used to simulate CGRAs of different sizes just by changing the configuration routine since the number of PEs influences not only the number of clocks needed to program the CGRA but also the data to be sent to the configuration chain. The input data and the filter's coefficients are randomly generated, with a maximum magnitude constraint according to the CGRA's bitwidth, and the output data are calculated according to those values. This golden module of the FIR filter may be generated using *Matlab*.

This section puts six CGRA variants through the frontend design flow. Every CGRA has 4 lines and the columns will range from 4 to 9. The generated reports after synthesising the design

[1]João Lopes, Configurable coarse-grained array architecture for processing of biological signals

CGRA	Cells	Low-power cells	Area	Clock frequency
4 x 4	6842	352	83795.558	333.333MHz
4 x 5	8532	440	104732.467	333.333MHz
4 x 6	10236	528	125440.819	333.333MHz
4 x 7	11929	616	146175.898	333.333MHz
4 x 8	13634	704	167023.411	333.333MHz
4 x 9	15313	792	187967.693	333.333MHz

Table 6.1: CGRA type vs Number of cells

allow conclusions to be taken about the definition of power domains. During this stage it is possible to consult the generated reports in order to know the impact on the design's area. However, since the gate-level netlist generated during synthesis does not contain power-connections information nor power-switch cells, these results do not necessarily match those that will be obtained after floorplan.

The UPF used for these designs allows every power domain to be switched on/off independently and defines isolation rules that clamp the values of every output of the powered-down domains to 0. Besides this, the UPF also defines a power net for each power domain with its own power-switch and groups PEs that share the same column in the same power domain.

In terms of area, it can be seen from table 6.1 that for each extra power domain there is an increase of 88 low-power cells which is the number of isolation cells of each power domain. This makes sense since the number of outputs of each PE is 22 (10 bits for the data output and 12 bits for the configuration output) and there are 4 PEs in each domain, which means that $22 \times 4 = 88$ isolation cells should be inserted per power domain.

Power consumption was also estimated using information about synthesis. These results can be seen in 6.2 and show that the leakage power is a small percentage of the total power. This can be explained by looking at figure 2.12 which shows that leakage power becomes a subject of major concern when using technology nodes of 40nm of smaller. However, these results were obtained using a 90nm library, meaning that the power-savings of this methodology will not be as high as if the design had been implemented in a smaller technology.

After synthesising the CGRAs, the smallest 3 were taken to floorplan and post-route simulation. To assess the power consumption the testbench mentioned earlier is executed and comparisons

Rows x Columns	Power Domains	Leakage Power(μW)	Dynamic Power(μW)	Total power(μW)
4 x 4	4	384,0	6874,7	7258,8
4 x 5	5	478,0	7513,5	7991,5
4 x 6	6	571,3	8469,6	9040,9
4 x 7	7	664,9	9234,3	9899,2
4 x 8	8	758,7	9843,1	10601,8

Table 6.2: Post-synthesis power report

	Leakage Power(μW)	Total Power(μW)
FIR	279.053	876.396
OFF	0.427	0.636

Table 6.3: Post-route power report 1

are made. In the 4 by 4 array the FIR is executed and a power result is calculated. After this execution and because no power domain can be shut down while maintaining the functionality of the circuit, it was decided to assess the power consumption while every power domain is off. This could be useful in cases where, for example, the data that should be calculated once every second is actually being calculated with a faster clock in 100ms, meaning that the CGRA could power down for the remaining 900ms. A comparison between the average power spent calculating a FIR filter and the average power when powered-down is depicted in table 6.3 and shows that when powered-down, the circuit consumes a tiny amount of power. About the powered-down CGRA, it would be expected to have values different than 0 only for the leakage power. However, since this design has top-level cells that show some activity, like isolation cells and header cells, this is not verified.

The 2 remaining CGRAs were tested using a different approach which consists of simulating the same FIR filter with all of the power domains turned on and comparing the power result with the ones obtained after running the algorithm while switching off the unused power domains. These results are shown in table 6.4. From these results shown in 6.4 it is possible to assess the efficiency of power-gating the columns of the CGRA. In the case of the 4 by 5 array, 1 of the 5 PEs is powered off, which means that theoretically a 20% reduction of leakage power is expected. However, the obtained result is a power reduction of approximately 18.9% which is due to the leakage power of the power switching cells which are, obviously, not ideal. In the case of the 4 by 6 array, 2 of the 6 columns are powered off, which would have an expected power reduction impact of approximately 33.3%. However, once again and for the same reasons, the power reduction is approximately 31.6%.

CGRA	Internal Power(μW)	Switching Power(μW)	Leakage Power(μW)	Total Power(μW)
4x5 FIR	397.5	214.2	343.6	955.3
4x5 OFF	397.5	214.2	278.6	890.3
4x6 FIR	390.9	212.4	406.8	1010.2
4x6 OFF	390.3	211.5	278.1	880.0

Table 6.4: Post-route power report 2

6.2 Use case #2

To test the functionality and adaptability of the proposed design flow, it was tested using a different RTL proposed by a colleague as previously mentioned. The CGRA proposed by João Lopes[2] presents a 4 by 4 array of PEs with two separate daisy-chained configuration registers. One of them sets the configuration of the PE and the other stores constant values that can be used for calculations, like for example the FIR filter's coefficients. In terms of power domains, each column will belong to a separate power domain. The infrastructure for the constant values is in a separate domain. The PE's configuration infrastructure belongs to the top power domain which is an always-on domain. This setup allows for configuration of the CGRA to be achieved even when the power domains are off, since the configuration chain is in the top domain which is always-on. The UPF file was generated after modifying the UPF-generating script to match the names of the instances in the Verilog code. For better understanding, refer to the PE's diagram depicted in figure 6.1.

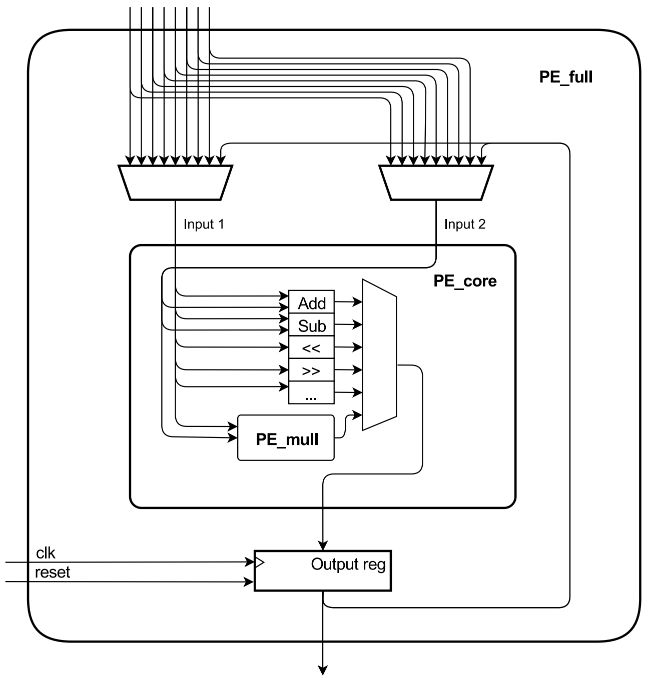


Figure 6.1: PE of the CGRA [3]

The final objective was to provide post-route power results of the architecture for comparison purposes. This was achieved by passing the design through the entire design-flow resulting in a post-route netlist which was then simulated by mapping a 4-tap FIR filter using a testbench designed for such purpose. From the design flow resulted the CGRA physical design depicted in figure 6.3, which is also shown in figure 6.2 without nets so that the power domains are visible, and

[2]ee12136@fe.up.pt
[3]Gently ceded by João Lopes, ee12136@fe.up.pt

provided power results that can be seen in table 6.5. These reports help estimating the advantage

CGRA	Leakage Power(μW)	Total Power(μW)
4x4 FIR	1107.198	2847.554
4x4 OFF	124.093	146.575

Table 6.5: Post-route power report

of speeding up the clock in order to benefit from a longer idle time during which the CGRA would be turned off.

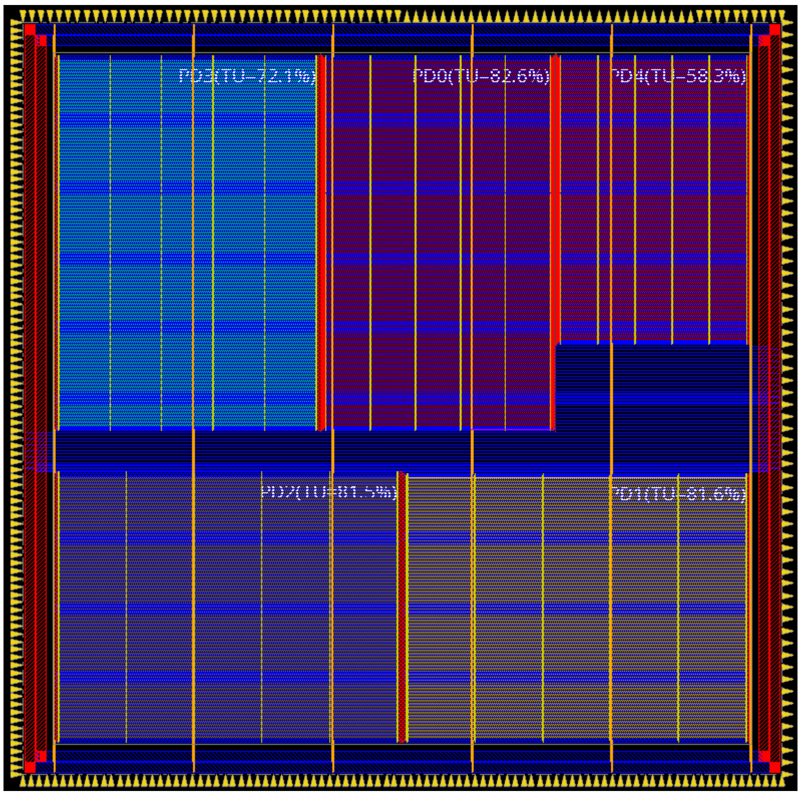


Figure 6.2: Power domains - floorplan

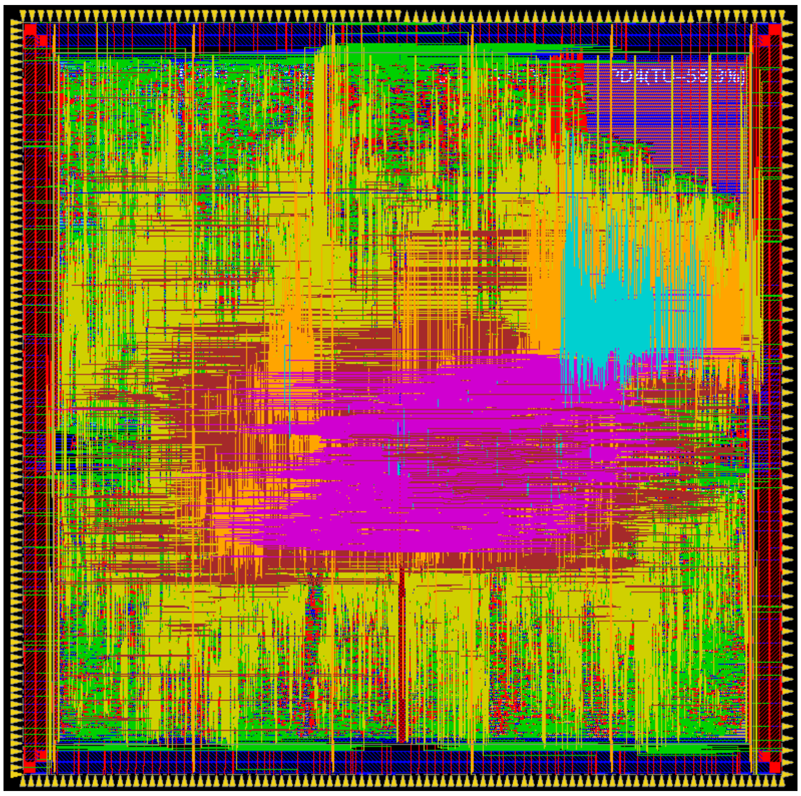


Figure 6.3: Final design - P&R layout

Chapter 7

Conclusions and future work

In this chapter there is a discussion about the project outcomes and a comparison between the objectives of the project and the achieved results. Besides this discussion, there is a proposal for future work which would benefit from this project and take it further.

7.1 Conclusions

The main objective of this project was to establish a design flow for low-power design of CGRAs that would take an HDL description of a CGRA and from it create a physical design as an IP core. The project should result in set of scripts that would automate the process as much as possible.

Besides this main objective there was also the objective of making use of the design flow to assess the impact of power-gating techniques on static power consumption after place and route both with estimates for network activity as well as with activity data taken from simulation tools and also test the adaptability of the proposed design flow and the proposed scripts by repeating the design process for different CGRA architectures.

It is possible to say that the objectives have been achieved as the proposed design flow is capable of generating a physical design of the proposed CGRA architecture and even adapt to parameter changing like the number of lines or columns of the array. It has also been shown in chapter 6 that this adaptability goes even further when a HDL code written by a designer without deep knowledge of low-power methodologies is physically implemented with a low-power intent which has 5 power domains that allow different parts of the circuit to be switched off.

Regarding the assessment of power-gating techniques, their impact is shown when 1 columns in a 5 column CGRA shows 18.9% reduction in terms of power consumption and turning off 2 columns in a 6 column array achieves savings in the order of 31.6%.

In conclusion, this project has shown its potential as a pillar for the design of CGRA architectures and may effectively reduce the overhead time which is critical when the designer is given strict time frames to develop the product, leaving margin for architectural improvements and further studies before tape-out.

7.2 Future work

The future work proposals are the following:

Low-power methodologies Having a well defined design flow is a huge step towards the evaluation of low-power methodologies. This work could be adapted to support other power-saving features like multi-voltage supply or voltage-frequency scaling which also suggest significant gains.

CGRA controller Creating a CGRA controller would allow the generated CGRAs to be tested with a larger set of algorithms. This would be the ideal situation as it would allow for power-efficiency comparisons to be made between different algorithms and data.

Power controller It would also be beneficial to implement a power controller as a block attached to the CGRA controller which, depending on the algorithm and timing constraints, among other variables, would set different power attributes to maximize efficiency.

Ecosystem Finally, one of the most important tasks proposed as future work consists of importing an IP core designed using this methodology to a bigger ecosystem where a general-purpose processor would exist. This would allow the CGRA to act as an accelerator coupled to the processor. This would also allow rail analysis to take place since the CGRA would no longer be a single IP but an IP within a complete chip. This proposal would also take the project to a stage in which it would be possible to create a product which could be fabricated and tested in real-world scenarios.

Appendix A

Front end script

The following script is used in the Genus tool to execute front end design. The name of the script is "*genus_script.tcl*" and has no arguments since the variables are changed in the code.

```
set enable_ieee_1801_support 1

set DESIGN cgra
set systemTime [clock seconds]
set currTime [clock format $systemTime -format %a%d%B%Y_][clock format\
 $systemTime -format %H-%M]
set REPORTS ../run/reports_${currTime}
file mkdir ${REPORTS}

#Library
create_library_domain {saed90nm_typ}
set_attr library \
{/home/diogo/Documents/SAED-EDK90/SAED90_EDK/SAED_EDK90nm/Digital_Stan\
dard_cell_Library/synopsys/models/saed90nm_typ.lib} \
saed90nm_typ

create_library_domain {saed90nm_min}
set_attr library \
{/home/diogo/Documents/SAED-EDK90/SAED90_EDK/SAED_EDK90nm/Digital_Stan\
dard_cell_Library/synopsys/models/saed90nm_min.lib } saed90nm_min

create_library_domain {saed90nm_max}
set_attr library \
{/home/diogo/Documents/SAED-EDK90/SAED90_EDK/SAED_EDK90nm/Digital_Stan\
dard_cell_Library/synopsys/models/saed90nm_max.lib } saed90nm_max


set_operating_conditions -min BEST -min_library saed90nm_min -max WORST\
 -max_library saed90nm_max
set_operating_conditions BEST -library saed90nm_min
set_operating_conditions TYPICAL -library saed90nm_typ

```

```
check_library

set_attr lef_library {  /home/diogo/Documents/SAED-EDK90/SAED90_EDK/SA\
ED_EDK90nm/Digital_Standard_cell_Library/lef/saed90nm_tech.lef \
/home/diogo/Documents/SAED-EDK90/SAED90_EDK/SAED_EDK90nm/Digital_Stan\
dard_cell_Library/lef/saed90nm.lef \
/home/diogo/Documents/SAED-EDK90/SAED90_EDK/SAED_EDK90nm/Digital_Stan\
dard_cell_Library/lef/saed90nm_lvt.lef  \
/home/diogo/Documents/SAED-EDK90/SAED90_EDK/SAED_EDK90nm/Digital_Stan\
dard_cell_Library/lef/saed90nm_hvt.lef \
}

read_hdl {        ../src/bottom_alu.v \
                  ../src/top_alu.v \
                  ../src/alu.v \
                  ../src/pe_array.v \
                  ../src/cgra.v \
                          }

set bitwidth 10
set rows 4
set columns 4
set DESIGN cgra
set DESIGN ${DESIGN}_bitwidth${bitwidth}_rows${rows}_columns${columns}

set_attr hdlin_enable_hier_naming true

read_power_intent ../synth_scripts/powerIntentBeforeSynth3.upf -1801\
 -module ${DESIGN} -version 2.0

elaborate cgra -parameters {10 4 4 }

source ../synth_scripts/constraints.sdc

apply_power_intent
set_attr library_domain saed90nm_typ TOP
set_attr library_domain saed90nm_typ PD1
set_attr library_domain saed90nm_typ PD2
set_attr library_domain saed90nm_typ PD3
set_attr library_domain saed90nm_typ PD4
commit_power_intent

check_design

syn_generic
syn_map
syn_opt

report_timing > ${REPORTS}/timing.rep
```

```
report_gates > ${REPORTS}/cell.rep
report_power > ${REPORTS}/power.rep

write_design ${DESIGN} -basename ../outputs/innovus/${DESIGN}\
_${currTime}/synth_cgra -innovus

exit
```

Appendix B

Back end script

The following script is used with the Innovus tool and transforms a gate-level netlist to a physical design. The name of the script is "*script_innovus.tcl*" and deals with the entire back end flow.

```
set init_gnd_net VSS
set init_io_file {../ipads/io_pads.io}
set LEF_PATH /home/diogo/Documents/SAED-EDK90/SAED90_EDK/SAED_EDK90nm/\
Digital_Standard_cell_Library/lef
set SRC_PATH ../src_afterSynth
set SCRIPTS_PATH ../floorplan_scripts
set lefs "${LEF_PATH}/saed90nm_tech.lef ${LEF_PATH}/saed90nm.lef ${LEF_\
PATH}/saed90nm_hvt.lef ${LEF_PATH}/saed90nm_lvt.lef"
set init_lef_file $lefs
set init_pwr_net {VDD PD1_VDD PD2_VDD PD3_VDD PD4_VDD}
set init_top_cell cgra_bitwidth10_rows4_columns4
set power_intent_files "../src_afterSynth/synth_cgra.upf"
set verilog_files "${SRC_PATH}/synth_cgra.v"
set init_verilog ${verilog_files}

init_design

floorPlan -site unit -r 1 0.7 30 30 30 30 -dieSizeByIoHeight max

read_power_intent -1801 ${power_intent_files}
commit_power_intent

planDesign

addPowerSwitch -ring -powerDomain PD1 -topSide 1
addPowerSwitch -ring -powerDomain PD2 -topSide 1
addPowerSwitch -ring -powerDomain PD3 -topSide 1
addPowerSwitch -ring -powerDomain PD4 -topSide 1

deleteAllFPObjects

read_power_intent -1801 ${power_intent_files}
```

```
commit_power_intent

modifyPowerDomainAttr TOP -rsExts {4 4 4 4}
modifyPowerDomainAttr PD1 -minGaps {4 4 4 4} -rsExts {4 4 4 4}
modifyPowerDomainAttr PD2 -minGaps {4 4 4 4} -rsExts {4 4 4 4}
modifyPowerDomainAttr PD3 -minGaps {4 4 4 4} -rsExts {4 4 4 4}
modifyPowerDomainAttr PD4 -minGaps {4 4 4 4} -rsExts {4 4 4 4}

setPlanDesignMode -useGuideBoundary fence -effort high -incremental false\
 -boundaryPlace true -fixPlacedMacros false -noColorize false -fenceSpacing 5
planDesign

set file_to_check "${SCRIPTS_PATH}/ios.io"
set check_value [file exist ${file_to_check}]

if { $check_value eq 1 } {              \
        echo "Importing␣IO␣file..."     \
        loadIoFile ${file_to_check}     \
}                                       \
else {                                  \
        echo "IO␣file␣not␣available..." \
}

addRing -skip_via_on_wire_shape Noshape -skip_via_on_pin Standardcell -stacke\
d_via_top_layer M9 -type core_rings -jog_distance 0.16 -threshold 0.16 -nets \
{VDD VSS} -follow core -stacked_via_bottom_layer M1 -layer {bottom M1 top M1 \
right M2 left M2} -width 10 -spacing 1 -offset 7

deselectAll
selectObject Group PD1
addRing -skip_via_on_wire_shape Noshape -skip_via_on_pin Standardcell -stacke\
d_via_top_layer M9 -around power_domain -jog_distance 0.16 -threshold 0.16 -t\
ype block_rings -nets {PD1_VDD VSS} -follow core -stacked_via_bottom_layer M1\
 -layer {bottom M1 top M1 right M2 left M2} -width 1.5 -spacing 0.45 -offset \
 0.45

deselectAll
selectObject Group PD2
addRing -skip_via_on_wire_shape Noshape -skip_via_on_pin Standardcell -stacke\
d_via_top_layer M9 -around power_domain -jog_distance 0.16 -threshold 0.16 -t\
ype block_rings -nets {PD2_VDD VSS} -follow core -stacked_via_bottom _layer M\
1 -layer {bottom M1 top M1 right M2 left M2} -width 1.5 -spacing  0.45 -offse\
t 0.45

deselectAll
selectObject Group PD3
addRing -skip_via_on_wire_shape Noshape -skip_via_on_pin Standardcell -stack\
ed_via_top_layer M9 -around power_domain -jog_distance 0.16 -threshold 0.16 \
-type block_rings -nets {PD3_VDD VSS} -follow core -stacked_via_bottom_layer\
 M1 -layer {bottom M1 top M1 right M2 left M2} -width 1.5 -spacing 0.45 -off\
 set 0.45

deselectAll
selectObject Group PD4
addRing -skip_via_on_wire_shape Noshape -skip_via_on_pin Standardcell -stack\
ed_via_top_layer M9 -around power_domain -jog_distance 0.16 -threshold 0.16 \
-type block_rings -nets {PD4_VDD VSS} -follow core -stacked_via_bottom_layer\
 M1 -layer {bottom M1 top M1 right M2 left M2} -width 1.5 -spacing 0.45 -off\
 set 0.45

deselectAll
selectObject Group PD1
addStripe -skip_via_on_wire_shape Noshape -block_ring_top_layer_limit M1 -max\
_same_layer_jog_length 0.9 -over_power_domain 1 -padcore_ring_bottom_layer_li\
mit M1 -number_of_sets 6 -skip_via_on_pin Standardcell -stacked_via_top_layer\
 M9 -padcore_ring_top_layer_limit M1 -spacing 0.45 -merge_stripes_value 0.16\
   -layer M4 -block_ring_bottom_layer_limit M1 -width 0.45 -area {} -nets \
   {PD1_VDD VSS} -stacked_via_bottom_layer M1

deselectAll
selectObject Group PD2
addStripe -skip_via_on_wire_shape Noshape -block_ring_top_layer_limit M1 -max\
_same_layer_jog_length 0.9 -over_power_domain 1 -padcore_ring_bottom_layer_li\
mit M1 -number_of_sets 6 -skip_via_on_pin Standardcell -stacked_via_top_layer\
 M9 -padcore_ring_top_layer_limit M1 -spacing 0.45 -merge_stripes_value 0.16 \
 -layer M4 -block_ring_bottom_layer_limit M1 -width 0.45 -area {} -nets \
 {PD2_VDD VSS} -stacked_via_bottom_layer M1

deselectAll
selectObject Group PD3
addStripe -skip_via_on_wire_shape Noshape -block_ring_top_layer_limit M1 -max\
_same_layer_jog_length 0.9 -over_power_domain 1 -padcore_ring_bottom_layer_li\
mit M1 -number_of_sets 6 -skip_via_on_pin Standardcell -stacked_via_top_layer\
 M9 -padcore_ring_top_layer_limit M1 -spacing 0.45 -merge_stripes_value 0.16 \
 -layer M4 -block_ring_bottom_layer_limit M1 -width 0.45 -area {} -nets \
 {PD3_VDD VSS} -stacked_via_bottom_layer M1

deselectAll
selectObject Group PD4
addStripe -skip_via_on_wire_shape Noshape -block_ring_top_layer_limit M1 -max\
_same_layer_jog_length 0.9 -over_power_domain 1 -padcore_ring_bottom_layer_li\
mit M1 -number_of_sets 6 -skip_via_on_pin Standardcell -stacked_via_top_layer\
 M9 -padcore_ring_top_layer_limit M1 -spacing 0.45 -merge_stripes_value 0.16 \
 -layer M4 -block_ring_bottom_layer_limit M1 -width 0.45 -area {} -nets \
 {PD4_VDD VSS} -stacked_via_bottom_layer M1

addStripe -skip_via_on_wire_shape Noshape -block_ring_top_layer_limit M1 -max\
_same_layer_jog_length 0.9 -padcore_ring_bottom_layer_limit M1 -number_of_sets\
 6 -skip_via_on_pin Standardcell -stacked_via_top_layer M9 -padcore_ring_top_\
 layer_limit M1 -spacing 0.45 -merge_stripes_value 0.16 -layer M6 -block_ring\
 _bottom_layer_limit M1 -width 0.45 -area {} -nets {VDD VSS} -stacked_via_bot\
 tom_layer M1


 placeDesign


# Inside each PowerDomain, associate each cell's VDD pin with the
# domain-specific power net
globalNetConnect PD1_VDD -type pgpin -pin VDD -powerDomain PD1
globalNetConnect PD2_VDD -type pgpin -pin VDD -powerDomain PD2
globalNetConnect PD3_VDD -type pgpin -pin VDD -powerDomain PD3
globalNetConnect PD4_VDD -type pgpin -pin VDD -powerDomain PD4
# Connect VDD to the VDD pin of all the TOP domain cells.
# Connect VSS to every cell in the design (since only header switches are used)
globalNetConnect VDD -type pgpin -pin VDD -powerDomain TOP -override
globalNetConnect VSS -type pgpin -pin VSS -all
# Connect the VDDG pin of the header cells to the PowerDomains power nets
globalNetConnect PD1_VDD -type pgpin -pin VDDG -inst *PD1_1_HEAD* -all -override
globalNetConnect PD2_VDD -type pgpin -pin VDDG -inst *PD2_1_HEAD* -all -override
globalNetConnect PD3_VDD -type pgpin -pin VDDG -inst *PD3_1_HEAD* -all -override
globalNetConnect PD4_VDD -type pgpin -pin VDDG -inst *PD4_1_HEAD* -all -override
# Connect the VDDG pin of the isolation cells to the PowerDomains power nets
globalNetConnect PD1_VDD -type pgpin -pin VDDG -inst *PD1_iso* -all -override
globalNetConnect PD2_VDD -type pgpin -pin VDDG -inst *PD2_iso* -all -override
globalNetConnect PD3_VDD -type pgpin -pin VDDG -inst *PD3_iso* -all -override
globalNetConnect PD4_VDD -type pgpin -pin VDDG -inst *PD4_iso* -all -override


sroute -connect { blockPin padPin padRing corePin floatingStripe } -layerChangeRange\
 { M1(1) M9(9) } -blockPinTarget { nearestTarget } -padPinPortConnect { allPort oneG\
 eom } -padPinTarget { nearestTarget } -corePinTarget { firstAfterRowEnd } -floating\
 StripeTarget { blockring padring ring stripe ringpin blockpin followpin } -allowJog\
 ging 1 -crossoverViaLayerRange { M1(1) M9(9) } -nets { PD1_VDD PD2_VDD PD3_VDD PD4_V\
 DD VDD VSS } -allowLayerChange 1 -blockPin useLef -targetViaLayerRange { M1(1) M9(9) }

setPGPinUseSignalRoute HEADX2:VDDG
routePGPinUseSignalRoute -nets {VDD PD1_VDD PD2_VDD PD3_VDD PD4_VDD VSS}

setNanoRouteMode -quiet -routeWithTimingDriven 1
setNanoRouteMode -quiet -routeWithEco 0
setNanoRouteMode -quiet -routeTdrEffort 3
setNanoRouteMode -quiet -drouteStartIteration default
setNanoRouteMode -quiet -routeTopRoutingLayer default
setNanoRouteMode -quiet -routeBottomRoutingLayer default
setNanoRouteMode -quiet -drouteEndIteration default
setNanoRouteMode -quiet -routeWithTimingDriven true
setNanoRouteMode -quiet -routeWithSiDriven false

routeDesign -globalDetail -viaOpt -wireOpt

addFiller -cell SHFILL1 -prefix FILLER -doDRC
set filler_cell_script "${SCRIPTS_PATH}/filler_cells.tcl"
source $filler_cell_script > ../reports/fillers.txt
```

Appendix C

Filler cells script

This script is used during back end design to place filler cells in the top power domain. The name of this script is "*filler_cells.tcl*" and serves as a complement to the backend script as it inserts the remaining filler cells that the tool can't place.

```
setLayerPreference allM0 -isVisible 0
setLayerPreference allM1Cont -isVisible 0
setLayerPreference allM1 -isVisible 0
setLayerPreference allM2Cont -isVisible 0
setLayerPreference allM2 -isVisible 0
setLayerPreference allM3Cont -isVisible 0
setLayerPreference allM3 -isVisible 0
setLayerPreference allM4Cont -isVisible 0
setLayerPreference allM4 -isVisible 0
setLayerPreference allM5Cont -isVisible 0
setLayerPreference allM5 -isVisible 0
setLayerPreference allM6Cont -isVisible 0
setLayerPreference allM6 -isVisible 0
setLayerPreference allM7Cont -isVisible 0
setLayerPreference allM7 -isVisible 0
setLayerPreference allM8Cont -isVisible 0
setLayerPreference allM8 -isVisible 0
setLayerPreference allM9Cont -isVisible 0
setLayerPreference allM9 -isVisible 0
setLayerPreference routeGuide -isVisible 0
setLayerPreference ptnPinBlk -isVisible 0
setLayerPreference ptnFeed -isVisible 0
setLayerPreference pwrdm -isVisible 0
setLayerPreference netRect -isVisible 0
setLayerPreference substrateNoise -isVisible 0
setLayerPreference powerNet -isVisible 0
setLayerPreference trackObj -isVisible 0
setLayerPreference nonPrefTrackObj -isVisible 0
setLayerPreference net -isVisible 0
setLayerPreference power -isVisible 0
setLayerPreference pgPower -isVisible 0
setLayerPreference pgGround -isVisible 0
setLayerPreference shield -isVisible 0
setLayerPreference unknowState -isVisible 0
setLayerPreference metalFill -isVisible 0
setLayerPreference clock -isVisible 0
setLayerPreference whatIfShape -isVisible 0
setLayerPreference cell -isVisible 0

set counter 0

set startx 30.08
set starty 30.08

set endx [expr 382.08-0.32]
set endy 372.8

set x [expr $startx + 0.32 + 0.1]
set y [expr $starty + 0.16]

set total 0

setLayerPreference inst -isVisible 1

while {1} {
        deselectAll

        if {$x >= $endx} {
                set x [expr $startx + 0.32 + 0.1]
                set y [expr $y + 2.88]

                if {$y >= $endy} {
                        return
                }
        }

        zoomBox [expr $x -2] [expr $y-2] [expr $x + 2] [expr $y + 2]

        gui_select -point [expr $x] [expr $y]

        set type [dbGet selected.objType]

        if {$ptr == "0x0"} {
                echo "Found␣hole␣-␣" $x $y
                set name "TOP_FILL_"
                append name $counter
                addInst -cell SHFILL1 -inst $name -loc $x $y

                set total [expr $total +1]

```

```
                        set counter [expr $counter + 1]
                        set x [expr $x + 0.32]

            } else {
                        echo "Found␣cell␣-␣" $x $y $ptr
                        set x [expr $x + [dbGet selected.box_sizex]]
            }
}
```

References

[1] A. Pantelopoulos and N.G. Bourbakis. A Survey on Wearable Sensor-Based Systems for Health Monitoring and Prognosis. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 40(1):1–12, January 2010. URL: http://ieeexplore.ieee.org/document/5306098/, doi:10.1109/TSMCC.2009.2032660.

[2] Joyce Kwong and Anantha P. Chandrakasan. An Energy-Efficient Biomedical Signal Processing Platform. *IEEE Journal of Solid-State Circuits*, 46(7):1742–1753, July 2011. URL: http://ieeexplore.ieee.org/document/5783951/, doi:10.1109/JSSC.2011.2144450.

[3] T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk, and P.Y.K. Cheung. Reconfigurable computing: architectures and design methods. *IEE Proceedings - Computers and Digital Techniques*, 152(2):193, 2005. URL: http://digital-library.theiet.org/content/journals/10.1049/ip-cdt_20045086, doi:10.1049/ip-cdt:20045086.

[4] Kunjan Patel and Chris J. Bleakley. Coarse Grained Reconfigurable Array Based Architecture for Low Power Real-Time Seizure Detection. *Journal of Signal Processing Systems*, 82(1):55–68, January 2016. URL: http://link.springer.com/10.1007/s11265-015-0981-9, doi:10.1007/s11265-015-0981-9.

[5] Changmoo Kim, Mookyoung Chung, Yeongon Cho, Mario Konijnenburg, Soojung Ryu, and Jeongwook Kim. ULP-SRP: Ultra Low-Power Samsung Reconfigurable Processor for Biomedical Applications. *ACM Transactions on Reconfigurable Technology and Systems*, 7(3):1–15, September 2014. URL: http://dl.acm.org/citation.cfm?doid=2664590.2629610, doi:10.1145/2629610.

[6] Hideharu Amano. A survey on dynamically reconfigurable processors. *IEICE Transactions on Communications*, E89-B(12):3179–3187, 12 2006. doi:10.1093/ietcom/e89-b.12.3179.

[7] Hartej Singh, Ming-Hau Lee, Guangming Lu, Fadi J. Kurdahi, Nader Bagherzadeh, and Eliseu M. Chaves Filho. MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE transactions on computers*, 49(5):465–481, 2000. URL: http://ieeexplore.ieee.org/abstract/document/859540/.

[8] PACT XPP Technologies. Xpp-iii processor overview. URL: https://courses.cs.washington.edu/courses/cse591n/06au/papers/XPP-III_overview_WP.pdf.

[9] Xinan Tang, Manning Aalsma, and Raymond Jou. *A Compiler Directed Approach to Hiding Configuration Latency in Chameleon Processors*, pages 29–38. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000. URL: http://dx.doi.org/10.1007/3-540-44614-1_4, doi:10.1007/3-540-44614-1_4.

[10] T. Sato, H. Watanabe, and K. Shiba. Implementation of dynamically reconfigurable processor dapdna-2. In *2005 IEEE VLSI-TSA International Symposium on VLSI Design, Automation and Test, 2005. (VLSI-TSA-DAT).*, pages 323–324, 2005.

[11] M. Takada H. Tanaka Y. Akita M. Sato T.Kodama, T. Tsunoda and M.Ito. Flexible engine: A dynamic reconfigurable accelerator with high performance and low power consumption.

[12] Zain-ul-Abdin and Bertil Svensson. Evolution in architectures and programming methodologies of coarse-grained reconfigurable computing. *Microprocessors and Microsystems*, 33(3):161–178, May 2009. URL: http://linkinghub.elsevier.com/retrieve/pii/S0141933108001038, doi:10.1016/j.micpro.2008.10.003.

[13] M. Saito, H. Fujisawa, N. Ujiie, and H. Yoshizawa, Cluster architecture for reconfigurable signal processing engine for wireless communication, Proc. FPL, pp.353–359, Sept. 2005.

[14] M. Motomura, A dynamically reconfigurable processor architecture, Microprocessor Forum, Oct. 2002.

[15] B. Levine, Kilocore: Scalable, high-performance, and power efficient coarse-grained reconfigurable fabrics, Proc. Int. Symp. on Advaned Reconfigurable Systems, pp.129–158, Dec. 2005.

[16] Francisco-Javier Veredas, Michael Scheppler, Will Moffat, and Bingfeng Mei. Custom implementation of the coarse-grained reconfigurable ADRES architecture for multimedia purposes. In *Field Programmable Logic and Applications, 2005. International Conference on*, pages 106–111. IEEE, 2005. URL: http://ieeexplore.ieee.org/abstract/document/1515707/.

[17] M. Petrov, T. Murgan, F. May, M. Vorbach, P. Zipf, and M. Glesner, The XPP architecture and its co-simulation within the simulink environment," Proc. FPL, pp.761–770, 2004.

[18] T. Stansfield, Using multiplexers for control and data in D-fabrix, Proc. FPL, pp.416–425, Sept. 2003.

[19] Ricardo S. Ferreira, João M.P. Cardoso, Alex Damiany, Julio Vendramini, and Tiago Teixeira. Fast placement and routing by extending coarse-grained reconfigurable arrays with Omega Networks. *Journal of Systems Architecture*, 57(8):761–777, September 2011. URL: http://linkinghub.elsevier.com/retrieve/pii/S1383762111000373, doi:10.1016/j.sysarc.2011.03.006.

[20] J.m. arnord, s5: The architecture and development flow of a software configurable proecssor, proc. icfpt, pp.121–128, dec. 2005.

[21] Gerry Kane. *MIPS RISC architecture*. Prentice-Hall, Englewood Cliffs, NJ, 1988. URL: http://cds.cern.ch/record/113345.

[22] Kunjan Patel, Chern-Pin Chua, Stephen Fau, and Chris J. Bleakley. Low power real-time seizure detection for ambulatory EEG. In *Pervasive Computing Technologies for Healthcare,*

2009. PervasiveHealth 2009. 3rd International Conference on, pages 1–7. IEEE, 2009. URL: http://ieeexplore.ieee.org/abstract/document/5191226/.

[23] Kunjan Patel, Séamas McGettrick, and Chris J. Bleakley. SYSCORE: A Coarse Grained Reconfigurable Array Architecture for Low Energy Biosignal Processing. pages 109–112. IEEE, May 2011. URL: http://ieeexplore.ieee.org/document/5771260/, doi:10.1109/FCCM.2011.38.

[24] Kunjan Patel and C. J. Bleakley. *Systolic Algorithm Mapping for Coarse Grained Reconfigurable Array Architectures*, pages 351–357. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. URL: http://dx.doi.org/10.1007/978-3-642-12133-3_33, doi:10.1007/978-3-642-12133-3_33.

[25] Kunjan Patel, Séamas McGettrick, and C.J. Bleakley. Rapid functional modelling and simulation of coarse grained reconfigurable array architectures. *Journal of Systems Architecture*, 57(4):383–391, April 2011. URL: http://linkinghub.elsevier.com/retrieve/pii/S1383762111000294, doi:10.1016/j.sysarc.2011.02.006.

[26] Nobuaki Ozaki, Y. Yoshihiro, Yoshiki Saito, Daisuke Ikebuchi, Masayuki Kimura, Hideharu Amano, Hiroshi Nakamura, Kimiyoshi Usami, Mitaro Namiki, and Masaaki Kondo. Cool Mega-Array: A highly energy efficient reconfigurable accelerator. In *2011 International Conference on Field-Programmable Technology (FPT),*, pages 1–8. IEEE, 2011. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6132668.

[27] Volker Baumgarte, Gerd Ehlers, Frank May, Armin Nückel, Martin Vorbach, and Markus Weinhardt. PACT XPP—A self-reconfigurable data processing architecture. *the Journal of Supercomputing*, 26(2):167–184, 2003. URL: http://link.springer.com/article/10.1023/A:1024499601571.

[28] Vasutan Tunbunheng, Masayasu Suzuki, and Hideharu Amano. Romultic: Fast and simple configuration data multicasting scheme for coarse grain reconfigurable devices. In *Proceedings - 2005 IEEE International Conference on Field Programmable Technology*, volume 2005, pages 129–136, 2005. doi:10.1109/FPT.2005.1568536.

[29] Clive Taylor | Electronic. Understanding Low-Power IC Design Techniques. URL: http://electronicdesign.com/power/understanding-low-power-ic-design-techniques,11July2013.

[30] Robert Aitken Alan Gibbons Michal Keating, David Flynn and Kaijian Shi. *Low Power Methodology Manual*. 2007.

[31] Semiconductor Engineering .:. As Nodes Advance, So Must Power Analysis. URL: http://semiengineering.com/as-nodes-advance-so-must-power-analysis/.

[32] Zhigang Hu, Alper Buyuktosunoglu, Viji Srinivasan, Victor Zyuban, Hans Jacobson, and Pradip Bose. Microarchitectural techniques for power gating of execution units. In *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, ISLPED '04, pages 32–37, New York, NY, USA, 2004. ACM. URL: http://doi.acm.org/10.1145/1013235.1013249, doi:10.1145/1013235.1013249.

[33] Steven Dropsho, Volkan Kursun, David H. Albonesi, Sandhya Dwarkadas, and Eby G. Friedman. Managing static leakage energy in microprocessor functional units. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO

35, pages 321–332, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. URL: http://dl.acm.org/citation.cfm?id=774861.774896.

[34] Adam Page, Nasrin Attaran, Colin Shea, Houman Homayoun, and Tinoosh Mohsenin. Low-power manycore accelerator for personalized biomedical applications. In *Proceedings of the 26th Edition on Great Lakes Symposium on VLSI*, GLSVLSI '16, pages 63–68, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2902961.2902986, doi:10.1145/2902961.2902986.

[35] Voltus IC Power Integrity Solution, 2017-06-16. URL: http://bit.ly/2sE50Qn.

[36] Incisive Enterprise Simulator, 2017-06-16. URL: http://bit.ly/2t8HInt.

[37] Genus Synthesis Solution, 2017-06-16. URL: http://bit.ly/2rP92mb.

[38] Conformal Equivalence Checker, 2017-06-16. URL: http://bit.ly/2s4AArR.

[39] Genus Synthesis Solution, 2017-06-16. URL: http://bit.ly/2t4hP7e.

[40] Innovus Implementation System (Hierarchical), 2017-06-16. URL: http://bit.ly/2tI2vuL.

[41] Aruno Dayan John. ASIC Floor Planning. November 2012. URL: https://usebackend.wordpress.com/2012/11/07/asic-floor-planning/.

[42] Innovus Implementation System, 2017-06-16. URL: http://bit.ly/2sOTnoc.

[43] Tempus Timing Signoff Solution, 2017-06-16. URL: http://bit.ly/2u2cpqB.